



UNIVERSIDAD  
Popular del cesar

# Ingeniería de Sistemas

**ESPECIALIZACION EN INGENIERIA DE SOFTWARE**  
**MODULO PATRONES DE DISEÑO DE SOFTWARE**



## EL DOCENTE



**JAIRO FRANCISCO  
SEOANES LEON**

[jairoseoanes@unicesar.edu.co](mailto:jairoseoanes@unicesar.edu.co)  
(300) 600 06 70



## Educación formal

- ✓ **Ingeniero de sistemas**, Universidad Popular del Cesar sede Valledupar, Feb 2002 – Jun 2009.
- ✓ **MsC en Ingeniería de Sistemas y Computación**, Universidad Nacional de Colombia, Bogotá, Feb 2011 – Mar 2015
- ✓ **PhD Ciencia, Tecnología e innovación**, Urbe, Venezuela, Mayo 2024

## Formación complementaria

- ✓ **AWS Academy Graduate** - AWS Academy Cloud Foundations, 2022  
<https://www.credly.com/go/p3Uwht36>
- ✓ **Associate Cloud Engineer Path** - Google Cloud Academy, 2022  
[https://www.cloudskillsboost.google/public\\_profiles/c7e7936c-3e37-4bad-b822-74d40c49d0db](https://www.cloudskillsboost.google/public_profiles/c7e7936c-3e37-4bad-b822-74d40c49d0db)
- ✓ **Fundamentos De Programación Con Énfasis En Cloud Computing** – AWS Academy y Misión Tic 2022
- ✓ **Google Cloud Computing Foundations** – Google Academy, 2022
- ✓ **Aplicación de cloud: retos y oportunidades de mejora para las empresas de software gestionando la computación en la nube** – Fedesoft, 2023
- ✓ **Desarrollo De Aplicaciones Web En Angular, Para El Nivel Frontend** – Universidad EAFIT, 2023
- ✓ **Microsoft Scrum Foundations** – Intelligent Training - MinTic , 2023

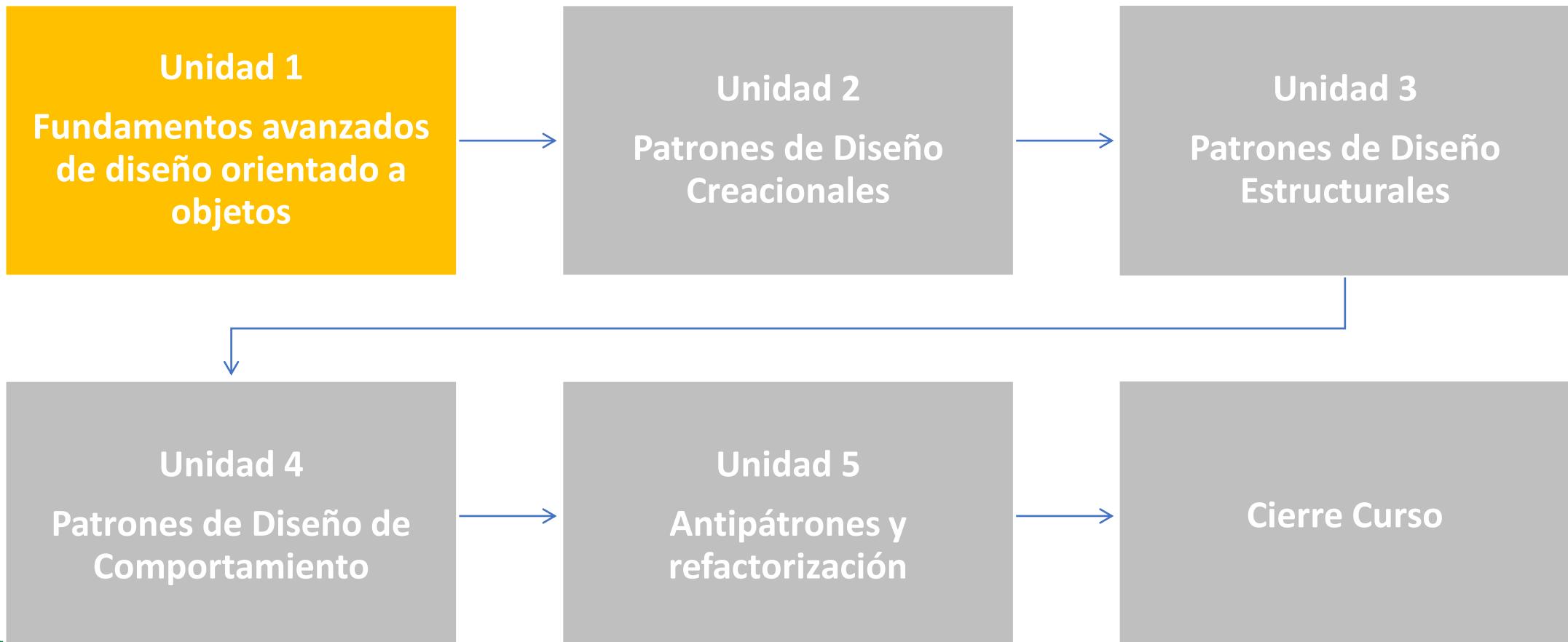
## Experiencia profesional

- ✓ **Docente Universitario**, Universidad Popular del Cesar sede Valledupar, marzo del 2013.
- ✓ **Técnico de Sistemas Grado 11**, Rama judicial Seccional Cesar, SRPA Valledupar, Junio del 2009

# MODULO DE PATRONES DE DISEÑO DE SOFTWARE



# MODULO DE PATRONES DE DISEÑO DE SOFTWARE



# Unidad 1. Fundamentos avanzados de diseño O.O

**1.1** Abstracción y encapsulamiento

**1.2** Acoplamiento y cohesión

**1.3** Delegación y responsabilidad

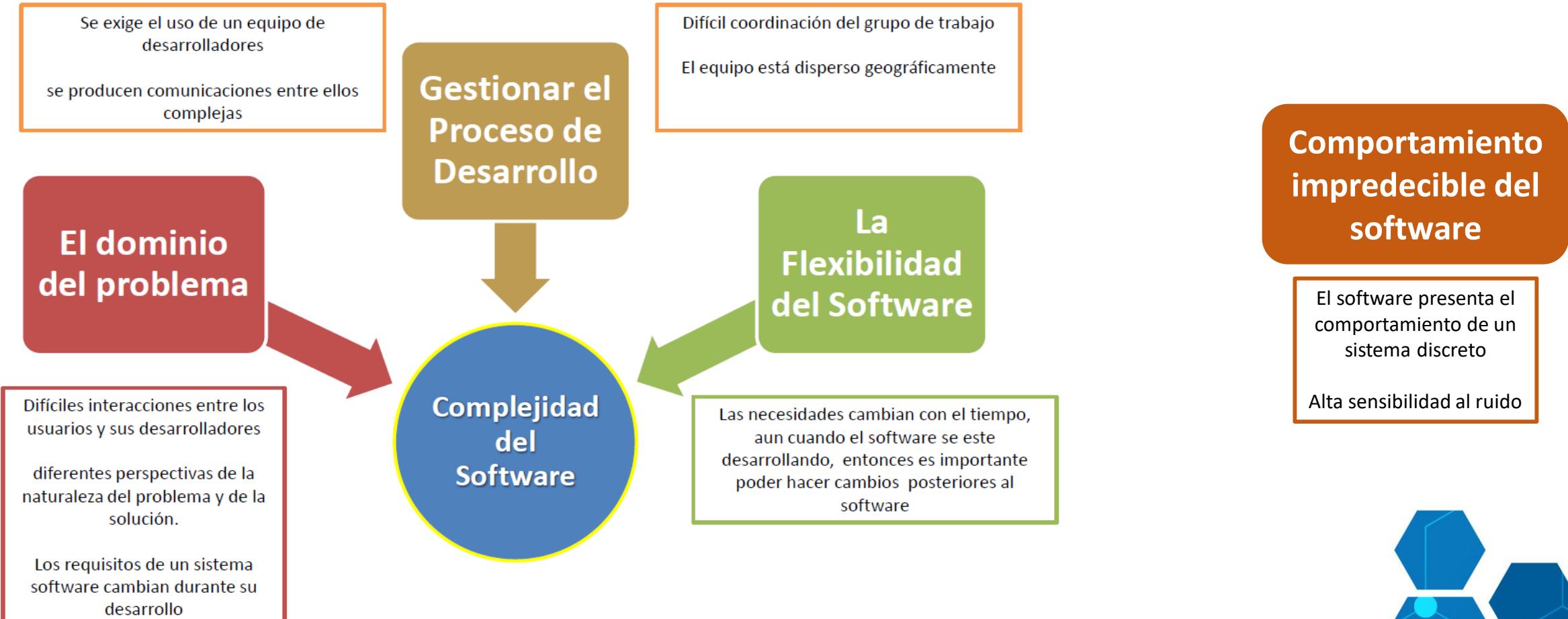
**1.4** Modelado en UML

**1.5** Principios SOLID

**1.6** Principios GRASP



# La complejidad del software



# Herramientas para tratar la complejidad

## Descomponer

Descomposición  
algorítmica

Descomposición en  
objetos

## Abstraer

Construir un modelo  
simplificado de la  
realidad

## Jerarquizar

Ordenar los elementos  
de un sistema



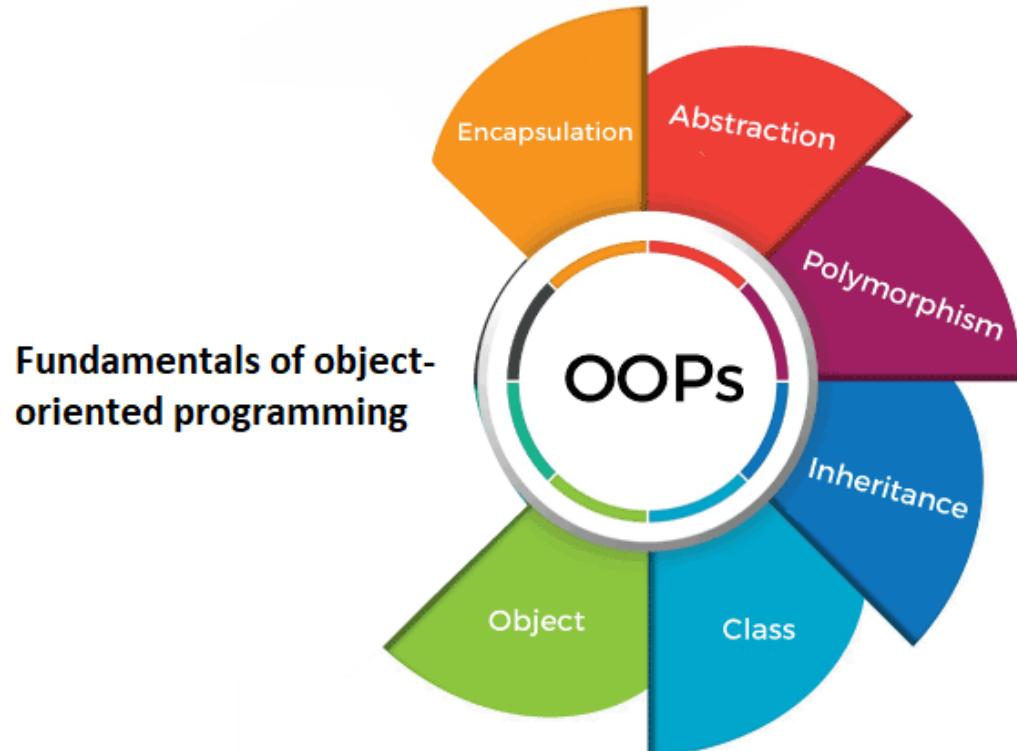
## Factores de calidad del software

- **Funcionalidad:** Capacidad del software para satisfacer los requisitos especificados.
- **Confiabilidad:** Capacidad del software para mantener su nivel de rendimiento bajo condiciones específicas y durante un período de tiempo determinado.
- **Usabilidad:** Capacidad del software para ser entendido, aprendido y utilizado de manera efectiva y eficiente por los usuarios.
- **Eficiencia:** Capacidad del software para realizar las funciones requeridas con la menor cantidad de recursos posibles.
- **Mantenibilidad:** Capacidad del software para ser modificado y mejorado de manera efectiva y eficiente.
- **Portabilidad:** Capacidad del software para ser transferido de un entorno a otro.

La norma ISO  
9126



# Programación orientada a objetos - POO

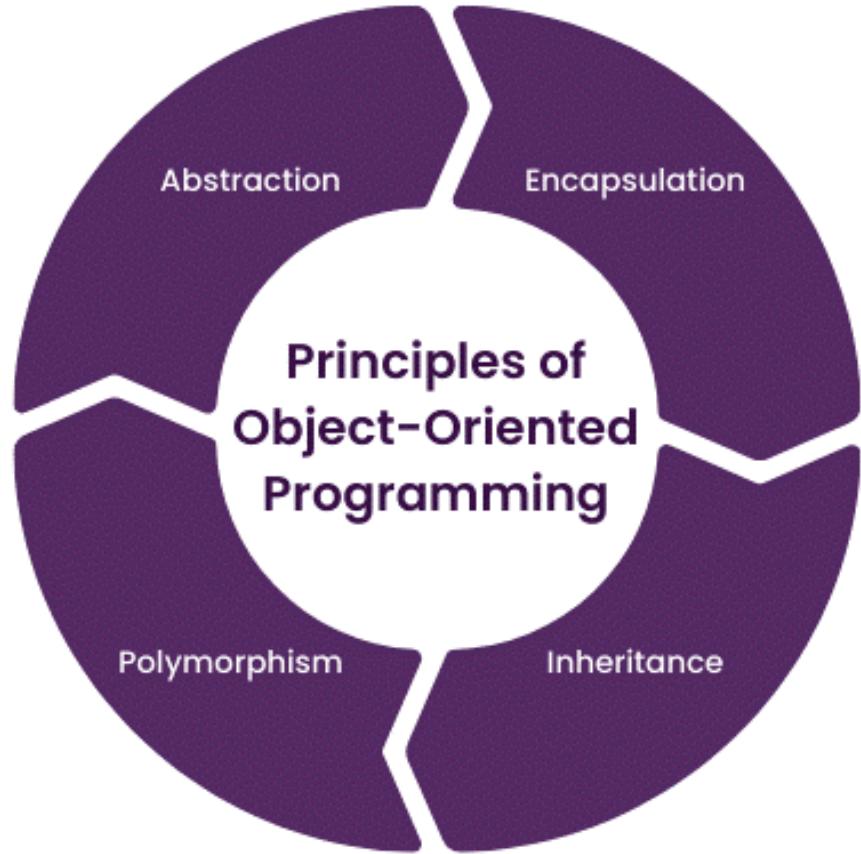


**La programación orientada a objetos - POO** es un paradigma de programación en el cual los programas expresan en un conjunto de objetos que colaboran entre si para resolver un problema.

<https://www.linkedin.com/pulse/fundamentals-object-oriented-programming-usman-malik-gd0gf>



# Principios fundamentales de la POO



<https://www.theknowledgeacademy.com/blog/principles-of-object-oriented-programming/>



**La Abstracción** es el modelo de un objeto o fenómeno del mundo real, limitado a un contexto específico, que representa todos los datos relevantes a este contexto con gran precisión, omitiendo el resto.

**La encapsulación** es la capacidad que tiene un objeto de esconder partes de su estado y comportamiento de otros objetos, exponiendo únicamente una interfaz.

**La herencia** es la capacidad de crear nuevas clases sobre otras existentes. La principal ventaja de la herencia es la reutilización de código.

