Universidad del Valle de Guatemala

Facultad de Ingeniería

Departamento de Ciencia de la Computación

Ingeniería de Software I

PATRONES DE DISEÑO

INFORME DE INVESTIGACIÓN

Pablo José Méndez Alvarado – 23975

Luis Fernando Palacios López – 239333

Roberto Samuel Nájera Marroquín – 23781

André Emilio Pivaral López – 23574

Catedrático: Erick Francisco Marroquín Rodríguez

Sección: 20

Nueva Guatemala de la Asunción, 5 de marzo de 2024

Índice

Resumen

Este documento presenta un análisis detallado de tres patrones de diseño en ingeniería de software: Builder, Mediator y Visitor. Se describe su intención, objetivos, aplicaciones, estructura, implementación y ejemplos prácticos. Además, se incluyen diagramas y código para facilitar la comprensión de cada patrón.

Introducción

Los patrones de diseño son soluciones reutilizables a problemas comunes en el diseño de software. En este documento, se analizan tres patrones de diseño específicos: Builder, Mediator y Visitor, explicando su propósito, aplicaciones y cómo pueden mejorar la estructura y mantenimiento del código.

Patrón Builder

Intención

El patrón Builder 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 (Buschmann et al, 1996; Gamma et al, 1994).

Objetivos

- Facilitar la creación de objetos complejos sin sobrecargar los constructores.
- Separar la construcción del objeto de su representación final.
- Aumentar la legibilidad y reutilización del código.

Conocido como

- Generador
- Constructor

Motivo

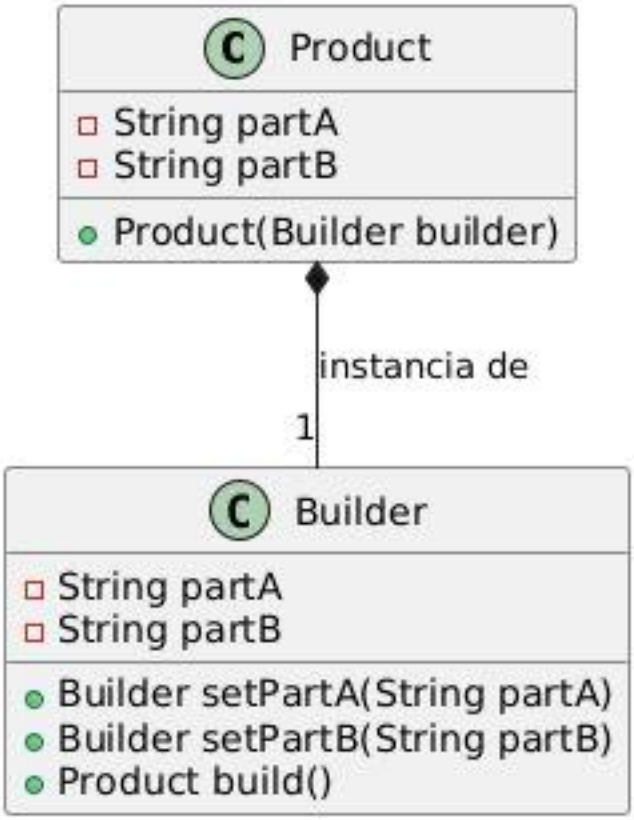
Cuando un objeto tiene múltiples parámetros opcionales o debe construirse de manera controlada, el uso de un constructor con demasiados parámetros puede hacer que el código sea

difícil de leer y mantener. Builder resuelve este problema proporcionando una forma clara y controlada de construir objetos (Freeman, 2004; Fowler, 2002).

Aplicaciones

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

Estructura



Participantes

- 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.

Colaboraciones

El *Director* utiliza el *Builder* para construir un *Product* siguiendo un conjunto de pasos definidos.

Consecuencias

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

Implementación

Decisiones de diseño:

- Usar métodos encadenados (Fluent Interface) mejora la legibilidad.
- Un Director es opcional, útil si hay configuraciones estándar.
- Se recomienda que el objeto construido sea inmutable.

Formas de codificación:

- En Java y C++, es común usar una clase estática anidada.
- En Python y JavaScript, se emplean métodos encadenados.

Código de Ejemplo

```
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;
    }
}
```

Usos conocidos

- StringBuilder en Java.
- Creación de objetos en frameworks de UI.

