



UNIVERSIDAD DE LAS FUERZAS ARMADAS (ESPE)

SEGUNDO SEMESTRE

CARRERAS TECNICAS

DEPARTAMENTO DE CIENCIAS EXACTAS

PRIMER PARCIAL

“Sistema de Gestión de Parking”

AUTORES:

Steven Guerrero

Samir Samande

PARALELO:

1322

DOCENTE:

LUIS ENRIQUE JARAMILLO MONTAÑO

SANGOLQUI – ECUADOR

1. Introducción

La actividad autónoma número 1, asignada como parte del primer parcial, tiene como objetivo principal el desarrollo de un sistema de gestión de parqueadero. Este proyecto integra conceptos fundamentales como la Programación Orientada a Objetos, el diseño de una interfaz gráfica de usuario y el manejo de bases de datos, desafiando a los estudiantes a aplicar de manera práctica los conocimientos adquiridos.

El sistema para desarrollar deberá contar con funcionalidades clave, como el registro, consulta y actualización de datos de los vehículos que utilizan el parqueadero. Además, los estudiantes deberán entregar su trabajo en un formato que incluya el código fuente y la documentación correspondiente, comprimidos en un archivo ZIP. Este proyecto fomenta no solo el aprendizaje técnico, sino también el desarrollo de habilidades para la organización y la documentación profesional de proyectos.

El trabajo será evaluado considerando aspectos importantes como la calidad del diseño UML, la implementación del código y la elaboración del informe final, promoviendo un enfoque integral hacia el desarrollo de software. Esta actividad constituye una oportunidad para consolidar competencias técnicas y metodológicas en un contexto académico.

2. OBJETIVOS

2.1 OBJETIVO GENERAL

Desarrollar un sistema de gestión de parqueadero que permita registrar, consultar y actualizar datos de vehículos, aplicando POO, interfaces gráficas y bases de datos.

3. OBJETIVOS ESPECÍFICO

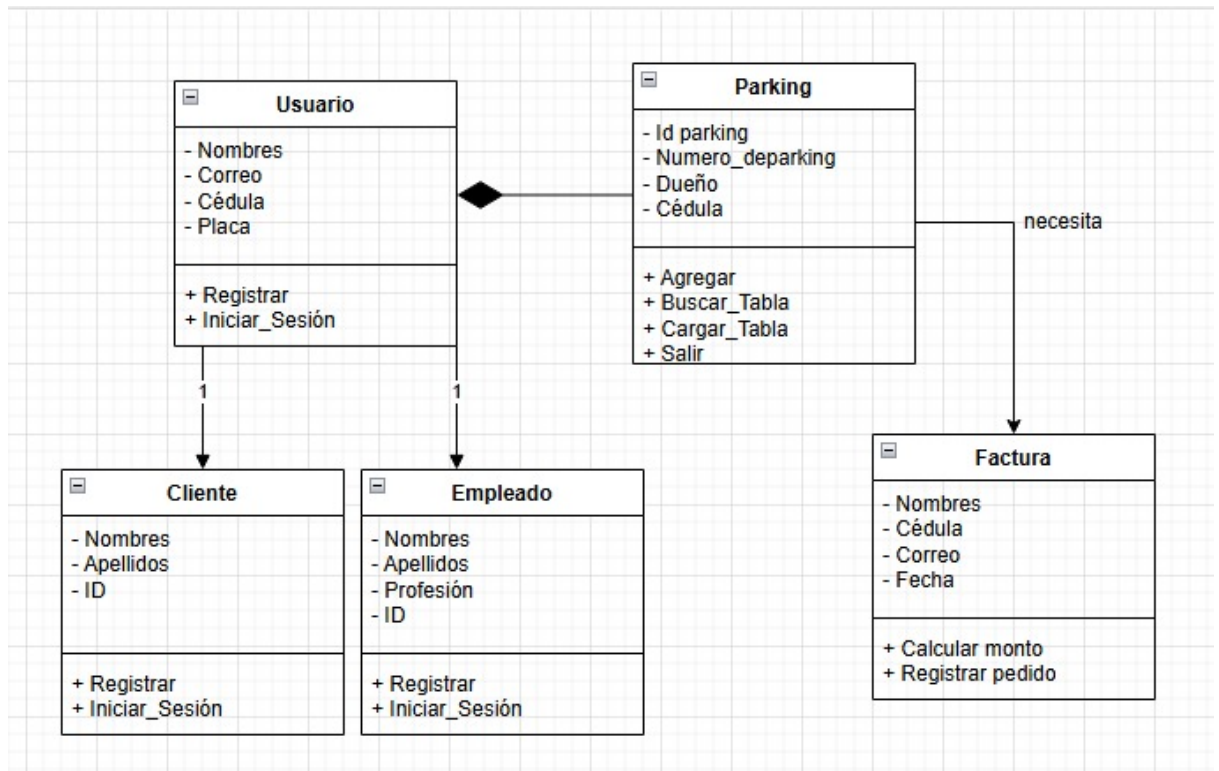
Implementar funcionalidades para el registro, consulta y actualización de datos de vehículos utilizando los principios de la Programación Orientada a Objetos .

Diseñar una interfaz gráfica intuitiva y conectar el sistema con una base de datos para garantizar la gestión eficiente de la información del parqueadero.

4. MARCO TEÓRICO

A continuación vamos a desarrollar una breve explicación sobre un sistema para gestionar un parqueadero utilizando POO, una interfaz gráfica, y bases de datos. El sistema debe incluir funcionalidades de registro, consulta y actualización de vehículos.

Diagrama UML



Clases principales

- **Usuario:**

Atributos:

- **Nombres:** Nombre(s) del usuario.
- **Correo:** Dirección de correo electrónico.
- **Cédula:** Número de identificación.

- Placa: Número de la placa del vehículo.

Métodos:

- Registrar: Permite registrar un nuevo usuario.
- Iniciar_Sesión: Permite al usuario autenticarse.

Parking:

Atributos:

- Id_parking: Identificador único del lugar de estacionamiento.
- Numero_departing: Número asociado al espacio de estacionamiento.
- Dueño: Propietario del vehículo.
- Cédula: Cédula del dueño.

Métodos:

- Agregar: Permite agregar un nuevo espacio de estacionamiento.
- Buscar_Tabla: Busca un registro en la tabla.
- Cargar_Tabla: Carga los datos existentes en la tabla.
- Salir: Permite salir del sistema.

Factura:**Atributos:**

- Nombres: Nombre del cliente asociado a la factura.
- Cédula: Identificación del cliente.
- Correo: Correo electrónico del cliente.
- Fecha: Fecha en la que se genera la factura.

Métodos:

- Calcular monto: Realiza el cálculo del monto a pagar.
- Registrar pedido: Registra un nuevo pedido o transacción.

Clases derivadas

- **Cliente** (Hereda de Usuario):

Atributos:

- Nombres: Nombre(s) del cliente.
- Apellidos: Apellido(s) del cliente.
- ID: Identificación única.

Métodos:

- Registrar: Permite registrar un cliente.
- Iniciar_Sesión: Permite al cliente autenticarse.
- **Empleado** (Hereda de Usuario):

Atributos:

- Nombres: Nombre(s) del empleado.
- Apellidos: Apellido(s) del empleado.
- Profesión: Cargo o función del empleado.
- ID: Identificación única.

Métodos:

- Registrar: Permite registrar un empleado.
- Iniciar_Sesión: Permite al empleado autenticarse.

