Lucía Montiel Asensio

PRÁCTICA 3

Ejercicio 1

Este modelo contiene 3 clases, un tipo enumerado y una clase de asociación.

Las relaciones que unen Paciente y Expediente son dos. Una es direccionada con Expediente haciendo el rol de ExpedienteAbierto, por lo que en la clase Paciente se introduciría un atributo de tipo Expediente que haga referencia a esto. La implementación de la relación no direccionada de un paciente a uno o muchos expedientes se haría con un atributo que fuera una lista de tipo Expediente en la clase Paciente.

La clase de asociación Acceso la modelaríamos como otra clase que se encargaría de conectar las clases Profesional y Expediente, permitiendo así acceder a las consultas que han hecho los profesionales a los expedientes.

El código Java que implementa este modelo tal cual lo hemos descrito sería:

- Tipo enumerado *TipoAcceso*

```
public enum TipoAcceso {

    consulta, modificacion, creacion, archivo;
}
```

- Clase *Profesional*

```
public class Profesional {

    private List <Acceso> accesos;

    public  Profesional(){

        accesos = new ArrayList<>();
    }

    public void addAcceso(Acceso a){

        accesos.add(a);
    }

    public void removeAcceso(Acceso a){

        if(accesos.contains(a)){

            accesos.remove(a);
        }else{
```

```
            throw new RuntimeException("El acceso no existe");
        }
    }

    public Iterable<Expediente> expedientesAccedidos(){

        List<Expediente> exp = new ArrayList<>();
        for(Acceso a : accesos){

            exp.add(a.getExpediente());
        }

        return exp;
    }

}
```

- Clase *Expediente*

```
public class Expediente {

    private List<Acceso> accesos;

    public Expediente(){

        accesos = new ArrayList<>();
    }

    public void addAcceso(Acceso acceso) {
        this.accesos.add(acceso);
    }

    public void removeAcceso(Acceso acceso){

       if(accesos.contains(acceso)){

           accesos.remove(accesos);
       }
       else{

           throw new RuntimeException("El acceso no existe");
       }
    }

    public Iterable<Acceso> getAccesosExpediente(){

        return accesos;
    }

}
```

- Clase *Acceso*

```
public class Acceso {

    private Profesional profesional;
    private Expediente expediente;
    private Date fecha;
    private TipoAcceso tipo;

    public Acceso (Profesional p, Expediente e, Date f, TipoAcceso a){

        fecha = f;
        profesional = p;
        expediente = e;
        tipo = a;
    }

    public Profesional getProfesional() {
        return profesional;
    }

    public void setProfesional(Profesional profesional) {
        this.profesional = profesional;
    }

    public Expediente getExpediente() {
        return expediente;
    }

    public void setExpediente(Expediente expediente) {
        this.expediente = expediente;
    }

    public Date getFecha() {
        return fecha;
    }

    public void setFecha(Date fecha) {
        this.fecha = fecha;
    }

    public TipoAcceso getTipo() {
        return tipo;
    }

    public void setTipo(TipoAcceso tipo) {
        this.tipo = tipo;
    }
}
```

