PROYECTO TECNOSTORE

Evidencias de Cumplimiento de Requisitos

Sistema de Gestión de Ventas de Celulares

RepositorioGitHubTecnoStore

Maria Alejandra Gomez Archila

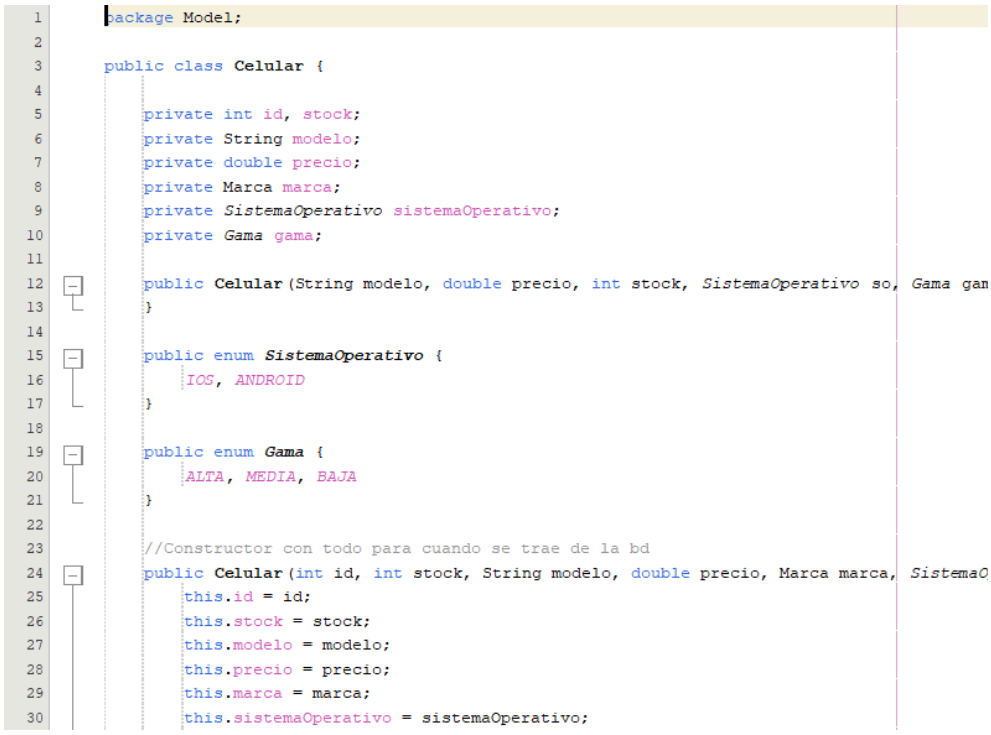
Ruta - JAVA / Sede - Cajasan / Docente - David Dominguez

1. GESTIÓN DE CELULARES

1.1 Clase Celular - Modelo

Archivo: Model/Celular.java
Líneas: 1-110 (clase completa)
Propósito: Representa la entidad Celular con todos sus atributos requeridos: id, marca, modelo, precio, stock, sistema operativo y gama.

Atributos principales:
• id (int): Identificador único del celular
• marca (Marca): Marca del celular (composición)
• modelo (String): Modelo del celular
• precio (double): Precio del celular
• stock (int): Cantidad disponible
• sistemaOperativo (enum): IOS o ANDROID
• gama (enum): ALTA, MEDIA o BAJA
(no se alcanza a ver completo, pero es prueba de que si existe)



```
1   package Model;
2
3   public class Celular {
4
5       private int id, stock;
6       private String modelo;
7       private double precio;
8       private Marca marca;
9       private SistemaOperativo sistemaOperativo;
10      private Gama gama;
11
12      public Celular(String modelo, double precio, int stock, SistemaOperativo so, Gama gam
13      }
14
15      public enum SistemaOperativo {
16          IOS, ANDROID
17      }
18
19      public enum Gama {
20          ALTA, MEDIA, BAJA
21      }
22
23      //Constructor con todo para cuando se trae de la bd
24      public Celular(int id, int stock, String modelo, double precio, Marca marca, SistemaO
25          this.id = id;
26          this.stock = stock;
27          this.modelo = modelo;
28          this.precio = precio;
29          this.marca = marca;
30          this.sistemaOperativo = sistemaOperativo;
```

1.2 Validación de Precio Positivo

Archivo: Utilities/Validator.java
Líneas: 32-34
Propósito: Valida que el precio sea mayor a 0 antes de registrar o actualizar un celular.

