v.1.1

Material de Referencia
Patrones de Diseño (Gamma et al.)

Alejandro Leoz
Martín Agüero
2019

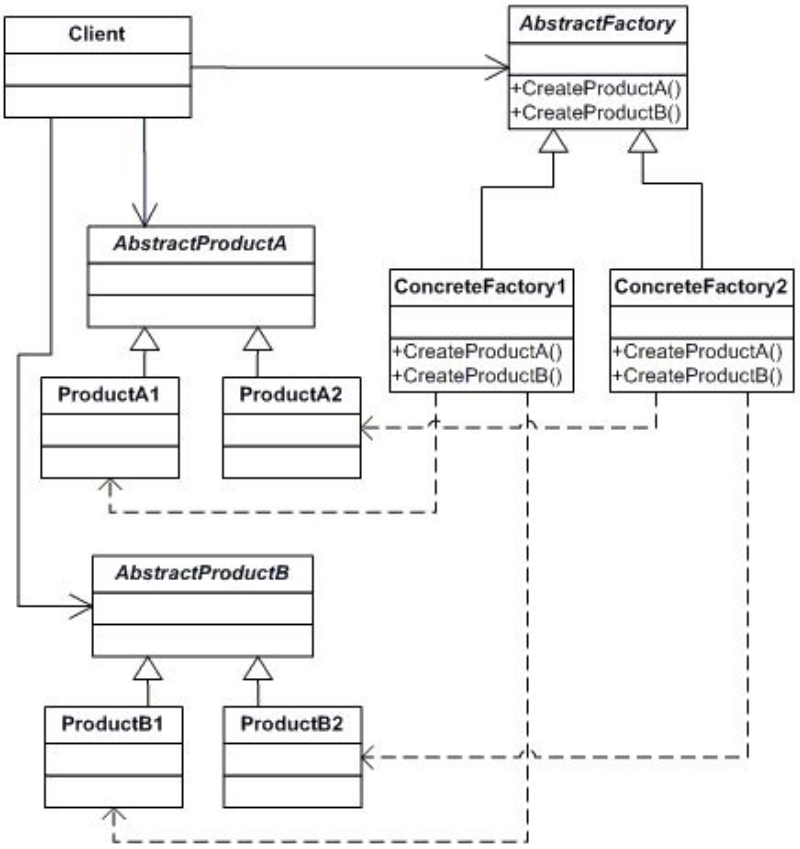
Índice:

Nota: Este texto tiene como objetivo presentar de manera muy resumida y a modo introductorio, cada uno de los patrones del texto ***Design Patterns: Elements of Reusable Object-Oriented Software***. Al final de cada resumen, la página del texto original (en inglés) y en castellano (ambos de la versión digital).

Creacionales

Abstract Factory

Tipo: Creacional
Propósito: Proveer una interfaz para la creación de familias de objetos relacionados, sin especificar la clase concreta.



Se usa cuando se quiere cambiar (usar otra) implementación en tiempo de ejecución.

Un ejemplo:

Supongamos que disponemos de una cadena de pizzerías. Para crear pizzas disponemos de un método abstracto en la clase Pizzería que será implementada por cada subclase de Pizzería.

```
abstract Pizza crearPizza();
```

Concretamente se creará una clase PizzeríaZona por cada zona, por ejemplo la Pizzería de New York sería PizzeriaNewYork y la de California PizzeríaCalifornia que implementarán el método con los ingredientes de sus zonas.
Las pizzas son diferentes según las zonas. No es igual la pizza de New York que la pizza de California. Igualmente, aunque usarán los mismos ingredientes (tomate, mozzarella...) no los obtendrán del mismo lugar, cada zona los comprará donde lo tenga más cerca. Así pues podemos crear un método creador de Pizza que sea:

```
Pizza(FactoriaIngredientes fi);
```

Como vemos utilizamos la factoría abstracta (no las concretas de cada zona, como podría ser IngredientesNewYork o IngredientesCalifornia). Pizza podrá obtener los ingredientes de la factoría independientemente de donde sea. Sería fácil crear nuevas factorías y añadirlas al sistema para crear pizzas con estos nuevos ingredientes. Efectivamente, en este ejemplo *cliente* es Pizza y es independiente de la Factoría usada.
El creador de la Pizza será el encargado de instanciar la factoría concreta, así pues los encargados de instanciar las factorías concretas serán las pizzerías locales. En PizzeríaNewYork podemos tener el método crearPizza() que realice el siguiente trabajo:

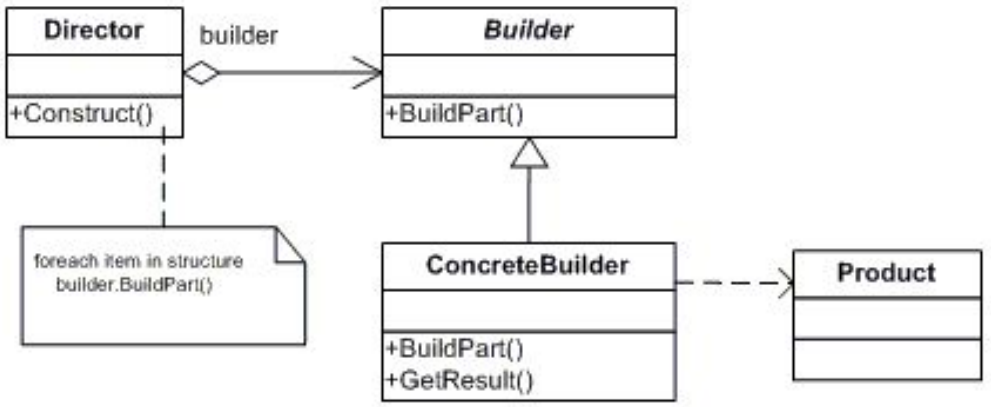
```
Pizza crearPizza() {
    FactoríaIngredientes fi = new IngredientesNewYork();
    Pizza pizza = new Pizza(fi); // Uso de la factoría
    pizza.cortar();
    pizza.empaquetar();
    return pizza;
}
```

Como conclusión podemos observar que gracias a la factoría de ingredientes crear una nueva zona, por ejemplo una pizzería en Barcelona, no nos implicaría estar modificando el código existente, solo deberemos extenderlo creando la subclase de Pizzería: PizzeríaBarcelona que al instanciar la factoría sólo debería escoger la factoría de Barcelona. Obviamente se debería crear la factoría de Barcelona que se encargaría de crear los productos obtenidos de Barcelona. Así que en ningún momento modificamos las pizzerías existentes, la superclase pizzería o las otras factorías o productos, solo creamos nuevas clases.

P. 88
P. 49

Builder

Tipo: Creacional
Propósito: Separar la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

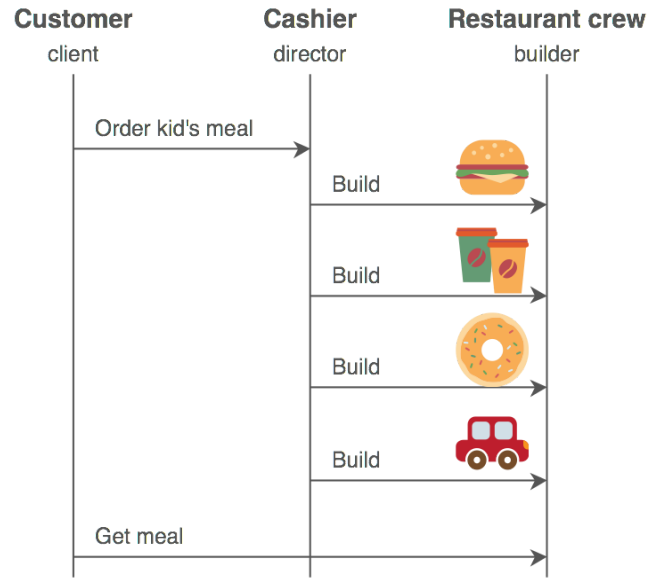


Ejemplo en C#:
Existe una clase Car. El problema es que un auto puede tener muchas opciones. La combinación de cada opción puede generar que el constructor deba recibir muchos parámetros. Por lo que se crea un Builder de Car, a fin de pasar los parámetros de las características de la nueva instancia de Car:

```
//Producto a crear por el Builder
public class Car
{
    public Car()
    {
    }
    public int Wheels { get; set; }
    public string Colour { get; set; }
}

//Interfaz Builder de Car
public interface ICarBuilder
{
    void SetColour([NotNull]string colour);
    void SetWheels([NotNull]int count);
    Car GetResult();
}

//Clase concreta de Builder
public class CarBuilder : ICarBuilder
{
    private Car _car;
    public CarBuilder()
    {
        this._car = new Car();
```

```
    }

    public void SetColour(string colour)
    {
        this._car.Colour = colour;
    }

    public void SetWheels(int count)
    {
        this._car.Wheels = count;
    }

    public Car GetResult()
    {
        return this._car;
    }
}

//El director
public class CarBuildDirector
{
    public Car Construct()
    {
        CarBuilder builder = new CarBuilder();
        builder.SetColour("Red");
        builder.SetWheels(4);
        return builder.GetResult();
    }
}
```

