Software Testing and Validation 2024/25
Instituto Superior Técnico

Project

Due: May 23rd, 2025

1 Introduction

The development of large information systems is a complex process that requires various levels of abstraction. A first step is the rigorous specification of the problem. Then, one should start coding the solution and testing it simultaneously in order to detect errors as early as possible.

The Software Testing and Validation course project consists of designing test cases based on a provided specification. This specification describes some entities of a given application. The main learning objective of this project is to give students hands-on experience designing test cases by applying the test case design techniques taught in lectures to a set of classes and methods whose behavior is described in this document.

This document is organized as follows. Section 2 describes the entities that participate in the system under test, as well as some implementation details that are important for designing the test cases. Section 3 identifies the tests to be carried out and explains how the project will be evaluated.

2 System Description

The system under test is responsible for managing a network of communication terminals. In general, the system should allow for the management and query of clients, terminals, and communications. The main entities of the system are described below.

2.1 The Client Entity

A client has a name with a maximum of 40 characters. The number of terminals a client may have can range from 1 to 9. A client can have *friends*, who are other clients. The maximum number of friends a client can have is determined by the number of terminals it owns: $5 * \#terminals - 3$. A client's friends list is initially empty, and a client cannot be friends with themselves. Additionally, a client has a certain number of points. Each client has a point count that ranges from 0 to 200. When a client is created, they start with 20 points. The number of points a client has affects the cost of communications.

The `Cliente` entity is represented by the class `Client`. Listing 1 shows the interface of this class.

The method `removeTerminal()` removes a terminal, if possible, from the client's list of terminals. The method returns a value indicating whether the removal was successful. A terminal can only be removed if it belongs to the client's list and has a non-negative balance. If the terminal does not belong to the client, the method does nothing and returns false. You also have to consider that the invocation of any method of this class that would put the invoked client in an invalid state should be aborted by throwing the `InvalidOperationException` exception, and the client's state must remain unchanged.

2.2 The Terminal Entity

The *Terminal* entity enables communications (voice or text) between two terminals: one initiates the communication, and the other receives it. For voice communication, both terminals maintain information about the

Listing 1: The interface of class *Client*.

```
package prr.core;

/**
 * This class represents a client.
 **/

public class Client {
  // Creates a client with a single terminal. The provided terminal cannot be null
  public Client(String name, int taxNumber, Terminal term) { /* ... */ }

  public void updateName(name n) { /* ... */ }

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

  // updates the number of points of the client. It can be a positive or negative number.
  public void updatePoints(int p) { /* ... */ }

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

  public void addFriend(Client c) { /* ... */ }

  public boolean removeFriend(Client c) { /* ... */ }

  public boolean hasFriend(Client c) { /* ... */ }

  public void addTerminal(Terminal terminal) { /* ... */ }

  public boolean removeTerminal(Terminal terminal) { /* ... */ }

  // returns the number of terminals of this client
  public int numberOfTerminals() { /* ... */ }
}
```

ongoing call. Each terminal is responsible for maintaining its own balance. The balance is defined as the difference between the value of communications made and the payments made by the client. Communications are charged based on the network's tariff. Each terminal has a unique key (a numeric string) within the network and is associated with a client. Listing 2 shows the interface of the class representing the *Terminal* entity.

A newly created terminal has a balance of zero. The following restrictions apply to method invocations in the `Terminal` class:

- When a terminal is created, it is *turned off* (`OFF`). In this mode, it can be turned on (`turnOn()`). When turned on, the terminal enters `NORMAL` mode. A payment (`pay()`) can only be made when the terminal is off and the amount paid is at least 5 (the amount is given in cents).
- A powered-on terminal may be in normal (`NORMAL`), silent (`SILENT`), or busy (`BUSY`) mode. The method `toggleOnMode()` allows switching between normal and silent mode. A terminal that is powered on and not busy can send text messages (`sendSMS()`) to any terminal, including itself. Sending a text message is instantaneous, but whether the message is received depends on the recipient terminal's state. A terminal can only receive messages (`receiveSMS()`) if it is powered on. If it is in silent mode (regardless of being busy), an additional restriction applies: the client of the terminal sending the message must be a friend of the recipient's client.
- A terminal that is powered on and not busy can initiate a voice call (`makeVoiceCall()`) if the target terminal can receive calls. A terminal may accept a voice call (`acceptVoiceCall`) only if it is in `NORMAL` mode. If the call is established, both terminals become busy and return to their previous states after the call ends (`ongoingCommunicationEnded()`).
- The balance of a terminal (`balance()`) can be queried only if the terminal is not busy. A powered-on and idle (`NORMAL` or `SILENT`) terminal can be turned off using the `turnOff()` method.

The `getMode()` method is always valid, independently of the mode of the terminal, and it should return the terminal's mode. If a method is invoked outside the allowed context, it must have no effect and throw the *InvalidInvocationException* exception.

