Universidad De Málaga

Escuela Superior de Ingeniería Informática

Ingeniería de Software

Práctica 4
Patrones de Diseño

Modelado y Diseño de Software

Grupo A3
Arjona Jiménez, Francisco de Paula
Castillo Berná, David
Ortuño Roig, Javier
Ramet Espinosa, Alejandro
Velasco López, Cristhian

17 de enero de 2021

Índice

1. Ejercicio 1

1.1. Apartado a

Java no dispone de mecanismos de exportación como tal, para la restricción de acceso de las diferentes clases a sus operaciones recurre a la visibilidad de los paquetes, (public, private, package, protected). Con esto una clase externa puede acceder tan solo a las operaciones públicas. La visibilidad protected indica que tan solo la misma clases y sus hijas pueden utilizarla y por último las operaciones private tan solo se pueden llamar en operaciones de la misma clase.

En eiffel al añadir "feature" podemos decir quienes podrán utilizar este método, con esto conseguimos una mayor simplicidad ya que al indicar directamente quienes pueden utilizarlo no tenemos que ir buscando si la clase exterior puede acceder o no a las operaciones de la clase inicial.

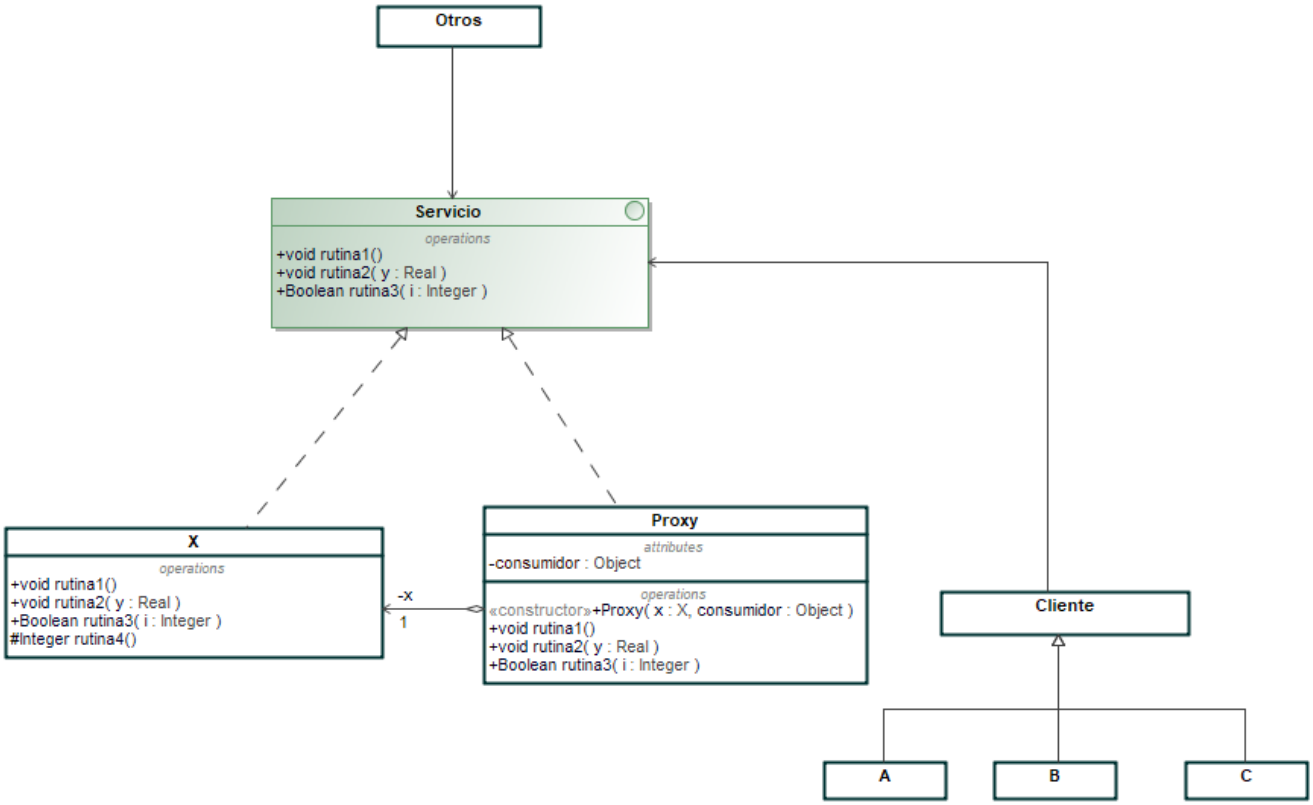
1.2. Apartado b

1.2.1. Explicación textual

El patrón que vamos a utilizar para resolver este ejercicio es el patrón estructural Proxy. Hemos utilizado este patrón ya que por propia definición se utiliza para el control de acceso al objeto principal. Es decir, la clase proxy indicará si una clase externa puede utilizar las operaciones de la clase principal o no.

