

Repaso sobre Diagramas

Ingeniería de Sistemas y Computación

Estudiante

Manuel Eduardo Alarcon Aza

Profesor

William Javier Matallana Porras

Universidad de Cundinamarca – UDEC

Chía

2026 - 1

## Tabla de contenido

|  |    |
|--|----|
| <b>Introducción .....</b>                    | 2  |
| <b>Tipos de diagramas de clase UML .....</b> | 2  |
| <b>Relaciones .....</b>                      | 3  |
| <b>Conclusión .....</b>                      | 11 |
| <b>Referencias.....</b>                      | 11 |

### **Introducción**

En el documento buscamos conocer que es y como se emplean los diagramas de clase UML en la programación orientada a objetos en java, revisando conceptos como lo son clases, objetos, herencia, encapsulamiento y polimorfismo y como podremos llegar a utilizarlos en proyectos de desarrollo de software.

### **Tipos de diagramas de clase UML**

Se dividen principalmente en dos familias: los que muestran la foto (estructura) y los que muestran la película (comportamiento).

#### **Diagramas de Estructura (La "Foto")**

Estos diagramas muestran qué elementos componen el sistema, como si abrieras el capó de un coche para ver las piezas sin encender el motor.

- **Diagrama de Clases:** Es el más famoso. Define los objetos del sistema (como "Usuario" o "Factura") y qué datos contienen.

- **Diagrama de Objetos:** Es una versión específica del de clases, pero con datos reales. No dice "un usuario tiene nombre", sino "Juan tiene 25 años".
- **Diagrama de Componentes:** Muestra cómo se agrupan las piezas de software en bloques más grandes, como bases de datos o librerías externas.
- **Diagrama de Despliegue:** Describe el hardware. Dónde vive el software (servidores, la nube o el teléfono del usuario).

### **Diagramas de Comportamiento (La "Película")**

Aquí es donde vemos el sistema en movimiento, cómo reacciona cuando alguien toca un botón o cómo fluye la información.

- **Diagrama de Casos de Uso:** Representa al usuario (el "actor") y qué quiere lograr. Es puramente funcional: "El cliente quiere comprar pan".
- **Diagrama de Secuencia:** Es el más usado para programar. Muestra el orden cronológico de los mensajes: primero la App pide datos, luego la Base de Datos responde, y al final se muestra el mensaje.
- **Diagrama de Actividades:** Es básicamente un diagrama de flujo profesional. Muestra decisiones (si pasa esto, haz aquello) y procesos paso a paso.
- **Diagrama de Estados:** Ideal para objetos con "humores" o ciclos de vida. Por ejemplo, una factura que pasa de "Pendiente" a "Pagada" o "Anulada".

### **Relaciones**

En el modelado de sistemas, las relaciones son los conectores lógicos que definen cómo las entidades interactúan y dependen entre sí. La Asociación es la relación estructural más común,

