Práctica 4

Nicolás Álvarez Romero
Víctor Madrona Cisneros
Nicolás Zhilie Zhao
Alfonso González Ortiz

1. Los interfaces Selectivos

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

Para ver el mecanismo de exportación selectiva en Java, vamos a ver las diferencias primero entre los modificadores de acceso:

- **Privado**: A una clase privada no se puede acceder desde ninguna otra clase
- **No declarado**: Si no se declara el modificador de acceso de la clase, la clase es visible y accesible desde clases del mismo paquete
- **Protected**: Una clase protected es visible y accesible desde clases del mismo paquete, o por una subclase de su clase en otro paquete.
- **Público**: Si declaramos una clase con el modificador public, la clase es visible para cualquier clase, y se pueden invocar sus métodos desde cualquier clase

En java la forma de realizar exportación selectiva es a través de los paquetes. Si quieres que una clase A pueda acceder a una clase B, declaras la clase B como protected y metes ambas clases en el mismo paquete.

En Eiffel es más preciso su mecanismo de exportación selectiva. Gracias a las *features* podemos definir dentro de una misma clase características que sean accesibles para todas las clases, características que sean accesibles para algunas clases y características que no sean accesibles para ninguna clase.
En cada *feature* defines para qué clases quieres que sean accesibles las características definidas en dicho *feature*

Entonces la diferencia entre ambos mecanismos de exportación selectiva es que en Java todas las características de la clase protected son accesibles desde las clases del mismo paquete. Mientras que en Eiffel puedes definir características accesibles desde una clase A, y en la misma clase definir otras características accesibles desde una clase B

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.

Para resolver este problema usaremos el patrón proxy, con este patrón tendremos un representante de la clase X, de forma que se encargará de seleccionar qué clases pueden acceder a las características y cuáles no. Por ejemplo al método rutina2 sólo podrán acceder las clases A y B, si es una de estas clases la que está accediendo a dicho método el proxy llamará a X para ejecutar la rutina2.

La gran ventaja de este método es que nos permite realizar exportación selectiva sin necesidad de agrupar las clases en distintos paquetes. Con este método tenemos la ventaja de exportación selectiva de Eiffel, una clase con distintos métodos donde cada método es solo accesible por las clases que especifiquemos. Esto sin el patrón definido no es posible.

La desventaja de la solución propuesta frente a Eiffel es que tenemos que crear una nueva clase auxiliar (Proxy) para comprobar qué clases pueden acceder a cada método. Dentro de esta clase tenemos que hacer un método auxiliar por cada método de la clase X, mientras que en Eiffel solo hay que incluir en la cabecera del *feature* las clases que pueden acceder a los métodos que serán definidos dentro del *feature*.

Código Java:

```
package Ejc1.Pack1;

import Ejc1.A;
import Ejc1.B;
import Ejc1.C;

public class Proxy {
  private X x;

  public Proxy(X x){
     this.x = x;
  }
  public void rutina1(){
     x.rutina1();
  }

  public void rutina2(double y, Object o){
     try {
        if(o instanceof A || o instanceof B){
           x.rutina2(y);
        }else {
           System.out.println("No autorizado");
           throw new Exception();
        }
     } catch (Exception e) {
        e.printStackTrace();
     }
  }
```

```
  public Boolean rutina3(int i, Object o){
     try {
        if (o instanceof A || o instanceof C) {
           return x.rutina3(i);
        } else {
           System.out.println("No autorizado");
           throw new Exception();
        }
     }
     catch(Exception e){
        e.printStackTrace();
     }
     return null;
  }

}
```

```
package Ejc1.Pack1;

public class X {

  public X(){

  }
  public void rutina1(){
     System.out.println("rutina1");
  }

  protected void rutina2(double y){
     System.out.println("rutina2");
  }

  protected boolean rutina3(int i){
     System.out.println("rutina3");
     return true;
  }

  private int rutina4(){
     System.out.println("rutina4");
     return 0;
  }
}
```

