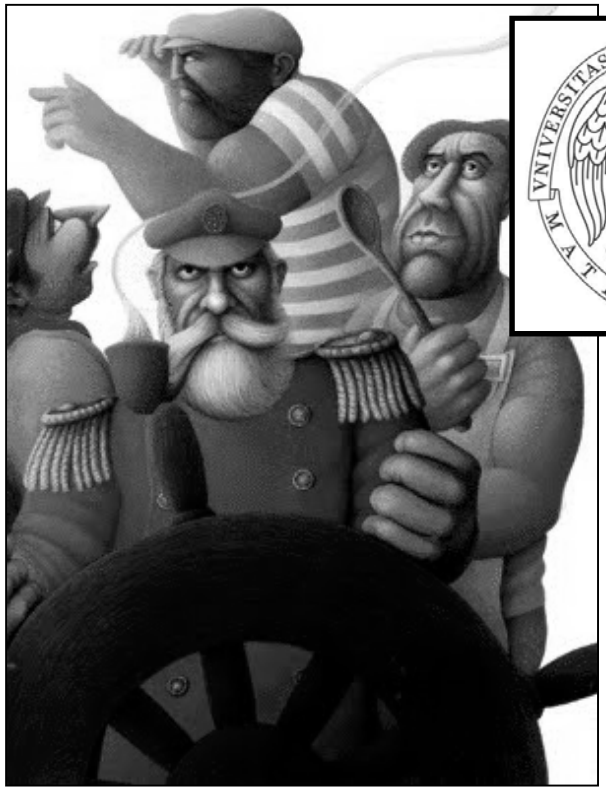
LPS: Patrones Básicos



Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

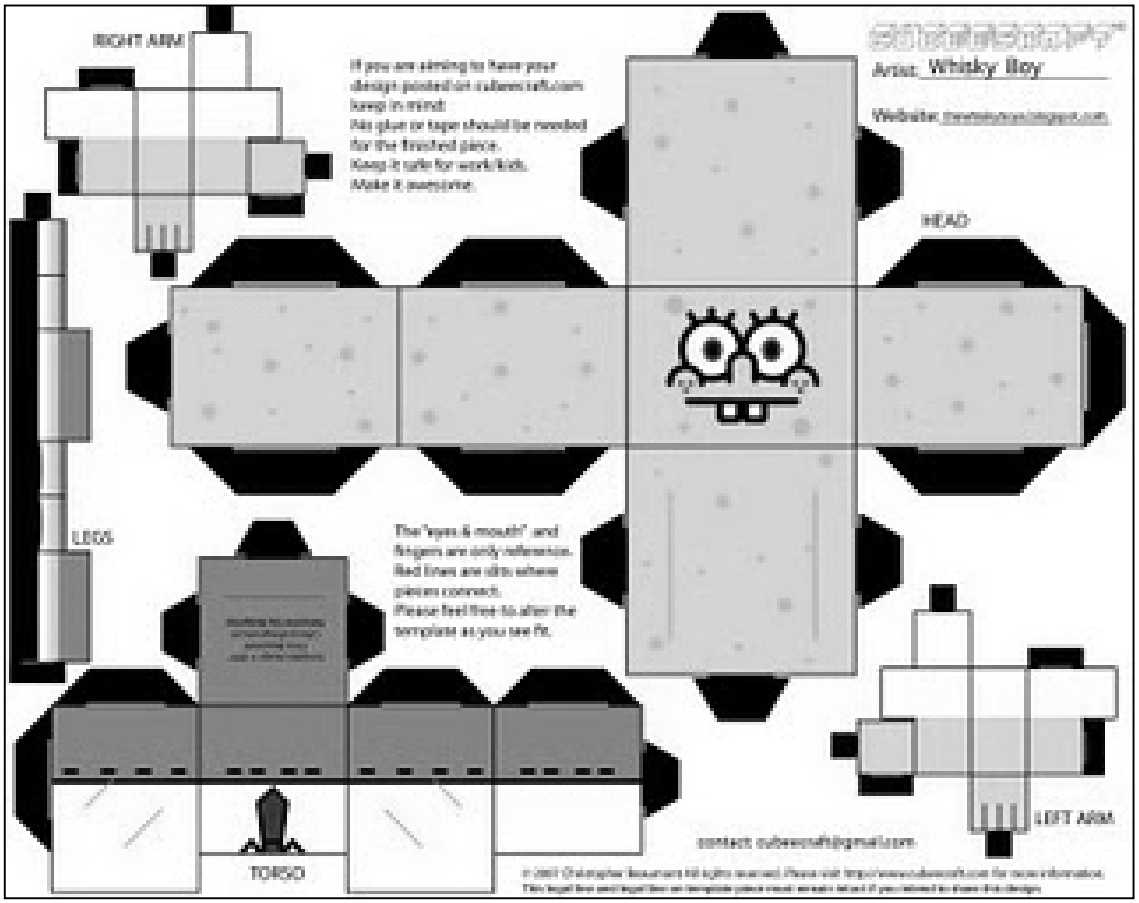
Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

Patrones Básicos

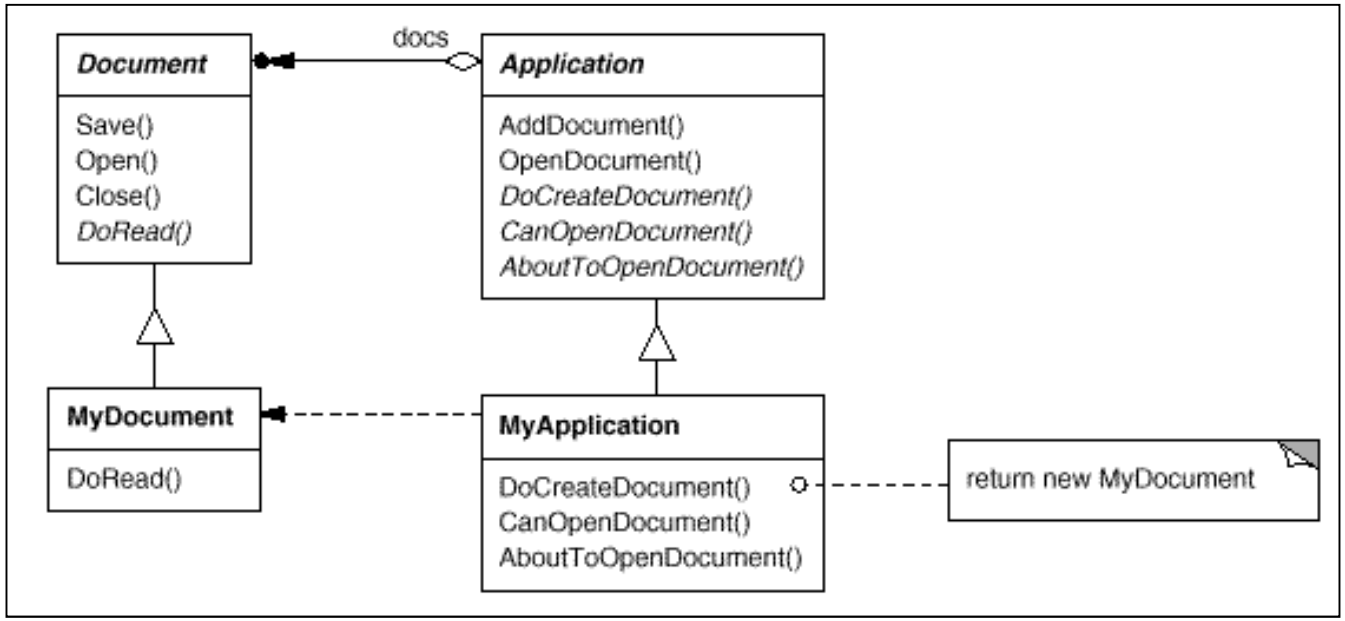
- Seguimos estudiando algunos de los patrones más populares del libro de *GoF*
  - Que los llamemos "básicos" aquí sólo quiere decir que parecen algo más frecuentes, pero *no* sencillos
- Mismas referencias que en la introducción a los Patrones de Diseño Software

