



Universidad del Valle de Guatemala

Ingeniería Software I
Sección 20

Patrones de diseño

Pablo José Méndez Alvarado – 23975

Luis Fernando Palacios López – 23933

Roberto Samuel Nájera Marroquín – 23781

André Emilio Pivaral López – 23574





Resumen e Introducción

Resumen: analiza tres patrones de diseño en ingeniería de software: Builder, Mediator y Visitor. Se describen su propósito, estructura, implementación y aplicaciones en el desarrollo de software. Cada patrón resuelve problemas específicos, mejorando la modularidad, escalabilidad y mantenimiento del código.

Introducción: En el desarrollo de software, los patrones de diseño proporcionan soluciones reutilizables a problemas recurrentes en la arquitectura de sistemas. Estos patrones ayudan a mejorar la organización del código, reducir el acoplamiento entre componentes y fomentar buenas prácticas de programación.

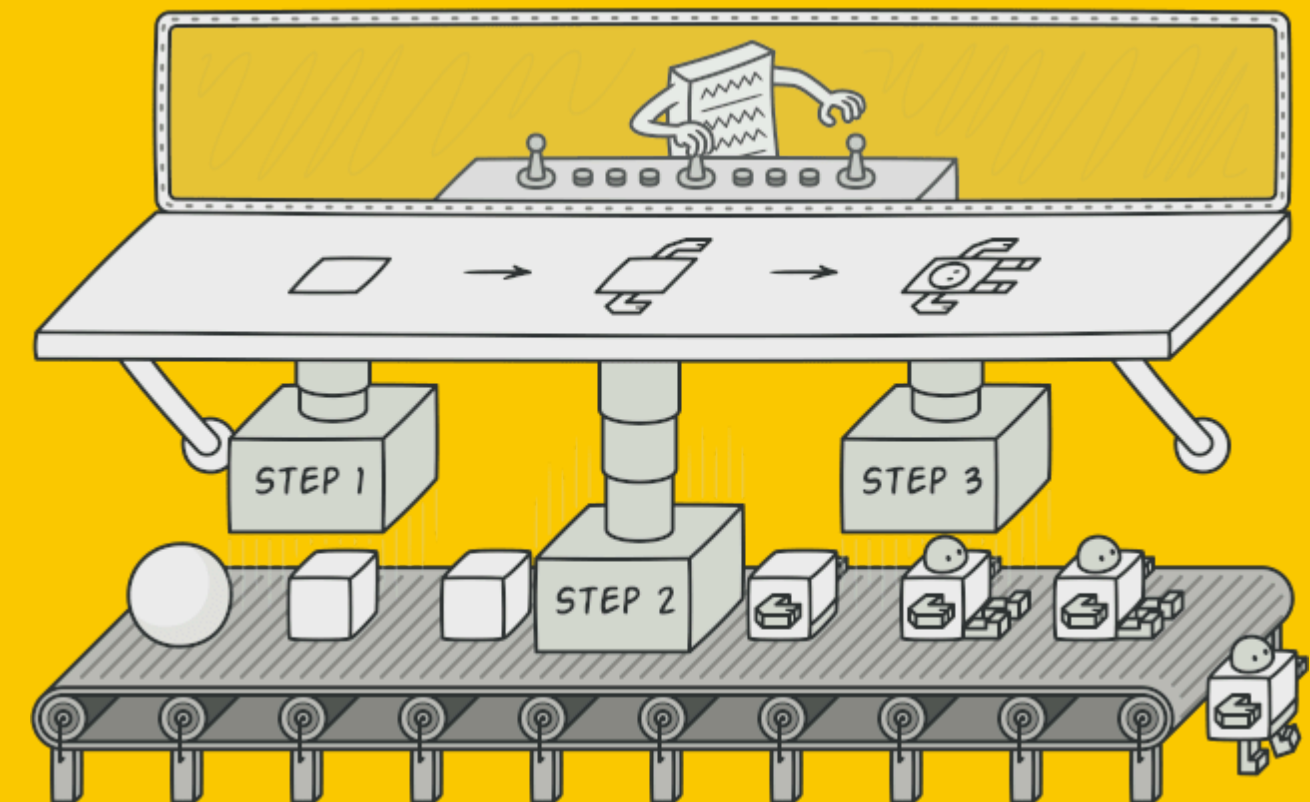
Builder

Intención: Permite la creación de objetos complejos paso a paso. Separa la construcción de un objeto de su representación, permitiendo diferentes representaciones para un mismo proceso de construcción.

