INFORME DE DESARROLLO E IMPLEMENTACIÓN DE UN SISTEMA DE GESTIÓN PARA UN PARQUEADERO

Katherine Cicero 192460

Jhoan Molina 192490

Abel Bonilla 192488

Adrián Rincón 192531

PRESENTADO A:

Ing. Jesús Eduardo Guerrero

UNIVERSIDAD FRANCISCO DE PAULA SANTANDER SECCIONAL OCAÑA

FACULTAD DE INGENIERÍAS – INGENIERÍA DE SISTEMAS

OCAÑA

2025

FECHA DE ENTREGA: 13/06/2025

ÍNDICE O TABLA DE CONTENIDO

OBJETIVO GENERAL

Desarrollar un sistema de gestión inteligente para parqueaderos que permita automatizar y optimizar los procesos de registro, control de entrada y salida de vehículos, cálculo de tarifas y generación de reportes, con el fin de reducir errores humanos, disminuir tiempos operativos y facilitar la administración del parqueadero tanto en contextos académicos como urbanos.

OBJETIVOS ESPECÍFICOS

1) Diseñar una arquitectura orientada a objetos que represente adecuadamente las entidades del parqueadero, incluyendo tipos de vehículos, tarifas, cupos y registros.
2) Implementar un módulo de registro que permita controlar con precisión la entrada y salida de vehículos, evitando duplicidad de placas y asegurando la disponibilidad de cupos.
3) Desarrollar un sistema de cálculo tarifario flexible que contemple tarifas diferenciadas por tipo de vehículo, recargos nocturnos y descuentos por mensualidades.
4) Proveer una interfaz de consola amigable para el encargado del parqueadero, que permita ingresar datos, buscar vehículos por placa y visualizar el estado actual del parqueadero.
5) Generar reportes financieros automáticos que muestren el total recaudado por categoría de vehículo, así como el total general, para facilitar el análisis y la toma de decisiones.
6) Establecer una base funcional para futuras mejoras tecnológicas, como la incorporación de reconocimiento automático de placas, pagos electrónicos o integración con sensores físicos.
7) Garantizar la integridad y trazabilidad de la información almacenada, permitiendo un seguimiento preciso de cada vehículo ingresado y retirado, con el fin de mejorar la seguridad y el control logístico del parqueadero.

MARCO TEÓRICO

El desarrollo del sistema ParkHub se fundamenta en los principios de la Programación Orientada a Objetos (POO), así como en el uso eficiente de estructuras de datos, control de flujo, y buenas prácticas de diseño de software.

- *Clases y Objetos*

Se crean clases como Vehiculo, Registro, Tarifas, Parqueadero y ParqueaderoParkhub, que se extrañen de entidades del mundo real (vehículos, tarifas, operaciones, sistema).

- *Herencia*

Carro, Moto y Bicicleta heredan de Vehiculo utilizando extends, lo que permite reutilizar atributos como placa y comportamientos comunes en una estructura jerárquica.

- *Encapsulamiento*

Uso de modificadores como ***private*** y ***protected*** para restringir el acceso directo a los atributos, y proporcionar métodos públicos como ***getPlaca***() o ***getTipo***() para acceder a ellos de forma controlada.

- *Polimorfismo*

Uso de la clase base Vehiculo para manejar diferentes tipos de vehículos.

ESTRUCTURAS DE DATOS

- *ArrayList*

Para almacenar los registros activos de los vehículos dentro del parqueadero

- *Arreglos*

Se utiliza un arreglo para llevar el control del dinero recaudado por tipo de vehículo

Control de Flujo

- *Condicionales (if, else, if-else)*

Se utilizan ampliamente para validar entradas del usuario, aplicar lógica de negocio como determinar si un vehículo tiene mensualidad, o si debe aplicarse un recargo nocturno. También se usan para diferenciar el tipo de vehículo (carro, moto, bicicleta) y calcular las tarifas correspondientes.

- *Bucles (while, for, switch)*

Se emplean para repetir operaciones del menú principal, recorrer listas de vehículos registrados y buscar información por placa.

1) while: Mantiene activo el menú principal del sistema mientras el usuario no elija la opción de salir.
2) for: Se utiliza para recorrer listas como vehiculos y mostrar información detallada.
3) switch: Organiza las opciones del menú principal del sistema.

ENTRADA Y SALIDA DE DATOS

- *Clase Scanner*

Se emplea para capturar la entrada del usuario por consola, como el tipo de vehículo, placa, o tipo de pago.

- *Validaciones*

Se validan los formatos y contenido de entradas mediante condiciones y manejo de excepciones. Se utiliza `try-catch` para validar que los datos sean numéricos cuando corresponde, y se controla que no haya entradas vacías o duplicadas:

MANEJO DE MÉTODOS

- *Métodos con parámetros y retorno*

Para operaciones como calcular el precio

- *Métodos void*

Ejecutan acciones como ingresar vehículos, mostrar reportes o listar el estado

ORGANIZACIÓN Y MODULARIDAD

1) El código está organizado por clases con funciones definidas como por ejemplo:

- Vehiculo → modelo base.
- Registro → entrada individual al parqueadero.
- Tarifas → lógica de precios.
- Parqueadero → lógica operativa principal.
- ParqueaderoParhub → interfaz con el usuario.

2) Se sigue el principio de responsabilidad única (SRP): cada clase y método se encarga solo de una función específica, facilitando el mantenimiento del sistema y la fluidez en el mismo a la hora de ejecutarse.