Método plantilla



Método plantilla

- **Nombre original:** Template Method
- **Propósito:** Permite implementar un algoritmo genérico/esquemático, delegando los detalles en subclases
  - Ejemplo: ¿Cómo diseñamos un armazón reutilizable para crear aplicaciones ofimáticas que trabajan con sus documentos específicos?

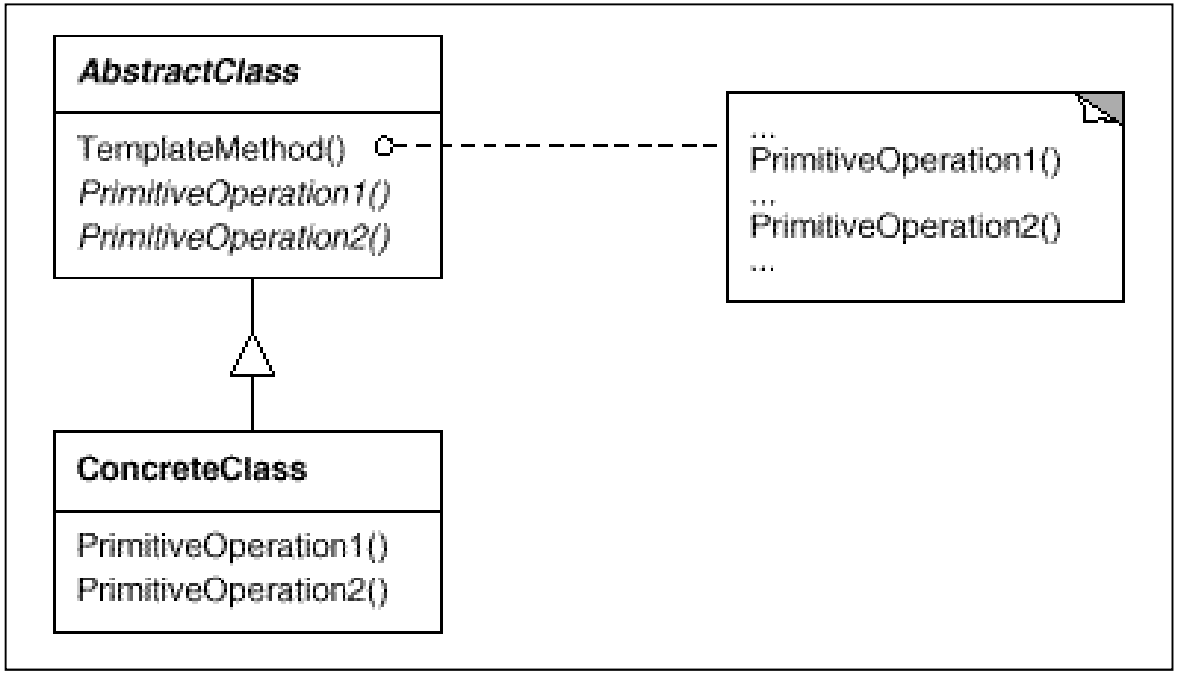


Método plantilla

```
class Application {
  public final void openDocument (String filename) {
    if (canOpenDocument()) {
      Document d = doCreateDocument();
      if (d) {
        _docs.addDocument(d);
        aboutToOpenDocument(d);
        d.open();
        d.doRead();
      }
    }
  }
}
```

