

1. Modelos de objetos

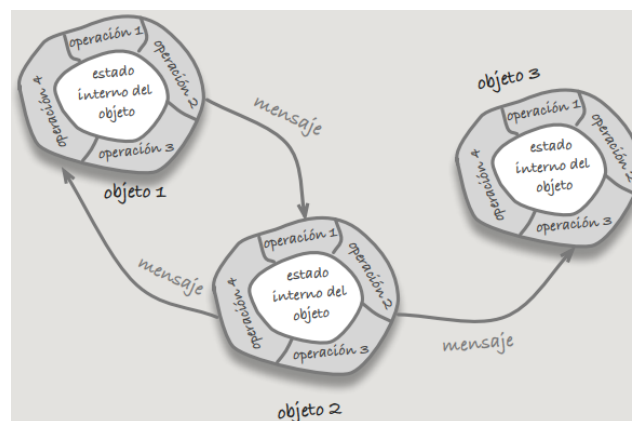
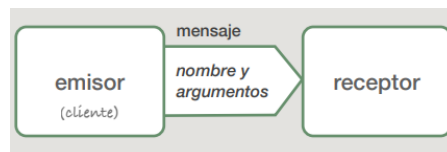
1.1 Diseño orientado a objetos

El **análisis** consiste en la investigación del problema y sus requisitos, mientras que el **diseño** consiste en una solución conceptual que satisface dichos requisitos.

El **diseño orientado a objetos** enfatiza la definición de objetos software y sus interrelaciones. Este diseño se puede presentar a través de diagramas UML. El proceso a seguir sería:

- a) Analizar el problema.
- b) Realizar un diseño orientado a objetos.
 - a. Este sistema de objetos satisface los requisitos.
 - b. Ha de contener documentación para el comportamiento público de los objetos.
 - c. Cómo se comunican entre sí.

A diferencia de la programación estructurada, que separa los datos de las rutinas que los manipulan, en el **OOD los objetos colaboran entre sí** con unos determinados métodos que los caracterizan. Estas operaciones se llevan a cabo cuando el objeto recibe una **petición o mensaje del cliente**. El estado interno del objeto está encapsulado, solo se puede cambiar a través de las operaciones, no se puede acceder directamente.



1.2 Interfaz de un objeto

Una **interfaz** es el conjunto de todas las signaturas públicas definidas por el objeto (nombre del método, parámetro, valor de retorno).

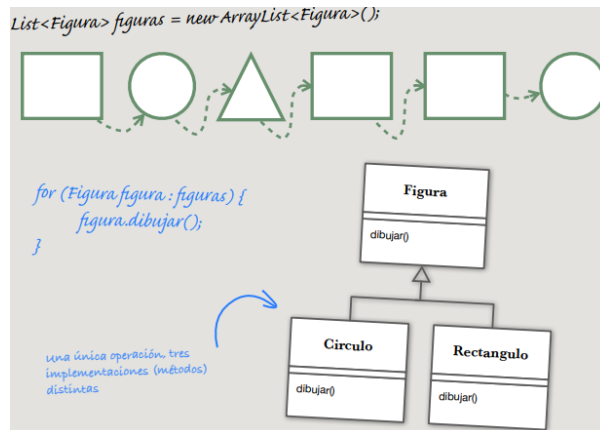
El **tipo** es el nombre de la interfaz. Es el conjunto de peticiones a las que un objeto puede responder. Un objeto puede tener muchos tipos, y muchos objetos distintos pueden pertenecer a un distinto tipo. Un tipo es subtipo de otro si su interfaz contiene la interfaz de su supertipo ('herencia').

1.3 Enlace dinámico y polimorfismo

El enlace dinámico es la **asociación en runtime** que tiene lugar entre la petición a un objeto y una de sus operaciones.

Esto permite sustituir en tiempo de ejecución un objeto por otro bajo la misma interfaz.

El polimorfismo es la **capacidad de 2+ tipos de objetos de responder al mismo mensaje, cada uno con su propia manera**.



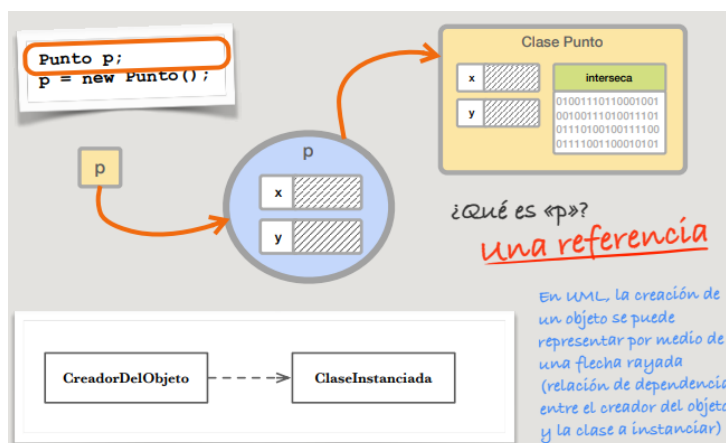
1.4 Clases, objetos y atributos

Una clase constituye la **implementación de los métodos** de un objeto de un determinado tipo.

La clase es una de las formas de crear tipos en Java. Para definir una clase:

- Determinar el nombre de la clase.
- Determinar los mensajes que pueden recibir:
 - Acciones: ordenar al objeto hacer algo.
 - Consultas: pedir al objeto alguna información.
- Implementación de los mensajes.

Cuando tenemos la clase creada, podemos crear los objetos: almacenamos en memoria las variables del objeto y asocia las operaciones de la clase a esos datos. **Se puede acceder al objeto guardado en memoria a través de un identificador que lo apunta en memoria, su referencia**.



Existen ciertos síntomas de una mala praxis en el diseño orientado a objetos, como si fuese programación estructurada:

- Métodos muy largos → no hay delegación ni métodos privados.
- Abundancia de llamadas a métodos de la propia clase → se guarda toda la funcionalidad en un solo objeto.
- Típica clase 'Util' → no se ha encontrado dónde asignar estas responsabilidades.

Respecto a los **datos del objeto**, estos se guardan en los **atributos**, que representan el estado interno del objeto.

Importante: no se ha de crear clases pensando en los atributos que poner. Los atributos son **PRODUCTO** del diseño de una clase de una determinada manera. Cada vez que la clase cambie, cambiarán también los atributos probablemente. Son usados exclusivamente para apoyar la implementación de los mensajes.

Otro fallo con respecto a los atributos: no confundir atributos con variables locales, pensar bien qué es realmente ese dato (¿afecta al resto?).

En resumen: lo importante de los objetos son sus **RESPONSABILIDADES**.

- Los objetos realizan un conjunto de operaciones (o tipos)
- Dichas operaciones las define el tipo del objeto
- La forma de utilizar un objeto es enviando mensajes a través de una referencia
- Cada mensaje tiene una implementación (por ahora)
- Los atributos son elementos auxiliares de dichas implementaciones y les permite actuar en función de los mensajes recibidos previamente Mantienen el estado del objeto

1.5 Mensajes, operaciones y métodos

Un objeto ha de estar **centrado en una funcionalidad**.

- Un **operando** es un objeto que el método necesita para operar.
 - Estos siempre permanecen estables.
- Una **opción** representa una forma de operar con los operandos.
 - Pueden variar a medida que la clase avance.
 - Los **parámetros** son opciones: información que se manda junto al mensaje.

Hemos de **procurar separar operandos de opciones**. Así, tenemos en cuenta el **estado del objeto**.

```
class Printer
{
    public void print(String document,
        int paperSize,
        boolean color,
        int resolution)
    {
        ...
    }
}
```

```
class Printer
{
    void setPaperSize(int size) { ... }
    void setResolution(int resolution) { ... }
    void setColor(boolean color) { ... }

    void print(String document)
    {
        // actúa en función del estado
    }
}
```

1.6 Estado del objeto

Como se ha indicado, **los atributos representan el estado interno del objeto**. Por tanto, los valores de los atributos determinarán el comportamiento del objeto ante los distintos mensajes recibidos.

Así, se determina qué mensajes son permitidos y cómo actuar con ellos. Guardar el estado del objeto supone guardar la suficiente información para que al recuperarla se obtengan exactamente las mismas respuestas ante la misma secuencia infinita de mensajes.

No obstante, no es solo el valor de los atributos: también depende de las referencias a otros objetos (delegación de mensajes en ellos, y las relaciones cruciales guardadas).

1.7 Métodos de acceso

Existen diversos criterios:

- **Encapsulación:** getters/setters.
 - o Permite controlar las modificaciones del atributo.
 - Posibles *IllegalArgumentExcepcion*, por ej.
 - o Permite modificar la implementación directamente.
- **Independencia:**
 - o La implementación de la clase es secreto: una clase no puede saber nada de la otra salvo los mensajes que puede recibir.

Criterio fundamental OO
Cuanto menos se sepa de la implementación de un objeto menos afectarán sus cambios
- **Estado del objeto:**
 - o Estado concreto: se define en función de los atributos.
 - o Estado abstracto: en función de las respuestas a los mensajes.

1.8 Cómo diseñar clases

DON'Ts:

- *Basarnos en objetos reales.*
- *Extraer nombres/verbos del lenguaje natural.*

DOs:

1. **Escoger una tarea a realizar: el QUÉ.**
 - a. Decidir QUÉ TIPO DE OBJETO lo hará
 - b. Si tenemos alguna clase que pueda realizar tal funcionalidad, añadimos dicha operación
 - c. Si no, creamos una clase nueva con una responsabilidad concreta
2. **Definir los mensajes**
 - a. Dividir la responsabilidad en operaciones atómicas

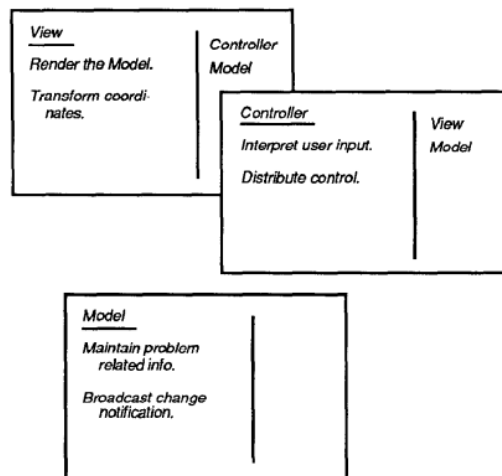
- b. Separar operandos de operaciones
- c. Sacar partido de que los objetos recuerden lo que se ha mandado para hacerlos más usables

3. Implementación de las operaciones

- a. Usar el constructor para inicializar siempre en un estado válido
- b. Al terminar una operación el estado ha de ser válido también
- c. Comprobar corrección de parámetros

Gracias a los **patrones de diseño**, se facilita la realización de los 2 primeros pasos: decisión de la funcionalidad (qué+quién) y la definición de mensajes.

Una herramienta útil son las **fichas CRC (Clase-Responsabilidades-Colaboraciones)**:



2. Jerarquías: interfaces y clases abstractas

2.1 Interfaces vs. clases abstractas

Gracias a los **patrones de diseño**, se facilita la realización de los 2 primeros pasos: decisión de la funcionalidad (qué+quién) y la definición de mensajes.

INTERFACES	CLASES ABSTRACTAS
Permiten comunicar objetos que no se conocen, aglutinados bajo un mismo tipo, y que comparten funcionalidad y capacidad para realizar una acción concreta	Dan una implementación concreta, lo que se quiere es generalizar y tener a objetos de distintas clases
Obliga a los objetos a 'cumplir un contrato' para poder ser utilizados en una situación dada	

2.2 Tipos de herencia: de interfaz

Consiste en que los subtipos carguen con el compromiso de implementar los métodos no implementados por la interfaz.

Se da así la **reutilización sin acoplamiento (coupling)**.

Utilizando este tipo de herencia, podemos evitar que los métodos exijan parámetros pertenecientes a una clase concreta. Si se aglutina bajo una determinada interfaz, esto se arregla.

2.3 Tipos de herencia: de implementación

El principal problema que podemos ver con el uso de **interfaces** es la duplicación innecesaria de código: se obliga a que la clase implemente todos los métodos definidos en la interfaz.

Para evitar esta duplicación, se utilizan las **clases abstractas**, que nos ayudan a factorizar nuestro propio código. También nos ayudan a facilitar la extensión, como se puede ver con el uso de frameworks y bibliotecas.

La clase base concreta (default/standard) es aquella que se puede instanciar directamente, y que da implementación a todos los mensajes. Las clases abstractas son aquellas que no se pueden instanciar directamente pueden quedar mensajes sin implementar.

Con las clases abstractas, se puede:

- **Aprovechar el código ya implementado**, es decir, heredar el método y no hacer ningún cambio en él.
- Redefinirlo y dar su **implementación completa**.

- Utilizar partes de la implementación y luego delegar otras en otras clases, es decir, **establecer una serie de pasos obligatorios** que han de realizar todos.
 - o Esto se puede conseguir: haciendo que la clase base tenga la parte obligatoria, las derivadas provean sus partes específicas **a través de un método protected**.

2.4 Extensión: añadiendo nuevos métodos

Se dice que una de las grandes ventajas del OOD es la facilidad de extender las clases añadiéndoles implementaciones de nuevos métodos.

El problema es que, en una jerarquía, **con lo que se interactúa son con las interfaces**. Si se extiende el código de una extensión, no se conocerá el código añadido a la implementación. Lo bueno es que, como estamos usando OO, siempre nos podemos aprovechar del enlace dinámico para no concretar el tipo concreto del objeto.

2.5 Jerarquías de interfaces

Sobre el **"ES UN"**: el gran problema de esta creación de jerarquía es que surge como mera clasificación. Hemos de tener en cuenta que **las interfaces surgen por la necesidad de declarar unas responsabilidades**, una determinada funcionalidad que cumpla unos determinados requisitos. Aparte, simplemente hemos de **ver si esta interfaz es usada por al menos un cliente** (implementada por alguna clase).

2.6 Jerarquías de clases abstractas

Las clases abstractas surgen, por ejemplo, del hecho de que existe duplicación de código en una herencia. Por ello, nacen como consecuencia de la factorización.

De hecho, esta jerarquía me da la idea de compartir código común para cambiarlo según se quiera.

2.7 Resumen: interfaz vs. Clase abstracta

	Interfaz	Clase abstracta
Situación en la que se originan	Extracción de responsabilidades	Facilitar nuevas implementaciones No repetir código
Creador	Analista/Diseñador	Programador
¿Raíz?	Sí	No
Importancia	Fundamental: es el enlace dinámico, lo que nos permite eliminar lógica condicional	Comodidad (es un mero cortar y pegar)
Operaciones	Añaden a las de sus ancestros	Implementan las de sus ancestros Añaden operaciones de implementaciones parciales
Impacto de los cambios	Alto; es un contrato que afecta a varios programadores	Bajo; la interfaz hace de cortafuegos

3. Principios de diseño

3.1 'Bad smells', o síntomas de un diseño pobre

Estos son los síntomas de un diseño pobre o con fallos:

- **Rigidez:** resistencia al cambio.
 - Un simple cambio causa una sucesión de cambios en cascada de otros módulos dependientes entre sí.
- **Fragilidad:** fácil que falle.
 - Un simple cambio en una parte del sistema causa un fallo en otras no relacionadas conceptualmente.
- **Inmovilidad:** difícil de reutilizar.
 - Es difícil separar el código en componentes que puedan ser reutilizados en otros sistemas.
 - Esto es especialmente importante para librerías y frameworks.
- **Viscosidad:** difícil hacer lo correcto.
 - Del software: Es difícil preservar y respetar el diseño original.
 - Del entorno: que sea lento e ineficiente (compilación, control de versiones), muy costoso.
- **Complejidad innecesaria:** sobrediseño.
 - Difícil de comprender y utilizar (probablemente ni siquiera se utilice).
- **Repetición innecesaria:** duplicación del código.
 - Absolutamente desastroso (en término de propensión a errores y costes)
- **Opacidad:** difícil de comprender un módulo/programa.
 - Hemos de mantener el código claro y expresivo, fácil de comprender.

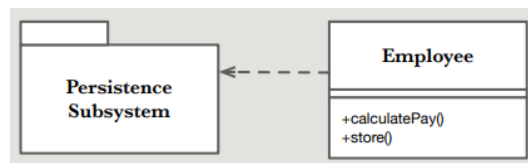
3.2 OOD: Principios SOLID

SRP: Principio de responsabilidad única

Una clase debe tener un único motivo para cambiar.

Una **responsabilidad** siendo una razón para el cambio. Si se nos ocurre más de un motivo por el que la clase debe cambiar, es porque la clase tiene más de una responsabilidad.

Tendemos a juntar las responsabilidades de manera natural. Lo que tenemos que hacer es encontrar y separar las responsabilidades.

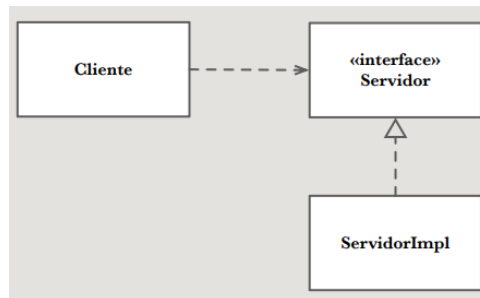


Ejemplo: la clase 'Employee' contiene la lógica de negocio, así como lógica de persistencia. Estas funcionalidades deberían ir separadas (ej. qué hacer si queremos cambiar el SGBD, o guardarlo en distintos tipos de ficheros).

OCP: Principio de abierto-cerrado

Las clases deberían estar abiertas para la extensión, pero cerradas para la modificación.

