Software Testing and Validation 2023/24
Instituto Superior Técnico

Project

Due: May 10th, 2024

1 Introduction

The development of large information systems is a complex process and demands several layers of abstraction. The 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 specification. Afterwards, one can start to implement the system, and the quality of the solution may be assessed by running tests to detect existing flaws.

The project for this course consists of designing 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 be tested is responsible for generating exams from a model and then managing the exams taken by students. Exams are composed of multiple-choice questions. The main entities of this system are described in the remaining of this section.

2.1 The Question entity

Each question is composed of the body of the question and a set of choices. The question also knows the correct choice. A question belongs to one or more topics and has a given weight. The weight is an integer greater than 0 and less than or equal to 15. Topics are simply represented by a string with at least 6 characters. A question cannot have repeated topics. A question can be associated with a maximum of 5 topics and may have between 2 and 8 choices. The interface of this entity is represented in Listing 1.

If the invocation of a method from the `Question` class invalidates any of the conditions presented before, then the method should have no effect and should throw the `InvalidInvocationException` exception.

2.2 The ExamModel entity

Each exam belongs to a given model. A model specifies the number of groups in the exam, the topic of each group, and the questions that should be part of each group. Listing 2 describes the interface of the *ExamModel* class. Each group has a unique identifier assigned when the group is added to the exam model. The identifier for each group is equal to the number of existing groups in the model at the time the group was added. When you add a group to an exam model, you have to specify the maximum number of questions the group can have and its topic. A group can only have questions that include the topic of the group. The `addQuestion()` method

Listing 1: The interface of the class *Question*.

```
public class Question {
    public Question(String body, List<String> choices, int correctChoice, String topic, int weight) throws
         InvalidOperationException { /* .... */ }

    // Removes a topic
    public void remove(String topic) throws InvalidOperationException { /* .... */ }

    // Adds a new topic
    public void add(String topic) throws InvalidOperationException { /* .... */ }

    // Returns all topics of this question
    public List<String> getTopics() { /* .... */ }

    // Computes the grade considering the weight of this question and the selected choice
    public float grade(int selectedChoice) { /* .... */ }

    // Changes the weight of this question
    public void setWeight(int weight) throws InvalidOperationException { /* .... */ }

    // Returns the weight of this question
    public int getWeight() { /* .... */ }

    // Returns the choices of this question
    public List<String> getChoices() { /* .... */ }
}
```

adds a new question to the concerned group if it is possible. Its return value shows whether the question was added or not.

The `ExamModel` class has the following behavior. A model can be open or closed. While it is open, its state can be changed, and new groups and questions can be added to the model, for example. Once it is closed, its content cannot be modified anymore. A model can only be closed if it is valid. The `validate()` method is responsible for validating an open exam model, changing its state to closed if it is valid. If the model is invalid, this method returns a string that describes why the model is invalid. If the model is valid, then this method simply returns the null reference (`null`). Thus, this method operates according to the following rules (in decreasing order of priority):

- If the model is already closed, it simply returns the `null` reference;
- If the number of groups of the model is less than 2 or greater than 8, it should return "Invalid number of groups";
- If the number of groups is between 2 and 8, then the following needs to be considered:
  - If the model score is different from 100, it should return "Invalid total exam score";
  - If the model score is 100 and if the average number of questions per group is less than or equal to 4, then the exam model is valid, and in that case, the method should change the state of this exam model to *closed* and return the `null` reference. If the average number of questions per group is greater than 4, then it should return "Invalid average number of questions".

The `addQuestion()` method adds a new question to the specified group. The addition of the new question is only performed if the model is still open, the indicated group is valid, and the topic of the group is one of the topics of the question to add. Otherwise, the state of the model should remain unchanged. The method returns a boolean value indicating whether the question was added or not.

The score of an exam model is equal to the sum of the scores of the model's groups. The score of a group is equal to the sum of the scores of the questions belonging to the group.

2.3 The ExamManager entity

The `ExamManager` class manages the exams taken by students. Each exam manager is associated with an exam model and is responsible for providing an exam to enrolled students, collecting completed exams from students, and making the results of submitted exams available. Listing 3 presents the interface of the `ExamManager` class.

Listing 2: The interface of the class *ExamModel*.

```
public enum ModelState {OPEN, CLOSED;}

public class ExamModel {
  public ExamModel() { /* .... */ }

  // Adds a new group and returns the number of the created group
  public int addGroup((int maxNumberOfQuestions, String topic) { /* .... */ }

  // Adds a question to the specified group of this exam model. groupId can range
  // between 1 and the number of groups of this exam model.
  public boolean addQuestion(Question q, int groupId) { /* .... */ }

  // Removes the question from the specified group of this exam model
  public void removeQuestion(Question q, int groupId) { /* .... */ }

  // Validates this exam model
  public String validate() { /* .... */ }

  // ...
}
```

Concerning the invocation of the methods of this entity, we have the following restrictions:

- When you create an exam manager, the manager is *open*. In this mode, you can enroll students for the exam managed by this manager. When the number of enrolled students is greater than or equal to 10, the invocation of the `close()` method ends the enrollment period and changes the mode of the exam manager to *evaluation*.
- When an exam manager is in *evaluation* mode, enrolled students can ask for their exams (method *getExam*) and they can also deliver their exams after they have answered the questions (method *submit*). If the *close()* method is invoked on a manager in *evaluation* mode, then the manager becomes *terminated*. Invoking the `cancel()` method on a manager in evaluation mode puts it back to `open` mode again. This is valid only if no student has yet requested his exam.
- In a *terminated* manager, you can put it back in *evaluation* mode (if you invoke the `cancel()` method) or put it in *published* mode if the `close()` method is invoked.
- Finally, an exam manager in *published* mode can evaluate the submitted exam of each student who took the exam through the `evaluate()` method.

The invocation of the `getMode()` method on an exam manager is always valid, independently of the mode of the exam manager, and it should return the current mode of the exam manager. You should also consider that any invocation of a method of this entity in a case not described here corresponds to an invalid invocation and should lead to throwing the *InvalidOperationException* exception.

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 `Question` class. Worth between 0 and 3.5 points.
- Class scope test cases for the `ExamManager` class. Worth between 0 and 6.5 points.
- Method scope test cases for the `validate()` method of the `ExamModel` class. Worth between 0 and 3.5 points.
- Method scope test cases for the `addQuestion()` method of the `ExamModel` class. Worth between 0 and 3.5 points.

Listing 3: The interface of the*ExamManager* class

```
public enum ManagerMode { OPEN, PUBLISHED, TERMINATED, EVALUATION; }

public class ExamManager {
  public ExamManager(ExamModel model) { /* ... */ }

  // Returns the mode of this exam manager
  public final ManagerMode getMode() { /* ... */ }

  // Enrolls a student for this exam
  public void enroll(Student t) { /* ... */ }

  // Cancels start and close operations
  public void cancel() { /* ... */ }

  // Returns an exam for the (enrolled) student
  public Exam getExam(Student student) { /* ... */ }

  // Submits an exam
  public void submit(Exam exam) { /* ... */ }

  // Finish current state (
  public void close() { /* ... */ }

  // Returns the evaluation of the exam made by the specified student
  public float evaluate(Student student) { /* ... */ }
}
```

- 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 `Question` 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 testing design strategy applied.
- 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 afterwards 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 may 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 the *Project* section of the course's webpage. The project delivery protocol is described in this section.

Have a Good Work!