Al realizar esta implementación podremos observar que tenemos un total control de las clases, pero comparado con eiffel es más tedioso, ya que en eiffel con un comando controlamos todo, mientras que en java necesitamos una clase proxy que lo controle.

1.2.2. Modelo



1.2.3. Código

Primero daremos una descripción y a continuación habrá imágenes con el código.

Las clases A,B y C, son los tres tipos de clientes que pueden acceder a la clase X, que es la clase que realmente aporta los servicios. La clase Cliente tan solo sirve para indicar que las clases A, B y C son sus hijas para no tener código repetido.

Como hemos indicado anteriormente a la clase Proxy le llega un cliente del tipo objeto, donde nosotros comprobamos de qué tipo es para saber si puede acceder a la operación de X que ha solicitado.

Por último X y Proxy son hijas de la interfaz Servicio. En esta interfaz es la que indica que servicios aporta el proxy. La interfaz Servicio es un objeto imaginario, y la clase X es el objeto real, y el Proxy tiene la función de crear un objeto real si el cliente puede acceder a los servicios que solicita en la interfaz.

Interfaz: Servicio.Java

```java
package Ejercicio1;

public interface Servicio {

    public void rutina1();
    public void rutina2(Double y);
    public Boolean rutina3(Integer i);
}
```

Clase: X.Java

```java
package Ejercicio1;

public class X implements Servicio {

    @Override
    public void rutina1() {
        System.out.println("Ha entrado en la rutina 1");

    }

    @Override
    public void rutina2(Double y) {
        System.out.println("Ha entrado en la rutina 2");

    }

    @Override
    public Boolean rutina3(Integer i) {
        System.out.println("Ha entrado en la rutina 3");
        return true;
    }

    public Integer rutina4() {
        System.out.println("Ha entrado en la rutina 4");
        return 1;
    }
}
```

Clase: Proxy.Java

```java
package Ejercicio1;

public class Proxy implements Servicio {
```

```
    X x;
    Object consumidor;

    public Proxy(X s, Object consumidor) {
        x = s;
        this.consumidor = consumidor;
    }

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

    @Override
    public void rutina2(Double y) {
        if (consumidor instanceof A || consumidor instanceof B) {
            this.x.rutina2(y);
        } else {
            System.out.println("El objeto no tiene permiso para
                usar esta clase.");
        }
    }

    @Override
    public Boolean rutina3(Integer i) {
        if (consumidor instanceof A || consumidor instanceof C) {
            return this.x.rutina3(i);
        } else {
            System.out.println("El objeto no tiene permiso para
                usar esta clase.");
            return null;
        }
    }
}
```

Interfaz: Proxy.Java

```
package Ejercicio1;

public class Cliente{
    //implementacion de la clase Cliente
}
```

Clase: A.Java

```java
package Ejercicio1;

public class A extends Cliente {
    //implementacion de la clase A
}
```

Clase: B.Java

```java
package Ejercicio1;

public class B extends Cliente{
    //implementacion de la clase B
}
```

Clase: C.Java

```java
package Ejercicio1;

public class C extends Cliente{
    //implementacion de la clase C
}
```

2. Ejercicio 2

2.1. Apartado a

2.1.1. Explicación textual

En este apartado vamos a utilizar el patrón de diseño “Estado”. Esto se debe a nuestra necesidad de querer alterar el comportamiento de la clase “Biestable” según el estado en el que se encuentre.

Concretamente, deseamos dos estados para el mismo, “Rojo” y “Verde”. Ambos se implementarán como subclases de una clase abstracta “Estado”, que contendrá los métodos propios de la clase “Biestable”, pues queremos que según el estado en el que se encuentre, actúe de una manera u otra.

Por lo tanto, nuestra clase “Biestable” contendrá una instancia de “Estado”, que podrá ser o “Rojo” o “Verde”.

En cuanto a los métodos del biestable:

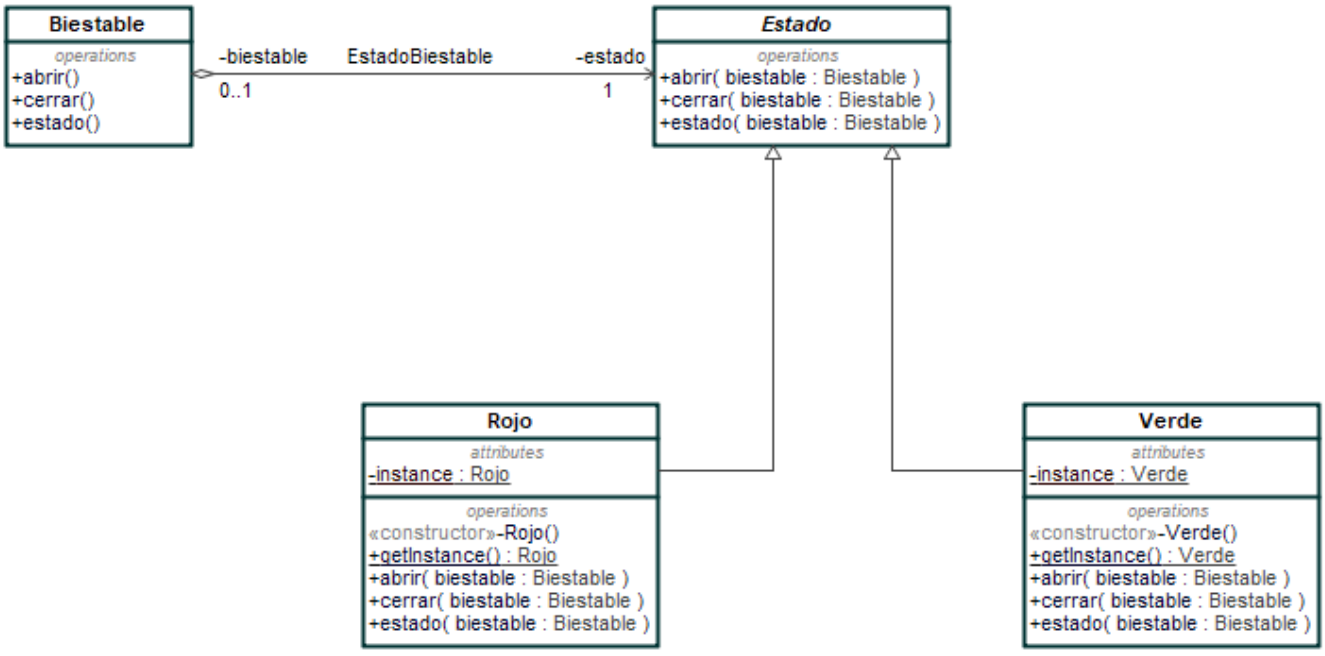
- abrir():
  - Estado “Rojo”: permitirá a un “Biestable” pasar de un estado “Rojo” a “Verde”.
  - Estado “Verde”: devolverá un mensaje de error.

- cerrar():
  - Estado “Rojo”: devolverá un mensaje de error.
  - Estado “Verde”: permitirá a un “Biestable” pasar de un estado “Verde” a “Rojo”.
- estado() : devolverá un string correspondiente al estado en el que me encuentro.

Adicionalmente, hemos utilizado el patrón “Singleton” para reutilizar las instancias de los estados, ya que en otro caso, con la llamada a los métodos, estaríamos creando constantemente nuevos estados.

Para aplicarlo, la clase contendrá un atributo estático correspondiente a una instancia de la misma clase. Por otro lado, encontramos un método getInstance(), que nos permitirá crear esta instancia estática si no se ha creado anteriormente (a través del constructor privado de la clase), o bien devolver dicha instancia ya creada.

2.1.2. Modelo



2.1.3. Pseudocódigo

Dado que el pseudocódigo es muy parecido al código en Java se dejará solo el código para no sobrecargar la memoria con información repetida.

2.1.4. Código

Clase: Biestable.Java

```
package Ejercicio2.a;

public class Biestable {

    Estado estado;

    public Biestable() {
        this.estado = Rojo.getInstance();
    }

    public void abrir() {
        estado.abrir(this);
    }

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

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

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

Clase abstracta: Estado.Java

```
package Ejercicio2.a;

public abstract class Estado {
    abstract public void abrir(Biestable biestable);
    abstract public void cerrar(Biestable biestable);
    abstract public void estado(Biestable biestable);
}
```

Clase: Rojo.Java

```
package Ejercicio2.a;
```

```
public class Rojo extends Estado {

    private static Rojo instance;

    private Rojo() {

    }

    public static Rojo getInstance() {
        if (instance == null) {
            instance = new Rojo();

        }
        return instance;
    }

    @Override
    public void abrir(Biestable biestable) {
        biestable.setEstado(Verde.getInstance());
    }

    @Override
    public void cerrar(Biestable biestable) {
        throw new RuntimeException("No se puede cerrar estando en
             el estado rojo");
    }

    @Override
    public void estado(Biestable biestable) {
        System.out.println("Cerrado");
    }
}
```

Clase: Verde.Java

```
package Ejercicio2.a;

public class Verde extends Estado {

    private static Verde instance;

    private Verde() { }

    public static Verde getInstance() {
```

```
        if (instance == null) {
            instance = new Verde();

        }
        return instance;
    }

    @Override
    public void abrir(Biestable biestable) {
        throw new RuntimeException("No se puede abrir estando en
            el estado verde");
    }

    @Override
    public void cerrar(Biestable biestable) {
        biestable.setEstado(Rojo.getInstance());
    }