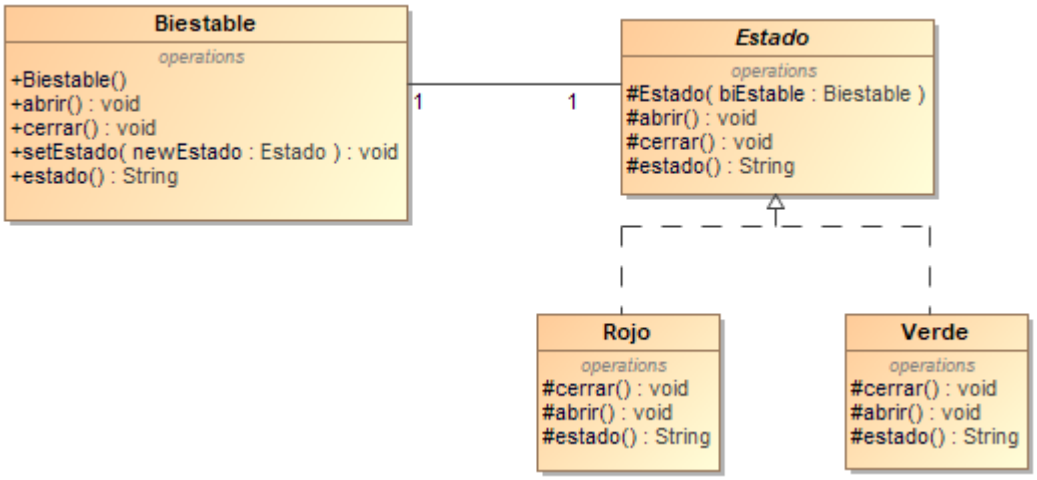
```
package Ejc1;

public class C {
}
```

```
package Ejc1;

public class B {
}
```

```
package Ejc1;

public class A {
}
```

2. Triestables

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.



Resolveremos el problema usando el patrón Estado, este patrón nos permite modificar el cuerpo de una operación en función al estado del objeto.

El biestable tiene un estado, este estado puede ser **Rojo** o **Verde**, estos estados implementan las mismas operaciones que el biestable de forma que cuando el biestable emplee alguna de estas se ejecutará en función al estado que tenga en dicho momento. En nuestro caso cuando el biestable esté en el estado verde y se ejecute cerrar() pasará a rojo, y si está en rojo y se ejecuta abrir() pasará a verde.

Los estados solo se pueden construir desde la clase biestable.

Código Java:

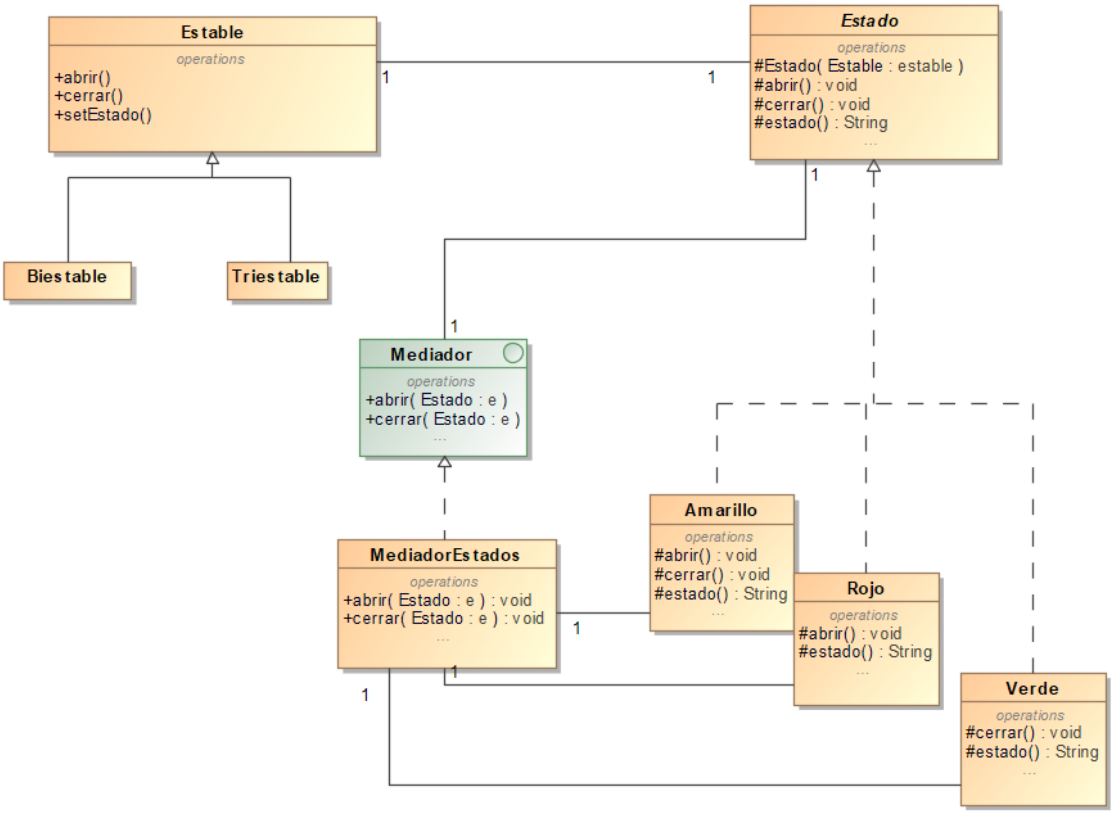
```
public class Amarillo extends Estado{
  protected Amarillo(Biestable b) {
     super(b);
```

```
    }

    @Override
    protected void abrir() {
        biest.setEstado(new TriVerde(biest));
    }

    @Override
    protected void cerrar() {
        biest.setEstado(new TriRojo(biest));
    }

    @Override
    protected String estado() {
        return "Precaucion";
    }
}

public class Rojo extends Estado{
    protected Rojo(Biestable b) {
        super(b);
    }

    @Override
    protected void abrir() {
        biest.setEstado(new Verde(biest));
    }

    @Override
    protected void cerrar() {

    }

    @Override
    protected String estado() {
        return "cerrado";
    }
}

public class Verde extends Estado{
    protected Verde(Biestable b) {
        super(b);
    }

    @Override
    protected void abrir() {

    }
```

```
    @Override
    protected void cerrar() {
        biest.setEstado(new Rojo(biest));
    }

    @Override
    protected String estado() {
        return "abierto";
    }
}

public class Biestable {

    private Estado estado;

    protected Biestable(){
        estado = new Rojo(this);
    }

    protected void setEstado(Estado nuevoEst){
        estado = nuevoEst;
    }

    protected void abrir(){
        estado.abrir();
    }

    protected void cerrar(){
        estado.cerrar();
    }

    protected String estado(){
        return estado.estado();
    }

}

public abstract class Estado {
    Biestable biest;

    protected Estado(Biestable b){
        biest = b;
    }

    protected abstract void abrir();
    protected abstract  void cerrar();
    protected abstract String estado();
```

}

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 todo 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.



Aplicamos el patrón mediador para que se encargue él de comprobar que tipo de estable está activo, de esta forma podemos añadir estados sin preocuparnos por la implementación de sus métodos ya que de esto se encargará el mediador.

Código Java:

```
public class Amarillo extends Estado {
  private Mediador mediador;

  protected Amarillo(Estable b, Mediador m) {
```

```
        super(b);
        setMediador(m);
    }

    public void setMediador(Mediador m){mediador=m;}

    @Override
    protected void abrir() {
        mediador.abrir(estado());
    }

    @Override
    protected void cerrar() {
        mediador.cerrar(estado());
    }

    @Override
    protected String estado() {
        return "Precaucion";
    }

}
public class Biestable extends Estable{
    public Biestable(){
        super();
    }
}
public abstract class Estable {

    private Estado estado;


    public Estable(){
        estado = new Rojo(this, new MediadorConcreto());
    }

    protected void setEstado(Estado nuevoEst){
        estado = nuevoEst;
    }

    protected void abrir(){
        estado.abrir();
    }

    protected void cerrar(){
        estado.cerrar();
    }
public abstract class Estado {
```

