PRACTICA 5

Genericidad, Colecciones, Lambdas y Patrones de Diseño

Alejandro Antonio Martín Almansa
Mario García Roque

Apartado 1:

En el apartado 1 de esta práctica desarrollamos las clases Graph, Node y Edge para trabajar con ellas en el apartado 1 y en el 2, realizando diferentes tareas como conectar nodos con ejes dentro de un grafo.
Código de la clase Graph:

```
package grafo;

import java.util.*;
import java.util.Map.Entry;

/**
 * Clase Graph <N, E>
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
public class Graph<N, E> implements Collection<Node<N>> {

	protected List<Node<N>> nodos;
	protected Map<Node<N>, List<Edge<E>>> map;

	/**
	 * Constructor de la clase Graph
	 */
	public Graph() {
		this.nodos = new ArrayList<Node<N>>();
		this.map = new HashMap<Node<N>, List<Edge<E>>>();

	}

	/**
	 * Metodo para saber si un nodo esta conectado a otro
	 *
	 * @param n1
	 *            primer nodo a comprobar
	 * @param n2
	 *            segundo nodo a comprobar
	 * @return true si esta conectado, false en caso contrario
	 */
	public boolean conectado(Node<N> n1, Node<N> n2) {

		for (Entry<Node<N>, List<Edge<E>>> entrada : map.entrySet()) {
			for (Edge<E> e : entrada.getValue()) {
				if (e.isConnnected(n1, n2))
					return true;
			}
		}

		return false;

	}

	/**
	 * Metodo para obtener los ejes entre dos nodos
	 *
	 * @param n1
	 *            nodo inicial
	 * @param n2
	 *            nodo destino
	 * @return lista de ejes entre los dos nodos
	 */
	public List<E> getEdges(Node<N> n1, Node<N> n2) {

		List<E> list = new ArrayList<E>();
```

```
		for (Entry<Node<N>, List<Edge<E>>> entrada : map.entrySet()) {
			if (entrada.getKey().equals(n1))
				for (Edge<E> e : entrada.getValue()) {
					if (e.getDestino().equals(n2))
						list.add(e.getInfo());
				}

		}

		return list;

	}

	/**
	 * Metodo para obtener la lista de nodos de un grafo
	 *
	 * @return la lista de nodos de un grafo
	 */
	public List<Node<N>> getNodos() {
		return this.nodos;
	}

	/**
	 * Metodo para obtener el mapa de un grafo
	 *
	 * @return mapa del grafo
	 */
	public Map<Node<N>, List<Edge<E>>> getMap() {
		return this.map;
	}

	/**
	 * Metodo para conectar dos nodos con un eje
	 *
	 * @param n1
	 *            nodo inicial
	 * @param e
	 *            eje para conectar los nodos
	 * @param n2
	 *            nodo final
	 * @return true si se ha conectado correctamente, false en caso contrario
	 */
	public boolean connect(Node<N> n1, E e, Node<N> n2) {

		Edge<E> edge = new Edge<E>(e, n1, n2);

		if ((nodos.contains(n1) && nodos.contains(n2)) == false) {
			return false;
		}

		if (map.containsKey(n1)) {
			map.get(n1).add(edge);
		} else {
			List<Edge<E>> lst = new ArrayList<Edge<E>>();
			lst.add(edge);
			map.put(n1, lst);
		}

		return false;
	}

	/**
	 * Metodo para eliminar un nodo
	 *
	 * @param n
	 *            nodo a eliminar
	 * @return true si se elimina correctamente, false en caso contrario
```

```
 */
public boolean remove(Node<N> n) {
	if (nodos.contains(n)) {
		nodos.remove(n);
		map.remove(n);
		return true;
	}
	return false;
}

/**
 * Metodo para eliminar un nodo de todo el grafo
 *
 * @param n
 *            nodo a eliminar
 * @return true si se elimina correctamente, false en caso contrario
 */
public boolean removeAll(Node<N> n) {
	if (nodos.contains(n)) {
		while (nodos.remove(n))
			;
		while (map.remove(n) != null)
			;
		return true;
	}
	return false;
}

/**
 * Metodo que anade un nodo a un grafo
 *
 * @param e
 *            nodo a anadir al grafo
 * @return true si se ha anadido correctamente, false en caso contrario
 */
@Override
public boolean add(Node<N> e) {
	if (nodos.contains(e))
		return false;
	nodos.add(e);
	e.setGraph(this);
	if (map.containsKey(e))
		System.out.println(map.put(e, null));
	return true;
}

/**
 * Metodo para anadir los nodos a un grafo
 *
 * @param c
 *            coleccion de nodos a anadir
 * @return true si se han anadido correctamente, false en caso contrario
 */
@Override
public boolean addAll(Collection<? extends Node<N>> c) {
	boolean ret = true;
	for (Node<N> n : c)
		ret = add(n);
	return ret;
}

/**
 * Metodo para limpiar los nodos
 */
@Override
public void clear() {
	this.nodos.clear();
```

```
}

/**
 * Metodo para ver si el grafo contiene un objeto
 *
 * @param o
 *            objeto a comprobar
 * @return true si lo contine, false en caso conraio
 */
@Override
public boolean contains(Object o) {
      if (nodos.contains(o))
            return true;
      return false;
}

/**
 * Metodo para ver si el grafo contiene los objetos
 *
 * @param c
 *            coleccion de nodos a comprobar
 * @return true si los contiene, false en caso contrario
 */
@Override
public boolean containsAll(Collection<?> c) {

      return this.nodos.containsAll(c);
}

/**
 * Metodo para comprobar si la lista de nodos esta vacia
 *
 * @return true si esta vacia, false en caso contrario
 */
@Override
public boolean isEmpty() {
      if (nodos.isEmpty())
            return true;
      return false;
}

/**
 * Metodo iterator sobre los nodos
 *
 * @return iterdor de la lista de nodos
 */
@Override
public Iterator<Node<N>> iterator() {
      Iterator<Node<N>> i = nodos.iterator();
      return i;
}

/**
 * Metodo para eliminar un nodo
 *
 * @param n
 *            nodo a eliminar
 * @return true si se elimina correctamente, false en caso contrario
 */
@Override
public boolean remove(Object o) {

      return this.nodos.remove(o);
}

/**
 * Metodo para eliminar un nodo de todo el grafo
```

```
 *
 * @param n
 *            nodo a eliminar
 * @return true si se elimina correctamente, false en caso contrario
 */
@Override
public boolean removeAll(Collection<?> c) {

	return this.nodos.removeAll(c);
}

/**
 * Metodo para ver si el grafo retiene los objetos de la coleccion
 *
 * @param c
 *            coleccion de objetos a comprobar
 * @return true si se ha comprobado correctamente, false en caso contrario
 */
@Override
public boolean retainAll(Collection<?> c) {

	return this.nodos.retainAll(c);
}

/**
 * Metodo para obtener el tamano de la lista de nodos
 *
 * @return tamano de la lista de nodos
 */
@Override
public int size() {

	return this.nodos.size();
}

/**
 * Metodo para converit la lista de nodos en array
 *
 * @return array de nodos convertido
 */
@Override
public Object[] toArray() {

	return this.nodos.toArray();
}

/**
 * Metodo para convertir en generico un array
 *
 * @return array convertido
 */
@Override
public <T> T[] toArray(T[] a) {

	return this.nodos.toArray(a);
}

/**
 * toString de la clase Grph
 */
public String toString() {
	String ret = new String();
	ret += "Nodes:\n";
	for (Node<N> n : nodos) {
		ret += n.toString() + "\n";
	}
	ret += "\nEdges:\n";
```

```
            for (Entry<Node<N>, List<Edge<E>>> e : map.entrySet()) {
                  for (Edge<E> edge : e.getValue()) {
                        ret += edge.toString() + "\n";
                  }
            }
            return ret;
      }
}
```