    @Override
    public void estado(Biestable biestable) {
        System.out.println("Abierto");
    }
}
```

2.2. Apartado b

2.2.1. Explicación textual

Al igual que en el apartado anterior, utilizaremos el patrón de diseño "Estado" para implementar la clase "Triestable", añadiendo esta vez una nueva subclase "Amarillo", que será un estado intermedio entre "Rojo" y "Verde".

Como queremos que en nuestro sistema haya tanto biestables como triestables, ambos tendrán un atributo "Estado". Sin embargo, debido a que un biestable no puede estar en un estado "Amarillo", hemos añadido una restricción OCL para evitar esto:

- *not self.estado.oclIsTypeOf(Amarillo):* En el contexto de un Biestable, su estado no podrá ser del tipo "Amarillo"

En cuanto a los métodos del triestable:

- abrir():

  - Estado “Rojo”: permitirá a un “Biestable” pasar de un estado “Rojo” a “Amarillo”.
  - Estado “Amarillo” : permitirá a un “Biestable” pasar de un estado “Amarillo” a “Verde”.
  - Estado “Verde”: devolverá un mensaje de error.

- cerrar():

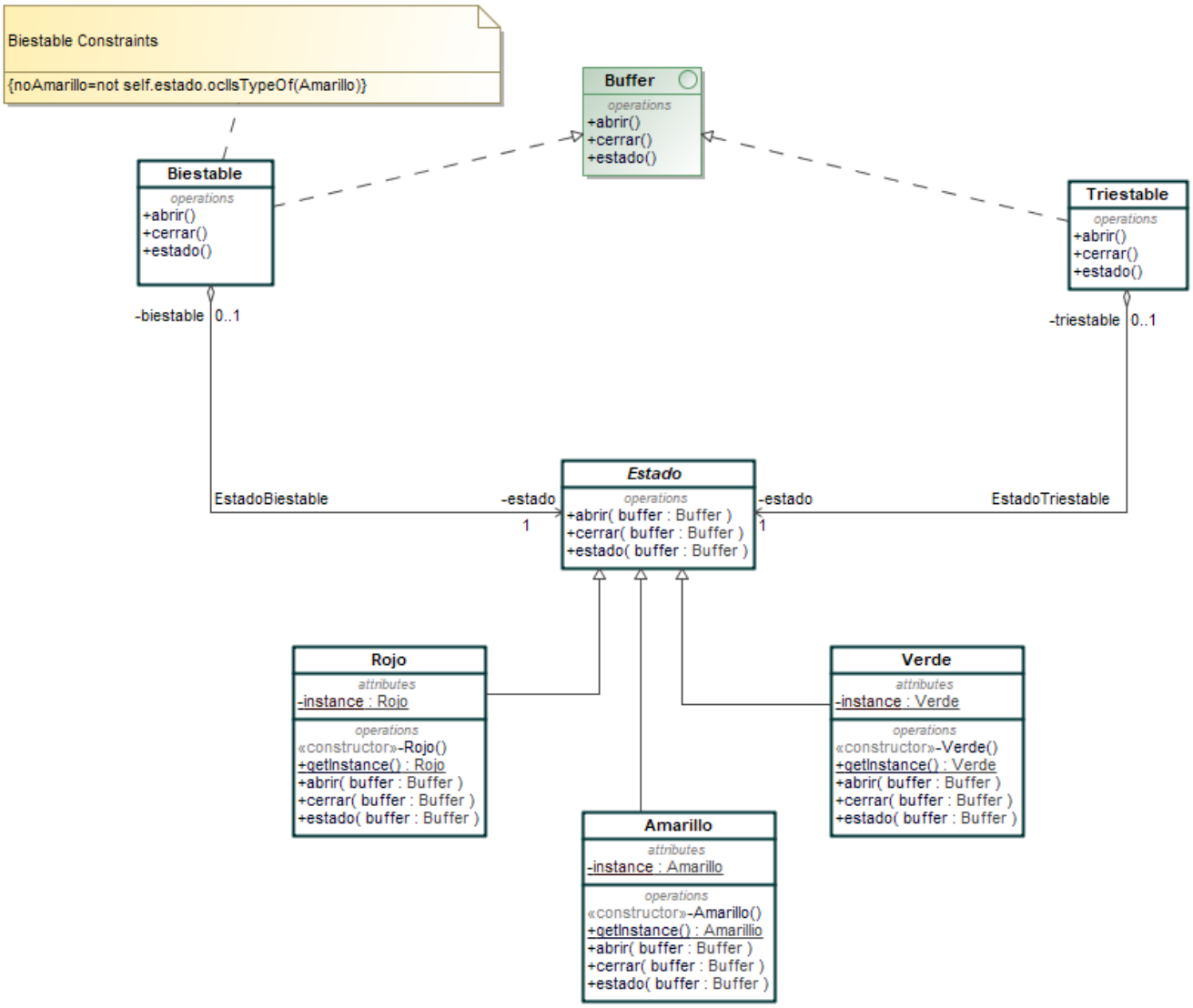
  - Estado “Rojo”: devolverá un mensaje de error.
  - Estado “Amarillo” : permitirá a un “Biestable” pasar de un estado “Amarillo” a “Rojo”.
  - Estado “Verde”: permitirá a un “Biestable” pasar de un estado “Verde” a “Amarillo”.

- estado() : devuelve una cadena string que muestra el estado en el que se encuentra el triestable.

Adicionalmente, hemos utilizado el patrón “Singleton” para reutilizar las instancias de los estados, ya que en otro caso, con la llamada a los métodos, estaríamos creando constantemente nuevos estados.

Para aplicarlo, la clase contendrá un atributo estático correspondiente a una instancia de la misma clase. Por otro lado, encontramos un método getInstance(), que nos permitirá crear esta instancia estática si no se ha creado anteriormente (a través del constructor privado de la clase), o bien devolver dicha instancia ya creada.

2.2.2. Modelo



2.2.3. Pseudocódigo

Dado que el pseudocódigo es muy parecido al código en Java se dejará solo el código para no sobrecargar la memoria con información repetida.

2.2.4. Código

Interfaz: Buffer.Java

