P4

Francisco Javier Hernández Martín, Jose Luis Bueno Pachón, Carlos Marín Corbera, Carmen González Ortega, Altair Bueno Calvente

UNIVERSIDAD
DE MÁLAGA

15 jan 2022

Contents

Preámbulo

Para la realización de esta práctica, hemos optado por utilizar un estilo de programación estandarizado para el lenguaje de programación Java, conocido como **Google Java Style Guide**[1]. Además, para verificar el correcto funcionamiento de la implementación de los distintos ejercicios, hemos creado una serie de test unitarios usando el framework de pruebas **jUnit5**[2].

Ejercicio 1: Los interfaces selectivos

Apartado A

En Java contamos con los **modificadores de acceso** para exportar de forma selectiva las distintas clases y atributos del sistema. Por ejemplo, para crear una clase `Foo` con visibilidad pública escribimos lo siguiente:

```
// Modificador de acceso público
// vvv
public class Foo {/* ... */}
```

En Java, los distintos modificadores de acceso son:

- **Visibilidad privada (`private`)**: Es la visibilidad más restrictiva. El elemento marcado con el atributo `private` será solo visible desde la clase en la que se declara.
- **Visibilidad de paquete**: Es la visibilidad por defecto de Java. Aquellos elementos que no han sido explícitamente marcados serán considerados visibles desde el paquete. Esto significa que el elemento se comporta con visibilidad pública en aquellas clases del mismo paquete, pero de forma privada con las clases externas al paquete.
- **Visibilidad protegida (`protected`)**: Al igual que la visibilidad de paquete, la visibilidad protegida impide que las clases externas accedan a aquellos elementos marcados con el atributo `protected`, aunque permite a sus subclases y clases del mismo paquete acceder a dichos elementos.
- **Visibilidad pública (`public`)**: Permite que el elemento marcado con el atributo `public` sea visible desde cualquier clase.

En la siguiente tabla se resumen las características principales de las distintas visibilidades de Java:

[1] https://google.github.io/styleguide/javaguide.html
[2] https://junit.org/junit5/

Modificador	Misma clase	Mismo paquete	Distinto paquete, subclase	Distinto paquete
Privado	Sí	No	No	No
De paquete	Sí	Sí	No	No
Protegido	Sí	Sí	Sí	No
Público	Sí	Sí	Sí	Sí

Por otra parte, el lenguaje Eiffel profundiza más en el mecanismo de exportación selectiva, otorgando al programador de un control granular sobre el acceso a los elementos de las clases. Los distintos modificadores de acceso Eiffel, según Wikipedia[3], son:

- **Visibilidad pública (`feature`)**: Al igual que en Java, permite que sea visible desde cualquier clase.
- **Visibilidad protegida (`feature {}`)**: Al igual que en Java, permite que sea visible desde sus subclases y clases del mismo paquete.
- **Visibilidad *selectiva* (`feature {class1,class2,class3...}`)**: Permite que dicho elemento sea visible a las clases seleccionadas, actuando como privada para las demás clases.

Podemos observar que ambos lenguajes tienen distintos modificadores de acceso; Java incluye visibilidad **privada** y **de paquete**, mientras que Eiffel incluye ***selectiva***, siendo esta última un preciso que las demás visibilidades. Podemos entender los mecanismos como distintos enfoques para controlar el acceso: los modificadores de acceso de Java dependen del rango o alcance (misma clase, subclases, paquete, fuera del paquete…) y los modificadores de Eiffel dependen de quién intente acceder a ese elemento/característica.

[3] https://en.wikipedia.org/wiki/Eiffel_(programming_language)#Scoping

Apartado B

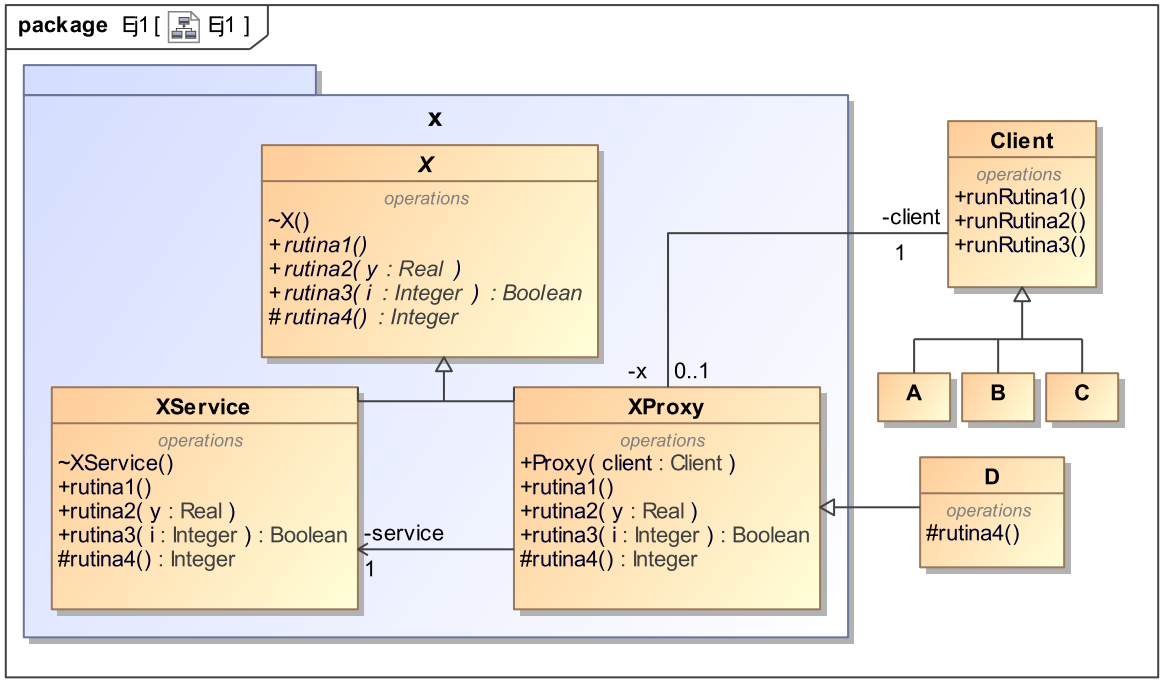

Figure 1: Modelo de diseño Ej1

Para poder emular el mecanismo de exportación selectiva de Eiffel, debemos controlar el acceso al servicio (la clase X) y para ello utilizamos el Patrón **Representante** o **Proxy** y el Patrón **Envoltorio** o **Wrapper**.

