Lab assignment 5

Genericity, Collections, Lambdas and Design Patterns

Start: Week of April 13th.
Duration: 4 weeks.
Submission: May 8th (all groups).
Weight: 30%

The objective of this assignment is to exercise more advanced object-oriented concepts, like

- *Design of generic and highly reusable classes,*
- *Use of the Java collections*
- *Use of design patterns and design strategies for APIs*
- *Use of lambda expressions*

Section 0. Introduction

In this assignment we will gradually develop an application for processing objects using simple rules. In our case, a rule is made of an application condition and an effect. If the object passed as a parameter meets the condition, then the rule is executed, applying its effect.

In the assignment we will apply this programming style to the realization of computations on graphs, and for this a generic structure will be built to store nodes and edges with different types of value.

Section 1. *Genericity and Collections*: A directed graph (3 points)

We will start by building a generic `Graph` class, which should allow storing `Nodes` of a certain type, linked by directed `Edges`. These edges will also contain information of another type, selected by the programmer. The graph must not support repeated nodes (considering referential equality), but a node can be connected multiple times with the same node. Design the class so that it has an API similar to the other Java collections, and is interoperable with them, allowing, for example, to create a list of nodes from a graph.

As an example, the following listing shows a graph with nodes that hold strings, and edges that hold integers.

```
Graph<String, Integer> g = new Graph<String, Integer>();
Node<String>  n1 = new Node<String>("s0");
Node<String>  n2 = new Node<String>("s1");

g.addAll(Arrays.asList(n1, n2, n1));       // does not admit repetitions, considering referential equality

g.connect(n1, 0, n1);                      // we connect n1 with n1 through an edge with value 0
g.connect(n1, 1, n2);
g.connect(n1, 0, n2);
g.connect(n2, 0, n1);
g.connect(n2, 1, n1);

System.out.println(g);                     // The graph contains 2 nodes and 5 edges

for (Node<String> n : g)                   // Collection of two nodes (n1 and n2)
       System.out.println("Node "+n);

List<Node<String>> nodes = new ArrayList<>(g);    // we can create a list from g
System.out.println(nodes);

// Two methods to check connectivity, the first receives the value of the Node
System.out.println("s0 connected with ‘s1’: "+ n1.isConnectedTo("s1"));
System.out.println("s0 connected with s1: "+ n1.isConnectedTo(n2));
System.out.println("neighbours of s0: "+n1.neighbours());
System.out.println("value of the edges from s0 to s1: "+n1.getEdgeValues(n2));
```

The execution of the previous program should produce the following output:

```
Nodes:
 0 [s0]
 1 [s1]
Edges:
 ( 0 --0--> 0 )
 ( 0 --1--> 1 )
 ( 0 --0--> 1 )
 ( 1 --0--> 0 )
 ( 1 --1--> 0 )

Node 0 [s0]
Node 1 [s1]
[0 [s0], 1 [s1]]
s0 connected with ‘s1’: true
s0 connected with s1: true
neighbours of s0: [0 [s0], 1 [s1]]
value of the edges from s0 to s1: [1, 0]
```

As you can see, internally the nodes must have an identifier (from zero onwards), which is useful when printing or debugging a graph. The nodes and edges are shown in insertion order. When a node is removed from a graph using the `remove`, or `removeAll` methods, that node must be removed as a neighbour of any other node in the graph. Only nodes that belong to the same graph are allowed to be connected.

Design an API for Node and Graph that allows performing common operations on nodes, for example: method `isConnectedTo` should tell us if one node is directly connected to another, method `neighbours` returns the collection of nodes directly connected to a given node, or `getEdgeValues` returns the values of the edges (without repetitions) linking one given node to another one.

