

## Taller de los principios SOLID



Universidad  
del Cauca

Vigilada Mineducación

### Laboratorio de Ingeniería de Software II

Presentado por:

Fabian Eduardo Guevara Muelas

Karold Dirley Delgado Arciniegas

Profesor:

Julio Ariel Hurtado Alegria

*Universidad del Cauca*

Facultad de Electrónica y Telecomunicaciones

Ingeniería de sistemas

Popayán, 30 de agosto 2024

## **ÍNDICE DE CONTENIDO**

### **1. Evidencia de ejecución (parte I)**

#### 1.1. Introducción

1.1.1. Breve descripción del sistema

1.1.2. Objetivo del documento

1.1.3. Configuración del entorno

#### 1.2. Ejecución del código sobre los principios SOLID

##### 1.2.1. Principio de responsabilidad única (SRP)

1.2.1.1. SRP con síntoma

1.2.1.2. SRP sin síntoma

##### 1.2.2 Principio abierto cerrado (OCP)

1.2.2.1. OCP con síntoma

1.2.2.2. OCP sin síntoma

##### 1.2.3. Principio de sustitución de Liskov (LSP)

1.2.3.1. LSP con síntoma

1.2.3.2. LSP sin síntoma

##### 1.2.4. Principio de segregación de interfaces (ISP)

1.2.4.1. ISP con síntoma

1.2.4.2 ISP sin síntoma

##### 1.2.5. Principio de inversión de dependencias (DIP)

1.2.5.1. DIP con síntoma

1.2.5.2. DIP sin síntoma

### **2. Evidencia de ejecución sin refactorizar (parte II)**

#### 2.1. Ejecución de operaciones CRUD

2.1.1. Crear producto

2.1.2. Leer producto

2.1.3. Buscar producto por ID

2.1.4. Actualizar producto

2.1.5. Eliminar producto

#### 2.2. Conclusiones

2.2.1. Resumen del funcionamiento del sistema

2.2.2. Áreas de mejora identificadas durante las pruebas

### **3. Evidencia de ejecución (parte II)**

3.1. Búsqueda de un producto por ID

3.2. Búsqueda de un producto por nombre

3.3. Acerca de la refactorización

3.4. Conclusiones

#### 3.4.1. Resumen del funcionamiento del sistema

### **4. Tabla de análisis**

- 4.1. Diagrama de módulos
- 4.2. Diagrama de clases
- 4.3. Diagrama de secuencias

## **1. Evidencia de ejecución (parte I)**

### **1.1. Introducción**

#### **1.1.1. Breve descripción del sistema:**

El sistema de gestión de productos es una aplicación Java diseñada para administrar un inventario de productos. Desarrollado por Libardo Pantoja y Julio Hurtado, este sistema permite realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) sobre una base de datos de productos utilizando SQLite como motor de base de datos.

Las principales características del sistema incluyen:

- a. Almacenamiento de información básica de productos, como nombre, descripción y precio.
- b. Capacidad para asociar productos con categorías, ubicaciones y usuarios.
- c. Interfaz de servicio (*ProductService*) que encapsula la lógica de negocio y las operaciones de base de datos.
- d. Manejo de persistencia de datos utilizando JDBC para interactuar con una base de datos SQLite.
- e. Funcionalidades para buscar productos por ID, listar todos los productos, y realizar operaciones de creación, actualización y eliminación de productos.

#### **1.1.2. Objetivo del documento**

El objetivo de este documento es proporcionar una evidencia clara y detallada de la ejecución del sistema de gestión de productos. Este documento servirá como una guía comprensiva para entender el funcionamiento práctico del sistema, validar su implementación y proporcionar una base para futuras iteraciones o mejoras del proyecto.

#### **1.1.3. Configuración del entorno**

El entorno en el que se está desarrollando esta práctica es en la última versión de Java, la cual es Java 8, con un entorno de desarrollo integrado en la versión Apache Netbeans IDE 22. Además, se maneja una base de datos de productos utilizando SQLite JDBC como motor de base de datos, versión 3.32.3.2. Tal como se aprecia en la siguiente imagen:

```
Product Version: Apache NetBeans IDE 22  
Java: 22.0.2; Java HotSpot(TM) 64-Bit Server VM 22.0.2+9-70  
Runtime: Java(TM) SE Runtime Environment 22.0.2+9-70  
System: Windows 11 version 10.0 running on amd64; UTF-8; es_CO (nb)
```

*Imagen 1: descriptiva del entorno de desarrollo utilizado.*

## 1.2. Ejecución del código sobre los principios SOLID

### 1.2.1. Principio de responsabilidad única (SRP)

El principio de responsabilidad única, es uno de los cinco principios de diseño que establece que una clase o módulo debe tener una única razón para cambiar, es decir, un solo propósito. En otras palabras, una clase debe centrarse únicamente en una tarea o funcionalidad específica y no estar encargada de múltiples cosas.

#### 1.2.1.1. SRP con síntoma

Como se aprecia en el siguiente ejemplo (imagen 2), el código no cumple con este principio, ya que la clase *service* tiene múltiples responsabilidades:

- **Lógica de negocios:** El método *calculateProductTax()* realiza calculo de impuestos sobre un producto.
- **Acceso a la base de datos:** Los métodos *saveProdcut()* y *listProducts()* gestionan el acceso a los datos y la interacción con la base de datos.
- **Inicialización de la base de datos:** El método *initDatabase()* establece la conexión y crea las tablas necesarias en la base de datos.

```

public class Service {

    private Connection conn;

    public Service() {
        initDatabase();
    }

    // Razon 1: A futuro podria cambiar la lógica para calcular el impuesto

    /** Una lógica de negocio sencilla: validad datos y calcular un impuesto del ...7 lines */
    public double calculateProductTax(Product product) {...10 lines }

    // Razon 2: A futuro podriamos cambiar el motor de base de datosm usar un
    // ORM, o cambiar la estructura de la base de datos
    /** Lógica de acceso a datos ...6 lines */
    public boolean saveProduct(Product newProduct) {...22 lines }

    public List<Product> listProducts() {...22 lines }

    private void initDatabase() {...21 lines }

}

```

*Imagen 2 del código de la clase Service sin aplicar el principio SRP*

```

--- exec:3.1.0:exec (default-cli) @ 1_UnicaRazon_Symptom ---
Ingrese el codigo
123
Ingrese el nombre del producto
Televisor
Ingrese el valor|
2000
Ingrese el codigo
321
Ingrese el nombre del producto
Nevera
Ingrese el valor
1800
Product{id=123, name=Televisor, price=2000.0}
Product{id=321, name=Nevera, price=1800.0}

-----
BUILD SUCCESS
-----

Total time:  34.085 s
Finished at: 2024-09-08T14:43:17-06:00
-----

