Practica 4

Practica realizada por Alba Ruiz Gutiérrez, Pablo Pardo Fernández, Pablo Ruiz Galiánez, Jorge Velázquez Jiménez y Pablo Robles

1.Los interfaces selectivos

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

Java no tiene un mecanismo específico de "exportación selectiva". Sin embargo podemos definir paquetes y modificar el acceso para obtener un resultado similar.

Un paquete en Java es un mecanismo de organización que se utiliza para agrupar clases y interfaces relacionadas. Este puede contener clases, interfaces, subpaquetes u otros recursos.

Java utiliza el modificador de acceso para controlar la visibilidad de clases, métodos y variables. Los modificadores de acceso más comunes son public, private, protected y el acceso predeterminado, que se usa cuando no se especifica ningún otro. Estos modificadores determinan la accesibilidad de los miembros de una clase.

- **Public**: accesible desde cualquier clase.
- **Private**: accesible dentro de la misma clase.
- **Protected**: accesible dentro de la misma clase, paquete y subclases.
- **Sin modificador (predeterminado)**: accesible dentro del mismo paquete.

Sin embargo en Eiffel se define un mecanismo de exportación selectiva específico mediante el cual una clase puede indicar, para cada una de sus características, que otras clases pueden acceder a ella. De tal manera podremos tener que una misma clase represente distintas interfaces, como por ejemplo interfaces protegidas, interfaces particulares para sus clientes e interfaces públicas.

b) Implementación de la clase usando un patrón en Java

El patrón Representante (Proxy) es una estrategia que nos permite controlar el acceso a ciertos componentes de nuestro sistema de manera similar al mecanismo de exportación selectiva de Eiffel. En este patrón, creamos una interfaz llamada "X", la cual es implementada por dos clases: "XService" y "XProxy".

La clase "XService" contiene la lógica real de la funcionalidad que queremos proporcionar. Por otro lado, la clase "XProxy" actúa como un representante de la clase service. Ambas clases implementan la interfaz "X". Cuando un cliente solicita acceder a la funcionalidad del servicio, interactúa con el "XProxy" en lugar de la clase "XService".

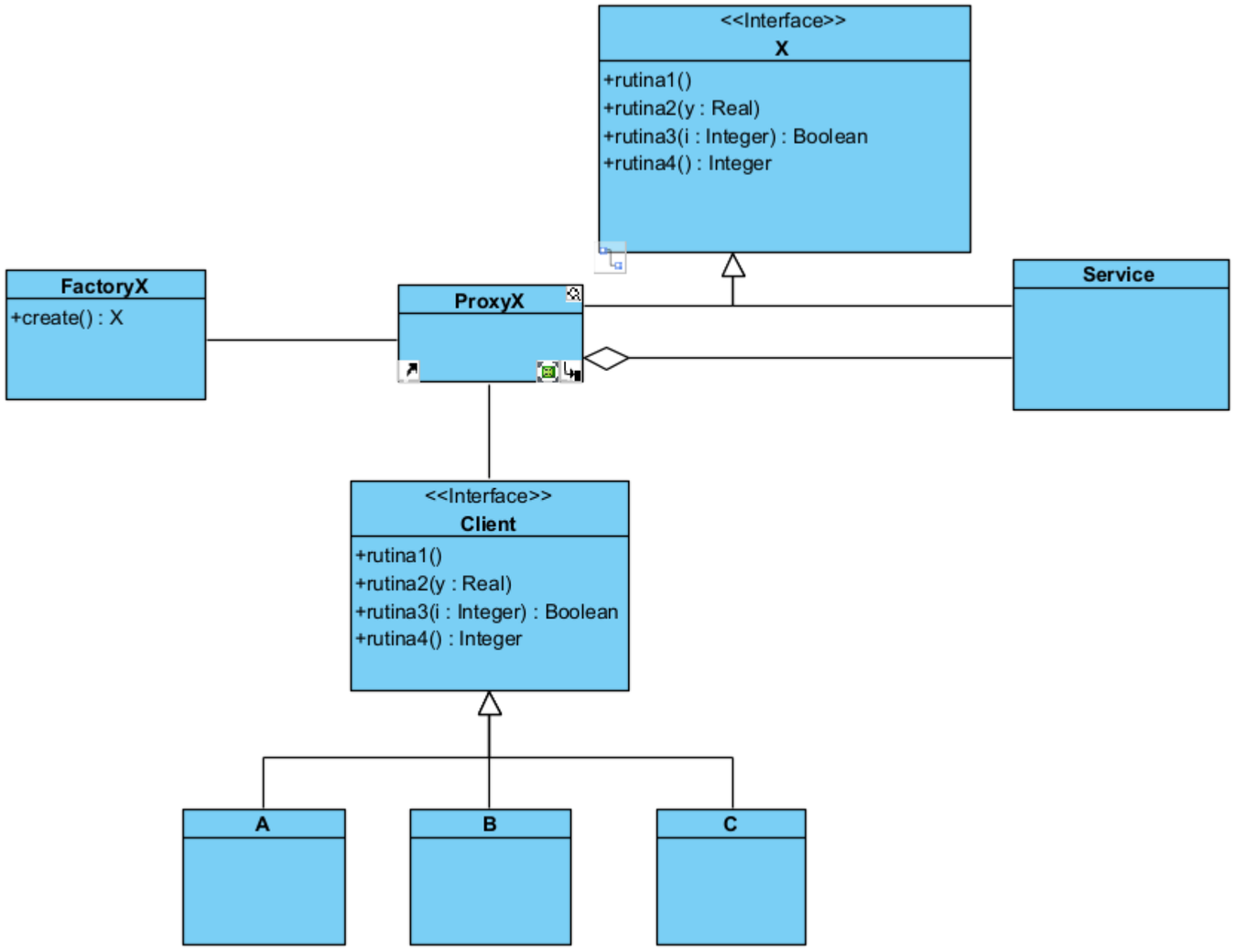
El "XProxy" se encarga de delegar la ejecución de los métodos al "XService", pero también controla el acceso a estos métodos. En otras palabras, el cliente interactúa con el representante (Proxy) sin conocer la complejidad interna de lo que sucede dentro del "XService". Este enfoque proporciona una capa adicional de control de acceso y permite gestionar el acceso a los componentes de manera más controlada.