- Clase *Expediente*

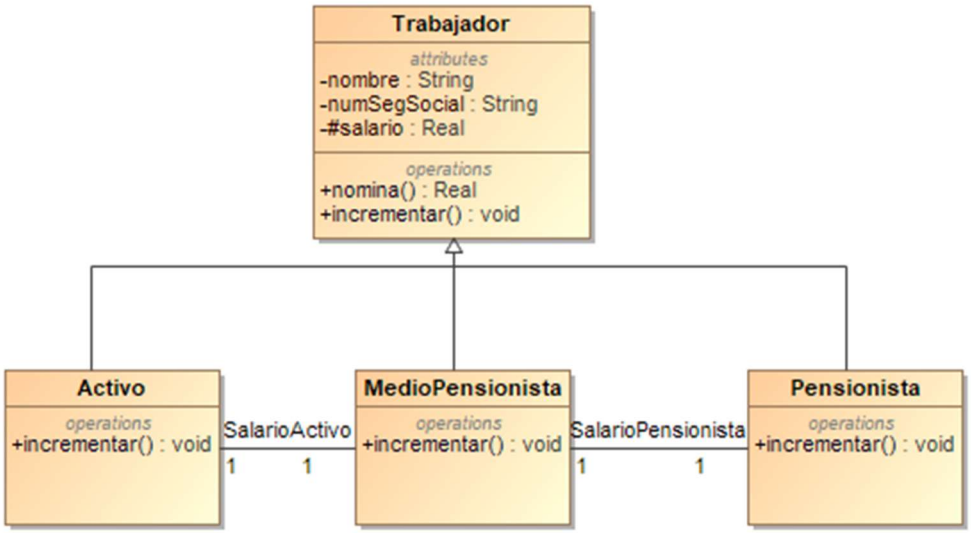
```
public class Paciente {

    private Expediente expedienteAbierto;
    private List<Expediente> expedientes;

    public Paciente() {

        expedienteAbierto = new Expediente();
        expedientes = new ArrayList<>();
    }

    public void addExpediente(Expediente e) {

        expedientes.add(e);
    }

    public Expediente getExpedienteAbierto() {
        return expedienteAbierto;
    }

    public void setExpedienteAbierto(Expediente expedienteAbierto) {
        if (expedientes.contains(expedienteAbierto)) {
            this.expedienteAbierto = expedienteAbierto;
        } else {
            throw new RuntimeException("El expediente no pertenece al
paciente");

        }
    }
}
```

Ejercicio 2

a) El modelo tal como está en el enunciado no se puede implementar ya que Java no permite la herencia múltiple.

b) Una solución válida sería que la clase MedioPensionista herede de Trabajador, al igual que Activo y Pensionista. Esta clase MedioPensionista se asociaría con Activo y Pensionista. La relación se implementaría con dos atributos, uno de tipo Activo y otro de tipo Pensionista, en la clase MedioPensionista que hicieran referencia al salario de estas dos clases para poder calcular así la nómina.

Un diagrama del modelo sería:



c) Implementación en Java:

- Clase *Trabajador*

```
public abstract class Trabajador {

    protected String nombre;
    protected String nSegSocial;
    protected float salario;

    public Trabajador(String nombre, String nSegSocial, float
salario){

        this.nombre = nombre;
        this.nSegSocial = nSegSocial;
        this.salario = salario;
    }

    public float nomina(){

        return salario;
    }
```

```
    public abstract void incrementar();
}
```

- Clase *Activo*

```
public class Activo extends Trabajador{

    public Activo(String nombre, String nSegSocial, float salario){

        super(nombre, nSegSocial, salario);
    }

    @Override
    public void incrementar() {
        super.salario *= 1.02;
    }
}
```

- Clase *Pensionista*

```
public class Pensionista extends Trabajador{

    public Pensionista (String nombre, String numSegSocial, float
salario){

        super(nombre, numSegSocial, salario);
    }

    @Override
    public void incrementar() {
        super.salario *= 1.04;
    }
}
```

- Clase *MedioPensionista*

```
public class MedioPensionista extends Trabajador{

    private Activo activo;
    private Pensionista pensionista;

    public MedioPensionista(String nombre, String numSegSocial, float
salarioActivo, float salarioPensionista){

        super(nombre, numSegSocial, salarioActivo +
salarioPensionista);
        this.activo = new Activo(nombre, numSegSocial, salarioActivo);
        this.pensionista = new Pensionista (nombre, numSegSocial,
salarioPensionista);
    }

    @Override
```

```
    public void incrementar() {

        activo.incrementar();;
        pensionista.incrementar();
        super.salario = activo.nomina() + pensionista.nomina();
    }
}
```