ACUMULADORES Y CONTADORES

- Se utilizan variables para contar vehículos activos y acumular ingresos según el tipo

```
dineroTipos[0] += precio; // Carros
dineroTipos[1] += precio; // Motos
dineroTipos[2] += precio; // Bicicletas
```

- Se controla el número de vehículos en tiempo real con la lista “vehículos”, y se verifica que no se sobrepase el cupo máximo de 200:

```
if (vehiculos.size() >= CUPOS_TOTALES) {
   System.out.println("Parqueadero lleno.");
}
```

INTRODUCCIÓN

La gestión manual de parqueaderos representa un desafío operativo en términos de eficiencia, control y reducción de errores. En muchos casos, los encargados deben registrar manualmente la entrada y salida de vehículos, calcular tarifas, controlar cupos disponibles y llevar reportes financieros, lo cual incrementa la posibilidad de equivocaciones, pérdida de información y retrasos en el servicio.

Con el fin de enfrentar estas problemáticas, se desarrolló ParkHub, un sistema de gestión de parqueaderos orientado a objetos que automatiza y optimiza los procesos fundamentales del servicio, permitiendo al encargado llevar un control preciso de los vehículos ingresados, aplicar tarifas dinámicas según el tipo de vehículo y el horario, así como generar reportes financieros confiables.

Aunque su implementación se enmarca inicialmente en un ejercicio práctico, ParkHub está diseñado con potencial para ser aplicado en entornos reales, como parqueaderos urbanos o institucionales, y cuenta con una estructura escalable para incorporar funciones avanzadas a futuro, como el reconocimiento automático de placas y pagos electrónicos.

EXPLICACIÓN DEL SISTEMA DE PARQUEADEROPARKHUB

De la línea 1 al 4 se importan librerías que sirven para:

- .LocalDateTime: Se usa para trabajar con fechas y horas, específicamente para registrar el momento en que un vehículo entra al parqueadero.
- .format.DateTimeFormatter: Se utiliza para definir cómo deben formatearse las fechas y horas para su visualización ("dd/MM HH:mm").
- .util.ArrayList;: Se usa para crear listas dinámicas de objetos, como la lista de vehículos actualmente parqueados.
- .util.Scanner;: Se utiliza para leer la entrada del usuario desde la consola (como cuando el usuario escribe una placa o una opción).

```
1  import java.time.LocalDateTime;
2  import java.time.format.DateTimeFormatter;
3  import java.util.ArrayList;
4  import java.util.Scanner;
5
```

- abstract class Vehiculo : Declara una clase **abstracta** llamada Vehiculo. Ser abstracta significa que no puedes crear un objeto directamente de Vehiculo; debes crear objetos de sus subclases (Carro, Moto, Bicicleta). Está diseñada para ser extendida.
- protected String tipo;: Declara una variable de tipo String **protected** llamada tipo para almacenar el tipo de vehículo (por ejemplo, "Carro", "Moto").
- protected significa que es accesible dentro de esta clase y sus subclases.
- protected String placa;: Declara una variable de tipo String **protected** llamada placa para almacenar la placa del vehículo.

```
6   // Clase padre - HERENCIA
7   abstract class Vehiculo {
8       protected String tipo;
9       protected String placa;
10
11      public Vehiculo(String tipo, String placa) {
12          this.tipo = tipo;
13          this.placa = placa;
14      }
```

- public String getTipo() return tipo; : Este es un método getter para tipo. Permite que otras partes del programa lean el tipo de un vehículo. Esto es un ejemplo de encapsulamiento, ya que proporciona un acceso controlado a los datos internos.

- public String getPlaca() { return placa; }: Un método getter para placa.

- @Override public String toString() : Este método sobrescribe el método toString() predeterminado heredado de la clase Objecto.

```
16      // ENCAPSULAMIENTO - getters
17      public String getTipo() { return tipo; }
18      public String getPlaca() { return placa; }
19
20      @Override
21      public String toString() {
22          return tipo + " - " + placa;
23      }
24  }
```

Estas clases **heredan** de la clase Vehículo, demostrando el concepto de **herencia**

- class Carro extends Vehículo: Declara una clase Carro que extends hereda de Vehículo.
- public Carro (String placa) super ("Carro", placa) Este es el constructor para Carro.

Cuando se crea un objeto Carro con una placa, este llama al constructor **super()** el constructor de la clase padre Vehiculo y le pasa "Carro" como el tipo y la placa proporcionada. La misma lógica se aplica a Moto y Bicicleta.

```
// Clases hijas
class Carro extends Vehiculo {
    public Carro(String placa) { super(tipo:"Carro", placa); }
}

class Moto extends Vehiculo {
    public Moto(String placa) { super(tipo:"Moto", placa); }
}

class Bicicleta extends Vehiculo {
    public Bicicleta(String placa) { super(tipo:"Bicicleta", placa); }
}
```

private final int PRECIO_CARRO = 2000; esto declara variables int **public** y **final** (constantes) para los precios base de cada tipo de vehículo y el recargo nocturno. private asegura que solo se pueda acceder a ellas dentro de esta clase, y final significa que sus valores no pueden cambiarse después de la inicialización.

```
39  // Clase para manejar precios
40  class Tarifas {
41      public final int PRECIO_CARRO = 2000;
42      public final int PRECIO_MOTO = 1000;
43      public final int PRECIO_BICI = 200;
44      public final int RECARGO_NOCTURNO = 1000;
```

public int calcularPrecio(String tipo, boolean esMensual, boolean esNocturno) Este es un método que calcula el precio del parqueo.

