

IES Francisco de los Ríos

Título del proyecto:

RetroDrip – Sistema de Pedidos y Gestión de Productos Retro



Alumno:

Manuel Laguna Prieto

Curso:

1º DAM – Desarrollo de Aplicaciones Multiplataforma

<https://github.com/Kmanuu/ProgramacionProyectoFinal.git>

ÍNDICE

1. Tecnologías Utilizadas	pág. 2
2. Resumen del Proyecto	pág. 3
3. Modelo E/R y Estructura de Tablas	pág. 4
4. Diseño Orientado a Objetos	pág. 5
5. DAOs y Consultas	pág. 6
6. Interfaz y Funcionalidades	pág. 7
7. Configuración Técnica	pág. 8
8. Fase de Desarrollo	pág. 9
9. Lista de Comprobación	pág. 10
10. Conclusiones	pág. 11

1 Tecnologías utilizadas:

- Java
- JavaFX
- MySQL
- JDBC
- JAXB
- XAMPP
- ChatGPT
- Maven
- IntelliJ

2. Resumen del Proyecto

RetroDrip es una aplicación de escritorio desarrollada con JavaFX que simula una tienda online de ropa retro, como camisetas de fútbol vintage y zapatillas. El sistema permite que los usuarios se registren como clientes o vendedores. Los clientes pueden ver productos disponibles y hacer pedidos, mientras que los vendedores pueden añadir, editar o eliminar productos del catálogo. La aplicación utiliza una base de datos MySQL y una conexión configurada externamente mediante XML y JAXB.

EXTRA :

Quiero dejar claro que la clase `PedidoProducto` puede parecer un poco rara o confusa al principio, porque no representa un objeto "principal" como lo sería un `Producto` o un `Pedido`, sino que es una tabla intermedia que relaciona ambos.

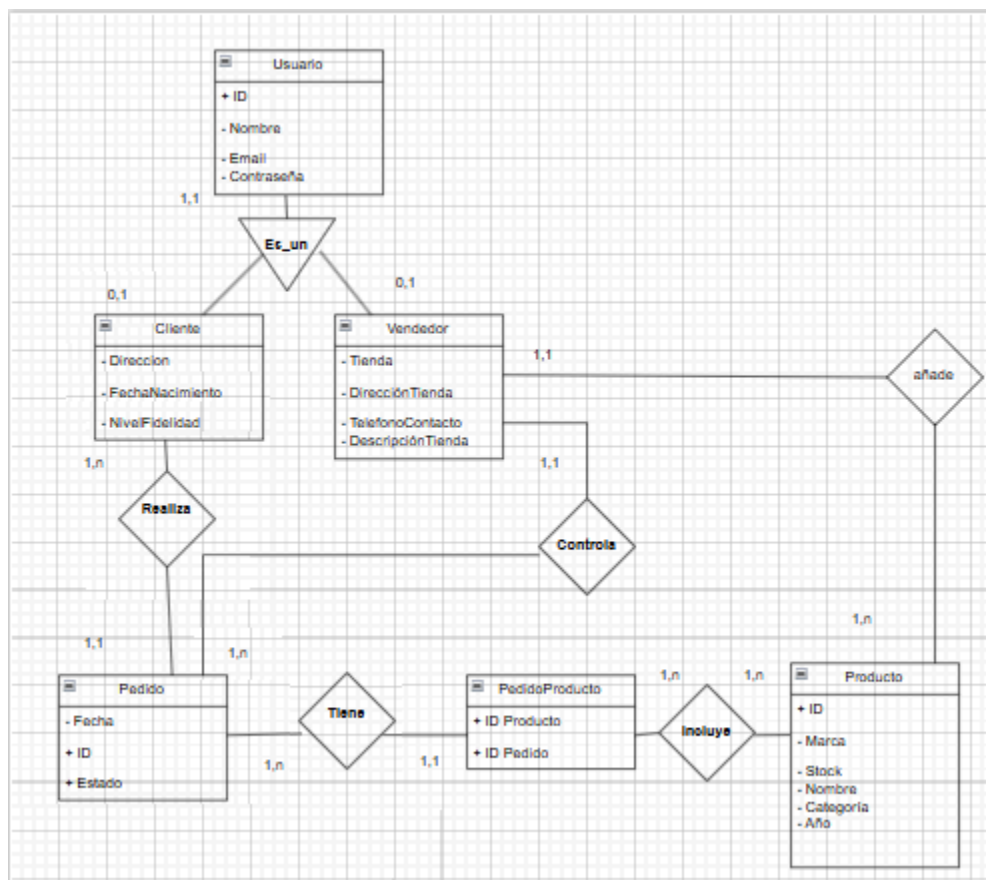
Aun así, es una de las piezas más importantes del sistema, porque:

- Gestiona la relación N:M entre pedidos y productos.
- Permite registrar cuántos productos hay en cada pedido.
- Ayuda a separar responsabilidades y no mezclar lógica en otras clases.

Sin esta clase, no podríamos saber qué productos tiene un pedido ni cuántas unidades se pidieron. Además, gracias a que está separada en su propio DAO, el código está mucho más limpio y es fácil de mantener o ampliar (por ejemplo, para añadir precios, descuentos o devoluciones en el futuro).

3. Modelo E/R y Estructura de Tablas

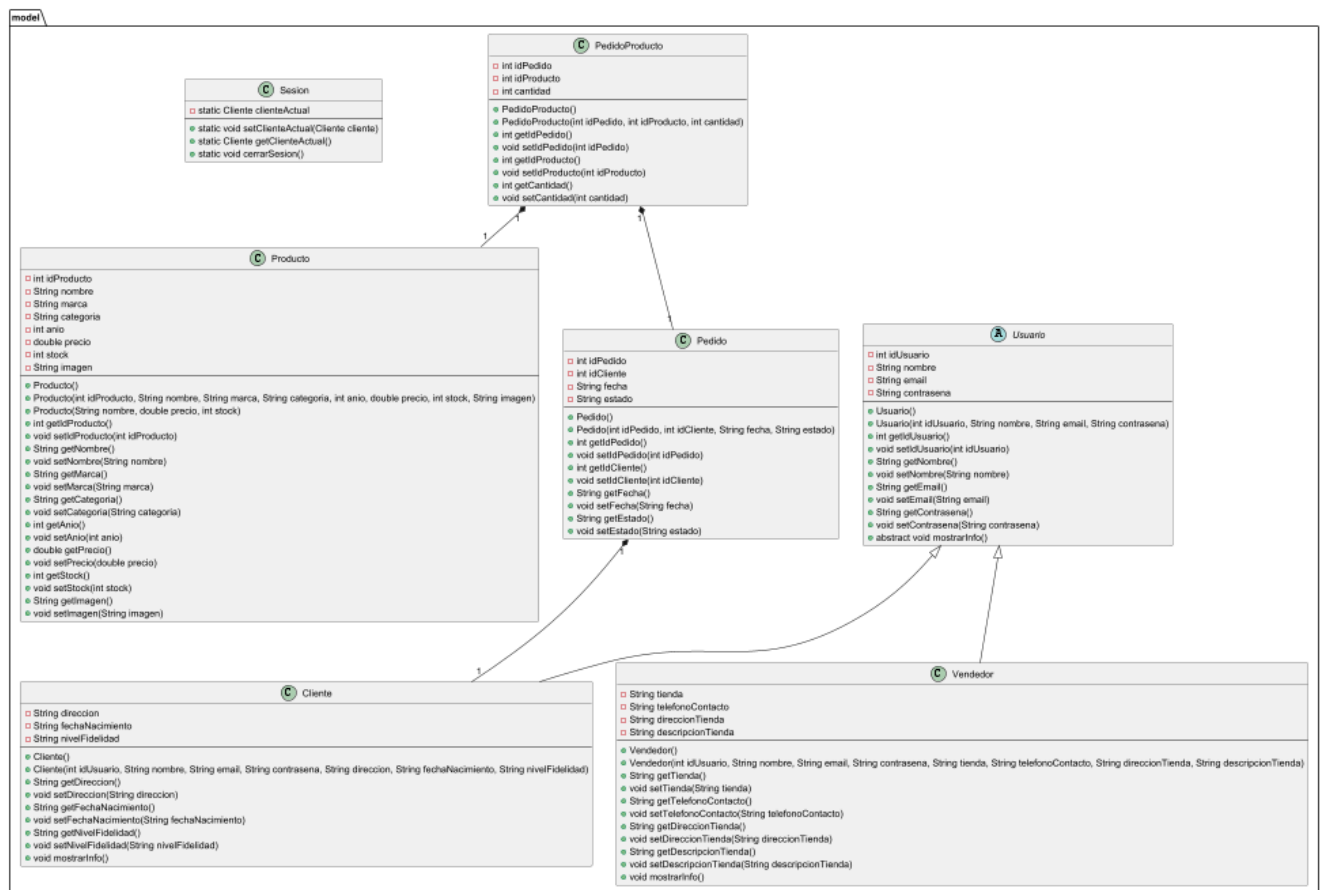
A continuación se muestra el modelo entidad-relación del sistema. Este modelo representa las relaciones entre usuarios, productos y pedidos.