Método de validación:
```
public static boolean validarPrecio(double precio) {
   return precio > 0;
}
```

Uso en MenuCelulares: View/MenuCelulares.java, líneas 111-114

```
33          // Valida que un precio sea positivo
34
35          public static boolean validarPrecio(double precio) {
36              return precio > 0;
37          }
...
109     if (!Validator.validarPrecio(precio)) {
110         System.out.println("-- El precio debe ser positivo --");
111         return;
112     }
```

(falta captura de la consola, donde se evidencie el funcionamiento)

1.3 Validación de Stock No Negativo

Archivo: Utilities/Validator.java
Líneas: 38-40
Propósito: Valida que el stock sea mayor o igual a 0 (no puede ser negativo).

Método de validación:
```
public static boolean validarStock(int stock) {
   return stock >= 0;
}
```

Uso en MenuCelulares: View/MenuCelulares.java, líneas 118-121

```
38
39          // Valida que el stock sea positivo o cero
40
41          public static boolean validarStock(int stock) {
42              return stock >= 0;
43          }
```

```
116     if (!Validator.validarStock(stock)) {
117         System.out.println("-- El stock debe ser positivo o cero --");
118         return;
119     }
120
```

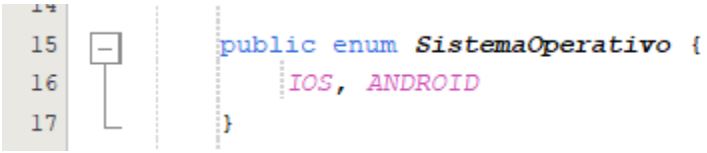
(falta captura de la consola, donde se evidencie el funcionamiento)

1.4 Enum Gama y Sistema Operativo

Archivo: Model/Celular.java
Líneas: 15-21 (enums internos)
Propósito: Define valores constantes para Sistema Operativo (IOS, ANDROID) y Gama (ALTA, MEDIA, BAJA).

Enum SistemaOperativo:
```
public enum SistemaOperativo {
    IOS, ANDROID
}
```

Enum Gama:
```
public enum Gama {
    ALTA, MEDIA, BAJA
}
```



2. GESTIÓN DE CLIENTES

2.1 Clase Cliente y Persona (Composición)

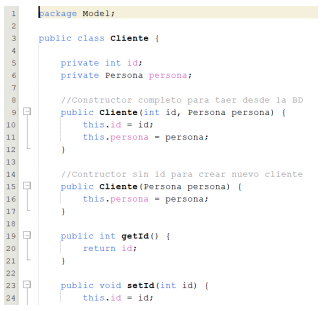
Archivos:
• Model/Cliente.java - Líneas 1-40
• Model/Persona.java - Líneas 1-73

Propósito: Demuestra composición POO. Cliente contiene un objeto Persona con los datos básicos (id, nombre, identificación, correo, teléfono).

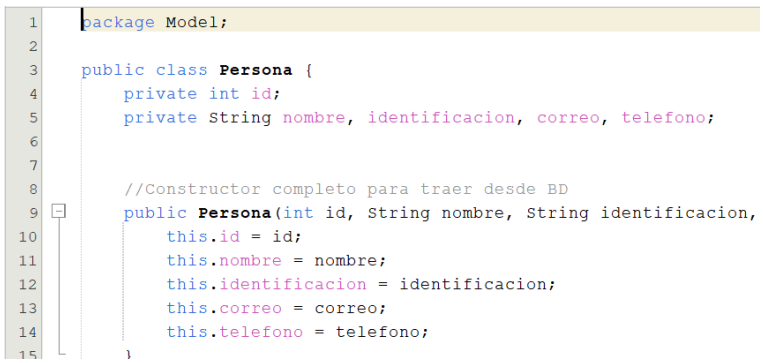
Estructura de Persona:
• id (int)
• nombre (String)
• identificacion (String)
• correo (String)
• telefono (String)

Composición en Cliente:

```
public class Cliente {
    private int id;
    private Persona persona;  // Composición
}
```



```
package Model;

public class Cliente {

    private int id;
    private Persona persona;

    //Constructor completo para taer desde la BD
    public Cliente(int id, Persona persona) {
        this.id = id;
        this.persona = persona;
    }

    //Contructor sin id para crear nuevo cliente
    public Cliente(Persona persona) {
        this.persona = persona;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
```



```
package Model;

public class Persona {
    private int id;
    private String nombre, identificacion, correo, telefono;


    //Constructor completo para traer desde BD
    public Persona(int id, String nombre, String identificacion,
        this.id = id;
        this.nombre = nombre;
        this.identificacion = identificacion;
        this.correo = correo;
        this.telefono = telefono;
    }
```

2.2 Validación de Formato de Correo (Regex)

Archivo: Utilities/Validator.java
Líneas: 8-21
Propósito: Valida el formato del correo electrónico usando expresiones regulares (Regex).

