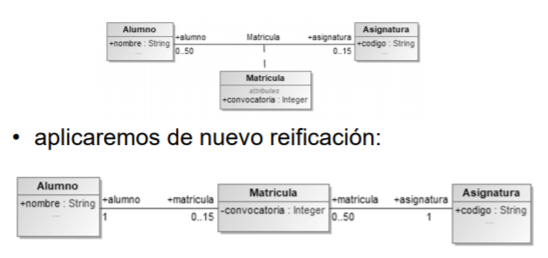
Pedro Díaz Gutiérrez

**Práctica 4**

**Ejercicio 1**

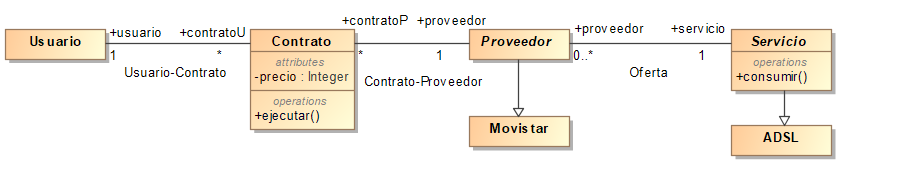
Para solucionar este problema, nos iremos al principal foco de problemas que es la clase de asociación Contrato, puesto que al ser la clase Servicio abstracta, no podremos acceder a esta. Para ello, transformaremos la clase de asociación Contrato en una clase entre Usuario y Proveedor, manteniendo las multiplicidades, es decir, realizaremos una reificación de la asociación como en este ejemplo:



Debido a la multiplicidad de la relación y la bidireccionalidad entre las clases Servicio y Proveedor , obligatoriamente tendremos un servicio, getServicio(), en la clase Proveedor, así sabremos lo que nos ofrece el Proveedor, además de la posibilidad poder acceder a este servicio desde la clase Contrato .

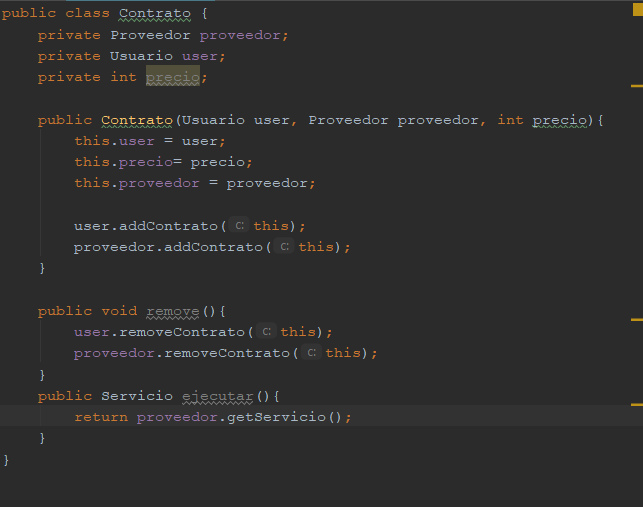
A continuación introduciremos nuestro modelo en UML y la implementación del código del modelo:

-Así quedaría nuestro modelo en UML.

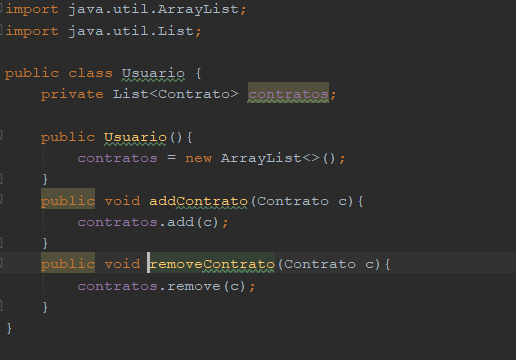


-Código:

