**EJERCICIO 1**

En sí Java no proporciona un mecanismo de exportación selectiva similar al que existe en Eiffel, ya que el control de acceso se basa en modificadores de acceso, como ‘public’, ‘protected’ o ‘private’. Estos modificadores operan a nivel de clase, no a nivel de método específico.

El patrón que hemos llevado a cabo se basa en el patrón método fábrica, en el que los tres tipos de clientes A, B y C, heredan de una interfaz Cliente que define todo tipo de cliente. Por otro lado, están las clases FabricaX, FabricaXA, FabricaXB y FabricaXC, en la que FabricaX es la clase base abstracta que contiene los métodos en común que todas las clases van a poder acceder, a partir de esa, surgen las otras fábricas que extienden a esa inicial, aportando en cada caso los métodos específicos para cada cliente. Por lo que la clase ClienteA va a poder acceder a todos los métodos públicos de FabricaX y a todos los métodos de FabricaXA, y así con cada tipo de cliente.

La ventaja de la solución propuesta es que te aseguras de que todo cliente puede acceder únicamente a los métodos que tiene permiso, ya que la creación de una clase por fábrica y por cliente limita ese aspecto con total seguridad. Sin embargo, tiene un inconveniente, y es que se crean demasiadas clases con poca información a nivel de código, meramente organizativas.

**EJERCICIO 2**

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

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

Usamos el patrón de diseño estado, que nos ayuda a localizar el comportamiento de los métodos implementados dependiendo del estado en el que se encuentre nuestro objeto de la clase “Dispositivo”.

**Diagrama:**Gráfico

Descripción generada automáticamente

**Código:**

Clase Dispositivo:

public class Dispositivo {  
 private Estado estado;  
 public Dispositivo() {  
 this.estado = new EstadoRojo();  
 }  
 private void setEstado(Estado estado) {  
 this.estado = estado;  
 }  
 private void abrir() {  
 setEstado(this.estado.abrir());  
 }  
 private void cerrar() {  
 setEstado(this.estado.cerrar());  
 }  
 private String estado() {  
 return this.estado.estado();  
 }  
}

Interfaz Estado:

public interface Estado {  
 public Estado abrir();  
 public Estado cerrar();  
 public String estado();  
}

Clase EstadoRojo:

public class EstadoRojo implements Estado{  
 @Override  
 public Estado abrir() {return new EstadoVerde();}  
 @Override  
 public Estado cerrar() {throw new RuntimeException("Cambio de estado invalido.");}  
 @Override  
 public String estado() {  
 return "Rojo";  
 }  
}

Clase EstadoVerde:

public class EstadoVerde implements Estado{  
 @Override  
 public Estado abrir() {  
 throw new RuntimeException("Cambio de estado invalido.");  
 }  
 @Override  
 public Estado cerrar() {  
 return new EstadoRojo();  
 }  
 @Override  
 public String estado() {  
 return "Verde";  
 }  
}

Como vemos, se inicializa el estado del Dispositivo a rojo, y el comportamiento de los métodos abrir y cerrar varía dependiendo del estado en el que nos encontremos. Por ejemplo, cuando el dispositivo está en el estado “rojo”, al abrir cambia al estado “verde”, pero al cerrar no sucede nada. También existe el método estado(), que nos devuelve una cadena con el estado actual del dispositivo.

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

Para reutilizar el máximo código posible, añadimos un atributo String “tipo”, el cual puede ser “Triestable” o “Biestable”, y dependiendo del valor de este atributo, el comportamiento de los estados variará. También hemos añadido un getter de tipo y el nuevo estado, “EstadoAmarillo”.

Por otra parte, hemos añadido como parámetro a los métodos abrir() y cerrar() un objeto de la clase Dispositivo, para poder comprobar su tipo.

**Diagrama:**

Diagrama

Descripción generada automáticamente

**Código:**

Clase Dispositivo:

public class Dispositivo {  
 private Estado estado;  
 private String tipo;  
 public Dispositivo(String tipo) {  
 assert (tipo.equals("Biestable") || tipo.equals("Triestable"));  
 this.tipo = tipo;  
 this.estado = new EstadoRojo();  
 }  
 public void setEstado(Estado estado) {  
 this.estado = estado;  
 }  
 private void abrir() {  
 this.estado.abrir(this);  
 }  
 private void cerrar() {  
 this.estado.cerrar(this);  
 }  
 private String estado() {  
 return this.estado.estado();  
 }  
 protected String getTipo() {  
 return this.tipo;  
 }  
}

Interfaz Estado:

public interface Estado {  
 public Estado abrir(Dispositivo dispositivo);  
 public Estado cerrar(Dispositivo dispositivo);  
 public String estado();  
}

