Instituto Superior Técnico

# Teste e Validação de Software

School Year 2017/2018 — 2nd Semester

2nd Test **A** - 2018 June, 14th — Duration: 1:30 hours

- Clearly identify the first page of this test.
- Each answer should be given using solely the blank space that follows it.

**I. (2.0+0.75+0.75 = 3.5 val.)**

a) Draw the control flow graph for the following method, clearly identifying its several code segments and the several *du-pairs* for the *m* variable. It may be useful to show in each segment if the variable was defined and/or used.

```
int computeSomething(int a, int b, int c) {
  int m = a + b;

  if (b == 0 && c == 0) {
    m = a;
  } else {
    if (m >= 0)
      m = b;

    if (c >= b)
      m = m + b + c;
  }

  return m;
}
```

**b)** Show a minimal set of paths that achieves *All-Defs* coverage for this method concerning just variable *m*.

**b)** Show a minimal set of paths that achieves *All-Uses* coverage for this method concerning just variable *m*.

Number:_____ Name:_____

**II. (3.5 val.)**

Consider the following class diagram


that represents the association between the *Employee* and *Company* entities for a given application. Each employee has a name and a salary. A company also has a name and it has a cost equal to the sum of the salaries of all of its employees. These classes have the following interfaces:

```
public class Company {
   /*...*/
  public Company(String name)  {/*...*/}

  // return the sum of the salaries of all
  // employees of this company
  public int getTotalSalaries() {/*...*/}

  // add an employee to this company
  public void add(Employee emp)  {/*...*/}

  // remove an employee from this company
  public void remove(Employee emp)  {/*...*/}

  // remove this company from the system
  public void destroy() {/*...*/}

  // return all existing companies
  static Company[] getAllCompanies() {/*...*/}
}
```

```
public class Employee {
   /*...*/

  public Employee(String name, int salary, Company comp)
  {/*...*/}

  // change the salary of this employee
  public void setSalary(int newSalary) {/*...*/}

  // change the name of this employee
  public void setName(String name) {/*...*/}

  // remove this employee from the system
  public void destroy() {/*...*/}

  // return all existing employees
  static Employee[] getAllEmployees()
  {/*...*/}
}
```

Consider that it is possible to directly remove instances of *Employee* and *Company* from the system using the *destroy* method. When an instance of *Company* is removed then all of its employees should also be automatically removed. Apply the correct test pattern and specify the several test cases that should be implemented to check the correct implementation of the association between these two classes.

**III. (5.5 val.)**

**a) a)** Suppose that you have developed a financial application. In this application the entity *Money* represents a certain amount of money and it is implemented by the following class:

```
public class Money {
  private int amount;

  public Money(int am) { amount = am; }
  public String toString() { return "" + amount; }
  public Money add(Money m) {
    return new Money(amount + m.amount);
  }
}
```

This class was tested and no bugs were found. Now, it is necessary to develop new functionalities for this application. These new functionalities require to deal with several amount of money that may belong to different currencies (e.g., euro, dollar, etc, ...). The currency should be represented by the following enumerated type:

```
public enum Currency {
 EUR, USD, CHF;
}
```

To be able to deal with operations that involve money in different currencies it is needed to develop a new entity that can determine the exchange rate between two currencies. This entity is represented by the following interface:

```
public interface CurrencyExchange {
  // returns the currency exchange rate betwen currencies to and from. Throws ExchangeCurrencyServerException
  // if this entity is not able to compute the currency exchange value.
 float computeExchange(Currency to, Currency from) throws ExchangeCurrencyServerException;
}
```

To support the new requirements, you have to implement new functionalities in the *Money* class: (i) each instance of this class has a given currency represented by the *Currency* type. It should be possible to know the currency of each instance of *Money*, which was specified when the instance was created; (ii) add two instances with the same currency (via the *add* method) must return a new instance of *Money* with the same currency and with an amount of money equal to the sum of the two instances of *Money* to add; (iii) If the two instances have distinct currencies then a new instance of *Money* is returned with the same currency of the object on which the *add* method was invoked. The amount of money of the new instance is equal to the sum of the amount of money of the first instance with the amount of money of the second instance converted to the currency of the first instance. The exchange rate is calculated using the *ExchangeCurrency* entity; (iv) Finally, if the invocation of *computeExchange* gives an error, then the *add* method should throw the exception *CannotProcessOperationException*.

For the sake of simplicity suppose that the *add* method of *Money* now receives an additional argument: the instance of *CurrencyExchange* to use if needed. You can consider that the *CannotProcessOperationException* and *ExchangeCurrencyServerException* exceptions are already implemented.

Apply the Test Driven Development (TDD) methodology and the *JMockit* framework to implement the new functionalities described above. Consider that the class that implements the *CurrencyExchange* interface is not developed yet and thus you have to use a test double to represent the functionality of this entity. Your answer should present the several steps performed during the application of the TTD, showing the new code and actions performed in each step.

**III. (1.5 + 1.5 = 3 val.)**

**a)** Describe the *Round-trip Scenario Test* pattern.

**b)** What is the goal of regression testing? When do you need to apply regression testing? Describe the test pattern *Retest Changed Code*.

**V. (2.25 + 2.25 = 4.5 val.)**

**a)** Describe the *Symbolic Execution* testing technique. Show also its main advantages and disadvantages.

**b)** Discribe the *Model Checking* testing technique, showing also its main advantages and disadvantages. Describe how the solution *Java PathFinder* (presented in paper Model Checking Java Programs) solve some of the limitations of this technique.