Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Introducción a la Programación y Computación 1

Sección F

Ing. William Escobar

Aux. Zenaida Chacón

Proyecto 2 – Manual técnico

Sancarlista Shop

Eldan André Escobar Asturias

Carné 202303088

24/10/2025

INTRODUCCIÓN

El presente sistema de gestión fue desarrollado con el objetivo de optimizar el control interno de una tienda que administra múltiples tipos de usuarios y procesos comerciales. La aplicación permite la interacción de tres roles principales: Administrador, Vendedor y Cliente, cada uno con acceso a funciones específicas que reflejan la estructura jerárquica y operativa de una tienda real.

El sistema facilita la administración del inventario de productos, la gestión de vendedores y clientes, y el registro de compras realizadas por los usuarios finales. A través del rol Administrador, es posible llevar a cabo tareas de creación, modificación y eliminación (CRUD) de vendedores y productos, además de generar reportes informativos. El rol Vendedor permite la atención del cliente a través de la gestión de cuentas, pedidos y actualización de existencias. Finalmente, el rol Cliente cuenta con acceso a un catálogo de productos, un carrito de compras dinámico y un historial propio de transacciones realizadas.

La aplicación fue implementada en el lenguaje Java, utilizando la biblioteca Swing para la construcción de la interfaz gráfica (GUI). Esta decisión favoreció el desarrollo de una plataforma intuitiva, modular y escalable, donde la comunicación entre ventanas y estructuras de datos se maneja de manera clara y eficiente. Para el almacenamiento de la información se emplearon vectores estáticos, permitiendo la organización de datos en memoria durante la ejecución del programa, sin requerir una base de datos externa.

Este sistema busca no solo resolver una necesidad técnica, sino también servir como base para el aprendizaje y reforzamiento de conceptos fundamentales como programación orientada a objetos, manejo de estructuras de datos, diseño de interfaces gráficas, encapsulamiento y separación lógica por capas. Gracias a su diseño modular, el programa puede ser ampliado y mejorado en futuras versiones, lo que lo convierte en una herramienta sólida tanto para uso práctico como académico.

GLOSARIO DE TÉRMINOS CLAVE

- package: Define un conjunto de clases agrupadas bajo un mismo espacio de nombres.
- import: Permite usar clases de otras librerías o paquetes en el programa.
- public: Modificador de acceso que indica que una clase, método o variable es accesible desde cualquier otra clase.
- private: Modificador de acceso que limita la visibilidad únicamente dentro de la misma clase.
- String: Clase en Java que representa cadenas de texto.
- extends: Palabra clave para indicar herencia entre clases. super: Permite acceder a métodos o constructores de la clase padre.
- void: Tipo de retorno que indica que un método no devuelve ningún valor.
- static: Permite acceder a un método o variable sin necesidad de crear una instancia de la clase. new: Operador que crea un objeto en memoria.
- this: Referencia al objeto actual de la clase.

DICCIONARIO DE MÉTODOS

- setVisible(true): Muestra la ventana gráfica en pantalla.
- addActionListener(): Asocia un evento (como clic en un botón) a un método que ejecuta una acción.
- getSelectedItem(): Obtiene el valor actual seleccionado en un JComboBox.
- append(): Agrega texto al final de un JTextArea.
- setText(): Establece el contenido de un campo de texto o etiqueta.
- getText(): Recupera el contenido escrito en un campo de texto.
- setEnabled(): Activa o desactiva un componente gráfico.
- setTitle(): Cambia el título de la ventana principal.
- dispose(): Cierra una ventana sin finalizar toda la aplicación.
- println(): Imprime un mensaje en consola.
- write(): Escribe datos en un archivo de texto.
- readLine(): Lee una línea de texto de un archivo o flujo de entrada.

POSIBLES MEJORAS A FUTURO

Una posible mejora importante sería migrar el almacenamiento de datos desde vectores hacia una base de datos, por ejemplo MySQL, SQLite o Firebase. Esto permitiría conservar la información de manera permanente incluso después de cerrar el programa, además de facilitar la escalabilidad y manejo de mayores volúmenes de datos.