**El polimorfismo** es la capacidad de un objeto para dar diferentes respuestas a un mismo mensaje

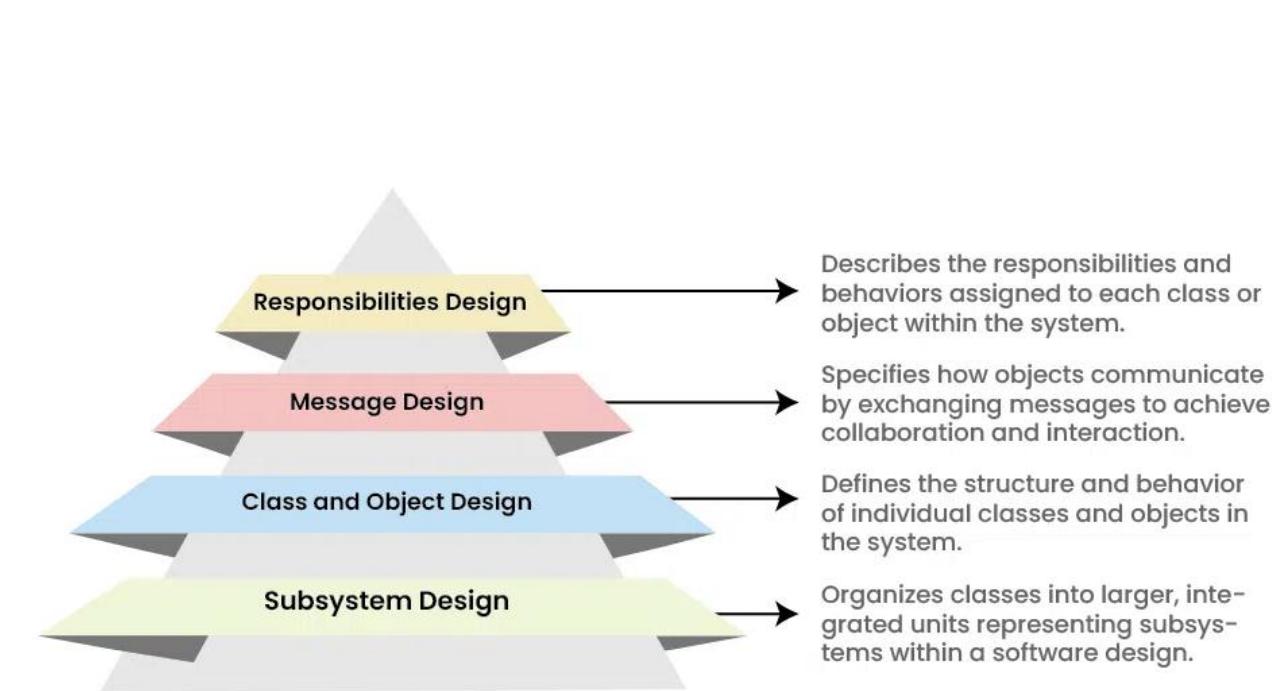
# Análisis y Diseño Orientado a Objetos - OOAD

El **análisis orientado a objetos (OOA)** es el proceso de comprender y analizar los requisitos del sistema observando el escenario del problema en términos de objeto:

- Identificar los objetos, sus atributos, comportamientos y relaciones

**OOD** es la implementación del modelo conceptual desarrollado en OOA. Transforma el modelo de análisis en un modelo de diseño detallado.

- Determinar los tipos de datos que contendrá cada objeto y cómo se relacionan entre sí.
- Descripción de procedimiento para cada operación que un objeto puede realizar



# Análisis y Diseño Orientado a Objetos - OOAD

## BENEFICIOS

Modularidad y la capacidad de mantenimiento

Representación abstracta de alto nivel de un sistema de software

Principios de diseño orientado a objetos y la reutilización de objetos

Trabajo colaborativo

software escalables

## DESAFIOS

Complicar un sistema de software

sobrecarga adicional y reducir la velocidad del software

curva de aprendizaje desafiante

proceso largo con mucha planificación

Mayor costo que otras metodologías



# Aspectos importantes para OOAD

Este enfoque facilita la gestión, la reutilización y el crecimiento del software



# Programación con Java

## Sintaxis básica

1. Variables y tipos de datos
2. Operadores y expresiones
3. Estructuras de control
4. Métodos
5. Arreglos



## POO en Java

1. Clases y objetos
2. Clases de api de Java
3. Asociación, agregación y composición
4. Herencia y polimorfismo
5. Interfaces
6. Colecciones
7. Lambdas
8. Record



# Programación con Java – Programación O.O

## Clase

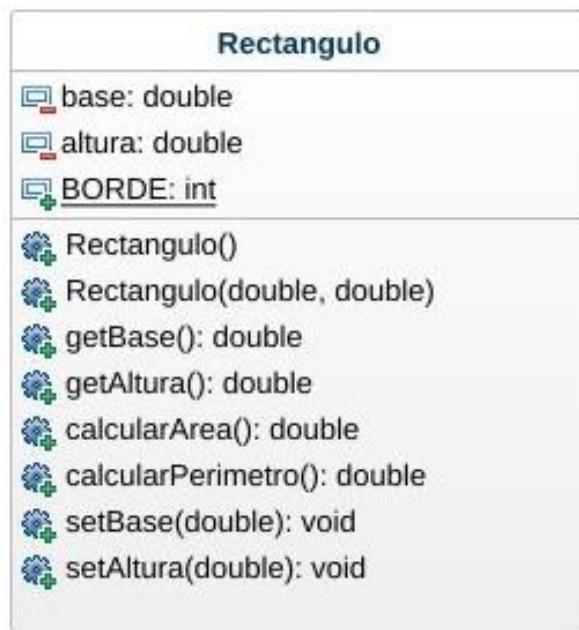


Diagrama de clase UML

Nombre de clase

Atributos

Métodos

```

3   public class Rectangulo {
4
5     private double base;
6     private double altura;
7     private static final int BORDE = 10;
8
9     public Rectangulo() { }
10
11    public Rectangulo(double base, double altura) {
12      this.base = base;
13      this.altura = altura;
14    }
15
16    public double getBase() { return base; }
17    public void setBase(double base) {this.base = base; }
18
19    public double getAltura() {return altura; }
20
21    public void setAltura(double altura) { this.altura = altura; }
22
23    public double calcularArea(){
24      return this.base*this.altura;
25    }
26
27    public double calcularPerimetro(){
28      return this.base*2 + this.altura*2;
29    }
30
31  }

```

public class [name]{  
}



# Programación con Java – Programación O.O

```
3  public class Rectangulo {  
4  
5      private double base;  
6      private double altura;  
7      private static final int BORDE = 10;  
8  
9      public Rectangulo() { }  
10  
11     public Rectangulo(double base, double altura) {  
12         this.base = base;  
13         this.altura = altura;  
14     }  
15  
16     public double getBase() { return base; }  
17     public void setBase(double base) {this.base = base;}  
18  
19     public double getAltura() {return altura;}  
20  
21     public void setAltura(double altura) { this.altura = altura;}  
22  
23     public double calcularArea(){  
24         return this.base*this.altura;  
25     }  
26  
27     public double calcularPerimetro(){  
28         return this.base*2 + this.altura*2;  
29     }  
30 }  
31 }
```

## Objetos y Constructores

**Rectangulo r = new Rectangulo( );**

**r.setBase(20);**

**r.setAltura(40);**

**System.out.println( r.getAltura() );**

**System.out.println( r.getBase() );**

**System.out.println( r.calcularArea() );**

**System.out.println( r.calcularPerimetro() );**

**System.out.println( Rectangulo.BORDE );**

# Programación con Java – Programación O.O

```
3  public class Rectangulo {  
4  
5      private double base;  
6      private double altura;  
7      private static final int BORDE = 10;  
8  
9      public Rectangulo() { }  
10  
11     public Rectangulo(double base, double altura) {  
12         this.base = base;  
13         this.altura = altura;  
14     }  
15  
16     public double getBase() { return base; }  
17     public void setBase(double base) {this.base = base;}  
18  
19     public double getAltura() {return altura;}  
20  
21     public void setAltura(double altura) { this.altura = altura;}  
22  
23     public double calcularArea(){  
24         return this.base*this.altura;  
25     }  
26  
27     public double calcularPerimetro(){  
28         return this.base*2 + this.altura*2;  
29     }  
30 }  
31 }
```

## Objetos y Constructores

**Rectangulo r = new Rectangulo( 20,10 );**

**System.out.println( r.getAltura() );**

**System.out.println( r.getBase() );**

**System.out.println( r.calcularArea() );**

**System.out.println( r.calcularPerimetro() );**

**System.out.println( Rectangulo.BORDE );**

# Programación con Java – Programación O.O

```
3   public class Rectangulo {  
4  
5       private double base;  
6       private double altura;  
7       private static final int BORDE = 10;  
8  
9       public Rectangulo() { }  
10  
11      public Rectangulo(double base, double altura) {  
12          this.base = base;  
13          this.altura = altura;  
14      }  
15  
16      public double getBase() { return base; }  
17      public void setBase(double base) {this.base = base;}  
18  
19      public double getAltura() {return altura;}  
20  
21      public void setAltura(double altura) { this.altura = altura;}  
22  
23      public double calcularArea(){  
24          return this.base*this.altura;  
25      }  
26  
27      public double calcularPerimetro(){  
28          return this.base*2 + this.altura*2;  
29      }  
30  
31  }
```

## Atributos de instancia

```
private double base ;  
private double altura;
```

```
Rectangulo r = new Rectangulo( );
```

```
System.out.println( r.getAltura() );
```

```
System.out.println( r.getBase() );
```

```
Rectangulo c = new Rectangulo( );
```

```
System.out.println( c.calcularPerimetro() );
```

```
System.out.println( c.getBase() );
```

# Programación con Java – Programación O.O

```
3  public class Rectangulo {  
4  
5      private double base;  
6      private double altura;  
7      private static final int BORDE = 10;  
8  
9      public Rectangulo() { }  
10  
11     public Rectangulo(double base, double altura) {  
12         this.base = base;  
13         this.altura = altura;  
14     }  
15  
16     public double getBase() { return base; }  
17     public void setBase(double base) {this.base = base;}  
18  
19     public double getAltura() {return altura;}  
20  
21     public void setAltura(double altura) { this.altura = altura;}  
22  
23     public double calcularArea(){  
24         return this.base*this.altura;  
25     }  
26  
27     public double calcularPerimetro(){  
28         return this.base*2 + this.altura*2;  
29     }  
30  
31 }
```

## Atributos de clase (static)

**public static final int BORDE = 10;**

**System.out.println( Rectangulo.BORDE );**

**private static int BORDE = 10;**

**public static int getBorde(){  
 return Rectangulo.BORDE;  
}**

**public static void setBorde(int borde){  
 Rectangulo.BORDE = borde;  
}**

**System.out.println( Rectangulo.getBorde() );**

# Programación con Java – Programación O.O

```
3  public class Rectangulo {  
4  
5      private double base;  
6      private double altura;  
7      private static final int BORDE = 10;  
8  
9      public Rectangulo() { }  
10  
11     public Rectangulo(double base, double altura) {  
12         this.base = base;  
13         this.altura = altura;  
14     }  
15  
16     public double getBase() { return base; }  
17     public void setBase(double base) {this.base = base;}  
18  
19     public double getAltura() {return altura;}  
20  
21     public void setAltura(double altura) { this.altura = altura;}  
22  
23     public double calcularArea(){  
24         return this.base*this.altura;  
25     }  
26  
27     public double calcularPerimetro(){  
28         return this.base*2 + this.altura*2;  
29     }  
30  
31 }
```

## Sobrecarga de métodos (overload)

```
public Rectangulo ( ){  
    }  
}
```

```
Rectangulo r = new Rectangulo();
```

```
public Rectangulo (double base, double altura){  
    this.base = base;  
    this.altura = altura;  
}
```

```
Rectangulo p = new Rectangulo(10,5);
```

# Programación con Java – Programación O.O

```
3  public class Rectangulo {  
4  
5      private double base;  
6      private double altura;  
7      private static final int BORDE = 10;  
8  
9      public Rectangulo() { }  
10  
11     public Rectangulo(double base, double altura) {  
12         this.base = base;  
13         this.altura = altura;  
14     }  
15  
16     public double getBase() { return base; }  
17     public void setBase(double base) {this.base = base;}  
18  
19     public double getAltura() {return altura;}  
20  
21     public void setAltura(double altura) { this.altura = altura;}  
22  
23     public double calcularArea(){  
24         return this.base*this.altura;  
25     }  
26  
27     public double calcularPerimetro(){  
28         return this.base*2 + this.altura*2;  
29     }  
30  
31 }
```

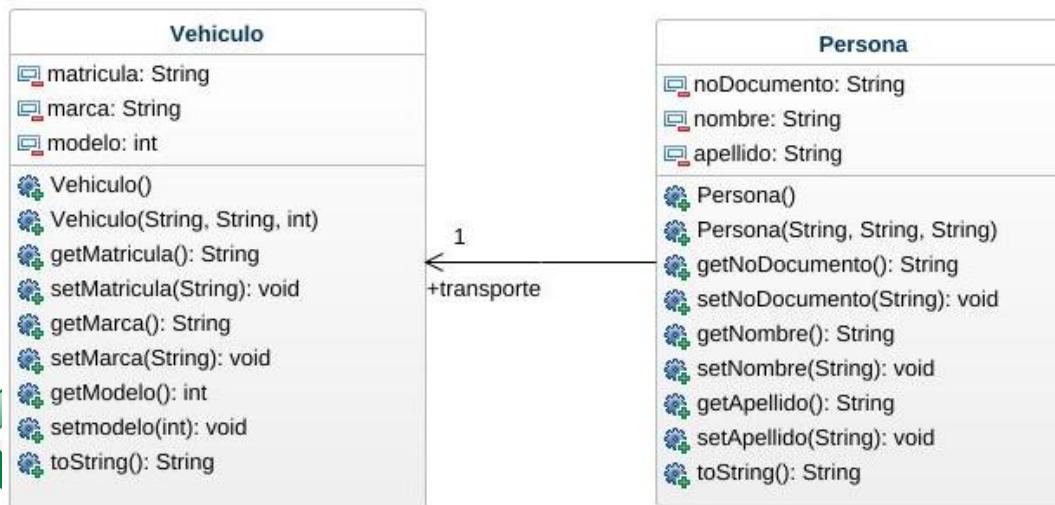
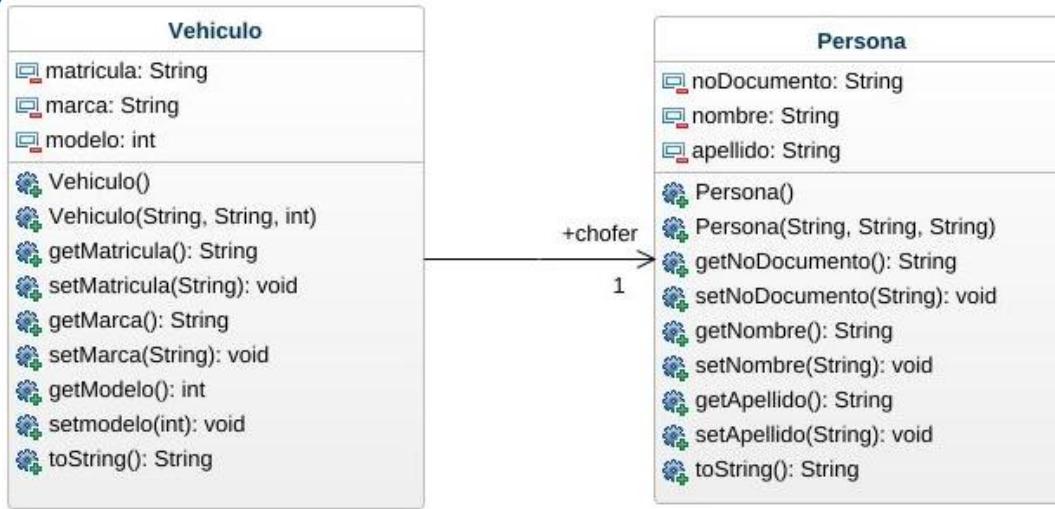
## Sobrecarga de métodos (overload)

```
public double sumaArea (Rectangulo r ){  
    return this.calcularArea() + r.calcularArea();  
}
```

```
public double sumaArea (double b, double a ){  
    Rectangulo r = new Rectangulo(b,a);  
    return this.sumaArea(r);  
}
```

```
public double sumaArea (double b){  
    return this.sumaArea(b,0);  
}
```

