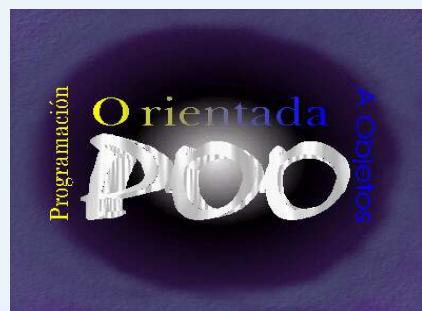


Universidad Centroccidental “Lisandro Alvarado”
Decanato de Ciencias y Tecnología
Departamento de Sistemas
Programación



Arquitectura Modelo - Vista - Controlador



Lapso 2025-2

Coordinación del Área de Programación

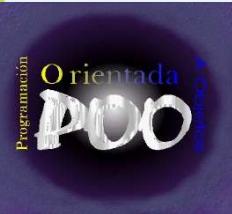
Metodología para el Desarrollo de Programas Orientado a Objetos

Objetivo General

Implementar soluciones con programación para problemas planteados aplicando la metodología para el desarrollo de programas orientados a objetos y la arquitectura MVC.

Objetivos Específicos

- Identificar los elementos que componen el patrón de diseño MVC (modelo, vista y controlador).
- Analizar el planteamiento de un problema y su diseño correspondiente con metodología MVC.
- Programar aplicaciones siguiendo un diseño con arquitectura MVC.
- Identificar en el diagrama de clases de un problema planteado los diversos componentes de la programación respectiva.
- Aplicar la arquitectura MVC usando estructuras de datos complejas.



Patrón de Diseño MVC



Contenido

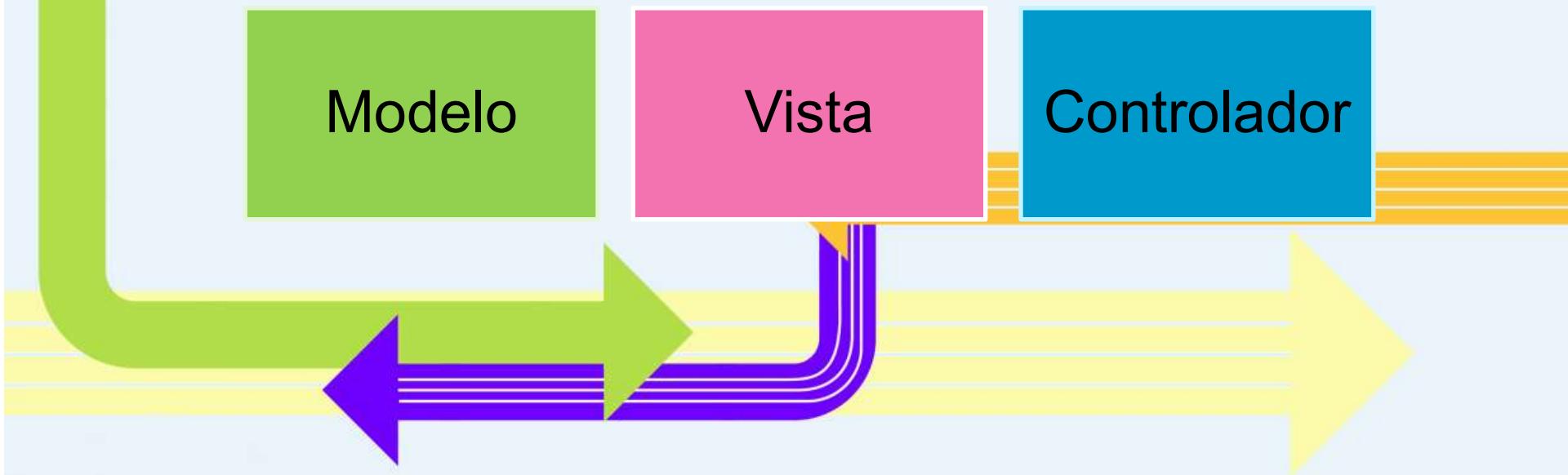
- Definición de Patrón MVC
- Componentes del Patrón MVC
- Ciclo de funcionamiento
- Análisis, Diseño e Implementación C++ usando el Patrón MVC
- Ventajas y Desventajas del Patrón MVC



Patrón de Diseño MVC

Definición

Es un patrón de diseño que ayuda a los programadores a organizar el código de una aplicación en tres partes principales, cada una con un trabajo específico.