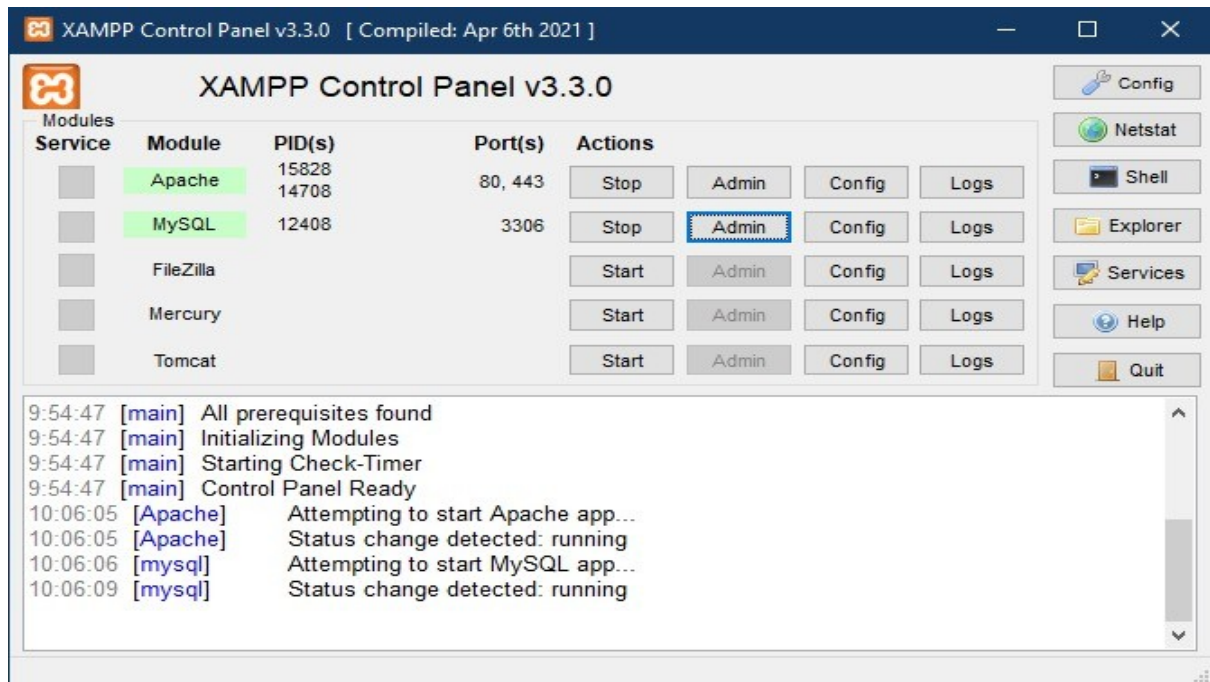
Relación entre Usuario y Parking:

- Representada con un rombo negro (composición), indicando que un usuario puede estar asociado a un estacionamiento.

Relación entre Parking y Factura:

- Representada con una flecha y el término "necesita", indicando que el estacionamiento genera o está asociado a una factura.

Panel de control de XAMPP



Servicios Activos:

Apache y **MySQL** están en estado "running" (en ejecución), lo que significa que ambos servicios están funcionando correctamente.

Los puertos en uso son:

- **Apache:** Puertos 80 y 443 (comúnmente usados para HTTP y HTTPS).
- **MySQL:** Puerto 3306 (usado para bases de datos).

Opciones Disponibles:

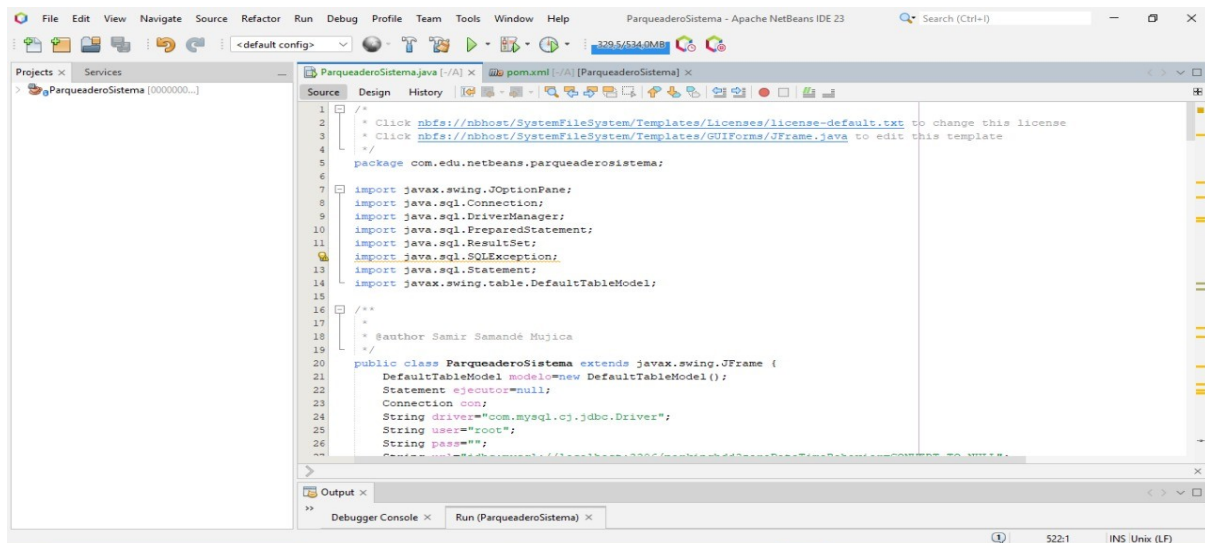
- **Stop:** Para detener un servicio en ejecución.
- **Admin:** Abre la interfaz administrativa (por ejemplo, el panel de phpMyAdmin para MySQL).
- **Config:** Configura ajustes del servicio.

- **Logs:** Muestra los registros de eventos del servicio.

Registro de Eventos:

- A las 9:54:47, se encontraron todos los requisitos y se inicializaron los módulos.
- Apache y MySQL se iniciaron correctamente a las 10:06:05 y 10:06:09, respectivamente.

Code



- **Declaración del paquete:**

El programa pertenece a un paquete llamado

com.edu.netbeans.parqueaderosistema. Esto organiza el código y lo agrupa con otras clases relacionadas dentro de la misma carpeta.

- **Importación de librerías**

Se usan librerías de Java Swing para crear la interfaz gráfica y librerías de Java SQL para manejar la conexión, consultas y resultados de la base de datos.

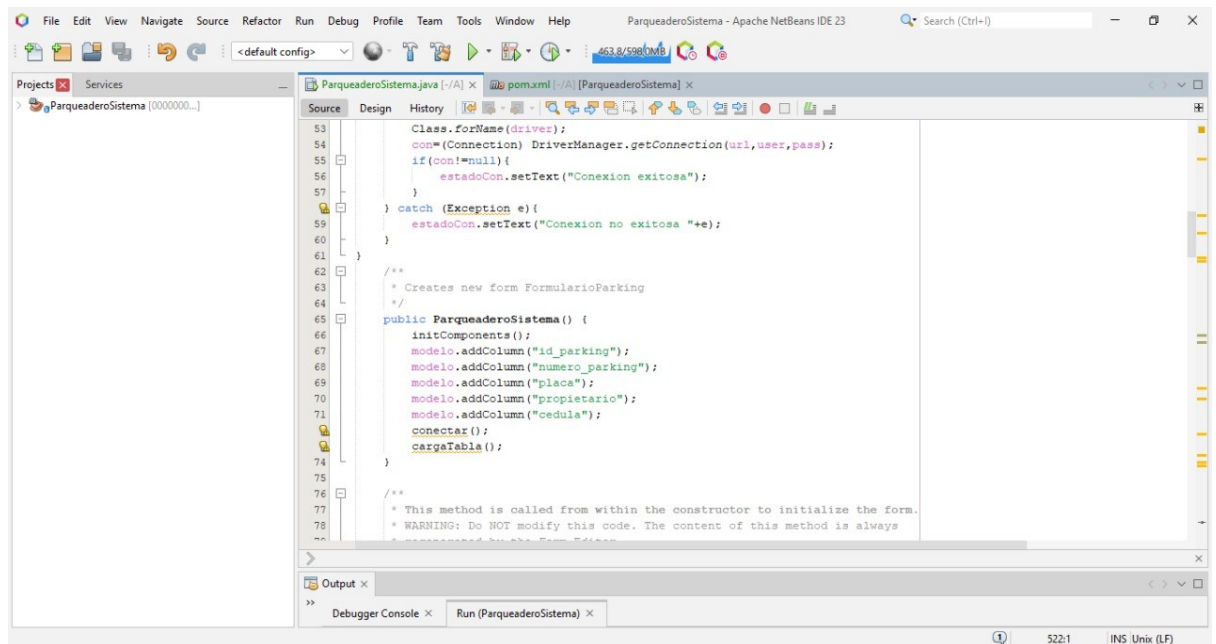
- **Clase principal**

La clase principal, llamada ParqueaderoSistema, crea una ventana gráfica (usando JFrame) que probablemente será el centro del sistema de parqueadero.

