PROGRAMACIÓN ORIENTADA A OBJETOS

Control de Lectura #2

Integrantes:

Jaya Taipe Jefferson Fernando

Ortiz Córdova Nick Mateo

Proaño Panchi Isaac Sebastian

Vilaña Puga Nayeli Estefania

Docente: Ing. Luis Enrique Jaramillo

Nrc:1322

Fecha: 25/02/2025

Objetivo general:

Entender el Modelo Vista Controlador (MVC) y otros patrones de diseño pertinentes, estudiando su influencia en la organización y estructuración del código durante el desarrollo de software.

Objetivos específicos:

Examinar la idea, estructura y operación del patrón de diseño MVC.

Reconocer y detallar otros modelos de diseño empleados en la creación de software.

Describir cómo la implementación de patrones de diseño potencia la mantención, escalabilidad y estructuración del código.

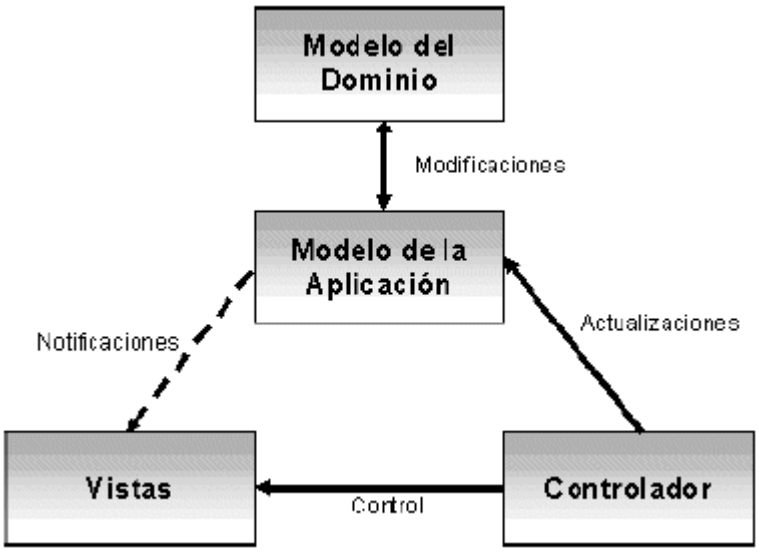
Elaborar una síntesis que condense los elementos fundamentales del patrón MVC y su uso en proyectos de software.

Marco Teórico

MVC es un patrón de diseño que se estructura mediante tres componentes: modelo, vista y controlador. Este patrón tiene como principio que cada uno de los componentes esté separado en diferentes objetos, esto significa que los componentes no se pueden combinar dentro de una misma clase. Sirve para clasificar la información, la lógica del sistema y la interfaz que se le presenta al usuario.

- Modelo: este componente se encarga de manipular, gestionar y actualizar los datos de una base de datos. No contiene ninguna lógica que describa como presentar los datos a un usuario.
- Vista: este componente presenta los datos del modelo al usuario. La vista sabe cómo acceder a los datos del modelo, pero no sabe que significa esta información o que puede hacer el usuario para manipularla.

- Controlador: este componente se encarga de gestionar las instrucciones que se reciben, atenderlas y procesarlas. Por medio del controlador se comunican el modelo y la vista: solicitando los datos necesarios, manipularlos para obtener los resultados y entregarlos a la vista para que pueda mostrarlos.



Ventajas

- Separación de preocupaciones:

MVC divide la lógica de la aplicación en tres capas independientes: Modelo, Vista y Controlador. Esto facilita la organización y el mantenimiento del código, ya que cada capa tiene una responsabilidad clara.

- Facilidad para el mantenimiento y la escalabilidad:

Debido a la separación de responsabilidades, es más fácil hacer cambios en una parte de la aplicación sin afectar otras. Esto facilita la escalabilidad a medida que el proyecto crece.

- Reutilización de código:

El modelo puede ser reutilizado en varias vistas sin tener que duplicar código, lo que favorece la eficiencia en el desarrollo.

Desventajas

- Complejidad inicial:

Para proyectos pequeños, usar el patrón MVC puede parecer innecesario. La estructura puede ser demasiado compleja si la aplicación no requiere una separación tan rigurosa, lo que puede resultar en un sobrecosto de tiempo y recursos al inicio.

- Sobrecarga de código:

Al dividir la aplicación en varias capas, puede haber más código para gestionar. Esto puede llevar a una sobrecarga, especialmente si la aplicación no necesita una separación tan detallada.

- Dificultad en la comunicación entre capas:

Aunque MVC promueve la separación de responsabilidades, a veces las capas necesitan comunicarse entre sí, lo que puede hacer que el flujo de información sea más complejo y difícil de manejar en aplicaciones grandes.

