**PATRONES DE DISEÑO**

**PATRONES CREACIONALES**

Estos patrones proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente.

* **FACTORY METHOD**

Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán

**¿CUÁNDO SE UTILIZA?**

1. No se sabe de antemano con que tipo de objetos se va a trabajar
2. Delegar la responsabilidad a las subclases de que objeto crear
3. Cuando se quiere ahorrar recursos del sistema (reutilizando objetos en vez de crear uno nuevo cada vez)

**VENTAJAS**

* Se evita un fuerte acoplamiento entre el creador y los productos concretos.
* Es fácilmente extensible ya que la arquitectura queda abierta a desarrollos horizontales con nuevas clases que extiendan a la factoría y a la familia de productos.
* Permite encapsular en las clases factorías toda la lógica de creación de objetos.

**DESVENTAJA**

Puede ser que el código se complique, ya que debes incorporar una multitud de nuevas subclases para implementar el patrón

**Diagrama

Descripción generada automáticamenteDIAGRAMA DE CLASES**

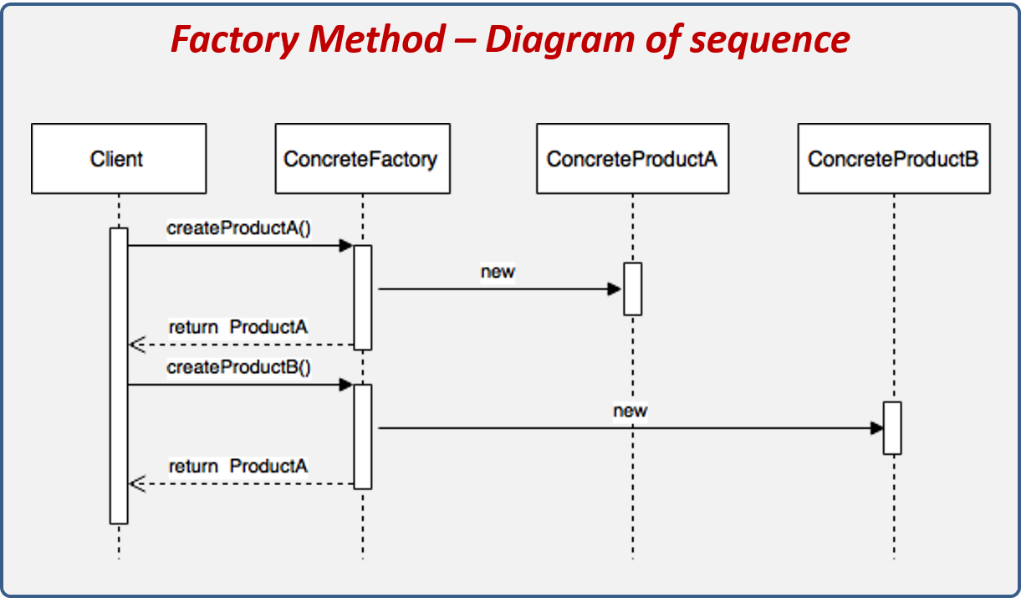
1. El **Product** declara la interfaz, que es común a todos los objetos que puede producir la clase creadora y sus subclases.

2. Los **Concrete Product** son distintas implementaciones de la interfaz de producto. Es el resultado final

3. La clase **Creator** declara el método fábrica que devuelve nuevos objetos del tipo producto. Es importante que el tipo de retorno de este método coincida con la interfaz de producto.

4. Los **Concrete Creator** redefinen el Factory Method base, de modo que devuelva un tipo diferente de producto.

**DIAGRAMA DE SECUENCIA**



1. El cliente se solicita al ConcreteFactory la creación del ProductA
2. El CF localiza la implementación concreta del ProductA y crea una nueva instancia
3. El CF regresa al ConcreteProductA creado
4. El cliente se solicita al ConcreteFactory la creación del ProductB
5. El CF localiza la implementación concreta del ProductB y crea una nueva instancia
6. El CF regresa al ConcreteProductB creado

**¿EN QUE CONSISTE?**

Consiste en definir una interfaz para crear objetos de tipo genérico permitiendo a las subclases decidir que tipo de objeto se va a crear

* **ABSTRACT FACTORY**

Es un patrón de diseño creacional el cual crea diferentes familias de objetos. Su objetivo principal es soportar múltiples estándares que vienen definidos por las diferentes jerarquías de herencia de objetos.