En nuestro contexto, para restringir el acceso a los métodos mediante el Proxy, es necesario que el Proxy tenga conocimiento sobre el tipo de objeto que está invocando el método deseado. Además, queremos asegurarnos de que la creación del servicio representante (Proxy) se realice correctamente, pasándole la referencia al servicio real, y evitando que las clases clientes instancien directamente el servicio real.

Para abordar esto, recurrimos al patrón Método Factory. Este patrón implica la creación de una interfaz llamada "factory" que tiene un método de creación (método factory) para un producto concreto.

En nuestro caso, la "Fábrica" sería responsable de crear instancias del servicio representante (Proxy) junto con su correspondiente servicio real. Así, al utilizar el método fábrica, podemos controlar la creación del servicio representante y asegurarnos de que la referencia al servicio real se proporcione adecuadamente, evitando que las clases clientes instancien directamente el servicio real. Este enfoque mejora la modularidad y la gestión de la creación de objetos en nuestro sistema.

El modelo implementado es el siguiente:



El código para implementar este sistema es el siguiente:

1. **Client**

```
public interface Client {

    public void rutina1();
    public void rutina2();
    public void rutina3();
    public void rutina4();
}
```

2. **ClientA**

```
public class ClientA implements Client{

    private X x;

    public ClientA(){
        FactoryX factoryX = new FactoryX();
        x = factoryX.create(this);
    }

    @Override
    public void rutina1() {
        x.rutina1();
    }

    @Override
    public void rutina2() {
        x.rutina2();
    }

    @Override
    public void rutina3() {
        x.rutina3();
    }

    @Override
```

```
    public void rutina4() {
        x.rutina4();
    }
}
```

3. **ClientB**

```
public class ClientB implements Client{

    private X x;

    public ClientB() {
        FactoryX factoryX = new FactoryX();
        x = factoryX.create(this);
    }

    @Override
    public void rutina1() {
        x.rutina1();
    }

    @Override
    public void rutina2() {
        x.rutina2();
    }

    @Override
    public void rutina3() {
        x.rutina3();
    }

    @Override
    public void rutina4() {
        x.rutina4();
    }
}
```

4. **ClientC**

```
public class ClientC implements Client{

    private X x;

    public ClientC() {
        FactoryX factoryX = new FactoryX();
        x = factoryX.create(this);
    }

    @Override
    public void rutina1() {
        x.rutina1();
    }

    @Override
    public void rutina2() {
        x.rutina2();
    }

    @Override
    public void rutina3() {
        x.rutina3();
    }

    @Override
    public void rutina4() {
        x.rutina4();
    }
}
```

5. **FactoryX**

```
public class FactoryX {
    public X create(Object obj){
        return new XProxy(obj,new XService());
    }
}
```

6. **X**

```
public interface X {
    public void rutina1();
    public void rutina2();
    public void rutina3();
    public void rutina4();


}
```

7. **XProxy**

```
public class XProxy implements X{

    Object obj;
    XService xService;

    public XProxy(Object obj, XService xService) {
    this.obj = obj;
    this.xService = xService;
    }

    @Override
    public void rutina1() {
        System.out.println("Acceso a rutina 1 concedido ");
        xService.rutina1();
    }

    @Override
    public void rutina2() {

        if( obj instanceof ClientA ||  obj instanceof ClientB){
            System.out.println("Acceso a rutina 2 concedido ");
            xService.rutina2();
        }else{
            System.out.println("Acceso a rutina 2 denegado ");
        }
```

```
}

@Override
public void rutina3() {

    if( obj instanceof ClientA ||  obj instanceof ClientC){
        System.out.println("Acceso a rutina 3 concedido ");
        xService.rutina3();
    }else{
        System.out.println("Acceso a rutina 3 denegado ");
    }
}

@Override
public void rutina4() {
    if( obj instanceof X){
        System.out.println("Acceso a rutina 4 concedido ");
        xService.rutina4();
        }else{
            System.out.println("Acceso a rutina 4 denegado ");
        }

    }
}
```

