**UNIVERSIDAD TECNOLÓGICA DEL PERÚ**

**FACULTAD DE INGENIERÍA DE SISTEMAS Y ELECTRÓNICA**


**CURSO INTEGRADOR I: SISTEMAS - SOFTWARE**

**TEMA:**

SISTEMA DE CONTROL DE STOCK

Avance de Proyecto Final 3

**ELABORADO POR:**


- SOTO LUQUE, ELVIS 100%
- RIVERA GUTIERREZ, EMMANUEL 100%
- SARDON MARTINEZ, RALFPH 100%
- QUISPE RODRIGUEZ, YINYER 100%


**DOCENTE:**

MAG. JESAMIN ZEVALLOS

**Repositorio de GitHub:** https://github.com/Emmanuel-Rivera-G/Proyecto-Final-Curso-Integrador-I-Sistemas-Software

**ÍNDICE**

## Introducción

En el entorno actual, la gestión de inventarios juega un papel crítico en la operación eficiente de las empresas, especialmente en las pequeñas y medianas empresas (MyPEs y PyMEs), donde una mala administración de los recursos puede resultar en pérdidas significativas. La implementación de un sistema de control de stock se presenta como una solución estratégica para abordar las dificultades inherentes a los métodos tradicionales de control manual. Este documento tiene como objetivo definir los requisitos funcionales y no funcionales de un sistema de control de stock para desktop, el cual permitirá automatizar y agilizar los procesos de entrada y salida de productos en almacenes de diversos sectores, tales como materias primas, productos terminados y existencias generales. A través de este sistema, se busca optimizar el desempeño de los responsables del almacén y facilitar el análisis logístico necesario para una proyección adecuada de los flujos de inventario.

## 1. Problema

Uno de los principales objetivos de toda empresa, es el poder generar la mayor cantidad de ventas posibles y que se tenga el mejor balance en stock disponible para no faltar a ninguna venta. Pero generalmente las MYPE's y algunas PYME's suelen presentar una ineficiencia en el control de stock, sobre todo en su etapa de inicios o incremento de actividades.

En la actualidad muchas empresas que no cuentan con la automatización digital de sus operaciones suelen tener dificultades en la gestión y administración de sus inventarios. Esto suele ser un desafío común en MYPE's y algunas PYME's, generando una serie de consecuencias negativas que impactan en la rentabilidad y competitividad de toda empresa que no se adapte a la tecnología.

Entre los principales problemas que se suelen generar ante la falta de un sistema de control de stock suelen ser:

- Gestión ineficiente de los inventarios: Los sistemas manuales suelen provocar desorganización en los registros de existencias, lo que dificulta el control adecuado de los productos en almacén y puede generar sobreabastecimiento o desabastecimiento.
- Errores humanos en el control físico: La falta de automatización en los procesos de entrada y salida de mercancías incrementa el riesgo de errores humanos, lo que puede traducirse en incongruencias entre el inventario físico y el inventario registrado.
- Falta de información en tiempo real para la toma de decisiones: La carencia de un sistema que ofrezca datos actualizados dificulta la capacidad de la empresa para realizar análisis efectivos y oportunos, afectando la toma de decisiones estratégicas.
- Ineficiencia en la generación de reportes: Al obtener realizar reportes manuales suele haber una mayor dificultad y demora en realizar los cálculos y verificar el flujo histórico de los productos, siendo este un proceso lento y tedioso.

Esta problemática puede llegar a afectar el nivel de ingresos que percibe la empresa. De aquí es donde nace la necesidad de poder contar con un sistema que facilite a los operarios almaceneros y analistas logísticos poder visualizar los productos disponibles en tiempo real, entradas y salidas de productos, rotación de inventarios y demás propio de un sistema de control de stock.

## 2. Solución

La solución propuesta es un sistema de control de stock robusto y escalable diseñado para computadoras desktop, capaz de adaptarse a las necesidades de MyPEs y PyMEs que requieren un manejo eficiente de sus inventarios. Este software permitirá a las empresas registrar, organizar y monitorear el flujo de productos o elementos de manera automatizada y precisa, reduciendo significativamente los errores humanos que suelen ocurrir con los sistemas manuales.

Este sistema está diseñado para optimizar las operaciones de almacenamiento al proporcionar una gestión eficiente de las entradas y salidas de productos, facilitando la actualización en tiempo real del inventario disponible. Entre sus principales características se encuentran:

- Registro automatizado de inventario: Permite que cada producto sea ingresado al sistema mediante códigos únicos o escaneos, asegurando que las entradas y salidas sean reflejadas de manera precisa y sin demoras.
- Organización flexible del stock: El sistema ofrece una interfaz intuitiva para organizar los productos en diferentes categorías, almacenes y ubicaciones específicas dentro del depósito, optimizando el proceso de búsqueda y acceso a la información.
- Monitoreo en tiempo real: La plataforma proporciona datos actualizados sobre las existencias, lo que permite tomar decisiones informadas y evitar tanto el sobreabastecimiento como el desabastecimiento, asegurando un flujo óptimo de los productos.

Al integrar este servicio de software, las empresas no solo podrán automatizar sus procesos de control de stock, sino también mejorar la eficiencia operativa, facilitando una mejor toma de decisiones a nivel logístico y administrativo. Esto es particularmente relevante en las empresas que manejan múltiples productos o materiales, donde el control preciso y actualizado es esencial para garantizar la rentabilidad del negocio.

## 3. Arquitectura MVC

### 3.1. Diagramas de secuencia

**Figura 1**

Diagrama de secuencia del Caso de Uso 01: Registrar Usuario

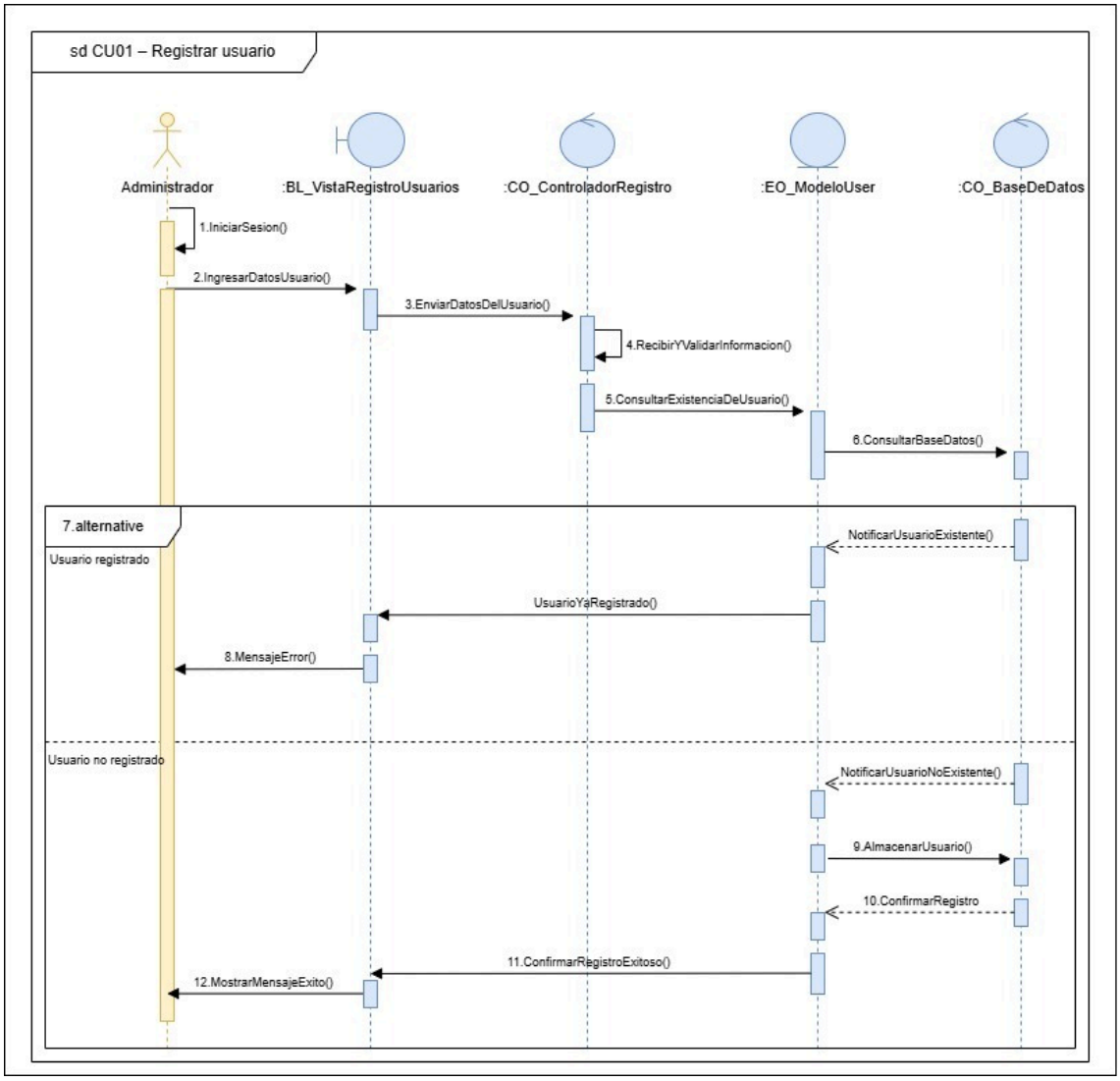


**Figura 2**

Diagrama de secuencia del Caso de Uso 02: Autenticar Usuario

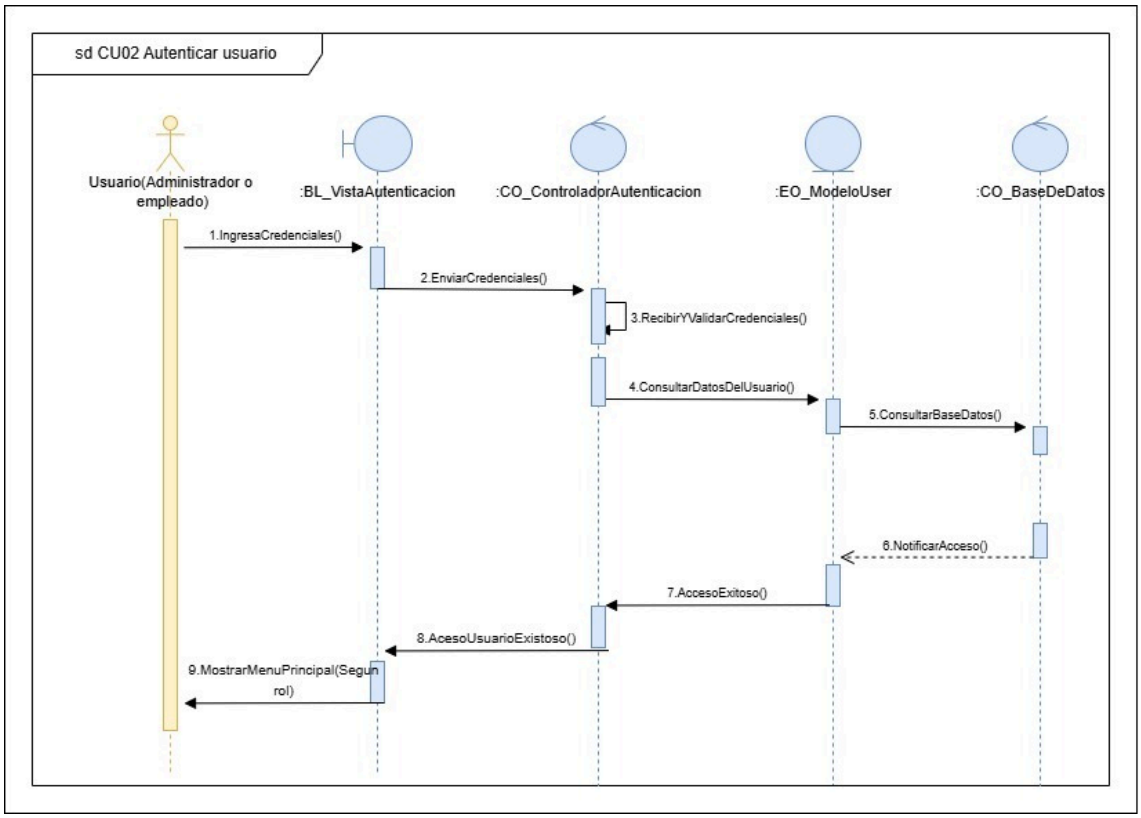
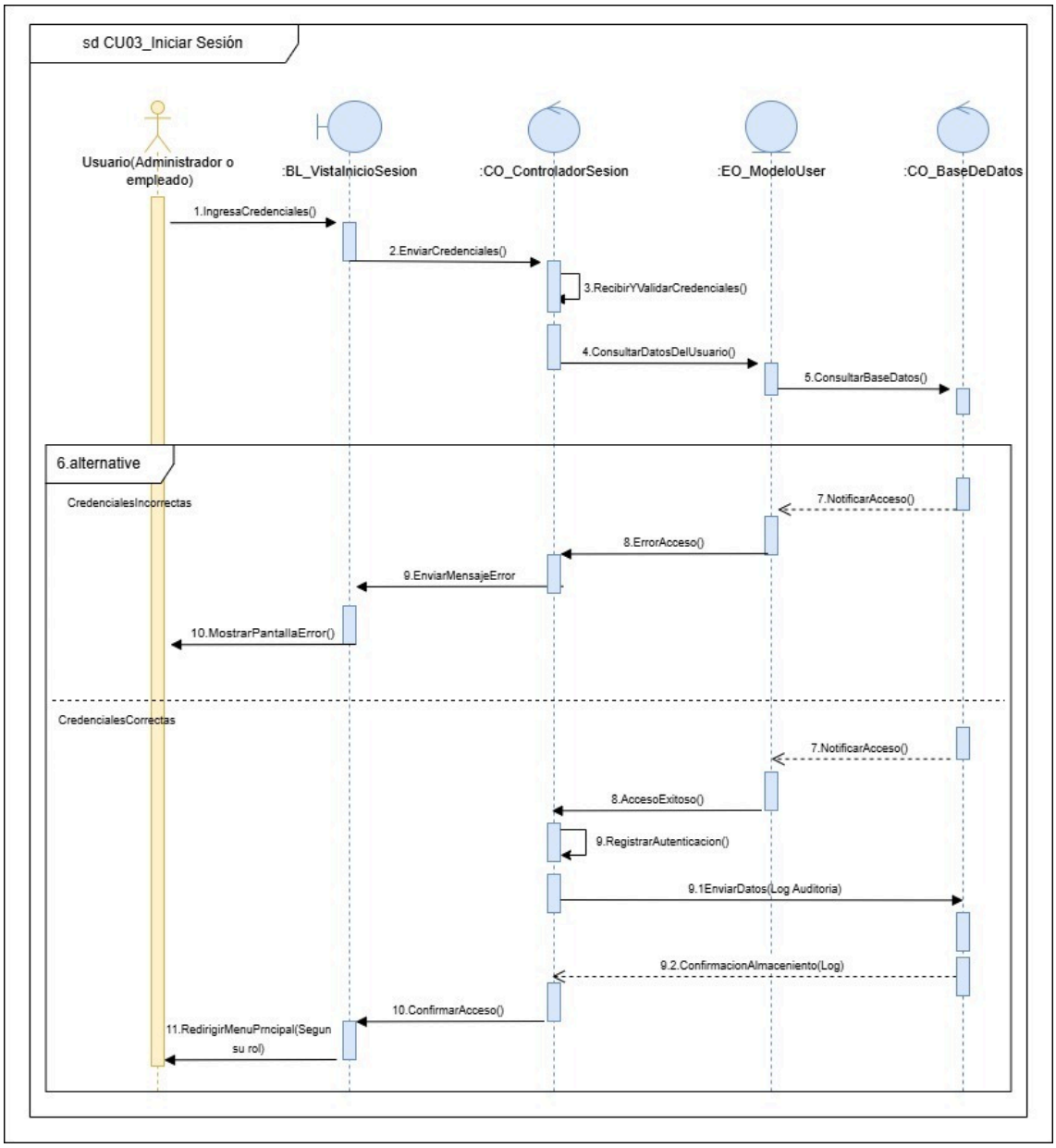


**Figura 3**

Diagrama de secuencia del Caso de Uso 03: Iniciar Sesión




se le otorga acceso a las funcionalidades del sistema.

**Figura 4**

Diagrama de secuencia del Caso de Uso 04: Registrar Productos

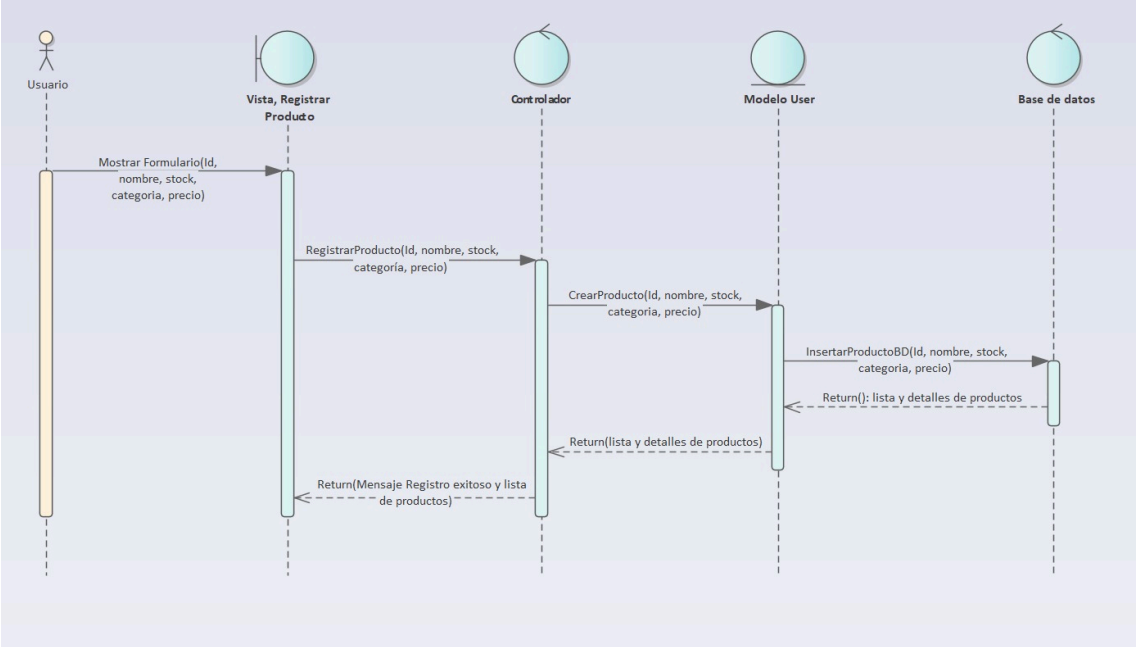
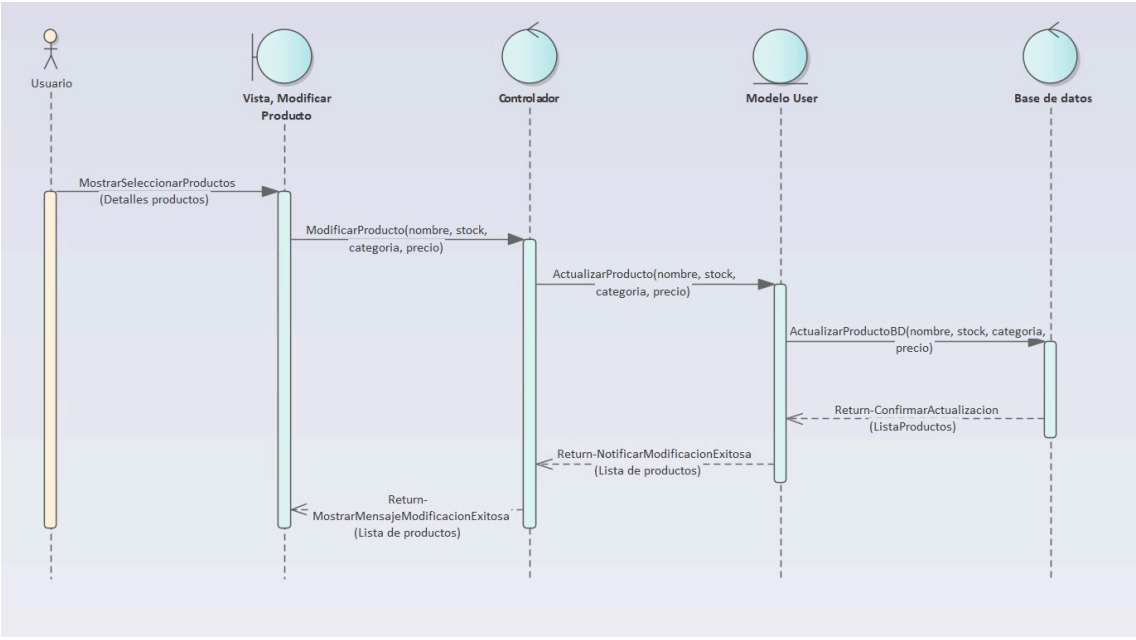