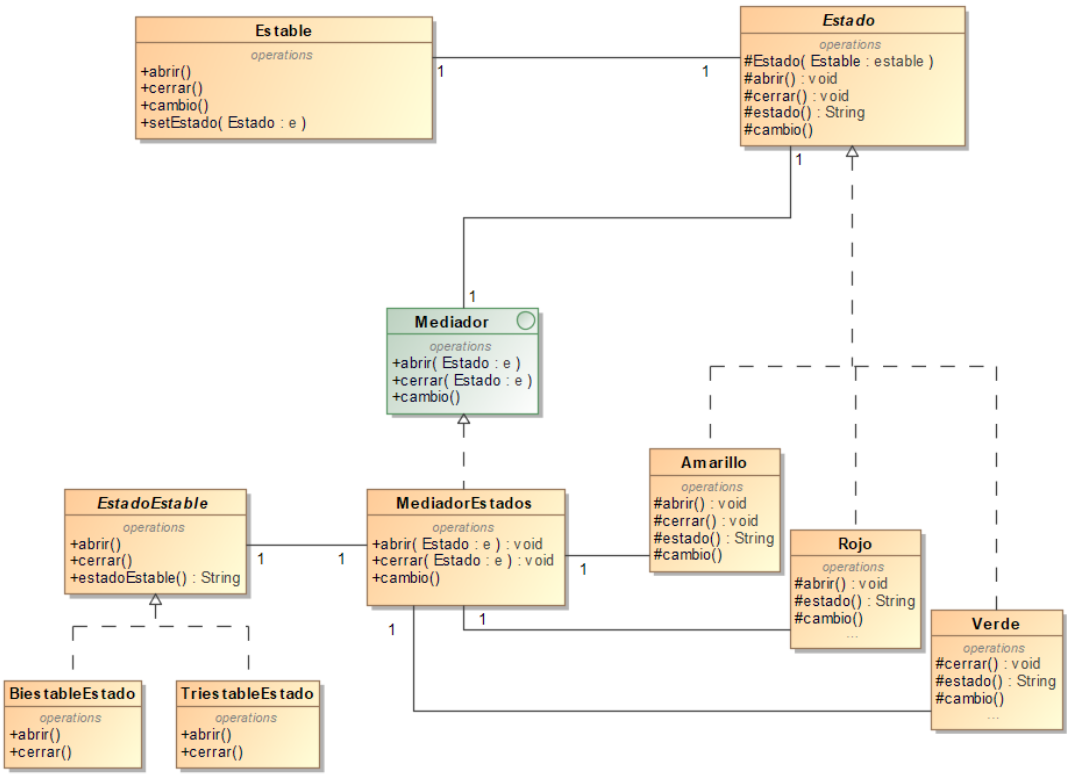
```
    Estable est;

    protected Estado(Estable e){
        est = e;
    }

    protected abstract void abrir();
    protected abstract  void cerrar();
    protected abstract String estado();

}

    protected String estado(){
        return estado.estado();
    }

}

public interface Mediador {
    void abrir(String e);
    void cerrar(String e);


}


public class MediadorConcreto implements Mediador {

    private Estado est;

    protected MediadorConcreto(){

    }
    @Override
    public void abrir(String e) {
        if (est.est instanceof Biestable){
            est.est.setEstado(new Verde(est.est, this));

        }else if(est.est instanceof Triestable){
            if(e.equals("cerrado")){
                est.est.setEstado(new Amarillo(est.est, this));
            }else if(e.equals("precaucion")){
                est.est.setEstado(new Verde(est.est, this));
            }
        }
    }
```

```
    @Override
    public void cerrar(String e) {
        if (est.est instanceof Biestable){
            est.est.setEstado(new Rojo(est.est, this));

        }else if(est.est instanceof Triestable){
            if(e.equals("abierto")){
                est.est.setEstado(new Amarillo(est.est, this));
            }else if(e.equals("precaucion")){
                est.est.setEstado(new Rojo(est.est, this));
            }
        }
    }

    protected Estado getEstado(){
        return est;
    }

    protected void setEstadoEstable(Estado e){
        est.est.setEstado(e);
    }

}

public class Rojo extends Estado {
    private Mediador mediador;

    protected Rojo(Estable b, Mediador m) {
        super(b);
        setMediador(m);
    }
    public void setMediador(Mediador m){mediador=m;}
    @Override
    protected void abrir() {
        mediador.abrir(estado());
    }

    @Override
    protected void cerrar() {

    }

    @Override
    protected String estado() {
        return "cerrado";
    }

}
```

```
public class Triestable extends Estable{
   public Triestable(){
      super();
   }
}
public class Verde extends Estado {

   private Mediador mediador;

   protected Verde(Estable b, Mediador m) {
      super(b);
      setMediador(m);
   }
   public void setMediador(Mediador m){mediador=m;}
   @Override
   protected void abrir() {

   }

   @Override
   protected void cerrar() {
      mediador.cerrar(estado());
   }

   @Override
   protected String estado() {
      return "abierto";
   }


}
```

