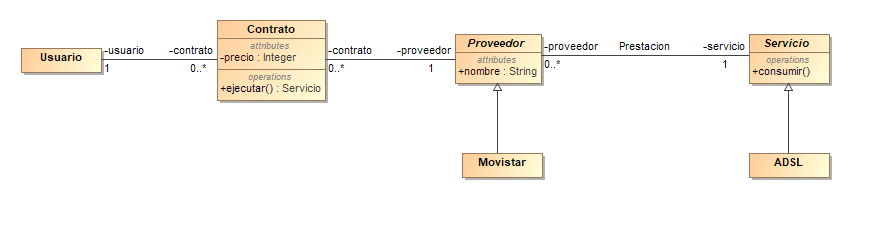
1. **LA ECONOMÍA DE SERVICIOS**



El problema surge al comprobar que, cuando llamamos al método *ejecutar()*, este no sabe que Servicio se ofrece en el contrato (puesto que no se ha inicializado). La solución propuesta se basa en crear un método *getServicio()* en la clase Proveedor que podremos llamar desde Contrato (al tener que pertenecer un Contrato a un determinado Proveedor).

Además, la clase de asociación la implantaremos igual que en la práctica anterior.

* USUARIO

public class Usuario {  
 public List<Contrato> contratos;  
  
 public Usuario(){  
 this.contratos = new ArrayList<>();  
 }  
  
 public void addContrato(Contrato c){  
 this.contratos.add(c);  
 }  
  
 public void removeContrato(Contrato c){  
 this.contratos.remove(c);  
 }  
}

* CONTRATO

public class Contrato {  
 public Usuario usuario;  
 public Proveedor proveedor;  
 public int precio;  
  
 public Contrato(Usuario usuario, Proveedor proveedor, int precio){  
 this.usuario = usuario;  
 this.proveedor = proveedor;  
 this.precio = precio;  
 this.usuario.addContrato(this);  
 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 {  
 public List<Contrato> contratos;  
 public Servicio servicio;  
 public String nombre;  
  
 public Proveedor(Servicio servicio, String nombre){  
 contratos = new ArrayList<>();  
 this.servicio = servicio;  
 this.nombre = nombre;  
 }  
  
 public void addContrato(Contrato c){  
 this.contratos.add(c);  
 }  
  
 public void removeContrato(Contrato c){  
 this.contratos.add(c);  
 }  
  
 public Servicio getServicio(){  
 return servicio;  
 }  
}

* MOVISTAR

public class Movistar extends Proveedor{  
 public Movistar(Servicio servicio, String nombre) {  
 super(servicio, nombre);  
 }  
}

* SERVICIO

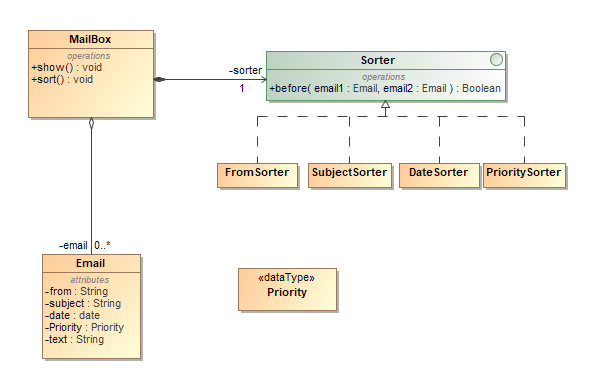
public abstract class Servicio {  
 List<Proveedor> proveedores;  
  
 public Servicio(){  
 proveedores = new ArrayList<>();  
 }  
  
 public void addProveedor(Proveedor p){  
 proveedores.add(p);  
 }  
  
 public void consumir(){  
 System.*out*.println("Consumido");  
 }  
}

* ADSL

public class ADSL extends Servicio{  
}

1. **CLIENTE DE CORREO E-LOOK**

Se ha decidido aplicar el patrón de Estrategia, ya que para una misma acción (ordenar emails) tenemos disponibles distintas estrategias (por el emisor, concepto, fecha y prioridad). Lo que se hará entonces será crear una interfaz que contenga el método *before(email1, email2)*. Tendremos de esta manera 4 clases (correspondiente a cada método de ordenación) que implementan dicha interfaz. La clase MailBox tendrá la referencia a la interfaz.