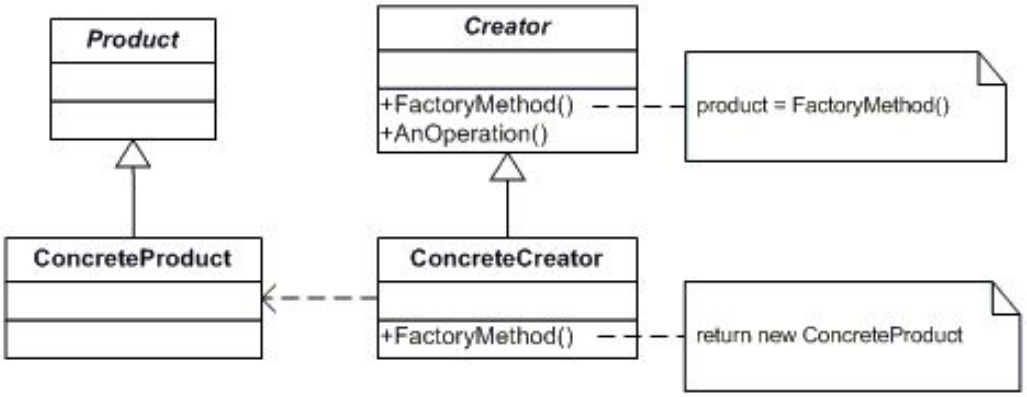
Ejemplo visual:



P. 97
P. 54

Factory Method

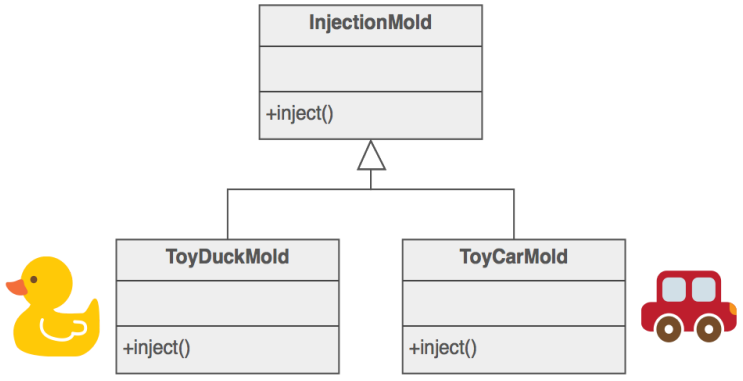
Tipo: Creacional
Propósito: Definir una *interfaz* para la creación de un objeto, pero delegando a las subclases la responsabilidad de crear las instancias apropiadas.



Es un caso particular de *Template Method*. La diferencia es el propósito: Template Method es de comportamiento y Factory Method es de creación.

A diferencia de *Abstract Factory*, Factory Method es sólo un método a cargo de la creación de instancias, mientras que Abstract Factory es un Objeto a cargo de la creación de familias de objetos.

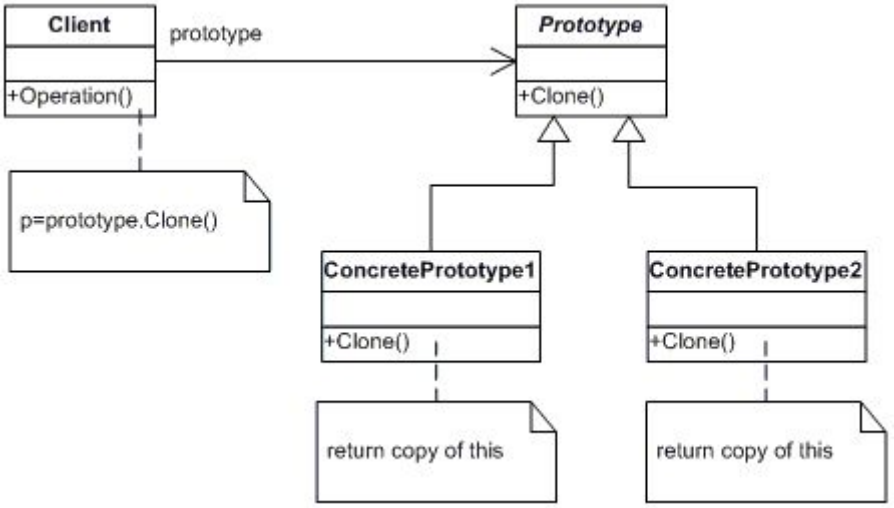
Ejemplo visual:



P. 106
P. 59

Prototype

Tipo: Creacional
Propósito: Crear nuevas instancias de objetos clonando otros objetos ya existentes.



Una ventaja es la performance: es más eficiente clonar que crear y setear valores.

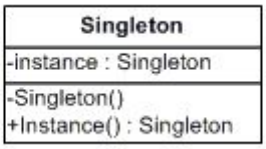
Una desventaja está relacionada a cómo se clona: hay que tener cuidado si la copia que se hace es *deep* o *shallow*.

- Shallow copy: dado un objeto A se crea un objeto B (de la misma clase que A) y se copian uno a uno los campos de A a B. Si el tipo de dato es primitivo, se copia el valor. Si el tipo de dato corresponde a un objeto, se copia la referencia, resultando que A y B apuntarán al mismo objeto en memoria.
- Deep copy: a diferencia de *shallow copy*, si el tipo de dato del campo que se está copiando corresponde a un objeto, se hace también una copia del objeto referenciado. Como resultado, A y B no apuntan al mismo objeto en memoria.

P. 115
P. 64

Singleton

Tipo: Creacional
Propósito: Asegurar que una clase sólo tenga una única instancia, y brindar un acceso global a la misma.



- El constructor es privado: de esta manera se evita que se creen instancias utilizando *new*.
- Se define un atributo de clase (estático), generalmente denominado *"instance"*, cuyo tipo es la misma clase singleton.
- Se define un método de clase (estático), generalmente denominado *"getInstance()"* que se ocupa de devolver el valor de *"instance"*. Previamente verifica que *instance* sea distinto de null, en cuyo caso llama al constructor (que por ser llamado dentro de la clase es accesible).

