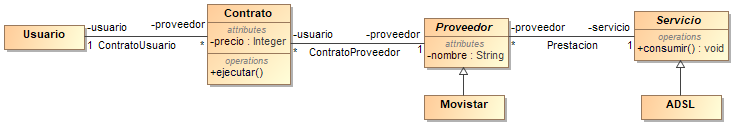
Ejercicio 1

El principal problema de este ejercicio es que desde la clase Contrato no podemos acceder a la clase Servicio ya que esta es abstracta. Sin embargo, por la relación entre Proveedor y Servicio, cada proveedor tendrá asociado únicamente un Servicio y, al ser bidireccional, tendremos un atributo servicio:Servicio dentro de la clase Proveedor, pudiendo utilizar un getter desde contrato.

La clase contrato se haría igual que en la práctica anterior la clase Acceso, transformándola en una clase entre Usuario y Proveedor y manteniendo la consistencia de la relación desde Contrato.



Siguiendo este diagrama, las clases quedarían como sigue:

* Usuario

public class Usuario {

private List<Contrato> contratos;

public Usuario() {

this.contratos = new ArrayList<>();

}

public void addContrato(Contrato contrato) {

this.contratos.add(contrato);

}

public void removeContrato(Contrato contrato) {

this.contratos.remove(contrato);

}

}

* Contrato

public class Contrato {

private Usuario usuario;

private Proveedor proveedor;

private int precio;

public Contrato(Usuario usuario, Proveedor proveedor, int precio {

this.usuario = usuario;

this.proveedor = proveedor;

this.precio = precio;

usuario.addContrato(this);

proveedor.addContrato(this);

}

public void remove(){

this.usuario.removeContrato(this);

this.proveedor.removeContrato(this);

}

public Servicio ejecutar(){

return this.proveedor.getServicio();

}

}

* *Proveedor*

public abstract class Proveedor {

private List<Contrato> contratos;

private String nombre;

private Servicio servicio;

public Proveedor(String nombre, Servicio servicio) {

this.nombre = nombre;

this.contratos = new ArrayList<>();

this.servicio = servicio;

servicio.addProveedor(this);

}

public Servicio getServicio() {

return servicio;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

public void addContrato(Contrato contrato) {

this.contratos.add(contrato);

}

public void removeContrato(Contrato contrato) {

this.contratos.remove(contrato);

}

}

* Movistar < *Proveedor*

public class Movistar extends Proveedor {

public Movistar(String nombre, Servicio servicio) {

super(nombre, servicio);

}

}

* *Servicio*

public abstract class Servicio {

private List<Proveedor> proveedores;

public Servicio() {

this.proveedores = new ArrayList<>();

}

public void addProveedor(Proveedor proveedor) {

this.proveedores.add(proveedor);

}

public void consumir(){

//El sysout está para que el método no esté vacío y poder probarlo

System.*out*.println("Método consumido");

}

}

* ADSL < *Servicio*

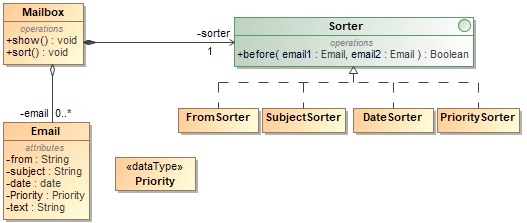
public class ADSL extends Servicio {

//No implementa nada según el enunciado, simplemente está.

}

Ejercicio 2

En este caso al querer tener la posibilidad de ordenar los correos mostrados en la clase MailBox mediante distintos métodos pero sólo uno a la vez. Así que podríamos usar el patrón de diseño Strategy, implementando una estrategia distinta del método *before* para cada método de ordenación, suponemos que no queremos ordenar los emails por su texto, es decir, su contenido.



