

Verification and Validation of Software

Assignment 2 Report

Beatriz Rosa 55313

May 20, 2024

1 Introduction

This project involves the verification and validation of a web application using automated JUnit tests developed in Java with HtmlUnit, DBSetup and Mockito.

For this project the following packages were created:

- `functions_for_tests` which has the auxiliary functions for used for tests.
- `htmlunit_tests` has the tests for question number 2.
- `dbsetup_tests` has the tests for question 3.
- `mockito_tests` has the tests for question 4.

2 Test Design

2.1 HtmlUnit tests (question 2)

Each test class in the package `htmlunit_tests` starts by calling the `setUpClass` method that has the annotation `@BeforeAll`, meaning that it will be executed before all tests in the class:

- it initializes a `WebClient` instance, which simulates a web browser for interacting with web pages in a headless manner.
- creates the Auxiliary Function Instances.

- initializes the Database by calling `initDatabase(webClient, customerFunctions)` from `Database_Aux_Functions_For_Tests` to reset the database to a known clean state. This involves fetching the list of all customers, extracting their VAT numbers, and deleting each customer, ensuring that any pre-existing data does not interfere with the test results which allows each test to start with a predictable and controlled environment.

2.1.1 (a) Insertion Of Addresses

Five tests were conducted to verify the correct insertion of two new addresses for an existing customer. The tests confirmed that the addresses were successfully added, resulting in an increase of two rows in the customer's address table.

- **testAddTwoAddresses:** This test verifies whether two new addresses can be successfully added to an existing customer and checks if the total count of addresses for that customer increases by two. The test first ensures that the customer exists and, if so, deletes the customer to start with a clean state. It then recreates the customer, adds two distinct addresses, and verifies that the number of addresses associated with the customer has increased by exactly two.
 - Passes: This confirms that the system successfully adds addresses to a customer.
- **testDataIntegrity:** Ensures that the addresses added to the system are stored accurately and displayed correctly on the web page. After adding an address, this test fetches the customer's details page and checks the content of the addresses table to verify that all the details (street, door number, postal code, locality) match the input data .
 - Passes: which means addresses are correctly added to the system without corruption of the fields.

- **testIdempotence:** Checks how the system handles the addition of duplicate addresses for a single customer by adding the same address twice to a customer and observing the outcome, which should ideally be the prevention of duplicate entries according to business rules.
 - Fails: the system allows duplicate addresses caused by a lack of validation in the address insertion process, therefore needing business rules enforcement . A solution for this would be for the system to first check that for that customer, if there is already that address, this would involve checking every address field.
 - (I reported this as a bug in Backlog [here](#))
- **testConcurrency:** To examine how the system handles concurrent requests, simulating two users adding different addresses to the same customer at the same time. This is implemented by executing two threads simultaneously, each adding a different address to the same customer, and then verifies the total number of addresses to ensure both are added without any loss of data.
 - Fails: There are no database locking strategies stated that prevent race conditions, so a solution for this would be when creating the database, including in the script a specific locking strategy.
- **testNegativeCases:** To validate the robustness of the application in handling incorrect or invalid address data (e.g., empty strings, invalid postal codes) this test checks if the system correctly rejects these entries.
 - Fails: the system accepts the invalid data, which highlights the absence of input validation and error handling mechanisms in the function responsible for adding the address to a customer.