Solución

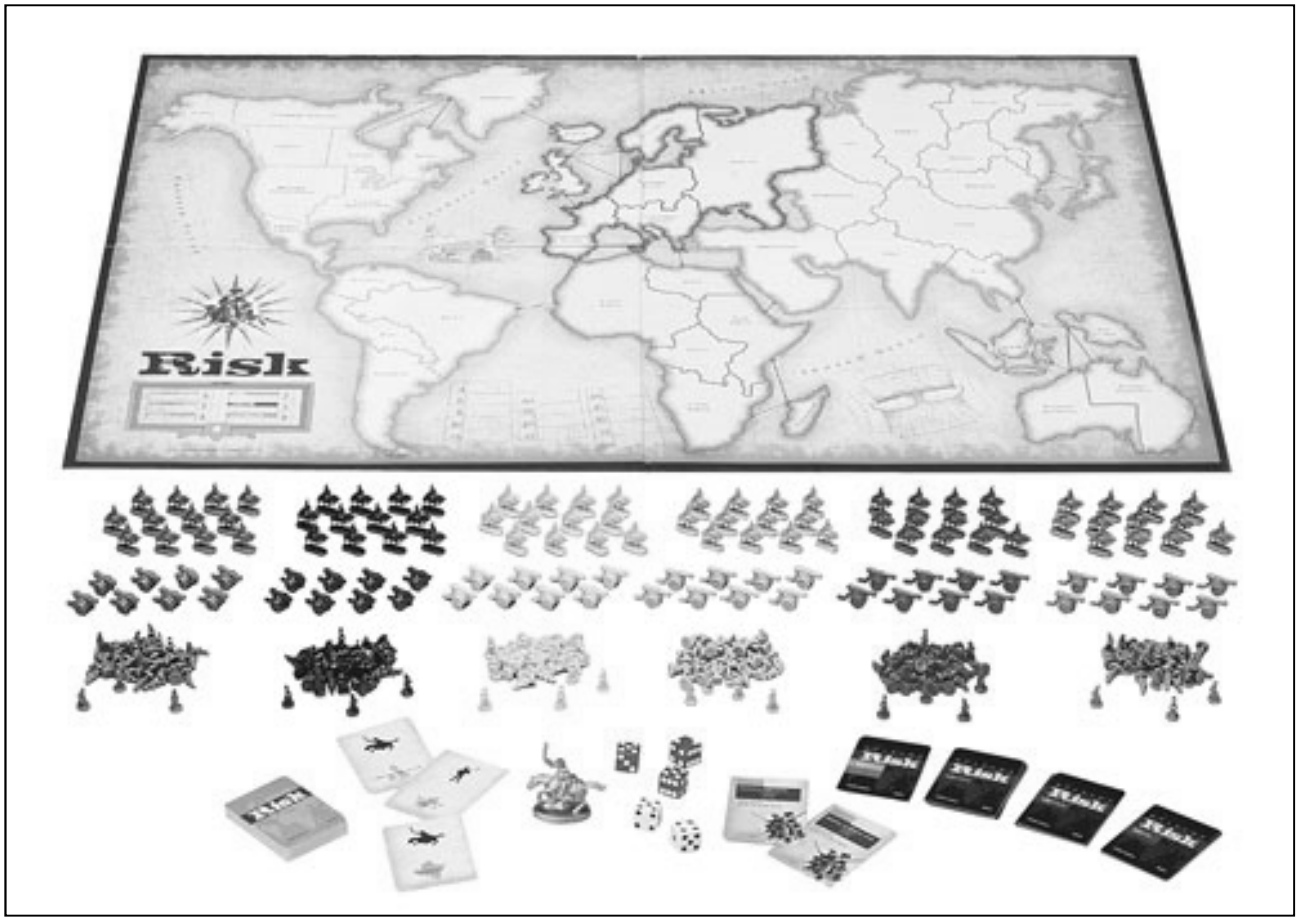
- *AbstractClass* declara en abstracto las operaciones primitivas que deben implementar sus subclases pero *implementa* (incluso de manera final) el método plantilla que las usa
- *ConcreteClass* implementa dichas operaciones primitivas, que encapsulan la única parte "variable" que admite la plantilla



Consecuencias

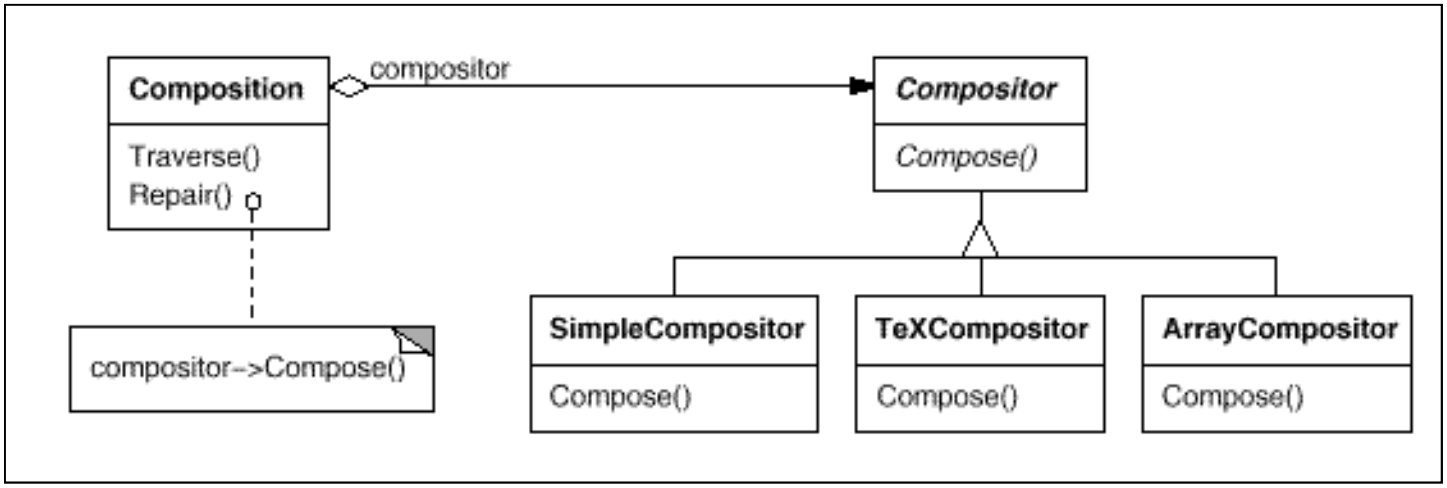
- Permite "extraer" comportamiento común a varias clases y llevarlo a un sólo método
  - Evitamos la repetición de código
- Da lugar a una "inversión" del flujo de control
  - Principio de Hollywood
    "*No nos llame; ya le llamaremos nosotros*"
- Las operaciones primitivas en las que se basa el método plantilla pueden ser de varios tipos
  - Operaciones abstractas, que han de ser definidas obligatoriamente por las clases concretas
  - Operaciones "gancho" (*hook*) que dan una implementación por defecto, aunque puede variar (= pueden ser sobreescritas)