**Figura 5**

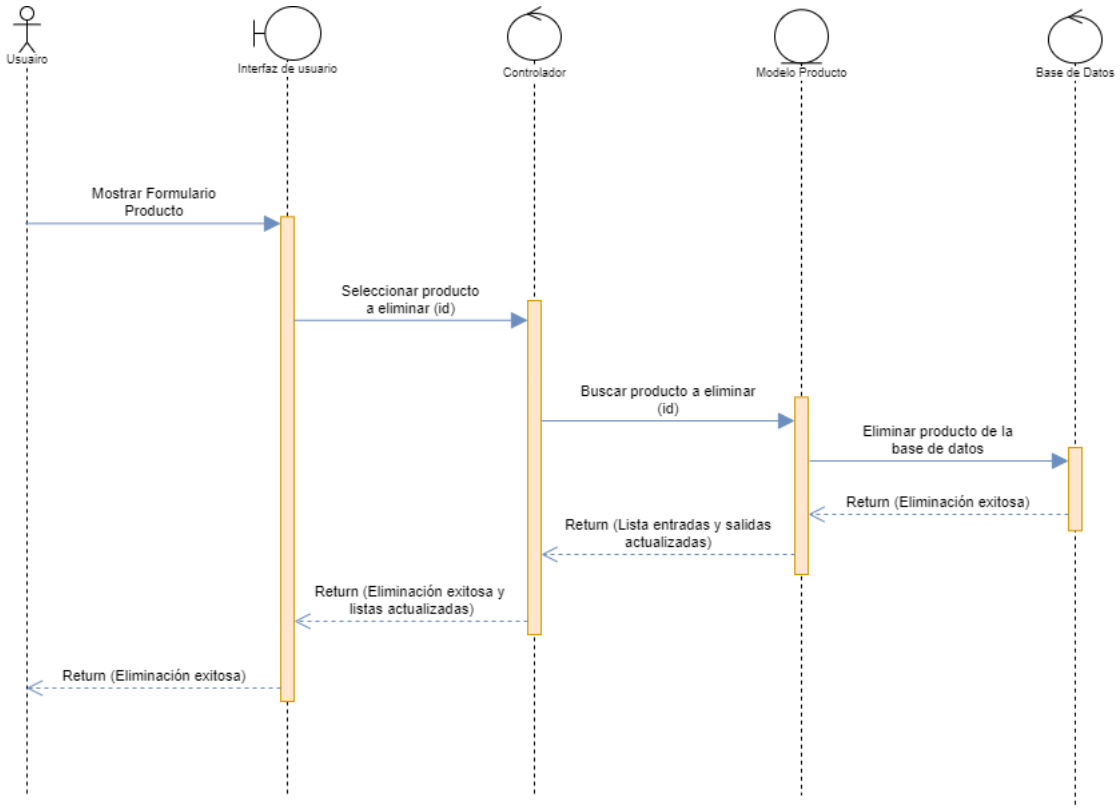
Diagrama de secuencia del Caso de Uso 05: Modificar Productos



selecciona el producto, actualiza los datos a través de la interfaz, y el controlador se encarga de realizar los cambios en el sistema.

**Figura 6**

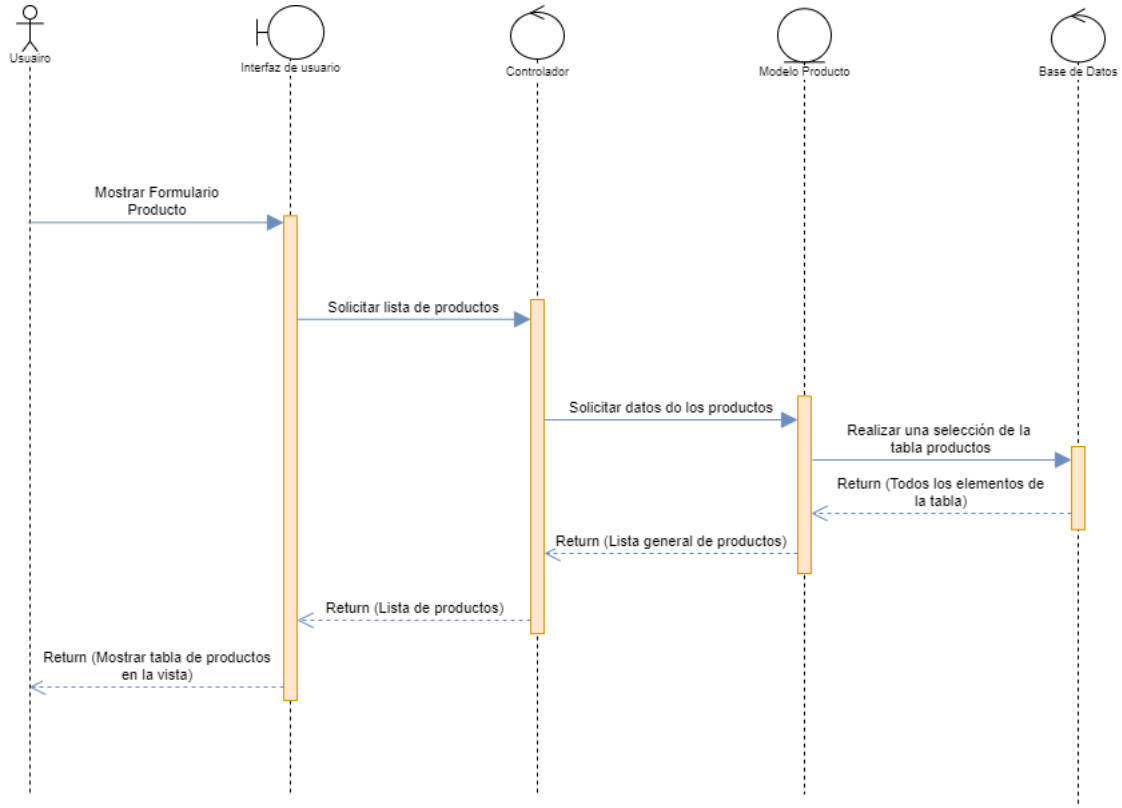
Diagrama de secuencia del Caso de Uso 07: Eliminar Productos




Nota. El diagrama de secuencia describe el flujo de acciones cuando un usuario desea eliminar un producto del sistema. Se observa cómo el usuario selecciona el producto a eliminar, confirma la acción, y el controlador se encarga de remover ese producto de la base de datos.

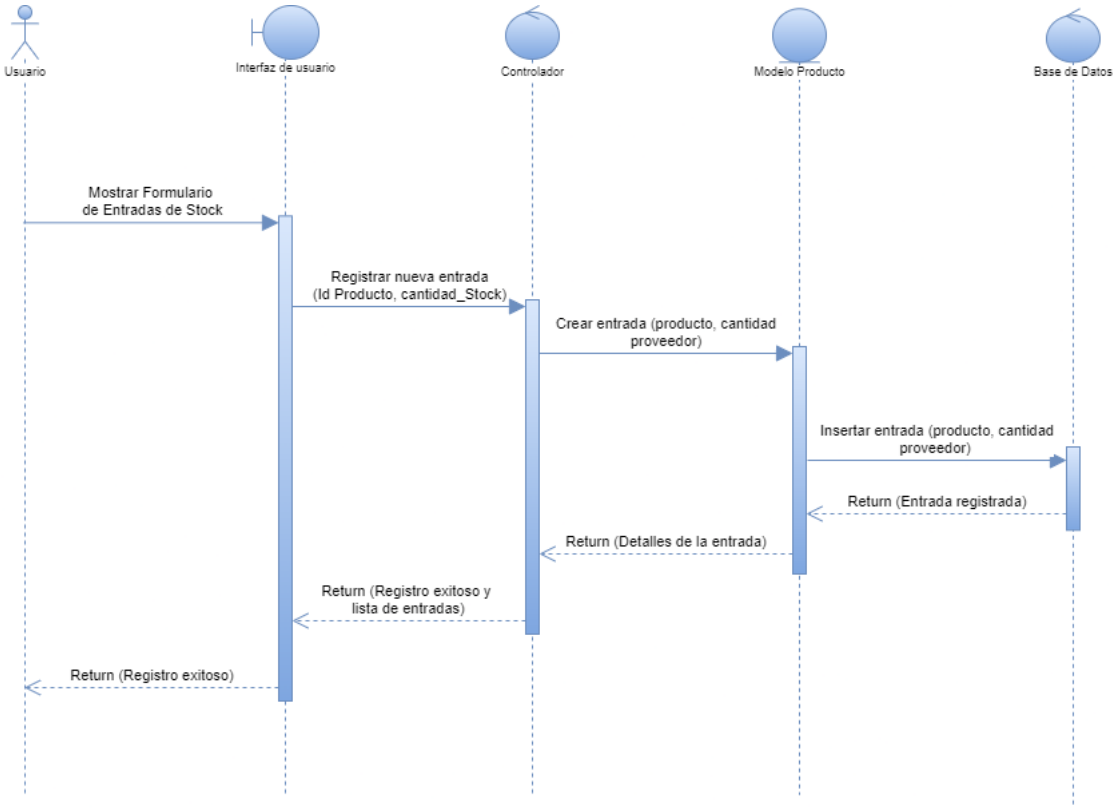
**Figura 7**

Diagrama de secuencia del Caso de Uso 07: Visualizar Productos




Nota. Este diagrama de secuencia representa el proceso de visualización de los productos disponibles en el sistema. Muestra cómo el usuario interactúa con la interfaz, solicita la lista de productos, y el controlador se encarga de recuperar y enviar esa información para su presentación.

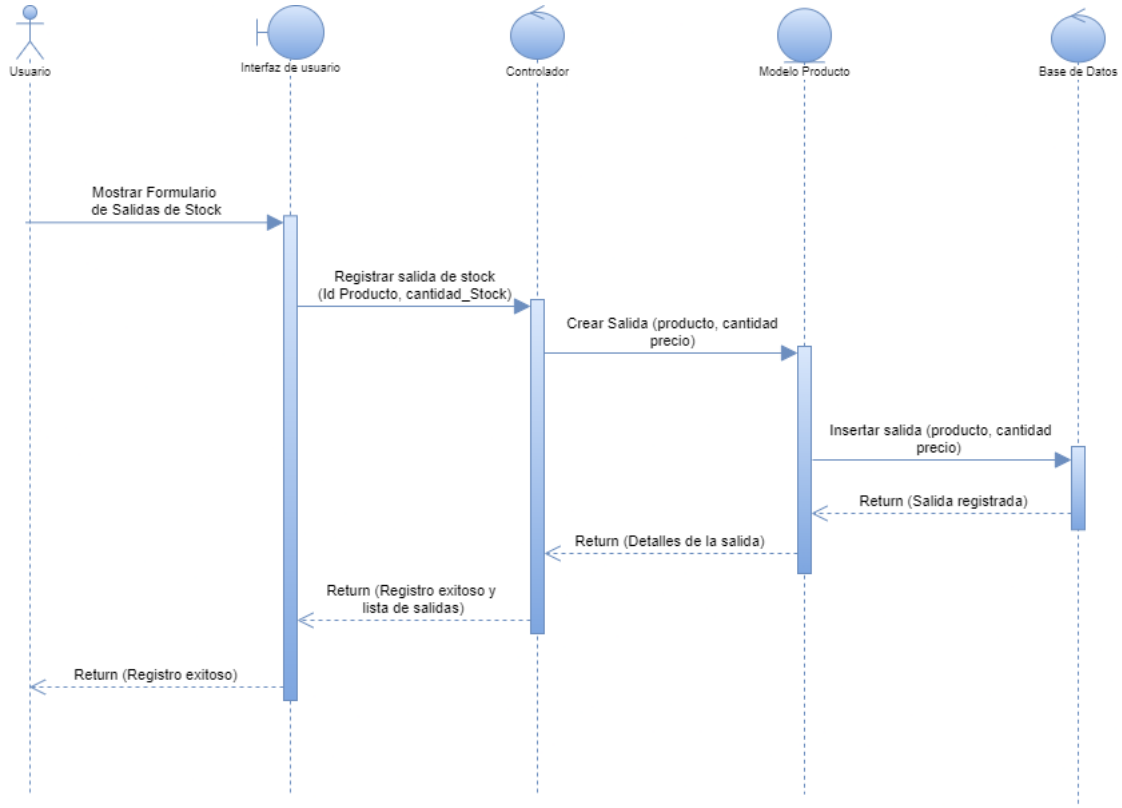
**Figura 8**




Nota. Este diagrama de secuencia ilustra el proceso de registro de una entrada de stock en el sistema. Muestra cómo el usuario ingresa los detalles de la nueva entrada, y el controlador actualiza los niveles de inventario correspondientes.

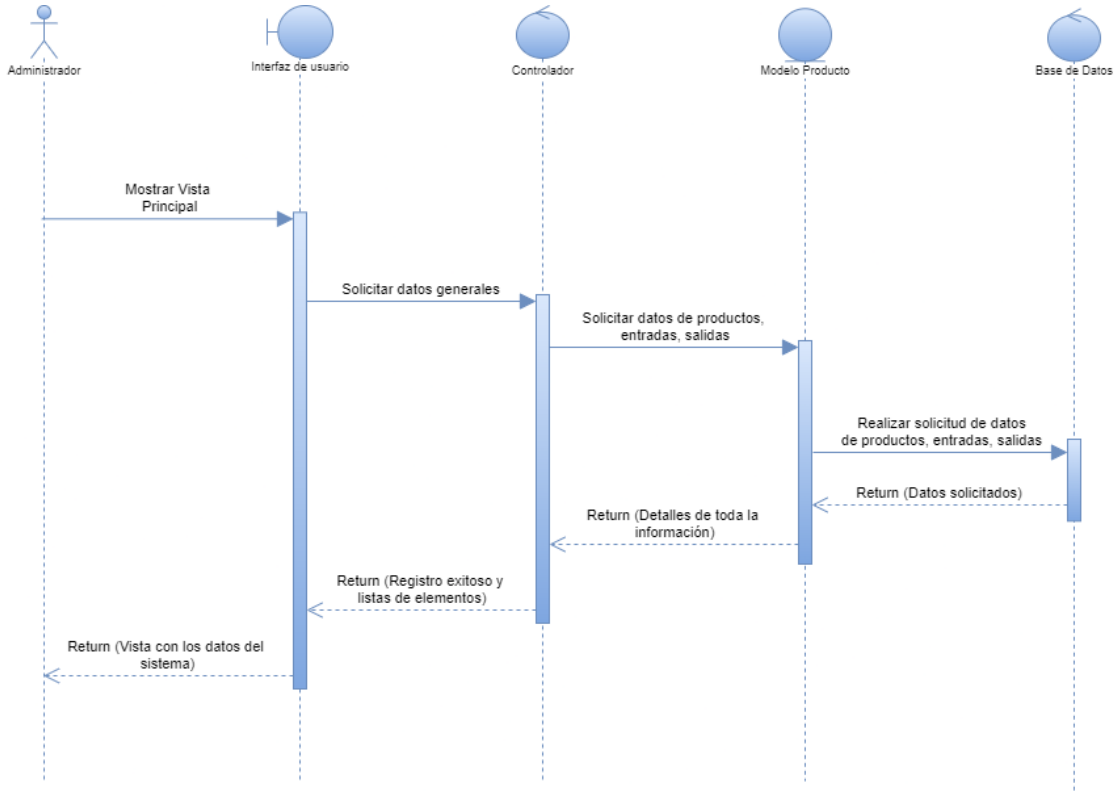
**Figura 9**

Diagrama de secuencia del Caso de Uso 09: Registrar Salida de Stock




Nota. El diagrama de secuencia describe el flujo de acciones cuando un usuario desea registrar una salida de stock en el sistema. Se aprecia cómo el usuario proporciona los detalles de la salida, y el controlador actualiza los niveles de inventario en consecuencia.

**Figura 10**




Nota. Este diagrama de secuencia ilustra el proceso de visualización de las estadísticas totales en el sistema. Muestra cómo el usuario interactúa con la interfaz para solicitar la información estadística, y el controlador se encarga de recuperar y enviar los datos agregados para su presentación.

## 3.2. TDD

### 3.2.1. Desarrollo Guiado por Pruebas (TDD)

Elaborado por Emmanuel Rivera Gutierrez

En este proyecto se ha implementado TDD para garantizar que la conexión a la base de datos funcione correctamente. Inicialmente, se escribió un test unitario utilizando JUnit para validar que la clase Conexion establece correctamente una conexión activa con la base de datos configurada. Este test verifica que no se lancen excepciones y que la conexión no sea nula. Una vez que el test falló, se procedió a implementar el código en la clase Conexion para satisfacer las condiciones del test, asegurando que el sistema pueda conectar con la base de datos de manera fiable antes de seguir avanzando con otras funcionalidades. Esto permite una mayor confianza en el correcto funcionamiento de la infraestructura del proyecto desde las primeras fases de desarrollo.

**Figura 11**

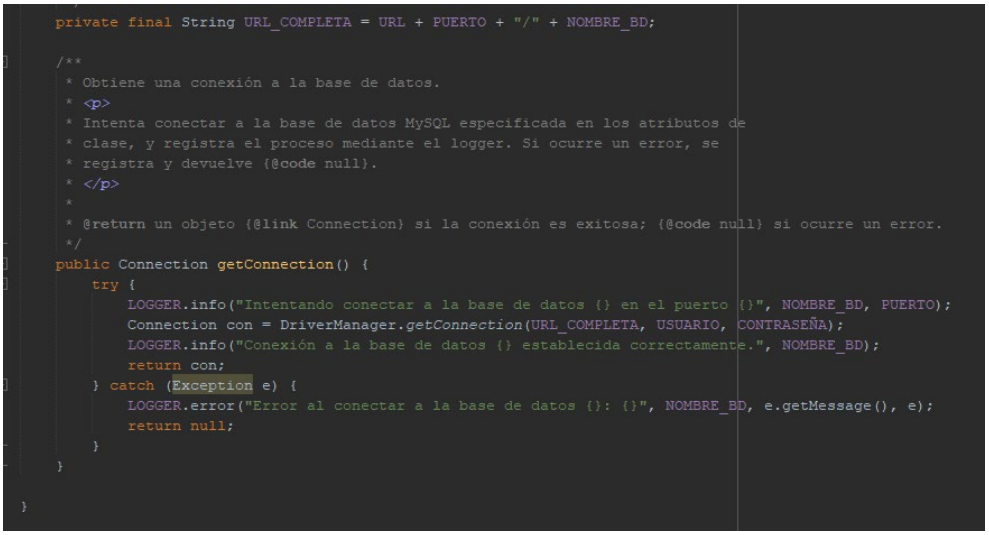

Nota. La figura ilustra el proceso de verificación de la conexión a la base de datos mediante TDD. Un test unitario inicial verifica que la clase Conexion establece una conexión activa sin errores. Este proceso asegura la fiabilidad del sistema desde el inicio.

**Figura 12**

```
public class ConexionTest {
    private ConexionBD conexion;
    private final Logger LOGGER = UtilLoggerManager.getLogger(ConexionTest.class);

    @Before
    public void setUp() {
        conexion = new ConexionBD();
    }

    @Test
    public void testGetConnection() {
        try {
            Connection con = conexion.conectarBaseDatos();
            Assert.assertNotNull("La conexión no debería ser null", con);
            LOGGER.info("Test de conexión exitoso.");
        } catch (Exception e) {
            LOGGER.error("Test fallido: Error al conectar con la base de datos.", e);
            Assert.fail("La conexión a la base de datos falló: " + e.getMessage());
        }
    }
}
```

La imagen muestra la implementación de la clase Conexion, la cual fue desarrollada después de que el test fallara. En este código se establece la lógica necesaria para conectar a la base de datos, asegurando que el test previamente escrito pueda ser superado. Esta implementación sigue el ciclo de TDD, donde primero se escriben las pruebas y luego el código para satisfacerlas.