Es similar al FM, solo que este esta orientado a combinar productos

**¿CUÁNDO SE UTILIZA?**

1. Un sistema se debe configurar con una de entre varias familias de productos
2. Una familia de productos que están relacionados y fueron hechos para utilizarse juntos

**VENTAJAS**

* Se oculta a los clientes las clases de implementación: los clientes manipulan los objetos a través de las interfaces o clases abstractas
* Evitas un acoplamiento fuerte entre productos concretos y el código cliente.
* Mejora la consistencia entre productos: el uso de la factoría abstracta permite forzar a utilizar un conjunto de objetos de una misma familia.

**DESVENTAJAS**

* No siempre es fácil soportar nuevos tipos de productos si se tiene que extender la interfaz de la factoría abstracta

**Diagrama

Descripción generada automáticamenteDIAGRAMA DE CLASES**

**AbstractFactory:** declara una interfaz para la creación de objetos de productos abstractos.

**ConcreteFactory:** implementa las operaciones para la creación de objetos de productos concretos.

**AbstractProduct**: declara una interfaz para los objetos de un tipo de productos.

**ConcreteProduct**: define un objeto de producto que la correspondiente factoría concreta se encargaría de crear, a la vez que implementa la interfaz de producto abstracto.

**Client**: utiliza solamente las interfaces declaradas en la factoría y en los productos abstractos.

Una única instancia de cada FactoryConcreto es creada en tiempo de ejecución. AbstractFactory delega la creación de productos a sus subclases FactoryConcreto.

**Diagrama

Descripción generada automáticamenteDIAGRAMA DE SECUENCIA**

1. El cliente solicita la creación del ConcreteFactory1 al AbstractFactory.
2. El AbstractFactory crea una instancia del ConcreteFactory1 y la regresa.
3. El cliente le solicita al ConcreteFactory1 la creación de un ProductA.
4. El ConcreteFactory1 crea una instancia del ProductA1 el cual es parte de la familia1 y lo regresa.
5. El cliente esta vez solicita la creación del ConcreteFactory2 al AbstractFactory.
6. El AbstractFactory crea una instancia del ConcreteFactory2 .
7. El cliente le solicita al ConcreteFactory2 la creación de un ProductA .
8. El ConcreteFactory2 crea una instancia del ProductA2 el cual es parte de la familia2 y lo regresa.

**¿EN QUE CONSISTE?**

Consiste en producir familias de objetos relacionados sin especificar sus clases concretas

**DIFERENCIA ENTRE FACTORY METHOD Y ABSTRACT FACTORY**

|  |  |
| --- | --- |
| **FACTORY METHOD** | **ABSTRACT FACTORY** |
| Patron de creación  Crea un solo producto  Usado en diseños sencillos | Patron de creación  Crea una familia de productos  Usado en diseños más complejos |

**PATRONES ESTRUCTURALES**

Tratan la manera en que los objetos se conectan con otros objetos, para asegurar que los cambios del sistema no requieren cambiar esas conexiones.

* **COMPOSITE**

Es un patrón de diseño estructural que sirve para construir algoritmos u objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol.

Además, permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

**¿CUÁNDO SE UTILIZA?**

1. Se busca representar una jerarquía de objetos como “parte-todo”.
2. Se busca que el cliente pueda ignorar la diferencia entre objetos primitivos y compuestos (para que pueda tratarlos de la misma manera).

**VENTAJAS**

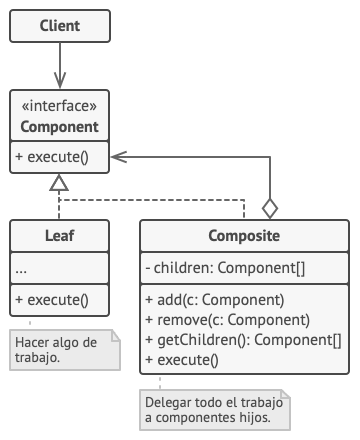
* Proporciona dos tipos de elementos básicos que comparten una interfaz común: hojas simples y contenedores complejos. Esto nos permite construir una estructura de objetos recursivos anidados parecida a un árbol.
* Todos los elementos definidos por este patrón comparten una interfaz común
* El cliente no tiene que preocuparse por la clase concreta de los objetos con los que funciona ya que para comunicarse con los componentes lo hace atreves de la interfaz component

**DESVENTAJAS**

* Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado. En algunos casos, tendrás que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender.

