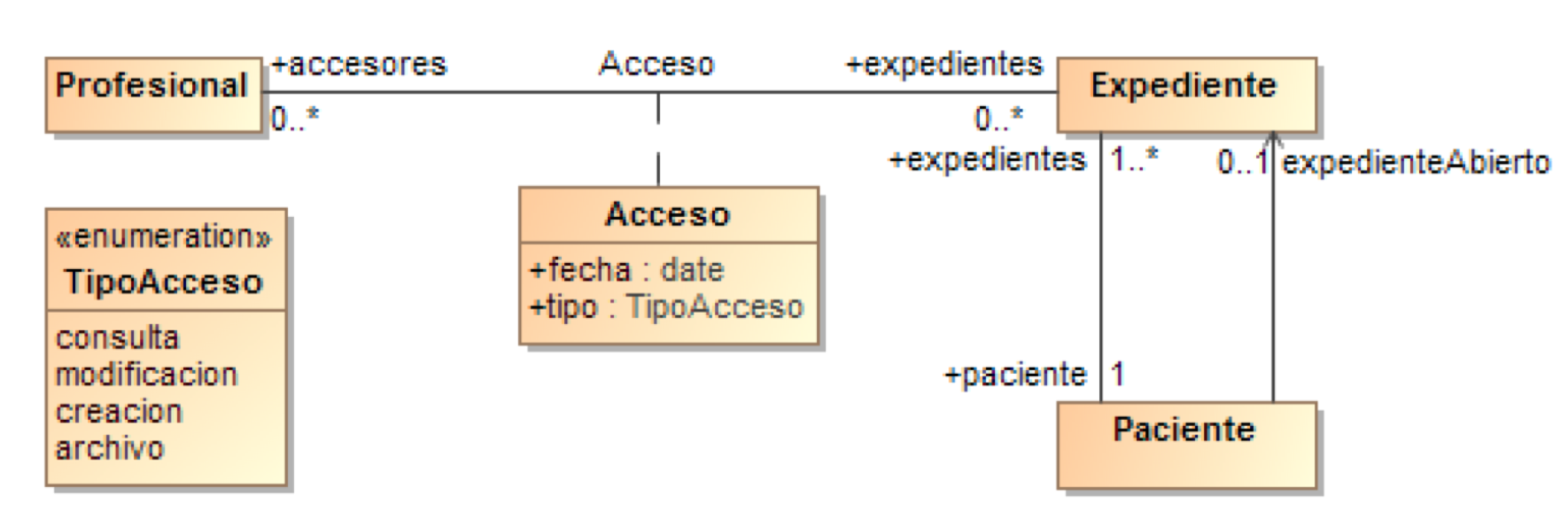
**Práctica tema 5**

1. **Discutir las diferentes formas de diseñar un esquema del código de andamiaje Java necesario para implementar este modelo, y en particular la relación *Acceso*. Implementar en Java la decisión que se considere mejor en este caso, y justificar las decisiones que han llevado a elegir dicha estrategia de implementación.**

Para poder implementar correctamente este modelo, primero debemos reificar la clase de asociación Acceso. Puesto que es una relación de muchos a muchos bidireccional entre Profesional y Expediente, debemos añadir referencias a las clases Profesional y Expediente en la clase Acceso para que ésta se haga cargo de la consistencia. Por último, debemos añadir una Lista de Accesos en las clases Profesional y Expediente para poder saber todos los expedientes a los que ha accedido un profesional y viceversa.

Entre Expediente y Paciente existen 2 relaciones, expedienteAbierto y expedientes. ExpedienteAbierto al ser una relación unidireccional, con tener en paciente una referencia a la clase Expediente, bastaría, sin embargo, en la relación de expedientes, debemos añadir una Lista de expedientes, y luego, en expedientes, una referencia al paciente al que pertenece dicho expediente.

A la hora de obtener los expedientes accedidos por un profesional hemos optado por realizar un método que obtiene todos los expedientes a los que ha accedido el profesional y los devuelve en un Iterable de Expedientes, de este modo, no permitimos al usuario modificar la lista de expedientes accedidos. De mismo modo, se realiza un método similar en la clase Expediente que obtiene todos los profesionales que han accedido a ese expediente devueltos en un Iterable de Profesionales.

Clase Acceso

*package* practica3.ejercicio1;

*import* java.util.Date;

*public class* Acceso {

*private* Date fecha;

*private* TipoAcceso tipo;

*private* Expediente accedido;

*private* Profesional usuario;

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

*this*.accedido = expediente;

*this*.usuario = profesional;

profesional.addAcceso(*this*);

expediente.addAcceso(*this*);

*this*.tipo = tipo;

*this*.fecha = fecha;

}

*public void* removeAcceso() {

accedido.removeAcceso(*this*);

usuario.removeAcceso(*this*);

}

*public void* setFecha(Date fecha) {

*this*.fecha = fecha;

}

*public void* setTipo(TipoAcceso tipo) {

*this*.tipo = tipo;

}

*public void* setAccedido(Expediente accedido) {

*this*.accedido = accedido;

}

*public void* setUsuario(Profesional usuario) {

*this*.usuario = usuario;

