Tecnología de Programación

Martín L. Larrea

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Patrones de diseño

Los patrones de diseño nombran, explican y evalúan
un diseño importante y recurrente
en los sistemas orientados a objetos.

Gang **O**f **F**our


Erich Gamma


Ralph Johnson


John Vlíssides


Richard Helm

Patrones GoF

Los siguientes son los patrones de diseño conocidos como GoF

		PROPÓSITO		
		CREACIONAL	ESTRUCTURAL	COMPORTAMIENTO
SCOPE	CLASE	Factory Method	Adapter	Interpreter Template Method
	OBJETO	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Patrones creacionales

Los *patrones creacionales* son patrones que abstraen el proceso de instanciación.

Procuran independizar el sistema de cómo sus objetos son creados, compuestos y representados.

Los patrones indican soluciones de diseño para
encapsular el conocimiento acerca de las clases que el sistema usa
y
ocultar cómo se crean instancias de estas clases.

Patrón Prototype

- Intención: **especifica los** *tipos* de objetos a crear utilizando una instancia prototipo, y crea nuevos objetos copiando esta instancia.

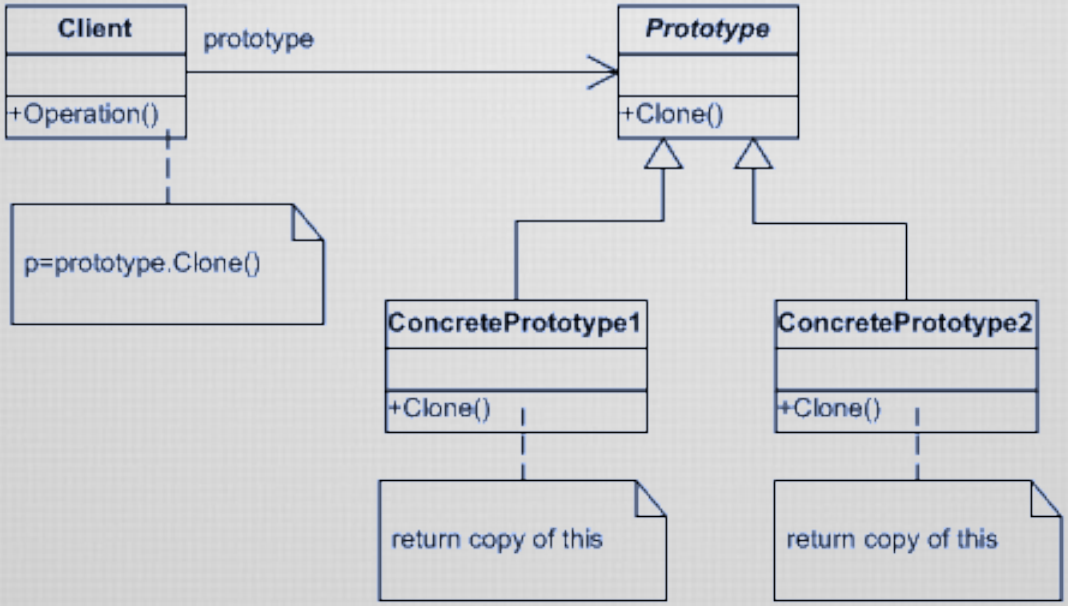
- Aplicabilidad: Usamos este patrón cuando
  - Cuando no es posible duplicar una instancia usando sólo sus métodos públicos.
  - Cuando instancias de una clase pueden tener sólo uno de muchas combinaciones de estados, lo que puede obtenerse clonando prototipos en lugar de hacerlo manualmente.

Este patrón delega el proceso de clonación al objeto que está siendo clonado.

Patrón Prototype

Pre-built prototypes can be an alternative to subclassing.

Patrón Prototype



Participantes

Prototype declara una interfaz para la autoclonación.

ConcretePrototype implementa una operación para clonarse a sí mismo.

Client crea un nuevo objeto solicitándole al prototipo que se clone a sí mismo.

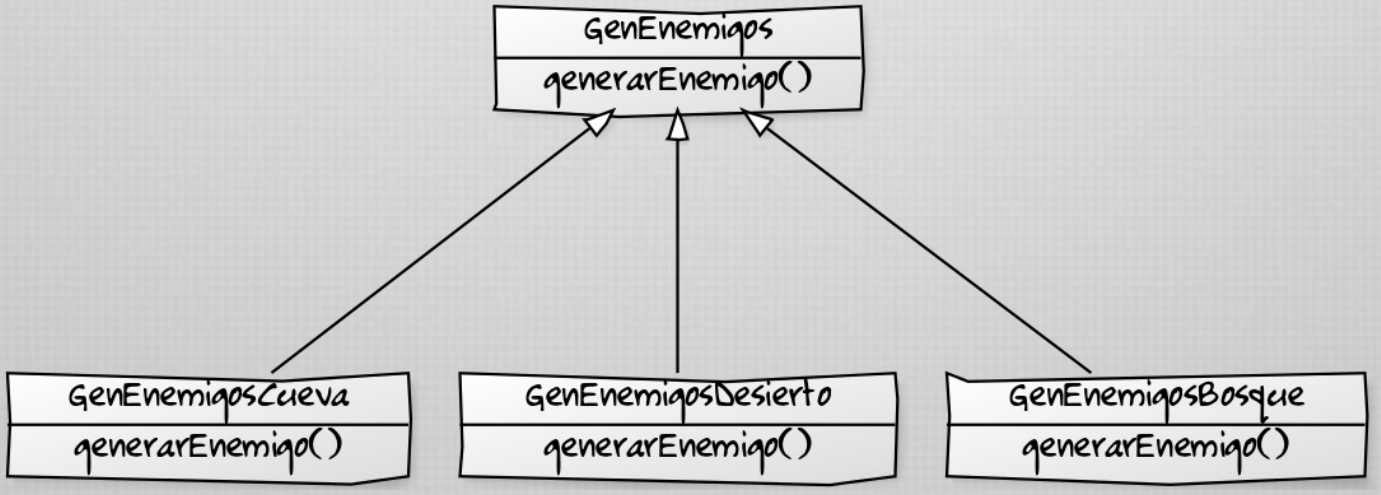
Patrón Prototype

Patrón Prototype

Problema

Generar enemigos para cada bioma, según sean
muy frecuentes, algo frecuentes o poco frecuentes

Alternativa de Solución

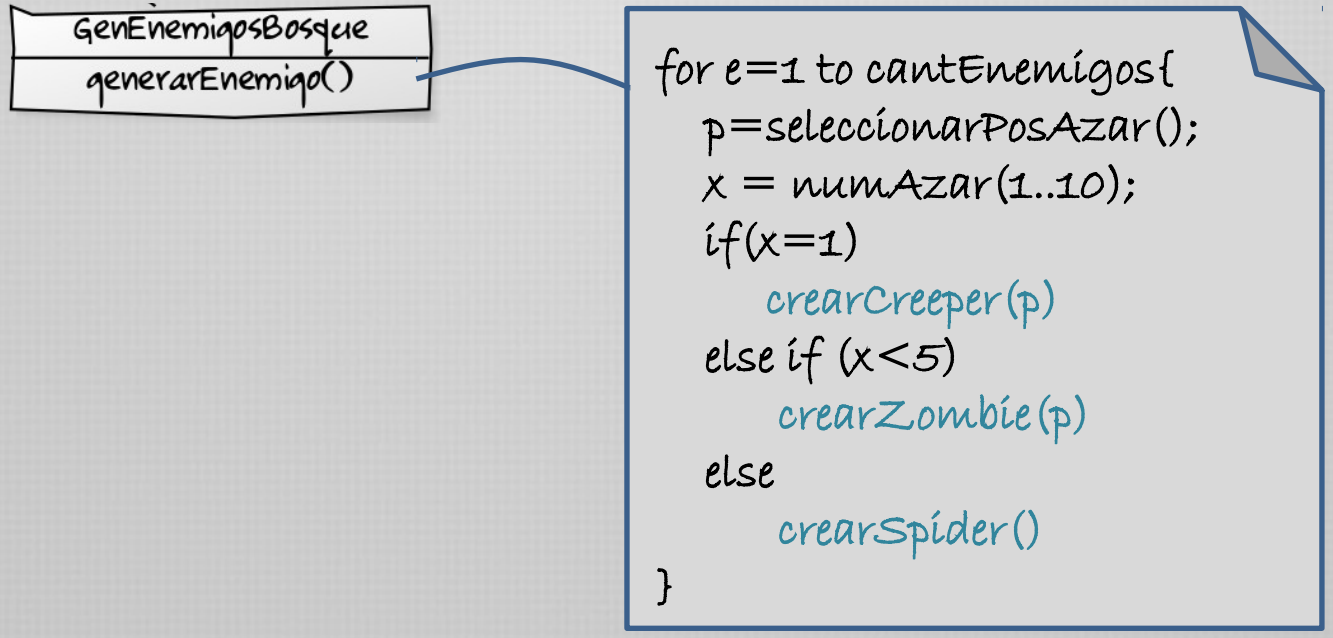


generarEnemigo() genera un enemigo en una posición al azar
Cada subclase genera mas de unos que de otros, según el bioma

Patrón Prototype

Problema

Generar enemigos para cada bioma, según sean muy frecuentes, algo frecuentes o poco frecuentes