**¿EN QUE CONSISTE?**

Este patrón busca representar una jerarquía de objetos conocida como “parte-todo”, donde se sigue la teoría de que las "partes" forman el "todo", siempre teniendo en cuenta que cada "parte" puede tener otras "parte" dentro.

**DIAGRAMA DE CLASES**

1. **COMPONENT:** implementa un comportamiento común entre las clases y declara una interface de manipulación a los padres en la estructura recursiva.
2. **LEAF:** representa los objetos “hoja” (no poseen hijos). Define comportamientos para objetos primitivos.
3. **COMPOSITE:** define un comportamiento para objetos con hijos. Almacena componentes hijos e implementa operaciones de relación con los hijos.
4. **CLIENTE:** manipula objetos de la composición a través de Component. Los clientes usan la interfaz de Component para interactuar con objetos en la estructura Composite. Si el receptor es una hoja, la interacción es directa. Si es un Composite, se debe llegar a los objetos “hijos”, y puede llevar a utilizar operaciones adicionales.

Diagrama

Descripción generada automáticamente**DIAGRAMA DE SECUENCIA**

* **PRINCIPIO DE INVERSIÓN DE DEPENDENCIA**

**¿DE DONDE PROVIENE?**

Provienen de los principios SOLID (conjunto de principios aplicables a la POO):

S 🡪 PRINCIPIO RESPONSABILIDAD ÚNICA

O 🡪 PRINCIPIO OPEN/CLOSED

L 🡪 PRINCIPIO DE SUSTITUCIÓN DE LISKOV

I 🡪 PRINCIPIO DE SEGREGACIÓN DE INTERFACES

D 🡪 PRINCIPIO DE INVERSIÓN DE DEPENDENCIAS

En este principio hay que tener en cuenta:

* Módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
* Abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones.

Puede implementarse con: Inyección De Dependencia e Inversión del Control

**INYECCIÓN DE DEPENDENCIA** 🡪 es un mecanismo que se encargar de extraer la responsabilidad de creación de instancias de un componente para delegar en otro. Separa la construcción de la ejecución

**INVERSIÓN DE CONTROL** 🡪 se refiere a que la app delega algun tipo de flujo de control de un tercero (framework). El framework se encarga de gestionar el ciclo de vida de la app e ira notificando eventos a la propia app para que esta actue en consecuencia

**DIAGRAMA DE CLASES**

Módulos de alto nivel 🡪 son aquellos q tienen dependencias y donde se invoca la acción

Módulos de bajo nivel 🡪 son aquellos de los que dependen y donde se realiza la acción

* Las clases importantes no deben depender de las clases menos importantes. Ambas deben depender de interfaces o clases abstractas
* Las clases más estables no deben depender de las clases más inestables. Ambas deben depender de interfaces o clases abstractas
* Las interfaces y clases abstractas no deben depender de las implementaciones. Las implementaciones deben depender de las interfaces y clases abstractas.

Diagrama

Descripción generada automáticamente

* **MODELO VISTA CONTROLADOR**

Es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos:

* **El Modelo:** que contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia.
* **La Vista (interfaz de usuario):**  que compone la información que se envía al cliente y los mecanismos de interacción con éste.  
  **El Controlador:** que actúa como intermediario entre el Modelo y la Vista, gestiona el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

API 🡪 Es una interfaz de programación de aplicaciones cuya función es conectar sistemas, software y aplicaciones. Le permite a un usuario final utilizar una aplicación o software agregando, modificando y almacenando datos sin que el usuario tenga que ingresar a ellos directamente

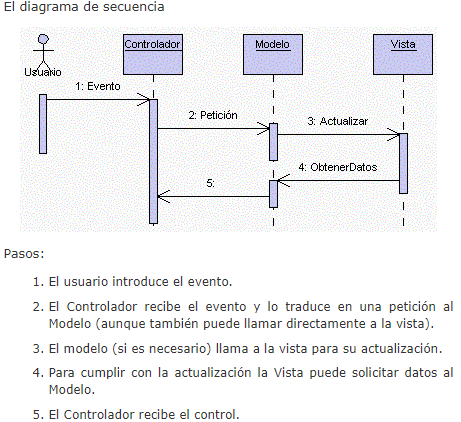
Diagrama

Descripción generada automáticamente**DIAGRAMA DE CLASES**

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)
2. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)
3. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra).