**Figura 13**

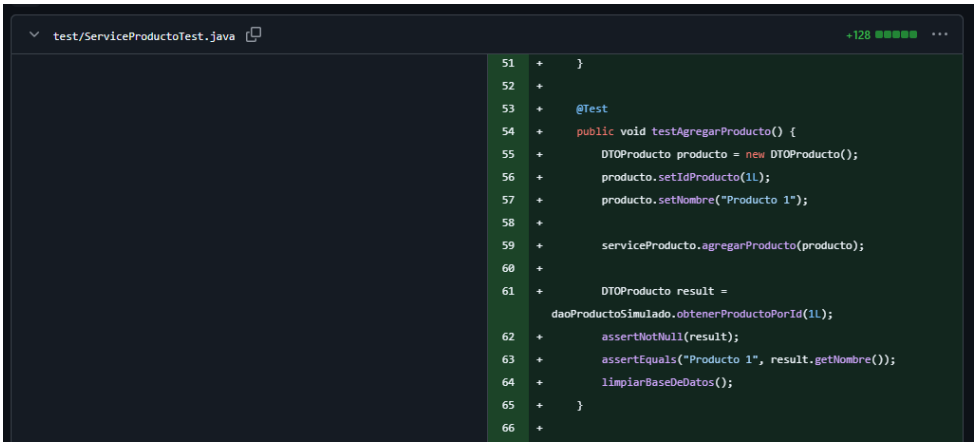


```
    private final String URL_COMPLETA = URL + PUERTO + "/" + NOMBRE_BD;

    /**
     * Obtiene una conexión a la base de datos.
     * <p>
     * Intenta conectar a la base de datos MySQL especificada en los atributos de
     * clase, y registra el proceso mediante el logger. Si ocurre un error, se
     * registra y devuelve {@code null}.
     * </p>
     *
     * @return un objeto {@link Connection} si la conexión es exitosa; {@code null} si ocurre un error.
     */
    public Connection getConnection() {
        try {
            LOGGER.info("Intentando conectar a la base de datos {} en el puerto {}", NOMBRE_BD, PUERTO);
            Connection con = DriverManager.getConnection(URL_COMPLETA, USUARIO, CONTRASEÑA);
            LOGGER.info("Conexión a la base de datos {} establecida correctamente.", NOMBRE_BD);
            return con;
        } catch (Exception e) {
            LOGGER.error("Error al conectar a la base de datos {}: {}", NOMBRE_BD, e.getMessage(), e);
            return null;
        }
    }

}
```

3.2.2. Test para ServiceProducto.java

**Figura 14**

Validación de ServiceProducto.java en commit de GitHub



test/ServiceProductoTest.java +128

```
51 +    }
52 +
53 +    @Test
54 +    public void testAgregarProducto() {
55 +        DTOProducto producto = new DTOProducto();
56 +        producto.setIdProducto(1L);
57 +        producto.setNombre("Producto 1");
58 +
59 +        serviceProducto.agregarProducto(producto);
60 +
61 +        DTOProducto result =
    daoProductoSimulado.obtenerProductoPorId(1L);
62 +        assertNotNull(result);
63 +        assertEquals("Producto 1", result.getNombre());
64 +        limpiarBaseDeDatos();
65 +    }
66 +
```

Nota: Captura de pantalla que muestra el historial de commits en GitHub relacionados con la implementación de pruebas para la clase ServiceProducto, evidenciando el desarrollo iterativo del código.

**Figura 15**

Prueba del método testAgregarProducto() en ServiceProductoTest.java

```java
@Test
public void testAgregarProducto() {
    DTOProducto producto = new DTOProducto();
    producto.setIdProducto(1L);
    producto.setNombre("Producto 1");

    serviceProducto.agregarProducto(producto);

    DTOProducto result = daoProductoSimulado.obtenerProductoPorId(1L);
    assertNotNull(result);
    assertEquals("Producto 1", result.getNombre());
    limpiarBaseDeDatos();
}
```

Nota. Esta prueba, ubicada en /test/ServiceProductoTest.java, evalúa el método testAgregarProducto() para confirmar la correcta adición de productos en la base de datos. La prueba permite validar que el método se desempeñe como se espera, facilitando la identificación de errores en la funcionalidad.

**Figura 16**

```java
@Test
public void testActualizarProducto() {
    DTOProducto producto = new DTOProducto();
    producto.setIdProducto(1L);
    producto.setNombre("Producto 1");

    serviceProducto.agregarProducto(producto);

    producto.setNombre("Producto Actualizado");
    serviceProducto.actualizarProducto(producto);

    DTOProducto result = daoProductoSimulado.obtenerProductoPorId(1L);
    assertEquals("Producto Actualizado", result.getNombre());
    limpiarBaseDeDatos();
}
```

Nota. En esta imagen se muestra la prueba testActualizarProducto(), parte del archivo de pruebas ServiceProductoTest.java. Evalúa la capacidad del sistema para actualizar correctamente los datos de un producto en la base de datos. Con ello, se asegura que los cambios en los productos se registren adecuadamente.

**Figura 17**

Prueba del método testEliminarProducto() en ServiceProductoTest.java

```java
@Test
public void testEliminarProducto() {
    DTOProducto producto = new DTOProducto();
    producto.setIdProducto(1L);
    producto.setNombre("Producto 1");

    serviceProducto.agregarProducto(producto);
    serviceProducto.eliminarProducto(1L);

    DTOProducto result = daoProductoSimulado.obtenerProductoPorId(1L);
    assertNull(result);
    limpiarBaseDeDatos();
}
```

Nota. La prueba testEliminarProducto() comprueba que el sistema pueda eliminar un producto de la base de datos de forma eficiente y sin errores. Esto es fundamental para garantizar que las operaciones de borrado funcionen correctamente en todas las condiciones necesarias del sistema.

### 3.2.3. Test para ServiceUsuario.java

Se comprueba la conexión a la base de datos y el funcionamiento de las funciones.

**Figura 18**

Validación de ServiceUsuario.java en commit de GitHub

test/ServiceUsuarioTest.java +165

```
57 +
58 +    @Test
59 +    public void testRegistrarUsuario() throws SQLException {
60 +        DTOUsuario usuario = new DTOUsuario();
61 +        usuario.setDocumento("123456");
62 +        usuario.setNombre("Usuario 1");
63 +        usuario.setApellido("Apellido 1"); // No debe ser nulo
64 +        usuario.setCorreo("usuario1@example.com");
65 +        usuario.setUsername("user1");
66 +        usuario.setPassword("password123");
67 +        usuario.setIdTipoUsuario(1);
68 +
69 +        boolean registrado = serviceUsuario.registrarUsuario(usuario);
70 +        assertTrue(registrado);
71 +
72 +        List<DTOUsuario> result = serviceUsuario.obtenerUsuarios();
73 +        assertEquals(1, result.size());
74 +        limpiarBaseDeDatos();
75 +    }
```

Nota: Captura de pantalla que muestra el historial de commits en GitHub relacionados con la implementación de pruebas para la clase ServiceUsuario, evidenciando el desarrollo iterativo del código.

**Figura 19**

Prueba del método testRegistrarUsuario() en ServiceUsuarioTest.java

```
@Test
public void testRegistrarUsuario() throws SQLException {
    DTOUsuario usuario = new DTOUsuario();
    usuario.setDocumento("123456");
    usuario.setNombre("Usuario 1");
    usuario.setApellido("Apellido 1");
    usuario.setCorreo("usuario1@example.com");
    usuario.setUsername("user1");
    usuario.setPassword("password123");
    usuario.setIdTipoUsuario(1);

    boolean registrado = serviceUsuario.registrarUsuario(usuario);
    assertTrue(registrado);

    List<DTOUsuario> result = serviceUsuario.obtenerUsuarios();
    assertEquals(1, result.size());
    limpiarBaseDeDatos();
}
```

Nota. La figura muestra el test testRegistrarUsuario() en ServiceUsuarioTest.java, que evalúa la creación de nuevos usuarios en la base de datos. Este test asegura que los datos ingresados para nuevos usuarios se validen y almacenen correctamente sin errores de integridad.

**Figura 20**

Prueba de actualización de usuario en ServiceUsuarioTest.java

```
@Test
public void testEditarUsuario() throws SQLException {
    DTOUsuario usuario = new DTOUsuario();
    usuario.setDocumento("123456");
    usuario.setNombre("Usuario 1");
    usuario.setApellido("Apellido 1");
    usuario.setCorreo("usuario1@example.com");
    usuario.setUsername("user1");
    usuario.setPassword("password123");
    usuario.setIdTipoUsuario(1);

    serviceUsuario.registrarUsuario(usuario);

    usuario.setNombre("Usuario Editado");
    boolean editado = serviceUsuario.editarUsuario(usuario);
    assertTrue(editado);

    DTOUsuario result = serviceUsuario.buscarUsuarioPorCriterios("Usuario Editado", null, null, null).get(0);
    assertEquals("Usuario Editado", result.getNombre());
    limpiarBaseDeDatos();
}
```

Nota. Esta imagen ilustra la prueba testEditarUsuario(), que verifica la capacidad de modificar datos de usuario existentes en la base de datos. Asegura que la información de los usuarios se actualice correctamente sin errores y que se mantenga la consistencia de los datos.

**Figura 21**

Validación de eliminación de usuario en ServiceUsuarioTest.java

```java
@Test
public void testEliminarUsuarioPorDocumento() throws SQLException {
    DTOUsuario usuario = new DTOUsuario();
    usuario.setDocumento("123456");
    usuario.setNombre("Usuario 1");
    usuario.setApellido("Apellido 1");
    usuario.setCorreo("usuario1@example.com");
    usuario.setUsername("user1");
    usuario.setPassword("password123");
    usuario.setIdTipoUsuario(1);

    serviceUsuario.registrarUsuario(usuario);

    boolean eliminado = serviceUsuario.eliminarUsuarioPorDocumento("123456");
    assertTrue(eliminado);

    List<DTOUsuario> result = serviceUsuario.obtenerUsuarios();
    assertEquals(0, result.size());
    limpiarBaseDeDatos();
}
```

Nota. En esta figura se presenta la prueba testEliminarUsuarioPorDocumento() en ServiceUsuarioTest.java. La prueba asegura que el sistema pueda eliminar registros de usuarios en la base de datos, confirmando que esta acción no afecte la integridad de otros datos relacionados.

### 3.3. DAO

**Patrón Objeto de Acceso a Datos (DAO)**

En el marco del desarrollo del sistema, se ha implementado el patrón DAO mediante la creación de la clase ProductoDAO. Esta clase se encarga de la interacción con la base de datos, utilizando ConexionBD para establecer la conexión a través del método conectarBaseDatos(). La implementación de ProductoDAO incluye métodos clave como listar() y agregar(Producto producto), que permiten recuperar y añadir productos a la base de datos de manera eficiente. A través de este enfoque, se asegura una separación clara entre la lógica de negocio y el acceso a datos, promoviendo un mayor modularidad y facilitando el mantenimiento del código a largo plazo.

- **Elaboración por Elvis Soto Luque**

**Figura 22**

```
public class ProductoDAO {
    ConexionBD conexion = new ConexionBD();//llamanos a la clase conexionBD instanciamos
    Connection con;
    PreparedStatement ps;
    ResultSet rs;

    public List listar(){
        String sql = "SELECT * FROM productos";
        List<Producto>lista = new ArrayList<>();
        try {
            con = conexion.conectarBaseDatos();
            ps = con.prepareStatement(sql);
            rs = ps.executeQuery();// devuelve lista de la base de datos productos
            // la lista se guardara en el Result Set "rs"
            //Ahora del RS debemos pasarlo al arraylist mediante un while
            while (rs.next()){//iterara a toda la lista de la base de datos productos
                //hara uso del constructor vacioO
                Producto producto = new Producto();// obj Producto llamado producto// aqui traigo DTO PRODUCTO
                producto.setId(rs.getInt(1));// referencia por la posicion de la columna tambien puede ir el nombre de la columna BD
                producto.setNombre(rs.getString(2));
                producto.setIdcategoria(rs.getInt(3));
                producto.setUndmedida(rs.getString(4));
                producto.setStock(rs.getInt(5));

                lista.add(producto);// aqui se carga todos los productos ARRAYLIST

            }
```

```
    //Metodo agregar
    public void agregar (Producto producto){
        String sql = "INSERT INTO productos(nombre,idCategoria,undMedida,Stock) VALUES (?,?,?,?)";
        try {
            con = conexion.conectarBaseDatos();
            ps = con.prepareStatement(sql);
            ps.setString(1,producto.getNombre());
            ps.setInt(2,producto.getIdcategoria());
            ps.setString(3,producto.getUndmedida());
            ps.setInt(4,producto.getStock());

            ps.executeUpdate();// EJECUTA LA CONSULTA O INSERCION SQL UPDATE

        } catch (SQLException e) {
            System.out.println("Error en proceso de agregacion productoDAO: "+e);
        }
    }//TERMINA EL METODO AGREGAR
```

- **Elaboración por Yinyer Quispe Rodriguez**

**Figura 24**

Commits de GitHub , relacionados con la conexión y la implementación de DAO -


**Figura 25**

Commits de GitHub , relacionados con la conexión.java

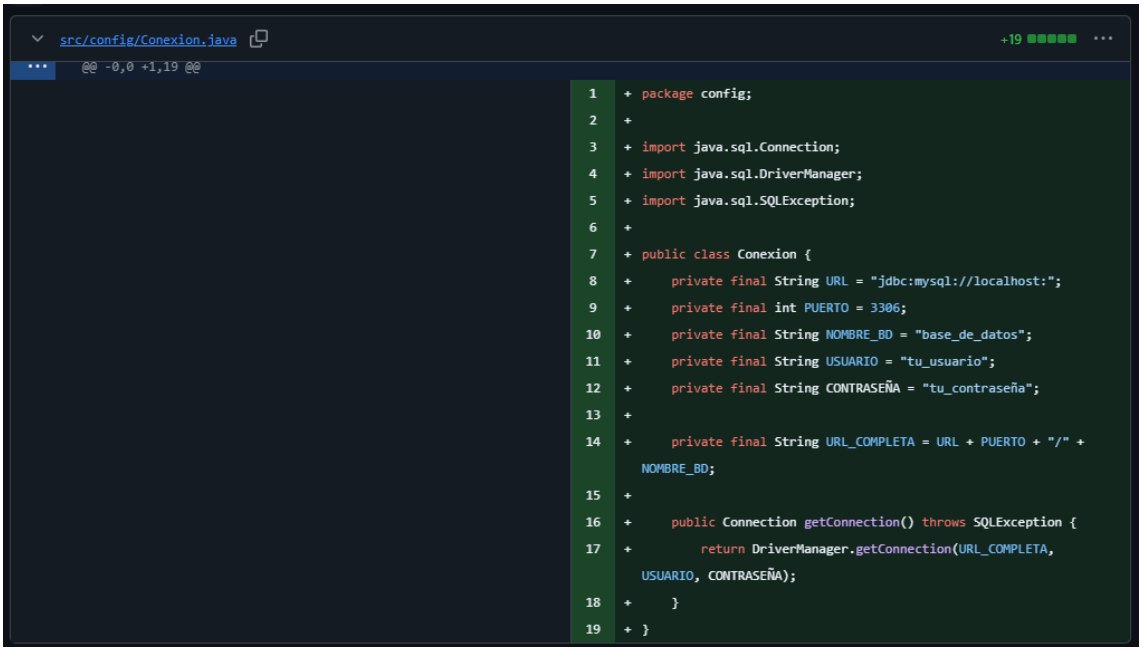


**Figura 26**

Commits de GitHub , relacionados con la implementación de DAOProductoImpl.java

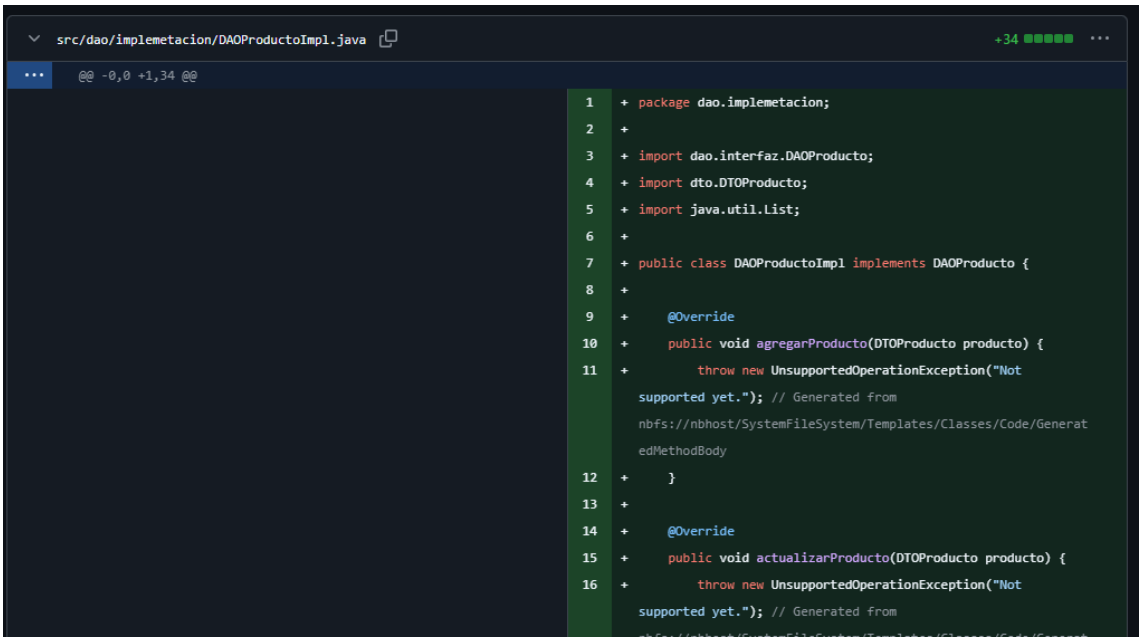



**Figura 27**

Commits de GitHub , relacionados con DAOProducto

- **Elaboración por Ralph Sardon Martinez**

**Figura 28**

Archivos y lineas modificadas para la implementación de la lógica para el usuario.

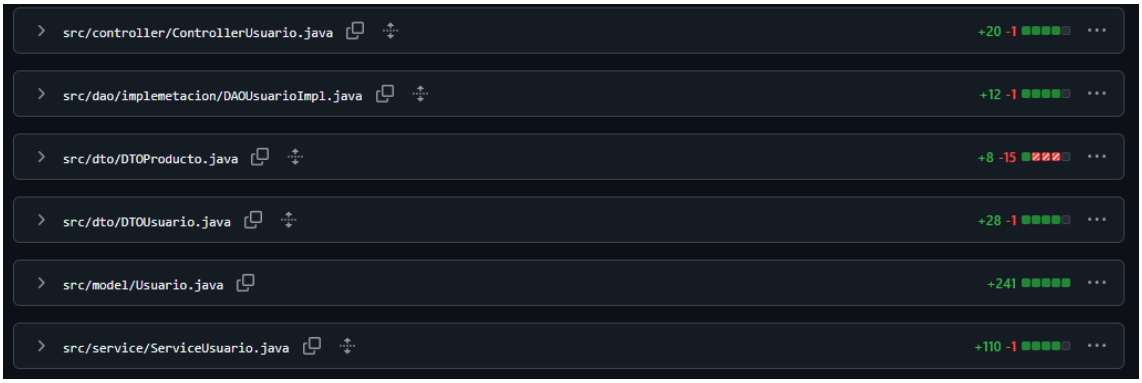



**Figura 29 –** Implementación de DTOUsuario para seguir el patrón de DAO

```java
private int idUsuario;
private String nombre;
private String apellido;
private String documento;
private String direccion;
private String telefono;
private String correo;
private int idTipoUsuario;
private String username;
private String password;

/**
 * Constructor vacio para crear un objeto {@code DTOUsuario} sin valores
 * iniciales.
 */
public DTOUsuario() {
}

/**
 * Constructor que inicializa un {@code DTOUsuario} a partir de un objeto
 * {@code Usuario}.
 *
 * @param usuario El objeto {@code Usuario} con los datos a transferir.
 */
public DTOUsuario(Usuario usuario) {
    this.idUsuario = usuario.getIdUsuario();
    this.nombre = usuario.getNombre();
    this.apellido = usuario.getApellido();
    this.documento = usuario.getDocumento();
    this.direccion = usuario.getDireccion();
    this.telefono = usuario.getTelefono();
    this.correo = usuario.getCorreo();
    this.idTipoUsuario = usuario.getIdTipoUsuario();
```

```java
public DTOUsuario(Usuario usuario) {
    this.idUsuario = usuario.getIdUsuario();
    this.nombre = usuario.getNombre();
    this.apellido = usuario.getApellido();
    this.documento = usuario.getDocumento();
    this.direccion = usuario.getDireccion();
    this.telefono = usuario.getTelefono();
    this.correo = usuario.getCorreo();
    this.idTipoUsuario = usuario.getIdTipoUsuario();
    this.username = usuario.getUsername();
    this.password = usuario.getPassword();
}

public DTOUsuario(int idUsuario, String nombre, String apellido, String do
    this.idUsuario = idUsuario;
    this.nombre = nombre;
    this.apellido = apellido;
    this.documento = documento;
    this.direccion = direccion;
    this.telefono = telefono;
    this.correo = correo;
    this.idTipoUsuario = idTipoUsuario;
    this.username = username;
    this.password = password;
}
```

### 3.4. SOLID

#### 3.4.1. Principio de Responsabilidad Única (Single Responsibility Principle)

La clase ServiceProducto es un ejemplo del Principio de Responsabilidad Única del marco SOLID, que establece que una clase debe tener una única razón para cambiar. En este caso, ServiceProducto se encarga exclusivamente de gestionar la lógica de negocio relacionada con los productos, actuando como un intermediario entre la capa de presentación y la capa de acceso a datos. Esta separación de responsabilidades permite que cualquier cambio en la lógica de negocio, como la adición de nuevas funcionalidades o la modificación de las existentes, no afecte a otras partes del sistema. Además, al delegar las operaciones de acceso a datos a la interfaz DAOProducto, se asegura que la clase permanezca enfocada en sus responsabilidades principales, facilitando su mantenimiento y ampliación en el futuro.

