<u>Práctica 1</u>: Uso de patrones de diseño en orientación a objetos

Alumno: Raúl Castro Moreno y Raúl Rodríguez Pérez
Grupo: GrupoPequeñoPrácticasDS2_2

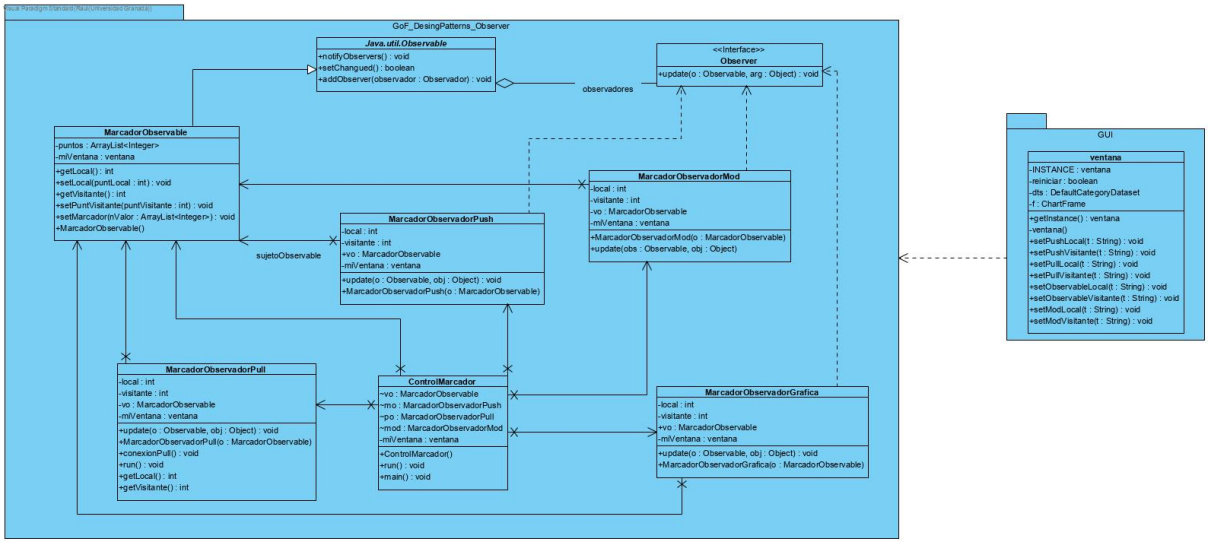
1. Sesión 1ª: Patrón Observador

Vamos a realizar, utilizando el patrón de diseño Observador y haciendo uso del lenguaje java, un programa con una GUI que simule la monitorización de datos del marcador de un partido de fútbol. Dicho programa tendrá un sujeto-observable con dos variables (local, visitante) y dos observadores: MarcadorPush y MarcadorPull. Cada vez que el sujeto actualiza el marcador, deberá notificar el cambio a los observadores que tenga suscritos (en este caso MarcadorPush), mediante el uso de la comunicación push. Por otro lado, el observador MarcadorPull no está suscrito, por lo que se comunicará con el sujeto observable de forma asíncrona (empleando la comunicación pull tal y como especifica su propio nombre).

En el desarrollo del problema, hemos elaborado los requisitos obligatorios, y los dos primeros requisitos opcionales. Por lo que finalmente en nuestro caso, hemos añadido dos nuevos observadores (ambos suscritos): por un lado, MarcadorObservadorMod, el cual es un observador que rompe con la filosofía del patrón de diseño, ya que es capaz de producir cambios en el modelo. Dicho observador modifica el atributo 'local' del sujeto observable cada vez que pulsamos el botón. Y por otro lado tenemos el MarcadorBarras, el cual es un observador que implementa un GUI más sofisticado, mostrando el marcador en una gráfica de barras.