Estrategia



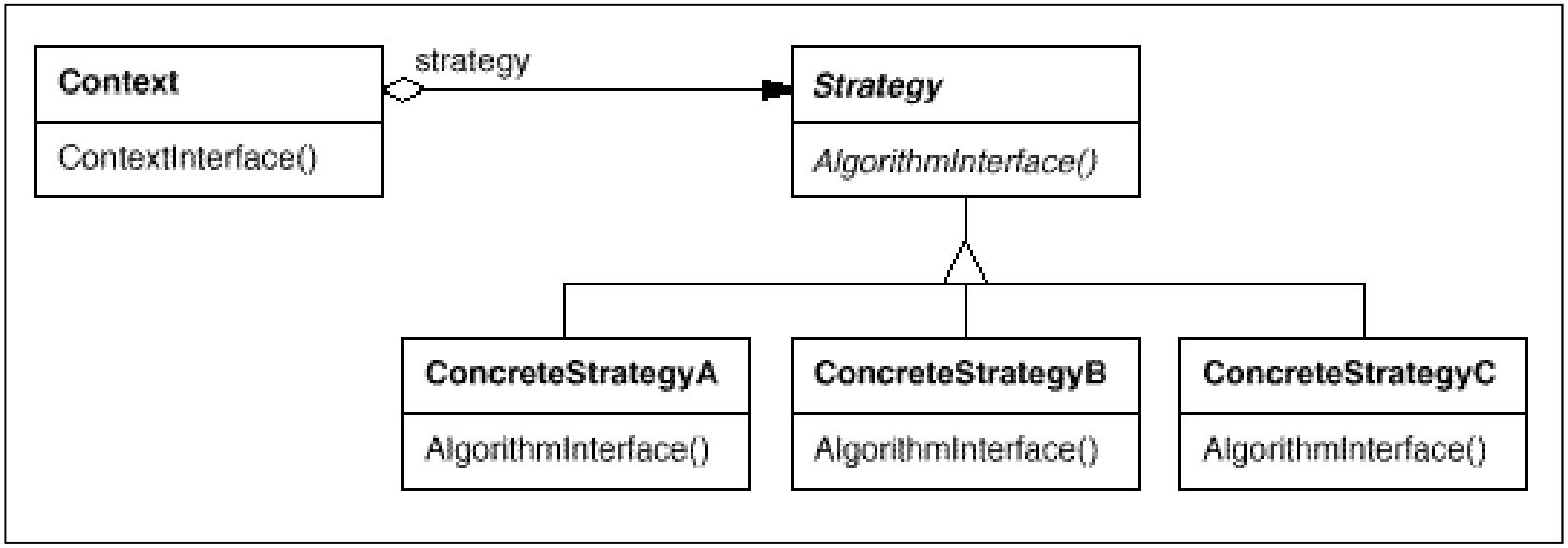
Estrategia

- **Nombre original:** Strategy
- **Propósito:** Permite la coexistencia de una "familia de algoritmos" (varios algoritmos pensados para resolver una misma tarea), encapsulándolos como objetos
  - Desacopla el código de los algoritmos del código del cliente
  - Ejemplo: ¿Cómo se divide en líneas un texto cuando lo mostramos en un cuadro de texto? Hay muchas formas posibles: línea a línea, párrafo a párrafo –como hace TeX-, palabra a palabra –tipo array/fila Excel-, etc.



Solución

- *Context* es la clase que necesita usar alguno de los algoritmos
- *Strategy* define una interfaz común a los distintos algoritmos, implementados por cada *ConcreteStrategy*
- Normalmente el cliente es quien decide qué estrategia usar, inicializando (o actualizando) el contexto adecuadamente



Consecuencias

- Al separar los algoritmos del contexto se hace más fácil definir nuevos algoritmos o variantes de los mismos
  - Hemos llevado "parte" del flujo de control a la jerarquía de clases
  - El patrón resulta más útil cuanto más complicados los algoritmos y/o más se apoyan en estructuras de datos complejas
    - Mediante herencia "extraemos" la parte común a todos los algoritmos
    - Se hace más clara la relación que existe entre los distintos algoritmos
- Sólo debemos dejar al cliente decidir la estrategia cuando es relevante para él conocer los distintos algoritmos disponibles (o poder alterar el contexto)
- Si hay mucha variación en el nivel de complejidad de los algoritmos puede ocurrir que a los más simples el contexto les pase mucha información innecesaria

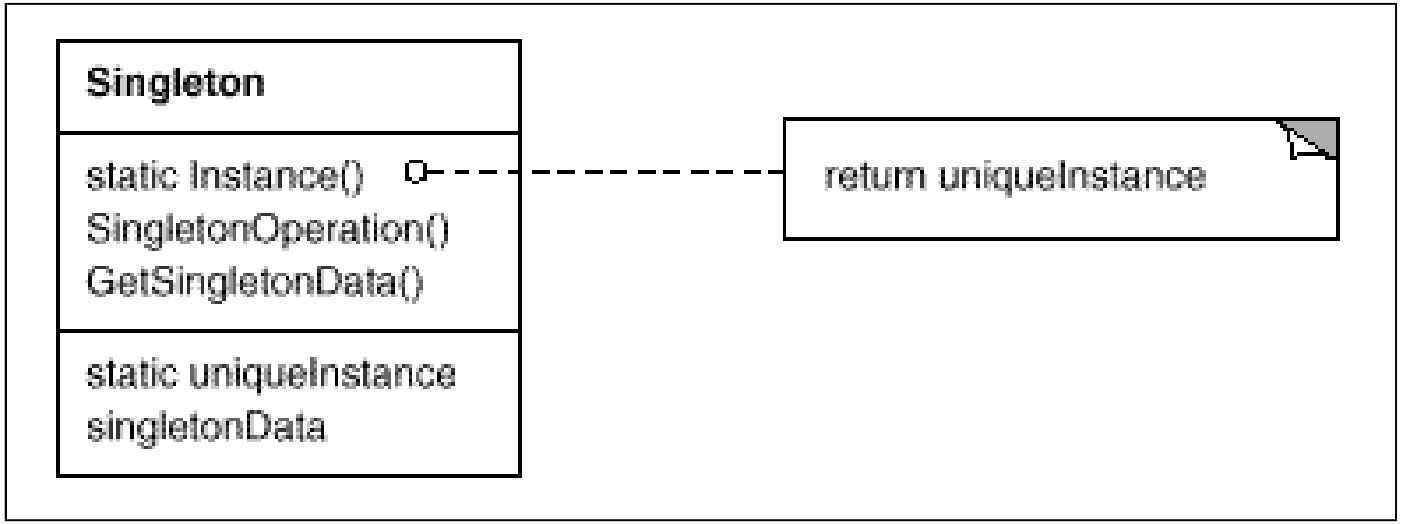
Ejemplar único

Ejemplar único

- **Nombre original:** Singleton
- **Propósito:** Garantiza que sólo existe un ejemplar (ó *N, número finito*) de una determinada clase y ofrece acceso a él
  - Ejemplo: ¿Cómo representamos en POO los recursos únicos de un sistema operativo como la cola de impresión, el sistema de archivos, el gestor de ventanas, etc.? Deben ser accesibles desde cualquier cliente, pero todos utilizan los mismos…

Solución

- Se define un método que permite acceder (e incluso crear si es necesario) al ejemplar único en cuestión



Un posible código en Java

```
public class Singleton {
  private static Singleton INSTANCE = null;
  private Singleton() { } //No ofrecemos constructor
  private synchronized static void createInstance() { //Sincronizado y todo
    if (INSTANCE == null) {
      INSTANCE = new Singleton(); //Inicialización diferida
    }
  }
  public final static getInstance() { //Final para que no lo sobreescriban
    if (instance == NULL)
      instance = new Singleton();
    return instance;
  }
  @Override
  public Object clone() throws CloneNotSupportedException { //Sin clonación
    throw new CloneNotSupportedException();
  }

  ...

}
```