- **Elaboración por Emmanuel Rivera Gutierrez**

**Figura 30**

```
public class ServiceProducto {
    private DAOProducto daoProducto;

    public ServiceProducto() {
        this.daoProducto = new DAOProductoImpl();
    }

    public void agregarProducto(DTOProducto productoDTO) {
        daoProducto.agregarProducto(productoDTO);
    }

    public void actualizarProducto(DTOProducto productoDTO) {
        daoProducto.actualizarProducto(productoDTO);
    }

    public void eliminarProducto(long idProducto) {
        daoProducto.eliminarProducto(idProducto);
    }

    public DTOProducto obtenerProductoPorId(long idProducto) {
        DTOProducto producto = daoProducto.obtenerProductoPorId(idProducto);
        return producto != null ? producto : null;
    }

    public List<DTOProducto> obtenerTodosLosProductos() {
        List<DTOProducto> productos = daoProducto.obtenerTodosLosProductos();
        return productos;
    }
}
```

- **Elaboración por Ralph Sardon Martinez**

**Figura 31**

```java
/**
 * La clase {@code ServiceUsuario} proporciona servicios para la gestión de
 * usuarios, incluyendo operaciones de autenticación, registro, edición y
 * exportación de datos.
 *
 * @author Ralfph
 */
public class ServiceUsuario {

    private DAOUsuarioImpl daoUsuario;

    /**
     * Constructor que inicializa el servicio de usuario, estableciendo la
     * conexión con la base de datos a través de {@code DAOUsuarioImpl}.
     *
     * @throws SQLException Si ocurre un error al establecer la conexión con la
     * base de datos.
     */
    public ServiceUsuario() throws SQLException {
        daoUsuario = new DAOUsuarioImpl();
    }

    /**
     * Verifica el inicio de sesión de un usuario basado en su nombre de
     * usuario, contraseña y tipo de usuario.
     *
     * @param usuario El nombre de usuario.
     * @param contrasena La contraseña del usuario.
     * @param tipoUsuario El tipo de usuario (ej. 1=administrador, 2=empleado).
     * @return {@code true} si las credenciales son válidas, {@code false} en
     * caso contrario.
     * @throws SQLException Si ocurre un error al acceder a la base de datos.
     */
public boolean login(String usuario, String contrasena, int tipoUsuario) throws SQLException {
    return daoUsuario.login(usuario, contrasena, tipoUsuario);
}

/**
 * Registra un nuevo usuario en el sistema.
 *
 * @param usuario El objeto {@code DTOUsuario} que contiene la información
 * del usuario a registrar.
 * @return {@code true} si el registro fue exitoso, {@code false} en caso
 * contrario.
 * @throws SQLException Si ocurre un error durante la inserción de datos en
 * la base de datos.
 */
public boolean registrarUsuario(DTOUsuario usuario) throws SQLException {
    return daoUsuario.registrarUsuario(usuario);
}

public List<DTOUsuario> buscarUsuarioPorCriterios(String nombre, String apellido, String documento, String correo) throws SQLException {
    return daoUsuario.buscarUsuarioPorCriterios(nombre, apellido, documento, correo);
}
```

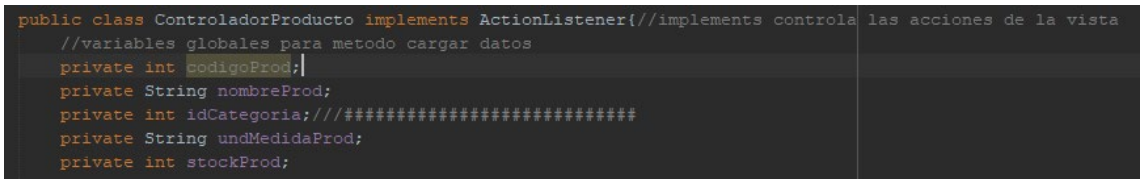
```java
/**
 * Elimina un usuario de la base de datos según su documento de
 * identificación.
 *
 * @param documento El documento de identificación del usuario a eliminar.
 * @return {@code true} si el usuario fue eliminado correctamente,
 * {@code false} en caso contrario.
 * @throws SQLException Si ocurre un error al eliminar el usuario en la base
 * de datos.
 */
public boolean eliminarUsuarioPorDocumento(String documento) throws SQLException {
    return daoUsuario.eliminarUsuarioPorDocumento(documento);
}

/**
 * Edita la información de un usuario existente.
 *
 * @param usuario El objeto {@code DTOUsuario} que contiene la información
 * actualizada del usuario.
 * @return {@code true} si la actualización fue exitosa, {@code false} en
 * caso contrario.
 * @throws SQLException Si ocurre un error al actualizar los datos en la
 * base de datos.
 */
public boolean editarUsuario(DTOUsuario usuario) throws SQLException {
    return daoUsuario.editarUsuario(usuario);
}
```

### 3.4.2. Principio de Abierto/Cerrado (Open/Closed Principle)

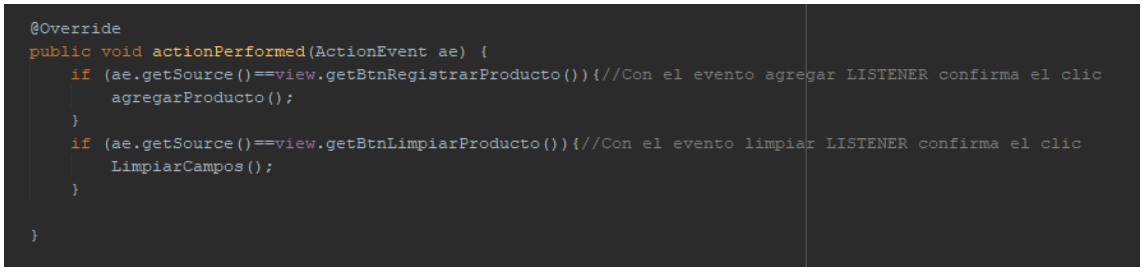
La clase ControladorProducto ilustra el Principio de Abierto/Cerrado del marco SOLID, que establece que las clases deben estar abiertas para la extensión, pero cerradas para la modificación. En esta implementación, el controlador se encarga de gestionar las acciones de la interfaz gráfica de usuario, delegando la lógica de negocio a la clase DAOProductoImpl para las operaciones de acceso a datos. Esto permite que nuevas funcionalidades, como la adición de métodos para editar o eliminar productos, se puedan implementar sin necesidad de modificar la clase ControladorProducto. En lugar de ello, se pueden agregar nuevos métodos o incluso crear subclases que extiendan el comportamiento del controlador existente, lo que favorece la mantenibilidad y la escalabilidad del sistema sin afectar el código ya implementado.

- **Elaboración por Elvis Soto Luque**

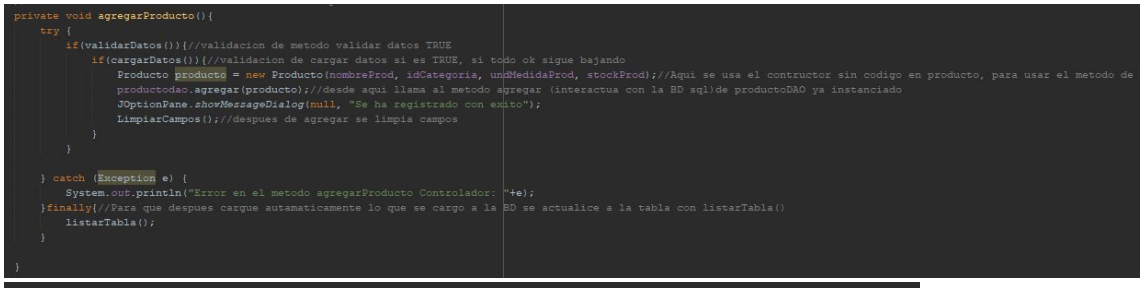
**Figura 32**



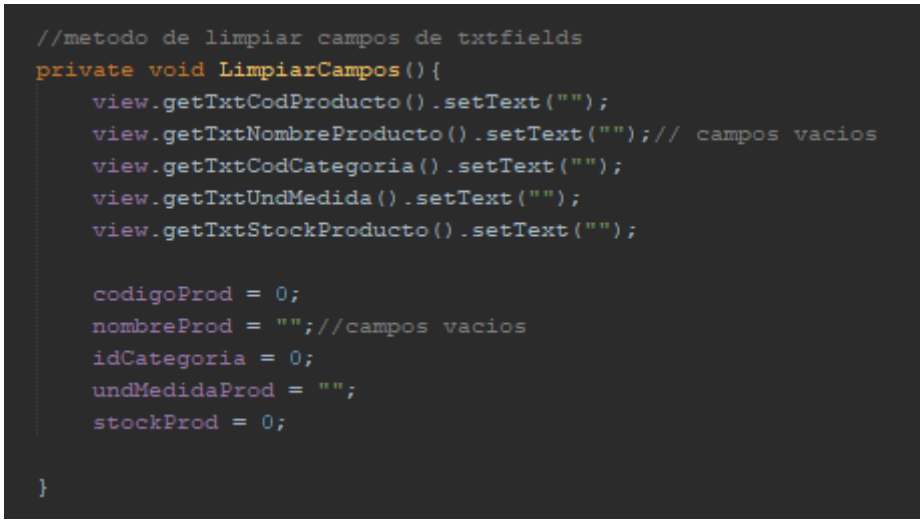
```
public class ControladorProducto implements ActionListener{//implements controla las acciones de la vista
    //variables globales para metodo cargar datos
    private int codigoProd;
    private String nombreProd;
    private int idCategoria;///###########################
    private String undMedidaProd;
    private int stockProd;
```



```
@Override
public void actionPerformed(ActionEvent ae) {
    if (ae.getSource()==view.getBtnRegistrarProducto()){//Con el evento agregar LISTENER confirma el clic
        agregarProducto();
    }
    if (ae.getSource()==view.getBtnLimpiarProducto()){//Con el evento limpiar LISTENER confirma el clic
        LimpiarCampos();
    }

}
```



```
private void agregarProducto(){
    try {
        if(validarDatos()){//validacion de metodo validar datos TRUE
            if(cargarDatos()){//validacion de cargar datos si es TRUE, si todo ok sigue bajando
                Producto producto = new Producto(nombreProd, idCategoria, undMedidaProd, stockProd);//Aqui se usa el contructor sin codigo en producto, para usar el metodo de
                productodao.agregar(producto);//desde aqui llama al metodo agregar (interactua con la BD sql)de productoDAO ya instanciado
                JOptionPane.showMessageDialog(null, "Se ha registrado con exito");
                LimpiarCampos();//despues de agregar se limpia campos
            }
        }

    } catch (Exception e) {
        System.out.println("Error en el metodo agregarProducto Controlador: "+e);
    }finally{//Para que despues cargue autamaticamente lo que se cargo a la BD se actualice a la tabla con listarTabla()
        listarTabla();
    }

}
```



```
//metodo de limpiar campos de txtfields
private void LimpiarCampos(){
    view.getTxtCodProducto().setText("");
    view.getTxtNombreProducto().setText("");// campos vacios
    view.getTxtCodCategoria().setText("");
    view.getTxtUndMedida().setText("");
    view.getTxtStockProducto().setText("");

    codigoProd = 0;
    nombreProd = "";//campos vacios
    idCategoria = 0;
    undMedidaProd = "";
    stockProd = 0;

}
```

### 3.4.3. Principio de Sustitución de Liskov (Liskov Substitution Principle)

La clase ServiceProducto ejemplifica el Principio de Sustitución de Liskov, que establece que las clases derivadas deben ser sustituibles por sus clases base sin afectar la funcionalidad del programa. En este caso, ServiceProducto utiliza una interfaz DAOProducto, lo que permite que cualquier implementación de esta interfaz pueda ser utilizada de manera intercambiable. Por ejemplo, si se desarrollara una nueva clase que implemente la interfaz DAOProducto, como DAOProductoMock, para propósitos de prueba, ServiceProducto seguiría funcionando correctamente sin necesidad de modificaciones. Este enfoque asegura que la lógica de negocio en ServiceProducto permanezca independiente de las implementaciones concretas de acceso a datos, promoviendo la reutilización y el mantenimiento del código a lo largo del ciclo de vida del desarrollo.

- **Elaboración por Emmanuel Rivera Gutierrez**

**Figura 33**

```
public class ServiceProducto {
    private DAOProducto daoProducto;

    public ServiceProducto() {
        this.daoProducto = new DAOProductoImpl();
    }

    public void agregarProducto(DTOProducto productoDTO) {
        daoProducto.agregarProducto(productoDTO);
    }

    public void actualizarProducto(DTOProducto productoDTO) {
        daoProducto.actualizarProducto(productoDTO);
    }

    public void eliminarProducto(long idProducto) {
        daoProducto.eliminarProducto(idProducto);
    }

    public DTOProducto obtenerProductoPorId(long idProducto) {
        DTOProducto producto = daoProducto.obtenerProductoPorId(idProducto);
        return producto != null ? producto : null;
    }

    public List<DTOProducto> obtenerTodosLosProductos() {
        List<DTOProducto> productos = daoProducto.obtenerTodosLosProductos();
        return productos;
    }
}
```

### 3.4.4. Principio de Segregación de Interfaces (Interface Segregation Principle)

- **Elaboración por Yinyer Quispe Rodriguez**

**Figura 34**

```java
package dao.interfaz;

import dto.DTOProducto;
import java.util.List;

public interface DAOProducto {
    public void agregarProducto(DTOProducto producto);
    public void actualizarProducto(DTOProducto producto);
    public void eliminarProducto(long idProducto);
    public DTOProducto obtenerProductoPorId(long idProducto);
    public List<DTOProducto> obtenerTodosLosProductos();
}
```

**Figura 35**

```java
/**
 *
 * @author Yinyer
 */
public interface DAOCategoria {
    /**
     * Agrega una nueva categoria a la base de datos.
     *
     * @param categoria el objeto {@link DTOCategoria} que representa la categoria a agregar
     */
    public void agregarCategoria(DTOCategoria categoria);

    /**
     * Actualiza los datos de una categoria existente en la base de datos.
     *
     * @param categoria el objeto {@link DTOCategoria} con los datos actualizados de la categoria
     */
    public void actualizarCategoria(DTOCategoria categoria);

    /**
     * Elimina una categoria de la base de datos según su identificador único.
     *
     * @param idCategoria el identificador único de la categoria a eliminar
     */
    public void eliminarCategoria(int idCategoria);
```

- **Elaboración por Ralph Sardon Martinez**

**Figura 36**

```java
package dao.interfaz;

import dto.DTOUsuario;
import java.sql.SQLException;
import java.util.List;

/**
 *
 * @author rasmx
 */
public interface DAOUsuario {

    boolean login(String usuario, String contrasena, int tipoUsuario) throws SQLException;

    boolean registrarUsuario(DTOUsuario usuario) throws SQLException;

    public List<DTOUsuario> buscarUsuarioPorCriterios(String nombre, String apellido, String documento,

    public boolean eliminarUsuarioPorDocumento(String documento) throws SQLException;

    public boolean editarUsuario(DTOUsuario usuario) throws SQLException;

    public List<DTOUsuario> obtenerUsuarios() throws SQLException;
}
```

3.4.5. Principio de Inversión de Dependencias (Dependency Inversion Principle)

Este principio se aplica cuando las clases de alto nivel no deben depender de las de bajo nivel directamente, sino de abstracciones. Al definir interfaces como DAOProducto y DAOUsuario, y luego inyectar dependencias de las clases concretas, como DAOProductoImpl, aplicas DIP. Aún mejor si utilizas un framework de inyección de dependencias como Spring para gestionar esas dependencias.

**Figura 37**

```java
public class ServiceProducto {
    private DAOProducto daoProducto;

    public ServiceProducto() {
        daoProducto = new DAOProductoImpl();
    }
```

**Figura 38**

```java
public class ServiceUsuario {

    private DAOUsuarioImpl daoUsuario;

    /**
     * Constructor que inicializa el servicio de usuario, estableciendo la
     * conexión con la base de datos a través de {@code DAOUsuarioImpl}.
     *
     * @throws SQLException Si ocurre un error al establecer la conexión con la
     * base de datos.
     */
    public ServiceUsuario() throws SQLException {
        daoUsuario = new DAOUsuarioImpl();
    }
```

## 4. Recursos Java

### 4.1. Google Guava

**Clase Strings**

-La clase Strings en Google Guava proporciona métodos esenciales para la manipulación de cadenas. Sus principales funcionalidades incluyen:

-Verificación de nulidad. isNullOrEmpty(String str) determina si una cadena es nula o vacía.

-Repetición de cadenas. repeat(String str, int times) genera una nueva cadena que repite la cadena original un número definido de veces.

**Clase CharMatcher**

CharMatcher ofrece herramientas para el manejo eficiente de caracteres. Sus características incluyen:

-Definición de patrones. Permite crear coincidentes para caracteres específicos, como letras o dígitos.

-Negación de patrones. Se pueden definir coincidentes que excluyen caracteres específicos.

**Figura 39**

Importación de librerías necesarias para la implementación de Google guava Strings y ChartMatcher

```
import com.google.common.base.CharMatcher;
import com.google.common.base.Strings;
```

**Figura 40**

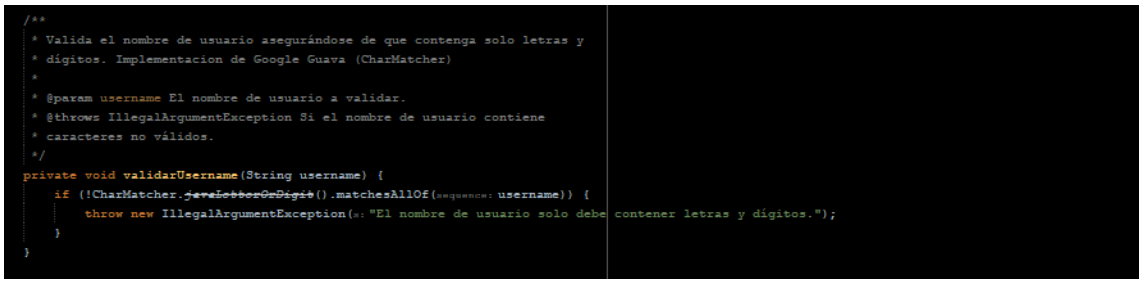
Implementación de Google Guava (Strings)

```
private String ValidarEntradaUsuario(String nombre, String apellido, String documento, String direccion, String telefono, String correo, int idTipoUsuario, String username, String password) {
    if (Strings.isNullOrEmpty(string:nombre)) {
        return "El nombre no debe ser nulo o vacío.";
    }
    if (Strings.isNullOrEmpty(string:apellido)) {
        return "El apellido no debe ser nulo o vacío.";
    }
    if (Strings.isNullOrEmpty(string:documento)) {
        return "El documento no debe ser nulo o vacío.";
    }
    if (Strings.isNullOrEmpty(string:direccion)) {
        return "La dirección no debe ser nula o vacía.";
    }
    if (Strings.isNullOrEmpty(string:telefono)) {
        return "El teléfono no debe ser nulo o vacío.";
    }
    if (Strings.isNullOrEmpty(string:correo)) {
        return "El correo no debe ser nulo o vacío.";
    }
    if (idTipoUsuario <= 0) {
        return "Selecciona un tipo de usuario válido.";
    }
    if (Strings.isNullOrEmpty(string:username)) {
        return "El nombre de usuario no debe ser nulo o vacío.";
    }
    if (Strings.isNullOrEmpty(string:password)) {
        return "La contraseña no debe ser nula o vacía.";
    }
    return null;
}
```

Nota. En la imagen, se observa el método ValidarEntradaUsuario que asegura de que todos los campos requeridos no sean nulos o vacíos, y que el tipo de usuario sea válido.