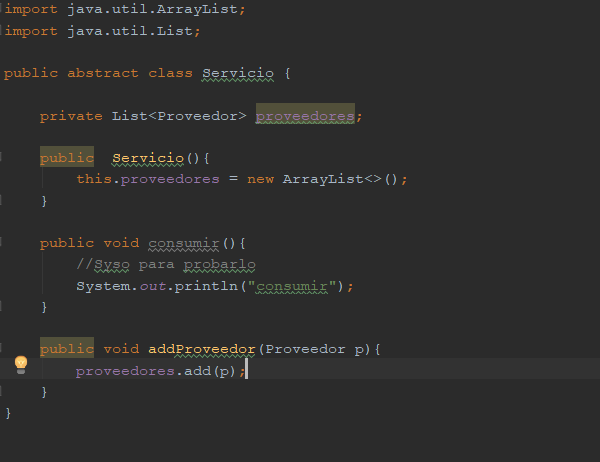
\*Clase Contrato



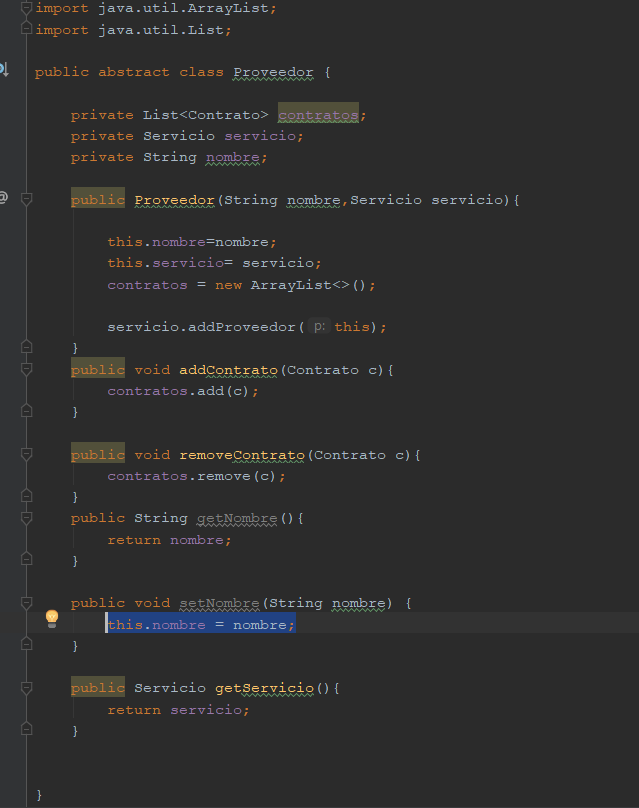
\*Clase Usuario



\*Clase Abstracta Servicio

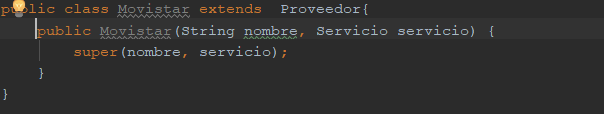


\*Clase Abstracta Proveedor



\*Clases de Generalización



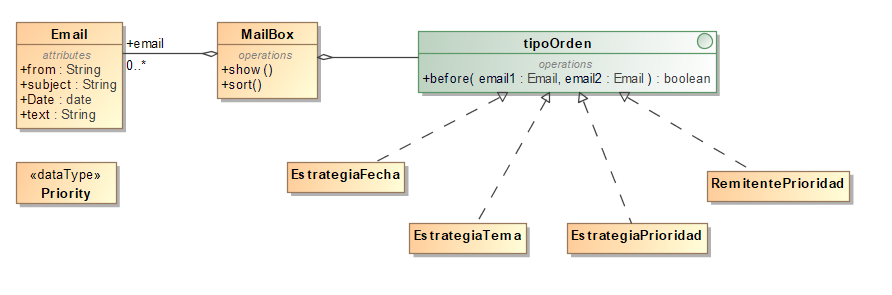


Ejercicio 2

Para resolver este ejercicio utilizaremos el patrón de diseño de Estrategia debido a que tenemos que gestionar los mensajes del Usuario en su Mail-Box, tendremos diversas maneras de ordenarlos (texto,fecha...), lo que significa que tendremos muchas maneras de abordar el problema utilizando este patrón.

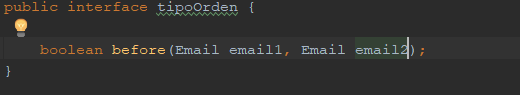
Siguiendo las opciones para ordenar el Mail-Box utilizaremos una Interfaz, tipoOrden, en la que utilizaremos la operación before(), para seleccionar el tipo de filtro para ordenar que utilizaremos.

UML:

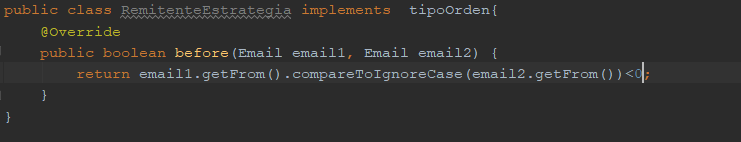


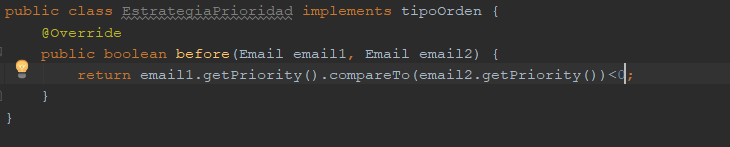
Código:

