Patrones de diseño

[Patrones creacionales: 1](#_Toc192884385)

[Singleton 1](#_Toc192884386)

[Factory Method 4](#_Toc192884387)

[Factory Parametrizado 9](#_Toc192884388)

[Abstract Factory 10](#_Toc192884389)

[Structural Patterns 14](#_Toc192884390)

[Adapter 14](#_Toc192884391)

[Façade 18](#_Toc192884392)

[Composite 20](#_Toc192884393)

[Patrones de comportamiento 23](#_Toc192884394)

[Command 23](#_Toc192884395)

[Observer 28](#_Toc192884396)

[State 31](#_Toc192884397)

# Patrones creacionales:

Proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente

Se cetra en la instanciación de clases (crear objetos (Carlos) a partir de una clase(Persona))

#### *Propósito*:

* Crear un objeto es una toma de decisión
* Separar los procesos de creación de objeto y de uso de un objeto

## Singleton

#### Propósito

Garantiza que una clase tenga una única instancia, a la vez que proporciona un punto global de acceso a ella

#### Aplicabilidad

* Necesita exactamente una sola instancia de una clase para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa
  + El patrón Singleton deshabilita el resto de las maneras de crear objetos de una clase, excepto el método especial de creación. Este método crea un nuevo objeto, o bien devuelve uno existente si ya ha sido creado.
* Es deseable una inicialización perezosa de la instancia
* Necesario un acceso global a la instancia
* Cuando necesites un control más estricto de las variables globales.
  + Garantiza que haya una única instancia de una clase. A excepción de la propia clase Singleton, nada puede sustituir la instancia en caché.
  + Ten en cuenta que siempre podrás ajustar esta limitación y permitir la creación de cierto número de instancias Singleton. La única parte del código que requiere cambios es el cuerpo del método getInstance.

#### Estructura

Diagrama

El contenido generado por IA puede ser incorrecto.Texto

El contenido generado por IA puede ser incorrecto.

1. La clase **Singleton** declara el método estático *ObtenerInstancia()* que devuelve la misma instancia de su propia clase
2. El constructor del **Singleton** debe estar culto del código cliente. La llamada a *ObtenerInstancia()* debe ser la única manera de obtener el objeto de Singleton

#### Problema

1. Garantiza que una clase tenga una única instancia: Controla el acceso a algún recurso compartido, por ejemplo, una base de datos

Imagina que has creado un objeto y al cabo de un tiempo quieres crear otro nuevo. En lugar de recibir un objeto nuevo, tendrás el que ya tenías.

Esto es imposible de hacer con un constructor normal ya que una llamada a él, devuelve un objeto nuevo por diseño

1. Proporciona un punto de acceso global a dicha instancia: son poco seguras porque podrías sobreescribir el contenido de las variables y descomponer la app

#### Solución

* Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton
  + private Singleton() {}
* Crear un método de creación estático the actúe como constructor. Este método invoca al constructor privado de crear un objeto y lo guarda en un campo estático
  + // instancia única

private static Singleton instancia = null;

public static Singleton getInstancia(){

if (instancia == null)

instancia = new Singleton();

return instancia; }

#### Analogía del mundo real

El gobierno es un ejemplo del patrón. Un país solo puede tener un gobierno oficial. Independientemente de las identidades personales de los individuos que forman el gobierno, el título “*Gobierno de X”* es un punto de acceso global que identifica al grupo de personas a cargo

#### Implementación

1. Añade un campo estático privado a la clase para almacenar la instancia Singleton.

// Static Area

private static DiceSingleton theDice; // A unique instance

1. Declara un método de creación estático público para obtener la instancia Singleton.

/\* A UNIQUE PUBLIC AND STATIC METHOD\*/

public static DiceSingleton getInstance(){

if (theDice == null) // LAZY CONSTRUCTION, IT IS DELAYED UNTIL THE FIRST ACCESS IS PERFORMED

theDice = new DiceSingleton();

return theDice;

}

1. Implementa una inicialización diferida dentro del método estático. Debe crear un nuevo objeto en su primera llamada y colocarlo dentro del campo estático. El método deberá devolver siempre esa instancia en todas las llamadas siguientes.
2. Declara el constructor de clase como privado. El método estático de la clase seguirá siendo capaz de invocar al constructor, pero no a los otros objetos.

/\*\*The constructor cannot be used by other classes \*/

private DiceSingleton(){

generator = new Random();

}

1. Repasa el código cliente y sustituye todas las llamadas directas al constructor de la instancia Singleton por llamadas a su método de creación estático.

DiceSingleton d3 = DiceSingleton.getInstance();

System.out.println("Singleton implementation -> Rolling result is "+ d3.roll());

#### Pros y contras

* Acceso controlado a la única instancia por la clase
* Evita inundar el código con variables globales
  + No deberían ser un “sustituto” de toda variable global
* Permite el refinamiento mediante herencia
  + El observador puede devolver un objeto Singleton o de una clase derivada especializada
* Permite un número de variable de instancias
  + Ej: Acceso a un objeto de comunicaciones, para no saturar un canal, el Singleton puede disponer internamente de 2 instancias que va alternando en cada solicitud.
* Son más flexibles que usar directamente la clase con métodos estáticos
  + Los métodos estáticos no se pueden reescribir en métodos heredados

## Factory Method

#### Propósito

* Define una interfaz para crear para crear objetos en una superclase mientras permite a las subclases alterar el tipo de objetos que se crearán
* Permite a una clase delegar la instanciación en sus subclases
* Define constructor virtual

#### Aplicabilidad

* Utiliza el Método Fábrica cuando no conozcas de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tu código.
  + El patrón Factory Method separa el código de construcción de producto del código que hace uso del producto. Por ello, es más fácil extender el código de construcción de producto de forma independiente al resto del código.
  + Por ejemplo, para añadir un nuevo tipo de producto a la aplicación, sólo tendrás que crear una nueva subclase creadora y sobrescribir el Factory Method que contiene
* Utiliza el Factory Method cuando quieras ofrecer a los usuarios de tu biblioteca o framework, una forma de extender sus componentes internos.
  + Veamos cómo funcionaría. Imagina que escribes una aplicación utilizando un framework de UI de código abierto. Tu aplicación debe tener botones redondos, pero el framework sólo proporciona botones cuadrados. Extiendes la clase estándar *Botón* con una maravillosa subclase *BotónRedondo*, pero ahora tienes que decirle a la clase principal *FrameworkUI* que utilice la nueva subclase de botón en lugar de la clase por defecto.
  + Para conseguirlo, creamos una subclase *UIConBotonesRedondos* a partir de una clase base del framework y sobrescribimos su método *crearBotón*. Si bien este método devuelve objetos Botón en la clase base, haces que tu subclase devuelva objetos *BotónRedondo*. Ahora, utiliza la clase *UIConBotonesRedondos* en lugar de FrameworkUI. ¡Eso es todo!
* Utiliza el Factory Method cuando quieras ahorrar recursos del sistema mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez.