En el paquete llamado **X** se encuentran las siguientes clases:

- `X`
- `XService`
- `XProxy`

La clase abstracta `X`, como debe permitir operaciones de visibilidad protegida, no puede ser una Interfaz, así que hemos decidido modelar el mismo comportamiento mediante una clase abstracta. Las dos clases que la implementan son `XProxy` (encargada de controlar el acceso) y `XService` (encargada de ofrecer el servicio), siendo esta clase la que denominan 'X' en el enunciado de la práctica. La clase `XProxy` se encargará de verificar si el cliente cuenta con los permisos suficientes para poder realizar dicha llamada, y en caso contrario, lanzará una excepción de tipo `IllegalCallerException`.

Las clases `A`, `B` y `C` heredan de la clase `Cliente` y se encontrarán en otro paquete para prote-

ger el acceso a los métodos de `XService`, teniendo así una referencia únicamente a la clase `XProxy`, a diferencia del patrón **Representante** donde debería interactuar el cliente con la interfaz. Esto implica que la clase `XProxy` actúe a su vez como una clase envoltorio, por lo que se ha aplicado el patrón **Envoltorio** de esta manera. El método `runRutina1` será visible para todos, los métodos `runRutina2` y `runRutina3` serán visibles desde el exterior, pero su ejecución solo será satisfactoria en caso de que el cliente cumpla los requisitos establecidos. Por último, la clase `D` no hereda de la clase `Cliente`, sino directamente de `XProxy`, e implementa el método `runRutina4()` con visibilidad protegida.

La ventaja que nos proporciona aplicar este Patrón es que los clientes no interactúan directamente con `XService`, relegando el control del acceso a la clase `XProxy` y dejando que `XService` únicamente se encargue de aplicar los métodos. La desventaja que podemos observar es que al no existir soporte en Java para el control granular de permisos, el acceso se comprueba en tiempo de ejecución en vez de en tiempo de compilación.

Ejercicio 2: Triestables

Apartado A

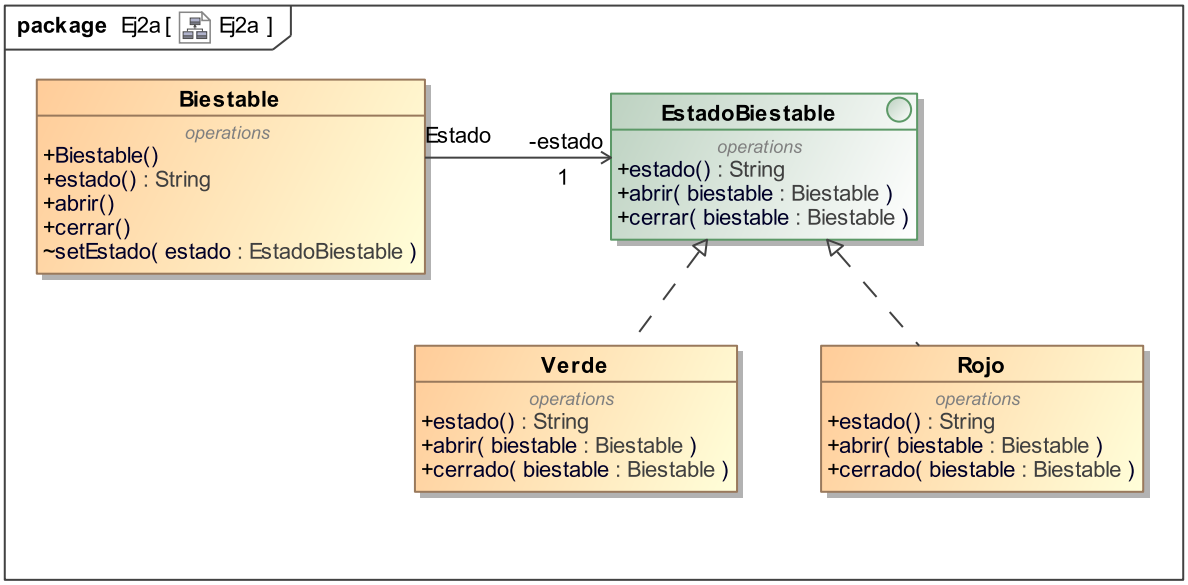

Figure 2: Modelo de diseño apartado A

A la hora de implementar este modelo hemos decidido utilizar el Patrón **Estado** que se asemeja notablemente con lo descrito en la práctica; un dispositivo software que alterne entre los

estados Rojo y Verde, semejante a un semáforo.

Para ello tenemos un objeto de la clase `Biestable` que representa el contexto del Patrón **Estado** y tiene los métodos `abrir()`, `cerrar()`, `estado()` y `setEstado()`. La función `estado()` devuelve un mensaje indicando en qué estado se encuentra (cerrado o abierto). El método `setEstado()` de visibilidad de paquete será llamado en el constructor de la clase, inicializándola a Rojo. Las funciones `cerrar()` y `abrir()` harán que el Biestable pase de un estado a otro dependiendo de si el cambio está permitido. Por ejemplo, si se intenta ejecutar el método `abrir()` en un Biestable que se encuentre en estado Verde, se elevará una excepción.

El comportamiento de estas funciones está descrito en las clases `Rojo` y `Verde`, que implementan la interfaz `EstadoBiestable`. Para poder ser utilizado por la clase `Biestable`, esta tiene un atributo referenciando a la interfaz.