* 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 + '}';  
 }  
}

* MAILBOX

public class MailBox {  
 private List<Email> emails;  
 private Sorter sorter;  
  
 public MailBox(Sorter sorter){  
 emails = new ArrayList<>();  
 this.sorter = sorter;  
 }  
  
 public void setSorter(Sorter sorter){  
 this.sorter = sorter;  
 }  
  
 public void show(){  
 sort();  
  
 for(Email email: emails){  
 System.*out*.println(email);  
 }  
 }  
  
 private void sort(){  
 for(int i = 0; i < emails.size() - 1; i++){  
 for(int j = 0; j < (emails.size() - i - 1); j++){  
 if (sorter.before(emails.get(j),emails.get(j + 1)) ){  
 Email e = emails.get(j + 1);  
 emails.set(j + 1, emails.get(j));  
 emails.set(j, e);  
 }  
 }  
 }  
 }  
}

* 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

public class FromSorter implements Sorter {  
 @Override  
 public boolean before(Email email1, Email email2) {  
 return email1.getFrom().compareToIgnoreCase(email2.getFrom()) > 0;  
 }  
}

* SUBJECTSORTER

public class SubjectSorter implements Sorter{  
 @Override  
 public boolean before(Email email1, Email email2) {  
 return email1.getSubject().compareToIgnoreCase(email2.getSubject()) > 0;  
 }  
}

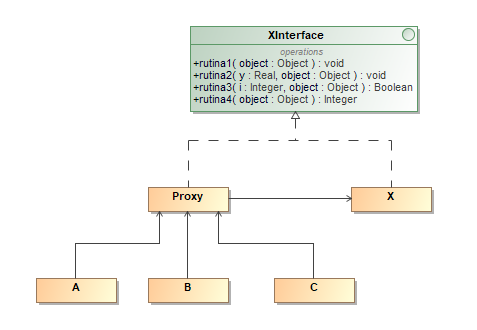
* DATESORTER

public class DateSorter implements Sorter{  
 @Override  
 public boolean before(Email email1, Email email2) {  
 return email1.getDate().after(email2.getDate());  
 }  
}

* PRIORITYSORTER

public class PriorityOrder implements Sorter{  
 @Override  
 public boolean before(Email email1, Email email2) {  
 return email1.getPriority().compareTo(email2.getPriority()) > 0;  
 }  
}

1. **LOS INTERFACES SELECTIVOS**
2. Java, como tal, no soporta la exportación selectiva. Sin embargo, esto no implica que no se pueda implementar en este lenguaje. Un ejemplo de implementación sería mediante el patrón Representante. En esté patrón se introduce una clase intermedia (entre la interfaz y las clases que hereden de ella) que administrará que funciones son usadas por cada clase. Por otro lado, Eiffel, al permitir administrar que funciones son utilizadas por cada clase, no precisaría de ningún patrón.
3. En el apartado anterior se ha indicado que para solucionar este problema emplearíamos el patrón Representante. La clara ventaja que presenta esta solución es el poder implementar la exportación selectiva en Java como se hace en Eiffel. Sin embargo, crear nuevas clases y nuevos métodos implicaría añadir nuevas reglas al proxy (siendo mucho menos instantáneo que con Eiffel). La solución propuesta es la siguiente:



* XINTERFACE

public interface XInterface {  
 public void rutina1(Object object);  
 public void rutina2(float y, Object object);  
 public boolean rutina3(int i, Object object);  
 public int rutina4(Object object);  
}

* X

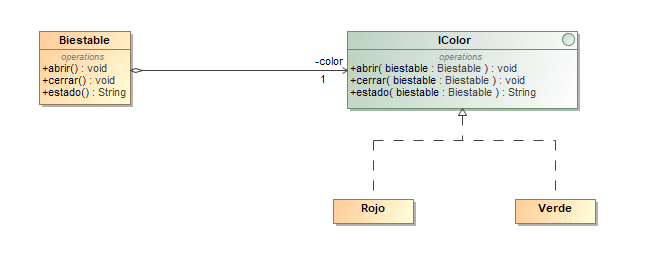
public class X implements XInterface{  
 @Override  
 public void rutina1(Object object) {  
  
 }  
  
 @Override  
 public void rutina2(float y, Object object) {  
  
 }  
  
 @Override  
 public boolean rutina3(int i, Object object) {  
 return false;  
 }  
  
 @Override  
 public int rutina4(Object object) {  
 return 0;  
 }  
}

* PROXY

public class Proxy implements XInterface{  
 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("Operacion restringida.");  
 }  
 }  
  