Si se nos da código, los cambios serán únicamente añadir código nuevo, no modificar lo anterior que funcionaba.



Ejemplo: gracias a esto, si queremos cambiar el servidor, podemos añadir nueva funcionalidad en su interfaz, que se implementa en la clase Impl y se usa en la clase Cliente.

LSP: Principio de sustitución de Liskov

Los subtipos deben poder sustituir a sus tipos base, o:

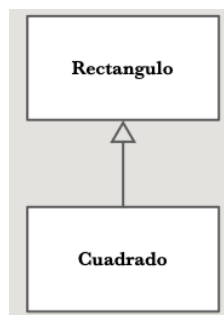
Las funciones que usen punteros o referencias a una clase base deberían poder utilizar un objeto de cualquier clase derivada sin saberlo.

Los objetos de un programa deberían ser REEMPLAZABLES por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.

También: debemos asegurarnos de que las clases derivadas extiendan la clase base sin alterar el comportamiento y se viole el contrato.

→ NO usar INSTANCEOF o hacer DOWNCASTS.

→ Es como si los métodos no fuesen conscientes de la existencia de los subtipos.



Para que esto sea correcto, cuadrado ha de poder sustituir a rectángulo en todos los casos.

DIP: Principio de inversión de dependencias

Los módulos de alto nivel no deben depender de los de bajo nivel: ambos deben depender de abstracciones.

Las abstracciones no dependen de los detalles, sino estos de las abstracciones.

Se dice 'inversión' porque los módulos de bajo nivel han de apoyarse en los de alto nivel.

Principio de Hollywood

Hay que depender de las abstracciones, no de las implementaciones.

■ PROGRAMAR PARA LA INTERFAZ, NO PARA LA IMPL.

Hemos de programar para la interfaz, ya que el tipo concreto ha de darnos igual, solo queremos que cumpla un determinado contrato (la interfaz). Solo conocemos la interfaz, no las implementaciones concretas. Así, evitamos las dependencias de implementaciones concretas.

ISP: Principio de segregación de interfaces

Es mejor usar muchas interfaces específicas que una con propósito general.

→ Los clientes no deberían depender de métodos que no usan.

No debemos obligar a muchos clientes a implementar un método que NO necesitan.

3.3 Patrones GRASP (*General Responsibility Assignment Software Patterns*)

Son unos PRINCIPIOS para asignar responsabilidades que siempre podemos aplicar. Divide las responsabilidades en dos tipos: 'hacer' y 'conocer'.

Una responsabilidad no es un método per se, más bien los métodos se definen para llevar a cabo las responsabilidades. Las responsabilidades pueden llevarse a cabo por un objeto concreto pero pueden colaborar varios para ello.

Se utilizan diagramas de interacción para ello.

- **GRASP 1: Experto en información**

Asignar una responsabilidad al experto (la clase que tiene la información necesaria para llevarla a cabo).

Miramos qué clases tienen la información necesaria en:

- El modelo de diseño (perspectiva software) si está lo suficientemente detallado, o
- El modelo de dominio (perspectiva conceptual), y vamos refinando los correspondientes diagramas de diseño

Problema: la solución dada por el 'Experto' no siempre es la mejor. A veces, damos una responsabilidad a una clase erróneamente, aumentando: la no cohesión, el acoplamiento, y la duplicación de código.

Ventaja: mantenemos el encapsulamiento y bajo acoplamiento, y se distribuye el comportamiento entre las clases, mayor cohesión y fácil entendimiento.

- **GRASP 2: Creador**

Asignar a la clase B la responsabilidad de crear una instancia de la clase A si se cumple alguna de las condiciones siguientes:

- B agrega objetos de A
- B contiene objetos de A
- B registra objetos de A
- B utiliza más estrechamente objetos de A
- B tiene los datos de inicialización de un objeto de A

Problema: a veces requeriremos 'factorías' para la fabricación auxiliar.

- **GRASP 3: Bajo acoplamiento (*dependencia de otros elementos*)**

Asignar una responsabilidad de manera que el acoplamiento permanezca bajo.

El **acoplamiento** tiene que ver con la dependencia de otras clases, por lo que a mayor acoplamiento:

- Cambios en una clase → cambios en muchas otras.
- Dificil aislamiento
- Dificil reutilización

El acoplamiento se da cuando se tiene un atributo referenciando a otro objeto, cuando se invoca un método de otra clase, o cuando se es una subclase o implementación de una interfaz.

A menor acoplamiento → más independencia entre clases.

Para el caso de herencia, las subclases están muy acopladas a sus superclases, aunque recompensa con la reutilización de código.

Siempre es necesario un poquito de acoplamiento.

- **GRASP 4: Alta cohesión**

Asignar una responsabilidad de manera que la cohesión permanezca alta.

La **cohesión** es la fuerza con que se relacionan las responsabilidades de un elemento. A menor cohesión, más difícil de entender, reutilizar, mantener y modificar.

→ La **modularidad** es la propiedad de un sistema que se ha descompuesto en un conjunto de módulos cohesivos y débilmente acoplados: cada método se crea con un único objetivo.

- **GRASP 5: Controlador**

Asignar la responsabilidad de recibir o manejar un evento a una clase que representa una de las siguientes opciones:

- Representa el sistema global, un dispositivo o un subsistema (controlador de fachada)
- Representa un escenario de caso de uso en el que tiene lugar el evento

Por ejemplo: caso en el que se gestionan los eventos producidos en una interfaz gráfica. La ventana donde se realiza la interacción y se produce el evento no ha de ser la que gestione la lógica, sino que ha de ser pasado a esa capa de dominio para solucionarlo.

- **GRASP 6: Polimorfismo**

Cuando las alternativas o comportamientos relacionados varían según el tipo de objeto, se asignará la responsabilidad a los tipos para los que varía el comportamiento, empleando operaciones polimórficas.

Básicamente: hay que usar polimorfismo, nada de if/switch/instanceof.

- **GRASP 7: Fabricación pura**

Se asignará un conjunto de responsabilidades altamente cohesivo a una clase artificial, de conveniencia, que no representa un concepto del dominio del problema, sino que se ha inventado para permitir esa alta cohesión, bajo acoplamiento y la reutilización de código.

- **GRASP 8: Indirección**

Se asignará la responsabilidad a un objeto intermedio que medie entre otros componentes o servicios de manera que no se acoplen directamente.

- **GRASP 9: Variaciones protegidas**

Identificar aquellos aspectos que varían y separarlos de lo que tiende a permanecer igual.

Es decir... encapsular y aislar el concepto que varía

→ Usar principios SOLID.

Un **buen diseño** es aquel realizado para el cambio y es fácil de comprender.

SOLO hay que cambiar un sitio.

→ Clases con pocas responsabilidades, métodos con nomenclatura clara y uso de patrones diseño.

4. Introducción a los patrones de diseño

- **Favorecer la composición de objetos frente a la herencia:**
 - La herencia se define en tiempo de compilación, por lo que es difícil de reutilizar.
 - La composición se define en tiempo de ejecución a través de objetos que guardan referencia a otros, a los que se accede con sus interfaces, y además favorece la **encapsulación y modularidad**.
- La **delegación** consiste en que 2 objetos sean encargados de responder a una petición, lo que permite variar fácilmente el comportamiento en runtime.
 - Lo malo es que requiere mucho rendimiento y es más difícil de entender.
 - Muchos patrones se basan en ella.
- **Estructura en tiempo de compilación:** estática, código, clases con una serie de relaciones entre ellas
- Estructura en tiempo de ejecución: dinámica, red de objetos que cooperan y se comunican entre sí

Cada **patrón** es una regla de 3 partes, que es una relación entre:

- Un **contexto** determinado
- Un **problema**
 - Requisitos que debe cumplir la solución,
 - Restricciones a considerar,
 - Propiedades deseables
- La **solución**
 - Estructura estática del patrón (clases + relaciones)
 - Su comportamiento dinámico (cooperación + comunicación)

Los patrones son esquemas que, según su abstracción (+ a -), pueden ser:

- **Arquitectónicos:** describen la arquitectura, estructura del sistema y subsistemas y relaciones entre ellos
 - Ej: Patrón MVC
- Patrones de diseño
- Centrados en el código (**idioms**): específicos de lenguajes determinados, para dar estilo uniforme, son convenciones de estilo
 - Ej: escoger cómo recorrer una colección (si con un bucle o con un iterador)
 - Ej: cómo utilizar los indent para las estructuras de control

4.1 Propiedades de los patrones

1.

Un patrón responde a un problema de diseño que se repite frecuentemente en determinadas situaciones, y presenta una solución a dicho problema.

2.

Los patrones sirven para documentar la experiencia previa, los diseños que ya se probaron útiles.

3.

Identifican y especifican abstracciones que están por encima de clases y objetos o componentes individuales.

4.

Proporcionan un vocabulario de diseño común.

5.

Permiten documentar nuestros diseños.

6.

Ayudan a construir arquitecturas complejas y heterogéneas.

4.2 Catálogo de patrones de diseño (GoF)

		Propósito		
		Creación	Estructural	Comportamiento
Ámbito	Clases	Factory Method	Adapter	Interpreter Template Method
	Objetos	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

4.3 Tipos de software

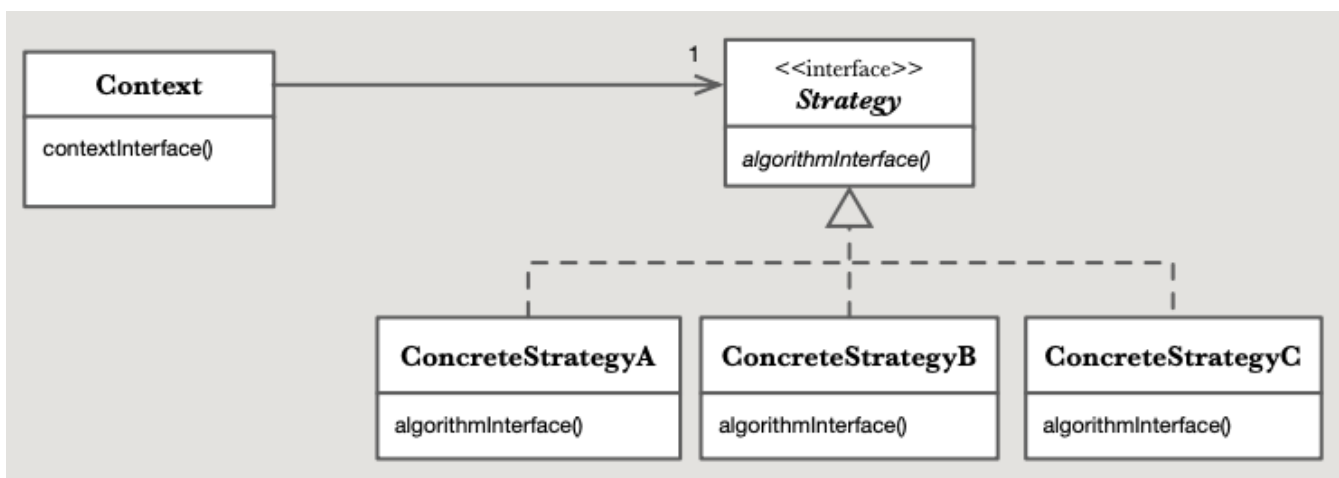
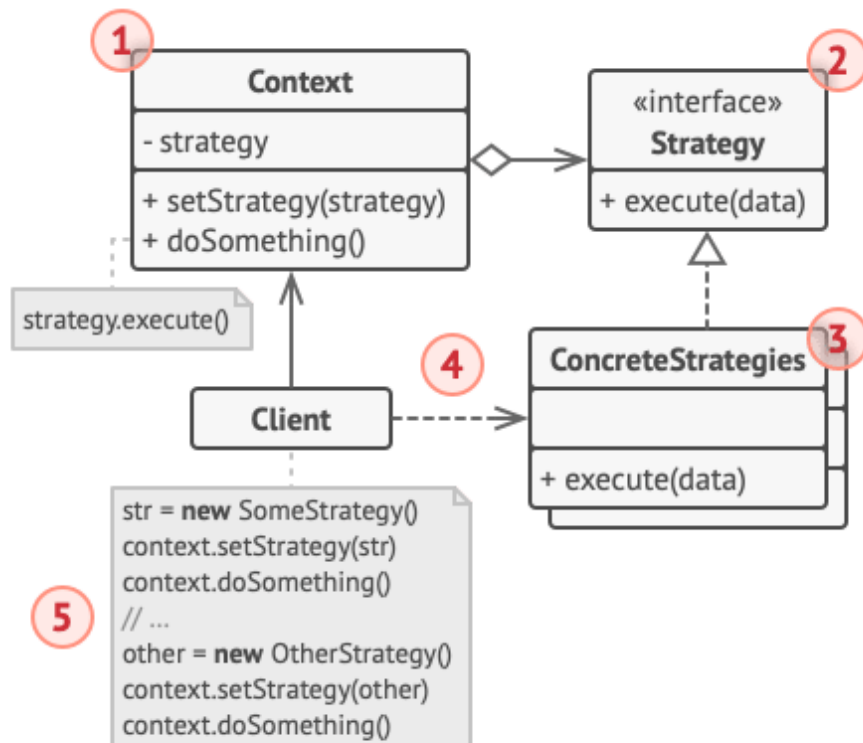
- **Aplicaciones:**
 - Implementan una determinada funcionalidad siguiendo un determinado diseño.
 - Prioridad: reutilización interna, mantenibilidad, extensibilidad.
- **Bibliotecas de clases:**
 - Contiene la FUNCIONALIDAD.
 - Se basan en la reutilización de código.
 - Evitar la dependencia.
 - Cuando utilizamos una biblioteca, escribimos el código principal y llamamos al código que queramos.
- **Frameworks:**
 - Clases cooperantes que constituyen un software reutilizable. Ej: Junit para tests.
 - Determina cómo será la arquitectura de la aplicación.
 - Contiene el DISEÑO.
 - Cuando utilizamos un framework, escribimos el código específico a ser llamado por el framework.
 - Cuando utilizamos un framework, escribimos el código específico a ser llamado por el framework.

5. Patrón STRATEGY

El **patrón Strategy** (*policy*) define una familia de algoritmos, los encapsula y hace intercambiables.

Permite que el algoritmo varíe de forma independiente a los clientes que lo usan.

Es un **patrón de comportamiento**.



Motivación:

Necesitamos distintas variantes de un algoritmo.

Participantes:

1. Compositor (Strategy):

Declara la interfaz común.

2. ConcreteStrategy (distintas formas de Compositor):

Implementación de cada estrategia (variante del algoritmo).

3. Context (Composition):

Configurado con una estrategia concreta.



Tiene una referencia al objeto de interfaz Strategy.

Puede definir operaciones (ej. Getters) para permitir a la estrategia acceder a los datos necesarios.

Colaboraciones:

- La estrategia y contexto colaboran para implementar el algoritmo escogido, o bien pasándole el contexto todos los datos a llamar a la estrategia o bien pasarse a sí mismo como referencia para que la estrategia acceda a esos datos cuando lo necesite.
- Los clientes pueden escoger la estrategia concreta a utilizar.

Consecuencias:

-  Define una familia de algoritmos relacionados.
- Alternativa a la herencia:
 - Contexto es más fácil de entender, modificar y mantener.
 - Evitamos duplicación de código.
 - Evitamos la eterna modificación y explosión de subclases (a la hora de añadir más métodos y funcionalidades a la superclase).
 - Se puede cambiar dinámicamente (todo depende de la interfaz), y esto lo puede escoger el cliente.
- Se eliminan los if/switch...
-  Los clientes deberían conocer y entender las distintas estrategias.
 - Realmente no tiene por qué, siempre que el cliente no tenga que cambiar él la estrategia.
- Podría comunicarse la comunicación entre contexto y estrategias.
- Aumenta el número de objetos.

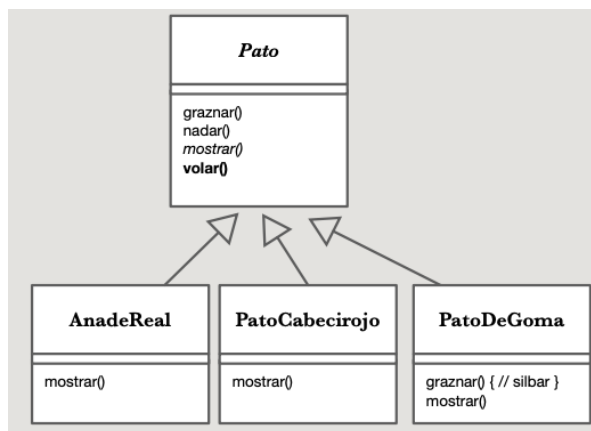
Ejemplos de uso: validación de formularios, separación de líneas en textos...

→ **Ejemplo del pato:**

Imagina una superclase pato donde cada pato grazna(), nada() y es mostrado [mostrar()]. Inicialmente los dos tipos de patos que tenemos graznan y nadan igual, y la implementación de mostrar() depende de cada uno.

