# Guía de Patrones de Diseño

*Documento de cátedra*

Profesor: Luciano Straccia

Fecha de actualización: 29/07/2018

# TABLA DE CONTENIDOS

# CATÁLOGO DE PATRONES

## Catálogo completo

En el siguiente cuadro puede hallarse el catálogo de patrones de diseño completo.

| *Ámbito* | *Creación* | *Estructural* | *Comportamiento* |
|---|---|---|---|
| Clase | Factory Method | Adapter | Interpreter<br>Template Method |
| Objeto | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

## Patrones para Diseño de Sistemas UTN

Los patrones de diseño que se encuentran en el siguiente listado son los considerados durante la cursada de Diseño de Sistemas. Esto forma parte de la decisión de la cátedra en su conjunto de priorizar los patrones asociados al diseño, dejando de lado aquellos más cercanos a la programación.

| *Creación* | *Estructural* | *Comportamiento* |
|---|---|---|
| Singleton<br>Prototype | Adapter<br>Composite<br>Decorator | Template Method<br>Command<br>Observer<br>State<br>Strategy<br>Visitor |

## Ejemplos de Implementación

Pueden hallarse ejemplos de patrones de diseño en los siguientes links:

NET: http://www.dofactory.com/net/design-patterns

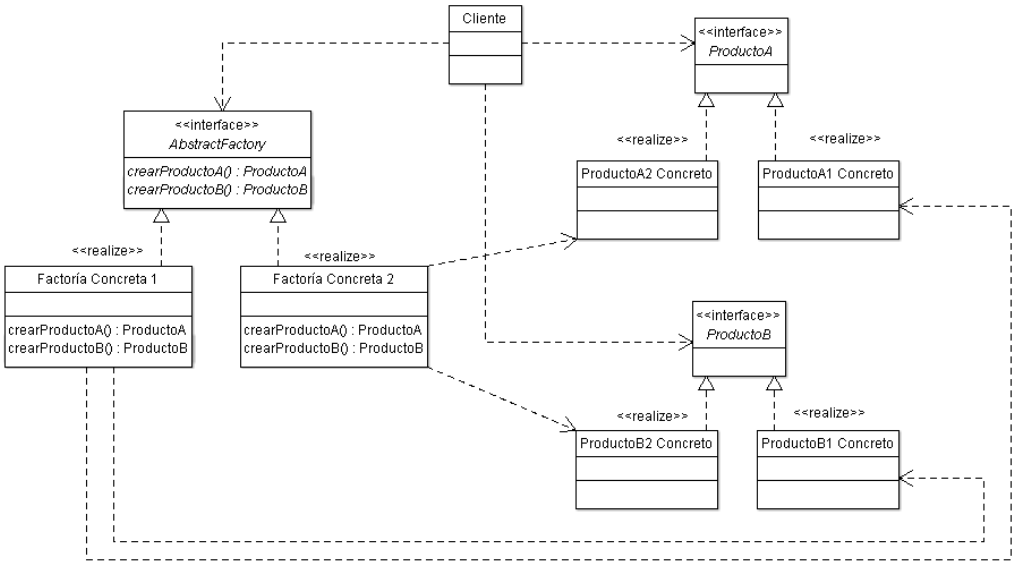
Java: http://www.arquitecturajava.com/categoria/arquitectura/design-patterns/

# ABSTRACT FACTORY

## Objetivo

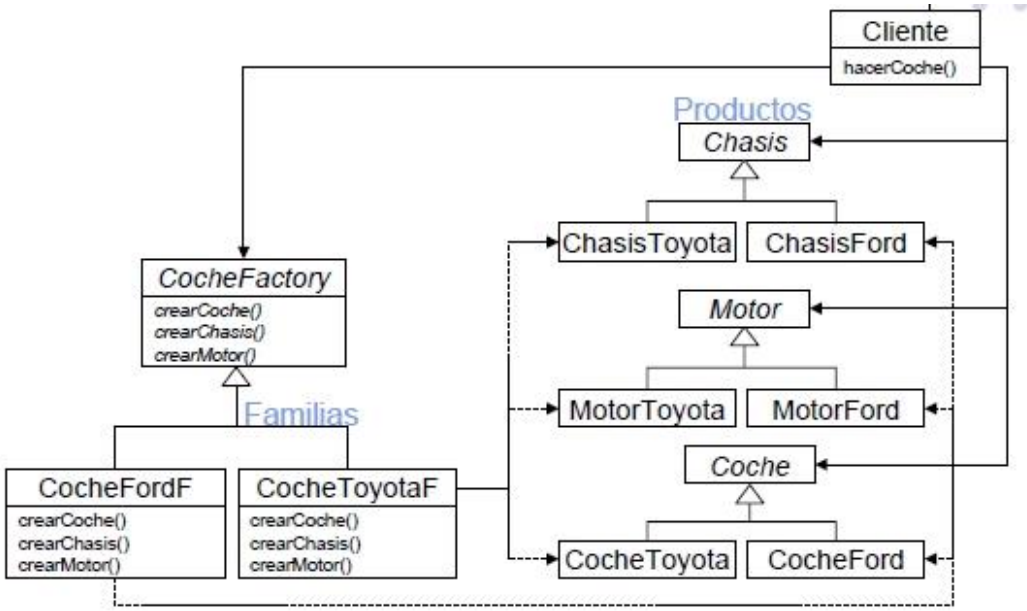
- Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas

## Estructura




## Ejemplo con implementación

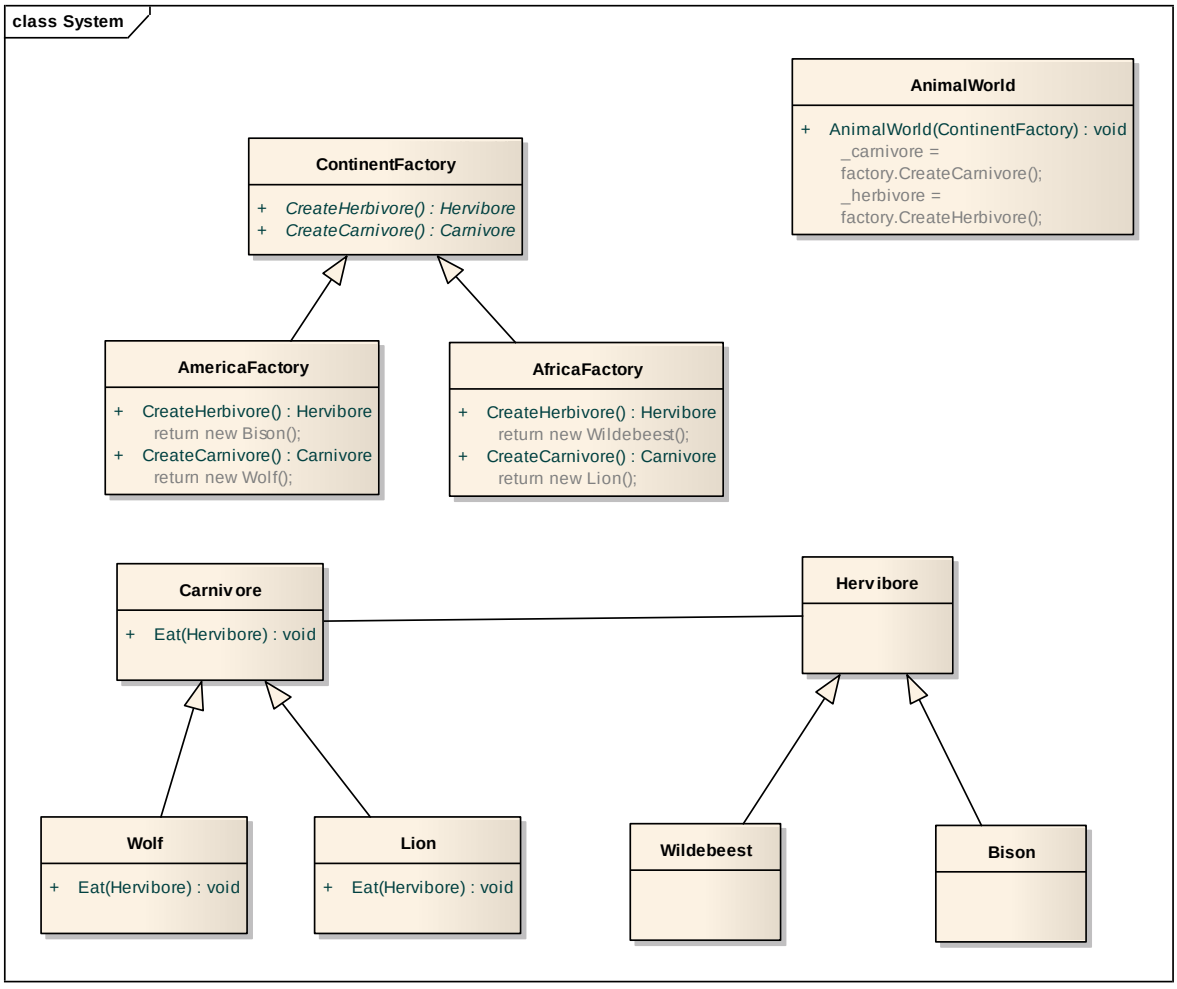
Construir una aplicación para construir coches mediante el ensamblado de sus partes (motor, chasis, etc.). Los componentes de un coche deben tener la misma marca del coche. Hay múltiples marcas. Es responsabilidad del cliente ensamblar las piezas.