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

   private Singleton() {
   }

   public static Singleton getInstance() {
      if(instance == null) {
         instance = new Singleton();
      }
      return instance;
   }
}
```

Ventaja:

- Performance: no se crean instancias innecesarias.

Desventajas:

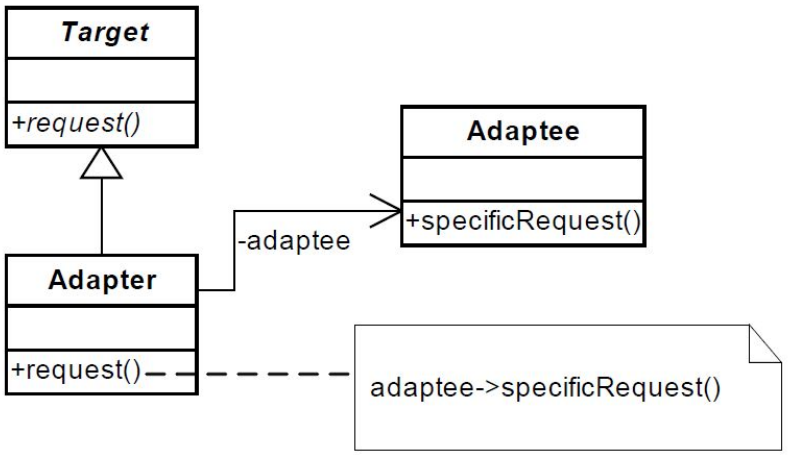
- Queda muy acoplado al sistema. Sacar un singleton de un sistema puede ser costoso debido a que el acceso es global.
- En entornos multi-threading puede generar deadlocks.

P. 124
P. 69

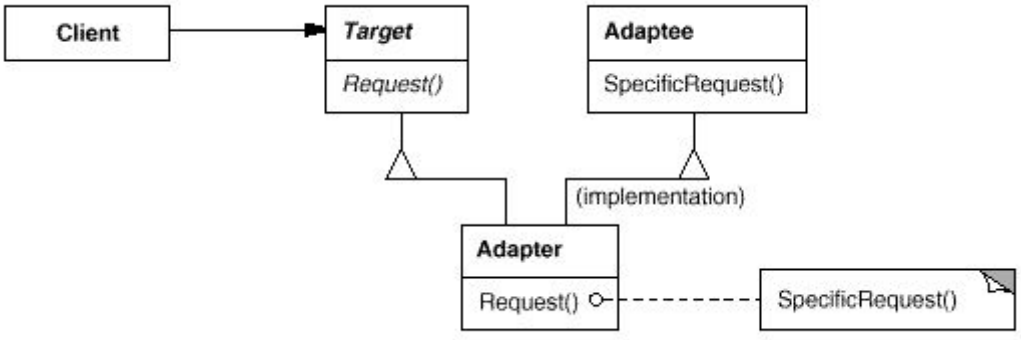
Estructurales

Adapter

Tipo: Estructural
Propósito: Convertir la interfaz de una clase en otra interfaz que el cliente espera recibir. Permite que dos clases incompatibles puedan funcionar en conjunto.



En el diagrama anterior el adapter está implementado por composición. Otra variante es la implementación por herencia:



Objetivo: Convertir la protocolo de una clase en otra,que es la que el objeto cliente espera.

- Problema: Muchas veces, clases que fueron pensadas para ser reutilizadas no pueden aprovecharse porque su protocolo no es compatible con el protocolo específico del dominio de aplicación con el que se está trabajando.
- Se desea utilizar una clase existente, cuyo protocolo no es compatible con el requerido.
- Se desea crear una clase que puede llegar a cooperar con otros objetos cuyo protocolo no se puede predecir de antemano.

Solución: Crear una clase que se encargue de “transformar” los nombres de los mensajes.

Consecuencias:

Puede utilizarse para adaptar el protocolo y los parámetros:

Ej 01/13/2001 >> 13/01/2001

Permite el reuso de un objeto como un componente.

No hace falta modificar el código del “cliente”.

Permite que un mismo Adapter trabaje con varios Adaptees.

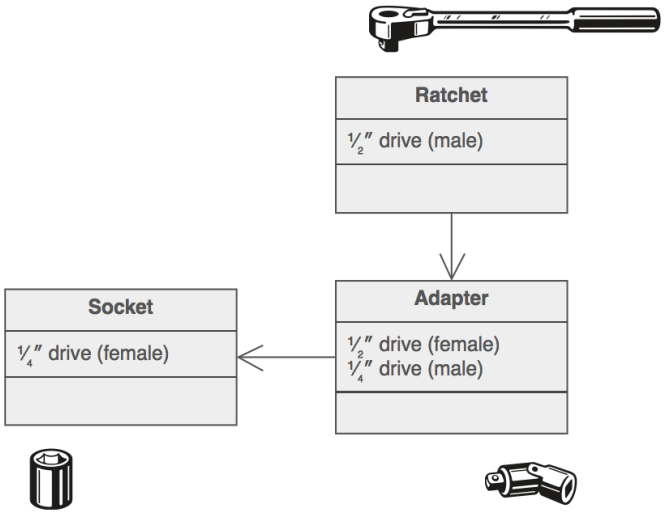
Mayor cantidad de objetos involucrados en resolver una operación.

Define mayor grado de indirección.

Generalmente es unidireccional.

Se puede concatenar varios adapters.

Ejemplo visual:

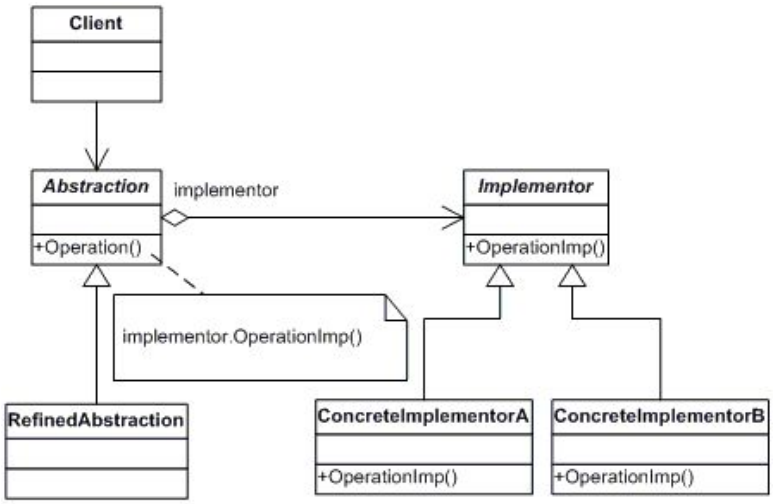


P. 135

P. 75

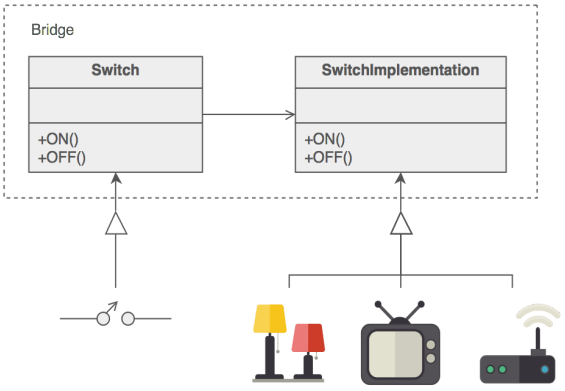
Bridge

Tipo: Estructural
Propósito: Desacoplar una abstracción de su implementación, de modo que ambas puedan variar de forma independiente.



Abstraction: define una interfaz abstracta. Mantiene una referencia a un objeto de tipo *Implementor*.
