### **1. ¿Qué es un patrón de diseño?**

Un **patrón de diseño** es una solución general y reutilizable a un problema común que surge en un contexto particular durante el desarrollo de software. Los patrones de diseño proporcionan una forma estándar de abordar problemas de diseño, ayudando a los desarrolladores a crear sistemas más mantenibles y escalables.

### **2. ¿Cuál es su clasificación?**

Los patrones de diseño se clasifican en tres categorías principales:

**Patrones creacionales**: Se enfocan en la creación de objetos, asegurando que se utilicen correctamente. Ejemplos: Singleton, Factory, Abstract Factory.

**Patrones estructurales**: Describen cómo ensamblar objetos y clases para formar estructuras más grandes y flexibles. Ejemplos: Adapter, Bridge, Decorator.

**Patrones de comportamiento**: Se centran en la comunicación y responsabilidad entre objetos. Ejemplos: Observer, Strategy, Command.

### **3. ¿Cuáles son sus ventajas?**

**Reutilización**: Los patrones de diseño ofrecen soluciones probadas que pueden ser reutilizadas, reduciendo el tiempo de desarrollo.

**Mantenibilidad**: Facilitan la creación de código más organizado y fácil de mantener.

**Flexibilidad**: Ayudan a hacer sistemas más flexibles y adaptables a cambios futuros.

**Comunicación clara**: Proporcionan un lenguaje común entre los desarrolladores para discutir soluciones.

### **4. ¿Quién inventó los patrones y cuándo?**

El concepto de patrones de diseño fue popularizado por el libro **"Design Patterns: Elements of Reusable Object-Oriented Software"** publicado en **1994** por **Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides**, conocidos como **Gang of Four (GoF)**. Aunque no inventaron los patrones de diseño, su trabajo fue fundamental para su reconocimiento y uso formal en el desarrollo de software.

### **5. ¿Se puede utilizar los patrones fuera del desarrollo de software? ¿Cómo se hace?**

Sí, los patrones de diseño no son exclusivos del software. Se pueden aplicar a otros campos como arquitectura, diseño gráfico y gestión de proyectos. Un ejemplo fuera del desarrollo de software sería el patrón **Singleton**, que puede ser utilizado para asegurar que solo una instancia de una entidad exista en un sistema de negocio, como un CEO en una organización. Los principios detrás de los patrones pueden ser adaptados a cualquier contexto donde haya problemas recurrentes que requieran soluciones estructuradas y optimizadas.

### **Patrones Creacional**

Los patrones creacionales son un tipo de patrones de diseño que se enfocan en el proceso de creación de objetos. Su objetivo principal es abstraer el proceso de instanciación de objetos, proporcionando una interfaz para crear objetos en lugar de utilizar los constructores directos. Esto ayuda a que el código sea más flexible y menos dependiente de las clases concretas que implementan los objetos, promoviendo así una mayor reutilización y mantenimiento del código.

### **Clasificación de los Patrones Creacionales**

Los patrones creacionales se pueden clasificar en cinco tipos principales:

1. **Singleton:**

**Propósito:** Garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.

**Aplicación:** Utilizado cuando se necesita una única instancia de una clase para coordinar acciones a través del sistema.

**Ejemplo:** Gestores de configuración, conexiones a bases de datos

1. **Factory Method:**

**Propósito:** Define una interfaz para crear un objeto, pero deja a las subclases la decisión de qué clase instanciar. Permite a una clase delegar la instanciación a sus subclases.

**Aplicación:** Utilizado cuando una clase no puede anticipar qué clase de objetos debe crear o cuando una clase quiere delegar la creación a sus subclases.

**Ejemplo:** Crear diferentes tipos de documentos o formas según el tipo de usuario.

1. **Abstract Factory:**

**Propósito:** Proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas. Permite la creación de varios tipos de objetos relacionados sin acoplar el código cliente a clases específicas.

**Aplicación:** Utilizado cuando el sistema debe ser independiente de cómo se crean, ensamblan y representan sus productos. Ideal para sistemas que necesitan trabajar con diferentes familias de productos.

**Ejemplo:** Crear interfaces de usuario para diferentes sistemas operativos (Windows, Mac).

1. **Builder:**

**Propósito:** Separa la construcción de un objeto complejo de su representación, permitiendo construir diferentes representaciones del objeto utilizando el mismo proceso de construcción.

**Aplicación:** Utilizado cuando un objeto debe ser construido paso a paso y cuando el objeto puede tener múltiples representaciones posibles.

**Ejemplo:** Construir un coche con diferentes características y opciones.

1. **Prototype**

**Propósito:** Permite la creación de nuevos objetos copiando un prototipo existente en lugar de instanciar objetos directamente. Facilita la clonación de objetos en lugar de crear nuevos desde cero.

**Aplicación:** Utilizado cuando la creación de un objeto es costosa y se puede utilizar un objeto existente como prototipo para crear nuevos objetos.

**Ejemplo:** Clonar objetos gráficos o copiar configuraciones de un sistema para nuevos entornos.

### **Resumen:**

**Patrones Creacionales:** Se enfocan en la creación de objetos y ayudan a gestionar la instanciación de manera más flexible y desacoplada.

