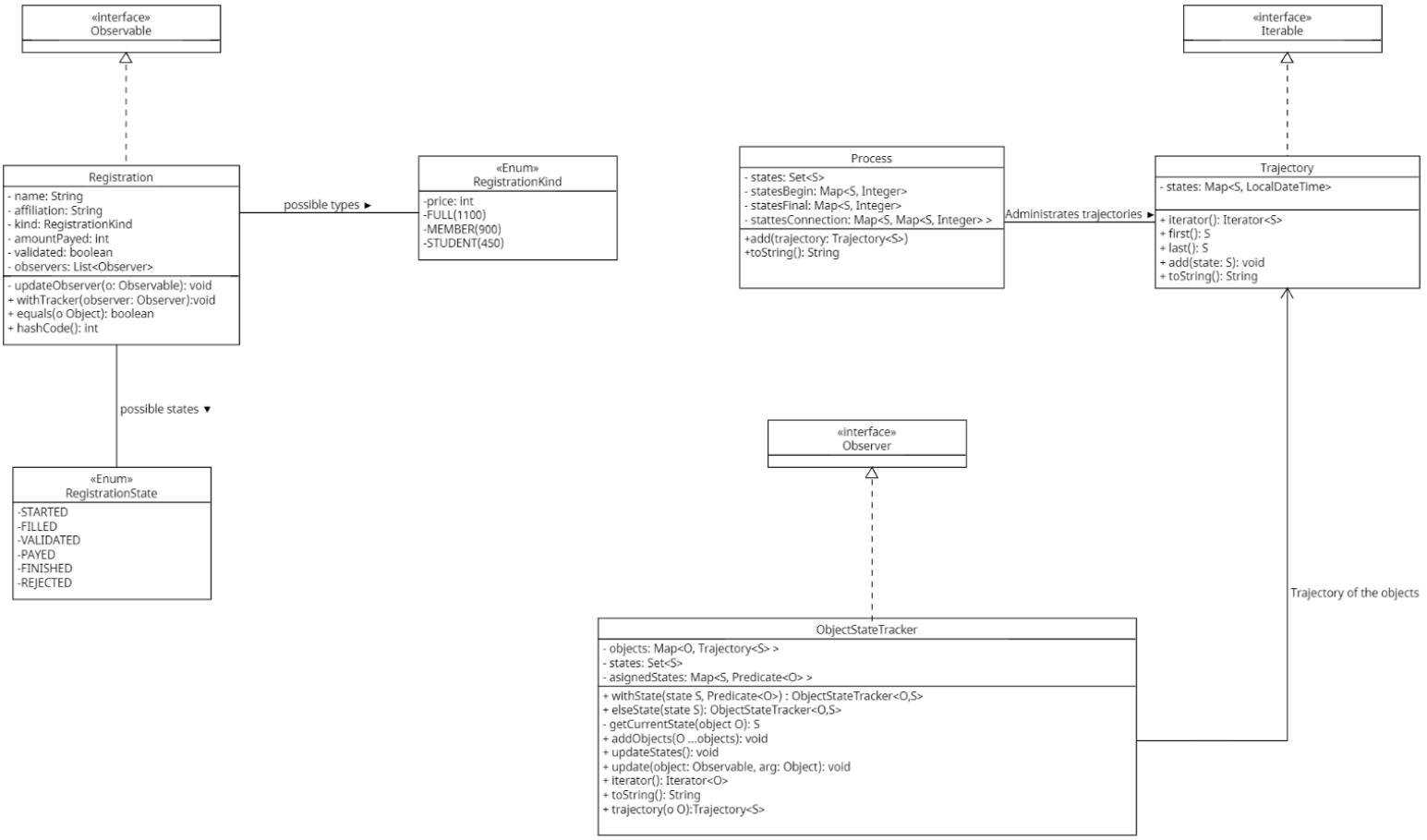
# MEMORIA PRACTICA 5 ANÁLISIS Y DISEÑO DEL SOFTWARE

**Luis Pastor**
**Gonzalo Jiménez**
**2º Ing. Informática**

DIAGRAMA UML:



## Apartado 1

Para este primer apartado se ha creado la clase ObjectStateTracker.