**DIAGRAMA DE SECUENCIA**

Diagrama

Descripción generada automáticamente

**PATRONES DE COMPORTAMIENTO**

Estos patrones tratan con algoritmos y la asignación de responsabilidades entre objetos.

* **CADENA DE RESPONSABILIDAD**

Es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

**¿CUÁNDO SE UTILIZA?**

1. Las peticiones emitidas por un objeto deben ser atendidas por distintos objetos receptores.
2. No se sabe a priori cual es el objeto que me puede resolver el problema.
3. Cuando un pedido debe ser manejado por varios objetos.

**VENTAJAS**

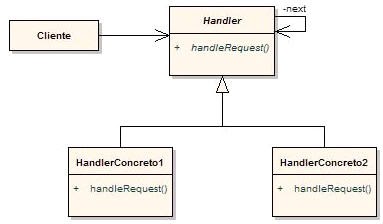
* Las peticiones emitidas por un objeto deben ser atendidas por distintos objetos receptores.
* No se sabe a priori cual es el objeto que me puede resolver el problema.
* Cuando un pedido debe ser manejado por varios objetos.

**DESVENTAJAS**

* ALGUNAS SOLICITUDES PUEDEN ACABAR SIN SER GESTIONADAS.

**¿EN QUE CONSISTE?**

Consiste en crear un sistema que pueda servir a diversas solicitudes de manera jerárquica. En otras palabras, si un objeto que es parte de un sistema no sabe cómo responder a una solicitud, la pasa a lo largo del árbol de objetos. Como el nombre lo implica, cada objeto de dicho árbol puede tomar la responsabilidad y atender la solicitud.

**DIAGRAMA DE CLASES**

**Handler**: define una interfaz para tratar las peticiones. Implementa el enlace al sucesor.

**HandlerConcreto**: trata las peticiones de las que es responsable. Si puede manejar la petición, lo hace, en caso contrario la reenvía a su sucesor. Además van a implementar las funciones del handler

**DIAGRAMA DE SECUENCIA**

**Diagrama

Descripción generada automáticamente**

* **Diagrama

  Descripción generada automáticamenteOBSERVER**

Es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando

**¿CUÁNDO SE UTILIZA?**

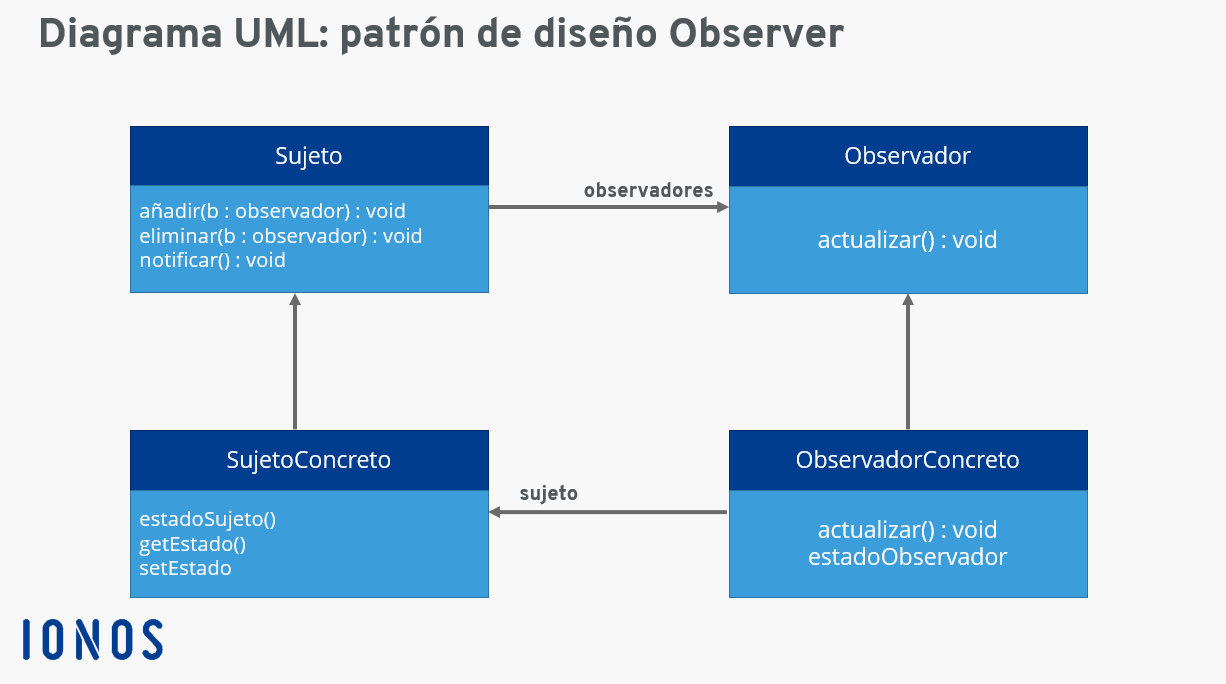
1. Un objeto necesita notificar a otros objetos cuando cambia su estado.
2. Cuando existe una relación de dependencia de uno a muchos que puede requerir que un objeto notifique a múltiples objetos que dependen de él cuando cambia su estado.