8. **XService**

```
public class XService implements X{
    @Override
    public void rutina1() {
        System.out.println("Codigo rutina numero 1");
    }

    @Override
    public void rutina2() {
        System.out.println("Codigo rutina numero 2");
    }
```

```
    @Override
    public void rutina3() {
        System.out.println("Codigo rutina numero 3");
    }

    @Override
    public void rutina4() {
        System.out.println("Codigo rutina numero 4");
    }
}
```

9. Main

```
public class main {

    public static void main(String[] args){
        ClientA clientA = new ClientA();
        ClientB clientB = new ClientB();
        ClientC clientC = new ClientC();

        System.out.println("Cliente A: ");
        clientA.rutina1();
        clientA.rutina2();
        clientA.rutina3();
        clientA.rutina4();

        System.out.println("Cliente B: ");
        clientB.rutina1();
        clientB.rutina2();
        clientB.rutina3();
        clientB.rutina4();

        System.out.println("Cliente C: ");
        clientC.rutina1();
        clientC.rutina2();
        clientC.rutina3();
        clientC.rutina4();
```

```
    }
}
```

- Ventajas: Esta solución ofrece una manera clara de establecer restricciones de acceso, similar al mecanismo de exportación selectiva utilizado en Eiffel. Además, el patrón Método Fábrica garantiza que el objeto "X" devuelto sea creado adecuadamente con un representante que controle el acceso a sus métodos.
- Inconvenientes: Sin embargo, un desafío de esta implementación es que cada vez que se desee agregar una nueva restricción de acceso a una clase, es necesario modificar el código de la clase "XProxy". Este proceso puede resultar engorroso y potencialmente propenso a errores si se realizan cambios frecuentes en las restricciones de acceso.

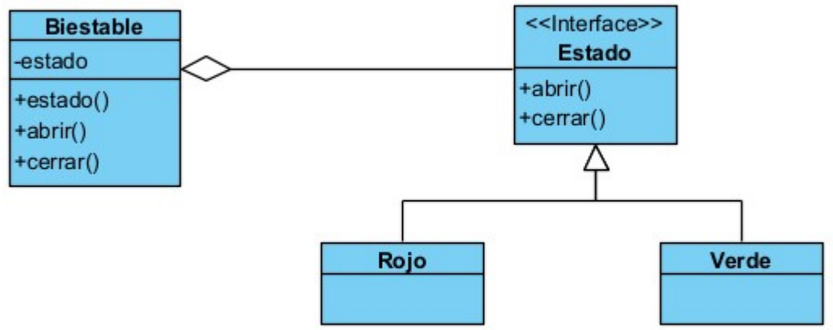
2.Triestables

a)Un patrón de diseño adecuado para nuestro ejercicio es el patrón de Estado el cual es utilizado para gestionar situaciones en las que el comportamiento de un sistema cambia según su estado interno. Este patrón implica la creación de una interfaz llamada "Estado", que es implementada por varias clases, cada una representando un estado específico del sistema y definiendo un comportamiento particular para las operaciones del sistema. El sistema, también conocido como contexto, delega la implementación de sus operaciones a un objeto que sigue la interfaz "Estado".

En términos más sencillos, el patrón Estado ayuda a organizar el comportamiento de un sistema en diferentes estados, donde cada estado tiene su propia forma de realizar ciertas operaciones. El sistema utiliza un objeto "Estado" para llevar a cabo estas operaciones, y este objeto puede cambiar dinámicamente a medida que el sistema atraviesa diferentes estados.

En las clases que implementan la interfaz "Estado", es común necesitar acceder a los atributos del sistema. Esto se puede lograr pasando el propio sistema como argumento a los métodos de las clases de estado o permitiendo que los estados tengan una referencia al sistema. Además, los estados se encargan de gestionar la transición del sistema al siguiente estado, modificando los atributos del sistema y actualizando la referencia al objeto de tipo "Estado".

Aplicando este patrón a nuestro ejercicio, obtenemos una clase Biestable y una interfaz estado que implementan dos clases, rojo y verde, la cual representa el estado en el que se encuentra el Biestable.



1. **Biestable**

```
package Java;

public class Biestable{
    Estado estado;
    public Biestable(){
        this.estado = new Rojo();
    }
    public void abrir(){
        this.estado.abrir(this);
    }
    public void cerrar(){
        this.estado.cerrar(this);
    }
    public void estado(){
        System.out.println(estado);
    }
}
```

2. **Rojo**

```
package Java;

public class Rojo implements Estado{
    private final String estado;

    public Rojo() {
        this.estado = "Rojo";
    }
    public void abrir(Biestable biestable){
        biestable.estado = new Verde();
    }
    public void cerrar(Biestable biestable){
        System.out.println("Ya esta cerrado");
    }
    @Override
    public String toString() {
        return estado;
    }
}
```

3. **Verde**

```
package Java;

public class Verde implements Estado{
    private final String estado;

    public Verde() {
        this.estado = "Verde";
    }
    public void abrir(Biestable biestable){
        System.out.println("Ya esta abierto");
    }
    public void cerrar(Biestable biestable){
        biestable.estado = new Rojo();
    }
    @Override
    public String toString() {
```

```
        return estado;
    }
}
```

4. **Estado**

```
package Java;

public interface Estado {
    //Metodos que van a tener que implementar las clases que implementen esta interfaz
    public void abrir(Biestable biestable);
    public void cerrar(Biestable biestable);
}
```