Apartado B

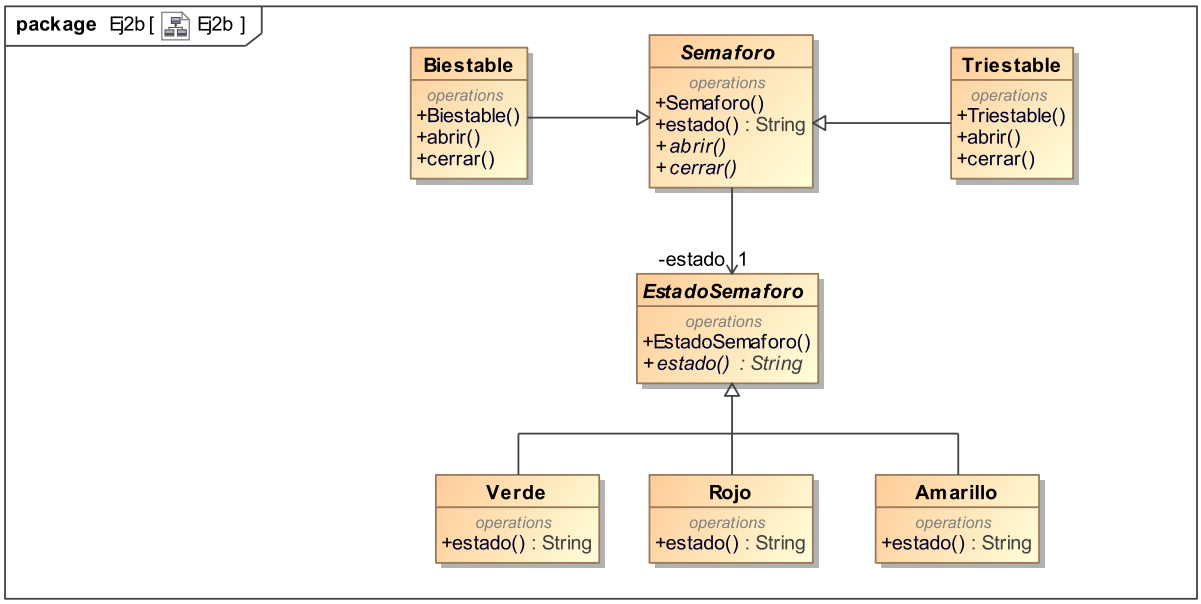

Figure 3: Modelo de diseño apartado B

Este apartado consiste en añadir un nuevo tipo de semáforo con tres estados.

Aprovechando el código del primer apartado, hemos decidido transformar `EstadoBiestable` en la clase abstracta `EstadoSemaforo`, manteniendo el Patrón **Estado**, pero esta vez relegando el comportamiento de abrir y cerrar a los semáforos concretos, es decir, el comportamiento de los métodos`abrir()` y `cerrar()` son implementados en las clases `Biestable` y `Triestable`. Estas

clases heredan de la clase abstracta `Semaforo`, que a su vez tiene un atributo referenciando a la clase abstracta `EstadoSemaforo`.

Al cambiar dónde se encuentra el peso de las operaciones encargadas de los cambios de estado, y ya no estar dentro de los propios estados, podemos considerar que el Patrón **Estado** aplicado funciona como un Patrón **Estrategia**, en el cual cada una de las estrategias o, en este contexto, cada uno de los estados del semáforo, son independientes entre ellos.

Apartado C

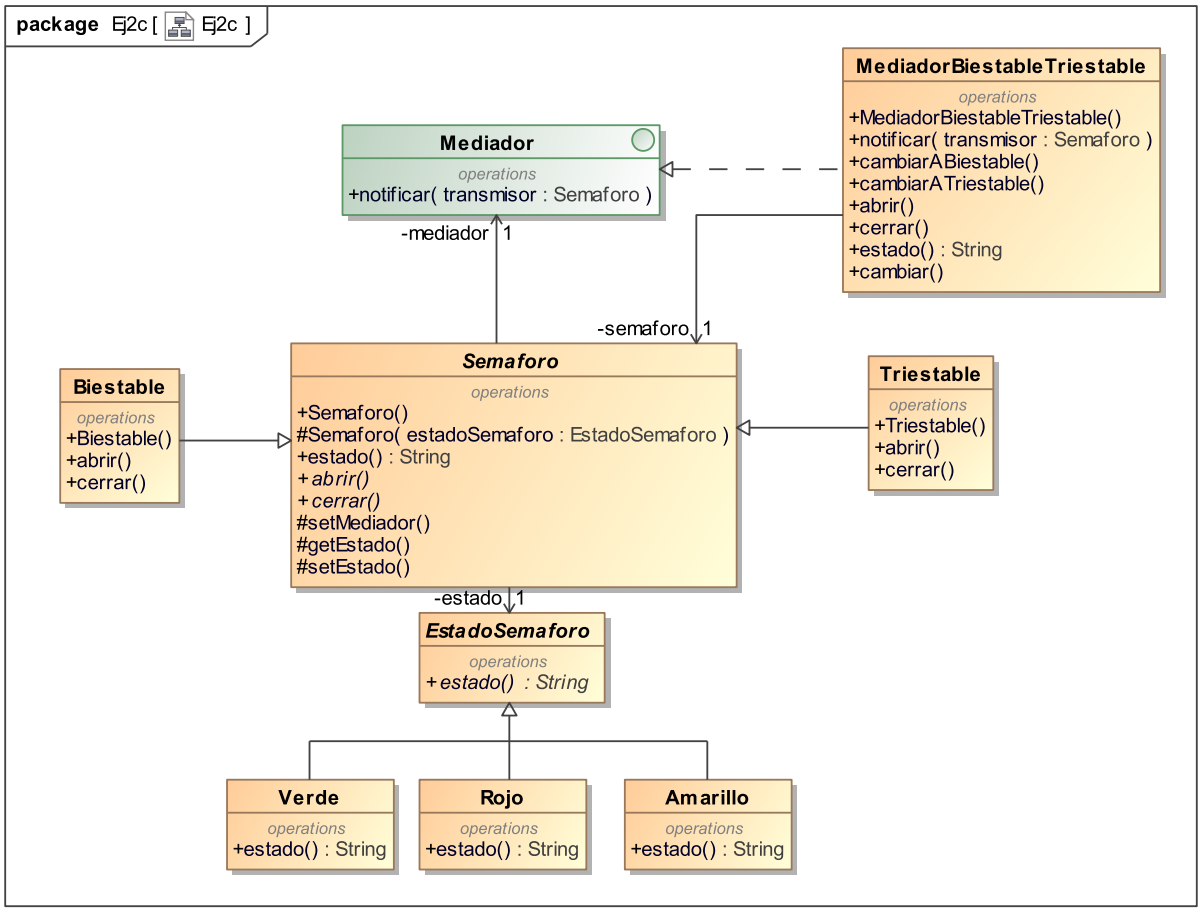

Figure 4: Modelo de diseño apartado C

Para este apartado hemos aplicado el Patrón **Estado** y el Patrón **Mediador** con el objetivo de poder cambiar dinámicamente, en tiempo de ejecución, el tipo de Semáforo.