Las tablas creadas en la base de datos son: usuario, cliente, vendedor, producto, pedido y pedido_producto. Se han definido claves primarias y foráneas que mantienen la integridad referencial. Por ejemplo, cada pedido se vincula a un cliente y cada producto a un vendedor. La tabla intermedia pedido_producto permite gestionar la relación N:M entre pedidos y productos. También se han implementado restricciones como NOT NULL, UNIQUE y claves compuestas cuando ha sido necesario.

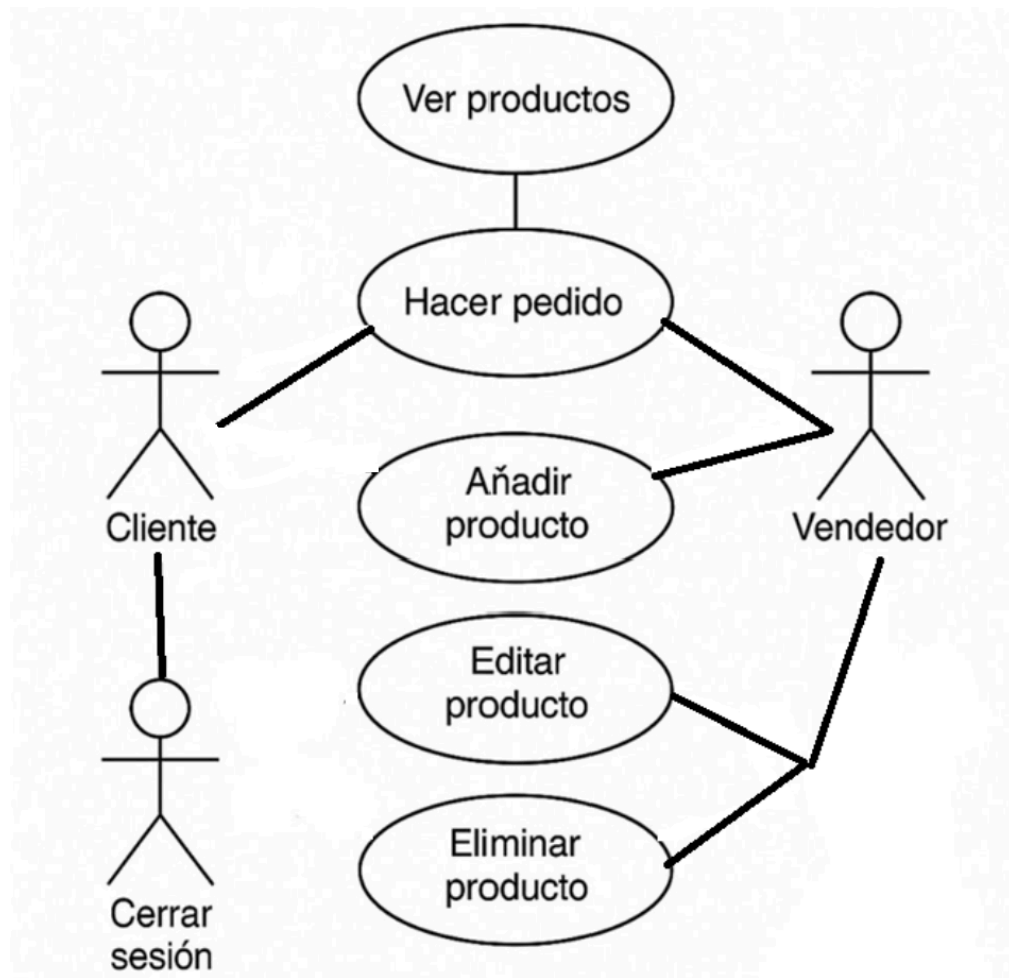
4. Diseño Orientado a Objetos

El proyecto está estructurado aplicando principios de programación orientada a objetos. Se ha utilizado herencia para representar las entidades Usuario (superclase), Cliente y Vendedor (subclases). Cada entidad cuenta con su clase DAO respectiva. También se han aplicado conceptos como encapsulamiento y reutilización de código.



El diseño incluye herencia clara: Usuario es la superclase, de la cual heredan Cliente y Vendedor. Las clases reflejan fielmente las tablas del modelo relacional y están conectadas con sus respectivos DAOs.

Diagrama de casos de Uso



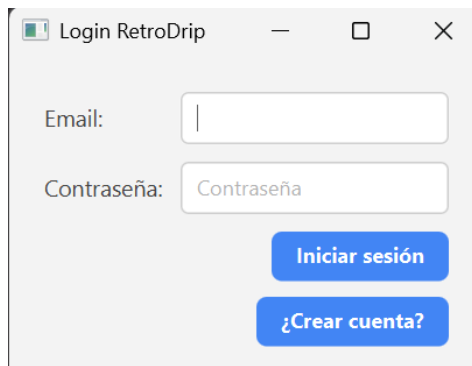
5. DAOs y Consultas