Patrón Regex utilizado:

```
private static final Pattern EMAIL_PATTERN =
    Pattern.compile("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$");
```

Método de validación:

```
public static boolean validarCorreo(String correo) {
    if (correo == null || correo.trim().isEmpty()) {
        return false;
    }
    return EMAIL_PATTERN.matcher(correo).matches();
}
```

Uso en MenuClientes: View/MenuClientes.java, líneas 93-97

```
7       // Patron para validar correos electronicos
8       private static final Pattern EMAIL_PATTERN
9               = Pattern.compile("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$");
10
11      // Patron para validar telefonos (10 dígitos)
12      private static final Pattern TELEFONO_PATTERN
13              = Pattern.compile("^[0-9]{10}$");
```

```
String identificacion = leerTexto("Identificacion: ");
if (!Validator.validarIdentificacion(identificacion)) {
    System.out.println("La identificacion debe tener entre 6 y 13 caracteres")
    return;
}

String correo = leerTexto("Correo: ");
if (!Validator.validarCorreo(correo)) {
    System.out.println("Correo invalido");
    return;
}
```

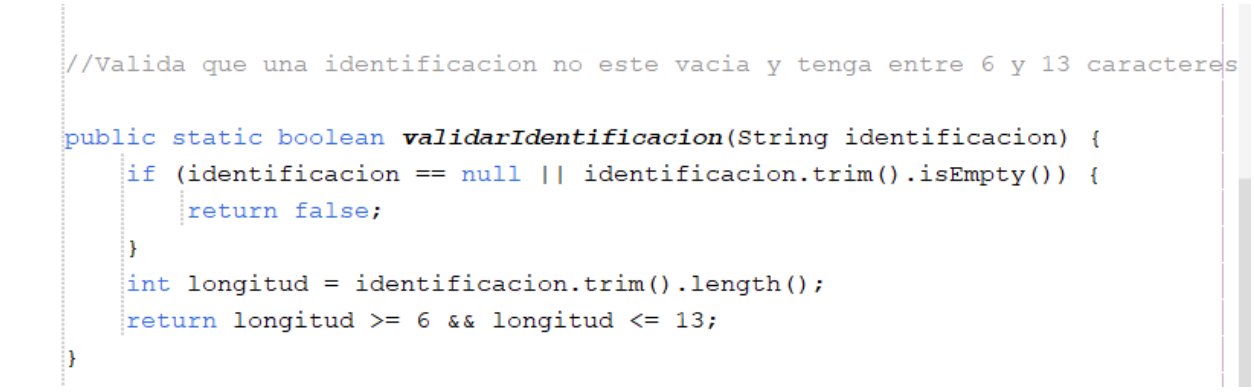
2.3 Validación de Identificación Única

Archivo: Utilities/Validator.java
Líneas: 44-51
Propósito: Valida que la identificación tenga entre 6 y 13 caracteres. La unicidad se verifica en la base de datos mediante constraints.

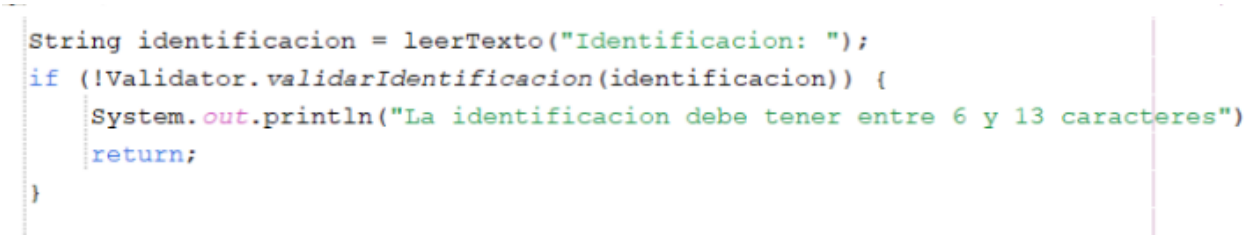
Método de validación:

```
public static boolean validarIdentificacion(String identificacion) {
    if (identificacion == null || identificacion.trim().isEmpty()) {
        return false;
    }
    int longitud = identificacion.trim().length();
    return longitud >= 6 && longitud <= 13;
}
```

Nota: La base de datos tiene un constraint UNIQUE en el campo identificacion para garantizar unicidad a nivel de persistencia.



```
//Valida que una identificacion no este vacia y tenga entre 6 y 13 caracteres

public static boolean validarIdentificacion(String identificacion) {
    if (identificacion == null || identificacion.trim().isEmpty()) {
        return false;
    }
    int longitud = identificacion.trim().length();
    return longitud >= 6 && longitud <= 13;
}
```



```
String identificacion = leerTexto("Identificacion: ");
if (!Validator.validarIdentificacion(identificacion)) {
    System.out.println("La identificacion debe tener entre 6 y 13 caracteres")
    return;
}
```