- **Variables clave**

Se usa un modelo de tabla (DefaultTableModel) para manejar datos que se mostrarán en la interfaz gráfica.

- Se configura una conexión a una base de datos MySQL utilizando un driver específico, junto con un usuario (root) y una contraseña vacía.
- También hay una referencia a las herramientas necesarias para ejecutar consultas y procesar resultados.

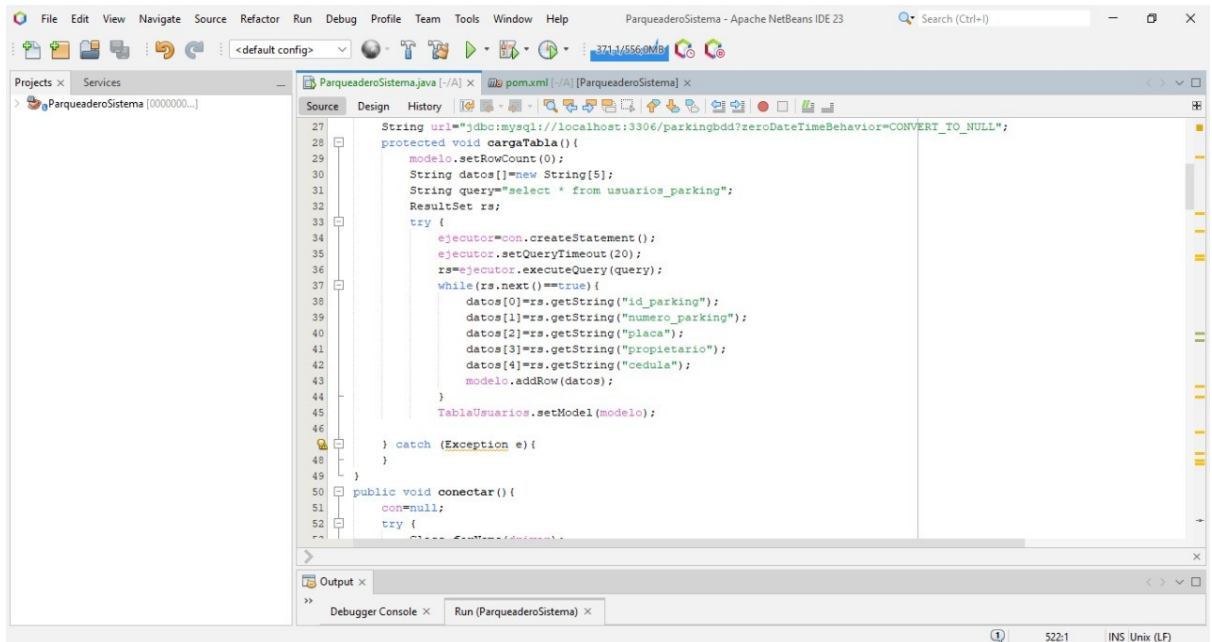


1. **Conexión con la base de datos:**

Se establece la conexión con la base de datos usando el controlador MySQL y las credenciales proporcionadas.

Descripción:

- Si la conexión es exitosa, se muestra el mensaje "**Conexión exitosa**" en un componente de texto (**estadoCon**).
- Si ocurre un error, se muestra "**Conexión no exitosa**" seguido del error específico



Declaración de la conexión a la base de datos

- **String**

url="jdbc:mysql://localhost:3306/parkingbdd?zeroDateTimeBehavior=CONVERT_TO_NULL";

- Define la dirección para conectar con la base de datos.
- El servidor es localhost (la máquina local).
- El puerto es 3306 (puerto predeterminado de MySQL).
- El nombre de la base de datos es parkingbdd.

2. Método cargaTabla()

Propósito:

- Este método se encarga de cargar datos desde la tabla usuarios_parking de la base de datos y mostrarlos en una tabla del sistema.

Pasos:

- **modelo.setRowCount(0);**
- Limpia el modelo de la tabla eliminando los datos previamente cargados.

Preparación de variables:

- Se define un arreglo datos de tamaño 5 para almacenar temporalmente los valores de las columnas.
- Se crea una consulta SQL con la instrucción: `select from usuarios parking`.

Ejecución de la consulta:

- Se utiliza un objeto Statement (llamado ejecutor) para ejecutar la consulta.
- Se establece un tiempo límite para la ejecución de 20 segundos con `setQueryTimeout(20)`.
- Los resultados de la consulta se almacenan en un objeto ResultSet (rs).

Iteración sobre los resultados:

- Mientras existan filas en el resultado (`while (rs.next() == true)`):
- **id_parking:** ID único del parqueadero.
- **Numero parking:** Número del parqueo.
- **placa:** Placa del vehículo.
- **propietario:** Nombre del propietario.
- **cedula:** Número de cédula del propietario.
- Los valores se almacenan en el arreglo datos.
- El arreglo se añade como una nueva fila al modelo de la tabla con `modelo.addRow(datos)`.

Actualización de la tabla en la interfaz:

- La tabla visible se actualiza con el modelo cargado utilizando
`TablaUsuarios.setModel(modelo).`

Manejo de excepciones:

- Si ocurre un error durante la ejecución del código, este es capturado por el bloque
`catch (Exception e).`

Método conectar()

- **Propósito:** Este método establece la conexión inicial con la base de datos.

Inicialización de la variable con:

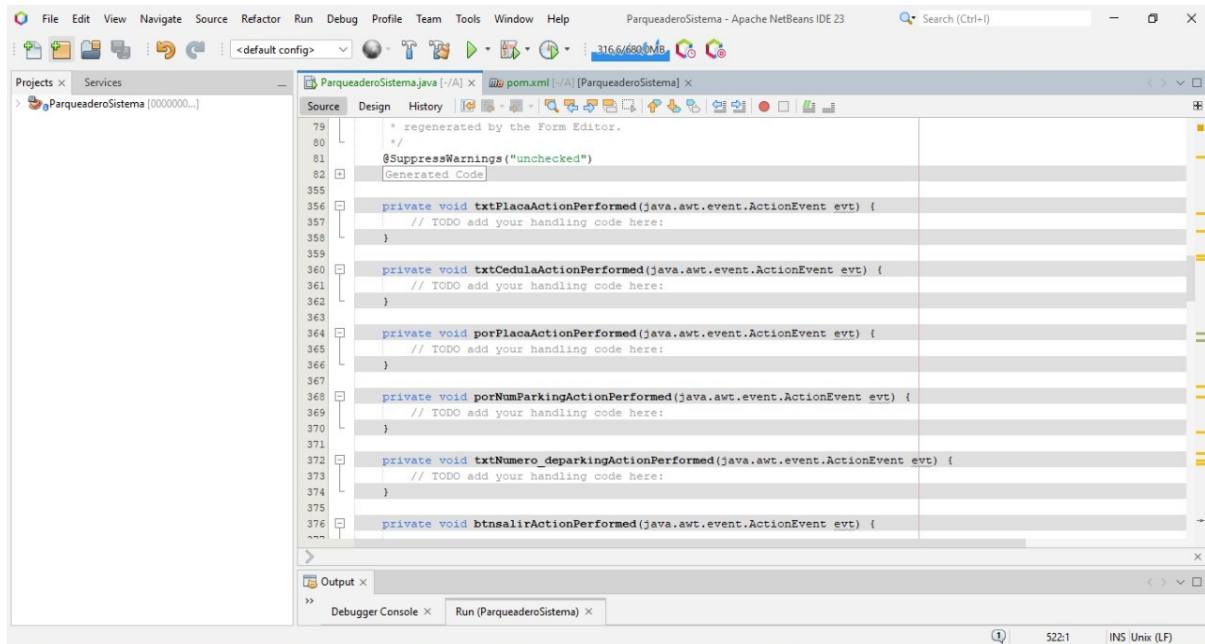
- La variable de conexión (con) se inicializa como null.

