Educational Offering

Design Pattern Primer

Exercise: Applying Strategy Pattern

Overview

In this exercise, you will apply the Strategy design pattern to the Calculator implementation.

User Requirement

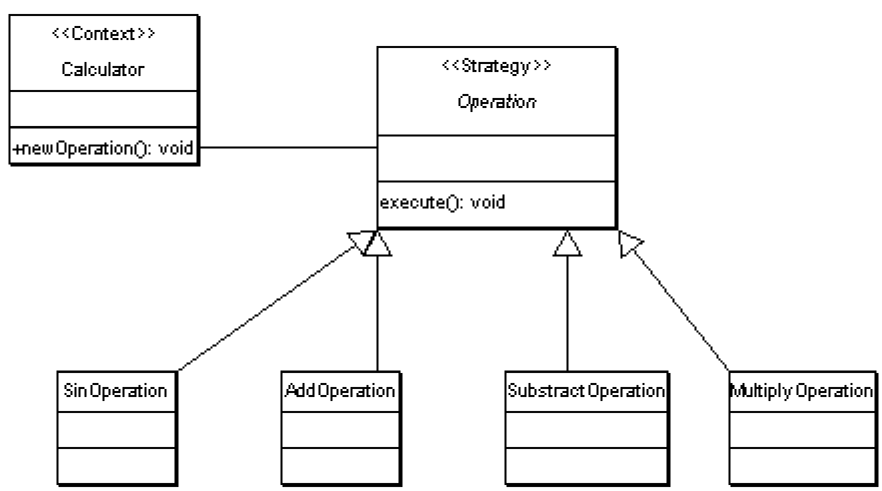
You need to utilize Java compiler and byte code interpreter to create and compile classes, methods, and fields.

Introduction

The previous calculator implementation utilized inheritance to define different calculator types. This exercise will apply the Strategy pattern so that individual groupings of operations can be configured with instances of a single calculator type.

This is accomplished by defining a separate hierarchy of operation classes. Each class represents a calculator operation. An abstract Operation class is defined with an abstract execute methods. Concrete operations classes provide an implementation of this method.

UML for the basic design is shown below:



Source for this exercise is defined in the db.lab.strategy package.

Exercise Instructions

1. Create Operation Hierarchy
In the db.lab.strategy package, implement a base class named Operation class, and then implement the concrete operation classes shown below:

```
public class AddOperation extends Operation{

	public AddOperation(){
		super("+");
	}

	public double execute(double leftValue,double rightValue) {

		return leftValue + rightValue;

	}

}

public class SinOperation extends Operation {

	/**
	 * Constructor for SinOperation.
	 * @param op
	 */
	public SinOperation() {
		super("sin");
	}

	/**
	 * @see dp.lab.strategy.Operation#execute(double, double)
	 */
	public double execute(double leftvalue, double rightValue) {
		return Math.tan(rightValue);
	}

}
```

2. Continue to extend the Operation class with SubtractOperation, DivideOperation, MultiplyOperation, and TanOperation and implement the appropriate execute methods. Notice that each operation class implements a constructor that initializes the name of the operation. The name specified for the operation is the name used to execute the operation. This will become more clear in the next step.

3. Testing the Calculator design
Since a single Calculator class will represent different calculator types, a set of Operation instances will have to be initialized to a calculator instance. A later exercise will provide a better mechanism, but for this exercise, operation instances will be installed in the test script implementation.

The source below implements a test class with an executable main method that creates a basic calculator instance.

```
public class Tester {

      public static void main(String[] args) {

            // basic calculator

            System.out.println("* * Basic Calculator * *");

            Calculator calc = new Calculator();
            // install operations

            calc.install(new AddOperation());
            calc.install(new SubtractOperation());
            calc.install(new MultiplyOperation());
            calc.install(new DivideOperation());

            // execute operations

            calc.execute("+",10.0);
            calc.print();
            calc.execute("+",10.0);
            calc.print();
            calc.execute("-",10.0);
            calc.print();
            calc.execute("/",2.0);
            calc.print();
  }

}
```

4. Modify the test class with the source shown below, implementing a scientific calculator. Execute the main method.

```
      // Scientific Calculator

            calc = new Calculator();

            calc.install(new AddOperation());
            calc.install(new SubtractOperation());
            calc.install(new MultiplyOperation());
            calc.install(new DivideOperation());
            calc.install(new SinOperation());
            calc.install(new TanOperation());
            calc.install(new LogOperation());

            System.out.println("* * Scientific Calculator * *");

            //calc.log(10.0)
            calc.execute("+",10.0);
            calc.execute("sin",20.0);
            calc.print();
```

```
        calc.execute("log",100);
        calc.print();
```

*** The programmers calculator is supported yet, later patterns will provide support for alternative operations.

*** Study the implementation to see how operations are obtained and executed ****