5.4 Punto de Vista Lógico

Propósito del Punto de Vista Lógico

Este punto de vista se enfoca en las abstracciones clave del sistema, representando las entidades fundamentales, sus atributos, operaciones y relaciones. Su objetivo es:
Definir la estructura de clases y objetos que componen la calculadora.
Establecer jerarquías de herencia para promover la reutilización de código.
Documentar los principios de diseño (SOLID, DRY) aplicados.
Mejorar en la mantenibilidad y escalabilidad del sistema.
Audiencia principal:
Desarrolladores: Para implementar las clases y métodos con precisión.
Equipo de pruebas: Para diseñar casos de prueba unitarios basados en contratos de métodos.

Diagrama de Clases Principal

El sistema se basa en una jerarquía de clases que representa las operaciones, el historial y la calculadora como orquestadora.

Entidades clave:

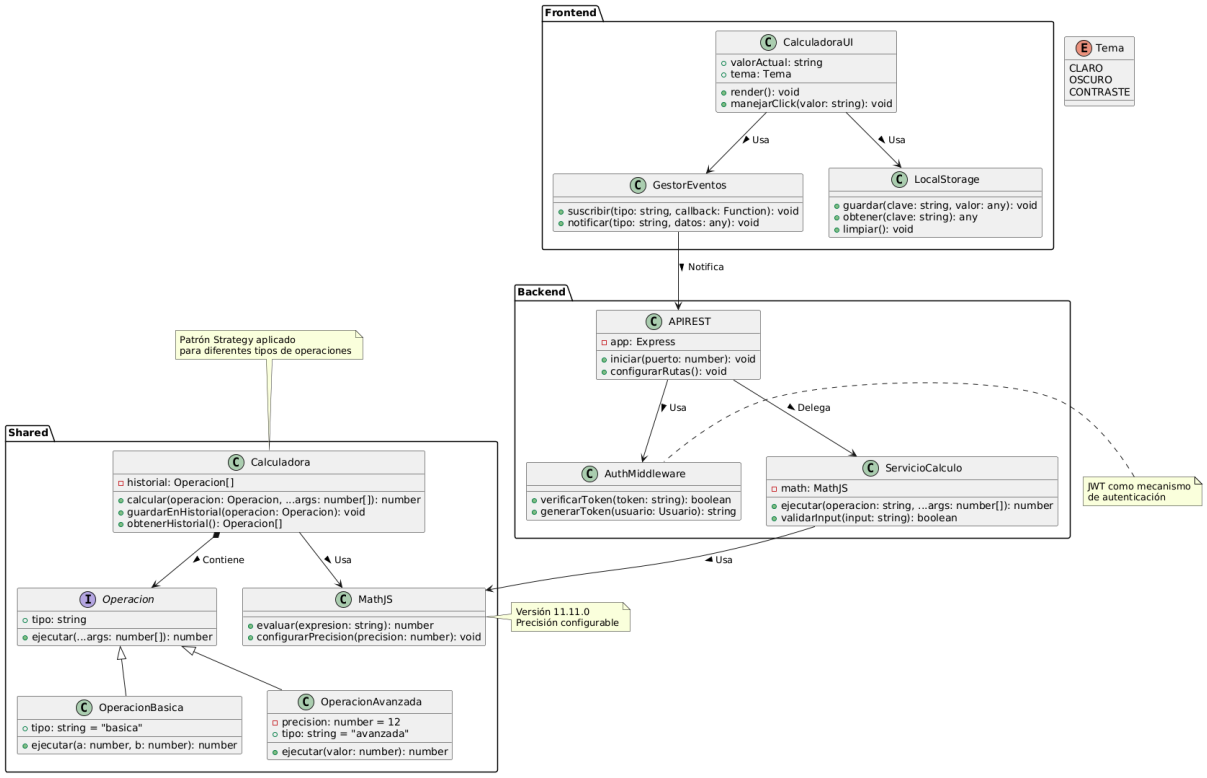
Operación: Clase abstracta base que define la estructura común de toda operación matemática.

Operación Básica: Hereda de Operación; implementa operaciones como suma, resta, multiplicación y división.

