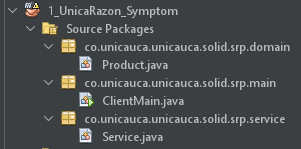
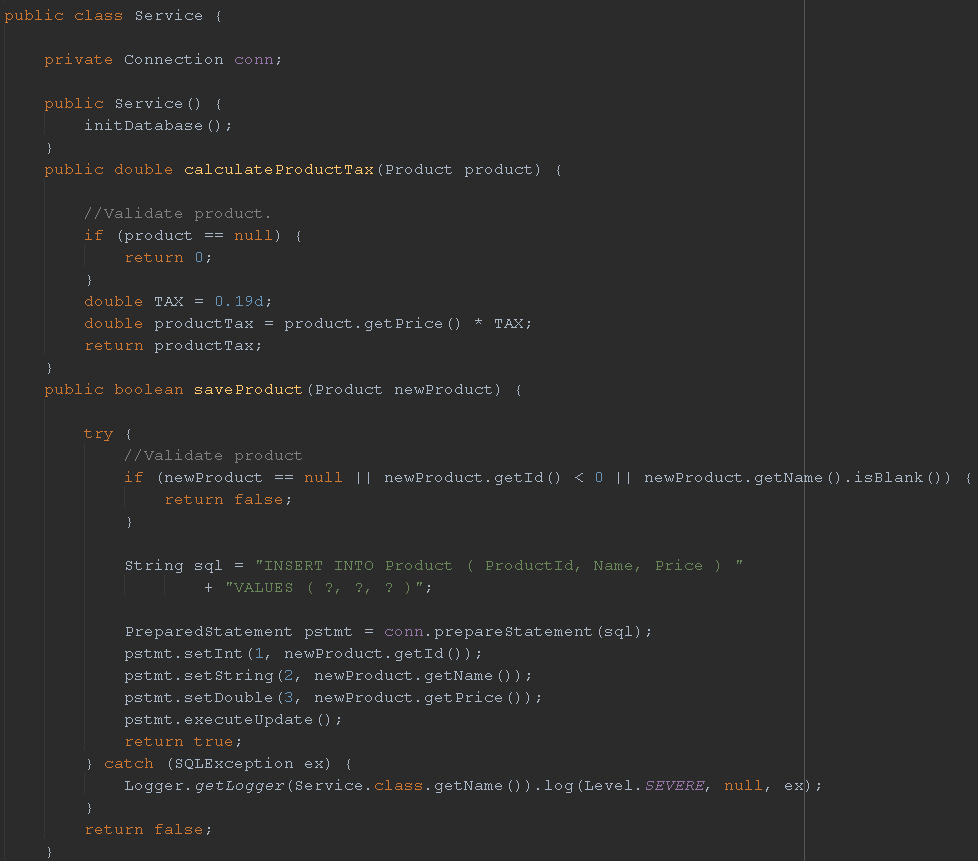
**Parte I**

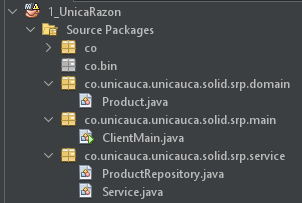
* Principio de Única Razón:



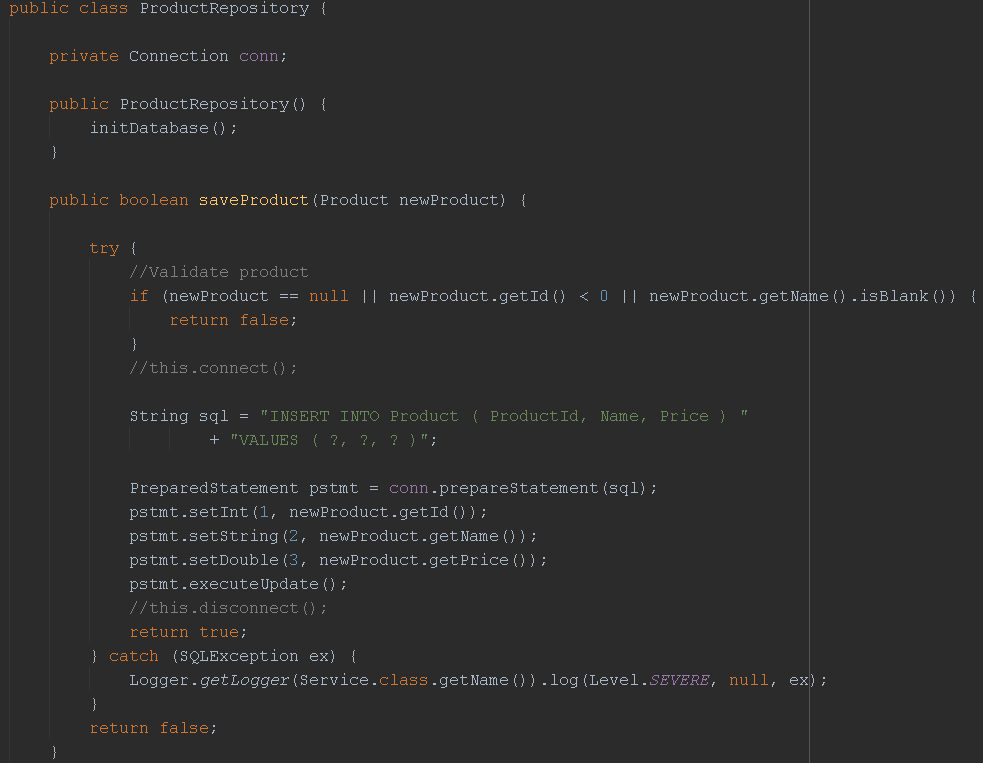
En la anterior imagen encontramos la estructura inicial del proyecto, en al cual al adentrarnos en la clase servicio encontrábamos que la lógica implementada en ella se extralimitaba en responsabilidades, presentando la lógica de negocios y la de acceso a datos en una misma clase, violando el principio de única razón como se ve a continuación:



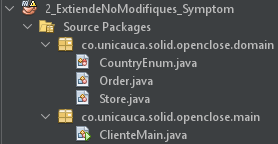
Ello se vio solucionado al crear una clase nueva llamada ProductRepository con la cual se encargaría de la lógica de acceso a datos, de la cual Service haría uso.

