Informe Final del Proyecto:
Sistema de Gestión de Telefonía Móvil Estudiantil Mov-UTPL

UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA

Ingeniería en ciencias de la computación.

TRABAJO FINAL DE II BIMESTRE

ASIGNATURA:

PROGRAMACION ORIENTADA A OBJETOS

TEMA:

Sistema de Gestión de Clientes y Facturación para Telefonía Móvil Estudiantil: Mov-UTPL

INTEGRANTES DEL GRUPO:

- CRISTIAN GRANDA
- CARLOS PIERDRA

DOCENTE:

WAGNER BUSTAMANTE

FECHA DE ENTREGA: 30 DE JULIO DEL 2025

1. Análisis del Problema / Solución

1.1. Problemática

La carrera de Telecomunicación de la UTPL ha impulsado un proyecto de comunicación móvil para su comunidad universitaria, disponiendo de la infraestructura tecnológica necesaria. Sin embargo, el proyecto se encuentra detenido debido a la carencia de un Sistema de Gestión de sus clientes y facturación, que debe considerar diversos planes celulares y la información específica de cada cliente.

1.2. Requisitos Identificados

Del enunciado de la problemática, se desprenden los siguientes requisitos funcionales y no funcionales clave para el sistema Mov-UTPL:

- **Gestión de Clientes:**
  - Almacenar datos básicos del cliente: nombres, pasaporte/cédula, ciudad.
  - **Atributos Adicionales:** Se han incorporado email y dirección para enriquecer el perfil del cliente.
  - Facilitar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para los clientes.
- **Gestión de Dispositivos Móviles:**
  - Asociar dispositivos móviles a clientes, incluyendo: marca, modelo, y número de celular.

- **Gestión de Planes Móviles (con Herencia y Polimorfismo):**
    - El sistema debe soportar cuatro tipos específicos de planes postpago, cada uno con atributos y lógicas de cálculo de pago mensual distintas:
        - PlanPostPagoMinutosMegasEconomico: minutos, costo minutos, megas (en gigas), costo por giga, porcentaje de descuento.
        - PlanPostPagoMinutos: minutos nacionales, costo minuto nacional, minutos internacionales, costo minuto internacional.
        - PlanPostPagoMegas: megas (en gigas), costo por giga, tarifa base.
        - PlanPostPagoMinutosMegas: minutos, costo minutos, megas (en gigas), costo por giga.
    - Facilitar operaciones CRUD para los planes.
    - **Restricción:** Un cliente puede disponer de **no más de 2 tipos de planes** simultáneamente.
- **Generación y Gestión de Facturas:**
    - Generación de facturas por cliente/plan, calculando el total a pagar en base a los planes activos del cliente.
    - Almacenar información de factura: ID, ID de cliente, fecha de emisión, total a pagar, estado (Pendiente/Pagada).

    - Permitir la visualización del detalle de una factura (qué planes la componen y sus costos).
    - Permitir marcar facturas como "Pagada".
- **Base de Datos:**
    - Utilizar **SQLite** como motor de base de datos para todo el sistema (clientes, planes, facturas, dispositivos, detalles de factura).
- **Arquitectura y Principios de Diseño:**
    - Aplicar la arquitectura **Modelo-Vista-Controlador (MVC)**.
    - Aplicar los pilares de la **Programación Orientada a Objetos (POO)**: Encapsulamiento, Herencia y Polimorfismo.
- **Control de Versiones:**
    - Utilizar un repositorio **GIT** para la gestión del código fuente.

1.3. Entidades Identificadas y sus Atributos Clave

Para la solución del problema, se identificaron las siguientes entidades principales, que se modelarán como clases en el sistema:

Entidad	**Atributos Clave**

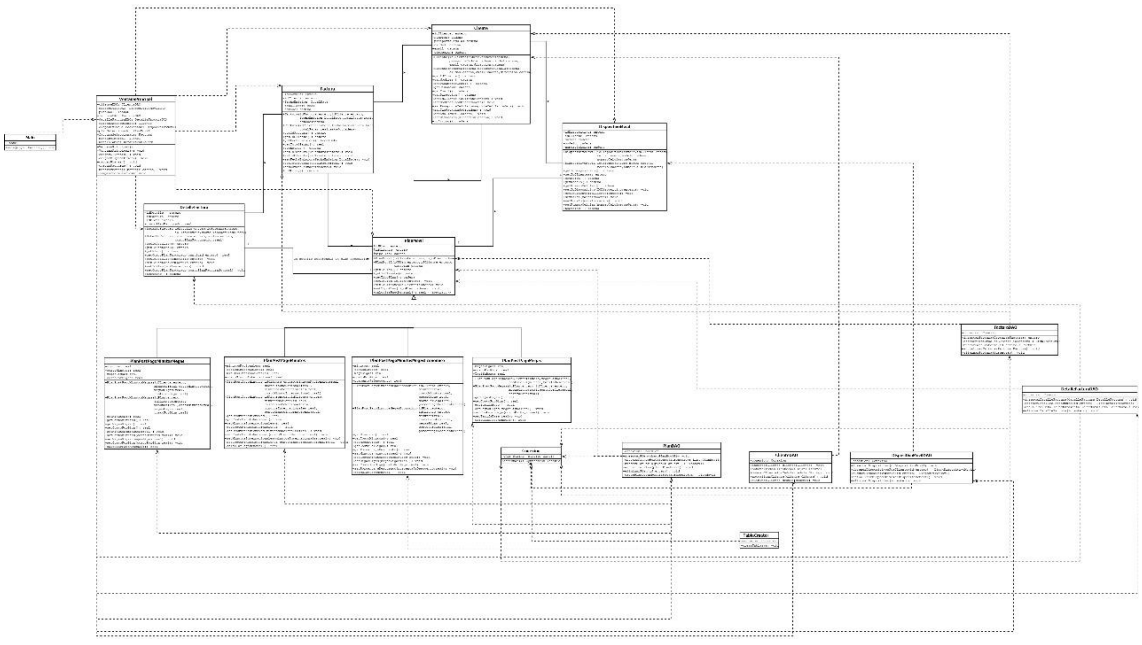
Cliente	idCliente, nombres, pasaporteCedula, ciudad, email, direccion
PlanMovil (Abstracta)	idPlan, idCliente, tipoPlan
PlanPostPagoMinutosMegasEconomico	minutos, costoMinutos, megasGigas, costoPorGiga, porcentajeDescuento
PlanPostPagoMinutos	minutosNacionales, costoMinutoNacional, minutosInternacionales, costoMinutoInternacional
PlanPostPagoMegas	megasGigas, costoPorGiga, tarifaBase
PlanPostPagoMinutosMegas	minutos, costoMinutos, megasGigas, costoPorGiga
DispositivoMovil	idDispositivo, idCliente, marca, modelo, numeroCelular