Motivo:

Cuando un objeto tiene múltiples parámetros opcionales o debe construirse de manera controlada, el uso de un constructor con demasiados parámetros puede hacer que el código sea difícil de leer y mantener. Builder resuelve este problema proporcionando una forma clara y controlada de construir objetos.

Patrones relacionados: Factory Method, Prototype






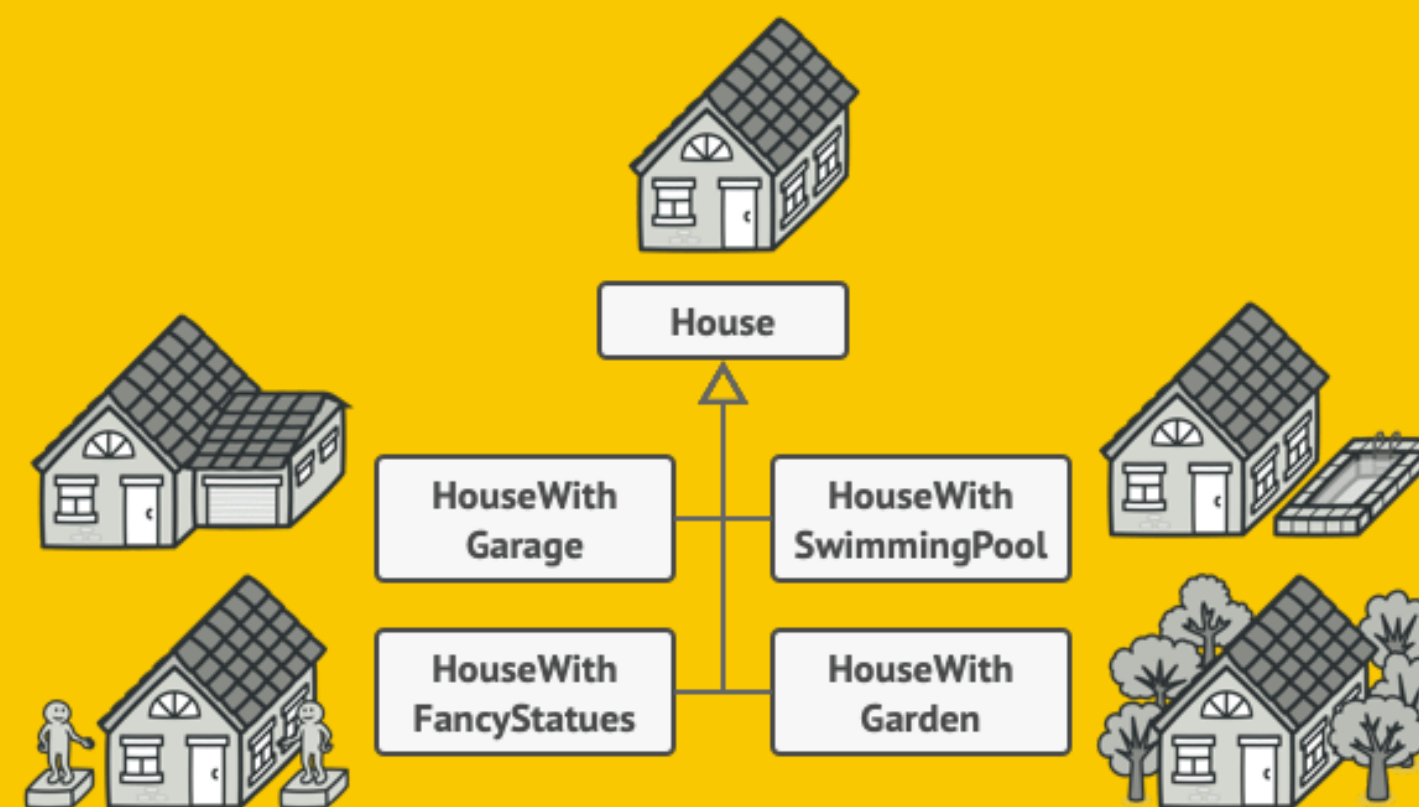
Builder

Aplicaciones y usos conocidos:

- Creación de objetos con múltiples configuraciones opcionales.
- Construcción de objetos inmutables.
- Uso en APIs para permitir configuraciones flexibles.
- StringBuilder en Java.
- Creación de objetos en frameworks de UI.