Otra mejora sería implementar control real de inventario, de manera que el stock de productos se actualice automáticamente cuando se registren compras o ingresos de mercancía. Esto haría el sistema más útil en un entorno comercial real.

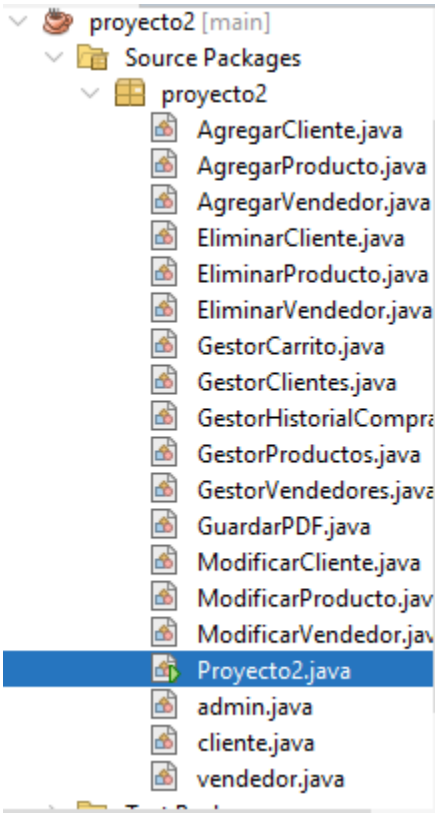
En cuanto a la seguridad, sería beneficioso cifrar las contraseñas utilizando algoritmos como SHA-256 o BCrypt, evitando que las credenciales se almacenen en texto plano dentro de los vectores.

También se podría implementar una generación automática de reportes avanzados en PDF, utilizando librerías como iText o Apache PDFBox, para proporcionar al administrador informes visuales sobre ventas, productos más vendidos, ingresos totales, entre otros.

A nivel de usabilidad, el sistema podría evolucionar hacia una versión web o móvil, utilizando frameworks como Spring Boot para el backend y Angular, React o Flutter para la interfaz. Esto permitiría acceso desde múltiples dispositivos y soporte multiusuario en tiempo real.

Finalmente, otra mejora importante sería separar el historial de compras por cliente, de manera que cada uno pueda visualizar únicamente sus propias transacciones, fortaleciendo la privacidad y precisión del registro de datos dentro del sistema.

EXPLICACIÓN DE CÓDIGO



El programa funciona gracias a estas 19 clases, siento proyecto2 la principal, en la cual se encuentra el login.

```
42          JLabel lblUsuario = new JLabel("Usuario:");
43          gbc.gridx = 0;
44          add(lblUsuario, gbc);
45
46          txtUsuario = new JTextField();
47          gbc.gridx = 1;
48          add(txtUsuario, gbc);
49
50          gbc.gridy++;
51          gbc.gridx = 0;
52          JLabel lblContrasena = new JLabel("Contraseña:");
53          add(lblContrasena, gbc);
54
55          txtContrasena = new JPasswordField();
56          gbc.gridx = 1;
57          add(txtContrasena, gbc);
58
59          gbc.gridy++;
60          gbc.gridx = 0;
61          btnIngresar = new JButton("Ingresar");
62          add(btnIngresar, gbc);
63
64          gbc.gridx = 1;
65          btnSalir = new JButton("Salir");
66          add(btnSalir, gbc);
67
68          btnIngresar.addActionListener(e -> validarCredenciales());
69
70          btnSalir.addActionListener(e -> System.exit(0));
71
72          setVisible(true);
73      }
```