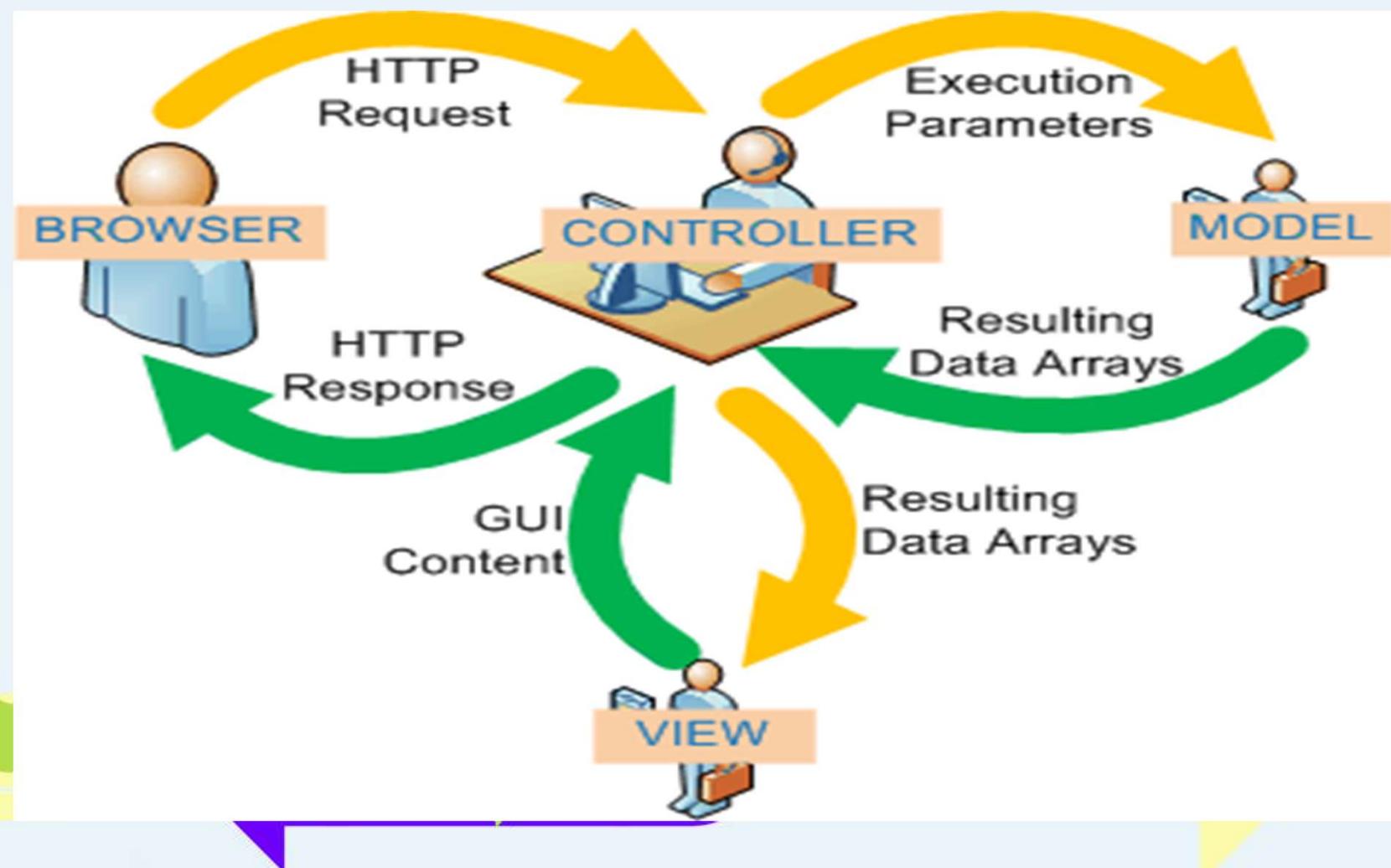
Patrón de Diseño MVC

Componentes:



Patrón de Diseño MVC

Ciclo de funcionamiento



Patrón de Diseño MVC



Conceptos clave desglosados

- **Patrón de arquitectura de software (patrón de diseño):** Imagina una receta de cocina para un pastel. La receta (el patrón) no es el pastel en sí, sino una guía o un conjunto de instrucciones bien probadas que te dicen cómo hacerlo de la mejor manera. De igual forma, un patrón de arquitectura es una "receta" probada para construir una aplicación de software, asegurando que sea robusta, escalable y mantenible.
- **Aplicación y software:** Son básicamente sinónimos en este contexto. Una aplicación o software es un programa de computadora diseñado para realizar una tarea específica. Un ejemplo sencillo es una aplicación de calculadora. El programa en sí es el software, y cuando lo ejecutas, se convierte en la aplicación que puedes usar.
- **Datos y lógica del negocio:**
 - **Los datos son la información pura.** En una aplicación de un blog, los datos serían el título de la publicación, el contenido del texto, el nombre del autor y la fecha.
 - **La lógica del negocio son las reglas que actúan sobre esos datos.** Siguiendo con el ejemplo del blog, una regla de negocio podría ser "solo el autor de la publicación puede editarla" o "el título no puede tener más de 100 caracteres". La lógica del negocio no se preocupa por cómo se ven los datos, sino por las reglas que los rigen.

Patrón de Diseño MVC



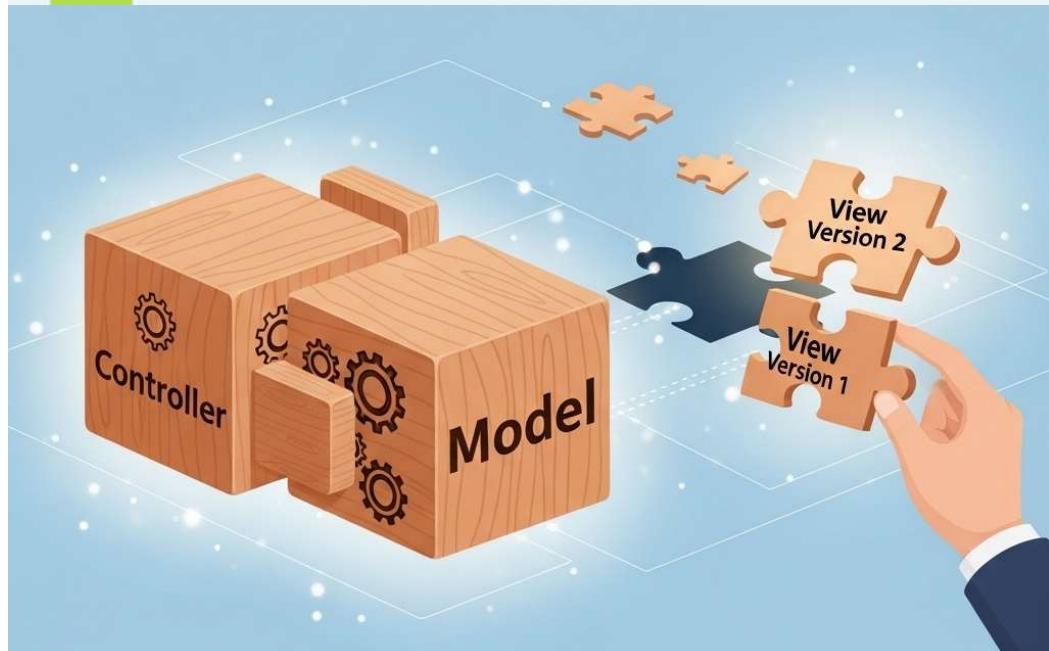
Conceptos clave desglosados

- **Interfaz de usuario:** Es la parte de la aplicación con la que el usuario interactúa. Es lo que ves y tocas en la pantalla: botones, menús, campos de texto, imágenes, etc. En un blog, la interfaz de usuario es la página web que muestra el contenido de las publicaciones, los botones para comentar y el formulario para crear una nueva entrada.
- **Componentes:** Piensa en una bicicleta. Un componente es una de sus partes, como el manubrio, el asiento o la rueda. Cada parte tiene una función específica y se puede reemplazar sin necesidad de cambiar toda la bicicleta. En el desarrollo de software, los componentes son las piezas de código que cumplen una función específica, como el Modelo, la Vista y el Controlador.

Patrón MVC: Metáfora

¡A toda la aplicación le cambiamos la vista, y tenemos una nueva GUI!

En esta imagen, el Modelo y el Controlador son el corazón y el cerebro de la aplicación. Son el motor interno, los engranajes que procesan la información y la lógica del negocio sin que nadie los vea. Son la maquinaria robusta y confiable que no necesita cambiar para seguir haciendo su trabajo.



La Vista, en cambio, es la máscara o el traje de un actor.

- Es lo que el público ve, el vestuario que se adapta a la escena.
- Gracias a la arquitectura MVC, podemos:
- Quitar la máscara de la "Vista Versión 1" (la primera interfaz de usuario)
- Y colocar una completamente diferente, la "Vista Versión 2", con tan solo un clic.
- El cerebro y el corazón de la aplicación nunca se enteran del cambio de vestuario, porque su trabajo es solo procesar y gestionar la información.
- El resultado es una aplicación con un nuevo rostro, una interfaz rediseñada, pero con la misma lógica y el mismo funcionamiento interno, sólido e inalterable.