**Clasificación:**

**Singleton:** Una única instancia global.

**Factory Method:** Interfaz para crear objetos, delegando la instanciación a subclases.

**Abstract Factory:** Interfaz para crear familias de objetos relacionados.

**Builder:** Construcción de objetos complejos paso a paso.

**Prototype:** Clonación de objetos para crear nuevos.

### **1. ¿Qué es un patrón de Método de Fábrica (Factory Method)?**

#### **Propósito:**

El patrón **Factory Method** proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que se crean. La idea es delegar la responsabilidad de la creación de instancias a las subclases para permitir una mayor flexibilidad y control sobre el proceso de creación.

#### **Problema:**

En aplicaciones donde se requiere crear diferentes tipos de objetos que comparten una misma interfaz o clase base, la instanciación directa puede hacer que el código sea rígido y difícil de mantener. Además, si el tipo de objeto que debe crearse cambia frecuentemente, modificar el código en todas las partes puede ser tedioso.

#### **Solución:**

El **Factory Method** define un método en una clase base que no especifica qué tipo de objeto se va a crear. Las subclases proporcionan la implementación concreta de este método, creando las instancias correspondientes. Esto permite desacoplar el código que necesita objetos del código que crea objetos.

#### **Estructura:**

**Product (Producto)**: Define la interfaz de los objetos que la fábrica va a crear.

**ConcreteProduct (Producto Concreto)**: Implementa la interfaz del producto.

**Creator (Creador)**: Declara el método de fábrica que devuelve un objeto de tipo Product. Puede ser abstracto o tener una implementación por defecto.

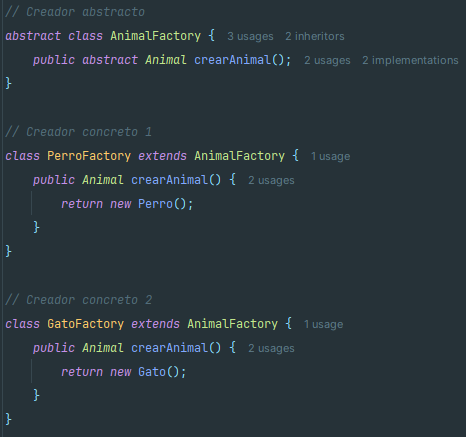
**ConcreteCreator (Creador Concreto)**: Sobrescribe el método de fábrica para devolver una instancia de ConcreteProduct.

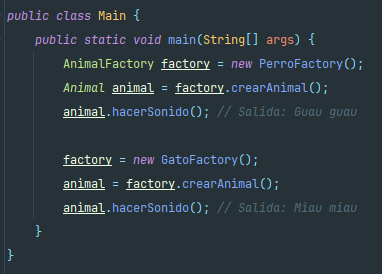
#### **Aplicabilidad:**

Cuando una clase no puede anticipar qué clase de objetos debe crear.

Para centralizar la lógica de creación de objetos y permitir la creación de familias de objetos relacionadas.

Cuando se quiere que las subclases decidan qué tipo de objeto crear.





### **2. Patrón de Fábrica Abstracto (Abstract Factory Pattern)**

#### **Propósito:**

El patrón de Fábrica Abstracta es un patrón de diseño creacional que proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Su propósito es abstraer la creación de objetos para que el código que los utiliza no dependa de las clases concretas, facilitando así la flexibilidad y la escalabilidad.

#### **Problema:**

En sistemas que requieren la creación de múltiples familias de objetos relacionados, puede ser complejo manejar la creación de estos objetos, especialmente cuando se deben garantizar que los objetos de una misma familia funcionen correctamente juntos. Si cada objeto debe ser creado por separado y se deben manejar sus interdependencias, el código puede volverse complicado y propenso a errores.

#### **Solución:**

El patrón de Fábrica Abstracta ofrece una solución al problema creando una interfaz abstracta para cada tipo de objeto, permitiendo a las subclases concretas implementar la creación de estos objetos. De esta forma, el código cliente no necesita conocer las clases concretas y puede operar de manera más flexible y extensible.

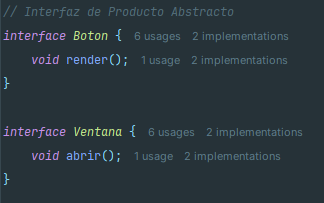
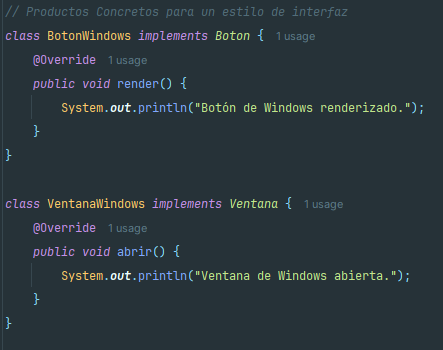
#### **Estructura:**

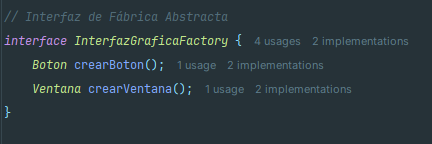
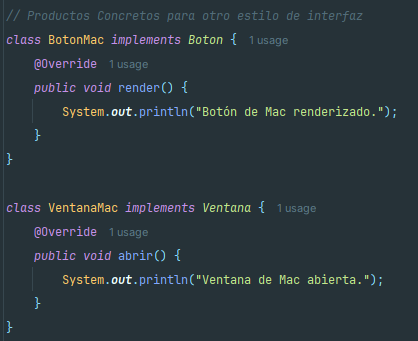
1. **Interfaz de Fábrica Abstracta (AbstractFactory):** Declara métodos para crear cada tipo de objeto.
2. **Fábricas Concretas (ConcreteFactory):** Implementan los métodos para crear instancias específicas de los objetos.
3. **Interfaz de Producto Abstracto (AbstractProduct):** Declara las interfaces que los productos deben implementar.
4. **Productos Concretos (ConcreteProduct):** Implementan las interfaces de productos específicos.
5. **Cliente:** Usa la interfaz de Fábrica Abstracta para obtener instancias de productos sin conocer las clases concretas.

