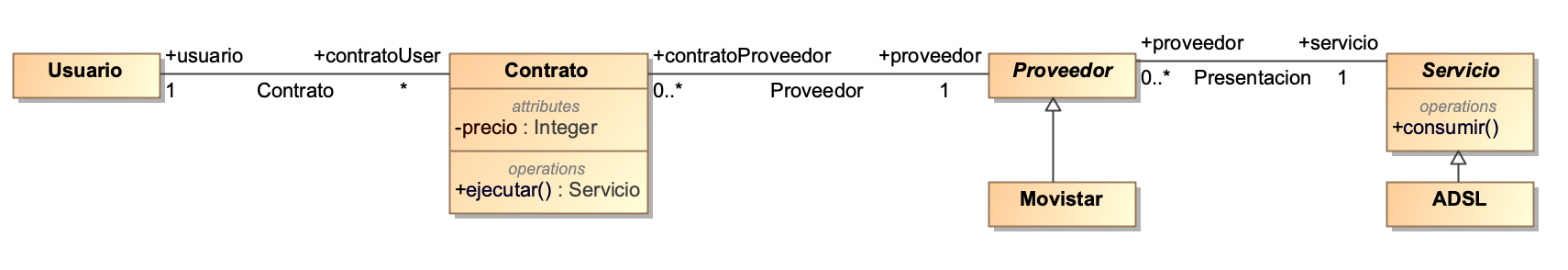
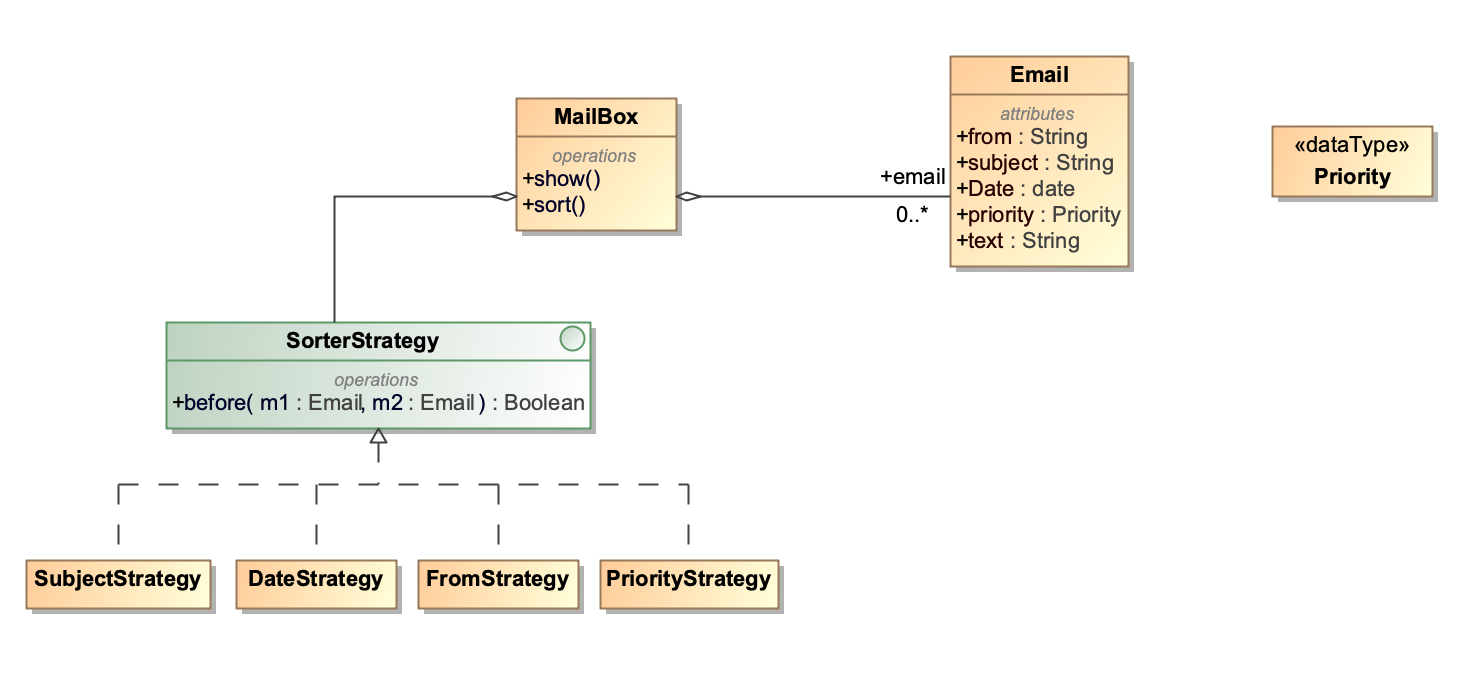
**Práctica 4**