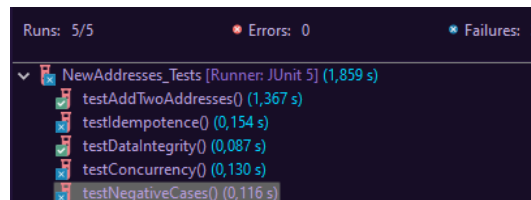


Figure 1: JUnit Tests results of class `NewAddresses_Tests`

2.1.2 (b) Insertion Of Customers

- **testInsertionOfTwoCustomers:** This test verifies that two customers with unique VAT numbers can be correctly inserted and listed in the database. It first ensures the uniqueness of VAT numbers for each customer, inserts two customers with these VAT numbers, and finally verifies that both customers appear in the ' List All Customers' table , ensuring the system handles multiple entries correctly.
 - Passes: Confirms that the database can handle multiple entries and displays the correct data in the ' List All Customers' option .
- **testInsertionOfTwoCustomersDuplicateVAT:** Ensures that the system blocks the addition of a second customer when the VAT number already exists in the database.
 - Passes: The system correctly prevents a second customer with a duplicate VAT from being added.
- **testInvalidVATType:** Assesses the application's ability to handle VAT numbers containing non-numeric characters.
 - Error: Results in a `ClassCastException`, indicating a significant deficiency in the application's error handling architecture. This specific case exposes that the application, when confronted with an invalid VAT type, does not generate any informative error message or page, but instead delivers a response with zero content length, as it is possible to see in figure 2 leading to the incorrect casting of a `TextPage` as an `HtmlPage` in the testing class. The `addCustomer` function is not prepared to receive an argument that has a VAT number with non numeric characters, so this type of validation should also be included in this function so that when this happens the application is capable of sending a descriptive error message and ask the user for a valid VAT number with 9 numeric digits.

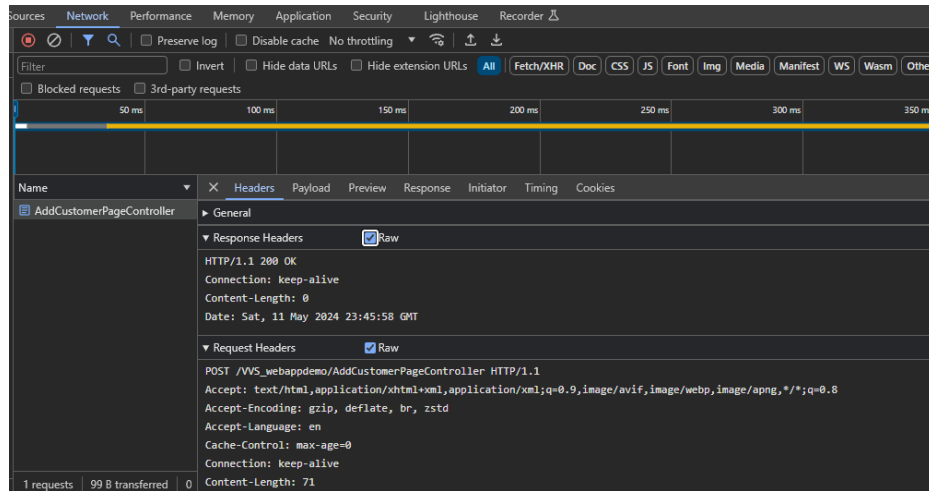
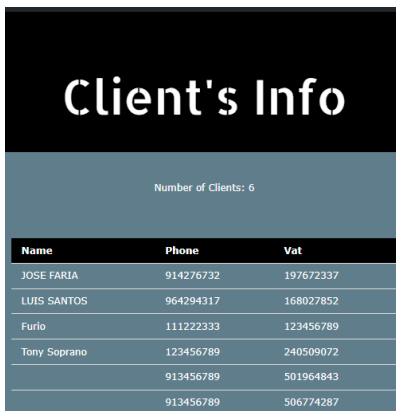


Figure 2: Empty response from server at http://localhost:8080/VVS_webappdemo/AddCustomerPageController upon submitting the customer form

- **testEmptyVAT:** Tests whether the system prevents the addition of a customer without a VAT number, a mandatory field. .
 - Passes: Validates the form’s requirement for a VAT number, ensuring no customer is added without this critical piece of information .
- **testVATCorrectSizeButInvalid:** Checks the system’s capability to distinguish between syntactically correct but semantically incorrect VAT numbers.
 - Passes: Confirms the system’s validation logic is robust enough to detect and reject these VAT numbers.
- **testVATTooLong:** Examines how the application handles VAT numbers that exceed the standard length of 9 digits.
 - Error: Encountered a **ClassCastException**, highlighting a failure in the application’s error management system. The application does not render an error page or message when presented with an overly long VAT number, instead it returns a blank page in the browser (Content Length 0) which is the same error obtained in the previous test **testInvalidVATType** as it is possible to see in figure 2. This inaction triggers a **ClassCastException** in the test framework which expects a standard HTML response. The **addCustomer** function is not prepared to receive an argument that has a VAT number with a length that is superior to the expected, so this type of validation should also be included in this function so that when this happens the application is capable of sending a descriptive error message and ask the user for a valid VAT number with 9 digits.
- **testVATTooShort:** Confirms that VAT numbers below the minimum length are not accepted.
 - Passes: Indicates that the system effectively enforces minimum length requirements for VAT numbers .
- **testVATWithMixedCharacters:** Assesses the application’s response to VAT numbers containing a mix of letters and numbers.
 - Error: Encounters a **ClassCastException** because the application fails to handle invalid VAT numbers appropriately. The lack of a proper error message suggests the application is not equipped to deal with this type of error, resulting in a blank response page (figure 2) and subsequently causing a **ClassCastException** in the testing code when it expects an **HtmlPage** but receives a **TextPage** due to the absence of HTML content. In order to prevent this type of exception to happen, the system should be prepared to not always get a VAT number that corresponds to the valid one in the **addCustomer** function guard that calls **isValidVAT** checks whether the VAT given as argument is valid or not, and in case it is not valid to warn the client of that by sending a error message.