```

*Imagen de la salida del programa al ejecutarlo*

### 1.2.1.2. SRP sin síntoma

```
public class Service {  
  
    private ProductRepository repository;  
  
    public Service() {  
        repository = new ProductRepository();  
    }  
    /**  
     * Maneja una lógica de negocio simple para calcular un impuesto del producto  
     * @param product producto a calcular impuesto  
     * @return  
     */  
    public double calculateProductTax(Product product) {...10 lines }  
    /**  
     * Graba el producto en la base de datos  
     * @param newProduct producto a ser grabado  
     * @return true si lo graba en la bd, false en caso contrario  
     */  
    public boolean saveProduct(Product newProduct) {...11 lines }  
    /**  
     * Listar productos  
     * @return Lista de Productos  
     */  
    public List<Product> listProducts() {...5 lines }  
  
}
```

*Imagen de la clase Service aplicando el principio SRP*

```

--- exec:3.1.0:exec (default-cli) @ 1_UnicaRazon ---
Ingrese el codigo:
345
Ingrese el nombre del producto:
Televisor
Ingrese el valor:
300000
Ingrese el codigo:
654
Ingrese el nombre del producto:
Lavadora
Ingrese el valor:
450000
Product{id=1, name=TV samgung RTR-123, price=2500000.0}
Product{id=2, name=Nevera LG FGET-233, price=2500000.0}
Product{id=10, name=Leche, price=34.0}
Product{id=20, name=Cal, price=34500.0}
Product{id=123, name=Televisor, price=2000.0}
Product{id=321, name=Nevera, price=1800.0}
Product{id=345, name=Televisor, price=300000.0}
Product{id=654, name=Lavadora, price=450000.0}

-----
BUILD SUCCESS
-----

Total time:  32.374 s
Finished at: 2024-09-08T14:50:24-06:00

```

*Imagen de la salida del programa al ejecutarlo*

### 1.2.2. Principio abierto/cerrado (OCP)

El principio OCP establece que un módulo, clase o función debe estar abierto para la extensión, pero cerrado para la modificación.

#### 1.2.2.1. OCP con síntoma

El siguiente código no cumple con el principio OCP dado que la clase *Store* está diseñada de tal forma que cualquier cambio en los países o en la lógica de cálculo de costos de entrega requerirá modificar el código de la misma.

- Si se añade un nuevo país, habría que modificar el *switch* dentro de *calculateDelivery()*.
- La lógica de entrega está mezclada dentro de una sola clase, lo que la hace difícil de modificar y extender sin tocar el código existente.



```

public double calculateDeliveryCost(Order order) {

    if (order == null) {
        return -1;
    }

    double result = 0;
    switch (order.getCountry()) {
        case COLOMBIA:
            result = order.getTotal() * 0.01;
            if (order.getWeight() <= 2) {
                result += 9900;
            } else {
                result += order.getWeight() * 5000;
            }

            break;
        case MEXICO:
            result = 98;
            break;
        case PERU:
            result = 0;
    }
}

```

*Imagen del código sin aplicar el principio OCP*

```

--- exec:3.1.0:exec (default-cli) @ 2_ExtiendeNoModifiques_Symptom ---
Order One cost:98.0 pesos mexicanos
Order Two cost: 5.0000000001E8pesos colombianos
-----
BUILD SUCCESS
-----
Total time:  1.063 s
Finished at: 2024-09-08T15:07:17-06:00

```

*Imagen de la salida del programa al ejecutarlo*

### 1.2.2.2. OCP sin síntoma

```
public class Store {  
    /**  
     * Calcular costo de entrega  
     * @param order orden sobre la cual se calcula el costo de entrega  
     * @return costo de entrega  
     */  
    public double calculateDeliveryCost(Order order) {  
  
        if (order == null) {  
            return -1;  
        }  
  
        // La fábrica devuelve una instancia de la jerarquía IDelivery  
        IDelivery delivery = Factory.getInstance().getDelivery(order.getCountry());  
        double result = delivery.calculateCost(order);  
  
        return result;  
    }  
}
```

*Imagen del código aplicando el principio OCP*

```
--- exec:3.1.0:exec (default-cli) @ 2_ExtiendeNoModifique ---  
Order One cost:98.0 pesos mexicanos  
Order Two cost: 5.000000001E8 pesos colombianos  
Order Three cost: 0.0 nuevos soles  
-----  
BUILD SUCCESS  
-----  
Total time: 1.244 s  
Finished at: 2024-09-08T15:18:52-06:00  
-----
```

*Imagen de la salida del programa al ejecutarlo*

### 1.2.3. Principio de sustitución de Liskov (LSP)

El principio LSP es el tercero de los principios SOLID, establece que los objetos de una clase derivada deben poder reemplazar a los objetos de su clase base sin alterar el correcto funcionamiento del programa.

#### 1.2.3.1. LSP con síntoma

El siguiente código no cumple con el principio LSP debido a que la subclase *SpecializedTruck* altera el comportamiento esperado de la clase *Truck*.

- En la clase *Truck*, se establece que el método *addTrip()* **no debe afectar el**

- odómetro** (el comentario indica explícitamente que hay otro componente responsable de ello). Sin embargo, en la subclase *SpecializedTruck*, el método *addTrip()* **modifica el odómetro**, lo que **rompe el contrato** definido por la clase base.
- Este comportamiento rompe el LSP porque un *SpecializedTruck* ya no puede ser sustituido por un *Truck* sin alterar el comportamiento esperado (modificar el odómetro es algo que no debería ocurrir según el diseño de *Truck*).

```
public class SpecializedTruck extends Truck {  
  
    public SpecializedTruck(String plateNumber, double odometer) {  
        super(plateNumber, odometer);  
    }  
    @Override  
    public void addTrip(Trip newTrip) {  
        //Update odometer.  
        odometer += newTrip.getDistance();  
        super.addTrip(newTrip);  
    }  
}
```

*Imagen de la subclase SpecializedTruck que viola el principio LSP*

```
--- exec:3.1.0:exec (default-cli) @ 3_TalPadreTalHijo_Symptom ---  
java.lang.Exception: Se incumple invariante  
    at co.unicauca.solid.liskov.domain.Truck.addTrip(Truck.java:51)  
    at co.unicauca.solid.liskov.domain.SpecializedTruck.addTrip(SpecializedTruck.java:16)  
    at co.unicauca.solid.liskov.presentation.ClienteMain.main(ClienteMain.java:16)  
Odometer has been modified. LSK violation  
-----  
BUILD SUCCESS  
-----  
Total time: 1.660 s  
Finished at: 2024-09-08T15:22:36-06:00
```