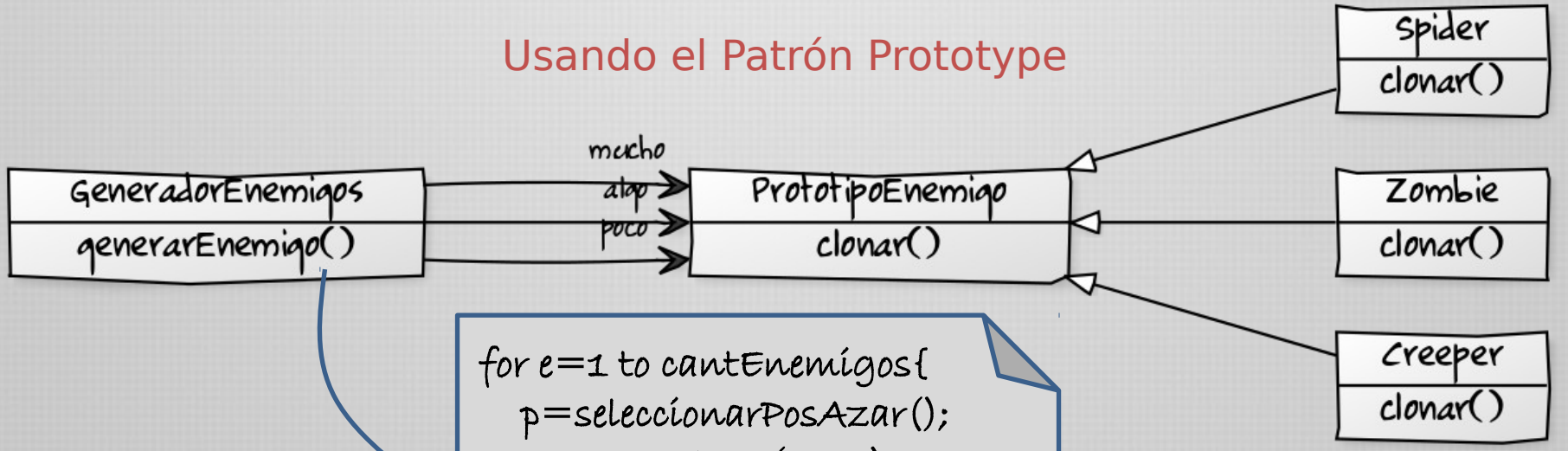


Patrón Prototype

Problema

Generar enemigos para cada bioma, según sean muy frecuentes, algo frecuentes o poco frecuentes

Usando el Patrón Prototype



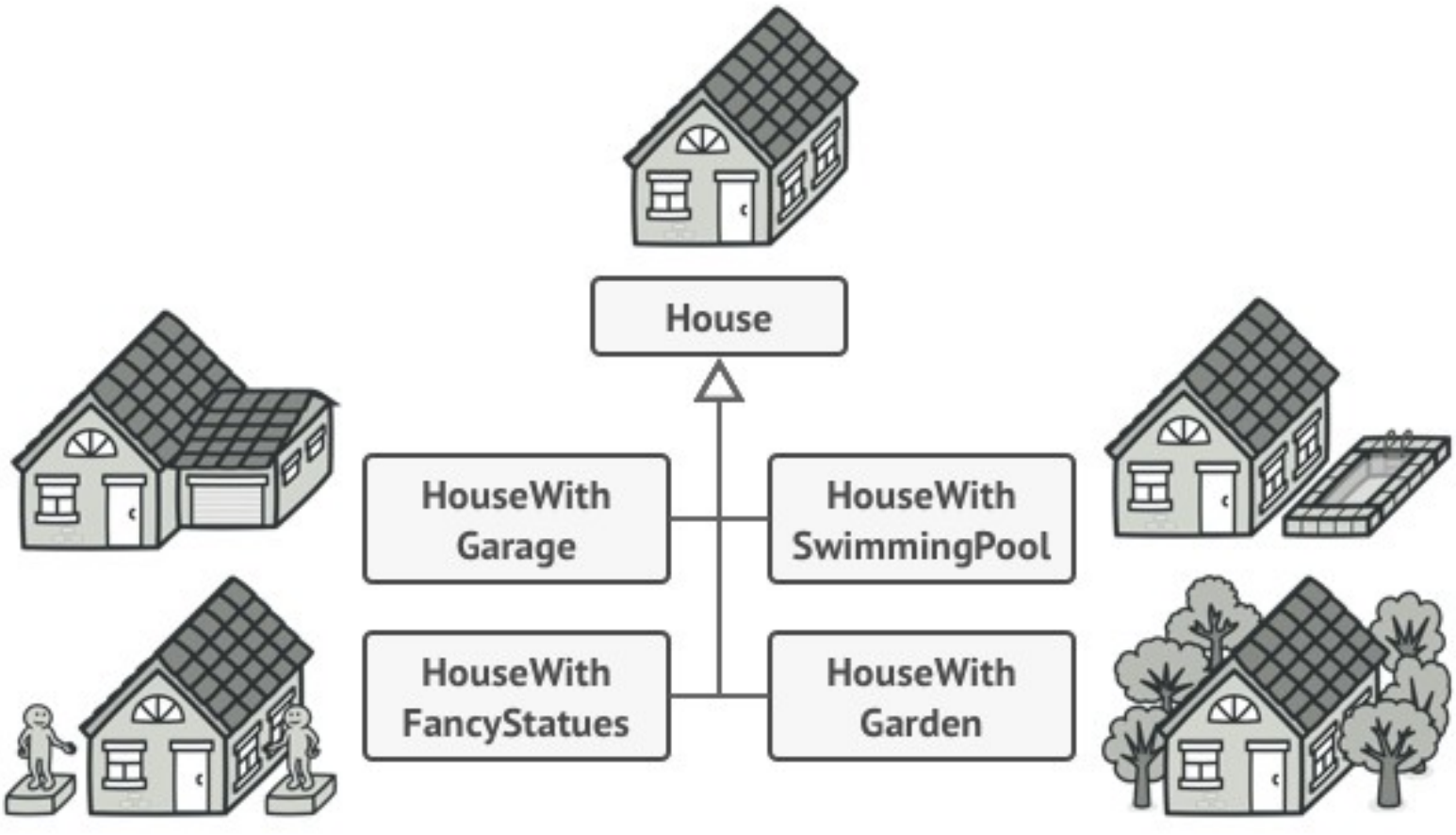

```
for e=1 to cantEnemigos{
   p=seleccionarPosAzar();
   x = numAzar(1..10);
   if(x=1)
       en=poco.clonar()
   else if (x<5)
       en=algo.clonar()
   else
       en=mucho.clonar()
   ubicar(en,p)
}
```

Patrón Builder

- Intención: simplificar la construcción de objetos complejos. Permite crear diferentes tipos de objetos usando los mismos pasos de construcción.

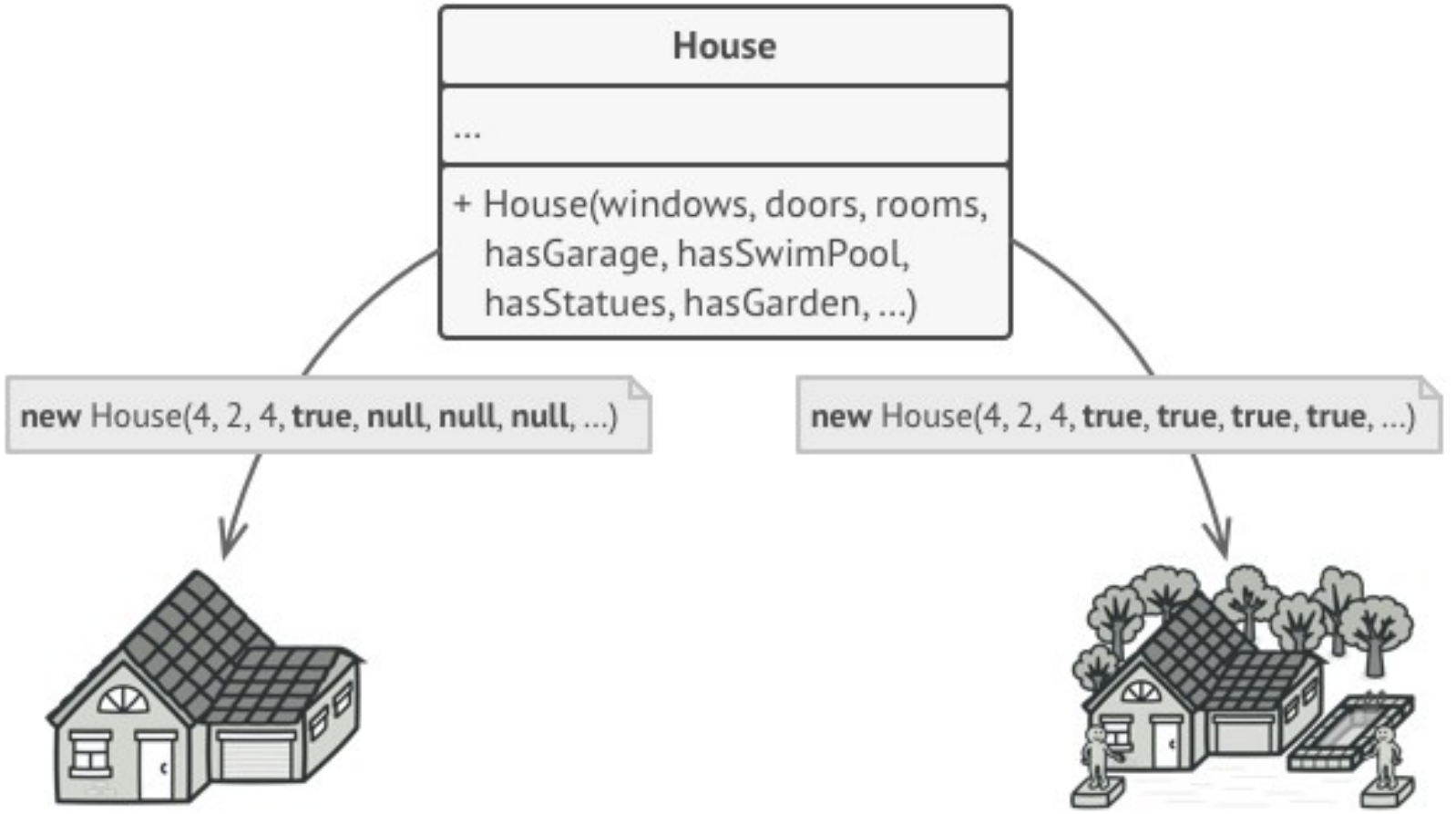
- Aplicabilidad: Usamos este patrón cuando
  - La construcción de un objeto requiere muchos parámetros que no siempre son instanciados.
  - La construcción de un objeto se hace mediante métodos en la clase del objeto que se quiere construir.