Este fragmento de código pertenece a la ventana de inicio de sesión del programa y lo que hace es colocar los elementos gráficos —la etiqueta y el campo para escribir el usuario, la etiqueta y el campo para la contraseña, y los botones de ingresar y salir— usando un diseño llamado GridBagLayout, que organiza los componentes por filas y columnas. Cada vez que se usa gbc.gridx y gbc.gridy se le dice al programa en qué posición colocar el elemento. Por ejemplo, las etiquetas van en la columna 0 y los campos de texto en la columna 1, formando dos filas para usuario y contraseña. Al final, se agregan los botones, uno en cada columna de la última fila. Además, al botón "Ingresar" se le asigna la acción de verificar las credenciales, mientras que el botón "Salir" cierra la aplicación. Con esto se construye la interfaz del login de manera ordenada.

```java
private void validarCredenciales() {
    String u = txtUsuario.getText().trim();
    String p = new String(txtContrasena.getPassword());

    //inicio de sesión para administrador
    if (u.equals("admin") && p.equals("IPC1F")) {
        JOptionPane.showMessageDialog(this, "Bienvenido administrador");
        new admin().setVisible(true);
        dispose();
        return;
    }

    //inicio de sesión para vendedor con datos registrados por administrador
    int idx = GestorVendedores.buscarIndicePorCodigo(u);
    if (idx == -1) {
        for (int i = 0; i < GestorVendedores.getTotal(); i++) {
            if (GestorVendedores.getNombres()[i].equalsIgnoreCase(u)) {
                idx = i;
                break;
            }
        }
    }

    if (idx != -1 && GestorVendedores.getContrasenas()[idx].equals(p)) {
        JOptionPane.showMessageDialog(this, "Bienvenido vendedor: " + GestorVendedores.getNombres()[idx]);
        new vendedor(GestorVendedores.getNombres()[idx]).setVisible(true);
        dispose();
        return;
    } else {
        JOptionPane.showMessageDialog(this, "Usuario o contraseña incorrectos", "Error", JOptionPane.ERROR_MESSAGE);
        txtContrasena.setText("");
    }

    //iniciar sesión cliente
    int idCliente = GestorClientes.buscarIndicePorCodigo(u);
    if (idCliente == -1) {
        for (int i = 0; i < GestorClientes.getTotal(); i++) {
            if (GestorClientes.getNombres()[i].equalsIgnoreCase(u)) {
                idCliente = i;
                break;
            }
        }
    }
}
```

Este método validarCredenciales() se encarga de comprobar si el usuario y contraseña ingresados coinciden con alguno de los tres tipos de usuario del sistema: administrador, vendedor o cliente. Primero, obtiene lo que se escribió en los campos de texto y lo guarda en variables. Luego, compara si el usuario es “admin” y la contraseña “IPC1F”; si coinciden, muestra un mensaje de bienvenida y abre la ventana del administrador. Si no es administrador, busca el usuario en la lista de vendedores utilizando sus vectores; si lo encuentra y la contraseña es correcta, entra al panel del vendedor. De lo contrario, muestra un mensaje de error. Finalmente, hace ese mismo proceso con los clientes, buscando su nombre y clave en los vectores correspondientes. En resumen, este bloque decide a qué ventana enviar al usuario dependiendo de sus credenciales o si mostrar un mensaje de error si son incorrectas.

```
32          //tabla primera pestaña
33          JPanel panelVend = new JPanel(new BorderLayout(8,8));
34          modeloVendedores = new DefaultTableModel(new String[]{"Código","Nombre","Género","Ventas Confirmadas"}, 0) {
35              @Override public boolean isCellEditable(int r,int c){ return false; }
36          };
37          tablaVendedores = new JTable(modeloVendedores);
38          panelVend.add(new JScrollPane(tablaVendedores), BorderLayout.CENTER);
39
40          JPanel accionesVend = new JPanel(new GridLayout(3,1,10,10));
41          JButton btnAddV = new JButton("Agregar");
42          JButton btnModV = new JButton("Modificar");
43          JButton btnDelV = new JButton("Eliminar");
44          accionesVend.add(btnAddV); accionesVend.add(btnModV); accionesVend.add(btnDelV);
45          panelVend.add(accionesVend, BorderLayout.EAST);
46
47          btnAddV.addActionListener(e -> { new AgregarVendedor().setVisible(true); dispose(); });
48          btnModV.addActionListener(e -> { new ModificarVendedor().setVisible(true); dispose(); });
49          btnDelV.addActionListener(e -> { new EliminarVendedor().setVisible(true); dispose(); });
50
51          tabs.add("Vendedores", panelVend);
```