*Imagen de la salida del programa al ejecutarlo*

### 1.2.3.2. LSP sin síntoma

```
public class SpecializedTruck extends Truck {
    private double totalDistance;

    public SpecializedTruck(String plateNumber, double odometer) {
        super(plateNumber, odometer);
        totalDistance = 0;
    }
    @Override
    public void addTrip(Trip newTrip) { ...6 lines }

    /**
     * @return the totalDistance
     */
    public double getTotalDistance() {
        return totalDistance;
    }

    /**
     * @param totalDistance the totalDistance to set
     */
    public void setTotalDistance(double totalDistance) {
        this.totalDistance = totalDistance;
    }
}
```

*Imagen del código de la clase SpecializedTruck aplicando el principio LSP*

```
--- exec:3.1.0:exec (default-cli) @ 3_TalPAdreTalHijoSol ---
Odometer is correct.
-----
BUILD SUCCESS
-----
Total time: 1.099 s
Finished at: 2024-09-08T15:53:52-06:00
```

*Imagen de la salida del programa al ejecutarlo*

### 1.2.4. Principio de segregación de interfaces (ISP)

El principio ISP es el cuarto de los principios SOLID, establece que **una clase no debería estar forzada a depender de interfaces que no utiliza**. En lugar de tener una interfaz grande y genérica, es mejor dividirla en interfaces más pequeñas y específicas para que las clases solo implementen lo que realmente necesitan.

#### 1.2.4.1. ISP con síntoma

El siguiente código no cumple con el principio ISP, se debe a que la interfaz *IRepository* es una **interfaz gorda**, que agrupa métodos relacionados con usuarios, proyectos y tareas en una sola interfaz, lo que la hace más difícil de mantener y menos flexible.

- Demasiadas responsabilidades en una sola interfaz: La interfaz *IRepository* tiene métodos para gestionar usuarios (*createUser*, *listUsers*, *deleteUser*), proyectos (*createProject*, *listProjectsByUser*), y tareas (*createTask*, *listCompletedTaskByUser*, *deleteTask*). Esto significa que una clase que implemente *IRepository* deberá proporcionar implementaciones para todos estos métodos, incluso si no necesita manipular usuarios, proyectos o tareas en algunos casos específicos.
- Volatilidad: Como se menciona en el comentario, esta interfaz es volátil porque puede cambiar con el tiempo. Si cambias la forma en que se gestionan los usuarios, proyectos o tareas, cualquier clase que implemente *IRepository* se verá afectada, incluso si no utiliza todos los métodos de esa interfaz.

```
public interface IRepository {  
  
    //User methods  
    void createUser(User user);  
  
    List<User> listUsers();  
  
    void deleteUser(User user);  
  
    //Project methods  
    void createProject(Project project);  
  
    List<Project> listProjectsByUser(int userId);  
  
    //Task methods  
    void createTask(Task task);  
  
    List<Task> listCompletedTasksByUser(int userId);  
  
    void deleteTask(Task task);  
}
```

*Imagen del código de la interfaz IRepository sin aplicar el principio ISP*

#### 1.2.4.2. ISP sin síntoma

```
public interface IProjectRepository {  
  
    //Project methods  
    void createProject(Project project);  
  
    List<Project> listProjectsByUser(int userId);  
  
}
```

*Imagen del código de la interfaz IRepository aplicando el principio ISP*

#### 1.2.5. Principio de inversión de dependencias (DIP)

El principio DIP, es el último de los principios SOLID, el cual establece dos reglas fundamentales:

- **Los módulos de alto nivel no deben depender de módulos de bajo nivel.** Ambos deben depender de abstracciones (interfaces o clases abstractas).
- **Las abstracciones no deben depender de los detalles.** Los detalles deben depender de las abstracciones.

##### 1.2.5.1. DIP con síntoma

El siguiente código no cumple con el principio DIP porque la clase *Service* depende directamente de una implementación concreta, *ProductRepository*, en lugar de una abstracción. Esto crea un fuerte acoplamiento entre la lógica de negocio (*Service*) y los detalles de acceso a datos (*ProductRepository*), lo que dificulta el mantenimiento y la flexibilidad del sistema.

```

public class Service {

    // Aqui hay una dependencia directa con un módulo inferior
    // Se debería reemplazar la dependencia con una abstracción
    private ProductRepository repository;

    public Service() {
        repository = new ProductRepository();
    }

    public double calculateProductTax(Product product) {

        //Validate product.
        if (product == null) { ...3 lines }
        double TAX = 0.19d;
        double productTax = product.getPrice() * TAX;
        return productTax;
    }

    public boolean saveProduct(Product newProduct) { ...11 lines }

    public List<Product> listProducts() { ...6 lines }

}

```

*Imagen de la clase Service sin aplicar el principio DIP*

```

--- exec:3.1.0:exec (default-cli) @ 5_AbstractoBueno_Symptom ---
sept 08, 2024 4:24:03 P.M. co.unicauca.solid.dip.domain.access.ProductRepository saveProduct
SEVERE: null
] org.sqlite.SQLiteException: [SQLITE_CONSTRAINT_PRIMARYKEY] A PRIMARY KEY constraint failed (UNIQUE constraint failed: Product.ProductId)
  at org.sqlite.core.DB.newSQLException(DB.java:1012)
  at org.sqlite.core.DB.newSQLException(DB.java:1024)
  at org.sqlite.core.DB.execute(DB.java:863)
  at org.sqlite.core.DB.executeUpdate(DB.java:904)
  at org.sqlite.jdbc3.JDBC3PreparedStatement.executeUpdate(JDBC3PreparedStatement.java:98)
  at co.unicauca.solid.dip.domain.access.ProductRepository.saveProduct(ProductRepository.java:44)
  at co.unicauca.solid.dip.domain.service.Service.saveProduct(Service.java:40)
  at co.unicauca.solid.dip.domain.main.ClientMain.main(ClientMain.java:22)

sept 08, 2024 4:24:03 P.M. co.unicauca.solid.dip.domain.access.ProductRepository saveProduct
SEVERE: null
] org.sqlite.SQLiteException: [SQLITE_CONSTRAINT_PRIMARYKEY] A PRIMARY KEY constraint failed (UNIQUE constraint failed: Product.ProductId)
  at org.sqlite.core.DB.newSQLException(DB.java:1012)
  at org.sqlite.core.DB.newSQLException(DB.java:1024)
  at org.sqlite.core.DB.execute(DB.java:863)
  at org.sqlite.core.DB.executeUpdate(DB.java:904)
  at org.sqlite.jdbc3.JDBC3PreparedStatement.executeUpdate(JDBC3PreparedStatement.java:98)
  at co.unicauca.solid.dip.domain.access.ProductRepository.saveProduct(ProductRepository.java:44)
  at co.unicauca.solid.dip.domain.service.Service.saveProduct(Service.java:40)
  at co.unicauca.solid.dip.domain.main.ClientMain.main(ClientMain.java:25)

co.unicauca.solid.dip.domain.Product@4cdbe50f
co.unicauca.solid.dip.domain.Product@66d33a
-----
BUILD SUCCESS