Siguiendo el diagrama, tendríamos las siguientes clases:

* Priority

public class Priority implements Comparable<Priority>{

private int priority;

public Priority(int priority) {

this.priority = priority;

}

public int getPriority() {

return priority;

}

public void setPriority(int priority) {

this.priority = priority;

}

@Override

public int compareTo(Priority priority) {

return this.priority-priority.priority;

}

}

* *Sorter*

public interface Sorter {

boolean before(Email email1, Email email2);

}

* FromSorter < *Sorter*

public class FromSorter implements Sorter {

@Override

public boolean before(Email email1, Email email2) {

return email1.getFrom().compareToIgnoreCase(email2.getFrom()) < 0;

}

}

* SubjectSorter < *Sorter*

public class SubjectSorter implements Sorter {

@Override

public boolean before(Email email1, Email email2) {

return email1.getSubject().compareToIgnoreCase(email2.getSubject()) < 0;

}

}

* DateSorter < *Sorter*

public class DateSorter implements Sorter {

@Override

public boolean before(Email email1, Email email2) {

return email1.getDate().before(email1.getDate());

}

* PrioritySorter < *Sorter*

public class PrioritySorter implements Sorter {

@Override

public boolean before(Email email1, Email email2) {

return email1.getPriority().compareTo(email2.getPriority()) < 0;

}

}

* Mailbox

public class Mailbox {

private List<Email> emails;

private Sorter sorter;

public Mailbox(Sorter sorter) {

this.sorter = sorter;

this.emails = new ArrayList<>();

}

public void setSorter(Sorter sorter) {

this.sorter = sorter;

}

public void show() {

sort();

for (Email email : emails) {

System.*out*.println(email);

}

}

public void sort() {

for (int i = 0; i < emails.size(); i++) {

for (int j = 0; j < (emails.size() - i - 1); j++) {

if (sorter.before(emails.get(j), emails.get(j - 1))) {

Email em = emails.get(j);

emails.set(j, emails.get(j - 1));

emails.set(j, em);

}

}

}

}

}

* Email

public class Email {

private String from;

private String subject;

private Date date;

private Priority priority;

public Email(String from, String subject, Date date, Priority 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 Date getDate() {

return date;

}

public void setDate(Date date) {

this.date = date;

}

public Priority getPriority() {

return priority;

}

public void setPriority(Priority priority) {

this.priority = priority;

}

@Override

public String toString() {

return "Email{" +

"from='" + from + '\'' +

", subject='" + subject + '\'' +

", date=" + date +

", priority=" + priority +

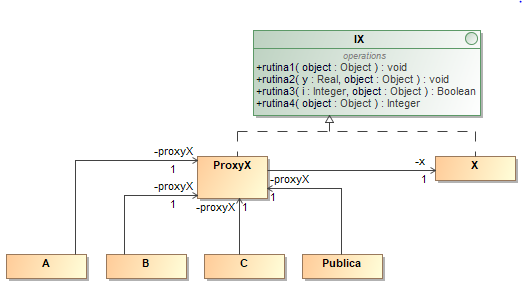
'}';

}

}

Ejercicio 3

1. En Java no es posible realizar una exportación selectiva como tal. Sin embargo, se puede usar el patrón Representante (o Proxy en inglés), el cual define una clase intermedia que se encarga de administrar qué clases hacen uso de determinadas funciones de otra clase compartiendo su interfaz para así definir claramente sus operaciones. Eiffel, por el contrario, permite directamente administrar qué funciones son utilizadas por cada clase y no haría falta insertar patrón alguno.
2. Tal y como se ha indicado en el apartado anterior, se puede utilizar el patrón Proxy para realizar una implementación de la exportación selectiva en Java. La solución aquí planteada tiene las mismas ventajas que la exportación selectiva de Eiffel, sin embargo, sus desventajas son claras. Por cada nueva clase u operación a realizar, se debe especificar dentro de la clase Proxy y la seguridad para verificar qué clase hace qué función es bastante baja. Así pues, la solución sería la siguiente.