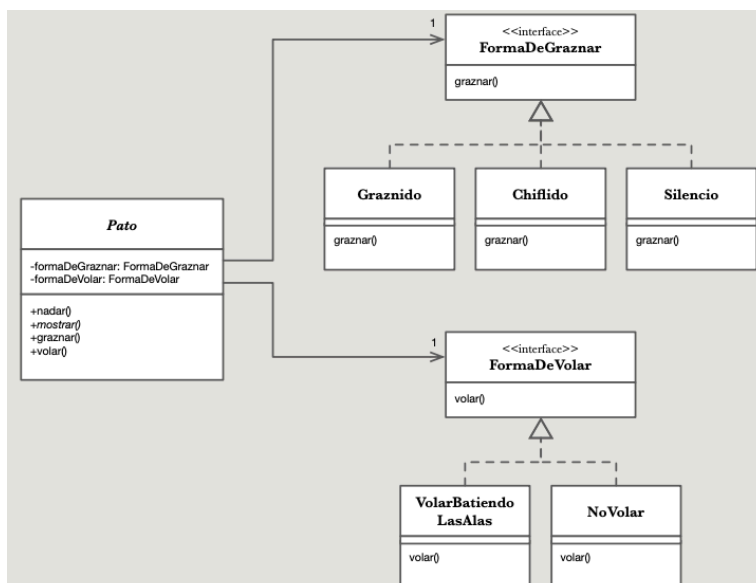
No obstante, queremos añadir una nueva funcionalidad: volar(). La solución más fácil sería implementarlo en la superclase y, automáticamente, todos los patos vuelan.

Problema: que ahora todos los patos que tengamos volarán, incluso aquellos que no deberían volar.



Como solución, podríamos redefinir el `volar()` del **PatoDeGoma** para que no haga nada.

Sin embargo, imagina que añadimos otro tipo de pato: **PatoDeReclamo**, que ni grazna ni vuela. Tendríamos que redefinir ambos métodos. Por tanto, los cambios aumentan exponencialmente y se hace más difícil añadir nuevas funcionalidades y satisfacer los requerimientos y funcionamientos de cada una. Aplicamos un principio básico: **identificar lo que varía y separarlo de lo que tiende igual → encapsulación**.

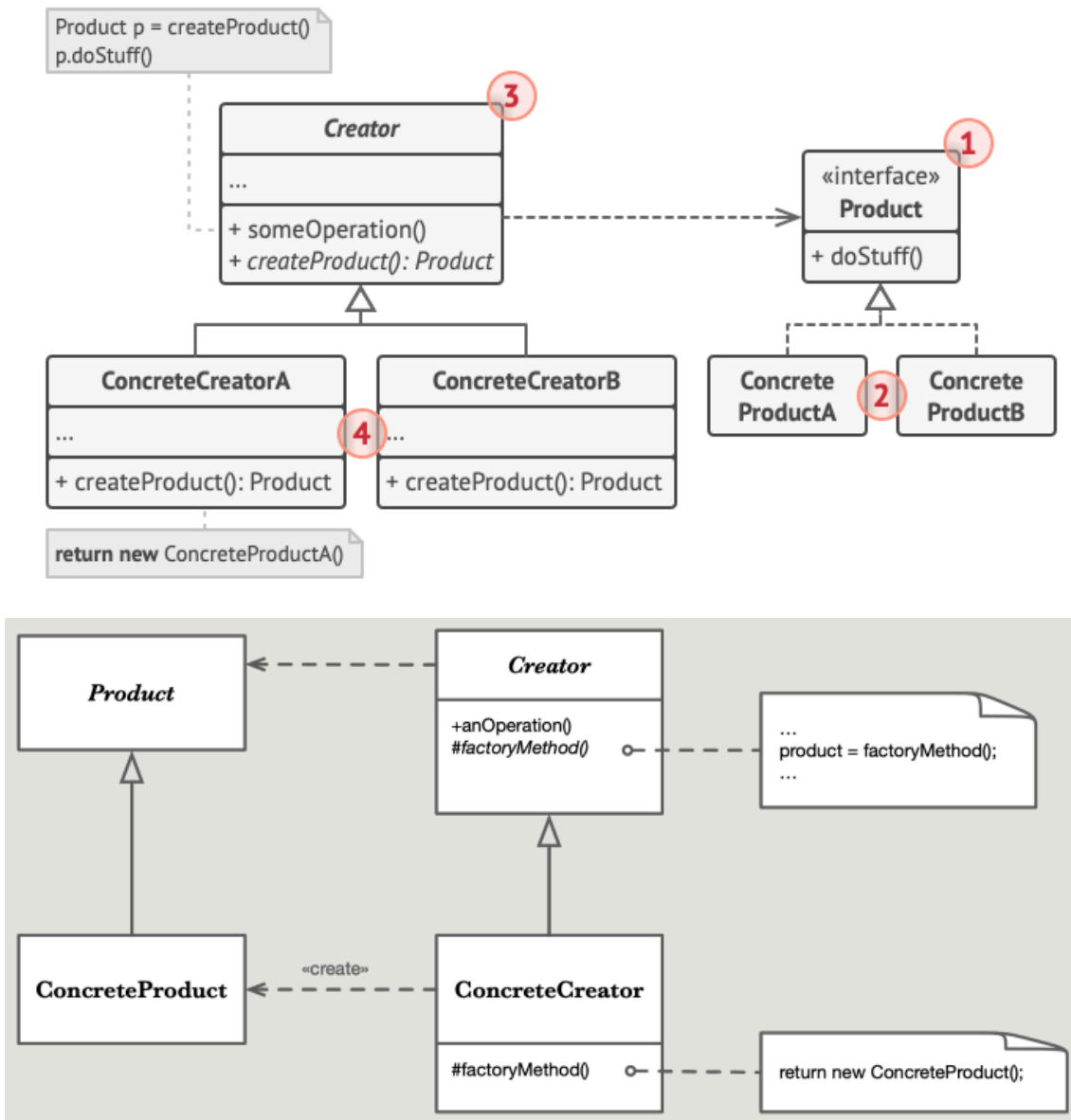


- **Encapsulación** del concepto que varía.
- **Composición** > herencia.
- **Programación para la interfaz**, no para la implementación.

6. Patrón FACTORY METHOD

El **Factory Method** define una interfaz para crear un objeto, dejando que las subclases sean quienes decidan la clase del objeto a crear.

Es un **patrón de creación de objetos**.



abstract Product factoryMethod(type)

Un «factory method» es abstracto: las subclases concretas deben redefinirlo por fuerza.

Devuelve un producto que normalmente es usado en los propios métodos de la superclase, sin falta de saber qué tipo concreto de producto es.

Opcionalmente, puede ser parametrizado (si hay que decidir entre distintas variaciones de un producto).

Motivación:

Definición de creación de objetos de tipos específicos desacoplada de la implementación. Se usa cuando hay clases que delegan responsabilidades en una o varias subclasses, y queremos saber quién es el delegado.

Participantes:

1. Product:

Declara la interfaz común de los objetos fabricados.

2. ConcreteProduct:

Implementación de cada producto.

3. Creator:

Declaración del método fábrica (abstracto) que crea nuevos objetos de la interfaz Producto.



4. ConcreteCreator:

Subclase de Creator para la redefinición de la implementación de fábrica para así devolver un objeto ConcreteProduct.

Colaboraciones:

- El creador se apoya en las subclasses para definir el método de fabricación que devuelve el objeto apropiado.

Consecuencias:

-  Se desacoplan ciertas clases (implementaciones) específicas del código, ya que solo se maneja la interfaz Product [entonces, podemos usar cualquier ConcreteProduct].
-  Tenemos que crear una subclase ConcreteCreator en los casos en los que esta no fuera necesaria al no aplicar el patrón.

Variante:

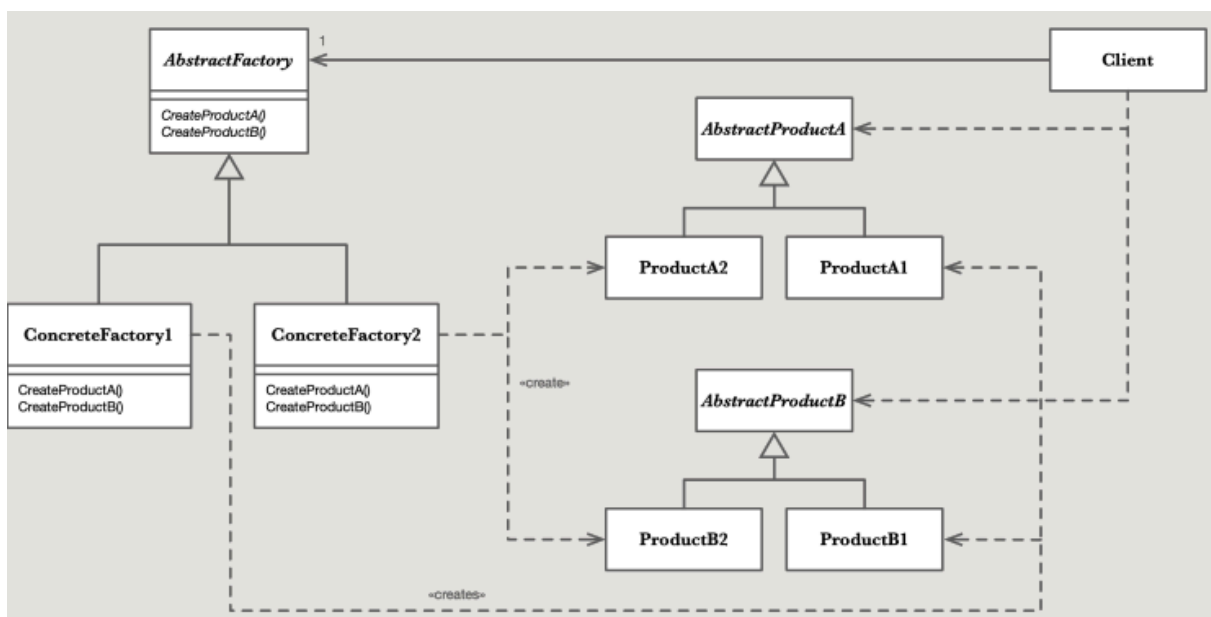
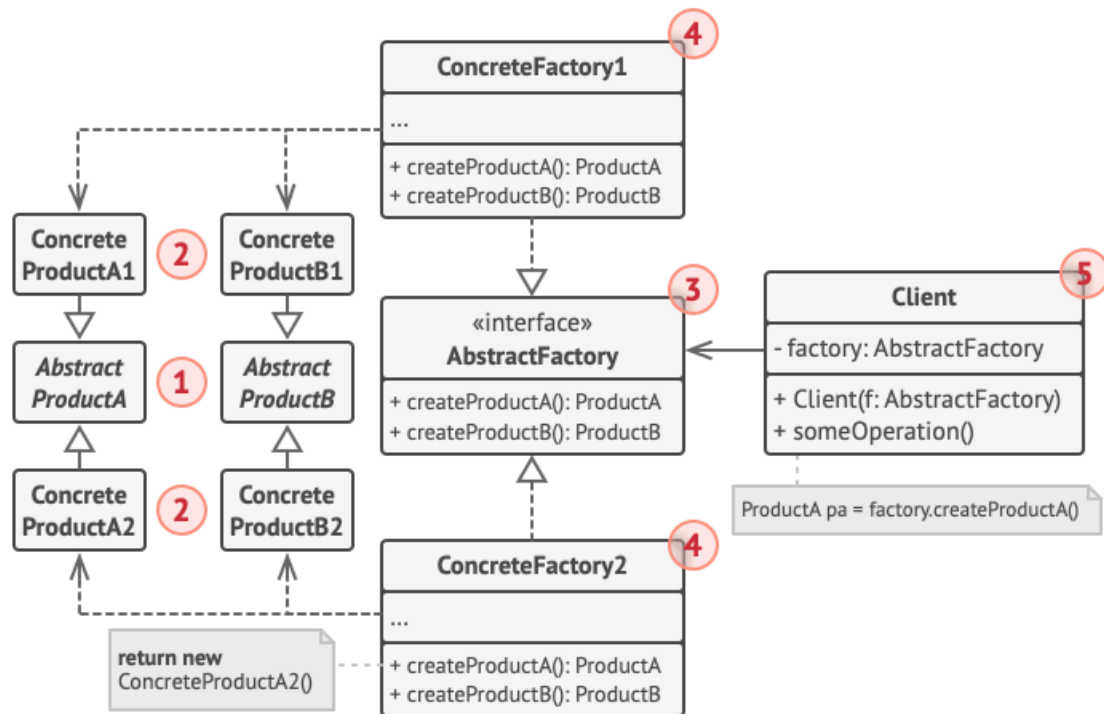
A veces, es el cliente quien llama al método de fabricación, en los casos de **jerarquías de clases paralelas**. Para ello podemos parametrizar el método de fabricación, indicando el tipo de objeto a crear. Hemos de aprovechar las características del lenguaje de implementación. Por ejemplo, en Java:

- Pasarle el nombre de la clase al método de fabricación → reflectividad.
- Guardar la clase en una variable estática de clase.
- Servirnos de métodos: valueOf, of, getInstance, newInstance, getType, newType...

7. Patrón ABSTRACT FACTORY

El patrón **Abstract Factory** define una interfaz para crear familias de objetos relacionados sin especificar clases concretas.

Es un patrón de creación de objetos.



Motivación:

Creación de familias de objetos relacionados entre sí que solo pueden cooperar con otros tipos concretos.

Participantes:

1. AbstractProduct:

Interfaces para grupos de productos diferentes relacionados.

2. ConcreteProduct:

Implementaciones distintas de productos abstractos agrupados por variantes relacionadas.

3. AbstractFactory:

Declaración de un grupo de métodos para crear cada uno de los productos abstractos.



4. ConcreteFactory:

Implementaciones de los métodos de creación de la AbstractFactory.

Colaboraciones:

- El Client se sirve de las ConcreteFactory, que crean objetos de una determinada interfaz AbstractProduct. Se encuentra desacoplado de las variantes específicas.

Consecuencias:

-  Aísla las clases concretas.
 - Los clientes solo manipulan los objetos a través de las interfaces abstractas.
- Permite intercambiar fácilmente familias de productos.
- Promueve la consistencia entre los productos.
-  Dificulta añadir nuevos tipos de productos.
 - Hay que cambiar la interfaz de la fábrica abstracta e implementar el nuevo método en todas las subclases.

Implementación:

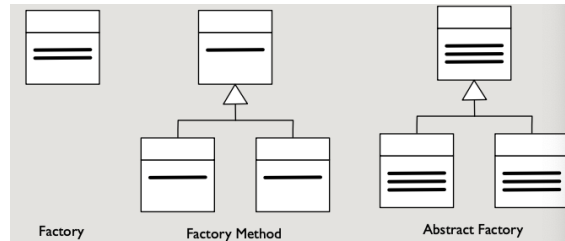
Las fábricas pueden ser *Singletons*.

Normalmente, cada AbstractFactory utiliza un FactoryMethod para cada producto (pero requiere una subclase por familia).

Se podría usar para cada distinto tipo de factoría el patrón Prototype.

Sin embargo, la forma más flexible de hacerlo sería con Fábricas extensibles: es decir, un único método de creación especificando el tipo de producto como parámetro. Sin embargo, es menor seguro, ya que todos los productos comparten el mismo tipo base y no hay comprobación estática de tipos.

Diferencias:



Cada línea negra siendo un método de creación de tipos.

Factorización → encapsulación de clases con Factory:

Hacer los constructores de clases no públicos (ej. `Protected`) y dejar que los clientes creen objetos a través de la factoría (dentro del paquete de dichas clases).

- **Motivación:** el cliente instancia directamente clases que residen en un paquete e implementan una interfaz común. El cliente no necesite conocer la existencia de clases concretas, sino solamente la interfaz.
- **Ventajas:** se ocultan clases innecesarias para el cliente, se programa para la interfaz, y se usan solamente métodos de creación.
- **Desventajas:** hay que añadir nuevo método de creación por cada nueva subclase.
- **Variación:** uso de clases internas (ej. `Java.util.Collections`).