Puede hallarse el código fuente en el repositorio de código.

## Ejemplo 2 con implementación

Para un videojuego necesitamos crear personajes según el continente que tiene asignado el jugador. Los personajes son diferentes animales carnívoros y herbívoros. En América hay bisontes (herbívoros) y lobos (carnívoros). En África hay antílopes denominados ñu (herbívoro) y leones (carnívoros). Tener en cuenta que se podrían agregar o modificar continentes y animales. Necesitamos que una clase Cliente pueda crear todos los animales que corresponden a un continente de manera atómica.




Puede hallarse el código fuente en el repositorio de código.

## Abstract Factory y Factory Method

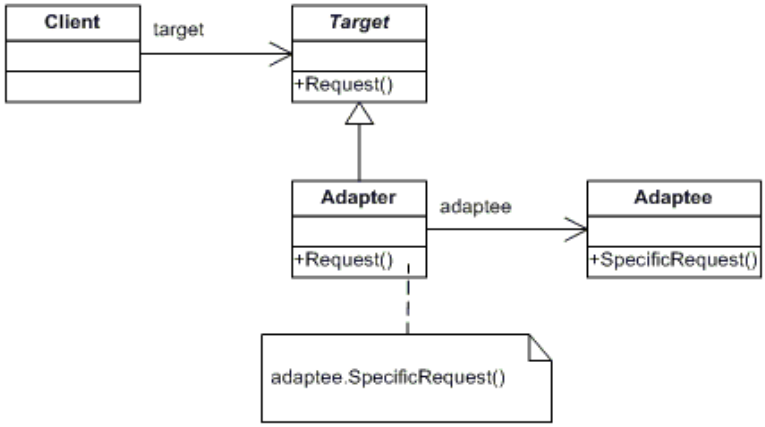
Abstract Factory generalmente se implementa utilizando Factory Method y por tanto provee al menos toda la flexibilidad de éste. La diferencia principal entre ambos es que el primero trata con familias de productos, mientras que el otro se preocupa por un único producto.

# ADAPTER

## Objetivo

- Convierte la interface de una clase en otra interface que es la que esperan los clientes. Permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles

## Estructura

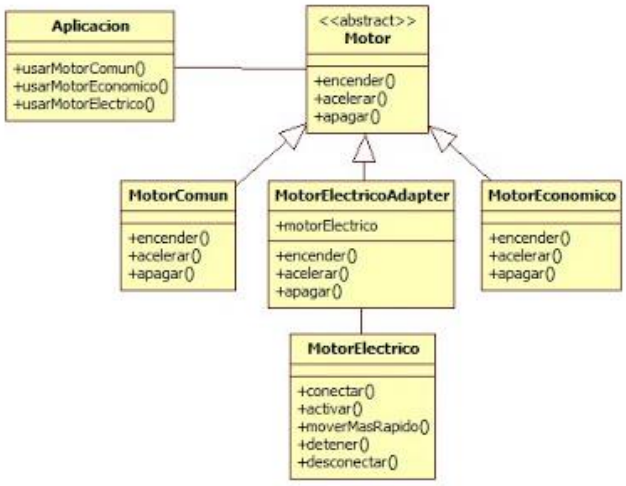



## Ejemplo

Tenemos tres tipos de motores para vehículos: común, eléctrico y económico. Supongamos que los motores comunes y económicos tienen funciones básicas: encender, acelerar, apagar.

Pero al querer incorporar a los eléctricos, tienen diferentes métodos: conectar, activar, moverMasRapido, detener y desconectar. Según los términos originales, encender equivale a conectar y activar; acelerar, a mover más rápido; y apagar a detener y desconectar.

No podríamos hacer heredar al motor eléctrico de la clase Motor. Excepto que utilicemos un adaptador que vincule los métodos de la superclase con los métodos de la clase que no es factible hacer heredar. En este caso podríamos poner una clase Adaptadora entre Motor y MotorEléctrico.




Aquí se puede hallar el código de la clase Adapter en Lenguaje Java:

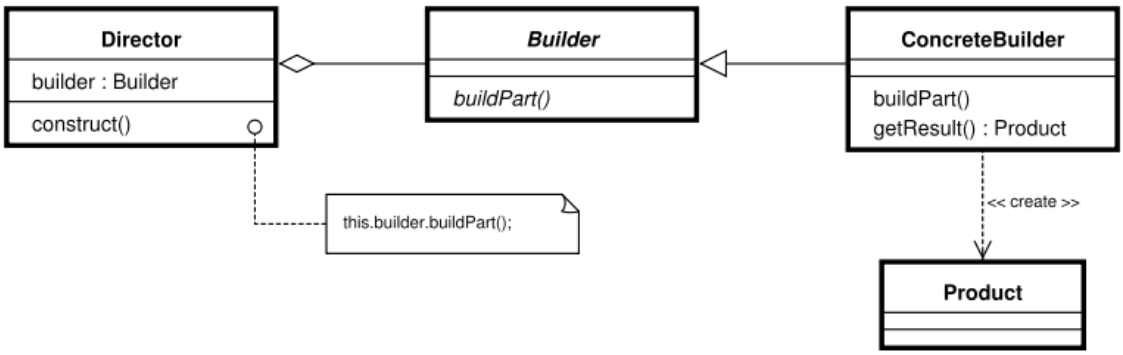
```
public class MotorElectricoAdapter extends Motor{
    private MotorElectrico motorElectrico;

    public MotorElectricoAdapter(){
        super();
        this.motorElectrico = new MotorElectrico();
        System.out.println("Creando motor Electrico adapter");
    }
    @Override
    public void encender() {
        System.out.println("Encendiendo motorElectricoAdapter");
        this.motorElectrico.conectar();
        this.motorElectrico.activar();
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando motor electrico...");
        this.motorElectrico.moverMasRapido();
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor electrico");
        this.motorElectrico.detener();
        this.motorElectrico.desconectar();
    }
}
```

# BUILDER

## Objetivo

- Permite la creación de un objeto complejo que se compone de una variedad de partes que contribuyen individualmente a la creación del objeto complejo
- Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto

## Estructura

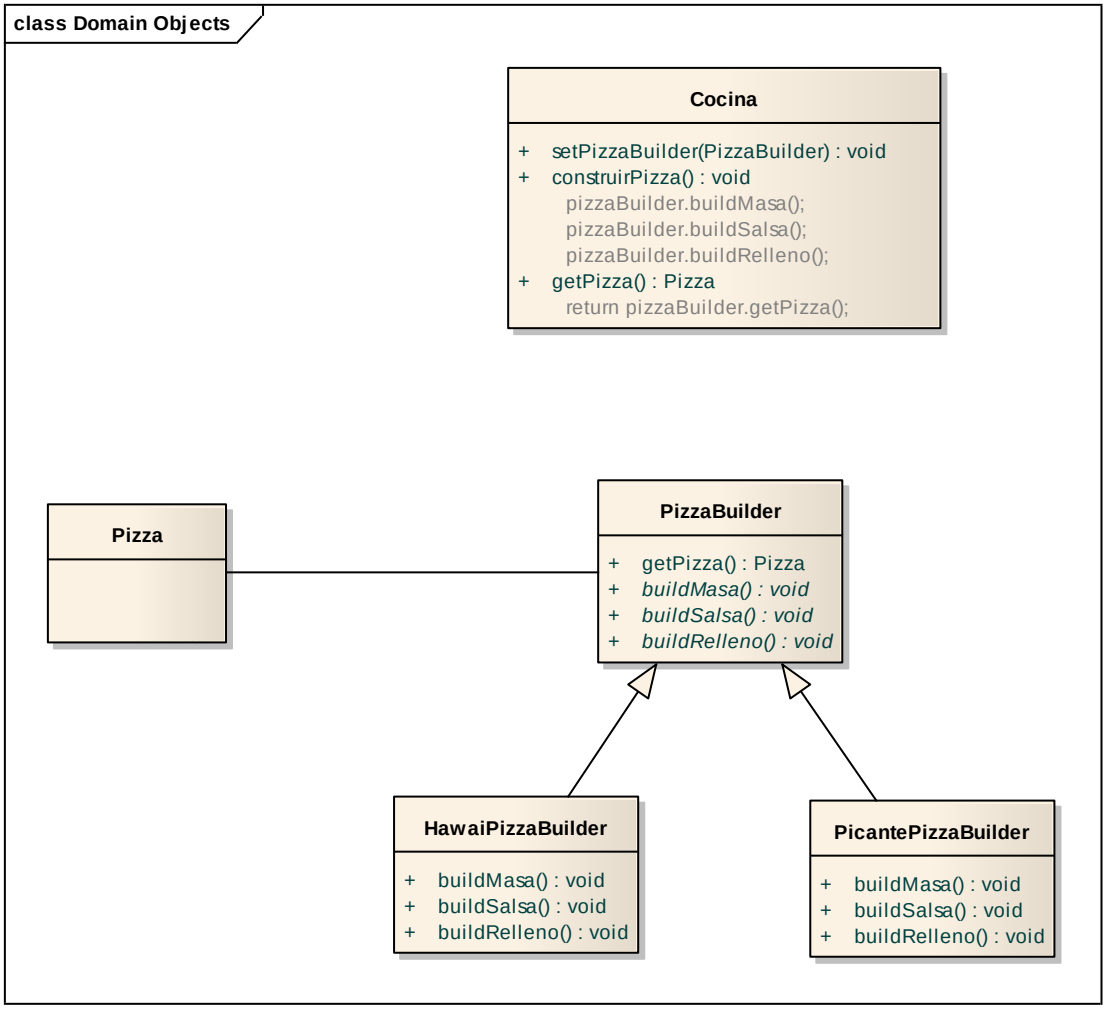



## Abstract Factory y Builder

- Builder se enfoca en construir un objeto complejo paso a paso. Mientras que Abstract Factory en una familia de productos (no un único objeto con un proceso de construcción a piezas o pasos).
- Generalmente Builder construye un objeto compuesto.