```
package Ejercicio2.b;
```

```
public interface Buffer {
    public void abrir();
    public void cerrar();
    public void estado();
    public void setEstado(Estado estado);
}
```

Clase: Biestable.Java

```
package Ejercicio2.b;

public class Biestable implements Buffer {

    Estado estado;

    public Biestable() {
        this.estado = Rojo.getInstance();
    }

    public void abrir() {
        estado.abrir(this);
    }

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

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

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

Clase: Triestable.Java

```
package Ejercicio2.b;

public class Triestable implements Buffer{

    Estado estado;
```

```
    public Triestable() {
        this.estado = Rojo.getInstance();
    }

    public void abrir() {
        estado.abrir(this);
    }

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

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

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

Clase abstracta: Estado.Java

```
package Ejercicio2.b;

public abstract class Estado {
    abstract public void abrir(Buffer buffer);
    abstract public void cerrar(Buffer buffer);
    abstract public void estado(Buffer buffer);
}
```

Clase: Rojo.Java

```
package Ejercicio2.b;

public class Rojo extends Estado {

    private static Rojo instance;

    private Rojo() {

    }

    public static Rojo getInstance() {
        if (instance == null) {
```

```
            instance = new Rojo();

        }
        return instance;
    }

    @Override
    public void abrir(Buffer buffer) {
        if (buffer instanceof Biestable) {
            buffer.setEstado(Verde.getInstance());
        } else {
            buffer.setEstado(Amarillo.getInstance());
        }
    }

    @Override
    public void cerrar(Buffer buffer) {
        throw new RuntimeException("No se puede cerrar estando en
             el estado rojo");
    }

    @Override
    public void estado(Buffer buffer) {
        System.out.println("Cerrado");
    }
}
```

Clase: Amarillo.Java

```
package Ejercicio2.b;

public class Amarillo extends Estado {

    private static Amarillo instance;

    private Amarillo() { }

    public static Amarillo getInstance() {
        if (instance == null) {
            instance = new Amarillo();

        }
        return instance;
    }
```

```
    @Override
    public void abrir(Buffer buffer) {
        if (buffer instanceof Biestable) {
            throw new RuntimeException("Un biestable no puede
                estar en el estado amarillo");
        } else {
            buffer.setEstado(Verde.getInstance());
        }
    }

    @Override
    public void cerrar(Buffer buffer) {
        if (buffer instanceof Biestable) {
            throw new RuntimeException("Un biestable no puede
                estar en el estado amarillo");
        } else {
            buffer.setEstado(Rojo.getInstance());
        }
    }

    @Override
    public void estado(Buffer buffer) {
        System.out.println("precaucion");
    }
}
```

Clase: Verde.Java

```
package Ejercicio2.b;

public class Verde extends Estado {

    private static Verde instance;

    private Verde() { }

    public static Verde getInstance() {
        if (instance == null) {
            instance = new Verde();

        }
        return instance;
```

```
    }

    @Override
    public void abrir(Buffer buffer) {
        throw new RuntimeException("No se pue de abrir estando en
            el estado Verde");
    }

    @Override
    public void cerrar(Buffer buffer) {
        buffer.setEstado(Amarillo.getInstance());
    }

