Practica 3

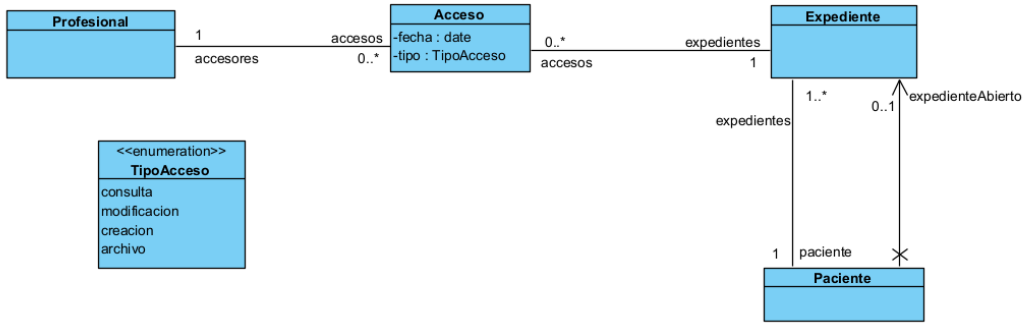
Practica realizada por Alba Ruiz Gutiérrez, Pablo Pardo Fernández, Pablo Ruiz Galiánez, Jorge Velázquez y Pablo Robles

Ejercicio 1

Discutir las diferentes formas de diseñar un esquema del código de andamiaje Java necesario para implementar este modelo. Realizar el diagrama de diseño (con cualquiera de las herramientas) y la implementación en Java con la decisión que se considere mejor en este caso, y justificar las decisiones que han llevado a elegir dicha estrategia de implementación.

Hemos elegido esta estrategia de implementación que utiliza andamiaje para construir asociaciones en un modelo orientado a objetos debido a que se enfoca en la privacidad de los atributos, con operaciones específicas como get(), set(), add(), nomenclatura basada en roles y clases, y destaca la dirección preferente de la asociación en la implementación. Al evitar asociaciones bidireccionales innecesarias se mejora la flexibilidad del diseño y se mantiene la coherencia con principios de diseño orientado a objetos, como encapsulamiento y bajo acoplamiento lo que hace que nuestro modelo sea más robusto y fácil de mantener.

Modelo:



Expediente.java:

```
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;

public class Expediente {
  private List<Acceso> acceso;
  private Paciente paciente;

  public Expediente(Paciente p) throws Exception {
    assert p != null;
    this.paciente = p;
    acceso = new ArrayList<Acceso>();
    this.paciente.abrirExpediente(this);
  }

  protected void addAcceso(Acceso a) {
    assert a != null;
    this.acceso.add(a);
  }

  public Enumeration<Acceso> getAcceso() {
```

```
    return java.util.Collections.enumeration(acceso);
  }

  public Paciente getPaciente() {
    return this.paciente;
  }
}
```

Acceso.java:

```
import java.util.Date;

public class Acceso {
  private Date fecha;
  private TipoAcceso tipo;
  private Profesional p;
  private Expediente e;

  public Acceso(Date fecha, TipoAcceso tipo, Profesional p, Expediente e) {
    assert fecha != null;
    assert tipo != null;
    assert p != null;
    assert e != null;

    this.fecha = fecha;
    this.tipo = tipo;
    this.p = p;
    this.e = e;
    p.addAcceso(this);
    e.addAcceso(this);
  }

  public Date getFecha() {
    return fecha;
  }

  public TipoAcceso getTipo() {
    return tipo;
  }

  public Profesional getP() {
    return p;
  }

  public Expediente getE() {
    return e;
  }
}
```