RefinedAbstraction: extiende la interfaz definida por *Abstraction*
Implementor: define la interface para la implementación de clases. Esta interface no se tiene que corresponder exactamente con la interfaz de *Abstraction*; de hecho, las dos interfaces pueden ser bastante diferente. Típicamente la interface *Implementor* provee sólo operaciones primitivas, y *Abstraction* define operaciones de alto nivel basadas en estas primitivas.
ConcreteImplementor: implementa la interface de *Implementor* y define su implementación concreta.

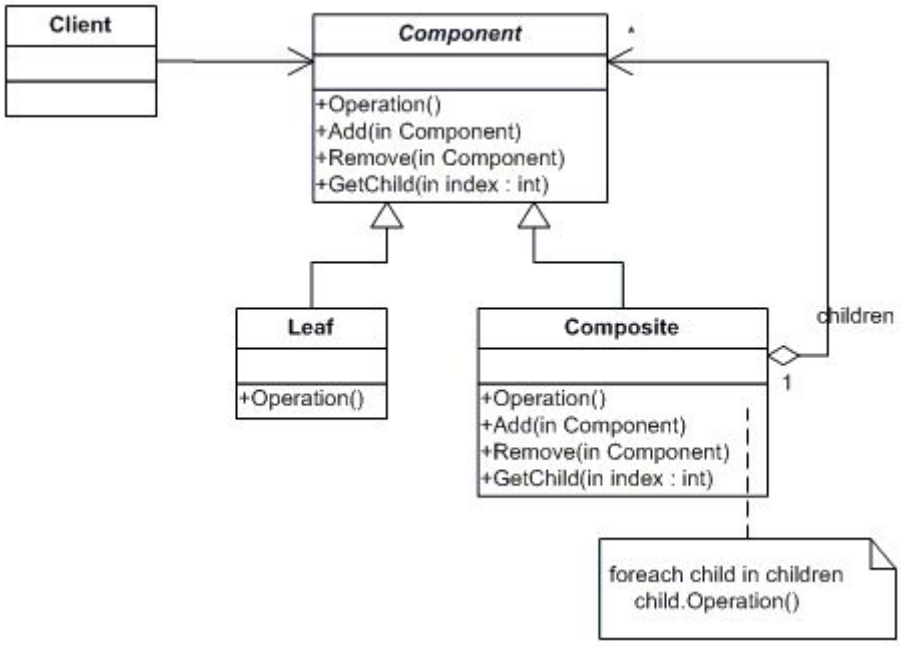
Ejemplo visual:



P.146
P.80

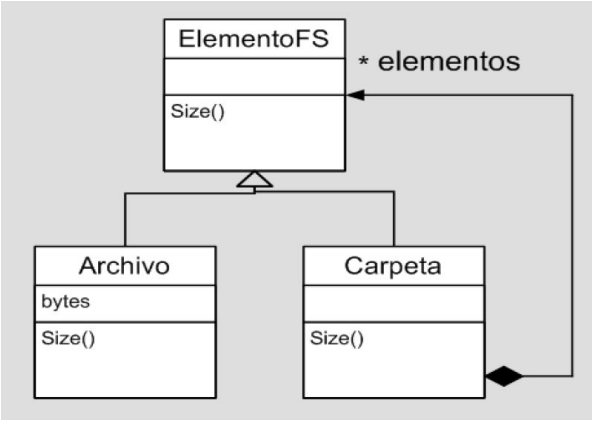
Composite

Tipo: Estructural
Propósito: Componer objetos en estructuras de árbol para representar jerarquías. Permite tratar objetos individuales o compuestos de la misma manera.



Desventaja: un *Component* puede aceptar cualquier otro *Component* (cualquier otra subclase o implementación de Component) debido a la herencia. Hay que hacer validaciones si es necesario.

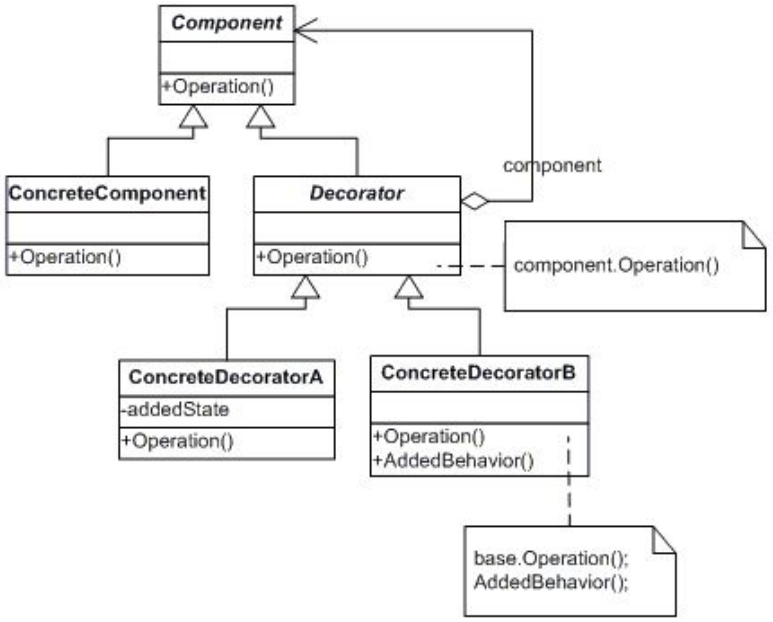
Ejemplo:



P. 156
P. 85

Decorator

Tipo: Estructural
Propósito: Agregar dinámicamente responsabilidades (funcionalidad) extra a un objeto. Es una forma flexible que sirve de alternativa a subclassing para extender funcionalidad.

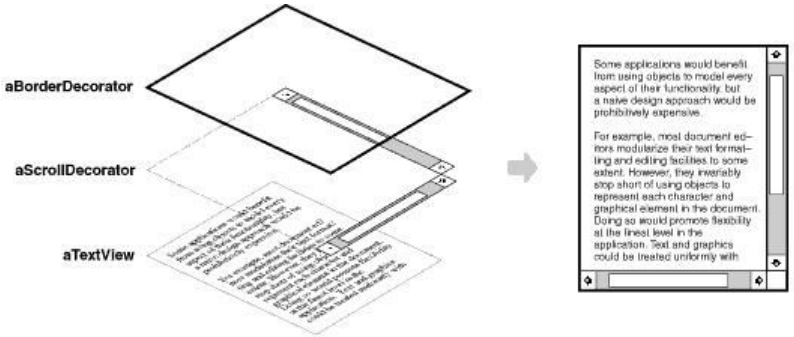


Nótese que tanto *ConcreteComponent* como *ConcreteDecoratorA* y *ConcreteDecoratorB* son *Component*. Los dos últimos, que extienden de *Decorator*, agregan algo.

Ventajas:

- Simplifica el código.
- Se pueden agregar *Decorators* dinámicamente de forma simple.

Ejemplo: dado un objeto *TextView* que sólo muestra texto, se le aplica un *ScrollDecorator* que agrega la funcionalidad de scrollbar. Luego, se aplica un *BorderDecorator* que agrega la funcionalidad de borde.

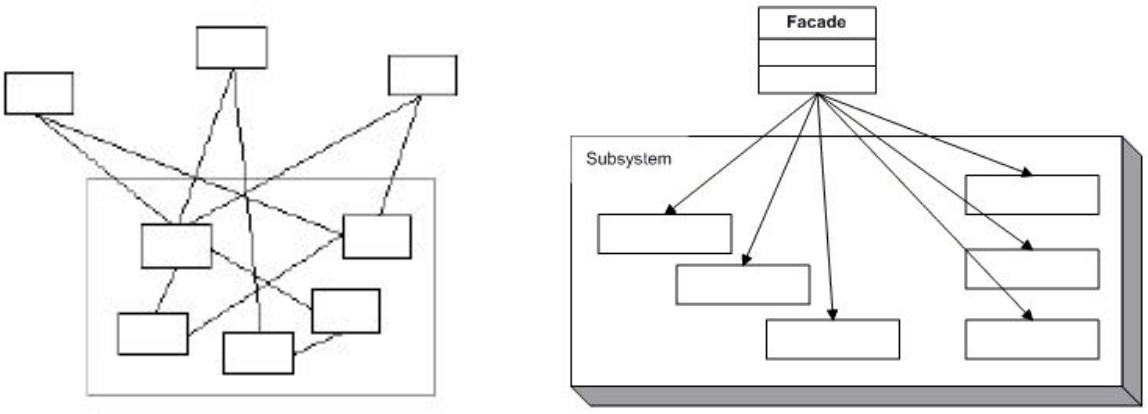


P. 166
P. 90

Facade

Tipo: Estructural
Propósito: Proveer una interfaz unificada a un set de interfaces en un subsistema. Define una interfaz de más alto nivel para simplificar el uso del sistema.

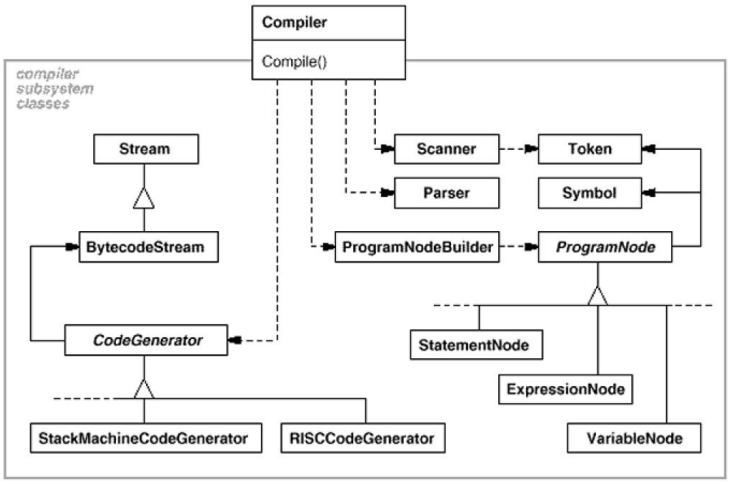
> Desacopla las clases "Cliente" del subsistema.
> **No** desacopla las clases del subsistema entre sí.



Ventaja

- Simplifica el modelo quitando visibilidad de métodos del subsistema que no se usan en clases cliente.
