Materia: Seminario de actualización en programación.

Profesores: Callejas, Marisa; Gutiérrez, Julio Marcelo.

Alumno: Ferrero Santiago

Legajo: VINF011991

Índice:

MVC Pág. 3\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_6

DER Pág. 7\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_8

DAO Pág. 8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_10

Patrones adicionales Pág. 10\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_13

**Componentes del Patrón MVC**

1. **Modelo (Model)**:
   * **Responsabilidad**: Representa la lógica de negocio y el estado de la aplicación. El modelo gestiona los datos, la lógica de negocio y las reglas del dominio de la aplicación.
   * **Ejemplo**: Clases como Cliente, Vehiculo, Turno, Mecanico, y FichaMecanica representan el modelo en el sistema. Estas clases contienen la estructura de los datos y se comunican con la base de datos a través de los DAOs.
   * **Función**: El modelo es independiente de la interfaz de usuario, es decir, puede funcionar sin que la vista ni el controlador interfieran en sus operaciones. Por ejemplo, puede almacenar datos de clientes o vehículos y actualizar estos datos sin interactuar directamente con la vista.
2. **Vista (View)**:
   * **Responsabilidad**: La vista se encarga de la interfaz de usuario y de mostrar la información al usuario. Recibe datos del modelo y los presenta en una forma comprensible (por ejemplo, una interfaz gráfica o una consola).
   * **Ejemplo**: En una implementación gráfica, la vista podría ser una interfaz de usuario (UI) que muestra listas de clientes, vehículos, o mecánicos, así como formularios para registrar nuevos datos. En el caso de la consola, se tiene un menú que interactúa con el usuario.
   * **Función**: La vista solo muestra la información y no debe realizar operaciones de lógica de negocio. También depende del controlador para enviar actualizaciones o peticiones.
3. **Controlador (Controller)**:
   * **Responsabilidad**: Actúa como intermediario entre la vista y el modelo. Procesa las entradas del usuario (por ejemplo, hacer clic en un botón) y las convierte en instrucciones para el modelo o la vista.
   * **Ejemplo**: El controlador en este caso podría ser un conjunto de clases o métodos que manejan las interacciones con el usuario, como crear, actualizar o eliminar registros de clientes o vehículos en respuesta a la entrada del usuario en la consola o UI.
   * **Función**: El controlador responde a las acciones del usuario, invoca métodos del modelo para modificar datos y actualiza la vista para reflejar los cambios.

**Flujo de Trabajo en MVC**

Para entender mejor cómo funciona el patrón MVC, veamos el flujo de trabajo típico cuando un usuario interactúa con la aplicación:

1. **Interacción del Usuario**:
   * El usuario realiza una acción (por ejemplo, selecciona "Agregar Cliente" en la interfaz de usuario).
2. **Controlador Procesa la Acción**:
   * La vista notifica al controlador sobre esta acción. El controlador interpreta la solicitud del usuario y llama al modelo para que realice la acción correspondiente (en este caso, crear un nuevo cliente).
3. **Actualización del Modelo**:
   * El controlador envía la información al modelo para crear un nuevo cliente. El modelo procesa la solicitud, aplica la lógica de negocio y, si es necesario, guarda el nuevo cliente en la base de datos.
4. **Notificación y Actualización de la Vista**:
   * Después de que el modelo ha actualizado los datos, el controlador o el modelo puede notificar a la vista que hay nueva información para mostrar.
   * La vista obtiene los datos actualizados del modelo (por ejemplo, la lista de clientes que ahora incluye el nuevo cliente) y actualiza la interfaz para reflejar los cambios.

**Modelo (Cliente)**: Representa la estructura de datos de un cliente y contiene solo atributos y métodos para acceder a ellos (getters y setters).

**Vista (ClienteView)**: Presenta los datos al usuario. En este caso, muestra la información del cliente y mensajes generales en la consola.

