**Identifica las características principales de los patrones de diseño más conocidos, indica para qué valen y un caso de uso para cada uno de ellos.**

Patrones de diseño más conocidos:

PATRONES CREACIONALES:

**--SINGLETON:**

Se trata de un patrón creacional, que se encarga de reutilizar un mismo objeto las veces que sea necesario, evitando que se creen más objetos de ese tipo y, reaprovechando constantemente el objeto (singleton)

Ej:

Si tenemos una clase que se encarga de convertir archivos de texto de formato txt a formato pdf.

Esa clase solo se instanciaría una vez. Después de haber creado el objeto de esa clase(conversorPDF), lo podríamos utilizar todas las veces que lo necesitemos con los archivos que sean, sin tener que crear un nuevo objeto de la clase cada vez.

**--BUILDER:**

Éste patrón, nos permite crear objetos complejos añadiendo según lo necesitemos, los atributos de manera funcional. En lugar de tener que usar un montón de constructores con diferentes combinaciones de parámetros cada uno.

Viene a ser la base de la programación funcional en Java.

Ej:

Si vamos a crear un objeto de tipo televisor utilizando éste parón.

En la declaración del objeto, podríamos construir un tv, sin conocer ninguno de sus atributos, conociendo solo algunos, o conociéndolos todos. También podremos añadirle, quitarle o modificarle, atributos como serían (smartTV, wifi, bluetooth, modo grabación, memoria externa, …, ). Todo ello usando la programación funcional.

**--FACTORY**

El patrón Factory sirve para crear una jerarquía de clases. Creando una interfaz, unas clases que la implementen de diferentes maneras y, una clase ClaseFactory que se encargará de crear el objeto de la clase que corresponda, según sea el caso.

Ej:

Si tenemos una tienda, podríamos implementar un patrón Factory para gestionar los envíos a todo el mundo. Crearíamos una interfaz Envío, que será implementada por diferentes clases específicas para cada zona geográfica o país.

Al crear un objeto de tipo envío, la clase factory se encargará de instanciar un objeto diferente dependiendo de la zona:

-Ej: Europa, EEUU, Rusia, África, Australia, …

Cada una de esas clases agrupará los costes, empresa de mensajería, trámites aduaneros y demás elementos que necesitaremos para enviar un artículo a cada lugar.

**--PROTOTYPE**

Tiene como objetivo que un objeto pueda clonarse a sí mismo tantas veces como sea necesario. Una vez clonado, podremos trabajar con los dos clones modificándolos o implementando sus métodos según necesitemos.

Ej:

Un software de control de cambios como git.

Cada vez que guardemos los cambios clonaría el estado actual de nuestro trabajo, para que en cualquier momento podamos volver a una versión anterior en caso de necesitarlo.

PATRONES ESTRUCTURALES:

**--Adapter**

El patrón adaptador, se usa para hacer que las clases que implementan una interfaz y las que no lo hacen, puedan trabajar con coherencia.

Usando una ClaseAdaptador que trabajará como intermediaria entre las clases que no implementan la misma interfaz y la función que las instancie. De manera que se pueda trabajar con esa clase que no implementa la interfaz de la misma manera que si la implementara.

Se usa habitualmente para adaptar código antiguo y trabajar con él como si fuera nuevo.

Ej:

Si tuviéramos un software de puntuación y logros de jugadores, similar al de Xbox.

Imaginemos que ese software se ha creado para contabilizar todos los puntos de los diferentes juegos de la actualidad y enumerar los logros de los jugadores.

Aparece un nuevo juego que usa un sistema de puntos y logros diferente al Standard actual.

Mediante un patrón adapter que haga de intermediario entre el juego y nuestro software (adaptará los puntos del nuevo juego a la equivalencia en nuestro software, adaptándolo al Standad adoptado por los demás juegos). Podríamos hacer que nuestro software, pueda trabajar con el sistema de puntuaciones y logros del nuevo juego de la misma forma que lo hace con los demás juegos.

**--Decorator**

El patrón decorador, se usa para modificar una clase padre envolviéndola con otras clases que tendrán diferentes atributos y/o métodos.

Usando un decorador u otro, nuestra clase padre obtendrá diferentes características

Y funcionalidades. Todos los decoradores deberán heredar de una claseDecorador que implementará la interfaz del Objeto.

EJ:

Tenemos una clase Lavadora que tiene como atributos rpm, Cert. Energético, Marca, Modelo, carcasa, …

El decorador (Eco): convierte el Cert. Energético de cualquier tipo a (A+++) y le añade modo ahorro.

El decorador (Metal): crea un atributo que modifica la carcasa y la convierte en carcasa de acero.

El decorador (silence): crea un programa de lavado especial que minimiza el ruido, bajando las rpm del lavado.

**--Farcade**

El patrón fachada, se usa para simplificar lo máximo posible el código que tendremos en nuestra clase Main, escondiendo la complejidad de nuestro software, mediante una clase ClaseFachada. Esta se encargará de ejecutar todo el código que hay detrás según sea necesario, sin necesidad de que el Main conozca nada de éste código.

En el Main, simplemente crearemos un objeto ClaseFachada y lo usaremos tantas veces como sea necesario para que la fachada se encargue de realizar todo el trabajo por detrás.

Ej:

Imaginemos que nuestro código representa una sala de fiestas, tenemos clases que se encargan de la música, las luces, la alarma del local, enviar una notificación al servicio de limpieza, controlar el stock de bebidas, máquina de hielo,…

Una fachada se encargaría de lidiar con todas esas clases según necesitemos.

En el Main, podríamos decir ClaseFachada ->abrimos al público:

La clase fachada, encendería las luces usando los métodos de luces, apagaría la alarma (usando su clase correspondiente), encender música (usando su clase correspondiente), notificar a limpieza que abandonen la sala,…

Así también al final de la noche, el main usaría un método de la clase fachada para apagar todo y activar la limpieza y a los reponedores de stock

**Crea una pequeña aplicación implementando el patrón Singlenton y demuestra que, efectivamente, no crea instancias nuevas. Puedes tomar la sesión 7 como referencia.**