Universidad de las Fuerzas Armadas-ESPE

Segundo Semestre de Ingeniería en Tecnologías de la Información y la Comunicación

Asignatura de Introducción a la Comunicación Académica

Control de lectura 2: Modelo Vista Controlador (MVC) y Patrones de Diseño

Integrantes:
Arequipa Tigasi Edwin Fernando
López Morales Dylan Joel
Pugachi Suarez Karol Elizabeth
Yar Zumba Evelyn Cristina

Nombre del docente:
Luis Enrique Jaramillo Montaño

Fecha: 25/02/2025

TÍTULO:

Modelo Vista Controlador (MVC) y Patrones de Diseño

¿Qué es el Modelo Vista Controlador (MVC)?

Se trata de un patrón de arquitectura de Software que separa el código dependiendo de sus responsabilidades, utilizado para implementar interfaces de usuario, datos y lógica de control, mediante tres capas que ejecutan una actividad especifica, modelos, vistas y controladores.

1.Cómo mejora la organización del código el patrón Modelo-Vista-Controlador (MVC)?

Al separar las responsabilidades de una aplicación en distintas partes se facilita la reutilización del código, el mantenimiento y la escalabilidad generando una mejor organización del código.

2.Para qué sirven?

Es un patrón de diseño de software que se utiliza para separar la lógica de una aplicación en tres componentes.

¿Cuándo se aplica?

- Se utiliza en aplicaciones web modernas
- Se utiliza en componentes gráficos básicos hasta sistemas empresariales
- Se utiliza en aplicaciones que necesitan ser escalables, mantenibles y fáciles de expandir

¿Cuándo no se aplica?

- No es recomendable para aplicaciones sencillas, ya que puede introducir complejidad innecesaria.
- Para aplicaciones sencillas, ya que separar la aplicación en tres capas puede no ser necesario.
- Para nuevos desarrolladores, ya que la curva de aprendizaje es mayor que otros modelos.
- Cuando la navegación por el código puede ser compleja, ya que hay más componentes y archivos.

Ejemplo:

MODELO

```java
package modelo;
public class Modelo {
    private int numero1,numero2,resultado;

    public int getNumero1() {
        return numero1;
    }

    public void setNumero1(int numero1) {
        this.numero1 = numero1;
    }

    public int getNumero2() {
        return numero2;
    }

    public void setNumero2(int numero2) {
        this.numero2 = numero2;
    }

    public int getResultado() {
        return resultado;
    }

    public void setResultado(int resultado) {
        this.resultado = resultado;
    }
    public int sumar(){
        resultado=numero1+numero2;
        return resultado;
    }
}
```

CONTROLADOR

```java
package controlador;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import modelo.Modelo;
import vista.Vista;

public class Controlador implements ActionListener{
    private Vista vista;
    private Modelo modelo;

    public Controlador(Vista vista, Modelo modelo) {
        this.vista = vista;
        this.modelo = modelo;
        vista.botonSumar.addActionListener(this);
    }

    public void iniciar(){
        vista.setTitle("MVC Sumar");
        vista.setLocationRelativeTo(null);
    }
    @Override
    public void actionPerformed(ActionEvent ae){
        modelo.setNumero1(Integer.parseInt(vista.cajaNumero1.getText()));
        modelo.setNumero2(Integer.parseInt(vista.cajaNumero2.getText()));
        modelo.sumar();
        vista.cajaResultado.setText(String.valueOf(modelo.getResultado()));
    }
}
```

VISTA

MAIN

```java
package mvc;

import controlador.Controlador;
import modelo.Modelo;
import vista.Vista;

public class Main {

    public static void main(String[] args) {
        Vista vista= new Vista();
        Modelo modelo = new Modelo();
        Controlador controlador = new Controlador(vista, modelo);

        controlador.iniciar();
        vista.setVisible(true);
    }
}
```

PATRÓN ESTRUCTURAL: Adapter Decorator

Los patrones estructurales son un tipo de patrón de diseño en programación que se enfocan en la composición de clases y objetos para facilitar la reutilización y la flexibilidad del código.

¿Cómo mejora la organización del código el patrón de diseño Adapter Decorator?

- Promueven un código modular y bien estructurado.
- Facilitan la reutilización al definir relaciones claras entre clases y objetos.
- Mejoran la mantenibilidad al separar responsabilidades y reducir dependencias acopladas.
- Hacen que el código sea más fácil de leer y comprender.

¿Para qué sirve el patrón de diseño Adapter Decorator?

- Para definir relaciones eficientes entre clases y objetos sin modificar su implementación.
- Para facilitar la interacción entre estructuras complejas.
- Para mejorar la extensibilidad del sistema, permitiendo agregar nuevas funcionalidades sin alterar el código existente.

¿Cuándo se aplica?

Se necesita organizar múltiples clases u objetos de manera eficiente.

Se requiere reutilización de código sin alterar su implementación original.

Se quiere mantener un bajo acoplamiento entre los componentes del sistema.