Operación Avanzada: Hereda de Operación; maneja cálculos más complejos como trigonometría, logaritmos y exponenciales.

Calculadora: Administra la ejecución de operaciones y el almacenamiento del historial.

Cada clase cuenta con sus propios atributos y métodos específicos para cumplir con su responsabilidad.



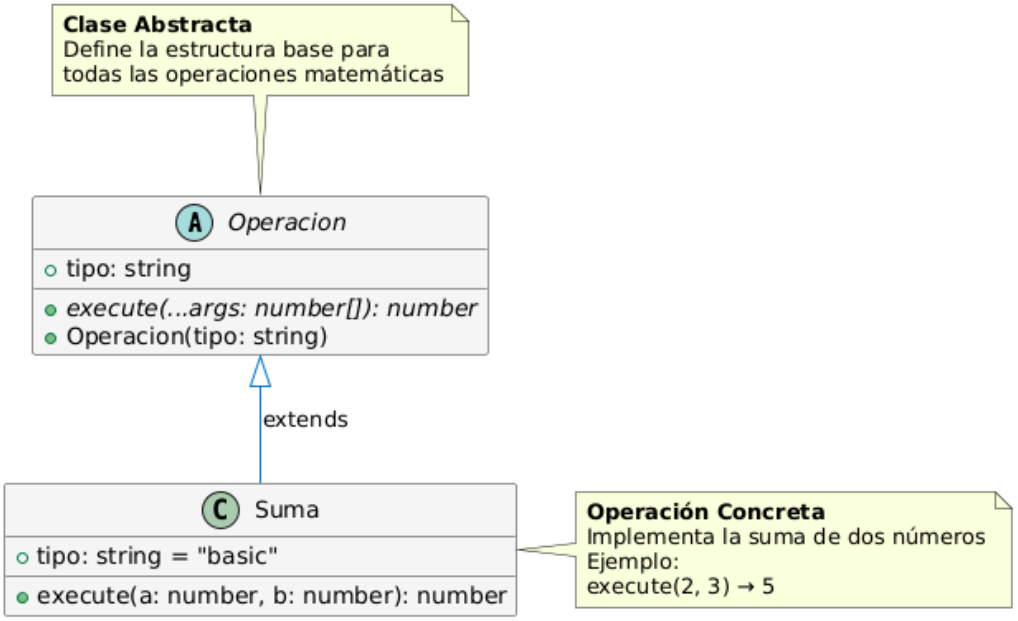
Descripción de Clases

Clase	Responsabilidad	Atributos/Métodos Clave
Operación	Clase abstracta para todas las operaciones.	tipo: String (basic, trig), execute()
Operación Básica	Implementa suma/resta/multiplicación/división .	execute(a, b) → a + b
Operacion Avanzada	Maneja trigonometría, logaritmos, etc.	precision: Number, execute(valor) → sin(valor)
Calculadora	Orquesta operaciones y gestiona historial.	calcular(op, valor), guardar en Historial()

Jerarquía de Herencia Detallada

La jerarquía de herencia permite organizar las operaciones según su complejidad y reutilizar código común. Se establece de la siguiente manera:

Operación es una clase abstracta general. Operación Básica y Operación Avanzada son clases hijas que implementan el método de ejecución de forma especializada. Nuevas operaciones (como raíces, potencias, factoriales, etc.) pueden añadirse fácilmente extendiendo esta jerarquía sin modificar clases existentes.