```

*Imagen de la salida del programa al ejecutarlo*

### 1.2.5.2. DIP sin síntoma

```
import co.unicauca.solid.dip.domain.Product;
import java.util.List;

public interface IProductRepository {
    boolean saveProduct(Product product);
    List<Product> listProducts();
}
```

*Imagen de la interfaz creada que define las operaciones*

```
public class ProductRepository implements IProductRepository {
    private Connection conn;

    public ProductRepository() {
        initDatabase();
    }

    @Override
    public boolean saveProduct(Product product) {...25 lines }

    private boolean productExists(int productId) throws SQLException {...10 lines }

    @Override
    public List<Product> listProducts() {...24 lines }

    private void initDatabase() {...18 lines }

    public void connect() {...12 lines }

    public void disconnect() {...10 lines }
}
```

*Imagen de la clase ProductRepository que implementa la interfaz IProductRepository*



```

public class Service {

    private IProductRepository repository;

    public Service(IProductRepository repository) {...3 lines }

    public double calculateProductTax(Product product) {...10 lines }

    public boolean saveProduct(Product product) {...6 lines }

    public List<Product> listProducts() {...3 lines }

}

```

*Imagen de la clase Service que implementa la interfaz IProductRepository*

```

--- exec:3.1.0:exec (default-cli) @ 5_AbstractoBueno_Symptom ---
Producto 1 guardado exitosamente.
Producto 2 guardado exitosamente.
Producto 1 actualizado exitosamente.

Lista de productos:
Product{productId=1, name='TV Samsung RTR-123 (Updated)', price=2600000.0}
Product{productId=2, name='Nevera LG FGET-233', price=2500000.0}
-----
BUILD SUCCESS
-----
Total time:  2.362 s
Finished at: 2024-09-08T17:31:00-06:00

```

*Imagen de la salida del programa aplicando el principio DIP*

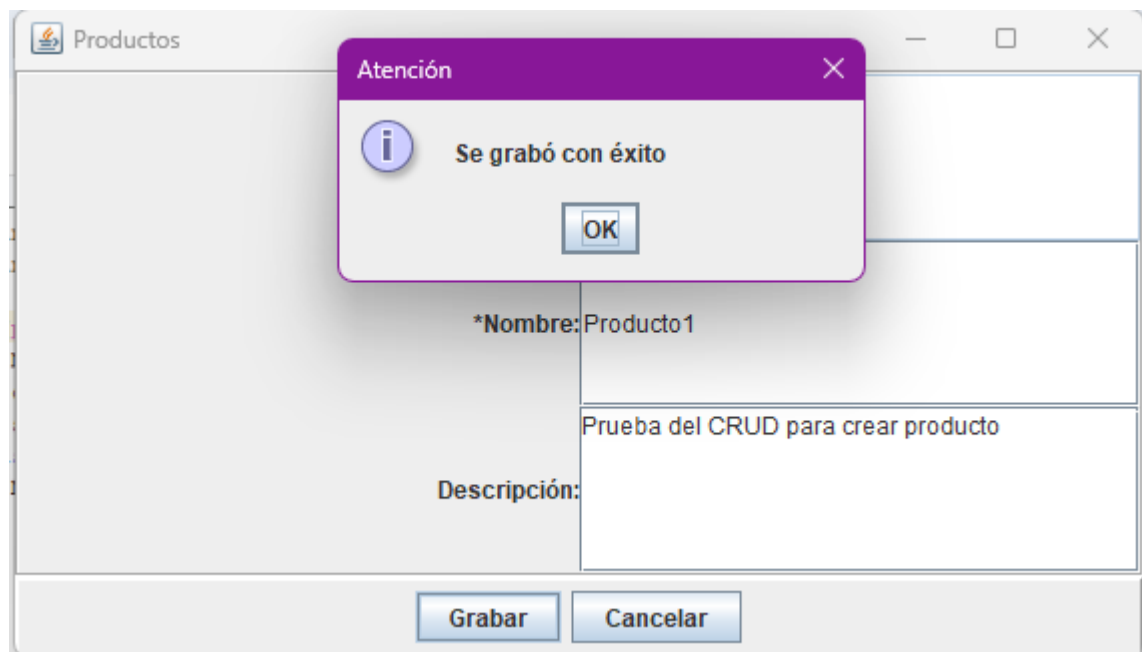
## 2. Evidencia de ejecución sin refactorizar (parte II)

### 2.1. Ejecución de operaciones CRUD

#### 2.1.1. Crear producto

```
private void addProduct() {  
    String name = txtName.getText().trim();  
    String description = txtDescription.getText().trim();  
  
    if (productService.saveProduct(name, description)) {  
        Messages.showMessageDialog("Se grabó con éxito", "Atención");  
        cleanControls();  
        stateInitial();  
    } else {  
        Messages.showMessageDialog("Error al grabar, lo siento mucho", "Atención");  
    }  
}
```

*Imagen 2: código que llama a saveProduct()*



*Imagen 3: resultado de la ejecución (éxito)*

### 2.1.2. Leer producto

```
private void btnSearchAllActionPerformed(java.awt.event.ActionEvent evt) {  
    fillTable(productService.findAllProducts());  
}
```

Imagen 4: código que llama a `findAllProducts()`

Id	Name	Description
1	Producto1	Prueba del CRUD para leer productos
2	Producto2	Otro producto

Imagen 5: resultado de la ejecución (lista de productos)

### 2.1.3. Buscar producto por ID

```
private void txtIdFocusLost(java.awt.event.FocusEvent evt) {  
    if (txtId.getText().trim().equals("")) {  
        return;  
    }  
    Long productId = Long.parseLong(txtId.getText());  
    Product prod = productService.findProductById(productId);  
    if (prod == null) {  
        Messages.showMessageDialog("El identificador del producto no existe", "Error");  
        txtId.setText("");  
        txtId.requestFocus();  
    } else {  
        txtName.setText(prod.getName());  
        txtDescription.setText(prod.getDescription());  
    }  
}
```