#### Problema

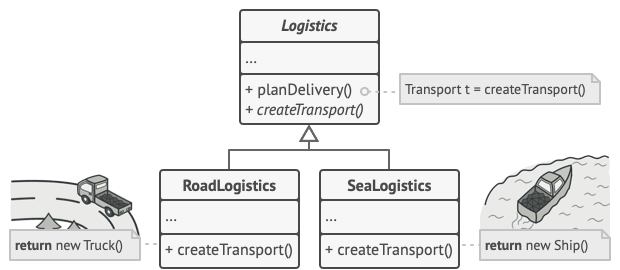
Imagina estar creando una app de gestión logística.

La primera versión solo es capaz de siportar el transporte camión y la mayoría del código está ahí

Luego quieres implementar transporte marítimo, tienes que pasar todo lo de camión a el nuevo y así si creas más.

#### Solución

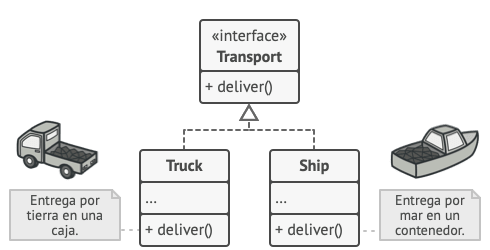
El patrón factory method sugiere que en lugar de llamar al operador *new* para construir objetos directamente se invoque el método de *fábrica especial*. Los objetos devueltos por por el método fábrica se denominan productos



Las subclases pueden alterar la clase de los objetos devueltos por el método fábrica.

Ahora puedes sobrescribir el método fábrica en una subclase y cambiar la clase de los productos creados por el método.

No obstante, hay una pequeña limitación: las subclases sólo pueden devolver productos de distintos tipos si dichos productos tienen una clase base o interfaz común. Además, el método fábrica en la clase base debe tener su tipo de retorno declarado como dicha interfaz.

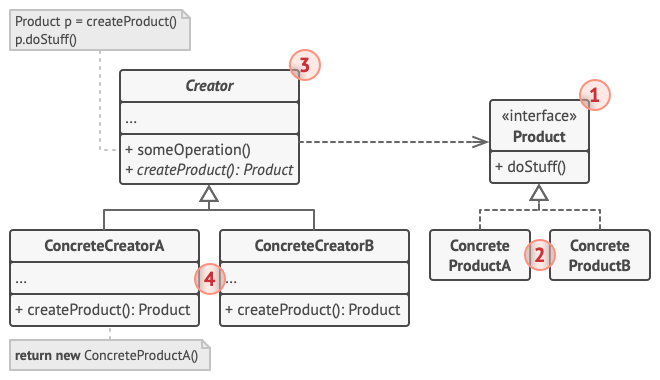


Todos los productos deben seguir la misma interfaz

Por ejemplo, tanto la clase *Camión* como la clase *Barco* deben implementar la interfaz *Transporte*, que declara un método llamado *deliver().* Cada clase implementa este método de forma diferente: los camiones entregan su carga por tierra, mientras que los barcos lo hacen por mar. El método fábrica dentro de la clase *RoadLogistics* devuelve objetos de tipo camión, mientras que el método fábrica de la clase *SeaLogistics* devuelve barcos.

El código que utiliza el método fábrica (a menudo denominado código cliente) no encuentra diferencias entre los productos devueltos por varias subclases, y trata a todos los productos como la clase abstracta *Transport*. El cliente sabe que todos los objetos de transporte deben tener el método *deliver(),* pero no necesita saber cómo funciona exactamente.

#### Estructura



1. El Producto declara la interfaz, que es común a todos los objetos que puede producir la clase creadora y sus subclases.
2. Los Productos Concretos son distintas implementaciones de la interfaz de producto.
3. La clase Creadora declara el método fábrica que devuelve nuevos objetos de producto. Es importante que el tipo de retorno de este método coincida con la interfaz de producto.
4. Los Creadores Concretos sobrescriben el Factory Method base, de modo que devuelva un tipo diferente de producto.

Observa que el método fábrica no tiene que crear nuevas instancias todo el tiempo. También puede devolver objetos existentes de una memoria caché, una agrupación de objetos, u otra fuente.

#### Implementación

1. Haz que todos los productos sigan la misma interfaz. Esta interfaz deberá declarar métodos que tengan sentido en todos los productos.
2. Añade un patrón Factory Method vacío dentro de la clase creadora. El tipo de retorno del método deberá coincidir con la interfaz común de los productos.
3. Encuentra todas las referencias a constructores de producto en el código de la clase creadora. Una a una, sustitúyelas por invocaciones al Factory Method, mientras extraes el código de creación de productos para colocarlo dentro del Factory Method.
4. Ahora, crea un grupo de subclases creadoras para cada tipo de producto enumerado en el Factory Method. Sobrescribe el Factory Method en las subclases y extrae las partes adecuadas de código constructor del método base.
5. Si hay demasiados tipos de producto y no tiene sentido crear subclases para todos ellos, puedes reutilizar el parámetro de control de la clase base en las subclases.

Por ejemplo, imagina que tienes la siguiente jerarquía de clases: la clase base Correo con las subclases CorreoAéreo y CorreoTerrestre y la clase Transporte con Avión, Camión y Tren. La clase CorreoAéreo sólo utiliza objetos Avión, pero CorreoTerrestre puede funcionar tanto con objetos Camión, como con objetos Tren. Puedes crear una nueva subclase (digamos, CorreoFerroviario) que gestione ambos casos, pero hay otra opción. El código cliente puede pasar un argumento al Factory Method de la clase CorreoTerrestre para controlar el producto que quiere recibir.

#### Standard

Forma

El contenido generado por IA puede ser incorrecto.

public abstract class Application {

private ArrayList<Document> docs; // collection of documents

public Application() { //Default Constructor

docs = new ArrayList<Document>(); }

public abstract Document createDocument(); // abstract method that generates a document depending on the application

public void newDocument() {

docs.add(/\* new WordDoc() \*/ createDocument()); } //New document

public void openDocument(String path) { //Open saved document, path to where it is

Document x = /\*new WordDoc()\*/ createDocument();

// read the document from the folder in -> x

x.open(path);

docs.add(x); }

}

public class Excel extends Application {

@Override

public Document createDocument() {

return new ExcelDoc(); }

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

public class Word extends Application {

@Override

public Document createDocument() {

return new WordDoc();}

}

Public interface Document{

void open(String path);

void close();

void save();

void undo();

}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

public class ExcelDoc implements Document {

public void open(String path) {// load the document using the path }

public void close() {// ... }

public void save() {// ... }

public void undo() {// ... }

}

Public class WordDoc implements Document{

public void open(String path) {// load the document using the path }

public void close() {// ... }

public void save() {// ... }

public void undo() {// ... }

}

## Factory Parametrizado

Diagrama

El contenido generado por IA puede ser incorrecto.Texto

El contenido generado por IA puede ser incorrecto.

