Software Analysis and Design 2020-2021

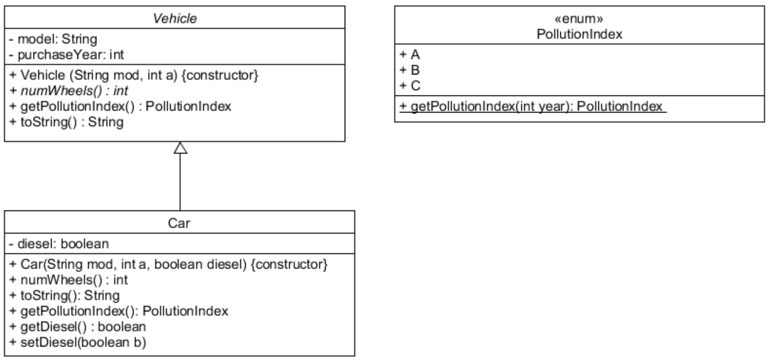
Second Assignment

Object oriented design

Start: Week of February 22nd.
Duration: 2 weeks.
Delivery: via Moodle, one hour before the start of the next assignment according to your group (week of March 8th).
Weight in the final grade: 15 %

The goal of this assignment is to introduce basic concepts of object orientation, from the perspective of design (using UML), as well as their corresponding Java constructs.

Part 1 (3.5 points):


Vehicle
- model: String
- purchaseYear: int
+ Vehicle (String mod, int a) {constructor}
+ numWheels() : int
+ getPollutionIndex() : PollutionIndex
+ toString() : String
«enum»
PollutionIndex
+ A
+ B
+ C
+ getPollutionIndex(int year): PollutionIndex
Car
- diesel: boolean
+ Car(String mod, int a, boolean diesel) {constructor}
+ numWheels() : int
+ toString(): String
+ getPollutionIndex(): PollutionIndex
+ getDiesel() : boolean
+ setDiesel(boolean b)

You need to write an application for a provincial traffic police department (DGT), that stores data of interest about different types of *vehicles*, as well as its *pollution index* (A, B or C, being 'A' the least polluting). For this purpose you are provided with the class diagram on the right.

As you can see, an abstract class *Vehicle* stores the *model* and *year* of purchase, and a subclass *Car* stores whether the car is diesel or gasoline. In addition to the model, the *Vehicle* class contains an abstract method *numWheels* (which must be implemented by all subclasses to indicate the number of wheels of the vehicle), a *getPollutionIndex* method, which returns an enumeration of type *PollutionIndex*, and the *toString* method, which you are familiar with from the previous assignment. Enumerations are similar to classes. *PollutionIndex* defines three possible values (A, B, C, which are actually objects of type *PollutionIndex*), and a static method *getPollutionIndex*. A static method does not need the context of an object to be invoked, but it is defined and invoked at the class level (an enumeration in this case).

The *Car* subclass defines its own constructor, implements the *numWheels* method (which had no implementation in the *Vehicle* class), defines a *getter* and a *setter* for the *diesel* attribute, and overrides the base class *toString* and *getPollutionIndex* methods.

The following listings show the complete implementation of these classes in Java. As you can see, the classes have been included in the package "*prac2.traffic*". Packages are used to organize the classes involved in the programs. In this way, the most related classes will be declared in the same package. The structure of the packages is hierarchical and corresponds to the physical structure of directories (folders) in the file system, i.e., the classes have to be stored in a directory '*prac2/traffic*'.

Constructors have the same name as the class, and are invoked by the operator **new** to construct an object (create an instance of the class) and initialize its attributes. Thus, the constructor of *Vehicle* takes two parameters and initializes the attributes *model* and *purchaseYear*, and the constructor of the class *Car* invokes via **super** (…) the constructor of its superclass *Vehicle* before initializing attribute *diesel*. The annotation (optional) **@Override** indicates that the programmer's intention is to override a method of one of the super-classes, which must have the same signature.

You can find below an example program and its output. Since all cars are vehicles, we can store them in an array of type *Vehicle*. This gives the possibility to add more *Vehicle* subclasses, and to handle its objects in a uniform way. The Java code for these classes is available in Moodle.