Designation Tests

- **testEmptyDesignationShouldNotAddCustomer:** Confirms that the system does not allow the addition of a customer without a designation.
 - Fails: The application incorrectly adds customers with an empty designation, indicating a significant validation flaw:



Client's Info		
Number of Clients: 6		
Name	Phone	Vat
JOSE FARIA	914276732	197672337
LUIS SANTOS	964294317	168027852
Furio	111222333	123456789
Tony Soprano	123456789	240509072
	913456789	501964843
	913456789	506774287

Figure 3: Customers with an empty designation are incorrectly being added, indicating a bug.

In order to prevent this from happening, a simple check or auxiliary function should be added to the **addCustomer** function that validates if the designation is valid or not before moving on to the addition of the customer.

- **testDesignationWithSpecialCharacters:** Ensures that customer designations containing special characters are rejected.
 - Fails: The test reveals that the application does not properly handle special characters within designations, incorrectly adding such customers without displaying error messages. Once more the **addCustomer** function lacks validation of designation field of the customer, that is necessary so that the instance is not incorrectly created. An error message or warning should also be implemented so that the user knows the cause of the error.
- **testExcessivelyLongDesignation:** Checks whether the application restricts excessively long designations.
 - Passes: Long designations are not accepted, which prevents database and display issues.

Phone Number Tests

- **testEmptyPhoneNumber:** Verifies that the application requires a phone number for customer registration.
 - Passes: Correctly prevents the addition of customers lacking a phone number, upholding the integrity of customer data .
- **testPhoneNumberWithLetters:** Tests the system's ability to reject phone numbers containing alphabetic characters.
 - Passes: Affirms the system's proper validation in excluding non-numeric characters from phone numbers.
- **testExcessivelyLongPhoneNumber:** Evaluates whether the system imposes a maximum length on phone numbers.
 - Passes: The system appropriately restricts the length of phone numbers, avoiding potential errors in data processing or display.

