

Patrones de Diseño

Antonio Espín Herranz

¿Qué es un patrón de diseño?

- “Un patrón describe un problema que ocurre una y otra vez en nuestro entorno y describe el núcleo de la solución al problema, de forma que puedes utilizar esta solución millones de veces” [C. Alexander].
- “Descripciones de clases y objetos que se comunican y que son adaptadas para resolver un problema de diseño general en un contexto particular”.

Elementos de un patrón

- Son 4 elementos:
 - Nombre del patrón: Se describe con una o dos palabras, un problema de diseño con sus soluciones y consecuencias. Nos permiten un mayor nivel de abstracción a la hora de diseñar.
 - Problema: Describe cuando aplicar el patrón. A veces el problema incluye una serie de condiciones para poder aplicar el patrón.
 - Solución: Describe los elementos que constituyen el diseño o una implementación en concreto.
 - Consecuencias: Los resultados, así como ventajas e inconvenientes que surgen al aplicar el patrón.

Descripción de los patrones de diseño

- A la hora de describir los patrones podemos utilizar los siguientes conceptos:
 - Nombre del patrón y Clasificación:
 - Un buen nombre es vital y la clasificación nos ayuda a entender mejor su comportamiento.
 - Propósito:
 - ¿Qué hace ese patrón?
 - ¿En qué se basa ese patrón?
 - ¿Qué problema de diseño resuelve?
 - Otros nombres que se le asignan.

La lista GoF

- **Gain of Fourth**, la pandilla de los 4,
- Recopilaron la lista de patrones:
 - Erich Gamma
 - Richard Helm
 - Ralph Johnson
 - John Vlissides

Catálogo de Patrones

PROPOSITO				
		DE CREACIÓN	ESTRUCTURALES	COMPORTAMIENTO
Ámbito	Clase	Factory Method	Adapter (de Clases)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (de Objetos) Bridge Composite Decorator Facade FlyWeight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Catálogo de Patrones

- Hacemos una primera clasificación por el **propósito**:
 - **Creación**: Tienen que ver con el proceso de creación de objetos.
 - **Estructurales**: Tratan de la composición de clases y objetos.
 - **Comportamiento**: Especifica la forma en que las clases y objetos interactúan y se reparten las responsabilidades.
- En cuanto al segundo criterio **ámbito** especifica si el patrón se aplica a clases u objetos.

Breve descripción I

- **Abstract Factory (Fábrica abstracta):**
 - Proporciona una interfaz de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
- **Adapter (Adaptador):**
 - Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.
- **Bridge (Puente):**
 - Desacopla una abstracción de su implementación, de manera que ambas pueden variar de forma independiente.
- **Builder (Constructor):**
 - Separa la construcción de un objeto de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

Breve descripción II

- **Chain of Responsibility (Cadena de responsabilidad):**
 - Evita el acoplar el emisor de una petición a su receptor, al dar a mas de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que esta sea tratada por algún objeto.
- **Command (Orden):**
 - Encapsula una petición de un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.
- **Composite (Compuesto):**
 - Combina objetos en estructura de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
- **Decorator (Decorador):**
 - Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Breve descripción III

- **Facade (Fachada):**
 - Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.
- **Factory Method (Método de fabricación):**
 - Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan que clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.
- **FlyWeight (Peso ligero):**
 - Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.
- **Interpreter (Intérprete):**
 - Dado un lenguaje, define una representación de su gramática junto con su intérprete que usa dicha representación para interpretar sentencias del lenguaje.

Breve descripción IV

- **Iterator (Iterador):**
 - Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
- **Mediator (Mediador):**
 - Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
- **Memento (Recuerdo):**
 - Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que este puede volver a dicho estado mas tarde.
- **Observer (Observador):**
 - Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualiza automáticamente todos los objetos que dependen de él.

Breve descripción V

- **Prototype (Prototipo):**
 - Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando de ese prototipo.
- **Proxy (Apoderado):**
 - Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.
- **Singleton (Único):**
 - Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.
- **State (Estado):**
 - Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

Breve descripción VI

- **Strategy (Estrategia):**
 - Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.
- **Template Method (Método plantilla):**
 - Define en una operación el esqueleto de una plantilla, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
- **Visitor (Visitante):**
 - Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

Aspectos de Diseño que los patrones permiten modificar

PROPÓSITO	Patrones de diseño	Aspectos que pueden variar
De creación	Abstract Factory	<i>La familia de los objetos producidos</i>
	Builder	<i>Cómo se crea un objeto compuesto</i>
	Factory Method	<i>La subclase del objeto que es instanciado</i>
	Prototype	<i>La clase del objeto que es instanciado</i>
	Singleton	<i>La única instancia de una clase</i>
Estructurales	Adapter	<i>La interfaz de un objeto</i>
	Bridge	<i>La implementación de un objeto</i>
	Composite	<i>La estructura y composición de un objeto</i>
	Decorator	<i>Las responsabilidades de un objeto sin usar la herencia</i>
	Facade	<i>La interfaz de un subsistema</i>
	Flyweight	<i>El coste de almacenamiento de los objetos</i>
	Proxy	<i>Como se accede a un objeto, su ubicación</i>
De comportamiento	Chain of responsibility	<i>El objeto que puede satisfacer una petición</i>
	Command	<i>Cuándo y cómo se satisface una petición</i>
	Interpreter	<i>La gramática e interpretación de un lenguaje</i>
	Iterator	<i>Cómo se recorren los elementos de un agregado</i>
	Mediator	<i>Qué objetos interactúan entre sí, y cómo</i>
	Memento	<i>Qué información privada se almacena fuera de un objeto, y cuando</i>
	Observer	<i>El número de objetos que dependen de otro, cómo se mantiene actualizado el objeto dependiente</i>
	State	<i>El estado de un objeto</i>
	Strategy	<i>Un algoritmo</i>
	Template Method	<i>Los pasos de un algoritmo</i>
	Visitor	<i>Las operaciones que pueden aplicarse a los objetos sin cambiar sus clases</i>

Patrones de Creación

Patrones de Creación

- Características principales:
 - Nos abstraen el proceso de creación de instancias.
 - Ayudan a hacer un sistema independiente de cómo se crean, se componen y se representan sus objetos.
 - Estos patrones se hacen mas importantes a medida que los sistemas pasan a ***depender mas de la composición de objetos que de la herencia*** entre clases.
 - Los patrones de creación dan mucha flexibilidad a qué es lo que se crea, quién lo crea y cuándo.

SINGLETON

- Singleton provee un mecanismo para limitar el número de instancias de una clase. Por lo tanto el mismo objeto es siempre compartido por distintas partes del código.
- Singleton puede ser visto como una solución más elegante para una variable global porque los datos son abstraídos por detrás de la interfaz de la clase singleton.

SINGLETON –Motivacion & Aplicabilidad

- Motivación

Algunas clases sólo pueden tener una instancia. Una variable global no garantiza que sólo se instancia una vez.

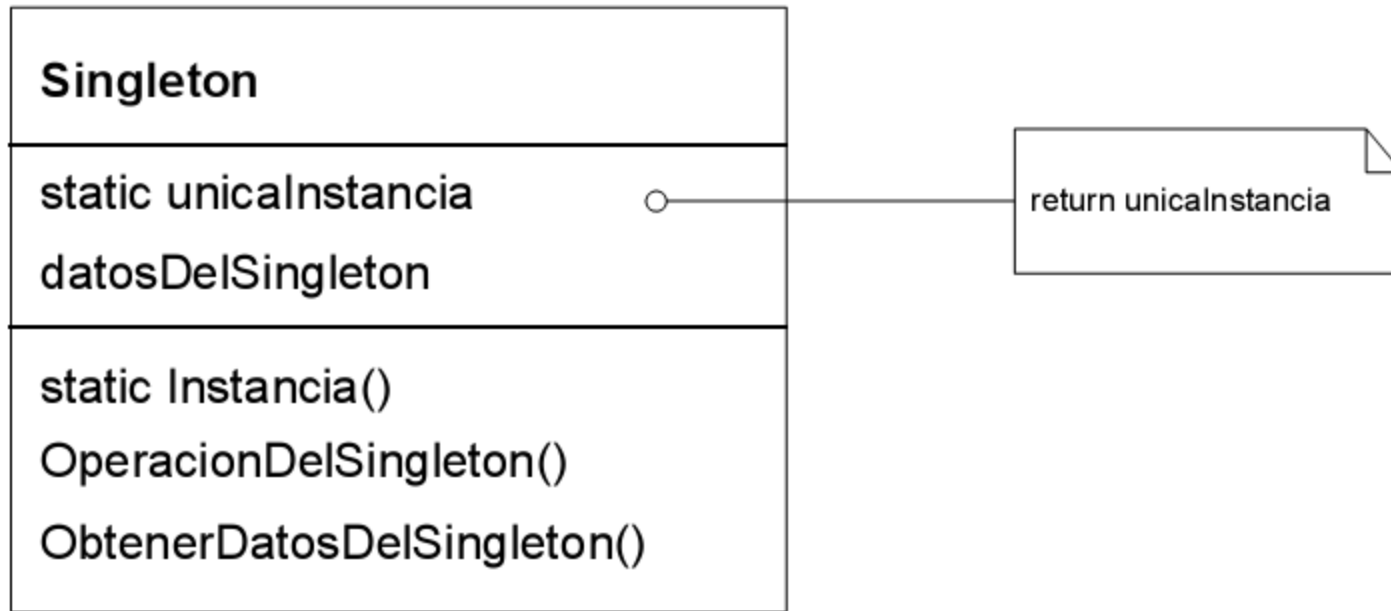
- Aplicabilidad: (utilizar cuando...)

Debe haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido.

SINGLETON –Ventajas e inconvenientes

- Acceso controlado a la única instancia.
- Espacio de nombres reducido: no hay variables globales.
- Puede adaptarse para permitir más de una instancia.

SINGLETON – Estructura



- **Participante:**
Singleton- define un método instancia que permite que los clientes accedan a su única instancia. Instancia es un método de clase estático.
- **Colaboración:**
Los clientes acceden a una instancia de un singleton exclusivamente a través de un método instancia de este.

SINGLETON – Implementación

- La **primera posibilidad** de implementar este patrón es con **una función que controle la inicialización o no** de un atributo de la función.
- **En python** teniendo en cuenta que **una función es un objeto** la podemos **asignar** un **atributo** como se de una clase se tratase.

SINGLETON – Implementación

- **Dependiendo del contexto** que nos encontremos podemos **dotar a la función de un parámetro** que indique si queremos **refrescar** el contenido de atributo.
- Por ejemplo, si utilizamos un singleton para descargar las cotizaciones del cambio de moneda de un banco, el singleton nos interesa para realizar una única conexión pero en momento dado podemos querer refrescar la información (en este caso mandamos el parámetro a True).