#### **Aplicabilidad:**

Cuando se necesita trabajar con varias familias de productos que están diseñados para ser utilizados juntos.

Cuando el sistema debe ser independiente de cómo se crean, ensamblan y representan sus productos.

Cuando se desea proporcionar una interfaz para crear una serie de productos relacionados o dependientes sin acoplar el código cliente a las clases concretas.



### **3. Patrón Builder**

#### **Propósito:**

El patrón Builder es un patrón de diseño creacional que permite construir un objeto complejo paso a paso. Su propósito es separar la construcción de un objeto de su representación, de modo que el mismo proceso de construcción puede crear diferentes representaciones del objeto.

#### **Problema:**

Cuando un objeto tiene una construcción compleja con múltiples pasos y varias configuraciones posibles, el código para crear este objeto puede volverse desordenado y difícil de manejar si se realiza en un solo constructor o método. Esto puede llevar a constructores con muchos parámetros, problemas de mantenimiento y código inflexible.

#### **Solución:**

El patrón Builder proporciona una solución al problema separando el proceso de construcción en diferentes fases. Utiliza un "constructor" que lleva a cabo el proceso de construcción paso a paso, y un "director" que dirige el proceso de construcción para crear un objeto específico. Esto permite construir objetos complejos de manera más flexible y manejable.

#### **Estructura:**

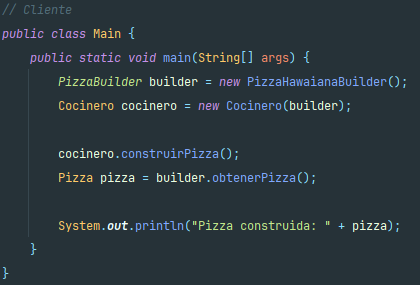
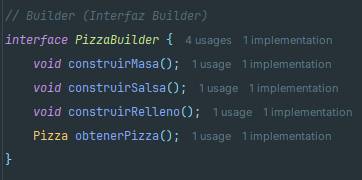
1. **Builder (Interfaz Builder):** Define los pasos necesarios para construir una parte del objeto.
2. **ConcreteBuilder (Constructor Concreto):** Implementa los pasos de la interfaz Builder y construye el objeto final.
3. **Director:** Utiliza un objeto Builder para construir un producto usando los pasos definidos.
4. **Product (Producto):** Representa el objeto complejo que está siendo construido.

#### **Aplicabilidad:**

Cuando un objeto necesita ser construido de manera compleja a partir de varias partes.

Cuando se deben construir diferentes representaciones del mismo objeto.

Cuando se quiere evitar constructores con muchos parámetros.



### **4. Patrón Singleton (Único)**

#### **Propósito**

El patrón Singleton garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia. Es útil cuando se necesita una única instancia para coordinar acciones en todo el sistema.

#### **Problema**

Cuando se necesita asegurar que una clase tenga solo una instancia y proporcionar un acceso global a esa instancia, puede ser complicado gestionar la creación y el acceso a la instancia única de manera eficiente.

#### **Solución**

El patrón Singleton asegura que solo se cree una instancia de una clase utilizando una instancia privada y estática. Proporciona un método público para acceder a la instancia única. El acceso concurrente se gestiona para evitar la creación de múltiples instancias.

#### **Estructura**

1. **Singleton:** Contiene una instancia privada y estática de la clase. Proporciona un método público para acceder a la instancia única y un constructor privado para evitar la creación de instancias adicionales.

#### **Aplicabilidad**

Cuando se necesita garantizar que solo haya una instancia de una clase en todo el sistema.

Para proporcionar un punto de acceso global a una instancia única.

En situaciones donde un objeto debe coordinar acciones en todo el sistema (por ejemplo, en la configuración de una aplicación).

### 

### **Patrón Estructural**

Los patrones estructurales son un tipo de patrones de diseño que se enfocan en la composición de clases y objetos para formar estructuras más grandes. Su objetivo es asegurar que si una parte del sistema cambia, el sistema completo no se vea afectado. Estos patrones ayudan a crear relaciones entre objetos y clases, facilitando la composición de nuevas estructuras sin modificar el código existente.

### **Clasificación de los Patrones Estructurales**

1. **Adapter (Adaptador):**

**Propósito:** Permite que dos interfaces incompatibles trabajen juntas. Convierte la interfaz de una clase en otra interfaz que el cliente espera.

**Aplicación:** Utilizado cuando se necesita integrar sistemas que tienen interfaces diferentes, sin cambiar su código.

**Ejemplo:** Adaptar una interfaz de un sistema de archivos para que funcione con un nuevo sistema de almacenamiento.

1. **Bridge (Puente):**

**Propósito:** Desacopla una abstracción de su implementación para que ambas puedan variar independientemente. Proporciona una manera de cambiar la implementación sin alterar el código cliente.

**Aplicación:** Utilizado cuando se quieren cambiar las implementaciones y abstracciones de manera independiente y se requiere la separación de la interfaz y la implementación.