Este fragmento corresponde a la pestaña de Vendedores dentro del panel del administrador. Primero, se crea un panel con BorderLayout para colocar la tabla en el centro y los botones en un área aparte. Luego se define el modelo de la tabla (DefaultTableModel) con las columnas “Código”, “Nombre”, “Género” y “Ventas Confirmadas”, y se sobrescribe el método isCellEditable para que ninguna celda sea editable directamente. Después, se crea la tabla basada en ese modelo y se agrega dentro de un JScrollPane para permitir el desplazamiento. En la parte derecha, se genera un panel adicional que utiliza GridLayout para colocar los botones Agregar, Modificar y Eliminar uno debajo del otro. Finalmente, a cada botón se le asigna un ActionListener que abre la ventana correspondiente (AgregarVendedor, ModificarVendedor o EliminarVendedor) y cierra la ventana actual. Todo esto se agrega al JTabbedPane bajo la pestaña llamada “Vendedores”.

```
93      private void cargarTablas() {
94          //tabla de primera pestaña
95          modeloVendedores.setRowCount(0);
96          String[] c = GestorVendedores.getCodigos();
97          String[] n = GestorVendedores.getNombres();
98          String[] g = GestorVendedores.getGeneros();
99          int[] v = GestorVendedores.getVentasConfirmadas();
100
101         for (int i = 0; i < GestorVendedores.getTotal(); i++) {
102             modeloVendedores.addRow(new Object[]{ c[i], n[i], g[i], v[i] });
103         }
104
105         //tabla de segunda pestaña
106         modeloProductos.setRowCount(0);
107         String[] pc = GestorProductos.getCodigos();
108         String[] pn = GestorProductos.getNombres();
109         String[] cat = GestorProductos.getCategorias();
110         String[] att = GestorProductos.getAtributoUnico();
111         double[] pre = GestorProductos.getPrecios();
112
113         for (int i = 0; i < GestorProductos.getTotal(); i++) {
114             modeloProductos.addRow(new Object[]{ pc[i], pn[i], cat[i], att[i], pre[i] });
115         }
116     }
117
118     @Override
119     public void setVisible(boolean b) {
120         super.setVisible(b);
121         if (b) cargarTablas();
122     }
```

Este método cargarTablas() se encarga de actualizar la información mostrada en las tablas de la pestaña de vendedores y la pestaña de productos. Primero, borra cualquier dato previo en la tabla de vendedores usando setRowCount(0) y luego obtiene los vectores con los códigos, nombres, géneros y ventas confirmadas desde GestorVendedores. Después recorre esos vectores con un ciclo for y agrega cada registro como una fila en la tabla. Luego hace lo mismo para la tabla de productos: limpia la tabla, obtiene los vectores de códigos, nombres, categorías, atributo único y precios desde GestorProductos, y agrega fila por fila a la tabla correspondiente. Finalmente, se sobrescribe el método setVisible() para que cada vez que esta ventana se muestre en pantalla, se vuelvan a cargar los datos actualizados en ambas tablas automáticamente.

```
11  public class GestorVendedores {
12      private static final int MAX = 200;
13      private static String[] codigos = new String[MAX];
14      private static String[] nombres = new String[MAX];
15      private static String[] generos = new String[MAX];
16      private static String[] contrasenas = new String[MAX];
17      private static int[] ventasConfirmadas = new int[MAX];
18      private static int total = 0;
19
20      public static boolean agregar(String codigo, String nombre, String genero, String contrasena) {
21          if (total >= MAX) return false;
22          if (buscarIndicePorCodigo(codigo) != -1) return false; //validacion de id
23          codigos[total] = codigo;
24          nombres[total] = nombre;
25          generos[total] = genero;
26          contrasenas[total] = contrasena;
27          ventasConfirmadas[total] = 0;
28          total++;
29          return true;
30      }
31
32      public static boolean modificar(String codigo, String nuevoNombre, String nuevaContrasena) {
33          int i = buscarIndicePorCodigo(codigo);
34          if (i == -1) return false;
35          nombres[i] = nuevoNombre;
36          contrasenas[i] = nuevaContrasena;
37          return true;
38      }
39
40      public static boolean eliminar(String codigo) {
41          int i = buscarIndicePorCodigo(codigo);
42          if (i == -1) return false;
43          for (int j=i; j<total-1; j++) {
44              codigos[j]=codigos[j+1];
45              nombres[j]=nombres[j+1];
46              generos[j]=generos[j+1];
47              contrasenas[j]=contrasenas[j+1];
48              ventasConfirmadas[j]=ventasConfirmadas[j+1];
49          }
50          total--;
51          return true;
```

