Instituto Superior Técnico

Teste e Validação de Software

School Year 2019/2020 — 2nd Semester

1st Exam - June 25, 2020 — Duration: 2:00h

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

I. (1.0+0.5+0.5+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* and corresponding *du-paths* for the z variable. It may be useful to show in each segment if the variable was defined and/or used.

```
public float doSomething(Scanner in) {
  float x = in.nextFloat();
  float y = in.nextFloat;
  float z = in.nextFloat;

  if (y > 0 && x > 0)
    z = z + y + x;

  if (z != 0)
    x = y / z;
  else
    x = y;

  return z + x;
}
```

b) Show a minimal set of paths that achieves path coverage for this method.

c) Show a minimal set of paths that achieves branch coverage for this method.

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

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

II. (4.0 val.)

Consider the following class *Baby*:

```
public class Baby {
  public Baby() { ... }

  public void play() throws InvalidOperation { ... }
  public void cry() { ... }
  public void wakeUp() throws InvalidOperation { ... }
  public void eat() throws InvalidOperation { ... }
  public void haveHungry() throws InvalidOperation { ... }
  public void sleep() throws InvalidOperation { ... }

  public String getState() { ... }
}
```

Thos class represents, in a very simplified way, the behavior of a baby. The behavior of this class is as follows: A baby may be happy, sad, hungry or sleeping. A happy, sad or sleeping baby that starts to cry (represented by the method `cry()`) then he gets sad. A hungry baby that cries is still hungry. A happy or sad baby that gets hungry (represented by the `haveHungry()` method) becomes hungry. A happy or sad baby who is sleepy (represented by the `sleep()` method) falls asleep. A happy or sad baby who starts playing (represented by the `play()` method) gets happy. A sleeping baby who wakes up (represented by the `wakeUp ()`) gets happy. A hungry baby who starts eating (represented by the `eat()` method) becomes happy. Finally, when a baby is created, the bay is happy. The `getStatus()` method is valid in any mode and returns the state of the baby (happy, sad, ...). You should also consider that any invocation of a method of this class in a case not described before corresponds to an invalid invocation and should lead to throwing the *InvalidOperationException* exception.

Draw the test suite that checks the behaviour of this class using the appropriate class scope test pattern. Describe the various steps of the applied test pattern.

III. (3.5 val.)

a) The `UserManager` class is responsible for managing users of a given system. Each user is identified uniquely in this entity by its *username*. The maximum number of users managed by this entity is specified at the creation time of this entity. The creation of new users is handled by following method of this class

```
public User createUser(String username, String password)
                throws InvalidUserOrPassword
```

This method creates, if possible, a new user with the unique *username* and *password* and returns the user created if successful. This method also takes into account that the *username* should be between 5 and 10 characters and the *password* should have at least 15 characters. If it is not possible to create the user, then this method should throw the exception `InvalidUserOrPassword`.

Using the test pattern more appropriate, and describing the several steps of the test pattern applied, determine the test suite that checks the correct behaviour of this method. You also need to implement two test cases of this test suite (one successful and another one unsuccessful) using the *TestNG* framework. For the implementation of theses test cases, you can assume the existence of any method for the *User* and *UserManager* classes that you deem necessary.
If you apply the *Combinational Function Test* or the *Category Partition Test* to solve this question, then in the **last step of each pattern** you only need to design the domain matrix for the first two variants (*Combinational Function Test*), or show the first 10 test cases resulting from the application of the *Category Partition Test*.

IV. (4.0 + 1.0 = 5.0 val.)

a) Consider the following application domain. An employee is represented by the *Employee* class. This class has two attributes, *name* and *salary*, and two methods for returning the values of these attributes. Information concerning the existing employees is persistently kept by the entity *EmployeeDAO*, which has the following interface:

```
public interface EmployeeDAO {
  public Employee[] findAll() throws NoExistingEmployeeException;
  public void addEmployee(Employee e);
}
```

The *findAll* method returns all existing employees. If there is no employee registered persistently, then this method throws the exception *NoExistingEmployeeException*.

Consider that you are developing the following class `EmployeeManager`:

```
public class EmployeeManager {
  private EmployeeDAO dao;

  public EmployeeManager(EmployeeDAO d) { dao = d; }
  public List<Employee> getEmployees(float sal) throws InvalidArgumentException { ...}
}
```

This class is associated with the entity *EmployeeDAO* in order to able to access the total set of employees. The association is performed when we create an instance of `EmployeeManager`. The `getEmployees` method of this class has the responsibility of determining the group of employees whose salary is less than or equal to the value of the`sal` parameter.

Regarding the `getEmployees` method of this class, you have to implement the following three test cases: (i) verify that only the employees registered in `EmployeeDAO` whose salary is less than or equal to the `sal` parameter are returned by the method; (ii) if there is no employee persistently stored, then this method should return the null reference; (iii) if the value of the`sal` parameter is less than or equal to 0, then it must throw the exception `InvalidArgumentException`. In the implementation of these test cases, the class `EmployeeManager` class must be exercised in isolation from its dependents. To be able to achieve this isolation, you have to use the emph JMockit framework. You can consider that all the exception classes and the class `Employee` are already implemented.

b) In the Test Driven Development methodology it is essential to write the test first and then the functionality. Why this should be made by this order and not the other way around (Write the functionality and then the test)?

IV. (2.0 + 1.0 + 1.0 = 4.0 val.)

a) Model checking is an automatic technique to perform software verification. Describe generically this technique, presenting its mais advantages and disadvantages. A possible implementation of this technique is presented in the paper M̈odel Checking Programs'.̈ How have this approach coped with the drawbacks of this technique?

b) Consider the `public void doSomething (RootClass obj)` method, where `RootClass` is the root class of a class hierarchy and this hierarchy was not developed by yourself. Suppose now that you want to test this method taking into account its dependency on the `RootClass` hierarchy. Which test pattern should be applied in this case? Describe it.

c) A problem that often exists in the design and implementation of test cases test is that you have a budget for this task, and therefore it is necessary to distribute the effort of creating the test cases by the whole application. What should be the best approach to follow in this case? Describe it, showing its main disadvantages.