**PRÁCTICA 5**

**ANÁLISIS Y DISEÑO DE SOFTWARE**

**Apartado 1:**

Construimos tres clases, **Graph<V, E>**, **Node<V>** y **Edge<E>**. **Graph** va a implementar **Collection**, para poder recorrer sus nodos como si fuesen una lista, y poder operar con ellos. Como un nodo solo puede pertenecer a un grafo, para no duplicar los datos se guardará una variable de tipo **Graph** en el nodo, que representa **null** si el nodo no pertenece a ningún grafo, o al grafo que pertenece.

Cada nodo tiene su propio **INDEX\_COUNT**, que es un id que lo representa de los demás.

La clase **Edge** tiene dos nodos, el **from** y el **to**, que representan el primer nodo que conecta al segundo, respectivamente. El peso de la arista es **weight**.

**Apartado 2:**

**a)** En esta sección del apartado 2, se ha creado la clase **ConstrainedGraph**, que hereda de **Graph**, ya que es un grafo que permite chequear propiedades en sus nodos. En esta clase se han implementado los métodos: **boolean one (Predicate<Node<V>> pred)**, **boolean forAll (Predicate<Node<V>> pred)**, y **boolean exists (Predicate<Node<V>> pred)**. Estos métodos chequean si solo un nodo (unitary), todos los nodos (universal) y si algún nodo (existential) cumple la propiedad pred pasada por argumento, respectivamente.

Además, se ha añadido un campo **Node<V> witness**, para que, si una propiedad existencial se satisface, se guarde uno de los nodos que la satisface. Y este valor se puede obtener con el método **Optional<Node<V>> getWitness ()**.

**b)** En esta segunda sección del apartado, se ha implementado la clase **BlackBoxComparator**, que implementa la interfaz **Comparator**, y compara dos **ConstrainedGraph**, y la enumeración **Criteria**, donde están los tipos de propiedades: unitary, universal, y existential.

La clase **BlackBoxComparator** tiene un atributo **Map<Criteria, Predicate<V>>** donde se mapea el tipo de propiedad, y la propiedad. Para añadir criterios/propiedades se ha creado el método **BlackBoxComparator<V, T> addCriteria (Criteria criterio, Predicate<Node<V>> pred)**,que devuelve el propio comparador para facilitar la programación fluida. Además, en esta clase, también se ha implementado el método compare, donde se compara dos **ConstrainedGraph**, y es mayor el que más propiedades cumple, es decir, cuyos nodos cumplan más propiedades.

**Apartado 3:**

En esta clase se han creado las clases **Rule<T>** y **RuleSet<T>**, donde T es el objeto sobre el que se aplican las reglas.

En la clase Rule, se han implementado los siguientes métodos: **<T> Rule<T> rule (String nombre, String rule)**, **Rule<T> when (Predicate<T>)**, y **Rule<T> exec (Predicate<T>)**. El primero crea una regla parametrizada con los parámetros pasados por argumento; el segundo guarda en una variable la condición que se tiene que cumplir para que se ejecute la regla; y el último, ejecuta la regla.

La clase **RuleSet** está formada por un conjunto de reglas, y en ella, se han implementado los siguientes métodos: **void setExecContext (Collection<T> str)**, **List<Rule<T>> add (Rule<T> exec)**, **y boolean process ()**. El primer método, guarda en una variable los objetos sobre los que se quieren aplicar las reglas; el segundo, añade una nueva regla al set; y el último, ejecuta las reglas sobre los objetos guardados anteriormente, y lo hace pasando por cada objeto de la lista, comprobando si cumple la condición de cada una de las reglas del conjunto, y si es así, se ejecuta la regla que cumple.

**Apartado 4:**

Implementamos **RuleSetWithStrategy**, que es una clase que hereda de **RuleSet** y que sigue una estrategia para ejecutar el método **process**. Hemos implementado un patrón de diseño llamado **strategy**, donde tenemos una interfaz funcional **Strategy** que tiene el método **executeMore**, que devuelve **true** si hay que ejecutar una vez más o **false** si no.

Hay dos estrategias que hemos implementado: **Sequence** y **AsLongAsPossible**. **Sequence** siempre devuelve **false**, así que primero se llama a **process** una vez, **Sequence** devuelve **false** y ya no se ejecuta más. **AsLongAsPossible** devuelve el retorno del último **process**, así que se va a ejecutar hasta que el **process** devuelva false. En caso del algoritmo del Ejercicio 4 es cuando la distancia de todos los nodos al primer nodo no disminuye más.

**Apartado 5:**

Hemos utilizado el patrón de diseño **Observer**. Tenemos la clase **TriggeredRule<T>** que implementa **Observer** y hemos cambiado la clase **Producto** para que herede de **Observable**. Así en el método de **setPrecio** hemos añadido las funciones **this.setChanged(); this.notifyObservers();** para que llame al método **update** de **TriggeredRule**, y así ejecutar la regla, en nuestro caso imprimiendo por pantalla la salida esperada.

Para que pueda funcionar, el genérico **T** debe de extender de **Observable**, así que se lo exigimos en la declaración de la clase **TriggeredRule**.

El diagrama de clases de esta práctica se encuentra en la siguiente página, pero se puede ver mejor en el diagrama\_de\_clase.png que está contenido en el zip (se puede ampliar más fácilmente).