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, y viceversa. 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. Discútase así mismo cómo tratar la situación de un Triestable que recibe un mensaje cambio() cuando se encuentra en estado Amarillo. 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.



Aplicamos el patrón mediador para abstraer la implementación de los estados del estable y que se encargue el mediador. Este mediador tiene también 2 estados el de biestable y el de triestable, en función al estado transitará de distintas formas en los semáforos.

Biestable : Rojo → Verde, Verde→Rojo
Triestable: Rojo→Amarillo→Verde, Verde→Amarillo→Rojo

Gracias a los estados del mediador podremos añadir en cualquier momento otro tipo de estable añadiendo un EstadoEstable más.

Código Java:

```
public class Amarillo extends Estado {
  private Mediador mediador;

  protected Amarillo(Estable b, Mediador m) {
     super(b);
     setMediador(m);
  }

  public void setMediador(Mediador m){mediador=m;}
```

```
    @Override
    protected void abrir() {
        mediador.abrir(estado());
    }

    @Override
    protected void cerrar() {
        mediador.cerrar(estado());
    }

    @Override
    protected String estado() {
        return "Precaucion";
    }

    @Override
    public void cambio() {

    }
}
public class Estable {

    private Estado estado;


    public Estable(){
        estado = new Rojo(this, new MediadorConcreto());
    }

    protected void cambio(){
        estado.cambio();
    }

    protected void setEstado(Estado nuevoEst){
        estado = nuevoEst;
    }

    protected void abrir(){
        estado.abrir();
    }

    protected void cerrar(){
        estado.cerrar();
    }

    protected String estado(){
        return estado.estado();
```

```
  }

}
public abstract class Estado {
  Estable est;

  protected Estado(Estable e){
     est = e;
  }

  protected abstract void abrir();
  protected abstract  void cerrar();
  protected abstract String estado();

  public abstract void cambio();
}
public class EstadoBiestable extends EstadoEstable{

  protected EstadoBiestable(MediadorConcreto m) {
     super(m);
  }

  @Override
  protected void cerrar(String e) {
     super.mediador.setEstadoEstable(new Rojo(super.mediador.getEstado().est,
super.mediador));
  }

  @Override
  protected void abrir(String e) {
     super.mediador.setEstadoEstable(new Verde(super.mediador.getEstado().est,
super.mediador));
  }

}
public abstract class EstadoEstable {
  MediadorConcreto mediador;

  protected EstadoEstable(MediadorConcreto m){ mediador=m;}

  protected abstract void cerrar(String e);

  protected abstract void abrir(String e);
}
public class EstadoTriestable extends EstadoEstable{

  protected EstadoTriestable(MediadorConcreto m) {
     super(m);
```

```
  }

  @Override
  protected void cerrar(String e) {
     switch (e) {
        case "precaucion":
           super.mediador.setEstadoEstable(new Rojo(super.mediador.getEstado().est,
super.mediador));
           break;
        case "abierto":
           super.mediador.setEstadoEstable(new Amarillo(super.mediador.getEstado().est,
super.mediador));
           break;
     }

  }

  @Override
  protected void abrir(String e) {
     switch (e) {
        case "precaucion":
           super.mediador.setEstadoEstable(new Verde(super.mediador.getEstado().est,
super.mediador));
           break;
        case "abierto":
           super.mediador.setEstadoEstable(new Amarillo(super.mediador.getEstado().est,
super.mediador));
           break;
     }
  }

}
public interface Mediador {
  void abrir(String e);
  void cerrar(String e);

  void cambio();
}
public class MediadorConcreto implements Mediador{
  private EstadoEstable estado;
  private Estado est;

  protected MediadorConcreto(){

     estado = new EstadoBiestable(this);
  }
  @Override
  public void abrir(String e) {
```

