Práctica 5

Genericidad, Colecciones, Lambdas y Patrones de Diseño

Inicio: Semana del 13 de abril.
Duración: 4 semanas.
Entrega: Viernes 8 de Mayo (todos los grupos).
Peso de la práctica: 30%

El objetivo de esta práctica es ejercitar conceptos de orientación más avanzados, como son

- *Diseño de clases genéricas, y altamente reutilizables*
- *Uso de la librería de colecciones Java*.
- *Empleo de patrones de diseño y estrategias de diseño de APIs*
- *Uso de expresiones lambda*

Apartado 0. Introducción

En esta práctica vamos a ir desarrollando progresivamente una aplicación para el procesamiento de objetos mediantes reglas sencillas. En nuestro caso, una regla está formada por una condición de aplicación y un efecto. Si el objeto que se pasa como parámetro cumple la condición, entonces se ejecuta la regla, aplicando su efecto.

En la práctica aplicaremos este estilo de programación a la realización de cómputos sobre grafos, y para ello se construirá una estructura genérica para almacenar nodos y enlaces con distintos tipos de valor.

Apartado 1. *Genericidad y Colecciones*: Un grafo dirigido (3 puntos)

Comenzaremos construyendo una clase genérica grafo (`Graph`), que permita almacenar nodos (`Nodes`) de cierto tipo, enlazados por enlaces (`Edges`) dirigidos. Dichos enlaces contendrán además información de otro tipo, configurable por el programador. El grafo no debe admitir nodos repetidos (considerando igualdad referencial), pero un nodo puede conectarse varias veces con el mismo nodo. Diseña la clase de manera que tenga un API similar al resto de colecciones, y sea interoperable con ellas, permitiendo por ejemplo crear una lista de nodos a partir de un grafo.

A modo de ejemplo, el siguiente listado muestra un grafo con nodos que contienen strings, y enlaces que contienen enteros.

```
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));       // no admite repetidos, considerando igualdad referencial

g.connect(n1, 0, n1);                      // conectamos n1 con n1 a través de enlace con valor 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);                     // El grafo contiene 2 nodos y 5 enlaces

for (Node<String> n : g)                   // Colección de dos nodos (n1 y n2)
       System.out.println("Nodo "+n);

List<Node<String>> nodos = new ArrayList<>(g);    // podemos crear una lista a partir de g
System.out.println(nodos);

// Dos métodos para chequeo de conexión, la primera recibe el valor del Nodo
System.out.println("s0 conectado con ‘s1’: "+ n1.isConnectedTo("s1"));
System.out.println("s0 conectado con s1: "+ n1.isConnectedTo(n2));
System.out.println("vecinos de s0: "+n1.neighbours());
System.out.println("valores de los enlaces desde s0 a s1: "+n1.getEdgeValues(n2));
```

La ejecución del programa anterior debe producir la siguiente salida:

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

Nodo 0 [s0]
Nodo 1 [s1]
[0 [s0], 1 [s1]]
s0 conectado con ‘s1’: true
s0 conectado con s1: true
vecinos de s0: [0 [s0], 1 [s1]]
valores de los enlaces desde s0 a s1: [1, 0]
```

Como ves, internamente los nodos deben tener un identificador (de cero en adelante), que es útil a la hora de imprimir o depurar un grafo. Los nodos y enlaces se muestran en orden de inserción. Cuando un nodo se elimina de un grafo con el método `remove`, o `removeAll`, se ha de eliminar dicho nodo como vecino de cualquier otro nodo del grafo. Sólo se permite conectar nodos que pertenezcan al mismo grafo.

Diseña un API para `Node` y `Graph` que permita realizar operaciones comunes sobre nodos, por ejemplo: `isConnectedTo` que nos diga si un nodo está directamente conectado a otro, `neighbours` que devuelva la colección de nodos directamente conectados a uno dado, o `getEdgeValues` que devuelva los valores de los enlaces (sin incluir repetidos) que une un nodo dado con otro.

Apartado 2. Comparadores y Lambdas: La clase *ConstrainedGraph* y *BlackBoxComparator* (2 puntos)

a) Tomando como base la clase `Graph`, crea una clase `ConstrainedGraph` que permita chequear diversas propiedades sobre los nodos de un grafo. Las propiedades serán de tres tipos: universales, existenciales y unitarias. Una propiedad universal se cumple si todos los nodos del grafo la cumplen. Una propiedad existencial se cumple, si al menos un nodo la cumple. Una propiedad unitaria se cumple si existe exactamente un nodo que la cumple.