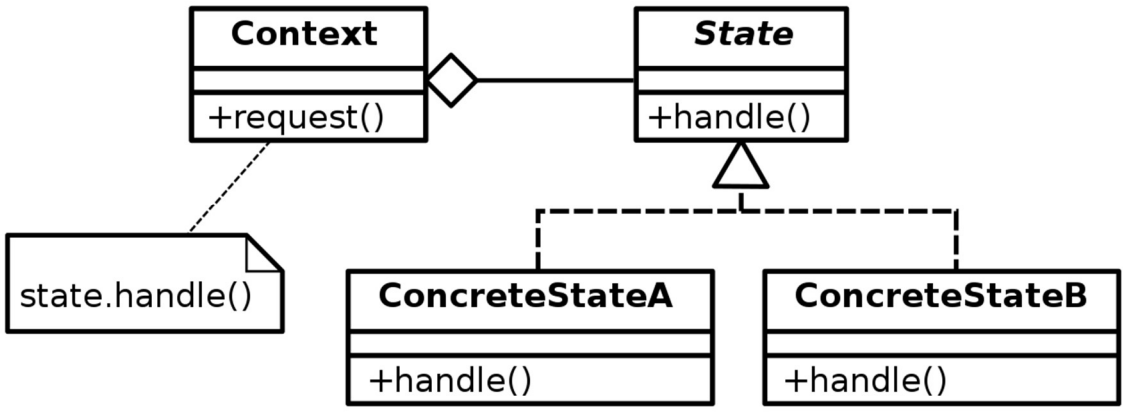
Ejercicio 3

Para este ejercicio existen varias estrategias de implementación:

- Comportamiento condicional: Consiste en realizar comprobaciones de las guardas dentro de la operación mediante bloques if/else para ejecutar la acción que el estado indica.

- Tablas de estado: Consiste en formar una tabla que indique para cada par de estados la acción necesaria para que transite de uno a otro.

- Patrón de diseño Estado: Es un patrón de diseño que delega las transiciones de estados a una clase Estado en la que se implementan los métodos de la clase principal. La clase Estado se especifica en subclases que, a la vez implementan los métodos de la clase Estado. Un esquema de este diseño sería:



El patrón Estado es la mejor opción ya que su implementación es más simple, siendo innecesario una tabla ni numerosos bloques if/else.

El código en Java sería:

- Clase Pieza:

```
public class Pieza {
}
```

- Clase Estado

```
public abstract class Estado {

    public abstract void put(Pieza pieza, Bandeja bandeja);
    public abstract Pieza get(Bandeja bandeja);
}
```

- Clase Empty

```
public class Empty extends Estado{


    @Override
    public void put(Pieza pieza, Bandeja bandeja) {

        if(bandeja.size()>1){

            bandeja.setEstado(new Normal());
        }else{

            bandeja.setEstado(new Full());
        }
        bandeja.getPiezas().add(pieza);
    }

    @Override
    public Pieza get(Bandeja bandeja) {
        throw new RuntimeException("No se puede recoger nada de una
bandeja vacía");
    }
}
```

- Clase Normal

```
public class Normal extends Estado{

    @Override
    public void put(Pieza pieza, Bandeja bandeja) {
        if(bandeja.getPiezas().size() == bandeja.getPiezas().size()-
1){

            bandeja.setEstado(new Full());
        }
        bandeja.getPiezas().add(pieza);
    }

    @Override
    public Pieza get(Bandeja bandeja) {

        Pieza p = bandeja.getPiezas().get(0);

        if(bandeja.getPiezas().size() == 1){

            bandeja.setEstado(new Empty());
        }

        bandeja.getPiezas().remove(0);

        return p;
    }
}
```

- Clase Full

```
public class Full extends Estado{

    @Override
    public void put(Pieza pieza, Bandeja bandeja) {

        throw new RuntimeException("No se puede incliur una nueva
pieza en una bandeja llena");
    }

    @Override
    public Pieza get(Bandeja bandeja) {
        Pieza p = bandeja.getPiezas().get(0);

        if(bandeja.size() > 1){

            bandeja.setEstado(new Normal());
        }else{

            bandeja.setEstado(new Empty());
        }
        bandeja.getPiezas().remove(0);

        return p;
    }
}
```