¿Cuándo no se aplica?

La solución es simple y no justifica la sobrecarga de un patrón.

No hay necesidad de cambiar la estructura de clases u objetos a futuro.

Se puede resolver el problema con una solución más directa y menos abstracta.

Ejemplo:

Imagina que tienes un enchufe europeo, pero viajas a EE.UU. Necesitas un adaptador. Lo mismo ocurre en la programación cuando dos clases no son compatibles, y ahí es donde entra el patrón Adaptador."

PATRÓN CREACIONAL: Factory Method – Método de Fabrica

Los patrones creacionales proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y reutilización de código existente, proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.
Permite la creación de objetos de un subtipo determinado a través de una clase Factory.
El patrón Factory Method mejora la organización del código al separar el código de construcción de productos del código que hace uso del producto. Es decir que el patrón ayuda a facilitar la extensión del código de construcción de forma independiente al resto del código.

¿Para qué sirve el patrón de diseño: Factory Method – Método de Fabrica?

- Permite la creación de objetos de un subtipo determinado a través de una clase Factory.
- Permite añadir nuevos tipos de objetos sin modificar el código existente.
- Permite alterar el tipo de objetos que se crearan por medio de las subclases.
- Permite manejar de forma genérica diferentes tipos de objetos.
- Facilita la creación de código modular y reutilizable.

¿Cuándo se aplica?

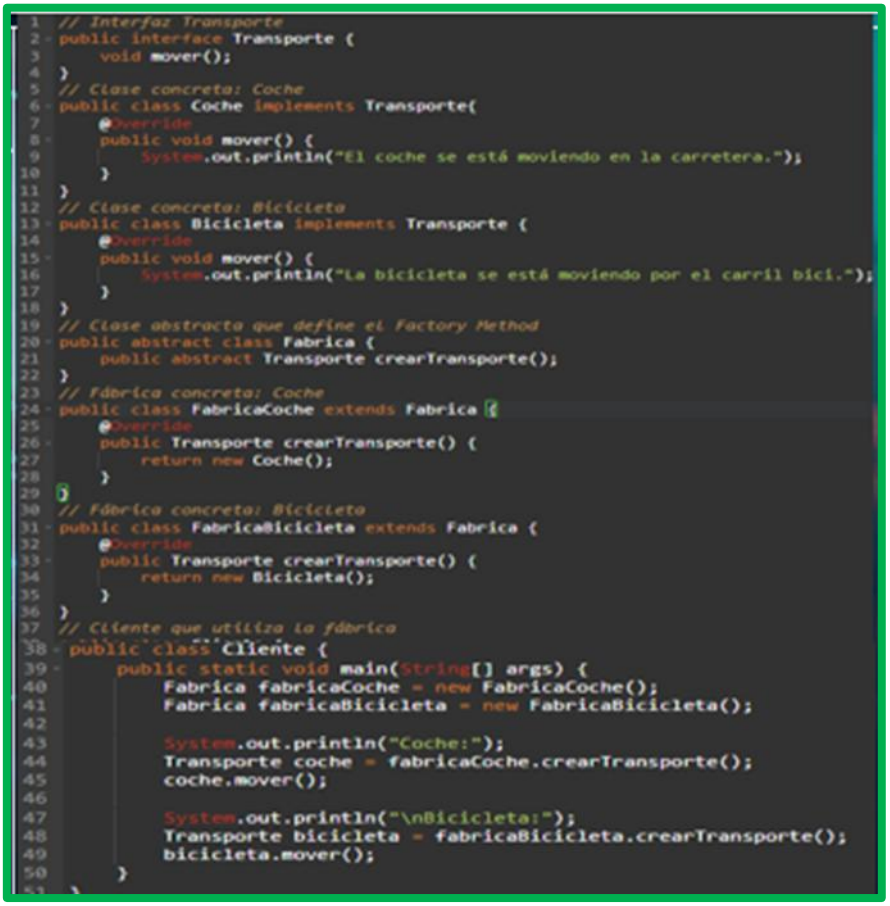
- Se aplica o se utiliza el método cuando no se conoce de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar el código.

- También se aplica cuando se quiere ofrecer al usuario una biblioteca o framework para poder extender los componentes internos.

¿Cuándo no se aplica?

- No se aplica el método cuando no se conoce el subtipo que se va a utilizar en un tiempo de diseño.
- Cuando no es necesario desacoplar la creación de objetos.
- Para las aplicaciones donde las clases no van a cambiar, un Factory Method podría ser un exceso.
- Si la creación de los objetos es simple y no requiere distintas variables, el método sería innecesario.