Código de la clase Node:

```
package grafo;

import java.util.*;

/**
 * Clase Node<N>
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
public class Node<N> {

      private static int numNodes = 0;
      private int id;
      private N info;
      private Graph<N, ?> graph = null;

      /**
       * Constructor de la clase Node
       *
       * @param i
       *            info a asignarle al nodo
       */
      public Node(N i) {
            this.info = i;
            this.id = Node.numNodes++;
      }

      /**
       * Metodo para obtener el id de un nodo
       *
       * @return el id del nodo
       */
      public int getId() {
            return id;
      }

      /**
       * Metodo para asigar un id a un nodo
       *
       * @param id
       *            a asignar al nodo
       */
      public void setId(int id) {
            this.id = id;
      }

      /**
       * Metodo para obtener el valor del nodo
       *
       * @return el valor del nodo
       */
      public N getInfo() {
            return info;
      }
```

```java
/**
 * Metodo para asignar un valor a un nodo
 *
 * @param info
 *            valor a asignar al nodo
 */
public void setInfo(N info) {
	this.info = info;
}

/**
 * Metodo para obtener el grafo al cual pertenece el nodo
 *
 * @return el grafo al cual pertenece el nodo
 */
public Graph<N, ?> getGraph() {
	return graph;
}

/**
 * Metodo para asignar un grafo a un nodo
 *
 * @param graph
 *            a asignar al nodo
 */
public void setGraph(Graph<N, ?> graph) {
	this.graph = graph;
}

/**
 * Metodo para comprobar si el nodo esta conectado a otro
 *
 * @param n
 *            info del nodo con el que se quiere comprobar
 * @return true si esta conectado, flase en caso contrario
 */
public boolean isConnectedTo(N n) {

	for (Node<N> nodo : graph.getNodos()) {
		if (nodo.info.equals(n)) {
			return graph.conectado(this, nodo);
		}
	}
	return false;

}

/**
 * Metodo para comprobar si el nodo esta conectado a otro
 *
 * @param n
 *            nodo con el que se quiere comprobar
 * @return true si esta conectado, flase en caso contrario
 */
public boolean isConnectedTo(Node<N> n) {

	return graph.conectado(this, n);
}

/**
 * Metodo para obtener los vecinos de un nodo
 *
 * @return lista de nodos vecinos
 */
public List<Node<N>> neighbours() {

	List<Node<N>> list = new ArrayList<Node<N>>();
```

```
			for (Node<N> nodo : graph.getNodos()) {
				if (this.isConnectedTo(nodo)) {
					list.add(nodo);
				}
			}

			return list;
		}

		/**
		 * Metodo para obtener los ejes de un nodo
		 *
		 * @param n
		 *            nodo del cual se quieren obtener los ejes
		 * @return lista con los ejes
		 */
		public List<?> getEdgeValues(Node<N> n) {

			return graph.getEdges(this, n);

		}

		/**
		 * toString de la clase Node
		 */
		public String toString() {
			return id + " [" + info.toString() + "]";
		}
}
```