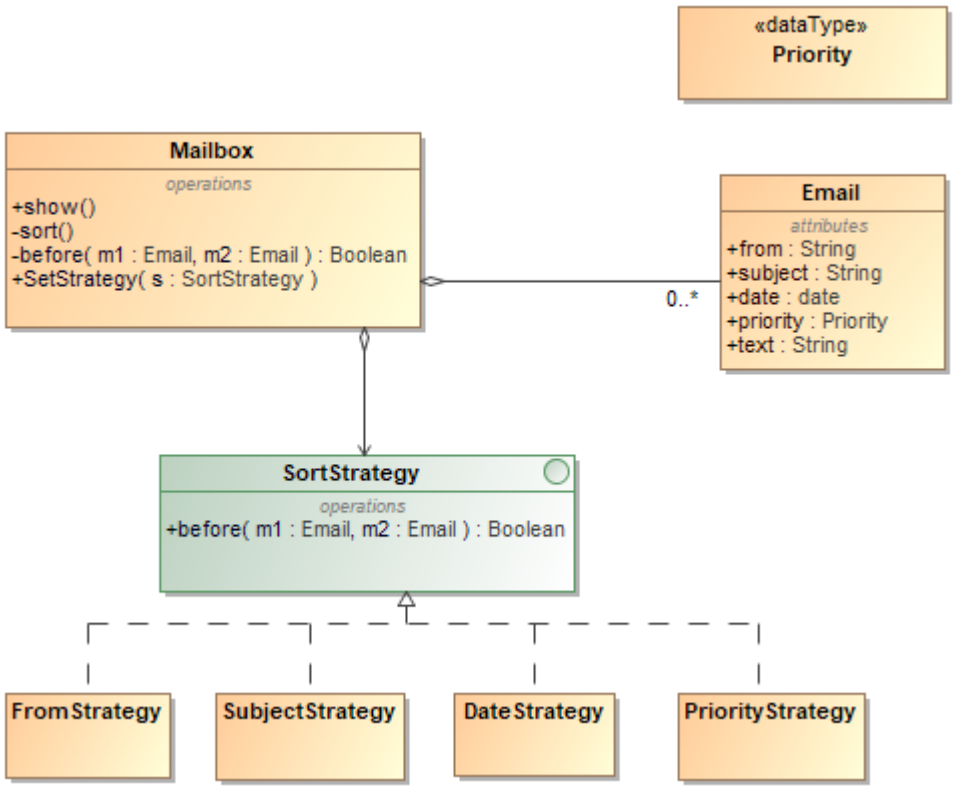
```
        estado.abrir(e);
    }

    @Override
    public void cerrar(String e) {
        estado.cerrar(e);
    }

    @Override
    public void cambio() {
        if(estado instanceof EstadoTriestable){
            estado = new EstadoBiestable(this);
        }else{
            estado = new EstadoTriestable(this);
        }
    }

    protected Estado getEstado(){
        return est;
    }
    protected void setEstadoEstable(EstadoEstable e){
        estado=e;
    }

    protected void setEstadoEstable(Estado e){
        est.est.setEstado(e);
    }

}
public class Rojo extends Estado {
    private Mediador mediador;

    protected Rojo(Estable b, Mediador m) {
        super(b);
        setMediador(m);
    }
    public void setMediador(Mediador m){mediador=m;}
    @Override
    protected void abrir() {
        mediador.abrir(estado());
    }

    @Override
    protected void cerrar() {

    }

    @Override
```

```
    protected String estado() {
        return "cerrado";
    }

    @Override
    public void cambio() {

    }
}
public class Verde extends Estado {

    private Mediador mediador;

    protected Verde(Estable b, Mediador m) {
        super(b);
        setMediador(m);
    }
    public void setMediador(Mediador m){mediador=m;}
    @Override
    protected void abrir() {

    }

    @Override
    protected void cerrar() {
        mediador.cerrar(estado());
    }

    @Override
    protected String estado() {
        return "abierto";
    }

    @Override
    public void cambio() {
        mediador.cambio();
    }
}
```

