Taller de Laboratorio 2. Principios SOLID

Universidad del Cauca ®

Presentado por:

Karen Nathalia Sandoval : 104622011416

Andrea Realpe Muñoz : 104622011442

Presentado a:

Julio Ariel Hurtado

Laboratorio de Ingeniería de Software 2

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Sistemas

Tabla de Contenido

Parte 1 2
- Principio de Única Responsabilidad 2
  - Imagen 1. Única razón vs Síntoma 3
  - Análisis 3
- Principio Abierto/Cerrado 3
  - Imagen 2. Abierto/Cerrado vs Síntoma 4
  - Análisis 4
- Principio de Sustitución de Liskov 4
  - Imagen 3. Liskov vs Síntoma 4
  - Análisis 4
- Principio de Segregación de Interfaces 5
  - Imagen 4. Segregación de Interfaces vs Síntoma 5
  - Análisis 5
- Principio de Inversión de Dependencias 6
  - Imagen 5. Inversión de Dependencias vs Síntoma 6
  - Análisis 6

Parte 2 8
- Tabla de Evidencia de la Violación a los Principios SOLID 8
- Diagrama de Módulos (Paquetes) 8
- Diagrama de Clases 8
- Diagrama de Secuencia 8

Parte 1

Principio de Única Responsabilidad

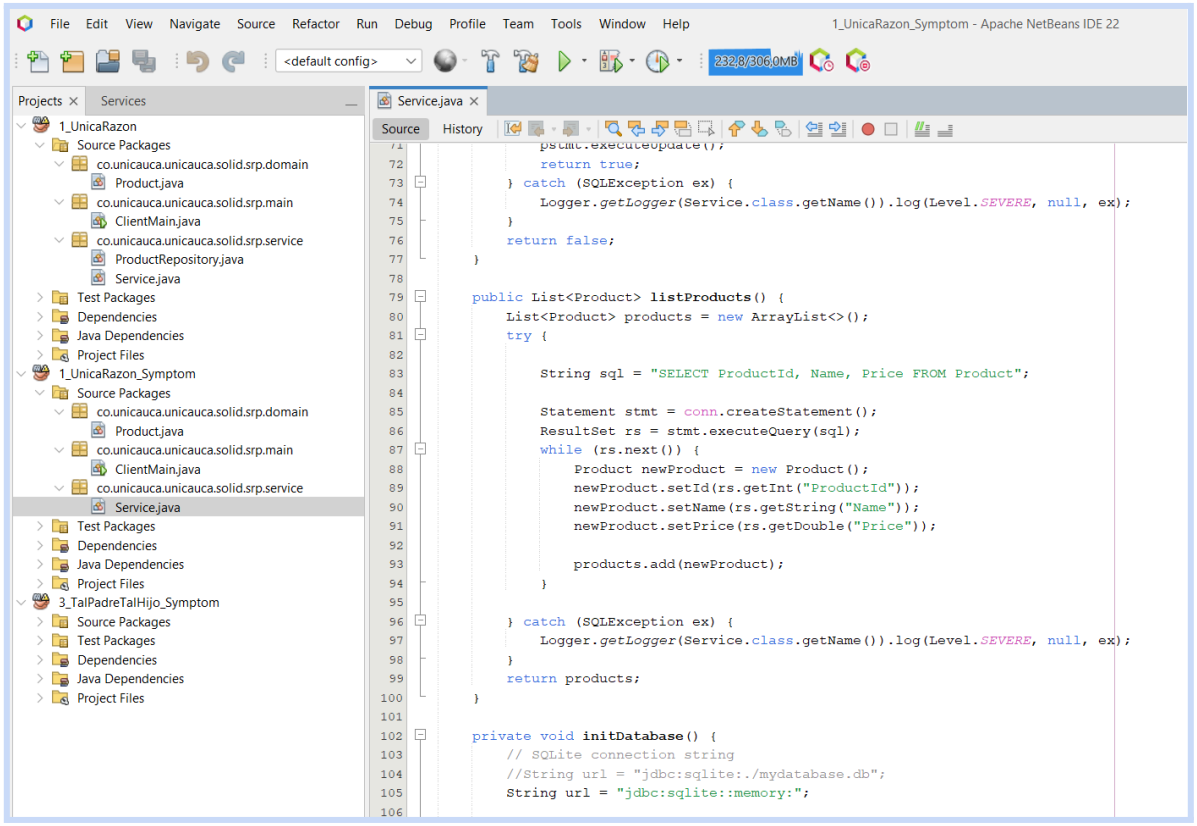

Imagen 1. Única razón vs Síntoma

Análisis

