Tecnología de Videojuegos
Práctica 6: Patrones de diseño

UAH, Departamento de Automática, ATC-SOL
`http://atc1.aut.uah.es`

Objetivos:

- Profundizar en la comprensión de los patrones de diseño
- Implementar algunos patrones sencillos ampliamente utilizados
- Saber identificar las situaciones en las que pueden aplicarse algunos patrones sencillos
- Comprender la relación entre cómo representamos los datos y la complejidad de los algoritmos resultantes

Comentario inicial

Los patrones de diseño son una herramienta extremadamente útil para realizar el diseño (desde la perspectiva de la programación) y la implementación de un videojuego. También son muy útiles en el desarrollo de cualquier aplicación orientada a objetos, siendo su utilización una competencia importante en el CV de un ingeniero que tenga labores de programación.

En esta práctica vamos a ampliar la implementación del videojuego incorporando la utilización de varios patrones de diseño. Idealmente, se verá en qué mejora el código utilizando los patrones, y se comprenderán mejor.

Ejercicio 1

¿Puedes identificar la aplicación de algún patrón de diseño en el código desarrollado hasta este momento?

Ejercicio 2

Limite el número de instancias del juego que pueden ejecutarse a una, para ello aplique el patrón *Singlenton* a la clase `Juego`. La aplicación de este patrón es muy sencilla. Defina el constructor de la clase `Juego` como privado, añada un atributo de tipo `Juego` inicialmente con valor [null] y un método estático llamado `getJuego()` que la devuelva instancia de juego. En caso de que no hubiese instancia de `Juego`, deberá crear una.

Actividad adicional: Prueba a crear más de una instancia del juego para testear la corrección del código.

Ejercicio 3

Centralice todo el código encargado de la creación tanto de los personajes como de las armas. Para ello vamos a utilizar el patrón `Factory`. Implemente dos clases, `PersonajeFactory` y `ArmaFactory`[1], cada uno con los métodos estáticos `Personaje crearPersonaje(String tipo)` y `Arma crearArma(String tipo)`.

Ejercicio 4

Aplique el patrón `Composite` para mantener un inventario de objetos distintos a las armas (por ejemplo pócimas, hechizos, monedas, llaves, etc). Implemente una pequeña jerarquía de clases con dichos objetos y cambie el código necesario para introducirlos en el mapa. Para simplificar la implementación, se sugiere crear una jerarquía de clases cuya raíz herede de una lista enlazada.

Ejercicio 5 (opcional)

Las estructuras de datos definidas hasta el momento no tienen una organización geométrica, lo que complica la visualización del mapa. Vamos a añadir una capa con información geométrica a los datos de manera que simplifique la visualización del mapa. Para ello se aplicará el patrón `Observer`.

Siga los siguientes pasos:

1. Cree una interfaz llamada `Posicion`, que contenga los dos métodos siguientes:
   - `int getX()`: Devuelve la posición X.
   - `int getY()`: Devuelve la posición Y.
2. Implemente la interfaz `Posicion`[2] en todos los objetos que puedan situarse en el mapa (`Personaje`, `Arma`, `Moneda`, etc)
3. Cree un nuevo atributo `posiciones` en la clase `Mapa` de tipo matriz de interfaces `Posicion`.
4. Aplique el patrón `Observer`: La clase `Mapa` debe implementar la interfaz `Observer` y la clases (`Personaje`, `Arma`, `Moneda`) deben heredar de la clase `Observable`. Cada vez que una de las clases observadas cambie su posición (x o y), se debe actualizar la clase `Mapa` para que lo posicione correctamente en la matriz. De esta manera, cada vez que cualquier objeto cambie de posición, automáticamente la clase `Mapa` actualizará la matriz `posiciones`.

[1] Es habitual utilizar el nombre del patrón en inglés en el nombre de la clase para poder identificar el patrón aplicado con más facilidad.

[2] La falta de tilde en el nombre de la clase es intencionada: Aunque Java permite utilizar tildes y eñe en el nombre de las clases, por lo general es mejor no utilizarlas para evitar problemas.