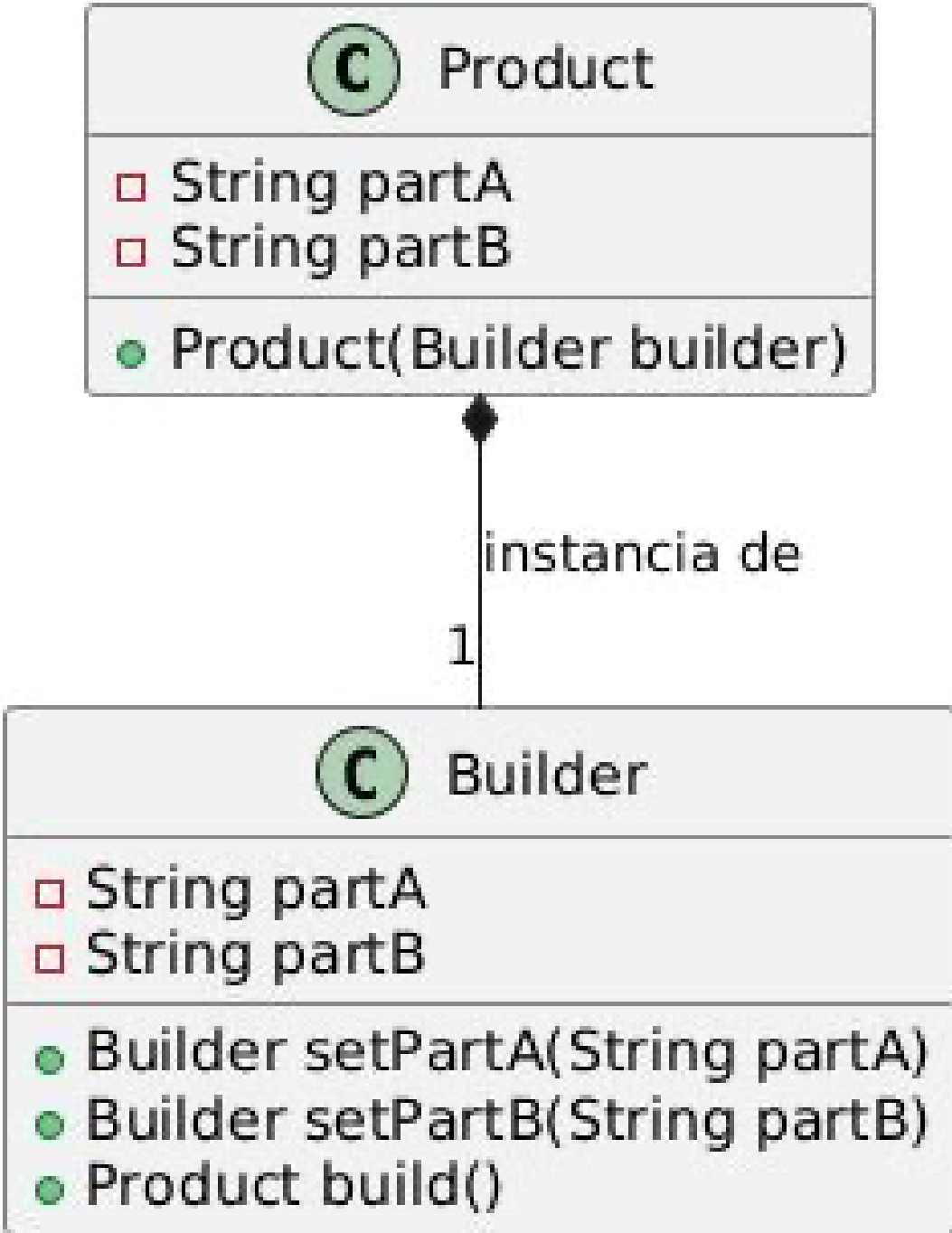
Consecuencias:

-  Mejora la legibilidad y modularidad.
-  Facilita la creación de objetos con múltiples configuraciones.
-  Puede aumentar la complejidad del código si se usa innecesariamente.



Estructura y código de ejemplo

- Builder: Declara una interfaz para la creación de partes del objeto.
- Concrete Builder: Implementa la interfaz de construcción.
- Director: Encapsula el proceso de construcción.
- Product: Representa el objeto complejo que se crea.



```
class Product {
    private String partA;
    private String partB;

    public static class Builder {
        private String partA;
        private String partB;

        public Builder setPartA(String partA) {
            this.partA = partA;
            return this;
        }

        public Builder setPartB(String partB) {
            this.partB = partB;
            return this;
        }

        public Product build() {
            return new Product(this);
        }
    }

    private Product(Builder builder) {
        this.partA = builder.partA;
        this.partB = builder.partB;
    }
}
```