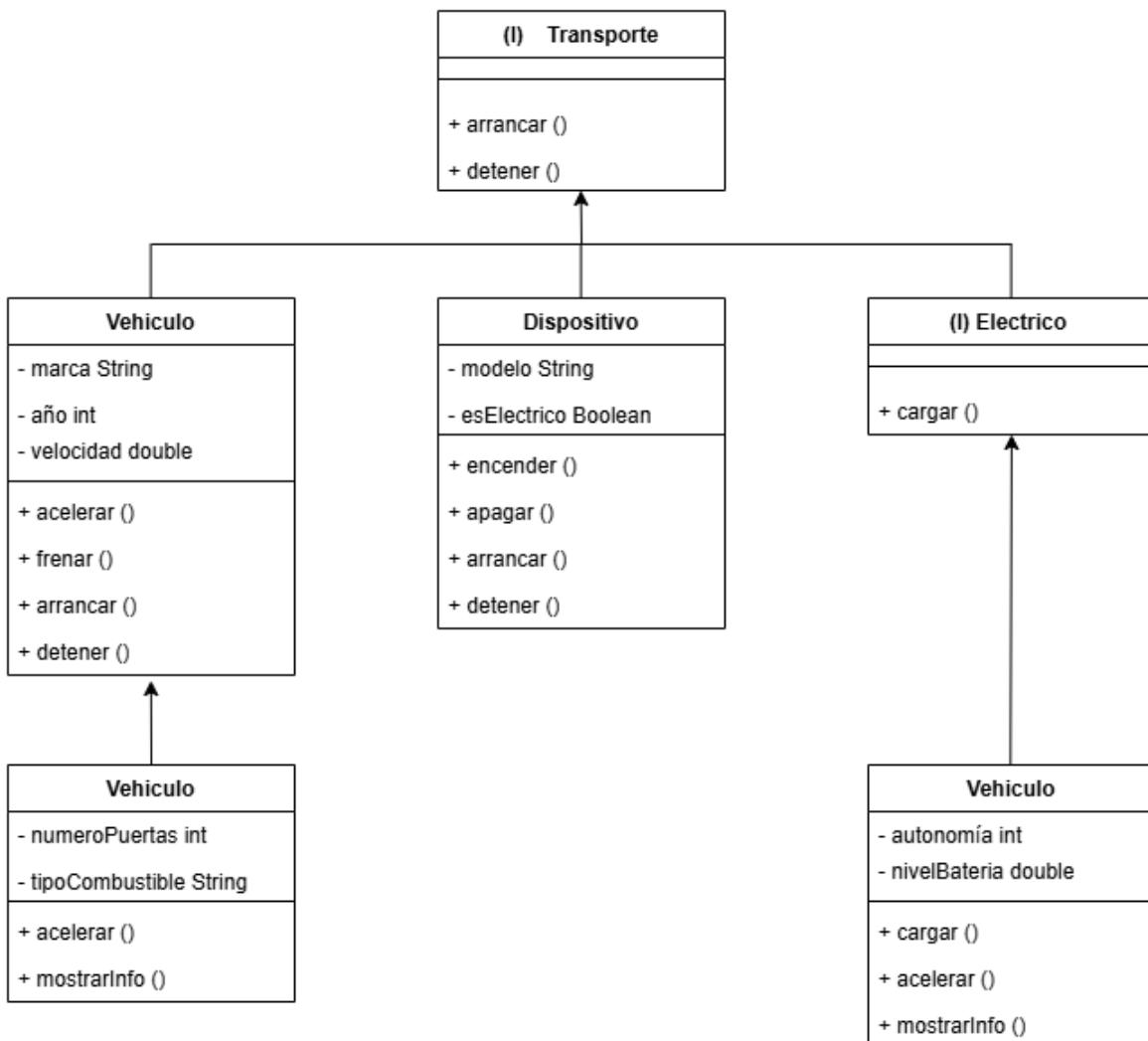
indicando que las instancias de una clase están conectadas con las de otra para cumplir una función; puede ser unidireccional o bidireccional y se caracteriza por tener "multiplicidad" (indicando cuántos objetos participan en el vínculo). Por su parte, la Generalización o Herencia representa la relación de taxonomía entre una superclase y una subclase, donde la subclase hereda los atributos y métodos de la superior, permitiendo el polimorfismo y la reutilización de código bajo el principio de "es un tipo de".

Un nivel más profundo de asociación se encuentra en la Agregación y la Composición, que modelan relaciones de "parte-todo". La Agregación es una forma de asociación débil donde la parte puede existir independientemente del todo (el ciclo de vida no está ligado). En contraste, la Composición es una asociación fuerte y restrictiva: la existencia de la "parte" está estrictamente subordinada a la vida del "todo", lo que implica que si el objeto contenedor se destruye, sus partes constituyentes se eliminan automáticamente, garantizando la integridad referencial del sistema.