Para ello partimos del apartado anterior añadiendo la clase `MediadorBiestableTriestable`, la cual implementa la interfaz `Mediador`, y tiene un atributo referenciando a la clase abstracta

`Semaforo`. Además, `Semaforo` añade los métodos `setMediador()`, `getEstado()`, `setEstado()` y un constructor, todos ellos de visibilidad protegida. Estos son utilizados por la clase `MediadorBiestableTriestable`, que funciona, en parte, como envoltorio de los métodos `abrir()`, `cerrar()`, y `estado()` de la clase `Semaforo`.

De esta manera, el cliente se comunicará con el dispositivo a través de la clase `MediadorBiestableTriestable` que se encargará de cambiar el estado del dispositivo según el Semáforo que esté activo en ese momento, y cambiará de tipo de Semáforo a través del método `cambiar()`, que llamará al método `notificar()`, definido en la interfaz `Mediador`, que a su vez llamará a `cambiarABiestable()` o `cambiarATriestable()` dependiendo del Semáforo activo.

Ejercicio 3: Cliente de correo e-look

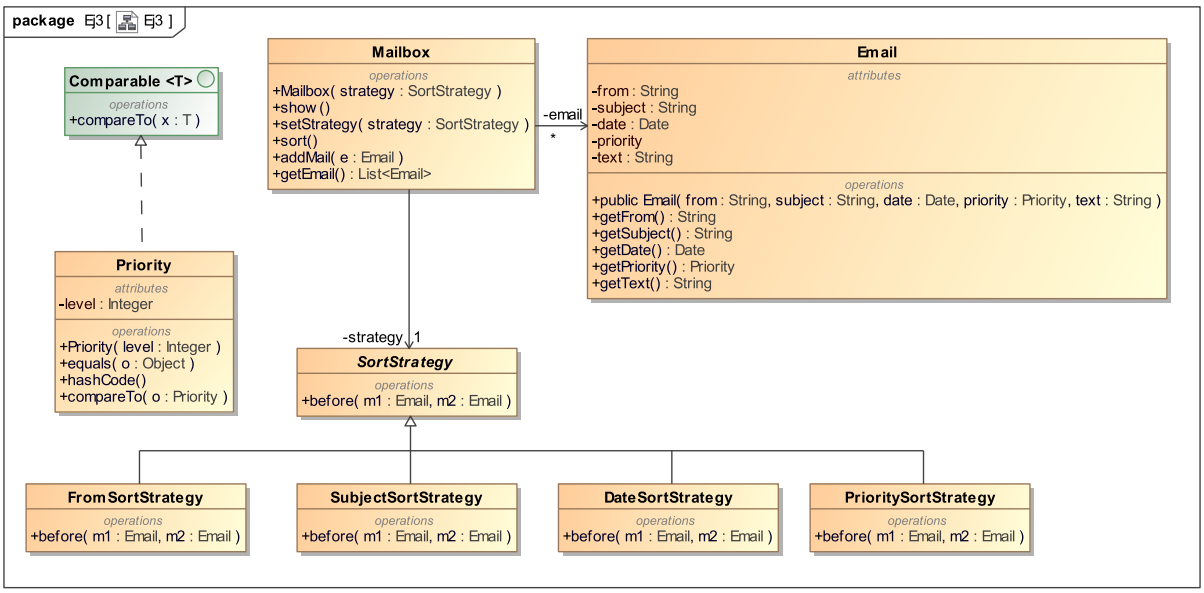

Figure 5: Modelo de diseño Ej3

Para la resolución de este ejercicio hemos decidido aplicar el Patrón **Estrategia**, pues necesitamos distintas variantes del algoritmo aplicado por la función `before()` y cambiar entre estas variantes en tiempo de ejecución.

La clase que representa el contexto del Patrón es la clase `Mailbox` con los métodos `show()`, `sort()`, `setStrategy()`, `getEmail()` y `addMail()`, con dos atributos referenciando uno a la clase `Email` y otro a la clase abstracta `SortStrategy`.

La clase `Email` representa un Email y posee únicamente sus atributos (`from`, `subject`, `date`, `priority` y `text`), el constructor y los getters correspondientes.

Esta clase posee un atributo referenciando a la clase abstracta `SortStrategy()` con el método `before()`, que recibe dos instancias de la clase `Email`. Este método es implementado de formas diferentes en las distintas clases que heredan de `SortStrategy()`, que son:

- `FromSortStrategy`: Devuelve `true` si los emails por parámetro están ordenados según su atributo `from`.
- `SubjectSortStrategy`: Devuelve `true` si los emails por parámetro están ordenados según su atributo `subject`.
- `DateSortStrategy`: Devuelve `true` si los emails por parámetro están ordenados según su atributo `date`.
- `PrioritySortStrategy`: Devuelve `true` si los emails por parámetro están ordenados según su atributo `priority`.

Para implementar la prioridad que es un `dataType` hemos creado la clase `Priority` que posee un atributo `level` que indica el nivel de prioridad del email. Además, implementa la interfaz `Comparable` de Java, para poder sobreescribir el método `compareTo()` y definir los niveles de prioridad.