Se han implementado clases DAO para todas las entidades principales. Cada clase contiene métodos para realizar operaciones CRUD (insertar, buscar, actualizar y eliminar). Además, las consultas están filtradas por campos clave, como el ID del cliente o del vendedor. Las relaciones entre pedidos y productos se manejan mediante una tabla intermedia y operaciones con Joins.

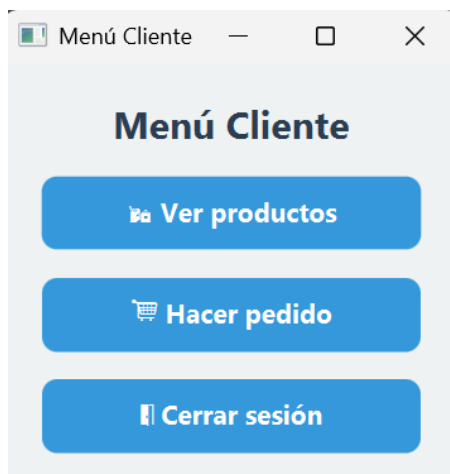
6. Interfaz y Funcionalidades

La interfaz depende de como te registres en el login, primero deberás de iniciar sesión siendo cliente o vendedor o creando una cuenta en la que defines tu rol, si eres cliente tendrás 3 opciones, ver productos, hacer un pedido y cerrar sesión; en cambio si eres vendedor tendrás más opciones : ver producto, editar/borrar producto, añadir producto y cerrar sesión a continuación veremos ambos caminos.

Cliente :



A screenshot of a web application window titled "Login RetroDrip". It features two input fields: "Email:" and "Contraseña:". Below the "Contraseña:" field are two blue buttons: "Iniciar sesión" and "¿Crear cuenta?".



A screenshot of a web application window titled "Menú Cliente". It displays three blue buttons stacked vertically: "Ver productos" (with a magnifying glass icon), "Hacer pedido" (with a shopping cart icon), and "Cerrar sesión" (with a door icon).

