---

# Taller Práctico: Relaciones en Diagramas de Clases usando Java

## Objetivos del Taller

1. Comprender y diferenciar las relaciones de agregación y composición.
2. Implementar estas relaciones en Java, respetando las reglas específicas de cada una.
3. Practicar con ejercicios para reforzar los conceptos.

## Conceptos y Ejercicios Revisados

### 1. Agregación (`has-a`)

- **Concepto**: Una relación débil entre clases donde una clase "Todo" puede tener objetos de una clase "Parte". Las instancias de "Parte" pueden existir independientemente de la instancia de "Todo". La clase "Todo" puede tener un constructor por defecto y otro que acepte objetos "Parte". Debe tener métodos para agregar, buscar, eliminar y eliminar todos los objetos "Parte".

**Ejemplo y Ejercicio:**

java
Copiar código

```java
// Clase Parte: Rueda
class Rueda {
    private String marca;

    public Rueda(String marca) {
        this.marca = marca;
    }

    public String getMarca() {
        return marca;
    }
}

// Clase Todo: Coche que tiene Ruedas (agregación)
class Coche {
    private List<Rueda> ruedas;

    // Constructor por defecto
    public Coche() {
        this.ruedas = new ArrayList<>();
    }

    // Constructor con Ruedas
    public Coche(List<Rueda> ruedas) {
        this.ruedas = ruedas;
    }

    // Método para agregar una rueda
    public void agregarRueda(Rueda rueda) {
        ruedas.add(rueda);
    }

    // Método para buscar una rueda por marca
    public Rueda buscarRueda(String marca) {
        return ruedas.stream()
                     .filter(rueda -> rueda.getMarca().equals(marca))
                     .findFirst()
                     .orElse(null);
    }

    // Método para eliminar una rueda
```

```
    public boolean eliminarRueda(Rueda rueda) {
        return ruedas.remove(rueda);
    }

    // Método para eliminar todas las ruedas
    public void eliminarTodasLasRuedas() {
        ruedas.clear();
    }
}
```

**Ejercicio:**

1. Crea una clase `Equipo` que tenga una lista de `Jugador`. Implementa métodos para agregar, buscar, eliminar y eliminar todos los `Jugadores`.

**2. Composición (strong `has-a`)**

- **Concepto**: Relación fuerte donde una clase "Todo" no puede existir sin sus partes. La clase "Todo" no puede tener un constructor por defecto; debe recibir las instancias de sus partes en el constructor. Debe tener métodos para agregar, buscar y eliminar partes. No puede tener un método para eliminar todas sus partes, y si se intenta, debe lanzar una excepción.

**Ejemplo y Ejercicio:**

java
Copiar código

```
// Clase Parte: Procesador
class Procesador {
    private String modelo;

    public Procesador(String modelo) {
        this.modelo = modelo;
    }

    public String getModelo() {
        return modelo;
    }
}

// Clase Todo: Computadora que se compone de un Procesador
class Computadora {
    private Procesador procesador;

    // Constructor que recibe Procesador
    public Computadora(Procesador procesador) {
        if (procesador == null) {
            throw new IllegalArgumentException("Una computadora no puede existir sin un procesador.");
        }
        this.procesador = procesador;
    }

    // Método para agregar un nuevo procesador
    public void cambiarProcesador(Procesador nuevoProcesador) {
        if (nuevoProcesador == null) {
            throw new IllegalArgumentException("El procesador no puede ser nulo.");
        }
        this.procesador = nuevoProcesador;
    }

    // Método para buscar el procesador actual
    public Procesador obtenerProcesador() {
```

```
        return this.procesador;
    }

    // Método para eliminar el procesador (lanzar excepción si se intenta eliminar)
    public void eliminarProcesador() {
        throw new UnsupportedOperationException("No se puede eliminar el procesador. La computadora dejaría
de existir.");
    }
}
```

**Ejercicio:**

1. Crea una clase `Casa` que se componga de una `Puerta`. Implementa los métodos para agregar, buscar, y eliminar la `Puerta`. Asegúrate de que eliminar la `Puerta` lance una excepción.

## 3. Relaciones Restantes

**Cardinalidad y Asociación**

- Las relaciones de cardinalidad y asociación se implementan naturalmente al diseñar las listas (uno a muchos) o al establecer referencias de objetos (uno a uno, muchos a muchos). En clases asociadas, también se maneja con clases intermedias.

**Clase Asociación (Relación de Muchos a Muchos)**

- **Concepto**: Relación de muchos a muchos usando una clase intermedia.

**Ejemplo:**

java
Copiar código

```
// Clase Proyecto
class Proyecto {
    private String nombre;

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

    public String getNombre() {
        return nombre;
    }
}

// Clase Empleado
class Empleado {
    private String nombre;

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

    public String getNombre() {
        return nombre;
    }
}

// Clase Asignacion para modelar la relación muchos a muchos
class Asignacion {
    private Empleado empleado;
    private Proyecto proyecto;
```

```
    public Asignacion(Empleado empleado, Proyecto proyecto) {
        this.empleado = empleado;
        this.proyecto = proyecto;
    }

    public void mostrarAsignacion() {
        System.out.println(empleado.getNombre() + " está asignado al proyecto " + proyecto.getNombre());
    }
}
```

**Ejercicio:**

1. Implementa una clase `Matrimonio` que modele una relación de muchos a muchos entre `Persona` y `Evento`. Asegúrate de que las clases respeten las relaciones establecidas.
2. Realizar el diagrama de clases correspondiente a cada ejemplo y ejercicio de este taller.