- Usa una sentencia **switch** para establecer el precio base según el tipo de vehículo.
- Luego, usa **condicionales** if else if para aplicar descuentos por pagos mensuales (25% de descuento sobre el precio diario por 30 días) o agregar un recargo nocturno

```
47    public int calcularPrecio(String tipo, boolean esMensual, boolean esNocturno) {
48        int precio = 0;
49
50        // SWITCH para precios base
51        switch (tipo) {
52            case "Carro": precio = PRECIO_CARRO; break;
53            case "Moto": precio = PRECIO_MOTO; break;
54            case "Bicicleta": precio = PRECIO_BICI; break;
55        }
56
57        // CONDICIONALES para modificar precio
58        if (esMensual) {
59            precio = (int)(precio * 30 * 0.75); // 25% descuento mensual
60        } else if (esNocturno) {
61            precio += RECARGO_NOCTURNO;
62        }
63
64        return precio;
65    }
```

- public boolean esNocturno() Este método verifica si la hora actual cae dentro del horario nocturno (8 PM a 6 AM).

- LocalDateTime.now().getHour() obtiene la hora actual. return hora >= 20 || hora < 6; devuelve true si la hora es 20 (8 PM) o más, o antes de las 6 AM.

```
67    public boolean esNocturno() {
68        int hora = LocalDateTime.now().getHour();
69        return hora >= 20 || hora < 6;
70    }
```

public void mostrarTarifas() Este método imprime las tarifas de parqueo diarias y las mensuales estimadas en la consola. Llama a calcularPrecio para obtener las tarifas mensuales.

```
    public void mostrarTarifas() {
        System.out.println(x:"\n===  TARIFAS ===");
        System.out.println(x:"DIARIAS:");
        System.out.println("• Carro: $" + PRECIO_CARRO);
        System.out.println("• Moto: $" + PRECIO_MOTO);
        System.out.println("• Bicicleta: $" + PRECIO_BICI);
        System.out.println(x:"\nMENSUALES (25% descuento):");
        System.out.println("• Carro: $" + calcularPrecio(tipo:"Carro", esMensual:true, esNocturno:false));
        System.out.println("• Moto: $" + calcularPrecio(tipo:"Moto", esMensual:true, esNocturno:false));
        System.out.println("• Bicicleta: $" + calcularPrecio(tipo:"Bicicleta", esMensual:true, esNocturno:false));
        System.out.println("\nRecargo nocturno (8PM-6AM): +$" + RECARGO_NOCTURNO);
    }
```

- private Vehiculo vehiculo: Almacena el objeto Vehiculo que está parqueado.
- private LocalDateTime entrada: Almacena la fecha y hora exactas en que el vehículo entró al parqueadero.
- private boolean esMensual: Un booleano que indica si el vehículo está pagando mensualidad.
- private boolean esNocturno: Un booleano que indica si se aplicó el recargo nocturno al momento de la entrada.
- public Registro (Vehiculo vehiculo, boolean esMensual, boolean esNocturno) Este es el **constructor** para Registro. Inicializa el vehículo, establece la hora de entrada a la hora actual y guarda si es un pago mensual o si es una entrada nocturna.

```

// Clase para cada vehículo registrado
class Registro {
    private Vehiculo vehiculo;
    private LocalDateTime entrada;
    private boolean esMensual;
    private boolean esNocturno;

    public Registro(Vehiculo vehiculo, boolean esMensual, boolean esNocturno) {
        this.vehiculo = vehiculo;
        this.entrada = LocalDateTime.now();
        this.esMensual = esMensual;
        this.esNocturno = esNocturno;
    }
```

- **Métodos getter**: getVehiculo(), getEntrada(), esMensual(), esNocturno() proporcionan acceso de solo lectura a los datos internos de un objeto Registro

- @Override public String toString() : Sobrescribe toString() para proporcionar una representación de cadena formateada de un objeto Registro, mostrando el vehículo, la hora de entrada y el tipo de pago (mensual/diario).

- **DateTimeFormatter.ofPattern("dd/MM HH:mm"):** Este es un método estático de la clase DateTimeFormatter.

- **.format(formato):** Se llama al método format() del objeto LocalDateTime (entrada), pasándole el DateTimeFormatter que creamos (formato). Esto convierte el objeto LocalDateTime en una cadena de texto según el patrón "dd/MM HH:mm"

```
100     // ENCAPSULAMIENTO - getters
101     public Vehiculo getVehiculo() { return vehiculo; }
102     public LocalDateTime getEntrada() { return entrada; }
103     public boolean esMensual() { return esMensual; }
104     public boolean esNocturno() { return esNocturno; }
105
106     @Override
107     public String toString() {
108         DateTimeFormatter formato = DateTimeFormatter.ofPattern(pattern:"dd/MM HH:mm");
109         return vehiculo.toString() + " | " + entrada.format(formato) +
110                 " | " + (esMensual ? "Mensual" : "Diario");
111     }
```

- private ArrayList<Registro> vehiculos;: Un **ArrayList** para almacenar todos los objetos Registro

- private Tarifas tarifas: Una instancia de la clase Tarifas para manejar los cálculos de precios.

- private final int CUPOS_TOTALES = 200: Una constante que define la capacidad máxima del parqueadero.

- public Parqueadero () { ... }: El **constructor** para Parqueadero. Inicializa el ArrayList, crea un objeto Tarifas e inicializa el arreglo dineroTipos con 3 elementos (todos inicialmente en 0).

```java
// Clase principal del parqueadero
class Parqueadero {
    private ArrayList<Registro> vehiculos; // ARRAYLIST
    private Tarifas tarifas;
    private double[] dineroTipos; // [carros, motos, bicis] - ARREGLO
    private final int CUPOS_TOTALES = 200;

    public Parqueadero() {
        vehiculos = new ArrayList<>();
        tarifas = new Tarifas();
        dineroTipos = new double[3]; // 0=carros, 1=motos, 2=bicis
    }
```

public boolean ingresarVehiculo(Vehiculo vehiculo, boolean esMensual) { ... }: Este **método reutilizable** maneja la admisión de un vehículo al parqueadero.