# Programación con Java – Programación O.O



## Asociación, agregación, composición

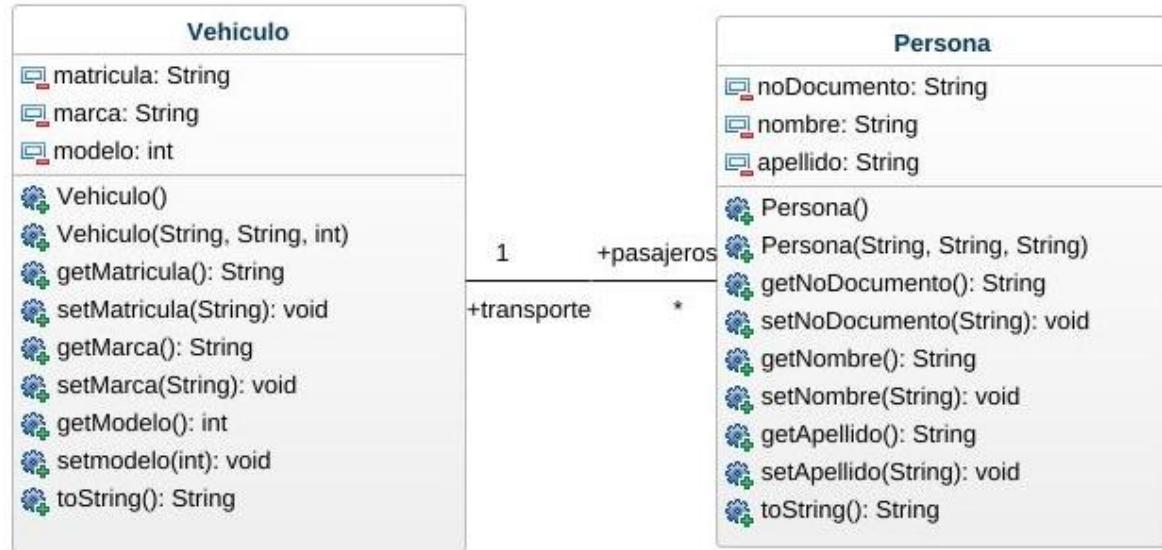
```

public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private Persona chofer;
    //
}
  
```

```

public class Persona {
    private String noDocumento;
    private String nombre;
    private String apellido;
    private Vehiculo transporte;
    //
}
  
```

# Programación con Java – Programación O.O



## Asociación, agregación, composición

```

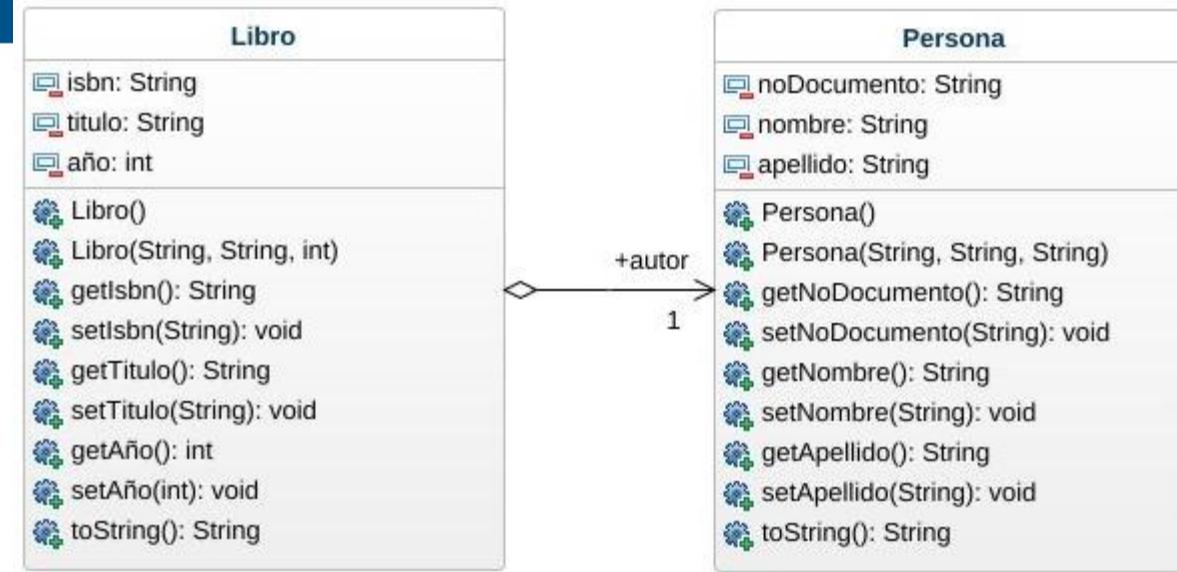
public class Vehiculo {
    private String matricula;
    private String marca;
    private String modelo;
    private Persona pasajeros[ ];
    //
}
  
```

```

public class Persona {
    private String noDocumento;
    private String nombre;
    private String apellido;
    private Vehiculo transporte;
    //
}
  
```



# Programación con Java – Programación O.O



```

public class Persona {
    private String noDocumento;
    private String nombre;
    private String apellido;
    //
}
  
```

## Asociación, agregación, composición

```

public class Libro {
    private String isbn;
    private String titulo;
    private int año;
    private Persona autor;
    //
    public Libro(String i, String t, int y, Persona a){
        //
        this.autor = a;
    }
}
  
```

# Programación con Java – Programación O.O

```
public class Libro {  
    private String isbn;  
    private String titulo;  
    private int año;  
    private Persona autor;  
    //  
    public Libro(String i, String t, int y, Persona a){  
        //  
        this.autor = a;  
    }  
  
}
```

```
public class Persona {  
    private String noDocumento;  
    private String nombre;  
    private String apellido;  
    //  
}
```

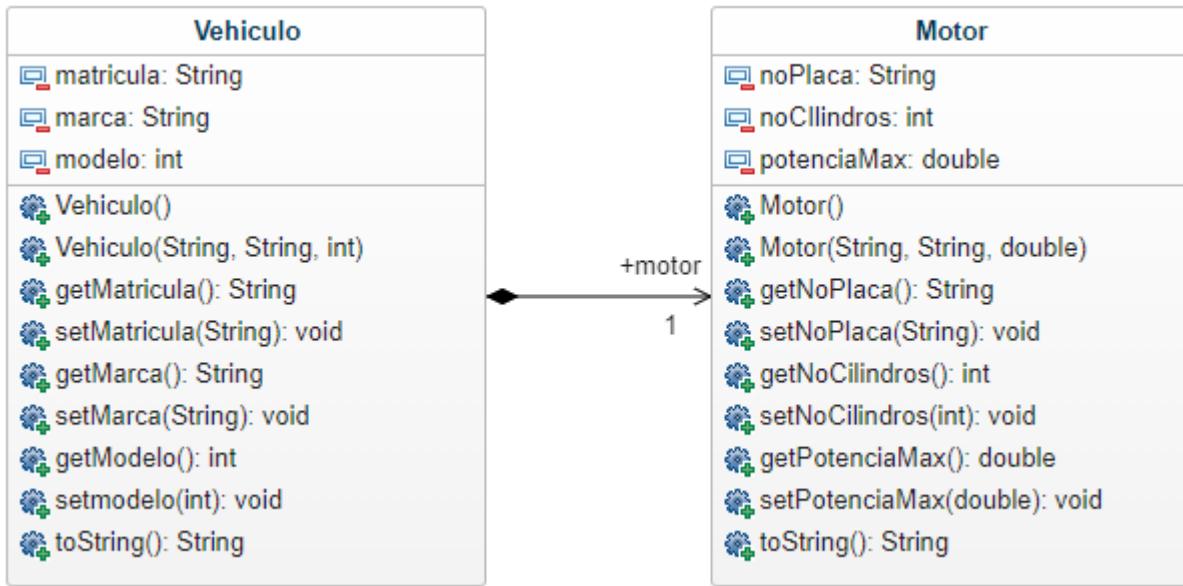
Asociación, **agregación**, composición

```
Persona p = new Persona("123", "Luis", "Perez");
```

```
Libro libroA = new Libro("xxx", "POO", 2020, p);
```

```
Libro libroB = new Libro("yyy", "PD", 2023, p);
```

# Programación con Java – Programación O.O



```

public class Motor {
    private String noPlaca;
    private int noCilindros;
    private double potenciaMax;
    //
}
  
```

## Asociación, agregación, **composición**

```

public class Vehiculo {
    private String matricula;
    private String marca;
    private int modelo;
    private Motor motor;
    //
    public Vehiculo(String mat, String marca, int mod, String nPlaca, int noCil, double pot){
        //
        this.motor = new Motor(nPlaca, noCil, pot);
    }
}
  
```

# Programación con Java – Programación O.O

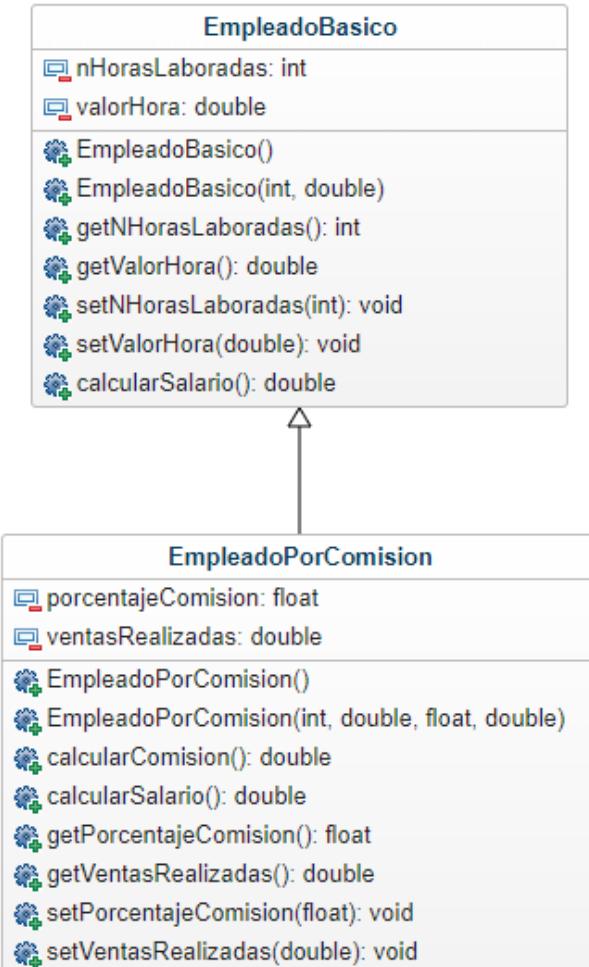
```
public class Vehiculo {  
    private String matricula;  
    private String marca;  
    private int modelo;  
    private Motor motor;  
    //  
    public Vehiculo(String mat, String marca, int  
        mod, String nPlaca, int noCil, double pot){  
        //  
        this.motor = new Motor(nPlaca, noCil, pot);  
    }  
}
```

```
public class Motor {  
    private String noPlaca;  
    private int noCilindros;  
    private double potenciaMax;  
    //  
}
```

Asociación, agregación, **composición**

```
Vehiculo v = new Vehiculo("123", "Toyota", 2020, "XBC-1", 4, 1500.8 );
```

# Programación con Java – Programación O.O



## Herencia entre Clases

```

public class EmpleadoBasico {
    private int nHorasLaboradas;
    private double valorHora;

    public EmpleadoBasico(){} }

    public EmpleadoBasico(int h, double v){
        this.nHorasLaboradas = h;
        this.valorHora = v;
    }

    // getter y setter

    public double calcularSalario(){
        return this.nHorasLaborada * this.valorHora;
    }
}
  
```

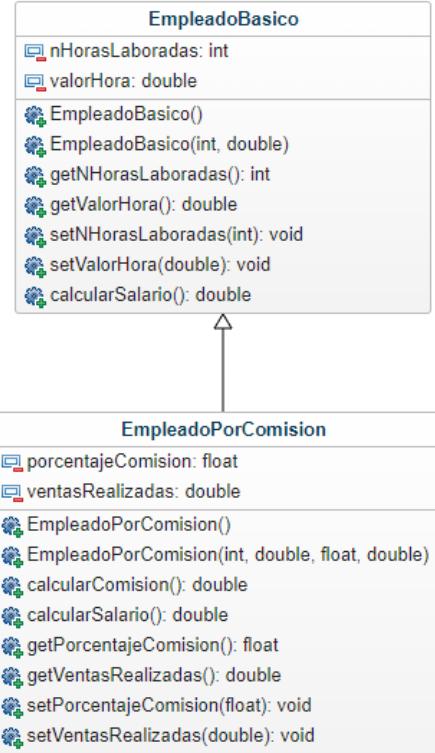
EmpleadoBasico eb = new EmpleadoBasico(40, 30000.5);

System.out.println( eb.calcularSalario() );

**Super clase**

# Programación con Java – Programación O.O

## Herencia entre Clases



```

public class EmpleadoPorComision extends EmpleadoBasico {
    private float porcentajeComision;
    private double ventasRealizadas;

    public EmpleadoPorComision() {
    }

    public EmpleadoPorComision(int h, double p, float c, double v) {
        super(h,p);
    }

    // getter y setter

    public double calcularComision(){
        return this.porcentajeComision * this.ventasRealizadas;
    }

    @Override
    public double calcularSalario(){
        return this.calcularComision() + super.calcularSalario();
    }
}
  
```

```

public class EmpleadoBasico {
    private int nHorasLaboradas;
    private double valorHora;

    public EmpleadoBasico(){}
}

public EmpleadoBasico(int h, double v){
    this.nHorasLaboradas = h;
    this.valorHora = v;
}

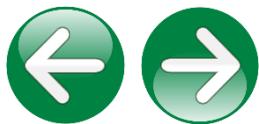
// getter y setter

public double calcularSalario(){
    return this.nHorasLaborada * this.valorHora;
}
  
```

EmpleadoPorComision ec;  
 ec = new EmpleadoPorComision(40, 30000.5, 0.1,7000000 );

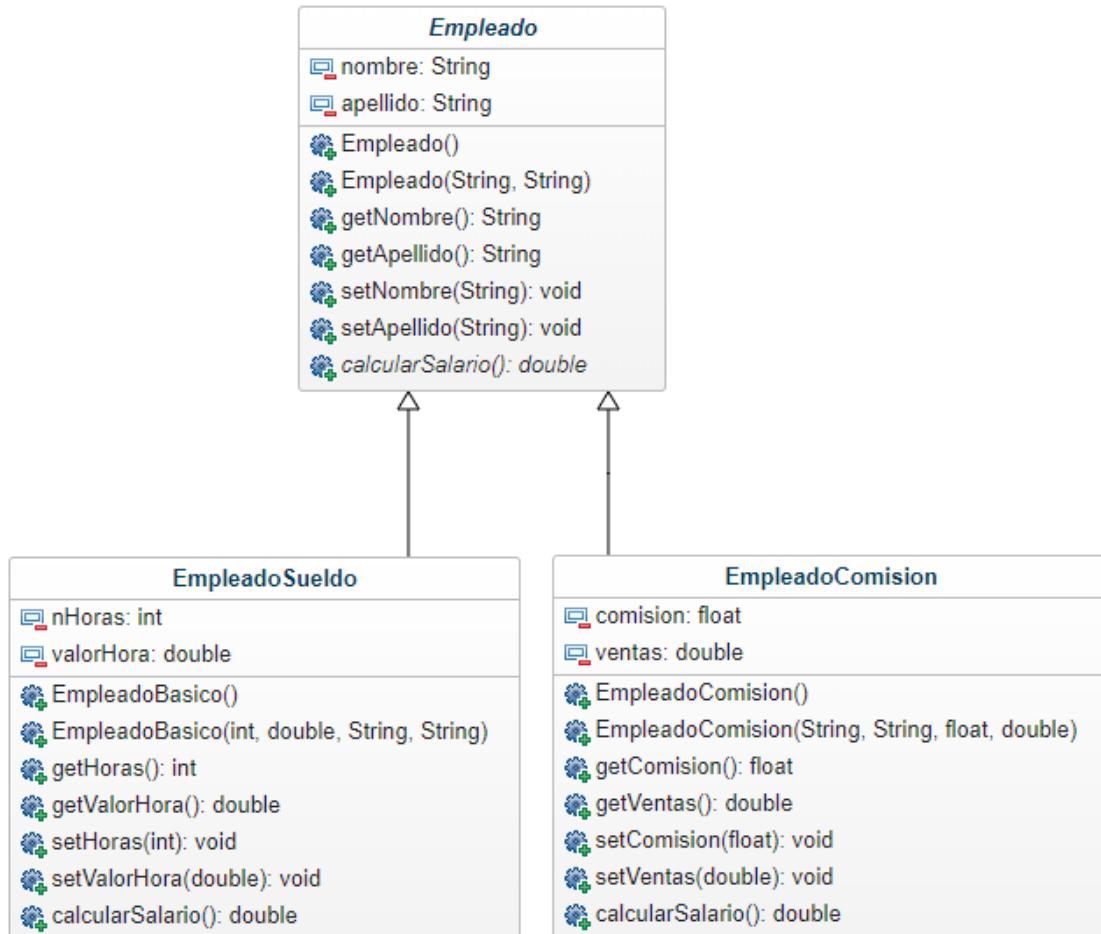
System.out.println( ec.calcularSalario() );