    @Override
    public void estado(Buffer buffer) {
        System.out.println("Abierto");
    }
}
```

2.3. Apartado c

2.3.1. Explicación textual

Al igual que en el apartado anterior, utilizaremos el patrón de diseño "Estado" para implementar las clases "Biestable" y "Triestable".

En este caso, para implementar el cambio de "Biestable" a "Triestable", utilizaremos el patrón de diseño "Decorador". Con esto conseguiremos añadir la funcionalidad de la clase "Triestable" a nuestra clase "Biestable".

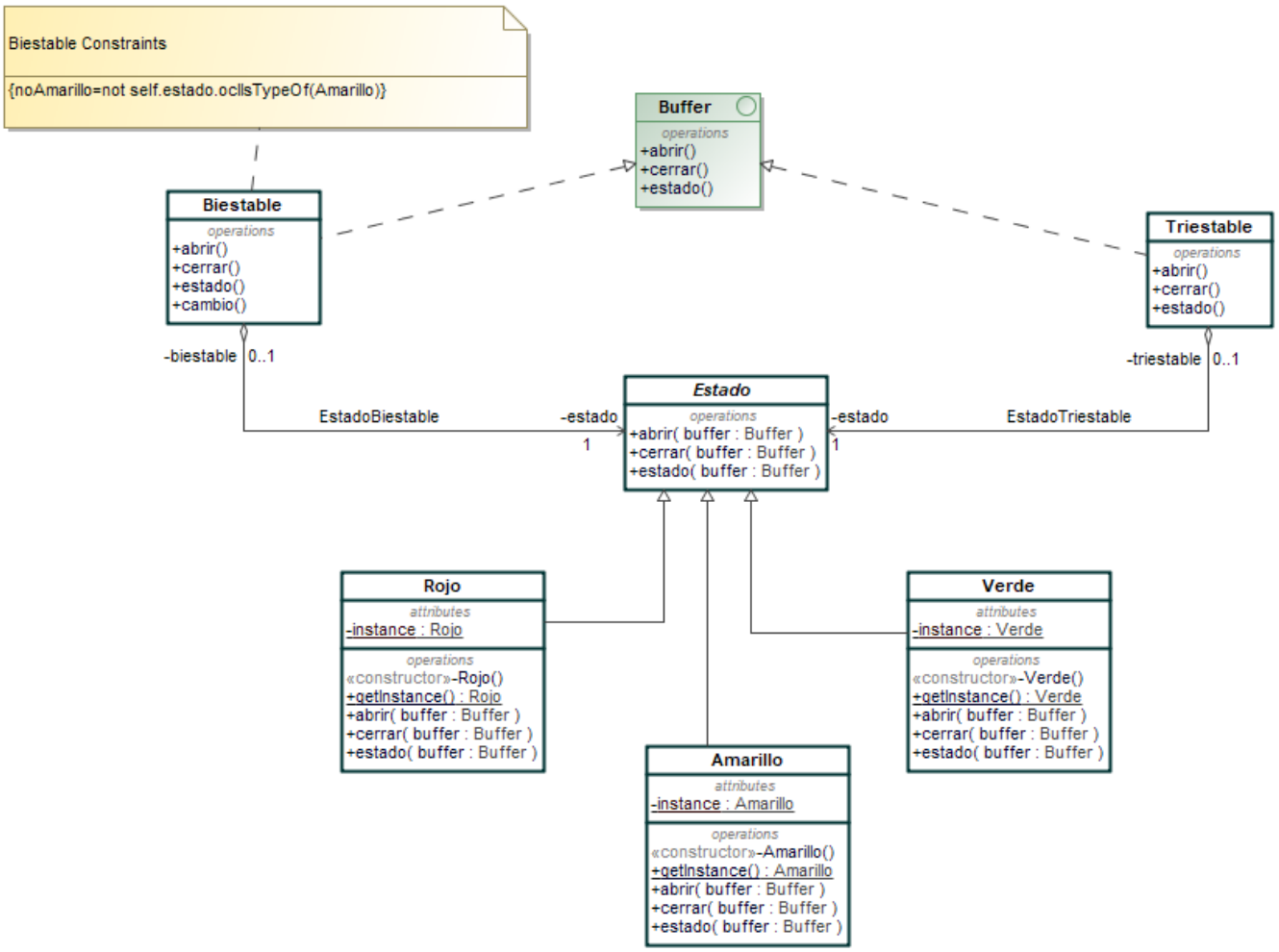
Para ello el "Biestable" contendrá un objeto que implementará la interfaz "Buffer", inicialmente a null.

Este objeto se inicializará a un objeto de tipo "Triestable" tras la llamada al método cambio() del "Biestable". De esta manera, cuando se vuelvan a invocar a los métodos abrir(), cerrar() y estado(), serán los métodos del buffer (Triestable) los que se ejecuten, actuando el "Biestable" como un "Triestable".

Adicionalmente, hemos utilizado el patrón "Singleton" para reutilizar las instancias de los estados, ya que en otro caso, con la llamada a los métodos, estaríamos creando constantemente nuevos estados.

Para aplicarlo, la clase contendrá un atributo estático correspondiente a una instancia de la misma clase. Por otro lado, encontramos un método getInstance(), que nos permitirá crear esta instancia estática si no se ha creado anteriormente (a través del constructor privado de la clase), o bien devolver dicha instancia ya creada.

2.3.2. Modelo



2.3.3. Pseudocódigo

Dado que el pseudocódigo es muy parecido al código en Java se dejará solo el código para no sobrecargar la memoria con información repetida.

2.3.4. Código

Interfaz: Buffer.Java

```
package Ejercicio2.c;

public interface Buffer {
    public void abrir();
    public void cerrar();
```

```
    public void estado();
    public void setEstado(Estado estado);
}
```

Clase: Biestable.Java

```
package Ejercicio2.c;

public class Biestable implements Buffer {

    Estado estado;
    Buffer buffer;

    public Biestable() {
        this.estado = Rojo.getInstance();
        this.buffer = null;
    }

    public void abrir() {
        if (buffer == null) {
            estado.abrir(this);
        } else {
            buffer.abrir();
        }
    }

    public void cerrar() {
        if (buffer == null) {
            estado.cerrar(this);
        } else {
            buffer.cerrar();
        }
    }