#### Consecuencias

* Permite variar la representación interna de un producto
* Aísla el código de construcción y representación
  + Los Clientes no necesitan conocer la estructura interna del Producto, tampoco aparecen en Constructor
  + ConstructorConcreto define todas las componentes y como se ensamblan
* Proporciona un control más fino sobre el proceso de construcción
  + El Constructor construye el producto paso a paso
  + El Cliente sólo obtiene el producto cuando está terminado
* Pueden devolver la misma instancia varias veces (Prototype)
* (Parametrizado) el método del Producto abstracto puede devolver objetos de diferentes subclases

## Abstract Factory

#### Propósito

Nos permite crear familias de objetos relacionados sin especificar sus clases concretas

#### Aplicabilidad

* Cuando hay necesidad de diferentes tipos de objetos que no quieren colaborar
* Configura un sistema escogiendo una familia de productos entre otros
* Divide los objetos por como están siendo usados
* Cuando queremos ofertar una librería de productos y solo queremos revelar sus interfaces y no las implementaciones

#### Familias de productos y sus variantes.Problema

Imagina un simulador de muebles

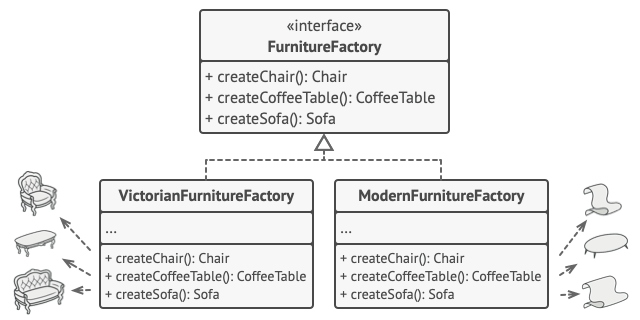
1. Una familia de productos relacionados, digamos: *Silla* + *Sofá* + *Mesilla*.
2. Algunas variantes de esta familia. Por ejemplo, los productos *Silla* + *Sofá* + *Mesilla* están disponibles en estas variantes: *Moderna*, *Victoriana*, *ArtDecó*.

Necesitamos una forma de crear objetos individuales de mobiliario para que combinen con otros objetos de la misma familia. Los clientes se enfadan bastante cuando reciben muebles que no combinan.

Además, no queremos cambiar el código existente al añadir al programa nuevos productos o familias de productos. Los comerciantes de muebles actualizan sus catálogos muy a menudo, y debemos evitar tener que cambiar el código principal cada vez que esto ocurra.

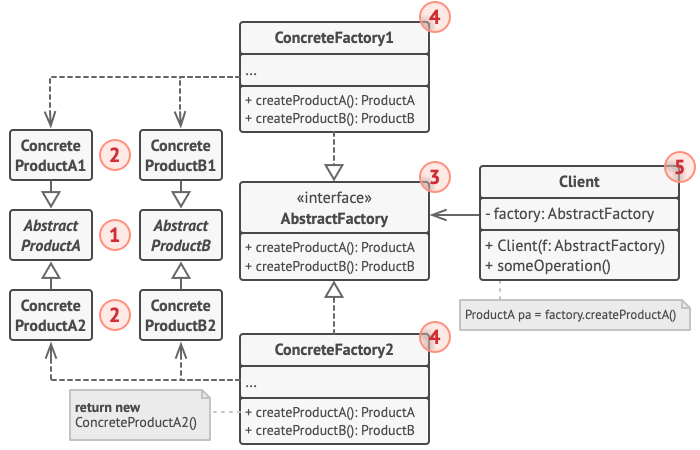
#### La jerarquía de la clase SillaSolución

Lo primero que sugiere el patrón Abstract Factory es que declaremos de forma explícita interfaces para cada producto diferente de la familia de productos (por ejemplo, silla, sofá o mesilla). Después podemos hacer que todas las variantes de los productos sigan esas interfaces. Por ejemplo, todas las variantes de silla pueden implementar la interfaz *Silla*, así como todas las variantes de mesilla pueden implementar la interfaz *Mesilla*, y así sucesivamente.

El siguiente paso consiste en declarar la Fábrica abstracta: una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos (por ejemplo, *crearSilla*, *crearSofá* y *crearMesilla*). Estos métodos deben devolver productos abstractos representados por las interfaces que extrajimos previamente: *Silla*, *Sofá*, etc.

Ahora bien, ¿qué hay de las variantes de los productos? Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz *FábricaAbstracta*. Una fábrica es una clase que devuelve productos de un tipo particular. Por ejemplo, la *FábricadeMueblesModernos* sólo puede crear objetos de *SillaModerna*, *SofáModerno* y *MesillaModerna*.

#### Estructura

1. Los Productos Abstractos declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.
2. Los Productos Concretos son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto (silla/sofá) debe implementarse en todas las variantes dadas (victoriano/moderno).
3. La interfaz Fábrica Abstracta declara un grupo de métodos para crear cada uno de los productos abstractos.
4. Las Fábricas Concretas implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante específica de los productos y crea tan solo dichas variantes de los productos.

Aunque las fábricas concretas instancian productos concretos, las firmas de sus métodos de creación deben devolver los productos abstractos correspondientes. De este modo, el código cliente que utiliza una fábrica no se acopla a la variante específica del producto que obtiene de una fábrica. El Cliente puede funcionar con cualquier variante fábrica/producto concreta, siempre y cuando se comunique con sus objetos a través de interfaces abstractas.

#### Código

Diagrama

El contenido generado por IA puede ser incorrecto.

interface FactoryOfSnacks {

Food giveFood();

Drinks giveDrink();

}

public abstract class Drinks {

public abstract void drink();

}

public class Chocolate extends Drinks {

public void drink() { /\* DRINKING CHOCOLATE\*/

System.out.println("Drinking chocolate");

}

}

public class FactoryOfHamburguers implements FactoryOfSnacks {

@Override

public Food giveFood() {

return new Hamburguer();

}

@Override

public Drinks giveDrink() {

return new Coke();

}

}

public class SnackClient {

private FactoryOfSnacks f = null;

SnackClient(String typeOfSnack) {

/\* We choose a kind of food and the drink is linked to it \*/

/\* They are already compatible, because they are created together \*/

//------------------------------------------------------

// SELECT THE FAMILY OF PRODUCTS TO CONSUME

switch (typeOfSnack) {

case "Hamburguer":

f = new FactoryOfHamburguers();

break;

case "Churros":

f = new FactoryOfChocolate();

break;

default: //nothing

}

}

public void createSnack() {

//------------------------------------------------------

// Creation of compatible products to consume

Food c = f.giveFood();

Drinks b = f.giveDrink();

//------------------------------------------------------

// Use them

c.eat();

b.drink();

}

}

public class Client {

public static void main(String[] args) {

//Client: Hamburguer

SnackClient c1 = new SnackClient("Hamburguer");

System.out.println("Snack for Client 1:");

c1.createSnack();

//Client: Churros

SnackClient c2 = new SnackClient("Churros");

System.out.println("Snack for Client 2:");

c2.createSnack();

//A WEIRD CLIENT IS NOT POSSIBLE NOW

}

}

1. Mapea una matriz de distintos tipos de productos frente a variantes de dichos productos.
2. Declara interfaces abstractas de producto para todos los tipos de productos. Después haz que todas las clases concretas de productos implementen esas interfaces.
3. Declara la interfaz de la fábrica abstracta con un grupo de métodos de creación para todos los productos abstractos.
4. Implementa un grupo de clases concretas de fábrica, una por cada variante de producto.
5. Crea un código de inicialización de la fábrica en algún punto de la aplicación. Deberá instanciar una de las clases concretas de la fábrica, dependiendo de la configuración de la aplicación o del entorno actual. Pasa este objeto de fábrica a todas las clases que construyen productos.
6. Explora el código y encuentra todas las llamadas directas a constructores de producto. Sustitúyelas por llamadas al método de creación adecuado dentro del objeto de fábrica.