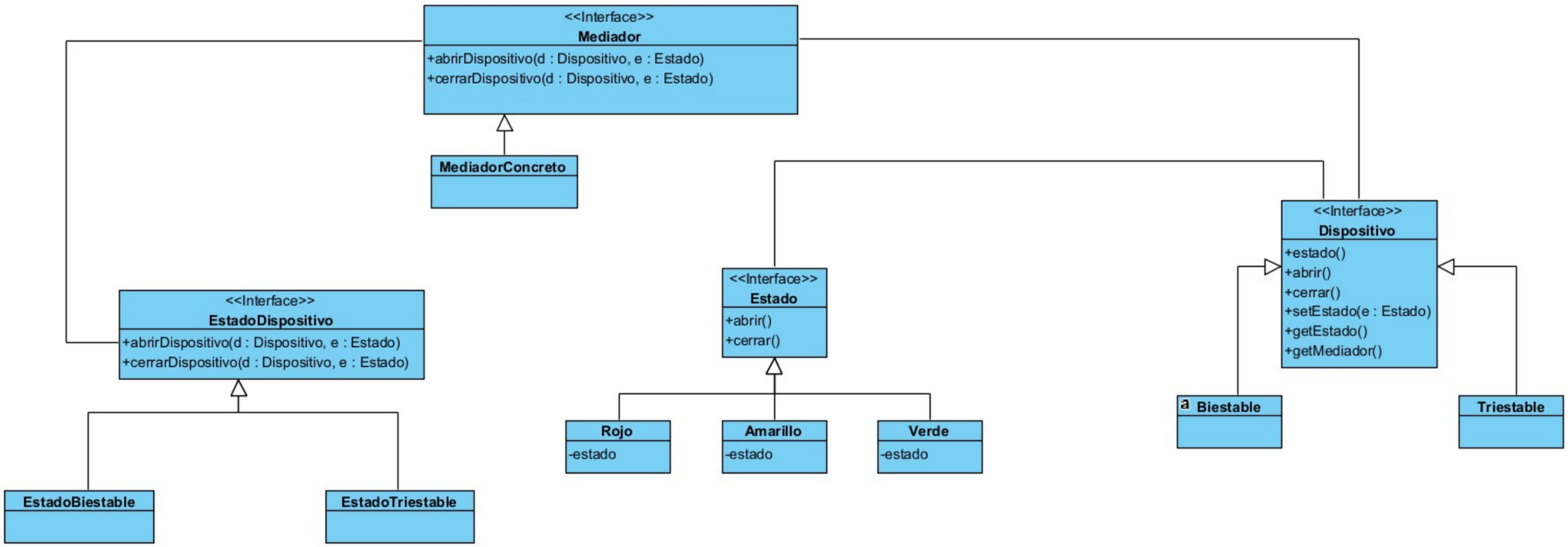
5. **Main**

```
package Java;

public class Main {
    public static void main(String[] args) {
        Biestable biestable = new Biestable();
        biestable.estado();
        biestable.abrir();
        biestable.estado();
        biestable.cerrar();
        biestable.estado();
    }
}
```

---

b) Para aplicarlo a nuestro ejercicio, primero creamos una interfaz Dispositivo que implementaran la clase que ya teníamos Biestable y una nueva clase Triestable, para que los estados reciban objeto del tipo Dispositivo independientemente de si sea un Biestable o un Triestable.
Posteriormente al aplicar el patrón Mediador, creamos una interfaz Mediador con los métodos "abrirDispositivo" y "cerrarDispositivo". Estos métodos toman como parámetros el dispositivo y su estado actual, y los implementamos en una clase concreta MediadorConcreto.
Para gestionar la transición de estados de manera condicional, introducimos una interfaz EstadoDispositivo. EstadoDispositivo ofrecerá los mismos métodos que el mediador, y sera implementadas por las clases EstadoBiestable y EstadoTriestable. La primera clase manejará la transición de estados para un dispositivo Biestable (de "Rojo" a "Verde"), mientras que la segunda lo hará para un dispositivo Triestable (transitando entre "Rojo", "Amarillo" y "Verde"). Este enfoque modular mejora la organización del código y facilita la reutilización al separar las responsabilidades de transición de estados en clases especializadas.



1. **Dispositivo**

```
    public interface Dispositivo {
    void abrir();
    void cerrar();
    public void setEstado(Estado estado);
    public Estado getEstado();
    public Mediador getMediador();

}
```

2. **Biestable**

```
public class Biestable implements Dispositivo{
    private Estado estado;
    private Mediador mediador;
```

```
    public Biestable(){
        this.estado = new Rojo();
        this.mediador = new MediadorConcreto(new EstadoBiestable());
    }
    public void abrir(){
        this.estado.abrir(this);
    }
    public void cerrar(){
        this.estado.cerrar(this);
    }

    public void setEstado(Estado estado){
        this.estado = estado;
    }

    public Estado getEstado() {
        return estado;
    }

    public Mediador getMediador() {
        return mediador;
    }

    public void estado(){
        System.out.println(estado);
    }
}
```