El propósito de este principio como su nombre lo indica, habla de que una clase debe tener una única razón para cambiar. Esto significa que una clase debe estar enfocada en hacer solo una cosa y hacerla bien. Por ello se puede observar que el ejemplo sympton en el paquete de servicios contiene una sola clase encargada tanto de la lógica de negocio como de conección/acceso a datos y a modo de correción en 1_UnicaRazon se crea ProductRepository para manejar el acceso a datos y quedando Service con una única responsabilidad, la de llevar la lógica de negocio.

Principio Abierto/Cerrado

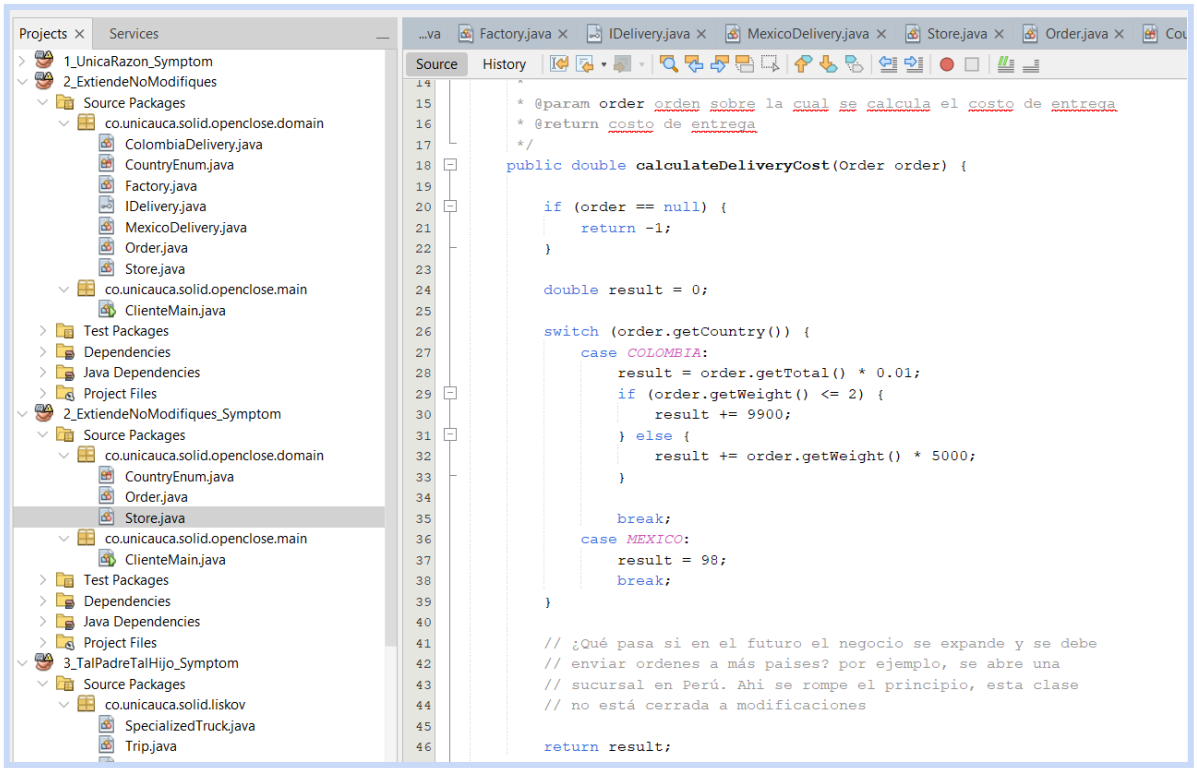

Imagen 2. Abierto/Cerrado vs Síntoma

Análisis

Aquí, las clases, módulos o funciones deben estar abiertas para su extensión y cerradas para su modificación. Esto permite que el comportamiento de una clase se amplíe sin modificar su código original y eso reduce el riesgo de introducir errores. Siendo así, en este programa que se encarga de calcular el costo de las órdenes en una tienda, el cual depende del país al que se vaya a enviar pasa de tener una clase abierta a modificaciones según dicho país, a volverse “independiente” siguiendo el principio abierto/cerrado de la siguiente manera. Para la versión mejorada, se implementó una interfaz IDelibery específicamente para calcular el costo, y la variación que puede tomar se implementa en las clases destinadas a cada país, así Store se cerraría a modificaciones.