- Primero, verifica si hay **cupos disponibles**.
- Luego, usa un **bucle for** para verificar si un vehículo con la misma placa ya está parqueado, evitando duplicados.
- **for (Registro r : vehiculos): "bucle for-each"**. Se lee como: "Por cada Registro (r) en la colección vehiculos".
- Si hay espacio y no hay duplicados, crea un nuevo objeto Registro y lo añade al ArrayList vehiculos.
- También verifica si la entrada es durante las horas nocturnas e imprime un mensaje si se aplicará un recargo.

```
127     // MÉTODO REUTILIZABLE 2
128     public boolean ingresarVehiculo(Vehiculo vehiculo, boolean esMensual) {
129         // Validar cupos
130         if (vehiculos.size() >= CUPOS_TOTALES) {
131             System.out.println(x:" No hay cupos disponibles");
132             return false;
133         }
134 
135         // BUCLE FOR - verificar duplicados
136         for (Registro r : vehiculos) {
137             if (r.getVehiculo().getPlaca().equalsIgnoreCase(vehiculo.getPlaca())) {
138                 System.out.println(x:" Este vehículo ya está parqueado");
139                 return false;
140             }
141         }
142 
143         boolean esNocturno = tarifas.esNocturno();
144         Registro registro = new Registro(vehiculo, esMensual, esNocturno);
145         vehiculos.add(registro);
146 
147         System.out.println(x:"Vehículo ingresado:");
148         System.out.println(registro.toString());
149         if (esNocturno && !esMensual) {
150             System.out.println(x:"Se aplicará recargo nocturno");
151         }
152 
153         return true;
154     }
```

public boolean sacarVehiculo(String placa) : Este método maneja la salida de un vehículo del parqueadero.

- **while (i < vehiculos.size()**

  Comienza un **bucle while**. El bucle continuará ejecutándose mientras la condición i < vehiculos.size() sea true.

  vehiculos.size(): Devuelve el número actual de elementos (registros de vehículos)

  en el ArrayList vehiculos.

Registro registro = vehiculos.get(i) Obtiene el objeto Registro en la posición actual i.

if (registro.getVehiculo().getPlaca().equalsIgnoreCase(placa)

Compara la placa del registro actual con la placa proporcionada como argumento para el método, ignorando mayúsculas y minúsculas.

Si las placas coinciden, significa que se encontró el vehículo a retirar.

int precio = tarifas.calcularPrecio(...): Llama al método calcularPrecio de la instancia tarifas. Le pasa el tipo de vehículo, si es mensual y si es nocturno (recuperados del objeto registro). El valor devuelto es el precio a pagar.

Si lo encuentra, calcula el precio usando tarifas.calcularPrecio(), registra el dinero recaudado para ese tipo de vehículo, elimina el objeto Registro del ArrayList e imprime el monto total a pagar.

```
156     public boolean sacarVehiculo(String placa) {
157         // BUCLE WHILE para buscar
158         int i = 0;
159         while (i < vehiculos.size()) {
160             Registro registro = vehiculos.get(i);
161 
162             if (registro.getVehiculo().getPlaca().equalsIgnoreCase(placa)) {
163                 // Calcular precio
164                 int precio = tarifas.calcularPrecio(
165                     registro.getVehiculo().getTipo(),
166                     registro.esMensual(),
167                     registro.esNocturno()
168                 );
169 
170                 // Registrar dinero por tipo
171                 registrarDinero(registro.getVehiculo().getTipo(), precio);
172 
173                 vehiculos.remove(i);
174 
175                 System.out.println(x:"Vehículo retirado:");
176                 System.out.println(registro.toString());
177                 System.out.println("Total a pagar: $" + precio);
178 
179                 return true;
180             }
181             i++;
182         }
183 
184         System.out.println("Vehículo con placa " + placa + " no encontrado");
185         return false;
186     }
```

Este método permite buscar un vehículo por su placa y mostrar su información junto con el precio estimado a pagar.

for (Registro registro: vehiculos)

Similar al método ingresarVehiculo, utiliza un **bucle for mejorado** para cada Registro en el ArrayList.

if (registro.getVehiculo().getPlaca().equalsIgnoreCase(placa)

Compara la placa del registro actual con la placa buscada, ignorando mayúsculas y minúsculas.

Si el vehículo es encontrado: Imprime "Vehículo encontrado:" y los detalles del registro usando su método toString().

Calcula el precio estimado: Llama a tarifas.calcularPrecio() con la información del vehículo para mostrar cuánto costaría hasta el momento.

return; Una vez que se encuentra el vehículo y se muestra su información, el método return sale de la función, ya que no hay necesidad de seguir buscando.

System.out.println("Vehículo no encontrado"); Si el bucle for termina sin encontrar la placa, significa que el vehículo no está en el parqueadero y se imprime que no es encontro

```
188     public void buscarVehiculo(String placa) {
189         // BUCLE FOR mejorado
190         for (Registro registro : vehiculos) {
191             if (registro.getVehiculo().getPlaca().equalsIgnoreCase(placa)) {
192                 System.out.println(x:"Vehículo encontrado:");
193                 System.out.println(registro.toString());
194 
195                 // Mostrar precio estimado
196                 int precio = tarifas.calcularPrecio(
197                     registro.getVehiculo().getTipo(),
198                     registro.esMensual(),
199                     registro.esNocturno()
200                 );
201                 System.out.println("Precio a pagar: $" + precio);
202                 return;
203             }
204         }
205         System.out.println(x:"Vehículo no encontrado");
206     }
```

Método private void registrarDinero(String tipo, double monto)

Este es un método privado (solo se puede llamar desde dentro de la clase Parqueadero) que se usa para acumular el dinero recaudado por cada tipo de vehículo.

switch (tipo)

Utiliza una instrucción **switch** para determinar qué tipo de vehículo es.