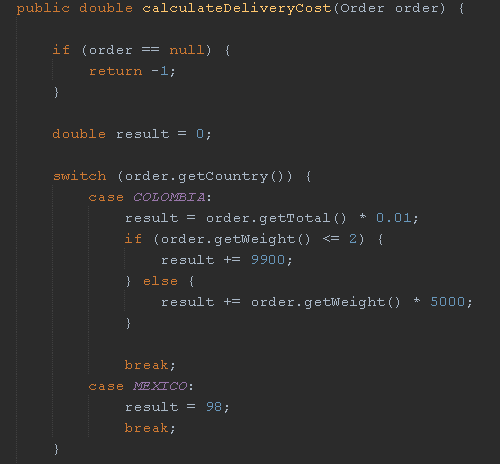




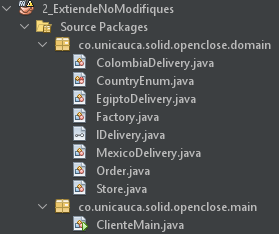


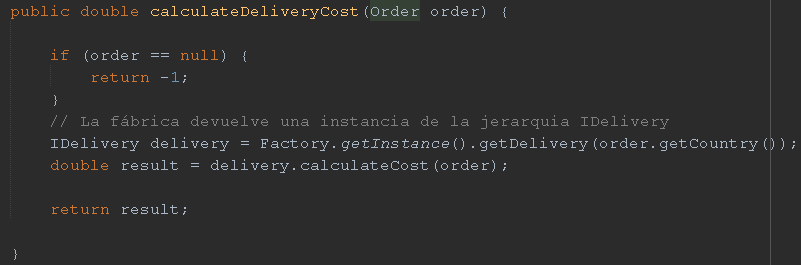
* Principio Abierto-Cerrado – OCP



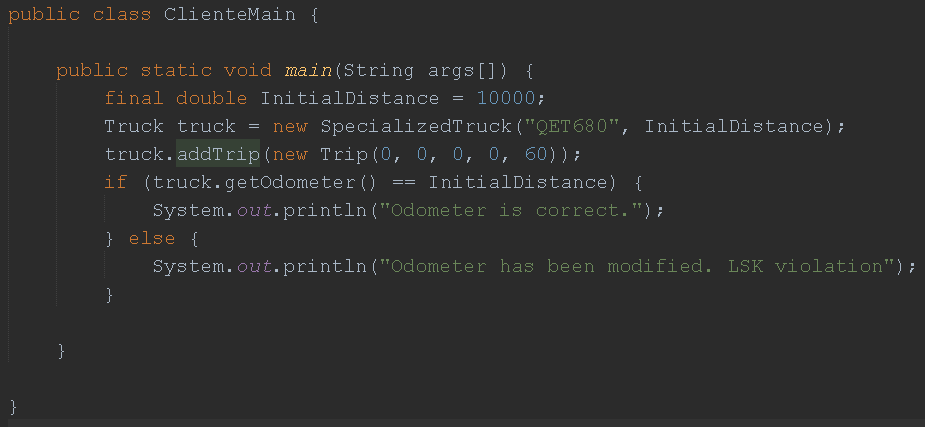


Dentro de la clase Store encontramos el método calculateDeliveryCost el cual hace muy engorrosa la modificabilidad del mismo a medida que se extiende a más países con distintas lógicas, siendo contrario al principio OCP, por ello en la segunda versión se optó el uso de un objeto abstracto, la interfaz IDelivery, la cual gracias a la clase Factory según la instancia de la clase de cada país se implementaría de un modo específico en lugar del switch en el método.



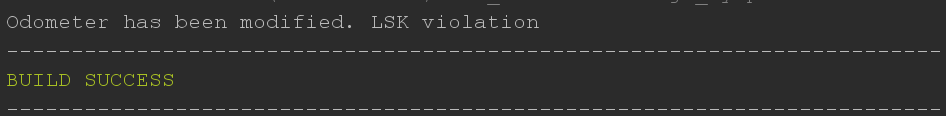


* Principio de Substitución de Liskov

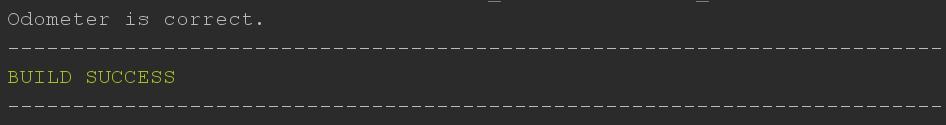


En el programa ClienteMain se tiene que SpecializedTruck es un subtipo de Truck entonces los objetos del tipo Truck deben poder ser reemplazados con objetos de tipo SpecializedTruck sin alterar el comportamiento del programa ClienteMain el cual habría de imprimir el mensaje “Odometer is correct.”

En el síntoma observamos que SpecializedTruck no se códifico del modo adecuado pues al ejecutarlo vemos que el el comportamiento del programa ClienteMain fue alterado



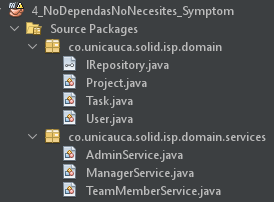
Al realizar una implementación correcta de la herencia en la solución vemos que el comportamiento es el esperado.



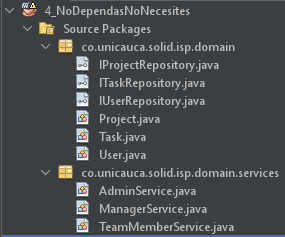
* Principio de Segregación de Interfaces

Podríamos resumir este principio en que es la aplicación del principio de única razón pero esta vez dirigido a interfaces.

Así pasamos de lo que era una única interfaz grande de la cual AdminService, ManagerService y ITaskRepository dependían, aunque cada una la usara parcialmente



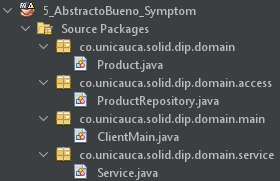
A una segregación de la interfaz anterior en múltiples que ahora sí cada servicio que utilice a cada cual haga un uso completo de cada una de ellas.



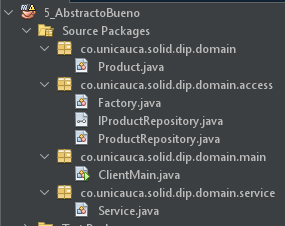
* Principio de Inversión de Dependencia

Módulos de alto nivel no dependen de módulos de bajo nivel. Ambos deberían depender de abstracciones, abstracciones no dependen de detalles. Detalles dependen de las abstracciones. Para nuestro caso las abstracciones vendrían a ser las interfaces y las clases sus detalles.

Así donde antes encontrábamos como en la clase Service una instancia de la clase ProductRepository



ahora encontraremos una de la interfaz IProductRepository.



**Parte II**

* En la Clase ProductService se violaba el principio de única razón, por lo que se creó la clase ProductRepository para encargarse del acceso a datos y en ProductService se agregó una dependencia a la interfaz también creada, llamada IProductRepository, para seguir el principio de Inversión de Dependencia, por lo mismo igualmente se crearon las interfaces ICategory, ILocation y así mismo IUser por lo tanto también fue necesaria la modificación de la creación del objeto producto en la clase Main y la creación de las clases FactoryProductRepository, FactoryCategory, FactoryLocation y FactoryUser.