Código de la clase Edge:

```
package grafo;

/**
 * Clase Edge <E>
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
public class Edge<E> {
		private static int numEdges = 0;
		private int id;
		private E info;
		private Graph<?, E> graph = null;
		private Node<?> origen;
		private Node<?> destino;

		/**
		 * Constructor de la clase Edge
		 *
		 * @param i
		 *            valor que cuesta recorrer el eje
		 */
		public Edge(E i) {
			this.info = i;
			this.id = Edge.numEdges++;
		}

		/**
		 * Contructor de la clase Edge
		 *
		 * @param i
		 *            valor que cuesta recorrer el eje
		 * @param n1
		 *            nodo inicial del eje
		 * @param n2
```

```
	 *            nodo final del eje
	 */
	public Edge(E i, Node<?> n1, Node<?> n2) {
		this.info = i;
		this.id = Edge.numEdges++;
		this.origen = n1;
		this.destino = n2;
	}

	/**
	 * Metodo para obtener el numero de ejes
	 *
	 * @return el numero de ejes
	 */
	public static int getNumEdges() {
		return numEdges;
	}

	/**
	 * Metodo para asignar un numero de ejes
	 *
	 * @param numEdges
	 *            numero de ejes a asignar
	 */
	public static void setNumEdges(int numEdges) {
		Edge.numEdges = numEdges;
	}

	/**
	 * Metodo para obtener el id de un eje
	 *
	 * @return el id del eje
	 */
	public int getId() {
		return id;
	}

	/**
	 * Metodo para asigar un id a un eje
	 *
	 * @param id
	 *            a asignar al eje
	 */
	public void setId(int id) {
		this.id = id;
	}

	/**
	 * Metodo para obtener el valor del eje
	 *
	 * @return el valor del eje
	 */
	public E getInfo() {
		return info;
	}

	/**
	 * Metodo para asignar un valor a un eje
	 *
	 * @param info
	 *            valor a asignar al eje
	 */
	public void setInfo(E info) {
		this.info = info;
	}

	/**
```

```
	 * Metodo para obtener el grafo al cual pertenece el eje
	 *
	 * @return el grafo al cual pertenece el eje
	 */
	public Graph<?, E> getGraph() {
		return graph;
	}

	/**
	 * Metodo para asignar un grafo a un eje
	 *
	 * @param graph
	 *            a asignar al eje
	 */
	public void setGraph(Graph<?, E> graph) {
		this.graph = graph;
	}

	/**
	 * Metodo para obtener el nodo origen del eje
	 *
	 * @return el nodo origen del eje
	 */
	public Node<?> getOrigen() {
		return origen;
	}

	/**
	 * Metodo para asignar un nodo origen a un eje
	 *
	 * @param origen
	 *            nodo origen a asignar
	 */
	public void setOrigen(Node<?> origen) {
		this.origen = origen;
	}

	/**
	 * Metodo para obtener el nodo destino del eje
	 *
	 * @return el nodo destino del eje
	 */
	public Node<?> getDestino() {
		return destino;
	}

	/**
	 * Metodo para asignar un nodo destino a un eje
	 *
	 * @param origen
	 *            nodo destino a asignar
	 */
	public void setDestino(Node<?> destino) {
		this.destino = destino;
	}

	/**
	 * Metodo para saber si un nodo esta conectado a otro
	 *
	 * @param o
	 *            nodo origen a comprobar
	 * @param d
	 *            nodo destino a comprobar
	 * @return true si esta conectado, false en caso contrario
	 */
	public boolean isConnnected(Node<?> o, Node<?> d) {
		if (o.equals(origen) && d.equals(destino))
```

```
                return true;
            return false;
      }

      /**
       * toString de la clase Edge
       */
      public String toString() {
            return "(" + origen.getId() + " --" + info.toString() + "--> "
                        + destino.getId() + ")";
      }
}
```

