Assignment 2

Mestrado em Engenharia Informática

Verificação e Validação de Software

2023/2024

Ricardo Sobral 56332

HtmlUnit Tests

As classes InsertNewAddressesTest.java, InsertNewCustomersTest.java, AddNewSaleTest.java, CloseSaleTest.java, AddNewDeliveryToNewCostumerSaleTest.java presentes no package HtmlUnit são as classes responsáveis por testar as narrativas (a), (b), (c), (d) e (e) presentes no ponto 2.2 do enunciado, respetivamente.

Para esta alínea (a), foi criada a classe InsertNewAddressesTest.java. Em primeiro lugar, foi utilizado o método getNumberRows() para obter o número inicial de linhas na tabela de endereços. Em seguida, com o método insertNewAddressesTest(), foram inseridos dois novos endereços para um cliente existente, verificando se os parâmetros foram corretamente adicionados à tabela de endereços. Após a inserção, foi obtido o número de linhas após a inserção e verificado se correspondia ao número inicial de linhas mais dois, que é o número de novos endereços adicionados. Este teste garante que a funcionalidade de adicionar novos endereços para um cliente está a funcionar corretamente

Para a alínea (b), foi criada a classe InsertNewCustomersTest.java. Nesta classe, foram inseridos dois novos clientes e verificado se todas as informações foram corretamente listadas.

Novamente método *getNumberRows*() foi utilizado, mas desta vez para obter o número de clientes existentes antes da execução do teste. Em seguida, foram preenchidos e submetidos dois formulários com informações de clientes. Depois disso, foi verificado se os novos clientes foram corretamente adicionados, verificando a página de listagem de clientes, também assegurando que todas as informações (VAT, designação e telefone) estavam presentes. Por fim, foi verificado se o número de linhas na tabela aumentou em dois, correspondendo aos dois novos clientes adicionados. Este teste garante que a funcionalidade de adicionar novos clientes está a funcionar corretamente e que as informações desses clientes foram corretamente inseridas.

Para a alínea (c), foi criada a classe AddNewSaleTest.java. Nesta classe, uma nova sale é adicionada e é verificado se ela é inserida como uma sale aberta para o cliente correspondente.

Novamente, o método goToNewSalePage() é utilizado, mas desta vez para obter o número inicial de sales do cliente. Em seguida, é preenchido um formulário com o

VAT do cliente e a sale é submetida. Após isso, é verificado se o número de sales do cliente aumentou em um, se a última sale adicionada está com o estado 'O' (aberto) e se o VAT da última sale é o mesmo do cliente. Este teste garante que uma nova sale é corretamente adicionada e é inserida como aberta para o cliente correspondente.

Para a alínea (d), foi desenvolvida a classe CloseSaleTest.java. Nesta classe, é verificado se há alguma sale com estado 'O' (aberto) e de seguida essa sale é fechada.

Inicialmente, a página para atualizar o estado da sale é acessada. Em seguida, é procurada a primeira linha de uma sale aberta na tabela de sales. Se uma sale aberta for encontrada, seu ID é guardado e é preenchido um formulário com esse ID para fechar a sale. Após submeter o formulário, é verificado se a sale foi fechada com sucesso, garantindo que após o fecho, a sale é listada como fechada.

Para atingir o objetivo proposto pela narrativa (e) foi criada a classe AddNewDeliveryToNewCostumerSaleTest, onde é realizada uma sequência de ações para testar o processo de criação de um novo cliente, uma nova sale para esse cliente, a inserção de uma entrega para essa sale e, por fim, a exibição da entrega da sale.

Inicialmente, é verificado se o cliente com o VAT específico não existe na página de clientes. Em seguida, é obtido o número inicial de sales para esse cliente.

Após isso, a classe simula a adição de um novo cliente, preenchendo um formulário com o VAT, designação e telefone do cliente. É verificado se o cliente foi adicionado corretamente, garantindo que as informações do cliente estejam presentes na página de clientes e que o número de linhas aumentou em 1.

Depois, uma nova sale é criada para esse cliente. O teste acessa a página de adicionar nova sale, preenche o formulário com o VAT do cliente, submete o formulário e verifica se a última sale foi aberta corretamente, verificando se o estado é 'O' (aberto) e se o VAT é o mesmo do cliente. Também é verificado se o número de sales aumentou em 1.