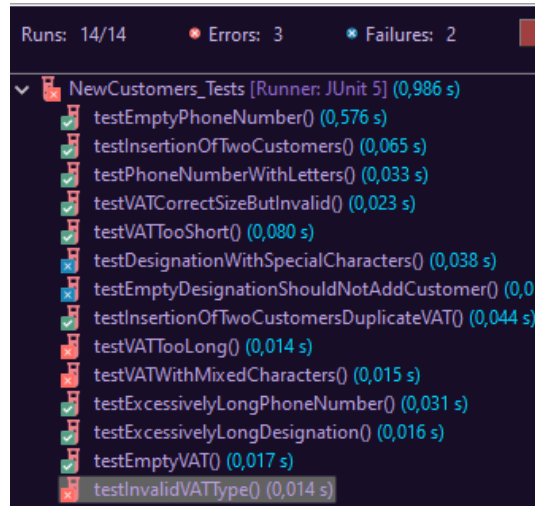


Figure 4: JUnit Tests results of class `NewCustomers_Tests`

2.1.3 (c) Insertion Of Sale

- **testNewSaleIsOpen:** Verifies that a sale, once created for a valid customer, is correctly listed as open.
 - Passes: The test successfully confirms that after creating a sale, the sale is marked as 'Open' and today's date is listed as the sale date. This ensures the application correctly tracks sale status immediately after creation.
- **testSaleOnCustomerWithEmptyDesignation:** Tests the application's ability to handle sales creation for a customer with an empty designation.
 - Passes: Although the application should reject the sale and this test should have failed due to the customer having an invalid empty designation, it instead does in fact add the sale, which indicates that the app does not check if a customer is valid before creating a sale for it. An error message to notify the user of this illegal customer should also have been presented.
- **testSaleOnCustomerWithSpecialCharactersInDesignation:** Examines if the application can handle sales creation for a customer whose designation includes special characters.
 - Fails: The application correctly prevents a sale from being registered for a customer with a problematic designation (special characters). However, it fails to provide an error message, which is inconsistent with good user interface practices, showing that the application's handling of such cases is not up to standard. This is just one more of the application's bugs where no error messages are shown to the user and a check that needs to be included in the function `addSale` implementation
- **testSaleCreationWithoutCustomer:** Tests the application's behavior when attempting to create a sale without providing a VAT number.
 - Error: Encounters a `ClassCastException` because the application fails to handle the absence of a VAT number in the form in an appropriate way. The lack of a proper error message indicates that the application is not equipped to deal with this case, resulting in a blank response page (figure 2). This blank page is what justifies the exception returned in this test, namely a `ClassCastException` in the testing code when it expects an `HtmlPage` but receives a `TextPage` due to the absence of HTML content.
- **testSaleCreationForNonExistentCustomer:** Tests the application's behavior when attempting to create a sale for a non-existent customer with a valid VAT number.
 - Fails: This test attempts to create a sale for a VAT number that is not associated with any customer in the system and then checks for the presence of an error message and the absence of sales records for the specified VAT number. Both assertions fail because the application fails to handle the validation of the existence of a customer properly and does not guide the user correctly by showing an error message. Additionally, the application incorrectly allows the creation of a sale for a non-existent customer. Again a lack of validation of the arguments given to the function is shown, there should be a auxiliary function

called in the beginning of `addSale` function for is example called `isCustomerValid` that checks whether the vat number given as an argument is valid plus if the customer with that number exists in database. In this case this auxiliary function would prevent the sale to be created from this incorrect state of the Customer object, by that would return false and then render a error message to the user alerting that the customer VAT number inserted in the form does not exist in the database.

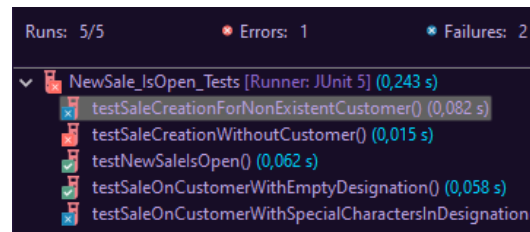


Figure 5: JUnit Tests results of class `NewSale_IsOpen_Tests`

2.1.4 (d) Closing Of a Sale

- **testSaleClosure:** Tests that an open sale can be successfully closed using the application's interface.
 - Passes: The test confirms the functionality to close a sale by using a dynamic retrieval of the latest sale ID for a given customer, ensuring the sale is closed correctly. The method works by first ensuring the customer and an open sale exist, then proceeding to close the sale, and finally verifying that the sale's status has been updated to 'Closed' in the system. This verification is done by examining the sales list table to ensure the sale's record reflects a 'Closed' status, thus confirming the application behaves as expected under normal operating conditions.
- **testClosingAlreadyClosedSale:** Tests the system's ability to handle repeated closure attempts on a sale that has already been marked as closed.
 - Passes: This test ensures that once a sale is closed, subsequent attempts to close the same sale do not affect its status. It involves initially closing a sale and then attempting to close it again, checking that the system does not permit a second closure. The test checks for idempotency by verifying that the count of 'Closed' entries for the sale remains one, demonstrating that the system's state management correctly prevents redundant state changes. This is crucial for maintaining the integrity and consistency of data within the application.
- **testClosedSaleRemainsClosed:** Verifies that a sale remains closed after subsequent operations.
 - Passes: The test ensures that once a sale is closed, it remains listed as closed, even after further interactions with the system. This confirms the stability of the sale's closure status.
- **testClosingSaleForNonExistentCustomer:** Tests the application's behavior when attempting to close a sale for a non-existent customer.
 - Fails: The application wrongly attempts to close a sale and does not display an error message to the user. Additionally, the application fails to check if the customer exists before attempting to close the sale. This reveals deficiencies in the validation and error handling mechanisms.