3. **Triestable**

```
public class Triestable implements Dispositivo{
    private Estado estado;
    private Mediador mediador;

    public Triestable(){
        this.estado = new Rojo();
        this.mediador = new MediadorConcreto(new EstadoTriestable());
    }
```

```
    @Override
    public void abrir() {
        this.estado.abrir(this);
    }

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

    public void setEstado(Estado estado){
        this.estado = estado;
    }

    public Estado getEstado() {
        return estado;
    }

    public Mediador getMediador() {
        return mediador;
    }

    public void estado(){
        System.out.println(estado);
    }
}
```

4. **Estado**

```
public interface Estado {
    //Metodos que van a tener que implementar las clases que implementen esta interfaz
    void abrir(Dispositivo dispositivo);
    void cerrar(Dispositivo dispositivo);
}
```

5. **Rojo**

```
public class Rojo implements Estado{
    private final String estado;

    public Rojo() {
        this.estado = "Rojo";
    }

    public void abrir(Dispositivo dispositivo){
        dispositivo.getMediador().abrirDispositivo(dispositivo,this);
    }
    public void cerrar(Dispositivo dispositivo){
        dispositivo.getMediador().cerrarDispositivo(dispositivo,this);
    }

    @Override
    public String toString() {
        return estado;
    }
}
```

6. **Amarillo**

```
public class Amarillo implements Estado{
    private final String estado;

    public Amarillo() {
        this.estado = "Amarillo";
    }
    public void abrir(Dispositivo dispositivo){
        dispositivo.getMediador().abrirDispositivo(dispositivo,this);
    }
    public void cerrar(Dispositivo dispositivo){
        dispositivo.getMediador().cerrarDispositivo(dispositivo,this);
    }

    @Override
    public String toString() {
        return estado;
```

```
    }
}
```

7. **Verde**

```
public class Verde implements Estado{
    private final String estado;

    public Verde() {
        this.estado = "Verde";
    }

    public void abrir(Dispositivo dispositivo){
        dispositivo.getMediador().abrirDispositivo(dispositivo,this);
    }
    public void cerrar(Dispositivo dispositivo){
        dispositivo.getMediador().cerrarDispositivo(dispositivo,this);
    }

    @Override
    public String toString() {
        return estado;
    }
}
```

8. **Mediador**

```
public interface Mediador {
    void abrirDispositivo(Dispositivo dispositivo, Estado estado);
    void cerrarDispositivo(Dispositivo dispositivo, Estado estado);
}
```

9. **MediadorConcreto**

```
public class MediadorConcreto implements Mediador{
    private EstadoDispositivo estadoDispositivo;
    public MediadorConcreto(EstadoDispositivo estadoDispositivo) {
        this.estadoDispositivo = estadoDispositivo;
```

```
    }

    @Override
    public void abrirDispositivo(Dispositivo dispositivo, Estado estado) {
        estadoDispositivo.abrirDispositivo(dispositivo,estado);
    }

    @Override
    public void cerrarDispositivo(Dispositivo dispositivo, Estado estado) {
        estadoDispositivo.cerrarDispositivo(dispositivo,estado);
    }
}
```

10. **EstadoDispositivo**

```
public interface EstadoDispositivo {
    void abrirDispositivo(Dispositivo dispositivo, Estado estado);
    void cerrarDispositivo(Dispositivo dispositivo, Estado estado);
}
```

11. **EstadoBiestable**

```
public class EstadoBiestable implements EstadoDispositivo{
    @Override
    public void abrirDispositivo(Dispositivo dispositivo, Estado estado) {
        if( estado instanceof Verde){
            System.err.println("No se puede abrir desde verde");
        }else {
            dispositivo.setEstado(new Verde());
        }
    }

    @Override
    public void cerrarDispositivo(Dispositivo dispositivo, Estado estado) {
        if( estado instanceof Rojo) {
            System.err.println("No se puede cerrar desde Rojo");
        }else {
            dispositivo.setEstado(new Rojo());
```

```
        }
    }
}
```

12. **EstadoTriestable**

```
public class EstadoTriestable implements EstadoDispositivo{
    @Override
    public void abrirDispositivo(Dispositivo dispositivo, Estado estado) {
        if( estado instanceof Verde){
            System.err.println("No se puede abrir desde verde");
        }else{
            if(estado instanceof Rojo){
                dispositivo.setEstado(new Amarillo());
            }else{
                dispositivo.setEstado(new Verde());
            }
        }
    }

    @Override
    public void cerrarDispositivo(Dispositivo dispositivo, Estado estado) {
        if( estado instanceof Rojo) {
            System.err.println("No se puede cerrar desde Rojo");
        }else{
            if(estado instanceof Verde){
                dispositivo.setEstado(new Amarillo());
            }else{
                dispositivo.setEstado(new Rojo());
            }
        }

    }
}
```

13. **Main**

```
public class Main {
    public static void main(String[] args) {
        Biestable biestable = new Biestable();
        biestable.estado();
        biestable.abrir();
        biestable.estado();
        biestable.cerrar();
        biestable.estado();
        System.out.println("-----------------------------------------");
        Triestable triestable = new Triestable();
        triestable.estado();
        triestable.abrir();
        triestable.estado();
        triestable.abrir();
        triestable.estado();
        triestable.cerrar();
        triestable.estado();
        triestable.abrir();
        triestable.estado();
        triestable.cerrar();
        triestable.estado();
        triestable.cerrar();
        triestable.estado();

    }
}
```

Ventajas: La solución permite reutilizar clases anteriores con cambios mínimos en "abrir" y "cerrar". Facilita la gestión de nuevos estados de color o tipos de dispositivos sin alterar las clases existentes.