Com o ID da última sale adicionada, o teste simula a adição de uma nova entrega para essa sale. Ele acessa a página de adicionar entrega para a sale, preenche o formulário com o ID do cliente e da sale, e insere a entrega. É verificado se a entrega foi adicionada corretamente, garantindo que as informações da sale e do endereço de entrega estejam presentes na página de informações de entregas.

DBSetup

De maneira a testar todos os casos propostos no ponto 2.3 do enunciado, foram criadas 3 classes, a DBSetupUtils para configurar o acesso à base de dados e definir as operações com fim de popular a base de dados, a classe CustomerDBTest que inclui os testes (a), (b), (c) ,(d), e a classe SaleDBTest para os testes (e), (f).

a) O teste *addCustomerWithExistingVAT* verifica se o sistema impede a adição de um novo cliente com um número de identificação fiscal (VAT) que já existe na DB, conforme solicitado no ponto (a) do enunciado. Primeiro, é definido um VAT existente na base de dados de teste. Em seguida, é verificado se o cliente com o VAT existente está presente na base de dados. Após essa verificação, o teste confirma se o sistema lança uma exceção do tipo ApplicationException ao tentar adicionar um novo cliente com o VAT existente.
b) Para o caso (b), foi criada a método *checkCustomerUpdate* onde primeiro é guardado o cliente na variável *customer*, de seguida atualiza-se o número de telefone do cliente e é inicializada a variável *otherCustomer* com o cliente já atualizado. Posteriormente são feitas as asserções para verificar que o cliente guardado nas duas variáveis não tem o mesmo número de telefone, e se o no caso do cliente com número atualizado (*otherCustomer)* tem o número de telefone igual ao número inserido para ser o novo número de telefone do cliente.
c) Para o caso (c), foi criada a método *deleteAllButOneCustomer,* em que inicialmente foi obtida uma lista de todos os clientes e verificado com asserções que a lista não estava vazia, ou seja, que existem clientes. Posteriormente são eliminados todos os clientes, com a método *removeCustomer* do *Enum CustomerService* e obtida novamente a lista de clientes a qual é usada para novas asserções e verificar que já não existe clientes.
d) Para o caso (d), foi criada a método *addDeletedCustomer* com intuído de verificar que se depois de remover um cliente é possível inseri-lo de volta sem problemas. Para fazer isso priemiro o cliente é obtido através do seu VAT (caso ele não se existe seria lançado um erro), posto isto o cliente é removido, verifica-se que de facto já não existe um cliente na lista de clientes com o VAT do cliente removido, e aí sim tenta-se adicionar o cliente, uma vez que a adição é concluída é verificado se existe um cliente na lista de clientes com o VAT do cliente adicionado.

e) Para cumprir o caso (e), foi criada a método *deleteCustomerAndDeliveries*. Neste método foi primeiro verificado se existia clientes e se um dos clientes tem sales e sale deliveries. Uma vez feitas as verificações é removidas o cliente e as suas sales e sales deliveries com as devidas funções (as funções para remover as sales e as sales deliveries, estavam em falta no SUT e por isso foram criadas). Após as remoções é verificado se o cliente e as suas sales/sales deliveries ainda existem.

f) Para concretizar o caso (f), foi criada a método addSaleDelivery. Neste método obtemos um cliente e a sua lista de sales, e é verificado se a lista de sales do cliente não está vazia. Posteriormente é adicionado uma nova sale delivery e é verificado se o número de sale deliveries aumentou por 1.

Para além dos casos indicados no enunciado forma criados 4 testes extras, 2 deles para testar sales deliveries, e outros 2 para testar sales.

O teste *addTwoSaleDeliveryWithSameSaleId*, verifica se é possível adicionar duas sales delivery com o mesmo número de sale com o id_addr.

```java
/**
 * Teste extra que tenta adicionar duas sales delivery com o mesmo número de salee com o id_addr
 *
 *
 * @throws ApplicationException
 */
@Test
public void addTwoSaleDeliveryWithSameSaleId() {
    int SaleId = 1;

    try {
        SaleService.INSTANCE.addSaleDelivery(SaleId, 200);
        SaleService.INSTANCE.addSaleDelivery(SaleId, 200);
    } catch (ApplicationException e) {
        assertEquals(e.getMessage(), "Can't add address to cutomer.");
    }

}
```