Class Vehicle

```java
package prac2.traffic;

public abstract class Vehicle {
	private String model;
	private int purchaseYear;

	public Vehicle(String mod, int a) {
		this.model = mod;
		this.purchaseYear = a;
	}

	@Override
	public String toString() {
		return "model "+this.model+", purchase year "+this.purchaseYear+", with "+
				this.numWheels()+" wheels, index:"+this.getPollutionIndex();
	}

	public abstract int numWheels();

	public PollutionIndex getPollutionIndex() {
		return PollutionIndex.getPollutionIndex(this.purchaseYear);
	}
}
```

Class Car

```java
package prac2.traffic;

public class Car extends Vehicle {
	private boolean diesel;

	public Car(String mod, int a, boolean diesel) {
		super(mod, a);
		this.diesel = diesel;
	}

	@Override public int numWheels() { return 4; }

	@Override public String toString() {
		return "Car "+(this.diesel ? "diesel" : "gasoline") + ", "+ super.toString();
	}

	@Override
	public PollutionIndex getPollutionIndex() {
		if (this.diesel) return PollutionIndex.C;
		return super.getPollutionIndex();
	}

	public boolean getDiesel() { return this.diesel; }
	public void setDiesel(boolean b) { this.diesel = b; }
}
```

Enumerator PollutionIndex

```java
package prac2.traffic;

public enum PollutionIndex {
	A, B, C;

	private static final int DATEA = 2018;
	private static final int DATEB = 2010;

	public static PollutionIndex getPollutionIndex(int year) {
		if (year >= DATEA) return A;
		if (year >= DATEB) return B;
		return C;
	}
}
```

Sample Program

```
package prac2.traffic;

public class Example1 {
        public static void main(String[] args) {
                Car fiat500x = new Car("Fiat 500x", 2019, true);
                Car minic = new Car("Mini Copper", 2015, false);
                Car xsara = new Car("Citroen XSara", 1997, true);

                Vehicle [] vehicles = { fiat500x, minic, xsara };

                for (Vehicle v : vehicles )
                        System.out.println(v);
        }
}
```

Output:

```
Car diesel, model Fiat 500x, purchase year 2019, with 4 wheels, index:C
Car gasoline, model Mini Copper, purchase year 2015, with 4 wheels, index:B
Car diesel, model Citroen XSara, purchase year 1997, with 4 wheels, index:C
```

You are required to: Extend the **UML design** and the **Java program** with two new types of vehicle: motorcycles and trucks. In addition to the model and year of purchase, a motorcycle stores whether it is electric or not. Motorcycles have two wheels, and if they are electric they have an 'A' pollution index, regardless of the year of purchase. If they are not, the general rules for other vehicles apply. Trucks have a model, a year of purchase and a number of axles. The number of wheels on a truck is given by twice the number of axles, and its pollution index is C if it has more than two axles. Otherwise, the general rules for other vehicles apply. Finally, we want to store the number plate of any type of vehicle.

As a guide, the following example program should give the output below.

```
public class Example2 {
  public static void main(String[] args) {
        Car fiat500x = new Car("Fiat 500x", 2019, "1245 HYN", true);

        Motorcycle moto1 = new Motorcycle("Harley Davidson", 2003, "0987 ETG", false);
        Motorcycle moto2 = new Motorcycle("Torrot Muvi", 2015, "9023 MCV", true);

        Truck camion1 = new Truck("MAN TGA410", 2000, "M-3456-JZ", 3);
        Truck camion2 = new Truck("Iveco Daily", 2010, "5643 KOI", 2);

        Vehicle [] vehicles = { fiat500x, moto1, moto2, camion1, camion2 };

        for (Vehicle v : vehicles )
                System.out.println(v);
  }
}
```

Output:

```
Car diesel, model Fiat 500x, number plate: 1245 HYN, purchase date 2019, with 4 wheels, index:C
Motorcycle, model Harley Davidson, number plate: 0987 ETG, purchase date 2003, with 2 wheels, index:C
Motorcycle electric, model Torrot Muvi, number plate: 9023 MCV, purchase date 2015, with 2 wheels, index:A
Truck with 3 axles, model MAN TGA410, number plate: M-3456-JZ, purchase date 2000, with 6 wheels, index:C
Truck with 2 axles, model Iveco Daily, number plate: 5643 KOI, purchase date 2010, with 4 wheels, index:B
```

Note:

In practice, a class diagram is an abstraction of the final code, so for ease of understanding not all the details that are in the code are included. This way, the setters and getters and constructors of the classes are usually omitted.

Additional reading:

The *toString*() method of the *Vehicle* class is an example of the 'Template Method' design pattern, which consists of writing code in a parent class that calls methods (perhaps abstract, such as *numWheels()*) that are implemented in subclasses. You have more information about design patterns in the book: Design Patterns: Elements of Reusable Object-Oriented Software Gamma, E., Helm, R., Johnson, R., Vlissides, J. Addison-Wesley, 2003.