Ver productos		
Nombre	Precio	Stock
Camiseta milan 2006	26.0	5
Argentina 2006 visitante	35.0	2
Fiorentina 1994	50.0	3
Madrid 1999	100.0	10
Vaqueros baggy grises	40.0	2
Volver al menú		

Hacer Pedido

Hacer Pedido

Camiseta milan 2006 (n... ▾)

1|

Confirmar

Volver

Y ahora vendedor :

Menú Vend...

Menú Vendedor

Ver productos

+ Añadir producto

Editar / Eliminar pr...

Cerrar sesión

Productos		
Nombre	Precio	Stock
Camiseta milan 2006	26.0	5
Argentina 2006 visitante	35.0	2
Florentina 1994	50.0	3
Madrid 1999	100.0	10
Vaqueros baggy grises	40.0	2

[Volver al menú](#)

 Añadir Producto

Nombre:

Precio:

Stock:

Guardar

Volver

Editor / Eliminar Producto

Nombre	Precio	Stock
Camiseta milan 2006	26.0	5
Argentina 2006 visitante	35.0	2
Florentina 1994	50.0	3
Madrid 1999	100.0	10
Vaqueros baggy grises	40.0	2

Editar
Eliminar
Volver

7. Configuración Técnica

El proyecto utiliza Maven como sistema de gestión de dependencias. La conexión con la base de datos está externalizada en un archivo XML y se carga utilizando JAXB. El archivo pom.xml incluye todas las dependencias necesarias como el conector de MySQL y las librerías de JAXB.

Puntos fuertes del diseño:

- Separación clara entre modelo, lógica de negocio y vista
- Uso de herencia real y funcional
- CRUD completo validado
- Conexión flexible con JAXB y XML
- Proyecto estructurado como Maven estándar

Funcionamiento de los botones

En general :

MenuClienteController → Botones de ver productos y hacer pedido

MenuVendedorController → Botones para ver, editar, borrar, añadir producto

HacerPedidoController → Botón para confirmar el pedido

ProductoFormController → Botón de guardar cambios al editar/añadir producto

Por ejemplo :

Botón "Confirmar" (Cliente)

Este botón permite hacer un pedido. Cuando se pulsa, se comprueba que se ha seleccionado un producto y una cantidad válida.

Después se crea el pedido, se guarda en la base de datos, se registra qué producto se ha pedido y se actualiza el stock.

Todo se hace con los métodos de los DAOs: PedidoDAO, PedidoProductoDAO y ProductoDAO.

```
@FXML  Manu, Yesterday • Versión final del proyecto funcionando con sesión

private void onConfirmar() {
    Cliente cliente = Sesion.getClienteActual();

    // Comprobamos que el cliente esté en sesión
    if (cliente == null) {
        mostrarAlerta( titulo: "Error",  mensaje: "No se detectó sesión del cliente.");
        return;
    }

    Producto producto = comboProductos.getValue();
    String cantidadStr = txtCantidad.getText();

    // Validamos que el producto y la cantidad sean válidos
    if (producto == null || cantidadStr.isEmpty()) {
        mostrarAlerta( titulo: "Error",  mensaje: "Selecciona un producto y escribe una cantidad.");
        return;
    }

    try {
        int cantidad = Integer.parseInt(cantidadStr);

        // Validamos que la cantidad sea positiva y no exceda el stock
        if (cantidad <= 0 || cantidad > producto.getStock()) {
            mostrarAlerta( titulo: "Error",  mensaje: "Cantidad inválida o sin stock.");
            return;
        }
    }
}
```