Factorización → sustituir constructores con método de creación:

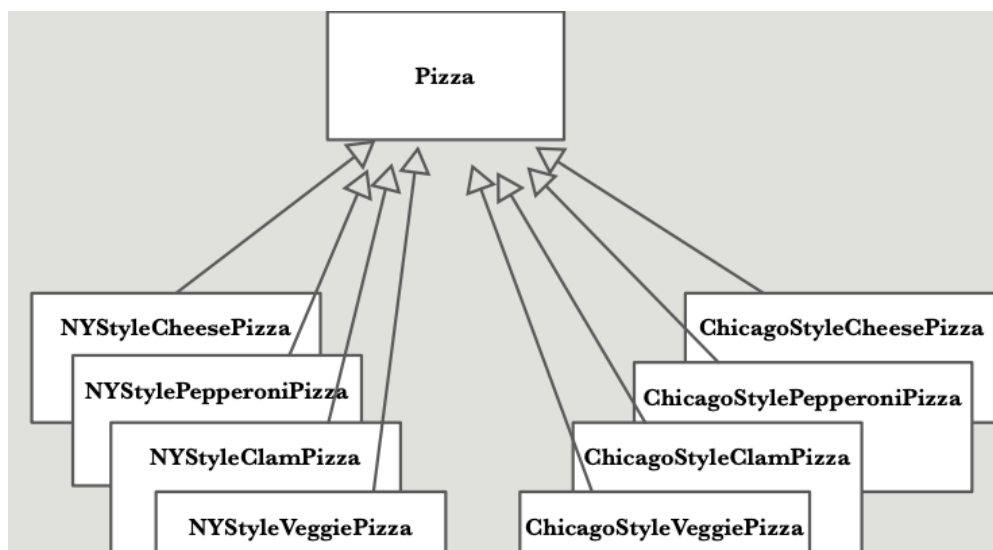
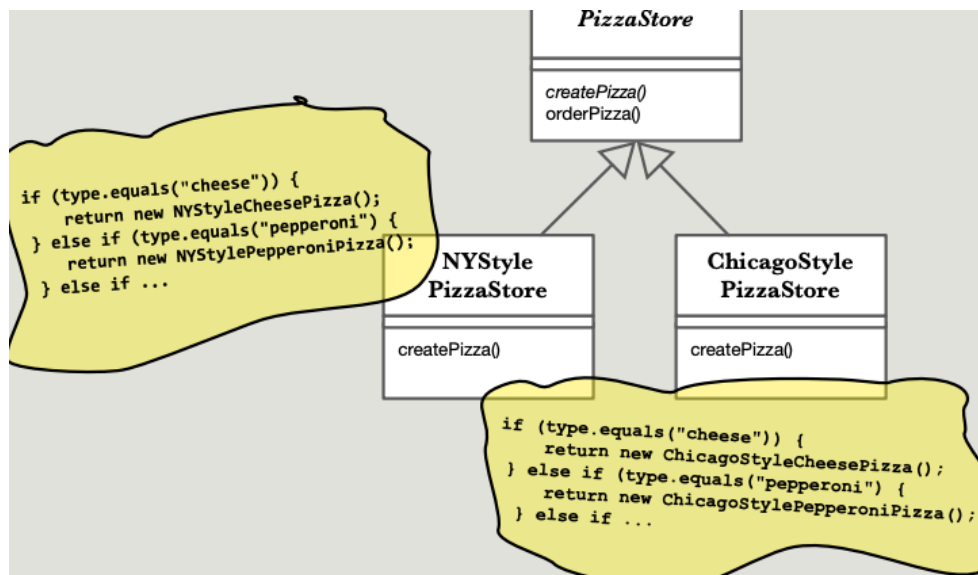
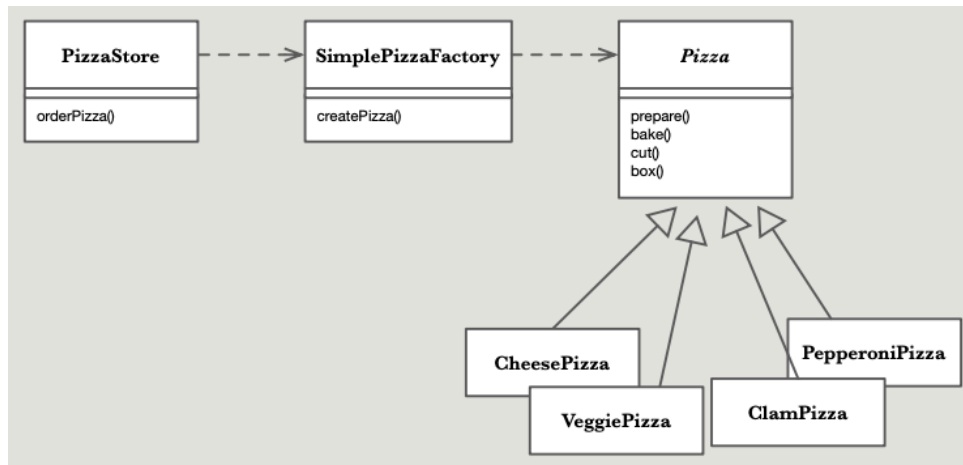
Reemplazar los constructores con métodos de creación que revelen la intención en su signatura y devuelvan instancias de objetos (ej. `createBigInteger`, `Boolean.valueOf()`).

- **Motivación:** hay demasiados constructores y no se sabe a cuál llamar.
- **Variaciones:**
 - Parameterized creation methods: dar métodos de creación a los más usuales.
 - Extract factory: sacar fuera la lógica de creación cuando la creación es más prominente que la responsabilidad propia.
- **Ventajas:**
 - Tienen nombres fáciles de comprender.
 - Si son estáticos, no necesitan crear un nuevo objeto para llamarlos.
 - Pueden devolver un objeto de cualquiera de sus subclases.

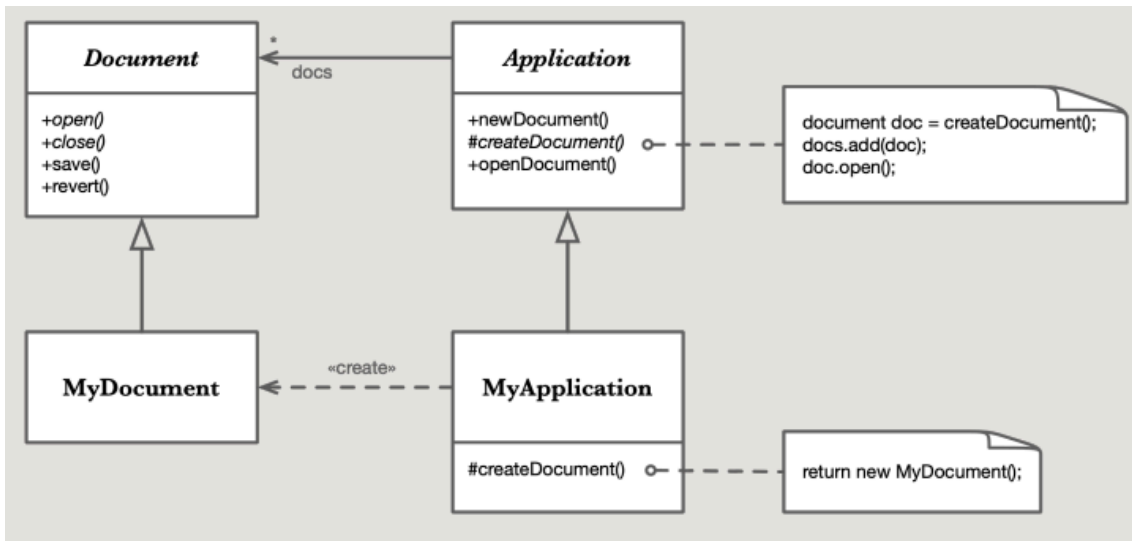
Luego, si los datos y código necesarios de creación están repartidos en numerosas clases, **mover toda esta lógica a una única factoría.**

Ejemplos de uso:

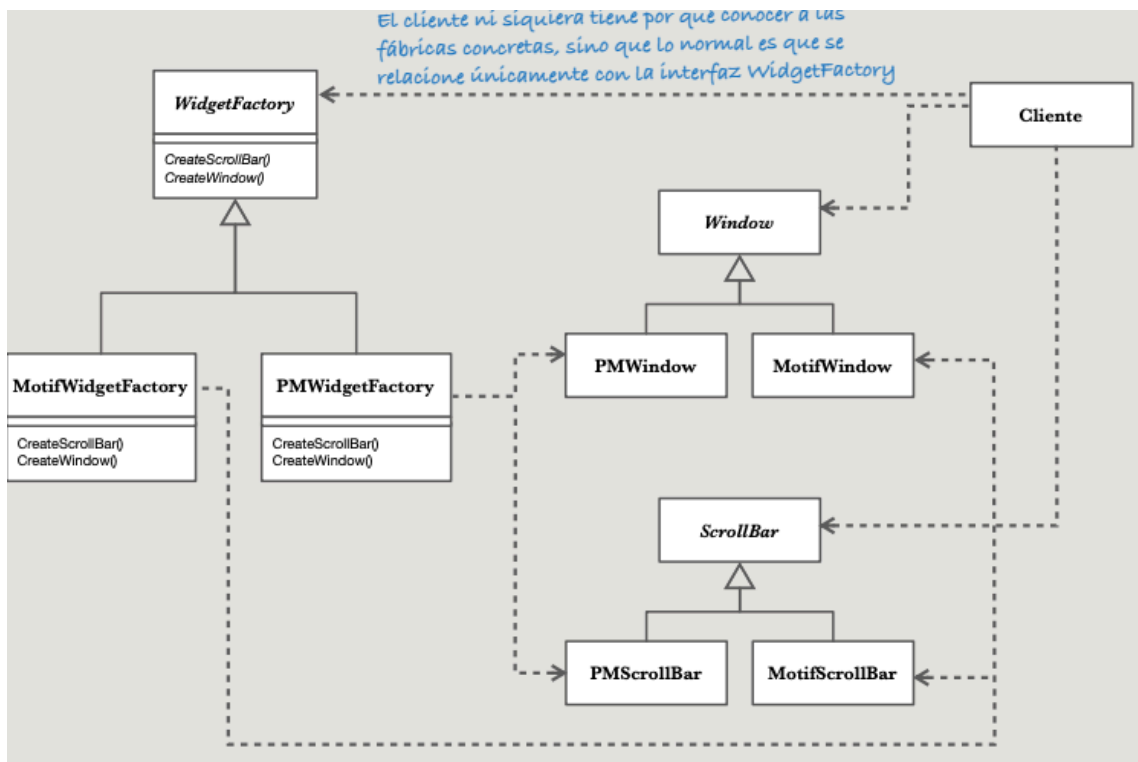
A) **Factory Method:** cadena de pizzerías de distintos estilos.



O: un framework para la construcción de editores de documentos de distintos tipos.



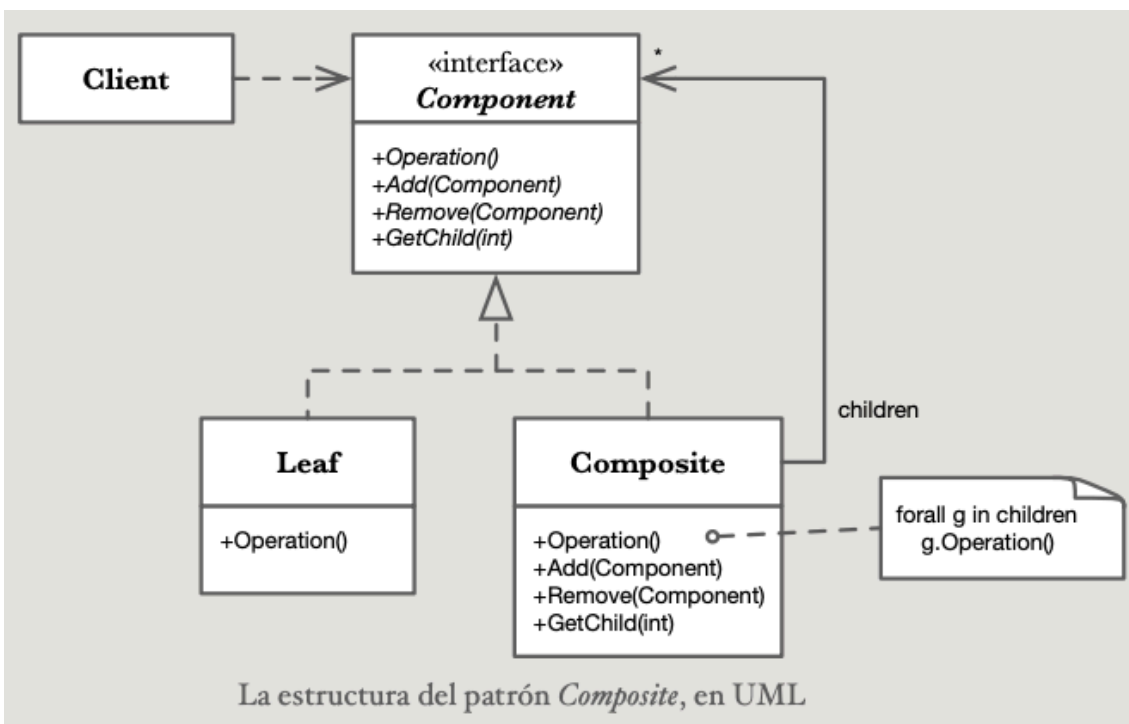
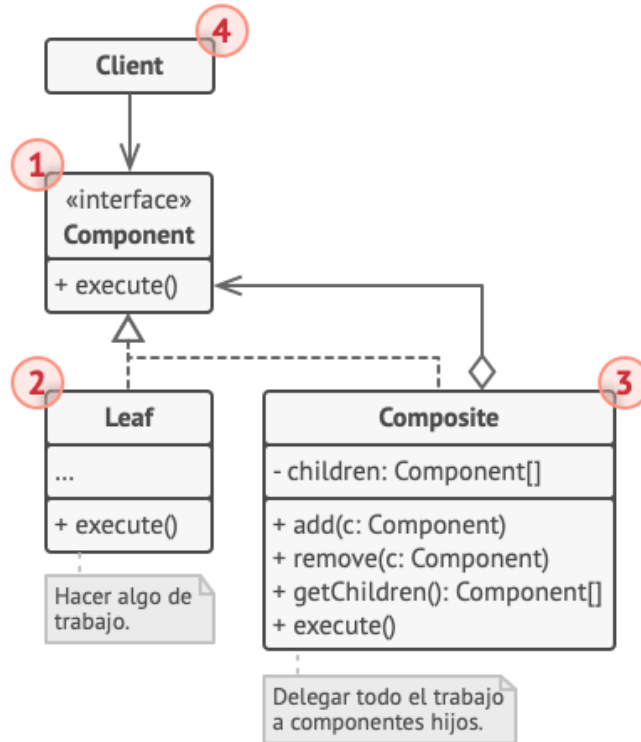
B) **Abstract Factory**: generación de interfaces de distintas ventanas con su respectivo scrollbar.



8. Patrón COMPOSITE

El **patrón Composite** permite componer objetos en estructuras arbóreas para presentar jerarquías todo/parte, de modo que los clientes puedan tratar a los objetos individuales y a los compuestos de manera uniforme.

Es un **patrón estructural de objetos**.



Motivación y aplicabilidad:

Queremos que los clientes no tengan que distinguir entre todos los elementos simples que formen un todo compuesto por dichos elementos simples. Es decir, que se trate por igual a elementos individuales y compuestos.

La estructura de objetos tiene forma de árbol.

Participantes:

1. Component:

- Declara la interfaz común e implementa el comportamiento común a todas las clases.
- A veces declara operaciones para acceder a los hijos, o para acceder al padre.

2. Leaf:

Elemento simple del árbol sin subelementos. No delegan el trabajo, lo realizan.

3. Composite:

Elemento compuesto por subelementos hijos (hojas, otros composite...). No conoce las clases concretas de sus hijos, funciona a través de la interfaz 'Component'. Implementa las operaciones relacionadas con sus hijos.



4. Client:

Manipula los objetos de la composición a través de la interfaz Component.

Implementación:

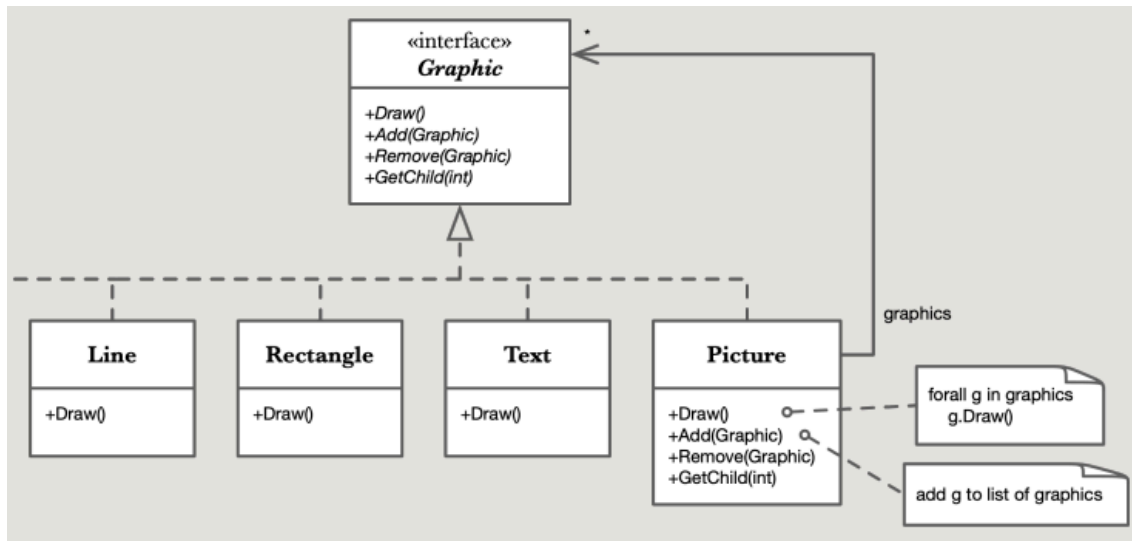
- Referencias explícitas al padre.
- Maximizar la interfaz de Component.
 - El problema es que puede haber operaciones comunes que unas subclases usen y otras no.
- Hay que tener en cuenta el modo de acceso y gestión a los hijos (ej. Un Iterador).

Consecuencias:

-  Permite jerarquías muy complejas.
- Simplifica el cliente.
- Se pueden añadir fácilmente nuevos componentes.
-  Podría generalizar mucho el diseño.
 - Por tanto, deberíamos explicitar nosotros las restricciones para que algunos componentes no puedan formar parte de un compuesto determinado.