Esta clase GestorVendedores funciona como el administrador de información de los vendedores dentro del programa. Aquí se guardan los datos en vectores estáticos como códigos, nombres, géneros, contraseñas y ventas confirmadas, y una variable total que indica cuántos vendedores están registrados. El método agregar() permite crear un nuevo vendedor siempre y cuando no exista ya un vendedor con el mismo código, y lo guarda en la siguiente posición disponible en los vectores. El método modificar() busca el vendedor por su código y, si lo encuentra, actualiza su nombre y contraseña. El método eliminar() también busca al

vendedor y, si existe, recorre los vectores desplazando los siguientes valores hacia arriba para llenar el espacio y luego reduce total en uno. Con esto, la clase se encarga de manejar el CRUD básico de los vendedores manteniendo la información organizada dentro del sistema.

```
54      public static int buscarIndicePorCodigo(String codigo) {
55          for (int i=0; i<total; i++) if (codigos[i].equals(codigo)) return i;
56          return -1;
57      }
58
59      //getters y setters
60      public static int getTotal() {
61          return total;
62      }
63
64      public static String[] getCodigos() {
65          return codigos;
66      }
67
68      public static String[] getNombres() {
69          return nombres;
70      }
71
72      public static String[] getGeneros() {
73          return generos;
74      }
75
76      public static int[] getVentasConfirmadas() {
77          return ventasConfirmadas;
78      }
79
80      public static String[] getContrasenas() {
81          return contrasenas;
82      }
83  }
84
```

Esta parte de la clase contiene el método buscarIndicePorCodigo(), que se encarga de recorrer el vector de códigos para encontrar la posición donde se encuentra un vendedor específico; si lo encuentra, devuelve el índice, y si no, devuelve -1 para indicar que no existe. Después aparecen los métodos getters, los cuales permiten acceder desde otras clases a los vectores y a la variable total. Por ejemplo, getTotal() devuelve la cantidad actual de vendedores registrados, mientras que métodos como getCodigos(), getNombres(), getGeneros(), getVentasConfirmadas() y getContrasenas() retornan los arreglos completos con esa información. Estos getters son importantes porque permiten que otras partes del programa, como las tablas en la interfaz, puedan obtener y mostrar los datos sin modificar directamente los vectores.

```java
public AgregarCliente() {
    //diseño de la ventana
    setTitle("Agregar Cliente");
    setSize(400, 300);
    setLocationRelativeTo(null);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    setLayout(new GridLayout(6,2,8,8));

    //componentes para texto
    add(new JLabel("Código:")); txtCodigo = new JTextField(); add(txtCodigo);
    add(new JLabel("Nombre:")); txtNombre = new JTextField(); add(txtNombre);
    add(new JLabel("Género:")); cmbGenero = new JComboBox<>(new String[]{"M","F"}); add(cmbGenero);
    add(new JLabel("Cumpleaños (dd/mm/aaaa):")); txtFecha = new JTextField(); add(txtFecha);
    add(new JLabel("Contraseña:")); txtContrasena = new JTextField(); add(txtContrasena);

    JButton btnGuardar = new JButton("Guardar");
    JButton btnVolver = new JButton("Volver");
    add(btnGuardar); add(btnVolver);

    btnGuardar.addActionListener(e -> guardar());
    btnVolver.addActionListener(e -> { new vendedor("Vendedor").setVisible(true); dispose(); });
}
```