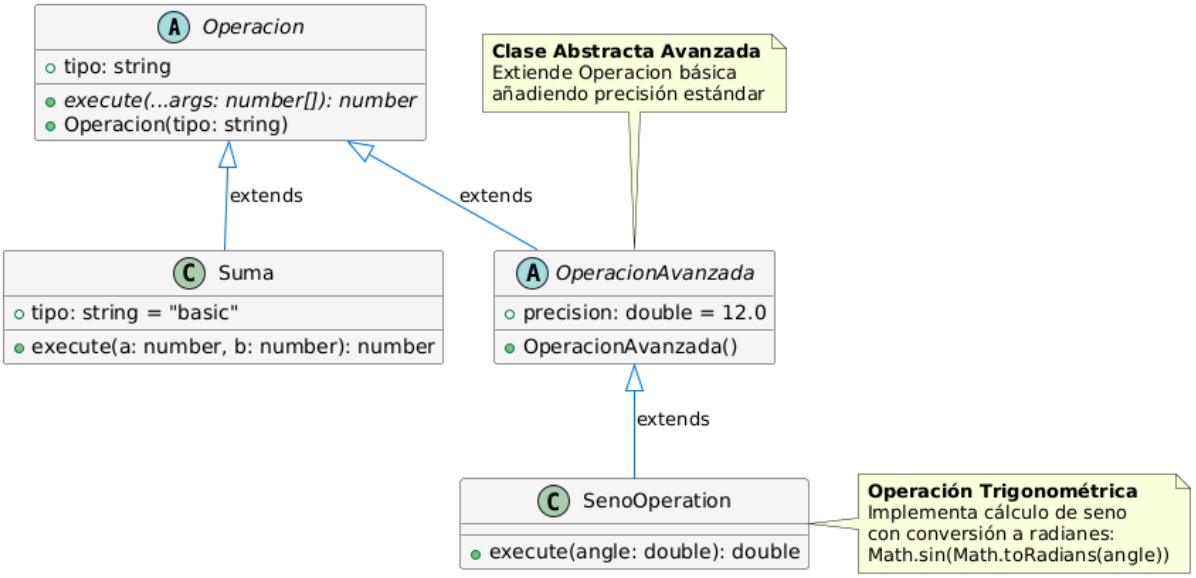


Operaciones Avanzadas

```
// Ejemplo en Java
public abstract class OperacionAvanzada extends Operacion {
    protected double precision = 12.0;

    public OperacionAvanzada() {
        super("advanced");
    }
}

public class SenoOperation extends OperacionAvanzada {
    @Override
    public double execute(double angle) {
        return Math.sin(Math.toRadians(angle));
    }
}
```



Patrones de Diseño Aplicados

Se aplican los siguientes patrones para mejorar la flexibilidad y robustez del diseño:

Strategy: Permite cambiar dinámicamente la lógica de cálculo según el tipo de operación seleccionada por el usuario.

Factory Method: Facilita la creación de instancias de operaciones a través de fábricas especializadas.

Singleton: Aplicado al historial para garantizar una única instancia compartida por todo el sistema.

Estos patrones aseguran bajo acoplamiento y alta cohesión entre clases.

Strategy Pattern

Contexto: Seleccionar algoritmos de cálculo en runtime.
Implementación:
python

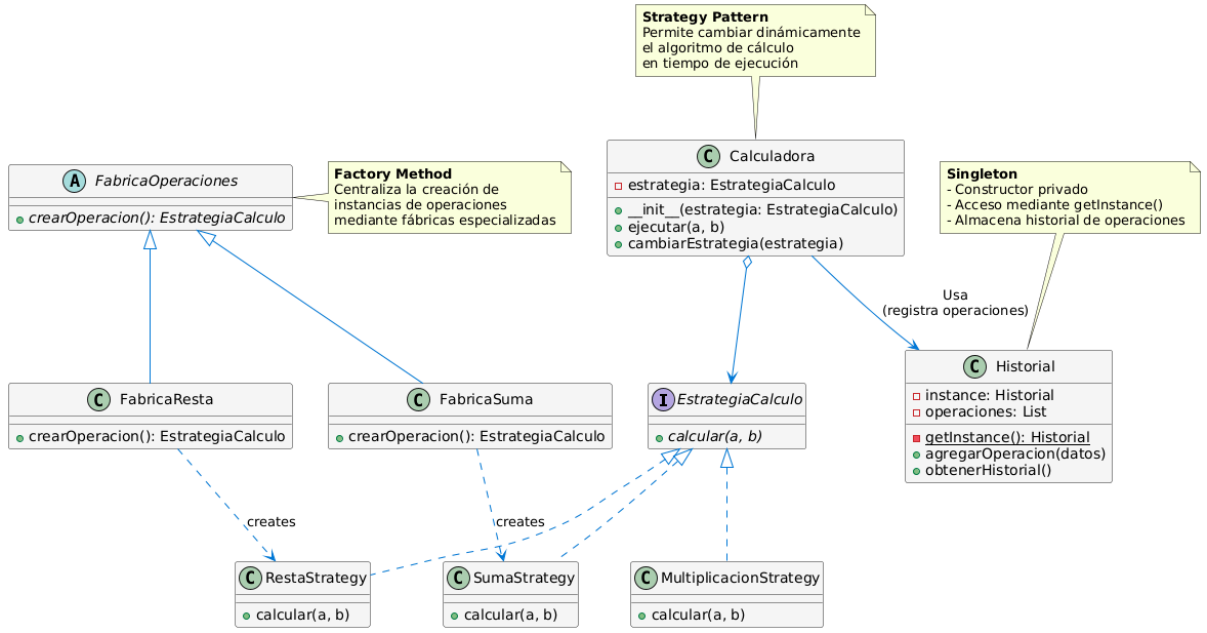
```
# Ejemplo en Python
class EstrategiaCalculo:
    def calcular(self, a, b):
        pass

class SumaStrategy(EstrategiaCalculo):
    def calcular(self, a, b):
        return a + b
```

```
class Calculadora:
    def __init__(self, estrategia: EstrategiaCalculo):
        self.estrategia = estrategia

    def ejecutar(self, a, b):
            return self.estrategia.calcular(a, b)
```