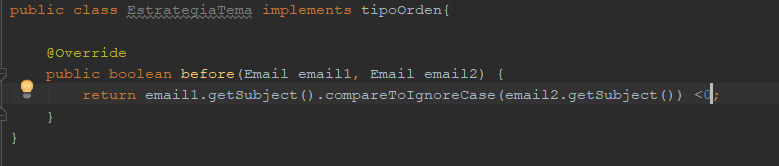
\*tipoOrden, Interface:

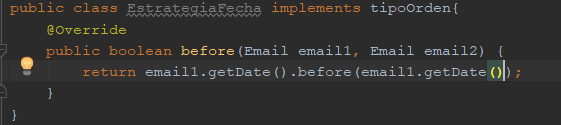


\*Estrategias para resolver el problema:

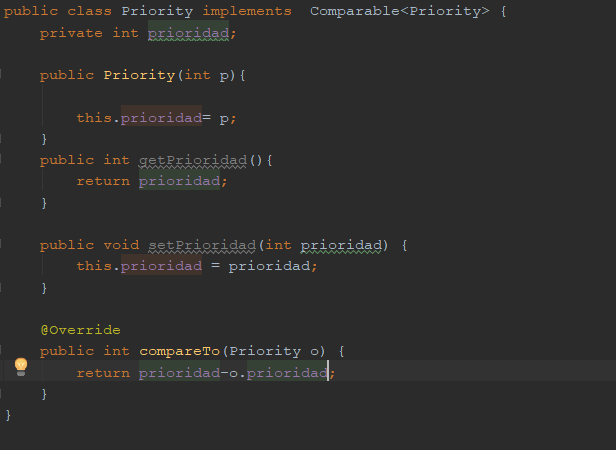




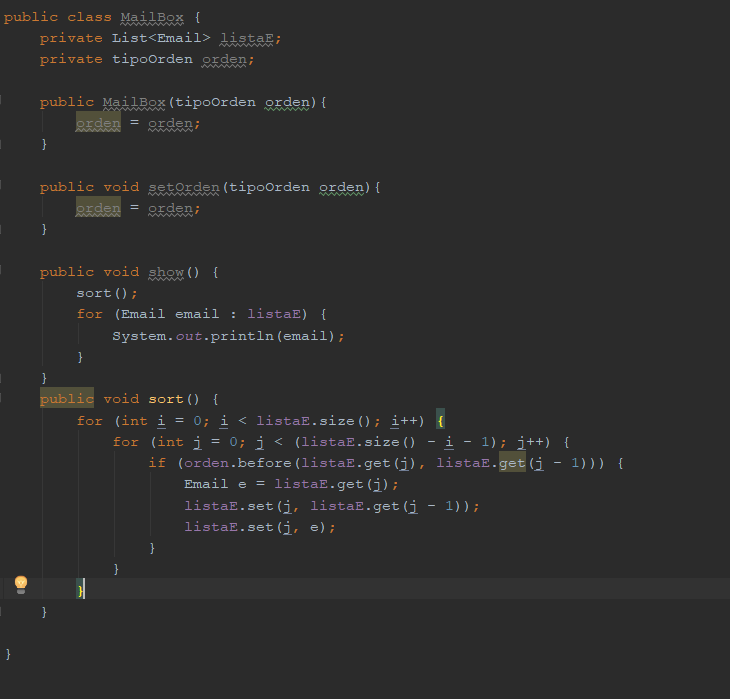




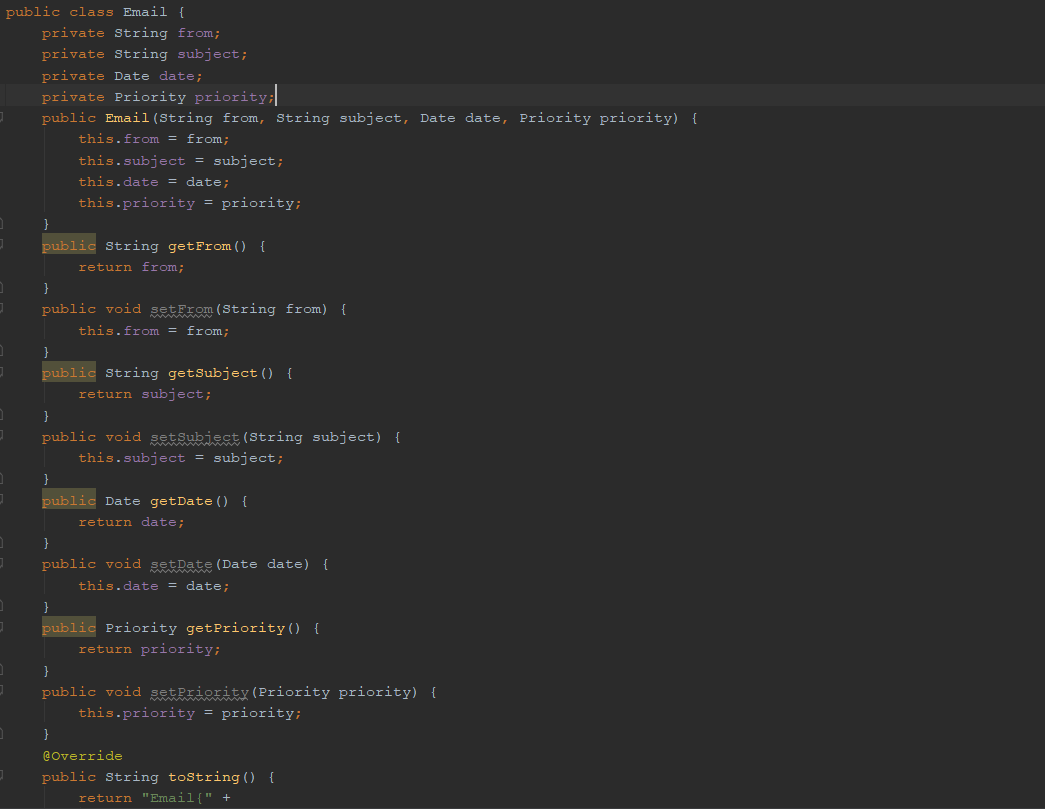
\*Clase Priority



\*Clase MailBox

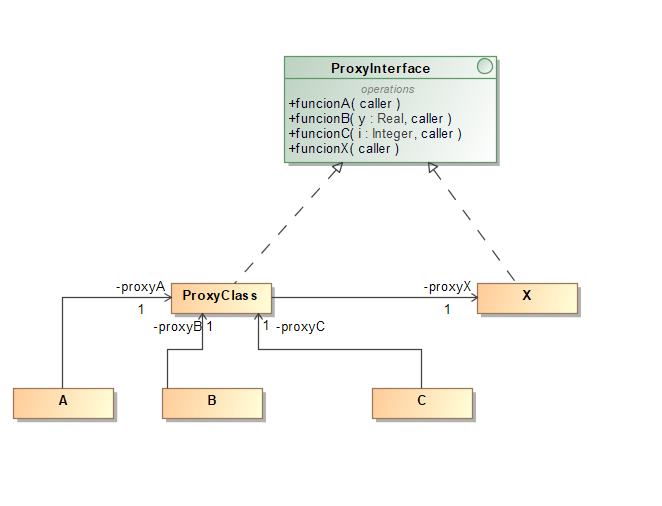


\*Clase Email

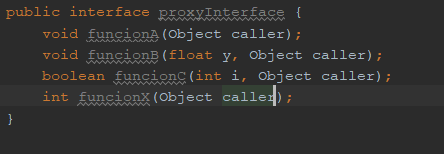


Ejercicio 3

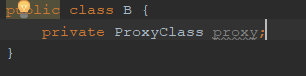
1. Mientras que en Eiffel es posible administrar los permisos de cada clase,en Java es imposible realizar una