Section 2. Comparators and Lambdas: Classes *ConstrainedGraph* and *BlackBoxComparator* (2 points)

a) Taking as a basis the `Graph` class, create a class `ConstrainedGraph` to support checking various properties on the nodes of a graph. The properties will be of three types: *universal*, *existential* and *unitary*. A *universal* property is fulfilled if all the nodes of the graph fulfill it. An *existential* property is true, if at least one node satisfies it. A *unit* property is true if there is exactly one node that satisfies it.

```
ConstrainedGraph<Integer, Integer> g = new ConstrainedGraph<Integer, Integer>();
Node<Integer>  n1 = new Node<Integer>(1);
Node<Integer>  n2 = new Node<Integer>(2);
Node<Integer>  n3 = new Node<Integer>(3);
g.addAll(Arrays.asList(n1, n2, n3));
g.connect(n1, 1, n2);
g.connect(n1, 7, n3);
g.connect(n2, 1, n3);

System.out.println("All nodes of g connected with n3? "+g.forAll(n -> n.equals(n3) || n.isConnectedTo(n3)));    // true
System.out.println("Is there exactly one node connected to n2? "+g.one( n -> n.isConnectedTo(n2)));             // true
System.out.println("Is there at least one node of g connected to n2? "+g.exists( n -> n.isConnectedTo(n2)));   // (*) true
```

As you see, the `exists`, `one` and `forall` methods receive a lambda expression as a parameter. As you probably know from the theory class, a lambda expression is nothing more than compact notation for an *anonymous* class. Thus, the line marked as (*) in the previous listing could be rewritten with an anonymous class as follows:

```
System.out.println("Is there at least one node of g connected to n2? "+
                   g.exists(new Predicate<Node<Integer>>() {
                        @Override public boolean test(Node<Integer> n) { return n.isConnectedTo(n2); }}));
```

To simplify later sections, we will take the convention that, if an existential property is satisfied, the node that satisfies it will be stored in an attribute of the graph, such that a call to the `getWitness()` method will return said node, or null. To unify both possibilities, the method must return an `Optional` object. For example, given the graph g above, the following 4 lines:

```
g.exists( n -> n.getValue().equals(89));        // Not satisfied: Optional is null
g.getWitness().ifPresent( w -> System.out.println("Witness 1 = "+g.getWitness().get()));
g.exists( n -> n.isConnectedTo(n2));            // Satisfied: Optional has value
g.getWitness().ifPresent( w -> System.out.println("Witness 2 = "+g.getWitness().get()));
```

should print: `Witness 2 = 0 [1],` where 0 is the identifier of the node and 1 its value

b) Create a graph comparator class called `BlackBoxComparator`. This class will be configured with existential, unitary and universal properties. The comparison criterion will be the number of properties (of any type) satisfied by the graphs. For example, if one graph g1 satisfies 2 properties, and another graph g2 satisfies 3, then g1 <g2. Thus, if we compare the previous graph g with the following graph g1 using the bbc comparator:

```
ConstrainedGraph<Integer, Integer> g1 = new ConstrainedGraph<Integer, Integer>();
g1.addAll(Arrays.asList(new Node<Integer>(4));

BlackBoxComparator<Integer, Integer> bbc = new BlackBoxComparator<Integer, Integer>();

bbc.addCriteria( Criteria.EXISTENTIAL, n -> n.isConnectedTo(1)).        // corrected
    addCriteria( Criteria.UNITARY, n -> n.neighbours().isEmpty()).
    addCriteria( Criteria.UNIVERSAL, n -> n.getValue().equals(4));
```

and use such comparator to order a list with g and g1, we should obtain a list where g appears first, and g1 second:

```
List<ConstrainedGraph<Integer, Integer>> cgs = Arrays.asList(g, g1);
Collections.sort(cgs, bbc);     // We use the comparator to order a list with two graphs
System.out.println(cgs);        // prints g (satisfies the 2nd property) and then g1 (satisfies the 2nd and 3rd) (corrected)
```

Section 3. Genericity and Lambdas: Rules and Rule Sets (3 points)