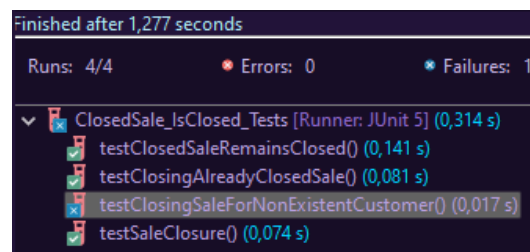


Figure 6: JUnit Tests results of class `ClosedSale_IsClosed_Tests`

2.1.5 (e) Insertion Of customer, address and Sale

- **testNewCustomerSaleAndDelivery:** Tests the complete workflow of adding a new customer, creating a sale, adding a delivery, and verifying the delivery details.
 - Passes: The test successfully verifies that a new customer can be added, a sale can be created for the customer, and a delivery can be inserted for that sale. The delivery details are correctly shown and match the expected details.
- **testNewDeliveryNotCreatedInSaleThatAlreadyHasIt:** Tests that a new delivery is not created if the sale already has a delivery.
 - Fails: The application creates a new delivery even though the sale already has one, indicating a lack of validation to prevent duplicate deliveries.
- **testInvalidAddressInformationHandling:** Tests the system's handling of invalid address information during address creation.
 - Fails: The system allows the creation of addresses with missing street address or empty postal code, showing a lack of validation for address data.
- **testDeliveryCreationWithoutSale:** Tests the system's behavior when attempting to create a delivery without associating it with any sale.
 - Error: The application does not handle the absence of a sale ID properly, leading to a **ClassCastException**. The error occurs because the system fails to provide an appropriate error message and instead returns a blank response.
- **testInvalidCustomerInformationHandling:** Tests the system's handling of invalid customer information during customer creation.
 - Error: The application does not display an error message when attempting to add a customer with a missing VAT number or empty designation. Instead, it results in a **ClassCastException** due to the lack of proper error handling.
- **testDeliveryCreationWithoutAddress:** Tests the application's behavior when attempting to create a delivery without associating it with any address.
 - Error: The application does not handle the absence of an address ID properly, leading to a **ClassCastException**. The error occurs because the system fails to provide an appropriate error message and instead returns a blank response.
- **testDeliveryCreationWithInvalidAddress:** Tests the system's behavior when attempting to create a delivery with an invalid address.
 - Error: The application does not handle invalid address IDs properly, leading to a **ClassCastException**. The error occurs because the system fails to provide an appropriate error message and instead returns a blank response.

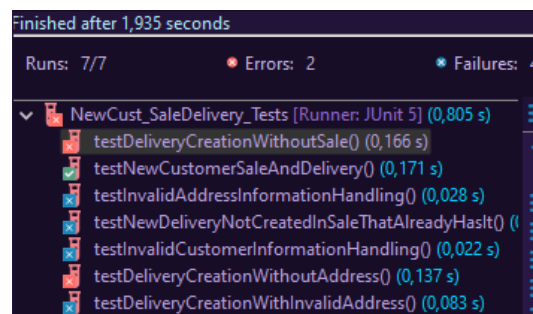


Figure 7: JUnit Tests results of class NewCust.SaleDelivery.Tests

2.1.6 Other backlog errors

Although not in the use cases previously stated on the exercise 2, in the backlog i also published:

- a bug on the Deletion of Customers ([here](#)) which explores the non cascading effect of the addresses upon a deletion of a customer with using its vat number.
- a bug on the Creation of a Sale Delivery ([here](#)) with an invalid customer state

2.2 DbSetup (question 3)

2.2.1 Setup

Project structure for this part: in package `dbsetup_tests` each class corresponds to a specific question of the project.

2.2.2 Database initialization and population

Class `DBSetupUtils` is designed to facilitate the setup of the database for testing purposes. It provides methods and operations to ensure a consistent and clean state of the database before tests are executed. It includes a method to start the application database if it has not already been started, ensuring a single active connection. The operations provided in the class, such as `DELETE_ALL`, `INSERT_CUSTOMER_SALE_DATA`, and `INSERT_CUSTOMER_ADDRESS_DATA`, are used to clear existing data and insert sample data into the database tables. This setup ensures that each test begins with a known state, making the tests reliable and repeatable. Additionally, the class includes static initializations for inserting customers, sales, and addresses, which helps in maintaining a predefined state of the database, important for validating the application's functionality during tests.

