# Ejercicio 1

En este caso nos encontramos ante 3 clases, un tipo definido mediante una enumeración y 3 relaciones, una de ellas con una clase de asociación.

El tipo definido TipoAcceso se hará mediante un enum en Java, indicando todos los tipos que éste contiene, quedando el código como sigue.

public enum TipoAcceso {

*consulta*, *modificacion*, *creacion*, *archivo*

}

Empecemos por las relaciones entre la clase Paciente y la clase Expediente:

Tal y como se nos indica en la relación con extremo *expedienteAbierto* que, además, es direccionada, se realizaría introduciendo un atributo expedienteAbierto de tipo Expediente en la clase Paciente.

Ahora, con la relación Acceso entre Profesional y Expediente sólo tendríamos una solución posible, la cual implicaría implementar la clase Acceso como una aparte y hacer que ésta se encargue de la consistencia entre Profesional y Expediente, ya que, como se indica en el enunciado del ejercicio,  *se puedan conocer los diferentes accesos a cada expediente y también conocer los expedientes a los que ha accedido un determinado trabajador del hospital*, es decir, existe bidireccionalidad*.*

Quedando de la siguiente manera el código de andamiaje de las clases Acceso, Profesional y Expediente, suponiendo además que no se puede cambiar el valor de los atributos mediante un setter:

* Acceso

public class Acceso {

private Profesional profesional;

private Expediente expediente;

private Date fecha;

private TipoAcceso tipo;

public Acceso(Profesional profesional, Expediente expediente, Date fecha, TipoAcceso tipo) {

this.fecha = fecha;

this.tipo = tipo;

this.profesional = profesional;

this.expediente = expediente;

expediente.addAcceso(this);

profesional.addAcceso(this);

}

public void remove(){

this.expediente.removeAcceso(this);

this.profesional.removeAcceso(this);

}

public Profesional getProfesional() {

return profesional;

}

public void setProfesional(Profesional profesional) {

throw new RuntimeException("No puedes sobreescribir este atributo");

}

public Expediente getExpediente() {

return expediente;

}

public void setExpediente(Expediente expediente) {

throw new RuntimeException("No puedes sobreescribir este atributo");

}

public Date getFecha() {

return fecha;

}

public void setFecha(Date fecha) {

throw new RuntimeException("No puedes sobreescribir este atributo");

}

public TipoAcceso getTipo() {

return tipo;

}

public void setTipo(TipoAcceso tipo) {

throw new RuntimeException("No puedes sobreescribir este atributo");

}

}

* Profesional

public class Profesional {

private List<Acceso> accesosProfesional;

public void addAcceso(Acceso acceso) {

this.accesosProfesional.add(acceso);

}

public Profesional(){

this.accesosProfesional = new LinkedList<>();

}

public void removeAcceso(Acceso acceso) {

this.accesosProfesional.remove(acceso);

}

}

public Iterable<Expediente> expedientesAccedidos() {

List<Expediente> expedientes = new ArrayList<>();

for (Acceso acceso : accesosProfesional) {

expedientes.add(acceso.getExpediente());

}

return expedientes;

}

* Expediente

public class Expediente {

private List<Acceso> accesosExpediente;

public void addAcceso(Acceso acceso) {

this.accesosExpediente.add(acceso);

}

public Expediente(){

this.accesosExpediente = new LinkedList<>();

}

public void removeAcceso(Acceso acceso) {

this.accesosExpediente.remove(acceso);

}

}

public Iterable<Acceso> getAccesosExpediente() {

return accesosExpediente;

}

* Paciente

public class Paciente {

private Expediente expedienteAbierto;

private List<Expediente> expedientes;

public Paciente() {

this.expedientes = new ArrayList<>();

new Expediente(this);

}

public void addExpediente(Expediente expediente){

this.expedientes.add(expediente);

}

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

1. Las clases no se pueden implementar directamente en Java puesto que no existe la herencia múltiple en este lenguaje.
2. La solución propuesta sería hacer que MedioPensionista heredara de Activo y, además, hacer una relación con Pensionista para poder tener el salario heredado desde Activo y el obtenido a través de la relación con Pensionista. Esto nos daría el problema de que a la hora de calcular el salario total, no separaríamos el de Activo y Pensionista, por lo que podríamos heredar desde Trabajador y relacionarlo con Activo y Pensionista
3. Implementación

* Trabajador

public abstract class Trabajador {

public String nombre;

public String numeroSeguridadSocial;

protected float salario;

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

this.nombre = nombre;

this.numeroSeguridadSocial = numeroSeguridadSocial;

this.salario = salario;

}

public float nomina(){

return this.salario;

}

public abstract void incrementar();

}

* Activo

public class Activo extends Trabajador {

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

super(nombre, numeroSeguridadSocial, salario);

}

@Override

public void incrementar() {

super.salario \*= 1.02f;

}

}

* Pensionista

public class Pensionista extends Trabajador {

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

super(nombre, numeroSeguridadSocial, salario);