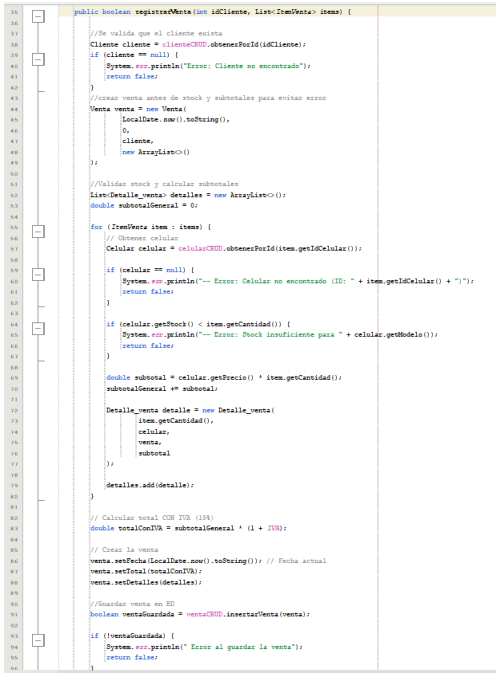
3. GESTIÓN DE VENTAS

3.1 Registro de Ventas con Detalle

Archivos principales:
• Controller/gestionVentas.java - Líneas 36-109
• Model/Venta.java - Líneas 1-90
• Model/Detalle_venta.java - Líneas 1-90

Propósito: Permite registrar una venta completa con cliente, fecha, múltiples productos (detalle_venta) y total.

Proceso del método registrarVenta():
1. Valida que el cliente exista
2. Verifica stock disponible para cada producto
3. Calcula subtotales por producto
4. Calcula total con IVA (19%)
5. Guarda la venta en la base de datos
6. Actualiza el stock de los celulares vendidos



```
public boolean registrarVenta(int idCliente, List<ItemVenta> items) {

    //Se valida que el cliente exista
    Cliente cliente = clienteCRUD.obtenerPorId(idCliente);
    if (cliente == null) {
        System.err.println("Error: Cliente no encontrado");
        return false;
    }
    //crear venta antes de stock y subtotales para evitar error
    Venta venta = new Venta(
            LocalDate.now().toString(),
            0,
            cliente,
            new ArrayList<>()
    );

    //Validar stock y calcular subtotales
    List<Detalle_venta> detalles = new ArrayList<>();
    double subtotalGeneral = 0;

    for (ItemVenta item : items) {
        // Obtener celular
        Celular celular = celularCRUD.obtenerPorId(item.getIdCelular());

        if (celular == null) {
            System.err.println("-- Error: Celular no encontrado (ID: " + item.getIdCelular() + ")");
            return false;
        }

        if (celular.getStock() < item.getCantidad()) {
            System.err.println("-- Error: Stock insuficiente para " + celular.getModelo());
            return false;
        }

        double subtotal = celular.getPrecio() * item.getCantidad();
        subtotalGeneral += subtotal;

        Detalle_venta detalle = new Detalle_venta(
                item.getCantidad(),
                celular,
                venta,
                subtotal

        );

        detalles.add(detalle);
    }

    // Calcular total CON IVA (19%)
    double totalConIVA = subtotalGeneral * (1 + IVA);

    // Crear la venta
    venta.setFecha(LocalDate.now().toString()); // Fecha actual
    venta.setTotal(totalConIVA);
    venta.setDetalles(detalles);

    //Guardar venta en BD
    boolean ventaGuardada = ventaCRUD.insertarVenta(venta);

    if (!ventaGuardada) {
        System.err.println(" Error al guardar la venta");
        return false;
    }
```

[CAPTURA 3.1B: Ejecución en consola del menú de ventas registrando una venta exitosa]

3.2 Cálculo de Total con IVA (19%)

Archivo: Controller/gestionVentas.java
Líneas: 26 (constante IVA) y 84-85 (cálculo)
Propósito: Calcula automáticamente el IVA del 19% sobre el subtotal de la venta.

Declaración de la constante IVA:
private static final double IVA = 0.19; // IVA DE 19%

Cálculo del total con IVA:
// Calcular total CON IVA (19%)
double totalConIVA = subtotalGeneral * (1 + IVA);

Ejemplo:
• Subtotal: $100,000
• IVA (19%): $19,000
• Total: $119,000

```
// Calcular total CON IVA (19%)
double totalConIVA = subtotalGeneral * (1 + IVA);
```

[CAPTURA 3.2B: Resumen de venta en consola mostrando Subtotal, IVA 19% y Total]

3.3 Actualización de Stock

Archivos:
• Controller/gestionVentas.java - Líneas 102-106
• Persistence/CelularCRUD.java - Líneas 209-228

Propósito: Actualiza automáticamente el stock de cada celular vendido en la base de datos.