**Figura 41**

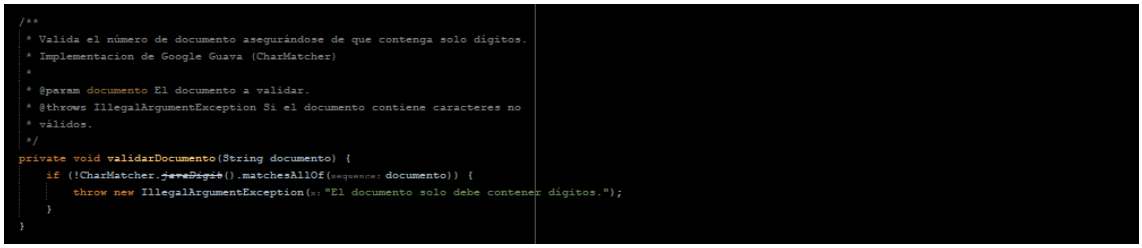
Implementación de Google Guava (CharMatcher) en el método validarUsername




Nota. En la imagen se muestra la creación del método validarUsername asegurando que el campo de username contenga solo letras y dígitos.

**Figura 42**

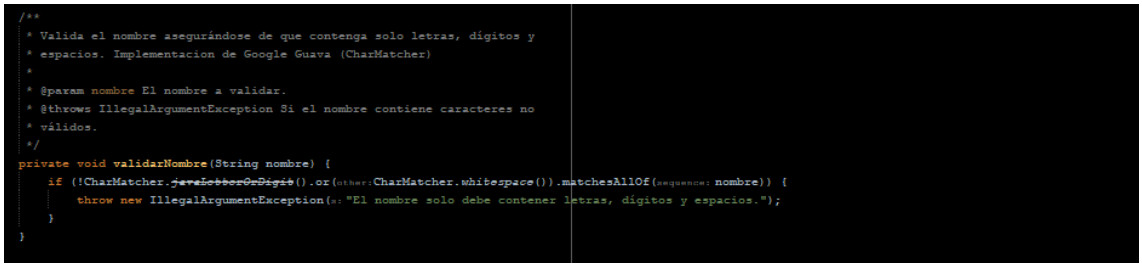
Implementación de Google Guava (CharMatcher) en el método validarDocumento




Nota. Valida el número de documento asegurando de que contenga solo dígitos.

**Figura 43**

Implementación de Google Guava (CharMatcher) en el método validarNombre




Nota. Valida el nombre asegurando de que contenga solo letras, dígitos y espacios

### 4.2. Apache POI

Apache POI es una biblioteca de Java diseñada para trabajar con documentos de Microsoft Office, específicamente Excel, Word y PowerPoint. Su versatilidad permite leer, crear y modificar archivos en formatos .xls, .xlsx, .doc, .docx y .ppt, entre otros.

-HSSFWorkbook. Utilizada para trabajar con archivos en formato .xls (Excel 97-2003), permite la lectura y escritura de datos de manera eficiente.

-XSSFWorkbook. Diseñada para archivos .xlsx (Excel 2007 en adelante), ofrece funcionalidades mejoradas, como un manejo más efectivo de grandes volúmenes de datos.

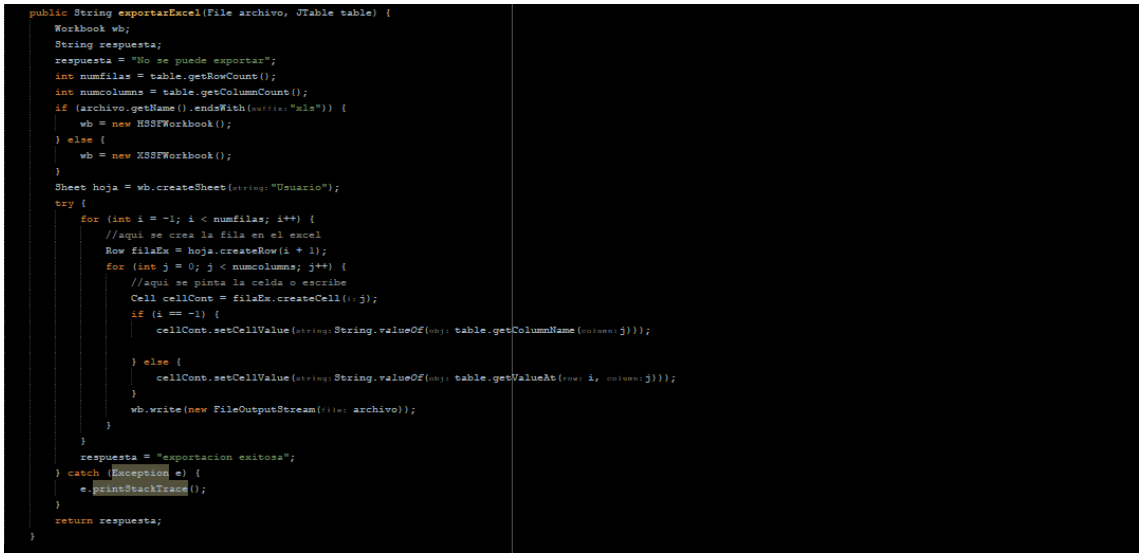
**Figura 44**

Importación de librerías necesarias para la implementación de Apache POI.

```
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
```

**Figura 45**

Implementación de Apache POI en el método exportarExcel

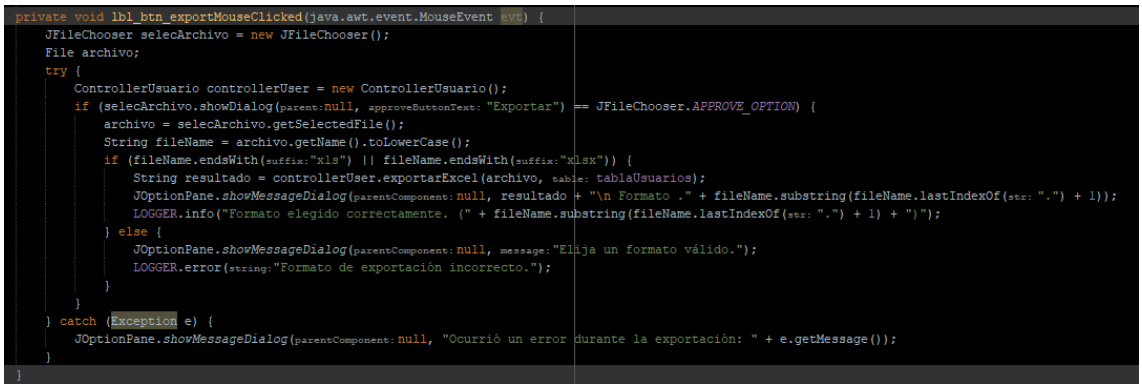


```
public String exportarExcel(File archivo, JTable table) {
    Workbook wb;
    String respuesta;
    respuesta = "No se puede exportar";
    int numfilas = table.getRowCount();
    int numcolumns = table.getColumnCount();
    if (archivo.getName().endsWith(suffix:"xls")) {
        wb = new HSSFWorkbook();
    } else {
        wb = new XSSFWorkbook();
    }
    Sheet hoja = wb.createSheet(string:"Usuario");
    try {
        for (int i = -1; i < numfilas; i++) {
            //aqui se crea la fila en el excel
            Row filaEx = hoja.createRow(i + 1);
            for (int j = 0; j < numcolumns; j++) {
                //aqui se pinta la celda o escribe
                Cell cellCont = filaEx.createCell(i:j);
                if (i == -1) {
                    cellCont.setCellValue(string:String.valueOf(obj: table.getColumnName(column:j)));

                } else {
                    cellCont.setCellValue(string:String.valueOf(obj: table.getValueAt(row:i, column:j)));
                }
                wb.write(new FileOutputStream(file: archivo));
            }
        }
        respuesta = "exportacion exitosa";
    } catch (Exception e) {
        e.printStackTrace();
    }
    return respuesta;
}
```

Nota. En la imagen se muestra el método que exporta los datos de una tabla a un archivo de Excel. Se reciben como parámetros el archivo de Excel de destino y la tabla cuyos datos se desean exportar. El método retorna un mensaje que indica si la exportación fue exitosa o si ocurrió un fallo durante el proceso.

**Figura 46**

Implementación del método exportarExcel en el la interfaz de ViewRegistroUsuario



```
private void lbl_btn_exportMouseClicked(java.awt.event.MouseEvent evt) {
    JFileChooser selecArchivo = new JFileChooser();
    File archivo;
    try {
        ControllerUsuario controllerUser = new ControllerUsuario();
        if (selecArchivo.showDialog(parent:null, approveButtonText: "Exportar") == JFileChooser.APPROVE_OPTION) {
            archivo = selecArchivo.getSelectedFile();
            String fileName = archivo.getName().toLowerCase();
            if (fileName.endsWith(suffix:"xls") || fileName.endsWith(suffix:"xlsx")) {
                String resultado = controllerUser.exportarExcel(archivo, table: tablaUsuarios);
                JOptionPane.showMessageDialog(parentComponent:null, resultado + "\n Formato ." + fileName.substring(fileName.lastIndexOf(str: ".") + 1));
                LOGGER.info("Formato elegido correctamente. (" + fileName.substring(fileName.lastIndexOf(str: ".") + 1) + ")");
            } else {
                JOptionPane.showMessageDialog(parentComponent:null, message:"Elija un formato válido.");
                LOGGER.error(string:"Formato de exportación incorrecto.");
            }
        }
    } catch (Exception e) {
        JOptionPane.showMessageDialog(parentComponent:null, "Ocurrió un error durante la exportación: " + e.getMessage());
    }
}
```

Nota. En la figura se observa, la instancia del metodo exportarExcel que permite exportar los datos de la tabla de usuarios a un archivo de Excel (.xls o .xlsx). El archivo es guardado en la ubicación seleccionada por el usuario.

## 4.3. Apache Commons

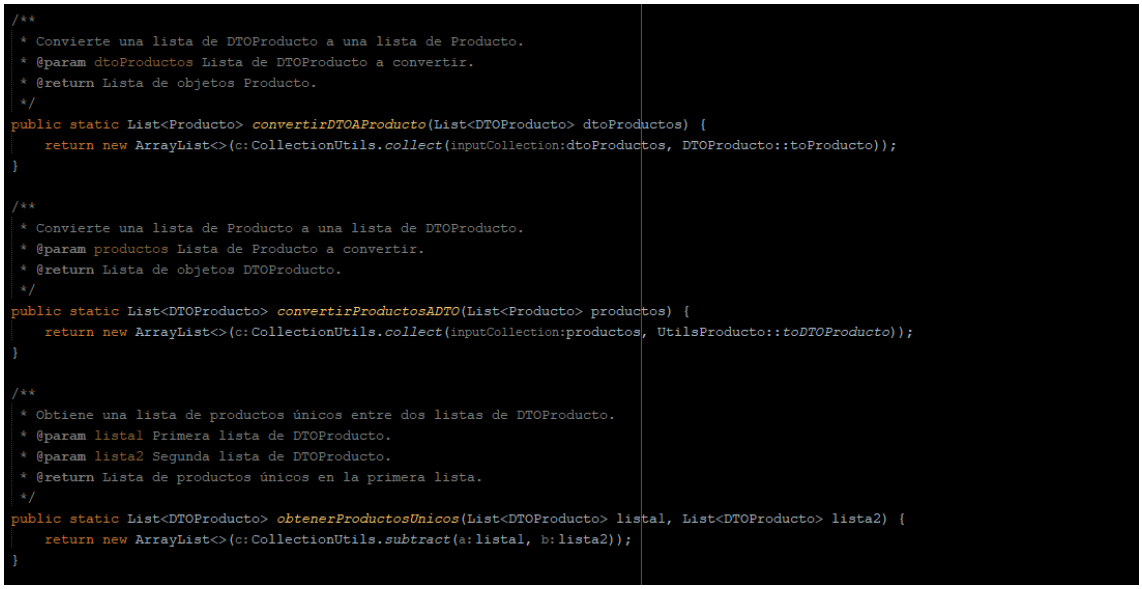
**Figura 47**

Importación de librerías necesarias para la implementación de Apache Commons.

```
import org.apache.commons.collections4.CollectionUtils;
import org.apache.commons.collections4.Predicate;
import org.apache.commons.lang3.StringUtils;
import org.apache.commons.lang3.math.NumberUtils;
```

**Figura 48**

Implementación de Apache Commons usando colecciones.

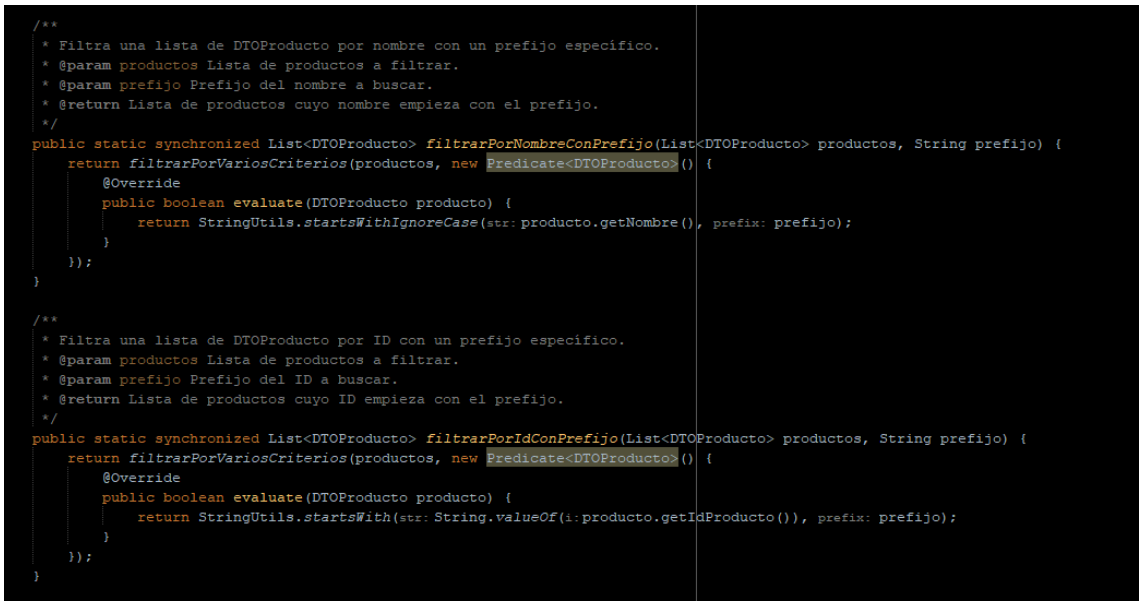


```
/**
 * Convierte una lista de DTOProducto a una lista de Producto.
 * @param dtoProductos Lista de DTOProducto a convertir.
 * @return Lista de objetos Producto.
 */
public static List<Producto> convertirDTOAProducto(List<DTOProducto> dtoProductos) {
    return new ArrayList<>(c: CollectionUtils.collect(inputCollection:dtoProductos, DTOProducto::toProducto));
}

/**
 * Convierte una lista de Producto a una lista de DTOProducto.
 * @param productos Lista de Producto a convertir.
 * @return Lista de objetos DTOProducto.
 */
public static List<DTOProducto> convertirProductosADTO(List<Producto> productos) {
    return new ArrayList<>(c: CollectionUtils.collect(inputCollection:productos, UtilsProducto::toDTOProducto));
}

/**
 * Obtiene una lista de productos únicos entre dos listas de DTOProducto.
 * @param lista1 Primera lista de DTOProducto.
 * @param lista2 Segunda lista de DTOProducto.
 * @return Lista de productos únicos en la primera lista.
 */
public static List<DTOProducto> obtenerProductosUnicos(List<DTOProducto> lista1, List<DTOProducto> lista2) {
    return new ArrayList<>(c: CollectionUtils.subtract(a: lista1, b: lista2));
}
```

Nota. Esta clase proporciona métodos para convertir listas de objetos entre DTOProducto y Producto, así como para obtener productos únicos de dos listas. Facilita la gestión de datos de productos, optimizando la interoperabilidad entre diferentes representaciones.

**Figura 49**

Implementación de Apache Commons (StringUtils)

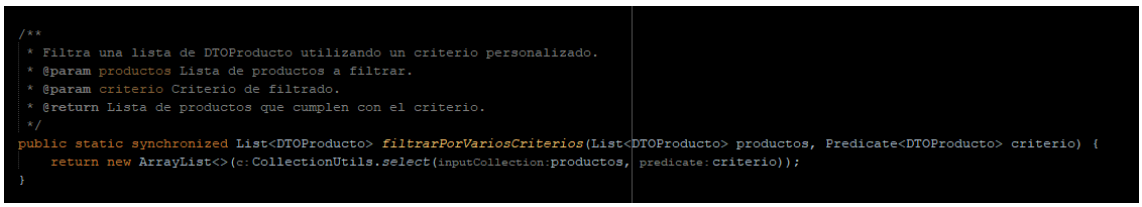


```java
/**
 * Filtra una lista de DTOProducto por nombre con un prefijo especifico.
 * @param productos Lista de productos a filtrar.
 * @param prefijo Prefijo del nombre a buscar.
 * @return Lista de productos cuyo nombre empieza con el prefijo.
 */
public static synchronized List<DTOProducto> filtrarPorNombreConPrefijo(List<DTOProducto> productos, String prefijo) {
    return filtrarPorVariosCriterios(productos, new Predicate<DTOProducto>() {
        @Override
        public boolean evaluate(DTOProducto producto) {
            return StringUtils.startsWithIgnoreCase(str: producto.getNombre(), prefix: prefijo);
        }
    });
}

/**
 * Filtra una lista de DTOProducto por ID con un prefijo especifico.
 * @param productos Lista de productos a filtrar.
 * @param prefijo Prefijo del ID a buscar.
 * @return Lista de productos cuyo ID empieza con el prefijo.
 */
public static synchronized List<DTOProducto> filtrarPorIdConPrefijo(List<DTOProducto> productos, String prefijo) {
    return filtrarPorVariosCriterios(productos, new Predicate<DTOProducto>() {
        @Override
        public boolean evaluate(DTOProducto producto) {
            return StringUtils.startsWith(str: String.valueOf(i:producto.getIdProducto()), prefix: prefijo);
        }
    });
}
```

Nota. Esta clase contiene métodos para filtrar listas de objetos DTOProducto. El método filtrarPorIdConPrefijo permite obtener productos cuyos IDs comienzan con un prefijo específico, mientras que filtrarPorVariosCriterios aplica un criterio de filtrado personalizado. Ambos métodos son sincronizados, garantizando un acceso seguro en entornos concurrentes, y retornan listas filtradas de productos.

**Figura 50**

Implementación de Apache Commons usando colecciones.



```java
/**
 * Filtra una lista de DTOProducto utilizando un criterio personalizado.
 * @param productos Lista de productos a filtrar.
 * @param criterio Criterio de filtrado.
 * @return Lista de productos que cumplen con el criterio.
 */
public static synchronized List<DTOProducto> filtrarPorVariosCriterios(List<DTOProducto> productos, Predicate<DTOProducto> criterio) {
    return new ArrayList<>(c:CollectionUtils.select(inputCollection:productos, predicate:criterio));
}
```

Nota. El método filtrarPorVariosCriterios permite filtrar una lista de objetos DTOProducto utilizando un criterio de filtrado personalizado. Acepta una lista de productos y un Predicate que define las condiciones del filtrado. Este método es sincronizado, lo que asegura un acceso seguro en entornos concurrentes.

