Assignment 4

Collections, genericity, lambda expressions and design patterns

Start: Week of April 26th
Duration: 2 weeks
Delivery: 9th of May, 23:55h (all groups)
Weight: 25%

The aim of this practice is to learn the use of different types of collections, designing generic programs that are able to adapt to parametric types, and using design patterns in a practical way. To do so, we will build some classes that help creating input forms, which take user input from the console, and that are then able to process the collected data.

Part 1: Generic forms for console data input (4.5 points)

We want to build a class called `Form`, which is a form for collecting input data from the console. The class must be configurable with the form fields, of type `Field`. These fields will be generic, parameterised with the expected type of the user response. In addition, the fields must be configured with a function (passed in the constructor) to convert from the `String` read from console, to the type of the field. Optionally, one or more validators may be added. These validators will contain a function, on the type of the field, which makes a correctness check (for example, that the age read in the field is over 18 years old) and the message to be presented to the user if the validation is not fulfilled.

The `Form` class must have an `exec` method, which presents (in order) each field to the user, collects the response (validating it) and returns a map with the text associated to each field as key, and the user's response (converted to the type of the field) as value.

The code below is an example program, and a possible execution is shown at the bottom. As you can see, the form does not ask repeated questions ("*are you married?*"), and is configured by means of calls to the `add` method. In this program, we have configured the converter and the validators of the fields by means of lambda expressions.

```
package forms;

import java.io.IOException;

public class FormMain {
  public static void main(String[] args) throws IOException {
        Form enrollForm = new Form();

        Field<Integer> age = new Field<Integer>(s -> Integer.valueOf(s)).
                              addValidation(a -> a > 18, "value should be bigger than 18").
                              addValidation(a -> a < 66, "value should be less than 66");
        Field<Boolean> yesNo = new Field<>(s -> s.toUpperCase().equals("YES"));
        enrollForm.add("What is your age?", age).
                   add("Are you married?", yesNo).
                   add("Do you have children?", yesNo).
                   add("Are you married?", yesNo);

        System.out.println(enrollForm.exec());
  }
}
```

```
What is your age? > 3

Invalid value: 3
  value should be bigger than 18
What is your age? > 34
Are you married? > yes
Do you have children? > no
{What is your age?=34, Are you married?=true, Do you have children?=false}
```

(the text shows in green the data introduced by the user)

Part 2: Processing the collected data (3 points)

In this section we will create a `DataAggregator` class that will group the answers to the forms, sorting each answer according to the *natural order* defined by each type of each field of the questionnaire. To do this, you must require that the data types of each `Field`, as well as the map data returned by the `exec` method, be compatible with the `Comparable` interface. For simplicity, the `DataAggregator` class will only present the data, but in a real application it would have many other functionalities, such as separating the data into intervals, or calculating statistics.

The following program exemplifies its use. You also need to fill in the `Address` class to obtain the output below. Note that the answers of type `Address` are sorted by postal code, and in case of equality, by alphabetical order of the street. `Integer` answers are sorted from smallest to largest.

```
class Address _____ /* Complete if needed */ {
        private String address;
        private int postalCode;

        public Address(String adr, int pc) {
                this.address = adr;
                this.postalCode = pc;
        }
        public int postalCode() { return this.postalCode; }
        public String toString() {
                return this.address+" at PC("+this.postalCode+")";
        }
        // Completar si es necesario
}

public class ProcessingMain {
  public static void main(String[] args) throws IOException {
        Form censusForm = new Form();

        Field<Address> adr = new Field<Address>(s -> { String[] data = s.split(";");
                                                       return new Address(data[0], Integer.valueOf(data[1].trim()));
                                                     }).
                             addValidation(a -> a.postalCode() >= 0, "Postal code should be positive");
        Field<Integer> np  = new Field<Integer>(s->Integer.valueOf(s)).
                             addValidation(s->s>0, "value greater than 0 expected");

        censusForm.add("Enter address and postal code separated by ';'", adr).
                   add("Number of people living in that address?", np);
        DataAggregator dag = new DataAggregator();
        for (int i=0; i<3; i++)
          dag.add(censusForm.exec());

        System.out.println(dag);
   }
}
```

```
Enter address and postal code separated by ';' > Main St.;45007
Number of people living in that address? > 6
Enter address and postal code separated by ';' > Baker St.;45007
Number of people living in that address? > 2
Enter address and postal code separated by ';' > Regent St.;23008
Number of people living in that address? > 1
{Enter address and postal code separated by ';'=[Regent St. at PC(23008), Baker St. at PC(45007), Main St. at
PC(45007)], Number of people living in that address?=[1, 2, 6]}
```

