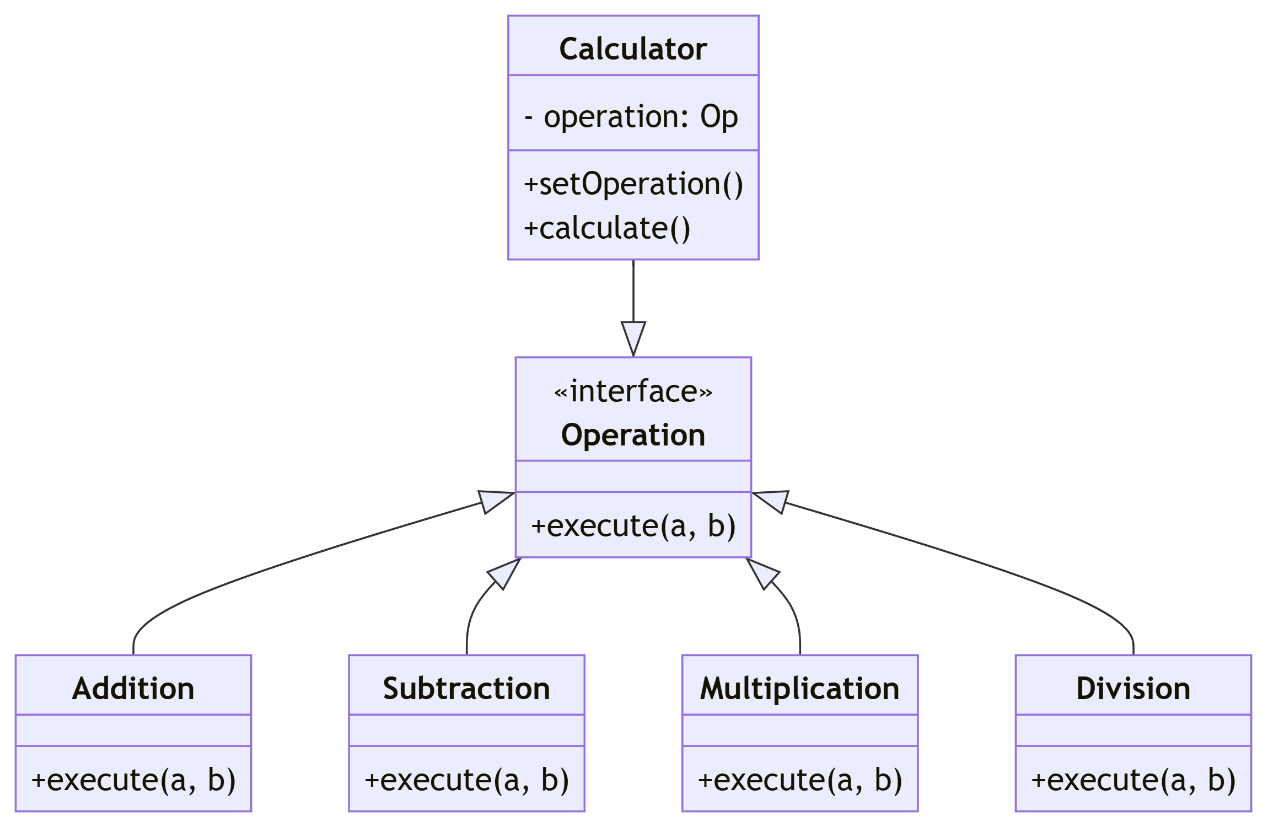
在这个例子中，最合适的设计模式是**策略模式**。策略模式可以定义一系列算法，并让它们在运行时可以互换使用。在这个例子中，可以将每个运算符（加、减、乘、除）抽象为一个独立的策略类，然后使用面向对象的方式来实现计算器。

**类图：**



|  |
| --- |
| **核心代码** |
| #include <iostream>  using namespace std;  // 抽象类，定义一个运算方法  class Operation {  public:  virtual double execute(int a, int b) = 0; // 纯虚函数  };  // 加法策略  class Addition : public Operation {  public:  double execute(int a, int b) override {  return a + b;  }  };  // 减法策略  class Subtraction : public Operation {  public:  double execute(int a, int b) override {  return a - b;  }  };  // 乘法策略  class Multiplication : public Operation {  public:  double execute(int a, int b) override {  return a \* b;  }  };  // 除法策略  class Division : public Operation {  public:  double execute(int a, int b) override {  if (b == 0) {  cerr << "Error: Division by zero" << endl;  return 0;  }  return (double)a / b;  }  };  // 计算器类，使用策略模式  class Calculator {  private:  Operation\* operation;  public:  // 设置当前使用的运算策略  void setOperation(Operation\* op) {  operation = op;  }  // 计算结果  double calculate(int a, int b) {  if (operation != nullptr) {  return operation->execute(a, b);  } else {  cerr << "Error: Operation not set" << endl;  return 0;  }  }  };  int main() {  int numa, numb;  char oper;  double result;  Calculator calculator;  cout << "输入两个操作数: ";  cin >> numa >> numb;  cout << "输入操作 (+, -, \*, /): ";  cin >> oper;  // 根据用户输入设置合适的运算策略  switch (oper) {  case '+':  calculator.setOperation(new Addition());  break;  case '-':  calculator.setOperation(new Subtraction());  break;  case '\*':  calculator.setOperation(new Multiplication());  break;  case '/':  calculator.setOperation(new Division());  break;  default:  cerr << "Invalid operation!" << endl;  return -1;  }  result = calculator.calculate(numa, numb);  cout << "Result: " << result << endl;  return 0;  } |