Código en gestionVentas.java:

```
// Actualizar stock de celulares vendidos
for (Detalle_venta detalle : detalles) {
    Celular celular = detalle.getCelular();
    int nuevoStock = celular.getStock() - detalle.getCantidad();
    celularCRUD.actualizarStock(celular.getId(), nuevoStock);
}
```

```
98      // Actualizar stock de celulares vendidos
99      for (Detalle_venta detalle : detalles) {
100         Celular celular = detalle.getCelular();
101         int nuevoStock = celular.getStock() - detalle.getCantidad();
102         celularCRUD.actualizarStock(celular.getId(), nuevoStock);
103     }
```

[CAPTURA 3.3B: Listado de celulares antes y después de una venta mostrando cambio en stock]

3.4 Persistencia con JDBC

Archivos principales:
• Persistence/VentaCRUD.java - Método insertarVenta (líneas 19-47)
• Persistence/DetalleVentaCRUD.java - Método insertarDetalleVenta (líneas 21-46)

Propósito: Guarda la venta y sus detalles en las tablas 'venta' y 'detalle_venta' de MySQL usando JDBC.

Inserción en tabla venta:
String sql = "insert into venta (id_cliente, fecha, total) values (?,?,?)";

Inserción en tabla detalle_venta:
String sql = "insert into detalle_venta (id_venta, id_celular, cantidad, subtotal) values (?,?,?,?)";

```
// C
public boolean insertarVenta(Venta venta) {

    String sql = "insert into venta (id_cliente, fecha, total) values (?,?,?)";

    try (PreparedStatement stmt = connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)

        stmt.setInt(1, venta.getCliente().getId());
        stmt.setString(2, venta.getFecha());
        stmt.setDouble(3, venta.getTotal());

        if (stmt.executeUpdate() > 0) {
            ResultSet rs = stmt.getGeneratedKeys();
            if (rs.next()) {
                venta.setId(rs.getInt(1));
            }
            System.out.println("-- Venta insertada correctamente --");
            return true;
        }

    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }

    return false;
}
```

[CAPTURA 3.4B: Consulta SELECT en base de datos mostrando ventas registradas]

4. REPORTES Y ANÁLISIS (STREAM API)

4.1 Celulares con Stock Bajo (filter)

Archivo: Utilities/ReportUtils.java
Líneas: 13-18
Propósito: Filtra celulares con stock menor a 5 unidades usando Stream API.

Código del método:

```
public static List<Celular> celularesStockBajo(List<Celular> celulares) {
    return celulares.stream()
            .filter(c -> c.getStock() < 5)
            .sorted(Comparator.comparingInt(Celular::getStock))
            .collect(Collectors.toList());
}
```

Operaciones Stream API utilizadas:
• filter() - Filtra celulares con stock < 5
• sorted() - Ordena por stock ascendente
• collect() - Convierte el stream a lista

```
12
13      // Obtener celulares con stock bajo (menor a 5 unidades) Usa Stream API
14      public static List<Celular> celularesStockBajo(List<Celular> celulares) {
15          return celulares.stream()
16                  .filter(c -> c.getStock() < 5)
17                  .sorted(Comparator.comparingInt(Celular::getStock))
18                  .collect(Collectors.toList());
19      }
```

[CAPTURA 4.1B: Reporte de stock bajo en consola mostrando celulares con menos de 5 unidades]

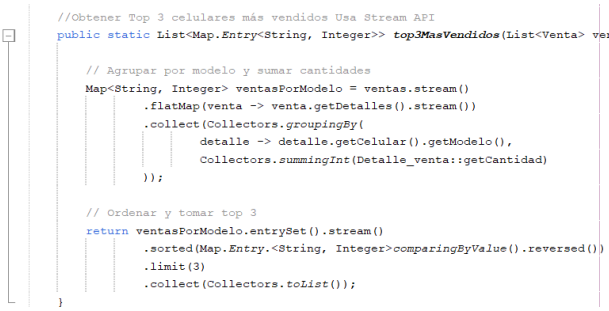
4.2 Top 3 Celulares Más Vendidos (flatMap, sorted, limit)

Archivo: Utilities/ReportUtils.java
Líneas: 21-37
Propósito: Calcula los 3 celulares más vendidos usando Stream API con operaciones avanzadas.

