**Ejercicio: Aplicación de Principios SOLID a Sistema de Información Viajero Frecuente**

El siguiente ejercicio práctico tiene como objetivo ejemplificar la aplicación de los principios SOLID para el desarrollo de software orientado a objetos.

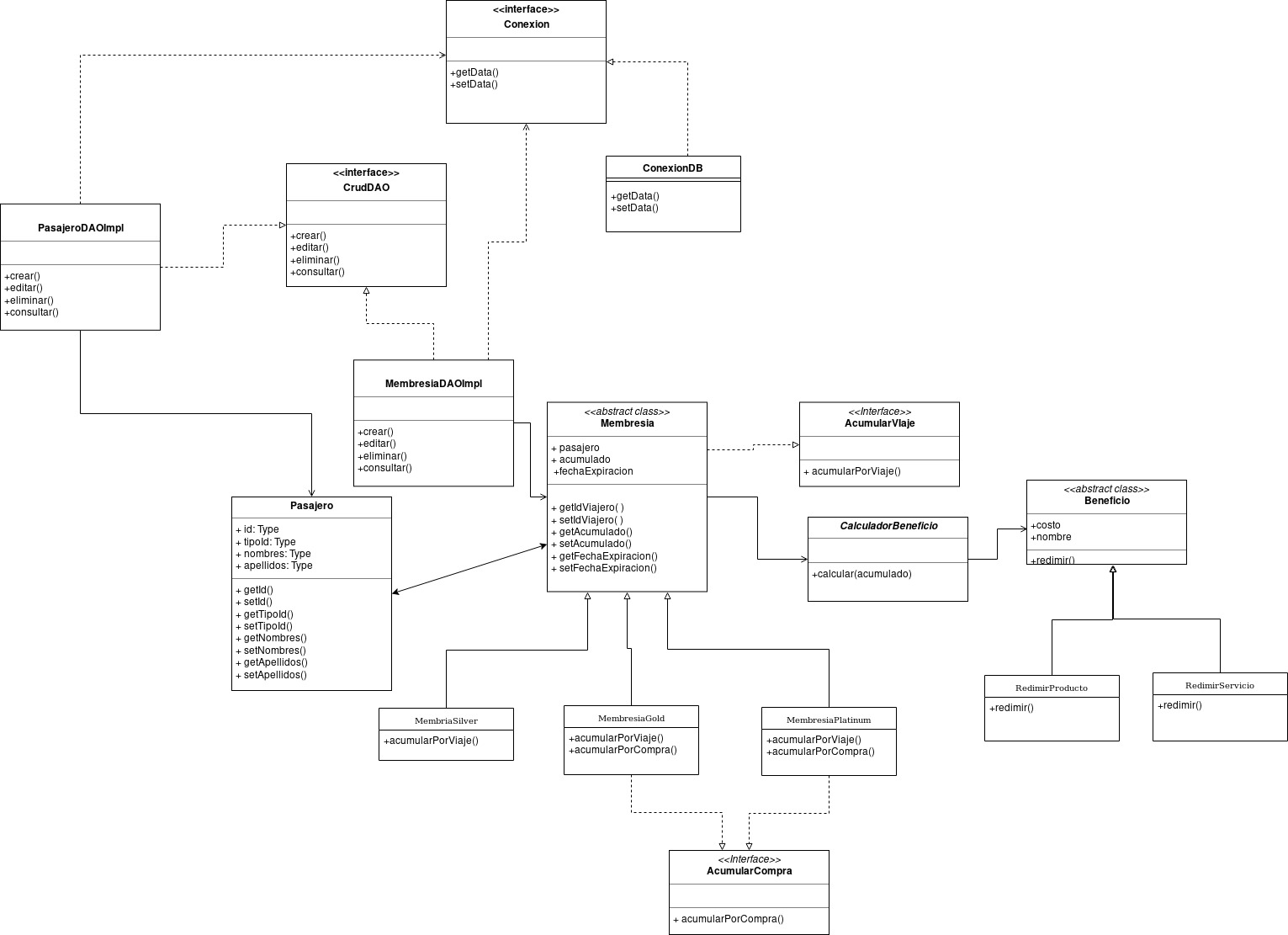
El tema en el que se le da aplicabilidad al ejercicio, es un fragmento de un sistema de información para viajeros frecuentes.