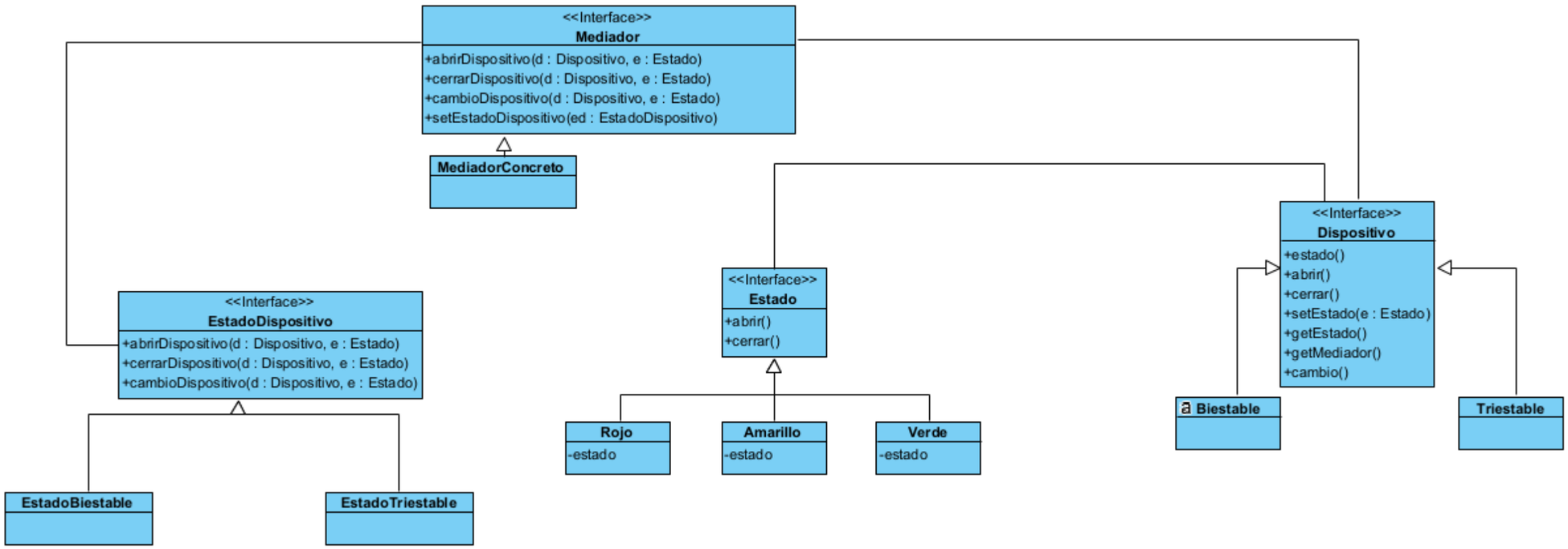
Inconvenientes: Se generan más clases de las aparentemente necesarias, ya que la transición entre estados, antes manejada por las clases "Rojo" y "Verde", ahora se gestiona a través del Mediador y se delega a los estados del Mediador.

---

c) Utilizando el patrón Estado para el mediador del paso anterior, logramos un cambio de comportamiento entre ambos dispositivos simplemente cambiando de estado. Ahora, incorporamos un método llamado "cambio()" a la interfaz de los dispositivos. Este método será delegado al mediador del dispositivo. La interfaz Mediador, , proporcionará un método "cambioDispositivo" que recibirá el dispositivo y su estado actual como parámetros. Este método, a su vez, delegará la responsabilidad al objeto de tipo "EstadoDispositivo" actualmente asociado.

Este objeto si es del tipo EstadoBiestable cambiara a Estado triestable y viceversa

Sin embargo, en el caso de que el estado del dispositivo sea "Amarillo", hemos decidido que cambie al estado "Rojo", ya que un dispositivo Biestable no puede estar en estado "Amarillo"



1. **Triestable**

```
public class Triestable implements Dispositivo{
    private Estado estado;
    private Mediador mediador;

    public Triestable(){
        this.estado = new Rojo();
        this.mediador = new MediadorConcreto(new EstadoTriestable());
    }
    @Override
    public void abrir() {
        this.estado.abrir(this);
    }

    @Override
```

```
    public void cerrar() {
        this.estado.cerrar(this);
    }

    public void setEstado(Estado estado){
        this.estado = estado;
    }

    public Estado getEstado() {
        return estado;
    }

    public Mediador getMediador() {
        return mediador;
    }

    @Override
    public void cambio() {
        mediador.cambioDispositivo(this,estado);
    }

    public void estado(){
        System.out.println(estado);
    }
}
```