**VENTAJAS**

* Permite añadir nuevos observadores en tiempo de ejecución, sin que esto afecte a ningún otro observador.
* Permite que dos capas de diferentes niveles de abstracción se puedan comunicar entre sí sin romper esa división.
* Permite modificar las clases subjects y las observers independientemente

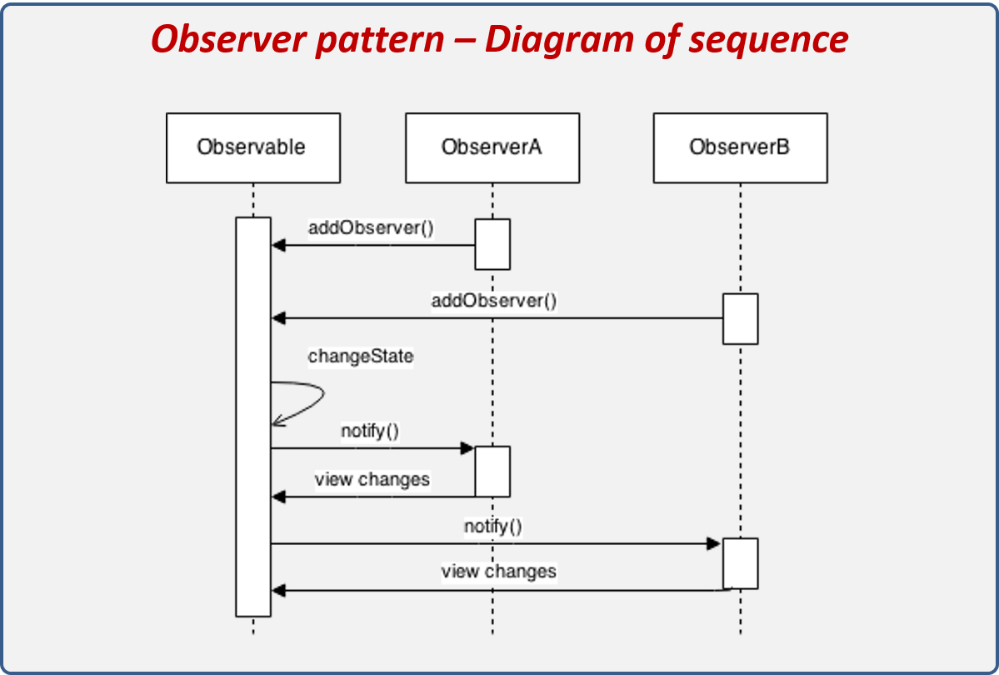
**DESVENTAJA**

* Los suscriptores son notificados en un orden aleatorio.

**DIAGRAMA DE CLASE**

* **Subject:** conoce a sus observadores y ofrece la posibilidad de añadir y eliminar observadores. Posee un método llamado attach () y otro detach () que sirven para agregar o remover observadores en tiempo de ejecución.
* **Observer:** define la interfaz que sirve para notificar a los observadores los cambios realizados en el Subject.
* **SubjectConcreto:** almacena el estado que es objeto de interés de los observadores y envía un mensaje a sus observadores cuando su estado cambia.
* **ObserverConcreto**: mantiene una referencia a un SubjectConcreto. Almacena el estado del Subject que le resulta de interés. Implementa la interfaz de actualización de Observer para mantener la consistencia entre los dos estados.

**DIAGRAMA DE SECUENCIA**



1. El ObserverA se registra con el objeto Observable para ser notificado de algun cambio
2. El ObserverB se registra con el objeto Observable para ser notificado de algún cambio
3. Ocurre algún cambio en el estado del Observble
4. Todos los Observers son notificados con el cambio ocurrido