Principio de Sustitución de Liskov

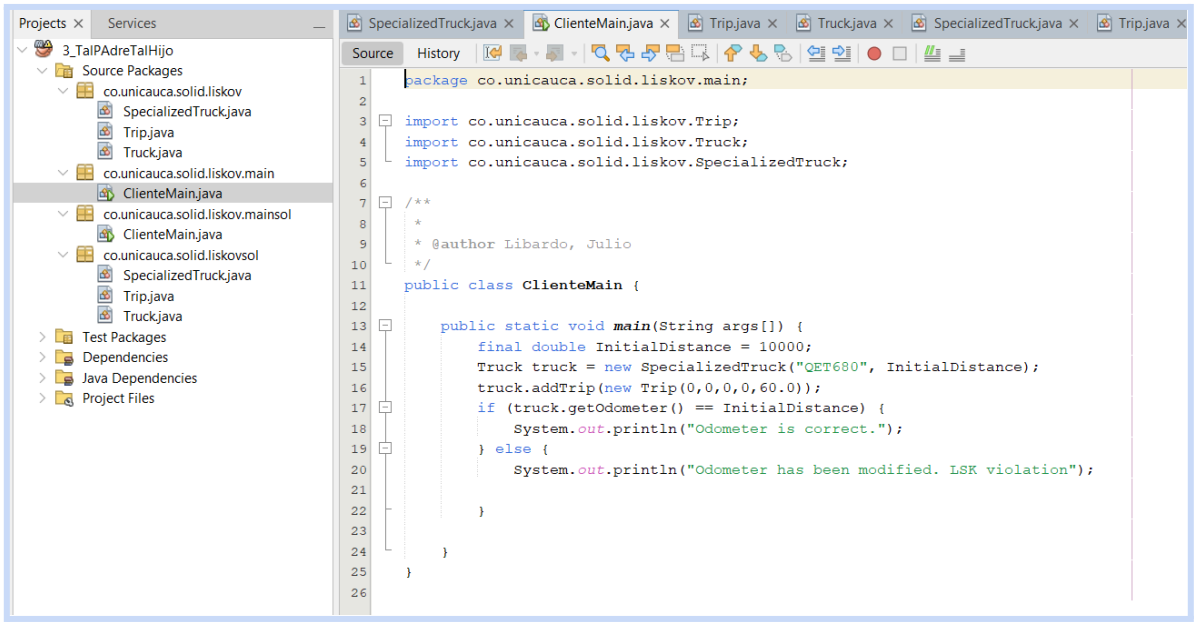

Imagen 3. Liskov vs Síntoma

Análisis

Para este principio los objetos de una clase derivada deben ser reemplazables por objetos de la clase base sin alterar el comportamiento del programa. En otras palabras, una clase hija debe poder sustituir a la clase padre sin que el sistema falle. En el ejemplo, la clase SpecializedTruck, que hereda de Truck, modifica el método addTrip, lo cual puede alterar el comportamiento que otras clases esperaban de Truck. Para cumplir con este principio, SpecializedTruck debería utilizar un atributo adicional para manejar las alteraciones, evitando así afectar el comportamiento de la clase base.