Patrón MVC: Metáfora

Piensa en el núcleo de un sistema operativo como la planta de energía de una ciudad

Esta planta genera la electricidad, gestiona la red, asegura que todo funcione y distribuye la energía a cada edificio. La planta de energía (el Modelo y el Controlador) es un sistema complejo y fundamental que no cambia a menudo. Su trabajo es que todo funcione de manera eficiente y segura.



El diseño de los edificios y los dispositivos en los hogares de la ciudad (la Vista) es lo que la gente ve y con lo que interactúa.

Los edificios pueden tener diseños antiguos, modernos, minimalistas o futuristas. La gente puede usar lámparas tradicionales o sistemas de iluminación inteligente.

Así como una ciudad puede modernizar todos sus edificios y su iluminación (pasar de Windows 10 a Windows 11) sin tener que rediseñar por completo la planta de energía, **la arquitectura MVC permite que el aspecto visual de una aplicación evolucione sin tocar la lógica de negocio y los datos que la sustentan**. El sistema subyacente sigue siendo el mismo; solo cambia la forma en que se presenta al usuario.

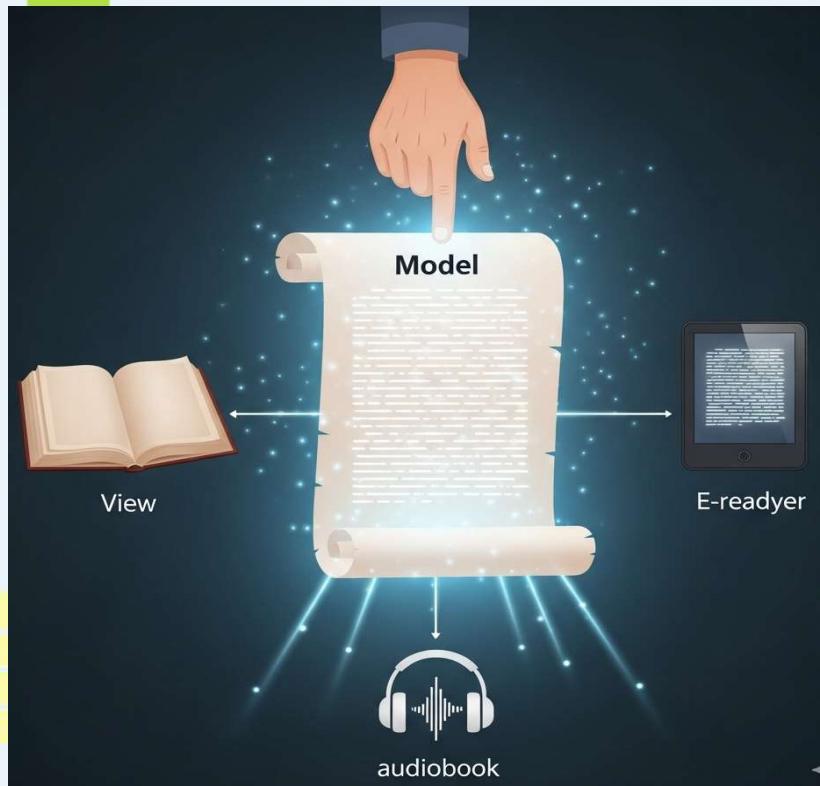


Patrón MVC: Metáfora



Pensemos en la analogía de un libro.

El Modelo es el contenido del libro: la historia, los personajes, los diálogos, los datos y las reglas del universo de la novela. El contenido existe y tiene sentido por sí mismo, independientemente de cómo se presente.



El Controlador es el lector. Tú, como lector, interactúas con el libro. Decides cuándo pasar de página (una acción), qué capítulo leer a continuación, o si quieras volver a un párrafo anterior. Tú, el lector, no cambias el contenido del libro, pero diriges cómo se accede a él.

La Vista es la edición física del libro: el tipo de papel, el tamaño de la letra, la portada, el diseño de la contraportada y la encuadernación. El mismo contenido (la historia) puede publicarse en una edición de tapa dura, un libro de bolsillo, un audiolibro o un libro electrónico. La historia sigue siendo la misma, pero la forma en que se te presenta (la Vista) es completamente diferente.

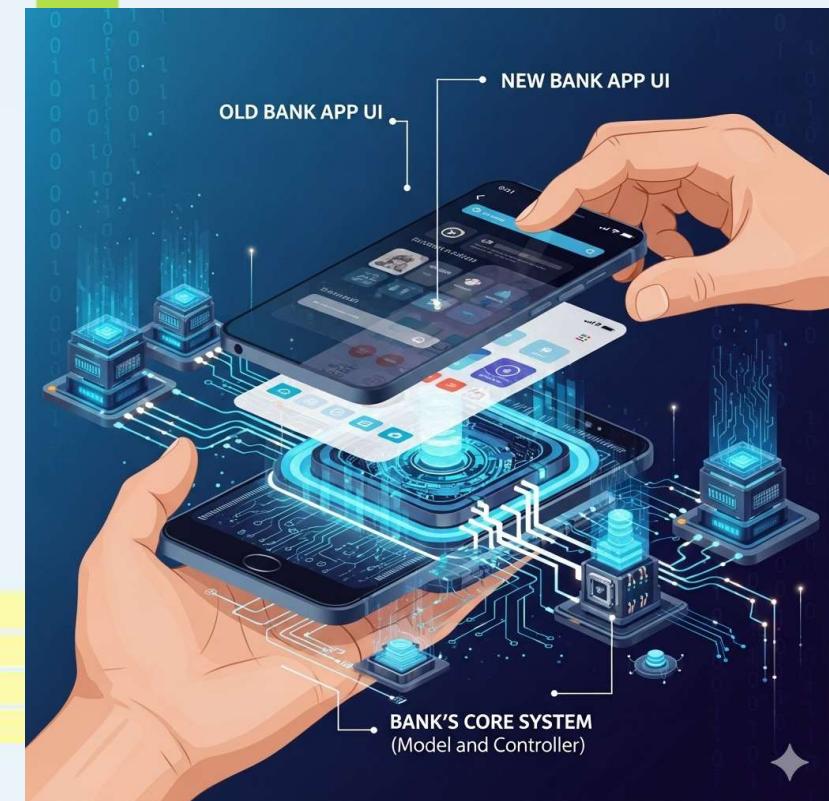
Esta analogía resalta cómo el Modelo (el contenido) y el Controlador (el lector) permanecen intactos, mientras que la Vista (la edición) puede ser cambiada, adaptada o actualizada para diferentes propósitos o tecnologías.

Patrón MVC: Metáfora



En esta metáfora, la seguridad es la clave.

Piensa en el corazón de un banco: las cámaras acorazadas, los servidores que gestionan las transacciones, la base de datos de los clientes y los complejos sistemas de seguridad. Esa es la maquinaria fundamental, sólida y confiable, que no cambia a menos que sea absolutamente necesario.