 @Override  
 public boolean rutina3(int i, Object object) {  
 if(object instanceof A || object instanceof C){  
 return x.rutina3(i, object);  
 }else{  
 throw new RuntimeException("Operacion restringida.");  
 }  
 }  
  
 @Override  
 public int rutina4(Object object) {  
 if(object instanceof X){  
 return x.rutina4(object);  
 }else{  
 throw new RuntimeException("Operacion restringida.");  
 }  
 }  
}

* A/B/C (Todas son iguales)

public class A {  
 private Proxy proxy;  
}

1. **TRIESTABLES**
2. Se nos presenta un biestable con dos posibles estados. Para implementarlo podemos usar el patrón de Estado de manera que el estado vaya cambiando de Rojo a Verde según las acciones realizadas. El diagrama sería el siguiente:

* BIESTABLE

public class Biestable {  
 private IColor color;  
  
 public Biestable(){  
 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 b);  
 void cerrar(Biestable b);  
 String estado(Biestable b);  
}

* ROJO

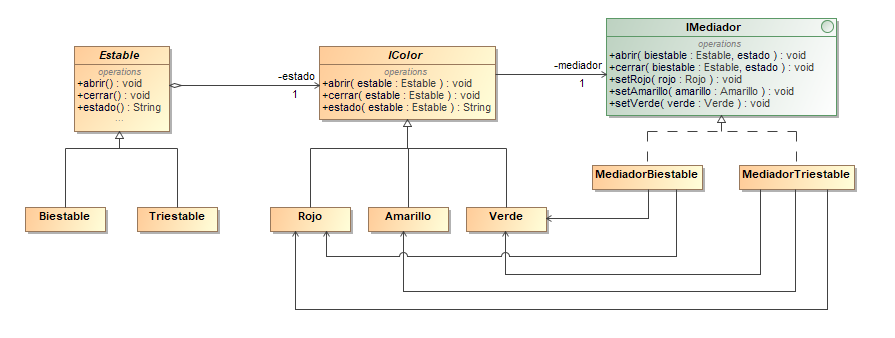
public class Rojo implements IColor {  
 @Override  
 public void abrir(Biestable b) {  
 b.setColor(new Verde());  
 }  
  
 @Override  
 public void cerrar(Biestable b) {  
 throw new RuntimeException("Objeto ya cerrado.");  
 }  
  
 @Override  
 public String estado(Biestable b) {  
 return "Cerrado";  
 }  
}

* VERDE

public class Verde implements IColor {  
 @Override  
 public void abrir(Biestable b) {  
 throw new RuntimeException("Objeto ya abierto.");  
 }  
  
 @Override  
 public void cerrar(Biestable b) {  
 b.setColor(new Rojo());  
 }  
  
 @Override  
 public String estado(Biestable b) {  
 return "Abierto";  
 }  
}

1. Para realizar este apartado reusaremos lo que se ha obtenido en el anterior realizando algunos cambios que nos permita añadir el patrón Mediador al cambio de estado. De esta manera tendremos dos mediadores, uno que se encargará del biestable y otro que se encargará del triestable.

Para ello, se ha creado una clase abstracta *Estable* de la que heredan *Biestable* y *Triestable*, se ha sustituido la interfaz *IColor* por una clase abstracta, se ha añadido la clase *Amarillo* (para el triestable) y una interfaz *IMediadorColor* que es implementada por los dos mediadores antes mencionados. El modelo y el codigo quedaría así:

* ESTABLE

public abstract class Estable {  
 public IColor estado;  
  
 public void abrir(){  
 estado.abrir(this);  
 }  
  
 public void cerrar(){  
 estado.cerrar(this);  
 }  
  
 public String estado(){  
 return estado.estado(this);  
 }  
  
 public IColor getEstado() {  
 return estado;  
 }  
  
 public void setEstado(IColor color) {  
 this.estado = color;  
 }  
}

* 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());  
 }  
}

* ICOLOR

public abstract class IColor {  
 IMediadorColor mediador;  
  
 public abstract void abrir(Estable b);  
 public abstract void cerrar(Estable b);  
 public String estado(Estable b) {  
 return null;  
 }  
}

* ROJO

public class Rojo extends IColor {  
 public Rojo(IMediadorColor mediador){  
 this.mediador = mediador;  
 this.mediador.setRojo(this);  
 }  
  