**Ejemplo:** Implementar diferentes tipos de dibujos gráficos y su representación en diferentes formatos.

1. **Composite (Composición):**

**Propósito:** Permite que los objetos se compongan en estructuras de árbol para representar jerarquías parte-todo. Permite que los clientes traten objetos individuales y composiciones de objetos de manera uniforme.

**Aplicación:** Utilizado cuando se quiere representar jerarquías de objetos y se necesita que los clientes puedan tratar tanto objetos individuales como composiciones de objetos de manera homogénea.

**Ejemplo:** Representar estructuras de archivos y directorios.

1. **Decorator (Decorador):**

**Propósito:** Permite agregar funcionalidad adicional a un objeto de manera dinámica. Proporciona una alternativa flexible a la subclasificación para extender funcionalidades.

**Aplicación:** Utilizado cuando se quiere agregar responsabilidades adicionales a los objetos sin modificar su código original.

**Ejemplo:** Añadir características adicionales a una ventana, como bordes o barras de desplazamiento.

1. **Facade (Fachada):**

**Propósito:** Proporciona una interfaz unificada a un conjunto de interfaces en un subsistema. Facilita el uso del subsistema al simplificar su interfaz.

**Aplicación:** Utilizado cuando se quiere proporcionar una interfaz más sencilla a un subsistema complejo.

**Ejemplo:** Simplificar el acceso a un conjunto complejo de APIs en un sistema de gestión de bases de datos.

1. **Flyweight (Peso Ligero):**

**Propósito:** Permite el uso compartido de objetos para ahorrar memoria. Utiliza objetos de manera compartida para reducir el costo de creación y almacenamiento de objetos similares.

**Aplicación:** Utilizado cuando se necesita crear un gran número de objetos similares y se quiere optimizar el uso de memoria.

**Ejemplo:** Representar caracteres en una interfaz de usuario donde cada carácter puede ser un objeto compartido.

1. **Proxy (Representante):**

**Propósito:** Proporciona un objeto sustituto que controla el acceso a otro objeto. Permite realizar operaciones adicionales antes o después de delegar la solicitud al objeto real.

**Aplicación:** Utilizado cuando se quiere controlar el acceso a un objeto, realizar operaciones adicionales o implementar el patrón de acceso diferido.

**Ejemplo:** Controlar el acceso a un objeto remoto o proteger el acceso a un recurso costoso en términos de tiempo o memoria.

### **Resumen:**

* **Patrones Estructurales:** Se centran en cómo las clases y objetos se componen para formar estructuras más grandes y complejas, asegurando que los cambios en las partes del sistema no afecten al sistema en su totalidad.
* **Clasificación:**

**Adapter:** Adaptar interfaces incompatibles.

**Bridge:** Separar abstracción de implementación.

**Composite:** Representar jerarquías parte-todo.

**Decorator:** Agregar funcionalidad adicional dinámicamente.

**Facade:** Proporcionar una interfaz simplificada a un subsistema.

**Flyweight:** Compartir objetos para reducir el uso de memoria

**Proxy:** Controlar el acceso a un objeto y realizar operaciones adicionales.

### **1. Patrón Adapter (Adaptador)**

#### **Propósito**

El patrón Adapter permite que dos interfaces incompatibles trabajen juntas, haciendo que una clase con una interfaz incompatible pueda ser utilizada como si tuviera una interfaz compatible con el sistema.

#### **Problema**

Cuando se tiene una clase con una interfaz incompatible con el sistema actual, pero se necesita utilizar esa clase, el problema es integrar esta clase en el sistema sin modificar su interfaz.

#### **Solución**

El patrón Adapter proporciona una solución mediante la creación de una clase adaptadora que convierte la interfaz de la clase incompatible (Adaptee) en una interfaz que el sistema espera (Target).

#### **Estructura**

1. **Target:** Interfaz que el cliente espera.
2. **Adapter:** Implementa la interfaz del Target y traduce las llamadas al Adaptee.
3. **Adaptee:** Clase con una interfaz incompatible.
4. **Client:** Utiliza la interfaz del Target.

#### **Aplicabilidad**

* Integrar clases con interfaces incompatibles.
* Reutilizar clases existentes con interfaces diferentes.
* Adaptar librerías de terceros a la interfaz de la aplicación.

### **2. Patrón Bridge (Puente)**

#### **Propósito**

El patrón Bridge se utiliza para desacoplar una abstracción de su implementación, permitiendo que ambas evolucionen independientemente. Esto es útil para evitar la explosión combinatoria de clases cuando se tienen múltiples dimensiones de variabilidad.

#### **Problema**

Cuando se tienen múltiples dimensiones de variabilidad (como diferentes tipos y plataformas), se puede crear una explosión combinatoria de clases si se combinan todas las posibilidades. Esto puede llevar a un diseño rígido y difícil de mantener.

#### **Solución**

El patrón Bridge resuelve este problema separando la abstracción de su implementación mediante un puente (Bridge) que delega las operaciones a una implementación concreta. Esto permite cambiar la implementación sin afectar a la abstracción y viceversa.

#### **Estructura**

1. **Abstraction:** Define la interfaz abstracta y mantiene una referencia al objeto Implementor.
2. **RefinedAbstraction:** Extiende la interfaz abstracta.
3. **Implementor:** Define la interfaz de implementación.
4. **ConcreteImplementor:** Implementa la interfaz del Implementor.

#### **Aplicabilidad**

* Cuando se necesita separar una abstracción de su implementación para evitar la explosión combinatoria de clases.
* Cuando se quiere permitir la evolución independiente de abstracciones e implementaciones.