Ejemplo

- Implementar un singleton que registra la hora del sistema (esto la hará la primera vez).
Posteriormente si hacemos retardos con `time.sleep` y repetimos la petición obtendremos la misma hora.
- Cuando hacemos una petición con `refresh` se actualiza la hora.

Código (Singleton con función)

```
import time
```

```
def singleton(refresh=False):
```

```
    if refresh:
```

```
        singleton.instancia=""
```

```
    if singleton.instancia:
```

```
        return singleton.instancia
```

```
    # Asignar tiempo a la variable:
```

```
    singleton.instancia = str(time.strftime("%H:%M:%S"))
```

```
    print( "Singleton creado" )
```

```
    return singleton.instancia
```

```
# Asocia un atributo a la función (ojo la función se declara previamente)
```

```
singleton.instancia=""
```


Prueba

```
print ("Sin refresco, siempre la misma hora")
print (singleton())
time.sleep(1)
print ("siguiente llamada")
print (singleton())
print()
print ("Con refresco, cada llamada cambia la hora")
print (singleton(True))
time.sleep(1)
print (singleton(True))
```

```
Sin refresco, siempre la misma hora
Singleton creado
19:02:09
siguiente llamada
19:02:09

Con refresco, cada llamada cambia la hora
Singleton creado
19:02:10
Singleton creado
19:02:11
```

Singleton (con POO)

- Hacer uso del decorador **@staticmethod**
- Un método **static** se considera un método de clase.
- **NO es necesario construir** un objeto de la clase.
- Los métodos static **NO** tienen **self**.

Ejemplo

```
class Singleton(object):
```

```
    __instance = None
```

```
    @staticmethod
```

```
    def getInstance(refresh = False):
```

```
        if Singleton.__instance == None or refresh:
```

```
            # Se crea el singleton
```

```
            Singleton.__instance = str(time.strftime("%H:%M:%S"))
```

```
        return Singleton.__instance
```

```
# La llamada
```

```
print Singleton.getInstance()
```

```
print Singleton.getInstance(True)
```

Otra posible implementación

- El patrón Singleton en python se puede definir utilizando **clases anidadas**.
- La clase externa tiene el **CONTROL** de la creación de la instancias (la clase interna).

```
class OnlyOne:
    class __OnlyOne:
        def __init__(self, arg):
            self.val = arg
        def __str__(self):
            return `self` + self.val
    instance = None
    def __init__(self, arg):
        if not OnlyOne.instance:
            OnlyOne.instance = OnlyOne.__OnlyOne(arg)
        else:
            OnlyOne.instance.val = arg
    def __getattr__(self, name):
        return getattr(self.instance, name)

x = OnlyOne('sausage')
print x
y = OnlyOne('eggs')
print y
z = OnlyOne('spam')
print z
```

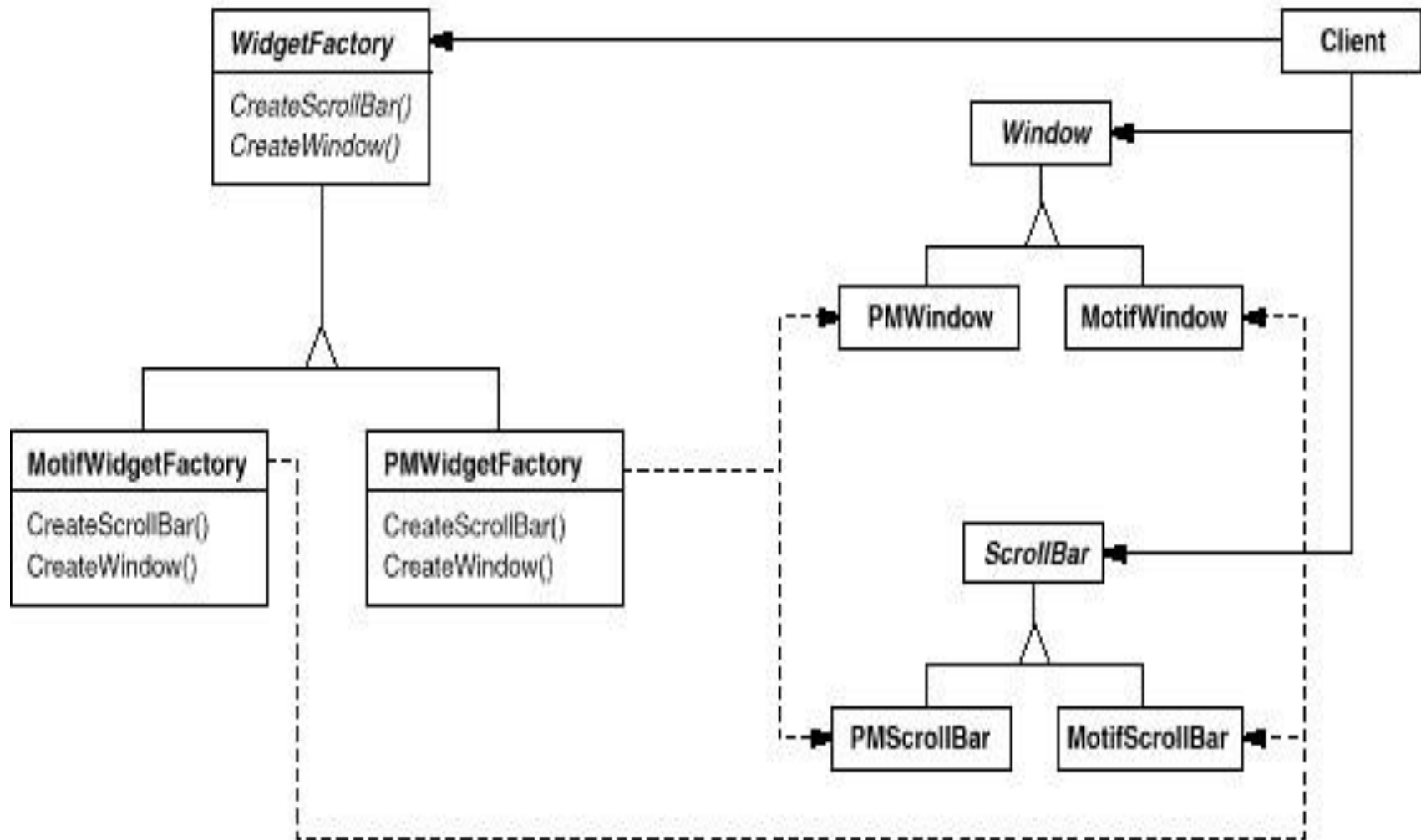
ABSTRACT FACTORY

- El patrón **Abstract Factory** ofrece una **interfaz para la creación de familias de productos relacionados o dependientes** sin especificar las clases concretas a las que pertenecen.
- Look-And-Feel múltiple: diseño de GUI en entorno de ventanas.
- Su objetivo es soportar múltiples estándares (MS-Windows, Motif, Open Look,...).
- Extensible para futuros estándares.
- Restricciones: cambiar el Look-and-Feel sin recompilar y cambiar el Look-and-Feel en tiempo de ejecución.

Ejemplo

- Ejemplo de partida: Creación de Widgets en aplicación cliente.
- Un almacén para interfaces gráficas que soporte distintos tipos de presentación (look-and-feel).
- Las aplicaciones cliente no deben cambiar porque cambie el aspecto de la interfaz de usuario.
- Los clientes no son conscientes de las clases concretas, sino sólo de las abstractas.
- Sin embargo los WidgetFactory imponen dependencias entre las clases concretas de los Widgets que contiene.

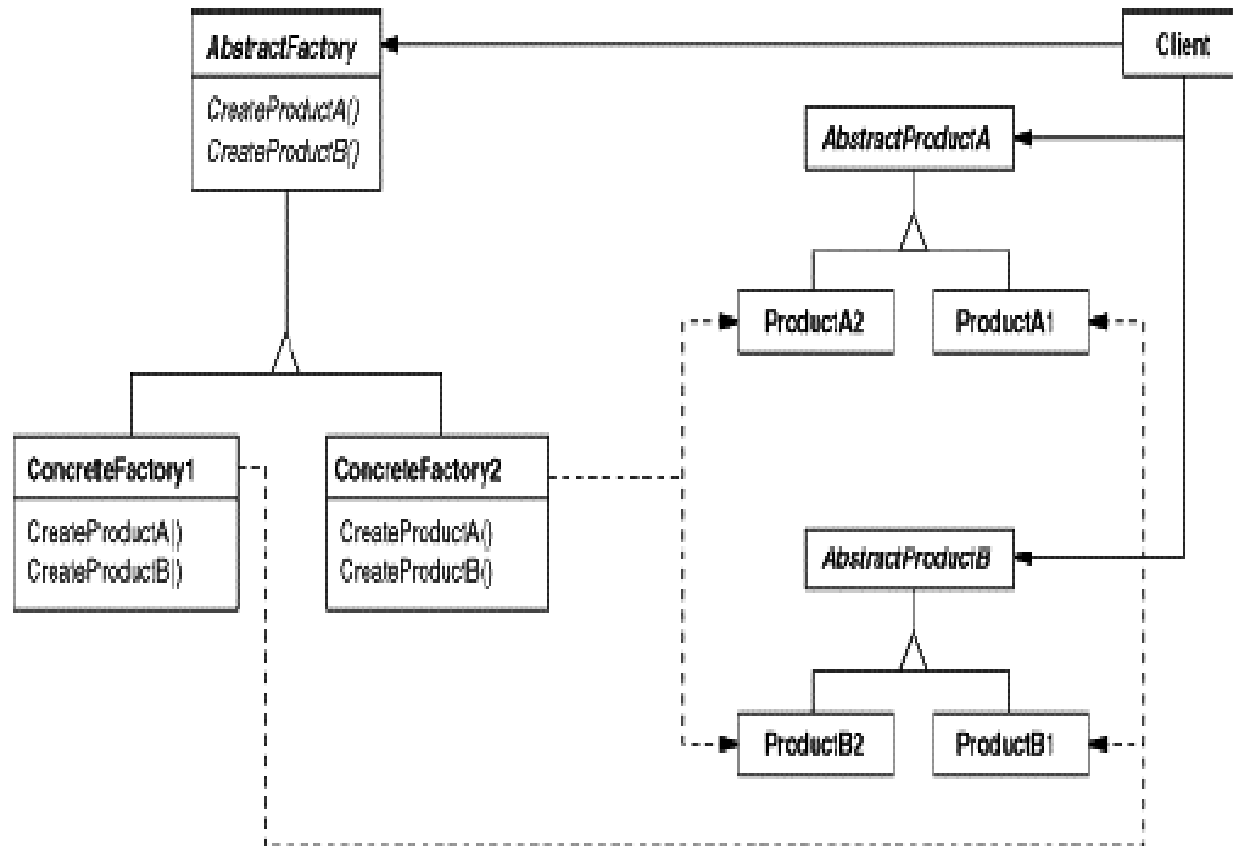
Ejemplo



Aplicaciones

- Un sistema debe ser independiente de los procesos de creación, composición y representación de sus productos.
- Un sistema debe ser configurado con una familia múltiple de productos.
- Una familia de productos relacionados se ha diseñado para ser usados conjuntamente (y es necesario reforzar esta restricción).
- Se quiere proporcionar una librería de productos y no revelar su implementación (simplemente revelando sus interfaces).