Patrón Builder



You might make the program too complex by creating a subclass for every possible configuration of an object.

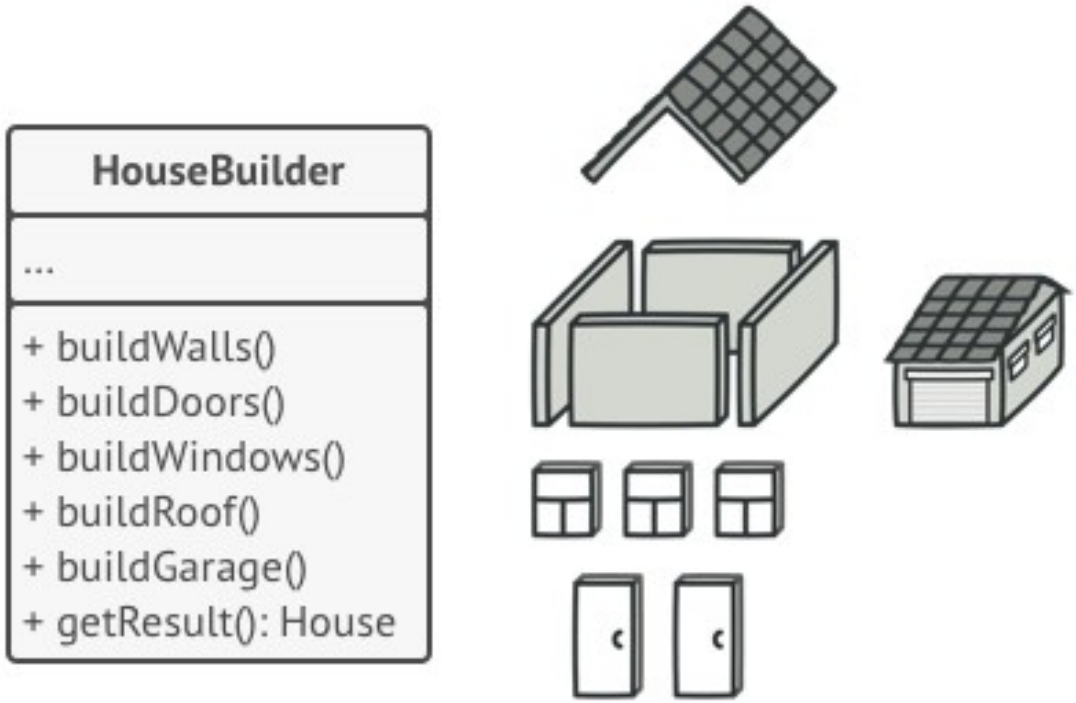
Patrón Builder



The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

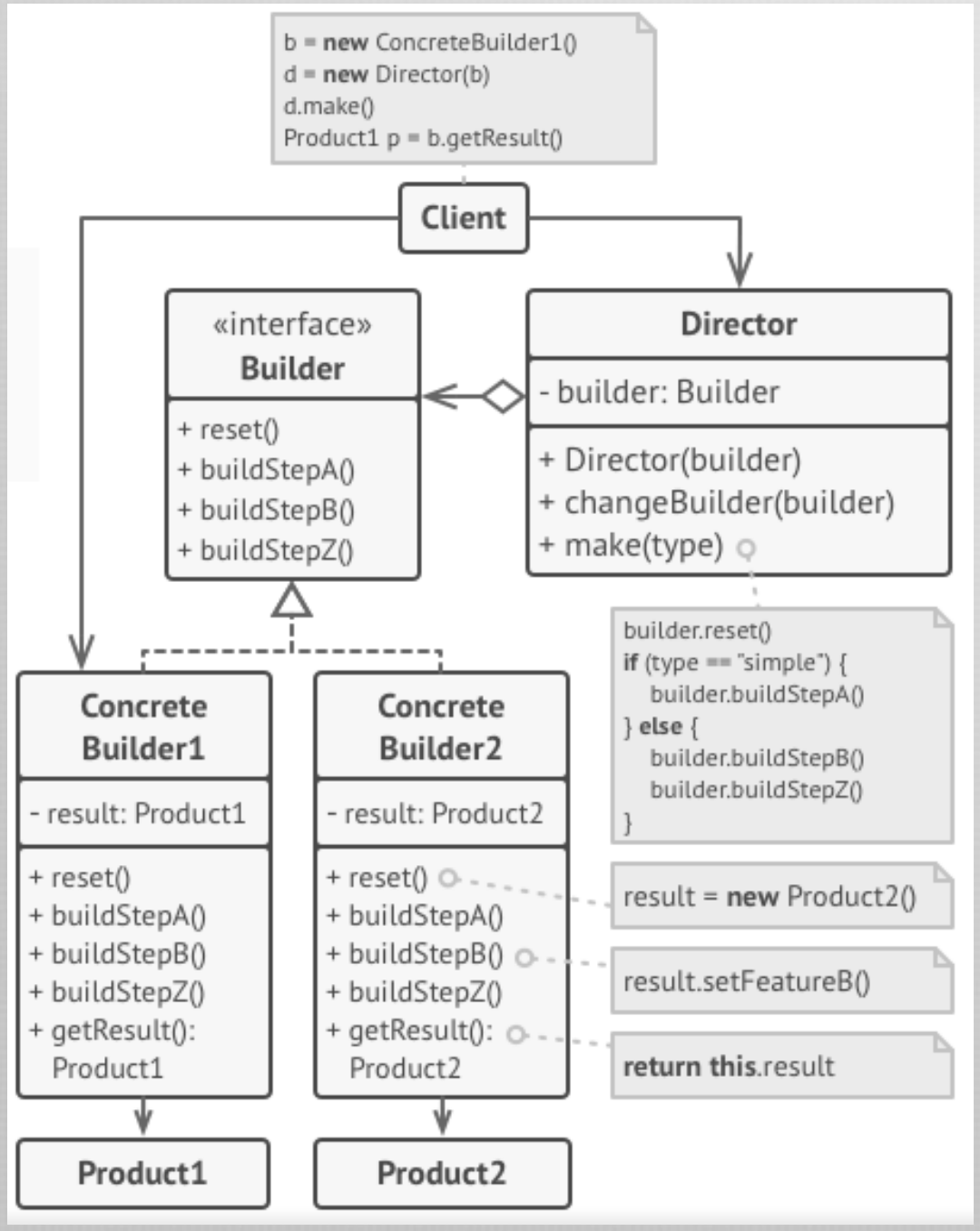
Patrón Builder

La propuesta de este patrón es sacar el código asociado a la construcción de la clase, de la clase y resolverlo en otra clase llamada *Builder.*



The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

Patrón Builder

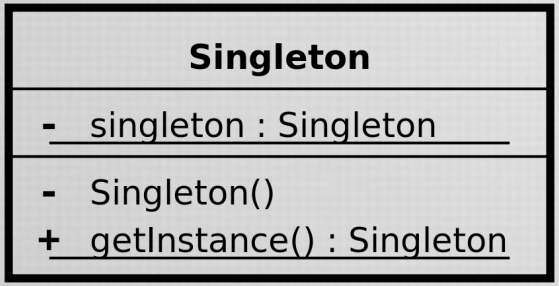


Patrón Singleton

- Intención: asegura que una clase tenga una única instancia a lo largo del sistema y ofrece un acceso global a tal instancia

- Aplicabilidad: Usamos este patrón cuando
  - Cuando queremos controlar el acceso a un recurso compartido.
  - Cuando no queremos tener dos instancias de diferentes tipos en un mismo sistema, por ejemplo dos *abstract factories*.