### **3. Patrón Composite (Compuesto)**

#### **Propósito**

El patrón Composite permite tratar objetos individuales y composiciones de objetos de manera uniforme. Es útil para representar jerarquías de objetos, donde los objetos individuales y las composiciones deben ser tratados de manera similar.

#### **Problema**

Cuando se necesita trabajar con estructuras de árbol donde los objetos individuales y las composiciones deben ser tratados de la misma manera, el problema es la complejidad de manejar tanto los objetos individuales como las composiciones.

#### **Solución**

El patrón Composite proporciona una solución al permitir que los objetos individuales y las composiciones de objetos (compuestos) implementen la misma interfaz. Esto simplifica el manejo de jerarquías complejas.

#### **Estructura**

1. **Component:** Define la interfaz común para los objetos individuales y compuestos.
2. **Leaf:** Implementa la interfaz Component para objetos individuales.
3. **Composite:** Implementa la interfaz Component y mantiene una colección de objetos Component.

#### **Aplicabilidad**

* Cuando se necesita tratar objetos individuales y composiciones de objetos de manera uniforme.
* Para representar jerarquías de objetos como árboles.

### **4. Patrón Decorator (Decorador)**

#### **Propósito**

El patrón Decorator permite agregar funcionalidades adicionales a un objeto de manera dinámica sin modificar su estructura. Es útil para extender las capacidades de un objeto de manera flexible.

#### **Problema**

Cuando se necesita agregar funcionalidades a un objeto sin modificar su estructura, el patrón Decorator proporciona una solución flexible. La alternativa, como la herencia, puede llevar a una explosión de clases y a un diseño menos flexible.

#### **Solución**

El patrón Decorator permite crear una serie de clases decoradoras que extienden la funcionalidad de un objeto. Cada decorador envuelve el objeto original y agrega nuevas funcionalidades sin modificar el objeto original.

#### **Estructura**

1. **Component:** Define la interfaz del objeto a decorar.
2. **ConcreteComponent:** Implementa la interfaz del Component.
3. **Decorator:** Mantiene una referencia al objeto Component y define la interfaz que puede ser extendida.
4. **ConcreteDecorator:** Extiende la funcionalidad del objeto Component.

#### **Aplicabilidad**

* Cuando se necesita agregar funcionalidades a un objeto de manera flexible y dinámica.
* Para evitar la explosión de clases que puede ocurrir con la herencia.

### **5. Patrón Facade (Fachada)**

#### **Propósito**

El patrón Facade proporciona una interfaz unificada que simplifica el acceso a un conjunto de interfaces en un subsistema. Es útil para ocultar la complejidad del sistema y proporcionar una interfaz más sencilla al cliente.

#### **Problema**

Cuando un subsistema es complejo y tiene múltiples interfaces, puede ser difícil para el cliente interactuar con él. El patrón Facade resuelve este problema proporcionando una interfaz simplificada.

#### **Solución**

El patrón Facade crea una clase fachada que ofrece métodos simplificados para interactuar con el subsistema. La fachada delega las llamadas a las clases del subsistema, simplificando así la interacción del cliente.

#### **Estructura**

1. **Facade:** Proporciona una interfaz simplificada para el cliente.
2. **Subsystem Classes:** Clases que realizan la funcionalidad real del subsistema.

#### **Aplicabilidad**

* Cuando un sistema es complejo y se desea simplificar su uso.
* Para proporcionar una interfaz más sencilla al cliente.

### **6. Patrón Flyweight (Peso Ligero)**

#### **Propósito**

El patrón Flyweight se utiliza para minimizar el uso de memoria al compartir objetos similares. Es útil cuando se tienen muchas instancias de objetos que comparten datos comunes.

#### **Problema**

Cuando se tienen muchas instancias de objetos con datos comunes, el uso de memoria puede ser elevado. El patrón Flyweight ayuda a reducir el uso de memoria mediante el compartir instancias.

#### **Solución**

El patrón Flyweight crea un conjunto de objetos compartidos (Flyweights) y permite que las instancias se compartan entre clientes. Solo los datos únicos se almacenan en cada instancia, mientras que los datos comunes se comparten.

#### **Estructura**

1. **Flyweight:** Define la interfaz para los objetos compartidos.
2. **ConcreteFlyweight:** Implementa la interfaz Flyweight y almacena el estado compartido.
3. **FlyweightFactory:** Crea y gestiona los objetos Flyweight compartidos.

#### **Aplicabilidad**

* Cuando se tienen muchas instancias de objetos que comparten datos comunes.
* Para reducir el uso de memoria en sistemas con gran número de objetos.

### **7. Patrón Proxy (Proxie)**

#### **Propósito**

El patrón Proxy proporciona un sustituto o representante de un objeto real para controlar el acceso a él. Es útil para implementar mecanismos de control de acceso o para retrasar la creación de un objeto costoso hasta que sea necesario.

#### **Problema**

Cuando se necesita controlar el acceso a un objeto real, o retrasar su creación hasta que sea necesario, el patrón Proxy ofrece una solución para gestionar el acceso y la creación.

#### **Solución**

El patrón Proxy crea una clase que actúa como intermediario entre el cliente y el objeto real. El Proxy puede implementar controles de acceso, almacenamiento en caché o realizar otras funciones antes de delegar la llamada al objeto real.