exportación selectiva, es decir, no podemos seleccionar libremente los permisos de las clases y a lo que pueden acceder. Para solventar esto podemos utilizar el patrón Representante
2. Como hemos dicho en el apartado anterior, podremos utilizar el patrón Representante/Proxy, aunque en Java nos encontraremos ciertos inconvenientes, ya que para definir los permisos de cada clase que se implemente se tendra que modificar el patrón con las nuevas reglas.



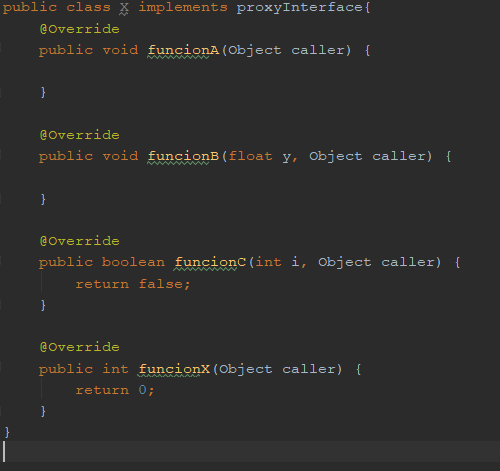
\*Interfaz, ProxyInterface:



\*A,B,C clases sin metodos.



\*Clase X



\*ProxyClass