Estructura



Participantes

- **AbstractFactory** (WidgetFactory): Declara un interfaz para las operaciones de creación de objetos de productos abstractos.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory): Implementa las operaciones para la creación de objetos de productos concretos.
- **AbstractProduct** (Window, ScrollBar): Declara una interfaz para los objetos de un tipo de productos.
- **ConcreteProduct** (MotifWindow, MotifScrollBar): Define un objeto de producto que creará la correspondiente *Concrete Factory*, a la vez que implementa la interfaz de *AbstractProduct*.
- **Client**: Usa solamente las interfaces declaradas por las clases *AbstractFactory* y *AbstractProduct*.

Colaboraciones

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

Consecuencias

- Ventajas:
 - Aísla las clases de implementación: ayuda a controlar los objetos que se creen y encapsula la responsabilidad de creación.
 - **Hace fácil el intercambio de familias de productos sin mezclarse, permitiendo configurar un sistema con una de entre varias familias de productos:**
cambio de factory \Rightarrow cambio de familia.
 - Fomenta la consistencia entre productos.

Consecuencias

- Desventajas:
 - Puede ser difícil incorporar nuevos tipos de productos (cambiar **AbstractFactory** y sus factorias concretas).
 - Posible Solución:
 - Pasarle un parámetro a los métodos de creación de productos \Rightarrow clase abstracta común \Rightarrow necesidad de downcast \Rightarrow solución no segura.

Detalles de Implementación

- **Factorías como singletons**
 - Sólo una instancia de **ConcreteFactory** por familia de productos.
- Definir factorías extensibles
 - Añadiendo un parámetro en las operaciones de creación que indique el tipo de objeto a crear: mas flexible y menos seguro.
- ¿Cómo crear los productos? Utilizando un *Factory Method* para cada producto.

BUILDER

- El patrón Builder es similar al patrón Abstract Factory.
- Ambos patrones se utilizan para construir **objetos complejos que se componen de otros objetos**.
- Builder no solo proporciona métodos para construir un objeto complejo, también **permite la representación** del objeto.
- La construcción de objeto se delega a partes de este más pequeñas que después se ensamblan en un objeto final.

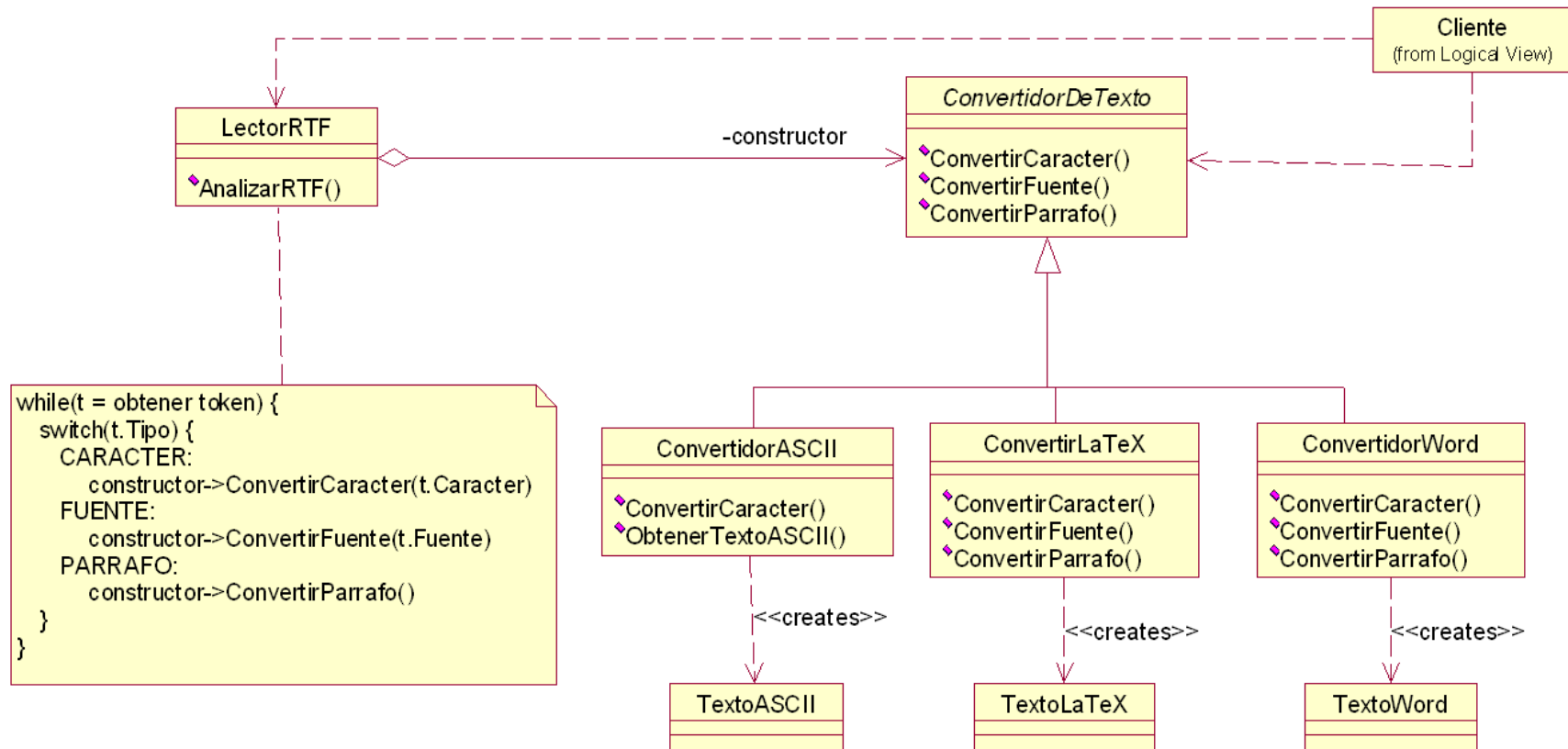
Ejemplo

- Por ejemplo, la construcción de coche implica la construcción de sus partes: Ruedas, Chasis, Motor y el ensamblaje de estas, dan como resultado un coche funcional y completo.
- Podemos tener distintos tipos de constructores para obtener distintos tipos de vehículos.

Builder

- Objetivo: Separa la construcción de un objeto complejo de su representación, de forma que puedan crearse distintas representaciones.
- También se le conoce con el nombre de constructor.
- Ejemplo: Convertir un fichero con formato RTF a otros formatos.

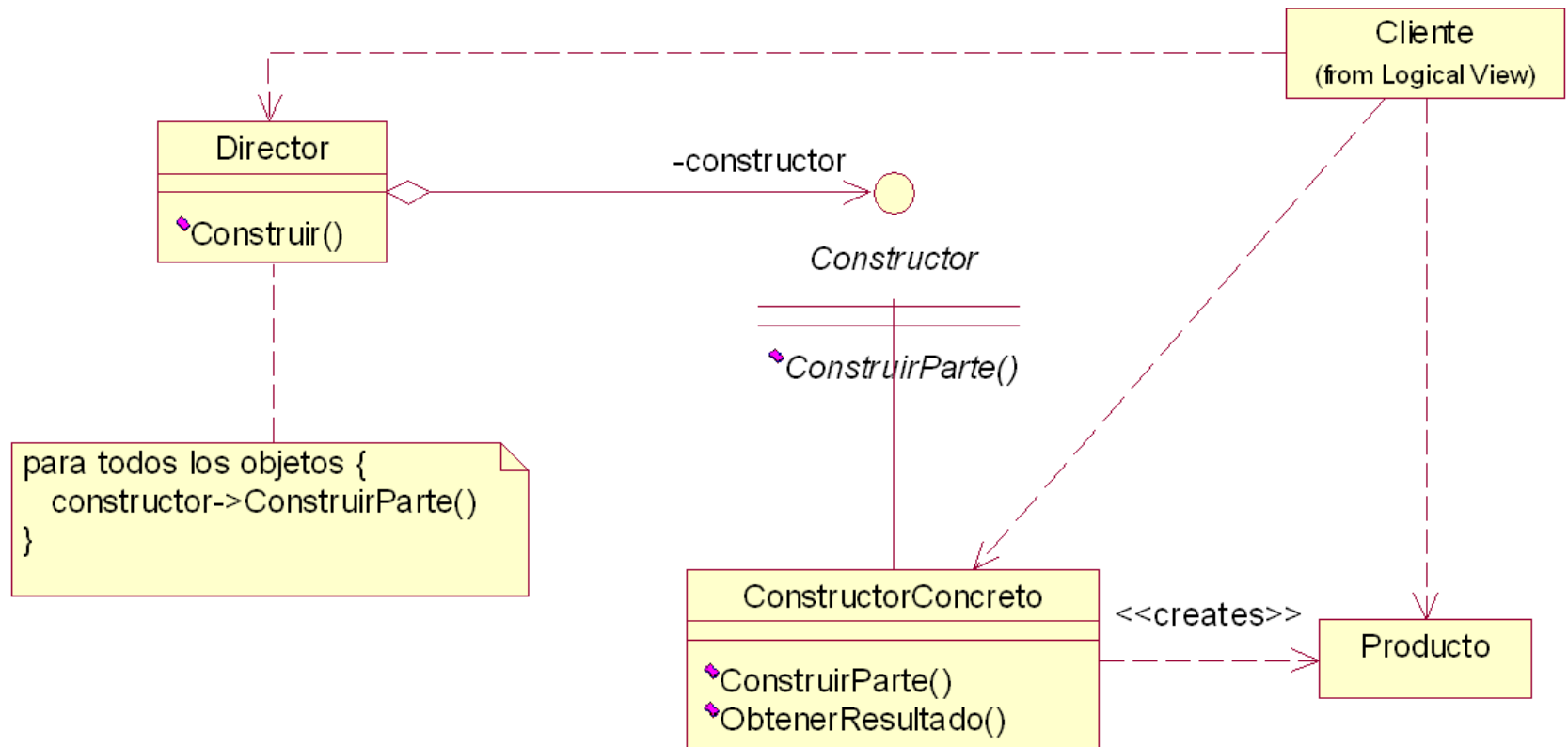
Ejemplo



Aplicabilidad

- El algoritmo para crear un objeto complejo debiera ser independiente de las partes que se compone dicho objeto y de cómo se ensamblan.
- El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.