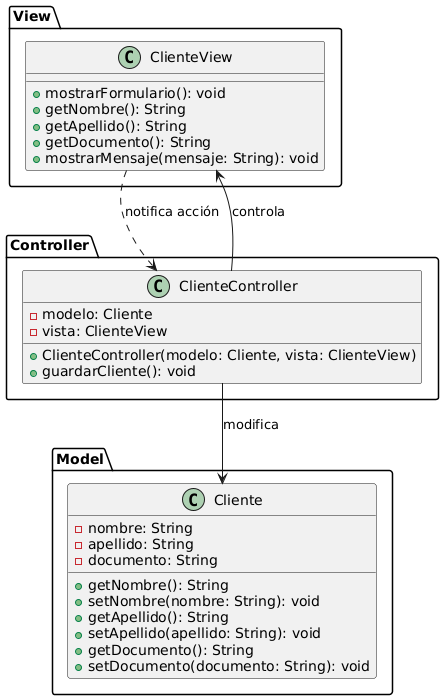
**Controlador (ClienteController)**: Actúa como intermediario entre la vista y el modelo. Contiene métodos como agregarCliente y mostrarClientes, que toman la entrada del usuario (a través de Main), actualizan el modelo (ClienteDAO), y luego notifican a la vista para actualizar la salida al usuario.

**Ventajas del Patrón MVC**

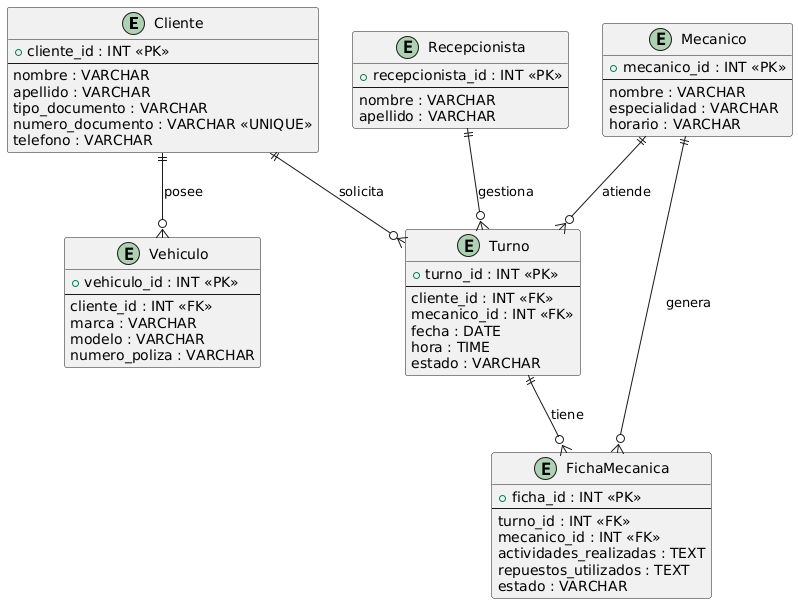
**Separación de Responsabilidades**: Cada componente tiene su rol específico, lo que facilita el mantenimiento y la escalabilidad.

**Reutilización de Código**: La lógica de negocio en el modelo puede ser utilizada por diferentes vistas o interfaces (consola, web, móvil).

**Flexibilidad**: Cambiar la vista o la lógica de negocio no afecta a los demás componentes, lo que facilita modificar y expandir la aplicación.



DER:



El Diagrama Entidad-Relación (DER) representa las tablas de la base de datos y sus relaciones, basándose en las entidades del sistema:

* **Entidad Cliente**: Representa a los clientes del sistema con atributos como id, nombre, apellido, y documento. La relación con Vehiculo es de uno a muchos (un cliente puede tener varios vehículos).
* **Entidad Vehiculo**: Contiene información sobre los vehículos, incluyendo el cliente\_id como clave foránea que se relaciona con Cliente. Tiene atributos como marca, modelo, y numero\_poliza.
* **Entidad Mecanico**: Representa a los mecánicos del sistema, con atributos como nombre, especialidad, y horario. Se relaciona con Turno y FichaMecanica.
* **Entidad Turno**: Representa los turnos asignados a los clientes y mecánicos. Contiene cliente\_id y mecanico\_id como claves foráneas que se relacionan con Cliente y Mecanico, respectivamente. Tiene atributos como fecha, hora, y estado.
* **Entidad FichaMecanica**: Almacena el historial de actividades y repuestos utilizados en cada turno. Tiene una relación con Turno y Mecanico mediante las claves foráneas turno\_id y mecanico\_id. Incluye atributos como actividades\_realizadas, repuestos\_utilizados, y estado.