- Permite dividir el sistema en capas.

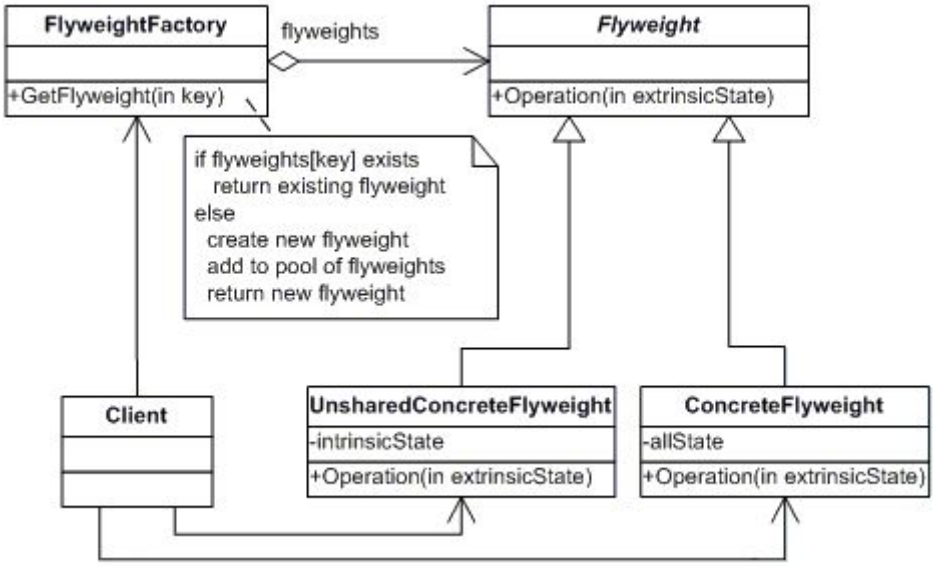
Ejemplo:
Compilador es la "fachada" a todo el conjunto de componentes que lo integra.



P. 175
P. 95

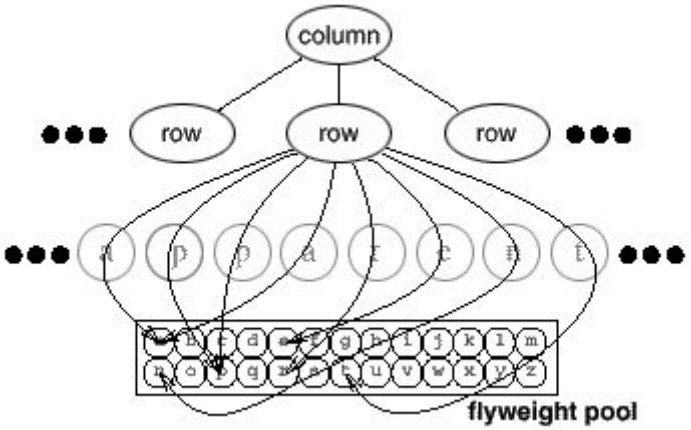
Flyweight

Tipo: Estructural
Propósito: Emplear objetos compartidos que son similares, en lugar de crear nuevas instancias. Se usa cuando se necesita crear muchos objetos similares. Mejora la performance, ya que reduce el uso de memoria.



La *Factory* guarda una única instancia (o un pool acotado de instancias) de los *ConcreteFlyweight* para un key determinado.
Los *ConcreteFlyweight* debería ser stateless ya que se pueden utilizar en diferentes contextos.

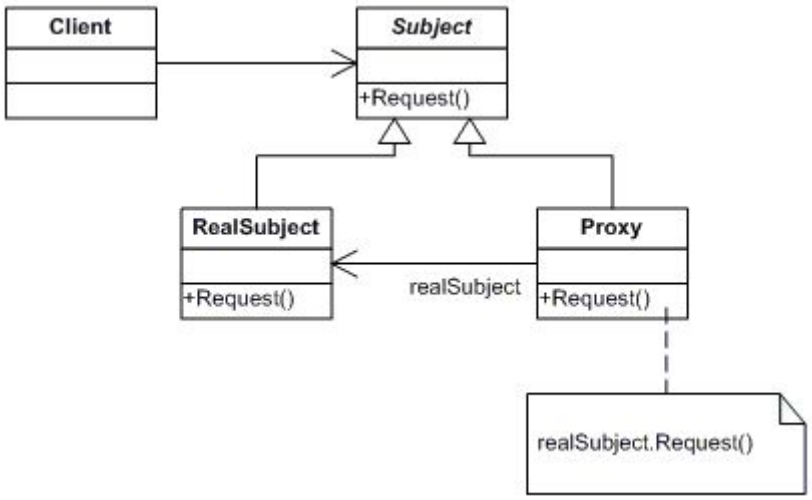
Ejemplo:
En un editor de texto, en lugar de repetir la instancia a cada símbolo, se define una referencia a ese en el flyweight pool.



P. 184
P. 99

Proxy

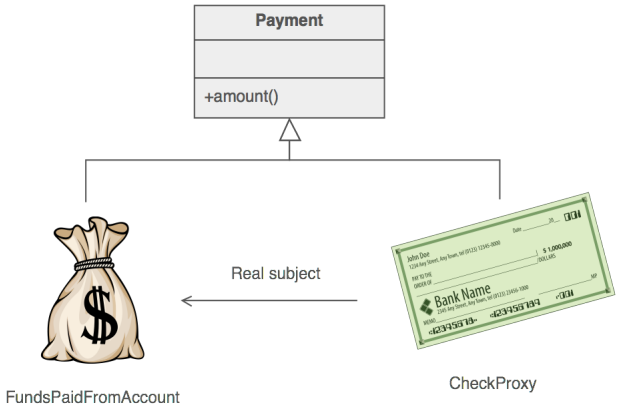
Tipo: Estructural.
Propósito: Proveer un sustituto de otro objeto para controlar el acceso al mismo.



Tipos de proxies:

- Remote proxy: representación local de un objeto remoto
- Virtual proxy: para crear objetos costosos on-demand.
- Protection proxy: controlar el acceso al objeto real. Se usa cuando hay diferentes permisos/políticas de acceso.
- Smart Reference: se usa como reemplazo de los punteros comunes, agregando acciones adicionales cuando se accede a un objeto. (esto es más de bajo nivel).

Ejemplo visual:

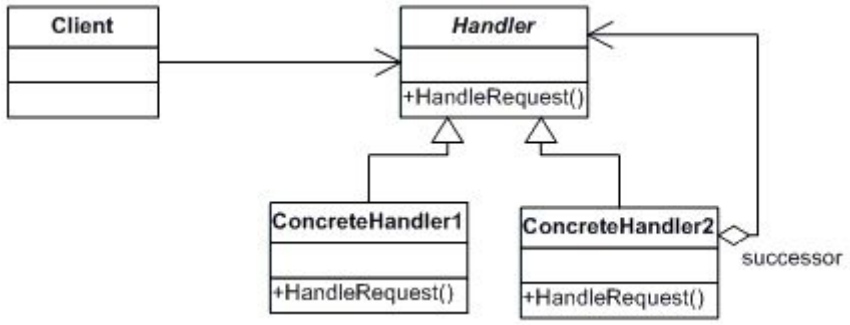


P. 196
P. 105

Comportamiento

Chain of Responsibility

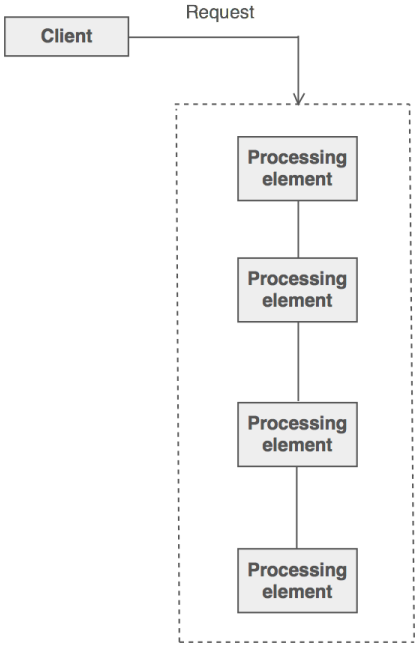
Tipo: Comportamiento
Propósito: Evita acoplar el emisor de una solicitud a su receptor, dando a más de un objeto la posibilidad de responder a la solicitud. Encadena los objetos receptores y pasa la solicitud a través de la cadena hasta que es procesada por algún objeto.



Consecuencias:

- Reduce el acoplamiento porque libera a un objeto de tener que saber qué otro objeto maneja una solicitud.
- Añade flexibilidad para asignar responsabilidades a objetos.
- No se garantiza la recepción dado que las solicitudes no tienen un receptor explícito no hay garantía de que sean manejadas por alguno de los objetos que forman parte de la cadena.

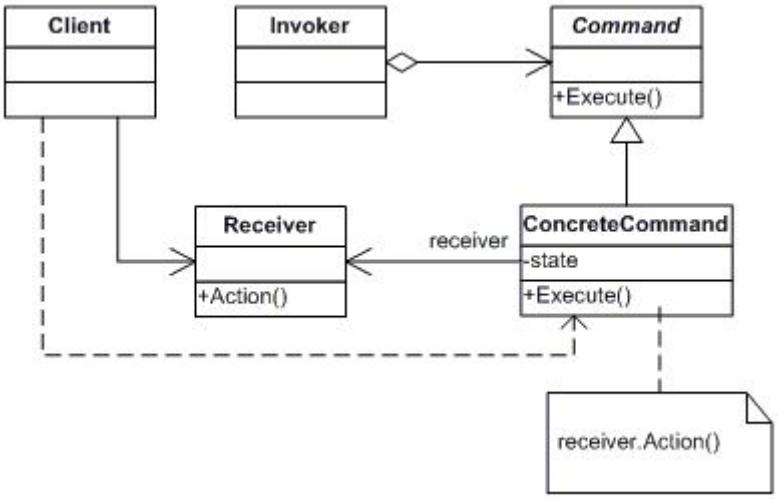
Ejemplo visual:



P. 209
P. 112

Command

Tipo: comportamiento