Según el tipo, suma el monto al elementos que corresponde en el arreglo dineroTipos:

- dineroTipos[0] para carros.
- dineroTipos[1] para motos.
- dineroTipos[2] para bicicletas.

```
208     private void registrarDinero(String tipo, double monto) {
209         // SWITCH para categorizar dinero
210         switch (tipo) {
211             case "Carro": dineroTipos[0] += monto; break;
212             case "Moto": dineroTipos[1] += monto; break;
213             case "Bicicleta": dineroTipos[2] += monto; break;
214         }
215     }
```

Método mostrarEstado()

Este método imprime información general sobre el estado actual del parqueadero.

- **System.out.println("Cupos totales: " + CUPOS_TOTALES);** Muestra la capacidad máxima del parqueadero.
- **System.out.println("Cupos ocupados: " + vehiculos.size());**Muestra cuántos vehículos están actualmente estacionados.
- **System.out.println("Cupos disponibles: " + (CUPOS_TOTALES - vehiculos.size(); :** Calcula y muestra cuántos cupos quedan libres.
- **Verificación de horario:** Llama a tarifas.esNocturno() para indicar si el parqueadero está operando en horario nocturno y si se aplica el recargo.

```
    public void mostrarEstado() {
        System.out.println(x:"");
        System.out.println(x:"----------------------------");
        System.out.println(x:"Estado del Parqueadero");
        System.out.println(x:"----------------------------");
        System.out.println(x:"");
        System.out.println("Cupos totales: " + CUPOS_TOTALES);
        System.out.println("Cupos ocupados: " + vehiculos.size());
        System.out.println("Cupos disponibles: " + (CUPOS_TOTALES - vehiculos.size()));

        if (tarifas.esNocturno()) {
            System.out.println(x:"Horario nocturno - Se aplica recargo");
        } else {
            System.out.println(x:"Horario diurno");
        }
    }
```

Método mostrarVehiculosActivos()

Este método es una lista para todos los vehículos que están actualmente estacionados.

if (vehiculos.isEmpty())

Verifica si el ArrayList vehiculos está vacío.

Si no hay vehículos, imprime "No hay vehículos parqueados".

else

Si hay vehículos, usa un **bucle for con contador** (for (int i = 0; i < vehiculos.size(); i++)

Para cada vehículo, imprime su número en la lista (i + 1) y su información utilizando el método toString() del objeto Registro.

```
    public void mostrarVehiculosActivos() {
        System.out.println(x:"");
        System.out.println(x:"----------------------------");
        System.out.println(x:"Vehículos Activos");
        System.out.println(x:"----------------------------");
        System.out.println(x:"");
        if (vehiculos.isEmpty()) {
            System.out.println(x:"No hay vehículos parqueados");
        } else {
            // BUCLE FOR con contador
            for (int i = 0; i < vehiculos.size(); i++) {
                System.out.println((i + 1) + ". " + vehiculos.get(i).toString());
            }
        }
    }
```

Método mostrarReporteFinanciero()

Este método presenta un resumen de los ingresos recaudados por cada tipo de vehículo y el total general.

- **double total = dineroTipos[0] + dineroTipos[1] + dineroTipos[2]; :** Suma los valores de los tres elementos del arreglo dineroTipos para obtener el total recaudado.
- **System.out.println("Carros: " + (int)dineroTipos[0]):** Muestra el dinero recaudado por carros, convertido a entero para una presentación más limpia.
- De manera similar, muestra el dinero para motos y bicicletas.
- **System.out.println("TOTAL RECAUDADO: $" + (int)total);** Imprime el monto total recaudado.

Método public Tarifas getTarifas()

- Este es un **método *getter*** simple que permite a otras clases acceder a tarifas de la clase Parqueadero. Es útil si alguna otra parte del programa necesita directamente la información de las tarifas.

```
250     public void mostrarReporteFinanciero() {
251         double total = dineroTipos[0] + dineroTipos[1] + dineroTipos[2];
252         System.out.println(x:"");
253         System.out.println(x:"----------------------------");
254         System.out.println(x:"Reporte Financiero");
255         System.out.println(x:"----------------------------");
256         System.out.println(x:"");
257         System.out.println("Carros: $" + (int)dineroTipos[0]);
258         System.out.println("Motos: $" + (int)dineroTipos[1]);
259         System.out.println("Bicicletas: $" + (int)dineroTipos[2]);
260         System.out.println("TOTAL RECAUDADO: $" + (int)total);
261     }
262
263     public Tarifas getTarifas() { return tarifas; }
264 }
265
```

Clase Principal Parqueadero

Esta es la CLASE PRINCIPAL (MAIN) de la aplicación, donde la ejecución del programa comienza.

- private static Parqueadero parqueadero = new Parqueadero();

- Declara una variable estática parqueadero y la inicializa con una nueva instancia de la clase Parqueadero. Al ser static, esta instancia es compartida por todos los métodos de la clase ParqueaderoParkhub

```
266  // CLASE PRINCIPAL (MAIN)
267  public class ParqueaderoParkhub {
268      public static Parqueadero parqueadero = new Parqueadero();
269      public static Scanner scanner = new Scanner(System.in);
270
```

Método public static void main (String [] args)

Este es el método principal donde el programa comienza su ejecución.

- **while (true)**

  Este es el **bucle principal del programa**. Se ejecuta indefinidamente (while (true)) hasta que el usuario elija salir explícitamente.

  try { ... } catch (NumberFormatException e) {catch (Exception e)

- Este es un bloque **TRY-CATCH para manejo de excepciones**. Envuelve el código que podría causar errores, como la entrada del usuario.

