Práctica 1

Uso de patrones de diseño en orientación a objetos

Fecha última actualización: 7 de marzo de 2021.

Para esta práctica se tomará como punto de partida el caso de estudio propuesto por el grupo pequeño de teoría al que pertenece el equipo de prácticas.

1.1. Programación y objetivos

Esta práctica constará de 4 partes, una por sesión. En cada sesión habrá al menos un ejercicio de aplicación de un patrón de diseño. Tendrá una puntuación en la nota final de prácticas de 4 puntos sobre 10. Se deben realizar diagramas de clase para cada ejercicio, que adapten el patrón a cada problema concreto.

1.1.1. Objetivos generales de la práctica 1

1. Familiarizarse con el uso de herramientas que integren las fases de diseño e implementación de código en un marco de Orientación a Objetos (OO)

2. Aprender a aplicar patrones de diseño de distintos tipos

3. Adquirir destreza en la práctica de diseño OO

4. Aprender a aplicar patrones de diseño en distintos lenguajes OO

1.1.2. Planificación y competencias específicas

Como en el resto de las prácticas, se espera del estudiante que en cada sesión de prácticas:

1. Entienda bien lo que se pide.
2. Realice el diseño que utilizará (diagrama de clases) junto con su compañero de prácticas y preguntando a los otros compañeros del grupo pequeño de teoría y al profesor sobre posibles decisiones a tomar.
3. Empiece a implementar. La implementación deberá ser terminada en tiempo de trabajo fuera de la sesión y antes de la siguiente sesión de prácticas.

NOTA: En la primera sesión también se recomienda que instale las herramientas necesarias para la realización completa de la práctica. La URL para descarga de VisualParadigm y obtención de licencia UGR es: https://ap.visual-paradigm.com/universidad-granada.

Sesión	**Semana**	**Competencias**
S1	5-10 marzo	Aplicar los patrones *observador* y *compuesto* en una aplicación Java con una GUI[1].
S2	12-17 marzo	Aplicar tres patrones creacionales en sendas aplicaciones con hebras en Java y Ruby entendiendo las relaciones entre los mismos.
S3	19-24 marzo	Aplicar el patrón *visitante* en c++ junto con otro patrón a elegir.
S4	26 marzo-7 abril	Aplicar el patrón combinado de *filtros de intercepción*.

1.2. Criterios de evaluación

Para superar cada parte será necesario cumplir con todos y cada uno de los siguientes criterios:

- Calidad (se cumplen los requisitos no funcionales)
- Capacidad demostrada de trabajo en equipo (reparto equitativo de tareas)
- Implementación completa y verificabilidad (sin errores de ejecución)

- Fidelidad de la implementación al patrón de diseño
- Reutilización de métodos (ausencia de código redundante)
- Validez (se cumplen los requisitos funcionales)

1.3. Plazos de entrega y presentación de la práctica

La práctica completa será subida a PRADO en una tarea que terminará justo antes del inicio de la primera sesión de la práctica 2 (a las 15:30 horas del día de la primera sesión). Será presentada posteriormente mediante una entrevista con el profesor de prácticas.

1.4. Formato de entrega

Se deben exportar los proyectos generados en cada ejercicio y archivarlos todos juntos, poniendo también en el archivo (válido cualquier formato) la documentación extra que se pida, como por ejemplo los diagramas de clase, en formato pdf. Es preferible un único pdf con toda la documentación de todos los ejercicios de la práctica.

1.5. SESIÓN 1ª: Patrón Observador (1 punto)

1.5.1. Requisitos obligatorios (0,6 puntos)

- Se usará Java como lenguaje de programación. Debe utilizarse la clase abstracta Observable de Java.util y la interfaz Observer de Java (ver Figura 1.1). Debe tenerse en cuenta que el método *notifyObservers* tiene como argumento el *Object* con la información a publicar (no aparece en la Figura 1.1) y que antes de llamar a ese método hay que invocar al método *setChanged* para dejar constancia de que se ha producido un cambio (si no se hace, el método *notifyObservers* no hará nada).
- Se definirán dos observadores, de la siguiente forma:
  1. Observador suscrito convencional (comunicación *push*) mediante método *notifyObservers* y
  2. Observador no suscrito (comunicación *pull*).