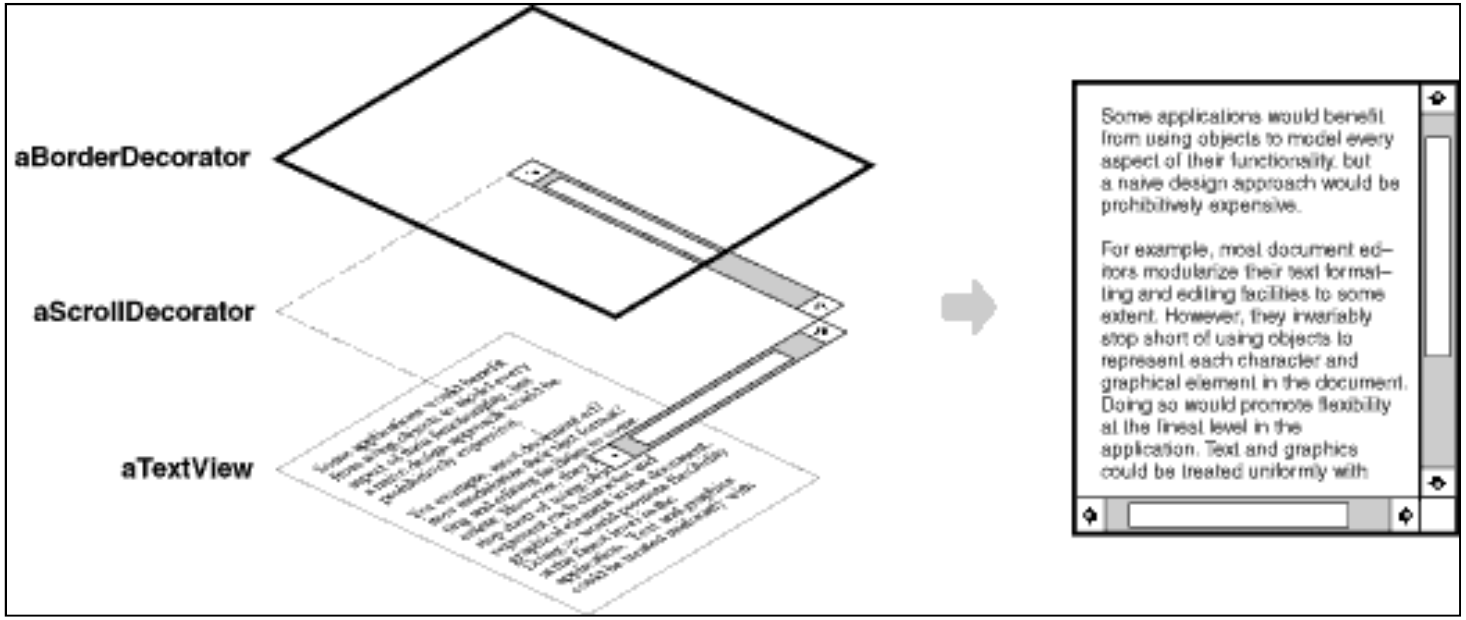
Consecuencias

- Tenemos acceso controlado al ejemplar único (o los *N ejemplares*) de dicha clase
  - Dicho *N* puede variar en ejecución, incluso
- Evita el uso de variables globales, siendo el acceso más elegante y tan "controlable" como el de cualquier otro objeto
- Podríamos definir subclases del ejemplar único (si la clase *no* es final) y determinar cual utilizar en tiempo de ejecución

Decorador

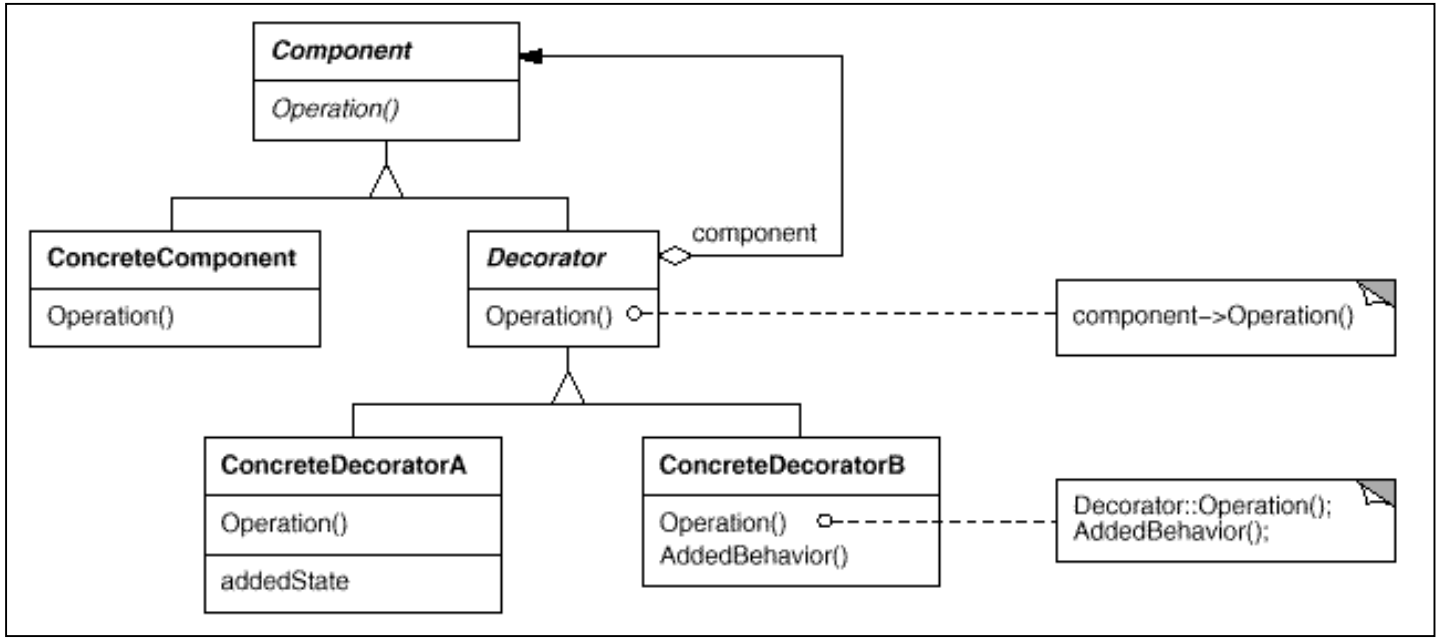
Decorador

- **Nombre original:** Decorator
- **Propósito:** Permite añadir dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa en tiempo de ejecución a la definición de subclases
  - Ejemplo: ¿Cómo podemos representar un panel de texto al que a veces se le añade una barra de desplazamiento, a veces también un borde, etc. sin necesidad de definir nuevas clases para cada combinación de elementos?