#### Contras

* Isolate specific classs from the client who uses them
* Facilitate the exchange of families of products
* Foster the consistency among products 🡪 families
* Its difficult to add new type of products
  + Imply the creation of new methods along the whole hierarchy of factories

# Structural Patterns

Explican como ensamblar objetos y clases en estructuras más grandes, mientras se mantiene la flexibilidad y eficiencia de la estructura original

* Desacoplar el sistema
* Obtener una estructura flexible
* Organizar

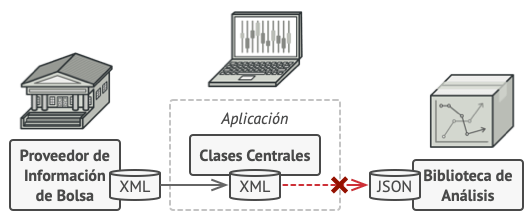
## Adapter

Permite la colaboración entre objetos con interfaces incompatibles. Convierte la interfaz de uno o muchas clases a otra interfaz que el cliente espera

#### Applicability

* Se requiere usar una clase existente cuya interfaz no es compatible con lo que se necesita
* Requiere la creación de una clase para colaborar con clases con quienes no comparten interfaces compatibles
* Cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase
* Object adapter
  + Necesitamos usar varias subclases existentes pero no es útil adaptar su interfaz heredando de cada una de ellas 🡪 Un objeto adaptador puede adaptar la interfaz de la clase padre
    - El Class Adapter es útil cuando se puede extender la clase directamente y se desea una solución más simple.
    - El Object Adapter es más flexible, ya que permite adaptar objetos sin necesidad de herencia, lo que hace que sea más adecuado cuando se manejan objetos de diferentes tipos o cuando no es posible modificar las clases base.

#### Problema

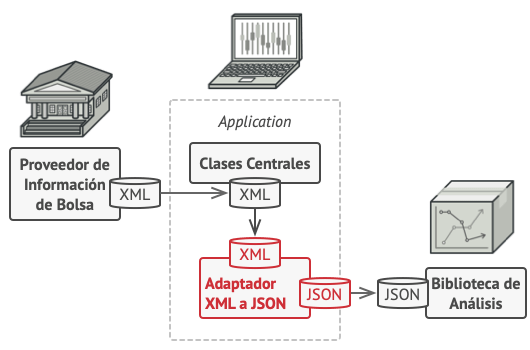
Imagina que estás creando una aplicación que descarga información desde varias fuentes en XML para presentarla al usuario con gráficos bonitos y diagramas

En cierto momento decides mejorar la app intefrando una biblioteca inteligente de análisis pero solo funciona con JSON

Podrías cambiar la biblioteca para que funcione con XML. Sin embargo, esto podría descomponer parte del código existente que depende de la biblioteca. Y, lo que es peor, podrías no tener siquiera acceso al código fuente de la biblioteca, lo que hace imposible esta solución.

#### Solución

Creas un adaptador. Es un objeto especial que convierte la interfaz de un objeto de forma que otro objeto pueda entenderla

Un adaptador envuelve uno de los objetos para esconder la complejidad de la conversión. El objeto envuelto ni siquiera es consciente de la existencia del adaptador. Puedes envolver un objeto que opera con metros y km con una adaptador que convierte los datos a pies y millas

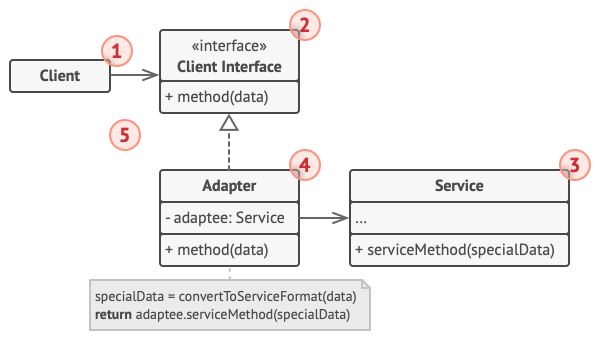
Los adaptadores no solo convierten datos a varios formatos, sino que también ayudan a objetos con distintas interfaces a colaborar.

1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes
2. Usando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador
3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto pero en un formato y orden que ese segundo objeto espera

#### Estructura

##### Adaptador de objetos

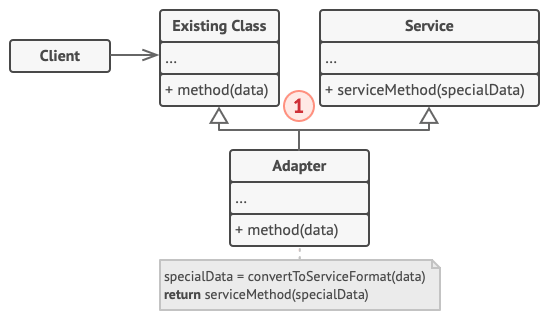
Esta implementación utiliza el principio de composición de objetos: el adaptador implementa la interfaz de un objeto y envuelve el otro. Puede implementarse en todos los lenguajes de programación populares.

1. La clase cliente contiene la lógica de negocio existente del programa
2. La interfaz con el cliente describe un protocolo que otras clases deben seguir para poder colaborar con el código cliente
3. Servicio es aguna clase útil (El cliente no puede usar esto pq tiene interfaz incompatible)
4. La clase Adapter es un puente entre cliente y servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio.
   1. La clase adaptadora recibe llamadas del cliente a través de la interfaz de cliente y las traduce en llamadas al objeto envuelto de la clase de servicio

Diagrama

El contenido generado por IA puede ser incorrecto.

##### Clase adaptadora

Esta implementación utiliza la herencia, porque la clase adaptadora hereda interfaces de ambos objetos al mismo tiempo.

1. La *Clase adaptadora* no necesita envolver objetos porque hereda comportamientos tanto de la clase cliente como de la clase de servicio. La adaptación tiene lugar dentro de los métodos sobrescritos. La clase adaptadora resultante puede utilizarse en lugar de una clase cliente existente.

Diagrama

El contenido generado por IA puede ser incorrecto.