**Ejercicio 1:**



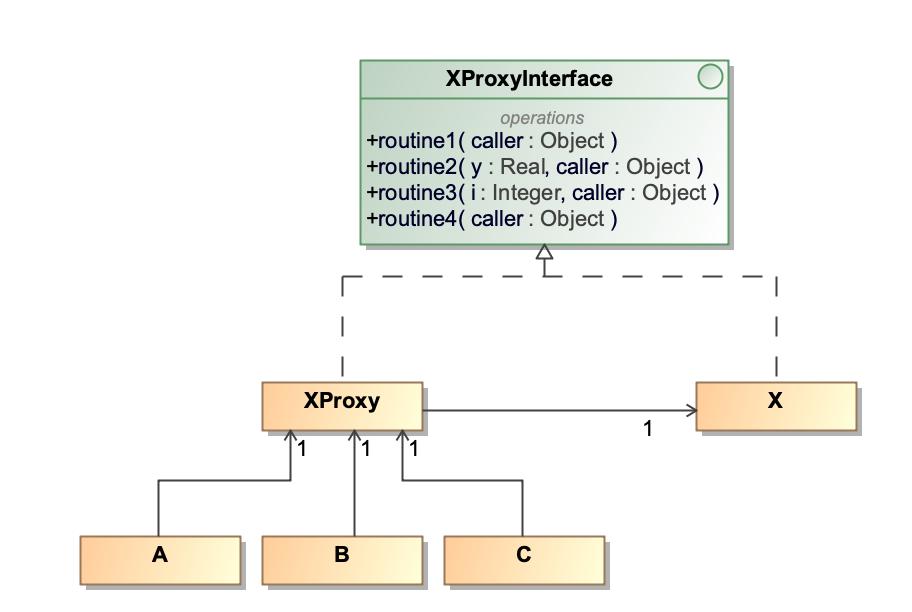
Para poder hacer una solución viable para el problema, primero reificaremos la clase de asociación Contrato. El problema que nos surge a la hora de llamar al método *ejecutar()* que, no sabe cuál es el Servicio del contrato puesto que no se ha inicializado aún. Puesto que la clase *Proveedor* debe tener un servicio obligatoriamente por la multiplicidad de la relación, podemos crear un método getServicio() en la clase Proveedor para saber qué servicio oferta ese proveedor. Además, como el contrato tiene que pertenecer también a un Proveedor, podemos acceder a getServicio() fácilmente desde contrato. En el constructor de *Proveedor* se debe guardar una referencia a un Servicio, al igual que en contrato, que tendrá que tener una referencia a *Proveedor*, entonces, la implementación del método *ejecutar()* es simplemente *return proveedor.getServicio();*.

**Ejercicio 2:**



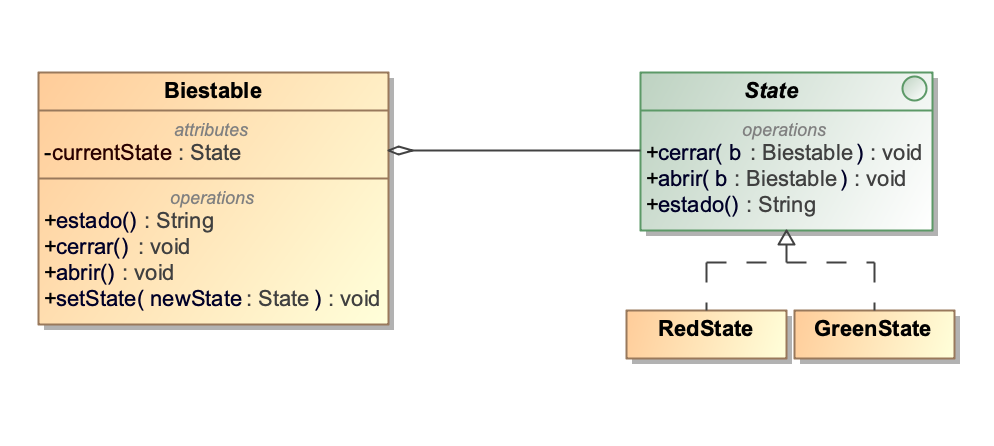
Para resolver este ejercicio utilizaremos el Patrón Estrategia, puesto que tenemos varias maneras de ordenar los correos mediante distintas estrategias (cada manera de ordenarlo equivale a una estrategia). Para ello crearemos una Interfaz con el método *before()*, que se implementará de distintas maneras dependiendo de la estrategia a usar. En la clase *MailBox* se guardará una referencia a esta Interfaz, dónde recibirá el tipo de estrategia a seguir en el constructor de *MailBox.* De la Interfaz heredarán tantas clases como maneras de ordenar los correos, en este caso hay 4 maneras, por prioridad, por Emisor, por fecha y por asunto, por tanto se crearán 4 clases que implementarán esa estrategia de ordenación.

