Software Testing and Validation 2019/20
Instituto Superior Técnico

Project

Due: April 10, 2020

1 Introduction

The development of large information systems is a complex process and demands several layers of abstraction. One first step is the specification of the problem in a rigorous way. After a rigorous specification of the problem, one may perform formal analysis and prove the correctness of the implementation. If the implementation is not correct, one may discover flaws that would not be easy to detect in any other way. Complementarily, and in particular when a solution has already been implemented, the quality of the solution may be assessed by running tests in order to detect existing flaws.

The project of this course consists of the design of a set of test suites given a specification. This specification models an application that manages a post office. The main goal of this project is for the students to learn the concepts related to software testing and acquire some experience in designing test cases.

Section 2 describes some entities present in the system to test and some implementation details that are important for testing these entities. Section 3 describes the test cases to design and implement and how the project will be evaluated.

2 System Description

The system to test is responsible for managing customers, products and invoices of a post office. All these entities have a unique identifier. The system is responsible for ensuring that no two entities of the same type exist with the same identifier.

2.1 The Invoice entity

The Invoice entity manages a particular purchase to be made by a given customer since the beginning of the interaction with the post office until the payment of the invoice or its cancellation. Figure 1 shows the interface of this entity.

The correct behavior of this class is as follows. When a client desires to make a purchase at the post office, the system creates an instance of `Invoice`. This instance is responsible for maintaining the list of products that the customer wants to buy. When an invoice is created, it is in the *open* mode. While an invoice is open, you can add (or remove) products to the invoice indicating the desired product and its amount. The product is represented by its name (the unique identifier of a product). In case of success (it was possible to add or remove the desired product), then the concerned method returns `true`. If it is not possible to add the product (because it does not exist in the post office or its available quantity is insufficient) or remove the product (because it is not present on the invoice) then the concerned method should have no effect and return `false`.

When a given customer wants to buy the products present in an open invoice, then he must confirm the invoice by invoking the `checkout` method. For security reasons, it is not possible to have invoices with a value greater than 100 euros. Thus, the invoice becomes confirmed if the purchase value is less than or equal to 100 euros. If is higher, the invoice remains open. The `checkout` method returns a boolean value to indicate whether it was

```
enum InvoiceMode {
  OPEN, CONFIRMED, PAYD, WAITING_FOR_PAYMENT, CANCELLED;
}

public class Invoice {

  // creates a new invoice with the specified identifier.
  public Invoice(int id) {/*....*/}

  // adds a product p with the given name and given quantity to this invoice if the invoice is open
  public boolean add(String name, quantity q) { /* .... */ }

  // removes the product whith the specified name  from this invoice if the invoice is open. If the product
  // does not exist in the invoice it does nothing.
  public boolean remove(String name) { /* .... */ }

  // canels a confimerd invoice
  public void cancel() { /* .... */ }

  // confirms this invoice if the invoice is open and ita value is less than 100 euros.
  // Returns true if it was possible to confirm the open invoice, false otherwise.
  public boolean checkout(Client client)  { /* .... */ }

  // pays a confirmed or "waiting for payment" invoice.
  public void pay() {/*....*/}

  // pays a confirmed invoice by credit.
  public void payByCredit() { /* .... */ }

  // computes the current value to pay for this invoice taking into account the client
  // and the number of days of delay.
  float computeCreditBill() { /* .... */ }

  // returns the list of products of this invoice
  List<Product> getProducts() { /* .... */ }

  // returns the mode of this invoice
  InvoiceMode getMode() { /* .... */ }

  ...
}
```

Figura 1: A interface da classe `Invoice`.

possible or not to confirm the invoice. It is possible to cancel a confirmed invoice, making it open again, using the `cancel` method.

A confirmed invoice can be paid in cash (represented by method `pay`) or payment can be made by credit (represented by method `payByCredit`), in which case the customer will have a deadline to pay the invoice. In the first case, the invoice is paid while in the second is awaiting payment. In the latter case, an *awaiting payment* invoice will become paid when the customer finally makes the payment which is done when the `pay` method is invoked on the *awaiting payment* invoice. Method `computeCreditBill` method is responsible for computing the amount to pay for an *awaiting payment* invoice in accordance with the business logic described at the end of this section.

You can always invoke methods `getProducts` and `getMode`. Method `getProducts` returns the list of products of an invoice (and their quantities) while method `getMode` returns the mode of an invoice (*open*, *confirmed*, ...).

You should also consider that any invocation of a method of this entity in a case not described before corresponds to an invalid invocation and should lead to throwing the *InvalidOperationException* exception.

The method `computeCreditBill`, which calculates the amount to pay for a given purchase made by credit, takes into account the number of purchases made by the client, the total value of the purchases and time taken to pay the invoice. A purchase made by credit has 14 days to be paid without any penalty. First, this method determines the base cost of the purchase. This cost is equal to the sum of the unit cost of each product present

in the purchase multiplied by the respective quantity. The method then determines the discount or penalty to be applied to this base cost as follows:

- If the purchase is paid on time, there is no penalty and the discount to be applied depends on the number of purchases already made by the client. If it is between 30 and 15, the discount is 3%. If it is less than 15, the discount is 1%. Finally, if this number exceeds 30, the discount depends on the total value of the purchases made by the client. If this value is greater than to 1000 euros, then the discount is 10%, otherwise, it is 6%.
- If the invoice is paid only one day late, then clients with more than 30 purchases have a discount of 2%. If the number of purchases is less than or equal to 30, then there a penalty of 4% (which can be seen as a -4% discount).
- Finally, if the number of days of delay is greater then one, then the penalty to apply is equal to 10%.