Diagrama



Singleton (Para el Historial)

javascript

```
// Ejemplo en JavaScript
class Historial {
    static instancia;

    constructor() {
        if (!Historial.instancia) {
            this.operaciones = [];
            Historial.instancia = this;
        }
        return Historial.instancia;
    }

    agregar(op) {
        this.operaciones.push(op);
    }
}
```

Contratos de Métodos Críticos

Para asegurar la correcta ejecución de las operaciones, se establecen contratos con:

Parámetros esperados.

Restricciones previas (precondiciones) como no dividir entre cero.

Resultados esperados (postcondiciones) con ejemplos concretos.

Esto ayuda a validar el comportamiento esperado y prevenir errores comunes.

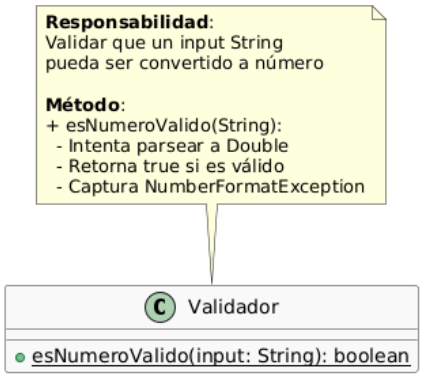
Operaciones Matemáticas

Método	Parámetros	Retorno	Precondiciones	Postcondiciones
execute(a: Number)	a: Ángulo en grados.	Number	-360 <= a <= 360	-1 <= retorno <= 1 (para seno/coseno)
execute(a: Number, b: Number)	a, b: Operandos.	Number	b != 0 (si es división)	retorno == a + b (ejemplo)

Validación de Input

java

```
public class Validador {
    public static boolean esNumeroValido(String input) {
        try {
            Double.parseDouble(input);
            return true;
        } catch (NumberFormatException e) {
            return false;
        }
    }
}
```



Modelado de Estados (Para Operaciones Complejas)

El sistema tiene estados definidos para reflejar la interacción del usuario con la calculadora:

Inactiva: No se ha ingresado ninguna expresión.

Validando: El sistema analiza la entrada.