Factura	idFactura, idCliente, fechaEmision, totalPagar, estado
DetalleFactura	idDetalle, idFactura, idPlan, costoPlanFacturado

2. Diseño del Sistema (UML)

El diseño del sistema Mov-UTPL se basa en la arquitectura MVC y los principios de la Programación Orientada a Objetos (POO), incluyendo encapsulamiento, herencia y polimorfismo. A continuación, se presenta el diagrama de clases UML que representa la estructura estática del sistema.

2.1. Diagrama de Clases General



2.2. Detalle de Clases y Relaciones por Paquete

2.2.1. Paquete modelo (Capa del Modelo - Entidades de Negocio)

Este paquete contiene las clases que representan las entidades fundamentales del negocio, encapsulando sus datos y comportamientos básicos.

- **Clase: Cliente**
  - **Propósito:** Representa la información básica de un cliente/estudiante.
  - **Atributos:** idCliente, nombres, pasaporteCedula, ciudad, email, direccion.
  - **Métodos:** Constructores, getters y setters para todos los atributos.
  - **Relaciones:**
    - **Asociación (Agregación) con DispositivoMovil:** Un Cliente puede tener cero a muchos DispositivoMovil (1 --o *). Un DispositivoMovil pertenece a un Cliente.
    - **Asociación (Agregación) con PlanMovil:** Un Cliente puede tener cero a dos PlanMovil (1 --o 0..2). Un PlanMovil pertenece a un Cliente.
    - **Asociación (Agregación) con Factura:** Un Cliente puede tener cero a muchas Factura (1 --o *). Una Factura pertenece a un Cliente.

- **Clase Abstracta: PlanMovil**
  - **Propósito:** Define la estructura y el comportamiento común para todos los tipos de planes móviles. Es la superclase abstracta de la jerarquía de planes.
  - **Atributos:** #idPlan, #idCliente, #tipoPlan.
  - **Métodos:** Constructores, getters, setters, y el método + _calcularPagoMensual(): double_ (abstracto).
  - **Principios POO:** Fundamento de la **Herencia** y el **Polimorfismo**.
  - **Relaciones:**
    - **Herencia de PlanPostPagoMinutosMegasEconomico, PlanPostPagoMinutos, PlanPostPagoMegas, PlanPostPagoMinutosMegas:** Estas cuatro clases son subclases que heredan de PlanMovil (Subclase --|> PlanMovil).
    - **Dependencia con DetalleFactura:** DetalleFactura hace referencia a un PlanMovil por su ID (DetalleFactura ---> PlanMovil).
- **Clases Concretas de Planes (Subclases de PlanMovil):**
  - PlanPostPagoMinutosMegasEconomico
  - PlanPostPagoMinutos
  - PlanPostPagoMegas
  - PlanPostPagoMinutosMegas

- **Propósito:** Cada una representa un tipo específico de plan con su lógica de cálculo de pago mensual.
- **Atributos:** Específicos para cada plan (e.g., minutos, costoMinutos, megasGigas, porcentajeDescuento para el plan Económico).
- **Métodos:** Constructores, getters/setters, y la implementación concreta de + calcularPagoMensual(): double.
- **Principios POO:** Demuestran **Herencia** (de PlanMovil) y **Polimorfismo** (al sobrescribir calcularPagoMensual()).
- **Relaciones:**
  - **Herencia de PlanMovil** (ya mencionada).
  - **Dependencia con PlanDAO:** PlanDAO necesita conocer estas clases para instanciarlas y manipularlas correctamente.

- **Clase: DispositivoMovil**
  - **Propósito:** Representa un dispositivo móvil asociado a un cliente.
  - **Atributos:** idDispositivo, idCliente, marca, modelo, numeroCelular.
  - **Métodos:** Constructores, getters y setters.
  - **Principios POO: Encapsulamiento**.
  - **Relaciones:**

        - **Asociación (Agregación) con Cliente:** Un DispositivoMovil pertenece a un Cliente (DispositivoMovil --o 1 Cliente).
- **Clase: Factura**
    - **Propósito:** Representa una factura generada para un cliente.
    - **Atributos:** idFactura, idCliente, fechaEmision, totalPagar, estado.
    - **Métodos:** Constructores, getters y setters.
    - **Principios POO: Encapsulamiento**.
    - **Relaciones:**
        - **Asociación (Agregación) con Cliente:** Una Factura pertenece a un Cliente (Factura --o 1 Cliente).
        - **Composición con DetalleFactura:** Una Factura está compuesta por uno o muchos DetalleFactura (Factura --* * DetalleFactura). La vida de los detalles depende de la factura.
- **Clase: DetalleFactura**
    - **Propósito:** Representa una línea de detalle dentro de una factura, vinculando un plan específico y registrando su costo al momento de la facturación.
    - **Atributos:** idDetalle, idFactura, idPlan, costoPlanFacturado.
    - **Métodos:** Constructores, getters y setters.

- **Principios POO: Encapsulamiento**.
- **Relaciones:**
  - **Composición con Factura:** Un DetalleFactura es parte de una Factura (DetalleFactura --* 1 Factura).
  - **Dependencia con PlanMovil:** Un DetalleFactura hace referencia al idPlan de un PlanMovil (DetalleFactura ---> PlanMovil).