}

*public* Date getFecha() {

*return* fecha;

}

*public* TipoAcceso getTipo() {

*return* tipo;

}

*public* Expediente getAccedido() {

*return* accedido;

}

*public* Profesional getUsuario() {

*return* usuario;

}

@Override

*public* String toString() {

*return* "Acceso{" +

"fecha=" + fecha +

", tipo=" + tipo +

'}';

}

}

Clase Expediente

*package* practica3.ejercicio1;

*import* java.util.ArrayList;

*import* java.util.LinkedList;

*import* java.util.List;

*public class* Expediente {

*private* List<Acceso> listaAccesos;

*private* Paciente paciente;

*public* Expediente(Paciente paciente) {

listaAccesos = *new* LinkedList<>();

*this*.paciente = paciente;

paciente.addExpediente(*this*);

}

@Override

*public* String toString() {

*return* "Expediente{" +

"listaAccesos=" + listaAccesos +

'}';

}

*public void* addPaciente(Paciente paciente) {

*this*.paciente = paciente;

}

*public void* addAcceso(Acceso acceso) {

*this*.listaAccesos.add(acceso);

}

*public void* removeAcceso(Acceso acceso) {

*this*.listaAccesos.remove(acceso);

}

*public void* eliminarExpediente(){

*this*.paciente.eliminarExpediente(*this*);

}

*public* Iterable<Acceso> getAllAccesosByProfesionales(){

*return* listaAccesos;

}

}

Clase Paciente

*package* practica3.ejercicio1;

*import* java.util.LinkedList;

*import* java.util.List;

*public class* Paciente {

*private* Expediente expedienteAbierto;

*private* List<Expediente> expedientes;

*public* Paciente() {

expedientes = *new* LinkedList<>();

Expediente expediente = *new* Expediente(*this*);

expediente.addPaciente(*this*);

}

*public void* addExpediente(Expediente expediente) {

*this*.expedientes.add(expediente);

}

*public void* eliminarExpediente(Expediente expediente) {

*this*.expedientes.remove(expediente);

}

@Override

*public* String toString() {

*return* "Paciente{" +

"expedienteAbierto=" + expedienteAbierto +

", expedientes=" + expedientes +

'}';

}

}

Clase Profesional

*package* practica3.ejercicio1;

*import* java.util.ArrayList;

*import* java.util.List;

*public class* Profesional {

*private* List<Acceso> listaAccesos;

*public void* addAcceso(Acceso acceso) {

*this*.listaAccesos.add(acceso);

}

*public void* removeAcceso(Acceso acceso) {

*this*.listaAccesos.remove(acceso);

}

*public* Iterable<Expediente> getAllAccesosExpedientes() {

List<Expediente> listaExpedientes = *new* ArrayList<>();

*for*(Acceso acceso : listaAccesos) {

listaExpedientes.add(acceso.getAccedido());

}

*return* listaExpedientes;

}

@Override

*public* String toString() {

*return* "Profesional{" +

"listaAccesos=" + listaAccesos +

'}';

}

}

**2.a)** No es posible realizar esa implementación en java puesto que la herencia múltiple de clases no está permitida en Java. Esto se debe precisamente para evitar que una clase pueda heredar de dos clases distintas, pero que implementen una misma función, con comportamientos distintos, haciendo que el comportamiento no sea el esperado (Problema del diamante).

**2.b)** El enunciado del problema nos dicen que el MedioPensionista tiene que tener el sueldo del Pensionista y del Activo, además de heredar de *Trabajador*, para implementar la operación *incrementar()*, por tanto una posible solución a este problema es extender la clase de MedioPensionista directamente de *Trabajador*, y, a la hora de crear el MedioPensionista, en el constructor crear 1 objeto de tipo Activo y otro de tipo Pensionista y almacenarlos en dos atributos de la clase. El constructor de MedioPensionista recibirá los 4 atributos, el número de la seguridad social, el nombre y 2 sueldos, el de activo y el de pensionista, para así poder crear los objetos Activo y Pensionista. De este modo, a la hora de realizar el incremento, llamamos a la función incremento de Activo y a la de Pensionista y se suman para guardarlo en el atributo saldo de MedioPensionista.

**2.c)**

Clase Activo:

*package* practica3.ejercicio2;

*public class* Activo *extends* Trabajador{

*public* Activo(String nombre, String numeroSeguridadSocial, *double* salarioActivo) {

*super*(nombre, numeroSeguridadSocial, salarioActivo);

}

@Override

*public void* incrementa() {

*super*.salario \*= 1.02;

}

}

Clase Medio Pensionista

*package* practica3.ejercicio2;

*public class* MedioPensionista *extends* Trabajador {

*private* Activo activo;

*private* Pensionista pensionista;

*public* MedioPensionista(String nombre, String numeroSeguridadSocial, *double* salarioActivo, *double* salarioPensionista) {

*super*(nombre, numeroSeguridadSocial, salarioActivo + salarioPensionista);

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

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

}

@Override

*public void* incrementa() {

activo.incrementa();

pensionista.incrementa();

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

}

}

Clase Pensionista

*package* practica3.ejercicio2;