1. Asegúrate de que tienes al menos dos clases con interfaces incompatibles
   1. Una útil clase servicio que no puedes cambiar (a menudo de un tercero, heredada o con muchas dependencias existentes).
   2. Una o varias clases cliente que se beneficiarían de contar con una clase de servicio.
2. Declara la interfaz con el cliente y describe el modo en que las clases cliente se comunican con la clase de servicio.
3. Crea la clase adaptadora y haz que siga la interfaz con el cliente. Deja todos los métodos vacíos por ahora.
4. Añade un campo a la clase adaptadora para almacenar una referencia al objeto de servicio. La práctica común es inicializar este campo a través del constructor, pero en ocasiones es adecuado pasarlo al adaptador cuando se invocan sus métodos.
5. Uno por uno, implementa todos los métodos de la interfaz con el cliente en la clase adaptadora. La clase adaptadora deberá delegar la mayor parte del trabajo real al objeto de servicio, gestionando tan solo la interfaz o la conversión de formato de los datos.

Las clases cliente deberán utilizar la clase adaptadora a través de la interfaz con el cliente. Esto te permitirá cambiar o extender las clases adaptadoras sin afectar al código cliente.

#### Participantes

* Objective
  + Define a specific interface from the Client domain.
* Client
  + Collaborate with objects which adjust to the Objective interface.
* Adaptable
  + Define an existing class with an interface which needs to be adapted.
* Adapter
  + Adapts the Adaptable interface to the Objective interface

#### Consecuencias

* Class Adapter:
  + An Adapter only is useful to adapt a specific Adaptable class, but not all Adaptable subclasses.
  + Adapter, as an Adaptable subclass, can redefine part of the inherited class.
  + Introduce a unique object, it does not need any reference to an object by composition.
* Object Adapter:
  + The same Adapter can work with multiple Adaptables(its subclasses).
  + Adapter can add functionality to all the adaptables, andhide the adapted object.
  + Introduce an object apart from the Adaptable.

## Façade

Proporciona una interfaz simplificada a una biblioteca de interfaces en un subsistema o cualquier otro grupo complejo de clases

Define un nivel alto de interfaces para facilitar el uso del subsistema

#### Aplicabilidad

* Hay una necesidad de proveer una interfaz sencilla en un subsistema complejo
* Hay muchas dependencias entre clientes y el subsistema
* Puedes dividirlo en una o más capas para definir un único input point a cada nivel del subsistema
* Cuando necesites una interfaz limitada pero directa a un subsistema complejo

#### Participantes

* Clientes
  + Usan el subsistema
* Fachada
  + Manda client request a las clases correspondientes
  + Additional work for adapting the interfaces
* Subsistem Classes
  + Implementa la funcionalidad del sistema
  + No distingue a la fachada de otros clientes

#### Estructura

Diagrama

El contenido generado por IA puede ser incorrecto.

1. El patrón Facade proporciona un práctico acceso a una parte específica de la funcoinalidad del subsistema. Sabe dirigir la petición del cliente y como operar todas las partes móviles
2. Puedes crear una fachada adicional para evitar contaminar una única fachada con funciones no relacionadas que podrían convertirla en otra estructura compleja. Las fachadas adicionales pueden utilizarse por clientes y por otras fachadas.
3. El Subsistema Complejo consiste en decenas de objetos diversos. Para lograr que todos hagan algo significativo, debes profundizar en los detalles de implementación del subsistema, que pueden incluir inicializar objetos en el orden correcto y suministrarles datos en el formato adecuado.
4. El Cliente utiliza la fachada en lugar de invocar directamente los objetos del subsistema.

#### Implementación

1. Comprueba si es posible proporcionar una interfaz más simple que la que está proporcionando un subsistema existente. Estás bien encaminado si esta interfaz hace que el código cliente sea independiente de muchas de las clases del subsistema.
2. Declara e implementa esta interfaz en una nueva clase fachada. La fachada deberá redireccionar las llamadas desde el código cliente a los objetos adecuados del subsistema. La fachada deberá ser responsable de inicializar el subsistema y gestionar su ciclo de vida, a no ser que el código cliente ya lo haga.
3. Para aprovechar el patrón al máximo, haz que todo el código cliente se comunique con el subsistema únicamente a través de la fachada. Ahora el código cliente está protegido de cualquier cambio en el código del subsistema. Por ejemplo, cuando se actualice un subsistema a una nueva versión, sólo tendrás que modificar el código de la fachada.
4. Si la fachada se vuelve demasiado grande, piensa en extraer parte de su comportamiento y colocarlo dentro de una nueva clase fachada refinada.

#### Consecuencias

* A client is apart from the subsystem components.
  + The subsystem is easier to use.
* Facilitate a low connection between the subsystem and the client.
  + Help to structure the software in layers and its dependencies.
  + Facilitate portability among platforms (only the layers that need it are reimplemented).
    - Ex: Music app: Collections, Files, Access to disk.
  + Allow to identify circular or complex dependencies.
* It does not avoid a direct access to the subsystem objects.

## Composite

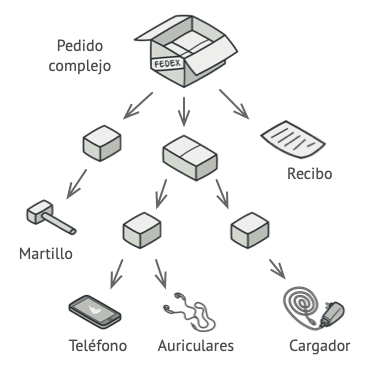
Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales

* Compose objects in tree structures to represent hierarchies part-whole.
  + Recursive composition
* Allow clients to address in a uniform way individual objects and the compounds.

#### Applicability

* Represent hierarchies part-whole.
* Cuando tengas que implementar una estructura en forma de árbol
* Clients do not see the difference between composition of objects and individual objects: every object is addressed in the same way.

#### Problema

Sólo tiene sentido si el modelo central de tu app puede representarse en forma de árbol

Por ejemplo, imagina que tienes dos tipos de objetos: *Productos* y *Cajas*. Una *Caja* puede contener varios *Productos,* así como cierto número de *Cajas* más pequeñas. Estas *Cajas* pequeñas también pueden contener algunos *Productos* o incluso *Cajas* más pequeñas, y así sucesivamente.

Puedes intentar la solución directa: desenvolver todas las cajas, repasar todos los productos y calcular el total. Esto sería viable en el mundo real; pero en un programa no es tan fácil como ejecutar un bucle. Tienes que conocer de antemano las clases de Productos y Cajas a iterar, el nivel de anidación de las cajas y otros detalles desagradables. Todo esto provoca que la solución directa sea demasiado complicada, o incluso imposible.

#### Solución

El patrón Composite sugiere que trabajes con Productos y Cajas a través de una interfaz común que declara un método para calcular el precio total.

Para un producto, sencillamente devuelve el precio del producto. Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos. Una caja podría incluso añadir costos adicionales al precio final, como costos de empaquetado.