Imagen 6: código que llama a `findProductById()`

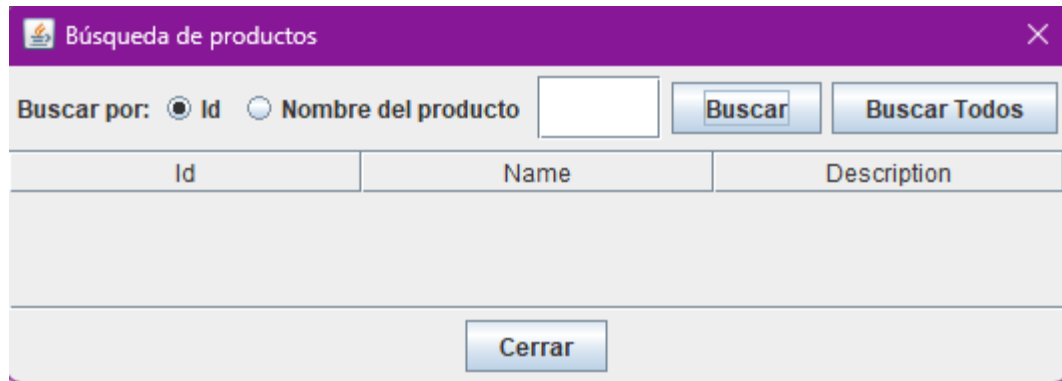


Imagen 7: resultado de la ejecución (detalles del producto - fallido)

#### 2.1.4. Actualizar producto

```
private void editProduct() {
    String id = txtId.getText().trim();
    if (id.equals("")) {
        Messages.showMessageDialog("Debe buscar el producto a editar", "Atención");
        txtId.requestFocus();
        return;
    }
    Long productId = Long.parseLong(id);
    Product prod = new Product();
    prod.setName(txtName.getText().trim());
    prod.setDescription(txtDescription.getText().trim());

    if (productService.editProduct(productId, prod)) {
        Messages.showMessageDialog("Se editó con éxito", "Atención");
        cleanControls();
        stateInitial();
    } else {
        Messages.showMessageDialog("Error al editar, lo siento mucho", "Atención");
    }
}
```

Imagen 8: código que llama a editProduct()