• Diagrama de clase

Antes de ponernos con el código, lo primero que hemos elaborado ha sido el diagrama de clases de nuestro problema. En dicho diagrama se puede apreciar tanto los distintos observadores, como el sujeto observable y la interfaces empleadas:



- **Implementación**

Una vez hecho el diagrama, el siguiente paso consistió en la implementación del código. Tuvimos que, en un primer momento, informarnos y enterarnos exactamente de en qué consistía el patrón observador, y mirar algunos ejemplos de cómo se llevaba a cabo la implementación de dicho patrón. Para ello consultamos diversas fuentes de información a partes de las proporcionadas en prado. Algunas de dichas fuentes las adjuntamos en los siguientes enlaces:

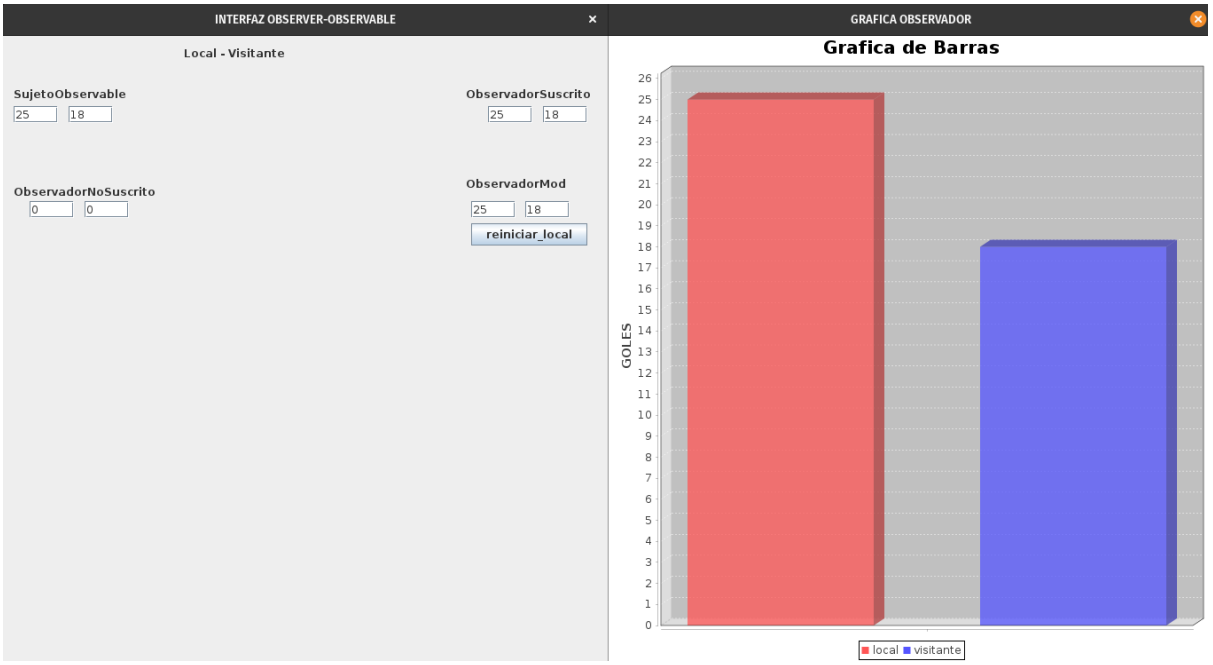
- https://mate.uprh.edu/~jse/cursos/4097/notas/java/javaEspanol/JavaTut/Apendice/ob_uso.html
- https://docs.oracle.com/javase/7/docs/api/java/util/Observable.html
- https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-el-patron-observer/
- https://informaticapc.com/patrones-de-diseno/observer.php

Una vez siendo conscientes de en qué consistía el uso de dicho patrón, nos pusimos manos a la obra y empezamos a trabajar en netbeans. El código final está adjunto en la carpeta por lo que no vamos a mostrar nada en esta documentación. Simplemente nos reiteramos en que, hemos realizado la comunicación entre el sujeto observable y sus observadores suscritos por medio de las funciones específicas “setChanged()” y “notifyObservers()”. Y, por otro lado, hemos realizado la comunicación con el observador no suscrito por medio de hebras tal y como se pide en el guión de la sesión.