Estructura



Participantes

- **Constructor** (ConvertidorDeTexto):
 - Especifica una interfaz abstracta para crear las partes de un objeto Producto.
- **ConstructorConcreto** (ConvertidorASCII, ConvertidorTeX, ConvertidorUtilDeTexto):
 - Implementa la interfaz Constructor para construir y ensamblar las partes del producto.
 - Define la representación a crear.
 - Proporciona una interfaz para devolver el producto (ObtenerTextoASCII, ObtenerUtilDeTexto).
- **Director**: (LectorRTF):
 - Construye un objeto usando una interfaz Constructor.
- **Producto**: (TextoASCII, TextoTeX, UtilDeTexto):
 - Representa el objeto complejo en construcción. El ConstructorConcreto construye la representación interna del producto y define el proceso de Ensamblaje.
 - Incluye las clases que definen sus partes constituyentes, incluyendo interfaces para ensamblar las partes del resultado final.

Consecuencias

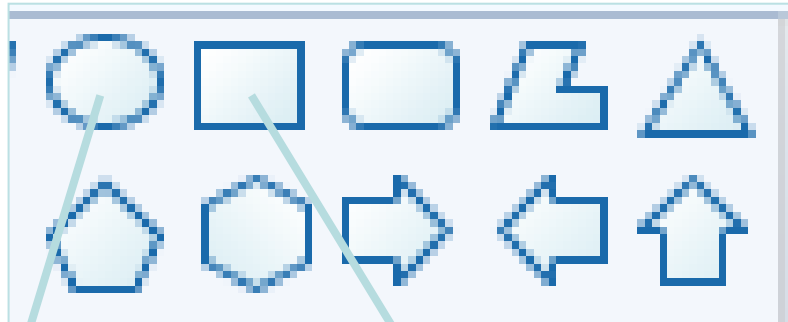
- El **constructor principal NO crea el objeto** define las operaciones para crear un objeto pero son las subclases del constructor las que realmente hacen la construcción.
- Tendremos un constructor específico de cada fabricante que hereda del constructor genérico.

PROTOTYPE

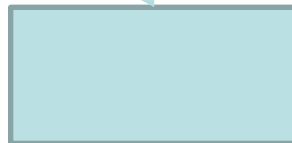
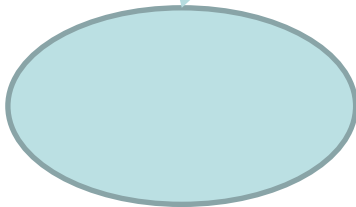
- Objetivo: Crear nuevos objetos clonando el original y después modificando el clon.
- Nombres: Prototipo.
- Ejemplo: Editor de figuras gráficas.
 - El usuario dispone de una paleta de figuras y la utiliza para arrastrar a la escena.
 - Cada que pincha en una figura se clona y se añade a la escena.

Ejemplo

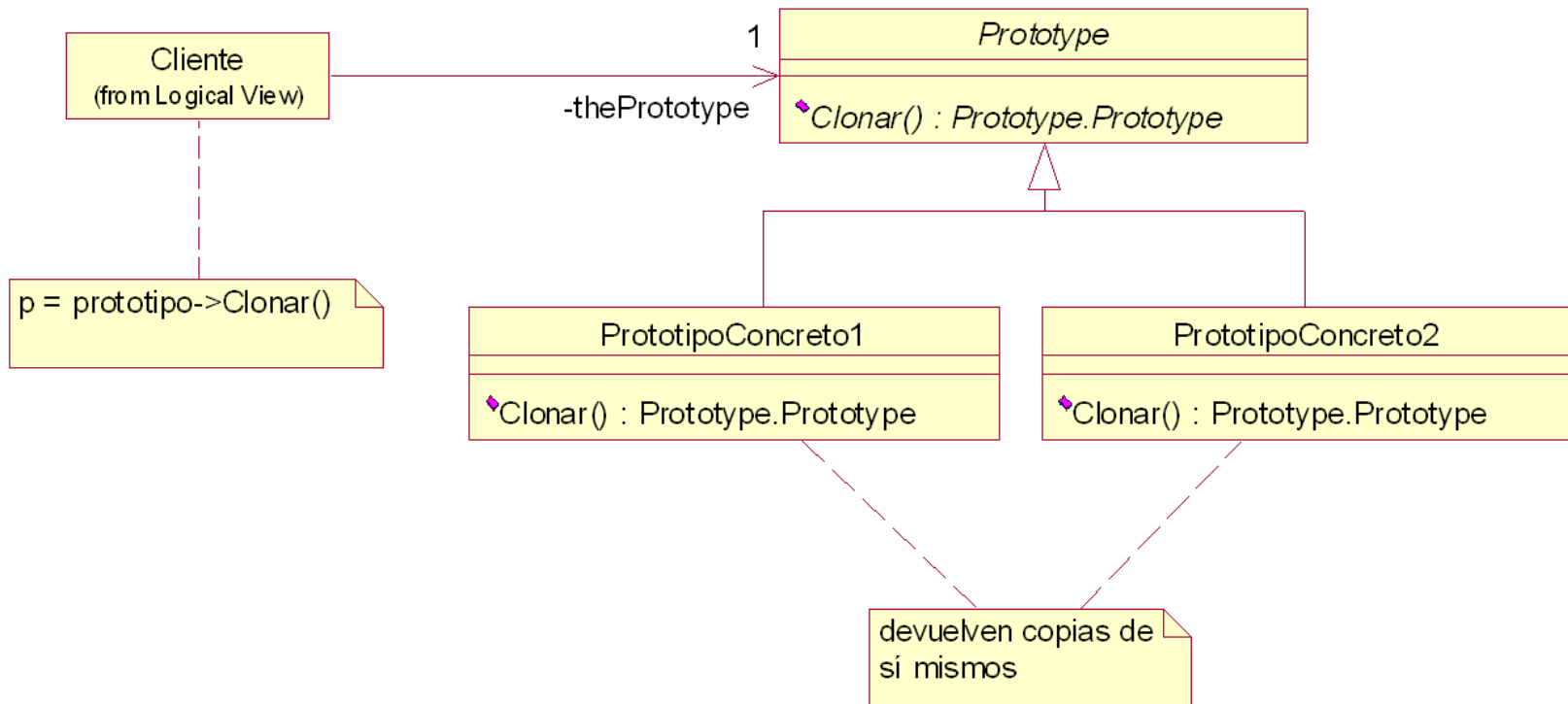
PALETA DE FIGURAS GRÁFICAS:



ESCENA



Estructura



Aplicabilidad

- Cuando las clases a instanciar sean especificadas en tiempo de ejecución.
- Para evitar jerarquías de clases de fábricas con jerarquía de clases de productos.
- Cuando las instancias de clases puedan tener un estado de entre un conjunto reducido.

Participantes

- Prototipo (Gráfico):
 - Declara la interfaz para clonarse.
- Prototipo Concreto (Circulo, Rectángulo, ...):
 - Implementa una operación para clonarse.
- Cliente (Herramienta Gráfica):
 - Crea un nuevo objeto pidiéndole a un prototipo que se clone.

Consecuencias

- **Añadir y eliminar productos en tiempo de ejecución.** El cliente puede instalar y eliminar prototipos en tiempo de ejecución.
- Especificar nuevos objetos modificando valores. Para sistemas dinámicos permiten definir un comportamiento nuevo mediante la composición de objetos.
- Especificar nuevos objetos variando la estructura.
- **Reduce la herencia:** El patrón Factory Method suele producir una jerarquía de clases Creador que es paralela a la jerarquía de clases de productos.
 - Con el patrón prototype podemos clonar un prototipo en vez de decirle a un método de fabricación que cree un nuevo objeto.
- Configurar dinámicamente una aplicación con clases. Algunos entornos permiten cargar clases en tiempo de ejecución.

Inconvenientes

- El principal inconveniente es que cada subclase del prototipo debe implementar la operación clonar, y la implementación puede ser compleja.

En python

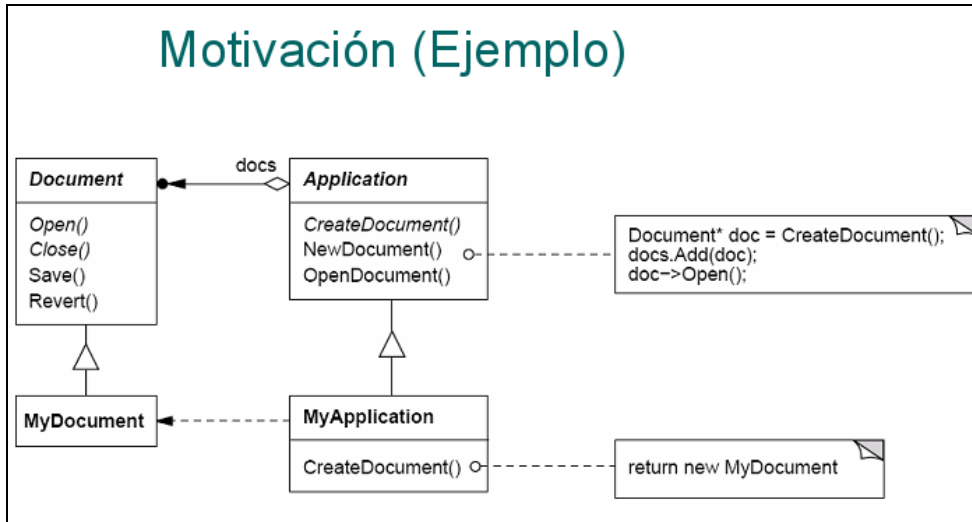
- Disponemos del módulo **copy** y el método **copy** que copia un objeto.
- Viene hacer las veces de un método clone.

VER EJEMPLOS - PRACTICA

FACTORY METHOD

- FACTORY METHOD: Define la interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan que clase concreta necesitan instancias.
- También conocido como: Método de fabricación.
- Es una simplificación del Abstract Factory, en la que la clase abstracta tiene métodos concretos que usan algunos de los abstractos; según usemos una u otra hija de esta clase abstracta, tendremos uno u otro comportamiento.