Part 2 (3 points):

You have to write an application for the development of statistical questionnaires and the collection of anonymous responses by users. A questionnaire is given by an introductory text, the estimated number of minutes to complete it, and the initial question. Each question has a text. Three types of questions will be considered: open-ended, single-choice, and multiple-choice. An open-ended question is configured with a maximum of characters for the answers, which by default will be 128. The choice-based questions will present 2 or more options, each one with a text. You may configure whether these options are presented to the user randomly or in the order in which they have been entered. Multiple-choice questions can be configured with a flag indicating whether it is possible to choose 'no option' as an answer.

In addition, when creating a questionnaire, you need to configure how to navigate from one question to the next. In the simplest case, a next question is selected, but the system must support conditional navigation. In this case, the next question depends on the options chosen by the user (in case of the choice-based questions). In this way, from a question of type choice it is possible to establish different navigations to other different questions, depending on the different options selected as an answer.

The system must store the users' responses to the different questionnaires anonymously. In particular, for a questionnaire, the start and end date is stored, as well as the start and end date of the answers to each question. In the case of choice questions, the selected options are stored, and in the case of open questions, the answered text.

<u>You are required to:</u>

a) Make a UML class diagram that reflects the design of the application. You do not need to include *constructors*, *getters* or *setters* (**2 points**).

b) Include methods in the classes of your diagram for (**0.6 points**):

1. Displaying the different types of questions on the screen.
2. Determine what the next question is, given a user answer to a question.
3. Obtain the time it took a user to answer a question, and to answer a questionnaire.
4. Obtain the deviation (in seconds) of the average response time of a questionnaire, with respect to the estimated time for its completion.

Bear in mind that in some cases it may be necessary to add helper methods. You don't need to include the method code in this exercise.

c) Make an object diagram that represents a questionnaire with 3 questions (open, simple and multiple choice), two navigations (conditional and simple), and an answer (**0.4 points**).

Part 3 (3.5 points):

You should build an application for managing virtual meetings, with video and chat support. When a user wants to use the application, she must enter a *nickname* (which must be unique in the system), by which she will be identified in the meetings in which she participates. Once inside the application, the user has the option to register, for which she will be asked for a name, email and password.

A registered user can create a meeting room, which is identified by a name. The meeting rooms can be public. There are three types of rooms: chat, video conference, and mixed. A chat room may or may not be moderated, and a conversation language – English, French, Spanish, German or other – must be established. A video conference room can have a virtual background image associated with it, which will be used automatically in all video conferences held in that room. Finally, a mixed room can contain chat rooms, video conferencing or other mixed rooms in a hierarchical way. The creator of a room, of any type, can prohibit entry to certain users, and the system will prevent these users from participating in any meeting held in that room.

The creator of a room can schedule meetings in it. A meeting is given by a name, a date and optionally a maximum duration in minutes. The meetings can be of two types: public and private. Private meetings can be organized in public or private rooms, but private rooms can only hold private meetings. In a private meeting, the list of users (registered or not) that can attend must be set. For public meetings, a maximum number of attendees should be established. In either case, a user cannot participate if she is prohibited entering the room in which the meeting is organized, or in any of the rooms contained in it (if the meeting room is a mixed room).

The system must record the content of the meetings. In particular, in any type of meeting, the messages created in the chat rooms will be recorded. A message is defined by the user who creates it, with a text and a date. Two types of messages are considered: public (reaching all participants in the room) and private (directed to a specific user). In either case, a message can be created in response to another one. In private meetings held in video conferencing rooms, it is possible to record video clips, which are identified by the start and end dates, as well as the name of the video file (automatically assigned by the system).

You are required to:

a) Design the system using a class diagram. Add the necessary methods in classes to obtain the functionality mentioned in the text. You do not need to include constructors, *getters* or *setters*. (**3 points**)

b) Specify the pseudocode (or you can use Java) of the following methods: (**0.5 points**)

1. Check that a user can access a meeting.
2. Obtain the users with prohibited entry into a room, and all those contained in it.

Delivery Rules:

- Submit parts 1, 2 and 3. Create a directory for each section.
- The delivery will be made by one of the students in the team, through Moodle.
- If the exercise asks for Java code, the documentation generated with ***javadoc*** must also be delivered. If the exercise asks for a design diagram, this should be submitted in PDF together with a short explanation of two or three paragraphs at the most.
- A single ZIP / RAR file must be delivered with everything requested, which must be called: `GR<group_number>_<student_name>.zip`. For example, Marisa and Pedro, from group 2261, would deliver the file: `GR2261_MarisaPedro.zip`