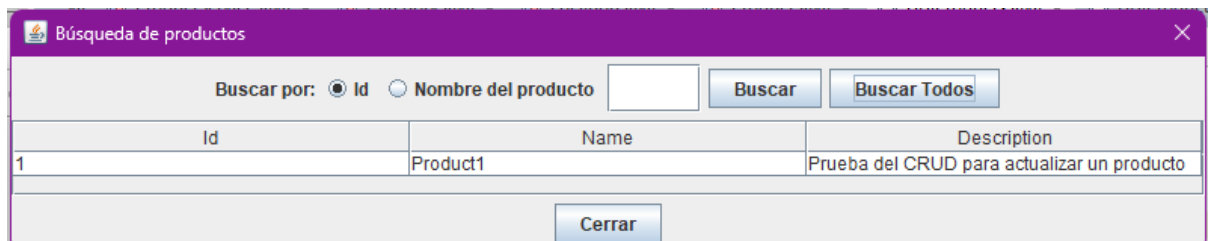


Imagen 9: producto guardado inicialmente

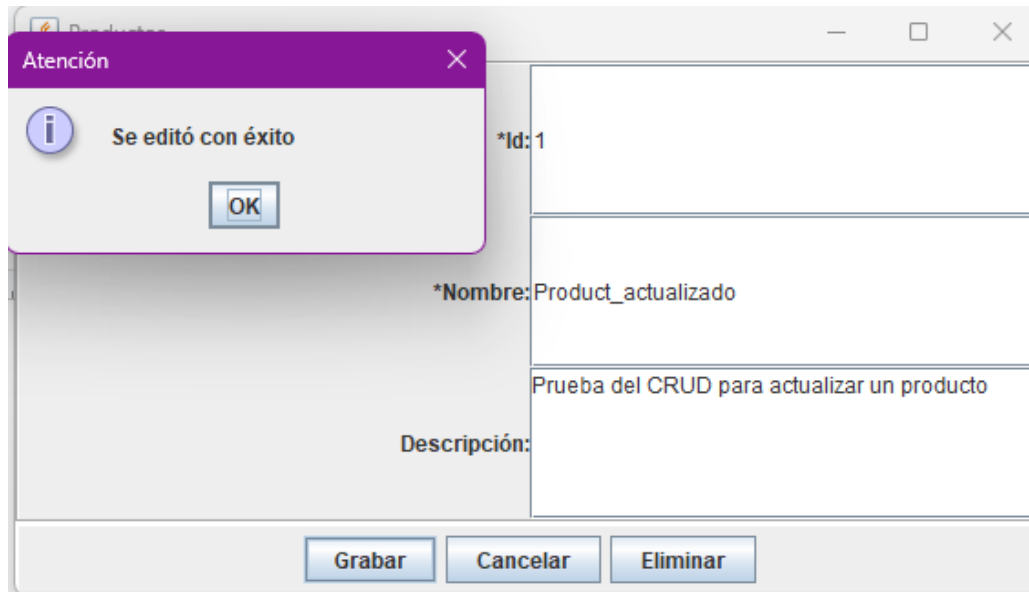


Imagen 10: producto editado con éxito

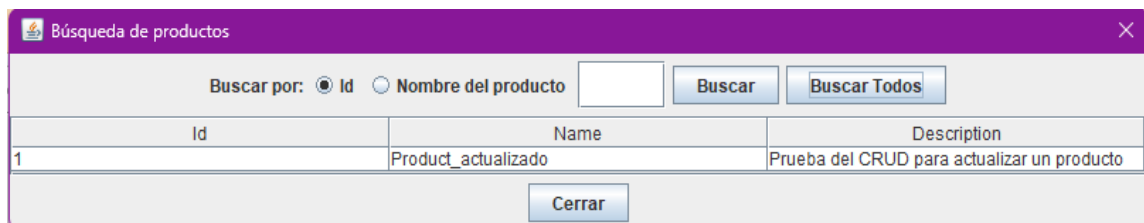
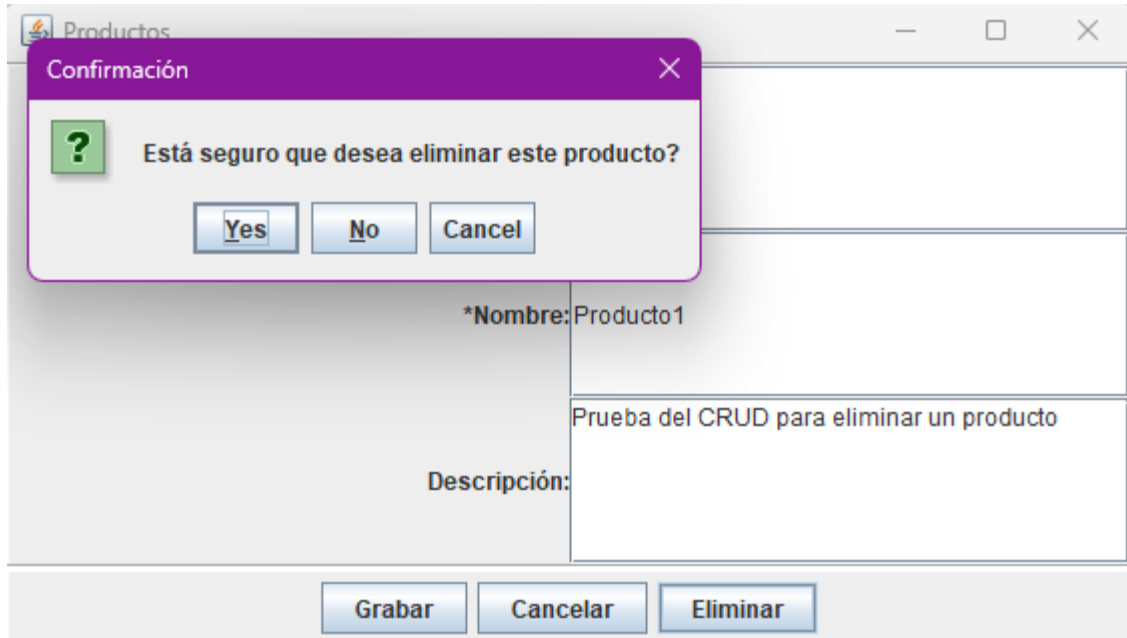


Imagen 11: final del producto actualizado con éxito

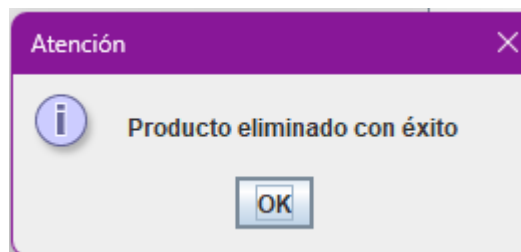
### 2.1.5. Eliminar producto

```
private void btnEliminarActionPerformed(java.awt.event.ActionEvent evt) {
    String id = txtId.getText().trim();
    if (id.equals("")) {
        Messages.showMessageDialog("Debe buscar el producto a eliminar", "Atención");
        txtId.requestFocus();
        return;
    }
    Long productId = Long.parseLong(id);
    if (Messages.showConfirmDialog("Está seguro que desea eliminar este producto?",
        "Confirmación") == JOptionPane.YES_NO_OPTION) {
        if (productService.deleteProduct(productId)) {
            Messages.showMessageDialog("Producto eliminado con éxito", "Atención");
            stateInitial();
            cleanControls();
        }
    }
}
```

Imagen 12: código que llama a deleteProduct()



*Imagen 13: proceso de eliminación del producto*



*Imagen 14: eliminación del producto*

## 2.2. Conclusiones

### 2.2.1. Resumen del funcionamiento del sistema

Tras una evaluación del Sistema de Gestión de Productos, podemos concluir que:

- a. **Funcionalidad CRUD:** El sistema demuestra una implementación efectiva de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para los productos. Cada operación se ejecuta correctamente, manteniendo la integridad de los datos en la base de datos SQLite. Sin embargo, da espacio a mejoras en la lectura de productos.
- b. **Interfaz de servicio:** La clase *ProductService* ofrece una interfaz clara y cohesiva para interactuar con los datos de los productos,

encapsulando adecuadamente la lógica de negocio y las operaciones de base de datos.

- c. **Modelo de datos:** La clase *Product* proporciona una representación adecuada de los productos, incluyendo relaciones con otras entidades como categorías, ubicaciones y usuarios.
- d. **Manejo de errores:** El sistema muestra capacidad para manejar entradas inválidas y situaciones de error, aunque hay margen para mejoras en esta área.

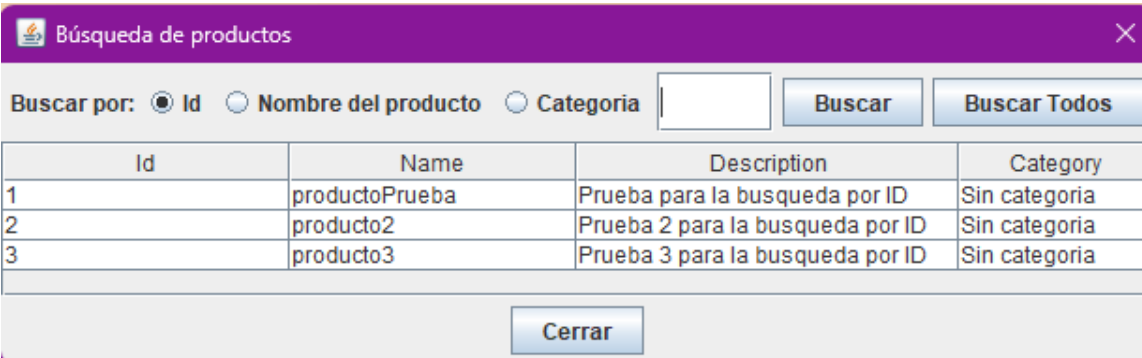
### 2.2.2. Áreas de mejora identificadas durante las pruebas

A pesar del funcionamiento general satisfactorio, se han identificado las siguientes áreas de mejora:

- a. **Separación de responsabilidades:** La clase *ProductService* actualmente maneja tanto la lógica de negocio como las operaciones de base de datos. Es recomendable separar estas responsabilidades en clases distintas para mejorar la modularidad y facilitar el mantenimiento.
- b. **Pruebas unitarias:** Desarrollar un conjunto completo de pruebas unitarias para garantizar la robustez del sistema y facilitar futuras modificaciones.