FACTORY METHOD – Motivación



- Sea un **framework** para la construcción de editores de documentos de distintos tipos.

- ¿Cuál es el problema?

Se nos presenta el dilema de que el framework es quien sabe, a través de su clase *Application*, cuándo se debe crear un nuevo documento, pero no sabe qué documento crear.

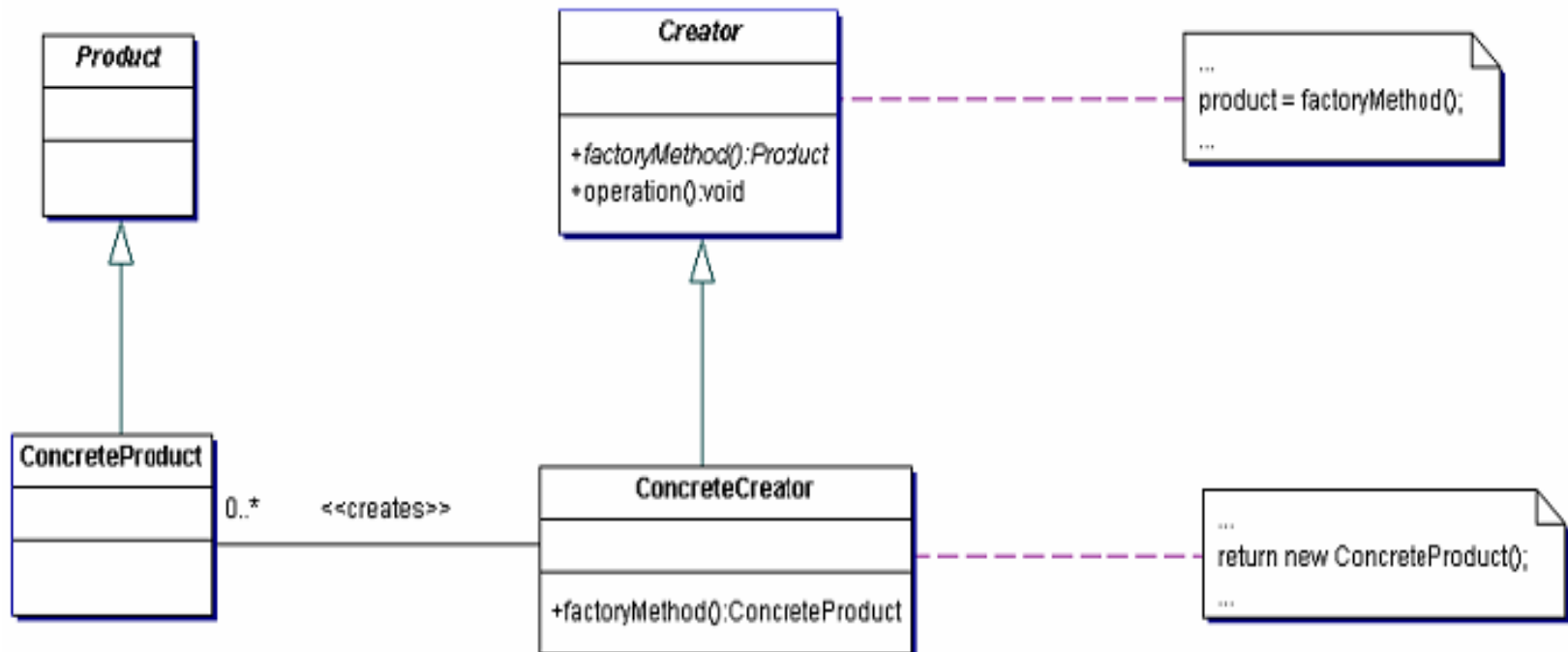
- Solución:

Encapsular el conocimiento de cuál es la subclase concreta del documento a crear y mover ese conocimiento fuera del framework

FACTORY METHOD – Aplicabilidad

- **Una clase no puede anticipar la clase de objeto que debe crear.**
- Una clase quiere que sus subclases especifiquen los objetos a crear.
- Hay clases que delegan responsabilidades en una o varias subclases.

FACTORY METHOD – Estructura



FACTORY METHOD – Participantes

- **Product:** Define la interfaz de los objetos creados por el método de fabricación (FactoryMethod()).
- **Concret Product:** Implementa la interfaz Product.
- **Create:** Declara el método de fabricación, que devuelve un objeto de tipo product. Puede llamar a dicho método para crear un objeto producto.
- **ConcretCreator:** Redefine el metodo de fabricación para devolver un objeto concretProduct.

FACTORY METHOD – Colaboración

- **Colaboración**: El creador se apoya en sus subclases para definir el método de fabricación que devuelve el objeto apropiado.

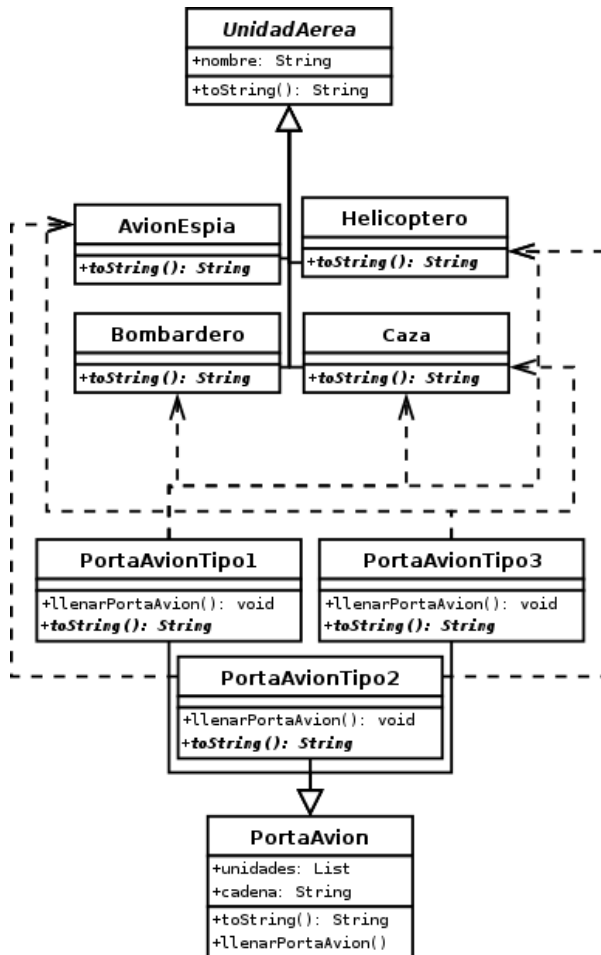
FACTORY METHOD –Ventajas e inconvenientes

- Elimina la necesidad de introducir clases específicas en el código del creador.
- Solo maneja la interfaz Product, por lo que **permite añadir cualquier clase ConcretProduct** definida por el usuario.
- Tener que crear una subclase de Creator en los casos en los que esta no fuera necesaria de no aplicar el patrón.

FACTORY METHOD – Implementación

- Convenios de **nominación**: Se suele usar el prefijo ***create***.
- Diversas clases se suele implementar como un Singleton.
- El Factory Method parametrizado (Una variante del patrón que permite al método de fabricación crear varios tipos de productos) protege de la variación del tipo de producto concreto.

FACTORY METHOD – Ejemplo



- En este ejemplo de porta-aviones simplemente creamos 3 porta-aviones de 3 tipos distintos, y nos olvidamos del tipo de aviones que tendrá cada uno de ellos dejando a las subclases barcoX decidir que tipo de aviones tendrán, ya no tenemos la responsabilidad de saber que aviones tiene cada barco .

Patrones Estructurales

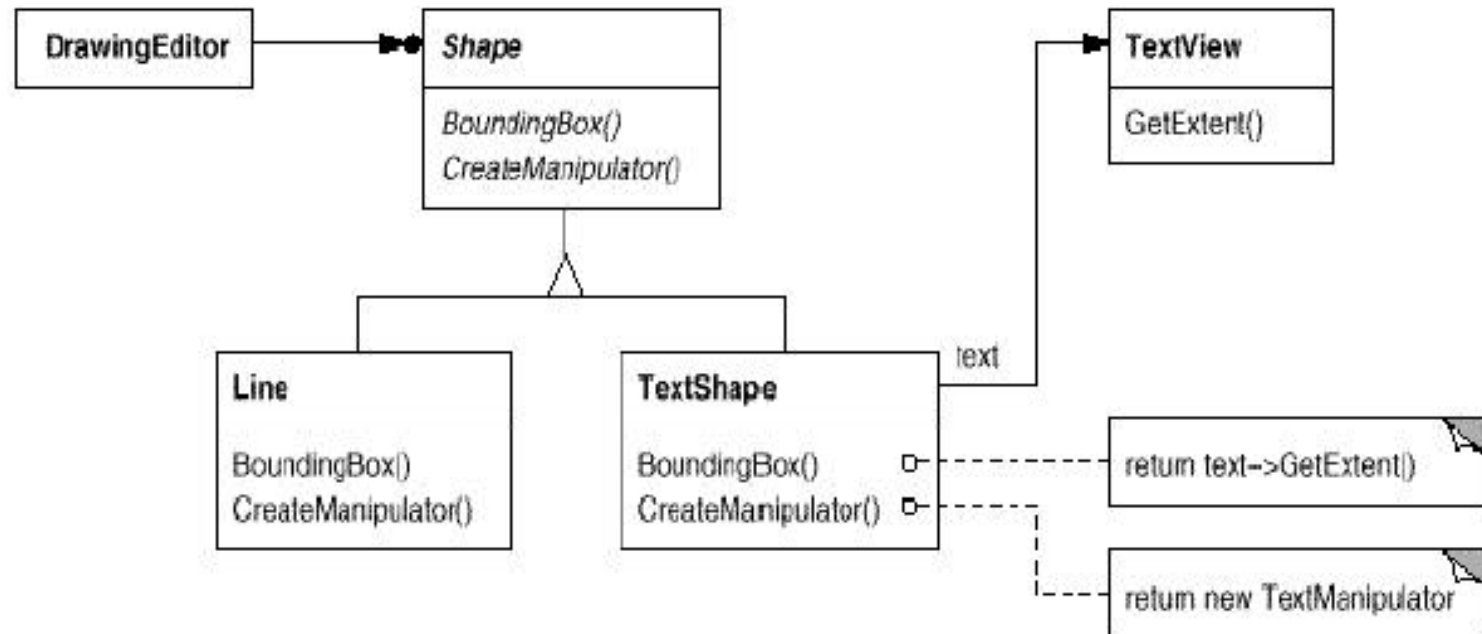
Patrones Estructurales

- Los patrones Estructurales se ocupan de cómo se combinan las clases y los objetos para formar estructuras mas grandes.
 - Estructurales **de Clase**: hacen uso de la herencia para componer interfaces o implementaciones.
 - Estructurales **de Objetos**: describen formas de componer objetos para obtener una nueva funcionalidad. Añaden flexibilidad al poder cambiar la composición en tiempo de ejecución.