Código Java

Ej1

`A.java`

```
package Ej1;

public class A extends Client {}
```

`B.java`

```
package Ej1;

public class B extends Client {}
```

C.java

```
package Ej1;

public class C extends Client {}
```

Client.java

```
package Ej1;

import Ej1.x.XProxy;

/** Clase base para los clientes de X */
public class Client {
  private final XProxy x;

  public Client() {
    x = new XProxy(this);
  }

  public void runRutina1() {
    x.rutina1();
  }

  public void runRutina2() {
    x.rutina2(2.0);
  }

  public void runRutina3() {
    x.rutina3(10);
  }
}
```

D.java

```
package Ej1;
```

```
import Ej1.x.XProxy;

public class D extends XProxy {
  /** Construye un nuevo proxy que guarda una instancia de XService */
  public D() {
    super(null);
  }

  @Override
  protected int rutina4() {
    return super.rutina4();
  }
}
```

X.java

```
package Ej1.x;

/** Define una interfaz común para el Proxy y su servicio */
abstract class X {
  X() {}

  public abstract void rutina1();

  public abstract void rutina2(double y);

  public abstract boolean rutina3(int i);

  protected abstract int rutina4();
}
```

XProxy.java

```
package Ej1.x;

import Ej1.A;
import Ej1.B;
```

```
import Ej1.C;
import Ej1.Client;

/** Protege la clase XService de accesos no deseados */
public class XProxy extends X {
  private final Client client;
  private final XService service;

  /**
   * Construye un nuevo proxy que guarda una instancia de XService
   *
   * @param client clase que llama los métodos
   */
  public XProxy(Client client) {
    this.client = client;
    this.service = new XService();
  }

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

  @Override
  public void rutina2(double y) {
    if (!(client instanceof A) && !(client instanceof B))
      throw new IllegalCallerException("Caller must be an instance of class A or B");
    service.rutina2(y);
  }

  @Override
  public boolean rutina3(int i) {
    if (!(client instanceof A) && !(client instanceof C))
      throw new IllegalCallerException("Caller must be an instance of class A or C");
    return service.rutina3(i);
  }

  @Override
```

```
  protected int rutina4() {
    return service.rutina4();
  }
}
```

XService.java

```
package Ej1.x;

/** Provee los servicios de X */
class XService extends X {
  XService() {}

  @Override
  public void rutina1() {
    // ...
  }

  @Override
  public void rutina2(double y) {
    // ...

  }

  @Override
  public boolean rutina3(int i) {
    // ...
    return false;
  }

  @Override
  protected int rutina4() {
    // ...
    return 0;
  }
}
```

Ej2a

Biestable.java

```
package Ej2a;

/**
 * Semáforo Biestable
 *
 * <p>El semáforo Biestable transita entre dos estados posibles utilizando los métodos {@lir
 * Biestable#abrir()} y {@link Biestable#cerrar()}
 */
public class Biestable {

  private EstadoBiestable estado;

  /** Crea un nuevo semáforo Biestable en el estado inicial Rojo */
  public Biestable() {
    this.estado = new Rojo();
  }

  /** Devuelve una cadena indicando el estado actual */
  public String estado() {
    return estado.estado();
  }

  /**
   * Abre el semáforo
   *
   * @throws IllegalCallerException Si la máquina de estados no permite esa transición
   */
  public void abrir() {
    estado.abrir(this);
  }

  /**
   * Cierra el semáforo
   *
```

```
   * @throws IllegalCallerException Si la máquina de estados no permite esa transición
   */
  public void cerrar() {
    estado.cerrar(this);
  }

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

EstadoBiestable.java

```
package Ej2a;

/** Interfaz común para los posibles estados de un semáforo Biestable */
public interface EstadoBiestable {
  /** Devuelve una cadena indicando el estado actual */
  String estado();

  /**
   * Realiza las transiciones de la operación {@link Biestable#abrir()} para un semáforo Bie
   *
   * @param biestable Semáforo a transitar
   */
  void abrir(Biestable biestable);

  /**
   * Realiza las transiciones de la operación {@link Biestable#cerrar()} para un semáforo Bi
   *
   * @param biestable Semáforo a transitar
   */
  void cerrar(Biestable biestable);
}
```

Rojo.java

```
package Ej2a;

class Rojo implements EstadoBiestable {
  @Override
  public String estado() {
    return "cerrado";
  }

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

  @Override
  public void cerrar(Biestable biestable) {
    throw new IllegalStateException("Transición no válida");
  }
}
```

Verde.java

```
package Ej2a;

class Verde implements EstadoBiestable {
  @Override
  public String estado() {
    return "abierto";
  }

  @Override
  public void abrir(Biestable biestable) {
    throw new IllegalStateException("Transición no válida");
  }

  @Override
  public void cerrar(Biestable biestable) {
```

```
    biestable.setEstado(new Rojo());
  }
}
```

Ej2b

Amarillo.java

```
package Ej2b;

public class Amarillo extends EstadoSemaforo {
  @Override
  public String estado() {
    return "precaución";
  }
}
```

Biestable.java

```
package Ej2b;

/**
 * Semáforo Biestable
 *
 * <p>El semáforo Biestable transita entre dos estados posibles utilizando los métodos {@lin
 * Biestable#abrir()} y {@link Biestable#cerrar()}
 */
public class Biestable extends Semaforo {
  @Override
  public void abrir() {
    if (this.estado instanceof Verde) {
      throw new IllegalStateException();
    } else {
      this.estado = new Verde();
    }
  }
```

```
  @Override
  public void cerrar() {
    if (this.estado instanceof Rojo) {
      throw new IllegalStateException();
    } else {
      this.estado = new Rojo();
    }
  }
}
```

EstadoSemaforo.java

```
package Ej2b;

/** Define el comportamiento común para los posibles estados de un semáforo */
public abstract class EstadoSemaforo {
  public EstadoSemaforo() {}

  public abstract String estado();
}
```

Rojo.java

```
package Ej2b;

public class Rojo extends EstadoSemaforo {
  @Override
  public String estado() {
    return "cerrado";
  }
}
```

Semaforo.java

```
package Ej2b;

/** Representa un semáforo que cambia de estado */
```

```
public abstract class Semaforo {
  protected EstadoSemaforo estado;

  /** Crea un nuevo semáforo y establece su estado inicial en Rojo */
  public Semaforo() {
    estado = new Rojo();
  }

  /** Devuelve una cadena indicando el estado actual */
  public String estado() {
    return estado.estado();
  }

  /** Abre el semáforo */
  public abstract void abrir();

  /** Cierra el semáforo */
  public abstract void cerrar();
}
```

Triestable.java

```
package Ej2b;