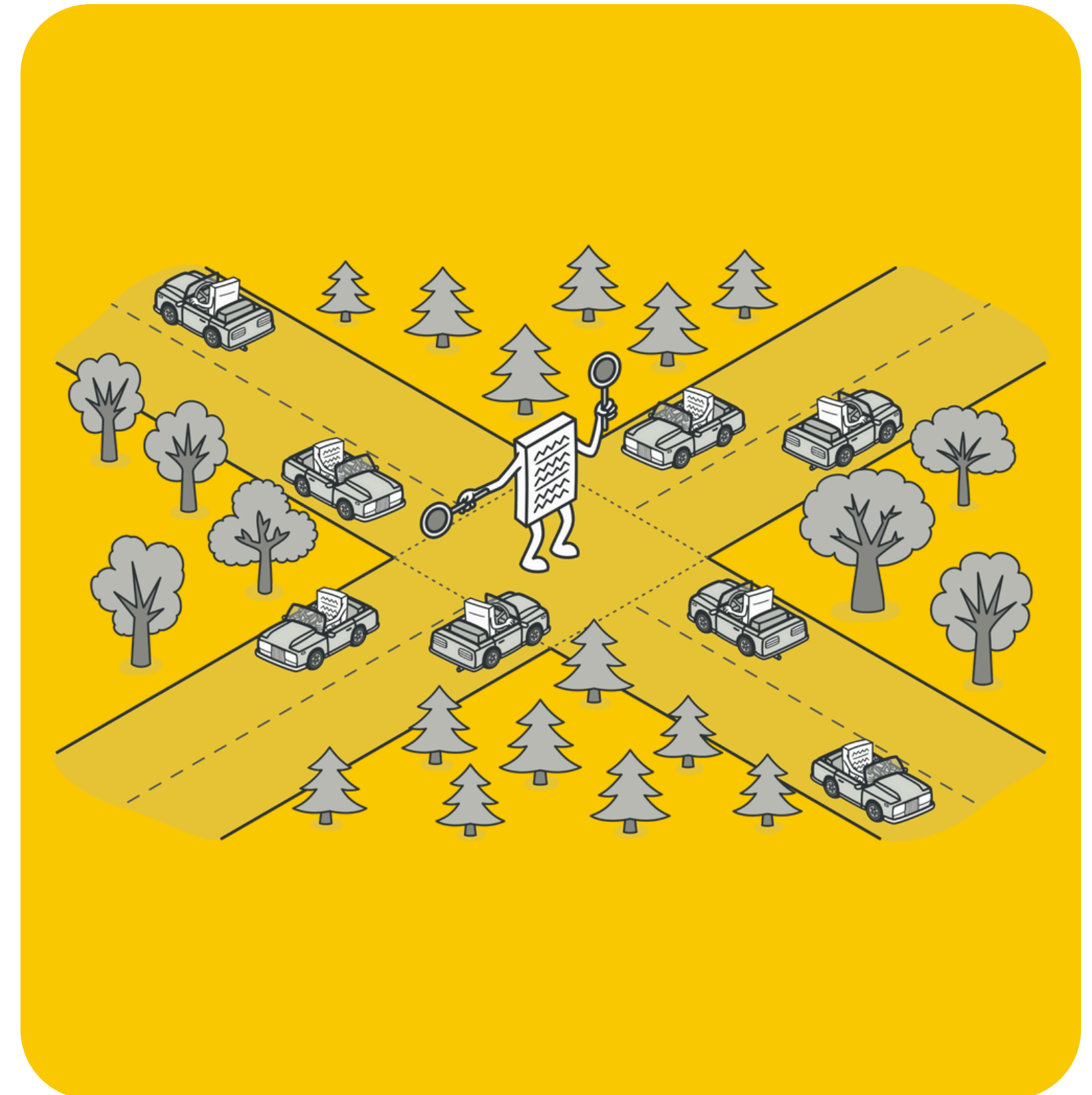
Mediator

Intención: Define un objeto que encapsula la comunicación entre múltiples objetos, promoviendo la reducción de dependencias directas entre ellos.

Motivo:

Cuando múltiples objetos interactúan entre sí, las dependencias pueden volverse difíciles de gestionar. Mediator centraliza la comunicación y simplifica la colaboración entre objetos.

