Task _1

Smell 1 — Duplicated code

Where: file content shows `BufferedImageCustom`, `Calculator`,

`SimpleJavaCalculator` repeated twice in the same file (or across files).

- Type: Duplicate code
- Why problem: Confusing, increases maintenance cost, compiler/package issues (multiple package declarations), may break build.
- Fix: Remove duplicates

Smell 2 — God object

- Where: `Calculator` class.
- Type: Single class handling many responsibilities: parsing stateful binary operations, many unary operations, special-case logic for trig handling, and side-effect-like control via returning `NaN`.
- Why problem: Violates Single Responsibility Principle (SRP) — hard to extend (add a new operator) and hard to test. Also `calculateBi` both mutates state and computes; mixing concerns.
- Fix: Extract operator behaviors into separate classes implementing a common interface (Strategy).

Smell 3 — Primitive Obsession & Conditional Logic

- Where: `calculateMono` and `calculateBiImpl` in `Calculator` — big if/else chains using enums.
- Type: Primitive obsession / long conditional (switch/if-else) — violates Open/Closed.
- Why problem: To add a new operator you must edit these big methods. Hard to unit test single operations.

Smell 4 — Resource handling not null-safe

- Where: `BufferedImageCustom.imageReturn()` — `getResourceAsStream` result used without null check.
- Type: Dangerous API usage / Lack of error handling.
- Why problem: If resource missing, `ImageIO.read(bis)` will NPE or IOException. App may crash at runtime.

Task _2

a) Violated principles

- Single Responsibility (SRP): `Calculator` does too many things. UI likely mixes view/controller actions.
- Open/Closed (OCP): `calculateMono` / `calculateBiImpl` are long `if` chains — extension requires code changes.
- Dependency Inversion (DIP): High-level UI / controller depends on concrete `Calculator` implementation details rather than abstractions.
- Interface Segregation (ISP): `Calculator` exposes many behaviors and internal state — clients only need parts of it.

b) Patterns to apply & why

PatternName:StrategyPattern and also we can appy Fectory pattern

3. 1.ConcreateStretegyMul

4.ContextClass

5.StrategyClass

6.StrategyPatternDemo

1. Strategy Pattern — for operators (binary and unary).
    - Problem addressed: Replaces long `if/switch` and enables easy extension.
    - Refactor targets: `Calculator` → `CalculatorEngine`

Task 3

Why this feature: It's a realistic calculator feature, demonstrates use of Command pattern plus History manager, and clearly uses design pattern(s) in a justifiable way.

Feature description

- **What:** Maintain a history list of completed calculations (input expression and result). Add UI area (a list panel or dropdown) to display history. Provide:
    - **Undo last operation** button — undoes last executed Command (if reversible) using Command pattern.
    - **Click history item** to repopulate display.

- ## Folder name :Operator

File name:

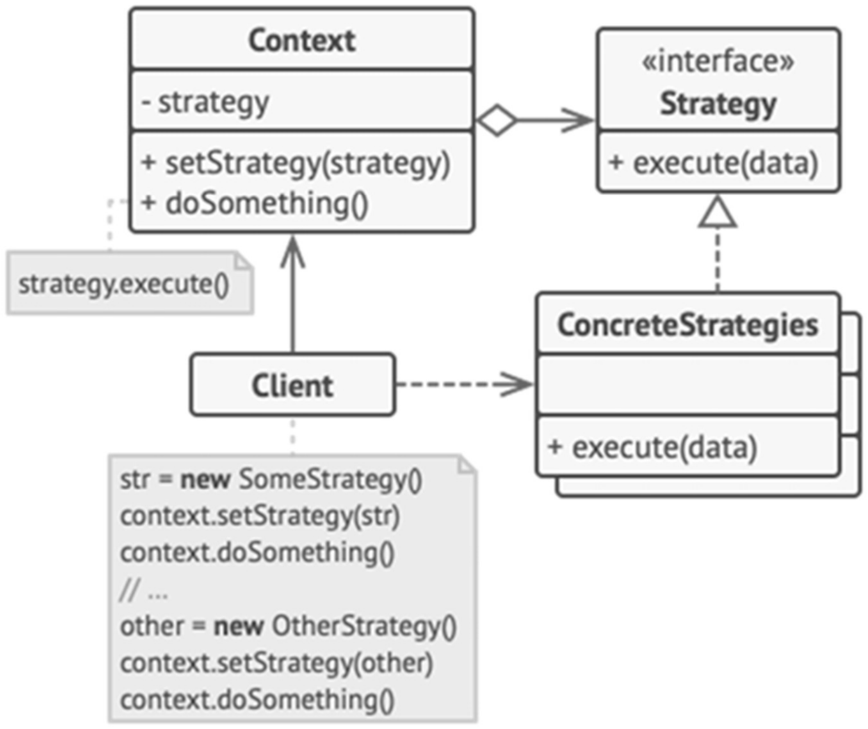
simplejavacalculator.java

CalculatorEngine.java

OperationCommand.java

ResourceLoader.java

Strategy UML diagram:


Context
- strategy
+ setStrategy(strategy)
+ doSomething()
strategy.execute()
«interface»
Strategy
+ execute(data)
ConcreteStrategies
+ execute(data)
Client
str = **new** SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = **new** OtherStrategy()
context.setStrategy(other)
context.doSomething()