2.2 The Client entity

Each client of `PostOffice` has a name (string) and is identified by his social security number (integer). A client also keeps the purchases already made at the post office. This entity is represented by the class Client, shown in Figure 2.

```
public class  Client {
  // Creates a client with the given name and unique identifier
  public Client(String name, int id) { /* .... */ }
  // adds an invoice bought by this client
  void addInvoice(Invoice v) { /* .... */ }
  // returns the name of this client.
  public void getName() { /* .... */ }
  // returns the unique identifier of this object.
  public int getId() { /* .... */ }
  ...
}
```

Figura 2: A interface da classe *Client*.

2.3 The Product entity

There are several products for sale at a post office, e.g. stamps, books and cd's. Each product has a name, a product description, a price and a critical amount (used in inventory management of the post office). Figure 3 presents the interface of this class.

```
public class Product {
  // creates a product
  public Product(String name, String description, int price, int criticalValue) { /* .... */ }

  public String getDescription() { /* .... */ }

  public int getPrice() { /* .... */ }

  public int setPrice(int newPrice) { /* .... */ }

  public int getCurrentQuantity() { /* .... */ }

  public void store(int numberOfUnits) { /* .... */ }

  public String getName() { /* .... */ }
  ...
}
```

Figura 3: A interface da classe *Product*.

2.4 The PostOffice entity

A post office is responsible for the management of the products it offers. A post office maintains the stocks of each product offered by the post office. In the scope of a post office, each product is identified by its name. Therefore, it is impossible two have two products with the same name registered in the same post office. The post office stores the number of units, critical quantity, price and name of each product offered by the post office. Figure 4 shows the interface of the class that represents a post office.

Distinct post offices may have different storage capacity. The total amount of products presented at a post office cannot exceed a given threshold. This limit depends on the post office. This maximum number of products can vary between 2 and 20 and it is specified when you create a post office. When you create a post office, you also have to give the initial set of products to be available in the new post office. Method `setMaxNumberOfProducts` can change this maximum number of products. The unit price and the number of units of a product cannot be a negative number.

It is possible to update the price and number of units of a given product using the `update` method. Method `addNewProduct(Product p)` adds a new product, referenced by p, to the post office if none of the conditions described before was violated. Additionally, product p should not already be part of the inventory of the concerned post office. It is also possible to remove a product (identified by its name) from the inventory of a post office if none of the conditions described before was violated and the number of units of the product to remove is equal to 0. Methods `addNewProduct`, `removeProduct`, `updateProduct` and `setMaxNumberOfProducts` return all a boolean value that show if the execution of the method was successful or not. When the execution of the method is invalid (e.g., update the price of an existing product of a post office to a negative value), then the method returns false and it should not have any effect in the concerned post office.

```
public class PostOffice {
  public PostOffice(int maxNumberOfProducts, List<Product> products) { /* .... */ }

  // adds a new product to this post office.
  public boolean addNewProduct(Product p) { /* .... */ }

  // updates quantity and/or price of the given product
  public boolean update(String productName, int newPrice, int newQuantity) { /* .... */ }

  // removes a product from this post office
  public boolean removeProduct(String productName) { /* .... */ }

  public boolean setMaxNumberOfProducts(int newMaxQuantity) { /* .... */ }

   // accessor methods
  public List<Product> getProducts() { /* .... */ }
  public int getMaxNumberOfProducts() { /* .... */ }
  public void getMaxNumberOfProducts() { /* .... */ }
  ...
}
```

Figura 4: A interface da classe *PostOffice*.

3 Project Evaluation

Students should develop the required test cases applying the most appropriate test patterns. The test cases to design are the following:

- Test classes *Invoice* and *PostOffice* at class scope.
- Test methods *addProduct* of class *PostOffice* and *computeCreditBill* of class *Invoice*.
- Finally, it is necessary to implement six test cases of the test suite that exercises the class *PostOffice* at class scope. This implementation should use the *TestNG* testing framework.

The project will be evaluated as follows:

1. Development of the test cases for class *PostOffice*: 0 to 3.5.
2. Development of the test cases for class *Invoice*: 0 to 6.5.
3. Development of the test cases for method *addNewProduct*: 0 to 3.5.
4. Development of the test cases for method *computeCreditBill*: 0 to 3.5.
5. Implementation of six test cases concerning the test suite that tests class *PostOffice* using the *TestNG* framework: 0 to 3.0.

For each method or class under testing, it is necessary to show the following:

- The name of test pattern applied.
- If applicable, the result of the different stages of the application of the test pattern using the format described in the lectures.
- Description of the test cases that result from the application of the chosen test pattern.

If one of the above items is only partially developed, the grade will be given accordingly.

3.1 Other Forms of Evaluation

It may be possible a posteriori to ask the students to individually develop test cases similar to the ones of the project. This decision is solely taken by the professors of this course. Students whose grade in the exam is lower than this project grade by more than 5 will have to make an oral examination. In this case, the final grade for the project will be individual and will be the one obtained in this evaluation.

3.2 Fraud and Plagiarism

The submission of the project presupposes the commitment of honor that the project was solely executed by the members of the group that are referenced in the files/documents submitted for evaluation. The forfeit of this commitment, i.e., the appropriation of work done by other groups, will have as consequence the exclusion of the involved students (including those that allowed the appropriation) from this course.

3.3 Handing-in the Project

The project is due April 10^{th}, 2020 at 23:59. The protocol for handing in the project will be available soon in section *Project* of the webpage of the course.

4 Final Remarks

All information regarding this project will be available in section *Project* of the webpage of the course. Extra material such as links, manuals, and FAQs will also be available in that same section.

Have a Good Work!