Ejemplos de uso:

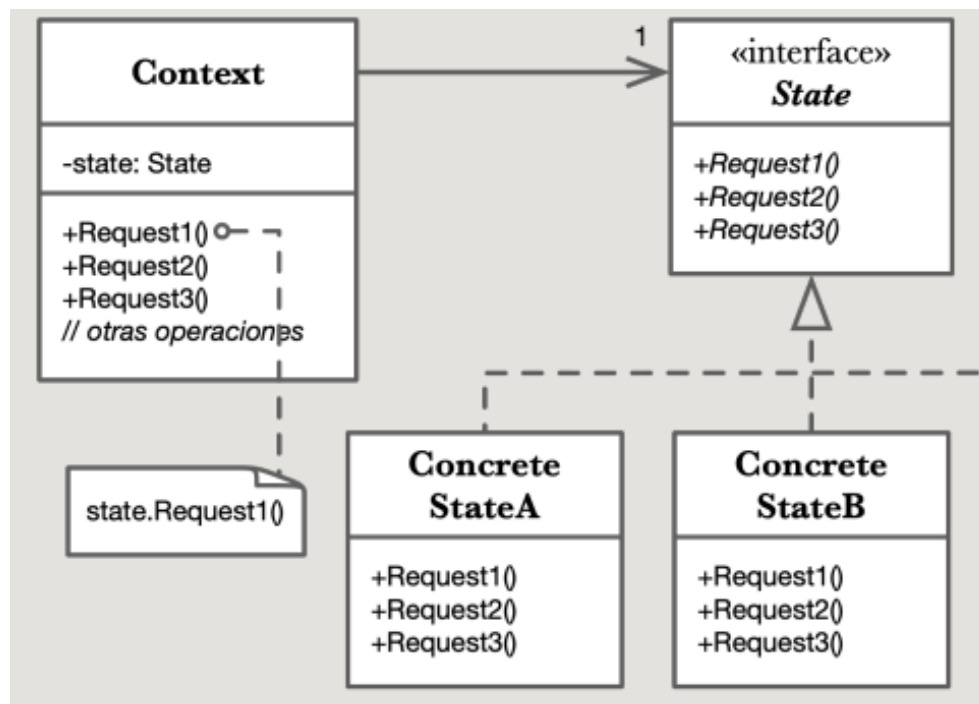
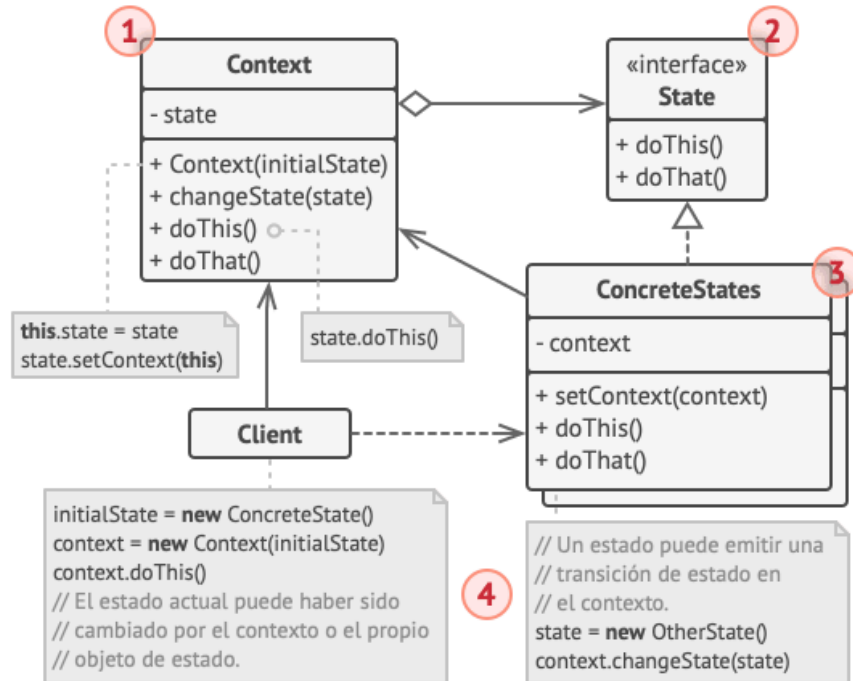
- A) **Editor de dibujo** que permite realizar: dibujos simples, o dibujos compuestos por dichos dibujos simples.



9. Patrón STATE

El **patrón State** permite a un objeto alterar su comportamiento cuando cambia su estado interno, como si el objeto cambiase de clase.

Es un **patrón de comportamiento de objetos**.



Motivación y aplicabilidad:

Se usa cuando:

- El comportamiento del objeto depende del estado, que cambia dinámicamente.
- Las operaciones tienen sentencias condicionales anidadas que tratan con los estados.

Participantes:

1. Context:

- Define la interfaz que interesa a los clientes.
- Mantiene una referencia a una subclase de estado concreto actual

2. State:

Interfaz que encapsula el comportamiento asociado al estado del contexto.


3. ConcreteState:


Distintas implementaciones de comportamiento para cada estado.

Colaboraciones:

- Las operaciones dependientes del estado se encuentran en el estado actual.
- El estado puede acceder al contexto actual si es necesario a través de una referencia.
- El cliente no necesita tratar directamente con el estado, el estado cambia acorde al contexto o los estados concretos.

Consecuencias:

-  Se aísla el comportamiento específico del estado (responsabilidad única).
- Se explicitan las transiciones entre estados.
- Principio abierto/cerrado (no se cambia lo existente), se puede extender fácilmente.
- Se disminuyen las sentencias condicionales.

-  Un poco excesivo si hay pocos estados, o raramente se cambia de estado.

Implementación:

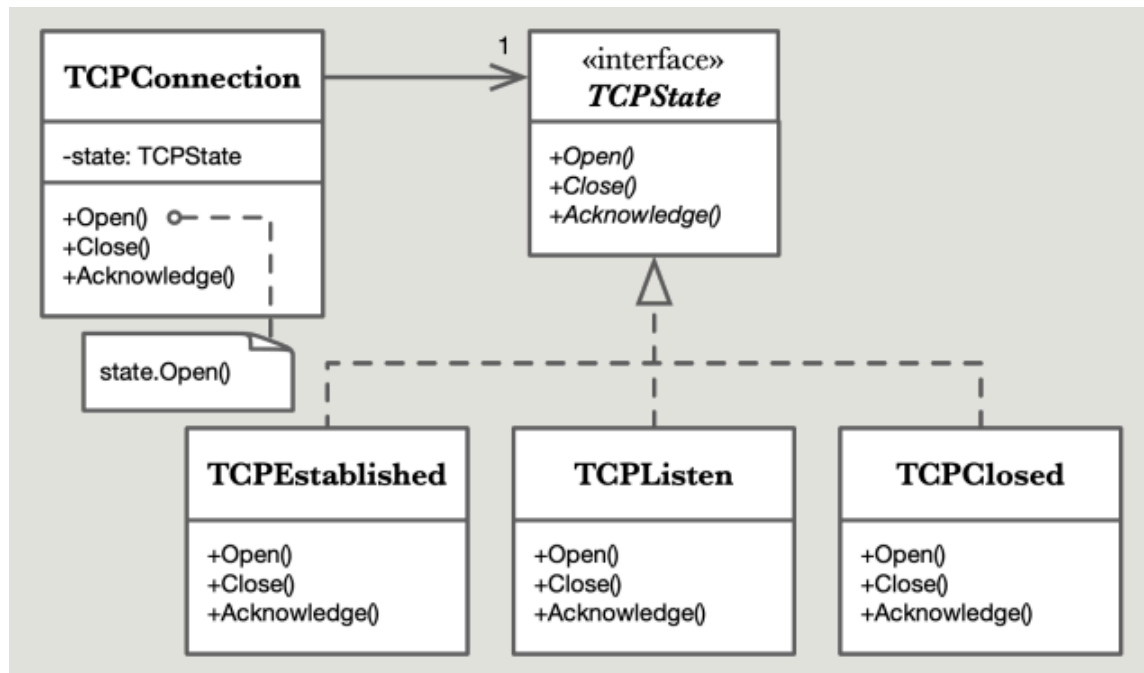
El contexto puede definir las transiciones, aunque las propias subclases de estado lo pueden hacer, lo que es más flexible, y fácil de extender y mantener. Lo malo de esto es que habría acoplamiento entre los estados concretos.

No sería necesario el patrón si el lenguaje permite cambiar de clase dinámicamente.

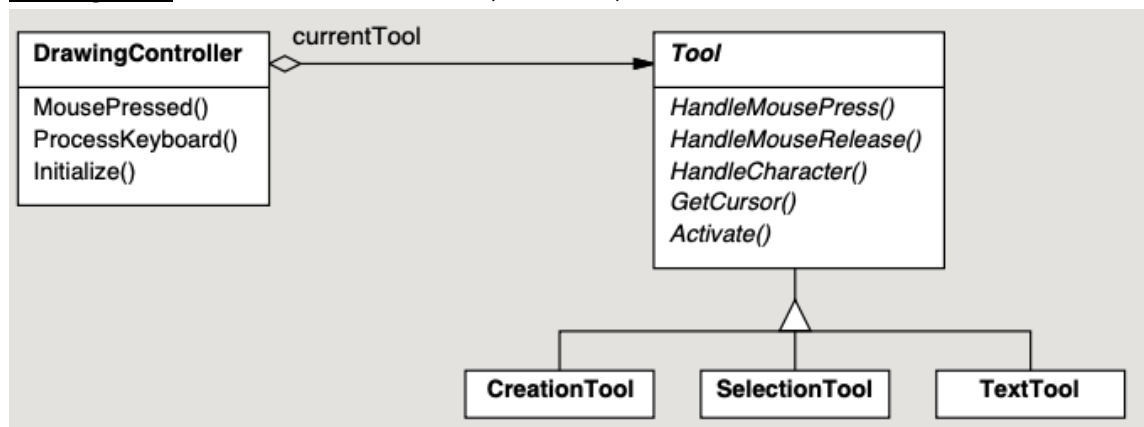
Para entrar/salir de estado se podrían añadir métodos exit() y/o entry().

Ejemplos de uso:

- A) **Comportamiento del protocolo TCP:** cambia según si la conexión está establecida, en escucha o cerrada.



- B) **Editor gráfico:** herramientas de creación, selección, texto..



Diferencia con respecto al STRATEGY:

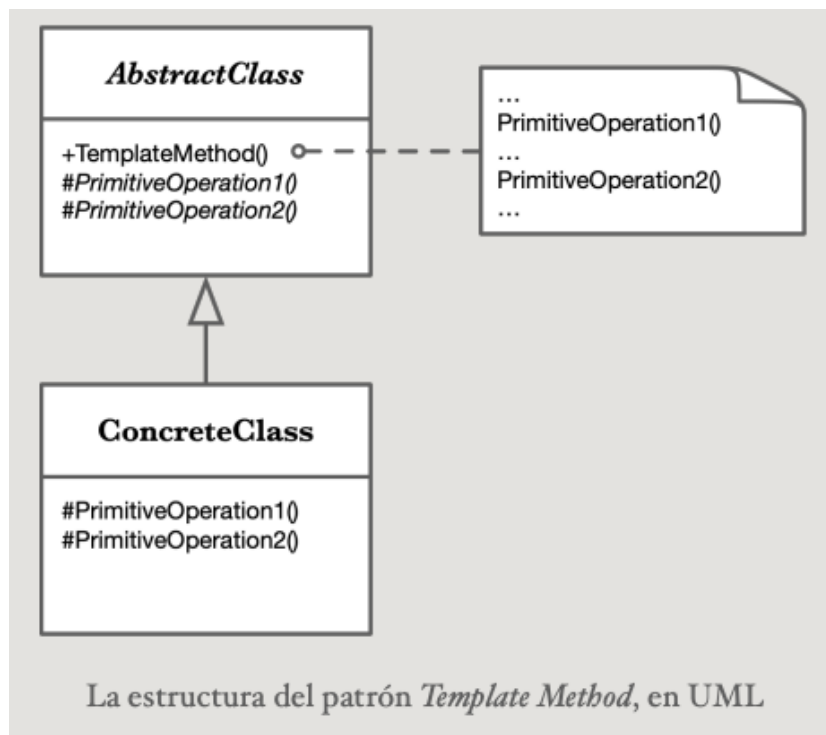
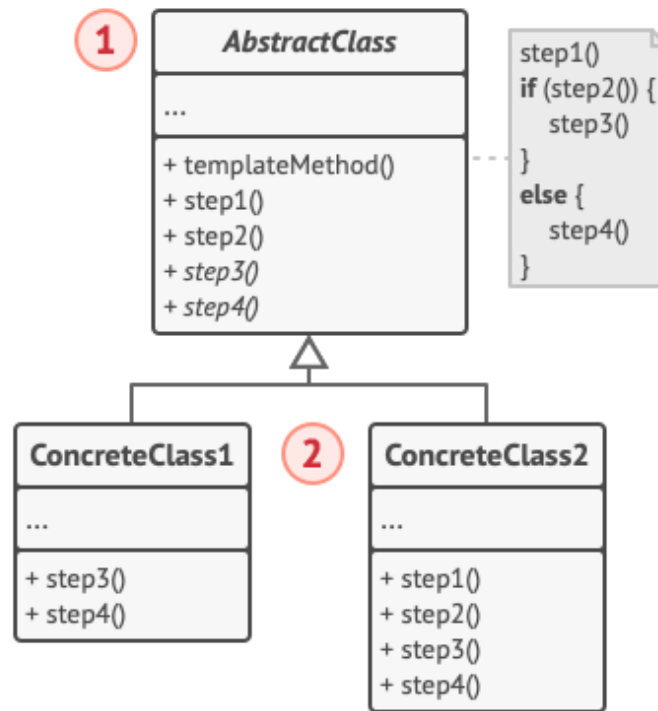
→ La estructura:

- Los métodos del State suelen llamarse igual y tener la misma signatura.

10. Patrón TEMPLATE METHOD

El patrón **Abstract Factory** define el esqueleto de un algoritmo en una operación, difiriendo algunos pasos hasta las subclases, que redefinen ciertos pasos del algoritmo sin cambiar la estructura en sí.

Es un **patrón de comportamiento de clases**.



Motivación y aplicabilidad:

Queremos que la estructura de un algoritmo persista, con cambios en algunas partes de este. Se utiliza cuando hay muchas clases con algoritmos idénticos pero diferencias mínimas.

Esto nos permite evitar código duplicado y controlar cómo extienden las subclases la clase base a partir de unos métodos plantilla.

Participantes:

1. AbstractClass:

Define las operaciones primitivas abstractas e implementa un método plantilla con el esqueleto del algoritmo.

2. ConcreteClass:

Implementa las operaciones primitivas y especifica los pasos que difieren en el algoritmo.



3. ConcreteState:

Distintas implementaciones de comportamiento para cada estado.

Colaboraciones:

- Las operaciones dependientes del estado se encuentran en el estado actual.
- El estado puede acceder al contexto actual si es necesario a través de una referencia.
- El cliente no necesita tratar directamente con el estado, el estado cambia acorde al contexto o los estados concretos.

Consecuencias:

-  Básico para la reutilización de código.
- Inversión de control (padre llama a hijos).
- Métodos plantilla pueden llamar a operaciones concretas de otra clase, sus métodos privados, operaciones concretas de la clase abstracta, operaciones abstractas, métodos de fabricación o de enganche.
-  El esqueleto puede limitar al cliente.
- Los métodos plantilla suelen ser más difícil de mantener

Implementación:

Los **métodos de enganche** son aquellos que, en la clase hija, se llama a la operación del padre para extenderla.

```
void operation()  
{  
    super.operation();  
    // DerivedClass extended behavior  
}
```

Para implementar:

- El método plantilla es protected.
- Minimizamos el nº de operaciones primitivas abstractas, que convencionalmente se llaman con el prefijo 'do'.

```
void View::Display()
{
    SetFocus();
    DoDisplay();
    ResetFocus();
}

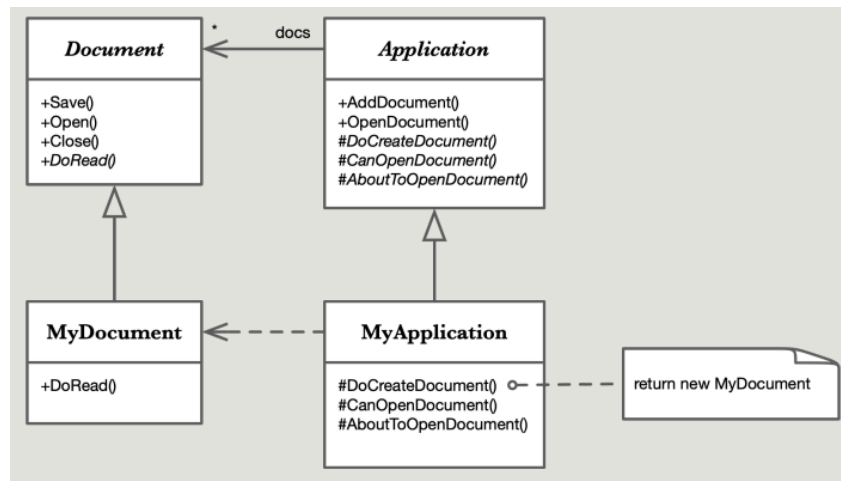
void View::DoDisplay() { }
```

Métodos relacionados:

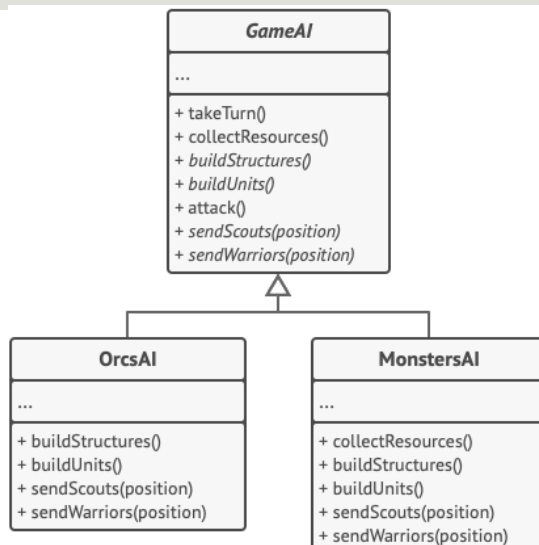
- Factory method: los métodos de fabricación suelen llamarse desde métodos plantilla.
- Strategy: mientras Template usa herencia para modificar un algoritmo, Strategy usa delegación para cambiarlo entero.