/**
 * Semáforo Triestable
 *
 * <p>El semáforo Triestable transita entre tres estados posibles utilizando los métodos {@l
 * Triestable#abrir()} y {@link Triestable#cerrar()}
 */
public class Triestable extends Semaforo {
  @Override
  public void abrir() {
    if (this.estado instanceof Rojo) {
      this.estado = new Amarillo();
    } else if (this.estado instanceof Amarillo) {
      this.estado = new Verde();
```

```
    } else {
      throw new IllegalStateException();
    }
  }

  @Override
  public void cerrar() {
    if (this.estado instanceof Verde) {
      this.estado = new Amarillo();
    } else if (this.estado instanceof Amarillo) {
      this.estado = new Rojo();
    } else {
      throw new IllegalStateException();
    }
  }
}
```

Verde.java

```
package Ej2b;

public class Verde extends EstadoSemaforo {
  public String estado() {
    return "abierto";
  }
}
```

Ej2c

Amarillo.java

```
package Ej2c;

public class Amarillo extends EstadoSemaforo {
  @Override
  public String estado() {
    return "precaución";
```

```
  }
}
```

Biestable.java

```
package Ej2c;

/**
 * Semáforo Biestable
 *
 * <p>El semáforo Biestable transita entre dos estados posibles utilizando los métodos {@lin
 * Ej2c.Biestable#abrir()} y {@link Ej2c.Biestable#cerrar()}
 */
public class Biestable extends Semaforo {
  public Biestable() {
    super();
  }

  Biestable(EstadoSemaforo estado) {
    super(estado);
  }

  @Override
  public void abrir() {
    if (this.getEstado() instanceof Verde) {
      throw new IllegalStateException();
    } else {
      this.setEstado(new Verde());
    }
  }

  @Override
  public void cerrar() {
    if (this.getEstado() instanceof Rojo) {
      throw new IllegalStateException();
    } else {
      this.setEstado(new Rojo());
```

```
    }
  }
}
```

EstadoSemaforo.java

```
package Ej2c;

/** Define el comportamiento común para los posibles estados de un semáforo */
public abstract class EstadoSemaforo {
  public abstract String estado();
}
```

Mediador.java

```
package Ej2c;

/** Interfaz necesaria para aplicar el patrón mediador*/
public interface Mediador {

  /** Método que alterna el semáforo entre biestable y triestable y que implementa {@link Ej
  void notificar(Semaforo transmisor);
}
```

MediadorBiestableTriestable.java

```
package Ej2c;

/** Interfaz que se encarga de alternar los semáforos entre biestable y triestable */
public class MediadorBiestableTriestable implements Mediador {
  private Semaforo semaforo;

  public MediadorBiestableTriestable() {
    semaforo = new Biestable();
    semaforo.setMediador(this);
  }
```

```
  @Override
  public void notificar(Semaforo transmisor) {
    semaforo = transmisor;
    if (transmisor instanceof Biestable) {
      cambiarATriestable();
    } else {
      cambiarABiestable();
    }
  }

  public void cambiarABiestable() {
    if (semaforo.getEstado() instanceof Amarillo) {
      throw new IllegalStateException();
    }
    semaforo = new Biestable(semaforo.getEstado());
  }

  public void cambiarATriestable() {
    semaforo = new Triestable(semaforo.getEstado());
  }

  public void abrir() {
    semaforo.abrir();
  }

  public void cerrar() {
    semaforo.cerrar();
  }

  public void cambiar() {
    notificar(semaforo);
  }

  public String estado() {
    return semaforo.estado();
  }
}
```

Rojo.java

```
package Ej2c;

public class Rojo extends EstadoSemaforo {
  @Override
  public String estado() {
    return "cerrado";
  }
}
```

Semaforo.java

```
package Ej2c;

/** Representa un semáforo que cambia de estado */
public abstract class Semaforo{
    private Mediador mediador;
    private EstadoSemaforo estado;

    /** Crea un nuevo semáforo y establece su estado inicial en Rojo */
    public Semaforo() {
        estado = new Rojo();
    }

    Semaforo(EstadoSemaforo estado){
        this.estado = estado;
    }

    protected EstadoSemaforo getEstado(){
        return this.estado;
    }

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

    protected void setMediador(Mediador mediador){
```

```
        this.mediador = mediador;
    }

    /** Devuelve una cadena indicando el estado actual */
    public String estado(){
        return estado.estado();
    }

    /** Abre el semáforo */
    public abstract void abrir();

    /** Cierra el semáforo */
    public abstract void cerrar();

}
```

Triestable.java

```
package Ej2c;

/**
 * Semáforo Triestable
 *
 * <p>El semáforo Triestable transita entre tres estados posibles utilizando los métodos {@l
 * Ej2c.Triestable#abrir()} y {@link Ej2c.Triestable#cerrar()}
 */
public class Triestable extends Semaforo {
  public Triestable() {
    super();
  }

  Triestable(EstadoSemaforo estado) {
    super(estado);
  }

  @Override
  public void abrir() {
```

```
    if (this.getEstado() instanceof Rojo) {
      this.setEstado(new Amarillo());
    } else if (this.getEstado() instanceof Amarillo) {
      this.setEstado(new Verde());
    } else {
      throw new IllegalStateException();
    }
  }

  @Override
  public void cerrar() {
    if (this.getEstado() instanceof Verde) {
      this.setEstado(new Amarillo());
    } else if (this.getEstado() instanceof Amarillo) {
      this.setEstado(new Rojo());
    } else {
      throw new IllegalStateException();
    }
  }
}
```

Verde.java

```
package Ej2c;

public class Verde extends EstadoSemaforo {
  @Override
  public String estado() {
    return "abierto";
  }
}
```

Ej3

DateSortStrategy.java

```
package Ej3;
```

```
/** Ordena los emails por fecha */
public class DateSortStrategy implements SortStrategy {
  @Override
  public boolean before(Email e1, Email e2) {
    assert e1 != null;
    assert e2 != null;

    return e1.getDate().compareTo(e2.getDate()) > 0;
  }
}
```

Email.java

```
package Ej3;

import java.util.Date;

