Práctica 4.

1. Los interfaces Selectivos

El lenguaje Eiffel dispone de un mecanismo de exportación selectiva mediante el cual una clase puede indicar, para cada una de sus características (p.ej. operaciones), qué otras clases pueden acceder a ellas.

```
class X
   feature -- Características públicas
   rutina1 is
         do ... end;
   feature {A, B} --Características que pueden invocar objetos de clases A y B
   rutina2(y: REAL) is
         do ... end;
   feature (A, C} -- Características que pueden invocar objetos de las clases A y C
   rutina3(i: INTEGER) : BOOLEAN is
         do ... end;
   feature {} -- Características visibles sólo para la propia clase X y sus derivadas
   rutina4 : INTEGER is
         do ... end;
   end
```

Esto permite que la clase X presente una interfaz protegida, tres interfaces distintas a cada uno de sus clientes A, B y C y una interfaz pública para el resto de las clases del sistema. Por el contrario, otros lenguajes como Java no disponen de mecanismos de exportación selectiva tan precisos, lo que puede causar problemas en ocasiones.

a) Describir los mecanismos de exportación selectiva en Java y compararlo con el de Eiffel.

b) Definir un patrón en Java para conseguir que una clase exporte sus funciones entre sus clientes, de manera realmente selectiva, tal y como se consigue en Eiffel. Utilizando el patrón, proporcionar una implementación para la clase X descrita. Discutir las ventajas e inconvenientes de la solución propuesta.

2. Triestables

Supongamos que necesitamos implementar un dispositivo software Biestable similar a un semáforo. Tal como muestra el diagrama de estados de la Figura (a), inicialmente el dispositivo estará en el estado Rojo, pasando a Verde tras recibir el mensaje `abrir()`. Desde este último estado, el dispositivo volverá a Rojo al recibir el mensaje `cerrar()`. El dispositivo será además capaz de reaccionar al mensaje `estado()`, devolviendo la cadena “cerrado” cuando está en Rojo y “abierto” cuando está en Verde.

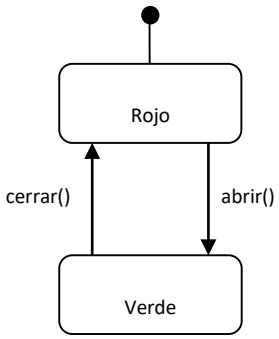


Figura (a). Biestable

a) Descríbase un patrón de diseño (mediante diagramas UML, pseudocódigo y las explicaciones textuales adecuadas) que permita implementar de manera satisfactoria dispositivos que, como el mencionado, reaccionan de forma distinta ante el mismo mensaje, dependiendo de su estado interno. Muéstrese el código Java correspondiente a una particularización de dicho patrón de diseño para implementar el dispositivo Biestable descrito.

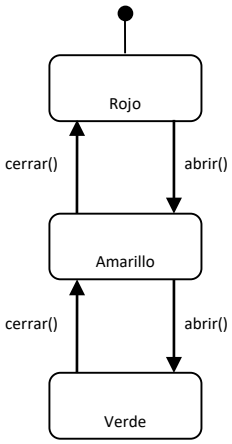


Figura (b). Triestable

b) Supongamos ahora que deseamos implementar un dispositivo Triestable. Tal como muestra la Figura (b), un Triestable incorpora un estado intermedio Amarillo en el que la respuesta al método `estado()` será la cadena "precaución". Amplíese la solución propuesta en el apartado anterior para reutilizar en la medida de lo posible el código ya desarrollado, teniendo en cuenta que en nuestro sistema deberemos disponer tanto de dispositivos Biestable como Triestable. Discuta las ventajas e inconvenientes de la solución propuesta, en particular comentando si la reutilización de código corresponde a lo que cabría esperar dadas las semejanzas de comportamiento de ambos dispositivos. Si conoce algún patrón de diseño que sea de utilidad para implementar la ampliación requerida, justifique su uso y documéntelo de nuevo con diagramas UML, pseudocódigo y las explicaciones textuales adecuadas.