Ejemplos de uso:

- A) Framework para abrir documentos almacenados y representar la información del fichero.



- B) Implementación de comportamiento de varios tipos de IA en un videojuego.

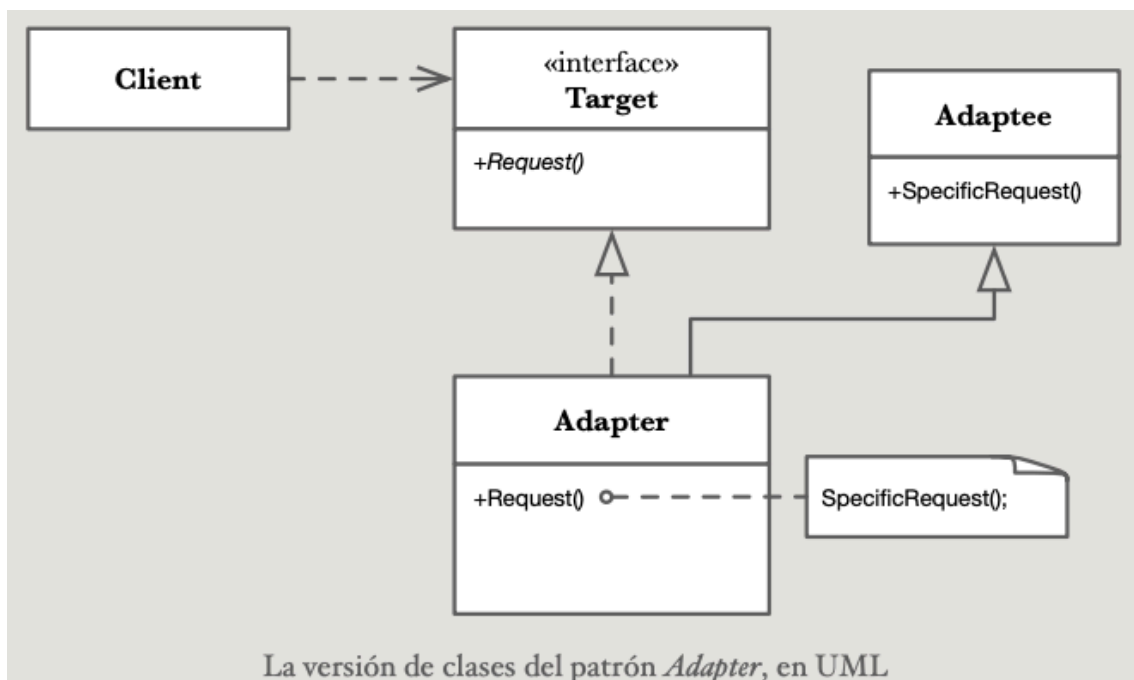
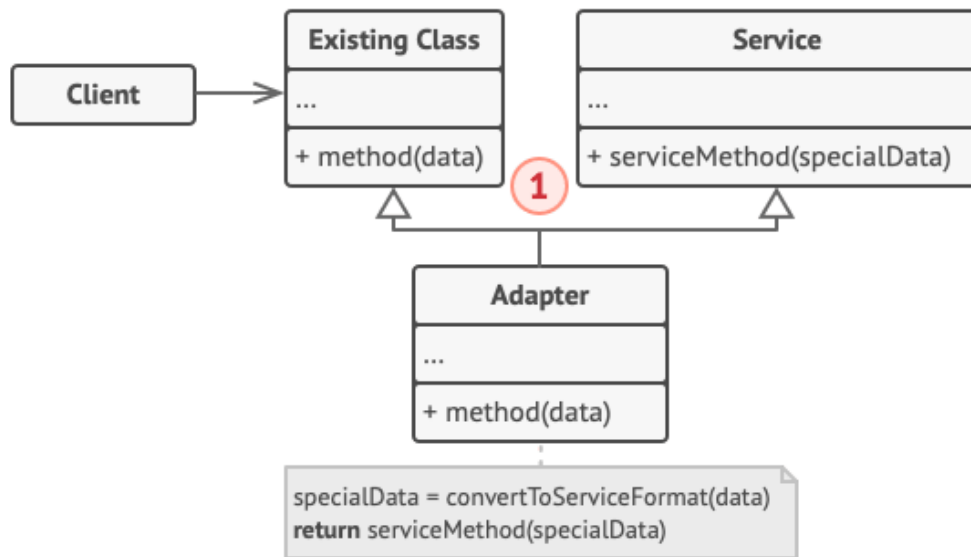


11. Patrón ADAPTER

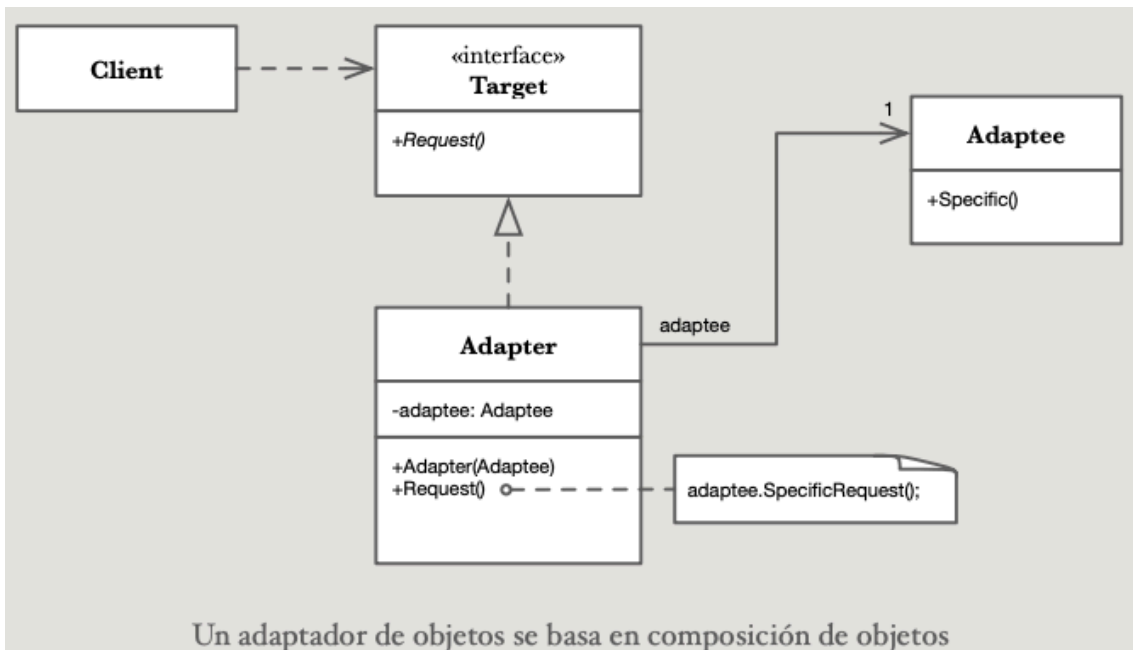
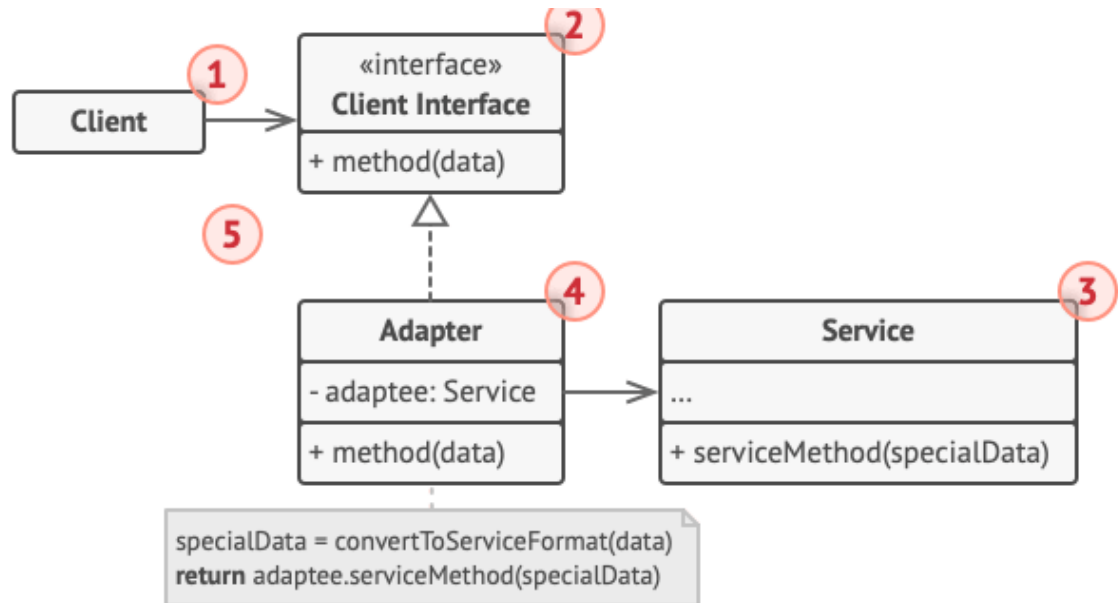
El **patrón Adapter** (*policy*) convierte la interfaz de una clase en otra que es la que esperan los clientes, para que trabajen juntas clases que no podrían al tener interfaces incompatibles, como un envoltorio.

Es un **patrón estructural para objetos y clases**.

→ CLASES:



→ OBJETOS:



Motivación y aplicabilidad:

Queremos usar una clase existente pero no usa la interfaz que queremos, por lo que queremos una interfaz envoltorio que agrupe a interfaces no compatibles.

Para la versión de objetos: necesitamos usar varias subclases existentes pero sin adaptar su interfaz creando una subclase para cada una, sino para la interfaz de la clase padre.

Participantes:

1. Target:

Interfaz específica del dominio a usar por el cliente.

2. Cliente:

Usa los Target.



3. Adaptee:

Clase existente que tiene que ser adaptada.

4. Adapter:

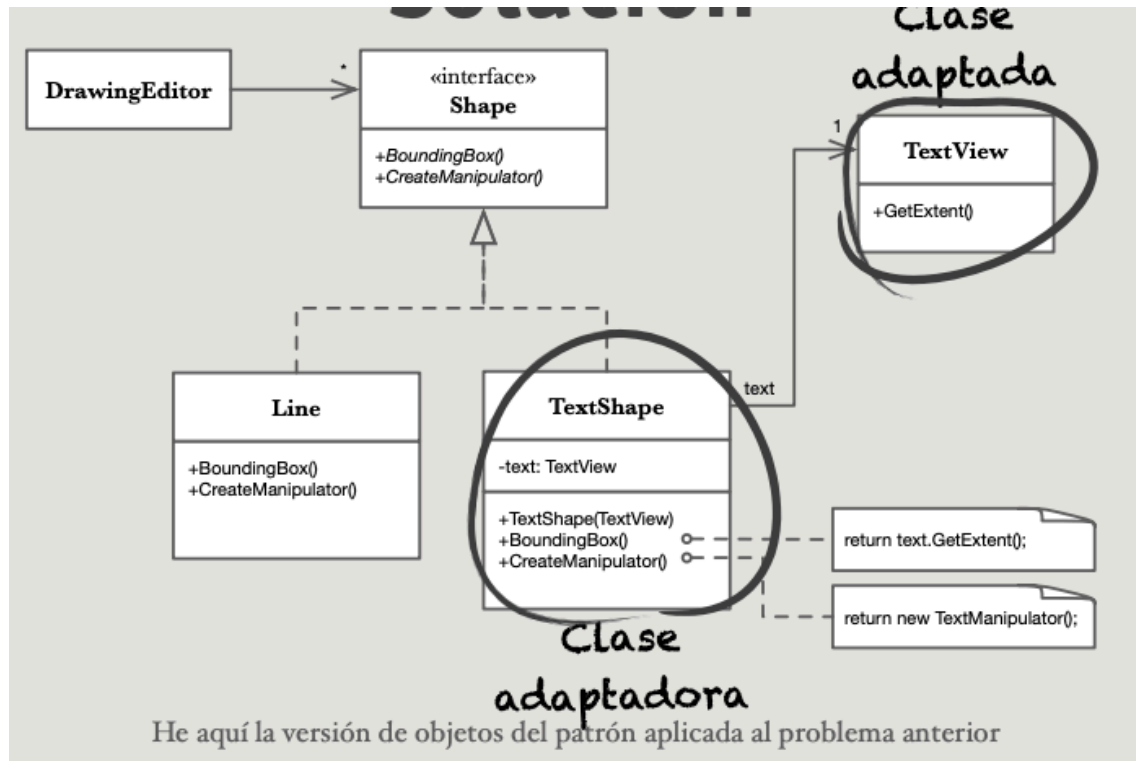
Adapta la interfaz de Adaptee a la de Target.

Consecuencias:

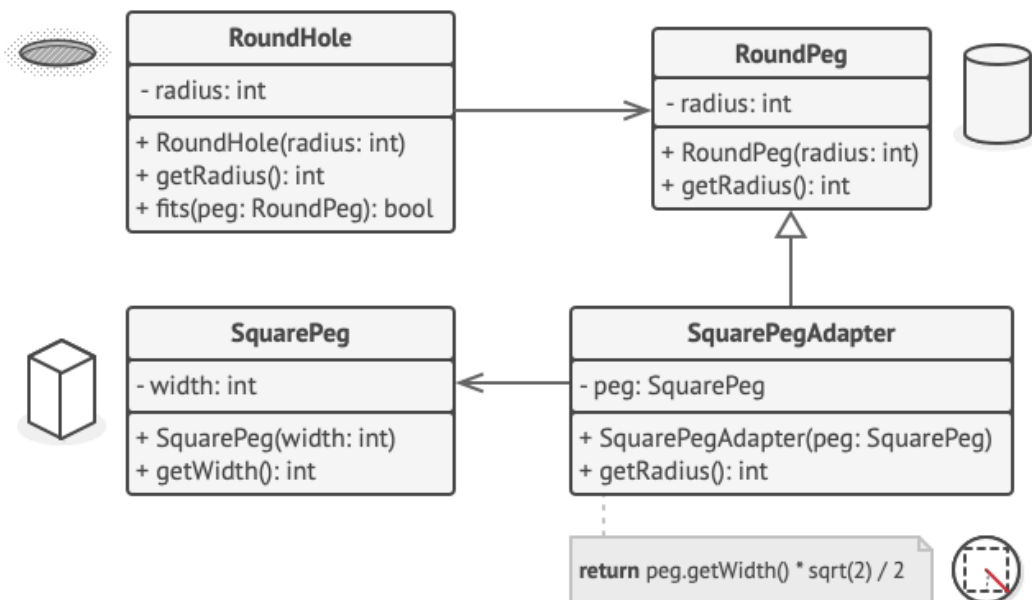
-  Adaptador de clases: introduce solo un objeto adicional, y permite que el adaptador redefina parte del comportamiento de la clase adaptada.
- Adaptador de objetos: permite adaptar objetos existentes, y permite que funcione para una clase adaptada y sus subclases.
-  Adaptador de clases: adapta una clase concreta, no se puede usar para adaptar una clase y todas sus subclases.
- Adaptador de objetos: no es del tipo del objeto adaptado.

Ejemplos de uso:

A)



B)

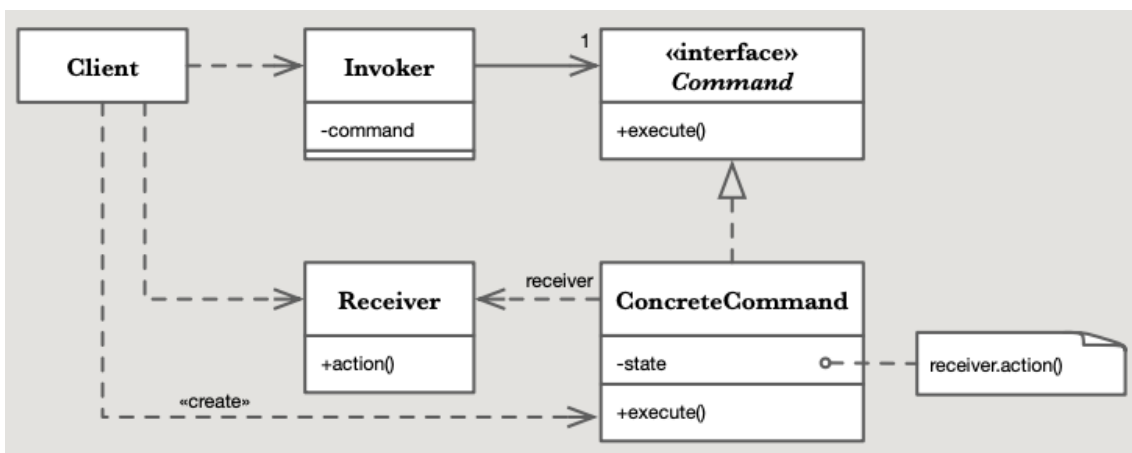
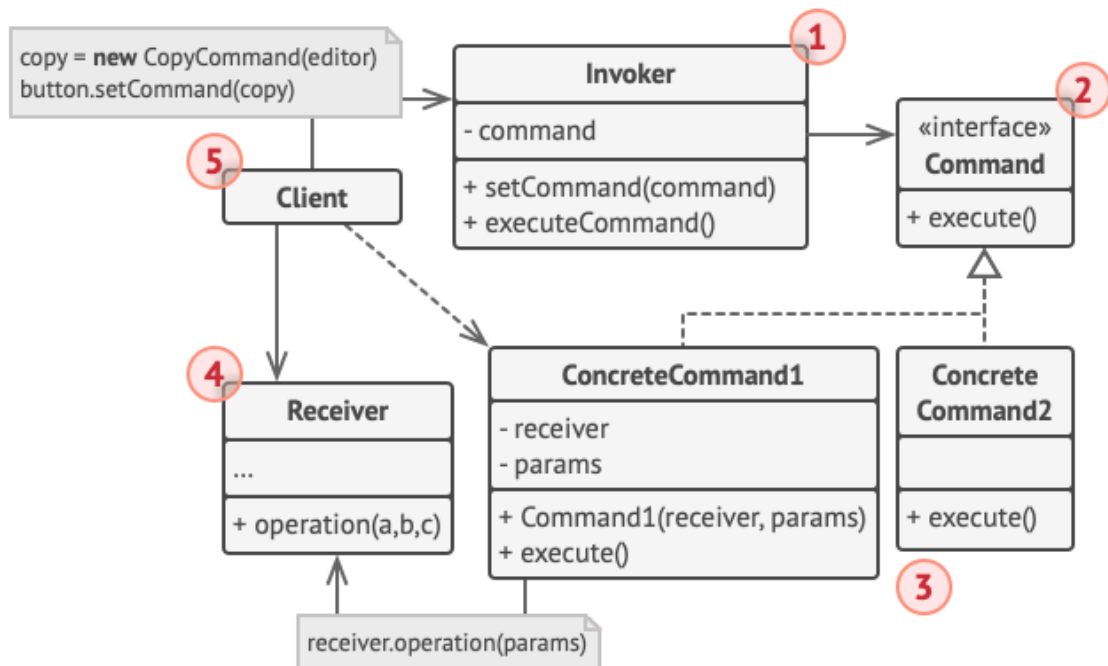


Adaptando piezas cuadradas a agujeros redondos.