Ejemplo en JAVA



```java
// Interfaz Transporte
public interface Transporte {
    void mover();
}
// Clase concreta: Coche
public class Coche implements Transporte{
    @Override
    public void mover() {
        System.out.println("El coche se está moviendo en la carretera.");
    }
}
// Clase concreta: Bicicleta
public class Bicicleta implements Transporte {
    @Override
    public void mover() {
        System.out.println("La bicicleta se está moviendo por el carril bici.");
    }
}
// Clase abstracta que define el Factory Method
public abstract class Fabrica {
    public abstract Transporte crearTransporte();
}
// Fábrica concreta: Coche
public class FabricaCoche extends Fabrica {
    @Override
    public Transporte crearTransporte() {
        return new Coche();
    }
}
// Fábrica concreta: Bicicleta
public class FabricaBicicleta extends Fabrica {
    @Override
    public Transporte crearTransporte() {
        return new Bicicleta();
    }
}
// Cliente que utiliza la fábrica
public class Cliente {
    public static void main(String[] args) {
        Fabrica fabricaCoche = new FabricaCoche();
        Fabrica fabricaBicicleta = new FabricaBicicleta();

        System.out.println("Coche:");
        Transporte coche = fabricaCoche.crearTransporte();
        coche.mover();

        System.out.println("\nBicicleta:");
        Transporte bicicleta = fabricaBicicleta.crearTransporte();
        bicicleta.mover();
    }
}
```

PATRÓN DE COMPORTAMIENTO: INTERPRETER

Los patrones de comportamiento se enfocan en la comunicación y la interacción entre objetos. El Patrón Observador permite establecer una relación de dependencia entre un objeto (sujeto) y múltiples observadores, de manera que todos sean notificados automáticamente cuando el estado del sujeto cambie.

Este patrón de diseño se utiliza para definir la gramática de un lenguaje y proporcionar un intérprete para procesar las instrucciones en ese lenguaje.

¿Cómo mejora la organización del código?

- Facilita un código modular y flexible.

- Reduce el acoplamiento entre clases, permitiendo que trabajen de forma independiente.
- Permite la reactividad en los sistemas, notificando cambios en tiempo real.
- Mejora la mantenibilidad y escalabilidad del software

¿Para qué sirve?

- Se usa para establecer comunicación eficiente entre objetos sin acoplarlos directamente.
- Es ideal para sistemas de eventos, interfaces gráficas y modelos de publicación/suscripción.
- Permite manejar múltiples suscriptores sin modificar la lógica del sujeto.

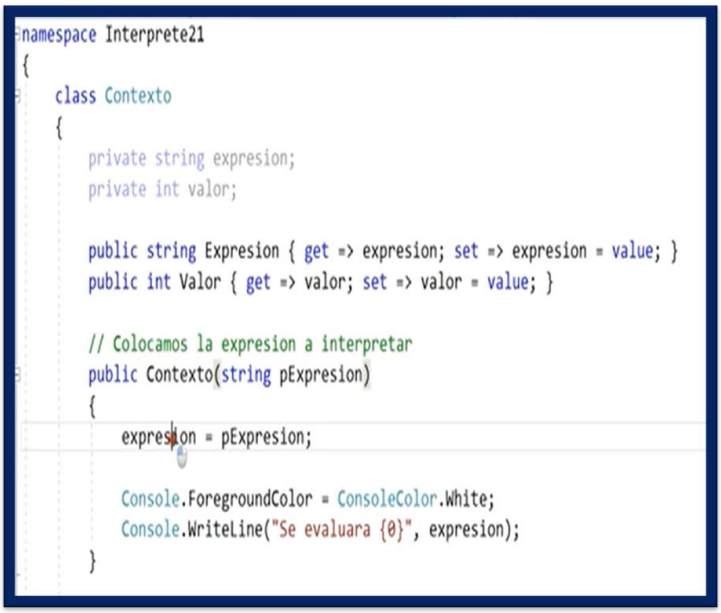
¿Cuándo se aplica?

- Se necesita que múltiples objetos reaccionen a cambios en otro objeto.
- Se diseñan sistemas dinámicos y escalables (notificaciones, juegos, interfaces).
- Se trabaja con modelos publicador-suscriptor en arquitectura de software.

¿Cuándo no se aplica?

- No es necesario que varios objetos respondan a un mismo evento.
- El número de observadores es pequeño y conocido de antemano.
- Se busca una solución más simple sin sobrecarga de patrones.

Ejemplo:



```
namespace Interprete21
{
    class Contexto
    {
        private string expresion;
        private int valor;

        public string Expresion { get => expresion; set => expresion = value; }
        public int Valor { get => valor; set => valor = value; }

        // Colocamos la expresion a interpretar
        public Contexto(string pExpresion)
        {
            expresion = pExpresion;

            Console.ForegroundColor = ConsoleColor.White;
            Console.WriteLine("Se evaluara {0}", expresion);
        }
```

Bibliografía

Hernández, U. (2015). MVC (Model, View, Controller) explicado: https://codigofacilito.com/articulos/mvc-model-view-controller-explicado

López, M. (2023). Estructura y funcionalidades del patrón Modelo Vista Controlador: https://immune.institute/blog/patron-modelo-vista-controlador/

REFACTORING. (2020). Patrones de Diseño. Patrones Creacionales: https://refactoring.guru/es/design-patterns/creational-patterns