Patrones relacionados

- Factory Method
- Prototype

Patrón Visitor

Intención

El patrón Visitor tiene como objetivo 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. Esto es útil cuando se necesita agregar nuevas funcionalidades a una estructura de clases sin modificar su código base, promoviendo así el principio de abierto/cerrado (open/closed principle) (Gamma, Helm, Johnson, & Vlissides, 1994; Freeman & Robson, 2004).

Objetivos

- Composite: El patrón Visitor a menudo se utiliza en combinación con el patrón Composite. Mientras que el Composite organiza los objetos en estructuras jerárquicas, el Visitor permite definir nuevas operaciones que pueden realizarse sobre todos los objetos de la jerarquía (Gamma et al., 1994).
- Iterator: Ambos patrones abordan el problema de recorrer una estructura de objetos. El Iterator proporciona una manera estándar de recorrer elementos de una colección, mientras que el Visitor permite realizar operaciones específicas sobre cada elemento de la estructura (Gamma et al., 1994).
- Strategy: El patrón Visitor puede considerarse una forma de estrategia que permite cambiar el algoritmo que se aplica a los objetos sin cambiar sus clases (Gamma et al., 1994).

Conocido como

- Operation as a Class
- Double Dispatch
- Acyclic Visitor
- External Polymorphism

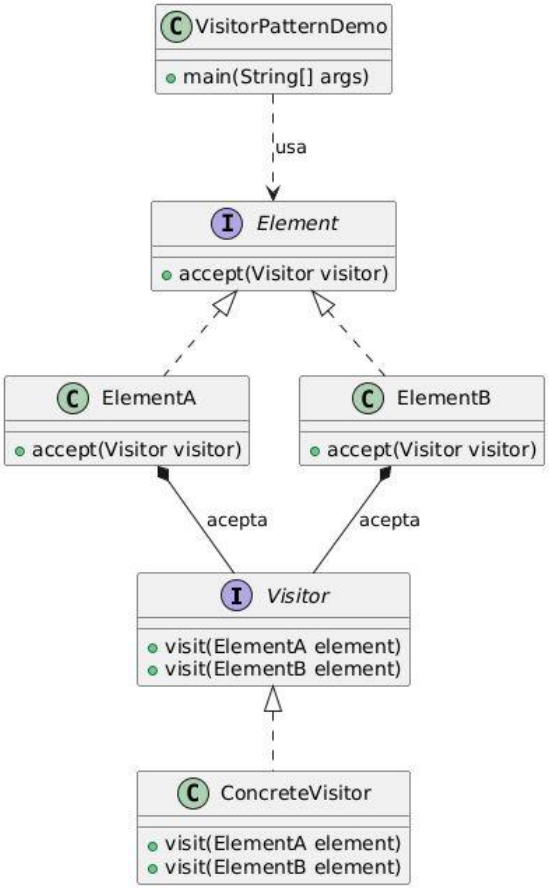
Motivo

- 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.
- Se necesita realizar operaciones que impliquen cambios en las clases sin violar su encapsulamiento.

Aplicaciones

- **Análisis léxico y sintáctico**: En compiladores y analizadores de código, el patrón Visitor se utiliza para realizar análisis léxico y sintáctico de programas. Se puede definir una estructura de clases para representar la gramática del lenguaje y utilizar visitantes para realizar operaciones como análisis de errores, generación de código intermedio o optimización.
- **Sistemas de archivos**: En sistemas de archivos, se puede utilizar el patrón Visitor para realizar operaciones sobre la jerarquía de directorios y archivos. Por ejemplo, se puede definir un visitante que calcule el tamaño total de un directorio incluyendo todos sus subdirectorios y archivos.
- **Aplicaciones gráficas**: En aplicaciones gráficas, el patrón Visitor puede aplicarse para realizar operaciones sobre los elementos gráficos de una interfaz de usuario, como dibujar, mover o redimensionar componentes gráficos sin modificar sus clases.
- **Procesamiento de documentos**: En aplicaciones de procesamiento de documentos, como editores de texto o sistemas de generación de informes, el patrón Visitor puede utilizarse para aplicar operaciones de formato, exportación o impresión a los elementos del documento, como párrafos, tablas e imágenes.

Estructura



Participantes

- Visitor
- ConcreteVisitor
- Element
- Cliente (VisitorPatternDemo)

Colaboraciones

- 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.