## 4.4. Logback

**Figura 51**

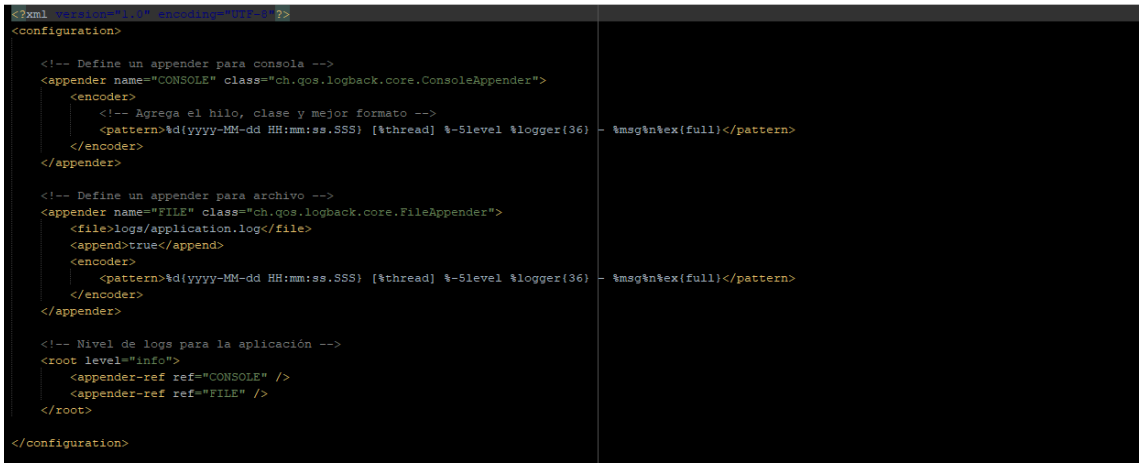
Implementación de LogBack en la clase UtilsLoggerManager

```java
package utils;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class UtilsLoggerManager {
    public static Logger getLogger(Class<?> clase) {
        return LoggerFactory.getLogger(clazz: clase);
    }
}
```

Nota. La clase UtilsLoggerManager proporciona una forma de obtener instancias de Logger utilizando SLF4J. El método getLogger recibe como parámetro una clase y retorna un logger asociado a esa clase. Esta implementación es flexible asi que se instanciar desde cualquier clase que lo requiera.

**Figura 52**

Creacion y configuracion de LogBack.xml.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <!-- Define un appender para consola -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!-- Agrega el hilo, clase y mejor formato -->
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n%ex{full}</pattern>
        </encoder>
    </appender>

    <!-- Define un appender para archivo -->
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>logs/application.log</file>
        <append>true</append>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n%ex{full}</pattern>
        </encoder>
    </appender>

    <!-- Nivel de logs para la aplicación -->
    <root level="info">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="FILE" />
    </root>

</configuration>
```

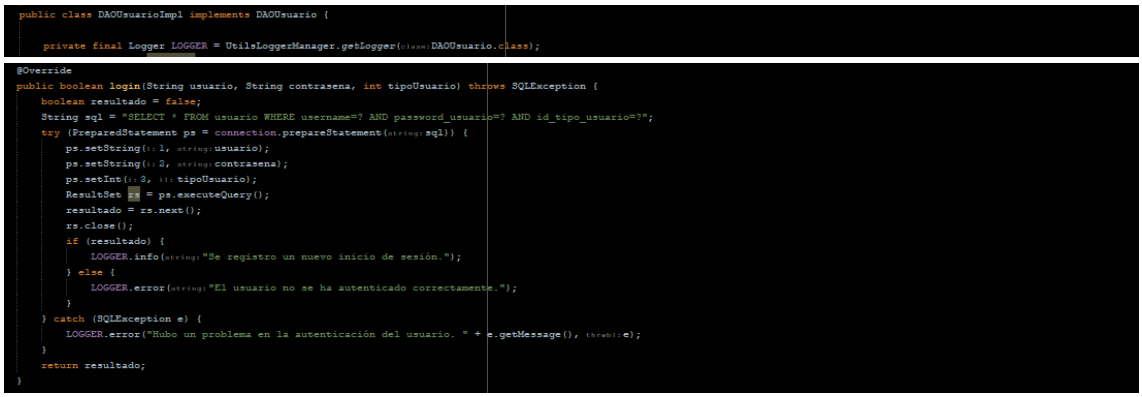
Nota. En esta imagen se presenta la configuración necesaria para implementar LogBack, permitiendo visualizar los logs de nivel ERROR e INFO en la consola, así como guardar cada registro en un archivo. Esta configuración optimiza el monitoreo y la persistencia de los eventos registrados en la aplicación.

**Figura 53**

Importación de librerías necesarias para la implementación de LogBack

```
import org.slf4j.Logger;
import utils.UtilsLoggerManager;
```

**Figura 54**



```
public class DAOUsuarioImpl implements DAOUsuario {

    private final Logger LOGGER = UtilsLoggerManager.getLogger(clase: DAOUsuario.class);
```

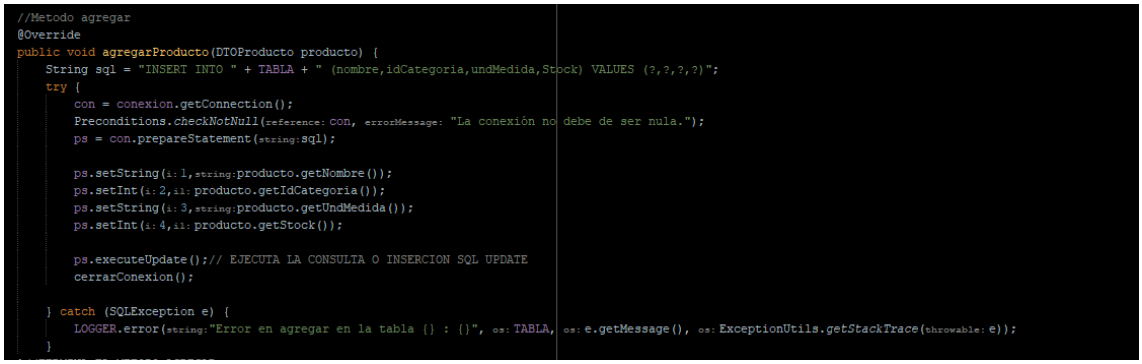
```
@Override
public boolean login(String usuario, String contrasena, int tipoUsuario) throws SQLException {
    boolean resultado = false;
    String sql = "SELECT * FROM usuario WHERE username=? AND password_usuario=? AND id_tipo_usuario=?";
    try (PreparedStatement ps = connection.prepareStatement(string: sql)) {
        ps.setString(i: 1, string: usuario);
        ps.setString(i: 2, string: contrasena);
        ps.setInt(i: 3, i1: tipoUsuario);
        ResultSet rs = ps.executeQuery();
        resultado = rs.next();
        rs.close();
        if (resultado) {
            LOGGER.info(string: "Se registro un nuevo inicio de sesión.");
        } else {
            LOGGER.error(string: "El usuario no se ha autenticado correctamente.");
        }
    } catch (SQLException e) {
        LOGGER.error("Hubo un problema en la autenticación del usuario. " + e.getMessage(), throwable: e);
    }
    return resultado;
}
```

Nota. En esta imagen se ilustra la instanciación de la clase UtilsLoggerManager para utilizar LogBack de manera flexible. En este caso, se aplica en el método login, donde se verifica la validez de la consulta a la base de datos. Si los datos ingresados son correctos,

se muestra un mensaje de nivel INFO en la consola; de lo contrario, se registra un mensaje de nivel ERROR.

**Figura 55**

Implementación de LogBack en la clase DAOProductoImpl.java



```
//Metodo agregar
@Override
public void agregarProducto(DTOProducto producto) {
    String sql = "INSERT INTO " + TABLA + " (nombre,idCategoria,undMedida,Stock) VALUES (?,?,?,?)";
    try {
        con = conexion.getConnection();
        Preconditions.checkNotNull(reference: con, errorMessage: "La conexión no debe de ser nula.");
        ps = con.prepareStatement(string:sql);

        ps.setString(i: 1,string:producto.getNombre());
        ps.setInt(i: 2,i1: producto.getIdCategoria());
        ps.setString(i: 3,string:producto.getUndMedida());
        ps.setInt(i: 4,i1: producto.getStock());

        ps.executeUpdate();// EJECUTA LA CONSULTA O INSERCION SQL UPDATE
        cerrarConexion();