Clase derivada



# Programación con Java – Programación O.O

## Clases abstractas y Polimorfismo



```

public abstract class Empleado {
    private String nombre;
    private String apellido;

    public Empleado (){}

    public Empleado ( String nombre, String apellido){
        this.nombre = nombre;
        this.apellido = apellido;
    }

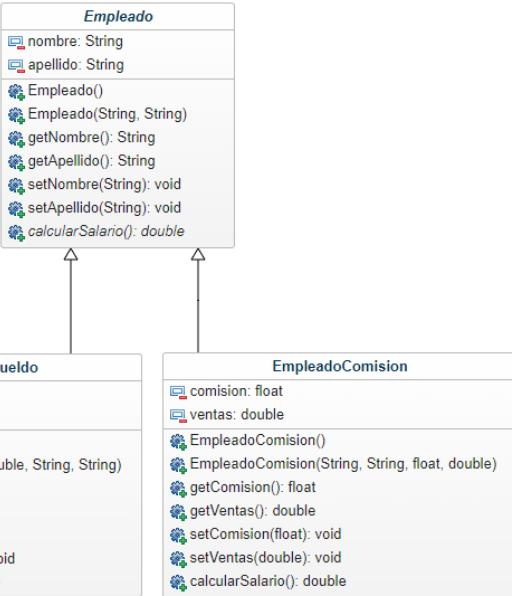
    public String getNombre(){
        return this.nombre;
    }
    // getter y setter

    public abstract double calcularSalario();
}
  
```

## Superclase Abstracta

# Programación con Java – Programación O.O

## Clases abstractas y Polimorfismo



```

public class EmpleadoSueldo extends Empleado {
    private int nHoras;
    private double valorHora;

    public EmpleadoSueldo(){} 
    public EmpleadoSueldo(int h, double v, String n, String a){
        super(n,a);
        this.nHoras = h;
        this.valorHora = v;
    }

    // getter y setter
    @Override
    public double calcularSalario(){
        return this.nHoras * this.valorHora;
    }
}
  
```

Clase derivada

```

public abstract class Empleado {
    private String nombre;
    private String apellido;

    public Empleado(){}
    public Empleado (String nombre, String apellido){
        this.nombre = nombre;
        this.apellido = apellido;
    }

    public String getNombre(){
        return this.nombre;
    }
    // getter y setter

    public abstract double calcularSalario();
}
  
```

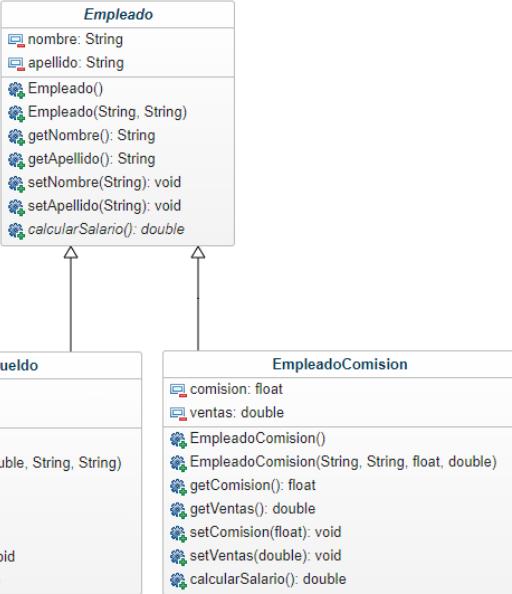
Empleado e = new EmpleadoSueldo(40, 30000.5, "Pedro", "Paz");

System.out.println( e.calcularSalario() );



# Programación con Java – Programación O.O

## Clases abstractas y Polimorfismo



```

public class EmpleadoComision extends Empleado {
    private float comision;
    private double ventas;

    public EmpleadoComision( ){ }
    public EmpleadoComision(float c, double v, String n, String a){
        super(n,a);
        this.comision = c;
        this.ventas = v;
    }

    // getter y setter
    @Override
    public double calcularSalario(){
        return this.nHoras * this.valorHora;
    }
}
  
```

Empleado e = new EmpleadoComision(40, 30000.5, "Pedro", "Paz");

System.out.println( e.calcularSalario() );

Clase derivada

```

public abstract class Empleado {
    private String nombre;
    private String apellido;

    public Empleado ( ){
    }
    public Empleado ( String nombre, String apellido){
        this.nombre = nombre;
        this.apellido = apellido;
    }

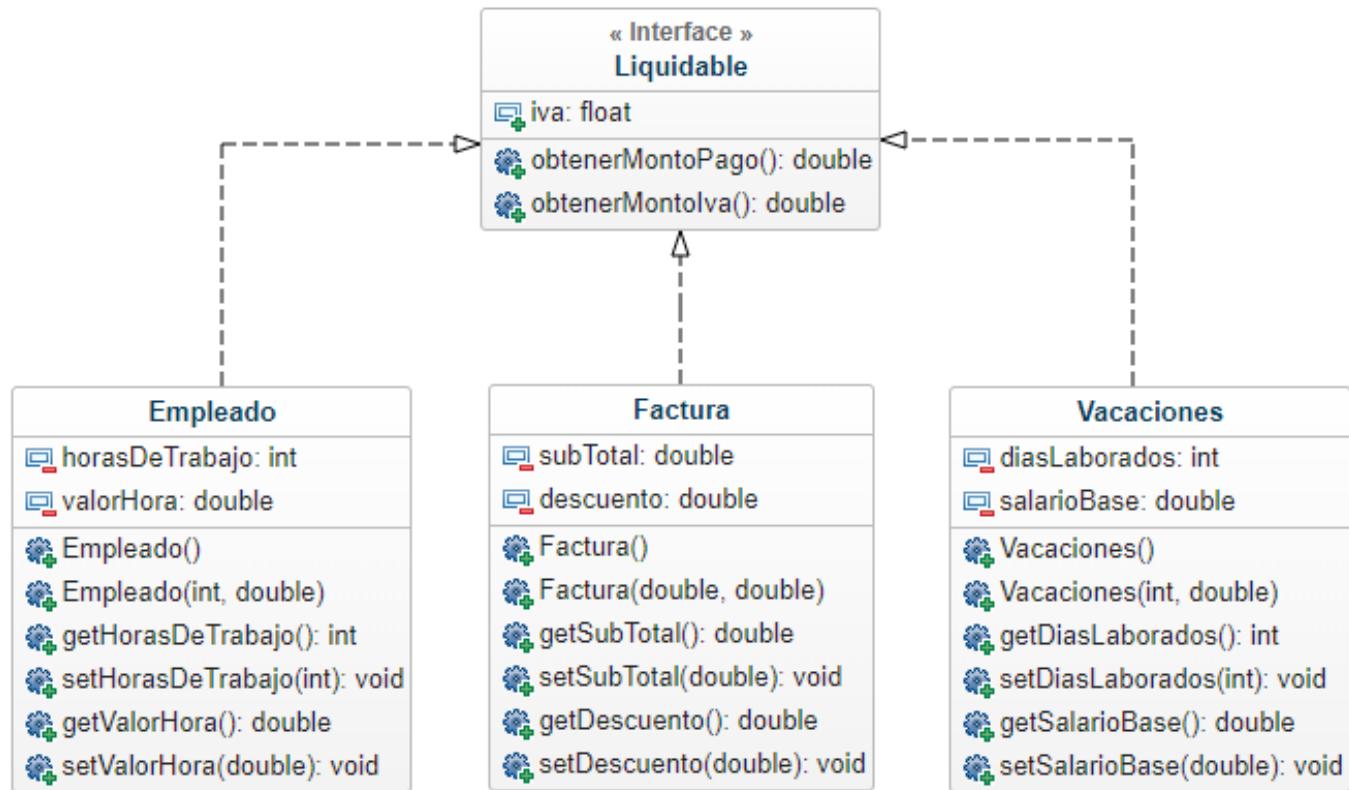
    public String getNombre(){
        return this.nombre;
    }
    // getter y setter

    public abstract double calcularSalario();
}
  
```



# Programación con Java – Programación O.O

## Interfaces y Polimorfismo



```
public interface Liquidable {
```

```
    float IVA=0.16;
```

```
    double obtenerMontoPago();
```

```
    double obtenerMontolva();
```

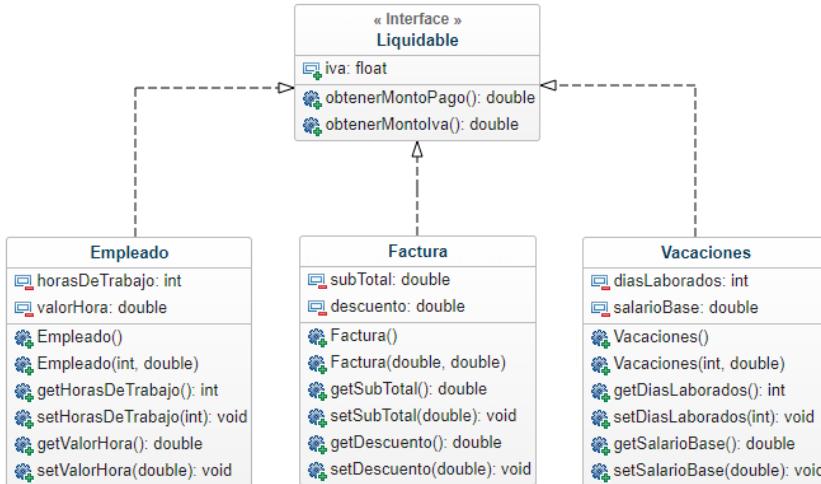
```
}
```

**Liquidable objetoLiquidable;**

**Interface**

# Programación con Java – Programación O.O

## Interfaces y Polimorfismo



```

public interface Liquidable {

    float IVA=0.16;
    double obtenerMontoPago();
    double obtenerMontolva();

}
  
```

```

public class Empleado implements Liquidable {
    private int horasDeTrabajo;
    private double valorHora;
  
```

```

    public Empleado () {
    }
    public Empleado ( int h, int v){
        this.horasDeTrabajo = h;
        this.valorHora = v;
    }
  
```

```

    // getter y setter
    @Override
    public double obtenerMontoPago(){
        return this.nHoras * this.valorHora;
    }
  
```

```

    @Override
    public double obtenerMontolva(){
        return Liquidable.IVA * this.obtenerMontoPago();
    }
  
```

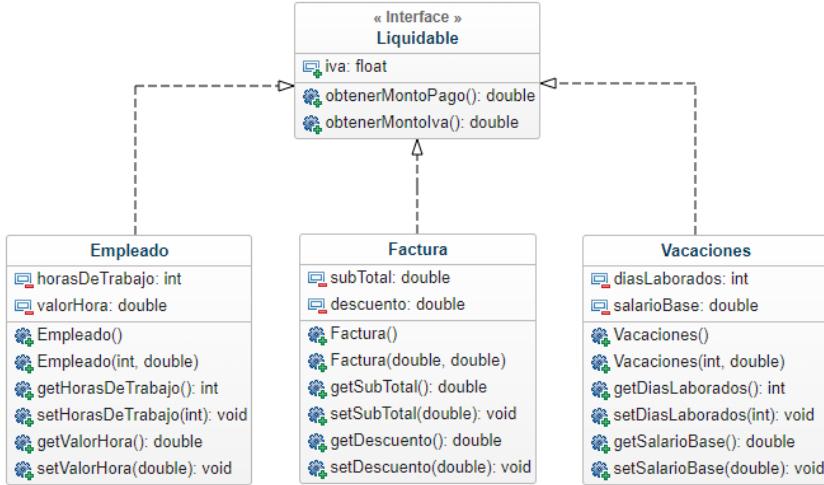
```

Liquidable objeto=new Empleado(8, 12000);
System.out.println( objeto.obtenerMontoPago() );
System.out.println( objeto.obtenerMontolva() );
  
```

**Implementación de interface**

# Programación con Java – Programación O.O

## Interfaces y Polimorfismo



```

public interface Liquidable {

    float IVA=0.16;
    double obtenerMontoPago();
    double obtenerMontolva();

}
  
```

```

public class Factura implements Liquidable {
    private double subTotal;
    private double descuento;
  
```

```

    public Factura (){
    }
  
```

```

    public Factura ( double s, double d){
        this.subTotal = s;
        this.descuento = d;
    }
    // getter y setter
  
```

```

    @Override
  
```

```

    public double obtenerMontoPago(){
        return this.subTotal - this.descuento;
    }
  
```

```

    @Override
  
```

```

    public double obtenerMontolva(){
        //
    }
  
```

```

Liquidable objeto=new Factura(1000, 100);
  
```

```

System.out.println( objeto.obtenerMontoPago() );
  
```

```

System.out.println( objeto.obtenerMontolva() );
  
```

Implementación de interface

# Programación con Java – Programación O.O

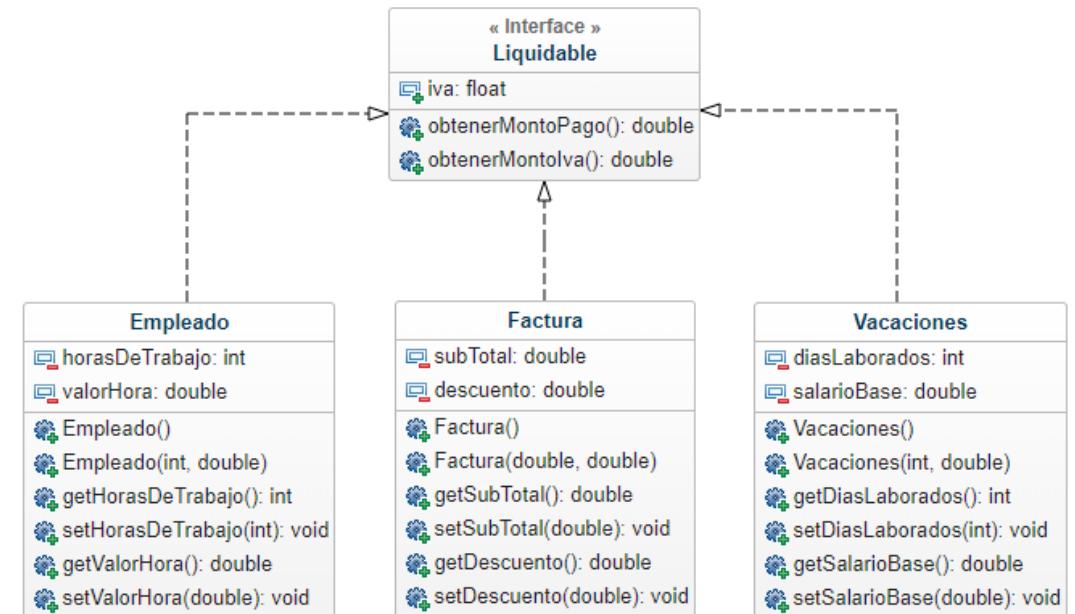
## Código genérico

```

public class Main {
    public static void main(String[] arg){
        List<Liquidable> list = new ArrayList();
        list.add(new Empleado(30, 5000));
        list.add(new Factura(30000,5000));
        ...
        verValorLiquidado(list);
    }

    public static double verValorLiquidado(List<Liquidable> list){
        for(Liquidable l : list){
            System.out.println("Valor= " + l.obtenerMontoPago() );
            System.out.println("Iva= " + l.obtenerMontolva() );
        }
    }
}