3. Cliente de correo e-look

Descríbase, 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). Mostrar de forma esquemática la implementación en Java del patrón propuesto al problema descrito.



Para resolver este ejercicio usaremos el patrón estrategia, para implementarlo creamos una interfaz con la operación before() con la que se decidirá el criterio de ordenación. A esta interfaz la implementan las distintas estrategias de ordenación, estas clases definirán el método before(), cada una con su propia estrategia.

De esta manera cuando se muestran los emails en el mailbox se organizará en función a la estrategia de ordenación indicada. Podremos cambiar la estrategia de ordenación del mailbox cambiando el sortStrategy de la clase Mailbox con el método SetStrategy(SortStrategy : s).

Además este patrón nos permite añadir en cualquier momento una nueva estrategia de ordenación, tan solo tendremos que crear una clase que implemente la interfaz SortStrategy.

Código Java:

```
public class DateStrategy implements SortStrategy{
   @Override
   public boolean before(Email m1, Email m2) {
      return m1.date.compareTo(m2.date) < 0;

   }
}
```

```
public class Email {
  protected String from, subject, text;
  protected Date date;
  protected Priority priority;

  protected Email(String from, String subject, Date date, Priority priority, String text){
    this.from = from;
    this.subject = subject;
    this.date = date;
    this.priority = priority;
    this.text = text;
  }

}
public class FromStrategy implements SortStrategy{
  @Override
  public boolean before(Email m1, Email m2) {
    return
m1.from.toLowerCase(Locale.ROOT).compareTo(m2.from.toLowerCase(Locale.ROOT)) <
0;
  }
}

public class Mailbox {

  private ArrayList<Email> listEmail;
  private SortStrategy strategy;

  protected Mailbox(){
    listEmail = new ArrayList<>();
  }

  protected void show(){
    System.out.println(listEmail);
  }

  private void sort(){
    for(int i=2; i<listEmail.size(); i++){
      for(int j=listEmail.size(); j>=i; j--){
        Email e1 = listEmail.get(j);
        Email e2 = listEmail.get(j-1);
        if(before(e1,e2)){
          Email aux = e1;
          listEmail.add(j,e2);
          listEmail.add(j-1,aux);
        }
      }
    }
```

```
  }

  private Boolean before(Email m1, Email m2){
    return strategy.before(m1,m2);
  }
}
public class Priority implements Comparable<Priority>{
  private int priority;

  protected Priority(int priority){
    this.priority = priority;
  }

  public boolean equals(Object o){
    return (o instanceof Priority) && ((Priority) o).priority == priority;
  }

  public int hashCode(){
    return Objects.hashCode(priority);
  }

  @Override
  public int compareTo(Priority p) {
    int res = 0;
    if(priority<p.priority){
      res = -1;
    }else if(priority<p.priority){
      res = 1;
    }else{
      res = 0;
    }
    return res;
  }
}
public class PriorityStrategy implements SortStrategy{
  @Override
  public boolean before(Email m1, Email m2) {
    return m1.priority.compareTo(m2.priority) < 0;
  }
}
public interface SortStrategy {
  boolean before(Email m1, Email m2);
}
public class SubjectStrategy implements SortStrategy{
  @Override
  public boolean before(Email m1, Email m2) {
```

```
        return
m1.subject.toLowerCase(Locale.ROOT).compareTo(m2.subject.toLowerCase(Locale.ROOT
)) < 0;

    }
}
```