 @Override  
 public void abrir(Estable e) {  
 mediador.abrir(e, this);  
 }  
  
 @Override  
 public void cerrar(Estable e) {  
 mediador.cerrar(e, this);  
 }  
  
 @Override  
 public String estado(Estable e) {  
 return "Cerrado";  
 }  
}

* AMARILLO

public class Amarillo extends IColor {  
 public Amarillo(IMediadorColor mediador){  
 this.mediador = mediador;  
 this.mediador.setAmarillo(this);  
 }  
  
 @Override  
 public void abrir(Estable e) {  
 mediador.abrir(e, this);  
 }  
  
 @Override  
 public void cerrar(Estable e) {  
 mediador.cerrar(e, this);  
 }  
  
 @Override  
 public String estado(Estable e) {  
 return "Precaucion";  
 }  
}

* VERDE

public class Verde extends IColor {  
 public Verde(IMediadorColor mediador){  
 this.mediador = mediador;  
 this.mediador.setVerde(this);  
 }  
  
 @Override  
 public void abrir(Estable e) {  
 mediador.abrir(e, this);  
 }  
  
 @Override  
 public void cerrar(Estable e) {  
 mediador.cerrar(e, this);  
 }  
  
 @Override  
 public String estado(Estable e) {  
 return "Abierto";  
 }  
}

* IMEDIADORCOLOR

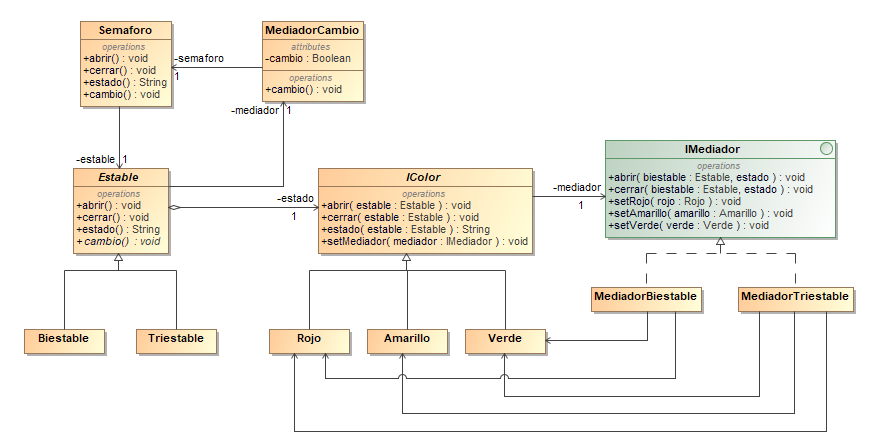
public interface IMediadorColor {  
 void abrir(Estable e, IColor estado);  
 void cerrar(Estable e, IColor estado);  
 void setRojo(Rojo rojo);  
 void setAmarillo(Amarillo amarillo);  
 void setVerde(Verde verde);  
}

* MEDIADORBIESTABLE

public class MediadorBiestable implements IMediadorColor{  
 private Rojo rojo;  
 private Verde verde;  
 @Override  
 public void abrir(Estable e, IColor estado) {  
 if(estado.equals(rojo)){  
 e.setEstado(verde);  
 }  
 }  
 @Override  
 public void cerrar(Estable e, IColor estado) {  
 if(estado.equals(verde)){  
 e.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 se puede asignar amarillo aqui.");  
 }  
 @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 e, IColor estado) {  
 if(estado.equals(rojo)){  
 e.setEstado(amarillo);  
 }  
 if(estado.equals(amarillo)){  
 e.setEstado(verde);  
 }  
 }  
 @Override  
 public void cerrar(Estable e, IColor estado) {  
 if(estado.equals(amarillo)){  
 e.setEstado(rojo);  
 }  
 if(estado.equals(verde)){  
 e.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. Aquí volveremos a usar el patrón Mediador de nuevo y nos ayudaremos de una clase *Semaforo* que contenga un atributo de tipo *Estable*. Añadiremos además un método *setMediador* en la clase abstracta *IColor* que (recibiendo como parámetro un mediador) mantiene el estado en el que se encuentra. El diagrama quedaría así:

A continuación, el código modificado (Biestable, Triestable, Semaforo, MediadorCambio y los colores -de estos solo el método añadido que se ha comentado antes-):

* 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 estableNuevo = new Triestable(semaforo);  
 IColor estadoActual = estable.getEstado();  
 estadoActual.setMediador(new MediadorTriestable());  
 estableNuevo.setEstado(estadoActual);  
 semaforo.setEstable(estableNuevo);  
 }  
 }  
}

* 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);  
}