Clase EstadoRojo:

public class EstadoRojo implements Estado {  
 @Override  
 public Estado abrir(Dispositivo dispositivo) {  
 switch (dispositivo.getTipo()) {  
 case "Biestable" -> {return new EstadoVerde();}  
 case "Triestable" -> {return new EstadoAmarillo();}  
 default -> {throw new RuntimeException("Tipo de dispositivo no reconocido");}  
 }  
 }  
 @Override  
 public Estado cerrar(Dispositivo dispositivo) {throw new RuntimeException("Cambio de estado invalido.");}  
 @Override  
 public String estado() {  
 return "Rojo";  
 }  
}

Clase EstadoVerde:

public class EstadoVerde implements Estado {  
 @Override  
 public Estado abrir(Dispositivo dispositivo) {throw new RuntimeException("Cambio de estado invalido.");}  
 @Override  
 public Estado cerrar(Dispositivo dispositivo) {  
 switch (dispositivo.getTipo()) {  
 case "Biestable" -> {return new EstadoRojo();}  
 case "Triestable" -> {return new EstadoAmarillo();}  
 default -> {throw new RuntimeException("Tipo de dispositivo no reconocido");}  
 }  
 }  
 @Override  
 public String estado() {  
 return "Verde";  
 }  
}

Clase EstadoAmarillo:

public class EstadoAmarillo implements Estado {  
 @Override  
 public Estado abrir(Dispositivo dispositivo) {  
 assert (dispositivo.getTipo().equals("Triestable"));  
 return new EstadoVerde();  
 }  
 @Override  
 public Estado cerrar(Dispositivo dispositivo) {  
 assert (dispositivo.getTipo().equals("Triestable"));  
 return new EstadoRojo();  
 }  
 @Override  
 public String estado() {  
 return "Amarillo";  
 }  
}

En las clases EstadoVerde y EstadoRojo, los métodos abrir y cerrar actúan de forma distinta dependiendo del tipo del dispositivo pasado por referencia. Es decir, en este caso estamos combinando el patrón de diseño Estado con una estrategia de comportamiento condicional.

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

Para este caso, podemos añadir un método cambio en la clase Dispositivo, que cambie su tipo de “Biestable” a “Triestable” o viceversa. Para el caso de “Triestable” a “Biestable” debemos controlar también que el dispositivo no se encuentre en estado Amarillo, ya que el cambio no sería posible. Para ello, también debemos añadir un método privado “setTipo(String tipo)”.

**Diagrama:**

Diagrama, Gráfico de cajas y bigotes

Descripción generada automáticamente

**Código:**

Clase Dispositivo:

public class Dispositivo {  
 private Estado estado;  
 private String tipo;  
 public Dispositivo(String tipo) {  
 assert (tipo.equals("Biestable") || tipo.equals("Triestable"));  
 this.tipo = tipo;  
 this.estado = new EstadoRojo();  
 }  
 private void setEstado(Estado estado) {  
 this.estado = estado;  
 }  
 private void abrir() {  
 setEstado(this.estado.abrir(this));  
 }  
 private void cerrar() {  
 setEstado(this.estado.cerrar(this));  
 }  
 private String estado() {  
 return this.estado.estado();  
 }  
 protected String getTipo() {  
 return this.tipo;  
 }  
 private void setTipo(String tipo) {  
 this.tipo = tipo;  
 }  
 private void cambio(){  
 switch (this.tipo) {  
 case "Biestable" -> {setTipo("Triestable");}  
 case "Triestable" -> {  
 assert (!this.estado.estado().equals("Amarillo"));  
 setTipo("Biestable");  
 }  
 default -> {  
 throw new RuntimeException("Tipo de dispositivo no reconocido");  
 }  
 }  
 }  
}

Interfaz Estado:

public interface Estado {  
 public Estado abrir(Dispositivo dispositivo);  
 public Estado cerrar(Dispositivo dispositivo);  
 public String estado();  
}

Clase EstadoRojo:

public class EstadoRojo implements Estado {  
 @Override  
 public Estado abrir(Dispositivo dispositivo) {  
 switch (dispositivo.getTipo()) {  
 case "Biestable" -> {return new EstadoVerde();}  
 case "Triestable" -> {return new EstadoAmarillo();}  
 default -> {throw new RuntimeException("Tipo de dispositivo no reconocido");}  
 }  
 }  
 @Override  
 public Estado cerrar(Dispositivo dispositivo) {throw new RuntimeException("Cambio de estado invalido.");}  
 @Override  
 public String estado() {  
 return "Rojo";  
 }  
}

Clase EstadoVerde:

public class EstadoVerde implements Estado {  
 @Override  
 public Estado abrir(Dispositivo dispositivo) {throw new RuntimeException("Cambio de estado invalido.");}  
 @Override  
 public Estado cerrar(Dispositivo dispositivo) {  
 switch (dispositivo.getTipo()) {  
 case "Biestable" -> {return new EstadoRojo();}  
 case "Triestable" -> {return new EstadoAmarillo();}  
 default -> {throw new RuntimeException("Tipo de dispositivo no reconocido");}  
 }  
 }  
 @Override  
 public String estado() {  
 return "Verde";  
 }  
}

