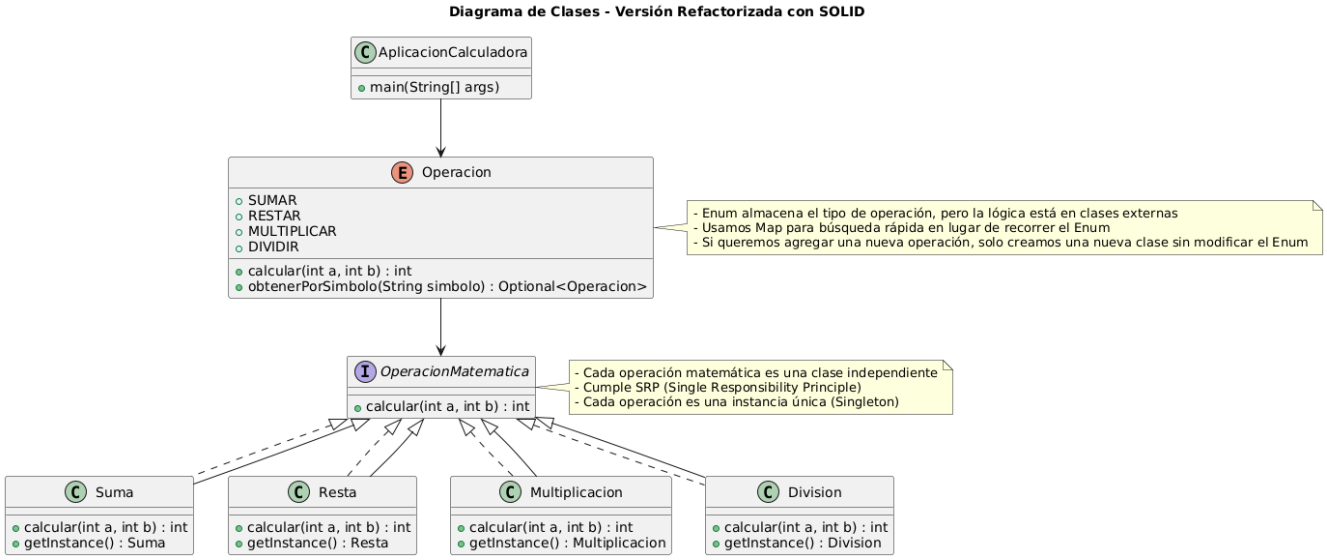
Resumen

Este documento describe la evolución y mejora de un sistema de operaciones matemáticas, aplicando principios **SOLID**, el patrón **Estrategia**, el uso de **Supplier**, y pruebas unitarias (**TDD**). Se comparan dos versiones del diseño y se justifica la refactorización para mejorar la extensibilidad, robustez y calidad del código.

Versión 1: Diseño Inicial sin refactoring como la version 2, mas debajo.



Problemas en la Versión 1

Violación de OCP (Open/Closed Principle) → Si se agrega una nueva operación, se debe modificar el `Enum`, lo que rompe la extensibilidad.
No hay manejo de excepciones → No se valida si los valores son correctos antes de calcular.
No hay pruebas unitarias (TDD) → No se verifica si el código funciona correctamente.
No hay separación de responsabilidades → `Enum Operacion` maneja tanto la lógica de selección como la ejecución de operaciones.