El Modelo y el Controlador son esa maquinaria.

Son el sistema central del banco, encargado de gestionar las cuentas, las transferencias, los saldos y las reglas de negocio. Esta infraestructura digital es la misma, ya sea que la uses desde una computadora, un cajero automático o un celular.

La aplicación móvil, en cambio, es solo la fachada del banco; la cara que te muestra.

Puede ser brillante, con nuevos colores, iconos y menús más modernos, pero por dentro sigue conectada al mismo sistema central.

La Vista es esa fachada de la aplicación móvil. Cuando el banco lanza una actualización de la app, simplemente está cambiando la apariencia (la Vista), sin tener que tocar ni una sola línea de código del motor que gestiona tus transacciones (el Modelo y el Controlador).

Patrón MVC: Análisis



Enunciado: **PEAJE CARDENAL**

El peaje cardenal cobra 3 tipos de tarifas (1, 2 ó 3) de vehículos: 1=bs.10 particulares, 2=bs.20 transporte, 3=bs.50 carga. Conociendo la placa y el tipo de vehículo se desea informar, por cada vehículo: su tarifa a pagar y por el peaje: el porcentaje de cada tipo de vehículo y el monto total para el municipio, sabiendo que es el 40% de todo lo cobrado.

Al ser consultada por la forma como desean que se presente la salida, el peaje cardenal suministra el siguiente formato, sobre la base de los siguientes datos:

(placa, tipo vehículo): (KBS11E, 1) (XXES12, 2) (YXZQE1, 3) (KBS23E, 1)

Título del ejercicio

PEAJE CARDENAL

Contexto

Un peaje procesa vehículos para calcular la tarifa de cada uno y generar estadísticas sobre los tipos de vehículos y los ingresos para el municipio.

Requerimientos

- R1. Por cada vehículo: su tarifa a pagar
- R2.a. **Por el peaje:** porcentaje de cada tipo de vehículo
- R2.b. **Por el peaje:** monto total para el municipio (40% de todo lo cobrado)

4. Ejemplo

placa	tipo	tarifaApagar()
KBS11E	1	10
XXES12	2	20
YXZQE1	3	50
KBS23E	1	10

Porcentajes:

tipo1=50%, tipo2=25%, tipo3=25%

Monto municipio:

$$(10+20+50+10)*0.4 = 36 \text{ Bs}$$

Patrón MVC: Análisis

Abstracción de lógica para cada requerimiento

R1. **tarifaAPagar** =

- 10 si tipo = 1
- 20 si tipo = 2
- 50 si tipo = 3

R2.a. **Porcentaje por tipo**

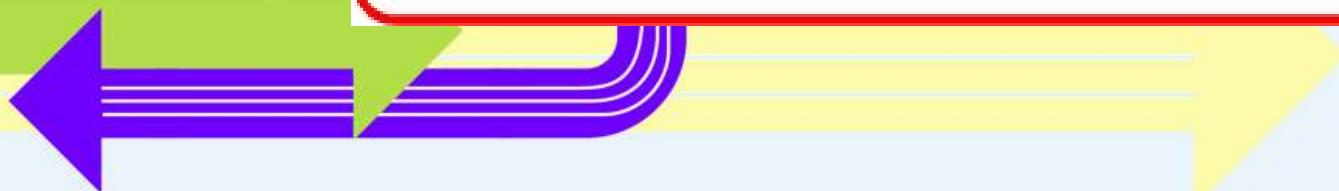
$$= (\text{cantidad de vehículos del tipo} / \text{total de vehículos}) * 100$$

R2.b. **Monto municipio**

$$= (\text{suma de todas las tarifas}) * 0.4$$

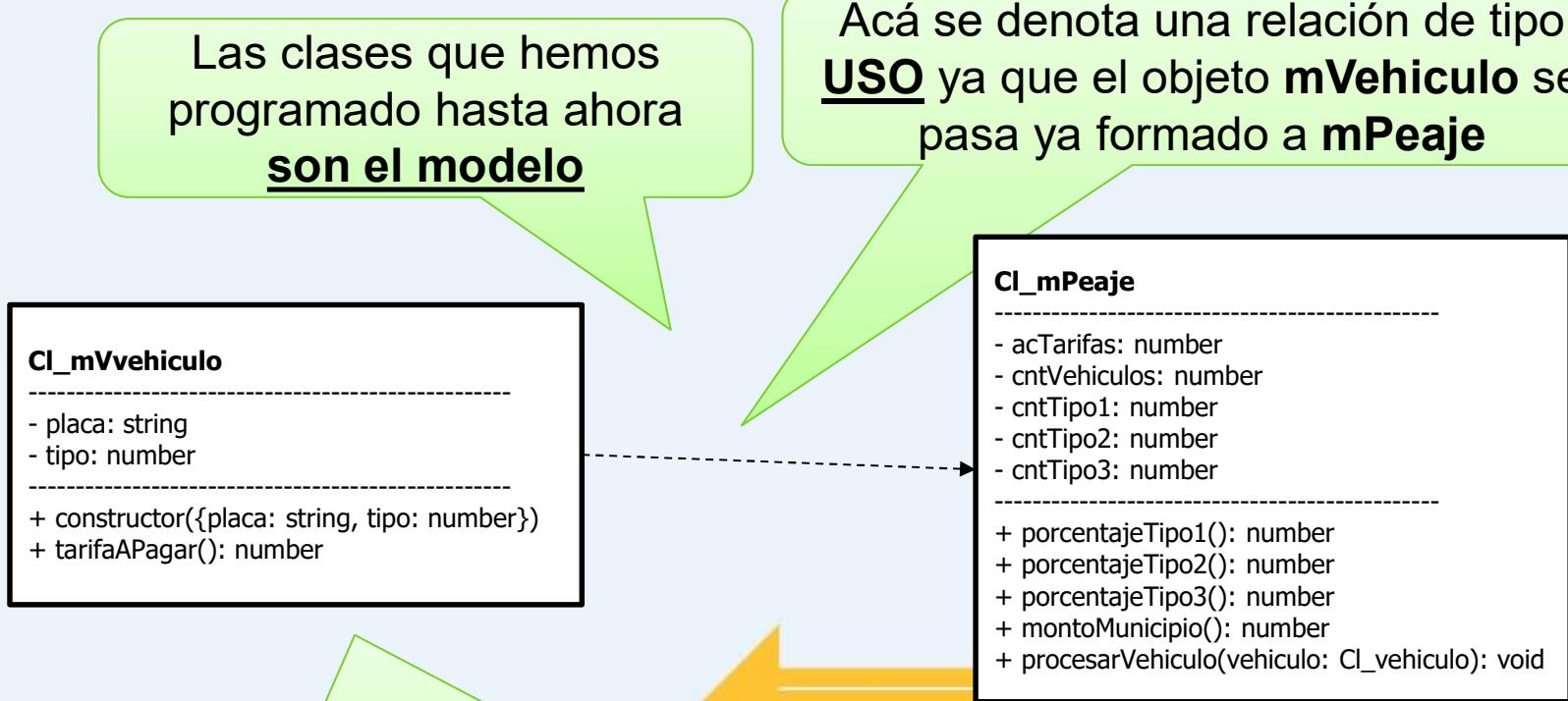
Atributos necesarios en clase mayor

- **acTarifas**: number (acumulador de todas las tarifas)
- **cntVehiculos**: number (contador total de vehículos)
- **cntTipo1**: number (contador de vehículos tipo 1)
- **cntTipo2**: number (contador de vehículos tipo 2)
- **cntTipo3**: number (contador de vehículos tipo 3)