Principio de Segregación de Interfaces

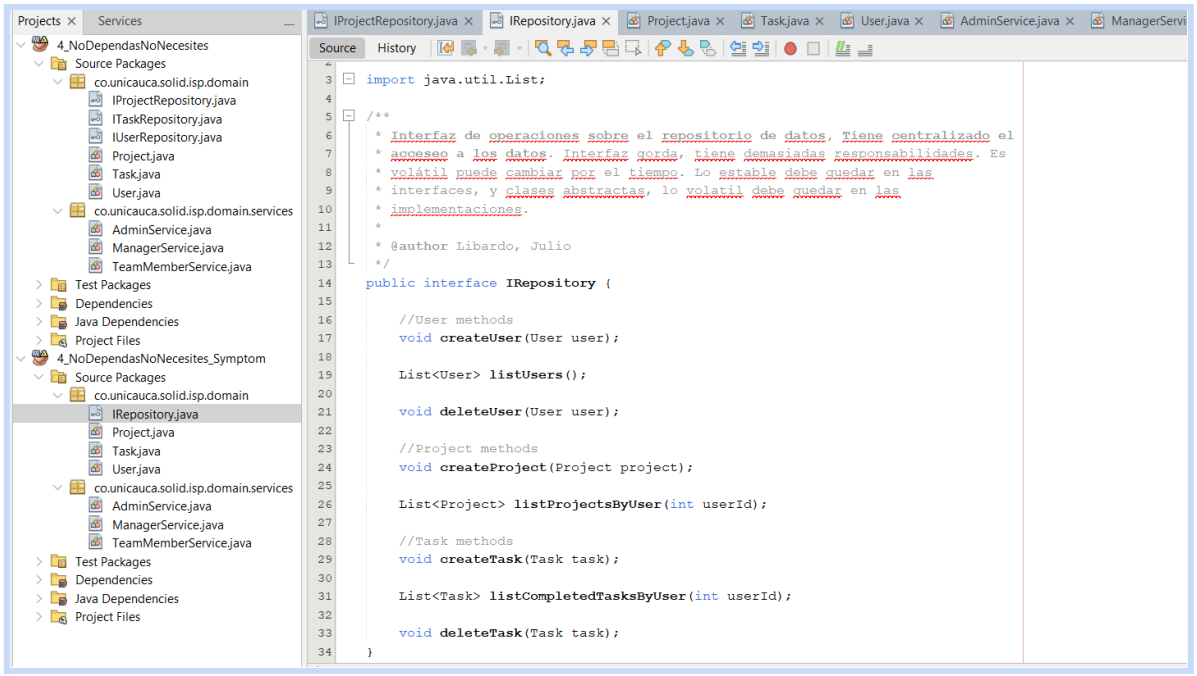

Imagen 4. Segregación de Interfaces vs Síntoma

Análisis

El objetivo aquí es no obligar a las clases a implementar interfaces que no necesitan. Es mejor tener interfaces más específicas (pequeñas y enfocadas) en lugar de una interfaz grande con métodos que algunas clases no usarán. En el programa, una interfaz Repositorio tiene demasiadas responsabilidades. La solución consiste en dividir esas responsabilidades en varias interfaces más específicas (ProjectRrepository, ITaskRepository, IUserRepository), cada una con una única función, para aplicar correctamente este principio.