#### **Estructura**

1. **Subject:** Define la interfaz que el Proxy y el RealSubject deben implementar.
2. **Proxy:** Implementa la interfaz Subject y controla el acceso al RealSubject.
3. **RealSubject:** Implementa la interfaz Subject y realiza la funcionalidad real.

#### **Aplicabilidad**

* Para controlar el acceso a un objeto real.
* Para implementar almacenamiento en caché.
* Para retrasar la creación de objetos costosos.

### **Patrones Comportamentales**

#### **Qué es un Patrón Comportamental**

Los patrones comportamentales se centran en los algoritmos y la asignación de responsabilidades entre los objetos. Se enfocan en cómo los objetos interactúan y colaboran para lograr un comportamiento específico. Estos patrones ayudan a definir cómo los objetos se comunican, se coordinan y se colaboran entre sí, facilitando el diseño de sistemas que son flexibles y fáciles de mantener.

### **Clasificación de los Patrones Comportamentales**

1. **Chain of Responsibility (Cadena de Responsabilidad):**
   * Permite que un objeto pase una solicitud a través de una cadena de objetos receptores. Cada objeto puede manejar la solicitud o pasarla al siguiente en la cadena.
2. **Command (Comando):**
   * Encapsula una solicitud como un objeto, lo que permite parametrizar clientes con diferentes solicitudes, encolar o deshacer operaciones.
3. **Interpreter (Intérprete):**
   * Define una gramática para un lenguaje y proporciona un intérprete para interpretar las oraciones del lenguaje. Es útil para implementar lenguajes de scripting o interpretaciones de expresiones.
4. **Iterator (Iterador):**
   * Proporciona un método para acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación subyacente.
5. **Mediator (Mediador):**
   * Define un objeto que encapsula cómo interactúan un conjunto de objetos. Permite que los objetos se comuniquen a través del mediador en lugar de referirse directamente unos a otros.
6. **Memento (Recuperador):**
   * Permite capturar y externalizar el estado interno de un objeto sin violar la encapsulación, para que el objeto pueda ser restaurado a este estado más tarde.
7. **Observer (Observador):**
   * Define una dependencia de uno a muchos entre objetos, de tal manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.
8. **State (Estado):**
   * Permite que un objeto altere su comportamiento cuando su estado interno cambia. El objeto parecerá cambiar su clase.
9. **Strategy (Estrategia):**
   * Define una familia de algoritmos, encapsula cada uno, y hace que sean intercambiables. Permite al algoritmo variar independientemente de los clientes que lo utilizan.
10. **Template Method (Método Plantilla):**
    * Define el esqueleto de un algoritmo en una operación, deferiendo algunos pasos a las subclases. Permite a las subclases redefinir ciertos pasos del algoritmo sin cambiar su estructura.
11. **Visitor (Visitante):**
    * Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera. Permite agregar funcionalidades a objetos sin modificar sus clases.

### **1. Patrón Chain of Responsibility**

**Propósito:** Permitir que múltiples objetos manejen una solicitud sin que el remitente de la solicitud tenga que saber cuál objeto la manejará. Los objetos se encadenan y la solicitud se pasa a lo largo de la cadena hasta que un objeto la maneje.

**Problema:** Cuando hay múltiples objetos que pueden manejar una solicitud, pero no se sabe cuál manejará la solicitud. Se necesita un mecanismo para evitar acoplar el emisor de la solicitud con el receptor.

**Solución:** Crear una cadena de objetos receptores donde cada objeto tiene la oportunidad de manejar la solicitud o pasarla al siguiente en la cadena.

**Estructura:**

1. **Handler (Manejador):** Define una interfaz para manejar la solicitud y opcionalmente mantiene una referencia al siguiente manejador en la cadena.
2. **ConcreteHandler (Manejador Concreto):** Implementa la lógica para manejar la solicitud y puede pasarla al siguiente manejador si no puede procesarla.
3. **Client (Cliente):** Envía la solicitud a la cadena de manejadores.

**Aplicabilidad:**

* Cuando se necesita una cadena de objetos para manejar una solicitud.
* Cuando el objeto que maneja la solicitud no necesita saber qué objeto está manejando la solicitud finalmente.

### **2. Patrón Command**

**Propósito:** Encapsular una solicitud como un objeto, permitiendo parametrizar clientes con diferentes solicitudes, encolar o deshacer operaciones.

**Problema:** Necesidad de separar la solicitud de la acción que la ejecuta. Esto permite que las solicitudes se traten como objetos, facilitando la parametrización, el deshacer y el registro de operaciones.

**Solución:** Crear una interfaz de comando que declare un método execute. Cada comando concreto implementa esta interfaz y encapsula una solicitud específica.

**Estructura:**

1. **Command (Comando):** Declara un método para ejecutar la operación.
2. **ConcreteCommand (Comando Concreto):** Implementa el método execute y define la acción que se debe realizar.
3. **Client (Cliente):** Crea un objeto de comando y configura el receptor.
4. **Invoker (Invocador):** Almacena y ejecuta los comandos.
5. **Receiver (Receptor):** Conoce cómo realizar las operaciones asociadas a la solicitud.

**Aplicabilidad:**

* Cuando se necesita parametrizar objetos con operaciones.
* Para implementar deshacer/rehacer operaciones.