Patrones relacionados: Observer, Facade, Colleague



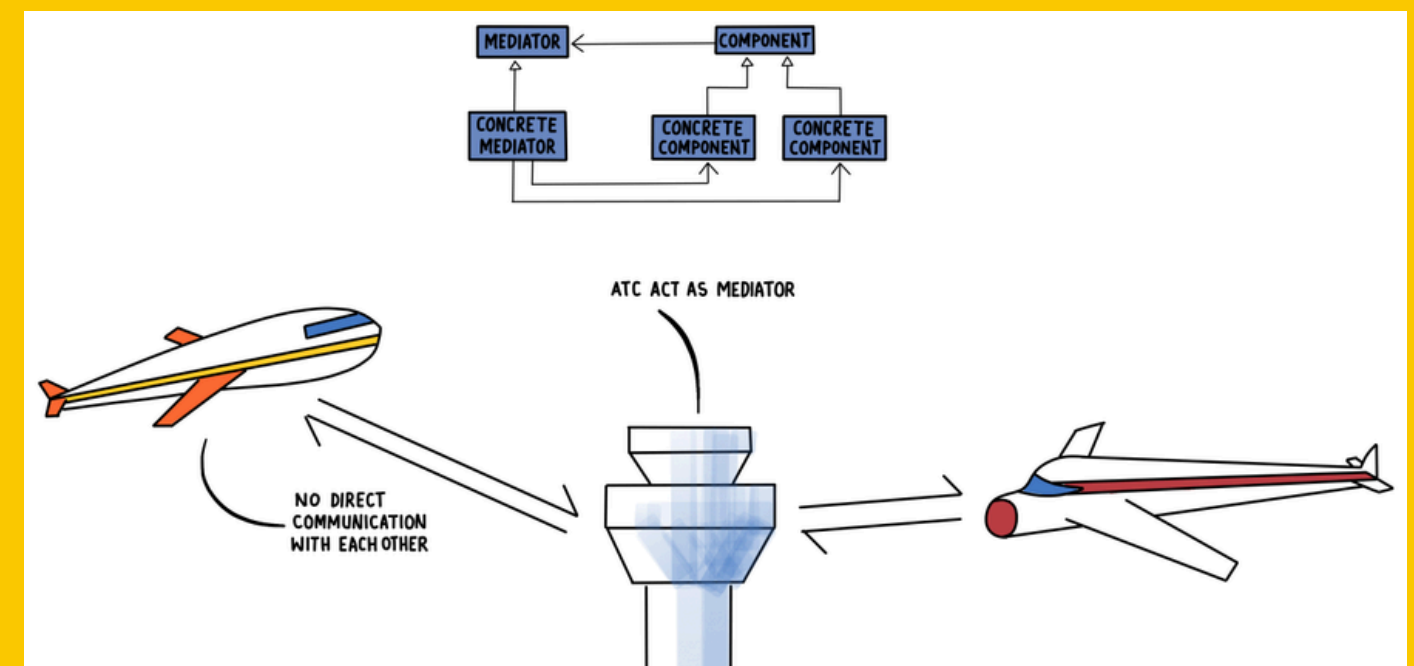
Mediator

Aplicaciones y usos conocidos:

- Sistemas de chat y comunicación.
- Gestión de eventos en interfaces gráficas.
- Coordinación de múltiples componentes en software.
- Controladores en *MVC*.
- Bibliotecas GUI como *Swing* y *Qt*.