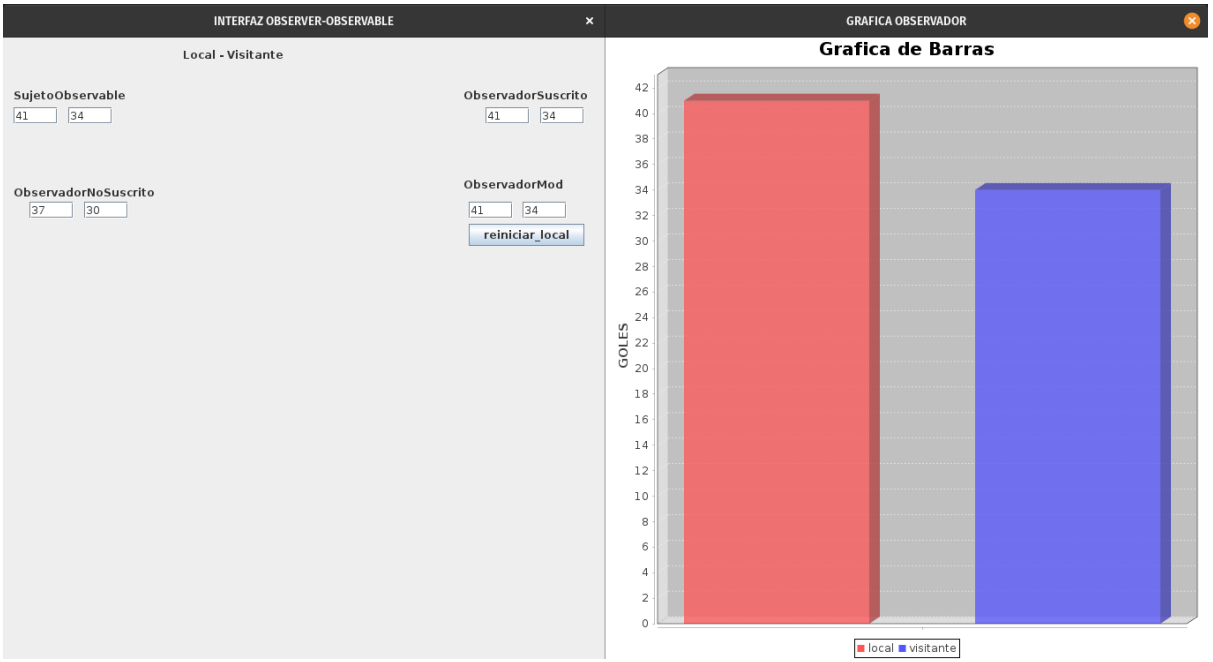
- **Capturas de la ejecución**

Una vez realizado todo el código, hemos querido realizar unas capturas de pantalla de la ejecución, para que quede constancia del resultado final:

En esta primera captura podemos ver como cuando se modifica el sujetoObservable se notifica al instante a los observadores suscritos (ObservadorSuscrito y ObservadorMod), modificando sus propios parametros. Además se aprecia también, como el observador no suscrito no ha realizado aún la comunicación con el sujetoObservable, sus valores siguen siendo 0. Además se ve como el observadorGrafica nos muestra los valores por medio de una gráfica de barras.



Por otro lado en esta segunda captura, observamos que el observadorNoSuscrito ya se ha comunicado con el sujetoObservable por lo que se ha modificado su valor (recordar que se establece la comunicación cada 6 iteraciones). Por otra parte vemos que el comportamiento con los observadores suscritos sigue siendo el mismo comentado anteriormente.



2. Sesión 2ª: Patrones Factoría Abstracta, Método Factoría y prototipo

Vamos a realizar la programación de 2 partidos simultáneos con un número 'N' de jugadores. 'N' no se conoce hasta que comienza el partido. De los dos partidos se retirarán un 20% y un 10% de los jugadores, respectivamente. Ambos partidos duran exactamente 60s. y todos los jugadores se retiran a la misma vez.

Supondremos que **FactoriaPistaYJugador**, una interfaz de Java, declara los métodos de fabricación públicos:

- **crearPista** que devuelve un objeto de alguna subclase de la clase abstracta **Pista**
- **crearJugador** que devuelve un objeto de alguna subclase de la clase abstracta **Jugador**.