## Ejemplo con implementación

Tenemos una cocina que produce pizzas. La producción de una pizza involucra diferentes pasos: su masa, su salsa, su relleno. Cada uno de estos pasos difiere según el tipo de pizza. Y queremos que esto sea transparente al Cliente que consume estas clases. Entonces crearemos un Builder. El Cliente deberá setear cuál builder concreto utilizar, deberá ejecutar el paso de construir la pizza y luego utiliza la pizza construida. Así el Cliente tiene un solo punto de entrada a la factoría sin importar los detalles de su implementación.

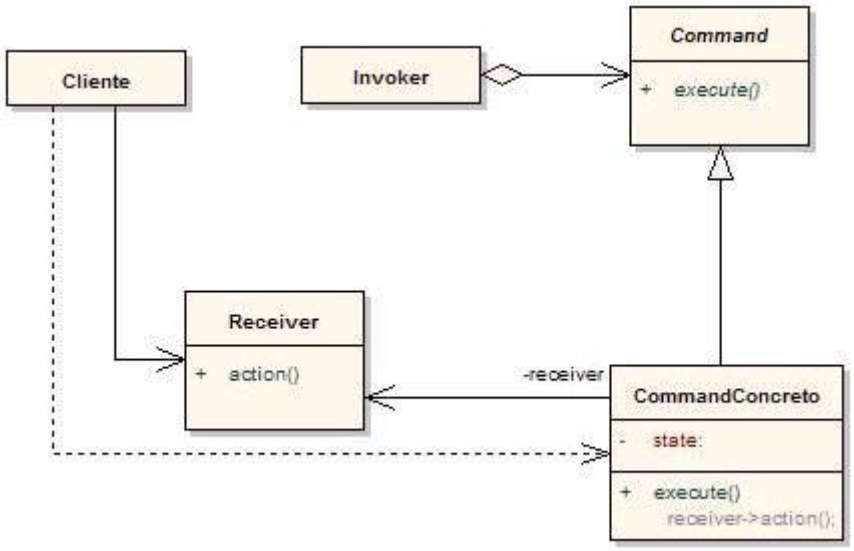


# COMMAND

## Objetivo

- Ofrece una interfaz común que permita invocar las acciones de forma uniforme y extender el sistema con nuevas acciones de forma más sencilla.
- Permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma.

## Estructura



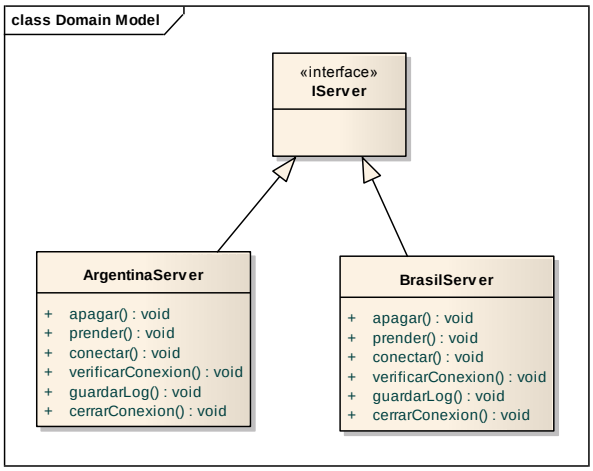
## Ejemplo

Una empresa maneja varios servidores y cada uno de ellos deben correr diversos procesos, como apagarse, prenderse, etc. Cada uno de estos procesos, a su vez, implica pequeños pasos como, por ejemplo, realizar una conexión a dicho servidor, guardar los datos en un log, etc.

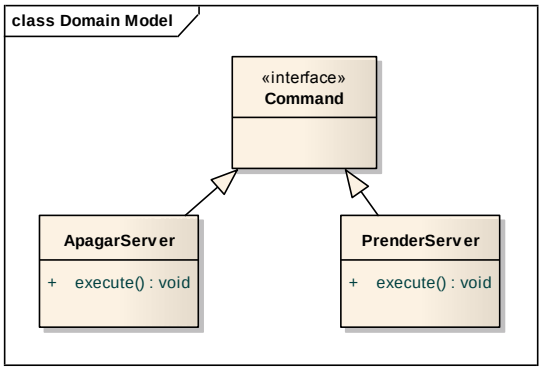
Apagar un servidor implica: conectar, verificar la conexión, guardar un log, apagar y cerrar la conexión.

Prender un servidor implica: conectar, verificar la conexión, prender, guardar un log y cerrar la conexión.

Cada servidor debe implementar cada una de las acciones (las llamables apagar y prender, pero también todos los pasos intermedios como verificar la conexión, guardar el log, cerrar la conexión, etc.). Cada servidor realiza cada uno de estos pasos de forma diferente, por lo cual se realiza una clase para cada servidor donde se implementa cómo realizar cada paso.




Además debemos armar los Command concretos, o sea, las acciones concretas que son invocables. En nuestro ejemplo son prender y apagar (las restantes acciones son sólo pasos de estas acciones invocables). Las clases del Command tendrán una referencia al Server concreto al momento de crearse, de esta forma se indicará sobre cuál server realizar las acciones.



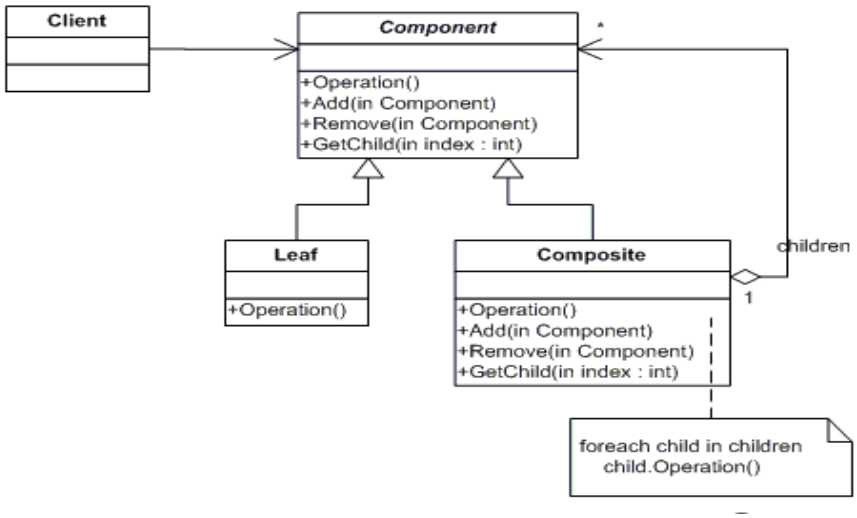

Finalmente el Invoker contiene el código que integra ambas estructuras y que es la base del Command.

# COMPOSITE

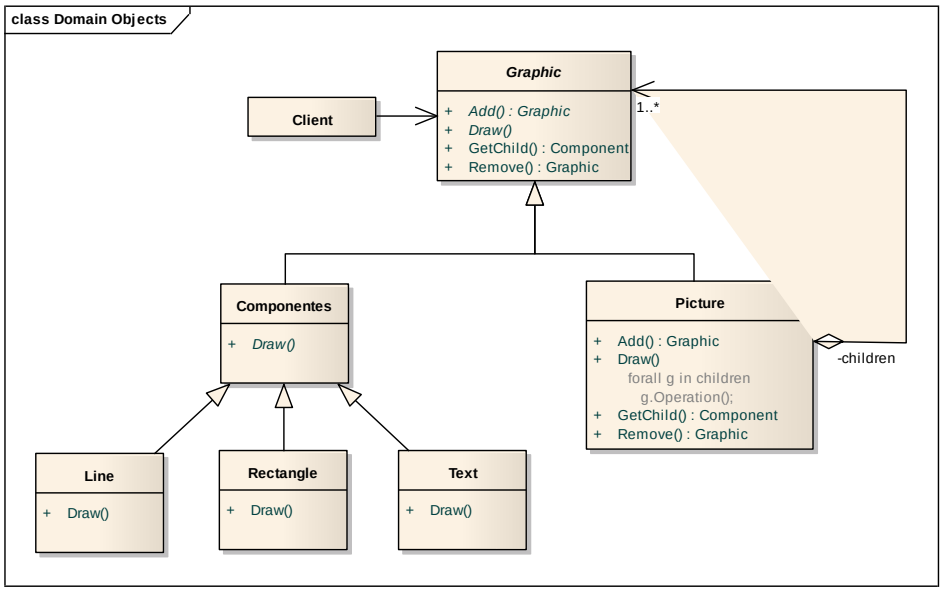
## Objetivo

- Permite componer objetos dentro de estructuras de árboles para representar jerarquías del tipo de "todo-parte". Les permite a los clientes tratar objetos individuales y objetos componentes de una única manera o forma.
- Sirve para construir algoritmos u objetos complejos a partir de otros más simples gracias a la composición y a una estructura de árbol.
- Al poseer todos los objetos una interfaz común se tratan todos de la misma manera.