Teniendo en la clase X la implementación de la funcionalidad de los métodos y en la clase ProxyX el filtro que permite a las clases A, B, C y Pública acceder a los métodos, quedando las clases como sigue, estando los métodos vacíos ya que no se indica su funcionalidad en el enunciado:

* *IX*

public interface IX {

void rutina1(Object object);

void rutina2(float y, Object object);

boolean rutina3(int i, Object object);

int rutina4(Object object);

}

* A, B, C, Pública

public class A/B/C/Publica {

private ProxyX proxyX;

}

* X

public class X implements IX {

@Override

public void rutina1(Object object) {

}

@Override

public void rutina2(float y, Object object) {

}

@Override

public boolean rutina3(int i, Object object) {

return false;

}

@Override

public int rutina4(Object object) {

return 0;

}

}

* ProxyX

public class ProxyX implements IX {

private X x;

@Override

public void rutina1(Object object) {

x.rutina1(object);

}

@Override

public void rutina2(float y, Object object) {

if (object instanceof A || object instanceof B) {

x.rutina2(y, object);

} else {

throw new RuntimeException("Este tipo de objetos no puede acceder a esta operacion");

}

}

@Override

public boolean rutina3(int i, Object object) {

if (object instanceof A || object instanceof C) {

return x.rutina3(i, object);

} else {

throw new RuntimeException("Este tipo de objetos no puede acceder a esta operacion");

}

}

@Override

public int rutina4(Object object) {

if (object instanceof X) {

return x.rutina4(object);

} else {

throw new RuntimeException("Este tipo de objetos no puede acceder a esta operacion");

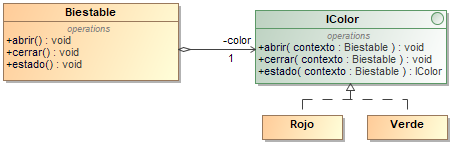
}

}

}

Ejercicio 4

1. Para este ejercicio tenemos un biestable con dos posibles estados, por lo que podemos introducir el Patrón Estado para la implementación, de manera que el biestable tenga un atributo de tipo *IColor* que se encargue de hacer los pertinentes cambios de estado de Rojo a Verde y viceversa, siguiendo el siguiente diagrama.



* Biestable

public class Biestable {

private IColor color;

public Biestable() {

this.color = new Rojo();

}

public void abrir() {

color.abrir(this);

}

public void cerrar() {

color.cerrar(this);

}

public String estado() {

return color.estado(this);

}

public IColor getColor() {

return color;

}

public void setColor(IColor color) {

this.color = color;

}

}

* *IColor*

public interface IColor {

void abrir(Biestable contexto);

void cerrar(Biestable contexto);

String estado(Biestable contexto);

}

* Rojo < *IColor*

public class Rojo implements IColor {

@Override

public void abrir(Biestable contexto) {

contexto.setColor(new Verde());

}

@Override

public void cerrar(Biestable contexto) {

throw new RuntimeException("No puedes cerrar más este dispositivo");

}

@Override

public String estado(Biestable contexto) {

return "Cerrado";

}

}

* Verde < *IColor*

public class Verde implements IColor {

@Override

public void abrir(Biestable contexto) {

throw new RuntimeException("No puedes abrir mas este dispositivo");

}

@Override

public void cerrar(Biestable contexto) {

contexto.setColor(new Rojo());

}

@Override

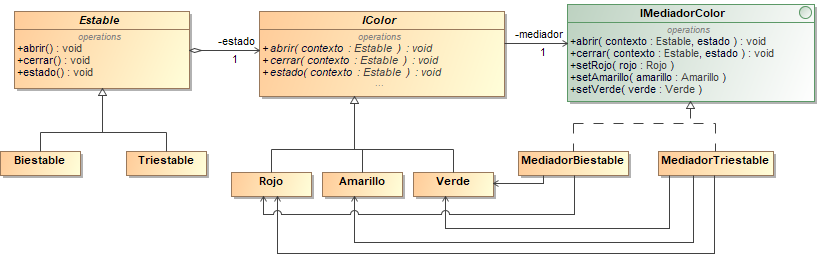
public String estado(Biestable contexto) {

return "Abierto";

}

}

1. Para este apartado reutilizaremos el anterior realizando algunos cambios para añadir el patrón Mediador al cambio de estado, de manera que tengamos un mediador que se encargue de los cambios del Biestable y otro que se encargue de los cambios del Triestable.