Patrón Singleton




```java
public final class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Patrones Estructurales

Structural Patterns

Patrones estructurales

Los patrones estructurales se refieren a cómo
las clases y los objetos son
organizados para conformar estructuras más grandes

Pueden ser
patrones de clases, basados en la utilización de herencia, o
patrones de objetos, basados en la técnica de composición.

La mayor variedad se encuentra en los patrones estructurales de objetos, en donde los objetos se componen para obtener nuevas funcionalidades.

Muchos de ellos están relacionados en cierta forma entre sí.

Recordemos nuevamente la importancia del concepto de interfaz.
Los objetos interactúan entre ellos por medio de sus interfaces.

Patrón Adapter

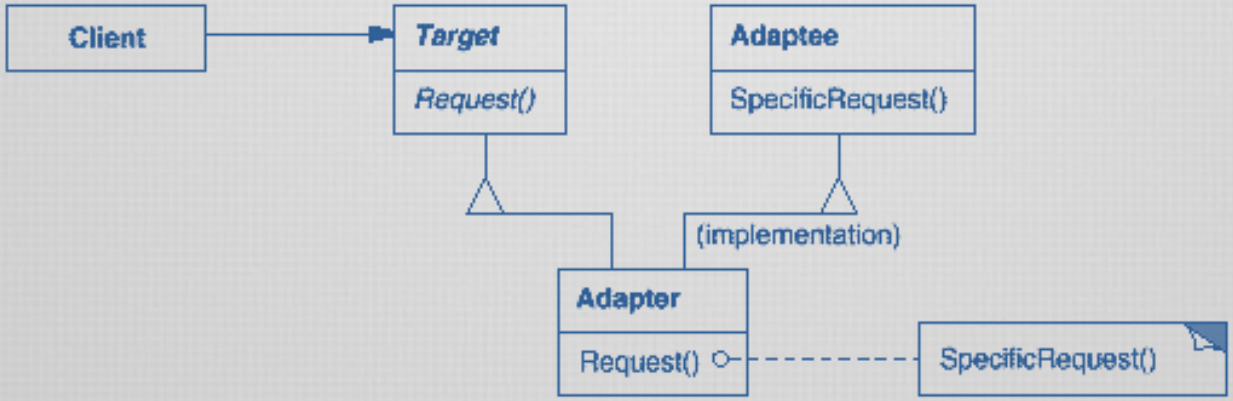
- Intención: Convertir la interfaz de una clase en otra interfaz que el cliente espera.
- Alias: *Wrapper*.
- Aplicabilidad: Usaremos este patrón cuando:
  - Deseamos usar una clase existente, pero su interfaz no es la que necesitamos.
  - Queremos crear una clase reutilizable que coopera con otras clases no relacionadas, probablemente con interfaces incompatibles.
  - Necesitamos usar varias subclases, pero no es práctico adaptar sus interfaces heredando de ellas, por lo que apelamos a un objeto *adaptador*. En este caso se utiliza la versión *Object Adapter*

Básicamente, es un encapsulado de los métodos y una traducción directa.

Lo usaremos cuando querramos adaptar una clase con interfaz diferente, a la interfaz que necesitamos.

Patrón Adapter

Estructura: *(versión Class-Adapter)*



Participantes:

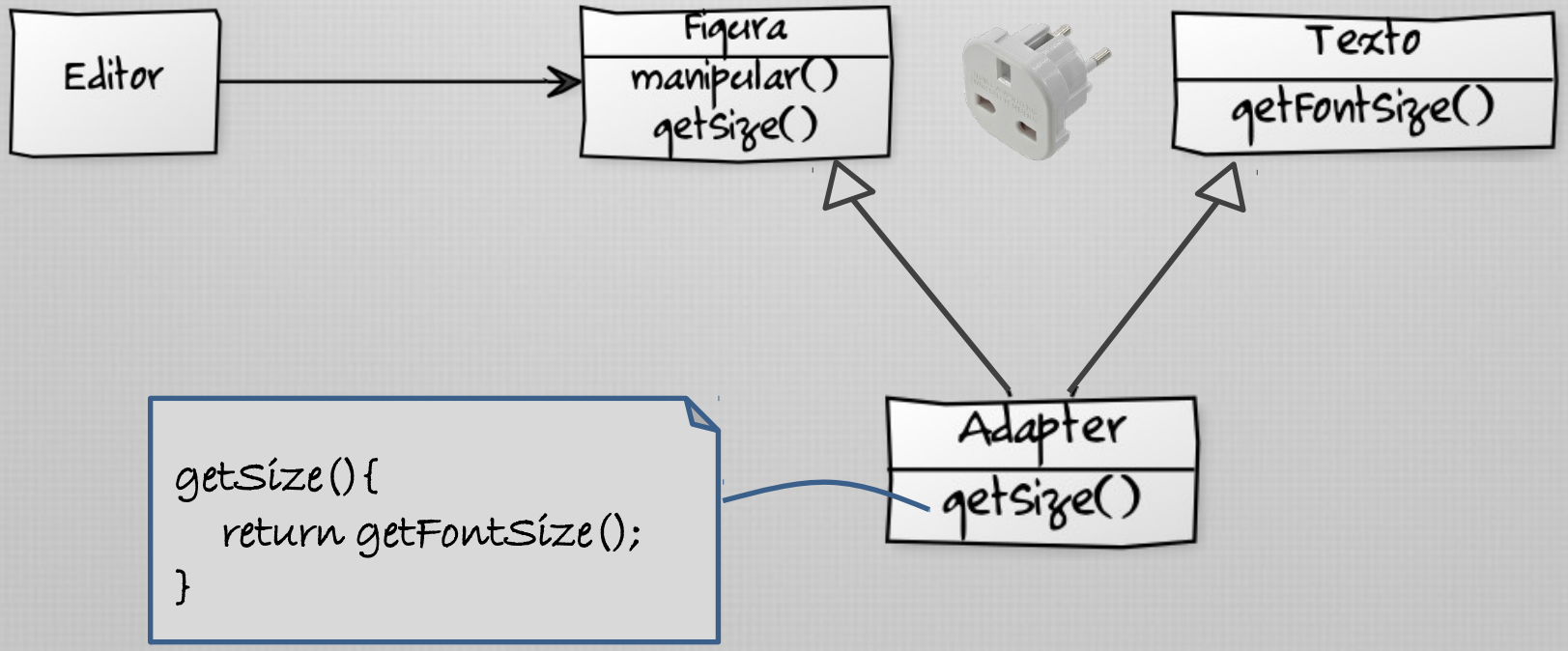
Target: define la interfaz que el cliente usa.

Adaptee: interfaz existente que necesita adaptarse.

Adapter: adapta la interfaz de *Adaptee* a la que necesita el cliente.

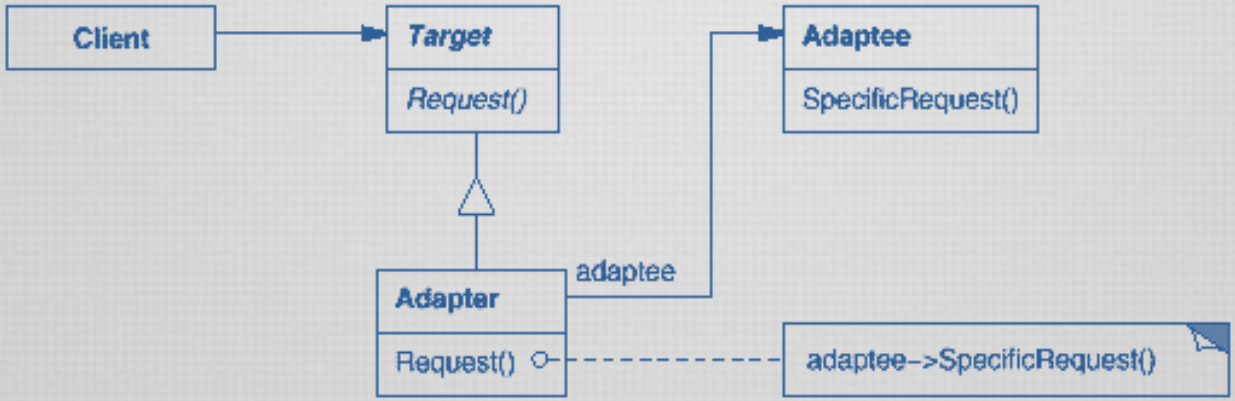
Patrón Adapter - ejemplo

UNS UNS UNS



Patrón Adapter

Estructura: *(versión Object-Adapter)*



Participantes:

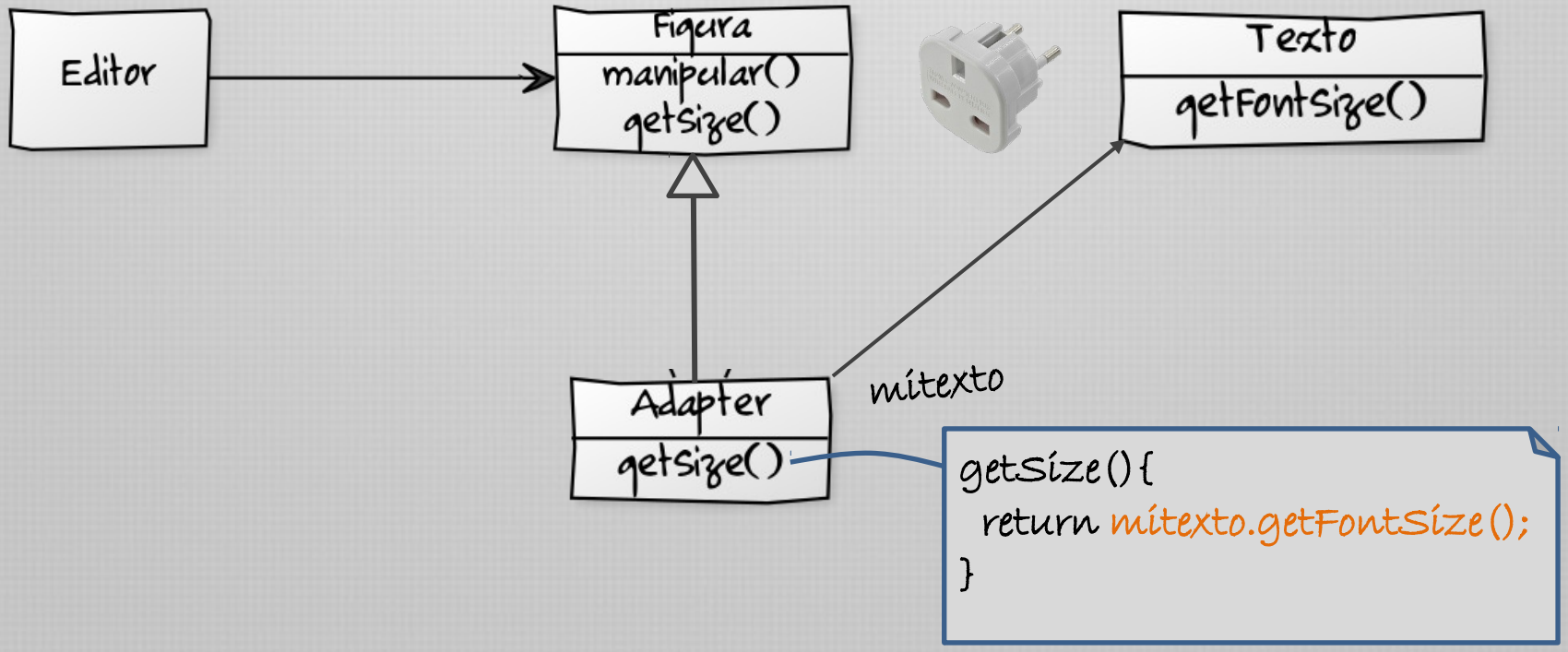
Target: define la interfaz que el cliente usa.

Adaptee: interfaz existente que necesita adaptarse.

Adapter: adapta la interfaz de Adaptee a la que necesita el cliente.

Patrón Adapter - ejemplo

UNS UNS UNS



Patrón Composite

Algunas aplicaciones necesitan manipular a la vez
- objetos y
- agrupaciones de objetos

de manera indistinta.

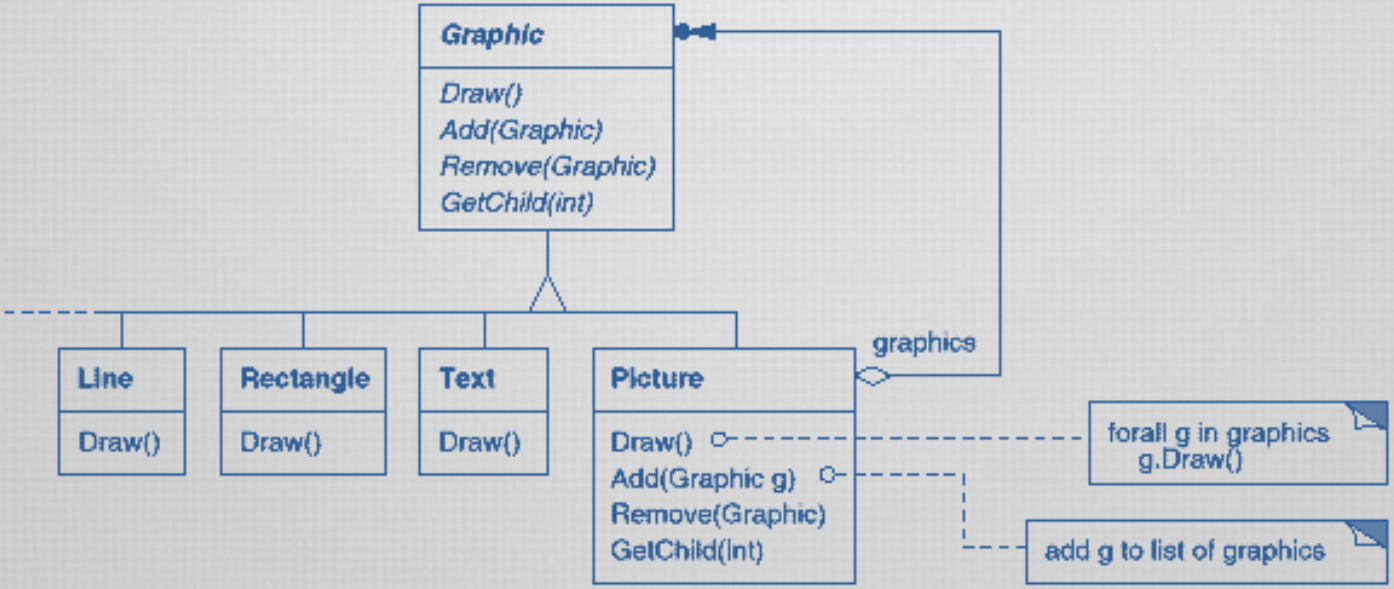
Por ejemplo, una aplicación gráfica necesita manipular líneas, círculos y rectángulos tanto como figuras compuestas por estos elementos, o por otras figuras compuestas.

La aplicación necesita distinguir estos objetos,
lo cual complica un poco el escenario general.

La clave en este caso es abstraerse de los dos elementos, y trabajar con una clase abstracta que represente tanto los objetos primitivos como los objetos compuestos.