12. Patrón COMMAND

El **patrón Command** encapsula una petición dentro de un objeto, permitiendo parametrizar a los clientes con distintas peticiones, encolarlas, guardarlas en un registro o implementar un mecanismo de deshacer/repetir.

Es un **patrón de comportamiento de objetos**.



Motivación y aplicabilidad:

Queremos desacoplar el objeto que invoca a la operación de aquel que la realiza.

Nos permite parametrizar objetos con una determina acción, especificar/guardar/ejecutar la petición en distintos momentos (<<desacoplamiento temporal>>) en un registro (log), usar transacciones y permite deshacer/repetir.

Participantes:

1. Command:

Interfaz para ejecutar una operación.

2. ConcreteCommand:

Asocia un objeto Receiver a una acción e implementa la ejecución llamando a las operaciones de dicho objeto receptor.

3. Client:

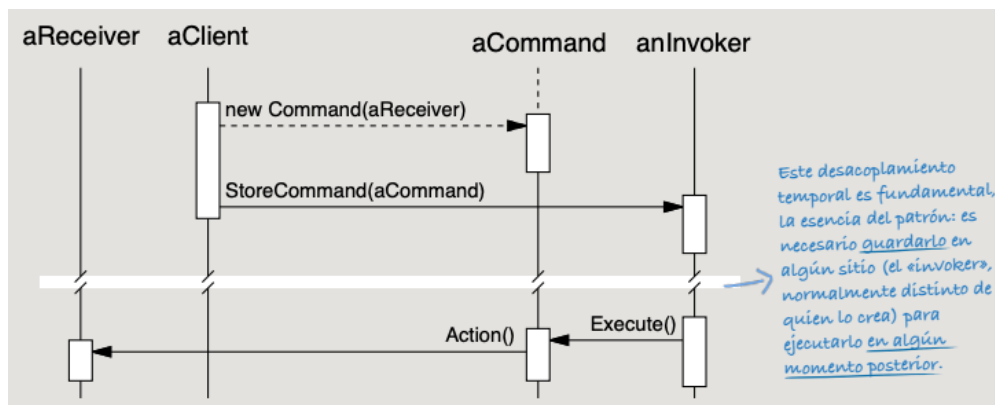
Crea un objeto ConcreteCommand y establece el Receiver.

4. Invoker:

Le pide al Command que se ejecute.

5. Receiver:

Es quien lleva a cabo la acción.



Consecuencias:

- Desacoplamiento.
- Se puede usar el Composite, o añadir nuevas acciones.

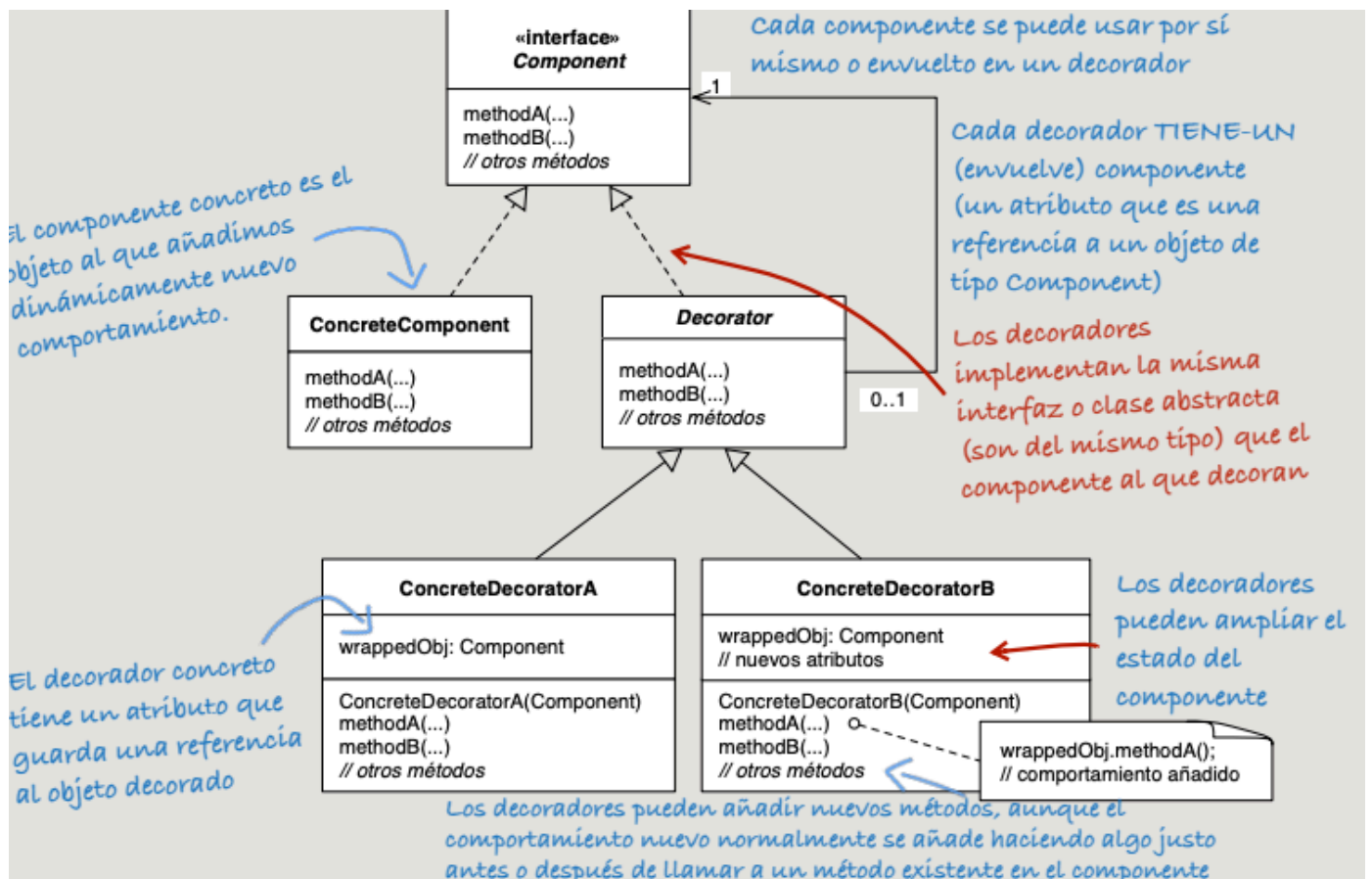
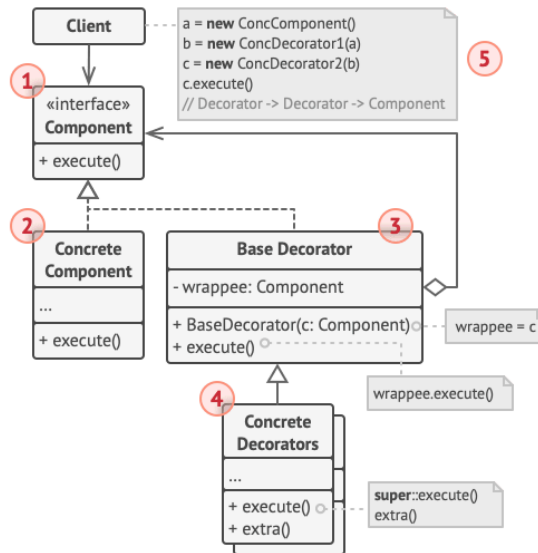
Implementación:

Para implementar undo/redo, podríamos tener una lista como un historial y haciendo copias del Command mediante el patrón Prototype.

13. Patrón DECORATOR

El **patrón Decorator** permite añadir responsabilidades a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Es un **patrón estructural de objetos**.



Motivación y aplicabilidad:

Queremos añadir responsabilidades a objetos individuales, no una clase entera.

No queremos usar la herencia, ya que sería inflexible y explotaríamos las clases. Entonces, envolvemos el componente en otro objeto que es quien añade el borde, con la misma interfaz que este, por lo que la presencia del decorador es transparente.

Participantes:

1. Component:

Define la interfaz de los objetos a los que añadir responsabilidades dinámicamente.

2. ConcreteComponent:

Implementación de los componentes.



3. Decorator:

TIENE-UN (envuelve) un objeto component, con su misma interfaz.

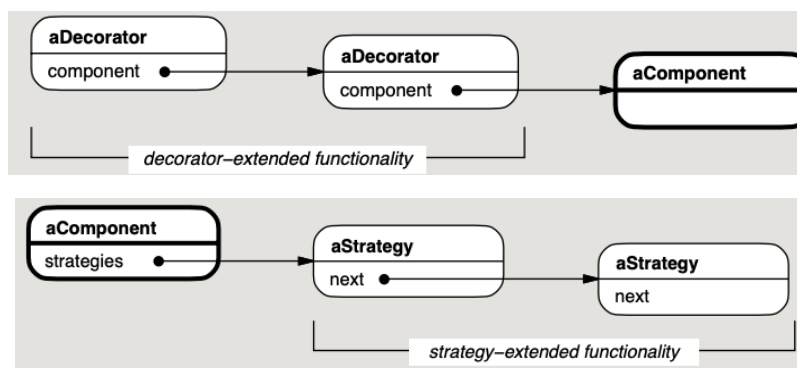
4. ConcreteDecorator:

Añade responsabilidades al componente.

Consecuencias:

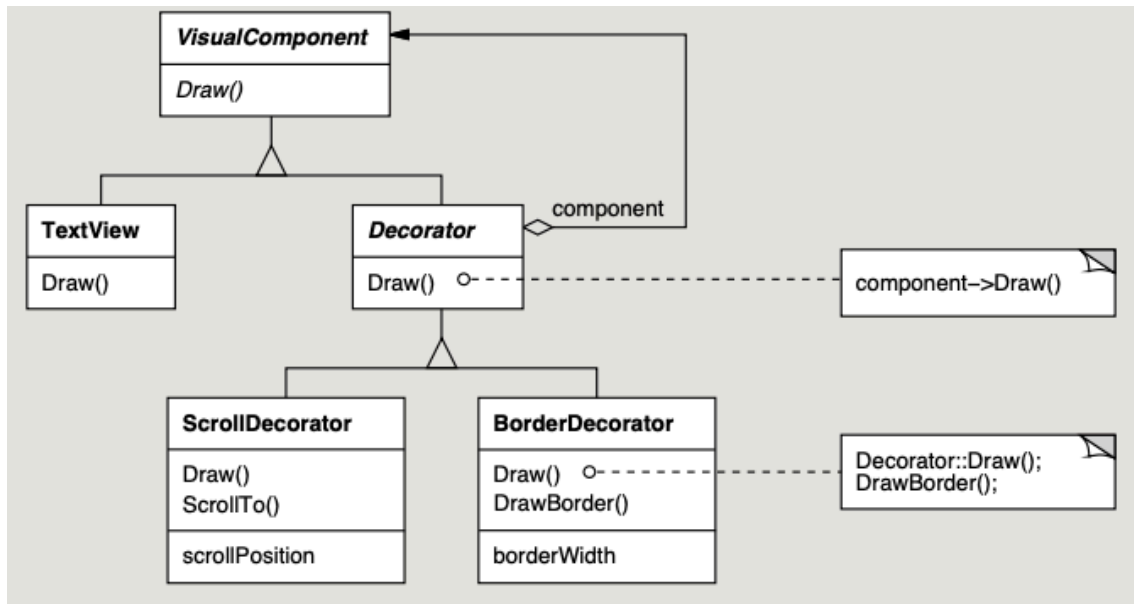
-  Más flexible que la herencia.
- Evita que las clases se llenen de nuevas funcionalidades.
-  Aunque Component y Decorator tengan la misma interfaz, no son lo mismo.
- Demasiados objetos pequeños.

Implementación:

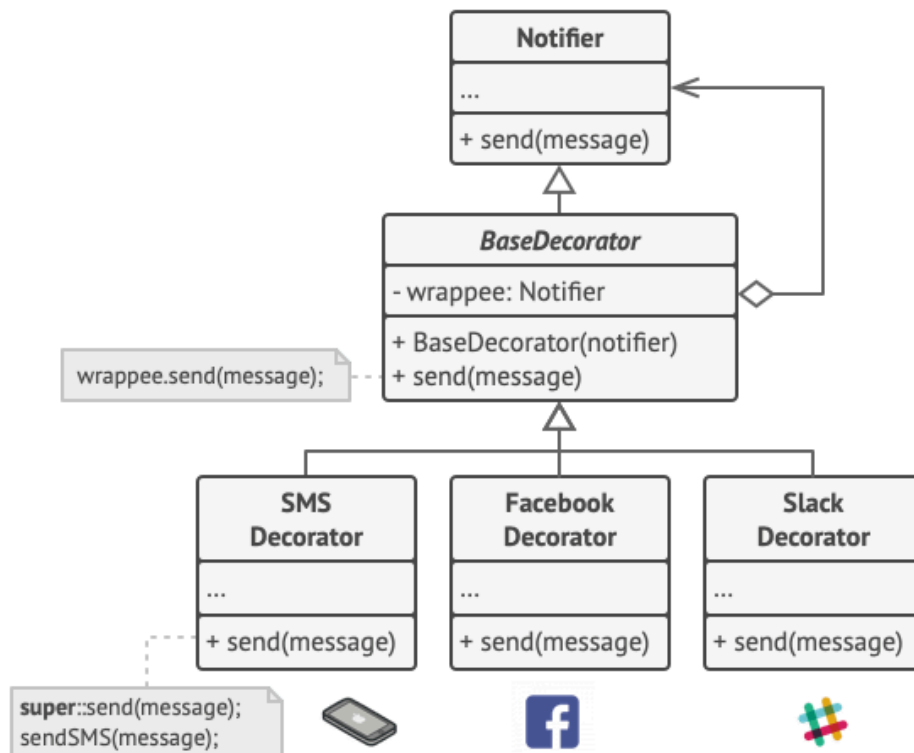


Ejemplos de uso:

A) Añadir decoraciones o funcionalidades concretas a elementos visuales:



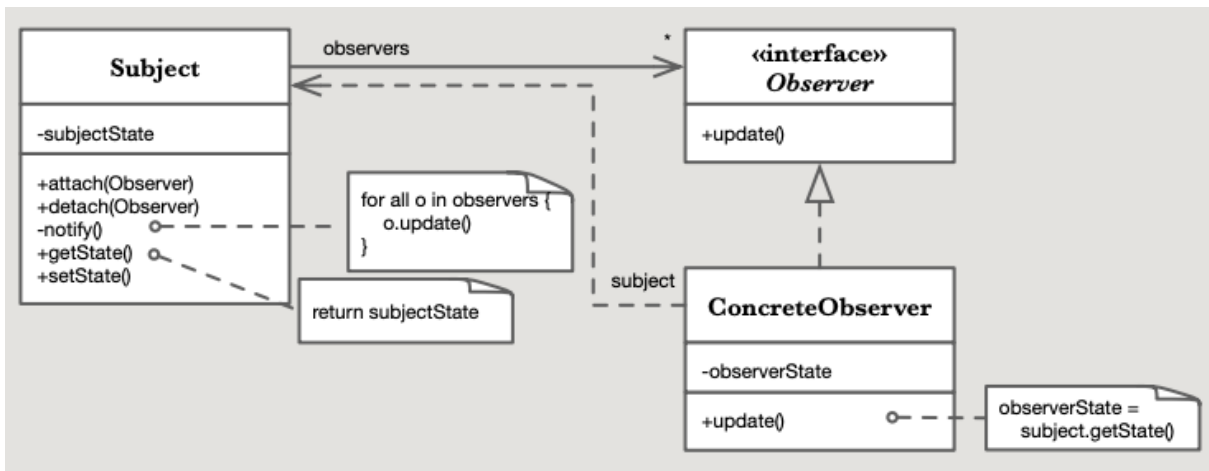
B) Añadir estilos a diferentes notificaciones:



14. Patrón OBSERVER

El **patrón Observer** define una dependencia one-to-many entre objetos de modo que, cuando uno cambie su estado, todos los dependientes se modifican y actualizan automáticamente.

