Prueba Diagnóstica para Tópicos Especiales de Programación (Semana 1)

Prof. Italo Visconti
17 de septiembre de 2025

1. Ejercicio: Evaluando Expresiones Aritméticas

Considere todas las posibles expresiones aritméticas posibles, por ejemplo:

```
(5 + 10 + (40 − 10) + (50 ∗ 10 ∗ 3) − (500/10))
```

Se le pide lo siguiente:

1. Diseño

1. Proponga un **diagrama de clases** para modelar expresiones como un **árbol**.
2. La solución debe favorecer **extensibilidad** (poder agregar operadores o funciones nuevas sin romper código existente).

2. Patrón de diseño

1. Indique qué patrón(es) usa y por qué.

3. Implementación

1. Implemente un método `eval()` que calcule el valor de la expresión.
2. Puede escribirlo en el lenguaje que prefiera (TypeScript/Java/Python) o en **pseudocódigo** claro.

Solución propuesta

1. Patrón de diseño

- **Composite**: Se trata a los objetos individuales (un número) y a los grupos de objetos (una operación) de manera uniforme, por esta razón todas son1 `Expression`
- **Interpreter**: cada clase concreta implementa `eval()` como la interpretación de su símbolo.

2. UML

Diagrama UML

3. Pseudocódigo

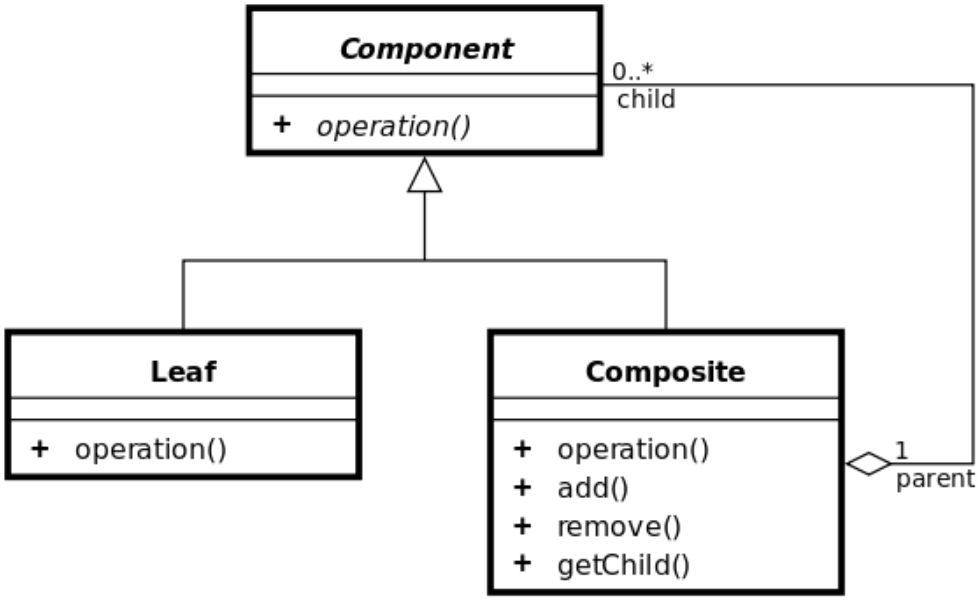
```
// componente
interface expression
    method eval() -> number

// hoja (leaf)
class NumericValue implements expression
    private value: number

    constructor(value: number)
        this.value = value

    method eval() -> number
        return this.value

// compuesto (composite)
abstract class BinaryOperation implements expression
    protected left: expression
    protected right: expression

    constructor(left: expression, right: expression)
        this.left = left
        this.right = right

    abstract method eval() -> number

// implementaciones concretas del compuesto
class Addition extends BinaryOperation
    method eval() -> number
        return this.left.eval() + this.right.eval()

class Subtraction extends BinaryOperation
    method eval() -> number
        return this.left.eval() - this.right.eval()

class Multiplication extends BinaryOperation
    method eval() -> number
        return this.left.eval() * this.right.eval()

class Division extends BinaryOperation
    method eval() -> number
        rightValue = this.right.eval()
        if rightValue is 0
            throw new Error("Error: División por cero.")
        return this.left.eval() / rightValue
```

4. Implementación en TS

```
interface Expression {
  eval(): number;
}
```

```
// Hoja (Leaf)
class NumericValue implements Expression {
  private value: number;

  constructor(value: number) {
    this.value = value;
  }

  eval(): number {
    return this.value;
  }
}

// Composite para operaciones binarias
abstract class BinaryOperation implements Expression {
  protected left: Expression;
  protected right: Expression;

  constructor(left: Expression, right: Expression) {
    this.left = left;
    this.right = right;
  }

  abstract eval(): number;
}

// Implementaciones concretas de los composite
class Addition extends BinaryOperation {
  eval(): number {
    return this.left.eval() + this.right.eval();
  }
}

class Subtraction extends BinaryOperation {
  eval(): number {
    return this.left.eval() - this.right.eval();
  }
}

class Multiplication extends BinaryOperation {
  eval(): number {
    return this.left.eval() * this.right.eval();
  }
}

class Division extends BinaryOperation {
  eval(): number {
    const rightValue = this.right.eval();
    if (rightValue === 0) {
      throw new Error("Error: División por cero.");
    }
    return this.left.eval() / rightValue;
  }
```

```
}

// Armamos la expresión
const finalExpression = new Subtraction(
    new Addition(
        new Addition(
            new Addition(
                new NumericValue(5),
                new NumericValue(10)
            ),
            new Subtraction(
                new NumericValue(40),
                new NumericValue(10)
            )
        ),
        new Multiplication(
            new Multiplication(
                new NumericValue(50),
                new NumericValue(10)
            ),
            new NumericValue(3)
        )
    ),
    new Division(
        new NumericValue(500),
        new NumericValue(10)
    )
);

console.log(`(5 + 10 + (40 - 10) + (50 * 10 * 3) - (500 / 10))`);
console.log(`Resultado: ${finalExpression.eval()}`); //1495

/*
* El patron Composite nos permite tratar tanto a los objetos individuales
* (NumericValue) como a las composiciones de objetos (BinaryOperation y sus subcla
* de manera uniforme a través de la interfaz `Expression`.
*/
```



composite-pattern.png