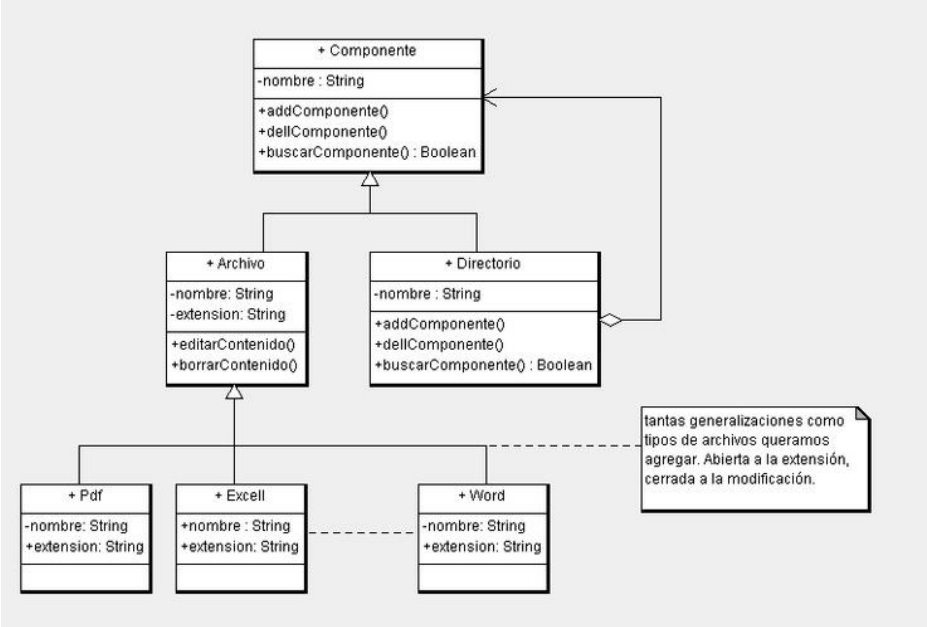
## Estructura



## Ejemplo 1

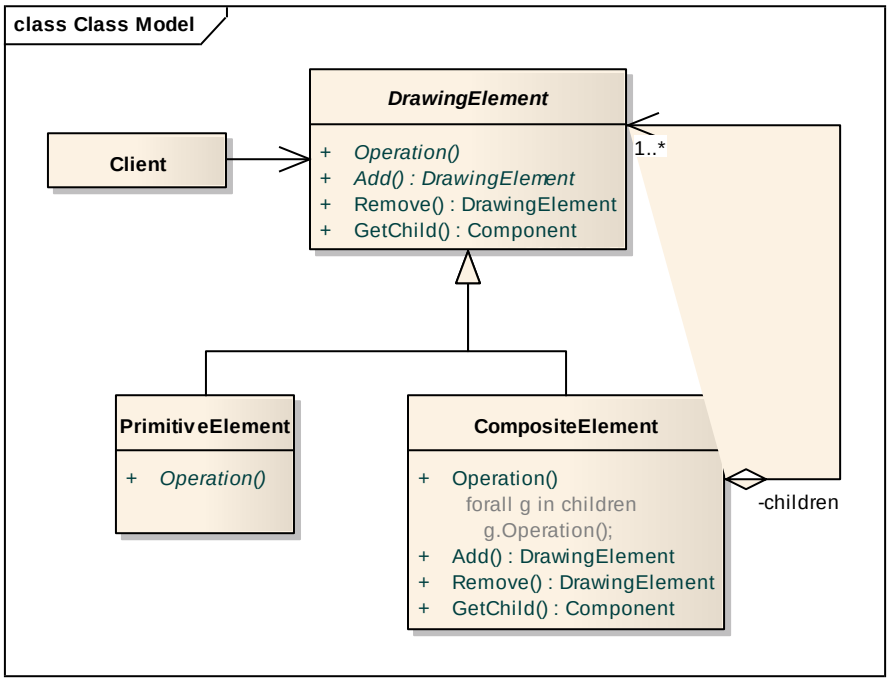



## Ejemplo 2



## Ejemplo 3 con implementación

En este ejemplo se utiliza al Patrón Composite en la construcción de una estructura de árbol gráfica compuesta de nodos primitivos (líneas, círculos, etc) y nodos compuestos (grupos de elementos de dibujo que forman elementos más complejos).




Puede hallarse la implementación en código C# de este ejemplo en http://www.dofactory.com

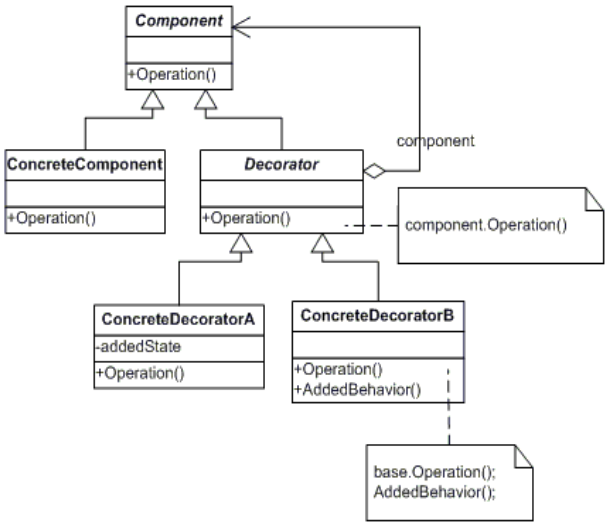
También ha sido creado el proyecto en el Repositorio de código fuente de la cursada

# DECORATOR

## Objetivo

- Asigna responsabilidades adicionales a un objeto en forma dinámica, proporcionando una alternativa flexible a la herencia para extender la funcionalidad
- Se utiliza cuando queremos añadir responsabilidad a objetos individuales en vez de a toda una clase.

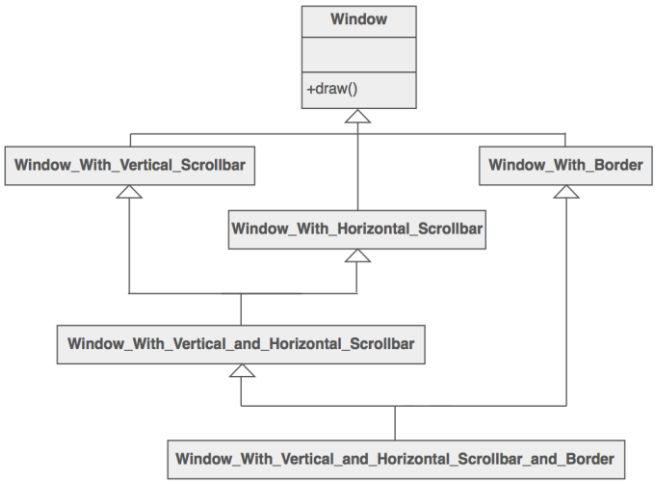
## Estructura



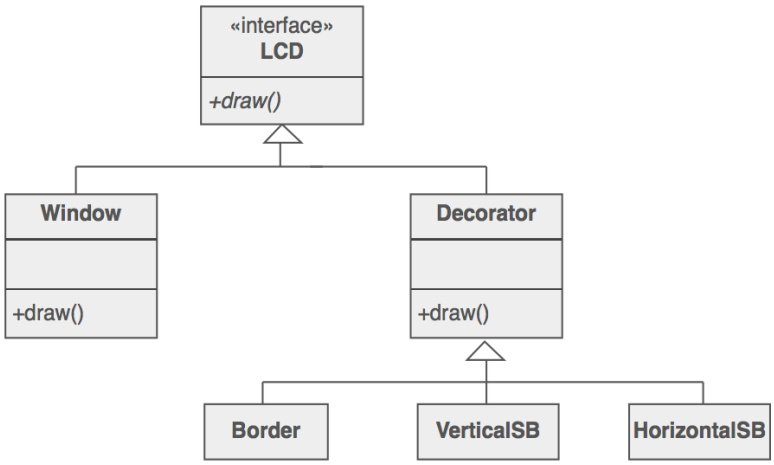
## Ejemplo

El siguiente ejemplo es sobre una pantalla a la cual se le puede agregar scrollbar y bordes.