DAO:

**Patrón de Diseño DAO (Data Access Object)**

El patrón **DAO (Data Access Object)** se utiliza para abstraer y encapsular el acceso a la base de datos, permitiendo que el resto de la aplicación no dependa de detalles de implementación específicos de la base de datos. En este proyecto, el patrón DAO tiene las siguientes ventajas:

* **Separación de Lógica de Acceso a Datos y Lógica de Negocio**: Cada entidad del sistema tiene una clase DAO dedicada que se encarga de todas las operaciones de acceso a datos. Esto permite que las clases de negocio (como Cliente, Vehiculo, Turno, Mecanico, y FichaMecanica) estén desacopladas de las operaciones de la base de datos, mejorando la modularidad.
* **Operaciones CRUD (Create, Read, Update, Delete)**: Cada DAO implementa operaciones básicas de CRUD que permiten crear, leer, actualizar y eliminar registros de la base de datos.

**Ejemplo de Implementación de DAO**

Para cada entidad del sistema, creamos una interfaz y su respectiva implementación DAO:

* **Interfaz DAO**: Define los métodos CRUD. Ejemplo:

public interface ClienteDAO {

void agregarCliente(Cliente cliente);

Cliente obtenerClientePorId(int id);

List<Cliente> obtenerTodosLosClientes();

void actualizarCliente(Cliente cliente);

void eliminarCliente(int id);

}

Implementación DAO: Implementa los métodos definidos en la interfaz y utiliza JDBC para interactuar con la base de datos. Ejemplo:

public class ClienteDAOImpl implements ClienteDAO {

@Override

public void agregarCliente(Cliente cliente) {

String sql = "INSERT INTO Cliente (nombre, apellido, documento) VALUES (?, ?, ?)";

try (Connection conn = ConexionBD.getInstancia().getConnection();

PreparedStatement stmt = conn.prepareStatement(sql)) {

stmt.setString(1, cliente.getNombre());

stmt.setString(2, cliente.getApellido());

stmt.setString(3, cliente.getDocumento());

stmt.executeUpdate();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

El DAO para cada entidad (Cliente, Vehiculo, Turno, Mecanico, FichaMecanica) sigue el mismo esquema, con métodos específicos de acuerdo con los atributos y las relaciones de cada entidad.

**Patrones de Diseño Adicionales**

Utilizamos varios patrones de diseño adicionales para mejorar la estructura y funcionalidad del proyecto.

**a. Singleton para la Conexión a la Base de Datos**

La clase ConexionBD implementa el patrón **Singleton**, lo que garantiza que solo exista una única instancia de conexión a la base de datos en toda la aplicación. Este patrón es útil para evitar múltiples conexiones no controladas y reducir el uso de recursos.

public class ConexionBD {

private static ConexionBD instancia;

private Connection connection;

private ConexionBD() throws SQLException {

this.connection = DriverManager.getConnection(URL, USER, PASSWORD);

}

public static ConexionBD getInstancia() throws SQLException {

if (instancia == null || instancia.getConnection().isClosed()) {

instancia = new ConexionBD();

}

return instancia;

}

}

**Factory Method para Crear DAO**

El **Factory Method** permite crear instancias de los DAOs sin que el código cliente dependa de las clases concretas. En este proyecto, DAOFactory define un método abstracto crearClienteDAO() que luego implementa ClienteDAOFactory para retornar la instancia de ClienteDAOImpl.

public abstract class DAOFactory {

public abstract ClienteDAO crearClienteDAO();

public static DAOFactory getDAOFactory() {

return new ClienteDAOFactory();

}

}

Command para Ejecución de Operaciones CRUD

Implementamos el Command Pattern para encapsular cada operación CRUD (como agregar, actualizar o eliminar) en una clase Command. Por ejemplo, CrearClienteCommand representa la acción de agregar un cliente.

public class CrearClienteCommand implements Command {

private ClienteDAO clienteDAO;

private Cliente cliente;

public CrearClienteCommand(ClienteDAO clienteDAO, Cliente cliente) {

this.clienteDAO = clienteDAO;

this.cliente = cliente;

}

@Override

public void ejecutar() {

clienteDAO.agregarCliente(cliente);

}

}

**Strategy para Generación de Informes**

El **Strategy Pattern** permite seleccionar el algoritmo o enfoque para la generación de informes en tiempo de ejecución. Definimos una interfaz GenerarInformeStrategy con estrategias específicas como InformeMensualStrategy e InformeAnualStrategy.

public class GeneradorDeInforme {

private GenerarInformeStrategy strategy;

public void setStrategy(GenerarInformeStrategy strategy) {

this.strategy = strategy;

}

public void generar() {

if (strategy != null) {

strategy.generarInforme();

}

}

}