Este constructor pertenece a la ventana AgregarCliente, y aquí se define cómo se verá la interfaz de esta pantalla. Primero se establece el título de la ventana, su tamaño, posición y el comportamiento al cerrarla. Luego, se usa un GridLayout para organizar los campos de entrada en forma de formulario. Se agregan etiquetas y cajas de texto para que el usuario pueda ingresar el código, el nombre, el género del cliente mediante un JComboBox, la fecha de cumpleaños y la contraseña. Después, se crean dos botones: Guardar, que al presionarse ejecuta el método guardar() para registrar al cliente, y Volver, que cierra esta ventana y regresa al panel del vendedor. En resumen, este código construye la interfaz visual y define la interacción básica para registrar nuevos clientes.

```java
private void guardar() {
    String c = txtCodigo.getText().trim();
    String n = txtNombre.getText().trim();
    String g = (String) cmbGenero.getSelectedItem();
    String f = txtFecha.getText().trim();
    String p = txtContrasena.getText().trim();

    if (c.isEmpty() || n.isEmpty() || f.isEmpty() || p.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Error, llene todos los campos"); return;
    }

    if (GestorClientes.agregar(c, n, g, f, p)) {
        JOptionPane.showMessageDialog(this, "Cliente agregado exitosamente");
        new vendedor("Vendedor").setVisible(true);
        dispose();
    } else {
        JOptionPane.showMessageDialog(this, "Error, codigo duplicado ", "Error", JOptionPane.ERROR_MESSAGE);
    }
}
}
```

Este método guardar() se encarga de tomar los datos que el usuario ingresó en los campos de texto de la ventana AgregarCliente. Primero obtiene el código, nombre, género, fecha de cumpleaños y contraseña, eliminando posibles espacios extra. Luego verifica que ninguno de estos campos esté vacío; si falta alguno, muestra un mensaje pidiendo completar todo antes de continuar. Si los datos están completos, llama al método GestorClientes.agregar() para intentar registrar al cliente en el sistema. Si la operación es exitosa, muestra un mensaje de confirmación y vuelve a la ventana principal del vendedor. Si el código ya existía en el sistema y no se puede agregar, muestra un mensaje de error indicando que el código está duplicado.

En resumen, este método valida la información y realiza el registro real del cliente en el sistema.

```
34      private void eliminar() {
35          String c = txtCodigo.getText().trim();
36          int i = GestorVendedores.buscarIndicePorCodigo(c);
37          if (i == -1) {
38              JOptionPane.showMessageDialog(this, "No existe el vendedor", "Error", JOptionPane.ERROR_MESSAGE);
39              return;
40          }
41          String resumen = "Código: " + c +
42                  "\nNombre: " + GestorVendedores.getNombres()[i] +
43                  "\nGénero: " + GestorVendedores.getGeneros()[i];
44          int op = JOptionPane.showConfirmDialog(this, "¿Eliminar?\n\n" + resumen, "Confirmación", JOptionPane.YES_NO_OPTION);
45          if (op == JOptionPane.YES_OPTION) {
46              if (GestorVendedores.eliminar(c)) {
47                  JOptionPane.showMessageDialog(this, "Vendedor eliminado exitosamente");
48                  new admin().setVisible(true);
49                  dispose();
50              } else {
51                  JOptionPane.showMessageDialog(this, "No se pudo eliminar", "Error", JOptionPane.ERROR_MESSAGE);
52              }
53          }
54      }
```

Este método eliminar() se encarga de borrar un vendedor del sistema. Primero toma el código ingresado y busca su posición en los vectores usando buscarIndicePorCodigo(). Si el vendedor no existe, muestra un mensaje de error y termina. Si sí existe, construye un resumen con sus datos (código, nombre y género) y muestra una ventana de confirmación para preguntar si realmente se desea eliminarlo. Si el usuario confirma, se llama al método GestorVendedores.eliminar(c) para removerlo de los vectores y luego se muestra un mensaje indicando que fue eliminado con éxito y se regresa a la ventana principal del administrador. Si el usuario elige que no, o si ocurre algún problema, se muestra un mensaje indicando que no se pudo eliminar. En resumen, este método confirma la acción y realiza la eliminación de forma segura para evitar errores.

```
41      private void cargarDatos() { //validación de datos
42          String codigo = txtBuscarCodigo.getText().trim();
43          int i = GestorVendedores.buscarIndicePorCodigo(codigo);
44          if (i == -1) {
45              JOptionPane.showMessageDialog(this, "No existe el vendedor", "Error", JOptionPane.ERROR_MESSAGE);
46              return;
47          }
48          txtNombre.setText(GestorVendedores.getNombres()[i]);
49          txtContrasena.setText(GestorVendedores.getContrasenas()[i]);
50      }
```