Patrón Composite - ejemplo

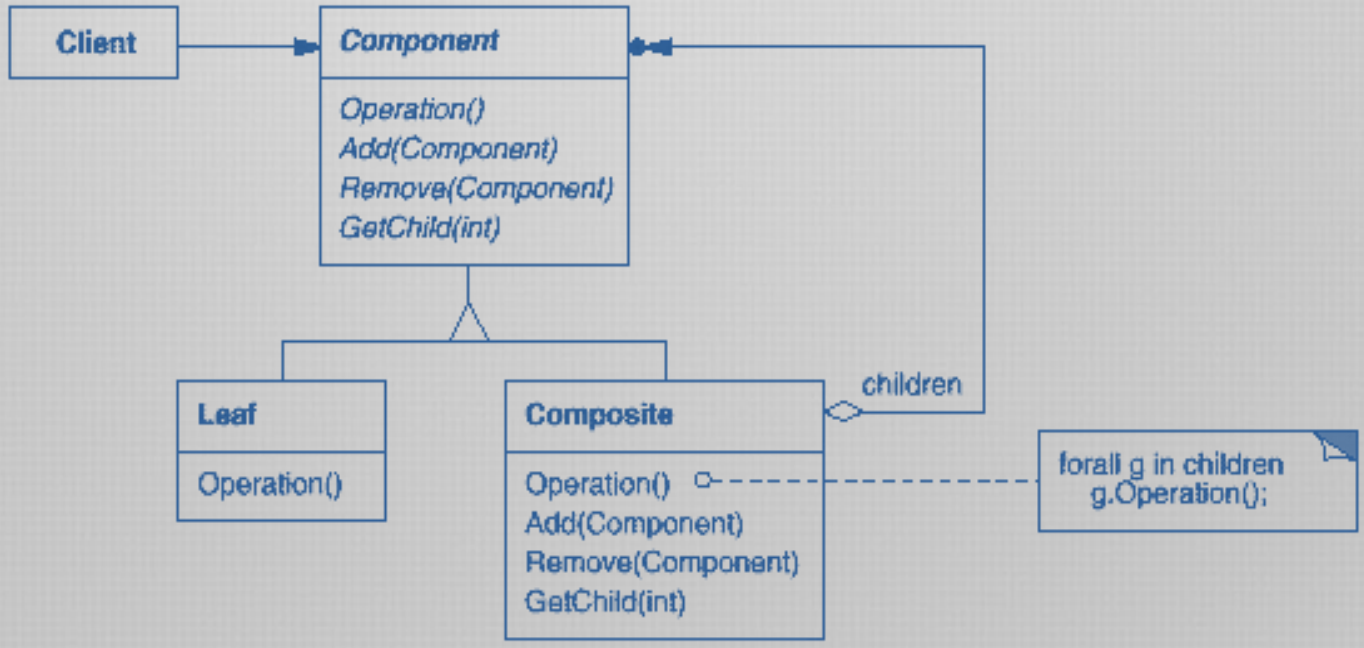
En el caso de la aplicación gráfica, la organización de clases puede ser:



Esta *composición recursiva* de objetos es la propuesta del patrón de diseño Composite.

Patrón Composite

- Intención: Compone objetos en estructuras de árbol para representar jerarquías de relación "*es-parte-de*".
- Aplicabilidad: Usaremos este patrón cuando:
  - Queremos representar *jerarquías de objetos modelando la relación "es-parte-de"* (*part-whole hierarchies*)
  - Queremos que el cliente *ignore la distinción entre objetos compuestos y objetos individuales*. El cliente tratará todos los objetos de la estructura compuesta de manera uniforme.
- Estructura:



Patrón Decorator

A veces es deseable agregar responsabilidades a objetos individuales,
y no a toda la clase a la que pertenece.

Un conjunto de herramientas para desarrollar interfaces gráficas debería permitir agregar bordes o comportamientos como el scrolling a alguna componente gráfica individual.

Una conexión particular a un servidor puede encriptar la información antes de enviarla.

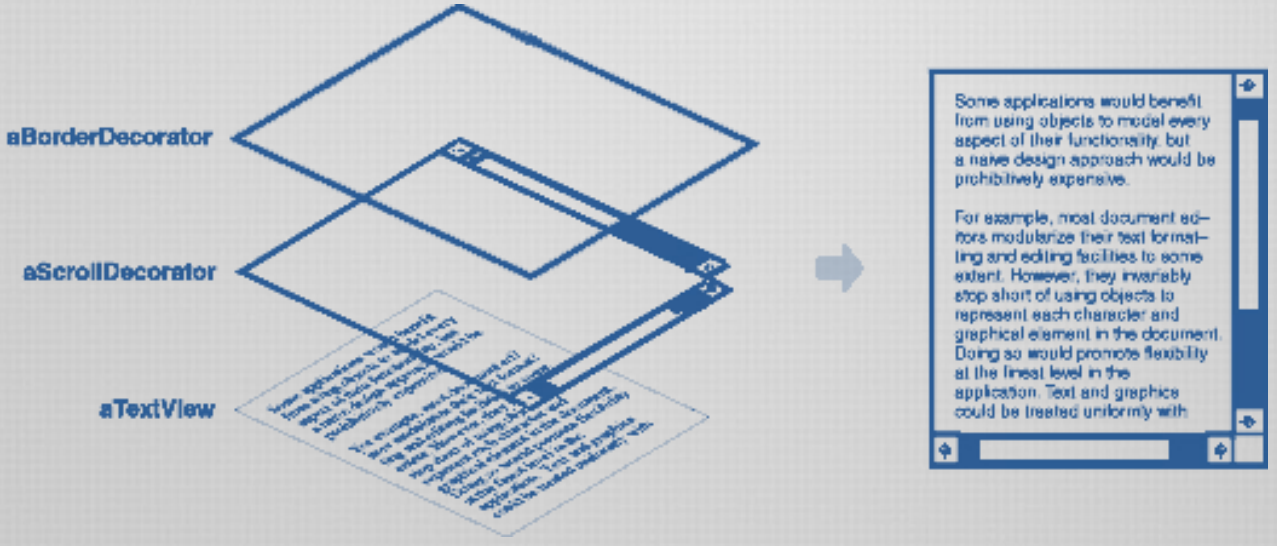
Una forma de agregar responsabilidades es por medio de la herencia.
Heredar un borde significa agregarlo a todas las subclases.

Pero la elección del borde se resuelve en forma estática.
El cliente no controla cómo y cuándo decorar el componente con un borde.

Una aproximación más flexible es “encerrar” el componente a decorar en otro objeto. Este último se denomina el objeto decorador.

Patrón Decorator

La idea es, por ejemplo, poder decorar dinámicamente un objeto de tipo *TextView* con otros objetos representando ciertas propiedades o comportamientos.

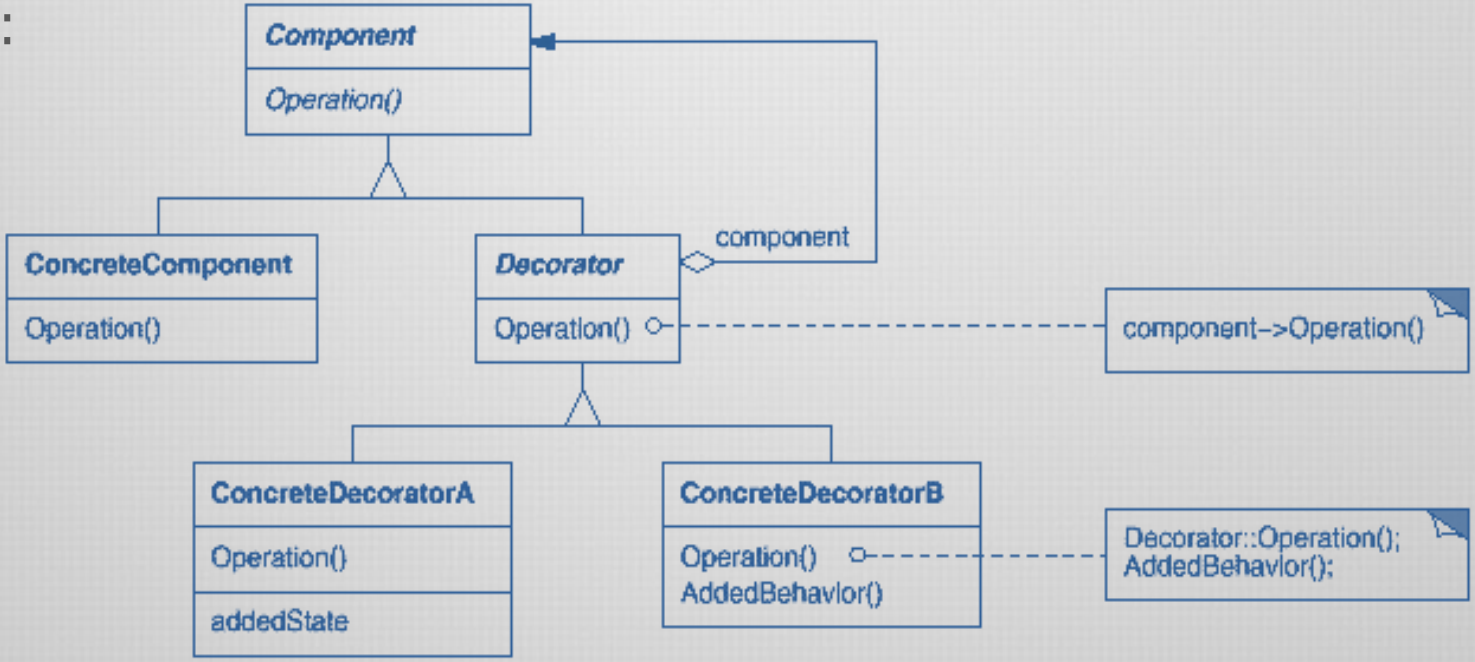


Patrón Decorator

- Intención: Agrega responsabilidades a un objeto de manera dinámica. Provee una alternativa a la herencia cuando deseamos agregar funcionalidad.
- Alias: *Wrapper*
- Aplicabilidad: Usaremos este patrón cuando:
  - Queremos agregar responsabilidades a objetos individuales de manera dinámica y transparente, sin afectar otros objetos.
  - Deseamos implementar responsabilidades que pueden removerse.
  - Cuando *la extensión utilizando herencia es impracticable*, por ejemplo, pues provocaría demasiadas variaciones y un gran número de clases.

Patrón Decorator

Estructura:



Participantes:

- Component: define la interfaz para los objetos que pueden tener responsabilidades agregadas dinámicamente.
- ConcreteComponent: define un objeto al cual se le pueden agregar responsabilidades dinámicamente.
- Decorator: mantiene una referencia a un objeto Component y define la interfaz que conforma la interfaz de Component.
- ConcreteDecorator: agrega responsabilidades al componente.

Patrón Decorator - ejemplo