Código del método:

```
public static List<Map.Entry<String, Integer>> top3MasVendidos(List<Venta> ventas) {
    // Agrupar por modelo y sumar cantidades
    Map<String, Integer> ventasPorModelo = ventas.stream()
        .flatMap(venta -> venta.getDetalles().stream())
        .collect(Collectors.groupingBy(
            detalle -> detalle.getCelular().getModelo(),
            Collectors.summingInt(Detalle_venta::getCantidad)
        ));
    // Ordenar y tomar top 3
    return ventasPorModelo.entrySet().stream()
        .sorted(Map.Entry.<String, Integer>comparingByValue().reversed())
        .limit(3)
        .collect(Collectors.toList());
}
```

Operaciones Stream API utilizadas:
• flatMap() - Aplana lista de detalles de todas las ventas
• groupingBy() - Agrupa por modelo de celular
• summingInt() - Suma cantidades vendidas por modelo
• sorted() - Ordena por cantidad descendente
• limit() - Toma solo los primeros 3



```
//Obtener Top 3 celulares más vendidos Usa Stream API
public static List<Map.Entry<String, Integer>> top3MasVendidos(List<Venta> ventas) {

    // Agrupar por modelo y sumar cantidades
    Map<String, Integer> ventasPorModelo = ventas.stream()
            .flatMap(venta -> venta.getDetalles().stream())
            .collect(Collectors.groupingBy(
                    detalle -> detalle.getCelular().getModelo(),
                    Collectors.summingInt(Detalle_venta::getCantidad)
            ));

    // Ordenar y tomar top 3
    return ventasPorModelo.entrySet().stream()
            .sorted(Map.Entry.<String, Integer>comparingByValue().reversed())
            .limit(3)
            .collect(Collectors.toList());
}
```

[CAPTURA 4.2B: Reporte en consola mostrando Top 3 con modelos y cantidades vendidas]

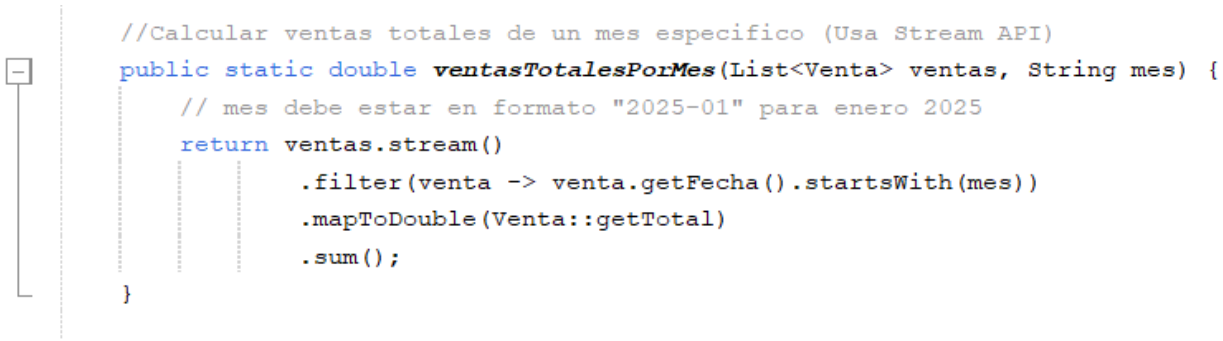
4.3 Ventas Totales por Mes (filter, mapToDouble, sum)

Archivo: Utilities/ReportUtils.java
Líneas: 40-46
Propósito: Calcula el total de ventas de un mes específico usando Stream API.

Código del método:

```
public static double ventasTotalesPorMes(List<Venta> ventas, String mes) {
    // mes debe estar en formato "2025-01" para enero 2025
    return ventas.stream()
        .filter(venta -> venta.getFecha().startsWith(mes))
        .mapToDouble(Venta::getTotal)
        .sum();
}
```

Operaciones Stream API utilizadas:
• filter() - Filtra ventas del mes especificado
• mapToDouble() - Extrae el total de cada venta
• sum() - Suma todos los totales



```
//Calcular ventas totales de un mes especifico (Usa Stream API)
public static double ventasTotalesPorMes(List<Venta> ventas, String mes) {
    // mes debe estar en formato "2025-01" para enero 2025
    return ventas.stream()
            .filter(venta -> venta.getFecha().startsWith(mes))
            .mapToDouble(Venta::getTotal)
            .sum();
}
```

[CAPTURA 4.3B: Consulta en consola de ventas por mes mostrando total calculado]

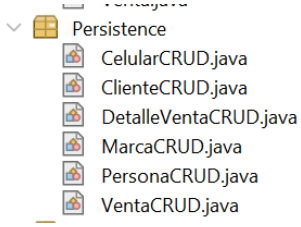
5. PERSISTENCIA Y ARCHIVOS

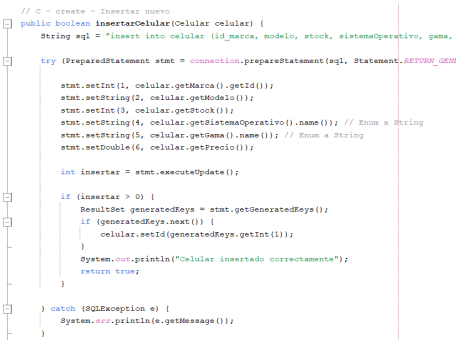
5.1 Patrón DAO (Data Access Object)

Archivos:
• Persistence/CelularCRUD.java
• Persistence/ClienteCRUD.java
• Persistence/VentaCRUD.java
• Persistence/DetalleVentaCRUD.java
• Persistence/MarcaCRUD.java
• Persistence/PersonaCRUD.java

Propósito: Separación de la lógica de acceso a datos. Cada clase CRUD encapsula todas las operaciones de base de datos (Create, Read, Update, Delete) para una entidad específica.