Propósito: Encapsular una única acción en un objeto, permitiendo que ésta sea parametrizable por las clases clientes y que también pueda ser deshecha (undo)



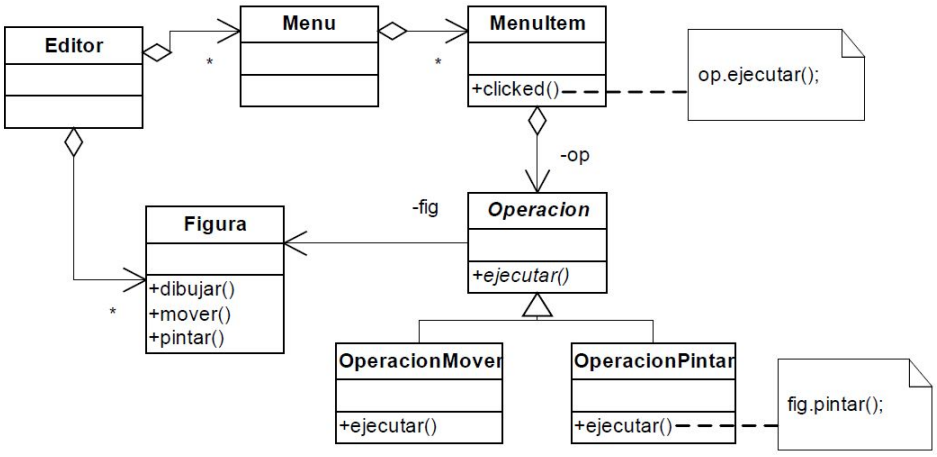
Se suele utilizar para:

- Encolar (queue) acciones o transacciones.
- Atomizar operaciones que se pueden hacer (do) y deshacer (undo).

Ventajas:

- Desacoplamiento de las clase cliente y la clase que realiza la acción.
- Se pueden agregar nuevos *Commands* sin modificar la estructura existente.

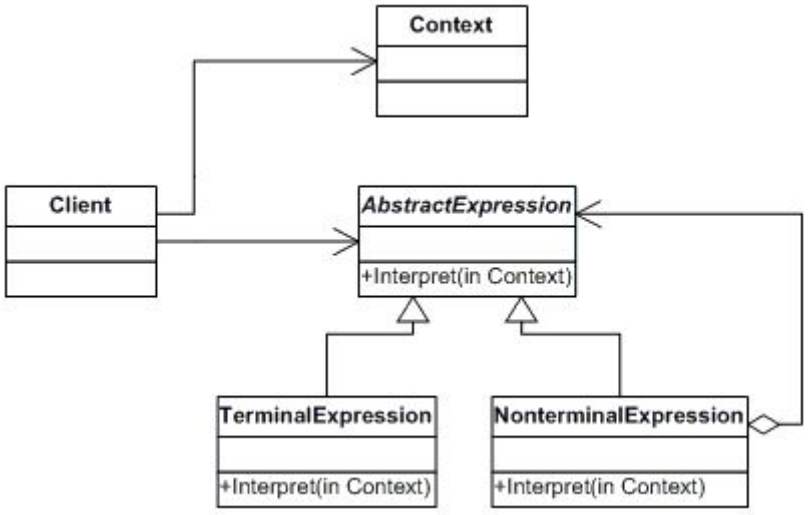
Ejemplo:



P. 219
P. 117

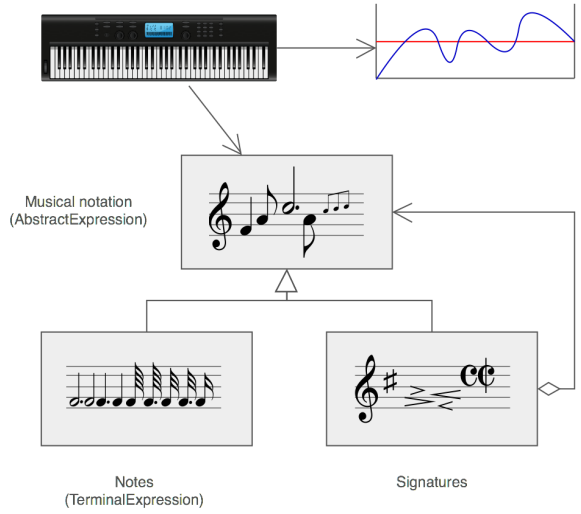
Interpreter

Tipo: Comportamiento
Propósito: Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.



Aplicabilidad:
Se emplea cuando hay que interpretar un lenguaje y las sentencias del mismo se pueden representar como árboles sintácticos abstractos. Este patrón funciona mejor cuando la gramática es simple.

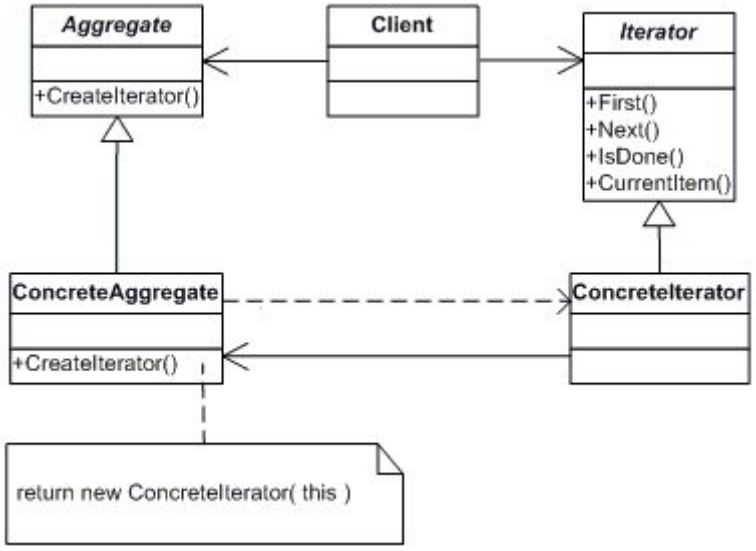
Ejemplo visual:
Los músicos son ejemplos de Intérpretes. El tono de un sonido y su duración puede ser representada en un pentagrama. Esta notación proporciona el lenguaje de la música.



P. 229
P. 122

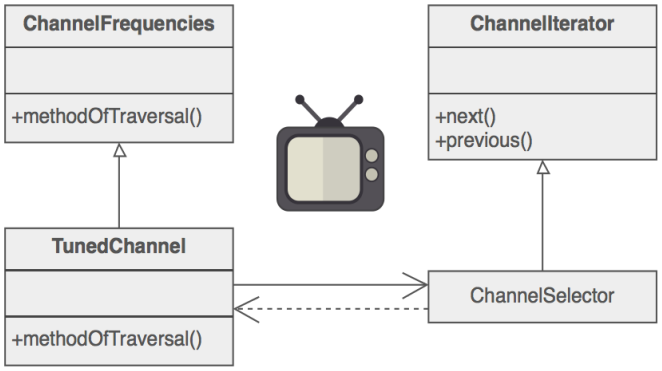
Iterator

Tipo: Comportamiento
Propósito: Proveer una forma de acceder a los elementos de una colección de objetos en orden secuencial, sin exponer la implementación real de la colección.



El iterator se encarga de saber cómo iterar la colección, siendo esta implementación totalmente transparente para el cliente.

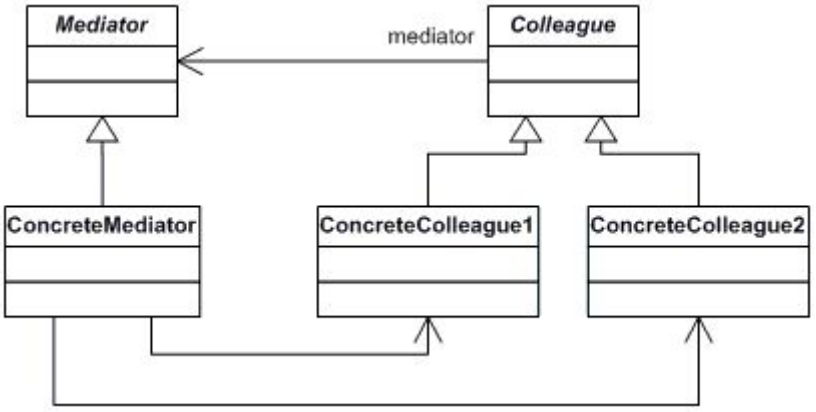
Ejemplo visual:
En los antiguos, se utilizaba un dial para cambiar de canal. En los televisores modernos, se utiliza un botón siguiente y anterior para iterar secuencialmente entre la colección de canales.



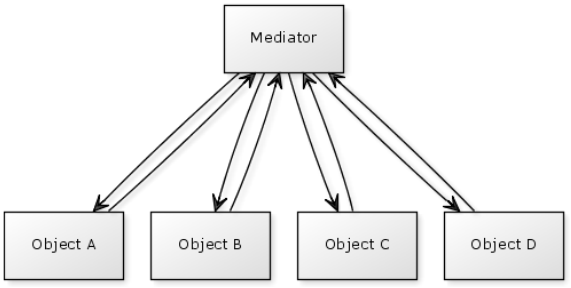
P. 241
P. 128

Mediator

Tipo: Comportamiento
Propósito: Definir un objeto que encapsula cómo un set de objetos interactúan. Provee bajo acoplamiento ta que se evita que los objetos se referencien entre sí explícitamente, y permite modificar la forma de interacción independientemente.



- *Mediator* sirve para desacoplar las clases *Colaboradoras (Colleague)* entre sí.
- Todas las colaboradoras conocen al mediator y el mediator conoce a todas las colaboradoras
- La comunicación está encapsulada en el mediador.