Versión 2; CALCULADORA REFACTORING

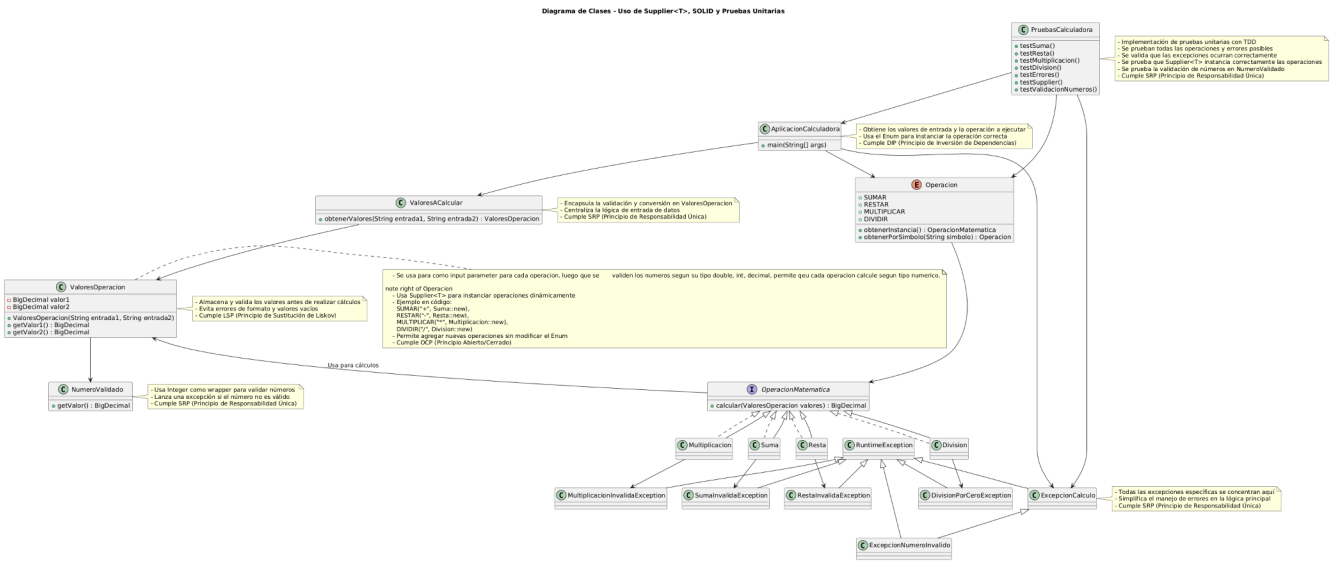
La clase `ValoresOperacion` encapsula dos valores numéricos (`valor1` y `valor2`) que se utilizarán en operaciones matemáticas dentro de la calculadora. Su propósito es garantizar que los valores sean representados correctamente como objetos `BigDecimal`, evitando errores de precisión y formato en cálculos aritméticos.

Responsabilidades

- **Conversión de entrada:** Recibe valores en formato `String` y los convierte a `BigDecimal` para cálculos precisos.
- **Encapsulación de datos:** Almacena los valores de manera segura, evitando modificaciones accidentales.
- **Interfaz de acceso:** Proporciona métodos para recuperar los valores (`getValor1()` y `getValor2()`).

Relación con otras clases

- **`ValoresACalcular`**: Se encarga de obtener y validar los valores antes de instanciarlos en `ValoresOperacion`.
- **`OperacionMatematica`**: Utiliza `ValoresOperacion` como entrada para realizar cálculos matemáticos.
- **`NumeroValidado`**: Puede validar los valores antes de que sean utilizados en operaciones.



Justificación.

✔ **Cumple mejor SOLID** → Separa responsabilidades, evita modificar el `Enum` y maneja errores correctamente.
✔ **Mayor extensibilidad** → Se pueden agregar nuevas operaciones sin modificar el código existente.
✔ **Mayor robustez** → Maneja excepciones y valida números antes de calcular.
✔ **Mayor calidad de código** → Se prueba con TDD para asegurar que todo funcione correctamente.
✔ **Uso del metodo** `Supplier` desde el Enum "Operacion" ejemplo: (**Suma::new**)→ Permite instanciar operaciones dinámicamente sin modificar el `Enum`.
✔ **Validación de números con** `NumeroValidado` → Evita errores de formato antes de calcular.
✔ **Pruebas unitarias (**`PruebasCalculadora`**)** → Se verifica que el código funcione correctamente.

CONCLUSION

La version 2, es mejor opcion, ya que cumple con los principios.

1 **Cumple SOLID** → Separa responsabilidades, evita modificar el `Enum` y maneja errores correctamente.

2 **Es más extensible** → Se pueden agregar nuevas operaciones sin modificar el código existente.

3 **Es más robusto** → Maneja excepciones y valida números antes de calcular.

4 **Tiene pruebas unitarias (TDD)** → Se asegura que el código funcione correctamente.