Este método cargarDatos() se usa dentro de la ventana para modificar la información de un vendedor. Lo primero que hace es obtener el código ingresado en el campo de texto y buscarlo dentro de los vectores de vendedores usando el método buscarIndicePorCodigo(). Si el código no existe, muestra un mensaje de error informando que el vendedor no fue encontrado y detiene la ejecución del método. Si sí existe, obtiene el nombre y la contraseña que corresponden a ese vendedor y los muestra automáticamente en los cuadros de texto del formulario, de modo que el usuario pueda ver y editar los datos fácilmente. En resumen, este método valida la existencia del vendedor y carga su información actual en los campos para permitir su modificación.

```
52      private void guardarCambios() { //extraer el texto del textbox y guardarlo en los vectores
53          String codigo = txtBuscarCodigo.getText().trim();
54          String nombre = txtNombre.getText().trim();
55          String pass = txtContrasena.getText().trim();
56          if (codigo.isEmpty() || nombre.isEmpty() || pass.isEmpty()) {
57              JOptionPane.showMessageDialog(this, "Error, llene todos los campos"); return;
58          }
59          if (GestorVendedores.modificar(codigo, nombre, pass)) {
60              JOptionPane.showMessageDialog(this, "Cambios guardados");
61              new admin().setVisible(true);
62              dispose();
63          } else {
64              JOptionPane.showMessageDialog(this, "Error, no se pudo modificar", "Error", JOptionPane.ERROR_MESSAGE);
65          }
66      }
67  }
```

Este método guardarCambios() se encarga de actualizar los datos de un vendedor después de haberlos cargado en los campos de texto. Primero obtiene el código, el nombre y la nueva contraseña ingresados por el usuario, eliminando espacios innecesarios. Luego revisa que ninguno de esos campos esté vacío; si falta algo, muestra un mensaje indicando que se deben completar todos los datos. Si todo está correcto, llama al método GestorVendedores.modificar() para actualizar la información en los vectores. Si la modificación se realiza con éxito, muestra un mensaje de confirmación y regresa a la ventana principal del administrador. Si por alguna razón no se puede modificar (por ejemplo, si el vendedor no existe), muestra un mensaje de error. En resumen, este método valida la edición y guarda los cambios realizados en los datos del vendedor.

Y así se repiten los códigos para las siguientes clases, solo varían en cosas mínimas.

CONCLUSIÓN

El sistema desarrollado cumple satisfactoriamente con las necesidades de gestión para los diferentes tipos de usuarios involucrados. Se logró implementar una interfaz gráfica clara y funcional que facilita el acceso y manipulación de la información relacionada con productos, vendedores, clientes y compras, permitiendo agilizar procesos que normalmente requerirían mayor tiempo y control manual.

El enfoque de desarrollo empleado, basado en Programación Orientada a Objetos y estructuras de datos en vectores, permitió construir una aplicación coherente, organizada y con una lógica de funcionamiento fácil de mantener y comprender. Además, la estructura del sistema se diseñó con una visión de escalabilidad, de manera que pueda incorporar nuevas funciones en futuras versiones, tales como conexión a bases de datos externas, mayor seguridad en el manejo de contraseñas o implementación de reportes más avanzados.

En definitiva, este sistema no solo representa una solución funcional para la gestión integral de una tienda, sino también una base educativa sólida que demuestra la correcta aplicación de los principios fundamentales de diseño y desarrollo de software. A medida que se continúe mejorando, este proyecto podrá evolucionar hacia una plataforma más robusta, adaptable e integrada con tecnologías modernas.