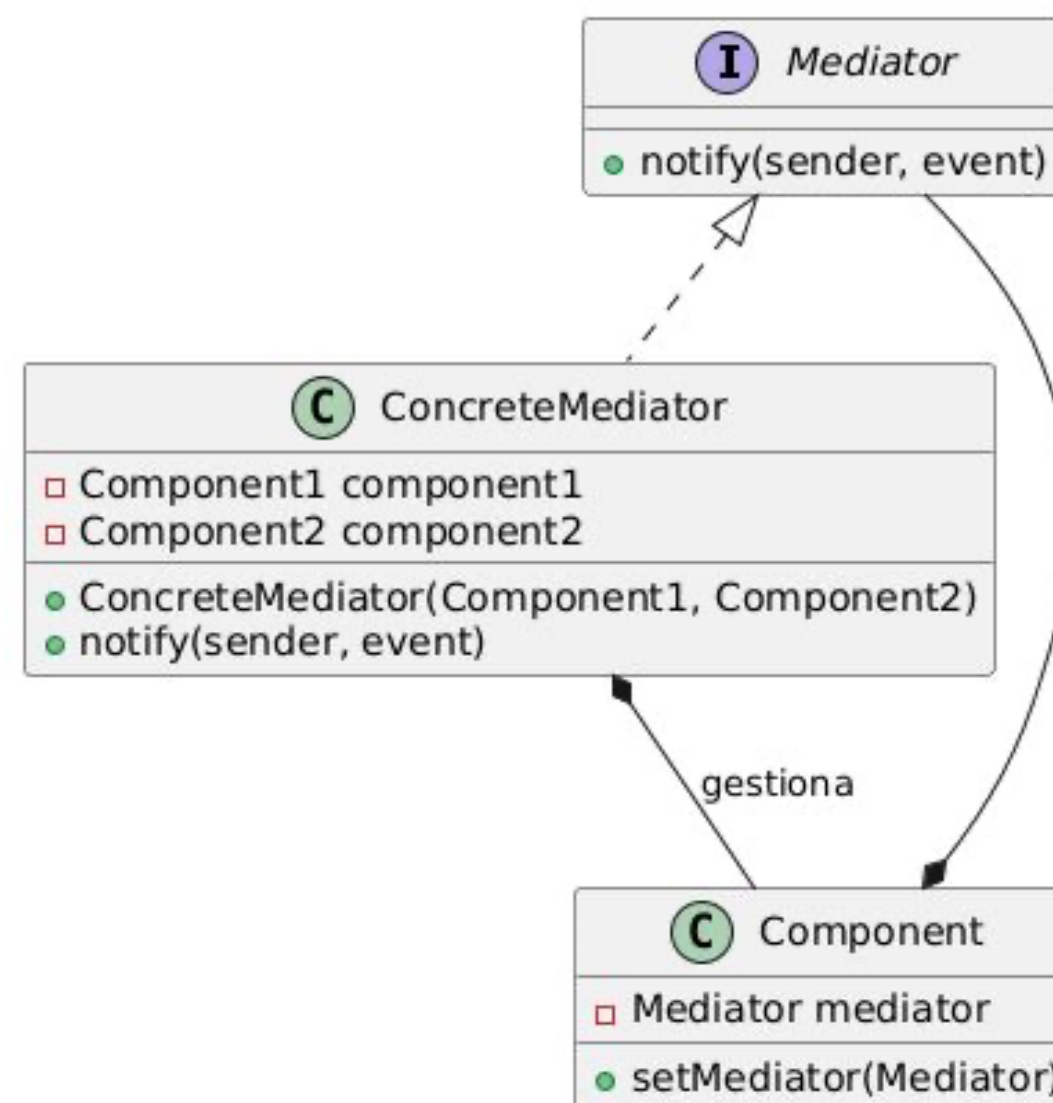
Consecuencias:

- ☒ Reduce la complejidad de la comunicación entre objetos.
- ☒ Fomenta el principio de responsabilidad única.
- ⚠ Puede convertirse en un punto único de fallo si no se diseña adecuadamente.



Estructura y código de ejemplo

- Mediator: Define la interfaz para la comunicación entre objetos.
- ConcreteMediator: Implementa la lógica de coordinación.
- Colleagues: Objetos que interactúan a través del Mediator.



```
class Mediator:
    def notify(self, sender, event):
        pass
```

```
class ConcreteMediator(Mediator):
    def __init__(self, component1, component2):
        self.component1 = component1
        self.component2 = component2
        self.component1.mediator = self
        self.component2.mediator = self
```

```
def notify(self, sender, event):
    if event == "A":
        print("Mediator responde al evento A y llama a Component2")
        self.component2.do_something()
    elif event == "B":
        print("Mediator responde al evento B y llama a Component1")
        self.component1.do_something()
```