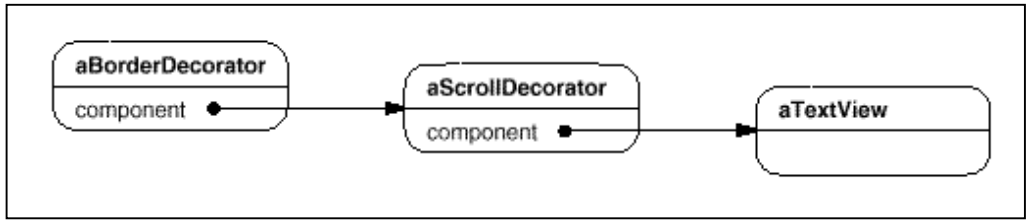


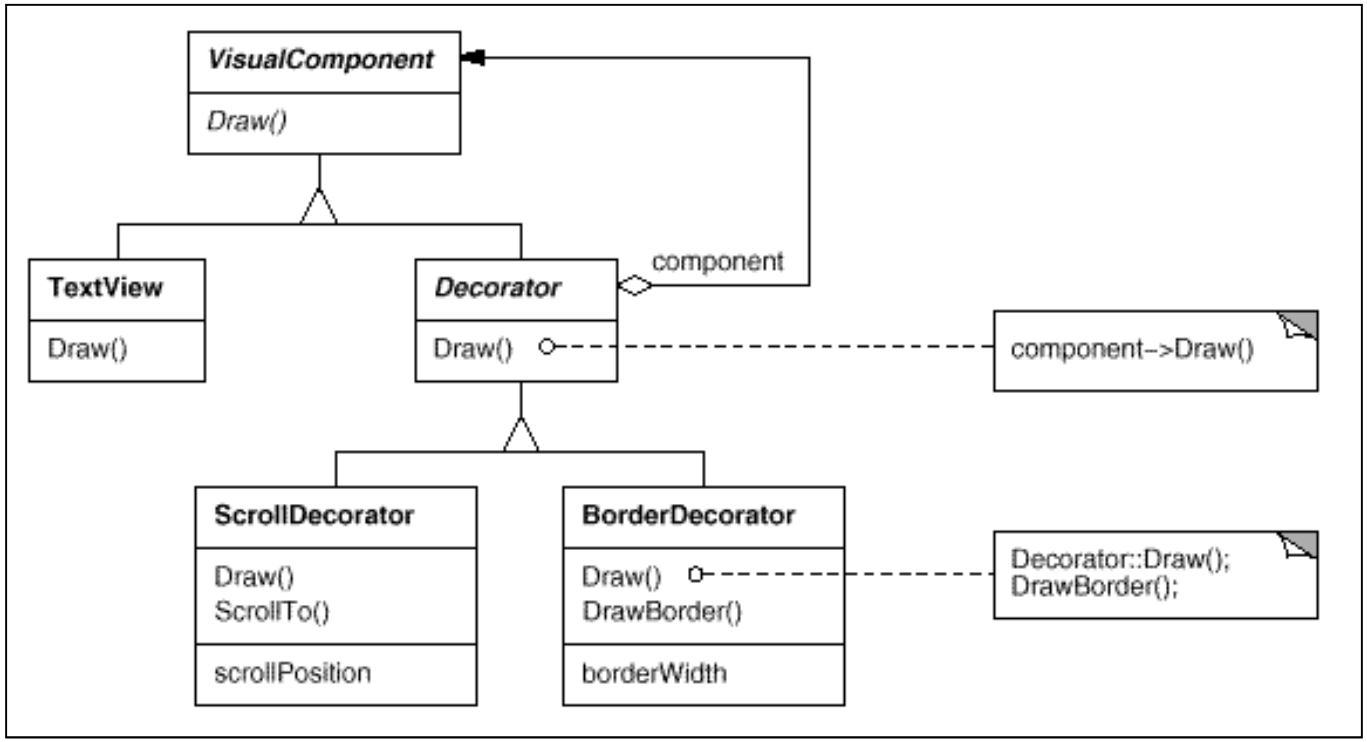
Solución

- *Component* define la interfaz de los objetos a los que se les va a poder añadir responsabilidades dinámicamente (= decoradores)
- *ConcreteComponent* define un objeto concreto de este tipo
- *Decorator* almacena una referencia a un objeto *Component* y define una interfaz que se ajusta a la de *Component* (puede ser más amplia)
- *ConcreteDecorator* añade responsabilidades al componente (más atributos y métodos que hagan falta)