Estructura de cada DAO:
• Conexión a base de datos mediante Singleton
• Métodos CRUD estándar:
  - insertar (Create)
  - obtenerPorId (Read)
  - obtenerTodos (Read)
  - actualizar (Update)
  - eliminar (Delete)





5.2 Try-with-resources

Archivo ejemplo: Persistence/CelularCRUD.java
Líneas: 30 (y en todos los métodos CRUD)
Propósito: Manejo automático de recursos con try-with-resources para cerrar conexiones, statements y resultsets automáticamente, evitando memory leaks.

Ejemplo de uso:
```
try (PreparedStatement stmt = connection.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS)) {
    // Configurar parámetros
    stmt.setInt(1, celular.getMarca().getId());
    stmt.setString(2, celular.getModelo());
    // ... más parámetros
    int insertar = stmt.executeUpdate();
    // PreparedStatement se cierra automáticamente al salir del bloque
} catch (SQLException e) {
    System.err.println(e.getMessage());
}
```

Ventajas:
• Cierre automático de recursos
• Previene memory leaks
• Código más limpio y legible

```java
try (PreparedStatement stmt = connection.prepareStatement(sql)) {
    ResultSet rs = stmt.executeQuery();

    while (rs.next()) {
        // Obtener la marca asociada
        Marca marca = marcaCRUD.obtenerPorId(rs.getInt("id_marca"));

        Celular celular = new Celular(
                rs.getInt("id"),
                rs.getInt("stock"),
                rs.getString("modelo"),
                rs.getDouble("precio"),
                marca,
                Celular.SistemaOperativo.valueOf(rs.getString("sistemaOperativo")),
                Celular.Gama.valueOf(rs.getString("gama"))
        );
        celulares.add(celular);
    }
```

5.3 PreparedStatement (Prevención SQL Injection)

Archivo ejemplo: Persistence/CelularCRUD.java
Líneas: 28-44 (método insertarCelular)
Propósito: Uso de PreparedStatement en todos los métodos CRUD para prevenir ataques de SQL Injection mediante parámetros seguros.

Ejemplo con parámetros:
String sql = "insert into celular (id_marca, modelo, stock, sistemaOperativo, gama, precio) values (?, ?, ?, ?, ?, ?)";

try (PreparedStatement stmt = connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {
    stmt.setInt(1, celular.getMarca().getId());   // Parámetro 1
    stmt.setString(2, celular.getModelo());   // Parámetro 2
    stmt.setInt(3, celular.getStock());   // Parámetro 3
    stmt.setString(4, celular.getSistemaOperativo().name()); // Parámetro 4
    stmt.setString(5, celular.getGama().name());   // Parámetro 5
    stmt.setDouble(6, celular.getPrecio());   // Parámetro 6
    int insertar = stmt.executeUpdate();
}

Ventajas de PreparedStatement:
• Previene SQL Injection
• Mejor rendimiento (queries pre-compiladas)
• Manejo automático de tipos de datos

```
public boolean insertarCelular(Celular celular) {
    String sql = "insert into celular (id_marca, modelo, stock, sistemaOperativo, gama,

    try (PreparedStatement stmt = connection.prepareStatement(sql, Statement.RETURN_GEN

        stmt.setInt(1, celular.getMarca().getId());
        stmt.setString(2, celular.getModelo());
        stmt.setInt(3, celular.getStock());
        stmt.setString(4, celular.getSistemaOperativo().name()); // Enum a String
        stmt.setString(5, celular.getGama().name()); // Enum a String
        stmt.setDouble(6, celular.getPrecio());

        int insertar = stmt.executeUpdate();

        if (insertar > 0) {
            ResultSet generatedKeys = stmt.getGeneratedKeys();
            if (generatedKeys.next()) {
                celular.setId(generatedKeys.getInt(1));
            }
            System.out.println("Celular insertado correctamente");
            return true;
        }
```

5.4 Generación de Archivo reporte_ventas.txt

Archivo: Utilities/UtilsFile.java
Líneas: 12-75
Propósito: Genera un archivo de texto con el resumen completo de todas las ventas realizadas, usando BufferedWriter y try-with-resources.