O teste *addSaleDeliveryInvalidSaleId*, verifica se é possível adicionar uma sale delivery com um id inválido de uma sale, caso não seja possível é enviado o erro adquado.

```java
/**
 * Teste extra que tenta adicionar uma sales delivery com um número de sale
 * inválido
 *
 * @throws ApplicationException
 */
@Test
public void addSaleDeliveryInvalidSaleId() {
    int invalidSaleId = 1971;

    // Verifica se a ApplicationException é lançada
    assertThrows(ApplicationException.class, () -> {
        SaleService.INSTANCE.addSaleDelivery(invalidSaleId, 200);
    });
```

O teste *addSale*, verifica que ao adicionar uma nova sale que o número de sales do cliente e sales totais aumentou mais um. Para executar isso obtemos

um cliente, a sua lista de sales e lista de todas as sales e é verificado que o cliente pelo menos uma sale e que o cliente não é null. Posteriormente é adicionada a sale e é verificado se o tamanho das listas de sales (do cliente e de todos os clientes) foram incrementados por 1.

```java
/**
 * Teste extra que verifica que ao adicionar uma nova sale verifica que o numero
 * de sales do cliente e sales totais aumentou mais um
 *
 *
 *
 * @throws ApplicationException
 */
@Test
public void addSale() throws ApplicationException {
    int vat = 197672337;
    // obter o primeiro customer de todos
    CustomerDTO customer = CustomerService.INSTANCE.getCustomerByVat(vat); // ele deve ter pelo menos uma
                                                                           // saledelivery
    int numberOfSalesCustomerBefore = SaleService.INSTANCE.getSaleByCustomerVat(vat).sales.size();
    int numberSalesCustomerAfter = 0;
    int numberSalesOverallBefore = SaleService.INSTANCE.getAllSales().sales.size();
    assertNotEquals(0, numberSalesOverallBefore);

    assertNotNull(customer);
    assertNotEquals(numberOfSalesCustomerBefore, numberSalesCustomerAfter);

    // adicionar nova sale delivery
    try {
        SaleService.INSTANCE.addSale(vat);
    } catch (ApplicationException e) {
        assertEquals(e.getMessage(), "Can't add customer with vat number " + vat + ".");
    }


    // verificar que existe mais uma sale delivery de antes

    numberSalesCustomerAfter += SaleService.INSTANCE.getSaleByCustomerVat(vat).sales.size();
    int numberSalesOverallAfter = SaleService.INSTANCE.getAllSales().sales.size();

    assertEquals(numberSalesCustomerAfter, numberOfSalesCustomerBefore + 1);
    assertEquals(numberSalesOverallAfter, numberSalesOverallBefore + 1);

}
```

Por último, o teste removeSales, verifica se é possível remover todas as sales de um cliente dado o seu VAT. Primeiro foi guardado na variável *customerVat* o VAT de um cliente, que em princípio deve ter pelo menos uma sale, mas de maneira a garantir que esse cliente está associado a uma sale é adicionada uma sale com o seu e através de asserções é verificado se a lista das suas sales não está vazia. Posto é chamado o método *removeSaleByCustomerId* dentro de um bloco *try catch*, de maneira a remover todas as sales do cliente. É de notar que este método chamado foi inicialmente criado para que fosse possível cumprir o caso (e) do enunciado. Uma vez que as sales foram removidas, verificamos através de asserções sobre a lista de sales do cliente, que esta já se encontra vazia.

```java
/**
 * Teste extra que verifica se é possível remover todas as sales de um cliente dado o seu vat
 *
 *
 *
 * @throws ApplicationException
 */
@Test
public void removeSales() throws ApplicationException {
    // Adiciona uma nova sale para teste
    int customerVat = 197672337;
    SaleService.INSTANCE.addSale(customerVat);
    // Obtém a última sale adicionada
    List<SaleDTO> salesList = SaleService.INSTANCE.getSaleByCustomerVat(customerVat).sales;
    int numberOfSalesBefore = salesList.size();
    assertNotNull(salesList);
    assertNotEquals(0, salesList.size());


    // Remove as sales do Cliente

    try {
        SaleService.INSTANCE.removeSaleByCustomerId(customerVat);;
    } catch (ApplicationException e) {
        assertEquals(e.getMessage(), "Sale of the customer with vat number " + customerVat + " don't exist.");
    }


    //verificar se todas as sales forma removidas
    try {
        SalesDTO salesDTO = SaleService.INSTANCE.getSaleByCustomerVat(customerVat);
        int numberOfSalesAfter = salesDTO.sales.size();
        assertEquals(0, numberOfSalesAfter);
        assertEquals(numberOfSalesAfter + 3, numberOfSalesBefore);
    }  catch (ApplicationException e) {
        assertEquals(e.getMessage(), "Invalid VAT number: " + customerVat);
    }
```

Apesar de já incluir os 4 testes extras pedidos no enunciado, após alguma investigação encontrei um erro grave no método *addSale* do Enum SaleService, este permitia adiciona uma sale com o VAT de um cliente inexistente. Dada esta descoberto foi alteada a função *addSale* e criado um teste *addSaleWithVatInvalid,* ao ficheiro que continha os testes executados sobre o enum *SaleService*.

