Software Testing and Validation 2022/23
Instituto Superior Técnico
Project

Due: May 26, 2023

1 Introduction

The development of large information systems is a complex process and demands several layers of abstraction. One first step is to have a rigorous specification of the problem. Having a rigorous specification of the problem, one may perform a 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 to detect existing flaws.

The project of this course consists of the design of a set of test suites given a specification of the main entities of the system to test. 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.

This document is organized as follows. Section 2 describes the entities present in the system under test and some implementation details concerning these entities that are important for testing them. Section 3 describes what should be tested and how the project is going to be evaluated.

2 System Description

The system to test it is a manager of a network of communication terminals. Generally, the program manages the clients, terminals, and communications of the network. Specifically, the system provides several services to its users, including: (i) registering clients; (ii) registering terminals; (iii) registering data about communications made; (iv) searching for communications made, and (v) accounting for the balance associated with terminals. The main entities of this system are: `TerminalNetwork`, `Terminal`, and `Client`. These entities are described below.

2.1 The Client entity

Each client has a name, and a level, and knows their terminals. The `Client` class represents this entity. Figure 1 shows the interface of this class.

The `computeCallUnitCost()` method is responsible for computing the unit cost of voice communications (cost in cents per minute). The computation of this cost depends on the client's level, the calls and *SMS*'s made by the client, and the number of terminals the client has. This cost is determined as follows:

- Client with the *Platinum* level: If the number of terminals the client has is greater than or equal to 3, then the cost is 5 if the number of calls is greater than or equal to 500, and 7 otherwise. If the number of terminals is less than 3, then the cost is 8 if the number of calls is greater than or equal to 500. If the number of calls is less than 500, then the cost is 12 if the number of *SMS*'s is less than 1000, or 10 if the number of *SMS*'s is greater than or equal to 1000.
- Client with the *Gold* level: If the number of calls is less than 500, then the cost is 25. If the number of calls is greater than or equal to 500, then the cost depends also on the number of *SMS*'s. If the number of *SMS*'s is greater than or equal to 600, the cost is 16, otherwise, the cost is 20.

```
public enum ClientLevel { NORMAL, GOLD, PLATINUM }

public class Client {

  public Client(String name, ClientLevel level)    // creates a client with the given name and level
  { /* .... */ }

  void setLevel(ClientLevel level) { /* .... */ }  // changes the level of this client

  public ClientLevel getLevel() { /* .... */ }     // gets the level of this client

  public String getName() { /* .... */ }           // returns the name of this client

  void addTerminal(Teminal t) { /* .... */ }       // adds a terminal to this client

  List<Terminal> getTerminals() { /* .... */ }     // returns the terminals of this client

  // returns the unpaid communications made by terminals of this client
  List<Communication> getUnpaidCommunications() { /* .... */ }

  // computes the unit call of voice communications made by these client
  public int computeCallUnitCost() { /* .... */ }
  ...
}
```

Figure 1: Interface of class *Client*.

- Client with the *Normal* level: If the number of calls is less than 700, then the cost is 36, otherwise, the cost is 30.

2.2 The TerminalNetwork entity

The network of terminals is represented by the `TerminalNetwork` class. The network maintains the list of clients and terminals registered on the network. Figure 2 shows the interface of this class.

```
public class TerminalNetwork {
  // creates an empty TerminalNetwork with a given name and maximum number of clients
  public TerminalNetwork(String name, int maxClients)   { /* .... */ }

  public void setMaxClients(int maxClients) { /* .... */ }   // changes the maximum number of clients
  public int getMaxClients() { /* .... */ }                  // returns the maximum number of clients

  public String getName() { /* .... */ }                     // returns the name of this network
  public String setName(String n) { /* .... */ }             // changes the name of this netwotk

  public void addClient(Client cl) { /* .... */ }            // adds a client to this network
  public void removeClient(Client cl) { /* .... */ }         // removes a client to this network

  public List<Client> getClients()  { /* .... */ }           // returns the clients of this network
  public List<Terminal> getTerminal()  { /* .... */ }        // returns the terminals of this network

  // register a terminal to this network and associates it to client cl
  public void addTerminal(Terminal t, String clientId) { /* .... */ }

  ...
}
```

Figure 2: Interface of class *TerminalNetwork*.

A terminal network has a maximum number of clients, which is indicated at the time of creation and can be changed later. The number of clients cannot exceed 50000, and the name of each client is a unique identifier within the context of the terminal network. A terminal network has a name, and the number of characters in the name must be greater than or equal to 3 and less than 10. Each terminal in a terminal network is associated with a client registered on the same network. If the invocation of one of the methods of the `TerminalNetwork` class invalidates any of these conditions, the method in question should have no effect and should throw the *InvalidInvocationException* exception.