Código del método:

```
public static boolean generarReporteVentas(List<Venta> ventas, String rutaArchivo) {
    // try-with-resources: cierra automáticamente el BufferedWriter
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(rutaArchivo))) {
        // Encabezado
        writer.write("***** REPORTE DE VENTAS - TECNOSTORE *****\n\n");
        // Fecha de generación
        LocalDateTime ahora = LocalDateTime.now();
        // ... escribir fecha y ventas
        for (Venta venta : ventas) {
            // Escribir detalles de cada venta
        }
        return true;
    } catch (IOException e) {
        return false;
    }
}
```

Contenido del archivo:
• Encabezado con título
• Fecha y hora de generación
• Total de ventas registradas
• Detalle de cada venta (ID, cliente, fecha, productos, total)
• Total general de todas las ventas

[CAPTURA 5.4B: Mensaje en consola confirmando generación exitosa del archivo]
[CAPTURA 5.4C: Archivo reporte_ventas.txt abierto mostrando su contenido]

6. PATRONES DE DISEÑO Y PRINCIPIOS POO

6.1 Patrón Singleton (ConexiónDB)

Archivo: Controller/ConecctionDB.java
Líneas: 1-34 (clase completa)
Propósito: Implementa el patrón Singleton para garantizar una única instancia de conexión a la base de datos durante toda la ejecución del programa.

Código del patrón:

```
public class ConecctionDB {
    // Instancia única (Singleton)
    private static ConecctionDB conectado;

    // Constructor privado (no se puede instanciar desde fuera)
    private ConecctionDB() {
    }

    // Método para obtener la única instancia
    public static ConecctionDB getInstance() {
        if (conectado == null) {
            conectado = new ConecctionDB();
        }
        return conectado;
    }
}
```

Características del Singleton:
• Constructor privado
• Variable estática privada
• Método público getInstance()
• Garantiza una sola instancia

Uso en las clases CRUD:

```
this.connection = ConecctionDB.getInstance().conectar();
```

```java
package Controller;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConecctionDB {

    //Instanc
    private static ConecctionDB conectado;

    private ConecctionDB() {
    }

    public static ConecctionDB getInstance() {
        if (conectado == null) {
            conectado = new ConecctionDB();
        }
        return conectado;
    }

    public Connection conectar() {
        Connection c = null;

        try {
            c = DriverManager.getConnection("jdbc:mysql://localhost:3306/tecnost

            System.out.println("Conexion con la base de datos se establecio corr
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return c;

    }
}
```

6.2 Herencia y Composición

COMPOSICIÓN:

1. Cliente contiene Persona
Archivo: Model/Cliente.java, líneas 5-6
private Persona persona; // Cliente contiene un objeto Persona

2. Celular contiene Marca
Archivo: Model/Celular.java, líneas 7
private Marca marca; // Celular contiene un objeto Marca

3. Venta contiene Cliente y Lista de Detalles
Archivo: Model/Venta.java, líneas 10-11
private Cliente cliente;
private List<Detalle_venta> detalles;

4. Detalle_venta contiene Celular y Venta
Archivo: Model/Detalle_venta.java, líneas 6-7
private Celular celular;
private Venta venta;

[CAPTURA 6.2A: Diagrama o código mostrando las relaciones de composición(no supe que captura va aca, creo que asi esta bien)]

6.3 Encapsulamiento

Archivos: Todas las clases del modelo (Model/*)

Propósito: Todas las clases del modelo implementan correctamente el principio de encapsulamiento con atributos privados y métodos públicos getters/setters.

Ejemplo en la clase Celular:

```
public class Celular {
    // Atributos PRIVADOS
    private int id, stock;
    private String modelo;
    private double precio;

    // Métodos PÚBLICOS para acceso controlado
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    // ... más getters y setters
}
```

Beneficios del encapsulamiento:

- Protección de datos
- Control de acceso
- Validación en setters
- Flexibilidad para cambios internos

```
public class Persona {
    private int id;
    private String nombre, identificacion, correo, telefono;


    //Constructor completo para traer desde BD
    public Persona(int id, String nombre, String identificacion,
        this.id = id;
        this.nombre = nombre;
        this.identificacion = identificacion;
        this.correo = correo;
        this.telefono = telefono;
    }
```

RESUMEN DE CUMPLIMIENTO

Este documento evidencia el cumplimiento completo de todos los requisitos del proyecto TecnoStore:

✓ Gestión de Celulares con validaciones de precio y stock
✓ Gestión de Clientes con validación de correo (Regex) e identificación única
✓ Gestión de Ventas con cálculo de IVA 19% y actualización de stock
✓ Reportes con Stream API (stock bajo, top 3, ventas por mes)
✓ Persistencia con JDBC, patrón DAO, try-with-resources y PreparedStatement
✓ Generación de archivo reporte_ventas.txt
✓ Patrón Singleton en la conexión a base de datos
✓ Aplicación correcta de POO: encapsulamiento, composición y enums