Este teste extra criado, limita-se apenas a chamar o método addSale com um Vat, inválido, antes das alterações ao SUT a exceção do catch não era levantada, mas devia pois como é obvio não devia ser possível adicionar uma *Sale* a um cliente que não existe.

```java
/**
 * Teste extra que tenta adicionar uma sale com um cliente inexistente.
 *
 * @throws ApplicationException
 */
@Test
public void addSaleWithVatInvalid() throws ApplicationException {
    int vat = 000000000;
    try {
        SaleService.INSTANCE.addSale(vat);
    } catch (ApplicationException e) {
        assertEquals(e.getMessage(), "Invalid VAT number: " + vat);
    }
}
```

Mockito

Dado que não é possível usar o mockito sobre Enums como é o caso do CostumerService e do SaleService então é necessário aplicar algumas alterações para tornar estes enums em classes convencionais, para que seja possível aplicar o mockito, vejamos por exemplo o que era necessário fazer ao enum CostumerService:

- **Remover a enumeração e o modificador INSTANCE:** O modificador INSTANCE deixa de ser necessário, pois CustomerService vai passar a ser uma classe;

- **Ajustar a invocação de métodos estáticos do CustomerService:** Como o INSTANCE deixa de ser necessário, podemos chamar os métodos estáticos de CustomerService diretamente.

```java
import java.util.List;

import webapp.persistence.AddressRowDataGateway;
import webapp.persistence.CustomerRowDataGateway;
import webapp.persistence.PersistenceException;

public class CustomerServiceMockito {

    // Removi o modificador INSTANCE e os métodos de enumeração

    public CustomerDTO getCustomerByVat(int vat) throws ApplicationException {
        if (!isValidVAT(vat))
            throw new ApplicationException("Invalid VAT number: " + vat);
        else {
            CustomerRowDataGateway customer = new CustomerRowDataGateway(vat, "Mock Customer", 123456789);
            return new CustomerDTO(customer.getCustomerId(), customer.getVAT(), customer.getDesignation(),
                    customer.getPhoneNumber());
        }
    }

```

Como podemos ver na figura acima, foi criada uma classe CustomerServiceMockito, com métodos semelhantes ao CustomerService original, mas neste caso o CustomerServiceMockito é uma public class, e onde também já não é usado modificador INSTANCE.

De maneira a demonstrar como seriam aplicados testes usando o mockito, produzi um simples teste do método getCustomerByVat da classe CustomerServiceMockito.

```java
import org.junit.Test;

import webapp.services.*;

public class CustomerServiceTest {

    @Test
    public void testGetCustomerByVat() throws ApplicationException {
        // Criar um mock para simular o CustomerServiceMockito
        CustomerServiceMockito mockedCustomerServiceMockito = mock(CustomerServiceMockito.class);

        // Definir o comportamento do metodo getCustomerByVat do CustomerServiceMockito
        when(mockedCustomerServiceMockito.getCustomerByVat(anyInt())).thenReturn(new CustomerDTO(1, 123456789, "Goncalo", 123456789));

        // Testar o metodo getCustomerByVat
        CustomerDTO customerDTO = mockedCustomerServiceMockito.getCustomerByVat(123456789);

        // Verificar se o método do CustomerServiceMockito foi chamado
        verify(mockedCustomerServiceMockito).getCustomerByVat(123456789);

        // Asserts para verificar os dados retornados
        assertEquals(1, customerDTO.id);
        assertEquals(123456789, customerDTO.vat);
        assertEquals("Goncalo", customerDTO.designation);
        assertEquals(123456789, customerDTO.phoneNumber);
    }
}
```

Neste simples teste *testGetCustomerByVat*, começou-se por mockar a classe criada anteriormente (*CustomerServiceMockito)*, e definir o comportamento do método getCustomerByVat da classe mockada. Após este pequeno *Setup*, é finalmente "testado" o método e verificamos com o auxílio do método *verify* disponibilizado pelo *mockito*, se o método *getCustomerByVat* foi de facto chamado. Posteriormente foram feitas asserções para garantir o bom funcionamento do método *getCustomerByVat*, de acordo com o comportamento estipulado acima.