    } catch (SQLException e) {
        LOGGER.error(string:"Error en agregar en la tabla {} : {}", o: TABLA, o: e.getMessage(), o: ExceptionUtils.getStackTrace(throwable: e));
    }
```

Nota. En esta imagen se muestra la implementación del método agregarProducto, que utiliza la clase UtilsLoggerManager para gestionar el registro de eventos con LogBack. Si los datos ingresados son correctos y la inserción en la base de datos se ejecuta exitosamente, se debería registrar un mensaje de nivel INFO. En caso de producirse un error durante la operación, se registra un mensaje de nivel ERROR, incluyendo detalles sobre la excepción generada.

## 5. Control de Versiones

### 5.1. README

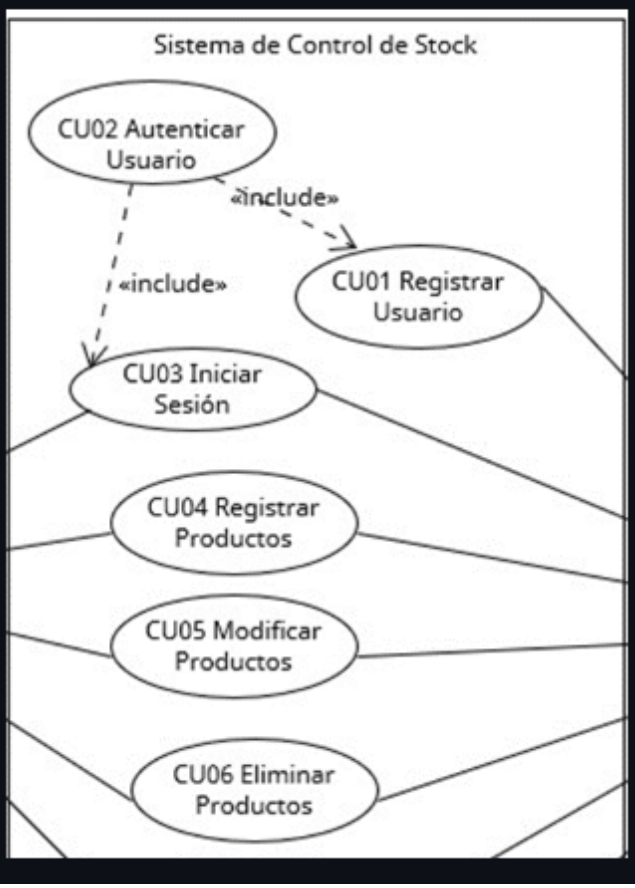
# Proyecto-Final-Curso-Integrador-I-Sistemas-Software | Sistema Control de Stock

> Este proyecto implementa un sistema de control de stock versátil, adecuado para diversas aplicaciones y entornos.

## Fase de Desarrollo (40%)

En esta fase se alcanzará un 40% del desarrollo del proyecto. Se han iniciado los trabajos y se han implementado varios casos de uso clave. Además, se han aplicado metodologías y patrones de diseño para asegurar un desarrollo sólido.

Imagen Referencial:



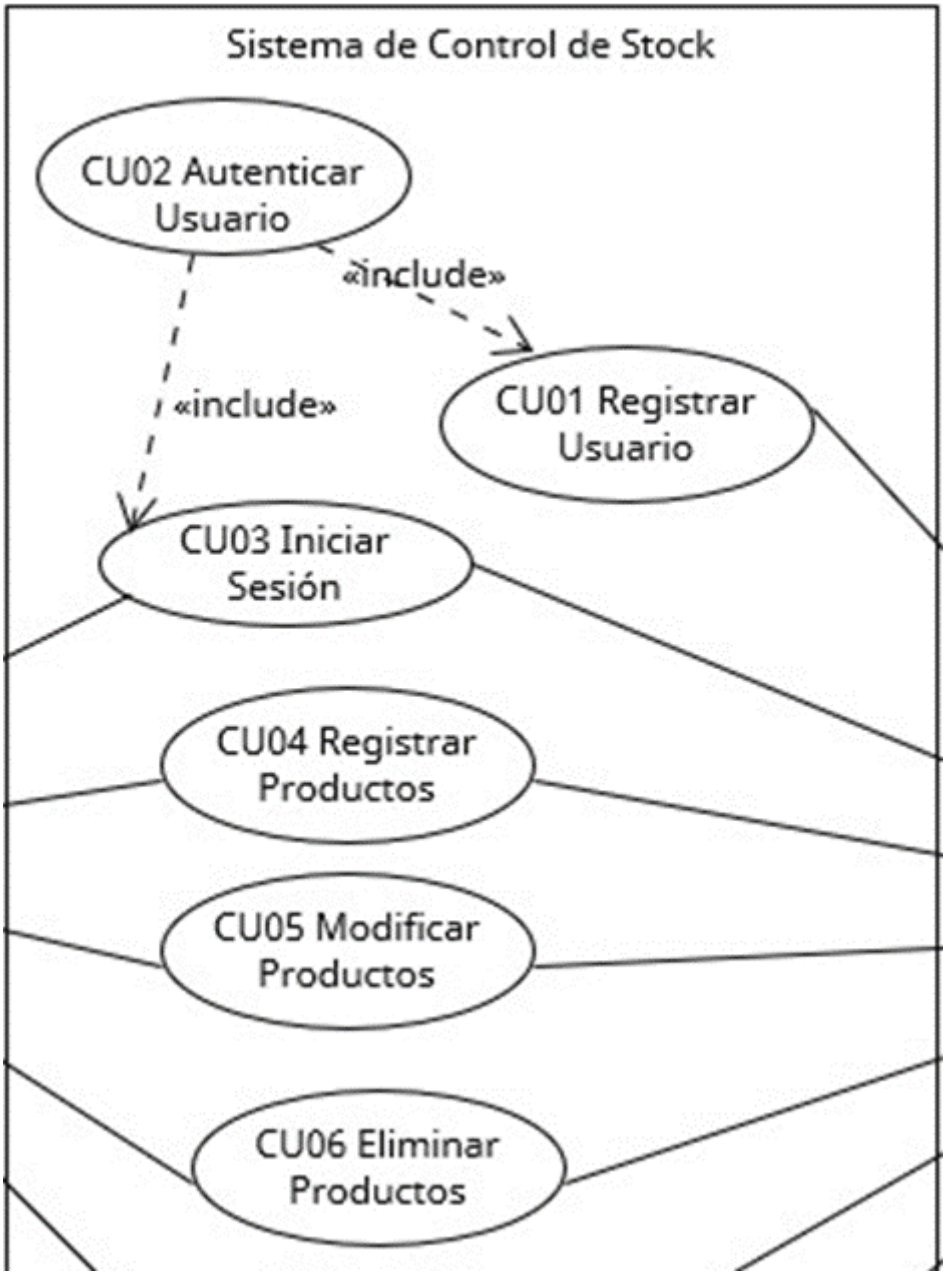

### Casos de uso a completar

- CU01 Registrar usuario
- CU02 Autenticar usuaio
- CU03 Iniciar sesión
- CU04 Registrar producto
- CU05 Modificar producto

### Casos de uso en proceso

- CU06 Eliminar producto (Interfaz gráfica)

**5.2. Diagrama CU especificando 40%**




**5.3. Ramas**

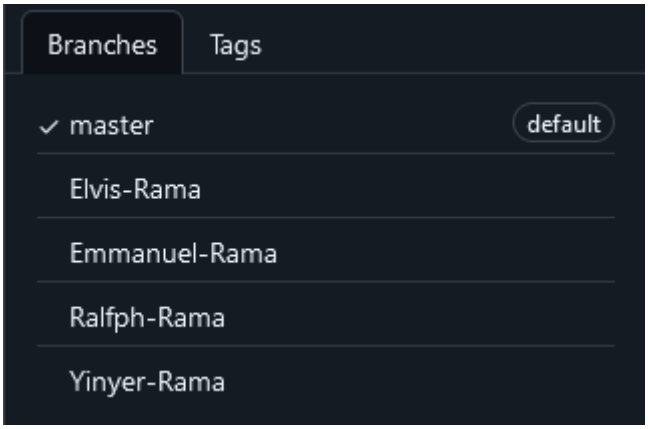
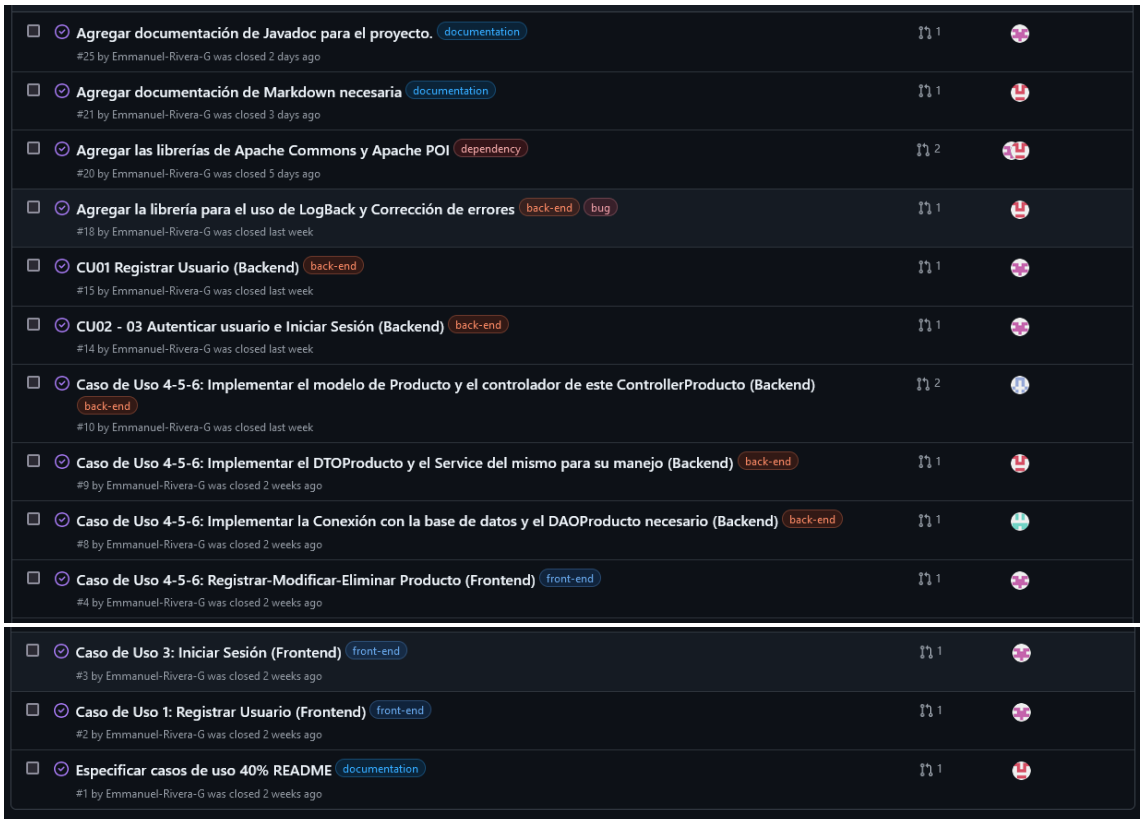


**Figura 56**

**Issues del proyecto relacionado con el avance del 40% de los casos de uso**



### 5.4. Gráfico de barras

**Figura 57**

Gráfico de Contribuciones de Commits Totales

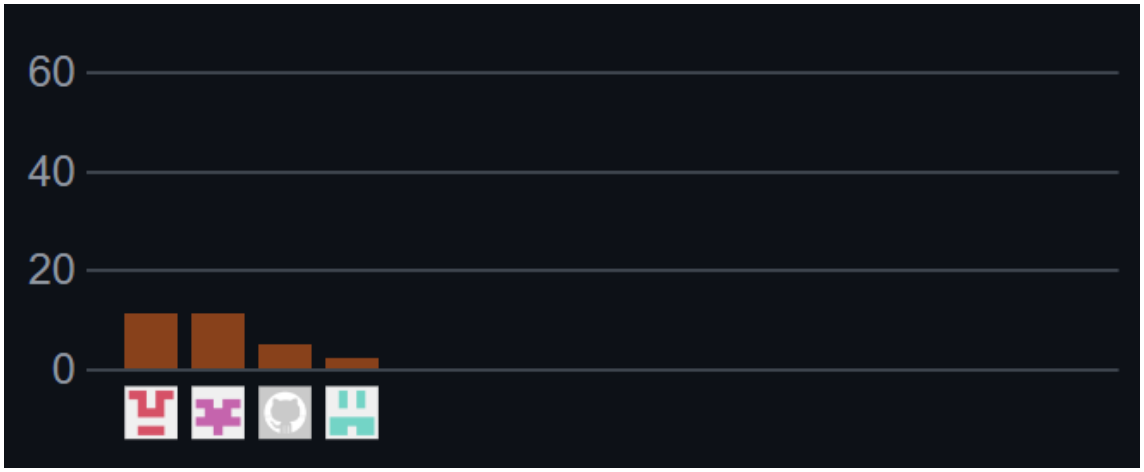



**Figura 58**

Gráfico de barras de Commits a lo largo del tiempo

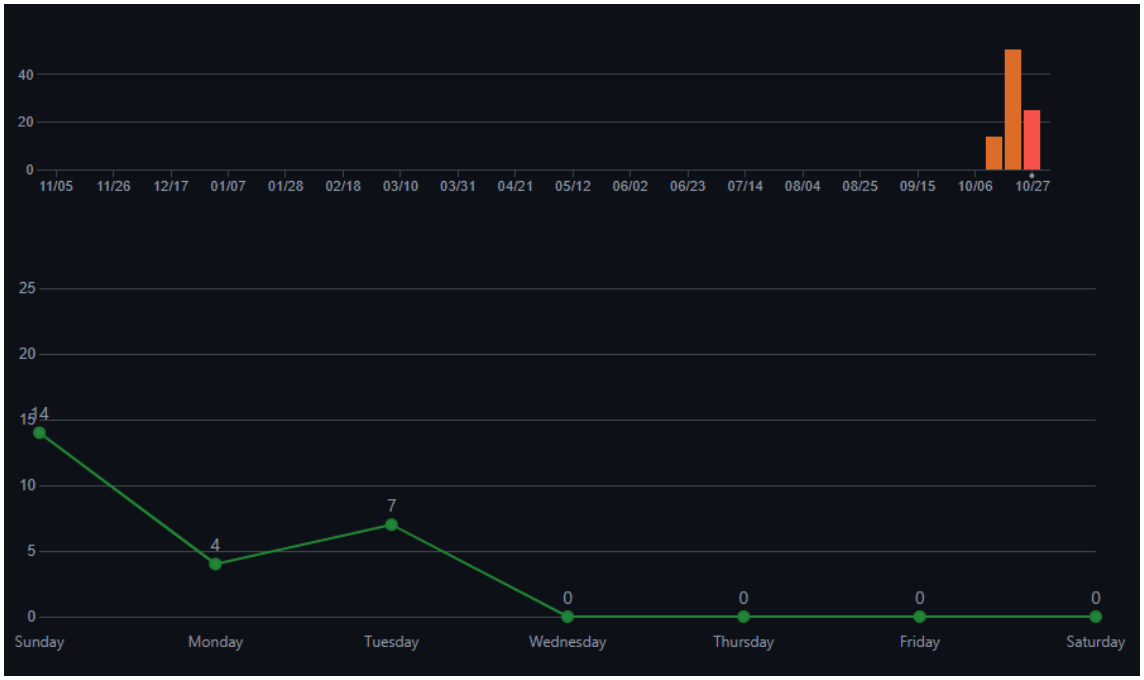


**Figura 59**

Gráfico de Contribuciones de Commits Semanales


**Figura 60**

Gráfico de Adiciones y Eliminaciones en los Commits

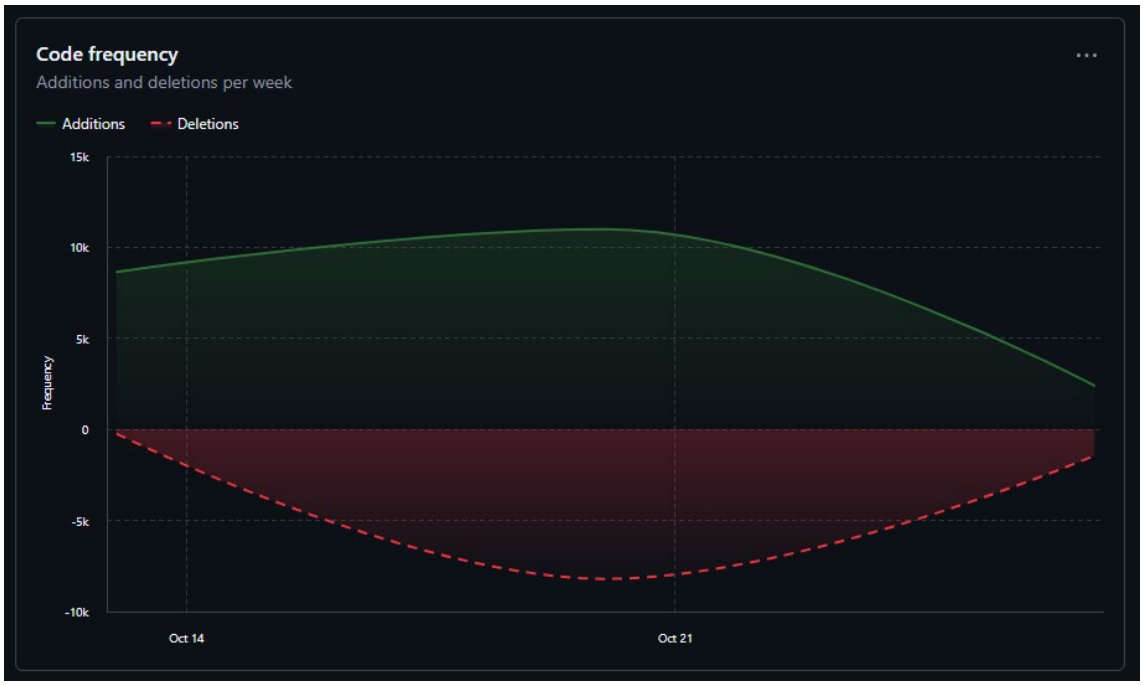


**Figura 61**

Gráfico de Red de Ramas

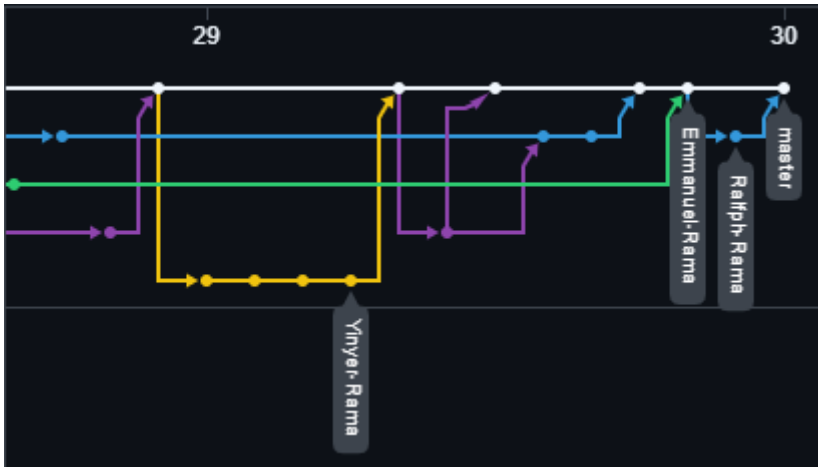
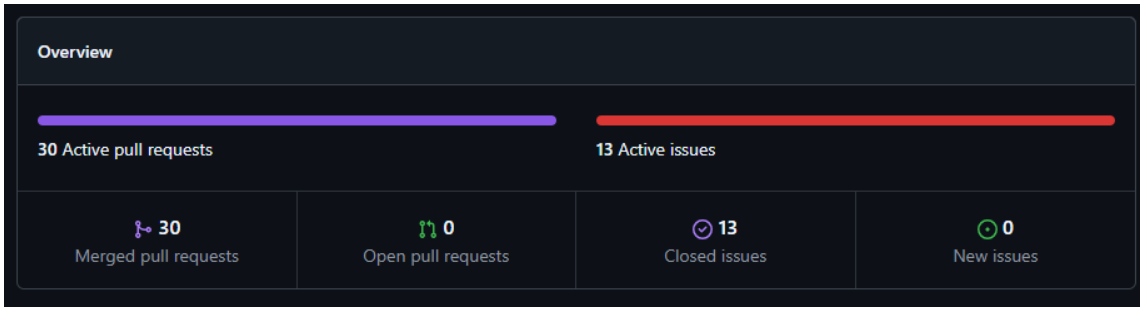



**Figura 62**

Gráfico de Issues creadas y Pull Requets unidas.




### 5.5. Commit de cada integrante

**Figura 63**

**Commit de Ralfph Sardon**



**Figura 64**

## Commit de Emmanuel Rivera

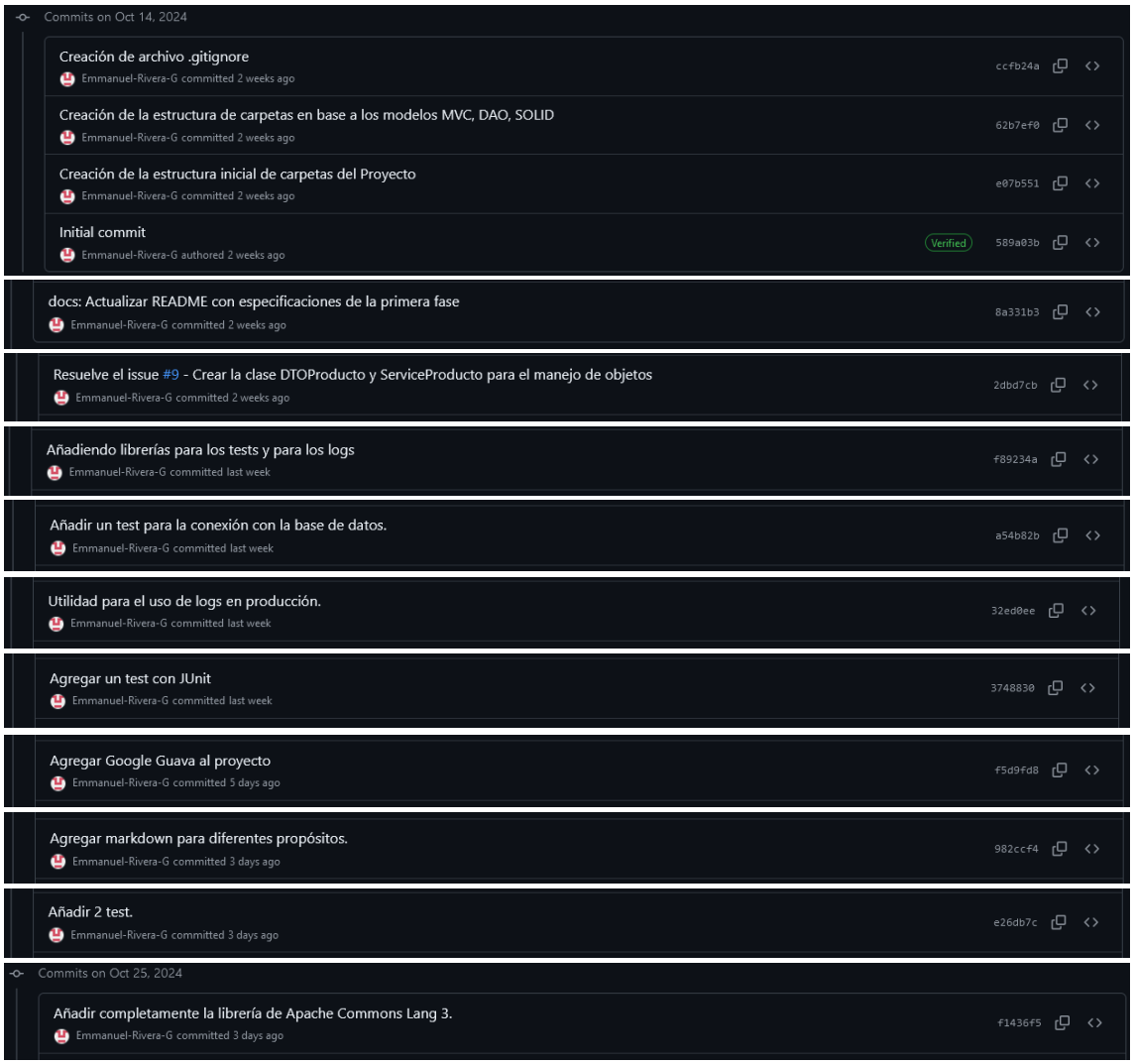


**Figura 65**

Commit de Elvis

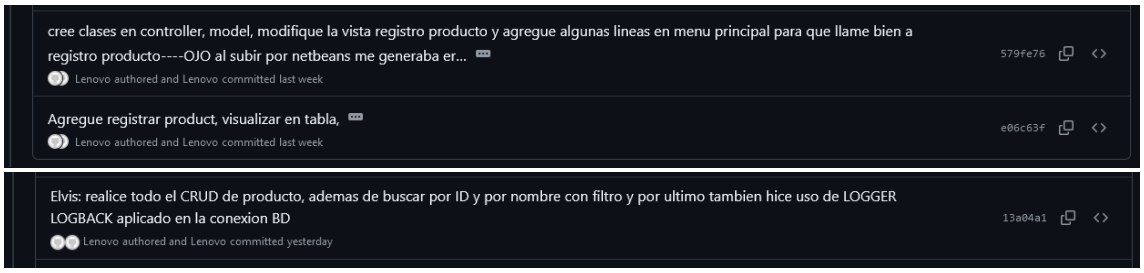
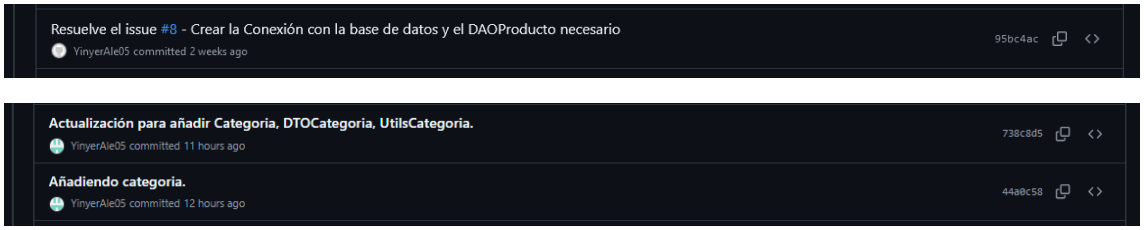



**Figura 66**

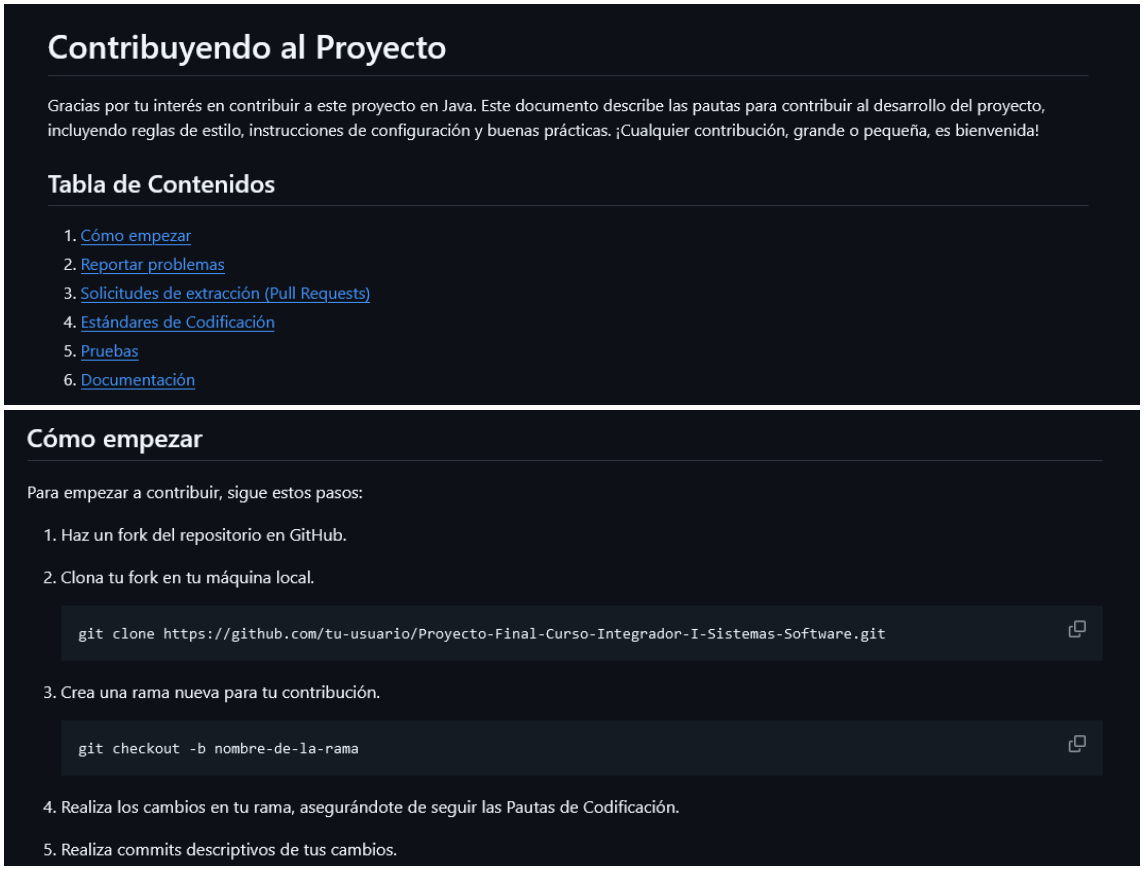
Commit de Yinyer




## 5.6. Documentación

**Figura 67**

Aplicación de Markdown para la documentación en GitHub del proyecto de sistema de control de stock Desarrollado en Java



6. Realiza pruebas para asegurar que tus cambios no rompan nada.
7. Abre un pull request en el repositorio original y describe los cambios realizados.

## Reportar problemas

Si encuentras un error o tienes una sugerencia de mejora, sigue estos pasos:

1. Asegúrate de que el problema o sugerencia no haya sido reportado anteriormente.
2. Si el problema es nuevo, abre un issue en el repositorio.
3. Describe el problema con el mayor detalle posible, proporcionando el contexto necesario para entenderlo y reproducirlo; si es posible, incluye el código fuente e imagenes de muestra.

### Formato de Issues

Para agilizar la gestión de los issues, proporciona la siguiente información:

- **Descripción del problema:** Explica el error o la sugerencia.
- **Pasos para reproducir:** Describe los pasos para replicar el problema.
- **Sistema y versión:** Indica el sistema operativo y la versión de Java utilizada.
- **Logs y capturas de pantalla:** Incluye cualquier log o captura que pueda ser útil para identificar el problema.

## Solicitudes de extracción (Pull Requests)

Para realizar una solicitud de extracción (Pull Request), sigue estas pautas:

1. Asegúrate de que tu código sigue los Estándares de Codificación y que no introduce errores en el proyecto.
2. Si es un cambio significativo, abre un issue primero para discutir la propuesta.
3. Incluye una descripción clara de los cambios realizados.
4. Agrega pruebas que cubran la nueva funcionalidad o los cambios realizados.
5. Asocia tu Pull Request a un issue existente, si aplica.
6. Asegúrate de que tu código pase todas las pruebas antes de enviar el Pull Request.

## Estándares de Codificación

Por favor, sigue estos estándares de estilo:

- **Formato de Código:** Utiliza spaces o tabs según como en el resto del proyecto.
- **JavaDoc:** Añade JavaDoc en todos los métodos y clases públicas, además de las clases o métodos privados más utilizados.
- **Nombres de Clases y Métodos:** Usa nombres descriptivos y sigue la convención de CamelCase para clases y camelCase para métodos y variables; la estructura de los nombres de clase tiene que ser primero su el *general* y luego el *específico* (ejemplo: ControllerModelo).
- **Estructura de Paquetes:** Organiza el código en paquetes según la funcionalidad y asegúrate de que cada clase esté en el paquete correspondiente (No modificar la estructura de paquetes del proyecto).
- **Limpieza de Código:** No dejes código comentado o en desuso, asegurate de que no haya errores de estilo y elimina comentarios de plantillas.
- **Control de Errores:** Maneja todas las excepciones correctamente y usa logs (logback) para reportar errores, en lugar de System.out.println.