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

    public void estado() {
        if (buffer == null) {
            this.estado.estado(this);
        } else {
            buffer.estado();
        }
```

```
    }

    public void cambio() { this.buffer = new Triestable(Amarillo
        .getInstance()); }
}
```

Clase: Triestable.Java

```
package Ejercicio2.c;

public class Triestable implements Buffer {

    Estado estado;

    public Triestable() {
        this.estado = Rojo.getInstance();
    }

    public Triestable(Estado estado) {
        this.estado = estado;
    }

    public void abrir() {
        estado.abrir(this);
    }

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

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

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

Clase abstracta: Estado.Java

```
package Ejercicio2.c;

public abstract class Estado {
    abstract public void abrir(Buffer buffer);
```

```
    abstract public void cerrar(Buffer buffer);
    abstract public void estado(Buffer buffer);
}
```

Clase: Rojo.Java

```
package Ejercicio2.c;

public class Rojo extends Estado {

    private static Rojo instance;

    private Rojo() {

    }

    public static Rojo getInstance() {
        if (instance == null) {
            instance = new Rojo();

        }
        return instance;
    }

    @Override
    public void abrir(Buffer buffer) {
        buffer.setEstado(Verde.getInstance());
    }

    @Override
    public void cerrar(Buffer buffer) {
        throw new RuntimeException("No se puede cerrar estando en
             el estado rojo");
    }

    @Override
    public void estado(Buffer buffer) {
        System.out.println("Cerrado");
    }
}
```

Clase: Amarillo.Java

```
package Ejercicio2.c;
```

```java
public class Amarillo extends Estado {

    private static Amarillo instance;

    private Amarillo() {

    }

    public static Amarillo getInstance() {
        if (instance == null) {
            instance = new Amarillo();
        }
        return instance;
    }


    @Override
    public void abrir(Buffer buffer) {
        if (buffer instanceof Biestable) {
            throw new RuntimeException("Un biestable no puede
                estar en el estado amarillo");
        } else {
            buffer.setEstado(Verde.getInstance());
        }
    }

    @Override
    public void cerrar(Buffer buffer) {
        if (buffer instanceof Biestable) {
            throw new RuntimeException("Un biestable no puede
                estar en el estado amarillo");
        } else {
            buffer.setEstado(Rojo.getInstance());
        }
    }

    @Override
    public void estado(Buffer buffer) {
        System.out.println("precaucion");
    }
}
```

Clase: Verde.Java

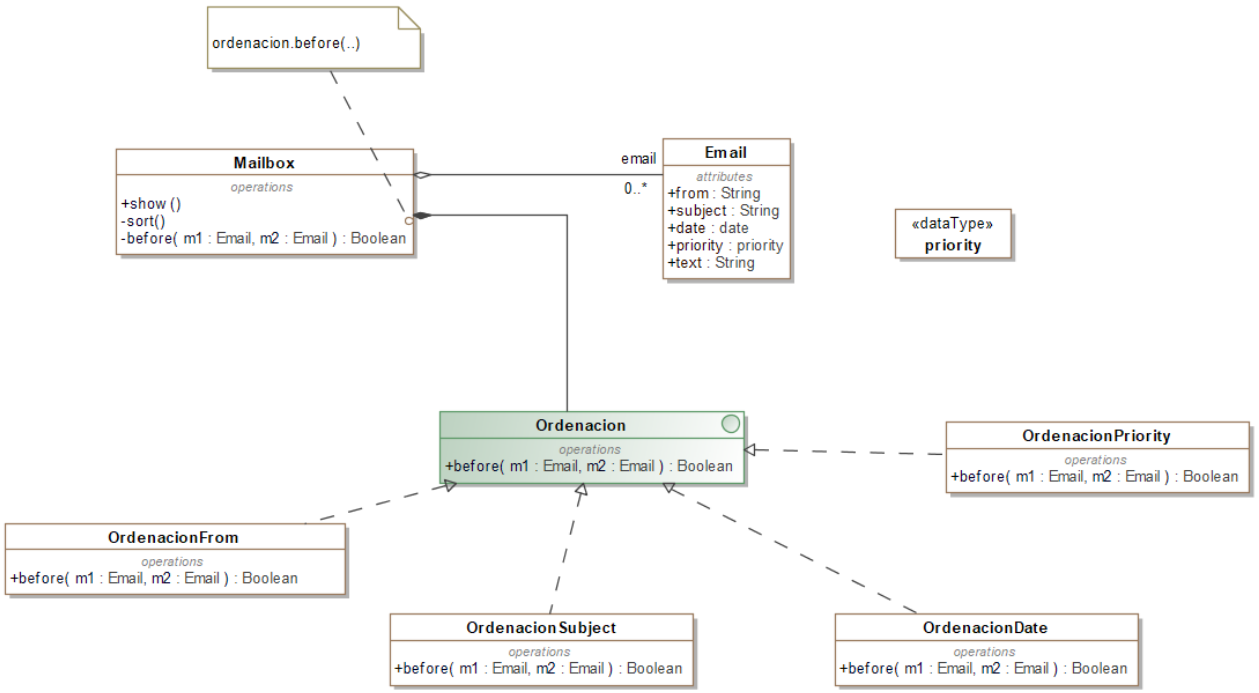
```
package Ejercicio2.c;

public class Verde extends Estado {

    private static Verde instance;

    private Verde() {

    }

    public static Verde getInstance() {
        if (instance == null) {
            instance = new Verde();
        }
        return instance;
    }

    @Override
    public void abrir(Buffer buffer) {
        throw new RuntimeException("No se puede abrir estando en
            el estado verde");
    }

    @Override
    public void cerrar(Buffer buffer) {

        if (buffer instanceof Biestable) {
            buffer.setEstado(Rojo.getInstance());
        } else {
            buffer.setEstado(Amarillo.getInstance());
        }
    }

    @Override
    public void estado(Buffer buffer) {
        System.out.println("Abierto");
    }
}
```