2. **MediadorConcreto**

```
public class MediadorConcreto implements Mediador{
    private EstadoDispositivo estadoDispositivo;
    public MediadorConcreto(EstadoDispositivo estadoDispositivo) {
        this.estadoDispositivo = estadoDispositivo;
    }

    @Override
    public void abrirDispositivo(Dispositivo dispositivo, Estado estado) {
        estadoDispositivo.abrirDispositivo(dispositivo,estado);
    }
```

```
    @Override
    public void cerrarDispositivo(Dispositivo dispositivo, Estado estado) {
        estadoDispositivo.cerrarDispositivo(dispositivo,estado);
    }

    @Override
    public void cambioDispositivo(Dispositivo dispositivo, Estado estado) {
        estadoDispositivo.cambioDispositivo(dispositivo,estado);
    }

    @Override
    public void setEstadoDispositivo(EstadoDispositivo estadoDispositivo) {
        this.estadoDispositivo = estadoDispositivo;
    }
}
```

3. **EstadoTriestable**

```
public class EstadoTriestable implements EstadoDispositivo{
    @Override
    public void abrirDispositivo(Dispositivo dispositivo, Estado estado) {
        if( estado instanceof Verde){
            System.err.println("No se puede abrir desde verde");
        }else{
            if(estado instanceof Rojo){
                dispositivo.setEstado(new Amarillo());
            }else{
                dispositivo.setEstado(new Verde());
            }
        }
    }

    @Override
    public void cerrarDispositivo(Dispositivo dispositivo, Estado estado) {
        if( estado instanceof Rojo) {
            System.err.println("No se puede cerrar desde Rojo");
        }else{
```

```
            if(estado instanceof Verde){
                dispositivo.setEstado(new Amarillo());
            }else{
                dispositivo.setEstado(new Rojo());
            }
        }
    }
    @Override
    public void cambioDispositivo(Dispositivo dispositivo, Estado estado) {
        if (estado instanceof Amarillo) {
            dispositivo.setEstado(new Rojo());
        }
        dispositivo.getMediador().setEstadoDispositivo(new EstadoBiestable());
    }
}
```

4. **EstadoBiestable**

```
public class EstadoBiestable implements EstadoDispositivo{
    @Override
    public void abrirDispositivo(Dispositivo dispositivo, Estado estado) {
        if( estado instanceof Verde){
            System.err.println("No se puede abrir desde verde");
        }else {
            dispositivo.setEstado(new Verde());
        }
    }

    @Override
    public void cerrarDispositivo(Dispositivo dispositivo, Estado estado) {
        if( estado instanceof Rojo) {
            System.err.println("No se puede cerrar desde Rojo");
        }else {
            dispositivo.setEstado(new Rojo());
        }
    }

    @Override
```

```
    public void cambioDispositivo(Dispositivo dispositivo, Estado estado) {
        dispositivo.getMediador().setEstadoDispositivo(new EstadoTriestable());
    }
}
```

5. **Biestable**

```
public class Biestable implements Dispositivo{
    private Estado estado;
    private Mediador mediador;
    public Biestable(){
        this.estado = new Rojo();
        this.mediador = new MediadorConcreto(new EstadoBiestable());
    }
    public void abrir(){
        this.estado.abrir(this);
    }
    public void cerrar(){
        this.estado.cerrar(this);
    }

    public void setEstado(Estado estado){
        this.estado = estado;
    }

    public Estado getEstado() {
        return estado;
    }

    public Mediador getMediador() {
        return mediador;
    }

    @Override
    public void cambio() {
        mediador.cambioDispositivo(this,estado);
    }
```

```
    public void estado(){
        System.out.println(estado);
    }
}
```

---

3. Cliente de correo e-look

Hemos implementado el patrón estrategia, pues ha resultado ser eficaz para proporcionar una solución a diferentes criterios de ordenación. Este enfoque permite a los usuarios personalizar la experiencia de gestión de correos electrónicos según sus preferencias individuales. La estructura modular del patrón facilita el mantenimiento y la incorporación de nuevas funcionalidades en el futuro.

Las ventajas del patrón son:

1. **Flexibilidad:** Permite cambiar dinámicamente la estrategia de ordenación en tiempo de ejecución.
2. **Extensibilidad:** Facilita la adición de nuevas estrategias de ordenación sin modificar el código existente.
3. **Reusabilidad:** Las estrategias de ordenación pueden reutilizarse en diferentes contextos.

Los elementos del patrón, en este caso son:

1. **Contexto (`Mailbox`):**
   - Contiene una lista de correos electrónicos y un atributo de estrategia de ordenación (`sortingStrategy`).
   - Provee métodos para agregar correos electrónicos, establecer la estrategia de ordenación y mostrar los correos electrónicos según la estrategia actual.

   Mailbox

```
import java.util.ArrayList;
import java.util.List;

public class Mailbox {
    private List<Email> emails;

    private SortingStrategy sortingStrategy;

    public Mailbox() {
        emails = new ArrayList<Email>();
```

```
    }

    public void addemail(Email email) {
        emails.add(email);
    }
    public void setSortingStrategy(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void show() {
        sortingStrategy.sort(emails);
        for (Email email : emails) {
            System.out.println(email.toString());
        }
    }
}
```

2. **Interfaz `SortingStrategy`:**

   - Define un contrato común para todas las estrategias de ordenación.

   SortingStrategy

```
import java.util.List;

interface SortingStrategy {
    void sort(List<Email> email);
}
```

3. **Estrategias Concretas (`DateSortingStrategy`, `FromSortingStrategy`, etc.):**

   - Implementan la interfaz `SortingStrategy` y proporcionan algoritmos específicos para ordenar los correos electrónicos según diferentes criterios.