Consecuencias

Ventajas:

Fácil adición de nuevas operaciones: Permite agregar nuevas operaciones sin modificar las clases de la estructura de objetos. Esto facilita la extensibilidad y cumple con el principio de abierto/cerrado (open/closed principle) (Gamma et al., 1994).

Separación de responsabilidades: El patrón Visitor separa las operaciones de los datos sobre los que operan, promoviendo un diseño más limpio y modular (Freeman & Robson, 2004).

Compatibilidad con estructuras de objetos complejas: Es particularmente útil para estructuras de objetos complejas que cambian frecuentemente, ya que permite definir nuevas operaciones sin alterar las clases existentes (Gamma et al., 1994).

Desventajas:

Dificultad para agregar nuevas clases: Agregar nuevas clases a la estructura de objetos puede ser complicado, ya que todas las clases de visitantes deben implementar una operación para la nueva clase (Gamma et al., 1994).

Rompe el encapsulamiento: El patrón Visitor puede violar el principio de encapsulamiento, ya que requiere acceso a las partes internas de los elementos de la estructura para realizar operaciones específicas (Freeman & Robson, 2004).

Complejidad adicional: Implementar el patrón Visitor puede añadir complejidad al código, especialmente en sistemas donde las estructuras de objetos son simples y las operaciones no cambian con frecuencia (Gamma et al., 1994).

Implementación

<u>Decisiones de Diseño</u>

Separación de Responsabilidades: Al usar el patrón Visitor, las operaciones sobre la estructura de objetos se separan de la estructura misma, promoviendo un diseño más modular.

Doble Despacho: Utiliza el mecanismo de doble despacho para asegurar que se invoque el método correcto de la clase visitante basado en el tipo del elemento visitado (Freeman & Robson, 2004).

Extensibilidad: Permite agregar nuevas operaciones sin modificar las clases de los elementos, siguiendo el principio de abierto/cerrado.

Encapsulamiento: Se debe tener cuidado de no violar el encapsulamiento de las clases de elementos al implementar visitantes.

Formas de Codificación

Interfaces y Clases Abstractas: Utiliza interfaces y clases abstractas para definir los contratos y asegurar que las clases concretas los implementen correctamente.

Colección de Elementos: Usa estructuras de datos como listas o conjuntos para almacenar los elementos de la estructura del objeto y permitir que sean visitados.

Ejecución Centralizada: Centraliza la ejecución de las operaciones en el visitante, permitiendo una gestión más sencilla de las operaciones.

Código de Ejemplo

```
// Interfaz Visitor

interface Visitor {

    void visit(ElementA element);

    void visit(ElementB element);

}

// Implementaciones de Visitor

class ConcreteVisitor implements Visitor {

    @Override

    public void visit(ElementA element) {
```

```
        System.out.println("Procesando ElementA");
    }
    @Override
    public void visit(ElementB element) {
        System.out.println("Procesando ElementB");
    }
}
// Interfaz Elemento
interface Element {
    void accept(Visitor visitor);
}
// Implementaciones de Element
class ElementA implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
class ElementB implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
```

```
    }
}
// Cliente que usa el patrón Visitor
public class VisitorPatternDemo {
    public static void main(String[] args) {
        Element[] elements = {new ElementA(), new ElementB()};
        Visitor visitor = new ConcreteVisitor();

        for (Element element : elements) {
            element.accept(visitor);
        }
    }
}
```

Usos conocidos

Usos Conocidos del Patrón Visitor

Compiladores:

Productos: GCC (GNU Compiler Collection), Clang

Descripción: En compiladores como GCC y Clang, el patrón Visitor se utiliza para realizar análisis y transformaciones sobre el árbol de sintaxis abstracta (AST). Los visitantes pueden realizar operaciones como la generación de código, optimización de código y detección de errores en diferentes etapas del proceso de compilación.

Sistemas de Gestión de Archivos:

Productos: Windows File Explorer, macOS Finder

Descripción: En sistemas de gestión de archivos, el patrón Visitor se puede utilizar para realizar operaciones sobre la jerarquía de directorios y archivos, como el cálculo del tamaño total, la búsqueda de archivos, la generación de informes, y más. Cada archivo o directorio puede aceptar un visitante que realice estas operaciones de manera uniforme.