```
class VisualComponent {
    public:
        VisualComponent();
        virtual void Draw();
        virtual void Resize();
        // ...
};
```

```
class Decorator : public VisualComponent
{
    public:
        Decorator(VisualComponent*);
        virtual void Draw();
        virtual void Resize();
        // ...
    private:
        VisualComponent* _component;
};
```

```
Draw() {
  _component->Draw()
}
```

Decorador Concreto

```
class BorderDecorator : public Decorator {
 public:
   BorderDecorator(VisualComponent* int borderWidth);
   virtual void Draw();
 private:
   void DrawBorder(int);
 private:
   int _width;
};
```

```
{
Super.draw();
DrawBorder(_width);
}
```

Patrón Decorator - ejemplo

Supongamos que tenemos esta operación en *Window*, para agregar componentes visuales

```
void Window::SetContents (VisualComponent* contents) {
    // ...
}
```

Creamos un objeto *TextView* y una ventana en la cual ubicarlo.

```
Window* window = new Window;
TextView* textView = new TextView;
window->SetContents(textView);
```

Pero como queremos un *TextView* con borde y con barra de desplazamiento (scroll bar), lo decoramos acordemente antes de ubicarlo en la ventana.

```
window->SetContents(
    new BorderDecorator( new ScrollDecorator(textView,1) );
```

Como *Window* accede a sus componentes por medio de la interfaz *VisualComponent*, no tiene *conciencia* de la decoración del objeto TextView.

Patrones de Comportamiento

Behavioral Patterns

Patrones de comportamiento

Los patrones de comportamiento se centran en los algoritmos y la asignación de responsabilidades entre los objetos.

Son patrones tanto de clases y objetos (similares a los anteriores) como de comunicación entre ellos.
Caracterizan flujo de control complejo.

Los patrones de comportamiento de clases *(behavioral class patterns)* utilizan herencia para distribuir el comportamiento entre las clases.