   DateSortingStrategy

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;

public class DateSortingStrategy implements SortingStrategy {
```

```
    public void sort(List<Email> email) {
        for (int i = 1; i < email.size(); i++ ){
            for (int j = email.size()-1; j >= i; j--){
                if (before(email.get(j), email.get(j - 1))) {
                    // intercambiar los mensajes j y j-1
                    Email temp = email.get(j);
                    email.set(j, email.get(j - 1));
                    email.set(j - 1, temp);
                }
            }
        }
    }

    private boolean before(Email email1, Email email2) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

        try {
            Date date1 = sdf.parse(email1.getDate());
            Date date2 = sdf.parse(email2.getDate());
            return date1.before(date2);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

FromSortingStrategy

```
import java.util.List;

public class FromSortingStrategy implements SortingStrategy {
        public void sort(List<Email> email) {
            for (int i = 1; i < email.size(); i++ ){
                for (int j = email.size()-1; j >= i; j--){
                    if (before(email.get(j), email.get(j - 1))) {
                        // intercambiar los mensajes j y j-1
```

```
                        Email temp = email.get(j);
                        email.set(j, email.get(j - 1));
                        email.set(j - 1, temp);
                    }
                }
            }
        }

    private boolean before(Email email1, Email email2) {
        return email1.getFrom().compareTo(email2.getFrom()) < 0;
    }
}
```

SubjectSortingStrategy

```
import java.util.List;

public class SubjectSortingStrategy implements SortingStrategy {
    public void sort(List<Email> email) {
        for (int i = 1; i < email.size(); i++ ){
            for (int j = email.size()-1; j >= i; j--){
                if (before(email.get(j), email.get(j - 1))) {
                    // intercambiar los mensajes j y j-1
                    Email temp = email.get(j);
                    email.set(j, email.get(j - 1));
                    email.set(j - 1, temp);
                }
            }
        }
    }

    private boolean before(Email email1, Email email2) {
        return email1.getSubject().compareTo(email2.getSubject()) < 0;
    }

}
```

PrioritySortingStrategy

```
import java.util.List;

public class PrioritySortingStrategy implements SortingStrategy {
    public void sort(List<Email> email) {
        for (int i = 1; i < email.size(); i++ ){
            for (int j = email.size()-1; j >= i; j--){
                if (before(email.get(j), email.get(j - 1))) {
                    // intercambiar los mensajes j y j-1
                    Email temp = email.get(j);
                    email.set(j, email.get(j - 1));
                    email.set(j - 1, temp);
                }
            }
        }
    }

    private boolean before(Email email1, Email email2) {
        return email1.getPriority() < email2.getPriority();
    }
}
```

4. **Cliente (`Main`):**
    - Instancia un objeto `Mailbox` y diferentes estrategias de ordenación (`dateSortingStrategy`, `fromSortingStrategy`, etc.).
    - Configura la estrategia de ordenación actual en el `Mailbox`.
    - Invoca el método `show` del `Mailbox` para mostrar los correos electrónicos según la estrategia seleccionada.

    Main

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {

        Mailbox mailbox = new Mailbox();
        mailbox.addemail(new Email("Juan", "juan saluda", "2023-12-03", 2));
        mailbox.addemail(new Email("Carlos", "pepe saluda", "2023-11-02", 1));
```

```
        SortingStrategy dateSortingStrategy = new DateSortingStrategy();
        SortingStrategy fromSortingStrategy = new FromSortingStrategy();
        SortingStrategy prioritySortingStrategy = new PrioritySortingStrategy();
        SortingStrategy subjectSortingStrategy = new SubjectSortingStrategy();

        // Configurar la estrategia de ordenación actual
        mailbox.setSortingStrategy(dateSortingStrategy);
        // Mostrar correos electrónicos ordenados por fecha
        mailbox.show();

        // Cambiar la estrategia de ordenación
        mailbox.setSortingStrategy(fromSortingStrategy);
        // Mostrar correos electrónicos ordenados por remitente
        mailbox.show();

        // Cambiar la estrategia de ordenación
        mailbox.setSortingStrategy(prioritySortingStrategy);
        // Mostrar correos electrónicos ordenados por prioridad
        mailbox.show();

        // Cambiar la estrategia de ordenación
        mailbox.setSortingStrategy(subjectSortingStrategy);
        // Mostrar correos electrónicos ordenados por asunto
        mailbox.show();

    }
}
```

La clase Email ha sido la siguiente

Email

```
public class Email {
        private String from;
        private String subject;
        private String date;
        private int priority;
```

```
    public Email(String from, String subject, String date, int priority) {
        this.from = from;
        this.subject = subject;
        this.date = date;
        this.priority = priority;
    }

    public String getFrom() {
        return from;
    }

    public void setFrom(String from) {
        this.from = from;
    }

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public String getDate() {
        return date;
    }

    public void setDate(String date) {
        this.date = date;
    }

    public int getPriority() {
        return priority;
    }

    public void setPriority(int priority) {
        this.priority = priority;
    }
```

```
    @Override
    public String toString(){
        return "From: "+from+"\nSubject: "+subject+"\nDate: "+date+"\nPriority: "+priority+"\n";
    }
}
```