```
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("Todos nodos de g conectados con n3? "+g.forAll(n -> n.equals(n3) || n.isConnectedTo(n3)));    // true
System.out.println("Existe exactamente un nodo conectado con n2? "+g.one( n -> n.isConnectedTo(n2)));             // true
System.out.println("Existe al menos un nodo conectado con n2? "+g.exists( n -> n.isConnectedTo(n2)));             // (*) true
```

Como ves, los métodos `exists`, `one` y `forall` reciben una expresión lambda como parámetro. Como probablemente sepas de la clase de teoría, una expresión lambda no es más que una notación compacta para una clase anónima. Así, la línea marcada como (*) en el listado anterior podría reescribirse con una clase anónima, como sigue:

```
System.out.println("Existe al menos un nodo conectado con n2? "+
                   g.exists(new Predicate<Node<Integer>>() {
                        @Override public boolean test(Node<Integer> n) { return n.isConnectedTo(n2); }}));
```

Para simplificar apartados posteriores, tomaremos la convención de que, si una propiedad existencial se satisface, el nodo que la satisface se almacenará en un atributo del grafo, de tal manera que una llamada al método `getWitness()` devolverá dicho nodo, o null. Para unificar ambas posibilidades el método deberá devolver un objeto `Optional`. Por ejemplo, dado el grafo g anterior, las siguientes 4 lineas:

```
g.exists( n -> n.getValue().equals(89));        // No se cumple: Optional es null
g.getWitness().ifPresent( w -> System.out.println("Witness 1 = "+g.getWitness().get()));
g.exists( n -> n.isConnectedTo(n2));            // Se cumple: Optional tiene valor
g.getWitness().ifPresent( w -> System.out.println("Witness 2 = "+g.getWitness().get()));
```

Deben imprimir: `Witness 2 = 0 [1],` donde 0 es el identificador del nodo y 1 su valor

b) Crea una clase comparadora de grafos llamada `BlackBoxComparator`. Esta clase se configurará con propiedades existenciales, unitarias y universales. El criterio de comparación será el número de propiedades (de cualquier tipo) satisfecha por los grafos. Por ejemplo, si un grafo g1 satisface 2 propiedades, y otro grafo g2 satisface 3, entonces g1 < g2. Así, si comparamos el grafo g anterior con el siguiente grafo g1 mediante el comparador bbc:

```
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)).        // corregido
    addCriteria( Criteria.UNITARY, n -> n.neighbours().isEmpty()).
    addCriteria( Criteria.UNIVERSAL, n -> n.getValue().equals(4));
```

y usamos dicho comparador para ordenar una lista con g y g1, debemos obtener una lista done g aparezca en primer lugar, y luego g1:

```
List<ConstrainedGraph<Integer, Integer>> cgs = Arrays.asList(g, g1);
Collections.sort(cgs, bbc);     // Usamos el comparador para ordenar una lista de dos grafos
System.out.println(cgs);        // imprime g (cumple la 2ª propiedad) y luego g1 (cumple la 2º y 3º)  (corregida)
```

Apartado 3. Genericidad y Lambdas: Reglas y Conjuntos de Reglas (3 puntos)

Una vez que hemos diseñado una estructura de datos genérica para los grafos, nos centraremos en definir reglas (Rule) y conjuntos de reglas (RuleSet). Como hemos adelantado, una regla se evalúa sobre objetos de un tipo dado. Si la guarda se satisface, entonces podemos ejecutar la regla. Una vez que hemos definido un conjunto de reglas, podemos evaluarlas sobre una colección de objetos del tipo correspondiente. El siguiente listado ilustra el uso de reglas y conjuntos de reglas. En él se usa una clase Producto, para mostrar que tus reglas no sólo serán aplicables a grafos, sino a objetos de cualquier otra clase.