Resolución sin decorador:



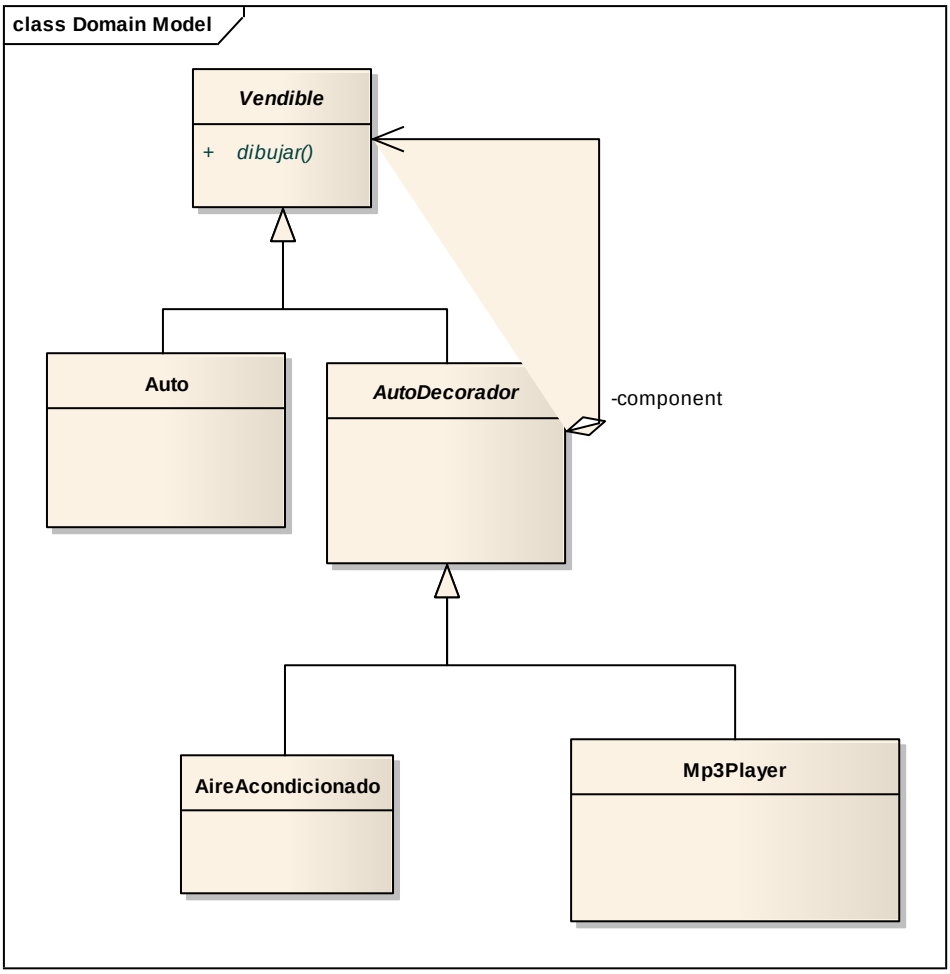

Con decorator:



## Ejemplo con implementación

Imaginemos que vendemos automóviles y el cliente puede opcionalmente adicionar ciertos componentes (aire acondicionado, mp3 player, etc). Por cada componente que se adiciona, el precio varía.

Puede hallarse la implementación en código C# en el Repositorio de código fuente de la cursada

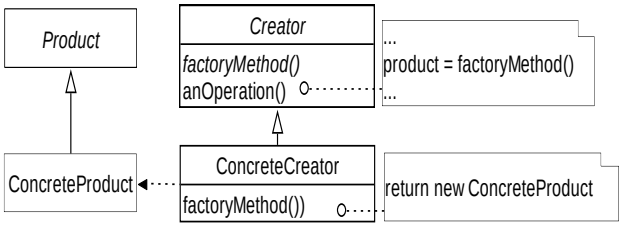


# FACTORY METHOD

## Objetivo

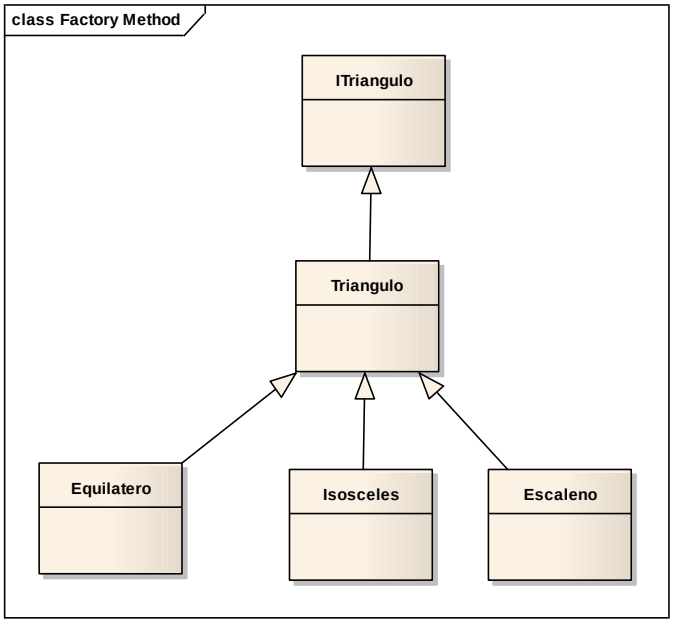
- Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar
- Permite que una clase delegue en sus subclases la creación de objetos

## Estructura




## Ejemplo con implementación

Imaginemos que tenemos que permitir a un cliente crear triángulos (equiláteros, escalenos e isósceles).




Puede hallarse la implementación en código C# en el Repositorio de código fuente de la cursada

## Abstract Factory y Factory Method

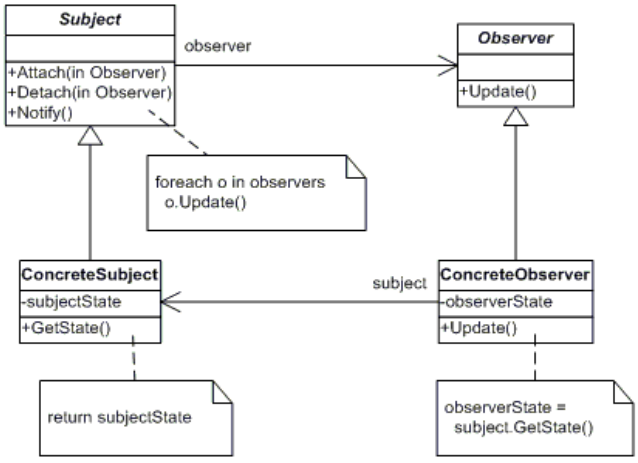
Abstract Factory generalmente se implementa utilizando Factory Method y por tanto provee al menos toda la flexibilidad de éste. La diferencia principal entre ambos es que el primero trata con familias de productos, mientras que el otro se preocupa por un único producto.

# OBSERVER

## Objetivo

- Define una dependencia entre objetos, de manera que cuando un objeto cambie su estado, se informe y se actualicen automáticamente los demás objetos que dependen de él
- Este patrón de diseño permite reaccionar a ciertas clases llamadas observadores sobre un evento determinado.
- Un cambio en un objeto implica un cambio en otro. Por ejemplo, si cambia el valor de la bolsa entonces cambia el valor del trigo.

## Estructura



## Observaciones

La única cosa que el objeto Sujeto conoce en relación al objeto observador, es que este implementa una determinada interface.

El objeto Sujeto es único, y cambia de estado cada vez que recibe una nueva información sobre los cambios de temperatura, humedad y presión y debe remitir este objeto a Observadores que generalmente son de estructura diferente, por lo tanto la información debe mandarse a una interface que los nuclee.

Se puede agregar nuevos objetos Observadores en cualquier momento, ya que el objeto Sujeto mantiene una lista de que implementa la interface Observer, pudiéndose agregar todos los objetos Observadores que se deseen. Lo mismo sucede con los objetos Observadores que necesitan ser dados de baja.

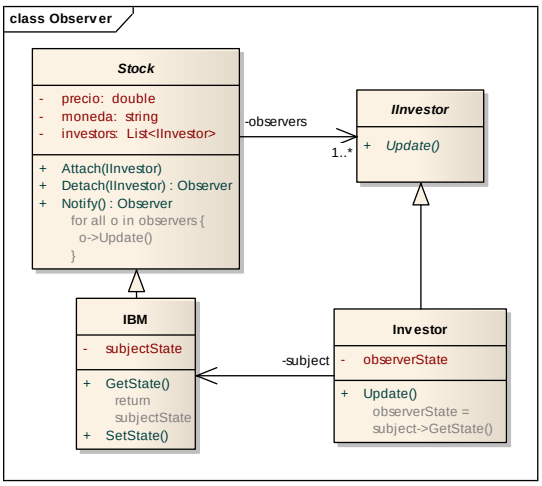
Los Observadores deben requerir ser dados de baja en la lista del Sujeto y también cuando en ella no están, y quieren estar, deben requerir al Sujeto que los incorpore.

## Ejemplo 1

Este ejemplo muestra un caso en que los inversores registrados son notificados cada vez que un valor cambia de valores.