Para probar las clases implementadas utilizamos el test que se nos da en el guion de la de la práctica.
Código de la clase de prueba testEjercicio1:

```
package tests;

import grafo.*;
import java.util.*;

/**
 * Clase de prueba del ejercicio 1
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
public class testEjercicio1 {
      public static void main(String[] args) {
            Graph<String, Integer> g = new Graph<String, Integer>();
            Node<String> s0 = new Node<>("s0");
            Node<String> s1 = new Node<>("s1");

            g.addAll(Arrays.asList(s0, s1, s0));
            g.connect(s0, 0, s0);
            g.connect(s0, 1, s1);
            g.connect(s0, 0, s1);
            g.connect(s1, 0, s0);
            g.connect(s1, 1, s0);
            System.out.println(g);
            for (Node<String> n : g)
                  System.out.println("Nodo " + n);
            List<Node<String>> nodos = new ArrayList<>(g);
            System.out.println(nodos);
            System.out.println("s0 conectado con 's1': " + s0.isConnectedTo("s1"));
            System.out.println("s0 conectado con s1: " + s0.isConnectedTo(s1));
            System.out.println("vecinos de s0: " + s0.neighbours());
            System.out.println("valores de los enlaces desde s0 a s1: "
                        + s0.getEdgeValues(s1));

      }

}
```

Salida: La salida esperada como podemos comprobar:

```
<terminated> testEjercicio1 (1) [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw.exe (25/4/2015 20:57:52)
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]
```

Apartado 2:

En este apartado desarrollamos dos nuevas clases: en el apartado a la clase ConstrainedGraph y en el apartado b la clase BlackBoxComparator.
En el apartado a desarrollamos la clase ConstrainedGraph para poder tratar con vaias propiedades de los nodos de un grafo.
Código de la clase ConstrainedGraph:

```
package grafo;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Optional;
import java.util.function.Predicate;

/**
 * Clase ConstrainedGraph <N, E>
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
public class ConstrainedGraph<N, E> extends Graph<N, E> {

	private Node<N> witness;

	/**
	 * Constructor de la clase ConstrainedGraph
	 */
	public ConstrainedGraph() {

		this.nodos = new ArrayList<Node<N>>();
		this.map = new HashMap<Node<N>, List<Edge<E>>>();
		this.witness = null;

	}

	/**
	 * Metodo para comprobar si todos los nodos cumplen una propiedad
	 *
	 * @param pred
	 *            propiedad que tienen que cumplir los nodos
```

```
	 * @return true si cumplen la propiedad todos los nodos, flase en caso
	 *         contrario
	 */
	public boolean forAll(Predicate<Node<N>> pred) {

		this.witness = null;

		for (Node<N> n : this.nodos) {
			if (pred.test(n) == false) {
				return false;
			}
		}

		return true;

	}

	/**
	 * Metodo para comprobar si al menos un nodo cumple una propiedad
	 *
	 * @param pred
	 *            propiedad que tiene que cumplir al menos un nodo
	 * @return true si cumple la propiedad al menos un nodo, flase en caso
	 *         contrario
	 */
	public boolean exists(Predicate<Node<N>> pred) {

		this.witness = null;

		for (Node<N> n : this.nodos) {
			if (pred.test(n) == true) {
				this.witness = n;
				return true;
			}
		}

		return false;

	}

	/**
	 * Metodo para comprobar si un unico nodo cumple una propiedad
	 *
	 * @param pred
	 *            propiedad que tiene que cumplir un unico nodo
	 * @return true si cumple la propiedad un unico un nodo, flase en caso
	 *         contrario
	 */
	public boolean one(Predicate<Node<N>> pred) {

		this.witness = null;

		List<Node<N>> list = new ArrayList<Node<N>>();

		for (Node<N> n : this.nodos) {
			if (pred.test(n) == true) {
				list.add(n);
			}
		}

		if (list.size() == 1) {
			return true;
		}

		return false;

	}
```

```
	/**
	 * Metodo para obtener el witness de un grafo
	 *
	 * @return nodo o null si no existe
	 */
	public Optional<Node<N>> getWitness() {
		return Optional.ofNullable(this.witness);
	}

}
```

En el apartado b desarrollamos la clase BlackBoxComparator como comparador de los grafos del apartado anterior.

Código de la clase BlackBoxComparator:

```
package grafo;

import java.util.*;
import java.util.function.Predicate;

/**
 * Clase BlackBoxComparator<N, E>
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
public class BlackBoxComparator<N, E> implements
		Comparator<ConstrainedGraph<N, E>> {

	/**
	 * Enumerado Criteria para comparar las propiedades de Grafos
	 */
	public enum Criteria {
		Existential {
			@Override
			public <N, E> boolean eval(ConstrainedGraph<N, E> g,
					Predicate<Node<N>> p) {
				return g.exists(p);
			}
		},
		Unitary {
			@Override
			public <N, E> boolean eval(ConstrainedGraph<N, E> g,
					Predicate<Node<N>> p) {
				return g.one(p);
			}
		},
		Universal {
			@Override
			public <N, E> boolean eval(ConstrainedGraph<N, E> g,
					Predicate<Node<N>> p) {
				return g.forAll(p);
			}
		};

		public abstract <N, E> boolean eval(ConstrainedGraph<N, E> g,
				Predicate<Node<N>> p);
	}

	private Map<Criteria, List<Predicate<Node<N>>>> criteria = new EnumMap<Criteria,
List<Predicate<Node<N>>>>(
			Criteria.class);

	/**
	 * Metodo para comparar dos ConstrainedGraph
	 *
	 * @param o1
```

```
	 *            primer grrafo a comparar
	 * @param o2
	 *            segundo grafo a comparar
	 * @return 1 si o1 cumple mas propiedades que o2, -1 si o1 cumple menos
	 *         propiedades que o2 y 0 si o1 y o2 cumplen las mismas propiedades
	 */
	@Override
	public int compare(ConstrainedGraph<N, E> o1, ConstrainedGraph<N, E> o2) {

		int contador_o1 = 0, contador_o2 = 0;

		for (Criteria c : this.criteria.keySet()) {
			for (Predicate<Node<N>> p : this.criteria.get(c)) {
				if (c.eval(o1, p)) {
					contador_o1++;
				}
				if (c.eval(o2, p)) {
					contador_o2++;
				}
			}
		}

		if (contador_o1 > contador_o2) {
			return 1;
		} else if (contador_o1 < contador_o2) {
			return -1;
		}

		return 0;
	}

	/**
	 * Metodo para anadir un predicado a la clave Criteria del mapa
	 *
	 * @param c
	 *            clave a la que se le va a anadir el predicado
	 * @param p
	 *            predicado a anadir a la clave criteria
	 * @return BlackBoxComparator a la cual se han anadido los elementos
	 */
	public BlackBoxComparator<N, E> addCriteria(Criteria c, Predicate<Node<N>> p) {

		if (this.criteria.containsKey(c) == false) {
			this.criteria.put(c, new ArrayList<Predicate<Node<N>>>());
		}

		this.criteria.get(c).add(p);

		return this;

	}

}
```