```
/**
 * Realiza una operación específica en el proyecto.
*
* @param parametro Un parámetro necesario para el método.
* @return Resultado de la operación.
* @throws MiExcepcion si ocurre un error durante la operación.
*/
public Resultado miMetodo(Tipo parametro) throws MiExcepcion {
    // Implementación
}
```

## Pruebas

Este proyecto utiliza JUnit para las pruebas unitarias. Asegúrate de que tus cambios pasen todas las pruebas antes de enviar un Pull Request. Si agregas una nueva funcionalidad, incluye pruebas unitarias para cubrirla.

Para ejecutar las pruebas, utiliza:

```
ant test
```

o

En netbeans presiona *click derecho* y ejecutar Test (Alt + F6).

### Especificaciones de pruebas

- **Cobertura:** Asegúrate de que la cobertura de las pruebas sea adecuada, especialmente en los métodos públicos.
- **Nombres Descriptivos:** Nombra los métodos de prueba de forma descriptiva para reflejar la funcionalidad que están validando.
- **Organización:** Organiza las clases de prueba siguiendo la estructura de paquetes del proyecto.

## Documentación

Documenta cada cambio de funcionalidad y asegúrate de actualizar los archivos README o cualquier otro archivo relevante si la funcionalidad cambia. Toda nueva clase pública debe tener su JavaDoc.

Para generar el JavaDoc completo del proyecto (en NetBeans con Ant):

1. Navega a la carpeta del proyecto.
2. Ejecuta:

   ```
   ant javadoc
   ```

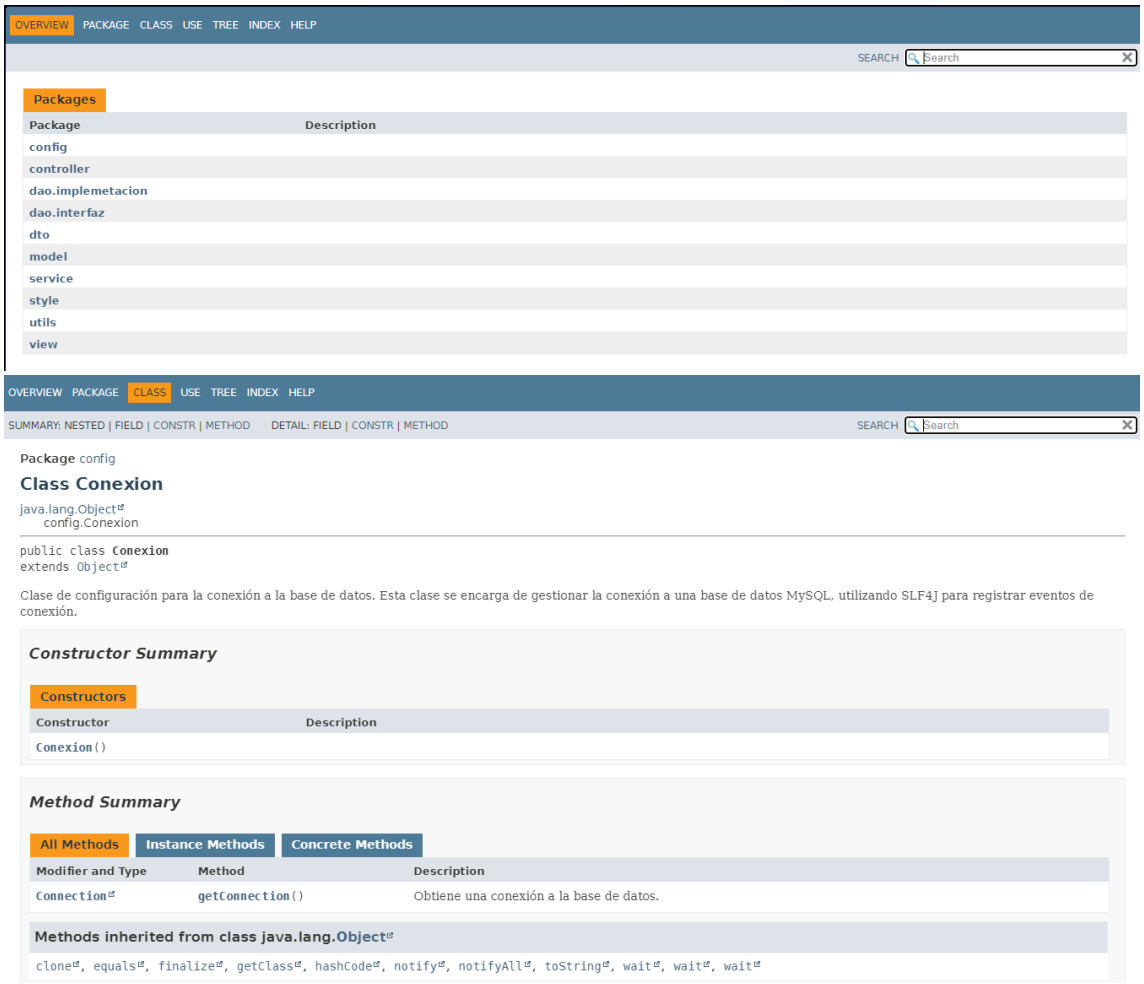
o

Presiona *click derecho* en la carpeta del proyecto y ejecutar JavaDoc.

Los archivos generados estarán en la carpeta ***dist/javadoc***.

**Figura 68**

Aplicación de Javadoc para la documentación en las clases del proyecto de sistema de control de stock Desarrollado en Java



Package dao.implemetacion

# Class DAOUsuarioImpl

java.lang.Object
  dao.implemetacion.DAOUsuarioImpl

**All Implemented Interfaces:**

DAOUsuario

---

```
public class DAOUsuarioImpl
extends Object
implements DAOUsuario
```

Implementación de la interfaz DAOUsuario que gestiona las operaciones de acceso a datos para los usuarios. Esta clase interactúa con la base de datos para realizar operaciones CRUD (Create, Read, Update, Delete) sobre la tabla de usuarios.

## Constructor Summary

Constructors

| Constructor | Description |
|---|---|
| DAOUsuarioImpl() | Constructor de la clase DAOUsuarioImpl. |

## Method Summary

All Methods | Instance Methods | Concrete Methods

| Modifier and Type | Method | Description |
|---|---|---|
| List<DTOUsuario> | buscarUsuarioPorCriterios(String nombre, String apellido, String documento, String correo) | Busca usuarios en la base de datos según los criterios proporcionados. |

## Constructor Details

### DAOUsuarioImpl

```
public DAOUsuarioImpl()
               throws SQLException
```

Constructor de la clase DAOUsuarioImpl. Establece una conexión a la base de datos mediante la clase Conexion.

**Throws:**

SQLException - Si ocurre un error al establecer la conexión con la base de datos.

## Method Details

### login

```
public boolean login(String usuario,
                     String contrasena,
                     int tipoUsuario)
              throws SQLException
```

Inicia sesión en el sistema verificando las credenciales del usuario.

**Specified by:**

login in interface DAOUsuario

**Parameters:**

usuario - el nombre de usuario

contrasena - la contraseña del usuario

Package controller

# Class ControllerUsuario

java.lang.Object
controller.ControllerUsuario

public class **ControllerUsuario**
extends Object

La clase `ControllerUsuario` es el controlador que gestiona las operaciones relacionadas con los usuarios, incluyendo login, registro, edición y exportación de datos. Se comunica con `ServiceUsuario` para ejecutar la lógica de negocio.

## Constructor Summary

Constructors

| Constructor | Description |
|---|---|
| `ControllerUsuario()` | Constructor que inicializa el controlador de usuario y crea una instancia de `ServiceUsuario`. |

## Method Summary

All Methods | Instance Methods | Concrete Methods

| Modifier and Type | Method | Description |
|---|---|---|
| `List<DTOUsuario>` | `buscarUsuarioPorCriterios(String nombre, String apellido, String documento, String correo)` | Busca usuarios según los criterios de búsqueda proporcionados. |
| boolean | `editarUsuario(String nombre, String apellido, String documento, String direccion, String telefono, String correo, int idTipoUsuario, String username, String password)` | Edita la información de un usuario existente. |

## Method Details

### login

```
public boolean login(String usuario,
                     String contrasena,
                     int tipoUsuario)
              throws SQLException
```

Verifica el inicio de sesión del usuario mediante el servicio.

**Parameters:**

`usuario` - El nombre de usuario.

`contrasena` - La contraseña del usuario.

`tipoUsuario` - El tipo de usuario administrador=1 , empleado = 2

**Returns:**

`true` si las credenciales son correctas, `false` en caso contrario.

**Throws:**

`SQLException` - Si ocurre un error al acceder a la base de datos.

**registrarUsuario**

```
public boolean registrarUsuario(String nombre,
                                String apellido,
                                String documento,
                                String direccion,
                                String telefono,
                                String correo,
                                int idTipoUsuario,
                                String username,
                                String password)
                         throws SQLException
```

Registra un nuevo usuario en el sistema.

**Parameters:**

`nombre` - El nombre del usuario.

`apellido` - El apellido del usuario.

`documento` - El documento de identificación del usuario.

`direccion` - La dirección del usuario.

`telefono` - El número de teléfono del usuario.

`correo` - El correo electrónico del usuario.

`idTipoUsuario` - El ID del tipo de usuario administrador=1 , empleado = 2

`username` - El nombre de usuario.

`password` - La contraseña del usuario.

**Returns:**

`true` si el registro fue exitoso, `false` en caso contrario.

**buscarUsuarioPorCriterios**

```
public List<DTOUsuario> buscarUsuarioPorCriterios(String nombre,
                                                  String apellido,
                                                  String documento,
                                                  String correo)
                                           throws SQLException
```

Busca usuarios según los criterios de búsqueda proporcionados.

**Parameters:**

`nombre` - El nombre del usuario (puede ser parcial).

`apellido` - El apellido del usuario (puede ser parcial).

`documento` - El documento del usuario (puede ser parcial).

`correo` - El correo del usuario (puede ser parcial).

**Returns:**

Una lista de usuarios que coinciden con los criterios de búsqueda.

**Throws:**

`SQLException` - Si ocurre un error al realizar la consulta en la base de datos.

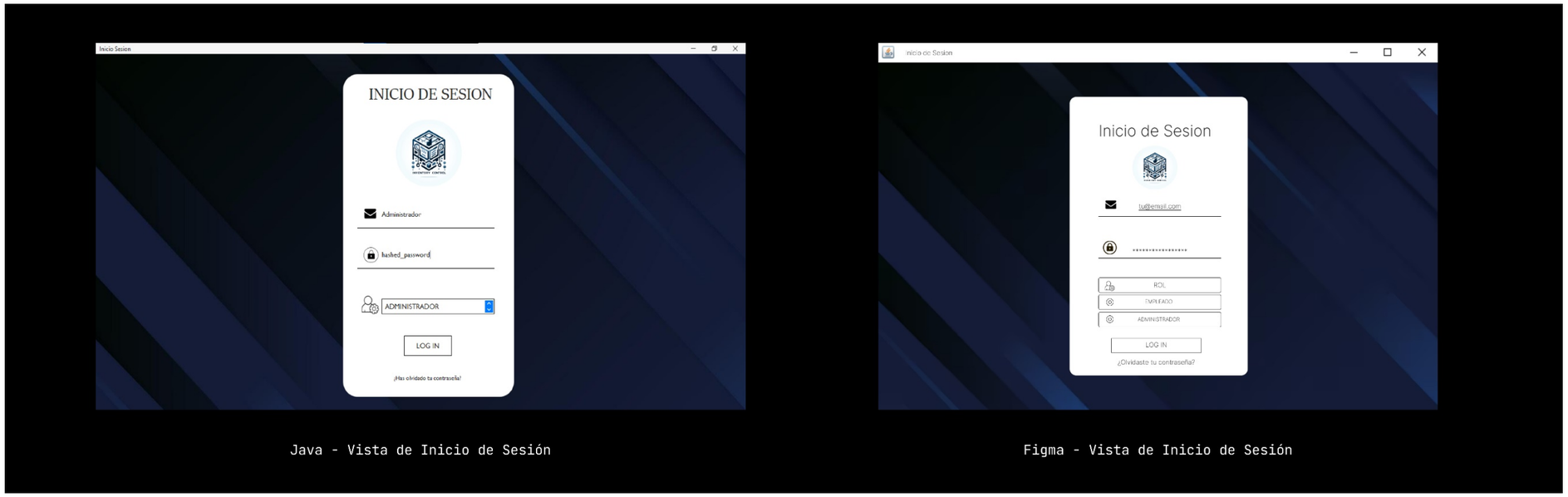
## 6. Comparación de implementación con el diseño

Link Interactividad en Figma:

https://www.figma.com/proto/xrxaX3t2tCiKt8jKjrFx4e/Integrador-Control-De-Stock-Prototipos?node-id=1-1292&node-type=canvas&t=9Iii9MjhVbHvFWin-0&scaling=scale-down&content-scaling=fixed&page-id=0%3A1&starting-point-node-id=15%3A1392

**Figura 69**

**Comparación de la implementación de las vistas de inicio de sesión del proyecto del sistema de control de stock en Java y Figma**




**Comparación de la implementación de las vistas de menú principal del proyecto del sistema de control de stock en Java y Figma**

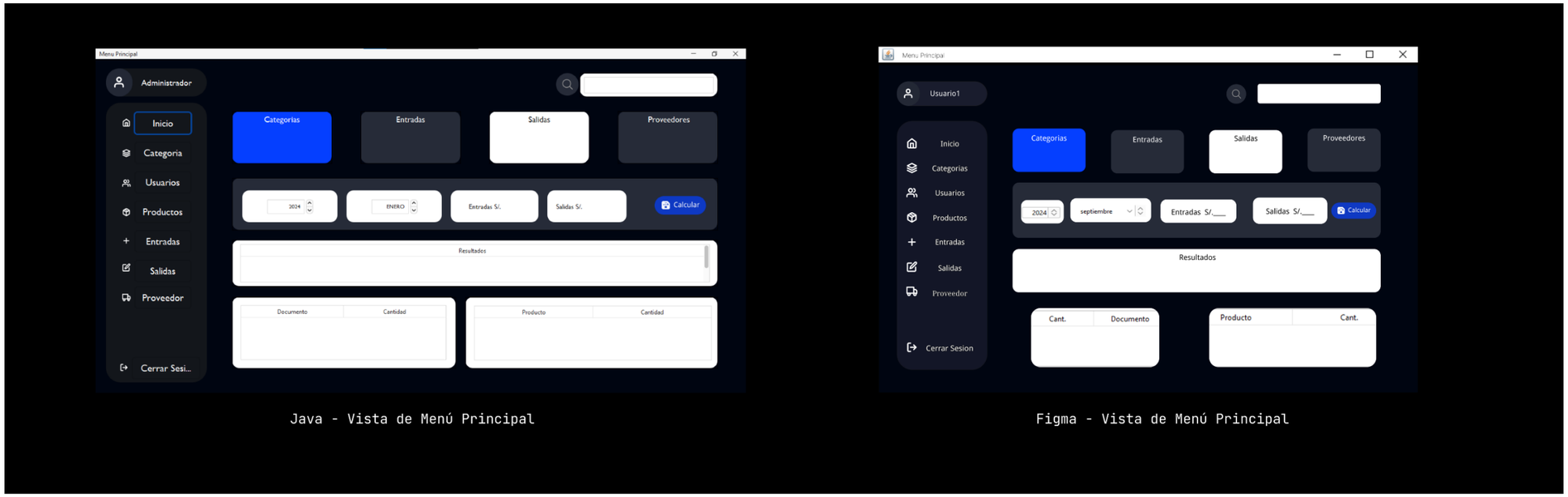
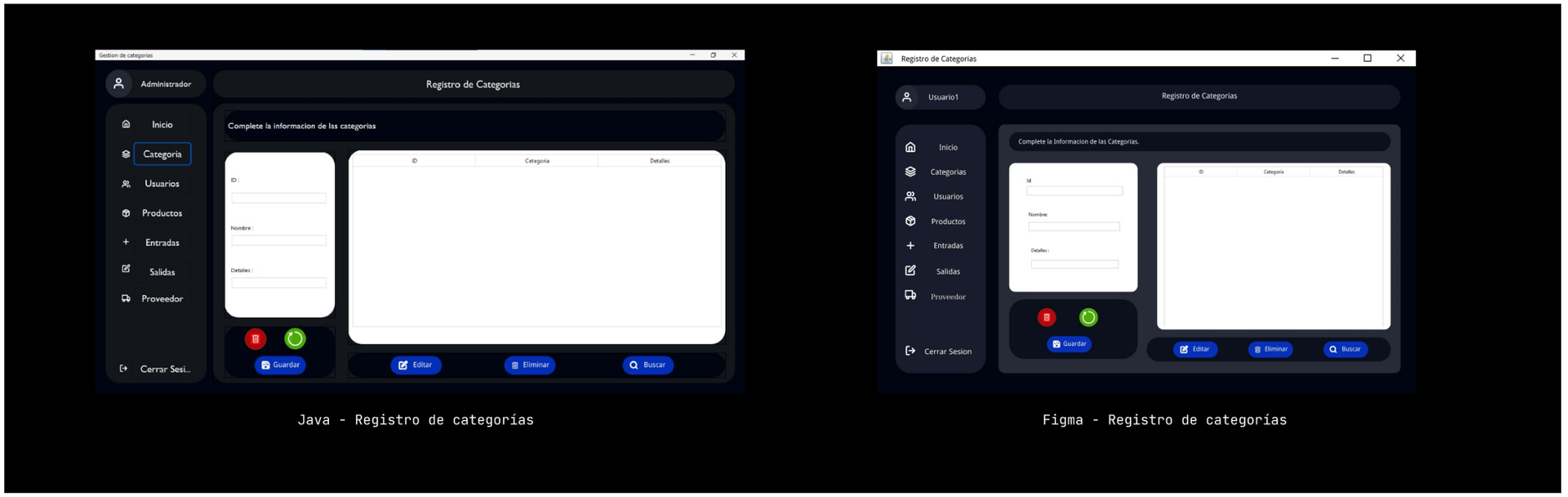


**Figura 71**

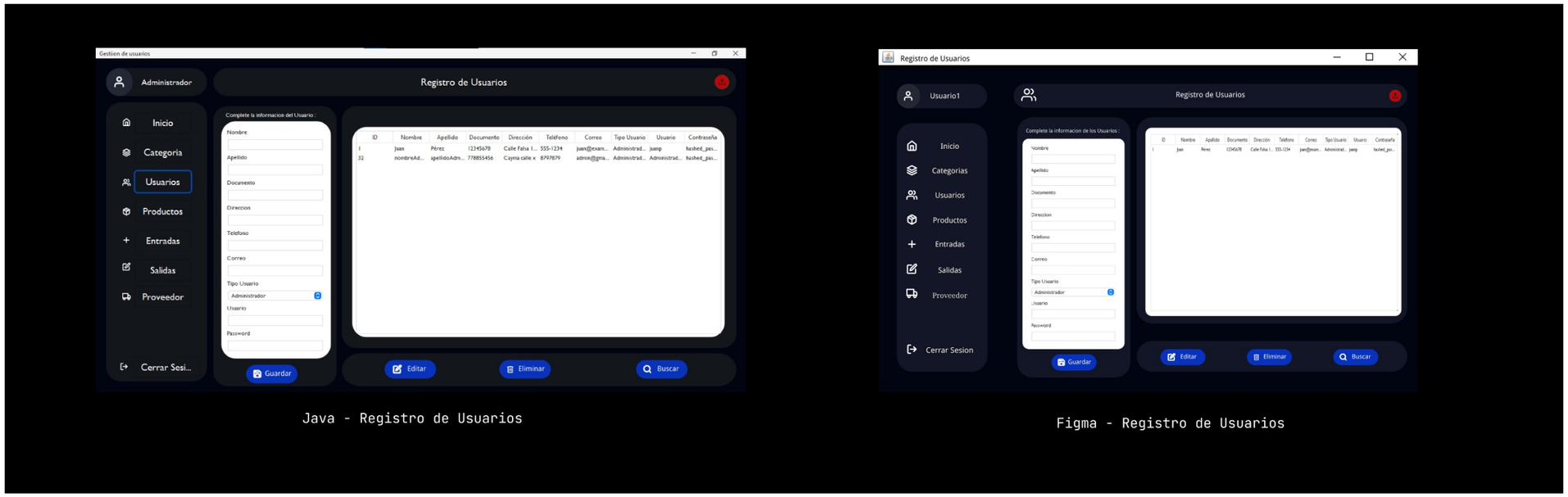
**Comparación de la implementación de las vistas de registro de categorías del proyecto del sistema de control de stock en Java y Figma**




Java - Registro de categorías

Figma - Registro de categorías

**Comparación de la implementación de las vistas de registro de usuarios del proyecto del sistema de control de stock en Java y Figma**




Java - Registro de Usuarios

Figma - Registro de Usuarios

**Figura 73**

**Comparación de la implementación de las vistas de registro de productos del proyecto del sistema de control de stock en Java y Figma**