### **3. Patrón Iterator**

**Propósito:** Proporcionar un método para acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación subyacente.

**Problema:** Necesidad de recorrer colecciones de objetos sin exponer la estructura interna de la colección.

**Solución:** Proveer una interfaz para iterar sobre los elementos de una colección sin conocer su estructura interna.

**Estructura:**

1. **Iterator (Iterador):** Define métodos para acceder a los elementos de la colección.
2. **ConcreteIterator (Iterador Concreto):** Implementa la interfaz Iterator y mantiene el estado de la iteración.
3. **Aggregate (Agregado):** Define una interfaz para crear un iterador.
4. **ConcreteAggregate (Agregado Concreto):** Implementa la interfaz Aggregate y proporciona una colección concreta para iterar.

**Aplicabilidad:**

* Cuando se desea recorrer una colección de objetos.
* Cuando la estructura de datos debe permanecer oculta.

### **4. Patrón Mediator**

**Propósito:** Definir un objeto que encapsula cómo interactúan un conjunto de objetos, promoviendo el acoplamiento débil al evitar que los objetos se refieran explícitamente entre sí.

**Problema:** Cuando se tiene un conjunto de objetos que deben interactuar y la complejidad de sus interacciones hace que sea difícil gestionar la comunicación.

**Solución:** Introducir un mediador que coordine la comunicación entre objetos y maneje las interacciones entre ellos.

**Estructura:**

1. **Mediator (Mediador):** Define una interfaz para comunicarse con los objetos.
2. **ConcreteMediator (Mediador Concreto):** Implementa la interfaz y coordina la comunicación entre los objetos.
3. **Colleague (Colega):** Objetos que interactúan a través del mediador.
4. **ConcreteColleague (Colega Concreto):** Implementa la comunicación a través del mediador.

**Aplicabilidad:**

* Cuando se necesita gestionar la comunicación entre múltiples objetos.
* Para reducir el acoplamiento entre objetos en sistemas complejos.

### **5. Patrón Memento**

**Propósito:** Capturar y externalizar el estado interno de un objeto sin violar la encapsulación, para que el objeto pueda ser restaurado a este estado más tarde.

**Problema:** Necesidad de restaurar un objeto a un estado anterior sin exponer su estructura interna.

**Solución:** Utilizar un objeto Memento para almacenar el estado y un objeto Originator para recuperar el estado desde el memento.

**Estructura:**

1. **Memento (Recuperador):** Almacena el estado interno del objeto.
2. **Originator (Originador):** Crea un memento para capturar su estado y utiliza el memento para restaurar el estado.
3. **Caretaker (Cuidador):** Guarda el memento y no interactúa con el estado interno.

**Aplicabilidad:**

* Para implementar funcionalidades de deshacer/rehacer.
* Cuando se necesita preservar el estado del objeto.

### **6. Patrón Observer**

**Propósito:** Definir una dependencia uno a muchos entre objetos de tal manera que cuando un objeto cambie de estado, todos sus dependientes sean notificados y actualizados automáticamente.

**Problema:** Cuando un objeto necesita notificar a otros objetos sobre cambios en su estado sin acoplar directamente al observador.

**Solución:** Utilizar un patrón de suscripción en el que los observadores se suscriben al sujeto y son notificados de los cambios.

**Estructura:**

1. **Subject (Sujeto):** Mantiene una lista de observadores y notifica a todos cuando su estado cambia.
2. **Observer (Observador):** Define una interfaz para recibir notificaciones.
3. **ConcreteSubject (Sujeto Concreto):** Implementa el estado y notifica a los observadores.
4. **ConcreteObserver (Observador Concreto):** Implementa el método de actualización en respuesta a los cambios en el sujeto.

**Aplicabilidad:**

* Cuando se necesita que varios objetos se sincronicen con el cambio en otro objeto.
* En interfaces gráficas para actualizar vistas en respuesta a cambios de datos.

### **7. Patrón State**

**Propósito:** Permitir que un objeto altere su comportamiento cuando su estado interno cambia. El objeto parecerá cambiar su clase.

**Problema:** Cuando un objeto cambia su comportamiento basado en su estado interno y el estado puede cambiar de manera significativa.

**Solución:** Definir una interfaz para el estado y usar objetos de estado concretos para implementar los comportamientos específicos.

**Estructura:**

1. **State (Estado):** Define una interfaz para el comportamiento asociado con un estado específico.
2. **ConcreteState (Estado Concreto):** Implementa el comportamiento específico del estado.
3. **Context (Contexto):** Mantiene una referencia al estado actual y permite cambiar de estado.

**Aplicabilidad:**

* Cuando el comportamiento de un objeto depende de su estado interno.
* Para gestionar estados y transiciones en máquinas de estado.

### **8. Patrón Strategy**

**Propósito:** Definir una familia de algoritmos, encapsular cada uno, y hacerlos intercambiables. Permite que el algoritmo varíe independientemente de los clientes que lo utilizan.

**Problema:** Cuando se necesitan diferentes algoritmos para una misma tarea y se quiere seleccionar el algoritmo adecuado en tiempo de ejecución.

**Solución:** Definir una interfaz para una estrategia y permitir que los clientes utilicen diferentes implementaciones de esta interfaz.

**Estructura:**

1. **Strategy (Estrategia):** Declara una interfaz para ejecutar el algoritmo.
2. **ConcreteStrategy (Estrategia Concreta):** Implementa el algoritmo específico.
3. **Context (Contexto):** Mantiene una referencia a una estrategia y permite cambiarla en tiempo de ejecución.

**Aplicabilidad:**

* Cuando se requiere seleccionar entre varias implementaciones de un algoritmo.
* Para encapsular algoritmos que pueden cambiar durante la ejecución.