#### Diagrama El contenido generado por IA puede ser incorrecto.Structure

* Component (Graphic)
  + Define an interface for the objects of the composition.
  + Implement a predefined behaviour for the common interface to every object.
  + Define an interface to access to the children nodes and manage them.
  + (optional) interface to access to the father node.
  + Describe operaciones que son comunes a elementos simples y complejos del árbol
* Leaf (Rectangle, Line, Text, etc.)
  + Represent final objects, without children
* Compound (Draw)
  + Define an interface to access and manage the children nodes.
  + Implement operations from the interface related to children.
  + Store children’s nodes
  + No conoce las clases concretas de sus hijos
  + Cuando recibe una solicitud, el compound delega el trabajo a sus subelementos, procesa los resultados y devuelve el resultado al cliente
* Client (Powerpoint)
  + Address compound objects through the Component interface.
  + Puede funcionar de la misma manera tanto con elementos simples como complejos

Diagrama

El contenido generado por IA puede ser incorrecto.

#### Implementación

1. Asegúrate de que el modelo central de tu aplicación pueda representarse como una estructura de árbol. Intenta dividirlo en elementos simples y contenedores. Recuerda que los contenedores deben ser capaces de contener tanto elementos simples como otros contenedores.
2. Declara la interfaz componente con una lista de métodos que tengan sentido para componentes simples y complejos.
3. Crea una clase hoja para representar elementos simples. Un programa puede tener varias clases hoja diferentes.
4. Crea una clase contenedora para representar elementos complejos. Incluye un campo matriz en esta clase para almacenar referencias a subelementos. La matriz debe poder almacenar hojas y contenedores, así que asegúrate de declararla con el tipo de la interfaz componente.
5. Al implementar los métodos de la interfaz componente, recuerda que un contenedor debe delegar la mayor parte del trabajo a los subelementos.
6. Por último, define los métodos para añadir y eliminar elementos hijos dentro del contenedor.

Ten en cuenta que estas operaciones se pueden declarar en la interfaz componente.

#### Consecuencias

* Define jerarquías de clases formadas por objetos primitivos y compuestos.
  + Los objetos primitivos pueden ser sustituidos por objetos compuestos por otros objetos primitivos o compuestos.
* Simplifica el cliente
  + Tratan de forma uniforme a todos los componentes de la composición por igual (la misma interfaz)
* Facilita añadir nuevos tipos de componentes
  + Basta con heredar de la clase componente Compuesto u Hoja
* Riesgo de excesiva generalidad en el diseño
  + No es posible restringir el tipo de cierto componentes de un determinado compuesto

# Patrones de comportamiento

Se encargar de la comunicación efectiva y la asignación de responsabilidades entre objetos

* Encapsula “lo que varía”
  + Cuando un comportamiento varía con frecuencia se encapsula un objeto
* Asignación de responsabilidad 🡪 Distribuir el comportamiento
* Comunicación entre instancias
* Se usa más la composición que la herencia

## Command

Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar

#### Purpose

* Necesito invocar operaciones pero…
  + El objeto (Emisor) que crea la petición no es el que lo ejecuta
  + El emisor pasa la solicitud a otro objeto (Receptor)
    - El receptor no necesita saber que operación es
    - Solo sabe como lanzarla
    - No tiene pq ejecutar inmediatamente la operación
  + La misma operación se puede ejecutar en diferentes Receptores
    - Misma opción desde el menú principal y botonera
  + Las operaciones a ejecutar son varias y queremos variar el orden de ejecución (sin recompilar)
* Utilizalo cuando quieras parametrizar objetos con operaciones
* Cuando quieras poner operaciones en cola, programar su ejecución o ejecutarlos de forma remota
* Cuando quieras implementar operaciones reversibles

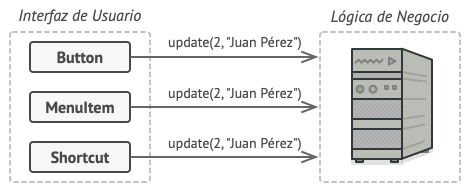
#### Problema

Estás trabajando en una nueva aplicación de edición de texto. Tu tarea actual consiste en crear una barra de herramientas con unos cuantos botones para varias operaciones del editor. Creaste una clase Botón muy limpia que puede utilizarse para los botones de la barra de herramientas y tb para los genéricos en diversos diálogos.

Todos los botones se parecen pero hacen cosas diferentes. Lo más sencillo es crear muchas subclases para cada lugar que se utilice el botón. Estas subclases contendrán el código que deberá ejecutarse con el click en un botón

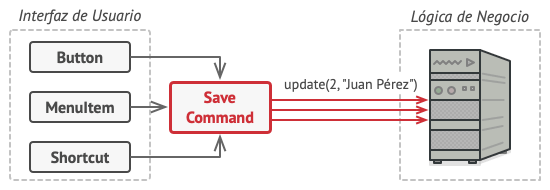
Pronto te das cuenta de que esta solución es muy deficiente. En primer lugar, tienes una enorme cantidad de subclases, lo cual no supondría un problema si no corrieras el riesgo de descomponer el código de esas subclases cada vez que modifiques la clase base Botón

#### Solución

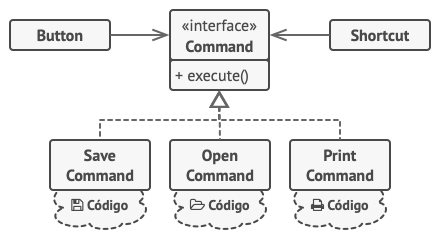
El buen diseño de software a menudo se basa en el principio de separación de responsabilidades, lo que suele tener como resultado la división de la aplicación en capas. El ejemplo más habitual es tener una capa para la interfaz gráfica de usuario (GUI) y otra capa para la lógica de negocio. La capa GUI es responsable de representar una bonita imagen en pantalla, capturar entradas y mostrar resultados de lo que el usuario y la aplicación están haciendo. Sin embargo, cuando se trata de hacer algo importante, como calcular la trayectoria de la luna o componer un informe anual, la capa GUI delega el trabajo a la capa subyacente de la lógica de negocio.

El código puede tener este aspecto: un objeto GUI invoca a un método de un objeto de la lógica de negocio, pasándole algunos argumentos. Este proceso se describe habitualmente como un objeto que envía a otro una solicitud.

El patrón Command sugiere que los objetos GUI no envíen estas solicitudes directamente. En lugar de ello, debes extraer todos los detalles de la solicitud, como el objeto que está siendo invocado, el nombre del método y la lista de argumentos, y ponerlos dentro de una clase comando separada con un único método que activa esta solicitud.



El siguiente paso es hacer que tus comandos implementen la misma interfaz. Normalmente tiene un único método de ejecución que no acepta parámetros. Esta interfaz te permite utilizar varios comandos con el mismo emisor de la solicitud, sin acoplarla a clases concretas de comandos. Adicionalmente, ahora puedes cambiar objetos de comando vinculados al emisor, cambiando efectivamente el comportamiento del emisor durante el tiempo de ejecución.