Puede hallarse la implementación en código C# de este ejemplo en http://www.dofactory.com

También ha sido creado el proyecto en el Repositorio de código fuente de la cursada

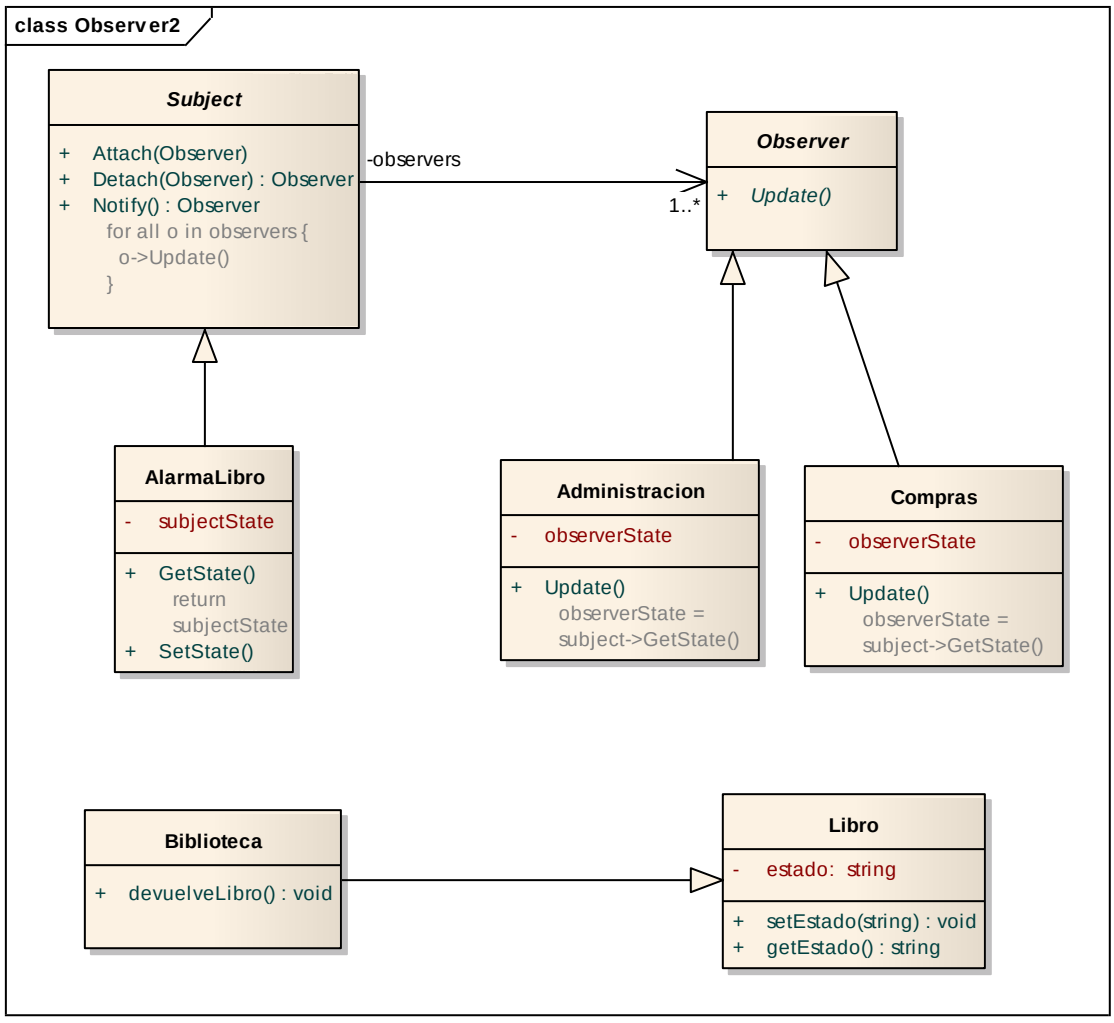


## Ejemplo 2

Vamos a suponer un ejemplo de una Biblioteca, donde cada vez que un lector devuelve un libro se ejecuta el método devuelveLibro(Libro libro) de la clase Biblioteca.

Si el lector devolvió el libro dañado entonces la aplicación avisa a ciertas clases que están interesadas en conocer este evento.

Puede hallarse la implementación en código C# en el Repositorio de código fuente de la cursada.

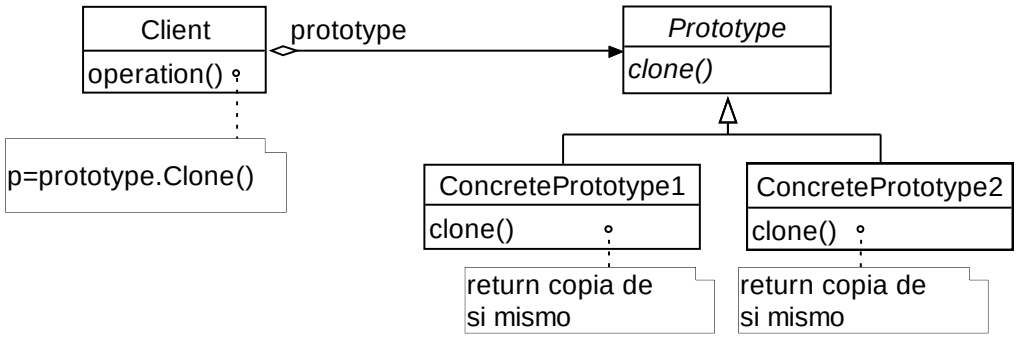


# PROTOTYPE

## Objetivo

- Crea nuevos objetos clonando una instancia previamente creada

## Estructura




## Observaciones

Desde la vista del diseño permite que el consumidor de la clase (*cliente/client)* no necesite conocer los atributos de una determinada instancia, sino sólo invoque su clonación

Existen lenguajes de programación que poseen el método clone(), como por ejemplo en Java donde se hereda de la clase Object, por lo cual ni siquiera hay que construir el método de clonación, sino invocar a este método heredado

## Ejemplo

Suponiendo que se esté desarrollando un juego interactivo que permite al usuario interactuar con personajes que juegan ciertos roles. Se desea incorporar al juego una facilidad para crear nuevos personajes que se añaden al conjunto de personajes predefinidos. En el juego, todos los personajes serán instancias de un pequeño conjunto de clases tales como Heroe, Villano, Principe o Monstruo. Cada clase tiene una serie de atributos como nombre, imagen, altura, peso, inteligencia, habilidades, etc. y según sus valores, una instancia de la clase representa a un personaje u otro, por ejemplo podemos tener los personajes príncipe bobo o un príncipe listo, o monstruo bueno o monstruo malo. Diseña una solución basada en patrones que permita al usuario crear nuevos personajes y seleccionar para cada sesión del juego personajes de una colección de personajes creados. Escribe el código de los métodos del catálogo de personajes que permiten añadir un nuevo personaje a la colección y seleccionar un personaje para jugar.

## Implementación

Se muestra la implementación de la función Clone en .NET

```
public override Prototype Clone()
{
  return (Prototype)this.MemberwiseClone();
}
```

El método Memberwise copia el objeto. Si no existiera este método deberíamos crear un nuevo objeto y copiar cada uno de los valores de sus atributos.

Pueden hallar en Wikipedia ejemplos de código para diversos lenguajes de programación:

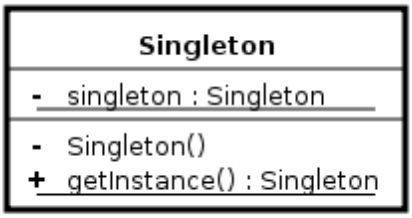
https://es.wikipedia.org/wiki/Prototype_(patr%C3%B3n_de_dise%C3%B1o)

# SINGLETON

## Objetivo

- Garantiza que una clase sólo tenga una instancia y proporciona un punto de acceso global a ella
- No permite instanciar 2 objetos de la misma clase. Por ejemplo, una clase de configuraciones del software.

## Estructura




## Implementación

```
public class Singleton
{
   private static Singleton instance = null;

   private Singleton() {}

   public static Singleton GetInstance()
   {
     if (instance == null)
        instance = new Singleton();

     return instance;
   }
}
```

## Singleton vs. Clases Estáticas

Una clase estática es aquella que no es instanciable, no contiene atributos y sus métodos son accedidos directamente con el nombre de la instancia. Es una clase que no crea objetos. No responde a los conceptos de la orientación a objetos, aunque es una práctica habitual. Además una clase estática es de tipo sealed y no puede heredarse.

Una clase Singleton es aquella que sí es instanciable, se puede crear 1 objeto y sólo 1. Puede contener atributos.

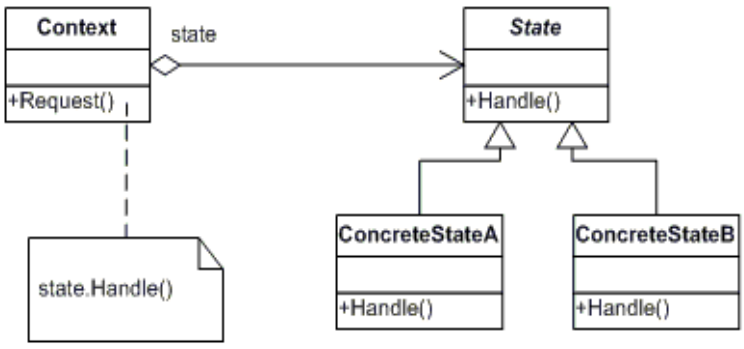
La diferencia principal radica en el manejo de atributos por parte de la clase Singleton y la posibilidad de crear clases que hereden de ella, lo cual no lo permite la clase estática. Además Singleton respeta conceptual al paradigma orientado a objetos. El uso de clases estáticas debiera reservarse a ciertas situaciones especiales.