}

@Override

public void incrementar() {

super.salario \*= 1.04f;

}

}

* MedioPensionista

public class MedioPensionista extends Trabajador {

private Activo activo;

private Pensionista pensionista;

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

super(nombre, numeroSeguridadSocial, salarioActivo + salarioPensionista);

this.activo = new Activo(nombre,numeroSeguridadSocial,salarioActivo);

this.pensionista = new Pensionista(nombre, numeroSeguridadSocial, salarioPensionista);

}

@Override

public void incrementar() {

activo.incrementar();

pensionista.incrementar();

super.salario = activo.nomina() + pensionista.nomina();

}

}

# Ejercicio 3

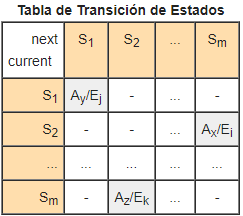
En este ejercicio existen 3 formas de realizar el código:

* Implementación condicional

Consiste en realizar comprobaciones de las guardas dentro de la operación mediante bloques if/else para realizar la ejecución que el estado mande realizar.

* Implementación mediante tabla de estado

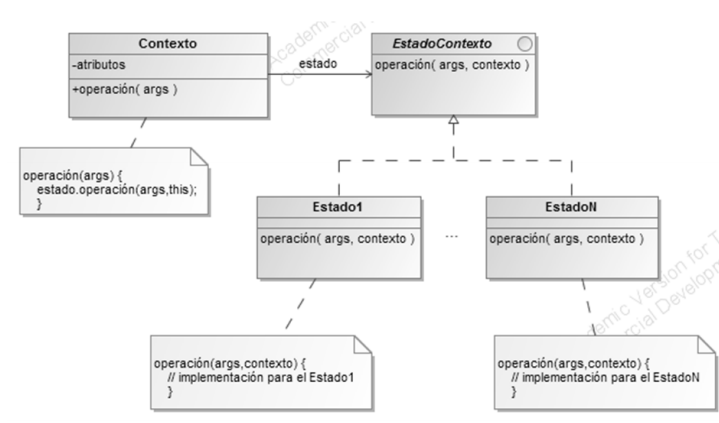
Consiste en implementar una tabla del siguiente estilo:



Mostrando para cada estado cual es el siguiente a razón del evento que le llegue.

* Implementación mediante el patrón de diseño Estado

Es un patrón de diseño que consiste en delegar las transiciones de estado a una clase externa Estado, en la que se implementen los distintos métodos de la clase principal y haya una implementación de cada método para el estado en el que se encuentre, siguiendo este patrón:



En este caso, el patrón estado es la mejor opción ya que no requiere de realizar una tabla compleja ni excesos de bloques If/else en nuestro código, simplemente una llamada a un método y éste se encarga de hacerlo todo. Lo único que habría que cambiar un poco la implementación de los métodos put() y get(), ya que a los estados habría que pasarles la capacidad de la cinta como parámetro, para así poder transitar de una forma o de otra.

Así pues, el código que quedaría es el siguiente:

* Pieza

Esta clase no contiene nada tal y como se indica en el diagrama

public abstract class Pieza {

}

* Bandeja

*public class Bandeja {*

*private int capacidad;*

*private List<Pieza> piezas;*

*private Estado estado;*

*public Bandeja(int capacidad) {*

*if (capacidad < 1) {*

*throw new RuntimeException("Una bandeja debe tener capacidad 1 o más");*

*} else {*

*this.capacidad = capacidad;*

*}*

*this.piezas = new ArrayList<>();*

*this.estado = new Empty();*

*}*

*public void put(Pieza pieza) {*

*estado.put(pieza, this);*

*}*

*public Pieza get() {*

*return estado.get(this);*

*}*

*public int size() {*

*return capacidad;*

*}*

*public void setEstado(Estado estado) {*

*this.estado = estado;*

*}*

*public List<Pieza> getPiezas() {*

*return piezas;*

*}*

*}*

* *Estado*

public abstract class Estado {

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

public abstract Pieza get(Bandeja bandeja);

}

* Empty < *Estado*

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 puedes coger nada de una bandeja vacía");

}

}

* Normal < *Estado*

public class Normal extends Estado {

@Override

public void put(Pieza pieza, Bandeja bandeja) {

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

bandeja.setEstado(new Full());

}

bandeja.getPiezas().add(pieza);

}

@Override

public Pieza get(Bandeja bandeja) {

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

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

bandeja.setEstado(new Empty());

}

bandeja.getPiezas().remove(0);

return primera;

}

}

* Full < *Estado*

public class Full extends Estado {

@Override

public void put(Pieza pieza, Bandeja bandeja) {

throw new RuntimeException("No puedes meter nada en una bandeja llena");

}

@Override

public Pieza get(Bandeja bandeja) {

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

if (bandeja.size() > 1) {

bandeja.setEstado(new Normal());

} else {

bandeja.setEstado(new Empty());

}

bandeja.getPiezas().remove(0);

return primera;

}

}