El alcance del ejercicio planteado, está en la aplicación de los principios SOLID, más que en la adecuabilidad y validez final del sistema (diagrama de clases) con respecto a las necesidades actuales de un viajero frecuente.

Para comenzar, nuestro diagrama de clases propuesto es: <ver imagen siguiente página, también adjunta en el mismo directorio de este documento>

Expliquemos brevemente el diagrama: un Pasajero posee un tipo de Membresia, lo que a través de Viajes o Compras, le permite acumular millas, y a su vez, le brinda acceso a diferentes tipos de Beneficios, ya sea en productos o servicios.

Por supuesto, toda esta información debe persistir en base datos, por lo cual, en el diagrama también se hace referencia a los artefactos que cooperan para persistir y obtener la información persistida.

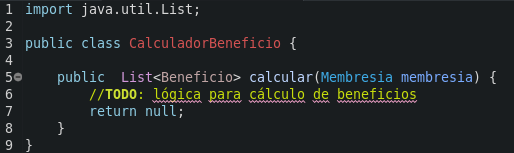
**Aplicación de SOLID:**

A continuación se hace mención de cada principio y su aplicación dentro del diagrama de clases visto.

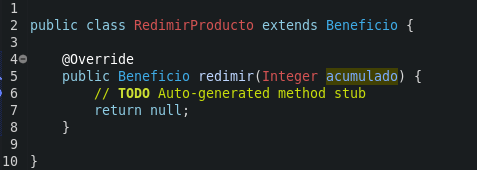
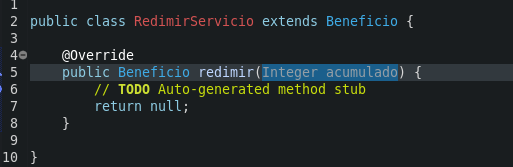
**SRP: Single Responsibility Principle:**

Las clases deberían tener una y solo una razón para cambiar.

* La clase CalculadorBeneficio tiene un nombre bastante específico y solo se encargará de calcular beneficios.

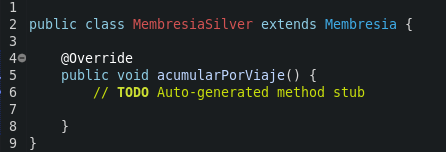
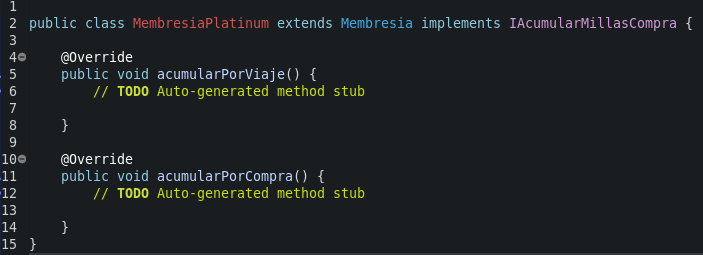
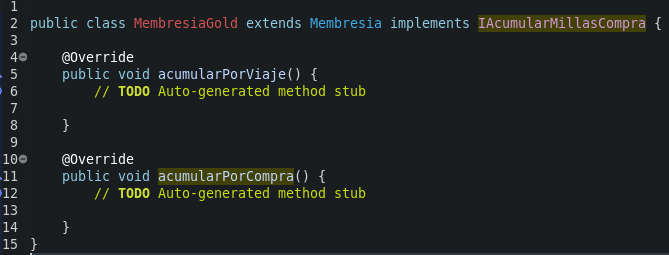
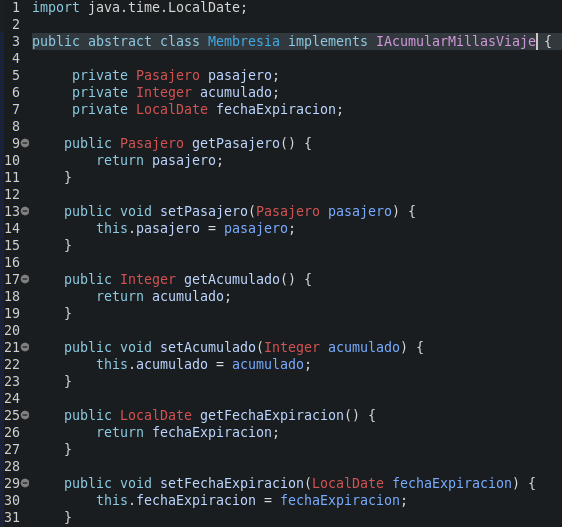


* Las clases RedimirProducto y RedimirServicio son lo suficientemente específicas, pues solo tendrán la responsabilidad de llevar a cabo la actividad de redención de millas a través de productos o servicios respectivamente.