### **9. Patrón Template Method**

**Propósito:** Definir el esqueleto de un algoritmo en una operación, deferiendo algunos pasos a las subclases. Permite a las subclases redefinir ciertos pasos del algoritmo sin cambiar su estructura.

**Problema:** Cuando se tiene un algoritmo con pasos fijos y algunos pasos pueden variar, pero la estructura del algoritmo debe mantenerse constante.

**Solución:** Definir el algoritmo en una clase base utilizando un método plantilla que llama a métodos abstractos o "hook" que pueden ser implementados por las subclases.

**Estructura:**

1. **AbstractClass (Clase Abstracta):** Define el método plantilla y algunos pasos del algoritmo como métodos abstractos o protegidos.
2. **ConcreteClass (Clase Concreta):** Implementa los métodos abstractos y proporciona comportamientos específicos.

**Aplicabilidad:**

* Cuando se tiene un algoritmo con una estructura fija y se desea permitir variaciones en ciertos pasos.
* Para evitar la duplicación de código en implementaciones similares.

### **10. Patrón Visitor**

**Propósito:** Permitir definir una nueva operación sin cambiar las clases de los elementos sobre los que opera. Permite agregar funcionalidades a los objetos sin modificar sus clases.

**Problema:** Cuando se necesita añadir nuevas operaciones a una estructura de objetos sin modificar las clases de los objetos.

**Solución:** Definir un visitante que implemente operaciones sobre los elementos de una estructura de objetos y permitir que los elementos acepten el visitante.

**Estructura:**

1. **Visitor (Visitante):** Declara un método de visita para cada tipo de elemento.
2. **ConcreteVisitor (Visitante Concreto):** Implementa las operaciones específicas.
3. **Element (Elemento):** Define un método para aceptar un visitante.
4. **ConcreteElement (Elemento Concreto):** Implementa el método para aceptar al visitante.
5. **ObjectStructure (Estructura de Objetos):** Mantiene una colección de elementos y permite la iteración sobre ellos.

**Aplicabilidad:**

* Cuando se necesitan agregar operaciones a una estructura de objetos sin cambiar sus clases.
* Para operaciones que necesitan ser aplicadas a una colección de objetos de diferentes tipos.

### **- Patrón MVC (Modelo – Vista - Controlador)**

**Propósito:** Separar la lógica de negocio, la presentación y el control en una aplicación para permitir un diseño más modular, facilitando la gestión, mantenimiento y escalabilidad del código.

**Problema:** Cuando se mezcla la lógica de negocio con la interfaz de usuario, lo que resulta en código acoplado y difícil de mantener. Cambios en la interfaz pueden requerir cambios en la lógica de negocio y viceversa.

**Solución:** Dividir la aplicación en tres componentes principales:

* **Modelo:** Representa los datos y la lógica de negocio. Maneja los datos y las operaciones sobre ellos.
* **Vista:** Se encarga de la presentación y la interfaz de usuario. Muestra los datos del modelo al usuario.
* **Controlador:** Actúa como intermediario entre el modelo y la vista. Procesa las entradas del usuario, actualiza el modelo y decide qué vista mostrar.

**Estructura:**

1. **Modelo (Model):** Contiene los datos y la lógica de negocio. Notifica a la vista sobre cambios en los datos.
2. **Vista (View):** Presenta los datos al usuario y recibe las entradas del usuario.
3. **Controlador (Controller):** Procesa las entradas del usuario, actualiza el modelo y selecciona la vista adecuada.

**Aplicabilidad:**

* En aplicaciones con interfaces de usuario complejas.
* Cuando se desea una separación clara entre la lógica de negocio y la interfaz de usuario.
* Para mejorar la mantenibilidad y escalabilidad del código.

### **- Patrón DAO (Data – Access - Object)**

**Propósito:** Proporcionar una abstracción para el acceso a datos y separar la lógica de negocio de la lógica de acceso a datos, facilitando así la gestión de datos y el mantenimiento de la aplicación.

**Problema:** La lógica de acceso a datos puede estar entremezclada con la lógica de negocio, lo que dificulta la gestión y el mantenimiento. También puede hacer que el código sea menos reutilizable y más difícil de probar.

**Solución:** Crear una interfaz DAO que defina los métodos para acceder y manipular los datos. Implementar esta interfaz con una clase concreta que maneje el acceso a la fuente de datos específica (como una base de datos).

### **Estructura:**

1. **DAO (Data Access Object):** Interfaz que define los métodos para acceder y manipular datos.
2. **ConcreteDAO (DAO Concreto):** Implementa la interfaz DAO y maneja el acceso a la fuente de datos.
3. **Model:** Representa los datos que se gestionan. Utiliza el DAO para realizar operaciones de acceso a datos.

**Aplicabilidad:**

* Cuando se necesita una capa de acceso a datos que permita cambios en la fuente de datos sin afectar a la lógica de negocio.
* Para desacoplar la lógica de acceso a datos de la lógica de negocio.
* Para facilitar la reutilización y prueba del código de acceso a datos.

**Resumen:**

* **MVC** separa una aplicación en tres componentes: Modelo, Vista y Controlador, mejorando la modularidad y mantenibilidad al separar la lógica de negocio, la presentación y el control.
* **DAO** proporciona una capa de abstracción para el acceso a datos, separando la lógica de negocio de la lógica de acceso a datos para mejorar la mantenibilidad y la flexibilidad.