Finalmente, existen relaciones de carácter funcional y de implementación como la Dependencia y la Realización. La Dependencia es una relación de uso transitoria donde un cambio en la definición de un elemento (el independiente) puede afectar al otro (el dependiente), aunque no exista un vínculo estructural permanente entre ellos. La Realización, por otro lado, es el vínculo entre una interfaz y la clase que la implementa; es una declaración formal de que la clase se compromete a ejecutar el comportamiento definido por la interfaz, actuando como un contrato técnico que asegura la compatibilidad operativa dentro de la arquitectura del software.

## Ejemplo en Java

Diagrama:



```

package org.example;

public class Main { & Alarconmanuel
    public static void main(String[] args) { & Alarconmanuel
        System.out.println(" SISTEMA DE TRANSPORTE \n");

        //Crear un Auto
        Auto auto1 = new Auto( marca: "Toyota", Anio: 2024, velocidad: 0, numeroPuertas: 4, tipoCombustible: "Gasolina");
        auto1.arrancar();
        auto1.acelerar();
        auto1.acelerar();
        auto1.mostrarInfo();
        auto1.detener();
        System.out.println(auto1.toString());

        System.out.println("\n Auto Electrico \n");

        //Crear un Auto Electrico
        AutoElectrico tesla = new AutoElectrico( marca: "Tesla", anio: 2024, velocidad: 0, autonomia: 500, nivelBateria: 80.0);
        tesla.arrancar();
        tesla.acelerar();
        tesla.acelerar();
        tesla.mostrarInfo();
        tesla.cargar();
        tesla.detener();
        System.out.println(tesla.toString());

        System.out.println("\n Dispositivo \n");
    }
}

```

```

//Crear un Dispositivo
Dispositivo patineta = new Dispositivo( modelo: "Patineta Electrica X1", esElectrico: true);
patineta.encender();
patineta.arrancar();
patineta.detener();
patineta.apagar();
System.out.println(patineta.toString());

//System.out.println("\n ejemploPolimorfismo \n");

//ejemploPolimorfismo
Transporte t1 = auto1;
Transporte t2 = tesla;
Transporte t3 = patineta;

t1.arrancar();
t2.arrancar();
t3.arrancar();

System.out.println("\n Interface Electrico \n");

Electrico e1 = tesla;
e1.cargar();
}

```

```

package org.example;

public interface Electrico {
    void cargar();
}