2.2.2. Paquete conexion (Capa de Conectividad a la Base de Datos)

- **Clase: Conexion**
  - **Propósito:** Gestiona la conexión con la base de datos SQLite.
  - **Atributos:** - URL: String {final} (la ruta a la DB).
  - **Métodos:** + conectar(): Connection {static}.
  - **Principios POO: Encapsulamiento** de los detalles de conexión.
  - **Relaciones:**
    - **Dependencia con todos los DAOs (ClienteDAO, DispositivoMovilDAO, PlanDAO, FacturaDAO, DetalleFacturaDAO) y TablaCreator:** Todas estas clases usan el método conectar() de Conexion para obtener una

conexión a la base de datos (DAO ---> Conexion).

2.2.3. Paquete dao (Capa del Modelo - Acceso a Datos)

Este paquete contiene las clases DAO (Data Access Objects) que interactúan directamente con la base de datos, implementando las operaciones CRUD para las entidades del modelo.

- **Clase: ClienteDAO**
  - **Propósito:** Proporciona los métodos para gestionar la persistencia de objetos Cliente.
  - **Métodos:** Implementa las operaciones **CRUD** para Cliente: insertarCliente (Create), obtenerTodosClientes, buscarClientePorCedula (Read), actualizarCliente (Update), eliminarCliente (Delete).
  - **Principios POO: Encapsulamiento** de la lógica de acceso a datos.
  - **Relaciones:**
    - **Dependencia con Cliente:** Opera sobre objetos Cliente.
    - **Dependencia con Conexion:** Utiliza Conexion para establecer la comunicación con la DB.
- **Clase: DispositivoMovilDAO**
  - **Propósito:** Gestiona las operaciones de persistencia para objetos DispositivoMovil.

    - **Métodos:** Implementa las operaciones **CRUD** para DispositivoMovil.
    - **Relaciones:**
        - **Dependencia con DispositivoMovil**.
        - **Dependencia con Conexion**.
- **Clase: PlanDAO**
    - **Propósito:** Gestiona las operaciones de persistencia para objetos PlanMovil y sus subclases, aprovechando el polimorfismo.
    - **Métodos:** Implementa las operaciones **CRUD** para PlanMovil (y sus subclases): insertarPlan (Create), obtenerPlanesPorCliente, buscarPlanPorId (Read), actualizarPlan (Update), eliminarPlan (Delete). Incluye el método auxiliar crearPlanDesdeResultSet() para la instanciación polimórfica.
    - **Principios POO:** Utiliza intensivamente el **Polimorfismo** para manejar los diferentes tipos de planes de forma uniforme.
    - **Relaciones:**
        - **Dependencia con PlanMovil** y sus subclases.
        - **Dependencia con Conexion**.
- **Clase: FacturaDAO**
    - **Propósito:** Gestiona las operaciones de persistencia para objetos Factura.

    - **Métodos:** Implementa las operaciones **CRUD** para Factura.
    - **Relaciones:**
        - **Dependencia con Factura**.
        - **Dependencia con Conexion**.
- **Clase: DetalleFacturaDAO**
    - **Propósito:** Gestiona las operaciones de persistencia para objetos DetalleFactura.
    - **Métodos:** Implementa las operaciones **CRUD** para DetalleFactura.
    - **Relaciones:**
        - **Dependencia con DetalleFactura**.
        - **Dependencia con Conexion**.
- **Clase: TablaCreator**
    - **Propósito:** Clase utilitaria que asegura la creación inicial de la estructura de la base de datos (tablas) si no existen.
    - **Métodos:** + crearTablas(): void.
    - **Relaciones:**
        - **Dependencia con Conexion:** Utiliza Conexion para ejecutar las sentencias SQL de creación de tablas.

2.2.4. Paquete vista (Capa de la Vista y el Controlador)

Este paquete contiene la interfaz gráfica de usuario y la lógica que maneja las interacciones del usuario.

- **Clase: VentanaPrincipal**
  - **Propósito:** Es la ventana principal de la aplicación, sirviendo como la **Vista** (presentación de la UI) y el **Controlador** (manejo de eventos y lógica de interacción).
  - **Atributos:** Instancias de todos los DAOs, atributos para los objetos seleccionados (clienteSeleccionado, etc.), y numerosos componentes Swing (JTextField, JButton, JTable, etc.) que componen la interfaz.
  - **Métodos:**
    - Constructor que inicializa la UI.
    - Métodos auxiliares para la UI (createStyledButton, addPlanField, actualizarCamposPlanes).
    - Métodos de carga y limpieza de datos en la UI (cargarClientesEnComboBoxes, limpiarCamposCliente, etc.).
    - agregarListeners(): Configura los manejadores de eventos.
    - **Métodos de Acción (Controlador):** Todos los métodos que responden a las interacciones del usuario (e.g., guardarCliente, generarFactura, modificarPlan, eliminarDispositivo,

verDetalleFactura). Estos métodos son el corazón del controlador, ya que:

1. Capturan la entrada del usuario desde los componentes de la UI.
2. Realizan validaciones básicas.
3. Invocan los métodos apropiados en las clases DAO (Modelo) para manipular los datos en la base de datos.
4. Actualizan los componentes de la UI (Vista) para reflejar los cambios o mostrar mensajes al usuario.

- **Relaciones:**
  - **Dependencia con todos los DAOs (ClienteDAO, DispositivoMovilDAO, PlanDAO, FacturaDAO, DetalleFacturaDAO):** VentanaPrincipal invoca los métodos de los DAOs para realizar las operaciones de negocio y persistencia.
  - **Dependencia con las clases de Modelo (Cliente, DispositivoMovil, PlanMovil, Factura, DetalleFactura):** VentanaPrincipal crea y manipula objetos de estas clases para mostrar y obtener datos de la UI.

2.2.5. Clase Main (Punto de Entrada del Sistema)

- **Clase: Main**
  - **Propósito:** Es la clase que inicia la ejecución de toda la aplicación.
  - **Métodos:**
    - + main(args: String[]): void: El método principal.
  - **Relaciones:**
    - **Dependencia con TablaCreator:** Llama a TablaCreator.crearTablas() para asegurar que la base de datos esté inicializada.
    - **Dependencia con VentanaPrincipal:** Crea una instancia de VentanaPrincipal y la hace visible, iniciando la interfaz gráfica.