2.2.3 Modifications in source code

- `CustomerDTO.java`: added getters for every attribute.
- `SaleDTO.java`: added getters for every attribute.
- `DataSource.java`: added a guard in the beginning of the function in which is verified if the connection is null, and in that case the connection will be attempted again. Also improved exception handling on this function in order make the errors more readable.
- `AddressDTO.java`: added getters for every attribute.

2.2.4 (a) Client with an existing VAT not allowed

- `testAddNewClientWithExistingVAT`: verifies the system's behavior when attempting to add a new customer with a VAT number that already exists in the database. The expected behavior is to throw an `ApplicationException`, ensuring no duplicate customers are added.
 - Passes: The test confirms the system correctly prevents adding a customer with a duplicate VAT number and properly throws an `ApplicationException`. This ensures data integrity and correct enforcement of business rules.

2.2.5 (b) Update of a customer contact

- `testUpdateCustomerPhoneNum`: This test first updates a customer's phone number using a valid VAT number and then retrieves the customer's data to verify the update.
 - Passes: Validates that the system correctly updates the phone number for an existing customer and reflects this change in the database.
- `testUpdateCustomerPhoneNumOnNonExistentInDBVAT`: Attempts to update the phone number for a VAT number that does not exist in the database, expecting an error.

- Error: Shows the correct exception handling when trying to update a non-existent customer’s phone number. This test fails and throws **ApplicationException**, indicating that the VAT number does not exist in the database. In order to prevent this a check should be added in which the VAT number is looked for in the database first before attempting the update, and if it is not present in it, a error message should be returned to the user .
- **testUpdateCustomerContactOnInvalidNumber**: Tests the system’s response to an invalid phone number update attempt, which should ideally enforce data validation.
 - Passes: This test should ideally fail to encourage implementing a check on the validity of the phone number before updating. But it passes which indicates a need to enforce data validation rules in the application.

2.2.6 (c) Deleting all customers

- **testDeleteAllCustomers**: Checks the total number of customers before and after executing a delete all operation, asserting that no customers remain.
 - Passes: Confirms that after executing the deletion operation, no customer records remain in the database, ensuring complete data removal as per the specified functionality.

2.2.7 (d) Deleting a certain customer

- **testDeleteAndAddCustomer**: This test deletes a customer by their VAT number and then attempts to re-add them to check both operations for robustness.
 - Passes: Ensures that after deleting a customer, the system allows re-adding them without issues, confirming that delete operations are clean and do not leave disruptive residuals.
- **testDeleteNotInDBCcustomerAndAdd**: This test first attempts to delete a customer using a VAT number that does not exist in the database, expecting to catch an **ApplicationException**. It then proceeds to add a new customer with the same VAT number to ensure that the system can handle additions following failed deletions correctly.
 - Passes: Confirms the robustness of the error handling mechanisms by properly throwing and catching the expected exception when attempting to delete a non-existent customer. It also verifies that subsequent additions are not hindered by prior failed deletion attempts.

2.2.8 (e) Deleting a certain customer and sales

- **testDeleteCustomerAndCheckSales**: : This test first checks if the targeted customer has any sales recorded. If no sales exist, the test initially creates two sales for this customer to establish a precondition. The customer is then deleted, and the test verifies whether all associated sales are also removed. This is crucial for maintaining data integrity and ensuring that orphan records do not persist in the database.
 - Fails: This failure suggests a problem with the database integrity constraints. To resolve this, the database schema is should be correctly set up with cascade delete options or that the service layer explicitly handles the deletion of all dependent records.

2.2.9 (f) Adding a new sale increases the total number of all sales

- **testAddNewSaleAndIsIncByOne**: The test begins by recording the initial count of sales. It then adds a new sale and checks that the total number of sales has increased by one. This is achieved by calling the **addSale** method on **SaleService** and subsequently fetching the updated sales list.
 - Passes: The test passes if the number of sales after adding one is exactly one more than the initial count, confirming that the system correctly records each new sale.

2.3 Additional tests added

The first two tests here described are specifically in the Sales concept and the last 2 are on Sale Deliveries.