- **mostrarMenu();**Llama al método mostrarMenu() para presentar las opciones al usuario.

- **int opcion = Integer.parseInt(scanner.nextLine()** Lee la línea completa ingresada por el usuario, luego intenta convertirla a un número entero. Si el usuario ingresa algo que no es un número, se lanzará una NumberFormatException.

- **switch (opcion) :**.
  - **case 1: ingresarVehiculo(); break;:** Si el usuario elige 1, llama al método ingresarVehiculo().
  - **case 2: sacarVehiculo(); break;:** Si el usuario elige 2, llama al método sacarVehiculo().
  - **case 3: buscarVehiculo(); break;:** Si el usuario elige 3, llama al método buscarVehiculo().
  - **case 4: parqueadero.mostrarEstado(); break;:** Si el usuario elige 4, llama al método mostrarEstado() de la instancia parqueadero.
  - **case 5: parqueadero.mostrarVehiculosActivos(); break;:** Si el usuario elige 5, llama al método mostrarVehiculosActivos() de la instancia parqueadero.
  - **case 6: parqueadero.getTarifas().mostrarTarifas(); break;:** Si el usuario elige 6, primero obtiene el objeto Tarifas de parqueadero y luego llama a su método mostrarTarifas().
  - **case 7: parqueadero.mostrarReporteFinanciero(); break;:** Si el usuario elige 7, llama al método mostrarReporteFinanciero() de la instancia parqueadero.
  - **case 8: System.out.println("¡Gracias por usar el sistema, no vuelva pronto por favor!"); return;:** Si el usuario elige 8, imprime un mensaje de despedida y return; **termina la ejecución del método main**, y por lo tanto, el programa.
  - **default: System.out.println("Opción inválida (1-8)");:** Si el usuario ingresa un número que no está entre 1 y 8, se imprime un mensaje de "Opción inválida".

```
Run | Debug | Run main | Debug main
public static void main(String[] args) {
    System.out.println(x:"----------------------------");
    System.out.println(x:"Bienvenido al Parqueadero ParkHub");
    System.out.println(x:"----------------------------");

    while (true) {
        try { // TRY-CATCH para manejo de excepciones
            mostrarMenu();
            int opcion = Integer.parseInt(scanner.nextLine());

            // SWITCH principal
            switch (opcion) {
                case 1: ingresarVehiculo(); break;
                case 2: sacarVehiculo(); break;
                case 3: buscarVehiculo(); break;
                case 4: parqueadero.mostrarEstado(); break;
                case 5: parqueadero.mostrarVehiculosActivos(); break;
                case 6: parqueadero.getTarifas().mostrarTarifas(); break;
                case 7: parqueadero.mostrarReporteFinanciero(); break;
                case 8:
                    System.out.println(x:"¡Gracias por usar el sistema, no vuelva pronto por favor!");
                    return;
                default:
                    System.out.println(x:"Opción inválida (1-8)");
            }

        } catch (NumberFormatException e) {
            System.out.println(x:"Por favor ingrese solo números");
        } catch (Exception e) {
            System.out.println("Error inesperado: " + e.getMessage());
        }
    }
}
```

Estas son las opciones que mostrara el método mostrarMenu()

```
// Se que hay muchos printlns, pero es para que se vea mas bonito el programa
    private static void mostrarMenu() {
        System.out.println(x:"");
        System.out.println(x:"----------------------------");
        System.out.println(x:"MENÚ PRINCIPAL");
        System.out.println(x:"----------------------------");
        System.out.println(x:"");
        System.out.println(x:"1. Ingresar vehículo");
        System.out.println(x:"2. Sacar vehículo");
        System.out.println(x:"3. Buscar vehículo");
        System.out.println(x:"4. Ver estado");
        System.out.println(x:"5. Ver vehículos activos");
        System.out.println(x:"6. Ver tarifas");
        System.out.println(x:"7. Reporte financiero");
        System.out.println(x:"8. Salir");
        System.out.print(s:"Opción: ");
    }
```

private static void ingresarVehiculo()

Este método estático y privado se encarga de la interacción con el usuario para ingresar un nuevo vehículo.

- try { ... } catch (NumberFormatException e) {
- Otro bloque try-catch para manejar posibles errores de entrada del usuario.
- Solicitud de tipo de vehículo Pide al usuario que elija el tipo de vehículo (Carro, Moto, Bicicleta) mediante un número.
- int tipo = Integer.parseInt(scanner.nextLine(): Lee la opción del tipo.
- Validación de tipo Si el número no está en el rango 1-3, imprime un mensaje de error y return; sale del método.
- Solicitud de placa : Pide al usuario la placa del vehículo.
- Validación de placa vacía : Si la placa está vacía, imprime un error y return; sale.
- Pregunta sobre mensualidad : Pregunta si es una mensualidad. scanner.nextLine().toLowerCase().startsWith("s") convierte la entrada a minúsculas y verifica si comienza con 's' (para "sí").
- Vehiculo vehiculo = null; : Declara una variable vehiculo de tipo Vehiculo y la inicializa como null.

```java
    private static void ingresarVehiculo() {
        System.out.println(x:"");
        System.out.println(x:"----------------------------");
        System.out.println(x:"Ingresar Vehículo");
        System.out.println(x:"----------------------------");
        System.out.println(x:"");

        try {
            System.out.println(x:"Tipos disponibles:");
            System.out.println(x:"1. Carro");
            System.out.println(x:"2. Moto");
            System.out.println(x:"3. Bicicleta");
            System.out.print(s:"Tipo (1-3): ");

            int tipo = Integer.parseInt(scanner.nextLine());

            if (tipo < 1 || tipo > 3) {
                System.out.println(x:"Tipo inválido");
                return;
            }

            System.out.print(s:"Placa: ");
            String placa = scanner.nextLine().trim();

            if (placa.isEmpty()) {
                System.out.println(x:"La placa no puede estar vacía");
                return;
            }

            System.out.print(s:"¿Mensualidad? (s/n): ");
            boolean esMensual = scanner.nextLine().toLowerCase().startsWith(prefix:"s");

```

switch (tipo) :

Este **SWITCH para crear vehículo** usa el número de tipo ingresado por el usuario para crear una instancia específica de Carro, Moto o Bicicleta (utilizando la **herencia**). Por ejemplo, si tipo es 1, se crea un new Carro(placa).