Intento de conexión:

- En las líneas siguientes (no visibles en la imagen), el método intentará conectarse a la base de datos utilizando los parámetros definidos en la URL.

Manejo de excepciones:

- Si ocurre algún problema al establecer la conexión, se capturará la excepción para evitar que el programa se detenga abruptamente.



Estos métodos están diseñados para manejar las acciones realizadas por el usuario en la interfaz gráfica, como escribir en campos de texto o hacer clic en botones.

txtPlacaActionPerformed(java.awt.event.ActionEvent evt)

- Método que responde cuando ocurre una acción en el campo de texto asociado a la **placa**.
- Actualmente está vacío y se muestra el comentario:
- `// TODO add your handling code here` (Agregar el código de manejo aquí).

txtCedulaActionPerformed(java.awt.event.ActionEvent evt)

- Método que se ejecuta cuando ocurre una acción en el campo de texto para la **cédula**.
- También está vacío y pendiente de implementación.

porPlacaActionPerformed(java.awt.event.ActionEvent evt)

- Responde a una acción asociada a un elemento (posiblemente un botón o acción relacionada con la **placa**).
- Está vacío y listo para agregar código.

porNumParkingActionPerformed(java.awt.event.ActionEvent evt)

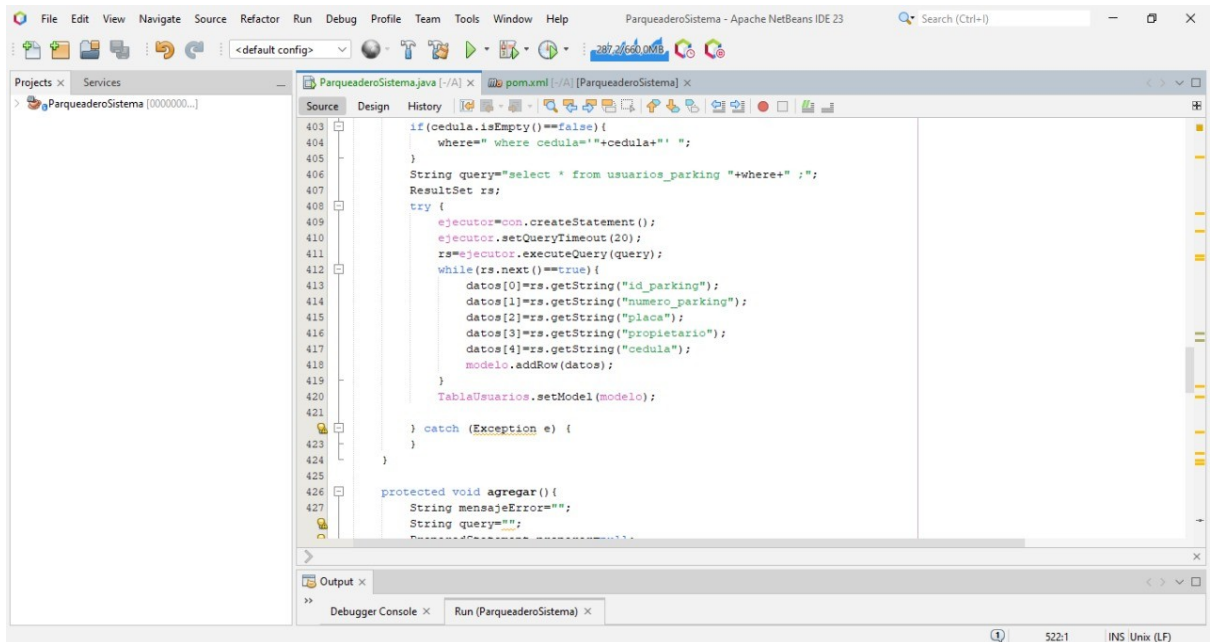
- Se activa cuando ocurre una acción relacionada con el **número de parqueo**.
- Como los demás, está vacío.

txtNumero_deparkingActionPerformed(java.awt.event.ActionEvent evt)

- Método que responde a una acción relacionada con el campo de texto del **número de parqueo**.
- Pendiente de código.

btnSalirActionPerformed(java.awt.event.ActionEvent evt)

- Método que responde cuando se realiza una acción en el botón **Salir**.
- También está pendiente de implementación.



Definición de filtro para cédulas y ejecución de la consulta

- **Verificación del filtro de cédula:** Comprueba si el valor de la cédula no está vacío. Si tiene un valor, añade un filtro WHERE a la consulta para buscar registros específicos por cédula.
- **Construcción dinámica de la consulta SQL:** Se genera una consulta SELECT que obtiene datos de la tabla usuarios_parking. Si no se especifica cédula, trae todos los registros.
- **Ejecución de la consulta:** Utiliza un objeto Statement para ejecutar la consulta con un límite de tiempo de 20 segundos.
- **Procesamiento de los resultados:** Recorre las filas devueltas por la consulta, extrae los valores de las columnas relevantes (id_parking, numero_parking, placa, propietario, cedula) y los añade al modelo de tabla.

Manejo de excepciones

- **Captura de errores:** Si ocurre un problema durante la ejecución de la consulta o la conexión, se captura mediante un bloque **catch**. Esto evita que el programa se bloquee por errores no controlados.

Método agregar

- **Definición inicial:** Se preparan variables para manejar mensajes de error y construir una consulta SQL de inserción (INSERT).
- **Posible objetivo del método:** Insertar nuevos datos en la tabla usuarios_parking con valores proporcionados por el usuario.

Método agregar

- **Definición inicial:** Se preparan variables para manejar mensajes de error y construir una consulta SQL de inserción (INSERT).
- **Posible objetivo del método:** Insertar nuevos datos en la tabla usuarios_parking con valores proporcionados por el usuario.

```

377
378
379
380
381
382 private void btnAgregarActionPerformed(java.awt.event.ActionEvent evt) {
383     agregar();
384 }
385
386 private void btnBuscarActionPerformed(java.awt.event.ActionEvent evt) {
387     buscarTabla(porNumParking.getText(), porPlaca.getText(), porPropietario.getText(), porCedula.getText());
388 }
389
390 protected void buscarTabla(String numero_parking, String placa, String propietario, String cedula) {
391     modelo.setRowCount(0);
392     String datos[] = new String[5];
393     String where = " where 1=1 ";
394     if (numero_parking.isEmpty() == false) {
395         where = " where numero_parking='" + numero_parking + "' ";
396     }
397     if (placa.isEmpty() == false) {
398         where = " where placa='" + placa + "' ";
399     }
400     if (propietario.isEmpty() == false) {
401         where = " where propietario='" + propietario + "' ";
402     }
403     if (cedula.isEmpty() == false) {
404         where = " where cedula='" + cedula + "' ";
405     }
406     String query = "select * from usuarios_parking " + where;
407     try {
408         PreparedStatement ps = con.prepareStatement(query);
409         ResultSet rs = ps.executeQuery();
410         while (rs.next()) {
411             datos[0] = rs.getString(1);
412             datos[1] = rs.getString(2);
413             datos[2] = rs.getString(3);
414             datos[3] = rs.getString(4);
415             datos[4] = rs.getString(5);
416             modelo.addRow(datos);
417         }
418     } catch (SQLException ex) {
419         JOptionPane.showMessageDialog(this, ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
420     }
421 }
422
423 System.exit(0);
424 }

```

Método btnagregarActionPerformed

Este método se ejecuta cuando se presiona un botón relacionado con la acción de "agregar". Invoca un método llamado agregar(), que probablemente contiene la lógica para añadir datos al sistema.