Patrón MVC: Diseño

Diagrama de Clases: **Modelo**



OBSERVA la nueva forma de inicializar un objeto: el constructor recibe otro objeto (datos entre llaves). Esto nos dará una serie de ventajas que iremos aprendiendo poco a poco.

Otro tipo de relación: **AGREGACIÓN...**
Se pasan los datos para que el otro objeto cree (agregue) al menor

Patrón MVC: Diseño

- Diseño de la VISTA
 - La interfaz la haremos lo más elemental posible.
 - Habrá que dibujarla con HTML.
 - Los estilos de la salida se perfilan en **styles.css**
 - Usaremos **inputs** para leer los datos (placa y tipo).
 - Un botón **Procesar** enviará estos datos al Peaje.
 - La salida se mostrará concatenada en un **div** del HTML
 - Mas o menos así:

Datos del vehículo

Placa

Tipo

Procesar

Información del Peaje

KBS11E: pagar 10

Porcentajes: Tipo1=100%, Tipo2=0%, Tipo3=0%

Monto municipio: 4 Bs

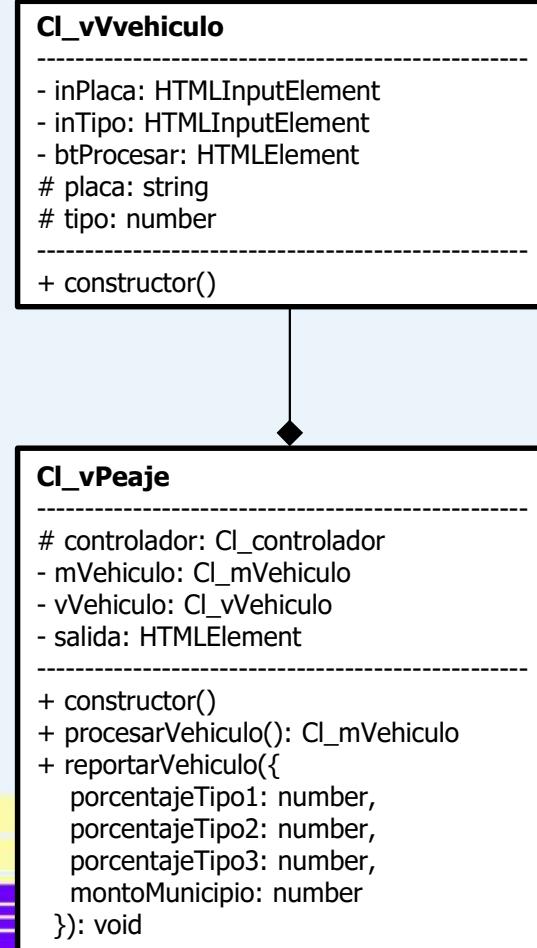
Patrón MVC: Diseño

Diagrama de Clases: Vista

La vista se encarga de “interactuar” con el usuario: leer datos y mostrar salidas

Ya que la clase menor tiene 1 requerimiento, entonces amerita el método reportar.
Observa que los datos a reportar son lo que me piden

Similar la vista para la clase mayor: el reportar recibe los datos a mostrar.



Patrón MVC: Controlador

Diagrama de Clases: **Controlador**

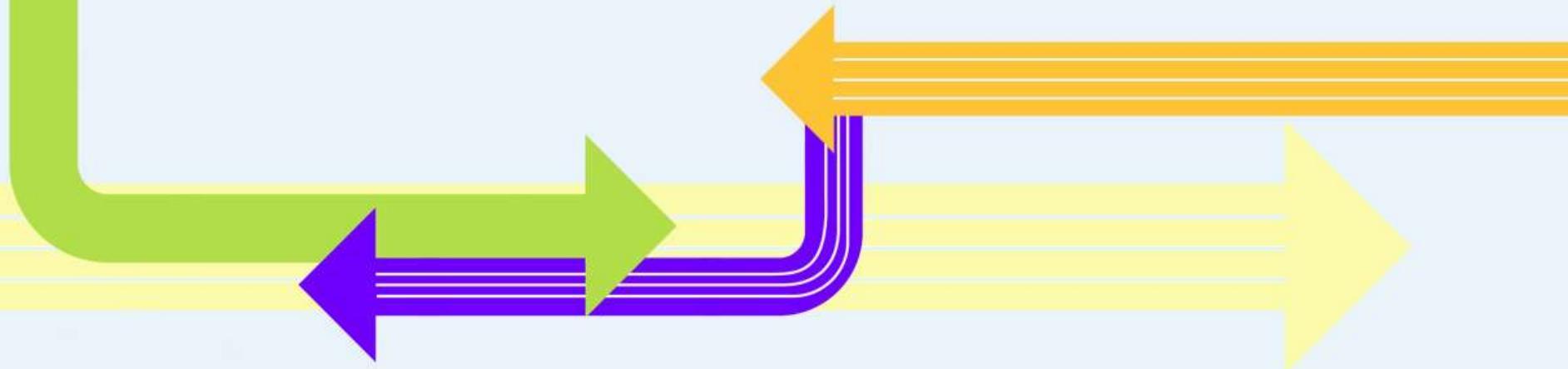
El controlador tiene como atributos las clases del modelo y de la vista (las necesarias).

El procesarVehiculo será el método que pone a funcionar tu programa: un evento click hasta que se termine con la entrada de datos.