Adapter

- **Convierte la interfaz de una clase en la interfaz de otra que espera un cliente:**
 - Muy útil si por ejemplo no se tiene acceso al código fuente de la interfaz inicial o **NO se puede modificar.**
- Se puede implementar de **dos formas**:
 - Por Composición.
 - Por Herencia múltiple. (recibe el nombre de clase **Adaptadora**)
- Es uno de los patrones denominados “wrappers” → **Envoltorio**, (otro por ejemplo es el patrón **Decorator**)

Ejemplo



“**TextShape**” es un adaptador para que un “**TextView**” se pueda considerar un “**Shape**”

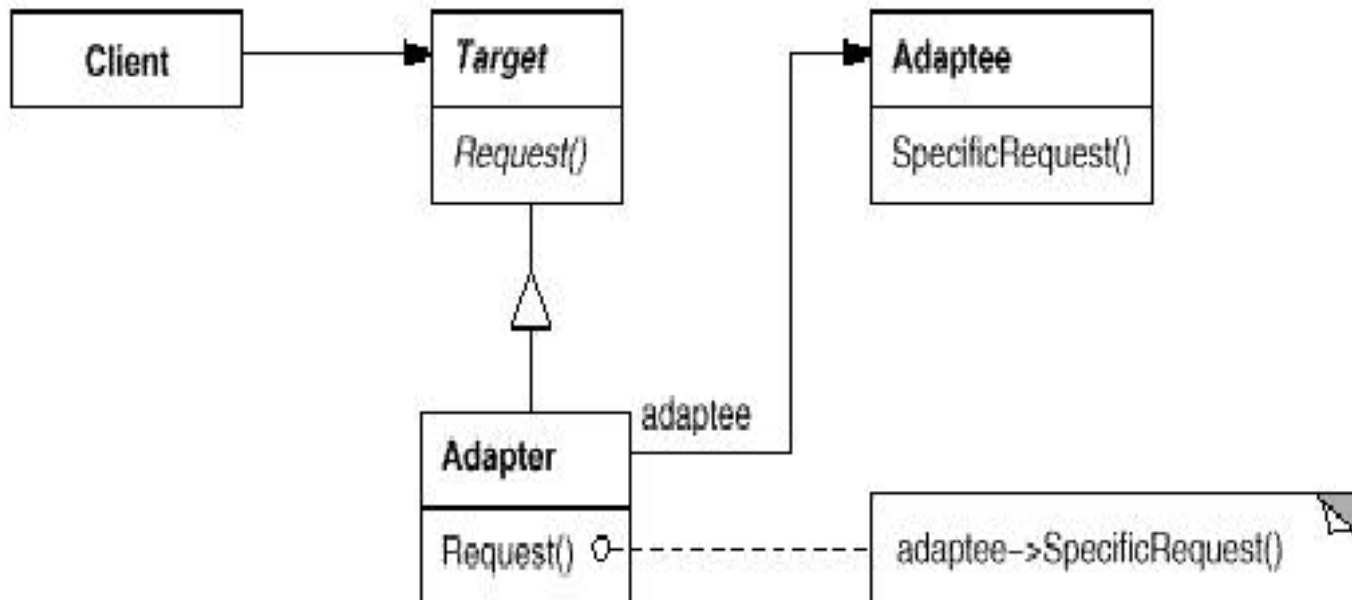
Aplicabilidad

- Este patrón debería usarse cuando:
 - Se quiere usar una clase existente y su interfaz no concuerda con la que necesita.
 - Pero la clase existente obviamente se puede utilizar (nos vale).
 - Se quiere crear una clase reutilizable que coopere con clases no relacionadas.

Participantes

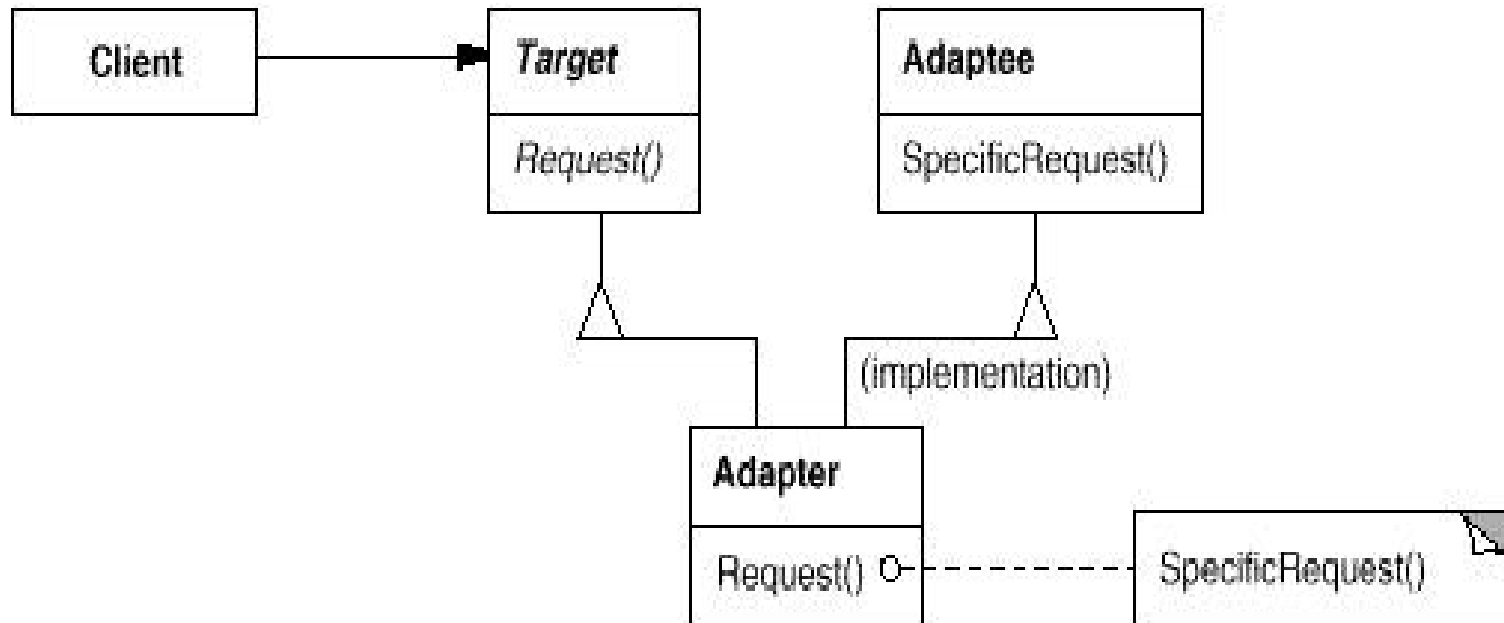
- Objetivo (Shape):
 - Define la interfaz específica del dominio que usa el cliente.
- Cliente (DrawingEditor):
 - Colabora con objetos que se ajustan a la interfaz Objetivo.
- Adaptable (TextView):
 - Define la interfaz existente que necesita ser adaptada.
- Adaptador (TextShape):
 - Adapta la interfaz de Adaptable a la interfaz Objetivo.

Estructura I



Se puede utilizar “**composición**”

Estructura II



... O alternatively **herencia múltiple**

Ejemplo

- Disponemos de una clase que representa un vector en 3D y necesitamos una clase VectorPlano que implemente una interface (clase con todos los métodos sin implementar) para un Vector2D.
- **Posibilidades:**
 1. La clase VectorPlano hereda de Vector2D y de Vector3D.
 2. La clase VectorPlano hereda de Vector2D y se compone de Vector3.

Fachada

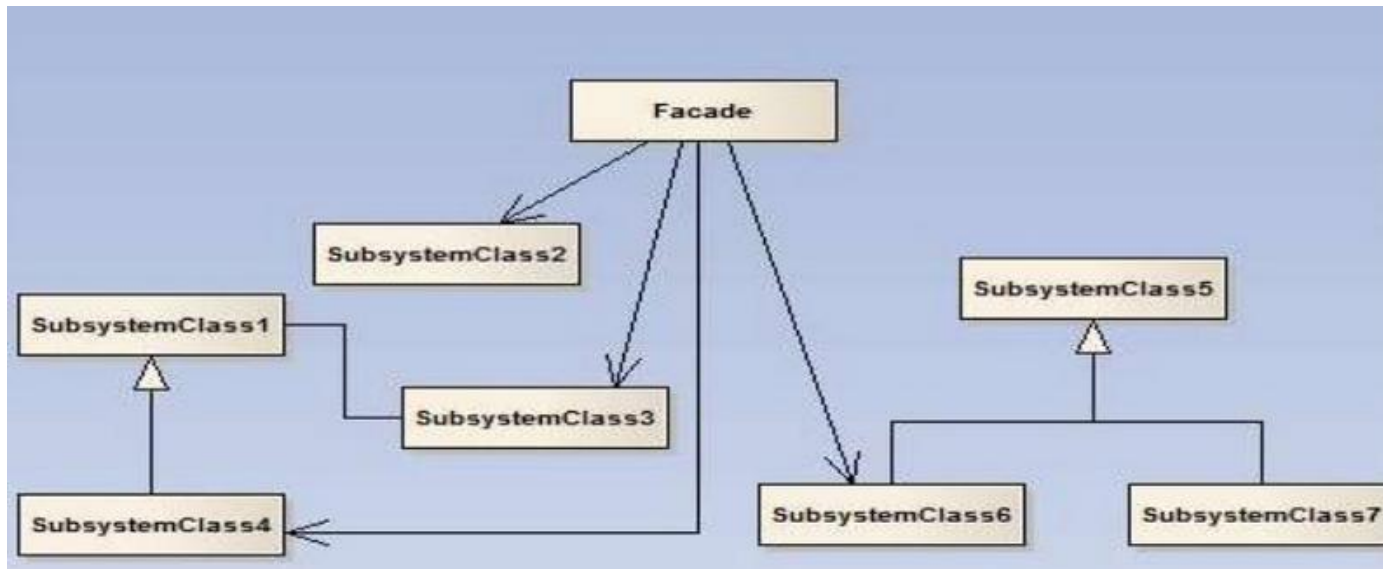
- Es otro patrón de diseño de tipo estructural.
- Sirve para proveer de una interfaz unificada sencilla que haga de intermediaria entre un cliente y una interfaz o grupo de interfaces mas complejas.