- **parqueadero.ingresarVehiculo(vehiculo, esMensual); :** Llama al método ingresarVehiculo() de la instancia parqueadero para registrar el vehículo creado.

- **catch (NumberFormatException e) (Líneas 40:** Captura el error si el usuario no ingresa un número para el tipo de vehículo.

```
357             switch (tipo) {
358                 case 1: vehiculo = new Carro(placa); break;
359                 case 2: vehiculo = new Moto(placa); break;
360                 case 3: vehiculo = new Bicicleta(placa); break;
361             }
362
363             parqueadero.ingresarVehiculo(vehiculo, esMensual);
364
365         } catch (NumberFormatException e) {
366             System.out.println(x:"Ingrese un número válido para el tipo");
367         }
368     }
```

private static void sacarVehiculo()

Este método estático y privado maneja la lógica para retirar un vehículo.

- Impresión de títulos: Imprime encabezados.
- Solicitud de placa: Pide al usuario la placa del vehículo a retirar.
- Validación de placa vacía: Si la placa está vacía, imprime un error y return; sale.
- parqueadero.sacarVehiculo(placa): Llama al método sacarVehiculo() de la instancia parqueadero para procesar la salida del vehículo.

```
370     private static void sacarVehiculo() {
371         System.out.println(x:"");
372         System.out.println(x:"----------------------------");
373         System.out.println(x:"Sacar Vehículo");
374         System.out.println(x:"----------------------------");
375         System.out.println(x:"");
376
377         System.out.print(s:"Placa del vehículo: ");
378         String placa = scanner.nextLine().trim();
379
380         if (placa.isEmpty()) {
381             System.out.println(x:"La placa no puede estar vacía");
382             return;
383         }
384
385         parqueadero.sacarVehiculo(placa);
386     }
```

Método private static void buscarVehiculo()

Este método estático y privado maneja la lógica para buscar un vehículo.

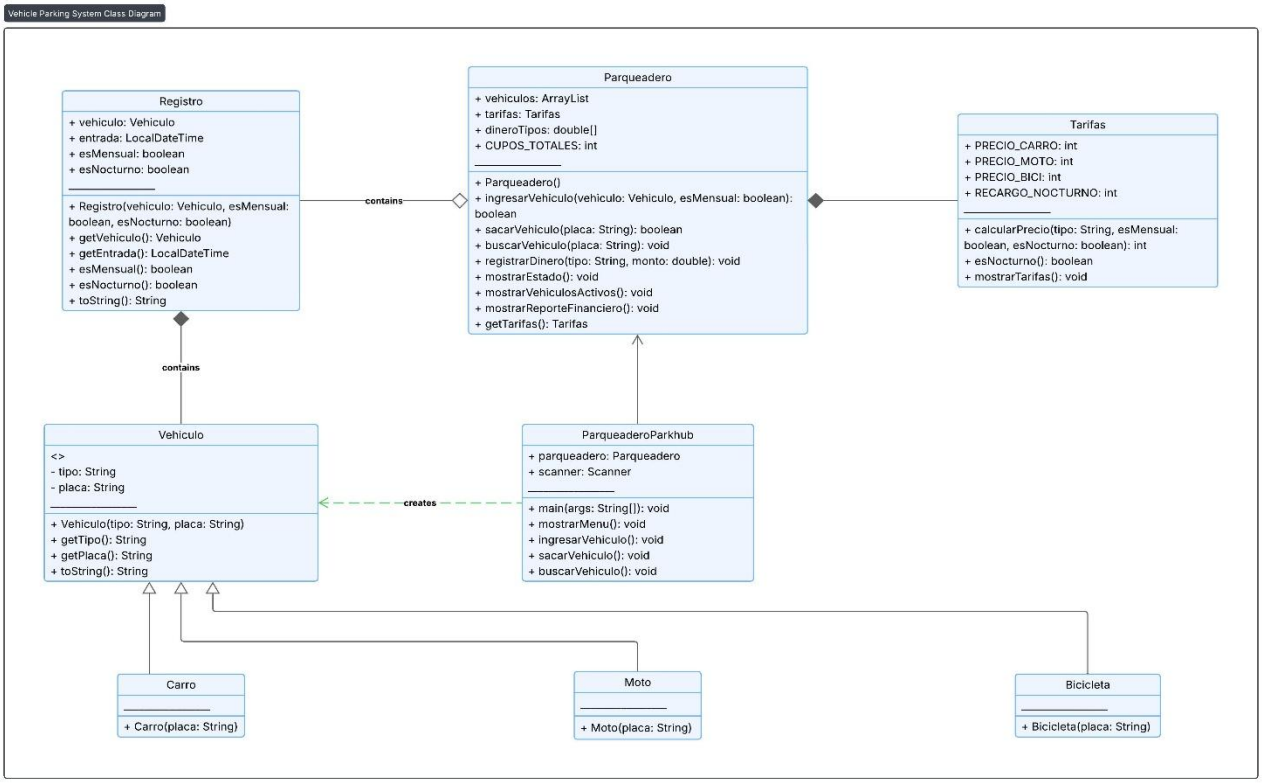
• Solicitud de placa: Pide al usuario la placa del vehículo a buscar.

