Week 4 Lab

Abstract Classes, Inheritance and more

Objectives

In this week's lab, you will:

- select and use a `Map` class
- implement an abstract base class and concrete subclasses
- design and implement your own solutions to satisfy requirements

Task 1. Preparation: Algorithm design and UML Class diagram

You must first have completed all tasks from the previous week's lab, including the design of the student list printer.

You must draw a UML class diagram showing your planned system at the end of this lab after implementing all tasks up to and including Task 4.

There are some more challenging aspects to designing the system this week, including:

- deciding which `Map` class you will use to store student details (so you will need to read the referenced documentation...)
- decide on a class to store the student IDs of students enrolled in a unit.
- deciding how you will store and access assessment marks.

This diagram should be quick and simple. It only needs to show the classes you plan to create and the relationships (associations) between them. You don't need to show attributes or methods. You are strongly encouraged to draw this diagram by hand and scan it in (or simply take a photo of it) for submission – you don't need to use a UML diagramming tool.

This must be submitted via the link on the FIT2099 Moodle page *at least 24 hours before the scheduled commencement of your lab class* so that your demonstrator can give you feedback on the quality of your design.

Task 2. Refactor to have a unified list of `Student` details

At the moment, student details are associated with `Units`. This means, for instance, that if a student's details need to be corrected, it may need to be corrected in multiple units. Furthermore, it makes it difficult to find a particular student's details for purposes other than dealing with their enrolment.

You should refactor the system so that the `University` contains a master collection of students, and each unit contains a collection of student IDs.

You will need to

- remove the old `students` attribute from `Unit`

- create an attribute called `enrolledStudents`, which contains a collection of student IDs. Choose an appropriate collection class for this attribute - there are a number of alternatives, some you have already used and others you haven't. Choose one, and explain your reasoning in comments.
- add a attribute called `students` to `University` to store all the `Students`. You are going to use a `Map` implementation, either HashMap or TreeMap. The keys will be the student ID, and the values will be the `Student` objects.
- add a comment to `Unit` to briefly justify your choice of `Map` implementation.
- add an `admitStudent(...)` method to `University`. This takes a `Student` as input and adds it to the map.
- modify the `enrolStudent(...)` method so that it takes an (admitted) student ID, rather than a `Student`.
- if you did Task 5 of last week's lab correctly, you will probably have written a method to return an array of enrolled `Students`. Modify this method so that it returns the same thing, but gets its data from the combination of the `enrolledStudent` collection and the `student Map` in the `University` (hint: how does the `Unit` know what `University` it's associated with)?

Task 3. Adding an `Assessment` class hierarchy

There are two types of assessment at Java University: exams and assignments. Every piece of assessment has an associated weight, which must be an integer between 1 and 100. Exams have a duration, which is an integer between 30 and 180 representing the duration of the exam in minutes. Assignments have a title, which is a string giving the assignment name.

You must implement these requirements using inheritance. You need to:

- create an abstract class called `Assessment` that has
  - an `int` attribute called `weight`, and getter and setter methods for this attribute.
  - an abstract function called `description()` that returns a `String`
- create a class `Exam` that inherits from `Assessment`. It needs to have:
  - a constructor that takes a weight and a duration as parameters, and sets the corresponding attributes
  - an `int` attribute called `duration`
  - an implementation of `description()` that returns a `String` such as: "Exam: duration 180 minutes, weight 60%"
- create an `Assignment` class that inherits from `Assessment`. It needs to have:
  - a constructor that takes a weight and a title as parameters, and sets the corresponding attributes
  - a title attribute
  - an implementation of description() that returns a string containing the title and weight, such as: "Assignment: Team Software Project, weight 40%"

Task 4. `AssessmentSchemes`

We now need to add further functionality to our `Unit` class. Every `Unit` needs to have an assessment scheme associated with it. An assessment scheme is a collection of assessments. An

`AssessmentScheme` object must be passed the `Assessment` objects that it consists of when it is created. Design an `AssessmentScheme` class that meets these requirements. Modify `Unit` so that the assessment scheme for the unit can be set. Modify `University` to test your new code.

`AssessmentScheme` doesn't have much in it yet, but we will refactor it later.

Getting marked
Once you have completed this task, you MUST submit your source code on Moodle. If you and your demonstrator have time, feel free to discuss it before submission.

You must submit your source code to Moodle no later than 24 hours after the *end* of your lab.

Note that to receive full marks, it is not sufficient merely to produce correct output. Your solution must implement the structure described above.

Task 5. Marks for assessments (homework)

When a student completes an assessment, a mark for that piece of assessment for that student needs to be recorded in the system. As well as a numerical mark, a comment can also optionally be recorded. Given an `Assessment` object, it must be possible to access the marks and comments for all the students who have done that assessment. Design and implement a `Mark` class that meets these requirements, as well as any changes needed to other classes. Modify `University` to test your new code.

Task 6. Total mark for a `Unit` (homework)

When a student has completed all the assessments in the assessment scheme for a unit, the mark for that student for that unit can be calculated by summing the student's marks for each assessment in the assessment scheme. Design and implement changes to the `Unit` class so that it is possible to query a `Unit` object to check whether a student has completed all assessments for that unit. Add a method that returns the student's mark for the unit. Modify `University` to test your new code.

Submitting homework

You should submit your complete source code for the homework tasks at the same time as next week's prep. There will be separate preparation and homework submission links on Moodle.

Marking
This lab is worth 2% of your final mark for FIT2099. Marks will be granted as follows:

- 0 marks if you do not complete all tasks, or if your demonstrator has not seen and approved your design
- 1 mark if all tasks are completed but the design is poor or the implementation is flawed
- 2 marks if all tasks are complete, the design is good, and the implementation is correct