3. Codificación

La codificación del proyecto se ha estructurado en paquetes lógicos para adherirse a la arquitectura MVC y facilitar la comprensión y el mantenimiento.

3.1. Estructura de Paquetes

- **modelo**: Contiene las clases de las entidades de negocio.
- **dao**: Contiene las clases de acceso a datos (Data Access Objects).
- **conexion**: Contiene la clase para la conexión a la base de datos.
- **vista**: Contiene la clase de la interfaz gráfica de usuario.

3.2. Fragmentos de Código Relevantes y Documentados

A continuación, se presentan ejemplos de fragmentos de código que ilustran la implementación de los principios de POO y las operaciones CRUD.

3.2.1. Herencia y Polimorfismo (Paquete modelo)

La clase PlanMovil define la estructura base y el método abstracto calcularPagoMensual().

```java
package modelo;

public abstract class PlanMovil {
    protected int idPlan;
    protected int idCliente;
    protected String tipoPlan;

    public PlanMovil(int idCliente, String tipoPlan) {
        this.idCliente = idCliente;
        this.tipoPlan = tipoPlan;
    }

    // Constructor para cuando se recupera de la BD (con ID)
    public PlanMovil(int idPlan, int idCliente, String tipoPlan) {
        this.idPlan = idPlan;
        this.idCliente = idCliente;
        this.tipoPlan = tipoPlan;
    }

    public int getIdPlan() {
        return idPlan;
    }

    public void setIdPlan(int idPlan) {
        this.idPlan = idPlan;
    }

    public int getIdCliente() {
        return idCliente;
    }

    public void setIdCliente(int idCliente) {
        this.idCliente = idCliente;
    }

    public String getTipoPlan() {
        return tipoPlan;
    }

    public void setTipoPlan(String tipoPlan) {
        this.tipoPlan = tipoPlan;
    }

    // Método abstracto para calcular el pago mensual, a ser implementado por subclases
    public abstract double calcularPagoMensual();
}
```

Un ejemplo de implementación polimórfica en una subclase, como PlanPostPagoMinutosMegasEconomico:

```java
package modelo;

public class PlanPostPagoMinutosMegasEconomico extends PlanMovil {
    private double minutos;
    private double costoMinutos;
    private double megasGigas;
    private double costoPorGiga;
    private double porcentajeDescuento;

    public PlanPostPagoMinutosMegasEconomico(int idCliente, double minutos, double costoMinutos, double megasGigas, double costoPorGiga, double porcentajeDescuento) {
        super(idCliente, "Economico");
        this.minutos = minutos;
        this.costoMinutos = costoMinutos;
        this.megasGigas = megasGigas;
        this.costoPorGiga = costoPorGiga;
        this.porcentajeDescuento = porcentajeDescuento;
    }

    // Constructor para cuando se recupera de la BD (con ID)
    public PlanPostPagoMinutosMegasEconomico(int idPlan, int idCliente, double minutos, double costoMinutos, double megasGigas, double costoPorGiga, double porcentajeDescuento) {
        super(idPlan, idCliente, "Economico");
        this.minutos = minutos;
        this.costoMinutos = costoMinutos;
        this.megasGigas = megasGigas;
        this.costoPorGiga = costoPorGiga;
        this.porcentajeDescuento = porcentajeDescuento;
    }

    // Getters y Setters
    public double getMinutos() { return minutos; }
    public void setMinutos(double minutos) { this.minutos = minutos; }
    public double getCostoMinutos() { return costoMinutos; }
    public void setCostoMinutos(double costoMinutos) { this.costoMinutos = costoMinutos; }
    public double getMegasGigas() { return megasGigas; }
    public void setMegasGigas(double megasGigas) { this.megasGigas = megasGigas; }
    public double getCostoPorGiga() { return costoPorGiga; }
    public void setCostoPorGiga(double costoPorGiga) { this.costoPorGiga = costoPorGiga; }
    public double getPorcentajeDescuento() { return porcentajeDescuento; }
    public void setPorcentajeDescuento(double porcentajeDescuento) { this.porcentajeDescuento = porcentajeDescuento; }

    @Override
    public double calcularPagoMensual() {
        double subtotal = (minutos * costoMinutos) + (megasGigas * costoPorGiga);
        return subtotal * (1 - (porcentajeDescuento / 100));
    }
}
```