Ejemplo Practico:

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Modelo;
public class Coche {
    private String marca;
    private String modelo;
    private int año;

    public Coche(String marca, String modelo, int año) {
        this.marca = marca;
        this.modelo = modelo;
        this.año = año;
    }

    public String getMarca() {
        return marca;
    }

    public String getModelo() {
        return modelo;
    }

    public int getAño() {
        return año;
    }

    @Override
    public String toString() {
        return año + " " + marca + " " + modelo;
    }
}
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Controlador;

import Modelo.Coche;
import Vista.VistaCoche;
import java.util.ArrayList;
import java.util.List;

public class ControladorCoche {
    private List<Coche> coches;
    private VistaCoche vista;

    public ControladorCoche(VistaCoche vista) {
        this.vista = vista;
        this.coches = new ArrayList<>();
    }

    public void agregarCoche(String marca, String modelo, int año) {
        Coche coche = new Coche(marca, modelo, año);
        coches.add(coche);
    }

    public void mostrarCoches() {
        vista.imprimirDetallesCoches(coches);
    }
}
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Vista;

import Modelo.Coche;
import java.util.List;

public class VistaCoche {
    public void imprimirDetallesCoches(List<Coche> coches) {
        System.out.println("Lista de coches:");
        for (Coche coche : coches) {
            System.out.println(coche);
        }
    }
}
```

Patrones de comportamiento

Entonces, descubrimos cómo se pueden crear objetos y cómo se pueden organizar las conexiones entre clases. Lo más interesante que queda es brindar flexibilidad para cambiar el comportamiento de los objetos. Y los patrones de comportamiento nos ayudarán con esto. Uno de los patrones mencionados con más frecuencia es el patrón " Estrategia ". Aquí es donde comienza el estudio de los patrones en el libro " La cabeza primero. Patrones de diseño ". Usando el patrón “Estrategia”, podemos almacenar dentro de un objeto cómo realizaremos la acción, es decir el objeto interno almacena una estrategia que se puede cambiar durante la ejecución del código. Este es un patrón que usamos a menudo cuando usamos un comparador:

```
import java.util.*;
class Main {
  public static void main(String[] args) {
    List<String> data = Arrays.asList("Moscow", "Paris", "NYC");
    Comparator<String> comparator = Comparator.comparingInt(String::length);
    Set dataSet = new TreeSet(comparator);
    dataSet.addAll(data);
    System.out.println("Dataset : " + dataSet);
  }
}
```

Este comportamiento tiene una estrategia predeterminada, que vemos en JavaDoc: ordenar en "orden natural" (para cadenas, este es el orden lexicográfico). Esto sucede si usa un constructor sin parámetros. Pero si queremos cambiar la estrategia, podemos pasar Comparator. En este ejemplo, podemos crear nuestro conjunto como new TreeSet(comparator)y luego el orden de almacenamiento de elementos (estrategia de almacenamiento) cambiará al especificado en el comparador. nosotros todos los días cuando usamos hilos:

```java
import java.util.logging.*;
class Main {
  public static void main(String[] args) {
    Logger logger = Logger.getLogger(Main.class.getName());
    ConsoleHandler consoleHandler = new ConsoleHandler(){
        @Override
            public void publish(LogRecord record) {
                System.out.println("LogRecord обработан");
            }
        };
    logger.addHandler(consoleHandler);
    logger.info("test");
  }
}
```

Como puede ver, los controladores se agregan a la lista de controladores de registradores. Cuando un registrador recibe un mensaje para procesar, cada mensaje pasa a través de una cadena de controladores. Otro patrón que vemos todos los días es el " Iterador ". Su esencia es separar una colección de objetos (es decir, una clase que representa una estructura de datos, por ejemplo, List) y recorrer esta colección.

```java
import java.util.*;
class Main {
  public static void main(String[] args) {
    List<String> data = Arrays.asList("Moscow", "Paris", "NYC");
    Iterator<String> iterator = data.iterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.next());
    }
  }
}
```

El ejemplo se parece más a un patrón de comando. Y la esencia del patrón "Mediador" está oculta en la implementación de Timer'a. Dentro del temporizador hay una cola de tareas TaskQueuey un hilo TimerThread. Nosotros, como clientes de esta clase, no interactuamos con

ellos, sino que interactuamos con Timerel objeto que, en respuesta a nuestra llamada a sus métodos, accede a los métodos de otros objetos de los que es intermediario.

La conclusión es que el código está escrito de tal manera que a los usuarios del código (desarrolladores) se les proporciona alguna plantilla de algoritmo, cuyos pasos pueden redefinirse. Esto permite a los usuarios del código no escribir el algoritmo completo, sino pensar solo en cómo realizar correctamente uno u otro paso de este algoritmo. Por ejemplo, Java tiene una clase abstracta AbstractListque define el comportamiento de un iterador mediante List. Sin embargo, el propio iterador utiliza métodos de hoja como: get, set, remove. El comportamiento de estos métodos lo determina el desarrollador de los descendientes AbstractList.

Veamos un ejemplo:

```java
import java.io.*;
import java.util.*;
class Main {
  public static void main(String[] args) throws IOException {
    ArrayList<String> list = new ArrayList<>();
    list.add("test");

    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    try (ObjectOutputStream out = new ObjectOutputStream(stream)) {
      out.writeObject(list);
    }

    byte[] bytes = stream.toByteArray();
    InputStream inputStream = new ByteArrayInputStream(bytes);
    try (ObjectInputStream in = new ObjectInputStream(inputStream)) {
      List<String> listNew = (List<String>) in.readObject();
      System.out.println(listNew.get(0));
    } catch (ClassNotFoundException e) {

    }
```

Recomendaciones

Considerar la implementación y uso del patrón MVC para mejorar la organización mediante la separación de responsabilidades.

Indagar sobre otros patrones de diseño e implementarlos en el proyecto o fusionarlos con otros patrones compatibles.

Evaluar que tan factible es implementar estos patrones en nuestro proyecto.

Conclusiones

El incluir un patrón de diseño permite realizar cambios en una parte sin afectar a otra, facilitando la escalabilidad y mantenimiento del código.

El MVC es un patrón muy usado debido a su capacidad de dividir responsabilidades entre el Modelo, la Vista y el Controlador.

Los patrones de diseño pueden reducir el tiempo de codificación gracias a su reutilización de código.