Para que se aplica

- Cuando un cliente necesita acceder a una parte de una funcionalidad de un sistema mas complejo.
- El problema se puede solucionar definiendo una interfaz que permita acceder a solamente esa funcionalidad.
- Se crea un código sencillo que interactúe entre el cliente y la librería interna. **Hace las veces de intermediario.**

Estructura

- Simplifica el acceso a un conjunto de clases proporcionando una única clase que todos utilizan para comunicarse con dicho conjunto de clases.



Ejemplo

- Para otorgar una Hipoteca a un cliente hay que realizar una serie de comprobaciones:
 - Comprobar en el banco si el cliente tiene ahorros suficientes.
 - Comprobar en el sistema de créditos si hay crédito positivo.
 - Comprobar en el sistema de préstamos si el cliente tiene impagos.

Aplicación

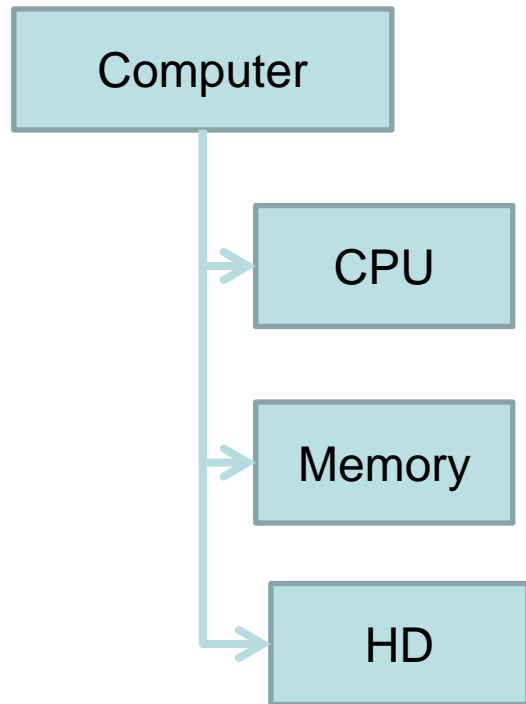
- El patrón Fachada se encarga de realizar todas las comprobaciones necesarias con todos los sistemas necesarios.
- **El cliente interactúa con la Fachada a través de un simple método que le dice si se le concede o no la Hipoteca.**

Mas ejemplos

- El desarrollo de un compilador también puede ser visto como un patrón Fachada.
- Un proceso de compilación se divide en varias fases y estas son complejas:
 - Análisis léxico, con la generación de Tokens.
 - Análisis sintáctico, con la construcción del árbol sintáctico.
 - Análisis semántico, generación de código, etc.
- La **fachada**, podría ser una clase que se pone en comunicación con las distintas fases pero suministra al **Ciente un método sencillo que se llama compilar y recibe el fichero fuente**.

Otro Ejemplo

*El arranque de un ordenador:
Es una operación compleja ...*



EJEMPLO FACHADA

```
class CPU(object):  
    def freeze(self): pass  
    def jump(self, position): pass  
    def execute(self): pass
```

```
class Memory(object):  
    def load(self, position, data): pass
```

```
class HardDrive(object):  
    def read(self, lba, size): pass
```

FACHADA

```
class Computer(object):  
    def __init__(self):  
        self.cpu = CPU()  
        self.memory = Memory()  
        self.hard_drive = HardDrive()  
  
    def start_computer(self):  
        self.cpu.freeze()  
        self.memory.load(0, \  
            self.hard_drive.read(0, 1024))  
        self.cpu.jump(10)  
        self.cpu.execute()
```

Utilizar Fachada cuando

1. Se quiera proporcionar una interfaz sencilla para un subsistema complejo.
2. Se quiera desacoplar un subsistema de sus clientes y de otros subsistemas, haciéndolo mas independiente y portable.
3. Se quiere dividir los sistemas en niveles: las fachadas serían el punto de entrada de cada nivel.

Decorator

- Se conoce también como **Wrapper**.
- Motivación:
 - Añadir responsabilidades a un objeto, pero no a toda la clase.
 - La herencia no es flexible, **añade nuevas responsabilidades pero de una forma estática**.
 - La solución consiste en rodear a un objeto con otro objeto que es al que se le añade la responsabilidad.
 - **El nuevo objeto es el decorador.**

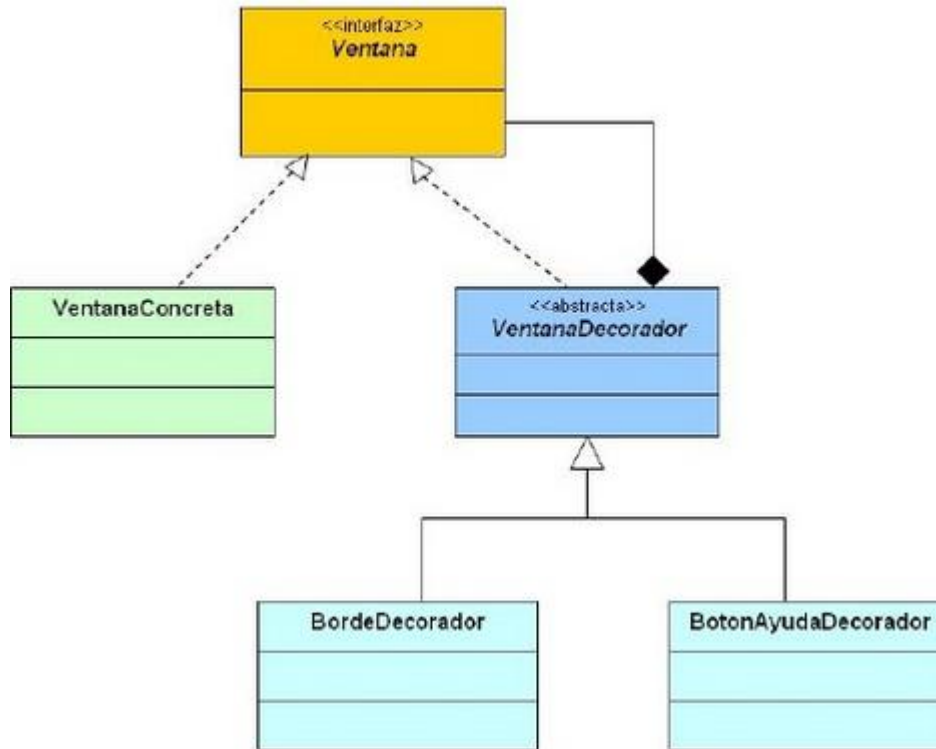
Se aplica cuando

- Añadir responsabilidades a objetos individuales dinámicamente sin afectar a otros objetos.
- Agregar responsabilidades que pueden ser retiradas.
- Cuando no es práctico añadir responsabilidades por medio de la herencia.

Ejemplo

- Partimos de una clase que representa una **Ventana gráfica** (*Lo vamos a simular con texto*).
- Queremos añadir una funcionalidad que sea un borde alrededor.
 - Aplicando herencia podríamos obtener **VentanaConBorde**.
- Si después queremos tener la posibilidad de añadir un botón de ayuda podríamos crear la clase **VentanaConBotonDeAyuda**.
 - **NO** cubre la posibilidad de tener ventanas con borde y con botón, podríamos seguir heredando, pero tampoco cubrimos todas las posibilidades: (multiplica las clases).
 - Ventana, VentanaConBorde, VentanaConBotonDeAyuda y VentanaConBordeYBotonDeAyuda.
 - **Para n funcionalidades se necesitan 2^n clases.**

Solución



Solución

- Se crea a partir de una ventana la subclase VentanaDecorator y, heredando de ella BordeDecorator, y BotonDeAyudaDecorator.
- VentanaDecorator encapsula el comportamiento de Ventana y utiliza **composición recursiva** para añadir todos los decoradores que necesitemos.

Solución

- En el main:
 - Podemos crear ventana de muchos tipos.
 - Aplicando composición recursiva.

```
if __name__ == "__main__":
```

```
    ventanaConDobleBorde2 = BordeDecorator(BordeDecorator(Ventana()))
```

```
    ventanaConDobleBorde2.dibujar()
```

- En tiempo de ejecución podemos crear Ventanas de distintos tipos:

Patrones de Comportamiento

Patrones de Comportamiento

- Los patrones de Comportamiento tienen que ver con Algoritmos y con la asignación de responsabilidades a objetos.
- Describen no solo patrones de clases y objetos, sino también patrones de comunicación entre ellos.

Estrategia

- El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.
- Permite seleccionar el algoritmo a aplicar.