*public class* Pensionista *extends* Trabajador {

*public* Pensionista(String nombre, String numeroSeguridadSocial, *double* salarioPensionista) {

*super*(nombre, numeroSeguridadSocial, salarioPensionista);

}

@Override

*public void* incrementa() {

*super*.salario \*= 1.04;

}

}

Clase Trabajador

*package* practica3.ejercicio2;

*public abstract class* Trabajador {

*public* String nombre;

*public* String numeroSeguridadSocial;

*protected double* salario;

*public* Trabajador(String nombre, String numeroSeguridadSocial, *double* salario) {

*this*.nombre = nombre;

*this*.numeroSeguridadSocial = numeroSeguridadSocial;

*this*.salario = salario;

}

*public double* nomina() {

*return* salario;

}

*public abstract void* incrementa();

}

**3)** En clase hemos visto 3 maneras distintas de implementar una clase con sus estados y transiciones correspondientes. Una de ellas es utilizando comportamiento condicional. Ésta solución, respecto al patrón estado, ensucia mucho el código, puesto que se tiene que guardar en un enumerado los estados, y en otro los cambios entre estados. A la hora de cambios de comportamiento en algunas funciones, se tendría que añadir sentencias de control concatenadas para cada estado, complicando el código a la hora de añadir nuevas funcionalidades y dificultando su mantenimiento. Otra de las maneras de implementar un *FSM* en java es utilizando una tabla en la que aparezcan todos los estados que representa por cada estado su siguiente estado dependiendo del evento que llegue. Sin embargo esta solución requiere construir la tabla, y si nuestro *FSM* tiene muchos estados y transiciones, complica su construcción. Por ello se utiliza el patrón estado, que, para cada estado tienes una clase que implementa una interfaz con las funciones cuyo comportamiento sea distinto dependiendo del estado en el que se encuentra la clase, facilitando su mantenimiento y expansión.

Clase Bandeja:

*package* practica3.ejercicio3;

*import* java.util.ArrayList;

*import* java.util.List;

*public class* Bandeja {

*private int* capacidad;

*protected* List<Pieza> piezaList;

*private* BandejaState currentState;

*public* Bandeja(*int* capacidad) {

piezaList = *new* ArrayList<>();

*this*.capacidad = capacidad;

*this*.currentState = *new* EmptyState();

}

*public void* put(Pieza p) {

currentState.put(p, *this*);

}

*public* Pieza get() {

*return* currentState.get(*this*);

}

*public int* getCapacidad() {

*return* capacidad;

}

*public void* setEstado(BandejaState newState) {

*this*.currentState = newState;

}

Interfaz BandejaState

*package* practica3.ejercicio3;

*public interface* BandejaState {

*void* put(Pieza p, Bandeja bandeja);

Pieza get(Bandeja bandeja);

}

Clase EmptyState

*package* practica3.ejercicio3;

*public class* EmptyState *implements* BandejaState {

@Override

*public void* put(Pieza p, Bandeja bandeja) {

*if*(bandeja.getCapacidad() == 1) {

bandeja.setEstado(*new* FullState());

} *else if*(bandeja.getCapacidad() > 1) {

bandeja.setEstado(*new* NormalState());

}

bandeja.piezaList.add(p);

}

@Override

*public* Pieza get(Bandeja bandeja) {

*throw new* RuntimeException("No se puede obtener piezas de una bandeja vacía");

}

@Override

*public* String toString() {

*return* "EmptyState{}";

}

}

Clase NormalState

*package* practica3.ejercicio3;

*public class* NormalState *implements* BandejaState {

@Override

*public void* put(Pieza p, Bandeja bandeja) {

*if*(bandeja.piezaList.size() == bandeja.getCapacidad() - 1) {

bandeja.setEstado(*new* FullState());

}

bandeja.piezaList.add(p);

}

@Override

*public* Pieza get(Bandeja bandeja) {

*if*(bandeja.piezaList.size() == 1) {

bandeja.setEstado(*new* EmptyState());

}

Pieza first = bandeja.piezaList.get(0);

bandeja.piezaList.remove(first);

*return* first;

}

@Override

*public* String toString() {

*return* "NormalState{}";

}

}

Clase FullState

*package* practica3.ejercicio3;

*public class* FullState *implements* BandejaState {

@Override

*public void* put(Pieza p, Bandeja bandeja) {

*throw new* RuntimeException("No se pueden añadir piezas a una bandeja llena");

}

@Override

*public* Pieza get(Bandeja bandeja) {

*if*(bandeja.getCapacidad() == 1) {

bandeja.setEstado(*new* EmptyState());

} *else* {

bandeja.setEstado(*new* NormalState());

}

Pieza first = bandeja.piezaList.get(0);

bandeja.piezaList.remove(first);

*return* first;

}

@Override

*public* String toString() {

*return* "FullState{}";

}

}

Clase Pieza

*package* practica3.ejercicio3;

*public abstract class* Pieza {

}

Podemos observar como en FullState lanza una excepción a la hora de añadir nuevas piezas, al igual que en EmptyState cuando intentamos obtener una pieza.