Regresemos a nuestro editor de textos. Tras aplicar el patrón Command, ya no necesitamos todas esas subclases de botón para implementar varios comportamientos de clic. Basta con colocar un único campo dentro de la clase base Botón que almacene una referencia a un objeto de comando y haga que el botón ejecute ese comando en un clic.

Como resultado, los comandos se convierten en una conveniente capa intermedia que reduce el acoplamiento entre las capas de la GUI y la lógica de negocio. ¡Y esto es tan solo una fracción de las ventajas que ofrece el patrón Command!

#### Ejemplo Real

1. En un restaurante un camarero se acerca y toma tu pedido en un papel
2. Se va a la cocina y pega el pedido a la pared
3. El pedido llega al chef que lo lee y prepara la comida
4. Cocinero coloca la comida en una bandeja junto al pedido
5. Camaero ve la bandeja, lo comprueba con el pedido y lo lleva a tu mesa

El pedido en papel hace la función del comando. Permanece en una cola hasta que el chef está listo para servirlo. Este pedido contiene toda la información relevante para prepara la comida. Permite al chef empezar a cocinar ded inmediato, en vez de correr de un lado a otro aclarando los detalles del pedido directamente contigo

#### Estructura

Diagrama

El contenido generado por IA puede ser incorrecto.

1. Podría haber una clase Invoker entre Cliente y Command que sería responsable de inicializar las solicitudes. Esta clase debe tener un campo para almacenar una referencia a un objeto de comando. El emisor activa este comando en vez de mandrlo directamente al recepto
2. La interfaz Commando normalmente declara un único método para ejecutar el comando
3. Comandos concretos implemntean varios tipos de solicitudes. Un comando concreto no se supone que tenga que realizar el trabajo por su cuenta, sino pasar la llamada a uno de los objetos de la lógica de nogocio. Estas clases se pueden fusionar.
   1. Los parámetros necesarios para ejecutar un método en un objeto receptor pueden declararse como campos en el comando concreto. Puedes hacer inmutables los objetos de comando permitiendo la inicialización de estos campos únicamente a través del constructor
4. La clase Receptora contiene cierta lógica de negocio. Casi cualquier objeto puede actuar como receptor. La mayoría de comandos solo gestiona los detalles sobre cómo se pasa una solicitud al receptor, mientras que el propio receptor hace el trabajo real
5. El cliente crea y configura los objetos de comandos concretos. El cliente debe pasar todos los parámetros de la solicitud, incluyendo la instancia del receptor, dentro del constructor del comando. Después de eso, el cuando resultante puede asociarse con uno o varios emisores

Texto

El contenido generado por IA puede ser incorrecto.



Diagrama

El contenido generado por IA puede ser incorrecto.

1. Declara la interfaz comando con único método de ejecución
2. Extrae solicitudes y ponlas dentro de clases que implementen Comando
   1. Cada clase debe tener un grupo de campos para almacenar los argumentos de las solicitudes junto con referencias al objeto receptor. Todos estos valores deben inicializarse a través del constructor del comando
3. Identifica clases que actúen como emisoras. Añade los campos para almacenar comandos dentro de estas clases
   1. Las emisoras deberán comunicarse con sus comandos tan solo a través de la interfaz de comando. Normalmente las emisoras no crean objetos de comando, los obtienen del código cliente
4. Cambia las emisoras de forma que ejecuten el comando en lugar de enviar directamente una solicitud al receptor
5. Cliente debe inicializar
   1. Receptores
   2. Crea comandos y asócialos a receptores
   3. Crea emisores y asócialos con comandos específicos

#### Alternativas

* Creacer un método deshacer la interfaz
* Puedo crear una pila de las últimas órdenes que se ejecutaron
* Puedo sacar de la pila órdenes ejecutadas y llamar al pétodo deshacer
* Combinando varios command puedes generar una orden Macro
  + Sería útil el Composite

Texto

El contenido generado por IA puede ser incorrecto.

#### Consecuencias

* Command desacopla el objeto que invoca la acción de aquiel que sabe cómo realizarla
* Las órdenes son objetos de primera clase
  + Pueden manipularse y extenderse como otro objeto
* Se pueden crear Macro órdenes combinando varios objetos de tipo Command 🡪 composite
* Es fácil crear nuevas órdenes, sin tocar las clases existentes
  + Herencia

## Observer

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.

* Define una dependencia de no a muchos entre objetos
* All cambiar un objeto automáticamente se notifica a todos los objetos que dependen de él: sus observadores

#### Aplicabilidad

* Una abstracción tiene 2 aspectos y uno depende del otro
  + Facilita que se puedan modificar y reutilizar por separado: desacopla el observador y el observado
* Cuando un cambio en un objeto requiera cambiar a otros, y a priori no sabemos cuentos necesitan cambiarse
* Cuando un objeto debería notificar a otros sin presuponer que objetos concretos son : desacoplarlos
* Cuando algunos objetos de tu aplicación deban observar a otros, pero solo durante un tiempo limitado o en casos específico

#### Problema

Imagina que tienes 2 tipos de objetos: objeto *Cliente*  y un objeto *Tienda*. El cliente está interesado en una marca particular (nuevo modelo de ihone) queestará en la tienda muy pronto

El cliente puede visitar la tienda cada día para comprobar la disponibilidad del producto. Pero mientras eel producto esté en camino, la mayoría de los viajes no servirán.

Por otro lado, la tienda podría enviar cientos de correos (lo cual se podría considerar spam) a todos los clientes cada vez que hay un nuevo producto disponible. Esto ahorraría a los clientes los interminables viajes a la tienda, pero, al mismo tiempo, molestaría a otros clientes que no están interesados en los nuevos productos.

#### Métodos de notificaciónSolución

El objeto que tiene un estado interesante suele denominarse sujeto, pero, como también va a notificar a otros objetos los cambios en su estado, le llamaremos notificador (en ocasiones también llamado publicador). El resto de los objetos que quieren conocer los cambios en el estado del notificador, se denominan suscriptores.

El patrón Observer sugiere que añadas un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de eventos que proviene de esa notificadora. ¡No temas! No es tan complicado como parece. En realidad, este mecanismo consiste en: 1) un campo matriz para almacenar una lista de referencias a objetos suscriptores y 2) varios métodos públicos que permiten añadir suscriptores y eliminarlos de esa lista.

Ahora, cuando le sucede un evento importante al notificador, recorre sus suscriptores y llama al método de notificación específico de sus objetos.

Las aplicaciones reales pueden tener decenas de clases suscriptoras diferentes interesadas en seguir los eventos de la misma clase notificadora. No querrás acoplar la notificadora a todas esas clases. Además, puede que no conozcas algunas de ellas de antemano si se supone que otras personas pueden utilizar tu clase notificadora.

Por eso es fundamental que todos los suscriptores implementen la misma interfaz y que el notificador únicamente se comunique con ellos a través de esa interfaz. Esta interfaz debe declarar el método de notificación junto con un grupo de parámetros que el notificador puede utilizar para pasar cierta información contextual con la notificación.