- **testCreateSaleForNonExistentVATInDB:** This test verifies that the system enforces business rules by rejecting the creation of a sale for a VAT number that does not correspond to any existing customer in the database.
 - Fails: An `ApplicationException` was expected but not thrown, indicating that the system incorrectly allows the creation of a sale for a non-existent customer VAT. This failure points to a missing validation mechanism in the sales processing logic. A possible solution for this problem would be to enhance the validation process within the `addSale` method of the `SaleService` class to ensure that a customer exists before proceeding with the sale creation. Implementing a pre-check for customer existence or integrating more robust error handling would prevent this issue.
- **testUpdateSaleStatus:** This test checks the functionality of updating a sale's status within the system. After adding a new sale and retrieving its ID, the test updates the status of this sale and verifies that the change is correctly reflected in the system.
 - Passes: Confirming the system's ability to update and reflect the sale's status accurately if the sale's status is updated to `CLOSED` and matches the expected status in subsequent retrievals.
- **testSingleDeliveryPerSale:** This test is designed to validate the application's constraint that restricts each sale to only one delivery. It verifies that attempting to associate a second delivery with an already delivered sale is properly handled by the system, enforcing business rules that prevent multiple deliveries for the same sale.
 - Fails: The test fails because an `ApplicationException` is not thrown as expected when a second delivery attempt is made for the same sale. This indicates that the `SaleService` does not adequately check for existing deliveries for a sale before allowing a new delivery to be added.
To solve this issue, the `addSaleDelivery` method in the `SaleService` should be enhanced to include a preliminary check that ensures no delivery exists for the sale before proceeding to add a new one. This can be implemented by querying the database for existing deliveries linked to the specific sale ID and rejecting the operation if a delivery record is found. Adding this validation step will prevent duplicate deliveries for a sale and ensure the application adheres to its intended business logic.
 - I reported this bug on Backlog ([here](#))
- **testGetSalesDeliveryByVat:** This test assesses the system's ability to retrieve sales deliveries by VAT number. It adds a delivery to a newly created sale and checks if this delivery can be accurately fetched using the customer's VAT.
 - Passes: The test is successful if the retrieval returns the correct number of deliveries associated with the VAT and the data matches the expected results.

2.4 Mockito (Question 4)

Yes. With some modifications in the code is possible to mock for example the business layer's module `CustomerService` and `AddCustomerPageController` because `AddCustomerPageController` depends on `CustomerService`.

2.4.1 Refactoring

In order to prove the last statement, instead of changing directly the source code of the classes of the project to not conflict with the tests done on the previous questions 2 and 3 and their respective results, it was decided to create duplicates of the following classes in order to answer this question:

- `CustomerService` - `CustomerServiceModified`
- `AddCustomerPageController` - `AddCustomerPageControllerModified`

And created the following classes:

- `ICustomerService` an interface.
- `AddCustomerPageControllerTest` test class that uses Mockito.

2.4.2 Challenges and modifications

- **ICustomerService:**

- Decoupling Code Dependencies: Originally, the `CustomerService` was a concrete singleton class that components depended directly upon. This tight coupling made it difficult to modify or test the behavior of these components independently from `CustomerService`. By defining an `ICustomerService` interface, the dependency is shifted from a specific implementation to a contract. This means that any class that performs operations involving customer services now depends on the abstraction (interface), not the concrete implementation. It decouples the service consumers from the details of how the services are implemented.
- Facilitating Unit Testing: Testing classes that rely on singletons can be problematic because the singleton's state persists across tests and its usage cannot be easily intercepted or replaced with a mocked implementation. With the interface `ICustomerService`, it becomes straightforward to use Mockito or any other mocking framework to create mock implementations for the service during testing. This allows tests to be isolated, focusing only on the behavior of the component under test, not on the underlying service logic or external dependencies.