La clase abstracta *Estable* contiene la definición de los métodos abrir, cerrar y estado exactamente igual que la clase Biestable antes y ahora Biestable y Triestable heredan de esta clase. Se ha sustituido la interfaz *IColor* por una clase abstracta, para poder introducir el atributo de tipo *IMediadorColor* que afecta solo a los estados Rojo y Verde*.* Quedando la implementación como sigue.

* Biestable

public class Biestable extends Estable {

public Biestable() {

this.estado = new Rojo(new MediadorBiestable());

}

}

* Triestable

public class Triestable extends Estable {

public Triestable(){

this.estado = new Rojo(new MediadorTriestable());

}

}

* Rojo

public class Rojo extends IColor {

public Rojo(IMediadorColor mediador) {

this.mediador = mediador;

this.mediador.setRojo(this);

}

@Override

public void abrir(Estable contexto) {

mediador.abrir(contexto, this);

}

@Override

public void cerrar(Estable contexto) {

mediador.cerrar(contexto, this);

}

@Override

public String estado(Estable contexto) {

return "CERRADO";

}

}

* Amarillo

public class Amarillo extends IColor {

public Amarillo(IMediadorColor mediador){

this.mediador = mediador;

this.mediador.setAmarillo(this);

}

@Override

public void abrir(Estable contexto) {

mediador.abrir(contexto, this);

}

@Override

public void cerrar(Estable contexto) {

mediador.cerrar(contexto, this);

}

@Override

public String estado(Estable contexto) {

return "PRECAUCION";

}

}

* Verde

public class Verde extends IColor {

public Verde(IMediadorColor mediador){

this.mediador = mediador;

this.mediador.setVerde(this);

}

@Override

public void abrir(Estable contexto) {

mediador.abrir(contexto, this);

}

@Override

public void cerrar(Estable contexto) {

mediador.cerrar(contexto, this);

}

@Override

public String estado(Estable contexto) {

return "ABIERTO";

}

}

* *IMediadorColor*

public interface IMediadorColor {

void abrir(Estable contexto, IColor estado);

void cerrar(Estable contexto, IColor estado);

void setRojo(Rojo rojo);

void setAmarillo(Amarillo amarillo);

void setVerde(Verde verde);

}

En las siguientes clases se hacen llamadas desde un setter a los otros para poder mantener el patrón Estado.

* MediadorBiestable

public class MediadorBiestable implements IMediadorColor {

private Rojo rojo;

private Verde verde;

@Override

public void abrir(Estable contexto, IColor estado) {

if(estado.equals(rojo)){

contexto.setEstado(verde);

}

}

@Override

public void cerrar(Estable contexto, IColor estado) {

if(estado.equals(verde)){

contexto.setEstado(rojo);

}

}

@Override

public void setRojo(Rojo rojo) {

this.rojo = rojo;

if(verde == null){

new Verde(this);

}

}

@Override

public void setAmarillo(Amarillo amarillo) {

throw new RuntimeException("No es posible asignar un amarillo en este objeto");

}

@Override

public void setVerde(Verde verde) {

this.verde = verde;

if (rojo == null){

new Rojo(this);

}

}

}

* MediadorTriestable

public class MediadorTriestable implements IMediadorColor {

private Rojo rojo;

private Amarillo amarillo;

private Verde verde;

@Override

public void abrir(Estable contexto, IColor estado) {

if(estado.equals(rojo)){

contexto.setEstado(amarillo);

}

if(estado.equals(amarillo)){

contexto.setEstado(verde);

}

}

@Override

public void cerrar(Estable contexto, IColor estado) {

if(estado.equals(amarillo)){

contexto.setEstado(rojo);

}

if(estado.equals(verde)){

contexto.setEstado(amarillo);

}

}

@Override

public void setRojo(Rojo rojo) {

this.rojo = rojo;

if(amarillo == null){

new Amarillo(this);