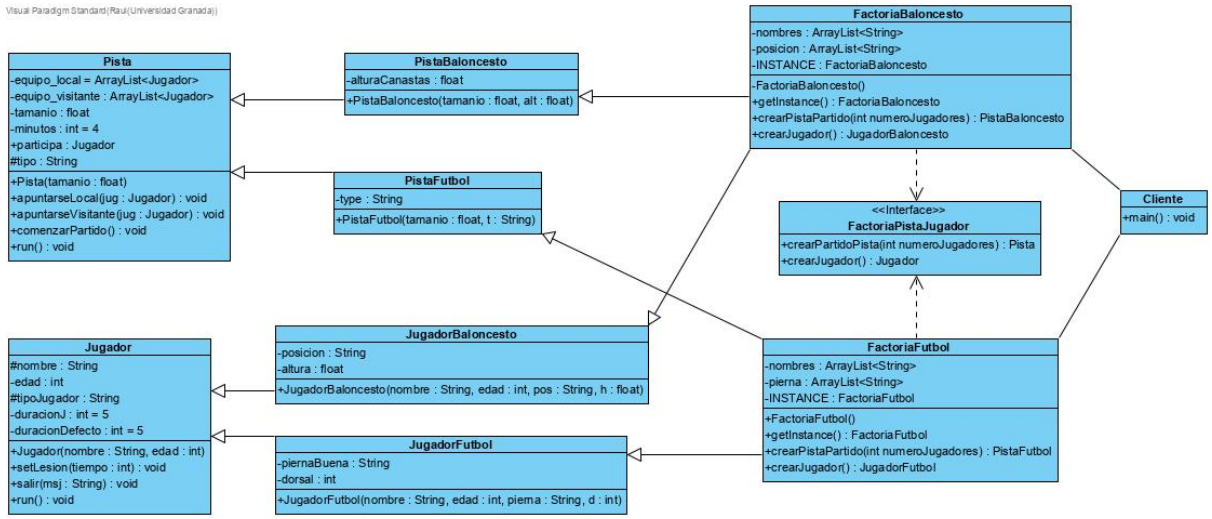
La clase **Pista** tiene al menos un atributo **ArrayList<Jugadores>**, con los jugadores que están jugando en esa pista. La clase **Jugadores** tiene al menos un identificador único de la pista en la que van a jugar. Las clases factoría específicas heredan de **FactoriaPistaYJugador** y cada una de ellas se especializa en un tipo de pista y jugador: las pistas y los jugadores de fútbol, y las pistas y los jugadores de baloncesto. Por consiguiente, asumimos que tenemos dos clases factoría específicas: **FactoriaFutbol** y **FactoriaBaloncesto**, que implementará cada una de ellas los métodos de fabricación **crearPista** y **crearJugador**. Por otra parte, las clases **Jugador** y **Pista** se deberían definir como clases abstractas y especializarse en clases concretas para que la factoría de pistas pueda crear productos **PistaFutbol** y **PistaBaloncesto** y la factoría de jugadores pueda crear productos

JugadorFubol y **JugadorBaloncesto**. Para programar la simulación de cada jugador en cada pista crearemos hebras en Java.

Hemos resuelto de la misma manera el ejercicio de la parte optativa, que consistía en emplear el patrón Prototipo en vez del método factoría, implementandolo esta vez en Ruby.

- ## Diagrama de clase

Antes de ponernos con el código, lo primero que hemos elaborado ha sido el diagrama de clases de nuestro problema.



3. Sesión 3ª: Estilos arquitectónicos: el estilo Filtros de Intercepción

Queremos representar el coste y el tiempo medio de alquiler que realizan diversos clientes en nuestra aplicación. Dichos parámetros serán previamente modificados (filtrados) con filtros capaces de aplicar, por ejemplo, descuentos según el número de personas que alquilen un encuentro (FiltroPersonas), impuestos como puede ser el IVA (FiltroIVA), o un aumento del coste del alquiler en función del tiempo del mismo.

A continuación se explican las entidades de modelado necesarias para programar el estilo Filtros de intercepción para este ejercicio:

- Objetivo (target): Representa la media en coste y tiempo de un alquiler
- Filtro: Interfaz implementada por las clases FiltroPersonas, FiltroIVA y FiltroTiempo. Estos objetos se aplicarán antes de que el objeto de la clase Objetivo ejecute sus propias tareas (mostrar la media del coste y tiempo).

- Cliente: Representa las acciones de realizar un alquiler, que serán interceptadas por los tres filtros comentados en el punto anterior, antes de que el Objeto de la clase objetivo realice su tarea propia.
- GestorFiltros: crea la cadena de filtros y se encarga de que se apliquen

- **Diagrama de clase**

Antes de ponernos con el código, lo primero que hemos elaborado ha sido el diagrama de clases de nuestro problema.