Los patrones de comportamiento de objetos *(behavioral object patterns)* utilizan composición de objetos en lugar de herencia.

Algunos describen cómo los objetos cooperan entre sí para realizar una tarea compleja, imposible para sólo uno de ellos.

Patrón Command

A veces es necesario solicitar servicios que desconocemos sobre objetos que ignoramos.

Por ejemplo, por medio de herramientas (toolkits) para implementar menús de usuario, con botones, listas, etc.

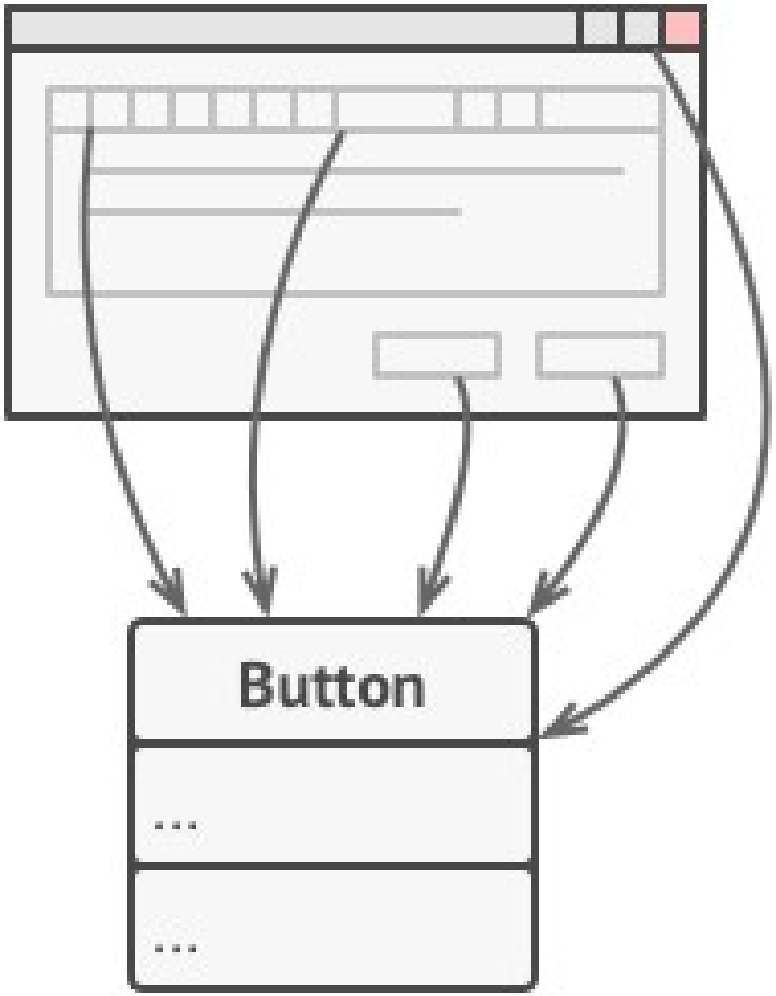
Obviamente, una opción del menú no implementa explícitamente la tarea. Sólo la aplicación que usa el toolkit sabe qué debe hacerse en qué objeto.

La idea es que el pedido mismo sea un objeto.

Podemos declarar una clase *Command* que declara una interfaz para ejecutar operaciones.

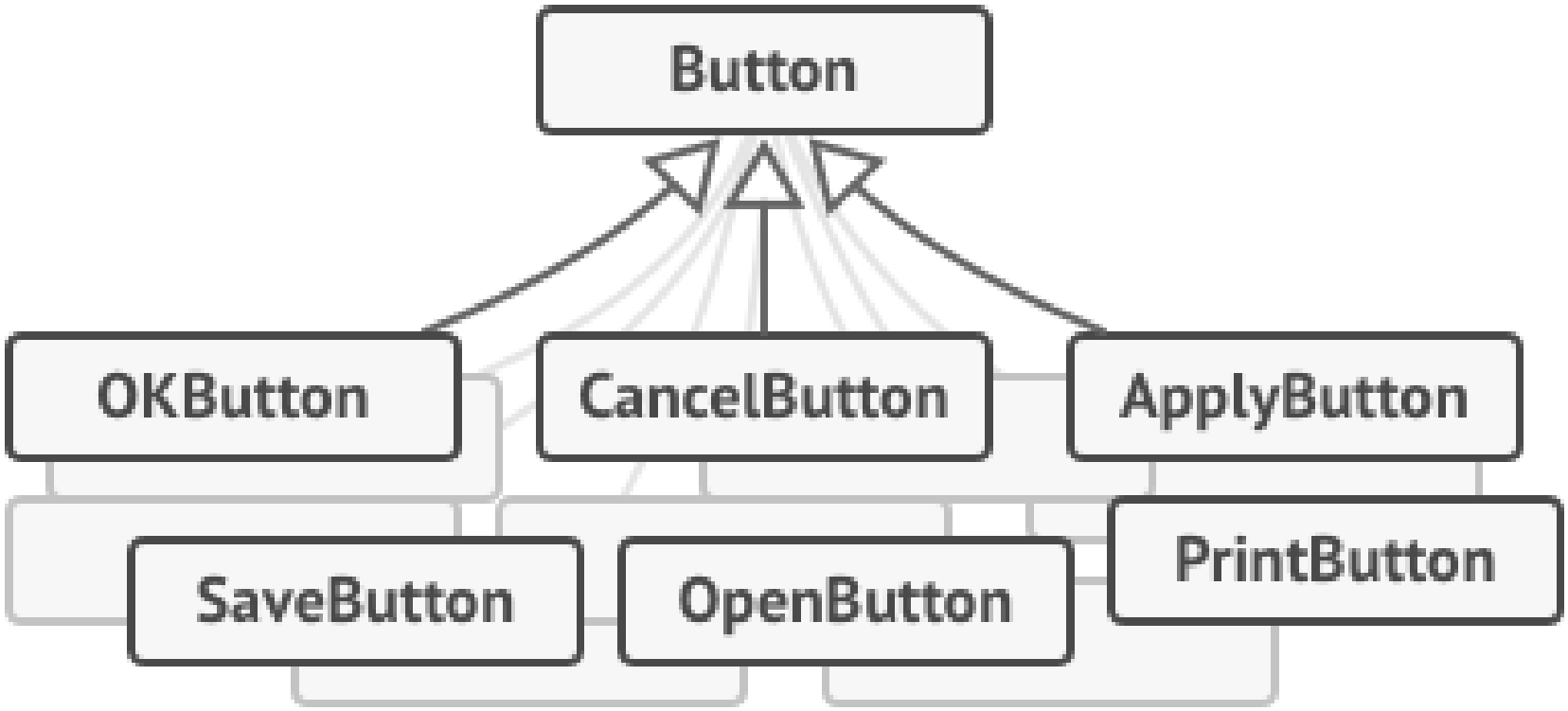
Las clases *Command* concretas (descendientes) especifican el receptor de la operación, vinculando el requerimiento con el objeto en cuestión.

Patrón Command



All buttons of the app are derived from the same class.

Patrón Command

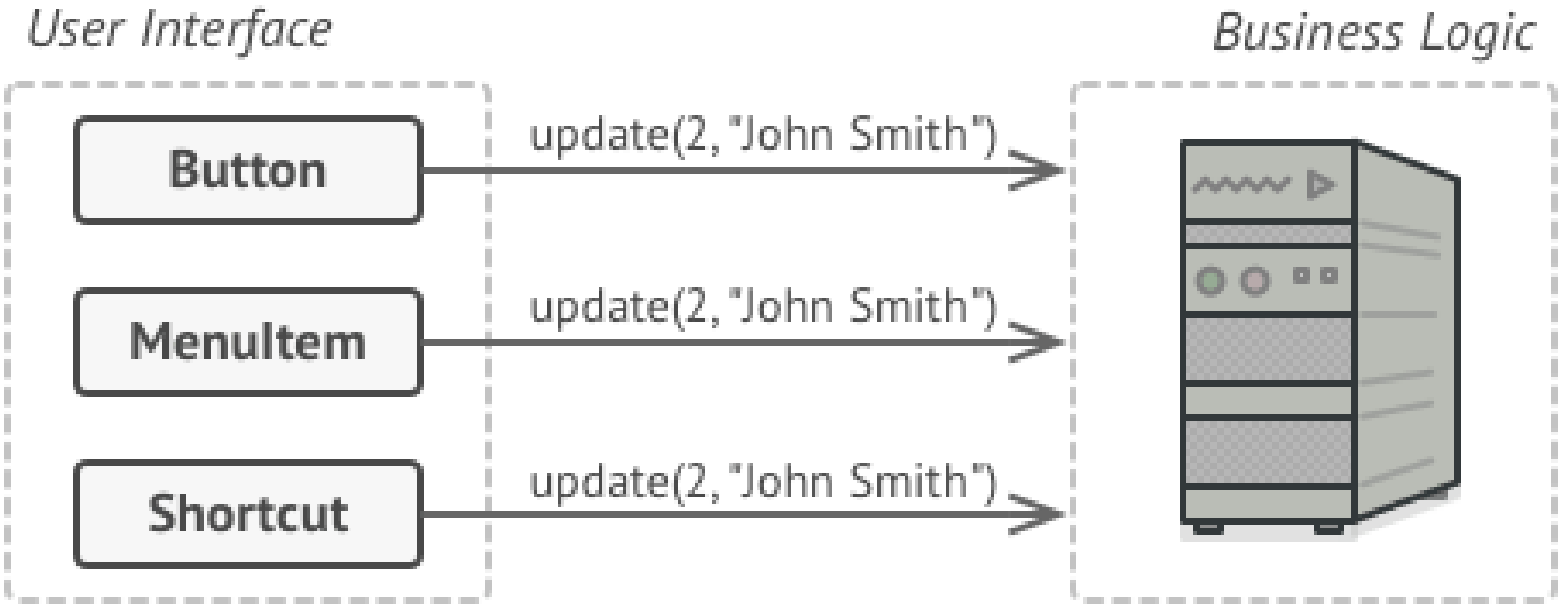


Lots of button subclasses. What can go wrong?

Patrón Command

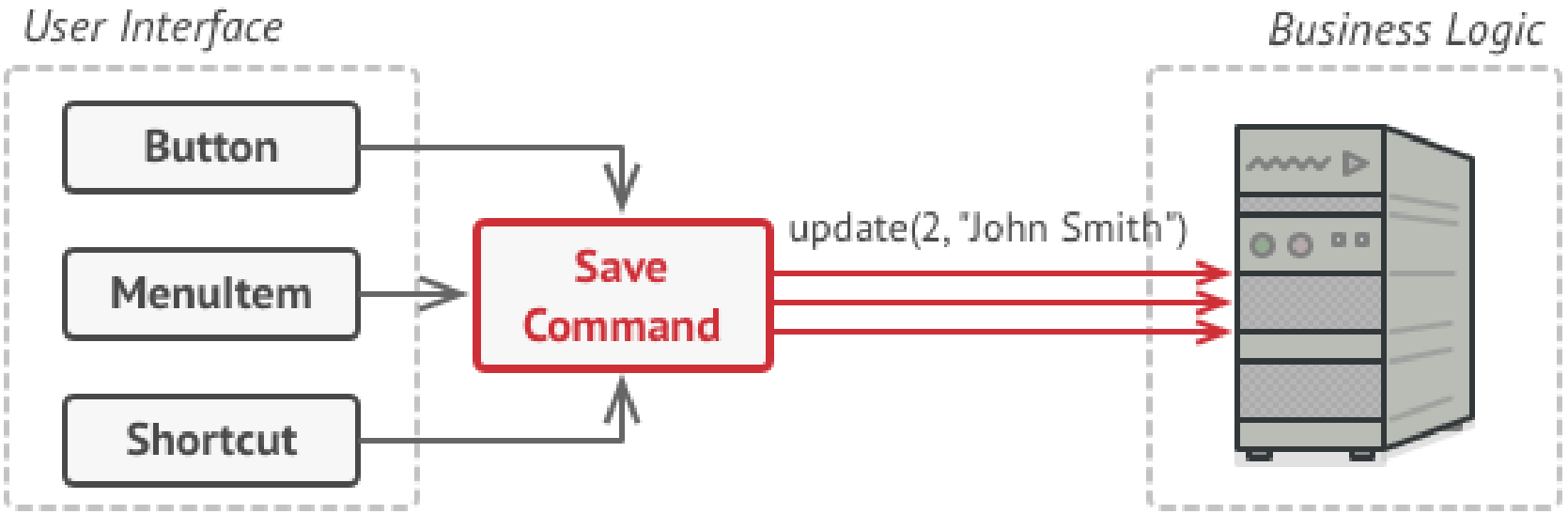
Several classes implement the same functionality.

Patrón Command



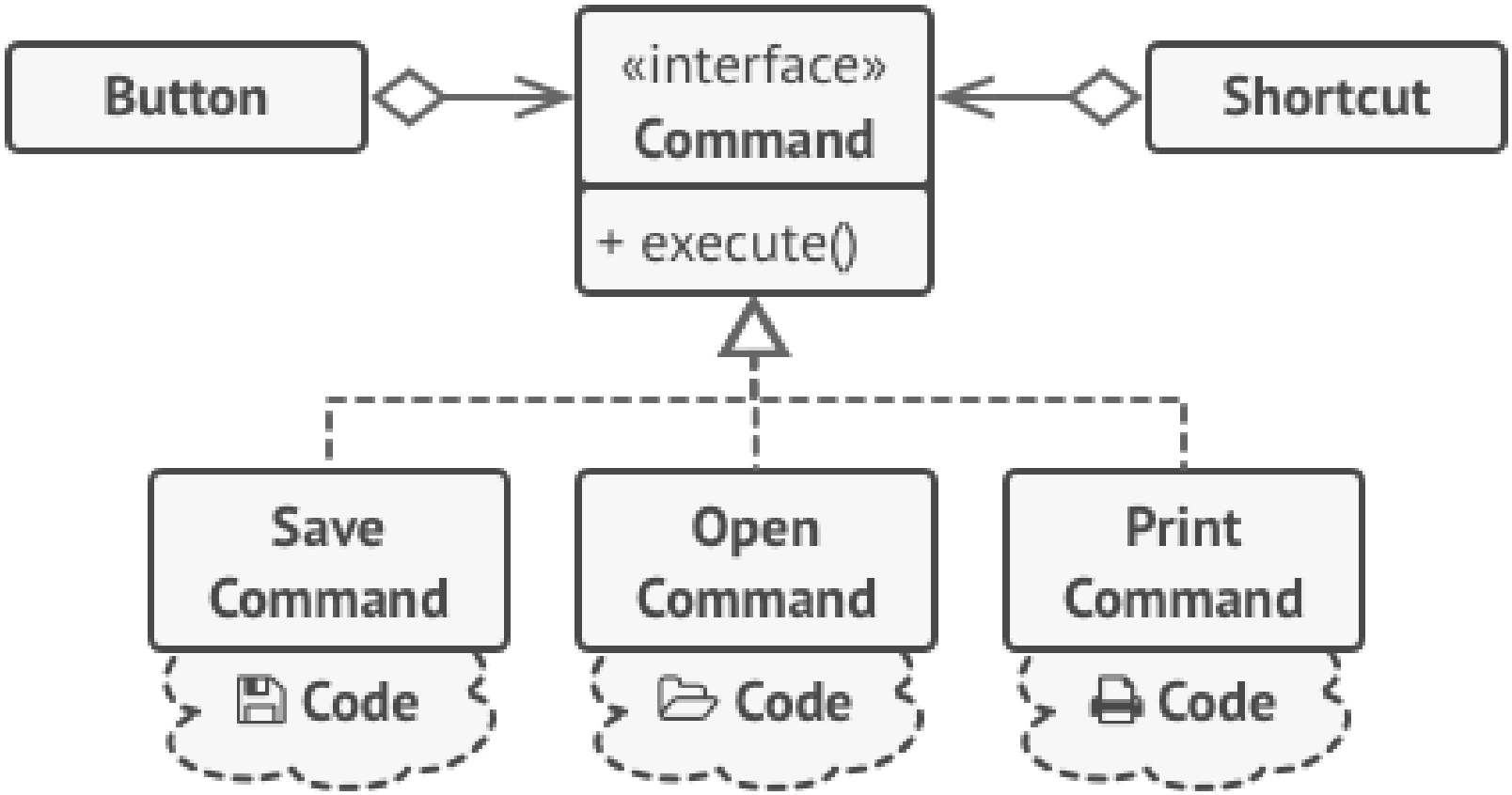
The GUI objects may access the business logic objects directly.

Patrón Command



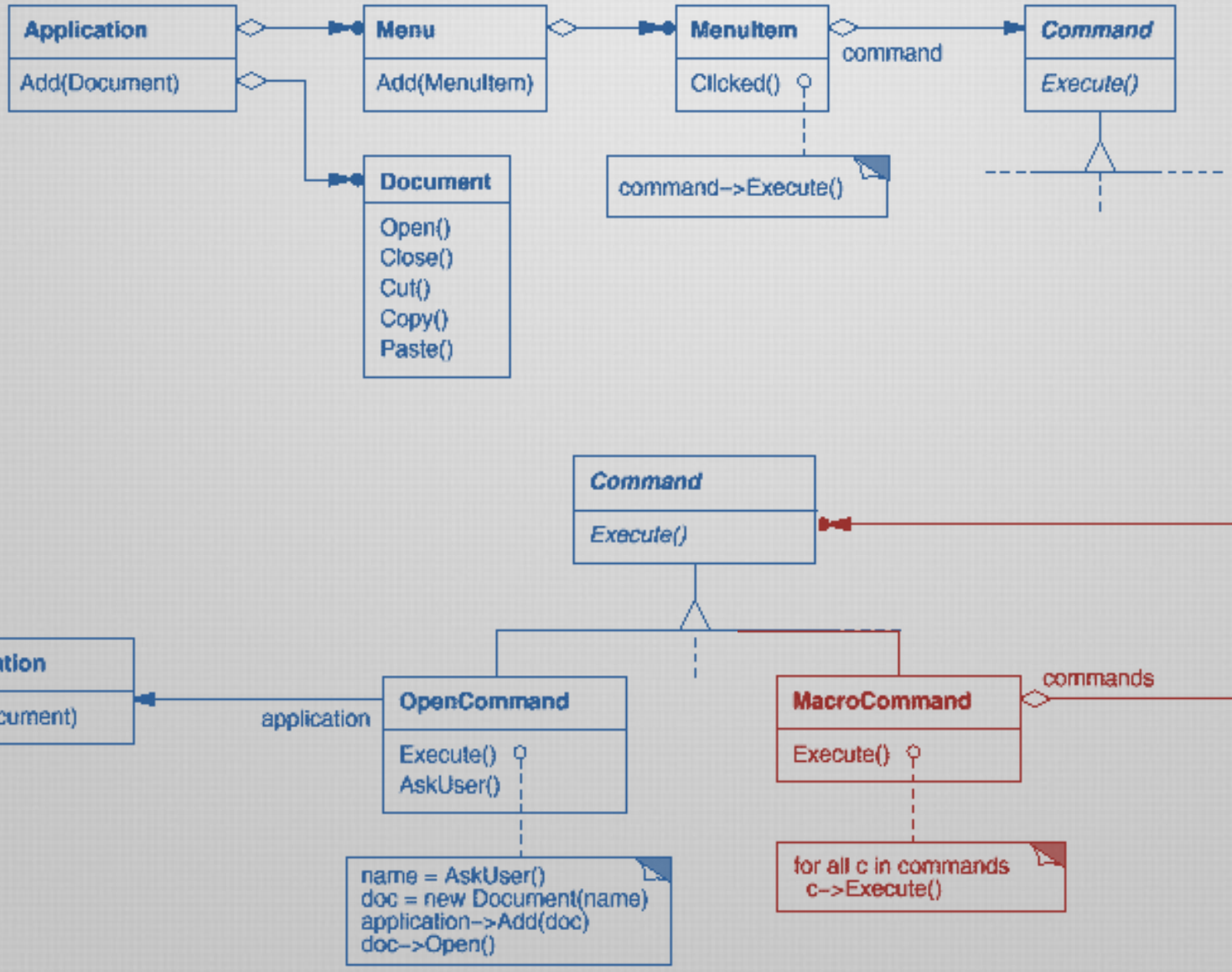
Accessing the business logic layer via a command.

Patrón Command



The GUI objects delegate the work to commands.

Patrón Command



Patrón Command

Intención: Encapsular el pedido (mensaje, solicitud) como un objeto, permitiendo parametrizar los clientes con diferentes pedidos, encolar o registrar los pedidos y proveer operaciones para deshacer pedidos previos.

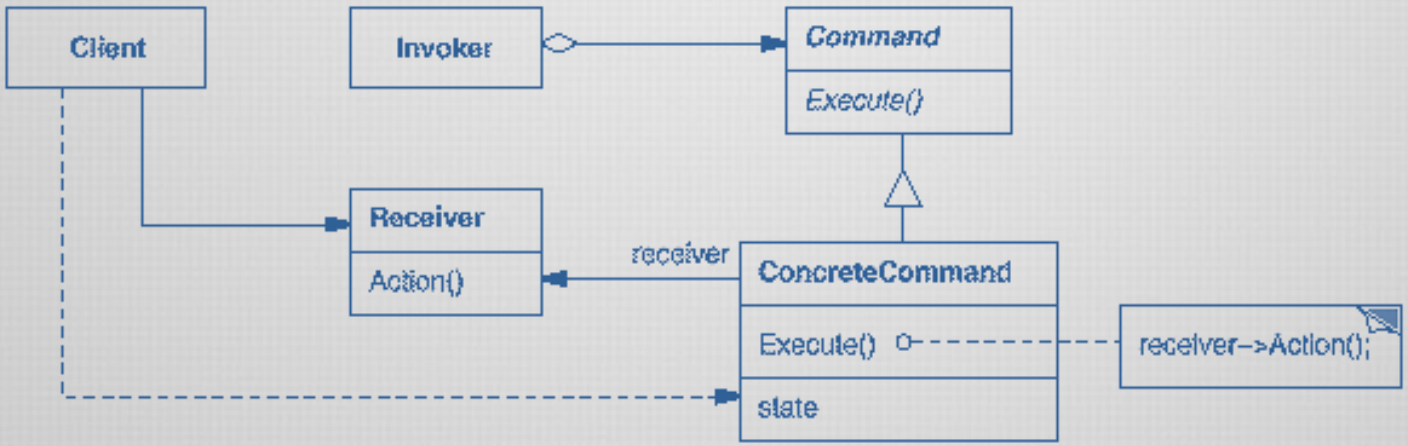
Alias: Action, Transaction.

Aplicabilidad: Usaremos este patrón cuando deseamos:

- Parametrizar objetos para una acción a realizar.