• Validación de placa vacía: Si la placa está vacía, imprime un error y return; sale.

• parqueadero.buscarVehiculo(placa); Llama al método buscarVehiculo() de la instancia parqueadero para realizar la búsqueda.

```
388     private static void buscarVehiculo() {
389         System.out.println(x:"");
390         System.out.println(x:"----------------------------");
391         System.out.println(x:"Buscar Vehículo");
392         System.out.println(x:"----------------------------");
393         System.out.println(x:"");
394 
395         System.out.print(s:"Placa del vehículo: ");
396         String placa = scanner.nextLine().trim();
397 
398         if (placa.isEmpty()) {
399             System.out.println(x:"La placa no puede estar vacía");
400             return;
401         }
402 
403         parqueadero.buscarVehiculo(placa);
404     }
405 }
```

EXPLICACIÓN UML



1. Relación de Herencia entre Vehiculo y sus subclases (Carro, Moto, Bicicleta)

Relación:

Carro, Moto y Bicicleta heredan de la clase abstracta Vehiculo.

Descripción:

Esta herencia permite reutilizar atributos como tipo y placa, así como métodos comunes como getPlaca() o toString().

Cada subclase representa un tipo específico de vehículo y se instancia según la elección del usuario.

Ejemplo en el código:

Vehiculo v = new Carro("ABC123");

Vehiculo m = new Moto("XYZ789");

2. Relación de Composición entre Parqueadero y Registro

Relación:

La clase Parqueadero contiene una lista de objetos Registro.

Descripción:

Cada Registro representa un vehículo que ha ingresado al parqueadero.

Esta es una relación de composición porque un registro no tiene sentido fuera del parqueadero y su existencia depende de él.

Ejemplo en el código:

ArrayList<Registro> vehiculos = new ArrayList<>();

3. Relación de Agregación entre Parqueadero y Tarifas

Relación:

La clase Parqueadero utiliza una instancia de la clase Tarifas.

Descripción:

El objeto Tarifas contiene las constantes de precios por tipo de vehículo y métodos para calcular la tarifa final según condiciones especiales (mensualidad o recargo nocturno).

La relación es de agregación porque Tarifas puede existir independientemente de Parqueadero.

Ejemplo en el código:

Tarifas tarifas = new Tarifas();

4. Relación de Creación entre ParqueaderoParhub y Parqueadero

Relación:

ParqueaderoParhub crea y utiliza una instancia de la clase Parqueadero.

Descripción:

Esta es una relación de uso (o dependencia). La clase ParqueaderoParhub actúa como punto de entrada (main) y controla toda la lógica de interacción con el usuario.

Delegando tareas a Parqueadero, se mantiene una buena separación entre lógica de negocio y presentación.

Ejemplo en el código:

Parqueadero parqueadero = new Parqueadero();

5. Asociación entre Registro y Vehiculo

Relación:

Cada objeto Registro contiene una referencia a un objeto Vehiculo.

Descripción:

Esta relación representa que un Registro siempre está asociado a un vehículo específico.

Permite recuperar los datos del vehículo ingresado (tipo y placa), así como la hora de entrada, si es mensual y si ingresó en horario nocturno.

Ejemplo en el código:

Registro r = new Registro(new Carro("ABC123"), true, false);

CONCLUSION

El desarrollo del sistema *ParkHub* representó mucho más que un ejercicio práctico: fue una oportunidad real para explorar, aplicar y consolidar conocimientos en programación orientada a objetos, manejo de clases, atributos y estructuras lógicas dentro de un sistema funcional.

Durante el proceso, surgieron desafíos importantes. Si bien el código como tal no representó grandes obstáculos, sí lo hizo la correcta modularización del sistema. Separar las responsabilidades en clases independientes sin perder la integridad del programa fue un reto clave, especialmente al enfrentar errores de ejecución y restricciones que interferían con la experiencia del usuario. Esta experiencia permitió comprender mejor la importancia del diseño estructurado del software y el equilibrio entre modularidad y funcionalidad.

Uno de los aspectos más útiles y satisfactorios del sistema fue precisamente su arquitectura modular, que facilita el mantenimiento y la escalabilidad. Cada clase cumple un rol claro, lo que permite mejorar o extender el sistema con mayor facilidad en el futuro. Aunque el resultado final no fue perfecto, representa un logro significativo dentro de las capacidades y conocimientos actuales del equipo.

Este proyecto motiva a continuar aprendiendo, especialmente en el uso de bibliotecas externas y herramientas que podrían hacer el sistema más visual, dinámico e interactivo. Reconocer lo que aún falta por aprender no es una debilidad, sino el primer paso hacia un crecimiento consciente y constante.

ParkHub es el reflejo de un trabajo comprometido, con visión de mejora continua. Y aunque el camino apenas comienza, este proyecto siembra una base sólida para futuros desarrollos más ambiciosos, con mayor preparación técnica y creatividad.

DEDICATORIA

Queremos dedicar este informe al profesor Yisus, por ser más que un guía académico: por ser un verdadero apoyo durante todo el desarrollo del proyecto.

Agradecemos profundamente su forma clara y práctica de enseñar, sus ejemplos reales, su disposición constante para resolver dudas, y sobre todo, su calidad humana. Es un profesor que transmite conocimiento con paciencia, accesible en todo momento, y con la capacidad de hacer que lo complejo parezca alcanzable.

Gracias por compartir su experiencia con nosotros y por motivarnos a seguir aprendiendo con entusiasmo. Este trabajo es también fruto de su enseñanza.