```

```
package org.example;

@l public class Vehiculo implements Transporte { 2 usages 2 inheritors & Alarconmanuel
    //Atributos
    private String marca; 6 usages
    private int anio; 4 usages
    private double velocidad; 11 usages

    //Constructor
    public Vehiculo() { 2 usages & Alarconmanuel
    }

    public Vehiculo(String marca, int anio, double velocidad) { 2 usages & Alarconmanuel
        this.marca = marca;
        this.anio = anio;
        this.velocidad = velocidad;
    }

    //Metodos
    > public String getMarca() { return marca; }

    > public void setMarca(String marca) { this.marca = marca; }

    > public int getAnio() { return anio; }

    @l public void setAnio(int anio) { no usages & Alarconmanuel
        this.anio = anio;
    }

    > public double getVelocidad() { return velocidad; }

    > public void setVelocidad(double velocidad) { this.velocidad = velocidad; }
```

```
    this.velocidad += 10;
    System.out.println("Acelerando... Velocidad: " + velocidad + " km/h");
}

public void frenar() { no usages & Alarconmanuel
    if(velocidad >= 10) {
        this.velocidad -= 10;
    } else {
        this.velocidad = 0;
    }
    System.out.println("Frenando... Velocidad: " + velocidad + " km/h");
}

@Override 6 usages & Alarconmanuel
public void arrancar() { System.out.println("El vehiculo " + marca + " ha arrancado"); }

@Override 3 usages & Alarconmanuel
public void detener() {
    this.velocidad = 0;
    System.out.println("El vehiculo " + marca + " se ha detenido");
}

@Override 2 overrides & Alarconmanuel
public String toString() {
    return "Vehiculo{" +
        "marca='" + marca + '\'' +
        ", anio=" + anio +
        ", velocidad=" + velocidad +
        '}';
}
```

```
1 package org.example;
2
3 public class Dispositivo implements Transporte { 2 usages & Alarconmanuel
4     //Atributos
5     private String modelo; 9 usages
6     private boolean esElectrico; 5 usages
7
8     //Constructor
9     public Dispositivo() { no usages & Alarconmanuel
10    }
11
12     public Dispositivo(String modelo, boolean esElectrico) { 1 usage & Alarconmanuel
13         this.modelo = modelo;
14         this.esElectrico = esElectrico;
15     }
16
17     //Metodos
18     public String getModelo() { no usages & Alarconmanuel
19         return modelo;
20     }
21
22     public void setModelo(String modelo) { no usages & Alarconmanuel
23         this.modelo = modelo;
24     }
25
26     public boolean isElectrico() { no usages & Alarconmanuel
27         return esElectrico;
28     }
29
30     public void setElectrico(boolean esElectrico) { no usages & Alarconmanuel
31         this.esElectrico = esElectrico;
32     }
```

```
33         System.out.println("Dispositivo " + modelo + " encendido");
34     }
35
36     public void apagar() { 1 usage & Alarconmanuel
37         System.out.println("Dispositivo " + modelo + " apagado");
38     }
39
40     @Override 6 usages & Alarconmanuel
41     public void arrancar() {
42         if(esElectrico) {
43             System.out.println("Dispositivo electrico " + modelo + " iniciado");
44         } else {
45             System.out.println("Dispositivo " + modelo + " arrancado");
46         }
47     }
48
49
50     @Override 3 usages & Alarconmanuel
51     public void detener() {
52         System.out.println("Dispositivo " + modelo + " detenido");
53     }
54
55
56     @Override & Alarconmanuel
57     public String toString() {
58         return "Dispositivo{" +
59             "modelo='" + modelo + '\'' +
60             ", esElectrico=" + esElectrico +
61             '}';
62     }
63 }
```

```
nmmit Alt+O e org.example;

public class Auto extends Vehiculo { 2 usages & Alarconmanuel
    //Atributos
    private int numeroPuertas; 5 usages
    private String tipoCombustible; 5 usages

    //Constructor
    public Auto() { no usages & Alarconmanuel
        super();
    }

    public Auto(String marca, int Anio, double velocidad, int numeroPuertas, String tipoCombustible) { 1 usage &
        super(marca, Anio, velocidad);
        this.numeroPuertas = numeroPuertas;
        this.tipoCombustible = tipoCombustible;
    }

    //Metodos
    >     public int getNumeroPuertas() { return numeroPuertas; }

    >     public void setNumeroPuertas(int numeroPuertas) { this.numeroPuertas = numeroPuertas; }

    >     public String getTipoCombustible() { return tipoCombustible; }

    >     public void setTipoCombustible(String tipoCombustible) { this.tipoCombustible = tipoCombustible; }

    @Override 4 usages & Alarconmanuel
    public void acelerar() {
        setVelocidad(getVelocidad() + 15);
        System.out.println("El auto acelera mas rapido! Velocidad: " + getVelocidad() + " km/h");
    }
}
```

```
36
37     @Override 4 usages & Alarconmanuel
38     public void acelerar() {
39         setVelocidad(getVelocidad() + 15);
40         System.out.println("El auto acelera mas rapido! Velocidad: " + getVelocidad() + " km/h");
41     }
42
43     public void mostrarInfo() { 1 usage & Alarconmanuel
44         System.out.println("== Informacion del Auto ==");
45         System.out.println("Marca: " + getMarca());
46         System.out.println("Año: " + getAnio());
47         System.out.println("Número de puertas: " + numeroPuertas);
48         System.out.println("Tipo de combustible: " + tipoCombustible);
49         System.out.println("Velocidad actual: " + getVelocidad() + " km/h");
50     }
51
52     @Override & Alarconmanuel
53     public String toString() {
54         return "Auto{" +
55             "marca='" + getMarca() + '\'' +
56             ", Anio=" + getAnio() +
57             ", velocidad=" + getVelocidad() +
58             ", numeroPuertas=" + numeroPuertas +
59             ", tipoCombustible='" + tipoCombustible + '\'' +
60         "}";
61     }
}
```

```

Commit Alt+O e org.example;

2
3  public class AutoElectrico extends Vehiculo implements Electrico { 2 usages & Alarconmanuel
4      //Atributos
5      private int autonomia; 5 usages
6      private double nivelBateria; 8 usages
7
8      //Constructor
9      public AutoElectrico() { no usages & Alarconmanuel
10          super();
11      }
12
13      public AutoElectrico(String marca, int anio, double velocidad, int autonomia, double nivelBateria) { 1 usage & Alarcon
14          super(marca, anio, velocidad);
15          this.autonomia = autonomia;
16          this.nivelBateria = nivelBateria;
17      }
18
19      //Metodos
20      > public int getAutonomia() { return autonomia; }
21
22      public void setAutonomia(int autonomia) { no usages & Alarconmanuel
23          this.autonomia = autonomia;
24      }
25
26      > public double getNivelBateria() { return nivelBateria; }
27
28      > public void setNivelBateria(double nivelBateria) { this.nivelBateria = nivelBateria; }
29
30      @Override 2 usages & Alarconmanuel
31      public void cargar() {
32          this.nivelBateria = 100.0;