```



# Programación con Java – Programación O.O

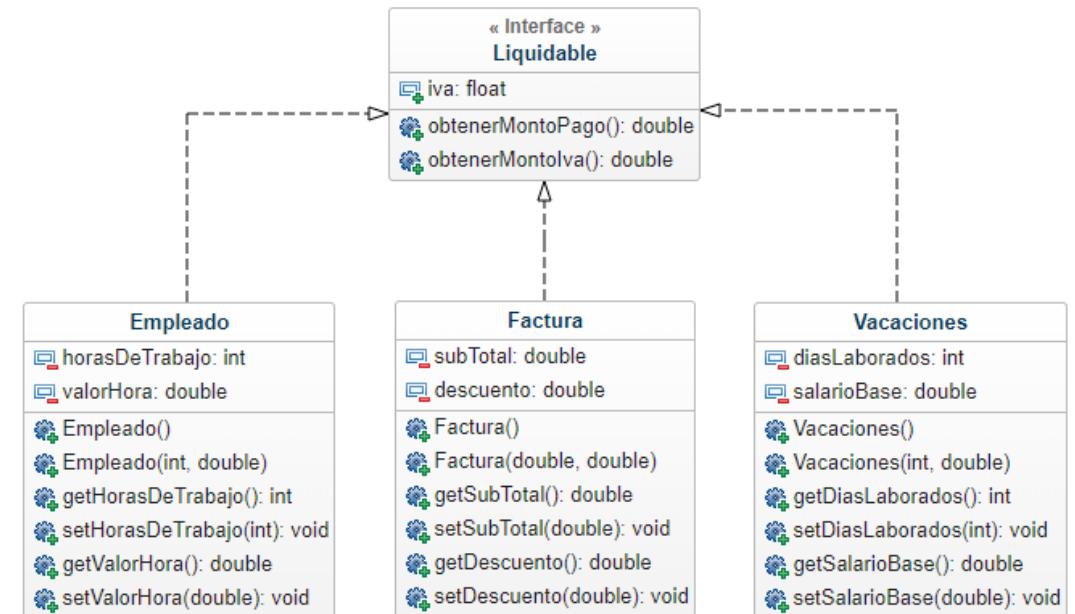
## Código genérico

```

public class Main {
    public static void main(String[] arg){
        List<Liquidable> list = new ArrayList();
        list.add(new Empleado(30, 5000));
        list.add(new Factura(30000,5000));
        ...
        verValorLiquidado(list);
    }

    public static double verValorLiquidado(List<Liquidable> list){
        for(Liquidable l : list){
            System.out.println("Valor= " + l.obtenerMontoPago());
            System.out.println("Iva= " + l.obtenerMontolva());
        }
    }
}

```



# Programación con Java – Programación O.O

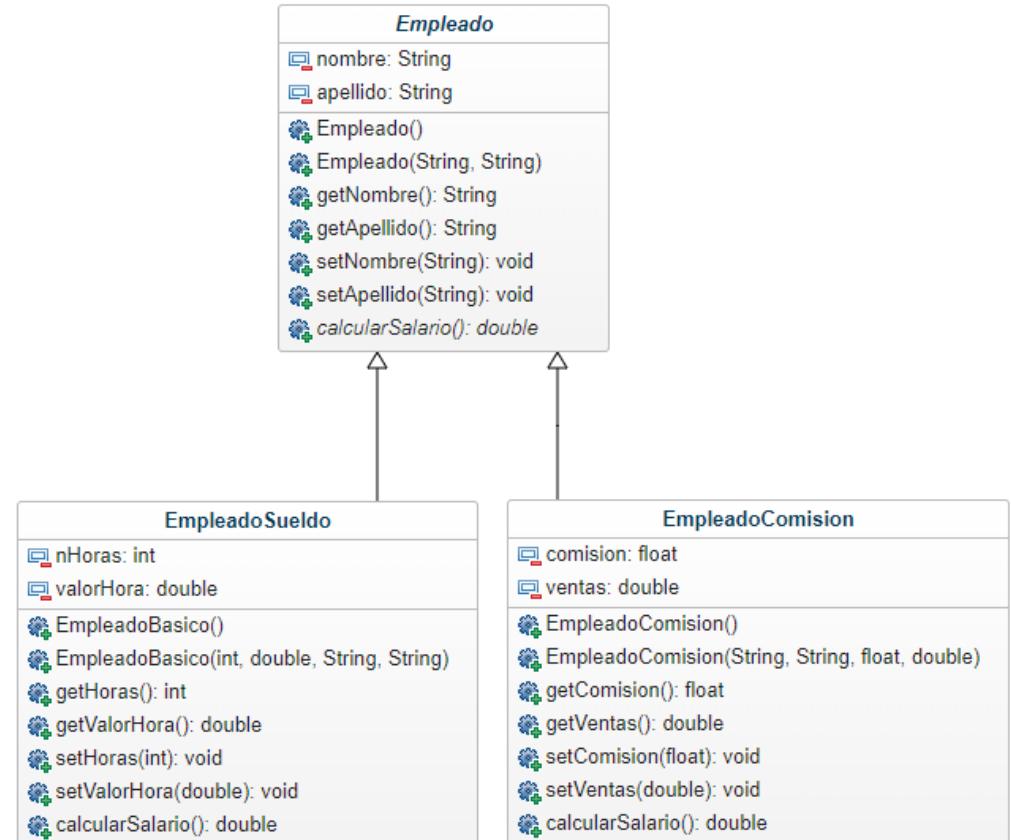
## Código genérico

```

public class Main {
    public static void main(String[] arg){
        List<Empleado> list = new ArrayList();
        list.add(new EmpleadoSueldo(30, 5000, "Pipe", "Paz" ));
        list.add(new EmpleadoComision(0.3,100000, "Luis", "Diaz") );
        ...
        verNomina(list);
    }

    public static double verNomina(List<Empleado> list){
        for(Empleado e : list){
            System.out.println("Nombre= " + e.getName() );
            System.out.println("Apellido= " + e.getApellido() );
            System.out.println("Salario= " + e.calcularSalario() );
        }
    }
}

```



# Programación con Java – Programación O.O

## Colecciones - Listas

ArrayList

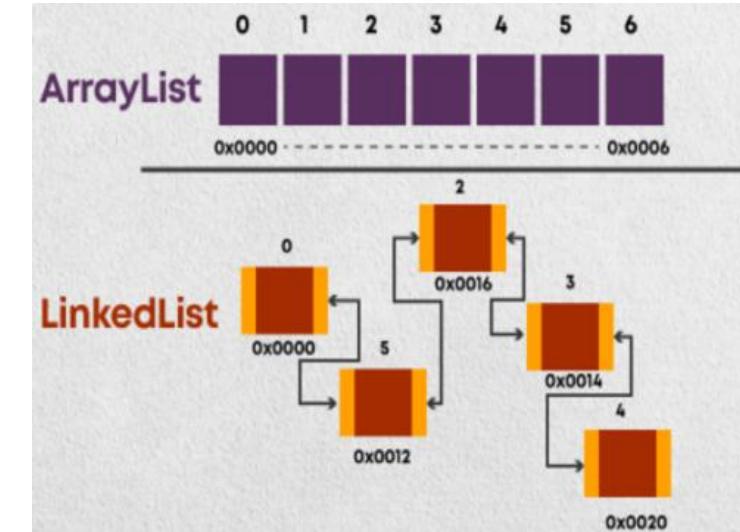
```
List <String> palabras=new ArrayList( );
```

LinkedList

```
List <String> palabras=new LinkedList( );
```

### Métodos :

- Object get (int index)**
- Object set (int index, Object element)**
- void add (int index, Object element)**
- Object remove (int index)**



Operations	ArrayList	LinkedList
	Complexity	Complexity
get(int index)	O(1)	O(n/4)
add(E element)	O(1)->O(n)	O(1)
remove(int index)	O(n/2)	O(n/4)
add(int index, E element)	O(n/2)	O(n/4)
Iterator.remove()	O(n/2)	O(1)
ListIterator.add(E element)	O(n/2)	O(1)



# Programación con Java – Programación O.O

## Colecciones - Mapas

HashMap

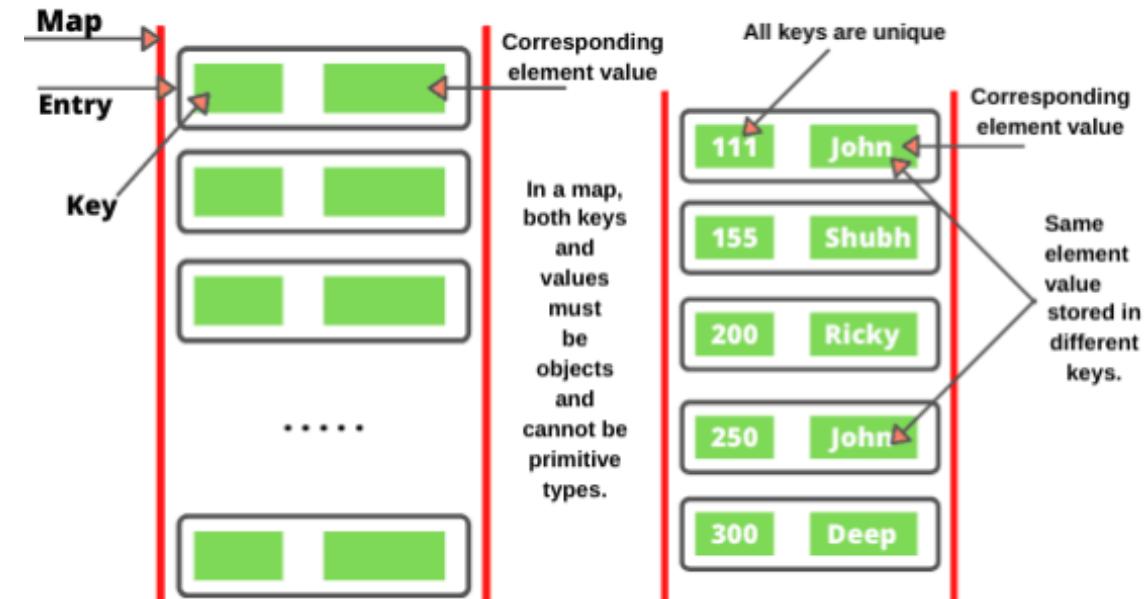
```
Map <int,String> palabras=new HashMap( );
```

TreeMap

```
Map <int,String> palabras=new TreeMap( );
```

Métodos :

- ✓ **put(K key, V value)**
- ✓ **remove(Object key)**
- ✓ **get(Object key)**
- int size()**



[https://www.scientecheeasy.com/2020/10/map-in-java.html/](https://www.scientecheasy.com/2020/10/map-in-java.html/)



# Programación con Java – Programación O.O

## Colecciones - Conjuntos

HashSet

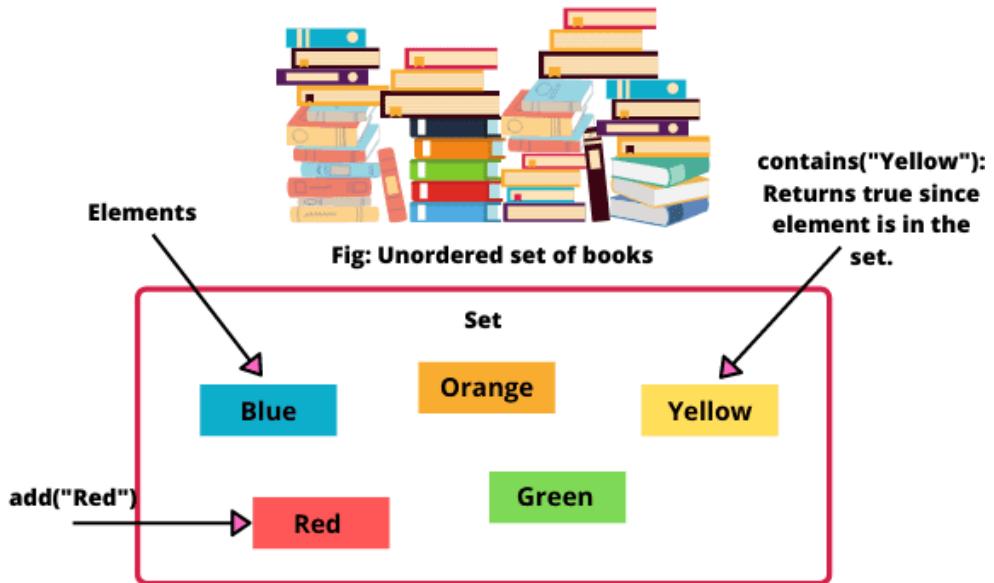
```
Set <String> palabras=new HashSet( );
```

TreeSet

```
Map <String> palabras=new TreeSet( );
```

Métodos :

- boolean add(Object o)
- boolean remove(Object o)
- boolean isEmpty()
- int size()



<https://www.scientecheeasy.com/2020/10/map-in-java.html/>

# Programación con Java – Programación O.O

## Colecciones – API Java

### Clases utilitarias

- String
- Scanner
- LocalDate
- LocalDateTime
- Random
- Math
- Calendar

### Arreglos y Colecciones

- Collections
- Arrays

### Interfaces

- Serializable
- Comparable
- Cloneable
- Comparator

### Avanzado

- Lambdas
- Streams
- Records



# Abstracción y Encapsulamiento

La **abstracción** es la capacidad de identificar las características y comportamientos esenciales de un objeto, ignorando los detalles irrelevantes.