Mudanças Efetuadas

Como foi explicado anteriormente, foi necessário fazer certas alterações ao SUT, nomeadamente a criação de métodos para remover as *sales* e as *delivery sales* de um cliente, e alteração do método *addSale* para que não fosse possível adicionar uma sale com um VAT ou um cliente inválido.

Os métodos para remoção de *sales* e *delivery sales* de um cliente, estão designados como *removeSaleByCustomerId* e *removeDeliverySalesByCustomerId*, respetivamente.

```java
public void removeSaleByCustomerId(int customerId) throws ApplicationException {
    try {
        List<SaleRowDataGateway> sales = new SaleRowDataGateway().getAllSales(customerId);
        for (SaleRowDataGateway sale : sales) {
            sale.deleteSale();
        }
    } catch (PersistenceException e) {
        throw new ApplicationException("Error removing sales for customer with VAT number " + customerId, e);
    }
}


public void removeDeliverySalesByCustomerId(int customerId) throws ApplicationException {
    try {
        List<SaleDeliveryRowDataGateway> deliverySales = new SaleDeliveryRowDataGateway().getAllSaleDelivery(customerId);
        for (SaleDeliveryRowDataGateway deliverySale : deliverySales) {
            deliverySale.deleteSaleDelivery();
        }
    } catch (PersistenceException e) {
        throw new ApplicationException("Error removing delivery sales for customer with VAT number " + customerId, e);
    }
}
```

Para executar este métodos foi preciso criar os devidos métodos nas respetivas classes de RowDataGateway, para as sales foi criado o método *deleteSale* no *SaleRowDataGateway*, que limita-se a apagar da DB a sale indicada, do cliente:

```java
private static final String DELETE_SALE_SQL =
        "delete from sale " +
        "where id = ?";

public void deleteSale() throws PersistenceException {
    try (PreparedStatement statement = DataSource.INSTANCE.prepare(DELETE_SALE_SQL)){
        statement.setInt(1, id);
        statement.executeUpdate();
    } catch (SQLException e) {
        throw new PersistenceException("Internal error deleting sale with ID " + id + ".", e);
    }
}
```

E para as sales deleveries, foi criado no SaleDeliveryRowDataGateway o método deleteSaleDelivery, que também se limita a apagar da DB a sale delivery indicada, do cliente:

```java
private static final String DELETE_SALE_DELIVERY_SQL =
        "delete from saledelivery " +
        "where id = ?";

public void deleteSaleDelivery() throws PersistenceException {
    try (PreparedStatement statement = DataSource.INSTANCE.prepare(DELETE_SALE_DELIVERY_SQL)){
        statement.setInt(1, id);
        statement.executeUpdate();
    } catch (SQLException e) {
        throw new PersistenceException("Internal error deleting sale delivery with ID " + id + ".", e);
    }
}
```

Já no caso da alteração do método *addSale,* passou se a verificar se o VAT era válido com o auxílio do método privado *isValidVat,* foi adicionado um bloco *try catch,* que tenta obter o cliente com o VAT indicado, caso esse cliente não exista é levantada uma *ApplicationException*.

```
public void addSale(int customerVat) throws ApplicationException {
    if (!isValidVAT(customerVat)) //verifica se o VAT é valido
        throw new ApplicationException("Invalid VAT number: " + customerVat);
    else try {
            try {
                CustomerService.INSTANCE.getCustomerByVat(customerVat); //tenta obter o cliente com o VAT
            } catch (Exception e) {
                throw new ApplicationException("Invalid VAT number: " + customerVat); //se não conseguir ou seja se o
                                                                                       //cliente não existir esta excecão é levantada
            }
            SaleRowDataGateway sale = new SaleRowDataGateway(customerVat, new Date()); //se o VAT e o cliente existirem e foram válidos
            sale.insert(); //adiciona a Sale

        } catch (PersistenceException e) {
            throw new ApplicationException("Can't add customer with vat number " + customerVat + ".", e);
        }
}
```

Possíveis melhorias

Para corrigir o bug, por mim identificado, de um novo cliente que é adicionado com o VAT de outro cliente que já tenha sido eliminado, permanece com as moradas do cliente antigo, teríamos de criar uma função que se remove as moradas do cliente, uma vez que a função que elimina cliente limita-se a remover o cliente da tabela CUSTOMERS.

