
Taller Práctico: Materialización de Relaciones en Diagramas de Clases usando Java

En este taller, aprenderás a transformar las relaciones de un diagrama de clases en código Java. Exploraremos las principales relaciones: cardinalidad, usabilidad, dependencia, herencia, implementación, agregación, composición y clase asociación.

Objetivos del Taller

1. Entender cómo se representan las diferentes relaciones de un diagrama de clases en Java.
2. Materializar relaciones como herencia, implementación, agregación, composición y más en código.
3. Practicar con ejercicios de codificación que refuercen los conceptos.

Conceptos y Ejercicios

1. Herencia (`extends`)

- **Concepto**: Una clase hereda atributos y métodos de otra clase.
- **Ejemplo:**

java
Copiar código

```java
// Clase base Vehiculo
class Vehiculo {
    public void mover() {
        System.out.println("El vehículo se está moviendo.");
    }
}

// Clase derivada Coche que hereda de Vehiculo
class Coche extends Vehiculo {
    public void tocarBocina() {
        System.out.println("El coche toca la bocina.");
    }
}
```

Ejercicio:

1. Crea una clase `Animal` y una clase `Perro` que herede de `Animal`. Añade métodos relevantes y llama a estos desde la clase `Perro`.

2. Implementación (`implements`)

- **Concepto**: Una clase implementa una o más interfaces.
- **Ejemplo:**

java
Copiar código

```java
// Interfaz Volador
interface Volador {
    void volar();
}

// Clase Avion que implementa Volador
class Avion implements Volador {
    @Override
    public void volar() {
        System.out.println("El avión está volando.");
    }
}
```

Ejercicio:

1. Crea una interfaz `Nadador` con un método `nadar()`, y una clase `Delfin` que implemente `Nadador`.

3. Dependencia

- **Concepto**: Una clase usa otra clase temporalmente, generalmente dentro de un método.
- **Ejemplo:**

java
Copiar código

```java
// Clase Motor
class Motor {
    public void encender() {
        System.out.println("El motor se enciende.");
    }
}

// Clase Auto que depende de Motor
class Auto {
    public void arrancar() {
        Motor motor = new Motor(); // Dependencia
        motor.encender();
        System.out.println("El auto está en marcha.");
    }
}
```

Ejercicio:

1. Crea una clase `Impresora` y una clase `Documento`. Simula que `Impresora` tiene un método `imprimir(Documento doc)` que usa `Documento`.

4. Agregación (`has-a`)

- **Concepto**: Relación débil donde una clase contiene una referencia a otra sin controlar su ciclo de vida.
- **Ejemplo:**

java
Copiar código

```java
// Clase Rueda
class Rueda {
    public void girar() {
        System.out.println("La rueda gira.");
    }
}

// Clase Bicicleta que tiene una o más Ruedas
class Bicicleta {
    private Rueda ruedaDelantera;
    private Rueda ruedaTrasera;

    public Bicicleta(Rueda ruedaDelantera, Rueda ruedaTrasera) {
        this.ruedaDelantera = ruedaDelantera;
        this.ruedaTrasera = ruedaTrasera;
    }

    public void mover() {
        ruedaDelantera.girar();
        ruedaTrasera.girar();
        System.out.println("La bicicleta se mueve.");
```

```
    }
}
```

Ejercicio:

1. Crea una clase `Aula` y una clase `Estudiante`. Implementa una relación de agregación donde `Aula` tiene múltiples `Estudiantes`.

5. Composición (strong `has-a`)

- **Concepto**: Relación fuerte donde una clase controla el ciclo de vida de otra clase.
- **Ejemplo:**

java
Copiar código

```java
// Clase Habitacion
class Habitacion {
    public void limpiar() {
        System.out.println("La habitación se está limpiando.");
    }
}

// Clase Casa que tiene Habitaciones (composición)
class Casa {
    private Habitacion habitacion;

    public Casa() {
        this.habitacion = new Habitacion(); // Casa controla la vida de Habitacion
    }

    public void mantener() {
        habitacion.limpiar();
        System.out.println("La casa está siendo mantenida.");
    }
}
```

Ejercicio:

1. Crea una clase `Computadora` y una clase `Procesador`. Simula que `Computadora` controla la creación y destrucción de `Procesador`.

6. Asociación

- **Concepto**: Relación general entre dos clases, puede ser unidireccional o bidireccional.
- **Ejemplo de Asociación Bidireccional:**

java
Copiar código

```java
// Clase Profesor
class Profesor {
    private String nombre;
    private Curso curso; // Asociación bidireccional

    public Profesor(String nombre) {
        this.nombre = nombre;
    }

    public void setCurso(Curso curso) {
        this.curso = curso;
```

```
    }
}

// Clase Curso
class Curso {
    private String nombre;
    private Profesor profesor; // Asociación bidireccional

    public Curso(String nombre) {
        this.nombre = nombre;
    }

    public void setProfesor(Profesor profesor) {
        this.profesor = profesor;
    }
}
```

Ejercicio:

1. Crea una clase `Jugador` y una clase `Equipo`. Implementa una asociación bidireccional entre ambas.

7. Clase Asociación (Relación de Muchos a Muchos)

- **Concepto**: Se usa para modelar relaciones de muchos a muchos, generalmente mediante una clase intermedia.
- **Ejemplo:**

java
Copiar código

```
// Clase Curso (uno de los extremos de la relación)
class Curso {
    private String nombre;

    public Curso(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
}

// Clase Estudiante (otro extremo de la relación)
class Estudiante {
    private String nombre;

    public Estudiante(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
}

// Clase Registro para modelar la relación muchos a muchos entre Curso y Estudiante
class Registro {
    private Estudiante estudiante;
    private Curso curso;
```

```
    public Registro(Estudiante estudiante, Curso curso) {
        this.estudiante = estudiante;
        this.curso = curso;
    }

    public void mostrarRegistro() {
        System.out.println(estudiante.getNombre() + " está inscrito en " + curso.getNombre());
    }
}
```

Ejercicio:

1. Crea una clase `Empleado` y una clase `Proyecto`. Implementa una clase intermedia `Asignacion` que permita relacionar varios empleados a varios proyectos.

8. Cardinalidad

- **Concepto**: Define el número de instancias que pueden estar asociadas en una relación.

Ejemplo de uno a muchos:

java
Copiar código

```java
// Clase Empresa
class Empresa {
    private List<Empleado> empleados = new ArrayList<>();

    public void contratar(Empleado empleado) {
        empleados.add(empleado);
    }

    public void mostrarEmpleados() {
        for (Empleado empleado : empleados) {
            System.out.println(empleado.getNombre());
        }
    }
}

// Clase Empleado
class Empleado {
    private String nombre;

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

    public String getNombre() {
        return nombre;
    }
}
```

Ejercicio:

1. Crea una clase `Biblioteca` que pueda contener múltiples `Libros`. Implementa los métodos para añadir libros y mostrar la lista de libros.
2. Realizar el diagrama de clases correspondiente a cada ejemplo y ejercicio de este taller.