En esta clase se guardan una serie de estados tanto en orden de inserción (un objeto mirara en qué estado está en función de cual se ha añadido primero) como por orden natural (al imprimirse lo hacen por orden natural).

Es por este motivo que hemos decidido usar dos variables para guardar la información:

· states: un set en el que se guardan todos los estados que se pasan por el constructor, almacenados en orden natural.

· asignedStates: LinkedHashMap en el cual se guardan los estados junto con su función asignada. El motivo por el que es un LinkedHashMap es para mantener el orden de inserción

Adicionalmente se tiene una variable extra, para marcar cual es el estado por defecto, en caso de no cuadrar con ninguna función lambda de los estados.

A la hora de guardar los objectos, estos se guardan en orden de inserción en un LinkedHashMap, a estos objetos (keys) se les asigna como valor la trayectoria que siguen.

Para optimizar la creación y lectura de trayectorias, hemos creado una clase donde se guarda la lista de estados por los que se han pasado junto con sus respectivos tiempos.

## Apartado 2

Para realizar el apartado 2 y evitar que se añadir objetos duplicados, hemos añadido a la clase "Register" las funciones "equals()", la cual recibe un objeto como parámetro y "hashCode()", para poder así comparar diferentes registros por los nombres.

Para este apartado no ha sido necesario ni crear más clases, ni variables ni métodos. Esto es debido a que los Maps usan estas funciones para ver si el elemento a insertar ya lo estaba.

## Apartado 3

La clase Process se encarga de dar información con respecto a una serie de trayectorias. Para ello, ha sido necesario usar las siguientes variables:

· states: lista de estados que se pasan por el constructor. Se guardan en un TreeSet para poder así mantener el orden natural.

· statesBegin: es un HashMap en el que se guarda cada uno de los estados existentes junto al número de veces en el que el estado es el primero de las trayectorias

· statesFinal: este mapa se usa de la misma forma que el mapa anterior, excepto porque este solo almacena el número de veces que cada estado ha sido el último de las trayectorias

· statesConnection: es un HashMap en el que se guarda cuantas veces cada estado ha saltado a otro estado. Por ello, la clave de cada entrada será el estado a analizar, y su valor es otro Map con el resto de estados y las veces que ha pasado a ellos.

Gracias a esta organización, el análisis de las trayectorias es sencillo, tan solo es necesasrio iterar e ir viendo, a que estado se está accediendo en función del estado anterior.

## Apartado 4

En el penúltimo apartado se pide que la clase ObjectStateTracker implemente la interfaz "Iterable". Esta ha sido implementada de tal forma que vaya recorriendo la lista de objetos que se han guardado en la clase.

Además de lo anterior y a pesar de que no se pedía, también lo hemos implementado en la clase "Trajectory", ya que hace más rápida y entendible la iteración sobre los estados.

## Apartado 5

Para este último apartado se pide la implementación del patrón de diseño "Observer". Para ello fue necesario modificar la clase "Registration" para que extienda la clase "Obsevable". A pesar de ello, no fue necesario implementar ninguna de las funciones de las que nos proviene, ya que, como se puede ver en el tester, los observadores están arraigados en a la clase, no a los objetos que se instancian de la clase. Es por esto que hemos decidido crear una lista estática de los observadores, y en cuanto se añade uno ("withTracker") se añade a esta lista. Además, cada vez que se realiza algún cambio a un objeto, la función "updateObserver" es llamada. Esta función ejecuta la función “update” de los observadores.

Por último, hemos hecho que nuestra clase "ObjectStateTracker" implemente la interfaz "Observer", y sobrescriba la función “update”. En caso de ejecutarse "update", se volverán a comprobarse los estados de todos los objetos.

A la hora de implementar esta función surgieron dos posibilidades:

a) Únicamente actualizar el objeto que se ha modificado

b) Modificar todos los objetos

Finalmente se tomó la decisión de llevar a cabo la segunda opción, porque no sabemos el tipo de objetos que se van a insertar, ni si entre ellos existen relaciones que, en caso de modificarse algo, puedan ser alteradas. Es decir, para que la implementación sea lo más genérica posible es necesario modificar todos los objetos.

El diagrama UML de la primera página ayuda a ilustrar el funcionamiento y el diseño que se ha tomado.