```
class Producto {  // Una clase para probar las reglas
  private double precio;
  private Date caducidad;  // Otra opción es usar Calendar

  public Producto (double p, Date c) {
        this.precio = p;
        this.caducidad = c;
  }

  public double getPrecio()         { return this.precio; }
  public void   setPrecio(double p) { this.precio = p; }
  public Date   getCaducidad()      { return this.caducidad; }

  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.precio+", caducidad: "+this.caducidad;        }
}

public class Main {
  public static void main(String...args) throws ParseException{
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
    RuleSet<Producto> rs = new RuleSet<Producto>();     // Un conjunto de reglas aplicables sobre Productos

    rs.add(
           Rule.<Producto>rule("r1", "Rebaja un 10% los productos con fecha de caducidad cercana o pasada").
                when(pro -> Producto.getDateDiff(Calendar.getInstance().getTime(), pro.getCaducidad(), TimeUnit.DAYS) < 2 ).
                exec(pro -> pro.setPrecio(pro.getPrecio()-pro.getPrecio()*0.1))
           ).add(
           Rule.<Producto>rule("r2", "Rebaja un 5% los productos que valen más de 10 euros").
                when(pro -> pro.getPrecio() > 10).
                exec(pro -> pro.setPrecio(pro.getPrecio()-pro.getPrecio()*0.05))
    );

    List<Producto> str = Arrays.asList( new Producto(10, sdf.parse("15/04/2020")),  // parseamos a un Date
                                        new Producto(20, sdf.parse("20/03/2021")));

    rs.setExecContext(str);     // indicamos que el conjunto de reglas rs se ejecutará sobre str
    rs.process();               // ejecutamos el conjunto de reglas

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

Como ves, tanto las reglas como los conjuntos de reglas se parametrizan por el tipo del objeto que tratan. Los métodos when y exec están preparados para recibir expresiones lambda. Para facilitar la legibilidad del código, y una mejor integración con el estilo de programación Java 8, diseña un API fluida (http://en.wikipedia.org/wiki/Fluent_interface), que permita encadenar llamadas. Así, los métodos rule, when y exec devuelven un objeto Rule para facilitar la construcción de un objeto Rule mediante encadenamiento de llamadas.

La salida del programa anterior es la siguiente (la primera regla aplica al primer objeto y la segunda al segundo):

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

Por el momento, considera una estrategia de aplicación de reglas que consiste en: iterar sobre todos los objetos a procesar, probando cada una de las reglas en el orden en el que se han insertado en el conjunto de reglas. Si una regla es aplicable se ejecuta y se pasa al siguiente objeto a procesar.

Apartado 4. Patrones de diseño: Estrategias de ejecución de reglas (2 puntos)

La estrategia de aplicación de reglas del apartado anterior puede no ser adecuada en algunos escenarios. Por ejemplo, a veces nos puede interesar aplicar las regla tanto como sea posible (es decir, seguir aplicándolas mientras el `when` se cumpla). Para ello, siguiendo el patrón de diseño *Strategy* (http://en.wikipedia.org/wiki/Strategy_pattern) extraeremos la estrategia de ejecución fuera de la clase RuleSet, y pasaremos la estrategia más adecuada en el constructor de `RuleSet`.