**Ejercicio 3:**

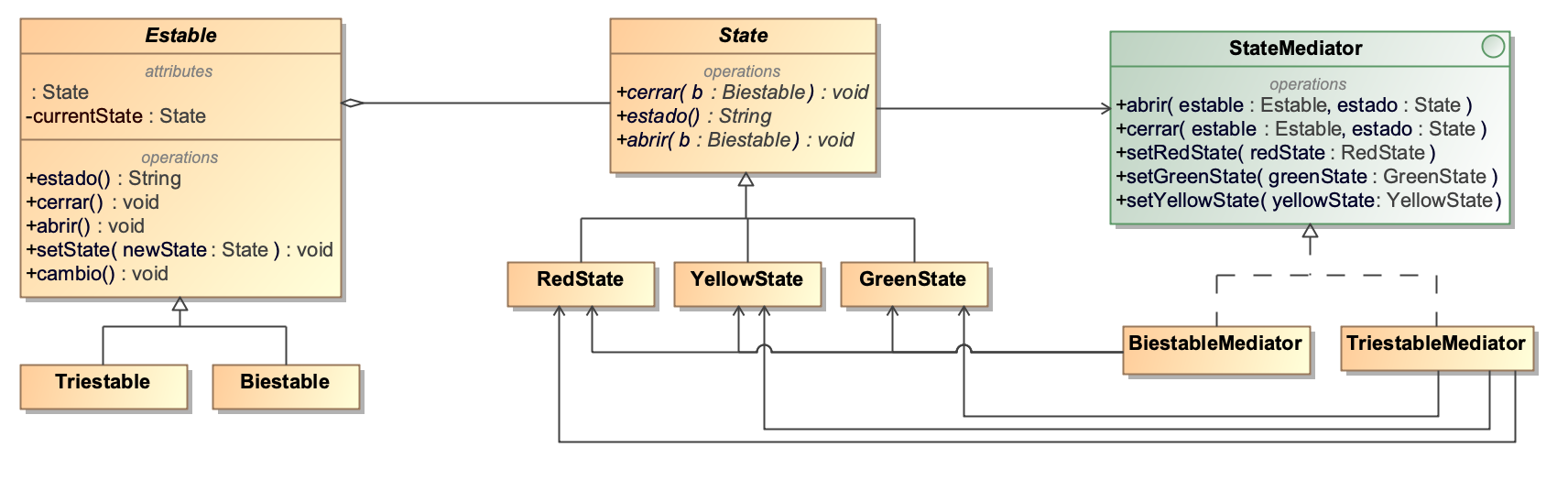
1. Java no soporta exportación selectiva, símplemente tiene 4 tipos de visibilidades, pero ninguna de ellas te permite seleccionar qué clases pueden acceder o no a según qué operaciones o atributos. Una manera de implementar una *exportación selectiva* en java es mediante un proxy de protección.
2. Como hemos comentado en el apartado anterior, haremos uso del patrón de diseño Proxy, en concreto, la variante del proxy de protección, que controla los accesos a nuestra clase original. La ventaja de esta solución es que te permite usar la exportación selectiva en java, y por tanto, tienes las mismas ventajas que en Eiffel, sin embargo, a la hora de añadir nuevas clases y nuevos métodos, se tendrían que añadir también reglas al proxy, por tanto, no es tan inmediato como en Eiffel. Además de eso, por mucho que exista el proxy, se puede “falsificar” el caller de *XProxyInterface* pasando por argumento un objeto que no sea el que realmente está llamando a la rutina. No hay manera de arreglar este “error” puesto que todos los componentes de una Interfaz deben ser públicos, y tampoco sirve realizarlo con una clase abstracta puesto que los componentes abstractos también deben ser públicos.

**Ejercicio 4:**

1. Para este ejercicio el patrón a usar es el patrón estado, puesto que para representar un autómata finito determinista (en adelante afd) es la mejor opción. El siguiente diagrama en UML representa la solución a desarrollar. Guardamos en la clase *Biestable* una referencia al estado actual para poder definir el comportamiento del afd puesto que cada vez que se llame a cerrar, abrir o estado, su comportamiento dependerá del estado en el que se encuentre en ese instante. En el momento en el que se llama a una de las operaciones de *State*, si existe transición, transita al nuevo estado asignando con el método *Biestable::setState()* la instancia del estado al que transita..



1. Para resolver este problema vamos a usar el patrón mediador. Para ello crearemos 2 mediadores, uno para el biestable y otro para el triestable. Los mediadores se encargarán a partir de ahora de las transiciones entre estados. Además, hay que convertir la antigua clase Biestable en una clase abstracta, de la que heredarán 2 clases nuevas, Biestable y Triestable. Los estados delegan en los mediadiores las transiciones para simplificar el código.



1. Para resolver el nuevo problema volvemos a utilizar un mediador usando un semáforo. Así cuando llamemos al método cambio, simplifica cambiar de tipo de estable. Sin embargo, vamos a tener que añadir en la clase *State* un método abstracto para asignar el mediador al estado y facilitar el cambio de un tipo de estable a otro.