- Ej: torre de control de un aeropuerto.



Ventajas

- Bajo acoplamiento + Lógica centralizada => fácil de mantener.

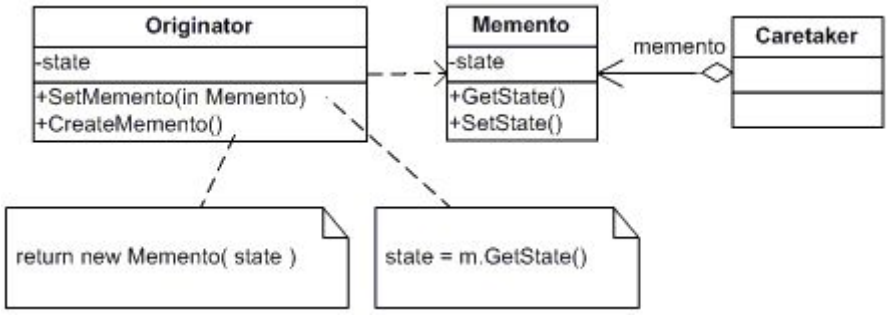
Desventajas

- Lógica centralizada -> puede generar código spaghetti.

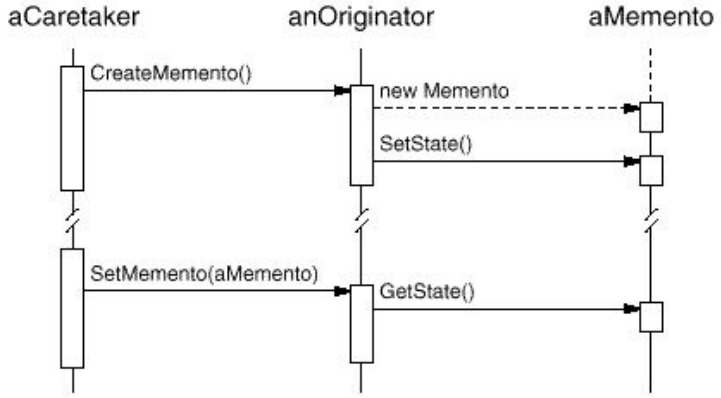
P. 255
P. 135

Memento

Tipo: Comportamiento
Propósito: Representa y externaliza el estado interno de un objeto sin saltear la encapsulación, de forma que éste pueda volver al anterior estado más tarde.



Ejemplo:
Un cuidador (caretaker) solicita un recuerdo de un originador, lo mantiene durante un tiempo, y se lo pasa de nuevo a la originador, como el siguiente diagrama ilustra la interacción:

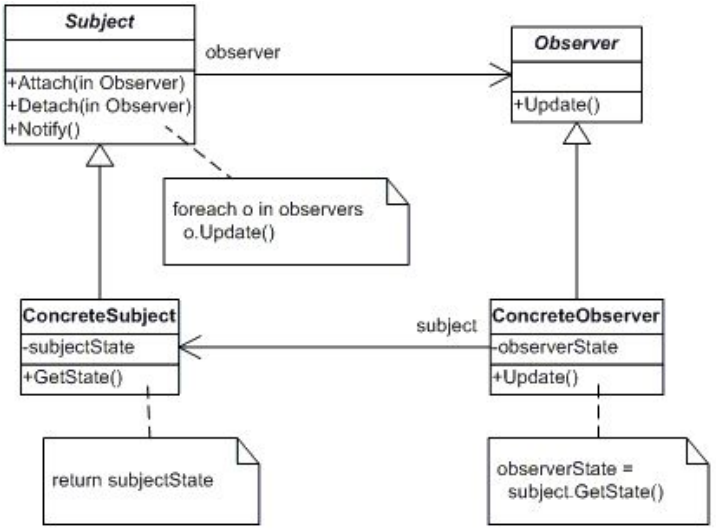


Los recuerdos son pasivos. Sólo el creador que creó un recuerdo asignará o recuperar su estado.

P. 265
P. 140

Observer

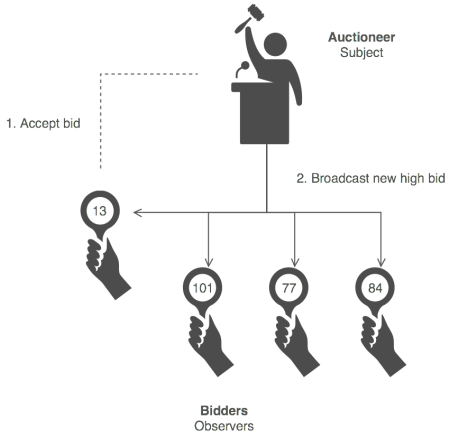
Tipo: Comportamiento
Propósito: Definir dependencias one-to-many entre objetos, de forma tal que cuando un objeto cambia su estado todos los objetos dependientes son notificados y actualizados inmediatamente



Ventajas

- Ayuda a desacoplar
- Mejora la performance, ya que el observer no tiene que hacer poll para verificar si hay cambios.

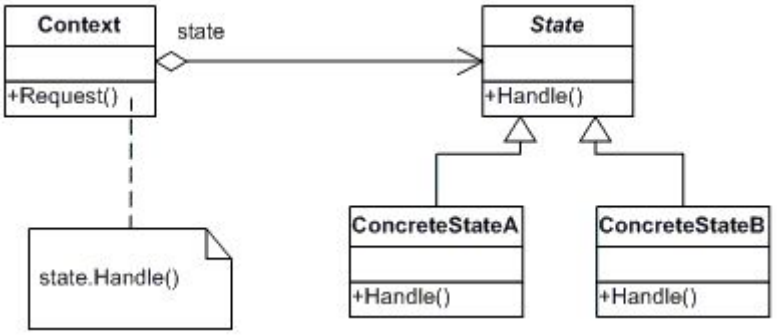
Ejemplo visual:
Cada oferente posee una paleta numerada que se utiliza para indicar una oferta. El subastador comienza la licitación, y "observa" cuando se eleva una paleta para aceptar la oferta. La aceptación de la oferta cambia el precio de la oferta que se emite a todos los licitadores.



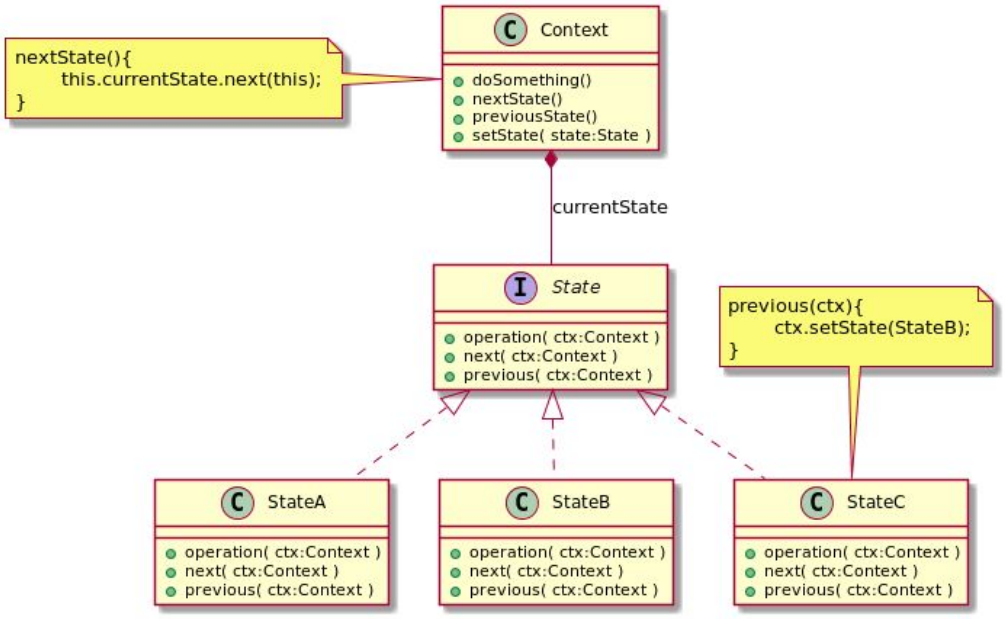
P. 273
P. 144

State

Tipo: Comportamiento
Propósito: Permitir que un objeto altere su comportamiento cuando su estado interno cambia. Permite modelar las transiciones entre estados.



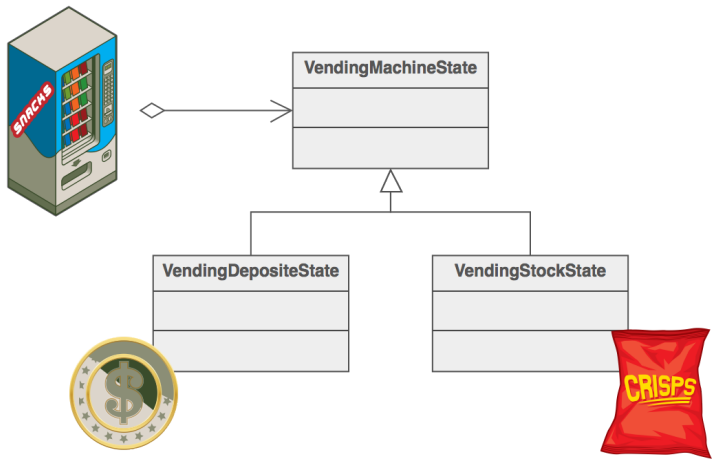
Una variante muy utilizada es que el método *Handle(..)* reciba como parámetro el objeto *Context*. De esta forma, las instancias de *State* no necesitan tener una relación hacia Context. Por ejemplo:



Ventajas

- Desacoplamiento
- Mantenimiento
- Instancias de objetos *State* se pueden compartir.

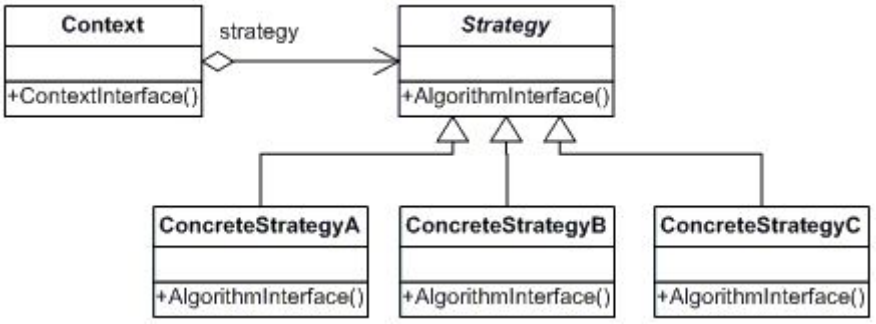
Ejemplo visual:
Una máquina expendedora funciona internamente como una máquina de estados.


VendingMachineState
VendingDepositeState
VendingStockState
SNACKS
CRISPS

P. 283
P. 149