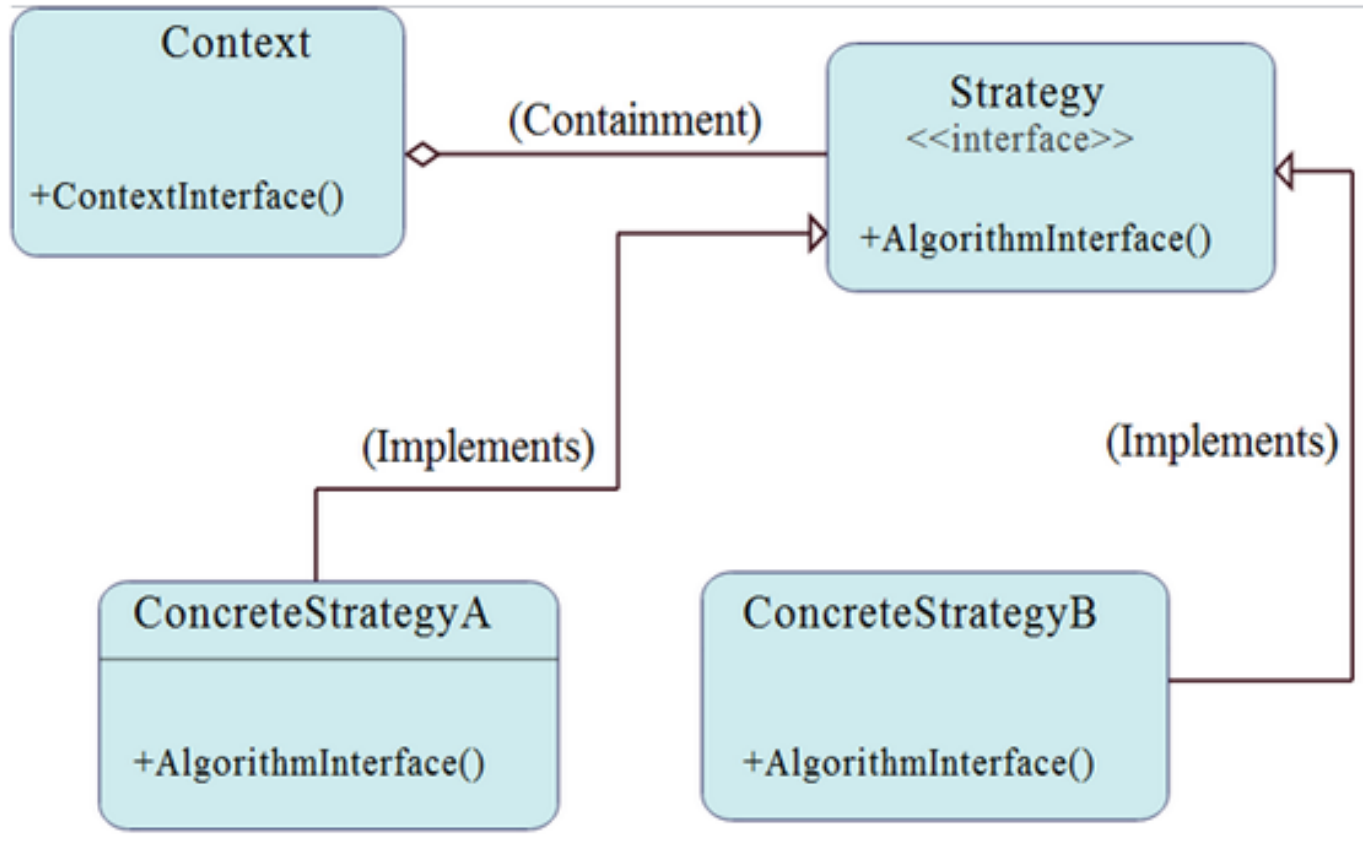
Estrategia

- Ejemplo real:
 - Objetivo ir al Aeropuerto
 - Tenemos 3 posibilidades → 3 Estrategias
 - Voy en **transporte público**: barato, algo más lento, tengo que transportar mi equipaje.
 - Voy en **coche particular**: lo más rápido, algo más caro, donde aparco el coche?
 - Voy en un **taxi**: cómodo, más caro.
 - Seleccionamos un modo de transporte según el tiempo del que dispongamos, si lo queremos más caro o barato, etc.

Cuando se aplica

- Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es **candidato** a utilizar el **patrón estrategia**.

Esquema UML



Participantes

- **Contexto** (*Context*) : Es el elemento que usa los algoritmos, por tanto, **delega en la jerarquía de estrategias**. Configura una estrategia concreta mediante una referencia a la estrategia necesaria.
- **Estrategia** (*Strategy*): Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el contexto para invocar a la estrategia concreta.
- **EstrategiaConcreta** (*ConcreteStrategy*): Implementa el algoritmo **utilizando la interfaz** definida por la estrategia.

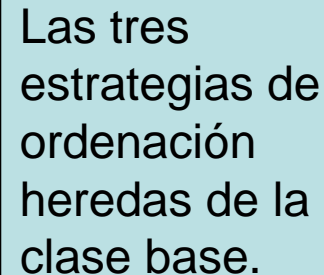
Pasar información

- Cuando es necesario el intercambio de información entre estrategia y contexto.
- Este **intercambio** puede realizarse de dos maneras:
 - Mediante **parámetros en los métodos** de la estrategia.
 - **Mandándose, el contexto**, a sí mismo a la estrategia.

Ejemplo

- Disponemos de una colección y lo podemos ordenar por distintos métodos de ordenación: burbuja, inserción directa y quicksort.

```
class InterfaceOrdenacion:  
    def ordenar(colección):  
        pass
```



Las tres
estrategias de
ordenación
heredas de la
clase base.

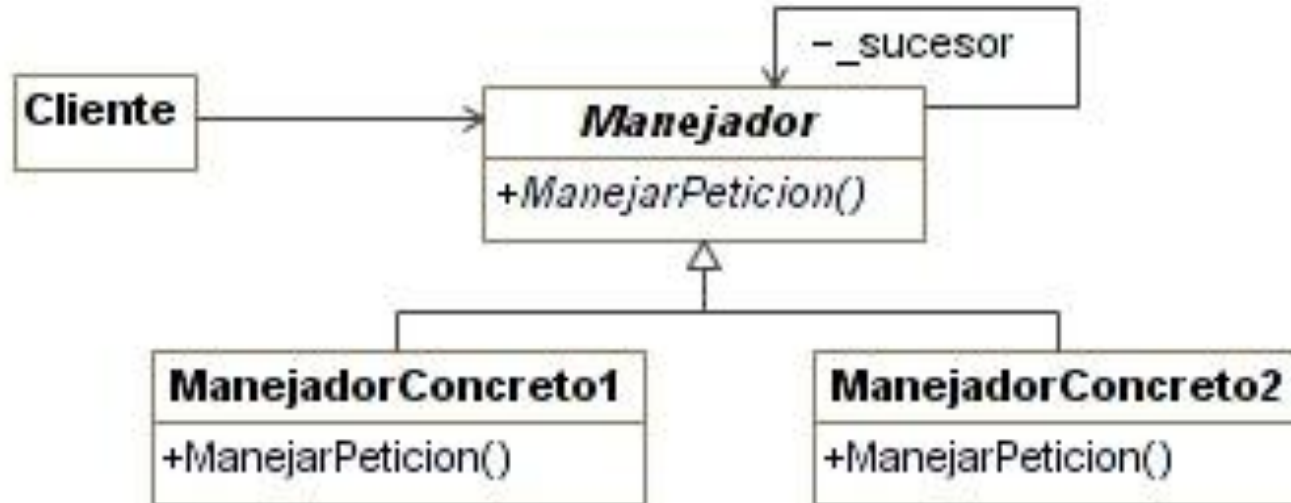
El contexto es el que selecciona la estrategia, método de ordenación a aplicar **Se puede combinar con Template para ampliar los tipos de algoritmos.**

VER EJEMPLO

Chain of Responsibility

- Es un patrón de comportamiento que evita acoplar el emisor de una petición a su receptor dando a más de un objeto la posibilidad de responder a una petición.
- Se encadenan los receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.
- Este **patrón es utilizado a menudo en el contexto de las interfaces gráficas** de usuario donde un objeto puede contener varios objetos. Según si el ambiente de **ventanas genera eventos, los objetos los manejan o los pasan.**

Esquema UML



Un objeto elige si procesa la petición o la pasa a su sucesor. La cadena de sucesores se configura desde fuera.

Nos podría valer para clasificar información, cada manejador podría realizar una serie de comprobaciones si no lo puede Clasificar lo manda a su sucesor.

Se usa cuando

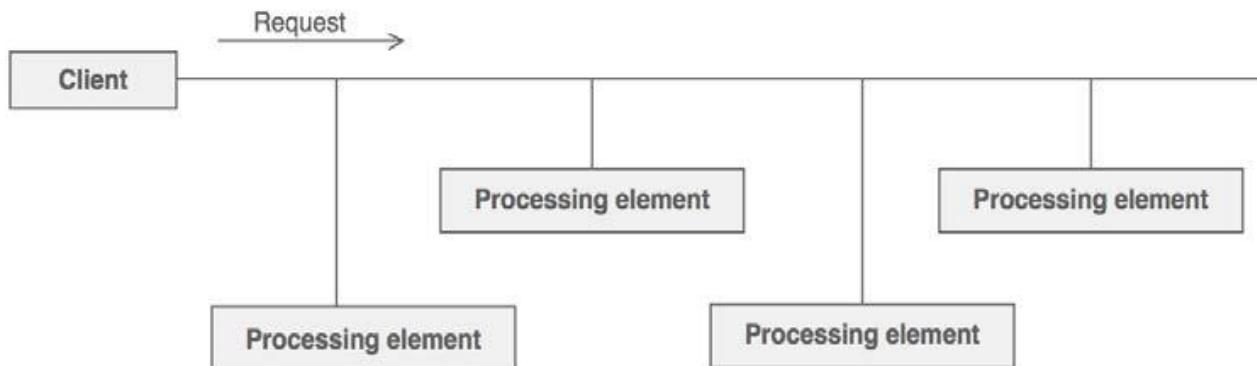
- Hay más de un objeto que puede manejar una petición, y el manejador no se conoce a priori, sino que debería determinarse automáticamente.
- Se quiere enviar una petición a un objeto entre varios sin especificar explícitamente el receptor.
- El conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

Ejemplo

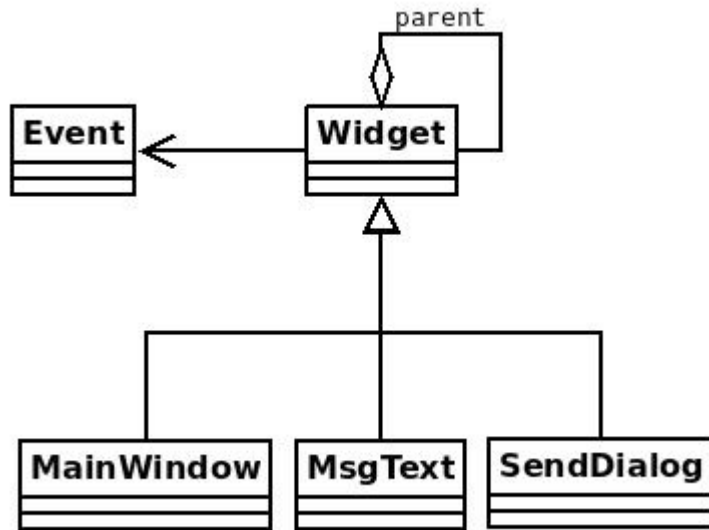
- En una red de ordenadores **TODOS** participan en la red pero solo atienden a la petición a los que se le ha hecho la petición y si no la vuelven a transmitir hasta que esta llegue al receptor.
 - Configuración en paso de testigo. Solo lo recoge el destinatario.

Esquema

- **El cliente SOLO tiene conocimiento del primer elemento de procesamiento.**
- Le envía la petición a este, si es para él la atiende, si no la transmite al siguiente elemento de procesamiento.



Implementación



- Generamos eventos en el sistema y se los lanzamos a las subclases de Widget.
- Si tienen un handler para ese evento lo atiende si no, lo envían por la cadena de responsabilidad.

Montar la cadena de responsabilidad

mw = MainWindow()

sd = SendDialog(**mw**)

msg = MsgText(**sd**)