}

if(verde == null){

new Verde(this);

}

}

@Override

public void setAmarillo(Amarillo amarillo) {

this.amarillo = amarillo;

if(rojo == null){

new Rojo(this);

}

if(verde == null){

new Verde(this);

}

}

@Override

public void setVerde(Verde verde) {

this.verde = verde;

if(amarillo == null){

new Amarillo(this);

}

if(rojo == null){

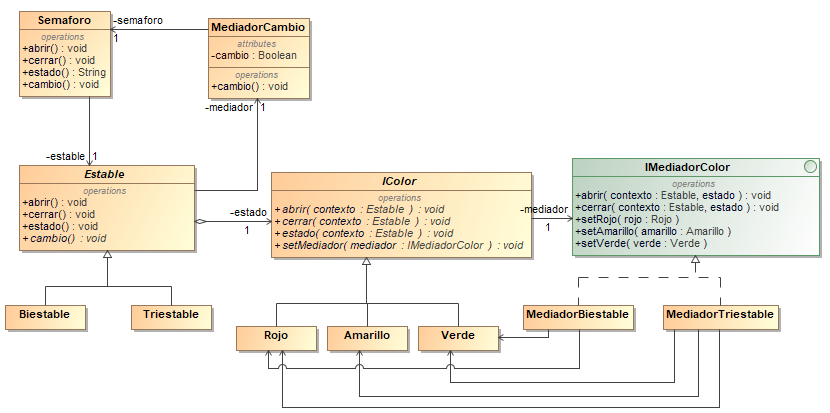
new Rojo(this);

}

}

}

1. Podemos introducir una clase Semaforo que contenga un atributo de tipo *Estable* para así, usando el patrón Mediador de nuevo, realizar el cambio del tipo del atributo. Para esto necesitamos un método setMediador en la clase *IColor* que reciba como parámetro un mediador para poder mantener el estado en el que nos encontramos. Quedando el diagrama como sigue. Siempre que se cree un nuevo Semaforo será un Biestable



Las clases que no están completas están así porque solo se ha añadido el código del documento y las que no aparecen no han sufrido ningún cambio.

* Biestable

public class Biestable extends Estable {

public Biestable(Semaforo semaforo) {

this.estado = new Rojo(new MediadorBiestable());

this.mediador = new MediadorCambio(false, semaforo);

}

@Override

public void cambio() {

mediador.cambio(this);

}

}

* Triestable

public class Triestable extends Estable {

public Triestable(Semaforo semaforo){

this.estado = new Rojo(new MediadorTriestable());

this.mediador = new MediadorCambio(true, semaforo);

}

@Override

public void cambio() {

mediador.cambio(this);

}

}

* Semaforo

public class Semaforo {

private Estable estable;

public Semaforo() {

this.estable = new Biestable(this);

}

public void abrir() {

estable.abrir();

}

public void cerrar() {

estable.cerrar();

}

public String estado(){

return estable.estado();

}

public void cambio() {

estable.cambio();

}

public void setEstable(Estable estable) {

this.estable = estable;

}

}

* MediadorCambio

public class MediadorCambio {

private boolean cambio;

private Semaforo semaforo;

public MediadorCambio(boolean cambio, Semaforo semaforo) {

this.cambio = cambio;

this.semaforo = semaforo;

}

public void cambio(Estable estable) {

if(!cambio){

Estable nuevoEstable = new Triestable(semaforo);

IColor estadoActual = estable.getEstado();

estadoActual.setMediador(new MediadorTriestable());

nuevoEstable.setEstado(estadoActual);

semaforo.setEstable(nuevoEstable);

}

}

}

* Rojo

public void setMediador(IMediadorColor mediador) {

this.mediador = mediador;

mediador.setRojo(this);

}

* Amarillo

public void setMediador(IMediadorColor mediador) {

this.mediador = mediador;

mediador.setAmarillo(this);

}

* Verde

public void setMediador(IMediadorColor mediador) {

this.mediador = mediador;

mediador.setVerde(this);

}