Ejemplo

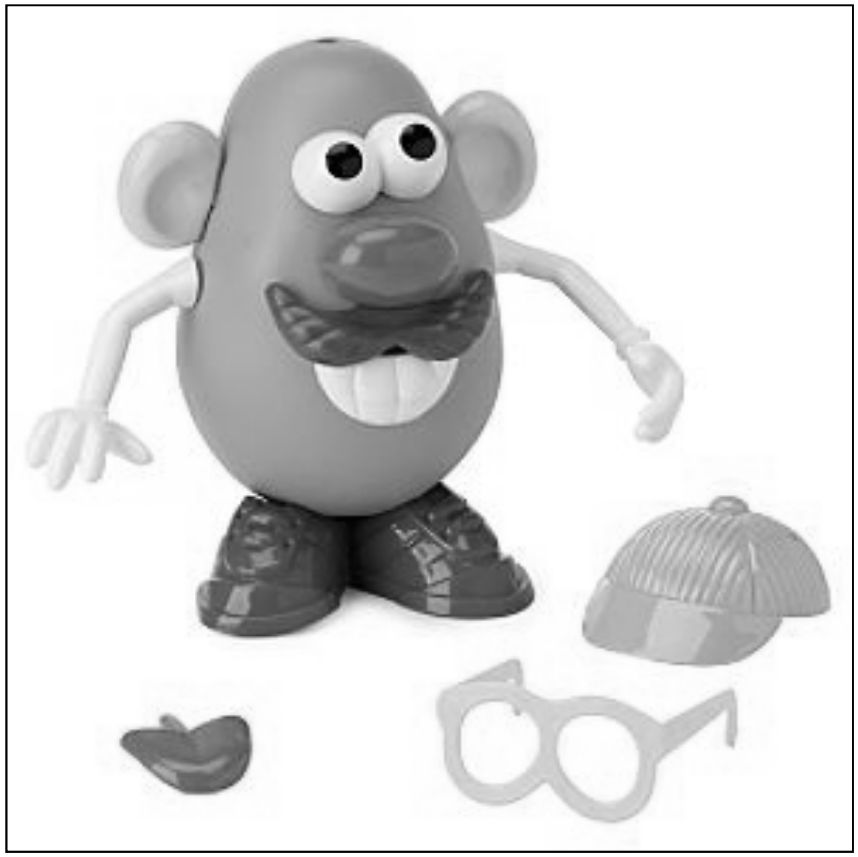

aBorderDecorator
component
aScrollDecorator
component
aTextView


VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

Consecuencias

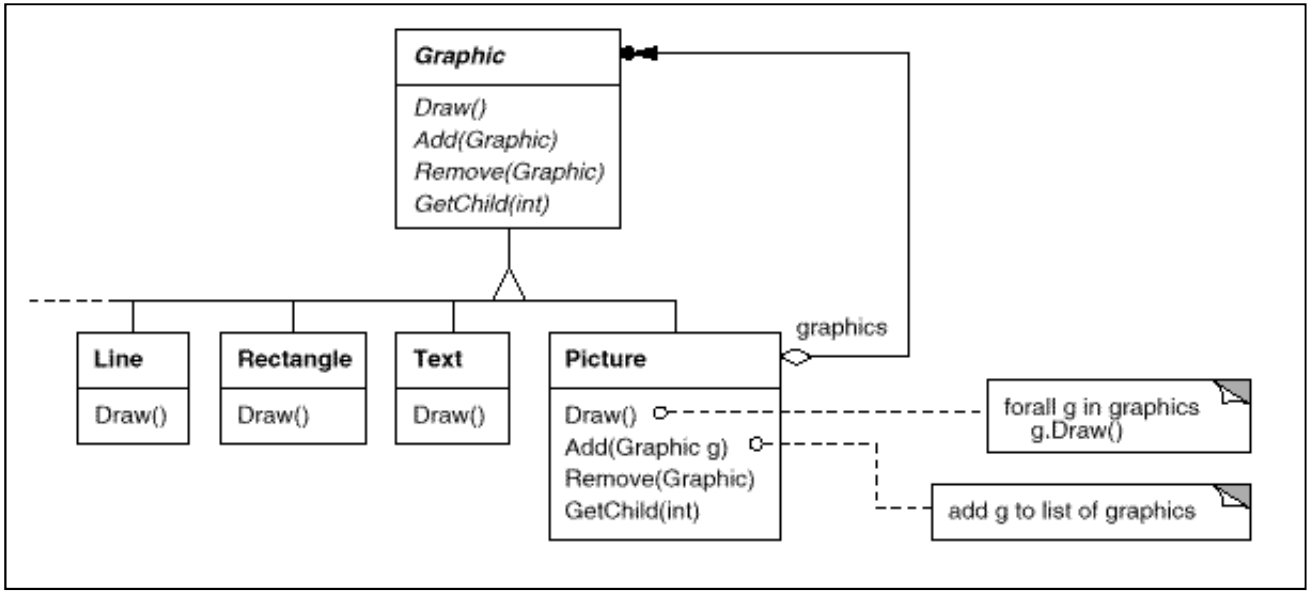
- Proporciona un mecanismo más flexible que la herencia para añadir responsabilidades
  - Se pueden añadir en tiempo de ejecución
  - Los decoradores no afectan a otros objetos coexistentes
- Evita sobrecargar las partes altas de la jerarquía de clases con funcionalidades a veces innecesarias
- ¡Cuidado al comparar objetos decorados!
  - Un individuo decorado no es idéntico al individuo en sí
- El uso extensivo de este patrón produce código más difícil de entender y depurar

Objeto compuesto



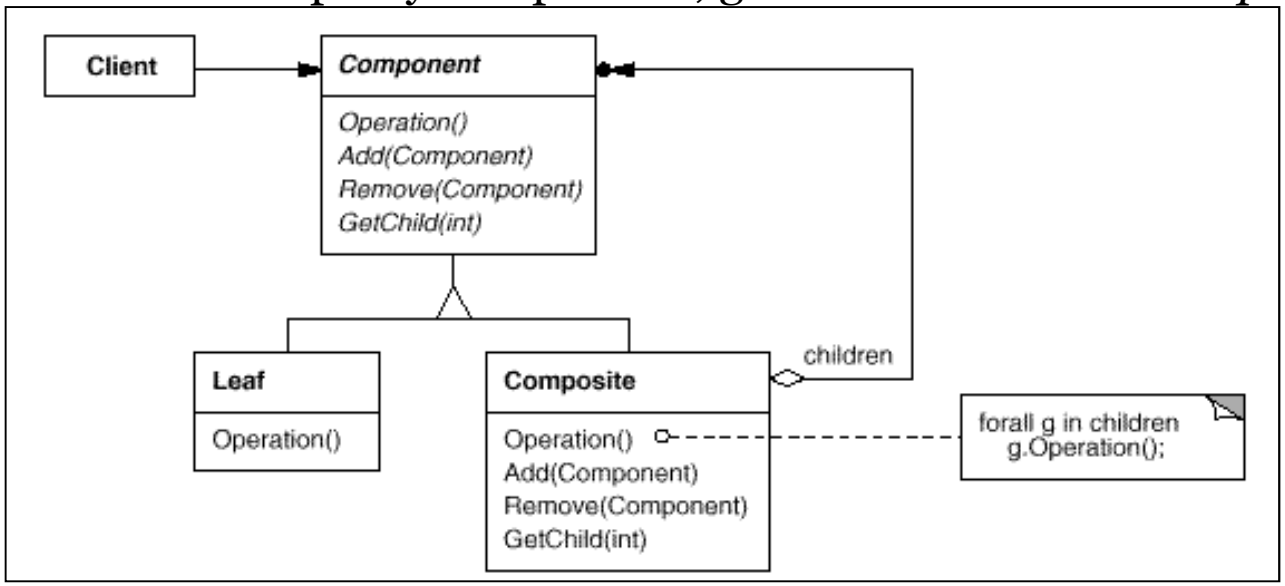
Objeto compuesto

- **Nombre original:** Composite
- **Propósito:** Permite representar estructuras *continente-contenido* donde los clientes pueden tratar de la misma forma a los individuos simples como a los compuestos
  - Ejemplo: ¿Cómo hacemos para que dentro de en un editor gráfico el código que manipula un rectángulo sea el mismo que el que trabaja sobre un montón de figuras geométricas agrupadas?