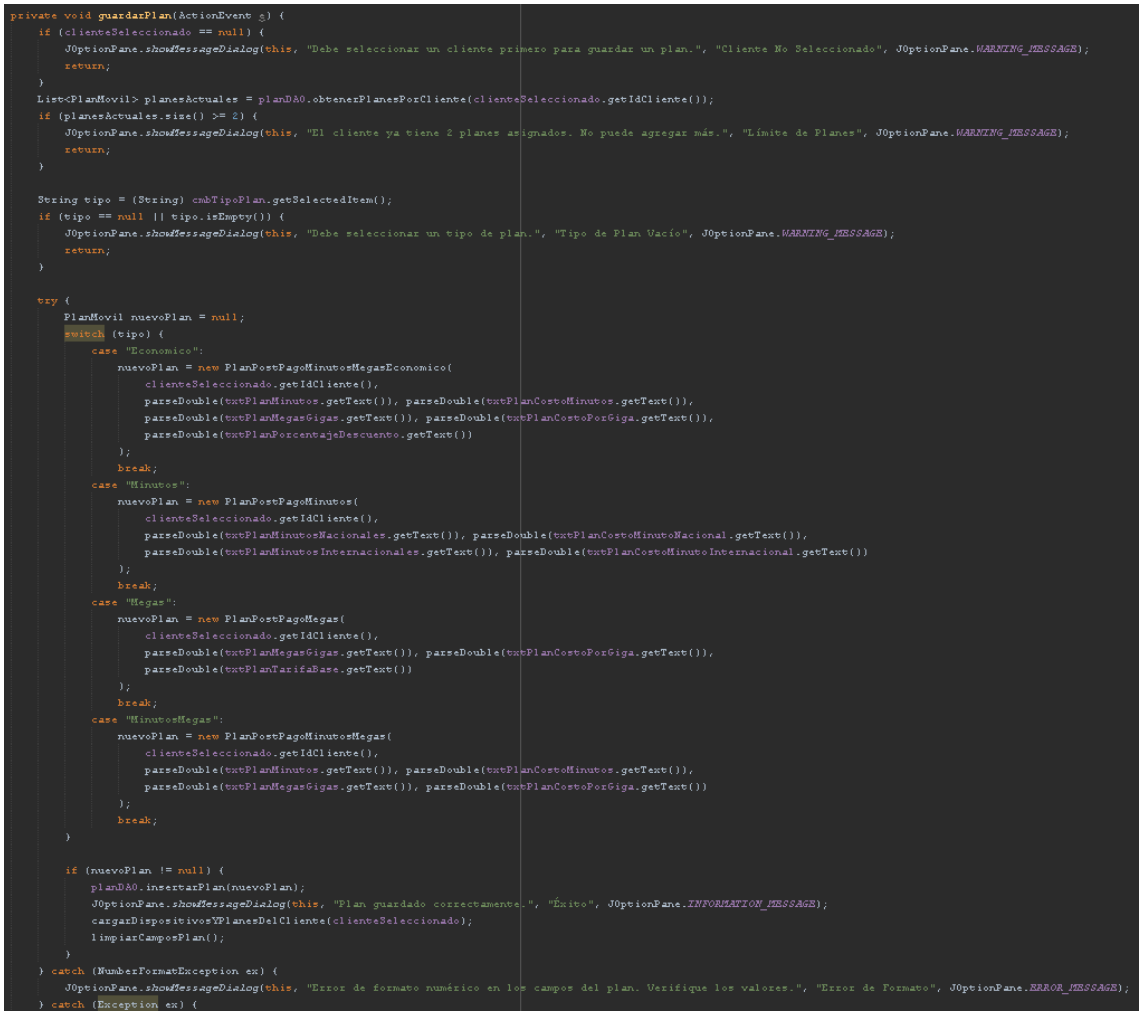
3.2.2. Operación CRUD (Create) en un DAO (Paquete dao)

Ejemplo de la operación Create para la entidad Cliente.

```java
public void insertarCliente(Cliente cliente) {
    String sql = "INSERT INTO clientes(nombres, pasaporte_cedula, ciudad, email, direccion) VALUES (?, ?, ?, ?, ?)";
    try (Connection conn = Conexion.conectar();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setString(1, cliente.getNombres());
        pstmt.setString(2, cliente.getPasaporteCedula());
        pstmt.setString(3, cliente.getCiudad());
        pstmt.setString(4, cliente.getEmail());
        pstmt.setString(5, cliente.getDireccion());
        pstmt.executeUpdate();
        System.out.println("Cliente insertado correctamente: " + cliente.getNombres());
    } catch (SQLException e) {
        System.err.println("Error al insertar cliente: " + e.getMessage());
        e.printStackTrace();
    }
}
```

3.2.3. Lógica del Controlador y Validación (Paquete vista)

Fragmento de VentanaPrincipal mostrando cómo se maneja la acción de guardar un plan, incluyendo la validación de la restricción de "no más de 2 planes".



```
private void guardarPlan(ActionEvent e) {
    if (clienteSeleccionado == null) {
        JOptionPane.showMessageDialog(this, "Debe seleccionar un cliente primero para guardar un plan.", "Cliente No Seleccionado", JOptionPane.WARNING_MESSAGE);
        return;
    }
    List<PlanMovil> planesActuales = planDAO.obtenerPlanesPorCliente(clienteSeleccionado.getIdCliente());
    if (planesActuales.size() >= 2) {
        JOptionPane.showMessageDialog(this, "El cliente ya tiene 2 planes asignados. No puede agregar más.", "Límite de Planes", JOptionPane.WARNING_MESSAGE);
        return;
    }

    String tipo = (String) cmbTipoPlan.getSelectedItem();
    if (tipo == null || tipo.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Debe seleccionar un tipo de plan.", "Tipo de Plan Vacío", JOptionPane.WARNING_MESSAGE);
        return;
    }

    try {
        PlanMovil nuevoPlan = null;
        switch (tipo) {
            case "Economico":
                nuevoPlan = new PlanPostPagoMinutosMegasEconomico(
                    clienteSeleccionado.getIdCliente(),
                    parseDouble(txtPlanMinutos.getText()), parseDouble(txtPlanCostoMinutos.getText()),
                    parseDouble(txtPlanMegasGigas.getText()), parseDouble(txtPlanCostoPorGiga.getText()),
                    parseDouble(txtPlanPorcentajeDescuento.getText())
                );
                break;
            case "Minutos":
                nuevoPlan = new PlanPostPagoMinutos(
                    clienteSeleccionado.getIdCliente(),
                    parseDouble(txtPlanMinutosNacionales.getText()), parseDouble(txtPlanCostoMinutoNacional.getText()),
                    parseDouble(txtPlanMinutosInternacionales.getText()), parseDouble(txtPlanCostoMinutoInternacional.getText())
                );
                break;
            case "Megas":
                nuevoPlan = new PlanPostPagoMegas(
                    clienteSeleccionado.getIdCliente(),
                    parseDouble(txtPlanMegasGigas.getText()), parseDouble(txtPlanCostoPorGiga.getText()),
                    parseDouble(txtPlanTarifaBase.getText())
                );
                break;
            case "MinutosMegas":
                nuevoPlan = new PlanPostPagoMinutosMegas(
                    clienteSeleccionado.getIdCliente(),
                    parseDouble(txtPlanMinutos.getText()), parseDouble(txtPlanCostoMinutos.getText()),
                    parseDouble(txtPlanMegasGigas.getText()), parseDouble(txtPlanCostoPorGiga.getText())
                );
                break;
        }

        if (nuevoPlan != null) {
            planDAO.insertarPlan(nuevoPlan);
            JOptionPane.showMessageDialog(this, "Plan guardado correctamente.", "Éxito", JOptionPane.INFORMATION_MESSAGE);
            cargarDispositivosYPlanesDelCliente(clienteSeleccionado);
            limpiarCamposPlan();
        }
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this, "Error de formato numérico en los campos del plan. Verifique los valores.", "Error de Formato", JOptionPane.ERROR_MESSAGE);
    } catch (Exception ex) {
```