Método btnbuscarActionPerformed

Este método se activa cuando se presiona un botón de "buscar". Llama al método buscarTabla () con los valores obtenidos de varios campos de texto (porNumParking, porPlaca, porPropietario, porCedula).

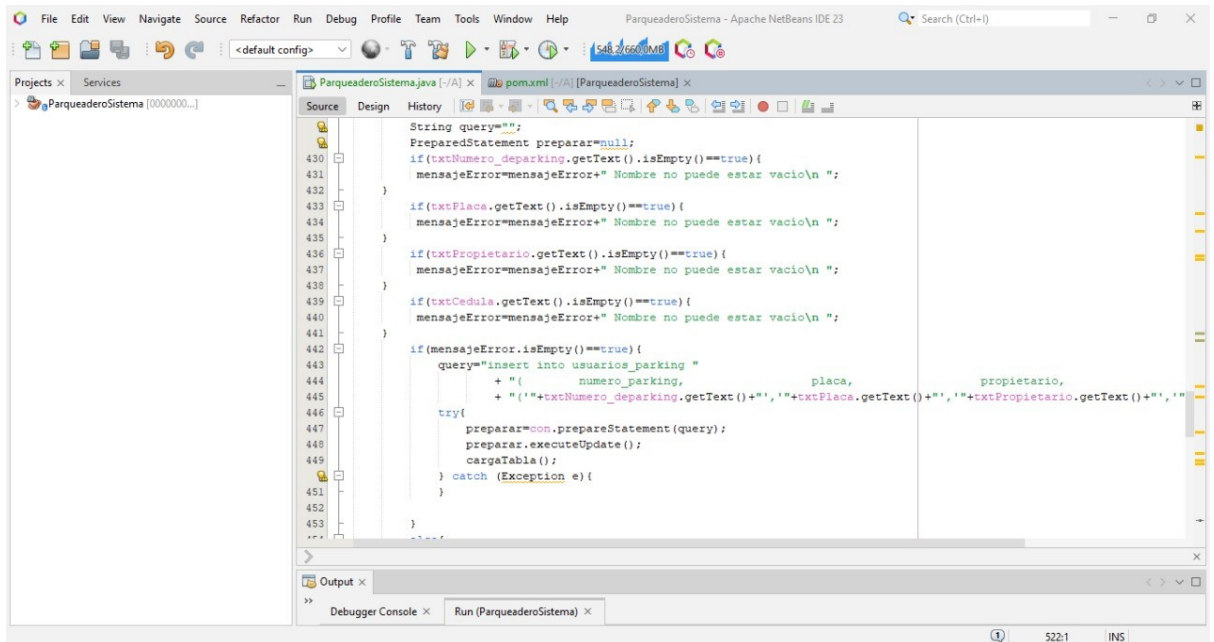
Método buscarTabla

Este método implementa la lógica de búsqueda en función de varios parámetros: número de parking, placa, propietario y cédula. Aquí está su explicación:

- **modelo.setRowCount(0):** Limpia cualquier dato previo de la tabla para mostrar solo los nuevos resultados.

Definición de variables:

- **datos[]:** Un arreglo para almacenar los datos de búsqueda.
- **where:** Una cadena que contendrá las condiciones para filtrar los resultados.
- **Condiciones de filtrado:** Dependiendo de si cada parámetro (numero_parking, placa, etc.) no está vacío (isEmpty() == false), se agrega una condición al filtro where:
- Si numero parking no está vacío, se añade una condición que filtra por ese número.
- Si placa no está vacía, se agrega la condición para la placa.
- Se hace lo mismo para propietario y cédula.

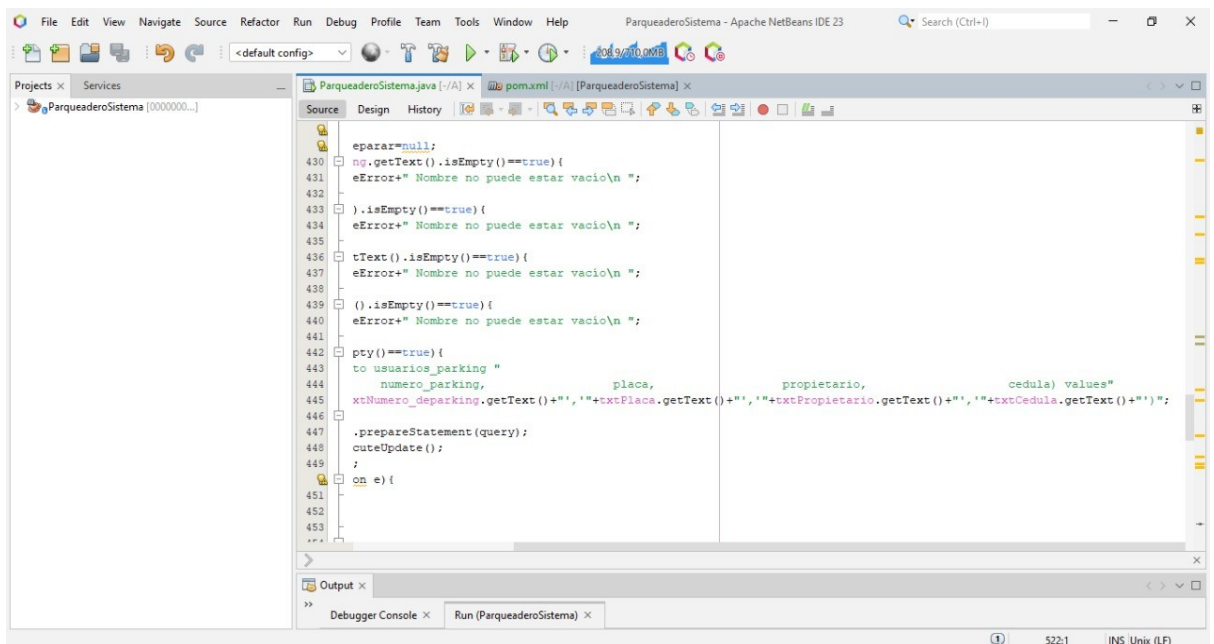


Análisis del código:

- **Validaciones de campos vacíos:** El programa verifica que todos los campos obligatorios (como número de parqueadero, placa, propietario y cédula) tengan datos antes de proceder. Si algún campo está vacío, se acumula un mensaje de error en una variable llamada mensaje Error, indicando qué información falta.
- **Condición para proceder:** Si no hay mensajes de error (es decir, todos los campos están completos), se permite que el sistema continúe con la operación de guardar los datos en la base de datos.
- **Insertión de datos:** Los datos ingresados por el usuario (número de parqueadero, placa, propietario y cédula) se preparan para ser guardados en una tabla de base de datos llamada usuarios_parking.
- **Ejecución del proceso:** El sistema intenta ejecutar la inserción en la base de datos utilizando un bloque de manejo de errores. Si todo funciona correctamente, los datos se guardan, y se actualiza la tabla para reflejar los nuevos registros. Si ocurre un error, este se captura y se puede manejar para evitar que el programa falle.