Para probar las distintas implementaciones de este apartado creamos una clase testEjercicio2.
Código de la clase de Prueba testEjercicio2:

```
package tests;

import java.util.*;
import grafo.*;
import grafo.BlackBoxComparator.Criteria;

/**
 * Clase de prueba del ejercicio 2
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
```

```
 */
public class testEjercicio2 {

	public static void main(String[] args) {

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

		g.exists(n -> n.getInfo().equals(89)); // No se cumple: el nodo witness
												// es null
		g.getWitness().ifPresent(
				w -> System.out.println("Witness 1 = " +
g.getWitness().get()));
		g.exists(n -> n.isConnectedTo(n2)); // Se cumple: el nodo witness estï¿½
											// definido
		g.getWitness().ifPresent(
				w -> System.out.println("Witness 2 = " +
g.getWitness().get()));

		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(2))
				.addCriteria(Criteria.Unitary, n -> n.neighbours().isEmpty())
				.addCriteria(Criteria.Universal, n -> n.getInfo().equals(4));

		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 1ï¿½ propiedad) y luego
									// g1 (cumple la 2ï¿½ y 3ï¿½)

	}
}
```

Salida: La esperada como podemos comprobar en la siguiente foto.

```
<terminated> testEjercicio2 [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw.exe (25/4/2015 21:07:45)
Todo nodo de g conectado con n3? true
Existe exactamente un nodo de g conectado con n2? true
Existe al menos un nodo de g conectado con n2? true
Witness 2 = 0 [1]
[Nodes:
0 [1]
1 [2]
2 [3]

Edges:
(0 --1--> 1)
(0 --7--> 2)
(1 --1--> 2)
, Nodes:
3 [4]

Edges:
]
```

Apartado 3:

En el apartado 3 hemos desarrollado las clases Rule y RuleSet para poder trabajar con las reglas, ejecutarlas, etc.

Código de la clase Rule:

```
package reglas;

import java.util.function.*;

/**
 * Clase Rule<T>
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
public class Rule<T> implements Consumer<T>, Predicate<T> {

	private String nombre;
	private String descripcion;
	private Predicate<T> when;
	private Consumer<T> exec;

	/**
	 * Metodo rule para crear una regla
	 *
	 * @param name
	 *            nombre de la regla
	 * @param description
	 *            descripcion de la regla
	 * @return Regla creada
	 */
	public static <T> Rule<T> rule(String name, String description) {

		Rule<T> r = new Rule<T>(name, description);

		return r;
	}

	/**
	 * Constructor de la clase Rule
	 *
	 * @param nombre
	 *            nombre de la regla
```

```
 * @param descripcion
 *            descripcion de la regla
 */
public Rule(String nombre, String descripcion) {

      this.nombre = nombre;
      this.descripcion = descripcion;
}

/**
 * Metodo que asigna una ejecucion a una regla
 *
 * @param c
 *            ejecucion a asignar
 * @return regla con la ejecucion
 */
public Rule<T> exec(Consumer<T> c) {

      this.exec = c;

      return this;

}

/**
 * Metodo que asigna un predicado a una regla
 *
 * @param p
 *            predicado a asignar
 * @return regla con el predicado
 */
public Rule<T> when(Predicate<T> p) {

      this.when = p;

      return this;

}

/**
 * Metodo que devuelve el predicado de una regla
 *
 * @return predicado de la regla
 */
public Predicate<T> getWhen() {
      return when;
}

/**
 * Metodo para obtener la ejecucion de una regla
 *
 * @return ejecucion de la regla
 */
public Consumer<T> getExec() {
      return exec;
}

/**
 * Metodo test
 *
 * @param t
 *            dato a testear
 */
@Override
public boolean test(T t) {

      return this.when.test(t);
```

```
	}

	/**
	 * Metodo accept
	 *
	 * @param arg0
	 *            dato a comprobar
	 */
	@Override
	public void accept(T arg0) {
		this.exec.accept(arg0);
	}

}
```