- Abstracción de datos
- Abstracción de procesos
- Abstracción conceptual

## Niveles de abstracción

## Técnicas avanzadas

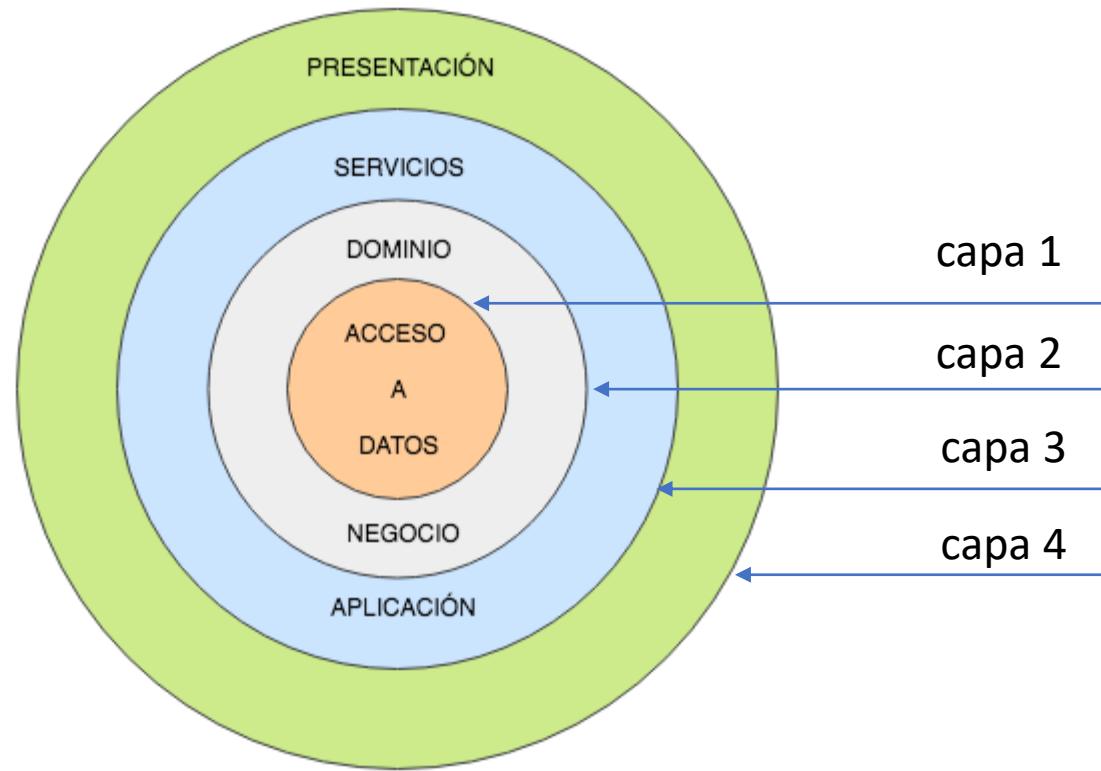
- Abstracción por especialización
- Abstracción por generalización
- Abstracción por análisis

- múltiples capas de abstracción

## Jerarquías de abstracción



# Abstracción y Encapsulamiento



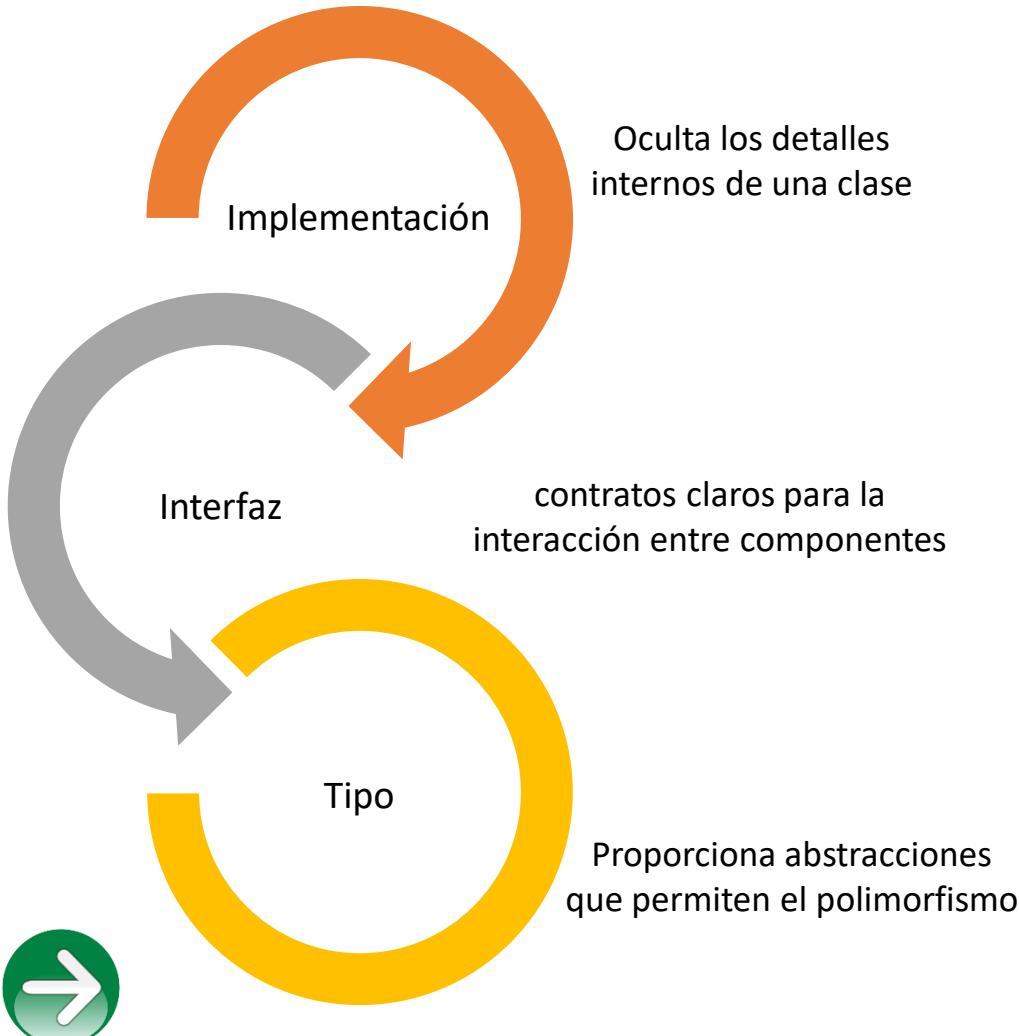
Las jerarquías de abstracción permiten manejar la complejidad a través de diferentes niveles de detalle.

Cada capa proporciona una vista coherente que oculta los detalles de las capas inferiores.



# Abstracción y Encapsulamiento

## NIVELES DE ENCAPSULAMIENTO



El **encapsulamiento** es el mecanismo que combina datos y comportamientos en una única unidad y restringe el acceso directo a algunos de los componentes del objeto

## Patrones Avanzados de Encapsulamiento

Tell, Don't Ask

Ley de  
Deméter

ENCAPSULA LO QUE  
VARIA

# Abstracción y Encapsulamiento

```
// Enfoque básico con violación de encapsulamiento
class UserBasic {
    public String name;
    public String email;
    public String password;
}
```

## BENEFICIOS ESTRATÉGICOS

Seguridad mejorada

Mantenibilidad superior

Reutilización estratégica

Evolución flexible

```
// Enfoque avanzado con encapsulamiento adecuado
class User {
    private String name;
    private Email email; // Tipo de valor encapsulado
    private Password password; // Tipo de valor encapsulado

    public User(String name, Email email, Password password) {
        this.name = name;
        this.email = email;
        this.password = password;
    }

    public void changeEmail(Email newEmail, AuthenticationService auth) {
        if (auth.isAuthenticated(this)) {
            this.email = newEmail;
            notifyEmailChange();
        } else {
            throw new SecurityException("Unauthorized email change attempt");
        }
    }
}
```



# Cohesión y Acoplamiento

El **acoplamiento** mide el grado de interdependencia entre módulos. Un bajo acoplamiento permite que los componentes sean más independientes, facilitando su mantenimiento y reutilización.



## Acoplamiento de datos

Módulos que se comunican pasando datos simples.

## Acoplamiento de marca

Módulos que comparten estructuras de datos complejos.

## Acoplamiento de control

Un módulo controla el flujo de otro enviando banderas o información de control

## Acoplamiento externo

Módulos que dependen de protocolos, dispositivos o formatos externos.

## Acoplamiento común

Módulos que comparten datos globales.

## Acoplamiento de contenido

Un módulo modifica directamente el estado interno de otro.

```
int calcularCuadrado(int numero)
```

```
void imprimirNombre(Empleado empleado)
```

```
void procesar(boolean modoDetallado)
```

```
public static Configuracion configGlobal;
```



# Cohesión y Acoplamiento

Tipo de Acoplamiento	Nivel de Acoplamiento	Deseable
1. Datos	Bajo	 Sí
2. Marca (Stamp)	Moderado	 Con cuidado
3. Control	Moderado/Alto	 Mejor evitar
4. Externo	Moderado/Alto	 Diseñar con precaución
5. Común	Alto	 No
6. Contenido	Muy Alto	 Nunca



# Cohesión y Acoplamiento

## ESTRATEGIAS

**Inversión de Dependencias:** Dependencia de abstracciones, no implementaciones concretas.

**Inyección de Dependencias:** Proveer las dependencias a un objeto en lugar de que éste las cree.

**Eventos y Observadores:** Comunicación indirecta entre componentes.

**Adaptadores y Fachadas:** Capas de dirección para aislar sistemas.



## METRICAS

**Acoplamiento Aferente (Ca):** Número de clases externas que dependen de clases dentro del módulo.

**Acoplamiento Eferente (Ce):** Número de clases dentro del módulo que dependen de clases externas.

**Inestabilidad (I):**  $Ce / (Ca + Ce)$ , donde 0 significa estable y 1 inestable.

Robert C. Martin ("Uncle Bob")



# Cohesión y Acoplamiento

La **cohesión** mide cuán relacionados están los elementos dentro de un módulo. Alta cohesión significa que los elementos de un módulo están fuertemente relacionados y enfocados en una única responsabilidad

**Cohesión coincidental:** Elementos agrupados sin relación lógica.

**Cohesión lógica:** Elementos que realizan tareas de categoría similar.

**Cohesión temporal:** Elementos que se ejecutan en el mismo momento.

**Cohesión procesal:** Elementos que participan en una secuencia de pasos.

**Cohesión comunicacional:** Elementos que operan sobre los mismos datos.

## Cohesión secuencial

La salida de un elemento es entrada para otro.



## Cohesión funcional

Todos los elementos contribuyen a una única tarea bien definida.

```
class Utilidades {
    void imprimir() { ... }
    void ordenar() { ... }
    void conectarDB() { ... }
}
```

```
void inicializar() {
    abrirArchivos();
    configurarRed();
    iniciarInterfaz();
}
```

```
void validar(String tipo) {
    if (tipo.equals("email")) { ... }
    else if (tipo.equals("telefono")) { ... }
}
```

```
void procesar() {
    validar();
    guardar();
    notificar();
}
```

```
public class ClienteManager {
    public void mostrarInformacion(Cliente cliente)
    public boolean validarCorreo(Cliente cliente);
    public void desactivarCliente(Cliente cliente);
}
```

```
void procesarFactura() {
    Datos datos = leerDatos();
    Factura f = generarFactura(datos);
    guardarFactura(f);
}
```

```
class Autenticador {
    boolean autenticar(String usuario, String clave) {
        return validarCredenciales(usuario, clave);
    }
}
```



# Cohesión y Acoplamiento

Tipo	Relación entre elementos	Calidad	Principio de Responsabilidad Única
Coincidencial	Ninguna	✗ Mala	Una clase debe tener solo una razón para cambiar.
Lógica	Categoría común, seleccionados por condicionales	✗ Baja	Extracción de Clases Mover funcionalidades relacionadas a nuevas clases especializadas
Temporal	Ejecutados al mismo tiempo	⚠ Media	
Procesal	Parte de una secuencia	⚠ Media	
Comunicacional	Operan sobre los mismos datos	✓ Buena	Composición sobre Herencia Usar composición para combinar comportamientos.
Secuencial	Salida de uno es entrada de otro	✓ Muy buena	
Funcional	Una única responsabilidad bien definida	✓ Excelente	Clases Cohesivas por Conceptos del Dominio Modelar clases según entidades del dominio del problema.





# Cohesión y Acoplamiento

**Alto acoplamiento :** Depende directamente de múltiples sistemas externos.

**Baja cohesión :** Realiza múltiples tareas no estrechamente relacionadas.

**Dificultad para pruebas :** Difícil de probar sin integración completa con sistemas externos.

```
// Diseño con ALTO acoplamiento y BAJA cohesión
class OrderProcessor {
    private Database db;
    private EmailService emailService;
    private PaymentGateway paymentGateway;
    private InventorySystem inventory;
    private TaxCalculator taxCalculator;

    public void processOrder(Order order) {
        // Verificar inventario
        boolean inStock = inventory.checkStock(order.getItems());
        if (!inStock) {
            throw new OutOfStockException();
        }

        // Calcular impuestos
        double tax = taxCalculator.calculateTax(order);
        order.setTax(tax);

        // Procesar pago
        PaymentResult result = paymentGateway.processPayment(order.getCustomer(),
            order.getTotalWithTax());
        if (!result.isSuccessful()) {
            throw new PaymentFailedException();
        }

        // Actualizar inventario
        inventory.updateStock(order.getItems());

        // Guardar orden
        db.save(order);

        // Enviar confirmación
        emailService.sendOrderConfirmation(order);
    }
}
```





# Cohesión y Acoplamiento

```
// Diseño mejorado con BAJO acoplamiento y ALTA cohesión

interface StockChecker {
    boolean checkAvailability(List<OrderItem> items);
    void updateInventory(List<OrderItem> items);
}

interface PaymentProcessor {
    PaymentResult process(Customer customer, Money amount);
}

interface OrderRepository {
    void save(Order order);
}

interface OrderNotifier {
    void notifyOrderConfirmation(Order order);
}
```

**Bajo acoplamiento :** Depende de abstracciones en lugar de implementaciones concretas.

**Alta cohesión :** Cada método tiene una única responsabilidad clara.

**Facilidad de pruebas :** Posibilidad de crear simulacros o stubs para los componentes.

**Flexibilidad :** Fácil adaptación a diferentes implementaciones de infraestructura.

```
class OrderProcessor {
    private final StockChecker stockChecker;
    private final TaxCalculator taxCalculator;
    private final PaymentProcessor paymentProcessor;
    private final OrderRepository orderRepository;
    private final OrderNotifier orderNotifier;

    // Inyección de dependencias
    public OrderProcessor(StockChecker stockChecker,
                         TaxCalculator taxCalculator,
                         PaymentProcessor paymentProcessor,
                         OrderRepository orderRepository,
                         OrderNotifier orderNotifier) {
        this.stockChecker = stockChecker;
        this.taxCalculator = taxCalculator;
        this.paymentProcessor = paymentProcessor;
        this.orderRepository = orderRepository;
        this.orderNotifier = orderNotifier;
    }

    public void processOrder(Order order) {
        ensureItemsAvailable(order);
        calculateAndApplyTax(order);
        processPayment(order);
        updateInventory(order);
        saveOrder(order);
        notifyCustomer(order);
    }

    private void ensureItemsAvailable(Order order) {
        if (!stockChecker.checkAvailability(order.getItems())) {
            throw new OutOfStockException();
        }
    }

    private void calculateAndApplyTax(Order order) {
        Money tax = taxCalculator.calculateTax(order);
        order.setTax(tax);
    }

    private void processPayment(Order order) {
```





# Cohesión y Acoplamiento

```
// Diseño mejorado con BAJO acoplamiento y ALTA cohesión
interface StockChecker {
    boolean checkAvailability(List<OrderItem> items);
    void updateInventory(List<OrderItem> items);
}

interface PaymentProcessor {
    PaymentResult process(Customer customer, Money amount);
}

interface OrderRepository {
    void save(Order order);
}

interface OrderNotifier {
    void notifyOrderConfirmation(Order order);
}
```

**Bajo acoplamiento :** Depende de abstracciones en lugar de implementaciones concretas.

**Alta cohesión :** Cada método tiene una única responsabilidad clara.

**Facilidad de pruebas :** Posibilidad de crear simulacros o stubs para los componentes.

**Flexibilidad :** Fácil adaptación a diferentes implementaciones de infraestructura.

```
class OrderProcessor {
    private final StockChecker stockChecker;
    private final TaxCalculator taxCalculator;
    private final PaymentProcessor paymentProcessor;
    private final OrderRepository orderRepository;
    private final OrderNotifier orderNotifier;

    // Inyección de dependencias
    public OrderProcessor(StockChecker stockChecker,
                         TaxCalculator taxCalculator,
                         PaymentProcessor paymentProcessor,
                         OrderRepository orderRepository,
                         OrderNotifier orderNotifier) {
        this.stockChecker = stockChecker;
        this.taxCalculator = taxCalculator;
        this.paymentProcessor = paymentProcessor;
        this.orderRepository = orderRepository;
        this.orderNotifier = orderNotifier;
    }

    public void processOrder(Order order) {
        ensureItemsAvailable(order);
        calculateAndApplyTax(order);
        processPayment(order);
        updateInventory(order);
        saveOrder(order);
        notifyCustomer(order);
    }

    private void ensureItemsAvailable(Order order) {
        if (!stockChecker.checkAvailability(order.getItems())) {
            throw new OutOfStockException();
        }
    }

    private void calculateAndApplyTax(Order order) {
        Money tax = taxCalculator.calculateTax(order);
        order.setTax(tax);
    }

    private void processPayment(Order order) {
```



# Delegación y responsabilidad

La **delegación** es un mecanismo mediante el cual un objeto transfiere la responsabilidad de una tarea específica a otro objeto. Es una alternativa a la herencia que promueve la composición sobre la jerarquía.

## Delegación Explícita:

El objeto delegante llama directamente a un método de otro objeto (el delegado).

**Características:** El código contiene una llamada explícita al método delegado. El delegante conoce al delegado.

```
class Motor {  
    void encender() {  
        System.out.println("Motor encendido");  
    }  
  
}  
  
class Auto {  
    private Motor motor = new Motor();  
  
    public void arrancar() {  
        motor.encender(); // Delegación explícita  
    }  
}
```



# Delegación y responsabilidad

## Delegación Implícita

La delegación ocurre de forma automática, normalmente usando composición y ocultando la llamada explícita. Se ve comúnmente en patrones como **Proxy** o **Decorator**.