/** Representa un email */
public class Email {
  private final String from;
  private final String subject;
  private final Date date;
  private final Priority priority;
  private final String text;

  public Email(String from, String subject, Date date, Priority priority, String text) {
    assert from != null;
    assert subject != null;
    assert date != null;
    assert priority != null;
    assert text != null;

    this.from = from;
    this.subject = subject;
    this.date = date;
    this.priority = priority;
```

```
    this.text = text;
  }

  public String getFrom() {
    return from;
  }

  public String getSubject() {
    return subject;
  }

  public Date getDate() {
    return date;
  }

  public Priority getPriority() {
    return priority;
  }

  public String getText() {
    return text;
  }
}
```

FromSortStrategy.java

```
package Ej3;

/** Ordena los emails por su remitente */
public class FromSortStrategy implements SortStrategy {
  @Override
  public boolean before(Email e1, Email e2) {
    assert e1 != null;
    assert e2 != null;

    return e1.getFrom().compareTo(e2.getFrom()) > 0;
  }
```

```
}
```

Mailbox.java

```
package Ej3;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/** Representa un buzón de email */
public class Mailbox {
  private final List<Email> email;
  private SortStrategy strategy;

  /**
   * Crea un nuevo buzón con la {@link SortStrategy} proporcionada
   *
   * @param strategy Estrategia para ordenar los emails
   */
  public Mailbox(SortStrategy strategy) {
    setStrategy(strategy);
    email = new ArrayList<>();
  }

  /**
   * Añade un nuevo email al buzón
   *
   * @param e email a añadir
   */
  public void addMail(Email e) {
    assert e != null;

    email.add(e);

    assert email.contains(e);
  }
```

```
  /** Muestra por consola los emails del buzón ya ordenados */
  public void show() {
    for (Email e : email) {
      System.out.println(e.toString() + "\n");
    }
  }

  public void setStrategy(SortStrategy strategy) {
    assert strategy != null;

    this.strategy = strategy;
  }

  public List<Email> getEmail() {
    return Collections.unmodifiableList(email);
  }

  /** Ordena los emails según la estrategia elegida */
  public void sort() {
    for (int i = email.size() - 1; i > 0; i--) {
      for (int j = 0; j < i; j++) {
        if (strategy.before(email.get(j), email.get(j + 1))) {
          var temp = email.get(j + 1);
          email.set(j + 1, email.get(j));
          email.set(j, temp);
        }
      }
    }
  }
}
```

Priority.java

```
package Ej3;

import java.util.Objects;
```

```
/** Representa la prioridad de un email */
public class Priority implements Comparable<Priority> {
  private final int level;

  /**
   * Crea una nueva instancia con un determinado nivel de prioridad
   *
   * @param level nivel de prioridad
   */
  public Priority(int level) {
    this.level = level;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Priority)) return false;
    Priority priority = (Priority) o;
    return level == priority.level;
  }

  @Override
  public int hashCode() {
    return Objects.hash(level);
  }

  @Override
  public int compareTo(Priority o) {
    return Integer.compare(this.level, o.level);
  }
}
```

PrioritySortStrategy.java

```
package Ej3;
```

```
/** Ordena los emails por {@link Priority} */
public class PrioritySortStrategy implements SortStrategy {
  @Override
  public boolean before(Email e1, Email e2) {
    assert e1 != null;
    assert e2 != null;

    return e1.getPriority().compareTo(e2.getPriority()) > 0;
  }
}
```

SortStrategy.java

```
package Ej3;

public interface SortStrategy {
  /**
   * Compara dos emails
   *
   * @param e1 primer email
   * @param e2 segundo email
   * @return true si el primer email está antes que el segundo email
   */
  boolean before(Email e1, Email e2);
}
```

SubjectSortStrategy.java

```
package Ej3;

/** Ordena los emails por su asunto */
public class SubjectSortStrategy implements SortStrategy {
  @Override
  public boolean before(Email e1, Email e2) {
    assert e1 != null;
    assert e2 != null;
```

```
    return e1.getSubject().compareTo(e2.getSubject()) > 0;
  }
}
```

jUnit tests

Ej1.java

```
import Ej1.A;
import Ej1.B;
import Ej1.C;
import Ej1.Client;
import org.junit.Test;

public class Ej1 {
  @Test
  public void rutina1Client() {
    var client = new Client();
    client.runRutina1();
  }

  @Test(expected = IllegalCallerException.class)
  public void rutina2Client() {
    var client = new Client();
    client.runRutina2();
  }

  @Test(expected = IllegalCallerException.class)
  public void rutina3Client() {
    var client = new Client();
    client.runRutina3();
  }

  @Test
  public void rutina1A() {
    var a = new A();
    a.runRutina1();
  }
```

```
@Test
public void rutina2A() {
  var a = new A();
  a.runRutina2();
}

@Test
public void rutina3A() {
  var a = new A();
  a.runRutina3();
}

@Test
public void rutina1B() {
  var b = new B();
  b.runRutina1();
}

@Test
public void rutina2B() {
  var b = new B();
  b.runRutina2();
}

@Test(expected = IllegalCallerException.class)
public void rutina3B() {
  var b = new B();
  b.runRutina3();
}

@Test
public void rutina1C() {
  var c = new C();
  c.runRutina1();
}

@Test(expected = IllegalCallerException.class)
```

```
  public void rutina2C() {
    var c = new C();
    c.runRutina2();
  }

  @Test
  public void rutina3C() {
    var c = new C();
    c.runRutina3();
  }
}
```

Ej2a.java

```
import Ej2a.Biestable;
import org.junit.Test;

public class Ej2a {
    @Test
    public void cambioDeEstadosCorrecto(){
        Biestable b = new Biestable();
        assert b.estado().equals("cerrado");
        b.abrir();
        assert b.estado().equals("abierto");
        b.cerrar();
        assert b.estado().equals("cerrado");
    }

    @Test (expected = IllegalStateException.class)
    public void noPuedeCerrarEnRojo(){
        Biestable b = new Biestable();
        b.cerrar();
    }

    @Test (expected = IllegalStateException.class)
    public void noPuedeAbrirEnVerde(){
        Biestable b = new Biestable();
```

```
        b.abrir();
        b.abrir();
    }
}
```

Ej2b.java

```
import org.junit.Test;

import Ej2b.*;

public class Ej2b {
    @Test(expected = IllegalStateException.class)
    public void testBiestable1(){
        Semaforo s = new Biestable();
        s.abrir();
        s.abrir();
    }