Código de la clase RuleSet:

```
package reglas;

import java.util.ArrayList;
import java.util.List;

/**
 * Clase RuleSet<T>
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
public class RuleSet<T> {

	private List<Rule<T>> reglas;
	private List<T> context;

	/**
	 * Constructor de la clase RuleSet
	 */
	public RuleSet() {
		this.reglas = new ArrayList<Rule<T>>();
		this.context = new ArrayList<T>();
	}

	/**
	 * Metodo para anadir una regla a un set de reglas
	 *
	 * @param rule
	 *            regla a anadir
	 * @return el set al cual se le ha anadido la regla
	 */
	public RuleSet<T> add(Rule<T> rule) {

		this.reglas.add(rule);

		return this;
	}

	/**
	 * Metodo para asignar un contexto de ejecucion a un RuleSet
	 *
	 * @param ctx
	 *            contexto a asignar
	 */
	public void setExecContext(List<T> ctx) {
		this.context = ctx;
	}

	/**
	 * Metodo para procesar un RuleSet
	 */
```

```
	public void process() {

		for (T t : this.context) {
			for (Rule<T> r : this.reglas) {
				if (r.getWhen().test(t) == true) {
					r.getExec().accept(t);
				}
			}
		}

	}

}
```

Para este apartado como para los anteriores también desarrollamos un test, testEjercicio3, para comprobar la correcta implementación de las clases.

Código de la clase de prueba testEjercicio3:

```
package tests;

import java.text.*;
import java.util.*;
import java.util.Date;
import java.util.concurrent.TimeUnit;
import reglas.*;

/**
 * Clase Producto para probar el ejercicio 3
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
class Producto {
	private double precio;
	private Date caducidad;

	/**
	 * Constructor de la clase Producto
	 *
	 * @param p
	 *            precio del producto
	 * @param c
	 *            fecha de caducidad del producto
	 */
	public Producto(double p, Date c) {
		this.precio = p;
		this.caducidad = c;
	}

	/**
	 * Metodo para obtener el precio de un producto
	 *
	 * @return precio de un producto
	 */
	public double getPrecio() {
		return this.precio;
	}

	/**
	 * Metodo para asignar un precio a un producto
	 *
	 * @param p
	 *            precio a asignar
	 */
	public void setPrecio(double p) {
		this.precio = p;
	}
```

```
	/**
	 * Metodo para obtener la fecha de caducidad de un producto
	 *
	 * @return fecha de caducidad de un producto
	 */
	public Date getCaducidad() {
		return this.caducidad;
	}

	/**
	 * Metodo para obtener la diferencia entr dos fechas
	 *
	 * @param date1
	 *            primer operando
	 * @param date2
	 *            segundo operando
	 * @param timeUnit
	 *            unidad de tiempo
	 * @return diferencia de las fechas
	 */
	public static long getDateDiff(Date date1, Date date2, TimeUnit timeUnit) {
		long diffInMillies = date2.getTime() - date1.getTime();
		return timeUnit.convert(diffInMillies, TimeUnit.MILLISECONDS);
	}

	/**
	 * toString de la clase Producto
	 */
	@Override
	public String toString() {
		return this.precio + ", caducidad: " + this.caducidad;
	}
}

/**
 * Clase de prueba del ejercicio 3
 *
 * @author Mario Garcia Roque
 * @author Alejandro Antonio Martin Almansa
 *
 */
public class testEjercicio3 {
	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
mÃ¡s 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("15/03/2015")), // parseamos a un

	// Date
				new Producto(20, sdf.parse("20/03/2016")));
		rs.setExecContext(str);
```

```
            rs.process();
            System.out.println(str);
        }
}
```

Salida: La esperada, como bien podemos comprobar en la siguiente foto.

```
<terminated> testEjercicio3 [Java Application] C:\Program Files\Java\jre1.8.0_40\bin\javaw.exe (25/4/2015 21:14:19)
[9.0, caducidad: Sun Mar 15 00:00:00 CET 2015, 19.0, caducidad: Sun Mar 20 00:00:00 CET 2016]
```