2.3 The Terminal entity

The behavior of a terminal is implemented by the `Terminal` class. Figure 3 shows the interface of this class. A terminal can establish a voice communication (i.e., make a call) with another terminal or send and receive *SMS* messages. An *SMS* can only be sent if ts length is between 10 and 200 characters. If the length is invalid, then the *SMS* sending operation is aborted and the *IllegalArgumentException* exception is thrown. Each terminal stores the *SMS* messages sent and received successfully. A terminal also keeps information regarding voice communications (calls) initiated by the terminal and successfully established.

A terminal can be in various modes: `off` or `on`. When `on`, it can be idle (`idle`), in silence (`silence`), or busy (`busy`). A terminal that is `on` but it is not busy can send *SMS* messages and make calls. The big difference between an `idle` terminal and a `silent` terminal is that the latter does not accept calls. A terminal that is `on` can receive *SMS* messages. The correct behavior of this class is described below. Consider that any invocation of a method of this class in an undescribed situation corresponds to an invalid invocation and should result in the *InvalidInvocationException* exception being thrown.

```
public class Terminal {

  Terminal(String id) { ... }

  public void turnOff() { ... }
  public void turnOn()  { ... }
  public void setSilence()  { ... }

  // if size of msg is invalid, then it throws IllegalArgumentException
  public boolean sendSMS(String msg, Terminal to)  { ... }
  void receiveSMS(String msg, Terminal from) { ... }

  public void makeCall(Terminal t) { ... }
  void acceptCall(Terminal t) { ... }
  public void endCall() { ... }
}
```

Figure 3: Interface of class *Terminal*.

When a terminal is created, it is turned off. A terminal can be turned on by invoking the `turnOn()` method, which puts it into the `idle` mode. A terminal in the `idle` or `silence` mode can be turned off by invoking the method `turnOff()`, which puts the terminal in the `off` mode. A non-busy turned-on terminal can send an *SMS* to another turned-on terminal through the `sendSMS(String msg, Terminal t)` method. The return value of this method indicates whether the *SMS* was successfully delivered or not. If the length of the *SMS* is valid, and the destination terminal is turned on, the method returns `true`. If the length of the *SMS* is valid but the destination terminal is invalid, the method returns `false`. Finally, if the length of the *SMS* is invalid, then this method throws the `IllegalArgumentException`. The delivery of an *SMS* to the destination terminal is handled by the method `receiveSMS(String msg, Terminal from)`, which is valid only if the terminal in question is turned on.

A non-busy turned on terminal can also make a call to another terminal through the `boolean makeCall(Terminal t)` method, provided that the destination terminal is in the `idle` mode. In this case, communication is established between both terminals, and the calling terminal goes into the `busy` mode. An `idle` terminal can accept an incoming call through the `acceptCall()` method, which puts it into the `busy` mode. The call ends when the `endCall()` method is invoked on both terminals and each terminal returns to the previous mode (the mode before the call was made). Finally, it is possible to put an `idle` terminal into the `silence` mode by invoking the method `setSilence()`, and put a terminal in the `silence` mode back into the `idle` mode through the `turnOn()` method. At any time, it is possible to know the mode of the terminal through the `getMode()` method.

3 Project Evaluation

All test cases must be determined by applying the most appropriate testing design strategy. The test cases to design are as follows:

- Class scope test cases for the `TerminalNetwork` class. Worth between 0 and 3.5 points.
- Class scope test cases for the `Terminal` class. Worth between 0 and 6.5 points.
- Method scope test cases for the `sendSMS(String msg, Terminal to)` method of the `Terminal` class. Worth between 0 and 3.5 points.
- Method scope test cases for the `computeCallUnitCost()` method of the `Client` class. Worth between 0 and 3.5 points.

- Additionally, it is necessary to implement 8 test cases (4 success cases and 4 failure cases) from the test suite designed to exercise the `TerminalNetwork` class at class scope. You should use the *TestNG* testing framework to implement these test cases. Worth between 0 and 3 points.

If the *Category Partition* testing pattern is applied, and if the number of generated test cases is greater than 30, only the first 30 combinations need to be presented and the total number of combinations must be indicated. For each method or class to be tested, the following must be indicated:

- The name of the used test pattern.
- If applicable, present the results of the several steps of the test design strategy applied, using the format described in the theoretical classes.
- The description of the test cases resulting from the chosen testing strategy.

If any of the points mentioned above are only partially satisfied, the grade will be given in proportion to what was accomplished.

3.1 Fraud and Plagiarism

The submission of a project presupposes the **commitment of honor** that the project was solely done by the the students referenced in the files/documents submitted for evaluation. Breaking this commitment will result in the failure of all students involved (including those who made the occurrence possible) in the course of Software Testing and Validation in this academic year.

4 *Final Remarks*

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 grade obtained in this evaluation.

All information regarding this project will be available in section *Project* of the course's webpage. The project delivery protocol is described in this section.

HAVE A GOOD WORK!