```

```

41
42      @Override 4 usages & Alarconmanuel
43      public void acelerar() {
44          setVelocidad(getVelocidad() + 20);
45          this.nivelBateria -= 2;
46          System.out.println("Auto electrico acelerando! Velocidad: " + getVelocidad() + " km/h - Bateria: " + nivelBateria + "%");
47      }
48
49      public void mostrarInfo() { 1 usage & Alarconmanuel
50          System.out.println("== Informacion del Auto Electrico ==");
51          System.out.println("Marca: " + getMarca());
52          System.out.println("Año: " + getAnio());
53          System.out.println("Autonomia: " + autonomia + " km");
54          System.out.println("Nivel de bateria: " + nivelBateria + "%");
55          System.out.println("Velocidad actual: " + getVelocidad() + " km/h");
56      }
57
58      @Override & Alarconmanuel
59      public String toString() {
60          return "AutoElectrico{" +
61              "marca='" + getMarca() + '\'' +
62              ", anio=" + getAnio() +
63              ", velocidad=" + getVelocidad() +
64              ", autonomia=" + autonomia +
65              ", nivelBateria=" + nivelBateria +
66              '}';
67      }
68  }

```

```
package org.example;

public interface Transporte {
    void arrancar(); 
    void detener();
}
```

## Conclusiones +++++

- La importancia del UML radica en su capacidad para actuar como un "filtro de errores" previo a la construcción. En términos prácticos, permite detectar inconsistencias lógicas o fallos de arquitectura en la fase de diseño, cuando el costo de corrección es mínimo. Al visualizar las relaciones y dependencias antes de escribir código, los equipos evitan retrabajos costosos y aseguran que la estructura del sistema sea escalable y fácil de mantener a largo plazo.
- El fin práctico de UML es servir como el "idioma universal" en el desarrollo de software. Su relevancia trasciende lo técnico, ya que permite que desarrolladores, arquitectos y analistas compartan una visión unificada sin ambigüedades. Esto garantiza que la documentación sea comprensible incluso para nuevos integrantes del equipo, facilitando la transferencia de conocimiento y asegurando que la implementación final sea fiel a los requerimientos del negocio.

## Referencias

- Miro. (s. f.). *How to Design UML Class Diagrams in Miro INSTANTLY with AI 🤖* [Vídeo].  
<https://miro.com/>. <https://miro.com/es/diagrama/que-es-diagrama-clases-uml/>
- *Tutorial de diagrama de clases UML*. (2026, 5 febrero). Lucidchart.  
<https://www.lucidchart.com/pages/es/tutorial-de-diagrama-de-clases-uml>

- *What is Class Diagram?* (s. f.). <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/>
- <https://platzi.com/cursos/patrones-diseno-software/relaciones-entre-clases/>