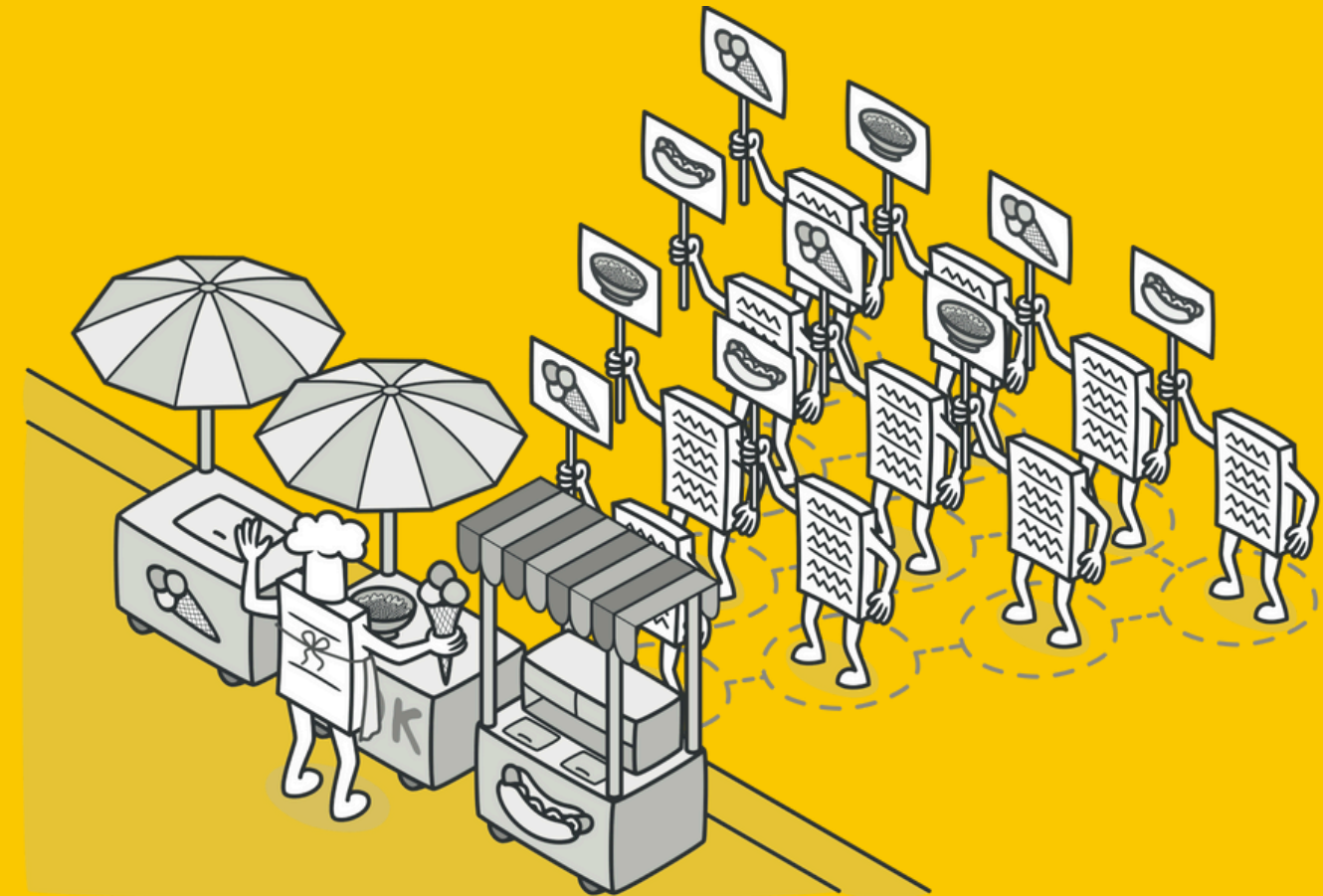

Visitor

Intención: Separar un algoritmo de una estructura de objetos sobre la cual opera. Permite definir nuevas operaciones sin cambiar las clases de los elementos sobre los que opera.

Motivo:

- Se requiere agregar nuevas operaciones a una estructura de objetos sin modificar sus clases.
- Hay una estructura de clases compleja y se desea definir operaciones independientes que puedan actuar sobre sus elementos.