- **CustomerServiceModified:**

- Singletons Pose a Challenge for Mocking: They carry state that persists across the application's lifecycle, which can lead to unintended side effects in tests since one test can alter the state for another adding to that, this pattern restricts the creation of an instance to one, making it hard to substitute with a mocked or a new instance during testing. So for these reasons it was decided to remove the singleton pattern, in this way multiple instances of this service `CustomerServiceModified` can be created as needed, which is useful in the scenario where each test case might require a fresh instance to ensure test isolation.
- Dependency Injection: By using the interface `ICustomerService`, it is possible to inject mock implementations of the service into the classes that depend on it during testing, significantly simplifying unit testing by isolating the service's consumers from its concrete implementation.
- With the refactoring into `CustomerServiceModified`, the tests in `AddCustomerPageControllerTest` can now mock it using Mockito without worrying about the persistent state that a singleton would have introduced.

- **AddCustomerPageControllerModified:**

- Dependency Injection via Constructor: The controller now receives an instance of `ICustomerService` through its constructor. This change supports dependency injection, allowing different implementations of the service (real or mocked) to be passed based on the context (production or testing), this facilitates unit testing by allowing mocks of `ICustomerService` to be injected during testing, ensuring that controller tests remain isolated from the service logic.
- Use of Interface Instead of Concrete Class: Instead of directly using `CustomerService`, the modified controller interacts with the `ICustomerService` interface. This abstraction further decouples the controller from the concrete implementation details of the service. This increases the flexibility of the controller's code and ensures that any service adhering to the `ICustomerService` interface can be used without modifying the controller. This is especially useful in scenarios where different service implementations might be needed
- Addition of a Public Test Method: called `testProcess` that, wraps the method protected `process`. This allows for direct invocation of the controller's processing logic during unit tests. What is beneficial about this is that while maintaining the encapsulation of the process method (as it remains protected), the `testProcess` method provides a pathway for explicitly testing the controller's request handling logic without needing to bypass access restrictions typically imposed on HTTP request handlers.

- **AddCustomerPageControllerTest:**

- **Annotations:**

- * `@RunWith(MockitoJUnitRunner.class)`: This annotation integrates the Mockito test runner with the JUnit framework. It initializes mocks and allows for the injection of mocked dependencies before the tests are executed. This is crucial for enabling dependency injection without manual setup, making the test class cleaner and easier to manage.

- * **@Mock:** Used to create mock instances of classes. In this test class, `CustomerServiceModified`, `HttpServletRequest`, `HttpServletResponse`, and `RequestDispatcher` are mocked. These mocks replace the actual implementations of these classes, removing the dependency on external systems or complex logic that might influence the test outcomes.
- * **@InjectMocks:** This annotation is used to create an instance of the class under test (`AddCustomerPageControllerModified`) and inject the mocked dependencies into it. It simplifies the setup by automatically injecting suitable mocked dependencies wherever required in the class being tested.
- **Setup Method:** The `setup()` method is annotated with **@Before**, ensuring it runs before each test method to configure the necessary preconditions for the tests. It sets up the behavior of the mocks using Mockito's `when()` method. For example, it configures the `customerService` mock to return a predefined `CustomerDTO` when `getCustomerByVat()` is called. This control over the mock's behavior allows the test to simulate different scenarios without relying on the actual service logic.
- **JUnit Tests**
 - * **testAddCustomerSuccess:** This method runs with success, its designed to test the successful addition of a customer using the `AddCustomerPageControllerModified` class. Its implementation starts by setting up the necessary request parameters for VAT number, phone number, and designation, then it invokes the `testProcess` method of the controller. Then it proceeds to verify that the `addCustomer` method of the `CustomerServiceModified` service is called with the correct parameters, ensuring the correct customer data is processed. Furthermore, makes sure that the request is forwarded to the `CustomerInfo.jsp` page, indicating a successful operation. Overall this test ensures that the controller behaves correctly when provided with valid customer data.
 - * **testAddCustomerFailure:** This test runs with success, verifying the behavior of the `AddCustomerPageControllerModified` class when an invalid VAT number is provided. The test arranges the necessary setup by mocking the `HttpServletRequest` to return an invalid VAT number, a phone number, and a designation when their respective parameters are requested. The `CustomerServiceModified` mock is configured to throw an `ApplicationException` when the `addCustomer` method is called with any integer VAT, string designation, and integer phone number. During the test execution, the `testProcess` method of the controller is called, and the behavior is verified to ensure that the `addCustomer` method of the `CustomerServiceModified` mock is invoked with the correct parameters, and the `RequestDispatcher` forwards the request to the "CustomerError.jsp" page, indicating proper error handling in the controller when faced with an invalid VAT number.