Esto son validaciones lo importante viene ahora,

```
18  public class HacerPedidoController { 1 usage  Manu
52  private void onConfirmar() {
77      }
78
79      // Creamos el pedido
80      Pedido pedido = new Pedido();
81      pedido.setIdCliente(cliente.getIdUsuario());
82      pedido.setFecha(LocalDate.now().toString());
83      int idPedido = PedidoDAO.insert(pedido);
84
85      // Asociamos el producto al pedido
86      PedidoProducto pedidoProducto = new PedidoProducto(idPedido, producto.getIdProducto(), cantidad);
87      PedidoProductoDAO.insert(pedidoProducto);
88
89      // Actualizamos el stock
90      producto.setStock(producto.getStock() - cantidad);
91      ProductoDAO.update(producto);
92
93      mostrarAlerta( titulo: "Pedido confirmado",  mensaje: "Se ha guardado tu pedido.");
94
95      } catch (NumberFormatException e) {
96          mostrarAlerta( titulo: "Error",  mensaje: "Introduce un número válido.");
97      } catch (Exception e) {
98          e.printStackTrace();
99          mostrarAlerta( titulo: "Error interno",  mensaje: "No se pudo procesar el pedido.");
100      }
101  }
```

Y luego en el fxml

```
<HBox spacing="10" alignment="CENTER">
    <Button text="Confirmar" onAction="#onConfirmar"/>
    <Button text="Volver" onAction="#onVolver"/>
</HBox>
```

Los botones se definen en <HBox>

- Confirmar: al pulsarlo ejecuta el método onConfirmar() del controlador.
- Volver: cierra la ventana llamando a onVolver(). El cual se haría igual que confirmar en el controlador

Ambos botones están conectados directamente a su lógica usando onAction="#onMetodo".

8. Fase de Desarrollo

Al principio me encargué de hacer el esquema entidad relación, el cual fue durante las prácticas y tomando una idea de como hacer el proyecto y de que iba a ser; cuando tomé la decisión de hacer un gestor de pedidos de ropa, luego empecé con las clases principales y sus respectivos DAOs, cuando acabé eso empecé con la interfaz los controladores y para acabar y decorar todo les puse un estilo con CSS para embellecer la interfaz.

9. Lista de Comprobación

1. Uso de herencia, abstract y constructores:

Se ha aplicado herencia entre Usuario (superclase), Cliente y Vendedor. Los constructores de las subclases llaman explícitamente al constructor de la clase padre. Aunque no se han utilizado clases abstract por simplicidad, el diseño es fácilmente adaptable para introducirlas y generalizar aún más el comportamiento.

2. Relaciones 1:N y N:M:

El sistema contiene relaciones 1:N (por ejemplo, un cliente tiene varios pedidos) y N:M (un pedido puede contener varios productos y un producto estar en varios pedidos). Esta última se maneja mediante la tabla intermedia pedido_producto.

3. DAO y control de relaciones:

Cada entidad tiene su DAO. Las operaciones N:M están controladas desde PedidoProductoDAO, el cual gestiona los enlaces entre pedidos y productos. Se implementan inserciones, eliminaciones y búsquedas cruzadas.

4. Mostrar, insertar y eliminar:

Las funcionalidades se realizan a través de los DAOs. Mostrar productos se hace con listas dinámicas desde la base de datos. Insertar y eliminar están controladas con validación y actualizan la vista en tiempo real.

5. Actualizar:

La edición de productos se realiza desde el menú del vendedor. Se selecciona el producto, se rellenan los nuevos datos y se actualizan en la base de datos con validación previa.

10. Conclusiones

Lo que más problemas he tenido en resumen ha sido con github, al subir los commits sobretodo, porque he cambiado de archivos mil veces y he cambiado cosas cada rato, también hubo un tiempo que no subí nada que fue el tiempo que estuve de prácticas iba un poco desorganizado la verdad aun así: Me siento muy orgulloso del resultado obtenido en el proyecto por mi parte ya que he aprendido muchas cosas, también me he frustrado mucho cuando algunas cosas del código no me funcionaban y al final logré arreglar y solventar todos mis problemas, la satisfacción final que tuve no tengo palabras para expresarlo, por último creo que se podrían aplicar mil cambios y mejoras pese a ello, me satisface esta versión final.

EXTRA que considero súper importante mencionar

He investigando como hacer muchas cosas en el proyecto, ya sea por el tiempo que estuve de prácticas programando sin ayuda de nadie, o porque me quedaba pillado sin saber que hacer, así que he hecho una lista mencionando algunas cosas que he aprendido a hacer ahora pero que antes no sabía :

1. Validación de email con expresiones regulares (Regex)

Implementé una validación automática para evitar que se registren correos electrónicos con formato incorrecto, utilizando una expresión regular sencilla en el controlador de registro.