Part 3: Protecting the forms by means of a password (2.5 points)

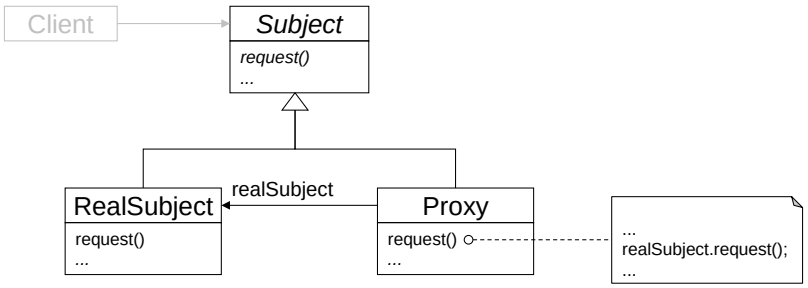
Finally, we will give the option to protect the form with a password. This design modification should not interfere with the code already developed, so the programs in sections 1 and 2 should continue to work. The idea is that the user responding to the form should enter a password before using the form, if s/he has not already done so. Three attempts are given for entering the password, and the questionnaire will be blocked if the password is not entered correctly in these three attempts. If there are several executions of the form, the password is only requested once.

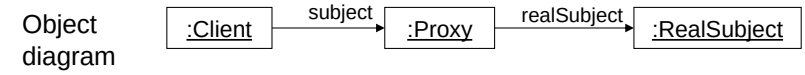
As an example, the program below shows how protected forms are used, and a possible implementation. A form is protected by the static method `ProtectedForm.protect`, where the form to be protected and the expected password are given. In the execution shown, the second time the form is executed, the password has already been entered, so it is not requested again. If the password would have been incorrect after the 3 attempts, the form would not be displayed, because it would be locked.

```java
public class ProtectedFormMain {
  public static void main(String[] args) throws IOException {
        Form enrollForm = new Form();

        Field<Integer> age = new Field<Integer>(s -> Integer.valueOf(s)).
                              addValidation(a -> a > 18, "value should be bigger than 18").
                              addValidation(a -> a < 66, "value should be less than 66");
        Field<Boolean> yesNo = new Field<>(s -> s.toUpperCase().equals("YES"));
        enrollForm.add("What is your age?", age).
                   add("Are you married?", yesNo);

        AbstractForm pf = ProtectedForm.protect(enrollForm, "qwerty");
        System.out.println(pf.exec());
        System.out.println(pf.exec());
  }
}
```

```
Enter password: admin
Invalid password (2 remaining attempts)
Enter password: qwerty
What is your age? > 23
Are you married? > no
{What is your age?=23, Are you married?=false}
What is your age? > 34
Are you married? > yes
{What is your age?=34, Are you married?=true}
```

Hint: A good option for this exercise is the use of the Proxy design pattern, which allows to provide a substitute for another object (with the same interface as this one) to protect its access. The following diagram shows a scheme of how it works. You can find more details of this pattern in the course slides.





Additional question: This is a very simplified assignment, which outlines an initial design for handling forms. How can we extend the design to improve the usability of the *framework* by avoiding the need to provide in the constructor of `Field` a converter for standard data types such as `Integer`, `Double` or `Boolean`?

Submission rules. You should submit:

- A **src** folder with all the Java code, including the test data and additional testers that you developed in the sections that require it.
- A **doc** folder with the generated documentation.
- A **report** in PDF format with a short description of the design decisions taken for the implementation of each section and the answer to the additional question in section 3.
- The **class diagram** of the final design.

Package everything in a single ZIP file, with name GR<group_number>_<student_names>.zip. For example Marisa and Pedro, from group 2213, would hand in the file: GR2213_MarisaPedro.zip. Inside the zip file there should be a directory src/, a folder doc/ and the PDF. Failure to deliver in this format will result in a penalty in the mark.