Educational Offering

Design Pattern Primer

Exercise: State

Overview

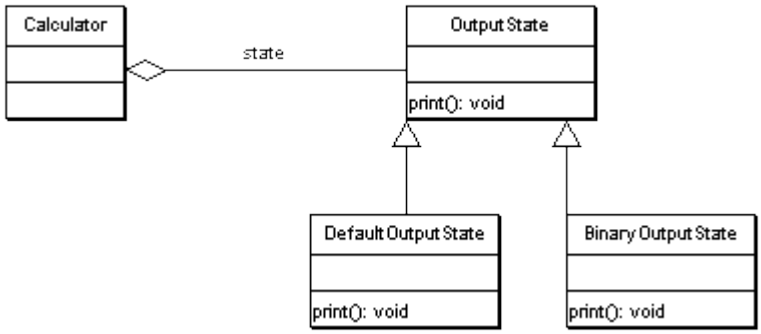
In this exercise, you will apply the State pattern to affect display-formatting capabilities of *Calculator* instances.

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 State pattern is applied to the calculator to support operation specific display behavior. The Calculator will be assigned a concrete *OutputState* object, and when sent the *print()* method, will forward the request to the current state for printing. UML is shown below.



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

Exercise Instructions

1. Execute the Test Script

In the ***db.lab.state*** package execute the Tester class and study the *Calculator print()* implementation. Notice how it forwards to the current state, which happens to be an instance of the *DefaultOutputState* class.

The binary operation changes simply changes the state of the *Calculator* to reference an instance of a *BinaryOutputState* instance. Implement the class *BinaryOutputState* class definition shown next.

```
	public class BinaryOutputState extends OutputState {

	/**
	 * @see dp.lab.state.OutputState#print(Calculator)
	 */
	public void print(Calculator calc) {

	    int itemp = new Integer((int) calc.getLeftValue()).intValue();
	      String sresult = " (bin) " + Integer.toBinaryString(itemp);
	      sresult += calc.getOperation();
	      itemp = new Integer((int) calc.getRightValue()).intValue();
	      sresult += Integer.toBinaryString(itemp);
	      sresult += " = ";
	      itemp = new Integer((int) calc.getResult()).intValue();
	      sresult += Integer.toBinaryString(itemp);

	      System.out.println(sresult);
	}

}
```

The Calculator's output state is transitioned to a BinaryOutputState when the BinaryOperation is executed. The *BinaryOperation* implementation is shown below, implement and take note of the execute() method.

```
public class BinaryOperation extends Operation {

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

	/**
	 * @see dp.lab.state.Operation#execute(Calculator)
	 */
	public void execute(Calculator calc) {
		calc.state = new BinaryOutputState();
	}

}
```

2. Configuring the Operation

The *BinaryOperation* is added to the programmers calculator by adding the highlighted code to the *PrototypeFactory* static initialization block

```
	static {

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


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

```
            // create programmer calculator
            programmer = basic.copy();
            programmer.install(new BinaryOperation());

    }
```

3. Execute the test class

Make sure the code below appears in the Tester classes main method, after the scientific calculator expressions, then execute the main method and review results.

```
// Programmer Calculator

      calc = PrototypeFactory.programmer();

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


      calc.execute("+", 10.0);
      calc.execute("+", 20.0);
      calc.execute("bin");
      calc.print();
```

4. Create other operations

Using the steps above, create Hex, Octal, and Decimal operations; add them to the factory and test in the testing class.