AP-20 – Programming Assignment #1

Exercise 1 (Java Beans) – Busses at the time of covid….

Due to the coronavirus pandemic, a bus capacity is reduced by 50%. We want to help the bus driver to keep travellers safe, by providing a system which allows passengers to get in only if there are available seats. The system controls the opening of the entrance door of the bus only.

Passengers at a bus stop can book the entrance to the bus. When the bus stops, the entrance door opens if allowed, and stays open for a few seconds, so that the passengers can enter the bus. Otherwise the door stays closed, and the booking is canceled. Passengers can always leave the bus.

The system to be designed is composed (at least) by a graphical dashboard and by two beans: the `Bus` and the `CovidController`, described as follows.

The Bus

The `Bus` is a non-visual bean having three properties, `capacity` (int, initially 50), `doorOpen` (boolean, initially false) and `numPassengers` (int, initially 20). Property `doorOpen` is bound, while `numPassengers` is bound and constrained.

`Bus` also has a method `activate()`. When activated, exploiting a timer, `numPassengers` is decreased randomly every few seconds (but never becomes negative).

When the property `numPassenger` is increased, if the new value does not exceed the `capacity` and the change is not blocked by a veto, just before the value of `numPassenger` is updated the property `doorOpen` has to be set to true for three seconds, and then back to false.

The CovidController

The `CovidController` is a non-visual bean having a single property: `reducedCapacity` (int, initially 25), and implementing `VetoableChangeListener`. The implementation of the abstract method `vetoChange` should forbid the change of the property `numPassenger` if the new value is larger than the `reducedCapacity`.

The Graphical Dashboard

The graphical dashboard has to be called `BusBoard` and must extend `JFrame`. It uses a `Bus` and a `CovidController` bean (you may import them from their `jar` files into the palette of NetBeans). The board has the following visual components:

- Two components which show the value of the properties `numPassenger` and `doorOpen` of Bus, respectively. You can use two labels, for example.
- A text field or another graphical component where the user can enter an integer in the range 1-5 (initial value and default: 1).
- A button, to request entering the bus. When the button is clicked, the button changes color immediately, and after two seconds the `setNumPassenger` method of `Bus` is invoked to

increase the property by the amount in the text field. Independently of the success or failure of this request, the text field and the button are reset.

Solution format

Three adequately commented source files (`Bus.java`, `CovidController.java` and `BusBoard.java`) and the three corresponding `jar` archives. Just for simplifying the integration of the three parts of the project, the `jar` archive of `BusBoard` can also contain a copy of the class files `Bus.class` and `CovidController.class`, extracted from the other `jars`.

Warning: When developing the `Bus` component, deal correctly with the `capacity` constraint, without assuming the presence of the `CovidController`. In fact the `Bus` could be reused in a completely different project.

Exercise 2 (Java Reflection and Annotations) – XML serialization

XML is a meta-language which can be used to describe structured documents in a machine-readable way. Information is packaged in elements. For a simple introduction to XML elements take a look at the following page:

XML Elements

[https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms766385(v=vs.85)]

An XML document contains one or more (possibly nested) elements. For instance, the following is a valid XML document:

```
<Student>
     <firstName type="String">Jane</firstName>
     <surname type="String">Doe</surname>
     <age type="int">42</age>
</Student>
```

As XML is widely used to export data, you must implement a Java serializer to export a class information in XML format. (Note that there are several Java APIs for XML serialization. Here we ask something different and much simpler).

Write a Java class `XMLSerializer` which offers a static method with signature

```
void serialize(Object [ ] arr, String fileName)
```

We assume that all the Objects of `arr` belong to the same class, say `C`. The method should introspect the class `C` searching for information (provided using annotations) to serialize the objects in `arr`. The output should be written in the file called `filename.xml`. Annotations are as follows:

- `@XMLable` provides information about the class. The presence of this annotation says that the objects of this class should be serialized. In this case the main tag should be the name of

the class itself. If the annotation is not present, no serialization is necessary and the method returns.

- `@XMLfield` identifies serializable fields (i.e., instance variables, only of primitive types or strings). The presence of this annotation states that the field has to be serialized. The annotation has a mandatory argument `type`, which is the type of the field (a `String`, for example `"int"`, `"String"`,…), and an optional argument `name`, also of type `String`, which is the XML tag to be used for the field. If the argument is not provided, the variable's name is used as a tag.

Once all the information about the class is collected, the program serializes all the objects of the arrays, writing them in the output file.

As an example, consider the XML element above: it should be the result of serializing an object of the following Java class, created by calling the constructor with parameters `"Jane"`, `"Doe"`, and `42`:

```
@XMLable
public class Student {
     @XMLfield(type = "String")
     public String firstName;
     @XMLfield(type = "String", name="surname")
     public String lastName;
     @XMLfield(type = "int")
     private int age;
     public Student(){}
     public Student(String fn, String ln, int age) {
          this.firstName = fn;
          this.lastName = ln;
          this.age = age;
     }
     ...
}
```

Solution format

- The Java files defining the annotations `@XMLable` and `@XMLfield`
- The class `XMLSerializer.java`

Additionally, for testing:

- A simple class annotated as described in the text above. The `@XMLfield` annotation has to be used, with fields of at least two different types, and both using the optional argument and not using it.
- A main class building an array of object of the class of the previous point and invoking `XMLSerializer.serialize()` on it.

Exercise 3 (Optional) – XML deserialization

In general, serialization only makes sense if accompanied with deserialization. In our case this would mean to be able to read an XML file obtained by `XMLSerializer.serialize()`, and to return an array of objects obtained by creating for each object encoded in the file a new instance of the same class with the same values for the serialized fields.

But this is possible only under certain conditions. Quite arbitrarily, we say that a class is "deserializable" if:

- It is annotated with `@XMLable`;
- It has a constructor with no arguments;
- All its fields are non-static and of primitive type or `String`;
- All its fields are annotated with `@XMLfield`**.**

For example, the class `Student` shown above is "deserializable" (if no other fields are added).

Write Java a program that reads an XML file produced by your `XMLSerializer.serialize()` method, then checks if the class of the serialized objects is "deserializable" (using introspection), and in this case returns an array with the corresponding deserialized objects.

Solution format: Free.