- Especificar, encolar y ejecutar pedidos en momentos diferentes.
- Proveer la posibilidad de deshacer acciones.
- Proveer registros de auditoría y salvaguarda.
- Estructurar un sistema en función de operaciones de alto nivel construidas en base a operaciones primitivas.

Patrón Command

Estructura



Participantes:

Command: declara una interfaz para ejecutar operaciones.

ConcreteCommand: define el vínculo entre el objeto *Receiver* y una acción. Implementa *Execute*() invocando las operaciones correspondientes sobre el Receiver.

Client: crea un objeto *ConcreteCommand* y setea el receptor.

Invoker: solicita al comando realizar su tarea.

Receiver: conoce cómo realizar operaciones asociadas al llevar a cabo un pedido. Cualquier clase puede actuar como Receiver.

Patrón Observer

Un problema recurrente en la programación orientada a objetos es el mantenimiento de la consistencia entre objetos relacionados y cooperativos, sin padecer de alto acoplamiento.

Este problema cobra también importancia en la representación visual de objetos complejos.
Cuando el objeto cambia, su representación visual (que también es un objeto) debe cambiar acordemente (ejemplo: gráficos en Excel)

Una forma de lograr esto es organizar los objetos en observadores de un aspecto particular (*observers-subject*)

La propuesta de organización se refleja en el patrón *Observer*.

Patrón Observer

Intención: Define una *dependencia entre objetos de uno-a-muchos* de forma tal que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados acordemente.

Alias: *Dependents, Publish-Suscribe*

Aplicabilidad: Usaremos este patrón cuando:

- Cuando una abstracción tiene dos aspectos, uno independiente del otro.
- Cuando un cambio a un objeto requiere cambios en otros, y no sabemos cuántos objetos necesitan ser cambiados.
- Cuando un objeto debería notificar a otros objetos sin realizar suposiciones de quiénes son esos objetos (evitar el acoplamiento)

Patrón Observer