Cl_controlador

- modelo: Cl_mPeaje
- vista: Cl_vPeaje

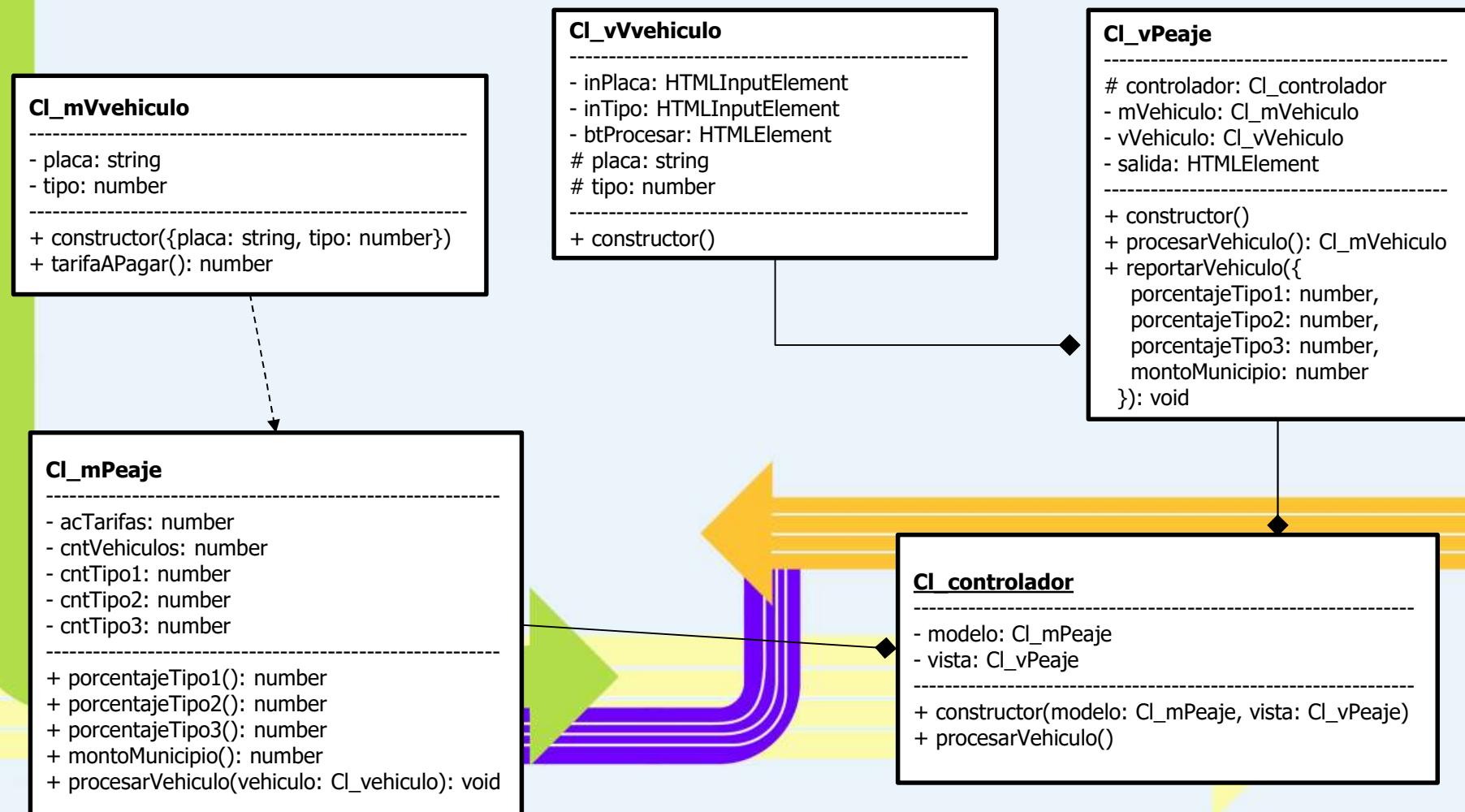
- + constructor(modelo: Cl_mPeaje, vista: Cl_vPeaje)
- + procesarVehiculo()



Solución 1: procesamiento básico

Patrón MVC: Diseño

Diagrama de Clases Completo con MVC



Cl_mVvehiculo

- placa: string
- tipo: number

+ constructor({placa: string, tipo: number})
+ tarifaAPagar(): number

Cl_mPeaje

- acTarifas: number
- cntVehiculos: number
- cntTipo1: number
- cntTipo2: number
- cntTipo3: number

+ porcentajeTipo1(): number
+ porcentajeTipo2(): number
+ porcentajeTipo3(): number
+ montoMunicipio(): number
+ procesarVehiculo(vehiculo: Cl_vVehiculo): void

Cl_vVvehiculo

- inPlaca: HTMLInputElement
- inTipo: HTMLInputElement
- btProcesar: HTMLElement
placa: string
tipo: number

+ constructor()

Cl_vPeaje

controlador: Cl_controlador
- mVehiculo: Cl_mVehiculo
- vVehiculo: Cl_vVehiculo
- salida: HTMLElement

+ constructor()
+ procesarVehiculo(): Cl_mVehiculo
+ reportarVehiculo({
porcentajeTipo1: number,
porcentajeTipo2: number,
porcentajeTipo3: number,
montoMunicipio: number
}): void

Cl_controlador

- modelo: Cl_mPeaje
- vista: Cl_vPeaje

+ constructor(modelo: Cl_mPeaje, vista: Cl_vPeaje)
+ procesarVehiculo()

REGLA IMPORTANTE:

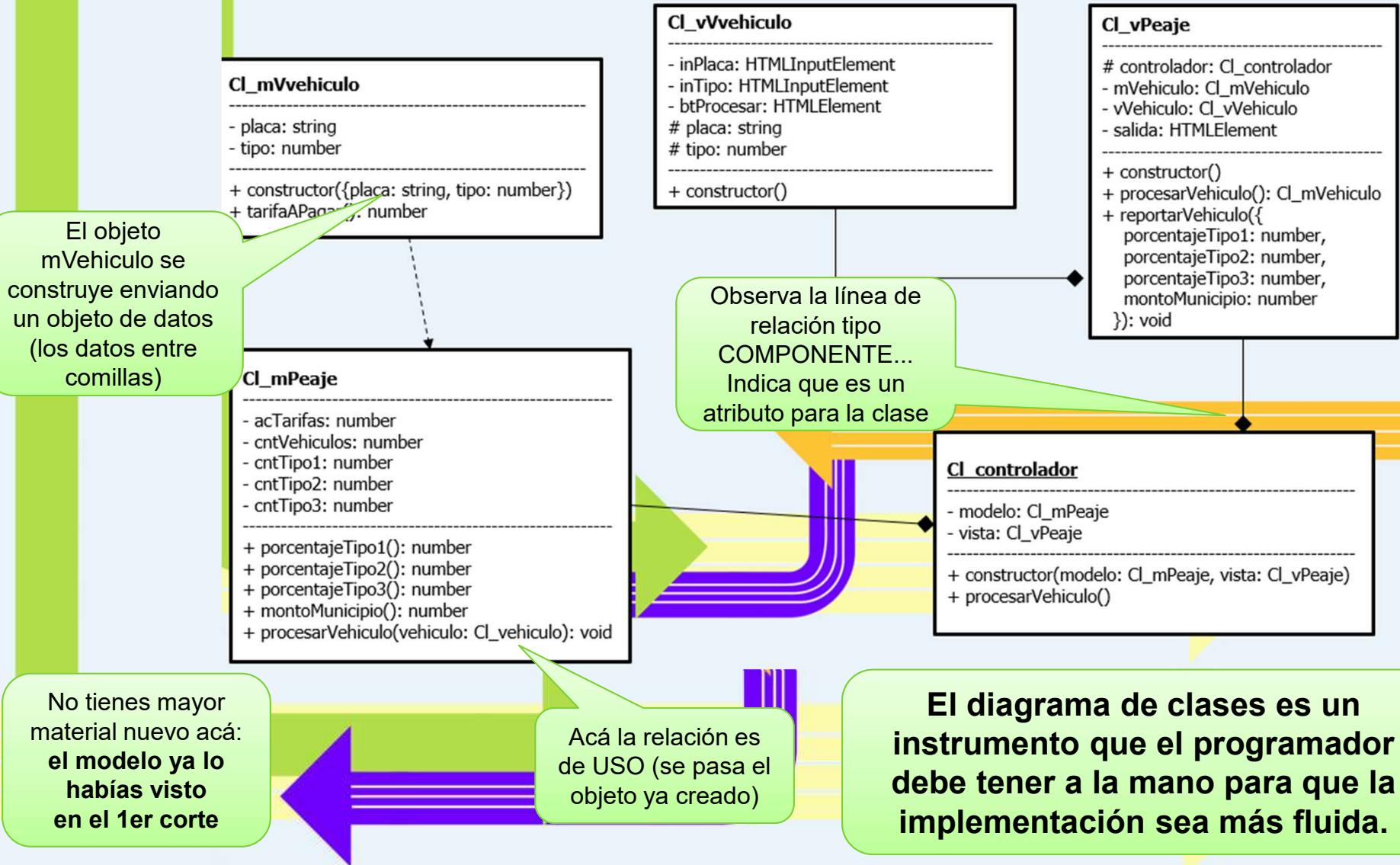
Diseña cuidadosamente tu diagrama de clases, será tu guía durante la implementación (es decir, durante la programación)