Patrones relacionados: Composite, Iterator, Strategy



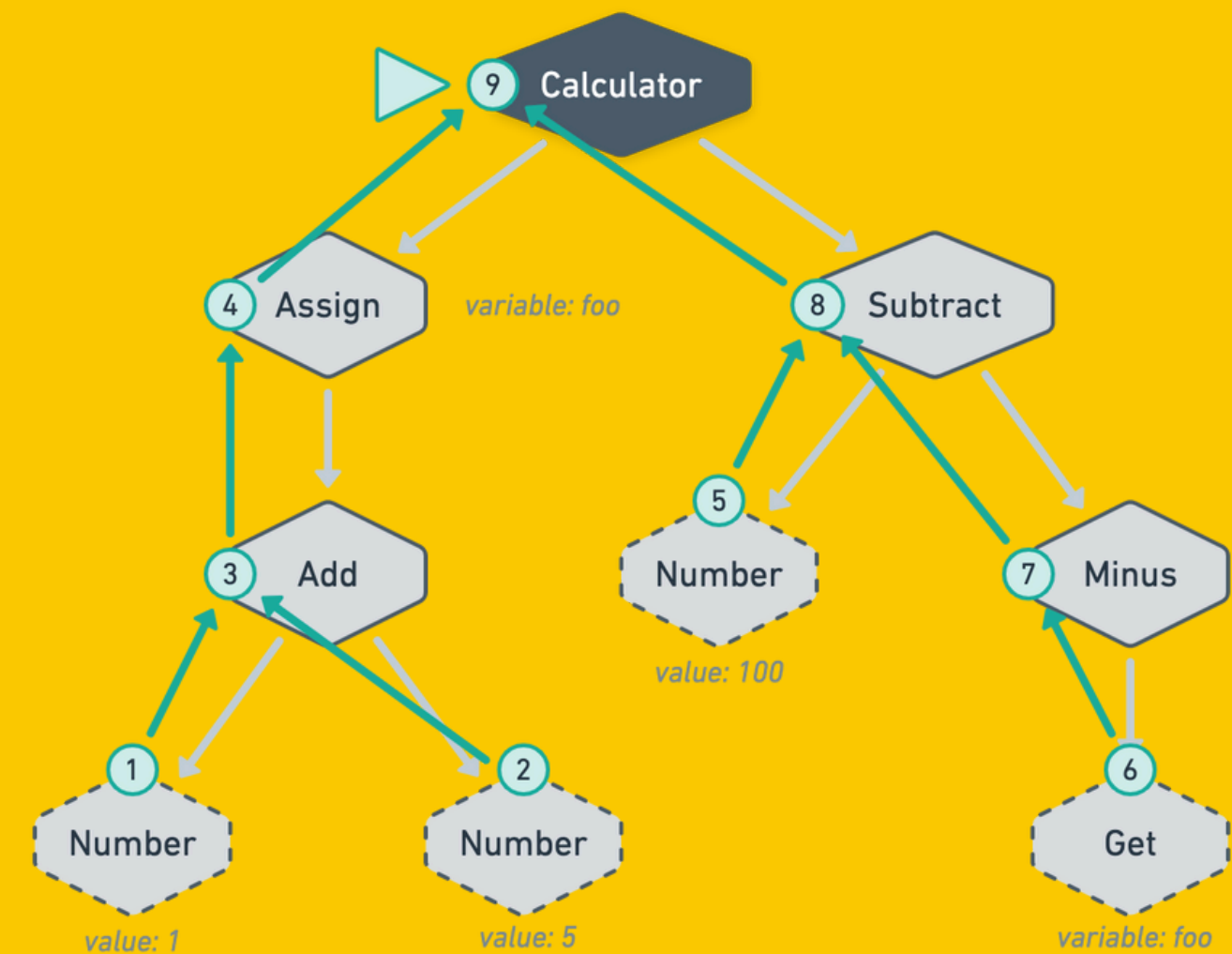
Visitor

Aplicaciones y usos conocidos:

- Análisis léxico y sintáctico
- Sistemas de archivos
- Aplicaciones gráficas
- Procesamiento de documentos

Consecuencias:

- Fácil adición de nuevas operaciones
- Separación de responsabilidades
- Compatibilidad con estructuras de objetos complejas
- Rompe el encapsulamiento



Estructura y código de ejemplo

- Visitor: Define una interfaz con métodos visit() para cada tipo de elemento que debe procesar.
- ConcreteVisitor: Implementa la lógica específica para cada tipo de elemento.
- Element: Declara el método accept(Visitor visitor).
- ConcreteElement (ElementA y ElementB): Implementan accept(), permitiendo que el Visitor los procese.
- Cliente (VisitorPatternDemo): Crea una lista de elementos y los recorre aplicando el visitante.

```
1 // Interfaz Visitor
2 interface Visitor {
3     void visit(ElementA element);
4     void visit(ElementB element);
5 }
6 // Implementaciones de Visitor
7 class ConcreteVisitor implements Visitor {
8     @Override
9     public void visit(ElementA element) {
10         System.out.println("Procesando ElementA");
11     }
12
13     @Override
14     public void visit(ElementB element) {
15         System.out.println("Procesando ElementB");
16     }
17 }
18 // Interfaz Elemento
19 interface Element {
20     void accept(Visitor visitor);
21 }
22 // Implementaciones de Element
23 class ElementA implements Element {
24     @Override
25     public void accept(Visitor visitor) {
26         visitor.visit(this);
27     }
28 }
29 class ElementB implements Element {
30     @Override
31     public void accept(Visitor visitor) {
32         visitor.visit(this);
33     }
34 }
35 // Cliente que usa el patrón Visitor
36 public class VisitorPatternDemo {
37     public static void main(String[] args) {
38         Element[] elements = {new ElementA(), new ElementB()};
39         Visitor visitor = new ConcreteVisitor();
40
41         for (Element element : elements) {
42             element.accept(visitor);
43         }
44     }
45 }
```