### Características:

La delegación se abstrae, el cliente no sabe que se está delegando. Utilizada para extender o controlar el comportamiento sin modificar el objeto original.

```
interface Servicio {  
    void ejecutar();  
}  
  
class ServicioBasico implements Servicio {  
    public void ejecutar() {  
        System.out.println("Ejecutando servicio básico");  
    }  
}  
  
class ServicioDecorado implements Servicio {  
    private Servicio servicio;  
  
    public ServicioDecorado(Servicio servicio) {  
        this.servicio = servicio;  
    }  
  
    public void ejecutar() {  
        servicio.ejecutar(); // Delegación implícita  
        System.out.println("Funcionalidad adicional");  
    }  
}
```



## Delegación Dinámica

La delegación se define o cambia en tiempo de ejecución, como en el patrón **State**, **Strategy**, o mediante reflexión.

### Características:

Alta flexibilidad: el objeto delegado puede variar en tiempo de ejecución.

Utilizada para cambiar comportamiento dinámicamente.

```
interface Estado {  
    void manejar();  
}  
  
class EstadoActivo implements Estado {  
    public void manejar() {  
        System.out.println("En estado activo");  
    }  
}  
  
class EstadoInactivo implements Estado {  
    public void manejar() {  
        System.out.println("En estado inactivo");  
    }  
}  
  
class Contexto {  
    private Estado estado;  
  
    public void setEstado(Estado estado) {  
        this.estado = estado;  
    }  
  
    public void operar() {  
        estado.manejar(); // Delegación dinámica  
    }  
}
```

# Delegación y responsabilidad

La **responsabilidad** se refiere a las obligaciones que tiene un objeto en el sistema. Una clase bien diseñada debe tener responsabilidades coherentes y claramente definidas.

## Basada en Conocimiento

“¿Quién tiene la información necesaria para llevar a cabo la tarea?”

**Idea clave:** La responsabilidad se asigna a la clase que posee o tiene acceso directo a los datos necesarios.

**Ejemplo:** Si una clase Factura contiene la lista de ítems, ella misma debería calcular el total, ya que tiene la información.

```
class Factura {  
    private List<Item> items;  
  
    public double calcularTotal() {  
        return items.stream().mapToDouble(Item::getPrecio).sum()  
    }  
}
```



# Delegación y responsabilidad

## Basada en Comportamiento

“¿Quién debe hacer esta acción?”

**Idea clave:** Se asigna la responsabilidad a la clase que naturalmente debe encargarse de la acción, desde el punto de vista del comportamiento deseado.

**Ejemplo:** Si queremos que un botón reaccione a un clic, la clase Botón debe tener el método clic(), no una clase externa.

```
class CuentaBancaria {  
    private double saldo;  
  
    public void transferirA(CuentaBancaria destino, double monto) {  
        if (saldo >= monto) {  
            this.saldo -= monto;  
            destino.saldo += monto;  
        }  
    }  
}
```

```
class Boton {  
    public void clic() {  
        System.out.println("Botón pulsado");  
    }  
}
```

## Basada en Dominio

“¿A qué concepto del dominio pertenece esta responsabilidad?”

**Idea clave:** Se asigna la responsabilidad según la semántica del dominio del problema (usando principios de diseño orientado al dominio).

**Ejemplo:** En un sistema bancario, si existe una clase CuentaBancaria, la lógica de transferencia de fondos pertenece al dominio de esa clase, no a una clase genérica como GestorOperaciones.

# Delegación y responsabilidad

```
// Sistema de generación de reportes con delegación avanzada

// Interfaz para estrategias de formato
interface ReportFormatter {
    String format(ReportData data);
}

// Implementaciones concretas
class HTMLReportFormatter implements ReportFormatter {
    @Override
    public String format(ReportData data) {
        StringBuilder html = new StringBuilder("<html><body>");
        html.append("<h1>").append(data.getTitle()).append("</h1>");
        // Implementación específica para HTML
        html.append("</body></html>");
        return html.toString();
    }
}

class PDFReportFormatter implements ReportFormatter {
    @Override
    public String format(ReportData data) {
        // Implementación específica para PDF
        return "PDF content...";
    }
}
```

```
// Interfaz para estrategias de distribución
interface ReportDistributor {
    void distribute(String formattedReport, List<String> recipients);
}

// Implementaciones concretas
class EmailDistributor implements ReportDistributor {
    private EmailService emailService;

    public EmailDistributor(EmailService emailService) {
        this.emailService = emailService;
    }

    @Override
    public void distribute(String formattedReport, List<String> recipients) {
        recipients.forEach(recipient ->
            emailService.sendEmail(recipient, "Report", formattedReport)
        );
    }
}
```

```
// Clase que coordina el proceso delegando responsabilidades
class ReportGenerator {
    private final ReportDataCollector dataCollector;
    private final ReportFormatter formatter;
    private final ReportDistributor distributor;

    public ReportGenerator(
        ReportDataCollector dataCollector,
        ReportFormatter formatter,
        ReportDistributor distributor) {
        this.dataCollector = dataCollector;
        this.formatter = formatter;
        this.distributor = distributor;
    }

    public void generateAndDistributeReport(
        ReportType type,
        Date startDate,
        Date endDate,
        List<String> recipients) {
        // Delega la recolección de datos
        ReportData data = dataCollector.collectData(type, startDate, endDate);

        // Delega el formateo
        String formattedReport = formatter.format(data);

        // Delega la distribución
        distributor.distribute(formattedReport, recipients);
    }
}
```



# Delegación y responsabilidad

```
// Cliente que configura y usa el sistema
class ReportingSystem {
    public void createMonthlyReport() {
        // Configuración de componentes
        ReportDataCollector collector = new DatabaseReportCollector(dbConnection);
        ReportFormatter formatter = new HTMLReportFormatter();
        ReportDistributor distributor = new EmailDistributor(emailService);

        // Creación del generador con delegación
        ReportGenerator generator = new ReportGenerator(collector, formatter, distributor);

        // Uso del generador
        generator.generateAndDistributeReport(
            ReportType.MONTHLY_SALES,
            getFirstDayOfMonth(),
            getLastDayOfMonth(),
            getRecipientList()
        );
    }
}
```

Delegación limpia  
Asignación clara de responsabilidades  
Flexibilidad mediante composición  
Bajo acoplamiento  
Alta cohesión

## Indicadores de Mala Asignación de Responsabilidades

**Clases Dios** : Objetos que conocen o hacen demasiado.

**Alta Frecuencia de Cambio** : Clases que cambian por múltiples razones no relacionadas.

**Métodos Utilitarios Abundantes** : Señal de que la responsabilidad no está en la clase correcta.

**Acoplamiento Excesivo** : Necesidad de conocer muchos otros objetos para completar sus tareas.



# Lenguaje unificado de modelado - UML



UML es una notación gráfica unificada para representar múltiples características de sistemas desarrollados con el paradigma de orientación a objetos.

Captura de requisitos - Análisis -  
Diseño – Implementación - Pruebas



# Lenguaje unificado de modelado - UML



**Vista de casos de uso:** Diagramas de Casos de Uso

**Vista lógica:**

Estructura: Diagrama de clases y diagramas de objetos

Interacción: Diagrama de estado, secuencia, colaboración y actividad

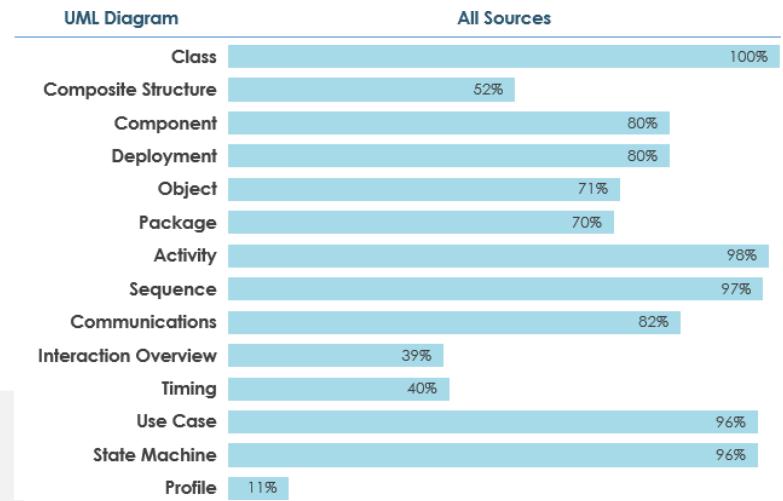
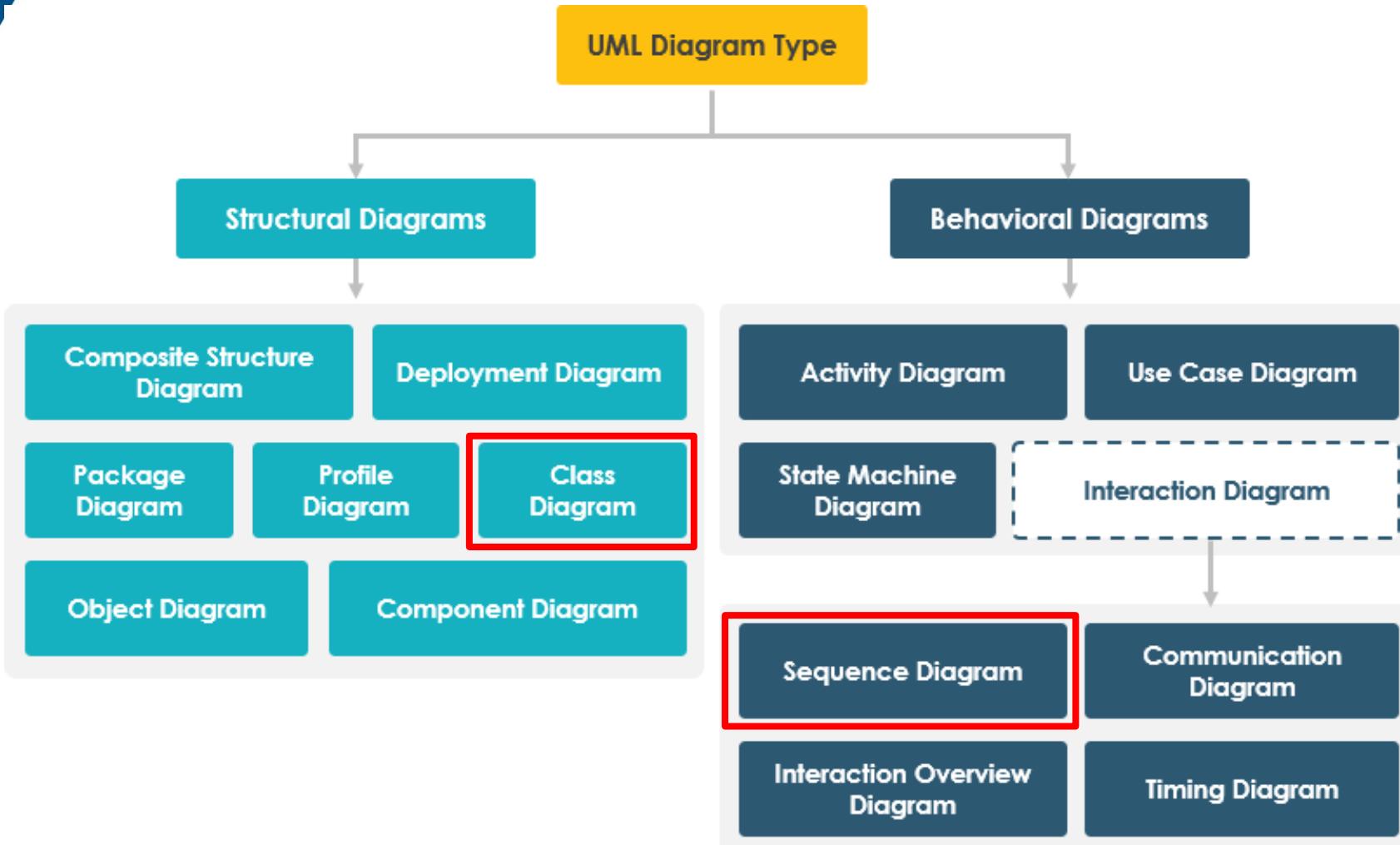
**Vista de implantación:** Diagrama de despliegue

**Vista de componentes:** Diagrama de componentes

**Vista de concurrencia:**

Vista lógica + Vista de componentes + Vista de implantación

# Lenguaje unificado de modelado - UML



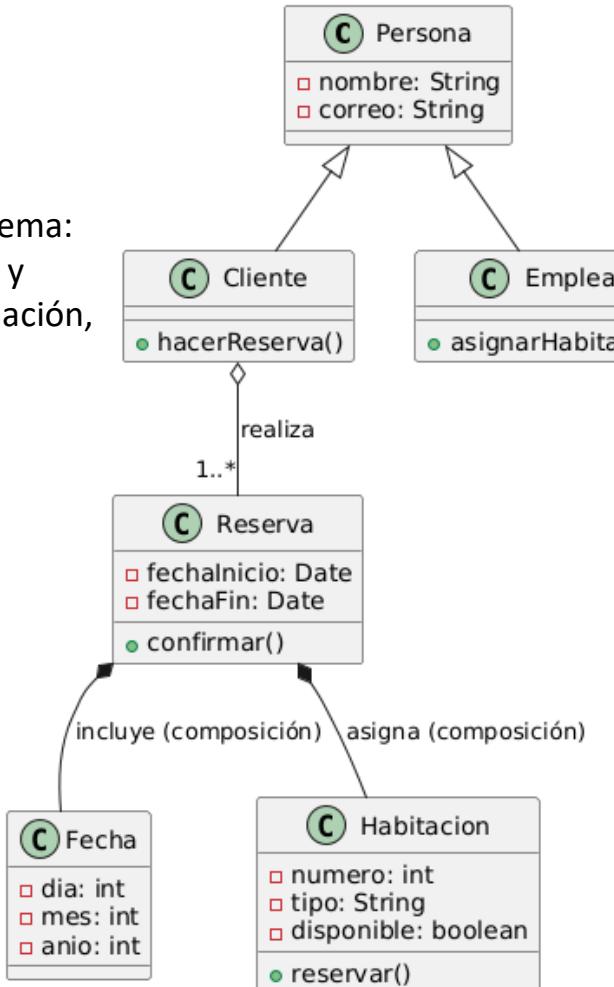
ampliamente utilizado, si es  $\geq 60\%$  de las fuentes  
escasamente utilizado si es  $\leq 40\%$  de las fuentes

<https://www.visual-paradigm.com/>

# Lenguaje unificado de modelado - UML

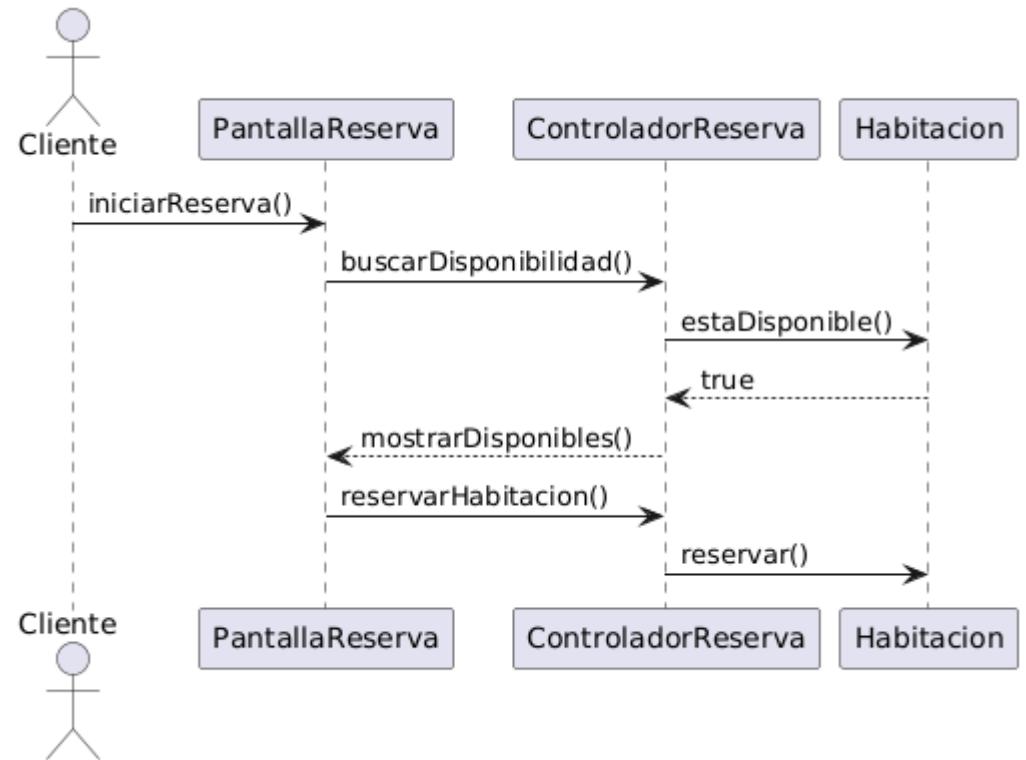
## Diagrama de Clases:

**Propósito:** Representa la estructura estática del sistema: clases, atributos, métodos y relaciones (herencia, asociación, composición, etc.).