Strategy

Tipo: Comportamiento
Propósito: Definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. Permite a las clases cliente cambiar el algoritmo en tiempo de ejecución.

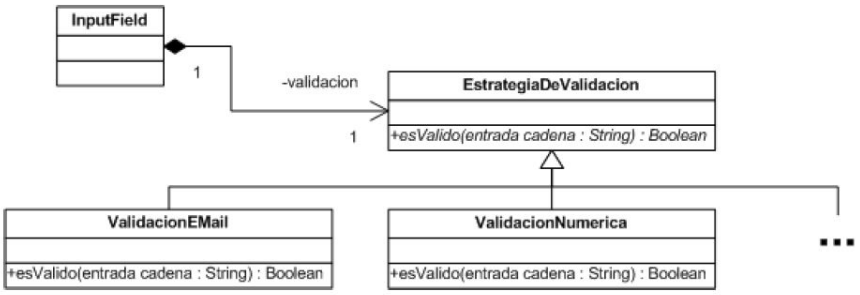


En el diagrama, la llamada desde context sería: `this.strategy.AlgorithmInterface()`.

Ventajas:

- Desacoplamiento
- Mantenimiento
- Alternativa a la herencia (subclassing)
- Ayuda a reducir el uso de condicionales

Ejemplo:

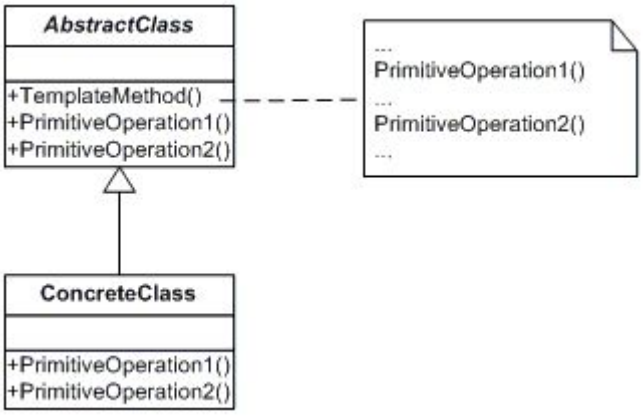


Consecuencias:
Las implementaciones concretas se pueden cambiar en forma dinámica.
Son independientes de los clientes que los usan.
Los clientes no conocen los detalles de implementación.
Se eliminan los condicionales.

P. 292
P. 154

Template Method

Tipo: Comportamiento
Intent: Definir el esqueleto de un algoritmo para una operación, delegando algunos pasos en las subclases. Las subclases definen la implementación de esos pasos sin cambiar la estructura del algoritmo.



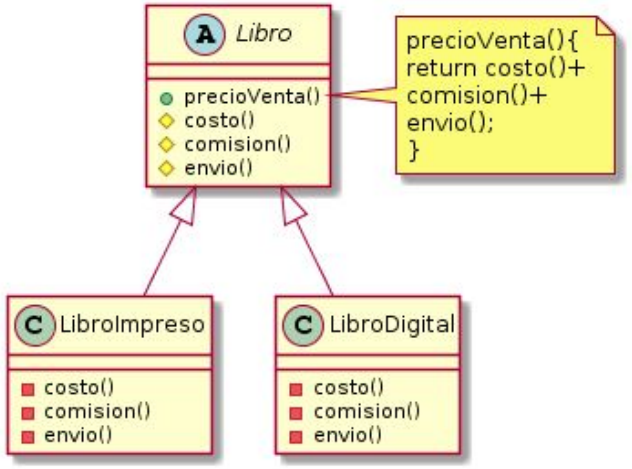
Se puede utilizar con dos enfoques:

- Hooks (gancho): son puntos de extensión y pueden sobreescribirse.
- Operaciones abstractas: definen al algoritmo y deben sobreescribirse.

Ventajas

- Reutilización de código.
- Evita código duplicado.

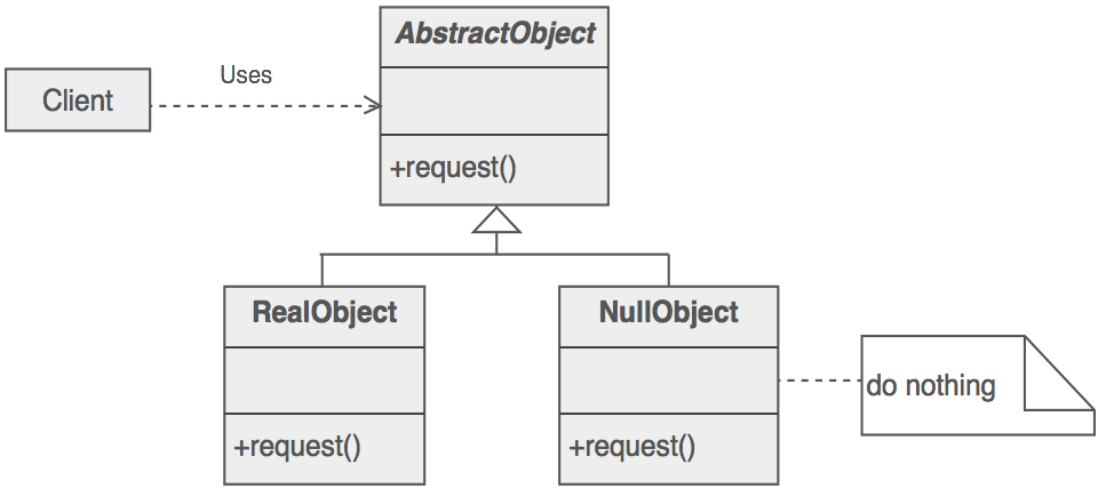
Ejemplo:
Para calcular el precio de las diferentes versiones de libros (impreso o digital) se define una clase abstracta con el método precioVenta() que definirá la plantilla para calcular el precio.



P. 301
P. 159

Null Object

Tipo: Comportamiento
Propósito: Encapsula la ausencia de un objeto mediante otro objeto que provee un comportamiento "do nothing"(no hacer nada) por omisión.



Suele emplearse en combinación con el patrón *State*.

Ventajas

- Evitar excepciones Null Pointer
- Minimiza el uso de condicionales

(Este patrón no está incluido en el libro de Gamma)

Referencias

Gamma, E. et al., Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley. 1994.

Balaguer, F., Garrido, A., Introducción a Patrones de Diseño.
Tópicos de Ingeniería de Software II. Postgrado Universidad Nacional de La Plata. 2011.

Metsker, S., The Design Patterns Java Workbook.
Addison-Wesley. 2002.

Freeman, E.,Head First Design Patterns.
O'Reilly. 2004.

Shvets, A., Design Patterns Explained Simply.
Source Making. 2015.

Do Factory http://www.dofactory.com/