Solución

- *Component* define la interfaz común e implementa el comportamiento por defecto, añadiendo los métodos necesarios para acceder a los posibles hijos y al posible padre de un individuo
- *Leaf* representa a los individuos simples (hojas del árbol, sin hijos) y define el comportamiento de los objetos primitivos de esta composición
- *Composite* define el comportamiento de los individuos compuestos, almacena a sus hijos e implementa las operaciones de acceso a ellos
- El cliente manipula todos los objetos de la composición, sin hacer distinciones entre simples y compuestos, gracias a la interfaz *Component*



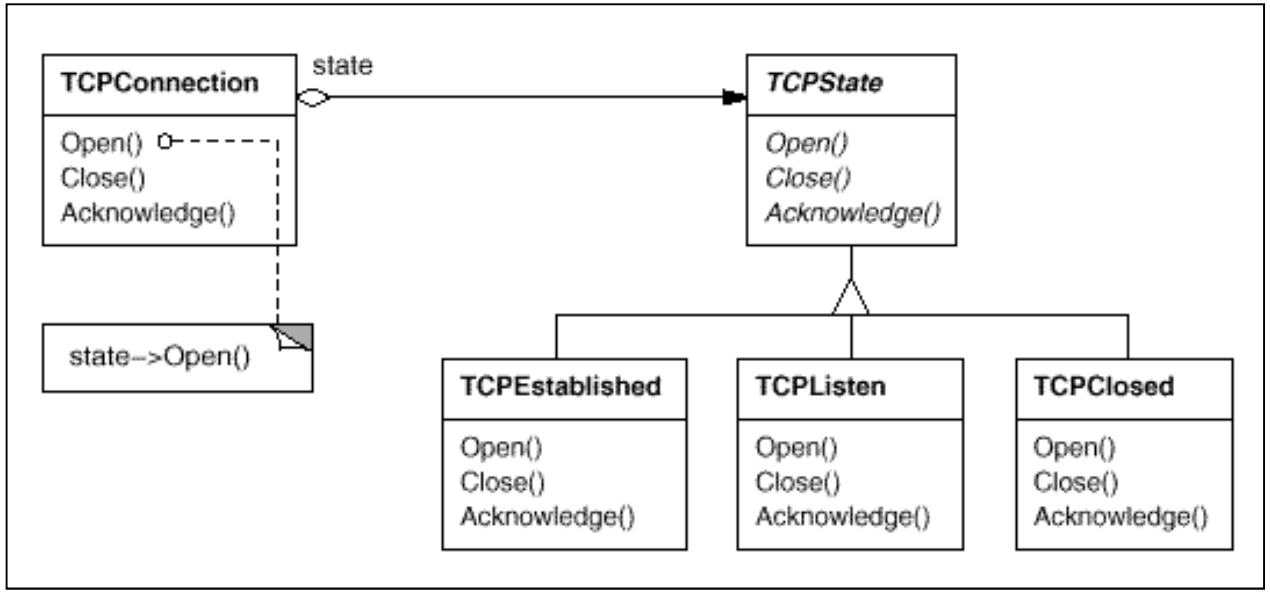
Consecuencias

- Permite tratar de manera homogénea a todos los objetos, sean simples o compuestos
- Simplifica el código del cliente, evitando distinciones de casos
- Facilita la definición de nuevos tipos de objetos (simples o compuestos) sin afectar al código del cliente
- Puede obligar a realizar un diseño demasiado generalista que dificulte, por ejemplo, la definición de compuestos donde se restrinje el tipo de componentes válidos

Estado

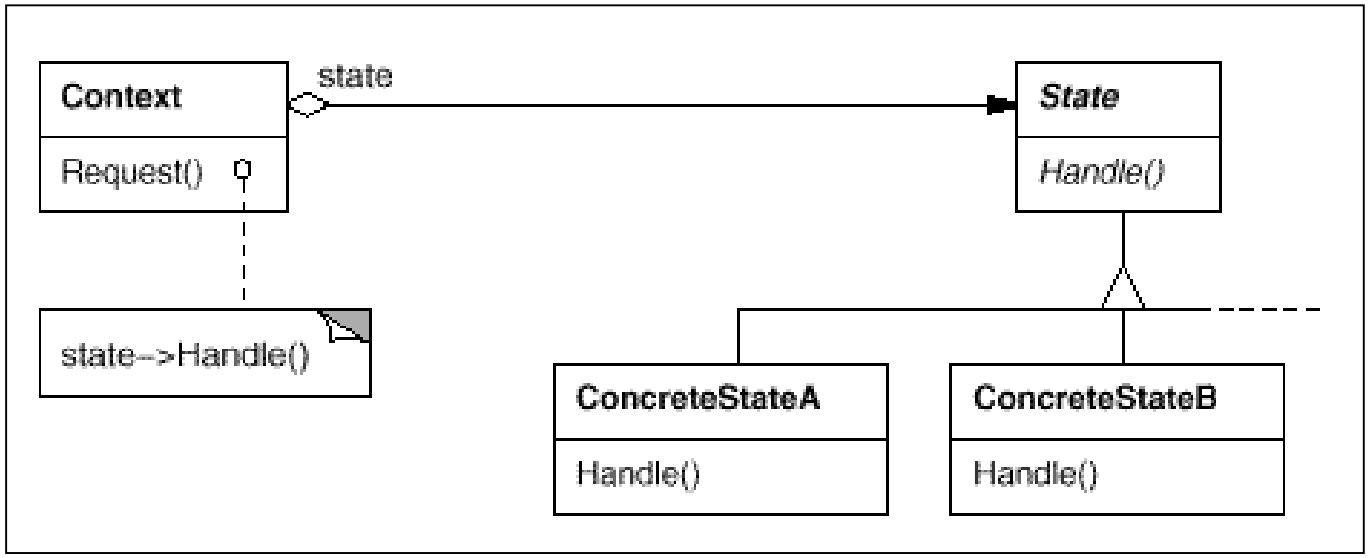
Estado

- **Nombre original:** State
- **Propósito:** Permite que un objeto se comporte de distinta forma (casi como si perteneciera a una clase distinta) dependiendo del estado en que se encuentre
  - Ejemplo: ¿Cómo representar una conexión por TCP mediante un objeto, si ha de comportarse de manera muy distinta según sea su estado?



Solución

- *Context* es la clase que ve el cliente y cuyos ejemplares tienen como atributo un objeto *State*, con el valor de estado adecuado para ese momento
- En *State* y sus especializaciones se encapsulan sólo aquellas operaciones dependientes del estado



Consecuencias

- Simplifica el código del contexto
- El flujo de control del código que depende del estado se hace patente en la jerarquía de clases de *State*
- Flexibiliza la definición de nuevos estados y también hace más explícitas las transiciones entre estados
  - Para cambiar de estado se destruye un objeto y se construye otro
  - Más seguro y consistente que codificar el estado en unas variables del contexto que luego hay que ir modificando convenientemente
- Su utilidad depende de cuántos estados diferentes hay y cuántas operaciones del contexto dependen del estado
  - Diseño menos compacto que no tendrá sentido con pocos estados

Críticas, dudas, sugerencias…

Federico Peinado
www.federicopeinado.es