Listing 2: The interface of class *Terminal*.

```
public enum TerminalMode { OFF, NORMAL, SILENT, BUS; }

public class Terminal {

  // creates a terminal with a given identifier and associated to the given client.
  Terminal(String id, Client client) { /* ... */ }

  // Returns the mode of this terminal
  public final TerminalMode getMode() { /* ... */ }

  // Decreases the debt of this terminal by the given amount. The amount must be a number greater than 5 cents
  public void pay(int amount) { /* ... */ }

  // returns the balance of this terminal
  public int balance() { /* ... */ }

  // send a SMS to terminal to with text msg. Returns if the SMS was successfully delivered.
  public boolean sendSMS(Terminal to, String msg) { /* ... */ }

  // receives a SMS from terminal from with text msg
  public void receiveSMS(Terminal from, String msg) { /* ... */ }

  // start a voice call with tetminal to
  public void makeVoiceCall(Terminal to) { /* ... */ }

  // to invoke over the receiving terminal of a voice call (represented by c). The voice
  // call is established if the terminal accepts the call, otherwise it throws an exception.
  void acceptVoiceCall(Communication c) { /* ... */ }

  // turns on this terminal
  public void turnOn() { /* ... */ }

  // turns off this terminal
  public void turnOff() { /* ... */ }

  // toggles the On mode: normal to silent or silent to normal
  public void toggleOnMode() { /* ... */ }

  // Ends the ongoing communication.
  public void endOngoingCommunication() { /* ... */ }
}
```

2.3 The Communication Entity

Communications between terminals are represented by the `Communication` class. A completed communication has a defined size. The size of a text communication equals the number of characters in the message (measured in hundreds, rounded up). For voice communications, the size is the duration (in seconds). Listing 3 shows the interface of this class:

The cost of communication depends on its type, duration, the number of points, and the number of friends of the client who initiated it. The cost is calculated as follows:

- If the size is 0, the cost is 0 cents;
- If the size is less than 10 and the client has more than 100 points, the cost is 1 cent. If the client has 100 points or fewer, the cost is 2 cents.
- If the size is 10 or more and less than 120, the cost depends on the client's points and the communication type. If the client has fewer than 75 points, the cost is 6 cents for a text and 12 cents for a voice call. If the client has at least 75 points, the cost is 4 cents for a text. For a voice call, if the client has fewer than 4 friends, the cost is 8 cents. Otherwise, it is 5 cents.
- If the size is 120 or more and the client has fewer than 150 points, the cost is 15 cents. Otherwise, it is 12 cents.

Listing 3: The interface of class *Communication*.

```
package prr.core;

public enum CommunicationType { SMS, VOICE; }

/**
 * This class represents a communication (text or voice) made between
 * two terminals.
 **/
public class Communication {

  private Communication(CommunicationType type, Terminal to, Terminal from) { /* ... */ }

  public static Communication textCommunication(Terminal to, Terminal from, int length) { /* ... */ }

  public static Communication textCommunication(Terminal to, Terminal from) { /* ... */ }

  public void duration(int duration) { /* ... */ }

  public Terminal to() { /* ... */ }

  public Terminal from() { /* ... */ }

  double computeCost() { /* ... */ }

  public double getCost() { /* ... */ }
}
```

3 Project Evaluation

All test cases must be designed using the most most appropriate testing design strategy for each situation. The project consists of specifying test cases to test some methods and classes at the class level. Some test cases must also be implemented. The test cases to be specified are:

- Class scope test cases for the `Client` 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 `computeCost()` method of the `Communication` class. Worth between 0 and 3.5 points.
- Method scope test cases for the `removeTerminal()` method of the `Client` class. Worth between 0 and 3.5 points.
- Additionally, it is necessary to implement 8 test cases (4 *success* test cases and 4 *failure* teste cases) from the test suite designed to exercise the `Client` 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* test design strategy is used and the number of combinations exceeds 30, only the first 30 need to be presented, along with the total number of combinations. For each method or class under test, the following must be indicated:

- The name of the test design strategy used;
- Where applicable, the result of each step of the applied strategy using the format presented in lectures;
- The description of the test cases resulting from the chosen testing strategy (typically the result of the final step).

If any of the above points are only partially met, the score will be proportional to the completion. 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:

3.1 Fraud and Plagiarism

Submitting a project implies a **statement of honor** that the submitted work was done by the students listed in the files/documents. Violation of this agreement—i.e., attempting to appropriate work done by others—will result in failing the course for all involved students (including those who enabled it).

3.2 Final Remarks

A discussion may be held to assess students' ability to perform tests similar to those in the project. This decision is at the discretion of the teaching staff. Students whose exam score is more than 5 points lower than the project score may be required to attend a discussion. If a discussion takes place, the final project grade may be individualized and will be based on the discussion outcome.

All information regarding the project will be available on the course webpage under the *Project* section. The submission protocol is also provided there.

Have a Good Work!