Motores de Renderizado de Gráficos:

Productos: Unity, Unreal Engine

Descripción: En motores de renderizado gráfico, el patrón Visitor se utiliza para aplicar operaciones sobre los elementos de la escena gráfica, como el renderizado, la actualización de transformaciones, y la detección de colisiones. Esto permite añadir nuevas funcionalidades de manera modular sin modificar las clases de los elementos gráficos.

Procesadores de Documentos:

Productos: Microsoft Word, Adobe Acrobat

Descripción: En procesadores de documentos, el patrón Visitor se utiliza para aplicar operaciones de formato, exportación, e impresión a los elementos del documento, como párrafos, tablas, imágenes, y más. Cada elemento del documento puede aceptar un visitante que realice estas operaciones, facilitando la extensión de funcionalidades sin cambiar la estructura del documento.

Herramientas de Análisis de Código:

Productos: SonarQube, ESLint

Descripción: En herramientas de análisis de código, el patrón Visitor se utiliza para recorrer la estructura de código fuente y aplicar verificaciones de estilo, detección de errores, y sugerencias de refactorización. Los visitantes pueden implementar diferentes reglas de análisis y aplicarlas a las clases, métodos, y otros componentes del código.

Sistemas que Utilizan el Patrón Visitor

Frameworks de Pruebas:

JUnit (Java): En JUnit, el patrón Visitor se puede utilizar para recorrer la estructura de pruebas y aplicar operaciones como la ejecución de pruebas, la recolección de resultados, y la generación de informes.

Sistemas de Modelado y Simulación:

MATLAB Simulink: En herramientas como MATLAB Simulink, el patrón Visitor se puede utilizar para aplicar operaciones sobre los elementos del modelo de simulación, como la generación de código, la ejecución de simulaciones, y la recolección de datos.

Plataformas de Comercio Electrónico:

Magento: En plataformas de comercio electrónico como Magento, el patrón Visitor se puede utilizar para aplicar operaciones sobre los elementos del catálogo de productos, como la actualización de precios, la generación de informes de inventario, y la exportación de datos de productos.

Patrones relacionados

Composite: El patrón Visitor a menudo se utiliza en combinación con el patrón Composite. Mientras que el Composite organiza los objetos en estructuras jerárquicas, el Visitor permite definir nuevas operaciones que pueden realizarse sobre todos los objetos de la jerarquía.

Iterator: Ambos patrones abordan el problema de recorrer una estructura de objetos. El Iterator proporciona una manera estándar de recorrer elementos de una colección, mientras que el Visitor permite realizar operaciones específicas sobre cada elemento de la estructura.

Strategy: El patrón Visitor puede considerarse una forma de estrategia que permite cambiar el algoritmo que se aplica a los objetos sin cambiar sus clases.

Patrón Mediator

Intención

El patrón Mediator, también conocido como patrón de mediador, es un diseño de comportamiento que se utiliza para reducir la complejidad de la comunicación entre múltiples objetos en un sistema, promoviendo un diseño más limpio y mantenible. Este patrón es particularmente relevante en el desarrollo de software orientado a objetos, donde la interacción directa entre objetos puede llevar a un acoplamiento estrecho, dificultando la escalabilidad y el mantenimiento (Gamma, et al., 1994).

Se define como un objeto que encapsula cómo un conjunto de objetos, denominados colegas, interactúan entre sí. Según la literatura, su propósito principal es evitar que los objetos se comuniquen directamente, lo que reduce las dependencias y facilita la modificación independiente de cada componente. Esto se logra introduciendo un mediador que actúa como un intermediario, centralizando la lógica de comunicación. Por ejemplo, en un sistema de chat, el mediador sería el servidor que gestiona los mensajes entre usuarios, evitando que cada usuario necesite conocer a los demás directamente (Freeman & Robson, 2004).

Objetivos

- Reducir el acoplamiento entre clases interdependientes.
- Centralizar la comunicación entre objetos.
- Facilitar la escalabilidad y mantenimiento.