Para no modificar la clase `RuleSet` (y facilitar así la corrección), créate una nueva clase `RuleSetWithStrategy`, tomando `RuleSet` como base. El siguiente listado muestra cómo usar dicha clase con una estrategia `AsLongAsPossible`. El listado calcula el camino mínimo desde un nodo de un grafo al resto de nodos, usando el algoritmo de Dijkstra (http://en.wikipedia.org/wiki/Dijkstra's_algorithm) codificado en una única regla. El nodo del que se parte se inicializa con el valor 0, y la distancia a este nodo se guarda como valor de los nodos (que se inicializan con la constante `INIT_CONSTANT`).

```
final int INIT_CONSTANT = 1000;
ConstrainedGraph<Integer, Integer> g = new ConstrainedGraph<Integer, Integer>();
Node<Integer>  n0 = new Node<Integer>(0);               // El valor del nodo es la longitud del camino. N0 es nodo inicial
Node<Integer>  n1 = new Node<Integer>(INIT_CONSTANT);   // inicializamos el resto a un valor alto, que iremos reduciendo
Node<Integer>  n2 = new Node<Integer>(INIT_CONSTANT);
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("Grafo inicial: \n"+g);
// Estrategia de ejecución “as long as possible”
RuleSetWithStrategy<Node<Integer>> rs = new RuleSetWithStrategy<Node<Integer>>(new AsLongAsPossible<>());

rs.add( Rule.<Node<Integer>>rule("r1", "disminuye el valor del nodo").  // Esta regla implementa Dijkstra!
             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("Nodos del grafo final: \n"+new ArrayList<>(g));
System.out.println("(Algunos) tests de corrección: ");
System.out.println("No hay nodos inalcanzables: "+g.forAll( n -> n.getValue() < INIT_CONSTANT));
System.out.println("Hay un sólo nodo inicial: "+g.one( n -> n.getValue().equals(0)));
```

En este apartado debes codificar dos estrategias de ejecución: `AsLongAsPossible` y `Sequence` (la estrategia por defecto del ejercicio anterior). Has de procurar que sea fácil añadir nuevas estrategias concretas de ejecución.

Comprueba que la regla calcula bien el camino mínimo usando la estrategia `AsLongAsPossible`, pero no con `Sequence` (ya que las reglas se deben aplicar iterativamente). La ejecución del programa anterior debe dar la siguiente salida:

```
Grafo inicial:
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 )

Nodos del grafo final:
[0 [0], 1 [1], 2 [3], 3 [6]]
(Algunos) tests de corrección:
No hay nodos inalcanzables: true
Hay un sólo nodo inicial: true
```

[Nota: errata en salida corregida, marcada en amarillo]

Apartado 5 (Opcional, 1 punto): Reglas con disparador y más patrones de diseño.

Crea reglas con disparador (`trigger`). Estas reglas se asocian a la modificación de un atributo de un objeto. De esta manera, cuando el atributo cambia, la guarda (`when`) de todas las reglas asociadas a dicho atributo se evalúa. Aquellas reglas cuya guarda se cumpla, se ejecutarán en secuencia.

A modo de ejemplo, el siguiente listado:

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

Producto p1 =   new Producto(10, sdf.parse("15/04/2020")); // “similar” a la clase Producto del apartado 2

TriggeredRule.<Producto>trigRule("r1").
              trigger(p1, "precio").
              when(pro -> Producto.getDateDiff(Calendar.getInstance().getTime(), pro.getCaducidad(), TimeUnit.DAYS) < 2 ).
              exec(pro -> { System.out.println("Ojo! cambias el precio de un producto próximo a caducar"); });

p1.setPrecio(17);
```

Produciría la siguiente salida:

```
Ojo! cambias el precio de un producto próximo a caducar
```

¿Qué patrón(es) de diseño has utilizado?

Normas de Entrega:

- Se deberá entregar
  - un directorio **src** con el código Java de cada apartado, incluidos los testers que hayas realizado
  - un directorio **doc** con la documentación generada
  - un archivo PDF con el **diagrama de clases** de tu diseño, una breve justificación de las decisiones que se hayan tomado en el desarrollo de la práctica, los problemas principales que se han abordado y cómo se han resuelto.
- Se debe entregar un único fichero ZIP con todo lo solicitado, que deberá llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2261, entregarían el fichero: GR2261_MarisaPedro.zip.