Educational Offering

Design Pattern Primer

Exercise: Prototype Factory

Overview
In this exercise, you will create different types of calculators using the Prototype pattern.

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

Introduction
This exercise will describe how the Prototype pattern can be applied to produce calculator instances that support a specific grouping of operations. Instances of calculator types are created by requesting a calculator type from a prototype factory. The factory creates a new prototype by returning a copy of the relevant calculator instance. The calculator instance in this exercise defines a copy() method that returns a new instance of itself along with a copy of the appropriate state.

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

Exercise Instructions

1. Create PrototypeFactory class
In the db.lab.prototype package implement the PrototypeFactory class shown below. Notice the usage of the copy() method defined in the CalculatorP class.

```
import dp.lab.strategy.*;

public class PrototypeFactory {

      static CalculatorP basic = null;
      static CalculatorP scientific = null;

      static {

            // Create basic calculator
            basic = new CalculatorP();

            // install operations

            basic.install(new AddOperation());
```

```
        basic.install(new SubtractOperation());
        basic.install(new MultiplyOperation());
        basic.install(new DivideOperation());

        scientific = basic.copy();
        scientific.install(new SinOperation());
        scientific.install(new TanOperation());
        scientific.install(new LogOperation());

    }

    public static CalculatorP basic() {

        return basic.copy();

    }

    public static CalculatorP scientific() {
        return scientific.copy();
    }

}
```

2. Implement the tester class shown below. This class utilizes the prototype factory to produce basic and scientific calculator instances.

```
import dp.lab.strategy.*;

public class Tester {

    public static void main(String[] args) {

        // basic calculator

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

        CalculatorP calc = PrototypeFactory.basic();
        // install operations

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

        calc = PrototypeFactory.scientific();

        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();

     }

}
```

3. Implement Custom Calculator
The prototype behavior can be exploited to create custom calculators based upon existing calculators. Add the expressions below to the tester class.

```
// Custom calculator

System.out.println("* * Custom Calculator * *");
calc = calc.copy();
calc.install(new TanOperation());
calc.execute("tan",20.00);
calc.print();
```