Conocido como

- Intermediario, Mediador

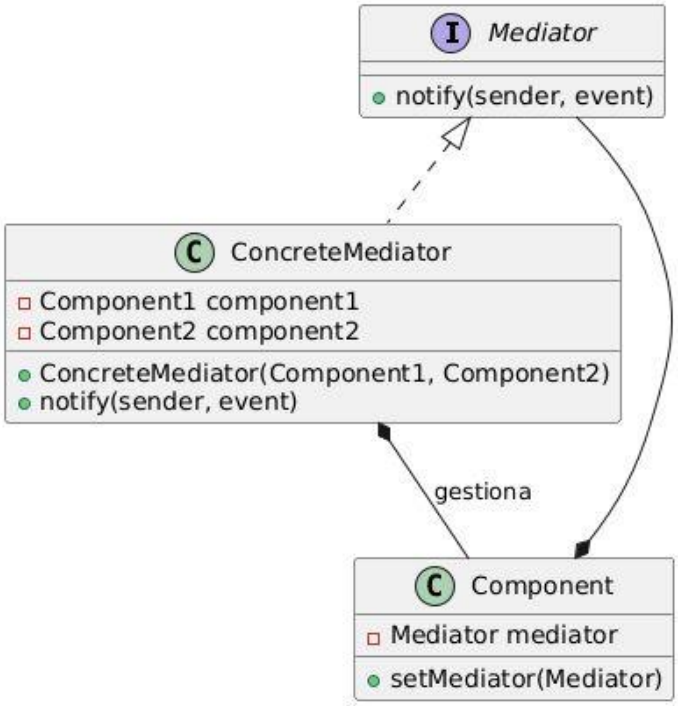
Motivo

Cuando múltiples objetos interactúan entre sí, las dependencias pueden volverse difíciles de gestionar. Mediator centraliza la comunicación y simplifica la colaboración entre objetos (Fowler, 2002).

Aplicaciones

- Sistemas de chat y comunicación.
- Gestión de eventos en interfaces gráficas.
- Coordinación de múltiples componentes en software.

Estructura



Participantes

- Mediator: Define la interfaz para la comunicación entre objetos.
- ConcreteMediator: Implementa la lógica de coordinación.
- Colleagues: Objetos que interactúan a través del Mediator (Freeman & Robson, 2004).

Colaboraciones

- Los Colleagues envían mensajes al Mediator, quien los redirige según corresponda.

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.

Implementación

El Mediator se implementa mediante una clase central que gestiona la comunicación entre los objetos sin que estos se refieran entre sí directamente.

Código de Ejemplo

```
(Ejemplo en Python)

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()
```

Usos conocidos

- Controladores en MVC.
- Bibliotecas GUI como Swing y Qt.

Patrones relacionados

- Observer

Conclusiones

Se analizaron los patrones Builder, Mediator y Visitor, explicando su propósito y aplicaciones en ingeniería de software. Cada uno de estos patrones proporciona una solución estructurada para problemas específicos en el diseño de software, promoviendo el mantenimiento y escalabilidad del código.

Anexos

Planificación

Tarea	Encargado	Fecha	Inicio	Fin
Asignación de tareas y organización concerniente a la entrega	André Pivaral	03/03/2025	23:00	00:00
Redacción de Resumen e Introducción	André Pivaral	04/03/2025	08:00	10:00
Investigación del Patrón de Diseño Builder	Roberto Nájera	04/03/2025	08:00	10:00
Investigación de la estructura del Patrón de Diseño Builder y Creación del ejemplo	Roberto Nájera	04/03/2025	10:00	12:00
Investigación del Patrón de Diseño Mediator	Luis Palacios	04/03/2025	08:00	10:00
Investigación de la estructura del Patrón de Diseño Mediator y Creación del ejemplo	Luis Palacios	04/03/2025	10:00	12:00
Investigación del Patrón de Diseño Visitor	Pablo Méndez	04/03/2025	08:00	10:00
Investigación de la estructura del Patrón de Diseño Visitor y Creación del ejemplo	Pablo Méndez	04/03/2025	10:00	12:00
Diseño de la presentación a utilizar en la exposición	Roberto Nájera y Pablo Méndez	05/03/2025	12:00	14:00
Corrección final del documento de la investigación	André Pivaral y Luis Palacions	05/03/2025	12:00	14:00

Bibliografía

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Freeman, E., & Robson, B. (2004). Head First Design Patterns. O'Reilly Media.

Bloch, J. (2017). Effective Java (3rd Edition). Addison-Wesley.

Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern-Oriented Software Architecture: A System of Patterns. Wiley.