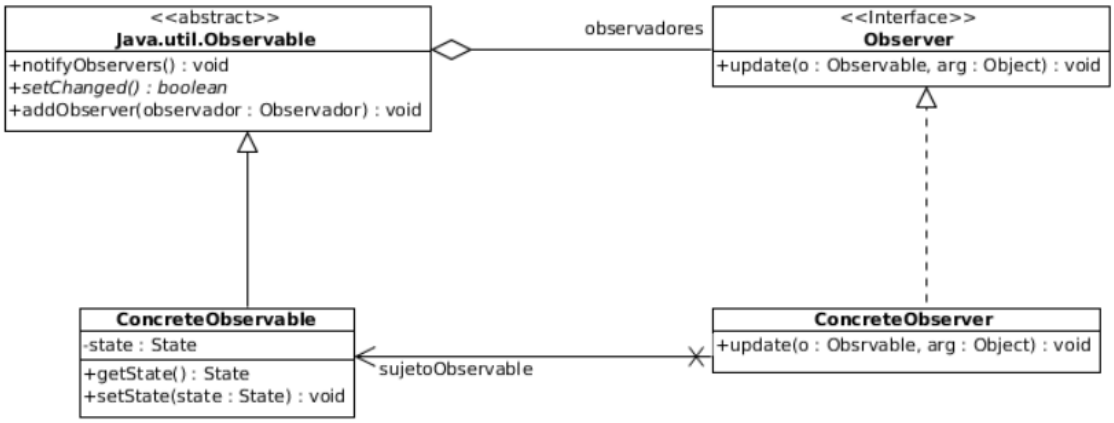

Figura 1.1: Diagrama de clases del patrón Java Observable-Observer.

- Se creará una GUI[2].
- Para programar la simulación de la actualización de datos por parte del modelo y la petición de datos por parte del observador no suscrito, crearemos hebras en Java.

1.5.2. Requisitos opcionales (0,4 puntos)

- Se romperá parte de la filosofía de este patrón de diseño mediante la inclusión de un tercer observador suscrito que pueda producir cambios en el modelo observado.
- Se añadirá un cuarto observador que haga uso de algún recurso GUI más sofisticado (mapas, librerías gráficas externas, etc.).
- Se creará un observador compuesto haciendo uso del patrón de diseño *compuesto*.

Ejemplo (parte obligatoria): Monitorización de datos meteorológicos

Programa en Java, utilizando el patrón de diseño Observador (ver Figura 1.1), un programa con una GUI que simule la monitorización de datos meteorológicos. El programa

[2]Se puede utilizar un asistente para desarrollar la GUI, como el que incluye el IDE NetBeans.

debe crear un sujeto-observable con una *temperatura* y dos observadores: *pantallaTemperatura* y *graficaTemperatura*. Cada vez que el sujeto actualiza su *temperatura*, lo que hace de forma regular (mediante una hebra), deberá notificar el cambio a los observadores que tenga suscritos (solamente *graficaTemperatura*) (comunicación push). El observador *pantallaTemperatura* no está suscrito, sino que se comunicará con el sujeto observable de forma asíncrona (comunicación pull). El observador *pantallaTemperatura* debe mostrar la temperatura tanto en grados centígrados como en Fahrenheit. El observador *graficaTemperatura* debe mostrar una gráfica con las últimas 7 temperaturas recibidas (una por semana), solo en grados Celsius.

Ejemplo (parte optativa): Ampliación con información geográfica

Se añadirán dos nuevos observadores suscritos. El primero, *botonCambio*, además de mostrar la temperatura en grados Celsius, podrá cambiar la temperatura del sujeto observable a petición del usuario. El segundo, *tiempoSatelital*, mostrará la temperatura sobre una posición de un mapa y accederá por suscripción a otros sujetos observables para mostrar las temperaturas en otras posiciones del mapa. Por último, se creará otro observador suscrito como combinación de *graficaTemperatura* y *botonCambio* utilizando el patrón *compuesto* (composite).

¿Cómo crear un hilo en Java?