3. Ejercicio 3

3.1. Explicación textual

En este ejercicio hemos usado el patrón estrategia porque nos requería diferenciar el algoritmo según diferentes criterios, permitiendo así definir una familia de algoritmos, encapsulandolos en clases distintas y haciéndolos intercambiables a su vez. Para desarrollarlo hemos creado la interfaz 'Ordenación', la cual implementan las subclases 'OrdenacionFrom', 'OrdenacionSubject', 'OrdenacionPriority' y 'OrdenacionDate', que tal y como su nombre indica cada subclase hace referencia a su propio algoritmo del método 'before()', definiendo así una clase estrategia por cada uno de ellos.

3.2. Modelo



3.3. Código

Clase: Email.Java

```
package Ejercicio3;
import java.util.Date;
```

```
public class Email {
    public String from;
    public String Subject;
    public Date date;
    public Priority priority;
    public String text;

}
```

Clase: Mailbox.Java

```
package Ejercicio3;
import java.util.ArrayList;
import java.util.List;

    public class Mailbox {
        public List<Email> email;
        private Ordenacion criterio;


        public Mailbox() {
            email = new ArrayList<Email>();
        }

        public Mailbox(List<Email> lista) {
            email = new ArrayList<Email>();
            email = lista;
        }


        public void setCriterio(Ordenacion criterio) {
            this.criterio = criterio;
        }

        // Operations

        public void show() {
        }


        private void sort() {
            for (int i = 2; i <= email.size(); i++) {
                for (int j = email.size(); j >= i; j--) {
```

```
                if (before(email.get(j), email.get(j-1))) { //
                    'at' cambiado por 'get' porque hemos
                   definido un ArrayList<Email>
                    Email aux = email.get(j);
                    email.set(j, email.get(j-1));
                    email.set(j-1, aux);
                }
            }
        }
    }

    private Boolean before(Email m1, Email m2) {
        return this.criterio.before(m1, m2);
    }

}
```

Interfaz: Ordenacion.Java

```
package Ejercicio3;

public interface Ordenacion {
    public Boolean before(Email m1, Email m2);
}
```

Clase: OrdenacionDate.Java

```
package Ejercicio3;

public class OrdenacionDate implements Ordenacion{

    public Boolean before(Email m1, Email m2) { // m1 va detras
        de m2
        boolean resultado;

        if(m1.date.compareTo(m2.date) > 0) { // <
            resultado = true;
        }else {
            resultado = false;
        }

        return resultado;
    }
}
```

Clase: OrdenacionFrom.Java

```
package Ejercicio3;

public class OrdenacionFrom implements Ordenacion {

    public Boolean before(Email m1, Email m2) { // m1 va detras
        de m2
        boolean resultado;

        if (m1.from.compareTo(m2.from) > 0) {
            resultado = true;
        } else {
            resultado = false;
        }

        return resultado;
    }
}
```

Clase: OrdenacionPriority.Java

```
package Ejercicio3;

public class OrdenacionPriority implements Ordenacion{

    public Boolean before(Email m1, Email m2) { // m1 va detras
        de m2
        boolean resultado;

        if(m1.priority.prioridad < m2.priority.prioridad) {
            resultado = true;
        }else {
            resultado = false;
        }

        return resultado;
    }
}
```

Clase: OrdenacionSubject.Java

```
package Ejercicio3;

public class OrdenacionSubject implements Ordenacion{
```

```
    public Boolean before(Email m1, Email m2) { // m1 va detras
        de m2
        boolean resultado;

        if(m1.Subject.compareTo(m2.Subject) > 0) {
            resultado = true;
        }else {
            resultado = false;
        }

        return resultado;
    }
}
```

Clase: Priority.Java

```
package Ejercicio3;

public class Priority {
    public int prioridad;
}
```

4. Patrones de diseño usados

Proxy: (https://refactoring.guru/es/design-patterns/proxy)
State: (https://refactoring.guru/es/design-patterns/state)
Singleton: (https://refactoring.guru/es/design-patterns/singleton)
Strategy: (https://refactoring.guru/es/design-patterns/strategy)