## Diagrama de Secuencia:

**Propósito:** Muestra la interacción entre objetos a lo largo del tiempo para un caso de uso específico.



# Lenguaje unificado de modelado - UML

Diagrama de Casos de Uso

**Propósito:** Representa las funcionalidades del sistema desde el punto de vista del usuario (actores y casos de uso).

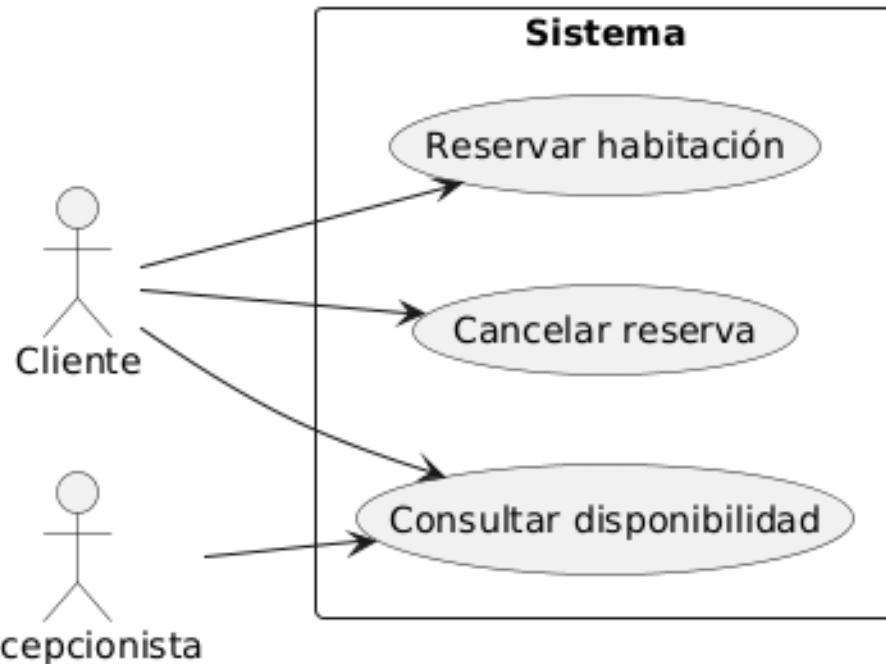
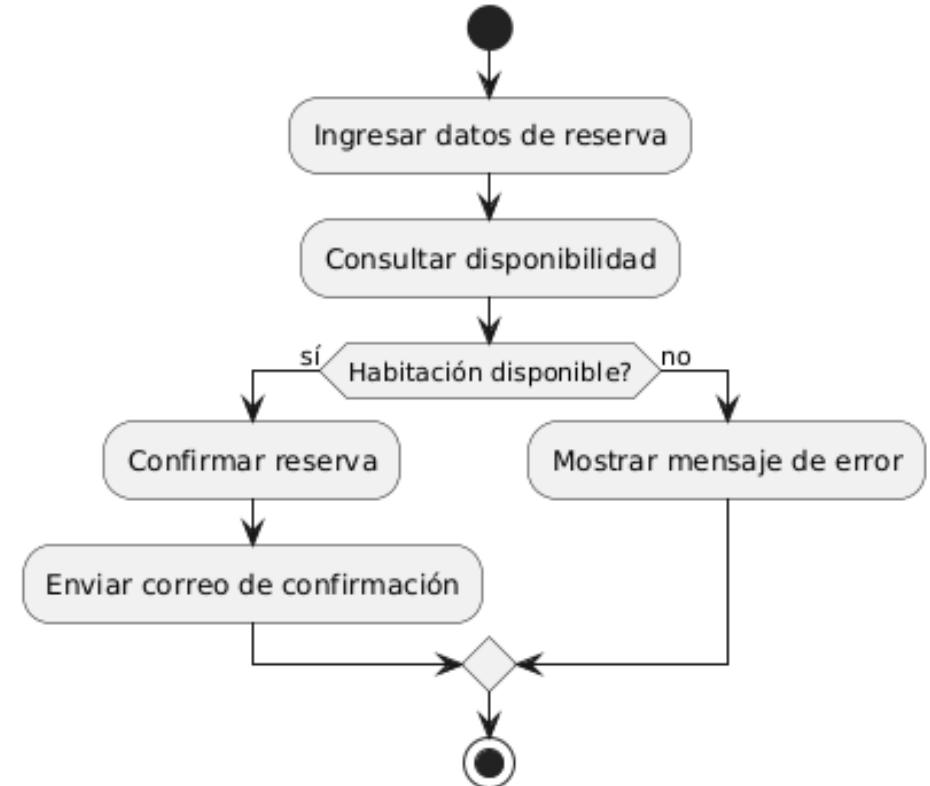


Diagrama de Actividades:

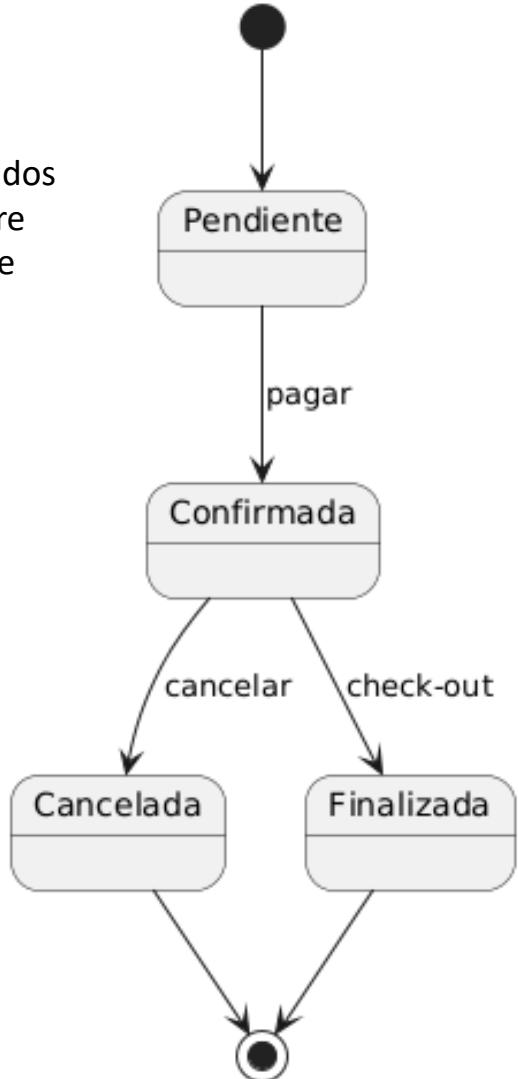
**Propósito:** Representa flujos de trabajo o procesos mediante actividades y decisiones. Importante para modelar lógica de negocio



# Lenguaje unificado de modelado - UML

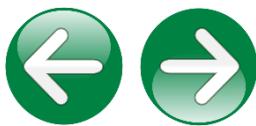
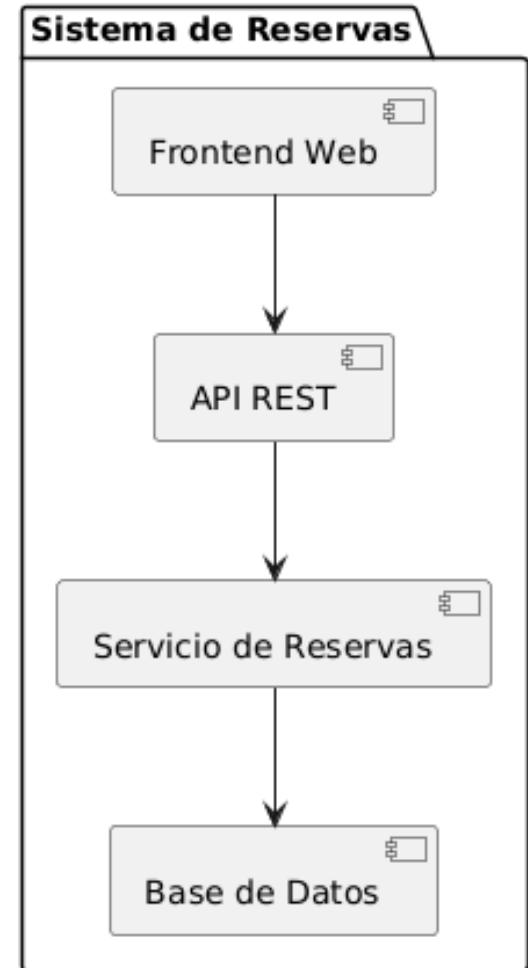
## Diagrama de Estados:

**Propósito:** Muestra los distintos estados de un objeto y las transiciones entre ellos. Modelar comportamiento de objetos y su ciclo de vida



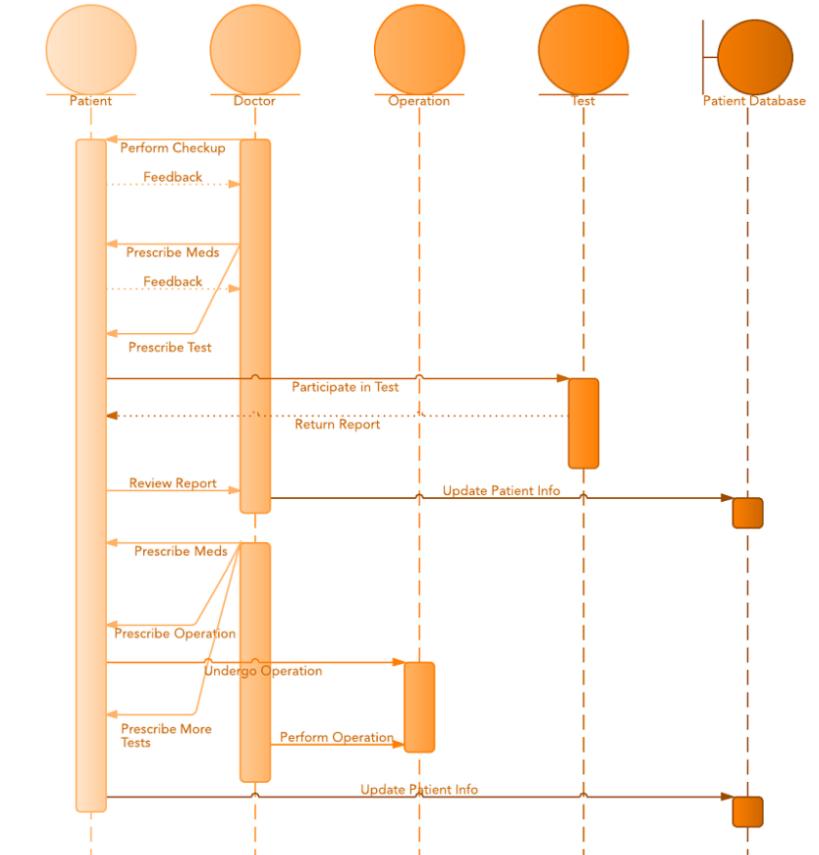
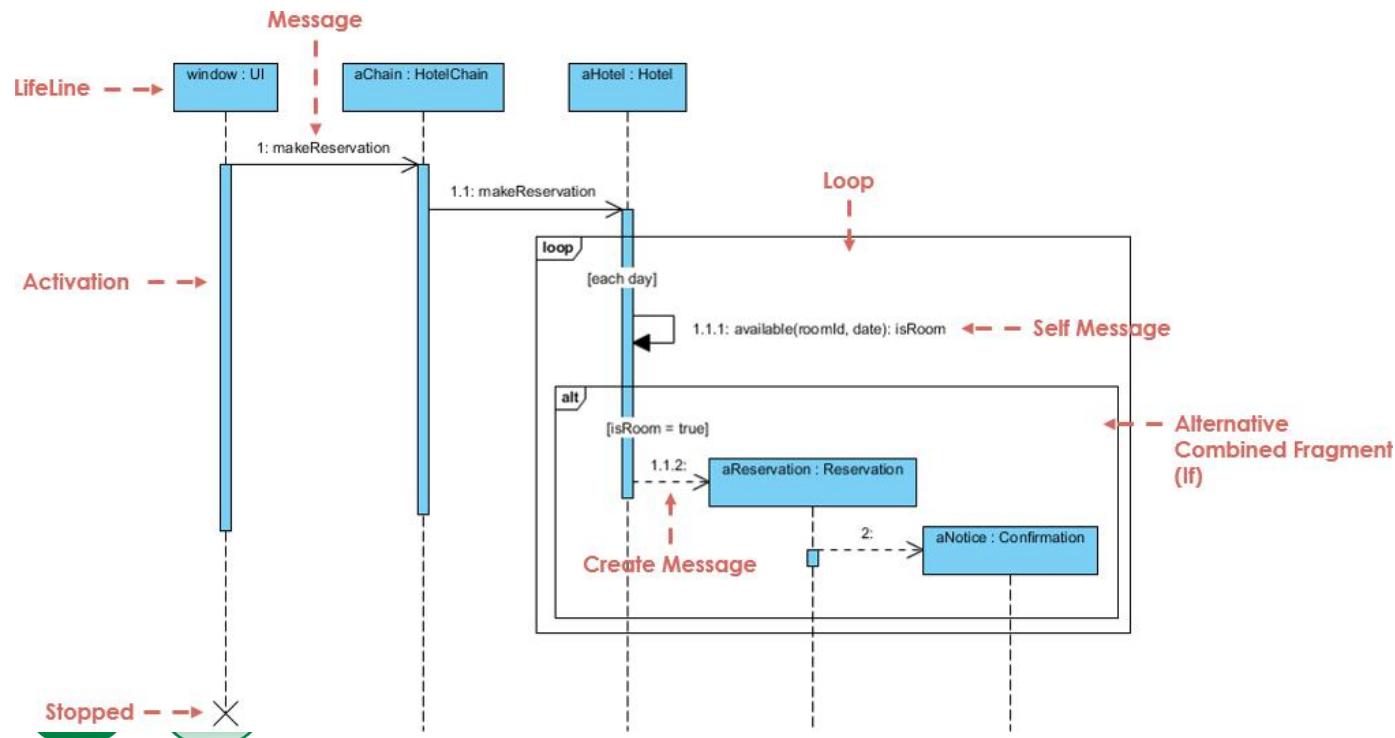
## Diagrama de Componentes

**Propósito:** Representa los componentes físicos del sistema y sus relaciones (paquetes, bibliotecas, módulos). Importante para diseñar arquitectura a nivel de despliegue lógico y modularidad del sistema



# Diagrama de secuencia - UML

**El diagrama de secuencia** modela la colaboración de objetos basándose en una secuencia de tiempo. Muestra cómo los objetos interactúan con otros en un escenario particular de un caso de uso.



Recursos para aprender POO con Java



Introducción a la programación orientada a objetos en Java

Universidad de los Andes

Introducción a UML

Universidad de los Andes



Academy

Ruta de aprendizaje Java





**UNIVERSIDAD**  
Popular del cesar