c) Supongamos por último que necesitamos realizar una nueva ampliación de nuestro sistema, en el que a la recepción de un mensaje `cambio()`, un dispositivo Biestable pasará a partir de ese momento a comportarse como un Triestable. Para ello, deberemos efectuar una fase de transición, tal como muestran las flechas discontinuas de la Figura (c), en el que pasaremos del diagrama de estados inicial (a la izquierda, con dos estados) al final (a la derecha, con tres estados) al recibir por primera vez los mensajes `abrir()` o `cerrar()` tras el mensaje `cambio()`. Discútase la mejor forma de modificar la solución propuesta en los apartados anteriores para tener en cuenta el nuevo requisito, de nuevo intentando la mayor reutilización de código posible. Si conoce algún patrón de diseño que sea de utilidad para implementar la ampliación requerida, justifique su uso y documéntelo de nuevo con diagramas UML, pseudocódigo y las explicaciones textuales adecuadas.

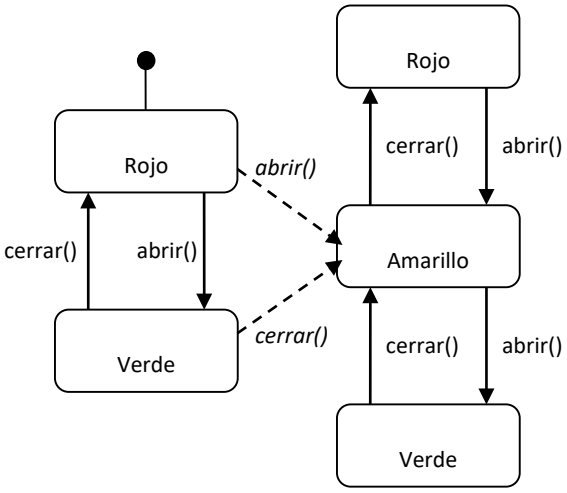
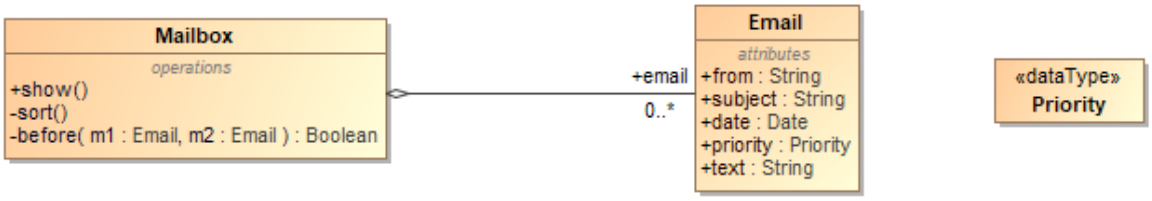


Figura (c). Transición de Biestable a Triestable

3. Cliente de correo e-look

e-look es un programa cliente de correo electrónico con el que el usuario puede gestionar sus mensajes de e-mail, almacenados en un *mailbox*.



Una de las funciones de *e-look* es la de permitir visualizar (método `show()`) los e-mails de un *`mailbox`* ordenados por diferentes criterios (from, subject, date, priority, …). Para ello, hemos implementado en la clase `Mailbox` una operación `sort()`, utilizando el llamado método de la burbuja:

```
private void sort() {
   for ( int i = 2; i <= email.size(), i++ )
       for ( int j = email.size(); j >= i; j-- )
           if ( before(email.at(j),email.at(j-1)) )
               // intercambiar los mensajes j y j-1
               ...
   }
```

que, como vemos, se basa a su vez en una operación `before()`. Como es lógico, el resultado que devuelva `before` dependerá del criterio de ordenación elegido para visualizar el *`mailbox`* (from, subject, date, priority, …), por lo que sería necesario parametrizar de alguna manera `sort()` de acuerdo a dicho criterio

Describe, por medio de diagramas, esquemas de código y descripciones textuales, un patrón de diseño que nos permita resolver la situación presentada, permitiendo incluso cambiar de un criterio de ordenación a otro mientras el usuario utiliza e-look*, y de forma que sea fácilmente extensible (por ejemplo para ordenar por otros criterios aparte de los indicados). Muestra de forma esquemática la implementación en Java del patrón propuesto al problema descrito.*