Clase EstadoAmarillo:

public class EstadoAmarillo implements Estado {  
 @Override  
 public Estado abrir(Dispositivo dispositivo) {  
 assert (dispositivo.getTipo().equals("Triestable"));  
 return new EstadoVerde();  
 }  
 @Override  
 public Estado cerrar(Dispositivo dispositivo) {  
 assert (dispositivo.getTipo().equals("Triestable"));  
 return new EstadoRojo();  
 }  
 @Override  
 public String estado() {  
 return "Amarillo";  
 }  
}

Como vemos, solo hemos necesitado cambiar un poco la clase dispositivo, por tanto, hemos conseguido reutilizar la mayoría del código. Por otra parte, hemos seguido la misma estrategia que en el caso anterior, hemos mantenido el patrón de diseño estado y hemos usado una estrategia de comportamiento condicional para el nuevo método “cambio()”.

**EJERCICIO 3**

El patrón de diseño aplicable a este ejercicio es el patrón de estrategia, gracias al cual podemos crear distintas clases SortXXX dependiendo del atributo del mensaje que queremos tener en cuenta para realizar la ordenación. Para no repetir código, el esquema utiliza una función sort() en Mailbox que implementa el algoritmo, sólo refiriéndose a la estrategia el método before(). Gracias a este patrón podemos crear las estrategias que queramos, incluso en casos donde Email se extienda con más atributos. Simplemente haría falta crear un nuevo criterio de comparación en una nueva clase SortXXX.

A screenshot of a computer

Description automatically generated

Clases:

Mailbox:

import java.util.ArrayList;

import static java.util.Collections.swap;

public class Mailbox {

private ArrayList<Email> email = new ArrayList<>();

private SortStrategy sortStrategy;

void setSortStrategy(SortStrategy sortStrategy) {

     this.sortStrategy = sortStrategy;

}

public void show(){

     sort();

     for (Email email : email) {

         System.out.println(email);

     }

}

private void sort() {

     for (int i = 2; i < email.size(); i++) {

         for (int j = email.size()-1; j >= i; j-- ){

             if (before(email.get(j-1),email.get(j)) ){

                 swap(email,j,j-1);

             }

         }

     }

}

private boolean before(Email email, Email email1) {

     return sortStrategy.sort(email, email1);

}

public void addEmail(Email email) {

     this.email.add(email);

}

}

Email:

import java.text.SimpleDateFormat;

import java.util.Date;

public class Email {

private String from;

private String subject;

private String date;

private boolean priority;

private String text;

public Email(String from, String subject, String date, boolean priority, String text) {

     this.from = from;

     this.subject = subject;

     this.date = date;

     this.priority = priority;

     this.text = text;

}

public long getDate() {

     String dateString = this.date;

     SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");

     try{

         Date date = dateFormat.parse(dateString);

         return date.getTime();

     } catch (Exception e) {

         return 0;

     }

}

public String getFrom() {return from;}

public String getSubject() {

     return subject;

}

public boolean getPriority() {

     return priority;

}

public String getText() {

     return text;

}

@Override

public String toString() {

     return "From: " + from + "\n" +

             "Subject: " + subject + "\n" +

             "Date: " + date + "\n" +

             "Priority: " + priority + "\n" +

             "Text: " + text + "\n";

}

}

Interfaz SortStrategy:

public interface SortStrategy {

public boolean sort(Email email1, Email email2);

}

SortDate:

public class SortDate implements SortStrategy  {

@Override

public boolean sort(Email email1, Email email2) {

     return email1.getDate()-email2.getDate() > 0;

}

}

SortFrom:

public class SortFrom implements SortStrategy{

@Override

public boolean sort(Email email1, Email email2) {

     String from1 = email1.getFrom();

     String from2 = email2.getFrom();

     int comparisonResult = from1.compareTo(from2); //should be earlier (a.b) < 0, should be after (b.a) > 0 (so should be swapped)

     return comparisonResult > 0;

}

}

SortPriority:

public class SortPriority implements SortStrategy {

@Override

public boolean sort(Email email1, Email email2) {

     return email1.getPriority();

}

}

SortSubject:

public class SortSubject implements  SortStrategy{

@Override

public boolean sort(Email email1, Email email2) {

     String subject1 = email1.getSubject();

     String subject2 = email2.getSubject();

     int comparisonResult = subject1.compareTo(subject2); //should be earlier (a.b) < 0, should be after (b.a) > 0 (so should be swapped)

     return comparisonResult > 0;

}

}