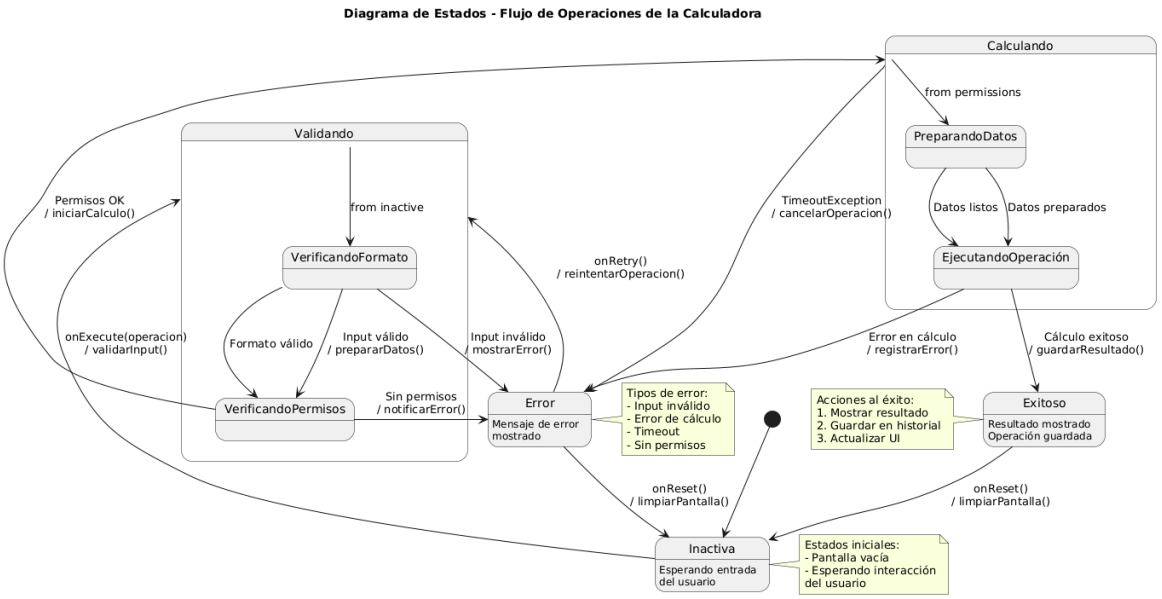
Calculando: Se realiza la operación.

Exitoso: Resultado válido mostrado.

Error: Entrada inválida o problema en el cálculo.

Cada transición entre estados está gobernada por eventos disparados desde la interfaz o el backen.

Diagrama de Estados (UML)



Transiciones de Estado

Estado	Evento	Acción	Nuevo Estado
Inactiva	onExecute(op)	Validar input.	Activa

Activa	onSuccess(result)	Guardar resultado.	Exitoso
Activa	onError(exception)	Mostrar mensaje de error.	Error

Ejemplos de Código por Lenguaje

El diseño lógico es agnóstico al lenguaje, pero la implementación concreta usa:

TypeScript/JavaScript en el frontend.

Java, Python o Node.js para la lógica en el backend.

Esto permite mapear las clases y métodos definidos en UML directamente a código en distintos lenguajes sin pérdida semántica.

Frontend (TypeScript)

typescript

```
// Clase para operaciones trigonométricas
class TrigCalculator {
    static seno(grados: number): number {
        return Math.sin(grados * Math.PI / 180);
    }
}
```

Backend (C#)

csharp

```
// Patrón Factory para operaciones
public interface IOperation {
    double Calculate(params double[] args);
}

public class Division : IOperation {
    public double Calculate(params double[] args) {
        if (args[1] == 0) throw new DivideByZeroException();
        return args[0] / args[1];
    }
}
```

Principios SOLID Aplicados

Single Responsibility: Cada clase tiene una única responsabilidad (por ejemplo, `Operación Básica` no gestiona historial).

Open/Closed: Se pueden agregar nuevas operaciones sin modificar código existente.

Liskov Substitution: Las clases hijas (`Operación Básica`, `OperacionAvanzada`) pueden sustituir a la clase padre (`Operación`) sin alterar el comportamiento del sistema.

Interface Segregation y Dependency Inversion se aplican especialmente en la separación del frontend y backend mediante interfaces claras.

1. Single Responsibility:
    - OperacionBasica solo sabe sumar/restar.
2. Open/Closed:
    - Nuevas operaciones (ej: Potencia) extienden Operacion.
3. Liskov Substitution:
    - OperacionAvanzada puede reemplazar a Operacion sin romper el sistema.

Escalabilidad del Diseño

El modelo lógico está preparado para crecer con nuevas funcionalidades:

Extensión para Nuevas Operaciones:

javascript

```
class OperacionPersonalizada extends Operacion {
  constructor(private formula: (x: number) => number) {
    super("custom");
  }

  execute(x: number): number {
    return this.formula(x);
  }
}

// Uso:
const op = new OperacionPersonalizada(x => x ** 2 + 2);
```