Para esta sesión y la siguiente, será necesario crear hebras. Hay dos modos de conseguir hilos de ejecución (threads) en Java. Una es implementando el interfaz Runnable, la otra es extender la clase Thread. Si optamos por crear hilos mediante la creación de clases que hereden de Thread, la clase hija debe sobreescribir el método run():

```
class MiThread extends Thread {
 @Override
 public void run() {
        . . .
        }
```

Los métodos java más importantes sobre un hilo son:

- *start*(): arranque explícito de un hilo, que llamará al método *run*()
- *sleep*(*retardo*): espera a ejecutar el hilo retardo milisegundos
- *isAlive*(): devuelve si el hilo está aun vivo, es decir, *true* si no se ha parado con *stop*() ni ha terminado el método *run*()

1.6. SESIÓN 2ª: Patrones *Factoría Abstracta, Método Factoría* y *Prototipo* (1 punto)

Esta sesión constará de dos ejercicios.

1.6.1. Ejercicio S2E1 (obligatorio - 0,6 puntos)

- Se implementará el patrón de diseño *Factoría Abstracta* junto con el patrón de diseño *Método Factoría*.
- Se usará Java como lenguaje de programación.
- Se utilizarán hebras.
- Se usará un interfaz de usuario de texto.
- Se implementarán solo dos familias/estilos de productos aunque en el futuro se podrán añadir otras [3].

1.6.2. Ejercicio S2E2 (opcional - 0,4 puntos)

- Se implementará una solución al mismo problema considerado en el ejercicio anterior, pero haciendo los siguientes cambios:
  - Debe usarse Ruby como lenguaje de programación.
  - Debe usarse el patrón de diseño *Factoría Abstracta* junto con el patrón de diseño *Prototipo*, en vez de hacerlo junto con el patrón de diseño *Método Factoría*, como en el ejercicio anterior.

Ejemplo ejercicio S2E1 (parte obligatoria): Patrón *Factoría Abstracta* y patrón *Método Factoría*

Programa utilizando hebras la simulación de 2 carreras simultáneas con el mismo número inicial (N) de bicicletas. N no se conoce hasta que comienza la carrera. De las

[3] Considérese que el concepto de producto en patrones de diseño, tiene una definición muy abierta, para que pueda ajustarse a cada problema concreto.

carreras de montaña y carretera se retirarán el 20 % y el 10 % de las bicicletas, respectivamente, antes de terminar. Ambas carreras duran exactamente 60 s. y todas las bicicletas se retiran a la misma vez.

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

- *crearCarrera* que devuelve un objeto de alguna subclase de la clase abstracta *Carrera* y
- *crearBicicleta* que devuelve un objeto de alguna subclase de la clase abstracta *Bicicleta*.

La clase *Carrera* tiene al menos un atributo $ArrayList < Bicicleta >$, con las bicicletas que participan en la carrera. La clase *Bicicleta* tiene al menos un identificador único de la bicicleta en una carrera. Las clases factoría específicas heredan de *FactoriaCarreraYBicicleta* y cada una de ellas se especializa en un tipo de carreras y bicicletas: las carreras y bicicletas de montaña y las carreras y bicicletas de carretera. Por consiguiente, asumimos que tenemos dos clases factoría específicas: *FactoriaMontana* y *FactoriaCarretera*, que implementarán cada una de ellas los métodos de fabricación *crearCarrera* y *crearBicicleta*.

Por otra parte, las clases *Bicicleta* y *Carrera* se deberían definir como clases abstractas y especializarse en clases concretas para que la factoría de montaña pueda crear productos *BicicletaMontana* y *CarreraMontana* y la factoría de carretera pueda crear productos *BicicletaCarretera* y *CarreraCarretera*.

Para programar la simulación de cada bicicleta en cada carrera crearemos hebras en Java.

Ejemplo ejercicio S2E2 (parte optativa): Patrón *Factoría Abstracta* y patrón *Prototipo* en Ruby

Diseña e implementa una aplicación inspirada en el ejercicio anterior que cumpla los siguientes requisitos:

- que aplique el patrón *Prototipo* en vez del patrón *Método Factoría*, junto con el patrón *Factoría Abstracta*,
- que no requiera programación concurrente, y
- que se implemente en Ruby[4].

[4]Debe tenerse en cuenta que en Ruby no puede declararse una clase como abstracta. La forma de

impedir instanciar una clase es declarando privado el constructor o utilizando algún mecanismo para que se lance una excepción si el cliente de la clase intenta instanciarla.