## 3. Evidencia de ejecución refactorizado (parte II)

### 3.1. Búsqueda de un producto por ID



Id	Name	Description	Category
1	productoPrueba	Prueba para la búsqueda por ID	Sin categoria
2	producto2	Prueba 2 para la búsqueda por ID	Sin categoria
3	producto3	Prueba 3 para la búsqueda por ID	Sin categoria

*Imagen de los productos agregados para la prueba de búsqueda por ID*

The screenshot shows a window titled "Búsqueda de productos" with a search interface. The "Buscar por:" section has three radio buttons: "Id" (selected), "Nombre del producto", and "Categoria". A text input field contains the value "2". To the right are two buttons: "Buscar" and "Buscar Todos". Below this is a table with the following data:

Id	Name	Description	Category
2	producto2	Prueba 2 para la búsqueda por ID	

At the bottom of the window is a "Cerrar" button.

*Imagen del resultado de la búsqueda por ID*

### 3.2. Búsqueda de un producto por nombre

The screenshot shows the same "Búsqueda de productos" window. The "Nombre del producto" radio button is now selected. The text input field contains "Producto3". The "Buscar" and "Buscar Todos" buttons are present. The table below displays a list of three products:

Id	Name	Description	Category
1	producto1	Prueba 1 para la búsqueda por nombre	Sin categoria
2	Producto2	Prueba 2 para la búsqueda por nombre	Sin categoria
3	Producto3	Prueba 3 para la búsqueda por nombre	Sin categoria

A "Cerrar" button is located at the bottom.

*Imagen de los productos agregados para la prueba de búsqueda por nombre*

This screenshot shows the "Búsqueda de productos" window with "Nombre del producto" selected and "Producto3" entered. The "Buscar" button is highlighted. The table below shows only the product matching the search criteria:

Id	Name	Description	Category
3	Producto3	Prueba 3 para la búsqueda por nombre	

The "Cerrar" button is at the bottom.

*Imagen del resultado de búsqueda por nombre del producto*

### 3.3. Acerca de la refactorización

Con el fin de cumplir con los principios SOLID, y habiendo identificado los principios vulnerados (principio de responsabilidad única -SRP-, principio de inversión de dependencias -DIP-), nos enfocamos en realizar lo siguiente:

- Principio de responsabilidad única (SRP): La clase *ProductService* debería



- manejar la lógica de negocio, no la gestión de la base de datos. Separamos esta lógica mediante la implementación de un repositorio para la persistencia de datos.
- Principio de inversión de dependencias (DIP): Para que *ProductService* no dependa directamente de una implementación específica de un repositorio (como una base de datos), usaremos una interfaz para el repositorio, lo que permitirá cambiar su implementación sin modificar la clase *ProductService*.

Pasos realizados:

- Crear una interfaz *ProductRepository* que definirá los métodos CRUD para los productos.
- Crear una implementación de *ProductRepository* que maneje la lógica de persistencia.
- Modificar *ProductService* para depender de la interfaz en lugar de la implementación concreta.

### 3.4. Conclusiones

#### 3.4.1. Resumen del funcionamiento del sistema

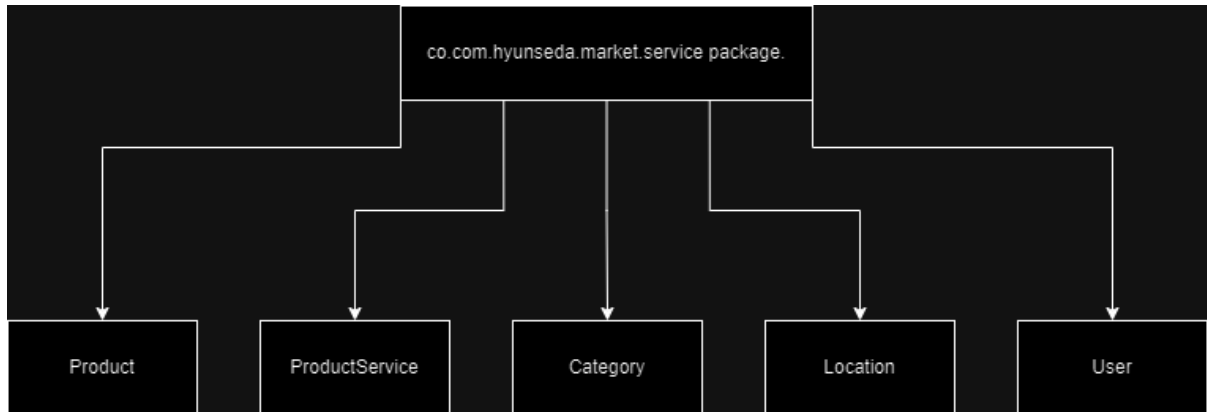
Este enfoque cumple con los principios SOLID al separar las responsabilidades y desacoplar la clase *ProductService* de la implementación específica de la base de datos.

### 4. Tabla de análisis

Violación	Principio	Acción para resolver
<i>ProductService</i> : La clase tiene demasiadas responsabilidades (operaciones de base de datos, lógica empresarial)	Principio de Única Responsabilidad ( <i>Single Responsibility Principle – SRP</i> )	Dividir <i>ProductService</i> en clases separadas: <i>ProductRepository</i> para operaciones de base de datos y <i>ProductService</i> para lógica empresarial
Acceso directo a la base de datos en <i>ProductService</i>	Principio de Inversión de Dependencias ( <i>Dependency Inversion Principle</i> )	Introducir una interfaz de repositorio y utilizar inyección de dependencia
Acoplamiento estrecho entre <i>Product</i> y otras	Alto acoplamiento	Uso de interfaces o clases abstractas para estas