Paciente.java

```
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;

public class Paciente {
  private List<Expediente> expedientes;
  private Expediente expedienteAbierto;

  public Paciente() {
    expedientes = new ArrayList<Expediente>();
    expedienteAbierto = null;
  }

  protected void addExpediente(Expediente e) {
    assert e != null;
    expedientes.add(e);
  }
```

```
  protected void abrirExpediente(Expediente e) throws Exception{
    assert e != null;
    if(expedienteAbierto == null) {
      expedienteAbierto = e;
      addExpediente(e);
    }else {
      throw new Exception("Ya hay un expediente abierto para este paciente");
    }
  }

  protected void cerrarExpediente() throws Exception{
    if(expedienteAbierto != null) {
      expedienteAbierto = null;
    }else {
      throw new Exception("No hay ning�n expediente abierto para este paciente");
    }
  }

  public Enumeration<Expediente> getExpedientes() {
    return java.util.Collections.enumeration(expedientes);
  }

  public Expediente getExpedienteAbierto() {
    return expedienteAbierto;
  }
}
```

Profesional.java

```
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;

public class Profesional {
  private List<Acceso> acceso;

  public Profesional() {
    acceso = new ArrayList<Acceso>();
  }

  protected void addAcceso(Acceso a) {
    assert a != null;
    this.acceso.add(a);
  }

  public Enumeration<Acceso> getAcceso() {
    return java.util.Collections.enumeration(acceso);
  }
}
```

TipoAcceso.java

```
public enum TipoAcceso {consulta,modificacion,creacion,archivo}
```

Ejercicio 2

a) Justificar por qué las clases descritas no pueden ser implementadas directamente en Java.

Java no admite herencia múltiple de clases, lo que significa que una clase solo puede heredar de una sola clase directamente. En este caso, la clase MedioPensionista necesitaría heredar tanto de Activo como de Pensionista, ya que tiene atributos y comportamientos específicos de ambas clases.

b) Discutir y desarrollar una solución que permita resolver la situación descrita. Como es lógico, la solución propuesta debe mantener la funcionalidad actualmente existente en las tres subclases de Trabajadores y asegurar la consistencia de los atributos y la reutilización de los métodos de las clases Trabajador, Activo y Pensionista.

En la implementación, hemos utilizado una jerarquía de clases para representar distintos tipos de trabajadores en un sistema de nóminas. La clase abstracta Trabajador define atributos y métodos básicos, mientras que las clases concretas Activo y Pensionista heredan de ella para modelar trabajadores activos y pensionistas respectivamente.

Para manejar el caso especial de MedioPensionista, donde necesitamos combinar características de Activo y Pensionista, hemos optado por una solución de composición. MedioPensionista contiene instancias de Activo y Pensionista como atributos internos. De esta manera, logramos evitar limitaciones de herencia múltiple y mantenemos una estructura coherente.

c) Implementar en Java la solución propuesta, realizando el diagrama de diseño correspondiente

Activo.java

```
package ejercicio2P3;

public class Activo extends Trabajador{


    public Activo(String nombre, String numeroSeguridadSocial, float salario) {
      super(nombre, numeroSeguridadSocial, salario);
    }

    // Implementación del incremento del salario para un trabajador activo
    @Override
    public void incrementar() {
      salario = (float) (salario * 1.02);
    }

    @Override
    public float nomina() {
      return this.salario;
    }
}
```