Problemas detectados:

- **Riesgo de seguridad (inyección SQL):** Los datos ingresados por el usuario se utilizan directamente en la consulta SQL sin ser validados correctamente, lo que podría permitir a un atacante ejecutar comandos maliciosos y comprometer la base de datos.
- **Falta de validaciones avanzadas:** Aunque verifica que los campos no estén vacíos, no asegura que los datos tengan el formato correcto. Por ejemplo:
 - El número de parqueadero debería ser un número válido.
 - La cédula debería cumplir con un formato específico.
- **Mensajes de error generales:** Aunque se avisa al usuario si falta información, los mensajes podrían ser más específicos y claros para mejorar la experiencia del usuario.



```
430 eparar=null;
431 ng.getText().isEmpty() == true){
432     eError+ " Nombre no puede estar vacio\n ";
433 }
434 .isEmpty() == true){
435     eError+ " Nombre no puede estar vacio\n ";
436 }
437 tText().isEmpty() == true){
438     eError+ " Nombre no puede estar vacio\n ";
439 }
440 ().isEmpty() == true){
441     eError+ " Nombre no puede estar vacio\n ";
442 }
443 pty() == true){
444     to usuarios_parking "
445         numero_parking,           placa,           propietario,           cedula) values"
446         xtNumero_deparking.getText() + "," + xtPlaca.getText() + "," + xtPropietario.getText() + "," + xtCedula.getText() + ")";
447     .prepareStatement(query);
448     cuteUpdate();
449     ;
450 }
451 on e){
452
453
454 }
```

Análisis del bloque de código:

Preparación de variables:

- Se inicializa un objeto para preparar la ejecución de consultas y una variable para construir un mensaje de error (eError).

Validaciones de campos:

- Cada campo (como el número de parqueadero, placa, propietario y cédula) es verificado para asegurarse de que no estén vacíos.
- Si un campo está vacío, se añade un mensaje de error indicando que el dato no puede estar vacío.

Condición para proceder:

- Si no se encontraron errores (es decir, eError sigue vacío), se procede a construir una consulta SQL para insertar los datos en una tabla llamada usuarios_parking.

Construcción y ejecución de la consulta:

- La consulta SQL se construye usando los valores ingresados en los campos de texto.
- Luego, se ejecuta esta consulta utilizando un objeto preparado para interactuar con la base de datos.

Manejo de errores:

- Si ocurre algún error durante la ejecución de la consulta, este es capturado en un bloque de manejo de excepciones para evitar que el programa falle inesperadamente.

Problemas identificados:

Riesgo de inyección SQL:

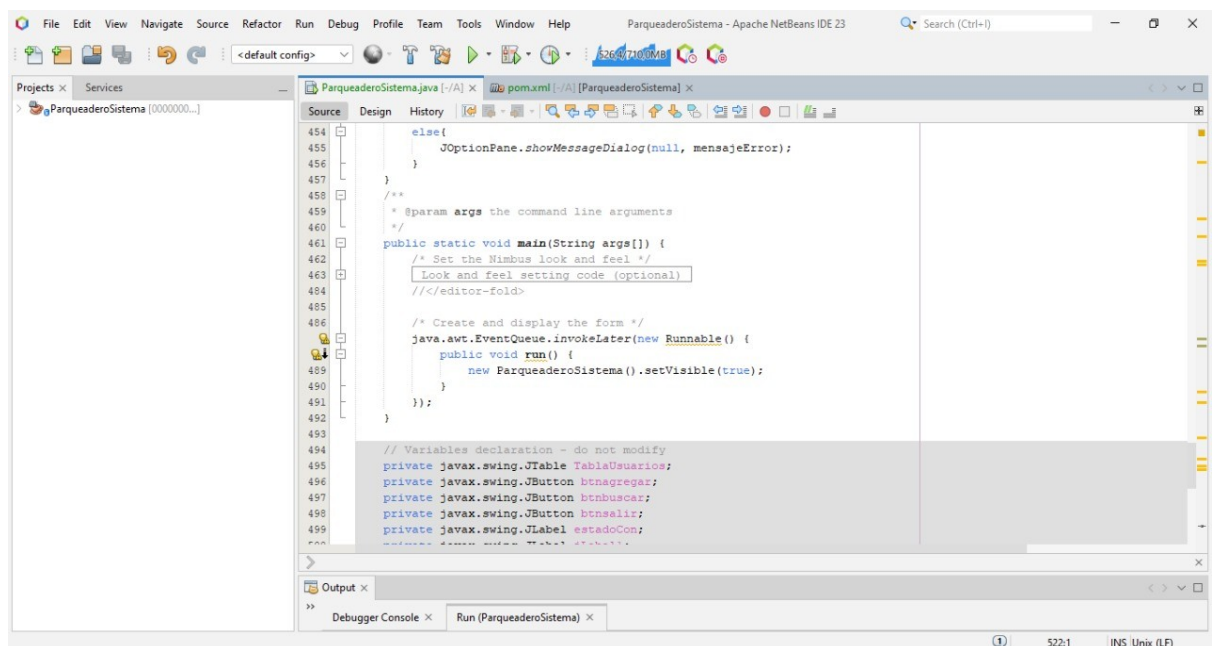
- La construcción de la consulta utiliza directamente los valores ingresados por el usuario, lo cual es vulnerable a ataques de inyección SQL. Esto podría permitir a un atacante ejecutar comandos maliciosos en la base de datos.

Mensajes de error básicos:

- Si hay un error, el usuario solo recibe un mensaje genérico. Sería ideal mostrar mensajes más descriptivos que indiquen qué ocurrió.

Validación limitada:

- Solo se valida que los campos no estén vacíos. No se verifica que los datos tengan el formato correcto, por ejemplo:
- El número de parqueadero debería ser un número.
- La placa podría requerir un formato alfanumérico específico.



Partes importantes del código y la interfaz:

1. Clase principal (public static void main):

- Es el punto de entrada del programa donde se inicializa la interfaz gráfica.
- Utiliza el método `java.awt.EventQueue.invokeLater` para garantizar que la GUI (interfaz gráfica de usuario) se ejecute en el hilo de eventos de Swing.
- Dentro de `run()`, se crea una nueva instancia de `ParqueaderoSistema` y se establece como visible llamando a `.setVisible(true)`.

2. Configuración de apariencia (comentado):

- Incluye un comentario que menciona el uso de Nimbus, una de las apariencias disponibles para las interfaces gráficas Swing. Sin embargo, el código para configurarlo está comentado.