# STATE

## Objetivo

- Que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.
- Desacoplar el estado de la clase en cuestión
- Implementar distintos métodos (o distintos comportamientos) según un estado de un atributo
- Cada subclase implementa el comportamiento asociado con un estado del contexto.

## Estructura

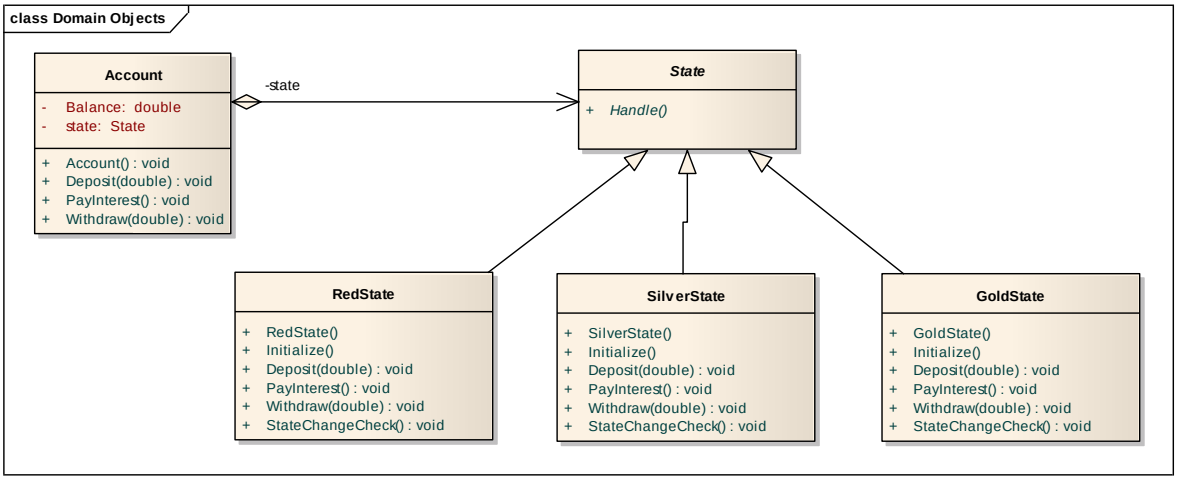


## Ejemplo

Puede hallarse la implementación en código C# de este ejemplo en http://www.dofactory.com

También ha sido creado el proyecto en el Repositorio de código fuente de la cursada

Una cuenta bancaria se comporta de manera diferente dependiendo de su balance. La diferencia en el comportamiento se delega en objetos de estados denominados RedState, SilverState y GoldState. Estos estados representan cuentas descubiertas, cuentas iniciales y cuentas en buen estado.



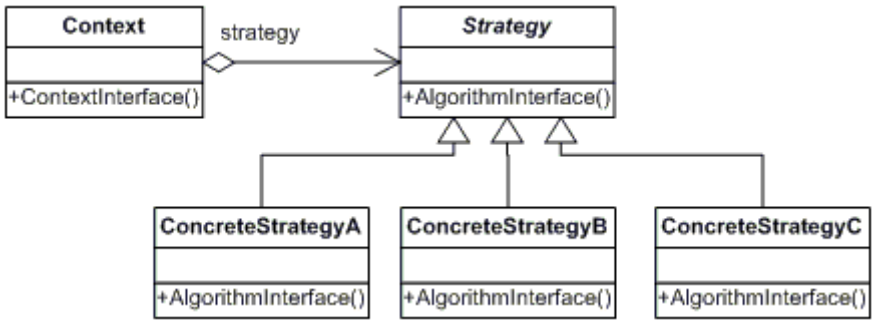

*Si no se aplicara el patrón State todo el comportamiento estaría sobrecargado sobre la cuenta Account.*

# STRATEGY

## Objetivo

- Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que los usan
- En base a un parámetro, que puede ser cualquier objeto, permite a una aplicación decidir en tiempo de ejecución el algoritmo que debe ejecutar.
- Implementar igual método con distintas estrategias

## Estructura

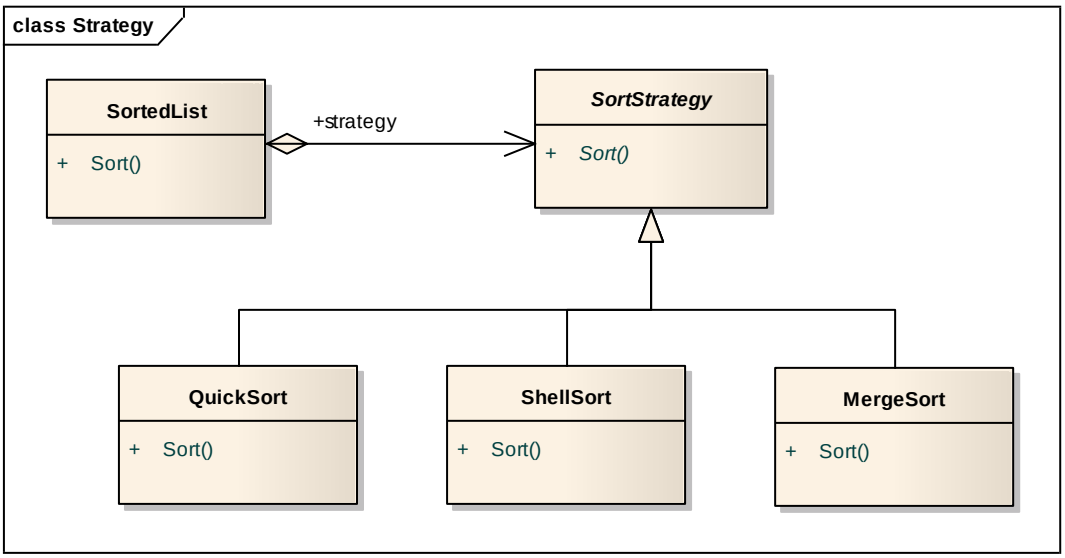


## Ejemplo

Puede hallarse la implementación en código C# de este ejemplo en http://www.dofactory.com

También ha sido creado el proyecto en el Repositorio de código fuente de la cursada

En este ejemplo las listas pueden utilizar diferentes algoritmos para su ordenamiento.




*Si no se aplicara el patrón Strategy todo el comportamiento estaría sobrecargado sobre la cuenta SortedList.*

## State vs. Strategy

Se suele decir que los patrones *Strategy* y *State* son hermanos que fueron separados al nacer. Esto se debe a que sus razones de ser son muy parecidas: modificar el funcionamiento de un objeto en tiempo de ejecución de forma transparente a través de un proceso de composición.

La diferencia fundamental, sin embargo, se encuentra en que el patrón *Strategy* tiene como objetivo **proporcionar alternativas para realizar una misma tarea**.

*State*, por su parte, tiene su razón de ser en **proporcionar comportamientos distintos** cuando la aplicación se encuentra en **estados distintos**. Con este patrón, la clase que implemente nuestro *State* realizará operaciones distintas dependiendo de su estado, no proporcionará "varias formas de hacer lo mismo". Por ello, *Strategy* y *State* son patrones distintos, aunque muy

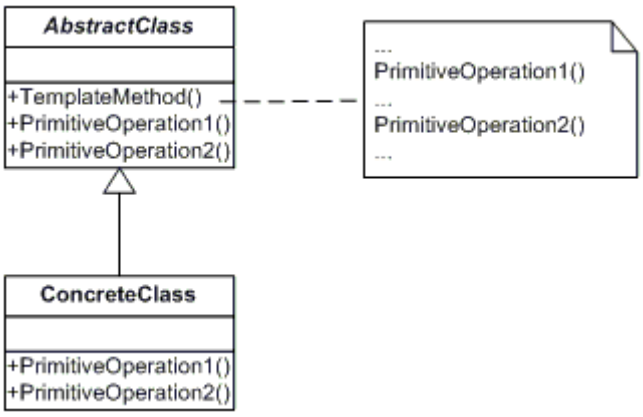
similares: *Strategy* encapsula el algoritmo, mientras que *State* encapsulará un comportamiento que variará dependiendo del estado en el que se encuentre el programa.