Principio de Inversión de Dependencias

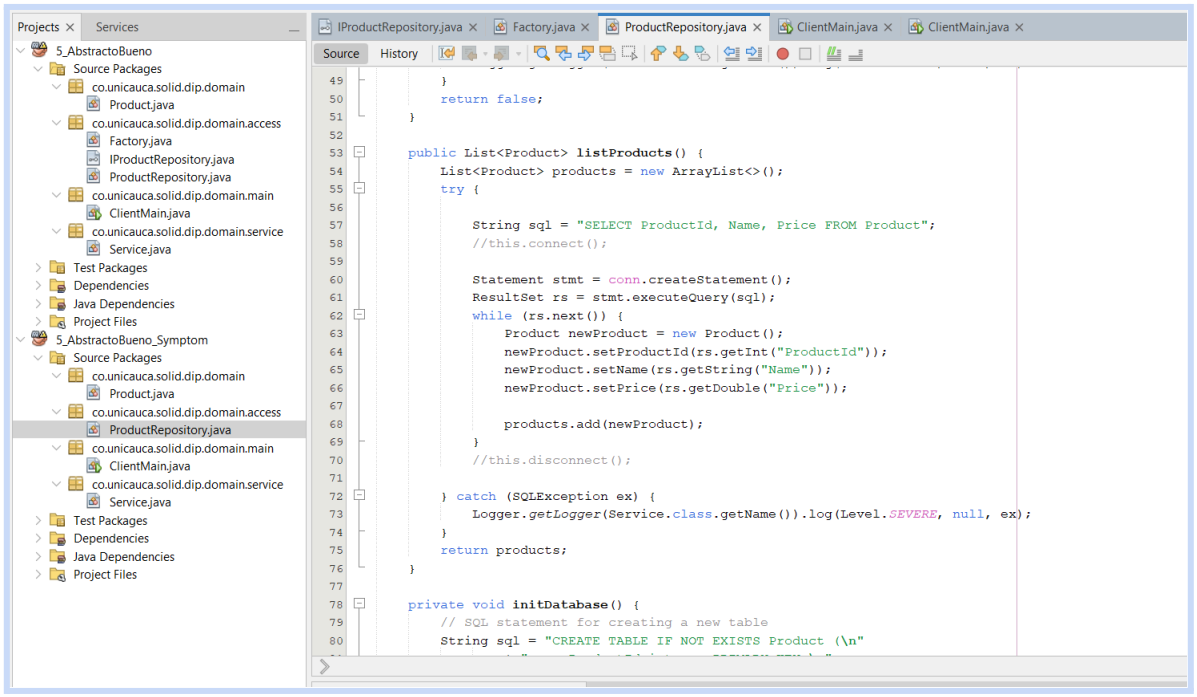

Imagen 5. Inversión de Dependencias vs Síntoma

Análisis

Según este principio, los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones. En el caso dado, el programa depende directamente de una conexión a una base de datos específica, lo que puede complicar la migración a otros sistemas de bases de datos en el futuro. La solución es crear una interfaz para la conexión a la base de datos, implementada de acuerdo con la sintaxis de cada tipo de base de datos, garantizando así que el programa está basado en abstracciones y no en detalles específicos. Finalmente es una “implementación que tiene libertad de hacer una implementación del contrato. Lo puede hacer con Sqlite, postgres, mysql, u otra tecnología”

Parte 2

Tabla de Evidencia de la Violación a los Principios SOLID

Ubicación	Principio Violado	Detalle	Refactorización
Class ProductService	Responsabilidad única	Tiene dos responsabilidades, el acceso a base de datos y la lógica de negocio (guardar, editar, eliminar productos).	Crear una clase exclusivamente para el acceso a base de datos DatabaseManager y para el crud se mantiene ProductService.
El método saveProduct dentro de ProductService	Abierto/Cerrado	Debido a que verifica el estado de los productos (por ejemplo, que el nombre no esté vacío). Si después se añade una nueva validación, tendría que modificarse el código	Validar en una clase separada ProductValidator, que pueda ser extendida sin modificar el código de ProductService.
Main y GUIProducts	Inversión de Dependencias	Dependen de una implementación ProductService, en lugar de depender de una abstracción.	Introducir interfaces para ProductService y hacer que Main dependa de una abstracción (interfaz) en lugar de una implementación concreta.

Diagrama de Módulos (Paquetes)