Pensionista.java

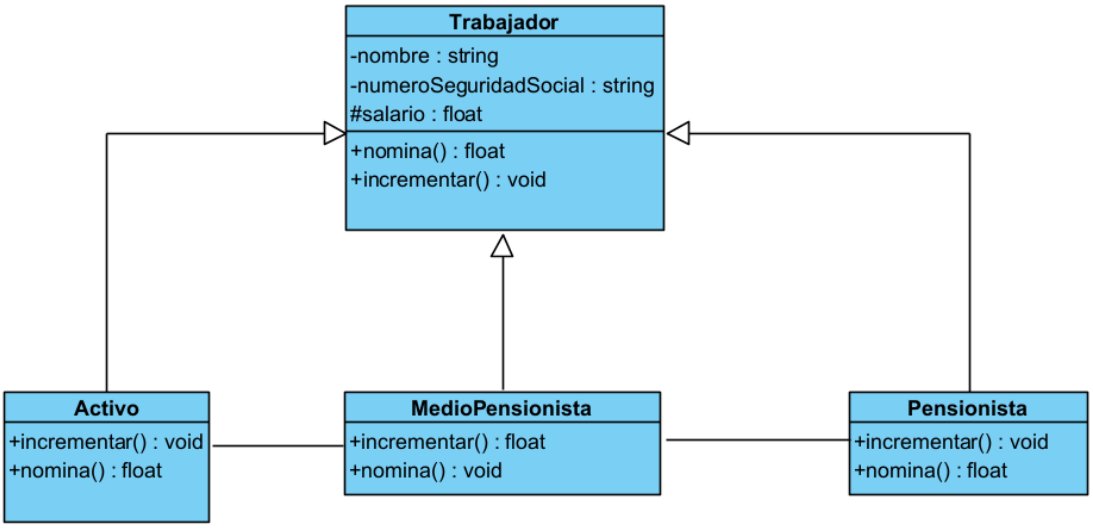
```
package ejercicio2P3;

public class Pensionista extends Trabajador{

  public Pensionista(String nombre, String numeroSeguridadSocial, float salario) {
    super(nombre, numeroSeguridadSocial, salario);
        }

    // Implementación del incremento del salario para un trabajador activo
    @Override
      public void incrementar() {
          salario = (float) (salario * 1.04);
      }
    @Override
    public float nomina() {
      return this.salario;
    }
}
```

Trabajador.java

```
package ejercicio2P3;

public abstract class Trabajador{
  private String nombre;
  private String numeroSeguridadSocial;
  protected float salario;
  public Trabajador(String nombre, String numeroSeguridadSocial, float salario) {
    assert nombre != null;
    assert numeroSeguridadSocial != null;
    assert salario > 0;

    this.nombre = nombre;
    this.numeroSeguridadSocial = numeroSeguridadSocial;
    this.salario = salario;
  }
  public Trabajador(String nombre, String numeroSeguridadSocial) {
    this.nombre = nombre;
    this.numeroSeguridadSocial = numeroSeguridadSocial;
  }
  public String getNombre() {
    return nombre;
  }
  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
  public String getNumeroSeguridadSocial() {
    return numeroSeguridadSocial;
  }
  public void setNumeroSeguridadSocial(String numeroSeguridadSocial) {
    this.numeroSeguridadSocial = numeroSeguridadSocial;
  }
  public abstract float nomina();
  public abstract void incrementar();
```

MedioPensionista.java

```
package ejercicio2P3;

public class MedioPensionista extends Trabajador{


  private Activo perfilActivo;
  private Pensionista perfilPensionista;



  public MedioPensionista(String nombre, String numeroSeguridadSocial, float salarioActivo, float salarioPensionista) {
    super(nombre, numeroSeguridadSocial,(salarioActivo + salarioPensionista));
    this.perfilActivo = new Activo(nombre,numeroSeguridadSocial, salarioActivo);
    this.perfilPensionista = new Pensionista(nombre,numeroSeguridadSocial, salarioPensionista);
  }



  @Override
  public void incrementar() {
    perfilActivo.incrementar();
    perfilPensionista.incrementar();
    this.salario = nomina();

  }
  @Override
  public float nomina() {
    return (perfilPensionista.nomina() + perfilActivo.nomina());
  }


}
```



Ejercicio 3

El ejemplo de la cadena de montaje de martillos planteaba el modelado de las bandejas que contenían las piezas de acuerdo al siguiente modelo.
Además, el diagrama de estados de una Bandeja distinguía los distintos estados por los que pasa una Bandeja y el efecto de las operaciones en cada uno de ellos.

Se pide implementar en Java dicho modelo de acuerdo al diagrama de estados, justificando la forma elegida para hacerlo de entre las tres posibles vistas en clase. Para ello, considerar la posibilidad de utilizar el patrón de diseño Estado, justificando las ventajas de esta solución frente a las otras posibles soluciones de diseño. Realizar asimismo el diagrama de diseño correspondiente.