# TEMPLATE METHOD

## Objetivo

- Permite definir el esqueleto de un algoritmo en una operación, aunque difieren algunos pasos en las subclases.
- Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

## Estructura



## Ejemplo

Un software que maneja un motor naftero debe tener en cuenta el siguiente algoritmo:

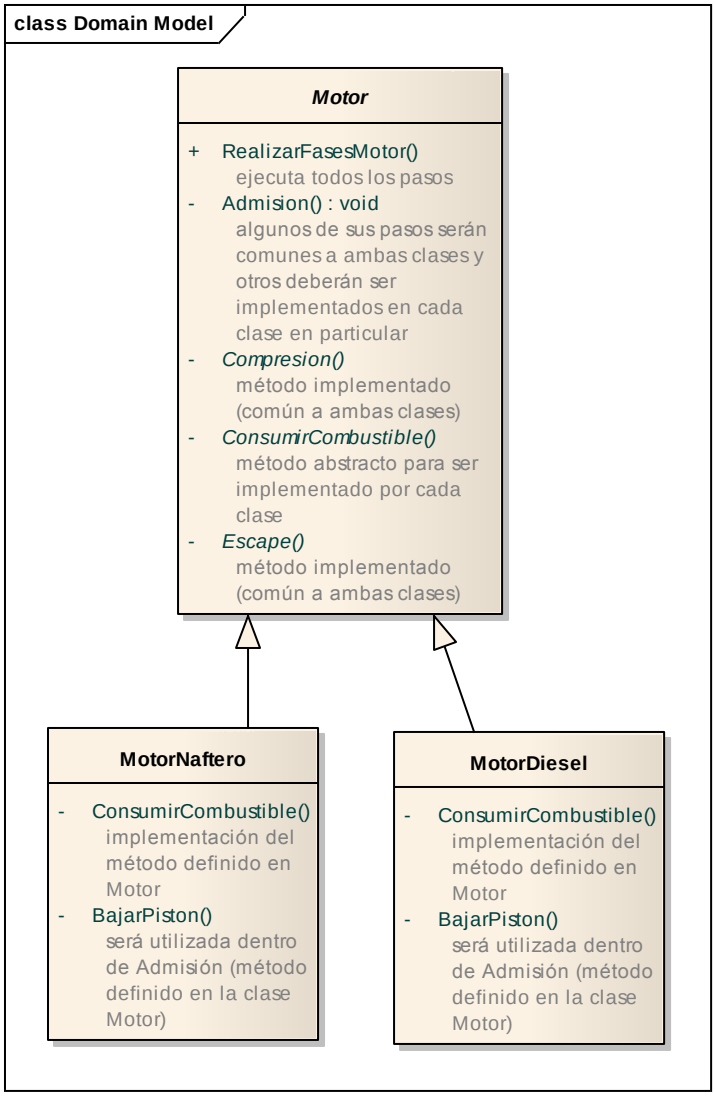
- **Admisión**: el descenso del pistón crea un vacío que aspira la mezcla de aire y combustible de la válvula de admisión. La válvula de escape permanece cerrada.
- **Compresión**: una vez que el pistón ha bajado hasta el final, se cierra la válvula de admisión. El pistón asciende, comprimiendo la mezcla y aumentando la presión.
- **Explosión**: el pistón alcanza la parte superior y la bujía produce una chispa que hace explotar la mezcla de aire y combustible, haciendo que el pistón vuelva a descender.
- **Escape**: la válvula de escape se abre. El pistón asciende nuevamente, empujando los gases resultantes de la explosión y comenzando un nuevo ciclo.

En tanto, un motor diesel de cuatro tiempos:

- **Admisión**: el descenso del pistón crea un vacío que aspira aire desde la válvula de admisión. La válvula de escape permanece cerrada.
- **Compresión**: una vez que el pistón ha bajado hasta el final, se cierra la válvula de admisión. El pistón asciende, comprimiendo el aire y aumentando la presión.
- **Combustión**: los inyectores pulverizan el combustible, haciendo que la presión se encargue de aumentar la temperatura, haciendo que se produzca la combustión y la expansión de los gases que fuerzan el descenso del pistón.
- **Escape**: la válvula de escape se abre. El pistón asciende nuevamente, empujando los gases resultantes de la explosión y comenzando un nuevo ciclo.

Tal y como observamos, ambos motores tienen un funcionamiento muy similar. Las fases 2 y 4 (compresión y escape) son idénticas, la fase 1 (admisión) varía ligeramente, mientras que la fase 3 (explosión en el motor naftero, combustión en el motor diesel) tiene un comportamiento diferente.

- En la clase Motor implementamos los métodos de fases comunes (compresión y escape) y el método que ejecuta todos los pasos (RealizarFasesMotor)
- En la clase Motor declararemos como abstracto el método que se implementa diferente (fase 3) y en cada clase se realizará su implementación según corresponda
- En la clase Motor implementaremos los métodos de la fase 1, pero algunos de sus pasos serán implementados en las clases específicas



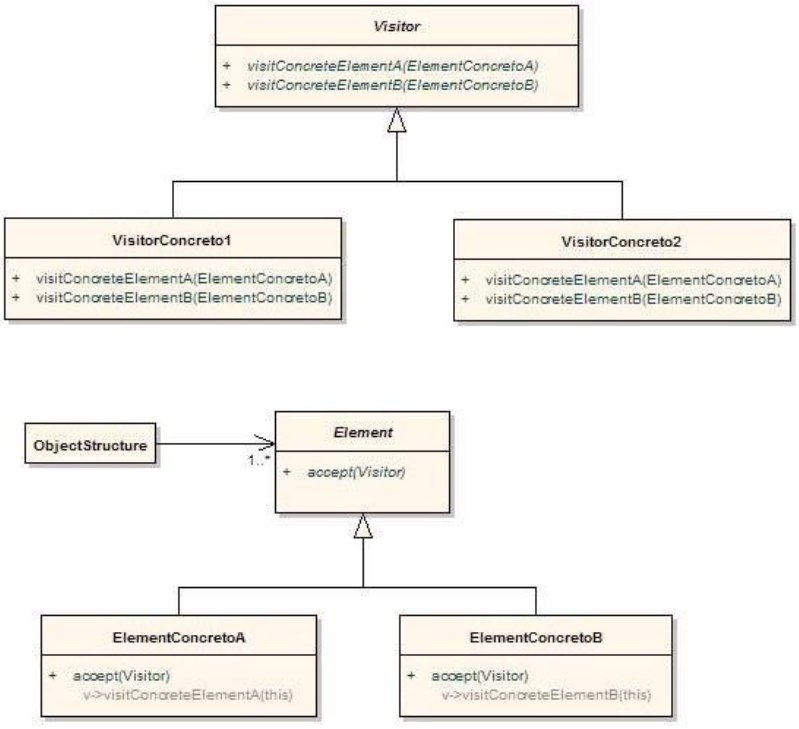

Puede hallarse la implementación en código C# de este ejemplo en el Repositorio de código fuente de la cursada.

# VISITOR

## Objetivo

- Se debe utilizar este patrón si se quiere realizar un cierto número de operaciones, que no están relacionadas entre sí, sobre instancias de un conjunto de clases, y no se quiere "contaminar" a dichas clases.
- Busca separar un algoritmo de la estructura de un objeto. La operación se implementa de forma que no se modifique el código de las clases sobre las que opera.

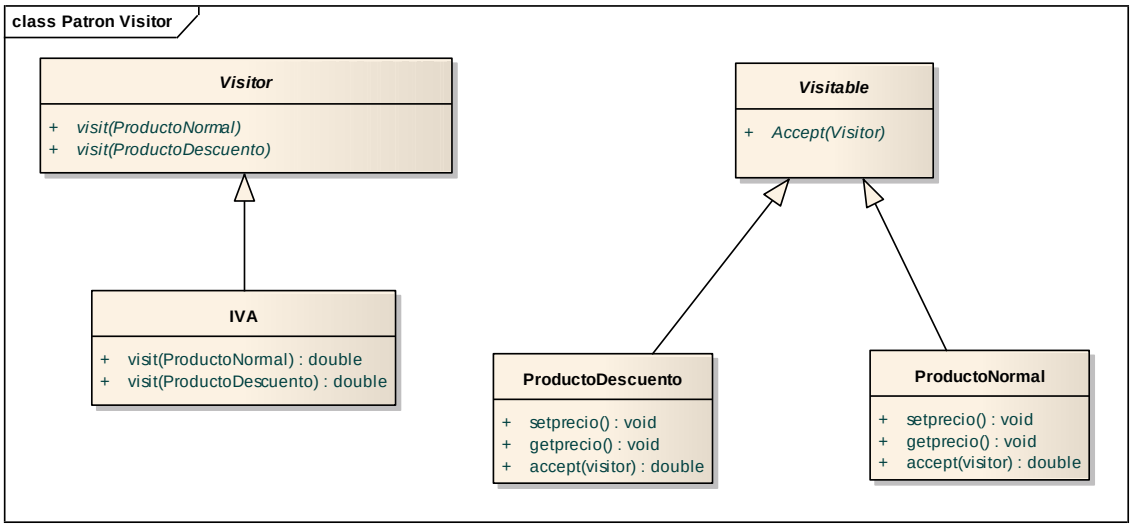
## Estructura



## Ejemplo

En Argentina todos los productos pagan IVA. Algunos productos poseen tasa reducida. Supongamos 21% de IVA para productos normales y 10% para tasa reducida.

Puede hallarse la implementación en el Repositorio de código fuente de la cursada




- El objetivo es calcular el precio de determinado producto incluyendo su IVA y desacoplar el cálculo del IVA de la propia clase (armando clases específicas que realicen el cálculo)
- El método accept del producto será el responsable de ejecutar al visit (realizar la ejecución del algoritmo que ha sido desacoplado)
- La clase que implementa el visitor lo hace para todas las clases visitables
- Cuando se modifiquen las reglas del IVA sólo cambio 1 clase y no tengo que modificar cada clase que requiere el cálculo de IVA