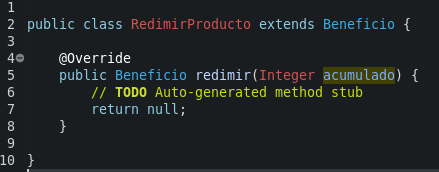
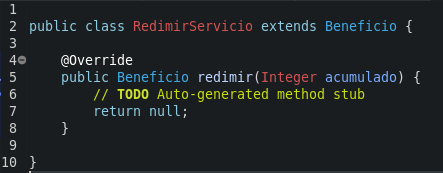
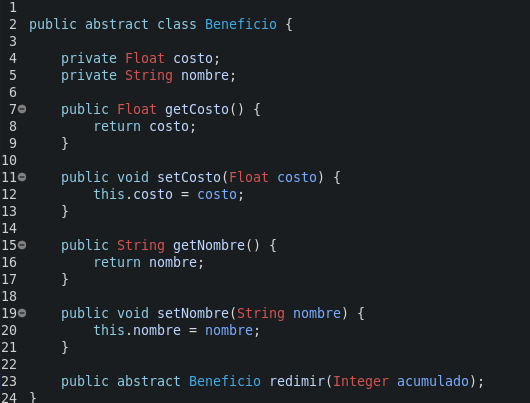
**OCP: Open/Closed Principle:**

Las clases deben estar abiertas para extensión y cerradas a modificación.

* La clase Membresia cumple con este principio dado que a través de las clases concretas MembresiaGold, MembresiaSilver y MembresiaPlatinum implementan el comportamiento declarado por la interfaz AcumularViaje y de esta forma, cada vez que se cree un nuevo tipo de Membresia, se creará una clase específica para el nuevo tipo, sin necesidad de modificar las demás clases relacionadas.
* En nuestro diagrama, también se aplica este principio para las clases AcumularBeneficio, Beneficio, RedimirProducto y RedimirServicio; pues en el momento en que se requiera agregar otro método de redención, no es necesario modificar la clase AcumularBeneficio donde se ejecuta el método redimir para cada forma de redimir (Producto, Servicio, etc).

**LSP: Liskov’s Substitution Principle:**

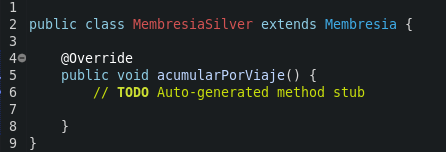
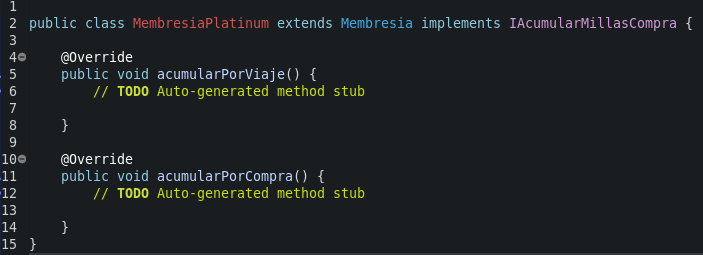
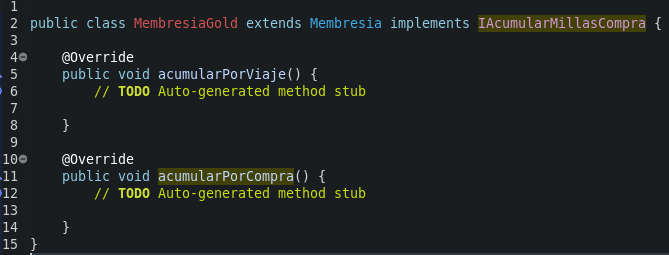
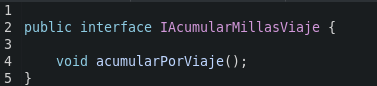
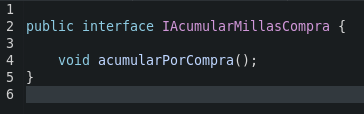
Una subclase puede ser reemplazada por su superclase.