La clase mPeaje **USA** la clase mVehiculo, mientras que el controlador tiene como **COMPONENTES** las clases principales

Solución 1: procesamiento básico

Patrón MVC: Diseño

Los métodos se llaman TAL como lo dice el planteamiento, no importa si es largo el identificador



Solución 1: procesamiento básico

Patrón MVC: Modelo en JavaScript

El MODELO fue lo aprendimos en las 1eras clases:
ES EXACTO IGUAL

```
export default class Cl_mVehiculo {  
    private _placa: string = "";  
    private _tipo: number = 1;  
    constructor({ placa = "", tipo = 1 }) {  
        this.placa = placa;  
        this.tipo = tipo;  
    }  
    public tarifaAPagar(): number {  
        if (this.tipo === 1) return 10;  
        if (this.tipo === 2) return 20;  
        if (this.tipo === 3) return 50;  
        return 0;  
    }  
    public set placa(placa: string) {  
        this._placa = placa;  
    }  
    public set tipo(tipo: number) {  
        this._tipo = tipo;  
    }  
    public get placa(): string {  
        return this._placa;  
    }  
    public get tipo(): number {  
        return this._tipo;  
    }  
}
```

básico en JavaScript

```
import Cl_mVehiculo from "./Cl_mVehiculo.js";
export default class Cl_mPeaje {
    private acTarifas: number;
    private cntVehiculos: number;
    private cntTipo1: number;
    private cntTipo2: number;
    private cntTipo3: number;
    constructor() {
        this.acTarifas = 0;
        this.cntVehiculos = 0;
        this.cntTipo1 = 0;
        this.cntTipo2 = 0;
        this.cntTipo3 = 0;
    }
    public procesarVehiculo(vehiculo: Cl_mVehiculo): void {
        const tarifa = vehiculo.tarifaAPagar();
        this.acTarifas += tarifa;
        this.cntVehiculos++;
        if (vehiculo.tipo === 1) this.cntTipo1++;
        if (vehiculo["tipo"] === 2) this.cntTipo2++;
        if (vehiculo.tipo === 3) this.cntTipo3++;
    }
    public porcentajeTipo1(): number {
        if (this.cntVehiculos === 0) return 0;
        return (this.cntTipo1 / this.cntVehiculos) * 100;
    }
    public porcentajeTipo2(): number {
        if (this.cntVehiculos === 0) return 0;
        return (this.cntTipo2 / this.cntVehiculos) * 100;
    }
    public porcentajeTipo3(): number {
        if (this.cntVehiculos === 0) return 0;
        return (this.cntTipo3 / this.cntVehiculos) * 100;
    }
    public montoMunicipio(): number {
        return this.acTarifas * 0.4;
    }
}
```

El MODELO fue lo aprendimos en las 1eras clases:
ES EXACTO IGUAL

```
import Cl_controlador from "./Cl_controlador.js";
export default class Cl_vVehiculo {
    public controlador: Cl_controlador | null = null;
    private inPlaca: HTMLInputElement;
    private inTipo: HTMLInputElement;
    public btProcesar: HTMLElement;

    constructor() {
        this.inPlaca = document.getElementById(
            "vehiculoForm_inPlaca"
        ) as HTMLInputElement;
        this.inTipo = document.getElementById(
            "vehiculoForm_inTipo"
        ) as HTMLInputElement;
        this.btProcesar = document.getElementById(
            "vehiculoForm_btProcesar"
        ) as HTMLElement;
        // Validar que los elementos existen
        if (!this.inPlaca || !this.inTipo || !this.btProcesar)
            throw new Error("Elementos del DOM no encontrados");
        this.btProcesar.onclick = () => {
            if (!this.controlador) throw new Error("No hay controlador");
            this.controlador.procesarVehiculo();
        };
    }
    get placa(): string {
        if (!this.inPlaca) throw new Error("Elemento inPlaca no encontrado");
        return this.inPlaca.value;
    }
    get tipo(): number {
        if (!this.inTipo) throw new Error("Elemento inTipo no encontrado");
        return +this.inTipo.value;
    }
}
```

Se crean los atributos desde el documento HTML

Verificar que existen los elementos en el HTML

Se le asigna la “acción” al hacer click en el botón “procesar”

La vista vVehiculo se encarga de leer datos

Es la INTERFAZ CON EL USUARIO

o GUI (Graphic User Interface)