Si tu aplicación tiene varios tipos diferentes de notificadores y quieres hacer a tus suscriptores compatibles con todos ellos, puedes ir más allá y hacer que todos los notificadores sigan la misma interfaz. Esta interfaz sólo tendrá que describir algunos métodos de suscripción. La interfaz permitirá a los suscriptores observar los estados de los notificadores sin acoplarse a sus clases concretas.

#### Mundo real

Si te suscribes a un periódico o una revista, ya no necesitarás ir a la tienda a comprobar si el siguiente número está disponible. En lugar de eso, el notificador envía nuevos números directamente a tu buzón justo después de la publicación, o incluso antes.

El notificador mantiene una lista de suscriptores y sabe qué revistas les interesan. Los suscriptores pueden abandonar la lista en cualquier momento si quieren que el notificador deje de enviarles nuevos números.

#### Estructura

Diagrama

El contenido generado por IA puede ser incorrecto.

1. Notificador (Sujeto) envía eventos de interés a otros objetos. Esos eventos ocurren cuando el notificador cambia su estado o ejecuta algunos comportamientos. Los notificadores contienen una infraestructura de suscripción que permite a nuevos y antiguos suscriptores abandonar en la lista
2. Cuando un nuevo evento, el notificador recorre la lista de suscripción e invoca el método de notificación declarado en la interfaz suscriptora en cada objeto suscriptor
3. La interfaz Suscriptora (Observador) declara la interfaz de notificación. En la mayoría de los casos, consiste en un único método actualizar. El método puede tener varios parámetros que permitan al notificador pasar algunos detalles del evento junto a la actualización.
4. Los Suscriptores Concretos realizan algunas acciones en respuesta a las notificaciones emitidas por el notificador. Todas estas clases deben implementar la misma interfaz de forma que el notificador no esté acoplado a clases concretas.
   1. Normalmente, los suscriptores necesitan cierta información contextual para manejar correctamente la actualización. Por este motivo, a menudo los notificadores pasan cierta información de contexto como argumentos del método de notificación. El notificador puede pasarse a sí mismo como argumento, dejando que los suscriptores extraigan la información necesaria directamente.
5. Normalmente, los suscriptores necesitan cierta información contextual para manejar correctamente la actualización. Por este motivo, a menudo los notificadores pasan cierta información de contexto como argumentos del método de notificación. El notificador puede pasarse a sí mismo como argumento, dejando que los suscriptores extraigan la información necesaria directamente.

#### Implementación

1. Repasa tu lógica de negocio e intenta dividirla en dos partes: la funcionalidad central, independiente del resto de código, actuará como notificador; el resto se convertirá en un grupo de clases suscriptoras.
2. Declara la interfaz suscriptora. Como mínimo, deberá declarar un único método actualizar.
3. Declara la interfaz notificadora y describe un par de métodos para añadir y eliminar de la lista un objeto suscriptor. Recuerda que los notificadores deben trabajar con suscriptores únicamente a través de la interfaz suscriptora.
4. Decide dónde colocar la lista de suscripción y la implementación de métodos de suscripción. Normalmente, este código tiene el mismo aspecto para todos los tipos de notificadores, por lo que el lugar obvio para colocarlo es en una clase abstracta derivada directamente de la interfaz notificadora. Los notificadores concretos extienden esa clase, heredando el comportamiento de suscripción.
5. Sin embargo, si estás aplicando el patrón a una jerarquía de clases existentes, considera una solución basada en la composición: coloca la lógica de la suscripción en un objeto separado y haz que todos los notificadores reales la utilicen.
6. Crea clases notificadoras concretas. Cada vez que suceda algo importante dentro de una notificadora, deberá notificar a todos sus suscriptores.
7. Implementa los métodos de notificación de actualizaciones en clases suscriptoras concretas. La mayoría de las suscriptoras necesitarán cierta información de contexto sobre el evento, que puede pasarse como argumento del método de notificación.
8. Pero hay otra opción. Al recibir una notificación, el suscriptor puede extraer la información directamente de ella. En este caso, el notificador debe pasarse a sí mismo a través del método de actualización. La opción menos flexible es vincular un notificador con el suscriptor de forma permanente a través del constructor.
9. El cliente debe crear todos los suscriptores necesarios y registrarlos con los notificadores adecuados.

#### Consecuencias

* Observer permite modificar los sujetos y sus observadores de forma independiente
  + Podemos añadir Observadores sin modificar el Sujeto.
  + Podemos reutilizar un mismo Observador con otros Sujetos y viceversa.
* Comunicación por difusión
  + Al Sujeto le da igual cuantos o qué Observadores tiene a la horade informar de cambios.
* Actualizaciones inesperadas
  + Los Observadores pueden modificar al Sujeto.
  + Una pequeña modificación puede disparar la “difusión” delcambio a múltiples Observadores, provocando una reacción en cadena con un alto coste computacional
* Sujetos complejos complican identificar “qué” cambió en ellos
  + Sujetos complejos complican identificar “qué” cambió en ellos

## State

Permite a un objeto alterar su comportamiento cuando su estado interno cambia: máquina de estados.

#### Aplicability

La complejidad de las transiciones hace inmanejable los cambios mediante sentencias if-then-else

No solo el comportamiento cambia con cada estado sino que las transiciones implican realizar acciones específicas

Parece como si el objeto cambiara de su clase

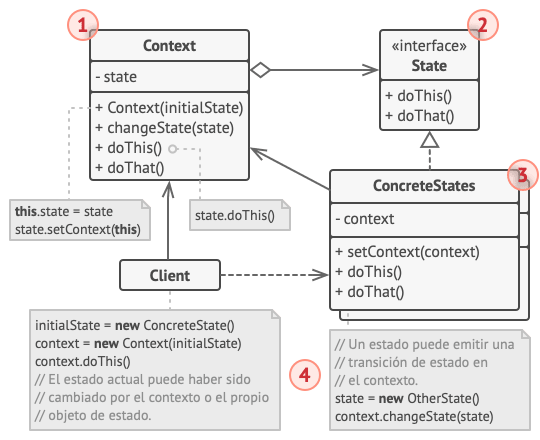
Diagrama

El contenido generado por IA puede ser incorrecto.

#### Structure

Diagrama

El contenido generado por IA puede ser incorrecto.



1. Clase contexto almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (setter) para pasarle un nuevo objeto de estado
2. La interfaz Estado declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.
3. Los Estados Concretos proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.
4. Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.

Texto

El contenido generado por IA puede ser incorrecto.

Diagrama

El contenido generado por IA puede ser incorrecto.

#### Consecuencias

* Hace explícitas las transiciones entre estados
  + Cada estado es un objeto
  + Cada estado define sus transiciones a otros estados
  + Robustez: Los cambios son atómicos (indivisibles)
    - Ahora el Contexto sólo cambia una variable: su Estado
  + Los objetos Estado pueden compartirse
    - Cuando todos los estados no tienen variables internas, puede compartirse un mismo objeto estado por varios contextos
  + Los objetos Estado pueden ser únicos
    - En lugar de generar uno nuevo cada vez se reutiliza el mismo objeto (Singleton)