El patrón de diseño Estado es preferible en este caso porque permite que un objeto cambie su comportamiento cuando su estado interno cambia, sin modificar su estructura. Esto promueve un código más limpio, mantenible y extensible, ya que cada estado se modela como una clase separada, lo que facilita la adición de nuevos estados en el futuro. Además, el uso del patrón Estado evita largas listas de condicionales que tendríamos usando otro patrón, lo que mejora la legibilidad del código.

Bandeja.java

```
package ejercicio3P3;

import java.util.ArrayList;
import java.util.Enumeration;
import java.util.LinkedList;
import java.util.List;

public class Bandeja {
  private int capacidad;
  public List<Pieza> piezas;
  private Estado estado;
  public Bandeja(int capacidad) {
    assert (capacidad > 0);
    this.capacidad = capacidad;
    piezas = new LinkedList<>();
    estado = new Empty(this);
  }
  public int getCapacidad() {
```

```
    return capacidad;
  }
  public void put(Pieza p) {
    estado.put(p);
  }
  public Pieza get() {

    return estado.get();
  }
  public int size() {
    return (piezas.size());
  }
  public void setEstado(Estado estado) {
    this.estado = estado;
  }

  public Enumeration<Pieza> getPiezas() {
    return java.util.Collections.enumeration(piezas);
  }
}
```

Estado.java

```
package ejercicio3P3;

public abstract class Estado {
    protected Bandeja bandeja;
    public Estado(Bandeja bandeja) {
        assert (bandeja != null);
        this.bandeja = bandeja;
    }

    public abstract void put(Pieza p);
    public abstract Pieza get();
    public int size(){
        return bandeja.size();
    }
}
```

Empty.java

```
package ejercicio3P3;

public class Empty extends Estado{
    public Empty(Bandeja bandeja) {
        super(bandeja);
    }

    @Override
    public void put(Pieza p) {
        assert (p != null);

        if(bandeja.getCapacidad()>1){
            bandeja.setEstado(new Normal(bandeja));
        }else{
            bandeja.setEstado(new Full(bandeja));
        }
        bandeja.piezas.add(p);
        p.setBandeja(bandeja);

    }

    @Override
    public Pieza get() {
        assert size() > 0;
        return null;
    }
}
```

Normal.java

```
package ejercicio3P3;

public class Normal extends Estado{
    public Normal(Bandeja bandeja) {
        super(bandeja);
    }

    @Override
    public void put(Pieza p) {
        assert (p != null);
        if(size()<bandeja.getCapacidad()-1){
            bandeja.setEstado(new Normal(bandeja));
        }else{
            bandeja.setEstado(new Full(bandeja));
        }
        bandeja.piezas.add(p);
        p.setBandeja(bandeja);


    }

    @Override
    public Pieza get() {
        if(size()>1){
            bandeja.setEstado(new Normal(bandeja));
        }else{
            bandeja.setEstado(new Empty(bandeja));
        }
        Pieza p = bandeja.piezas.remove(0);
        p.setBandeja(null);
        return p;
    }
}
```

Full.java

```
package ejercicio3P3;

public class Full extends Estado{
    public Full(Bandeja bandeja) {
        super(bandeja);
    }

    @Override
    public void put(Pieza p) {
        assert p!=null;
        assert size() < bandeja.getCapacidad();
    }

    @Override
    public Pieza get() {
        if(bandeja.getCapacidad()>1){
            bandeja.setEstado(new Normal(bandeja));
        }else{
            bandeja.setEstado(new Empty(bandeja));
        }

        Pieza p = bandeja.piezas.remove(0);
        p.setBandeja(null);
        return p;
    }
}
```

Pieza.java

```
package ejercicio3P3;

public class Pieza {
  private Bandeja bandeja;
  public Pieza(Bandeja b) {
    bandeja = b;
  }

  public void setBandeja(Bandeja bandeja) {
    this.bandeja = bandeja;
  }
}
```