Once we have designed a generic data structure for the graphs, we will focus on defining rules (class `Rule`) and rule sets (class `RuleSet`). As we have already mentioned, a rule is evaluated on objects of a given type. If the condition is satisfied, then the rule can be executed. Once we have defined a set of rules, we can evaluate them on a collection of objects of the corresponding type. The following listing illustrates the use of rules and rule sets. We use an example `Product` class is used, to show that your rules will not only be applicable to graphs, but to objects of any other class.

```
class Product {  // A sample class to test the rules
  private double price;
  private Date expiration;  // A better option is to use Calendar

  public Product (double p, Date c) {
        this.price = p;
        this. expiration = c;
  }

  public double getPrice()          { return this.price; }
  public void   setPrice(double p)  { this.price = p; }
  public Date   getExpiration ()      { return this. expiration; }

  public static long getDateDiff(Date date1, Date date2, TimeUnit timeUnit) {
    long diffInMillies = date2.getTime() - date1.getTime();
    return timeUnit.convert(diffInMillies,TimeUnit.MILLISECONDS);
  }
  @Override public String toString() { return this.price+", expiration: "+this. expiration;      }
}

public class Main {
  public static void main(String...args) throws ParseException{
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
    RuleSet<Product> rs = new RuleSet<Product>();       // A set of rules applicable to Products

    rs.add(
           Rule.<Product>rule("r1", "10% discount on products with an expiration date close or past").
                when(pro -> Product.getDateDiff(Calendar.getInstance().getTime(), pro.getCaducidad(), TimeUnit.DAYS) < 2 ).
                exec(pro -> pro.setPrice(pro.getPrice()-pro.getPrice()*0.1))
           ).add(
           Rule.<Product>rule("r2", " Discount on products with price more than 10 euros").
                when(pro -> pro.getPrice() > 10).
                exec(pro -> pro.setPrice(pro.getPrice()-pro.getPrice()*0.05))
    );

    List<Product> str = Arrays.asList(  new Product(10, sdf.parse("15/04/2020")),  // parses a date
                                        new Product(20, sdf.parse("20/03/2021")));

    rs.setExecContext(str);     // indicates that the rule set rs will be executed over str
    rs.process();               // we execute the rule set

    System.out.println(str);  // prints str
  }
}
```

As you can see, both the rules and the rule sets are parameterized by the type of the object they treat. The `when` and `exec` methods are prepared to receive lambda expressions. To facilitate the readability of the code, and better integration with the Java 8 programming style, design a fluid API (http://en.wikipedia.org/wiki/Fluent_interface), which allows chaining calls. Thus, the `rule`, `when`, and `exec` methods return a `Rule` object to facilitate the construction and configuration of a `Rule` object by chaining calls.

The output of the above program is as follows (the first rule applies to the first object and the second to the second):

```
9.0, expiration: Wed Apr 15 00:00:00 CET 2020
19.0, expiration: Sat Mar 20 00:00:00 CET 2021
```

For the moment, consider a rule application strategy that consists of: iterating over all the objects to be processed, checking each of the rules in the order in which they were inserted in the rule set. If a rule is applicable it is executed and passed to the next object to be processed.

Section 4. Design patterns: Rule execution strategies (2 points)

The rule execution strategy in the previous section may not be appropriate in some scenarios. For example, sometimes we may want to apply the rules as long as possible (that is, continue to apply them as long as the `when` condition is met). To do this, following the *Strategy* design pattern (http://en.wikipedia.org/wiki/Strategy_pattern) we will extract the execution strategy outside the `RuleSet` class, and we will pass the most appropriate strategy in the `RuleSet` constructor.