R **Ricardo Sobral** added a new issue 15 days ago

VVS2122_43-4 New Client with the same VAT as an older client maintains the older client's addresses

This bug was found while i was adding and deleting users for test purposes.

Pre-requisites
Existing client with at least one address.

Steps to reproduce
Delete a client, and add another client with the same VAT

Expected result
New client without addresses

Actual result
The new client has the same addresses as the old client.

IssueOldAddresses.pdf

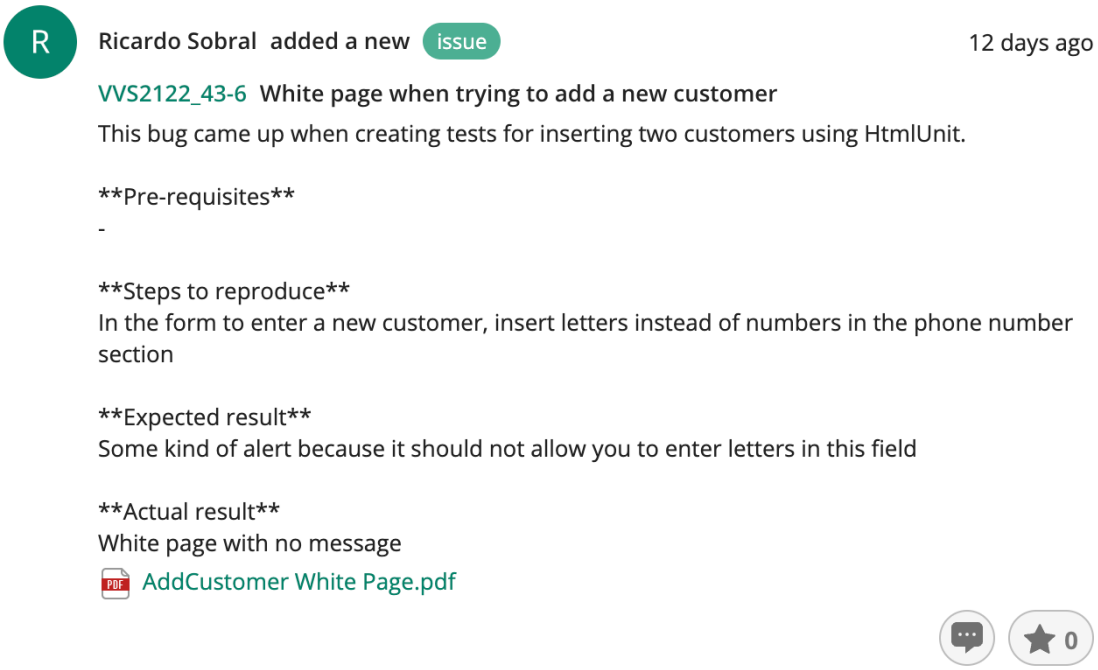
0

Para evitar a criação de mais métodos como foi feita para a remoção de *sales* e *sale deliveries* quando um cliente era removido, também podia ser alterada a estrutura da DB.

Para que esta alternativa funcionasse certas tabelas como na tabela SALE e na SALEDELIVERY e ADDRESS, tinham de passar a ter uma referência ao VAT_NUMBER do cliente que constava na tabela CUSTOMER, através de uma FOREIGN KEY a qual podia ser indiciada com o conceito ON DELETE CASCADE, para que assim um efeito cascata quando um cliente fosse eliminado. Isto permitia que ao eliminar um cliente todas as outras informações associadas a ele também fossem eliminadas.

Em alternativa à maneira que eu utilizei no método *addSale,* para verificar que um cliente existia, e esperar que fosse levantada uma exceção caso ele não existisse, podia ser a criação de um método privado no CustomerService e no SaleService para verificar que um cliente constava na lista de clientes. A criação deste método tornaria o código mais modelar e fácil de ler.

Outra verificação que deveria ser feita, era verificar se ao adicionar um cliente o conteúdo inserido, nomeadamente o número de telefone do novo cliente, era de facto um número e não outro tipo de caracteres. Esta verificação resolveria o outro bug que reportei no backlog:



Isto era algo que podia ser ultrapassado através de uma simples verificação no form do próprio HTML.