Esto me permitió mejorar la integridad de los datos y asegurar que los emails introducidos por los usuarios sean válidos.

Código usado:

```
if (!email.matches("^[\\w.-]+@[\\w.-]+\\. [a-zA-Z]{2,}$")) {  
    mostrarAlerta("Email inválido", "El formato del email no es  
válido.");  
    return;  
}
```

2. Uso de **Stream** y **filter** para manipular colecciones (Java 8)

Para cumplir con los estándares modernos de Java, utilicé un método con programación funcional (**stream().filter()**) para filtrar productos sin stock dentro del DAO de productos.

Esta técnica me permite manejar listas de forma más limpia y declarativa.

Ejemplo:

```
public static List<Producto> getSinStock() {  
    return getAll().stream()  
        .filter(p -> p.getStock() == 0)  
        .toList();  
}
```

3. Clase **Sesion** para mantener el usuario logueado entre ventanas

Como el paso de objetos entre vistas de JavaFX no es directo, investigué y apliqué una solución mediante una clase estática llamada **Sesion**.

Esta clase me permite mantener el cliente logueado en memoria y acceder a él desde cualquier controlador, sin necesidad de pasar manualmente los datos.

4. Validación dinámica para que solo se escriban números en campos sensibles

Para evitar errores en campos numéricos como el de cantidad de productos, añadí un `listener` al `TextField` que bloquea automáticamente cualquier carácter no numérico. Con esto se evitan errores comunes por parte del usuario.

```
txtCantidad.textProperty().addListener((obs, oldVal, newVal) -> {  
    if (!newVal.matches("\\d*")) {  
        txtCantidad.setText(newVal.replaceAll("[^\\d]", ""));  
    }  
});
```

5. Uso de métodos `initialize()` en los controladores FXML

En lugar de cargar datos directamente desde los constructores (algo no recomendado en JavaFX), utilicé el método `initialize()` con la anotación `@FXML` para preparar las interfaces justo cuando se cargan.

Esto me permitió:

- Rellenar combobox con productos.
- Asignar listeners o validadores.
- Preparar tablas con `TableColumn.setCellValueFactory()`.

@FXML

```
public void initialize() {  
    comboProductos.getItems().setAll(ProductoDAO.getAll());  
}
```

6. Carga dinámica de vistas usando FXMLLoader con paso de datos

Para mantener el flujo de ventanas entre controladores y pasar datos entre ellos (como el cliente logueado), utilicé el objeto **FXMLLoader**, accediendo a su **getController()** antes de mostrar la vista.

Esto me permitió acceder al controlador Java del nuevo FXML y configurar sus atributos antes de mostrarlo, algo que no se suele explicar en ejercicios básicos.

```
FXMLLoader loader = new  
FXMLLoader(getClass().getResource("/controller/hacerPedido.fxml")  
);
```

```
Parent root = loader.load();
```

```
HacerPedidoController controller = loader.getController();
```

```
controller.setClienteLogueado(cliente); // Paso de datos entre  
controladores
```

✓ 7. Separación lógica de cada pantalla en su propio controlador

En lugar de poner toda la lógica en una clase central, estructuré el proyecto usando un controlador por cada pantalla (FXML), lo cual mejora la mantenibilidad y el orden del código.

Cada controlador está conectado a su vista y tiene sus propias acciones con **@FXML**.

✓ 8. Creación y uso de alertas informativas reutilizables

Centralicé la lógica de mostrar mensajes al usuario usando el componente **Alert** de JavaFX, lo que mejora la experiencia de usuario y evita errores silenciosos.

Este patrón no suele enseñarse directamente pero es fundamental en apps reales.

```
private void mostrarAlerta(String titulo, String mensaje) {  
    Alert alerta = new Alert(Alert.AlertType.INFORMATION);  
    alerta.setTitle(titulo);  
    alerta.setHeaderText(null);  
    alerta.setContentText(mensaje);  
    alerta.showAndWait();  
}
```