To avoid modifying the `RuleSet` class (and thus facilitate our grading), create a new `RuleSetWithStrategy` class, taking `RuleSet` as a basis. The following listing shows how to use such a class with an `AsLongAsPossible` strategy. The listing calculates the minimum path from one node of a graph to the other nodes, using Dijkstra's algorithm (http://en.wikipedia.org/wiki/Dijkstra's_algorithm) encoded in a single rule. The starting node is initialized with the value 0, and the distance to this node is saved as the value of the nodes (which are initialized with the `INIT_CONSTANT` constant).

```
final int INIT_CONSTANT = 1000;
ConstrainedGraph<Integer, Integer> g = new ConstrainedGraph<Integer, Integer>();
Node<Integer>  n0 = new Node<Integer>(0);               // The value of the node is the path length. N0 is the initial node
Node<Integer>  n1 = new Node<Integer>(INIT_CONSTANT);   // we initialise the rest to a high value, that will be reduced…
Node<Integer>  n2 = new Node<Integer>(INIT_CONSTANT);   // progresively
Node<Integer>  n3 = new Node<Integer>(INIT_CONSTANT);

g.addAll(Arrays.asList(n0, n1, n2, n3));

g.connect(n0, 1, n1);
g.connect(n0, 7, n2);
g.connect(n1, 2, n2);
g.connect(n1, 10, n3);
g.connect(n2, 3, n3);

System.out.println("Initial graph: \n"+g);
// Execution strategy “as long as possible”
RuleSetWithStrategy<Node<Integer>> rs = new RuleSetWithStrategy<Node<Integer>>(new AsLongAsPossible<>());

rs.add( Rule.<Node<Integer>>rule("r1", "reduces the value of the node").  // This rule implements Dijkstra’s algorithm!
            when(z -> g.exists( x -> x.isConnectedTo(z) &&
                                     x.getValue() + (Integer)x.getEdgeValues(z).get(0) < z.getValue() ) ).
            exec(z -> z.setValue(g.getWitness().get().getValue()+
                                 (Integer) g.getWitness().get().getEdgeValues(z).get(0))));

rs.setExecContext( g );
rs.process();

System.out.println("Nodes of the initial graph: \n"+new ArrayList<>(g));
System.out.println("(Some) correctness tests: ");
System.out.println("No unreachable nodes: "+g.forAll( n -> n.getValue() < INIT_CONSTANT));
System.out.println("Only one initial node: "+g.one( n -> n.getValue().equals(0)));
```

In this section you must code two execution strategies: `AsLongAsPossible` and `Sequence` (the default strategy from the previous exercise). Make a design that facilitates adding new concrete implementation strategies.

Check that the rule calculates correctly the minimum path using the `AsLongAsPossible` strategy, but not with `Sequence` (since the rules must be applied iteratively). The execution of the previous program should produce the following output:

```
Initial graph:
Nodes:
 0 [0]
 1 [1000]
 2 [1000]
 3 [1000]
Edges:
 ( 0 --1--> 1 )
 ( 0 --7--> 2 )
 ( 1 --2--> 2 )
 ( 1 --10--> 3 )
 ( 2 --3--> 3 )

Nodes of the initial graph:
[0 [0], 1 [1], 2 [3], 3 [6]]
(Some) correctness tests:
No unreachable nodes: true
Only one initial node: true
```

[typo in the output corrected in yellow]

Section 5 (Optional, 1 point): Triggered rules and more design patterns.

Create trigger rules (`trigger`). These rules are associated with the modification of an attribute of an object. In this way, when the attribute changes, the guard (`when`) of all the rules associated with such attribute is evaluated. The rules whose condition is fulfilled will be executed in sequence.

As an example, the following listing:

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");

Product p1 =    new Product(10, sdf.parse("15/04/2020")); // “similar” to class Product of section 2

TriggeredRule.<Product>trigRule("r1").
              trigger(p1, "price").
              when(pro -> Product.getDateDiff(Calendar.getInstance().getTime(), pro.getCaducidad(), TimeUnit.DAYS) < 2 ).
              exec(pro -> { System.out.println("Beware! you are changing the price of a product that is about to expire ");
});
p1.setPrice(17);
```

would produce the following output:

```
Beware! you are changing the price of a product that is about to expire
```

Which design pattern(s) have you used?

How to submit:

You should submit

- an **src** folder with the Java code of all sections, including test data, and additional testers that you have developed
- a **doc** folder with the generated documentation
- a PDF file with with the **class diagram** of your design, a brief justification of the decisions that have been made in the development of the assignment, the main problems that have been addressed and how they have been solved
- If you have done the optional section, submit it in another project or folder

You have to pack everything in a unique ZIP file, named as follows: GR<groupnumber>_<studentnames>.zip. For example, for Marisa and Pedro of group 2213, the file name is: GR2213_MarisaPedro.zip.