Es un **patrón de comportamiento de objetos**.



Motivación y aplicabilidad:

Lo utilizamos cuando un objeto necesita notificar a otros cambios en su estado, sin hacer presunciones sobre quiénes son dichos objetos (es decir cuando queremos bajo acoplamiento). También cuando en la abstracción hay 2 aspectos dependientes entre sí.

Participantes:

1. Subject:

Conoce a sus observadores y da una interfaz para que se suscriban o borren sus observadores.

2. Observer:

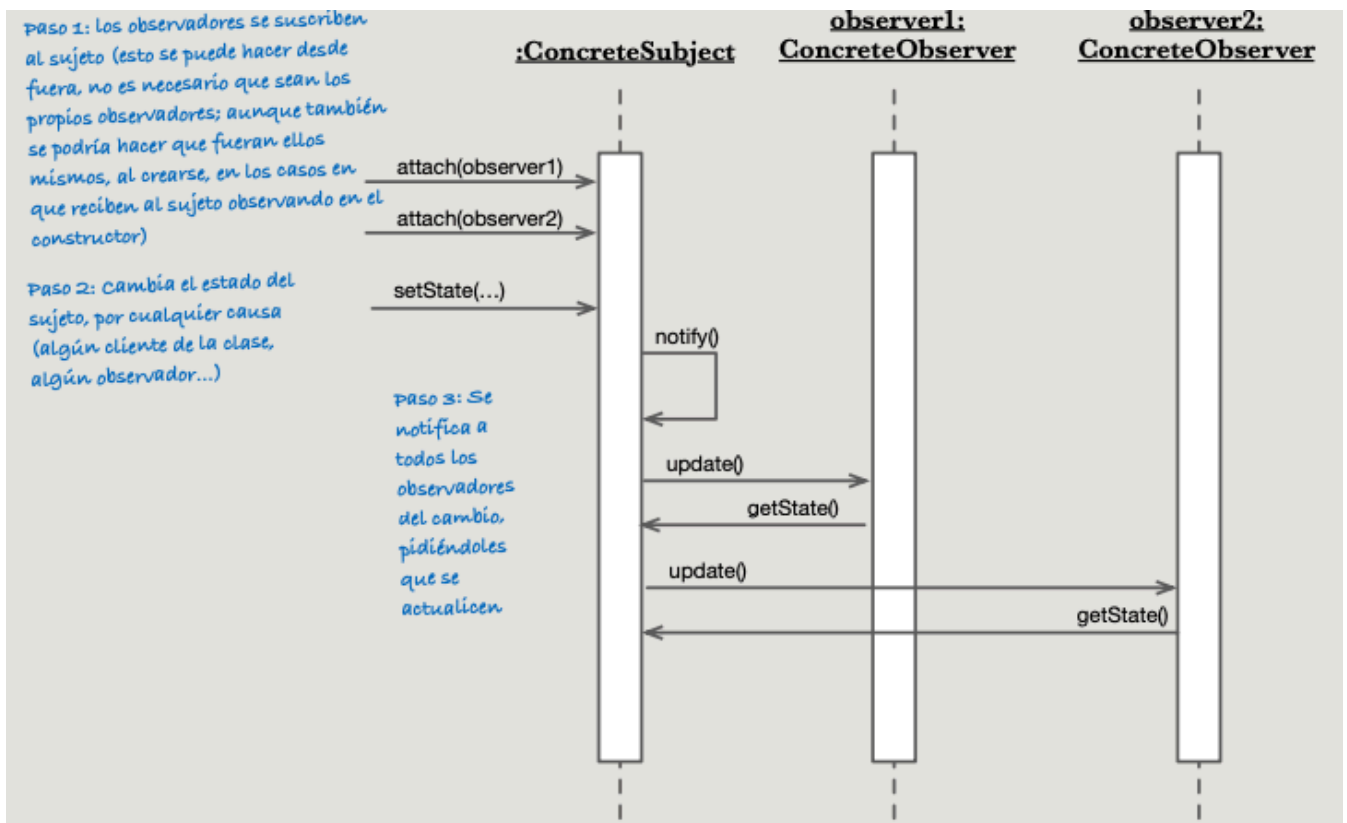
Define una interfaz para actualizar los objetos que deben ser notificados por cambios en el objeto Subject.

3. ConcreteSubject:

Guarda el estado de interés para los ConcreteObserver y envía una notificación a sus observadores cuando cambia de estado.

4. ConcreteObserver:

Mantiene una referencia a ConcreteSubject e implementa la interfaz observer para actualizarse y ser consistente con el objeto observado. Pueden pedir información al ConcreteSubject para reconciliarse.

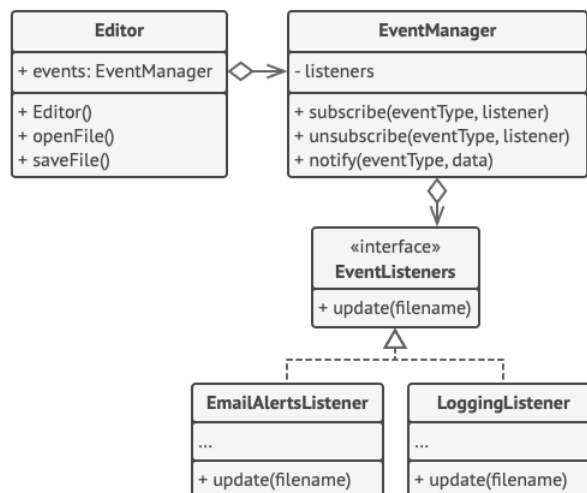


Consecuencias:

- ✓ Permite varios objetos observados y observadores.
- Hay acoplamiento abstracto entre Subject y Observer.
- No se especifica el receptor de una actualización, se manda a todos.
- ✗ Podría haber actualizaciones en cascada muy ineficientes.

Ejemplos de uso:

A)

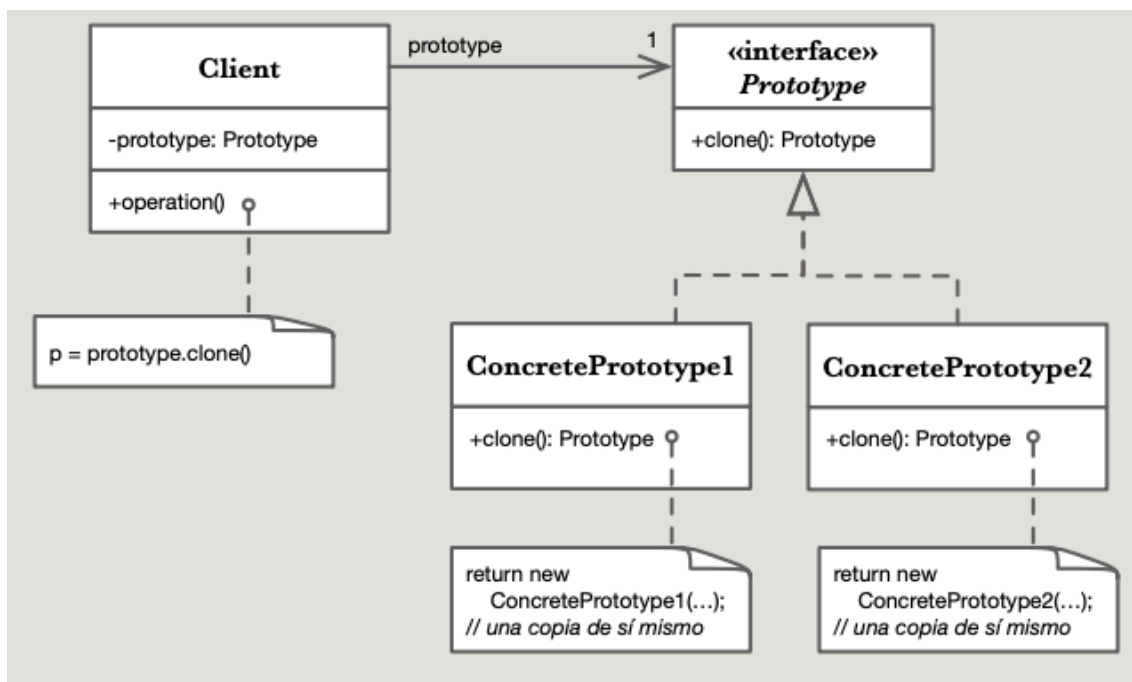
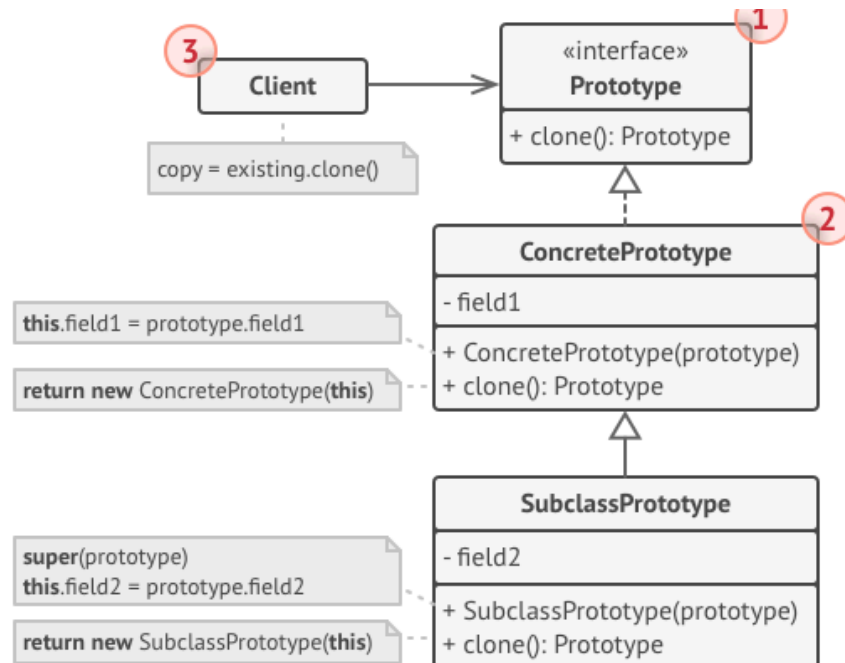


Notificar a objetos sobre eventos que suceden a otros objetos.

15. Patrón PROTOTYPE

El **patrón Prototype** especifica los tipos de objetos a crear usando una instancia prototípica, y crea objetos copiando dicho prototipo.

Es un **patrón de creación de objetos**.



Motivación y aplicabilidad:

Queremos que el sistema no pueda/deba conocer cómo se crean, componen y representan los productos, y además:

- Las clases a instanciar se definen dinámicamente, o
- Queremos construir jerarquía paralela de factorías
- Cuando las instancias tengan unos pocos estados y queramos evitar crear subclases para clonar prototipos con el estado necesario

Participantes:

1. Prototype:

Declara la interfaz para clonarse



2. ConcretePrototype:

Implementa la clonación

3. Client:

Crea un nuevo objeto diciendo al prototipo que se clone

Consecuencias:

-  Al igual que AbstractFactory, oculta las clases concretas de producto al cliente.
- Permite añadir/eliminar productos dinámicamente.
- Permite especificar nuevos objetos modificando sus propiedades y estructura, es como crear nuevas clases sin instanciarlas explícitamente.
- Reduce las subclases.
-  La clonación puede no ser fácil.

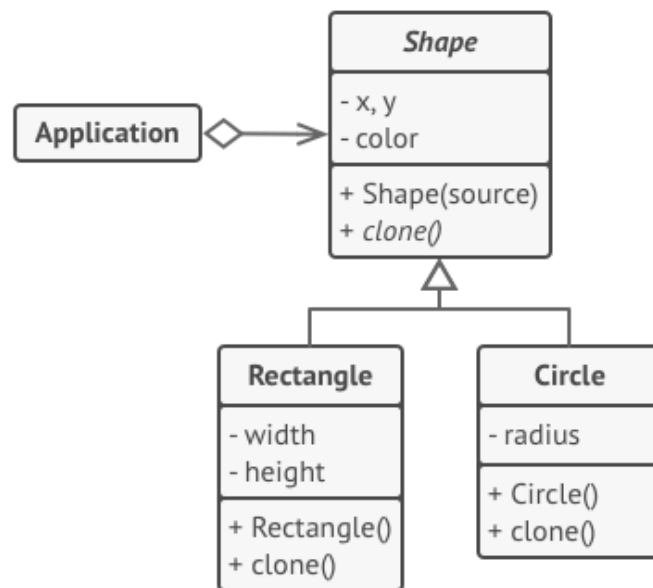
Implementación:

Hay que usar un registro de prototipos cuando estos no son fijos.

Hay que saber implementar la clonación (profunda o superficial).

Hay que saber inicializar los prototipos.

Ejemplos:

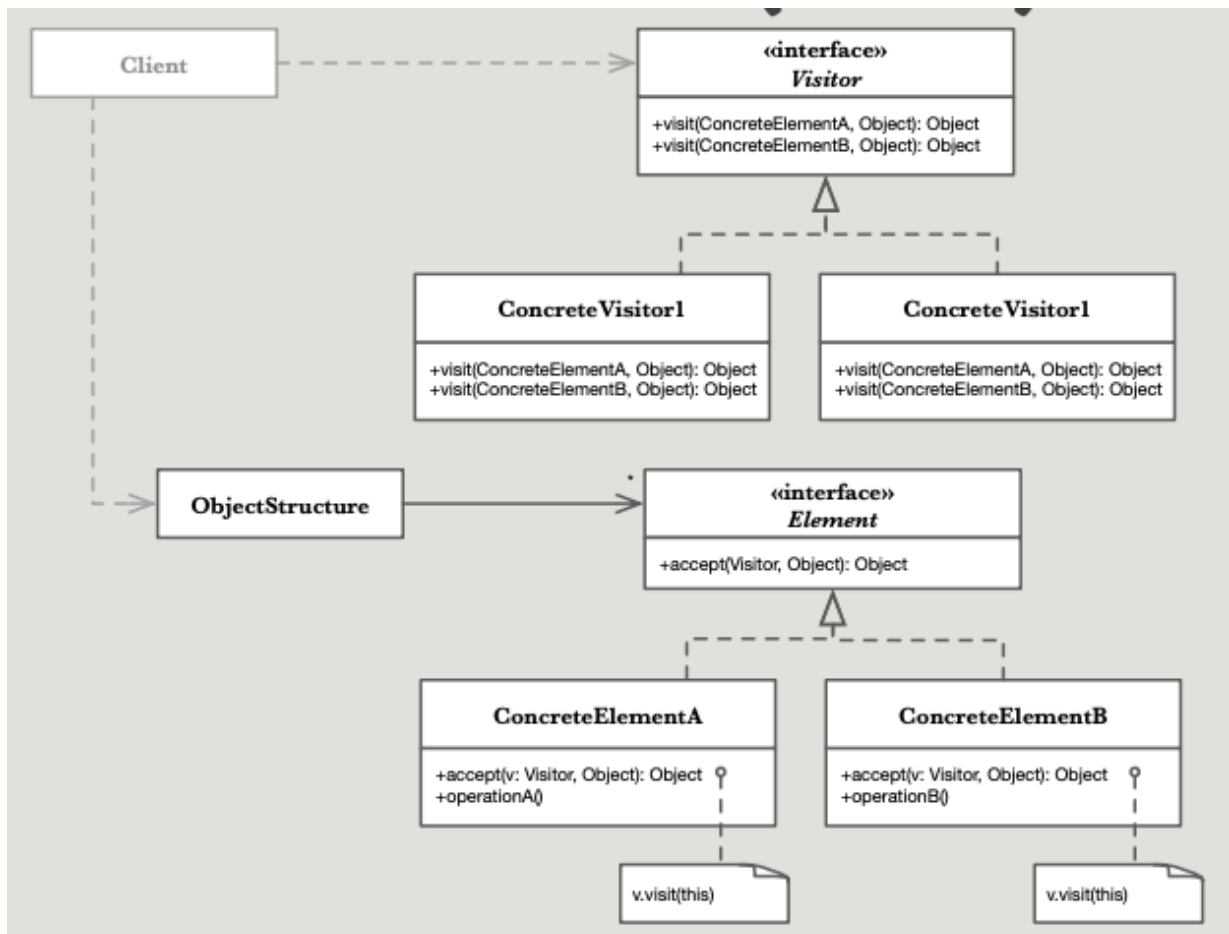


Clonación de un grupo de objetos que pertenece a una jerarquía de clase.

16. Patrón VISITOR

El **patrón Visitor** representa una operación a realizar sobre una estructura de objetos, que permite definir nuevas operaciones sin modificar las clases de los elementos sobre los que opera.

Es un **patrón de comportamiento de objetos**.



Motivación y aplicabilidad:

Lo aplicamos cuando existen muchas clases con distintas interfaces y queremos realizar operaciones que dependen de la clase concreta, y queremos evitar contaminar las clases con estas operaciones. Estas operaciones son juntadas en una clase.

Participantes:

1. Visitor:

Declara la operación 'visit' para determinar la clase concreta de elemento que está siendo visitada.

2. ConcreteVisitor:

Implementa cada operación declarada por Visitor, con un algoritmo definido para cada clase que se puede visitar.

3. Element:

Define una operación 'accept' que recibe un Visitor como argumento.



4. ConcreteElement:

Implemente la operación accept..

5. ObjectStructure:

Contiene a los elementos para que el visitante los pueda visitar, puede ser un compuesto (Composite) o una colección.

Consecuencias:

-  El visitante permite añadir nuevas operaciones.
- Agrupamos operaciones relacionadas de las que no lo están.
-  Es difícil añadir nuevas clases de elementos concretos.