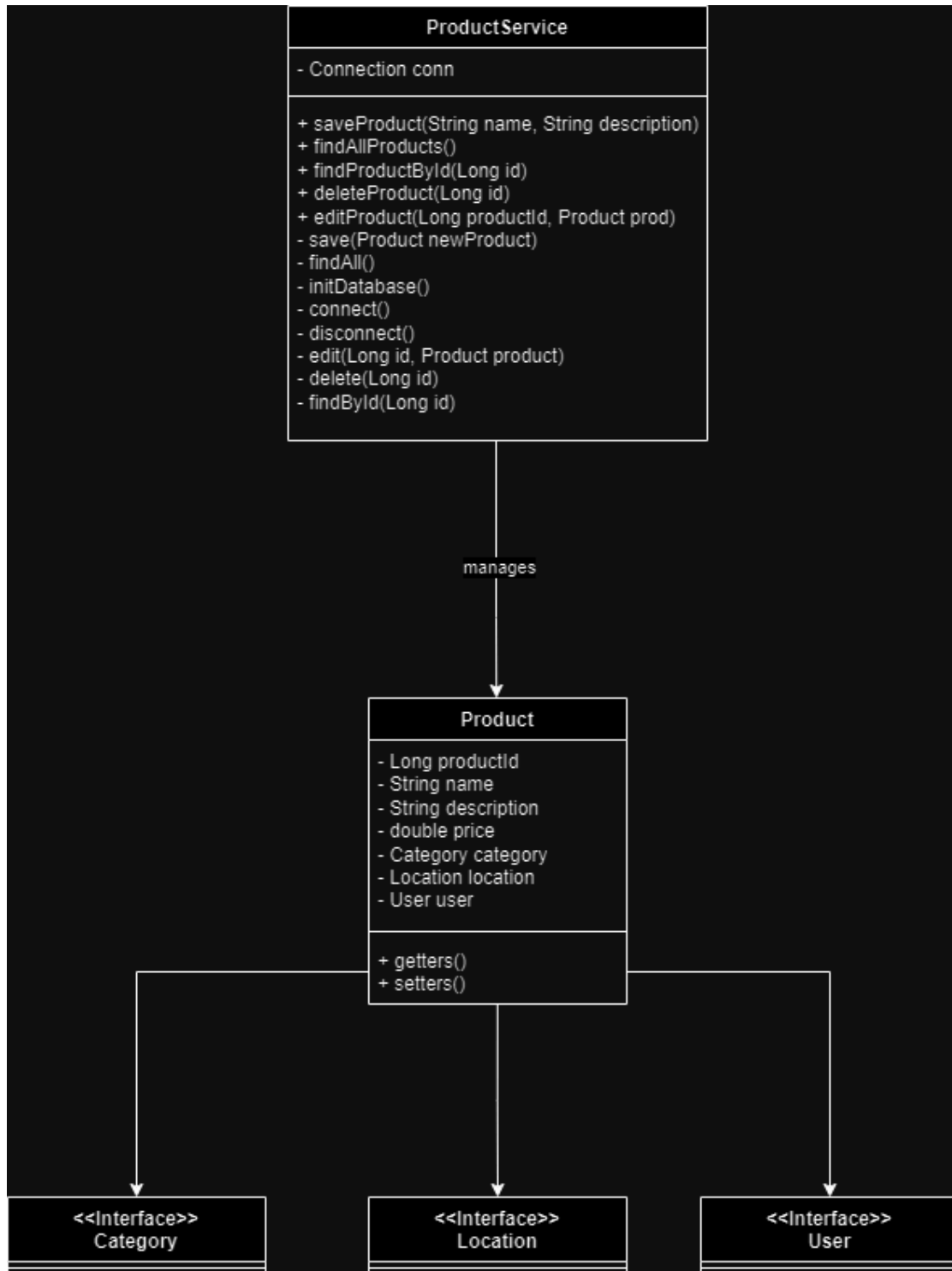
entidades ( <i>Category</i> , <i>Location</i> , <i>User</i> )		relaciones.
--	--	-------------

#### 4.1 Diagrama de módulos



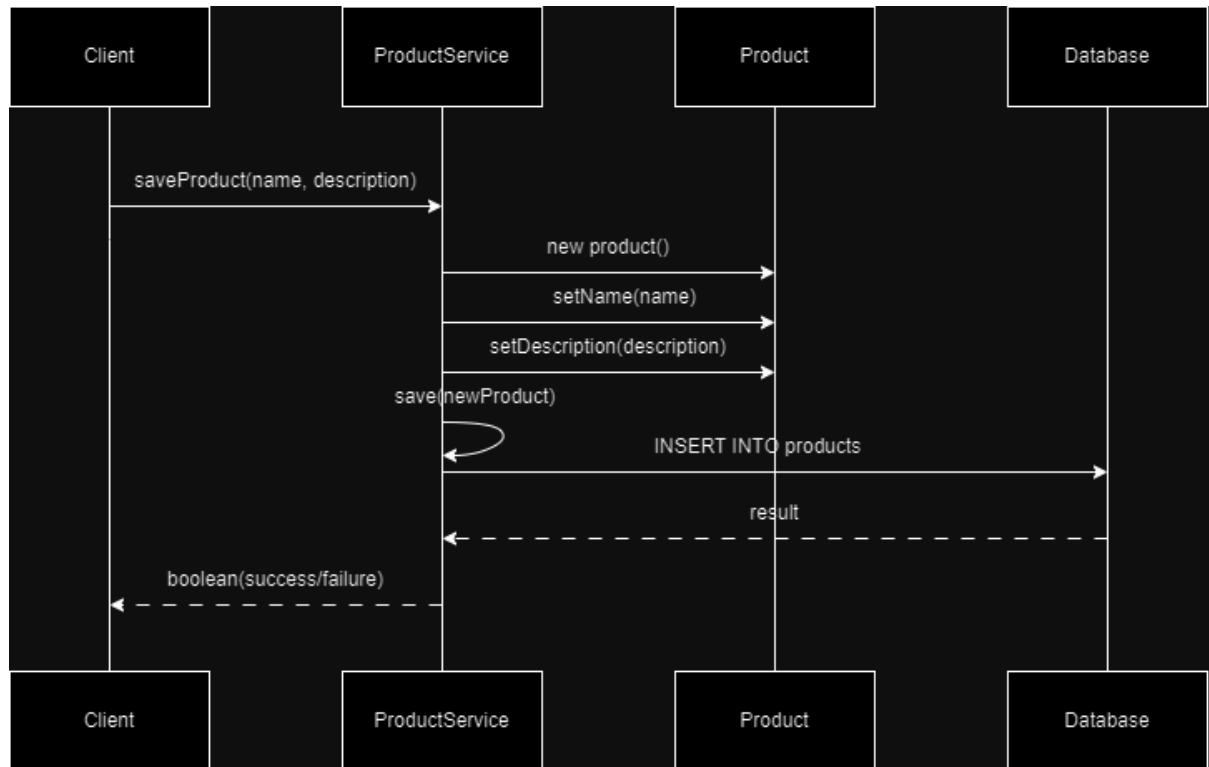
*Este diagrama de módulo muestra la estructura actual del paquete  
co. com. hyunseda. market. service*

#### 4.2 Diagrama de clases



*Este diagrama de clases muestra las relaciones entre las clases en el diseño actual*

### 4.3 Diagrama de secuencias



*Este diagrama de secuencia ilustra el proceso de creación de un nuevo producto utilizando el diseño actual*

Los diagramas anteriores (diagrama de módulos, diagrama de clases y diagrama de secuencia) y la tabla de análisis proporcionan una descripción general completa del diseño actual y sus problemas. Para la mejora del diseño se han de implementar las acciones planteadas en la tabla, de modo que se obtenga un código más modular, fácil de entender y extensible que se adhiera mejor a los principios de diseño.