    @Test (expected = IllegalStateException.class)
    public void testBiestable2(){
        Semaforo s = new Biestable();
        s.cerrar();
    }

    @Test
    public void testBiestable3(){
        Semaforo s = new Biestable();
        s.abrir();
        assert s.estado().equals("abierto");
        s.cerrar();
        assert s.estado().equals("cerrado");
        s.abrir();
        assert s.estado().equals("abierto");
    }

    @Test (expected = IllegalStateException.class)
```

```
    public void testTriestable1(){
        Semaforo s = new Triestable();
        assert s.estado().equals("cerrado");
        s.cerrar();
    }

    @Test (expected = IllegalStateException.class)
    public void testTriestable2(){
        Semaforo s = new Triestable();
        assert s.estado().equals("cerrado");
        s.abrir();
        assert s.estado().equals("precaución");
        s.abrir();
        assert s.estado().equals("abierto");
        s.abrir();
    }

    @Test
    public void testTriestable3(){
        Semaforo s = new Triestable();
        assert s.estado().equals("cerrado");
        s.abrir();
        assert s.estado().equals("precaución");
        s.abrir();
        assert s.estado().equals("abierto");
        s.cerrar();
        assert s.estado().equals("precaución");
        s.cerrar();
        assert s.estado().equals("cerrado");
    }




}
```

Ej2c.java

```
import org.junit.Test;
import Ej2c.*;

public class Ej2c {
    @Test
    public void test1(){
        MediadorBiestableTriestable mediador = new MediadorBiestableTriestable();
        assert mediador.estado().equals("cerrado");
        mediador.abrir();
        assert mediador.estado().equals("abierto");
        mediador.cerrar();
        assert mediador.estado().equals("cerrado");
        mediador.cambiar();
        assert mediador.estado().equals("cerrado");
        mediador.abrir();
        assert mediador.estado().equals("precaución");
        mediador.abrir();
        assert mediador.estado().equals("abierto");
        mediador.cambiar();
        assert mediador.estado().equals("abierto");
        mediador.cerrar();
        assert mediador.estado().equals("cerrado");
    }


    @Test (expected = IllegalStateException.class)
    public void test2(){
        MediadorBiestableTriestable mediador = new MediadorBiestableTriestable();
        assert mediador.estado().equals("cerrado");
        mediador.cerrar();
    }

    @Test (expected = IllegalStateException.class)
    public void test3(){
        MediadorBiestableTriestable mediador = new MediadorBiestableTriestable();
        assert mediador.estado().equals("cerrado");
```

```
        mediador.abrir();
        assert mediador.estado().equals("abierto");
        mediador.abrir();
    }

    @Test (expected = IllegalStateException.class)
    public void test4(){
        MediadorBiestableTriestable mediador = new MediadorBiestableTriestable();
        assert mediador.estado().equals("cerrado");
        mediador.abrir();
        assert mediador.estado().equals("abierto");
        mediador.cambiar();
        assert mediador.estado().equals("abierto");
        mediador.abrir();
    }

    @Test (expected = IllegalStateException.class)
    public void test5(){
        MediadorBiestableTriestable mediador = new MediadorBiestableTriestable();
        assert mediador.estado().equals("cerrado");
        mediador.abrir();
        assert mediador.estado().equals("abierto");
        mediador.cambiar();
        assert mediador.estado().equals("abierto");
        mediador.cerrar();
        assert mediador.estado().equals("precaución");
        mediador.cambiar();
    }

    @Test (expected = IllegalStateException.class)
    public void test6(){
        MediadorBiestableTriestable mediador = new MediadorBiestableTriestable();
        assert mediador.estado().equals("cerrado");
        mediador.abrir();
        assert mediador.estado().equals("abierto");
        mediador.cambiar();
        assert mediador.estado().equals("abierto");
        mediador.cerrar();
```

```
        assert mediador.estado().equals("precaución");
        mediador.cerrar();
        assert mediador.estado().equals("cerrado");
        mediador.cerrar();
    }




}
```

Ej3.java

```
import Ej3.*;
import org.junit.Before;
import org.junit.Test;

import java.util.Date;

public class Ej3 {
  Email e1;
  Email e2;
  Email e3;

  @Before
  public void init() {
    e1 = new Email("f1", "BBB", new Date(3), new Priority(10), "Hello");
    e2 = new Email("f2", "CCCC", new Date(2), new Priority(1), "World");
    e3 = new Email("f3", "AAA", new Date(1), new Priority(3), "Something");
  }

  @Test
  public void dateSort() {
    var mailbox = new Mailbox(new DateSortStrategy());
    mailbox.addMail(e1);
    mailbox.addMail(e2);
    mailbox.addMail(e3);
```

```
    mailbox.sort();
    var list = mailbox.getEmail();
    assert list.get(0).equals(e3);
    assert list.get(1).equals(e2);
    assert list.get(2).equals(e1);
  }

  @Test
  public void fromSort() {
    var mailbox = new Mailbox(new FromSortStrategy());
    mailbox.addMail(e1);
    mailbox.addMail(e2);
    mailbox.addMail(e3);
    mailbox.sort();
    var list = mailbox.getEmail();
    assert list.get(0).equals(e1);
    assert list.get(1).equals(e2);
    assert list.get(2).equals(e3);
  }

  @Test
  public void subjectSort() {
    var mailbox = new Mailbox(new SubjectSortStrategy());
    mailbox.addMail(e1);
    mailbox.addMail(e2);
    mailbox.addMail(e3);
    mailbox.sort();
    var list = mailbox.getEmail();
    assert list.get(0).equals(e3);
    assert list.get(1).equals(e1);
    assert list.get(2).equals(e2);
  }

  @Test
  public void prioritySort() {
    var mailbox = new Mailbox(new PrioritySortStrategy());
    mailbox.addMail(e1);
    mailbox.addMail(e2);
```

```
    mailbox.addMail(e3);
    mailbox.sort();
    var list = mailbox.getEmail();
    assert list.get(0).equals(e2);
    assert list.get(1).equals(e3);
    assert list.get(2).equals(e1);
  }
}
```