Los atributos protegidos retornan el valor formateado.
Protegidos de sólo lectura (get)

```

import Cl_vVehiculo from "./Cl_vVehiculo.js";
import Cl_mVehiculo from "./Cl_mVehiculo.js";
import Cl_controlador from "./Cl_controlador.js";
// Define una interfaz para el controlador
export default class Cl_vPeaje {
    private _controlador: Cl_controlador | null = null;
    private salida: HTMLElement;
    private vVehiculo: Cl_vVehiculo;
    private mVehiculo: Cl_mVehiculo | null = null;
    constructor() {
        this.salida = document.getElementById("mainForm_salida") as HTMLElement;
        if (!this.salida) throw new Error("Elemento salida no encontrado");
        this.vVehiculo = new Cl_vVehiculo();
    }
    set controlador(controlador: Cl_controlador | null) {
        this._controlador = controlador;
        this.vVehiculo.controlador = controlador;
    }
    get controlador(): Cl_controlador | null {
        return this._controlador;
    }
    procesarVehiculo(): Cl_mVehiculo {
        this.mVehiculo = new Cl_mVehiculo({
            placa: this.vVehiculo.placa,
            tipo: this.vVehiculo.tipo,
        });
        return this.mVehiculo;
    }
    reportarVehiculo({
        porcentajeTipo1 = 0,
        porcentajeTipo2 = 0,
        porcentajeTipo3 = 0,
        montoMunicipio = 0,
    }): void {
        if (!this.mVehiculo) throw new Error("No hay artículo procesado");
        this.salida.innerHTML += `<br><br>${{
            this.mVehiculo.placa
        }}: pagar ${this.mVehiculo.tarifaAPagar()}
<br><br>Porcentajes: Tipo1=${porcentajeTipo1}%, Tipo2=${porcentajeTipo2}%, Tipo3=${porcentajeTipo3}%
<br>Monto municipio: ${montoMunicipio} Bs`;
    }
}

```

Todos los atributos deben inicializarse en el constructor, o acá; si es acá, el compilador lo pondrá **en el respectivo .js** dentro del constructor.

El constructor debe inicializar todos los atributos.
Si no existe el elemento HTML se dispara un error en la consola.

La vista principal (la que opera el controlador) se encarga de difundir el acceso al controlador entre el resto de las vistas.

El objetivo acá es retornar un objeto menor desde la GUI; lo recibirá el controlador y lo pasa al modelo.

CADA DISEÑO DE GUI tiene completa consideración de los objetos que están en el documento HTML; es decir, **nada es casual ni estándar**, todo depende de la lógica que se le da a cada vista.

La vista vPeaje se encarga de acceder la vista menor y reportar a la clase mayor (en este ejemplo no hay nada que leer de la clase mayor)

Solución 1: procesamiento básico

Patrón MVC: Controlador en JavaScript

```
import Cl_mPeaje from "./Cl_mPeaje.js";
import Cl_vPeaje from "./Cl_vPeaje.js";
export default class Cl_controlador {
    public modelo: Cl_mPeaje;
    public vista: Cl_vPeaje;
    constructor(modelo: Cl_mPeaje, vista: Cl_vPeaje) {
        this.modelo = modelo;
        this.vista = vista;
    }
    procesarVehiculo() {
        this.modelo.procesarVehiculo(this.vista.procesarVehiculo());
        this.vista.reportarVehiculo({
            porcentajeTipo1: this.modelo.porcentajeTipo1(),
            porcentajeTipo2: this.modelo.porcentajeTipo2(),
            porcentajeTipo3: this.modelo.porcentajeTipo3(),
            montoMunicipio: this.modelo.montoMunicipio(),
        });
    }
}
```

Cada vez que procesa un objeto menor, se reporta la salida completa.

Esta organización de los objetos (patrón MVC) te permite cambiar muy fácilmente la VISTA

El controlador recibe las instancias y crea atributos de las clases (en el constructor).

Recuerda que el constructor se ejecuta al crear la instancia de cada objeto

Luego espera por los eventos del botón agregar para procesar los objetos de la clase menor.

Existen diversas maneras de plantear la vista al usuario final.

Piensa en una APP carrito de compras... Debes haber visto docenas de maneras de presentarlo

Solución 1: procesamiento básico

Patrón MVC: Programa Principal

```
import Cl_vPeaje from "./Cl_vPeaje.js";
import Cl_mPeaje from "./Cl_mPeaje.js";
import Cl_controlador from "./Cl_controlador.js";
export default class Cl_principal {
    constructor() {
        let vista = new Cl_vPeaje();
        let modelo = new Cl_mPeaje();
        let controlador = new Cl_controlador(modelo, vista);
        vista.controlador = controlador;
    }
}
```

El programa principal “dispara” el procesamiento por parte del controlador.

El controlador sirve de puente entre la vista y el modelo.

Principal

Con solo cambiar las vistas entonces podrá tenerse varias versiones de la misma aplicación.
(se ven diferente pero funcionan igual)

Cambiar las vistas = usar otro código que maneja una GUI diferente

Solución 1: procesamiento básico

Archivo HTML

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Peaje Cardenal</title>
    <link rel="stylesheet" href="styles.css" />
</head>
<body>
    <section id="vehiculoForm">
        <h3>Datos del vehículo</h3>
        <input type="text" id="vehiculoForm_inPlaca" placeholder="placa del vehículo"></input>
        <input type="text" id="vehiculoForm_inTipo" placeholder="Tipo (1, 2, 3)"></input>
        <button id="vehiculoForm_btProcesar">Procesar</button>
    </section>
    <section id="mainForm">
        <h3>Info del Peaje</h3>
        <div id="mainForm_salida"></div>
    </section>
    <script type="module">
        import Cl_principal from "./dist/principal.js";
        new Cl_principal();
    </script>
</body>
</html>
```

El index.html, se encarga de dibujar los objetos en pantalla.

- **El tipo de elemento** se le llama TAG, hay uno de apertura y otro de cierre.
- **Elemento HTML section:** para dividir en secciones lógicas el documento HTML.
- **Elemento HTML input:** para hacer lecturas.
- **En el tag de apertura** se colocan las propiedades HTML.
- **El tag link** en este caso enlaza el archivo de estilos

- **Elemento HTML button:** dibuja un botón en el documento.
- **Elemento HTML div:** un lugar donde colocar texto.
- **El ID** se usa para acceder al elemento desde JavaScript.
- Observe que los elementos pueden anidarse.

- **Elemento HTML script:** permite colocar código JavaScript, o leerlo desde un archivo.
- **En este caso** el código se importa desde principal.js, y se crea la instancia de la clase **principal**, y se ejecuta el constructor.
- Entonces **se activan los onclick** y el programa comienza a funcionar.

Solución 1: procesamiento básico

Salida

Datos del vehículo

KBS23E
1
Procesar

Info del Peaje

KBS11E: pagar 10

Porcentajes: Tipo1=100%, Tipo2=0%, Tipo3=0%
Monto municipio: 4 Bs

XXES12: pagar 20

Porcentajes: Tipo1=50%, Tipo2=50%, Tipo3=0%
Monto municipio: 12 Bs

YXZQE1: pagar 50

Porcentajes: Tipo1=33.33333333333333%, Tipo2=33.33333333333333%, Tipo3=33.33333333333333%
Monto municipio: 32 Bs

KBS23E: pagar 10

Porcentajes: Tipo1=50%, Tipo2=25%, Tipo3=25%
Monto municipio: 36 Bs

La salida para la
solución 1:
RUDIMENTARIA