* Las clases RedimirProducto y RedimirServicio pueden ser reemplazadas por su superclase Beneficio, debido a que sobreescriben el único método disponible para ser sobreescrito, usando la misma firma que la superclase para sobreescribirlo. De esta forma, podemos intercambiar las subclases por la superclase sin que el aplicativo genere algún tipo de error o excepción.

**ISP: Interface Segregation Principle:**

Un cliente nunca debe ser forzado a implementar una interface/método que no use.

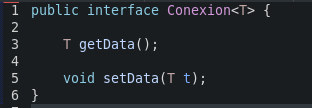
* Las interfaces AcumularCompra y AcumularViaje inicialmente hacian parte de una sola interfaz. Lo que se realizó fue dividirlas en dos interfaces con lo cual se logra que cualquier cliente pueda implementar selectivamente alguno de los comportamientos declarados sin que sea obligatorio implementar ambos a la vez.

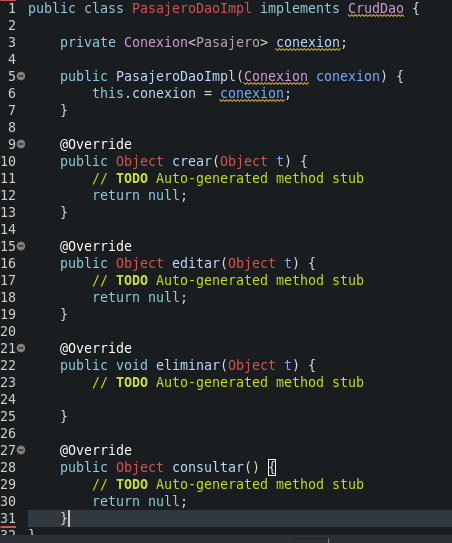
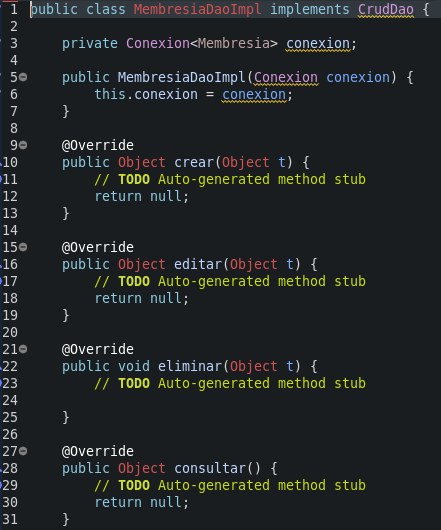
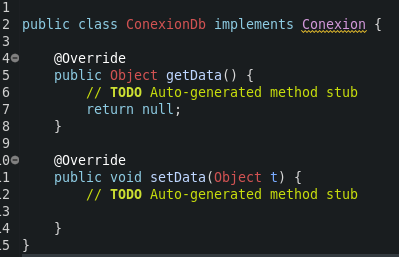
Como se puede apreciar en nuestro caso, las clases MembresiaGold y MembresiaPlatinum implementan ambas interfaces, sin embargo, la clase MembresiaSilver solo implementa la interfaz AcumularViaje.

**DIP: Dependency Inversion Principle:**

Las entidades deben depender de abstracciones y no de concreciones.

* Las clases PasajeroDaoImpl y MembresiaDaoImpl dependen de la interfaz Conexion (abstracción), y no de la clase concreta ConexionDB (concreción), con lo cual se logra desacople entre los diferentes tipos de conexiones que pudiesen existir, sin afectar a las entidades que dependen de la conexión; y no de la forma en que esta se pueda implementar.





Cabe resaltar que además de los puntos de aplicabilidad mencionados para cada principio, pueden haber más aplicaciones de los principios en el diagrama de clases, sin necesidad de que estos fuesen mencionados.

**Conclusiones**

La aplicación de SOLID fue llevada a cabo en este ejercicio, teniendo en cuenta el contexto y no de una forma netamente sistemática, pues entendemos a SOLID como un conjunto de principios que nos ayudan a mejorar mantenibilidad, extensibilidad y legibilidad de los productos de software, más que como unas leyes que tengan que aplicarse simplemente.