3.2.4. Inicialización de la Base de Datos (Paquete dao)

La clase TablaCreator asegura que la base de datos y sus tablas estén listas al iniciar la aplicación.

```
public void crearTablas() {
    String sqlClientes = "CREATE TABLE IF NOT EXISTS clientes ("
            + "id_cliente INTEGER PRIMARY KEY AUTOINCREMENT,"
            + "nombres TEXT NOT NULL,"
            + "pasaporte_cedula TEXT NOT NULL UNIQUE,"
            + "ciudad TEXT NOT NULL,"
            + "email TEXT,"
            + "direccion TEXT"
            + ");";

    String sqlDispositivos = "CREATE TABLE IF NOT EXISTS dispositivos_moviles ("
            + "id_dispositivo INTEGER PRIMARY KEY AUTOINCREMENT,"
            + "id_cliente INTEGER NOT NULL,"
            + "marca TEXT NOT NULL,"
            + "modelo TEXT NOT NULL,"
            + "numero_celular TEXT NOT NULL UNIQUE,"
            + "FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente) ON DELETE CASCADE"
            + ");";

    String sqlPlanes = "CREATE TABLE IF NOT EXISTS planes ("
            + "id_plan INTEGER PRIMARY KEY AUTOINCREMENT,"
            + "id_cliente INTEGER NOT NULL,"
            + "tipo_plan TEXT NOT NULL,"
            + "minutos REAL,"
            + "costo_minutos REAL,"
            + "megas_gigas REAL,"
            + "costo_por_giga REAL,"
            + "porcentaje_descuento REAL,"
            + "minutos_nacionales REAL,"
            + "costo_minuto_nacional REAL,"
            + "minutos_internacionales REAL,"
            + "costo_minuto_internacional REAL,"
            + "tarifa_base REAL,"
            + "FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente) ON DELETE CASCADE"
            + ");";

    String sqlFacturas = "CREATE TABLE IF NOT EXISTS facturas ("
            + "id_factura INTEGER PRIMARY KEY AUTOINCREMENT,"
            + "id_cliente INTEGER NOT NULL,"
            + "fecha_emision TEXT NOT NULL,"
            + "total_pagar REAL NOT NULL,"
            + "estado TEXT NOT NULL,"
            + "FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente) ON DELETE CASCADE"
            + ");";

    String sqlDetalleFactura = "CREATE TABLE IF NOT EXISTS detalle_factura ("
            + "id_detalle INTEGER PRIMARY KEY AUTOINCREMENT,"
            + "id_factura INTEGER NOT NULL,"
            + "id_plan INTEGER NOT NULL,"
            + "costo_plan_facturado REAL NOT NULL,"
            + "FOREIGN KEY (id_factura) REFERENCES facturas(id_factura) ON DELETE CASCADE,"
```

```
            + "FOREIGN KEY (id_plan) REFERENCES planes(id_plan) ON DELETE CASCADE"
            + ");";

    try (Connection conn = Conexion.conectar();
         Statement stmt = conn.createStatement()) {
        if (conn != null) {
            stmt.execute(sqlClientes);
            System.out.println("Tabla 'clientes' creada o verificada.");
            stmt.execute(sqlDispositivos);
            System.out.println("Tabla 'dispositivos_moviles' creada o verificada.");
            stmt.execute(sqlPlanes);
            System.out.println("Tabla 'planes' creada o verificada.");
            stmt.execute(sqlFacturas);
            System.out.println("Tabla 'facturas' creada o verificada.");
            stmt.execute(sqlDetalleFactura);
            System.out.println("Tabla 'detalle_factura' creada o verificada.");
        } else {
            System.err.println("No se pudo establecer la conexión a la base de datos para crear las tablas.");
        }
    } catch (SQLException e) {
        System.err.println("Error al crear las tablas: " + e.getMessage());
        e.printStackTrace();
    }
  }
}
```

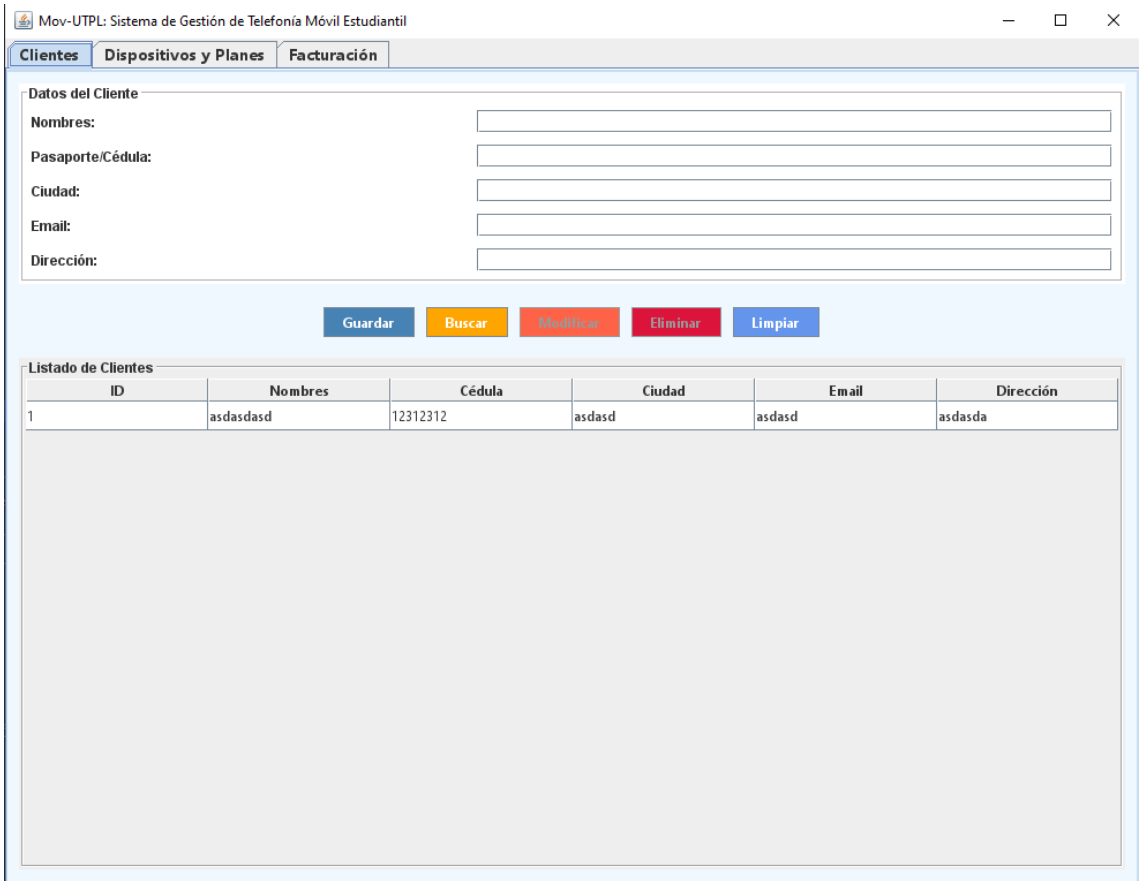
URL del Repositorio GIT:

https://github.com/ProOrientadaObjetos-P-E-AA2025/aab2-25-proyecto-bimestral-2do-bim-grupal-XaviCabrera05.git

4. Resultados

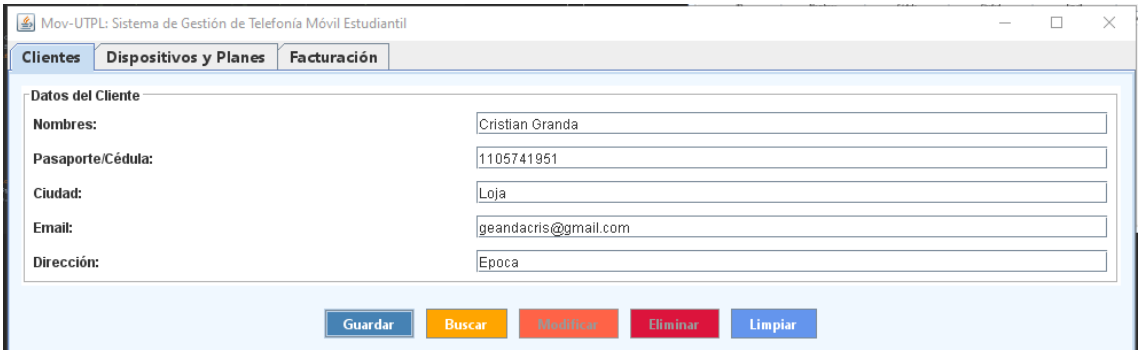
Esta sección presenta capturas de pantalla del sistema en funcionamiento, evidenciando la implementación de los requisitos y la interacción del usuario.