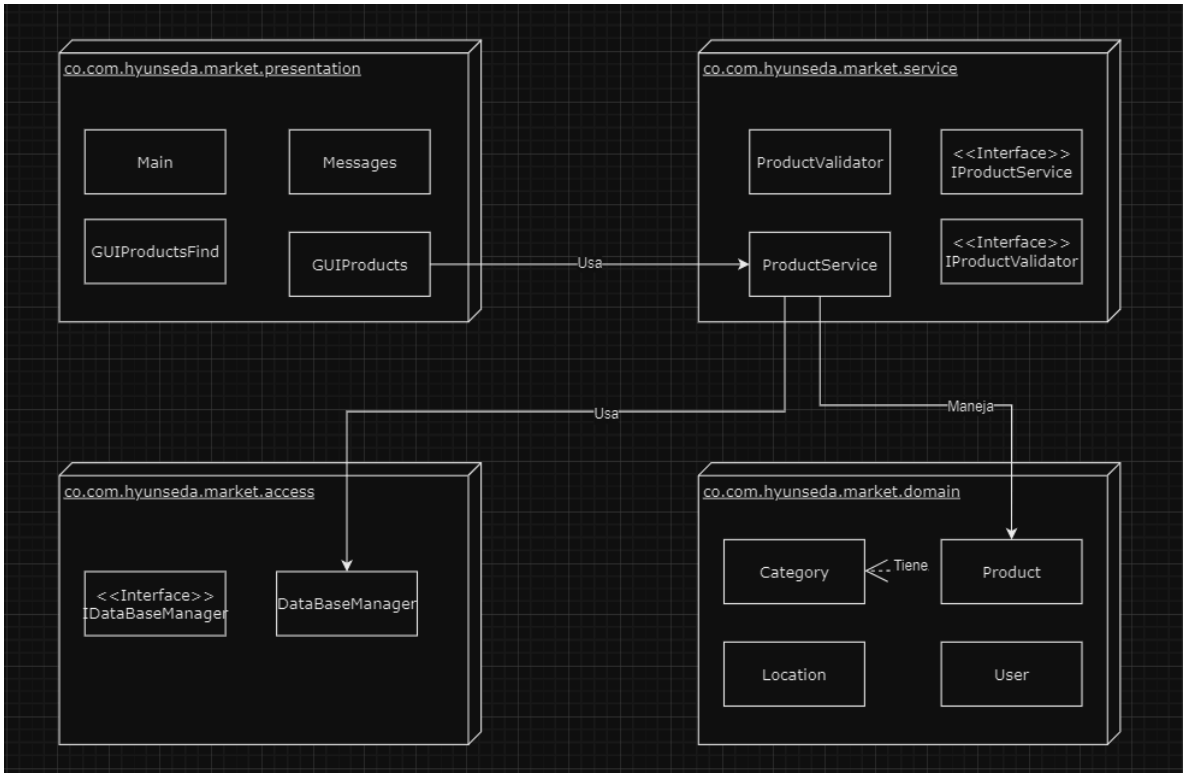

Diagrama de Clases

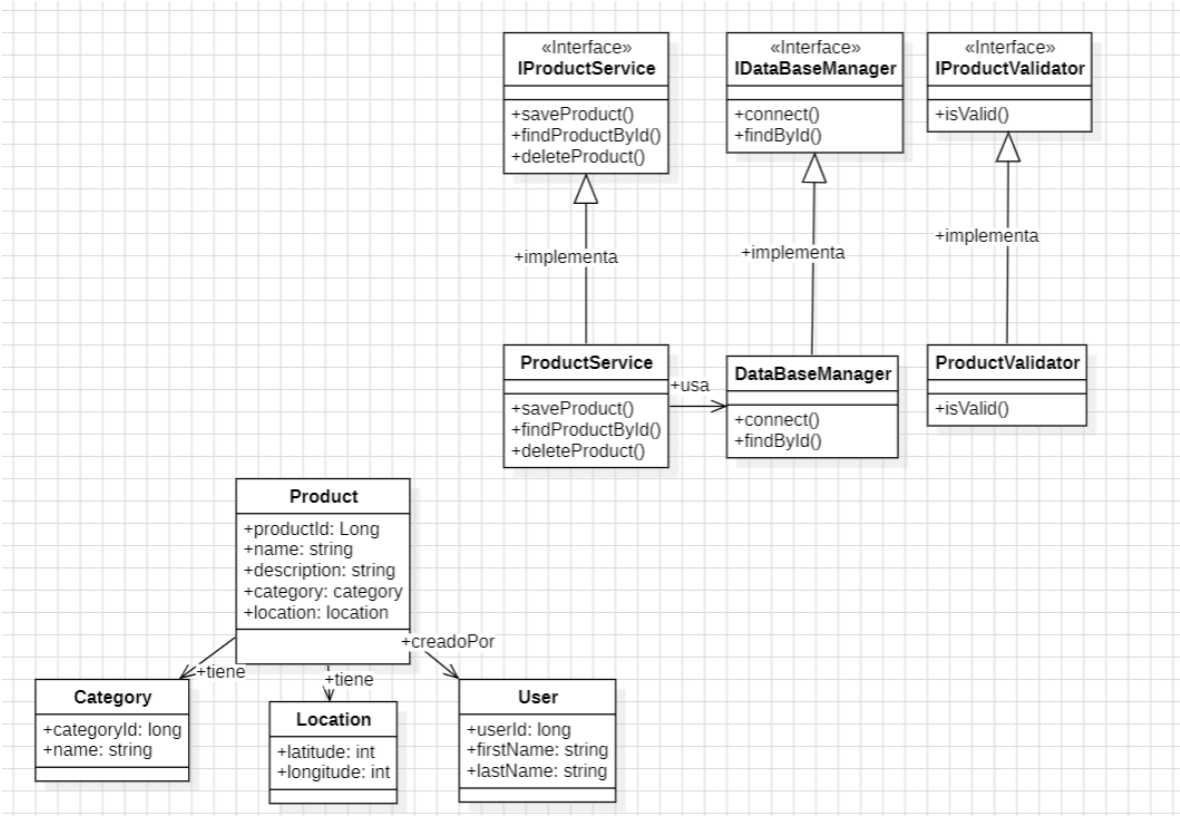

Diagrama de Secuencia