3. Declaraciones de variables:

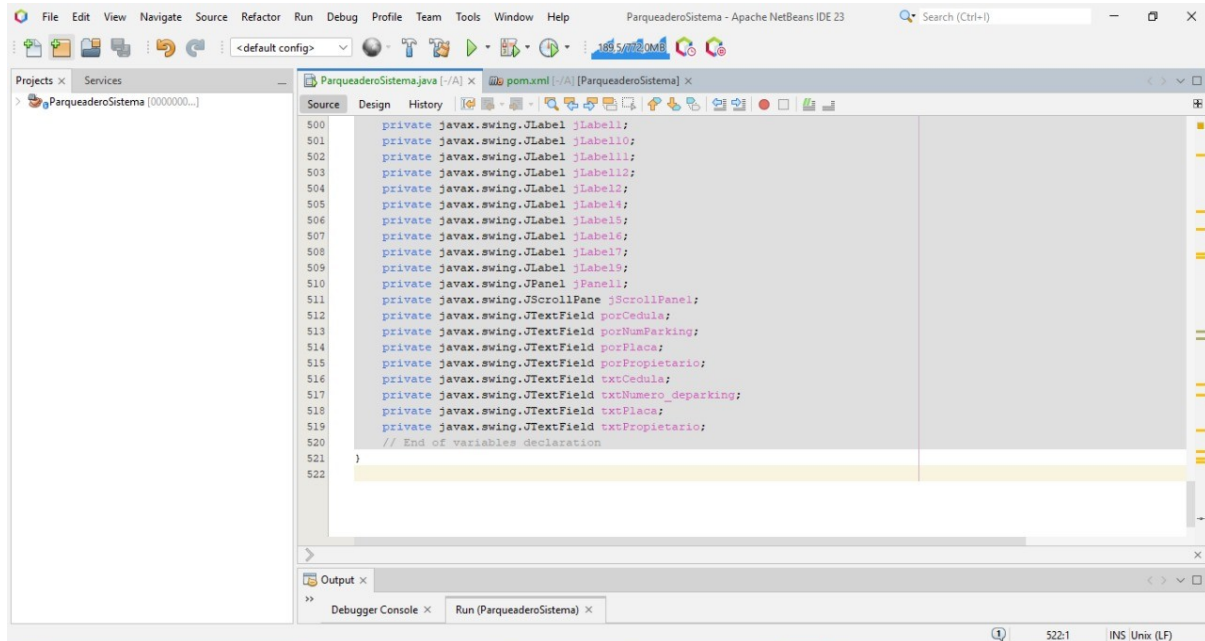
- Hay una sección de código al final con variables privadas relacionadas con elementos de Swing, como `JTable`, `JButton` y `JLabel`.
- Estas variables parecen representar componentes de la interfaz gráfica del sistema, como una tabla (`TablaUsuarios`), botones (`btnagregar`, `btnbuscar`, etc.), y etiquetas (`estadoCon`).

IDE (NetBeans):

- En la barra lateral izquierda se ve que el proyecto se llama **ParqueaderoSistema**.
- Se utilizan herramientas como el botón de depuración (debug) y de ejecución (run), que se encuentran en la barra superior.
- El área inferior tiene paneles para la consola de salida y depuración, lo que facilita el monitoreo de errores y la ejecución del programa.

Marcadores y advertencias:

- En el margen derecho del editor se aprecian indicadores de color amarillo que podrían representar advertencias (warnings) en el código. Estas podrían ser sobre prácticas no óptimas, código no utilizado o potenciales errores.



Detalles del código:

1. Variables JLabel:

- Se han declarado múltiples etiquetas (JLabel), como `jLabel1`, `jLabel2`, ..., hasta `jLabel17`.
- Estas etiquetas probablemente se utilizan para mostrar texto estático en la interfaz gráfica, como nombres de campos, encabezados, o instrucciones.

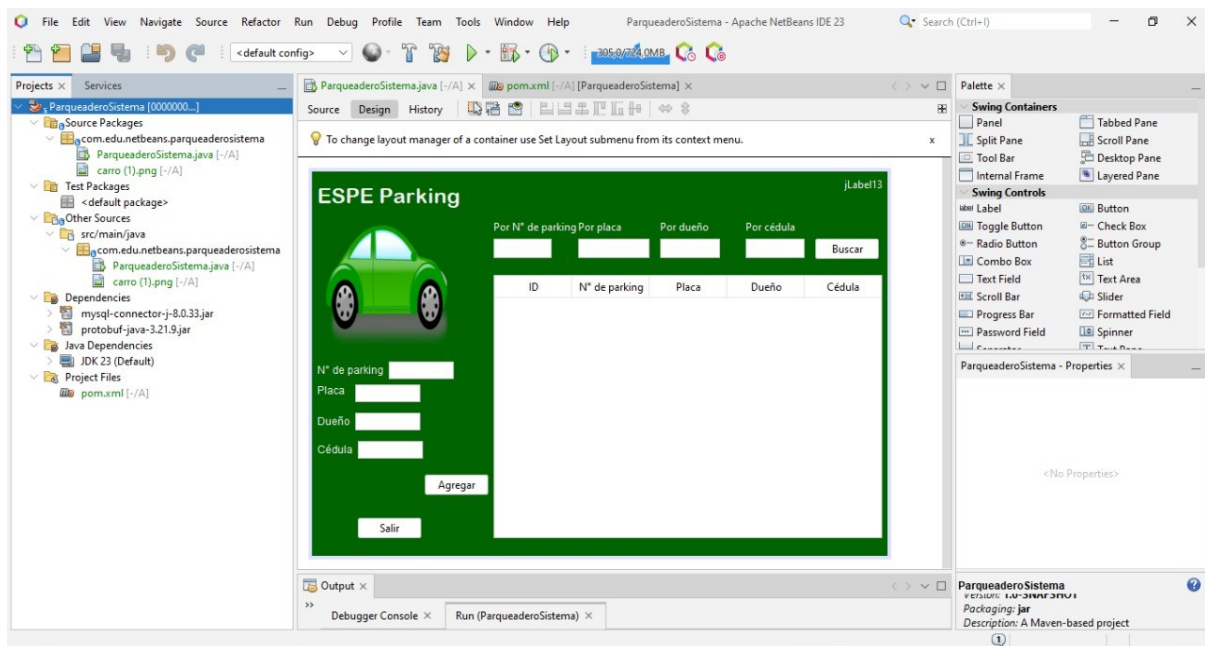
2. Paneles y componentes auxiliares:

Hay una variable de tipo `JPanel` (`jPanel1`), que se usa normalmente para organizar componentes en la GUI.

También se incluye un `JScrollPane` (`jScrollPane1`), que permite añadir barras de desplazamiento a elementos como tablas o listas.

3. Campos de texto (JTextField):

- Se observa la declaración de varios campos de texto (JTextField) para capturar datos como:
- porCedula (posiblemente para buscar por cédula).
- porNumParking (buscar por número de estacionamiento).
- porPlaca (buscar por placa).
- porPropietario (buscar por propietario).



Elementos claves:

Proyecto en NetBeans:

- El proyecto está nombrado como ParqueaderoSistema.
- Se observa que utiliza dependencias como mysql-connector-java y protobuf-java, lo que sugiere que la aplicación interactúa con una base de datos MySQL.

Diseño de la Interfaz:

- La ventana principal del sistema tiene un fondo verde con una imagen de un auto en el encabezado.

Contiene campos para:

- N° de parking.
- Placa.
- Dueño.
- Cédula.

Botones:

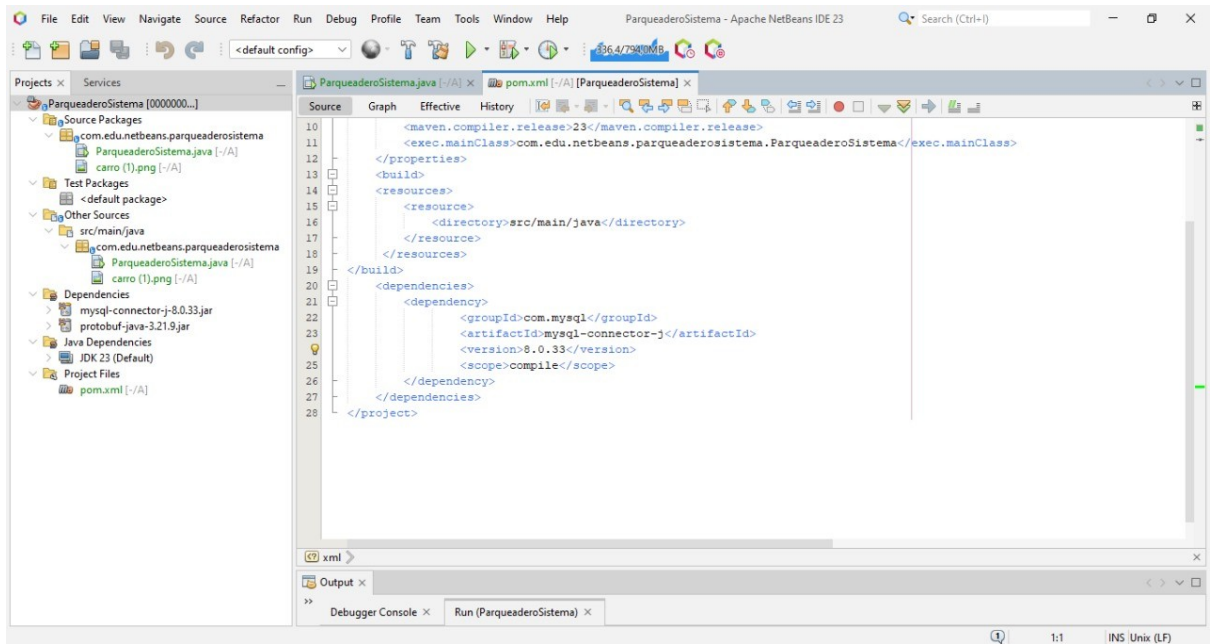
- Hay un botón para agregar información al parqueadero y otro para salir.
- Una sección para buscar registros con filtros (por número de parking, placa, dueño o cédula).
- Una *tabla* donde se mostrarán los registros, con columnas como ID, N° de parking, Placa, Dueño y Cédula.

Herramientas:

- La paleta de componentes Swing a la derecha.
- Propiedades de la interfaz seleccionada (ParqueaderoSistema).
- El proyecto usa Maven, como se nota en el archivo pom.xml.

Propósito del Sistema:

- Gestionar y registrar datos relacionados con un parqueadero.
- Permitir la búsqueda y visualización de los datos de manera organizada.



Clase Principal:

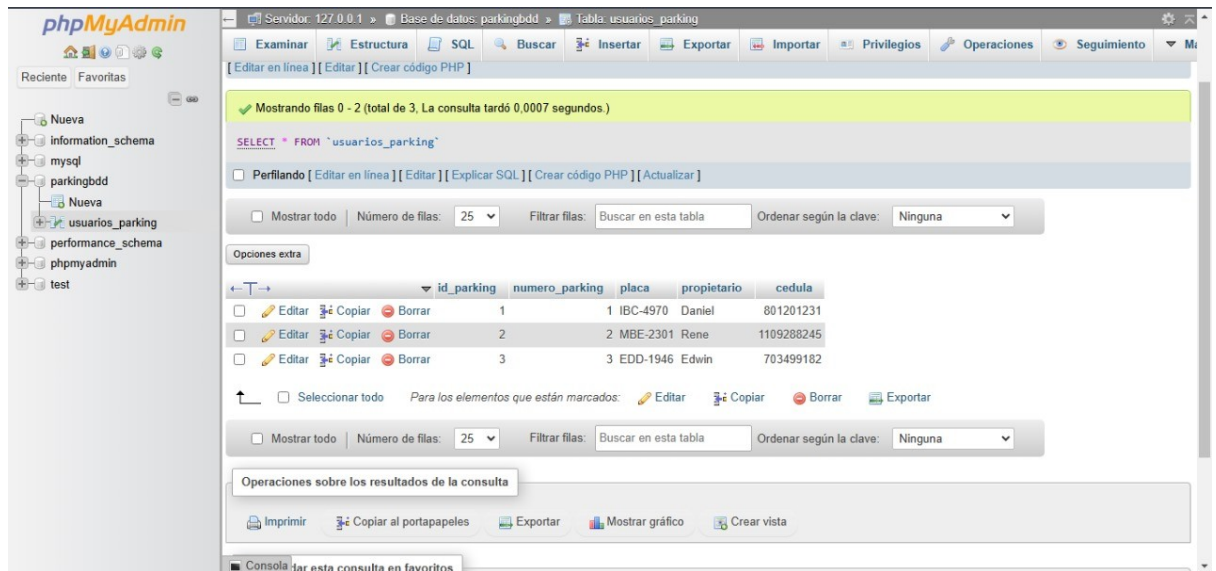
- El archivo indica que la clase principal del proyecto (la que contiene el método main) es **ParqueaderoSistema**, ubicada dentro del paquete **com.edu.netbeans.parqueaderosistema**.

Dependencias:

- MySQL Connector:** Este archivo incluye una librería que permite que el programa interactúe con una base de datos MySQL.
- Protobuf:** Otra librería añadida, utilizada para manejar datos estructurados de manera eficiente (por ejemplo, serialización de datos).

Recursos del Proyecto:

- Especifica que el código fuente y los archivos de recursos del proyecto están ubicados en la carpeta estándar **src/main/java**.



A continuación, vamos a describir un poco de lo que podemos observar en esta imagen

Base de datos seleccionada:

- La base de datos activa es parkingbdd.

Tabla seleccionada:

- Dentro de esta base de datos, se ha seleccionado la tabla usuarios_parking.

Consulta SQL ejecutada:

- La consulta SQL ejecutada es: `SELECT * FROM usuarios_parking`, que recupera todas las filas y columnas de la tabla usuarios_parking.

Contenido de la tabla:

- La tabla contiene cinco columnas: id_parking, numero_parking, placa, propietario, y cedula.

Hay tres registros o filas en la tabla:

- El registro 1 tiene un numero_parking de 1, la placa es "IBC-4970", el propietario es "Daniel", y la cedula es "801201231".
- El registro 2 tiene un numero_parking de 2, la placa es "MBE-2301", el propietario es "Rene", y la cedula es "1109288245".
- El registro 3 tiene un numero_parking de 3, la placa es "EDD-1946", el propietario es "Edwin", y la cedula es "703499182".

Opciones disponibles:

- Para cada registro, se pueden realizar acciones como **Editar**, **Copiar**, o **Borrar**.
- Hay opciones adicionales para ordenar, filtrar o exportar los datos.

Estructura de navegación:

- En el lado izquierdo de la pantalla, se puede observar la lista de bases de datos y tablas. Algunas bases de datos mostradas son: information_schema, mysql, performance_schema, y phpmyadmin.

5. CONCLUSIONES

El desarrollo del sistema de gestión de parqueadero permitió consolidar habilidades en programación orientada a objetos, diseño de interfaces gráficas y manejo de bases de datos, esenciales para el desarrollo de software eficiente y funcional.

La integración de diferentes tecnologías y herramientas en un proyecto práctico fomenta el aprendizaje autónomo y refuerza competencias técnicas y metodológicas aplicables a la solución de problemas reales.

6. RECOMENDACIONES

Antes de iniciar el desarrollo del sistema, es recomendable planificar detalladamente el diseño mediante diagramas UML, asegurando que la estructura del software sea clara y eficiente desde el inicio.

Durante la implementación, se sugiere realizar pruebas constantes de cada funcionalidad para identificar y corregir errores a tiempo, garantizando un producto final estable y funcional.

7. BIBLIOGRAFÍA

DuBois, P. (2013). *MySQL: The complete reference*. McGraw-Hill Education.

Oracle Corporation. (2024). *MySQL reference manual*. Recuperado de <https://dev.mysql.com/doc/>

phpMyAdmin Developers. (2024). *phpMyAdmin documentation*. Recuperado de <https://docs.phpmyadmin.net/>

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database system concepts* (7^a ed.). McGraw-Hill Education.

Beaulieu, A. (2009). *Learning SQL*. O'Reilly Media.

8. ANEXOS

Link de Github

Samir Samande : <https://github.com/sssamande/actividad-autonomo>

Steven Guerrero: <https://github.com/dsguerrero4/index-hub.git>