4.1. Interfaz Principal del Sistema

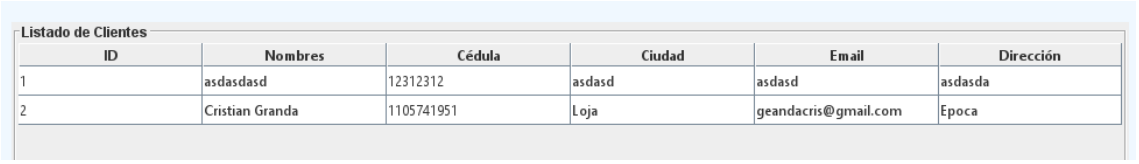


4.2. Gestión de Clientes (Operaciones CRUD)

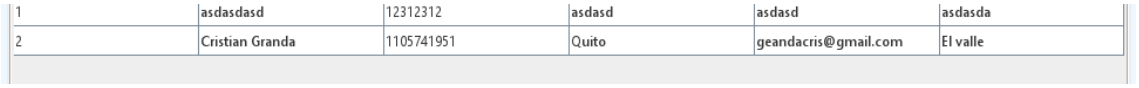
- **Creación de un Nuevo Cliente:**



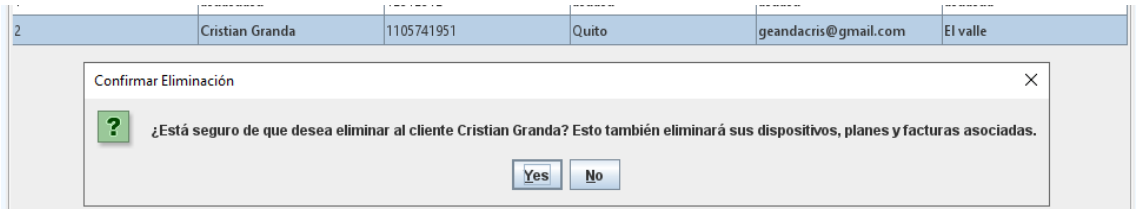
- **Listado de Clientes (Read**



- **Modificación de un Cliente:**



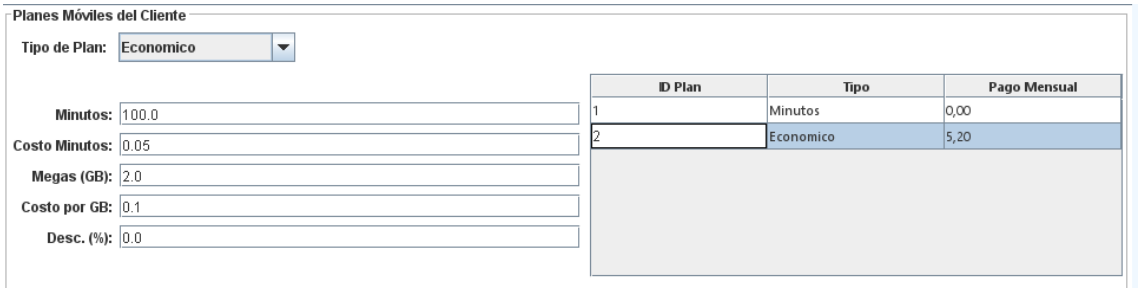
- **Eliminación de un Cliente**



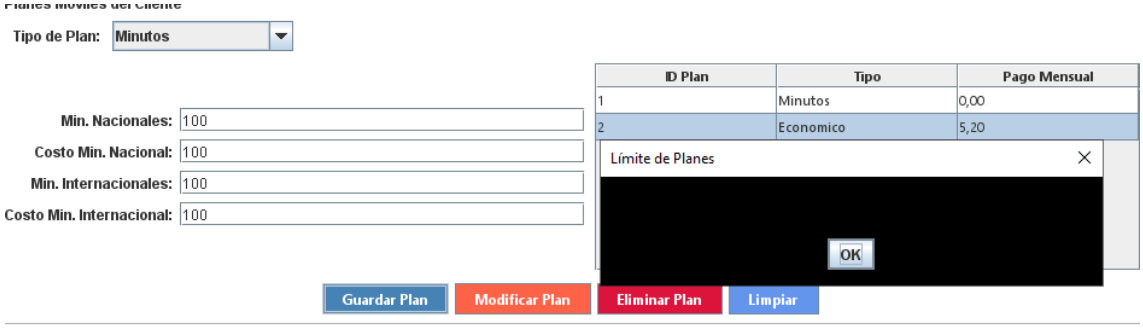
- ## 4.3. Gestión de Dispositivos y Planes

- ## Carga de Cliente y Adición de Dispositivo:

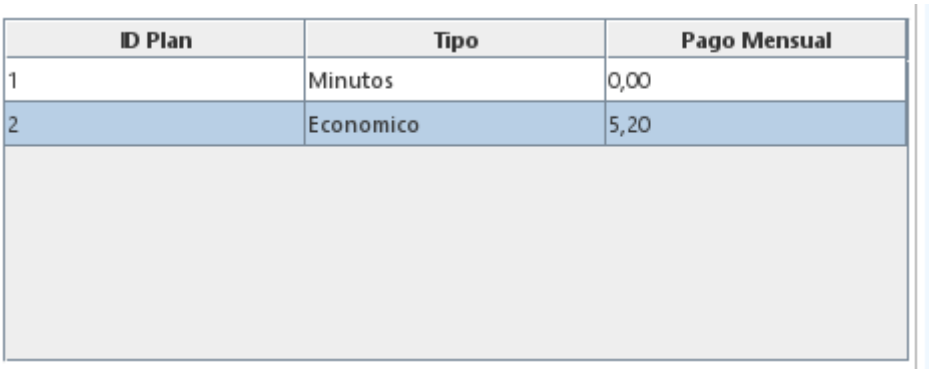
- ## Adición de Plan



- ## Intento de agregar un tercer plan a un cliente



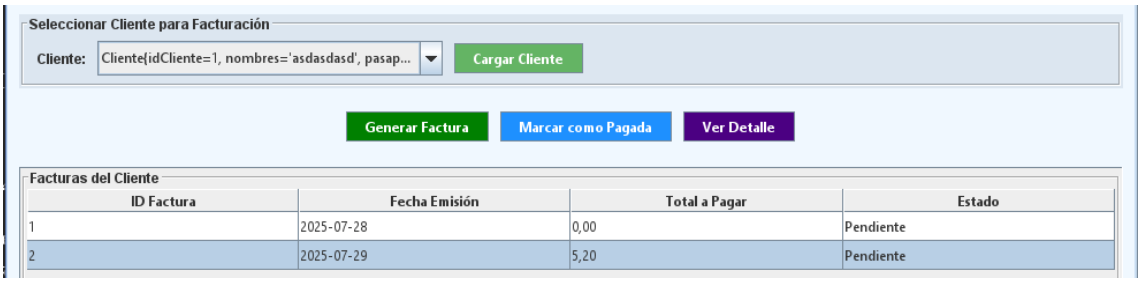
- **Visualización de Dispositivos y Planes Asociados:**



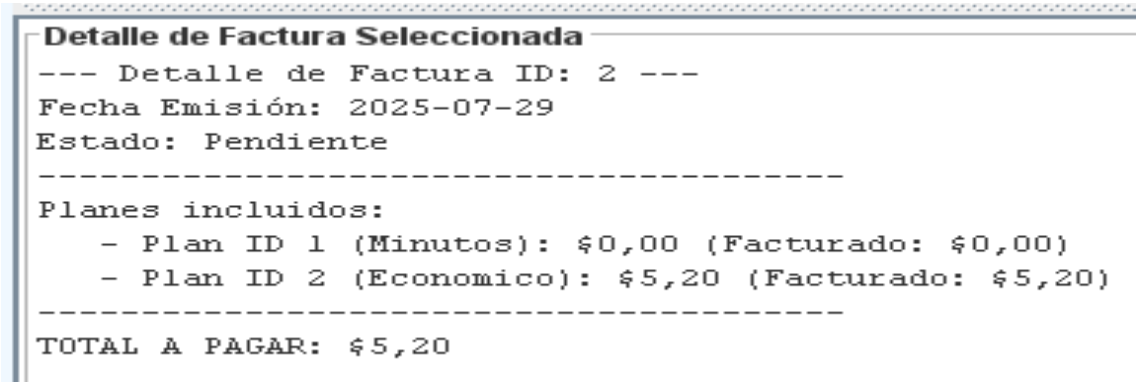
ID Plan	Tipo	Pago Mensual
1	Minutos	0,00
2	Economico	5,20

- **4.4. Gestión de Facturación**

- **Generación de Factura:**



- **Visualización de Facturas y Detalle:**



- **Marcado de Factura como Pagada**

4.5. Consola de Salida

```
Tabla 'clientes' creada o verificada.
Tabla 'dispositivos_moviles' creada o verificada.
Tabla 'planes' creada o verificada.
Tabla 'facturas' creada o verificada.
Tabla 'detalle_factura' creada o verificada.
Cliente insertado correctamente: Cristian Granda
Cliente actualizado correctamente: Cristian Granda
Cliente eliminado correctamente con c�dula/pasaporte: 1105741951
Plan insertado correctamente para cliente ID: 1 Tipo: Economico con ID de Plan: 2
Factura insertada correctamente para cliente ID: 1 con ID: 2
Detalle de factura insertado correctamente para factura ID: 2
Detalle de factura insertado correctamente para factura ID: 2
Factura actualizada correctamente ID: 2
```