```java
package operator;

/**
 * This is the root class of the hierachy of all operators. All the hierarchy
 * contains checkShouldExecute and exec methods: checkShouldExecute checks if an
 * operator (or a group of operator) should be executed, exec execute the
 * operator (in the deepest defition in the hierarchy). The only non-abstract
 * method (execute) calls the checkShouldExecute() on the operator and then if
 * necessary the exec method().
 * <p>
 * --- Hierarchy --- (lexicographic) <br>
 * Operator <br>
 * --CalculOperator<br>
 * ----OneOperandCalculOperator<br>
 * ------SignOperator<br>
 * ------SqrtOperator<br>
 * ------SquareOperator<br>
 * ------OneOverXOperator<br>
 * ----TwoOperandCalculOperator<br>
 * ------DivOperator<br>
 * ------MinusOperator<br>
 * ------PlusOperator<br>
 * ------TimesOperator<br>
 * --ControlOperator <br>
 * ----BackSpaceOperator<br>
 * ----CEOperator<br>
 * ----COperator<br>
 * ----EnterOperator<br>
 * ----MROperator<br>
 * ----MSOperator<br>
 * --NumberOperator<br>
 * ----DigitOperator<br>
 * ----DotOperator<br>
 * <p>
 * Note: only the leaf are non-abstract, all the other are abstract.
 *
 * @author Sacha Bron
 * @author Valentin Minder
 */

 // multi-page printing of all operators TO SAVE TREES !!!

import main.State;

public abstract class Operator {
    public void execute() {
        if (checkShouldExecute()) {
            exec();
        }
    }

    abstract void exec();

    abstract boolean checkShouldExecute();
```

```
55 }
56
57 public abstract class CalculOperator extends Operator {
58     abstract void exec();
59     abstract boolean checkShouldExecute();
60 }
61
62 public abstract class OneOperandCalculOperator extends CalculOperator {
63     abstract void exec();
64     boolean checkShouldExecute() {
65         // checks that a one-operand-operator could be executed
66         return State.getInstance().beforeOneOperand();
67     }
68 }
69
70 public class SignOperator extends OneOperandCalculOperator {
71     void exec() {
72         State.getInstance().inverseSign();
73     }
74 }
75
76 public class SqrtOperator extends OneOperandCalculOperator {
77     void exec() {
78         State.getInstance().operandSqrt();
79     }
80 }
81
82 public class SquareOperator extends OneOperandCalculOperator {
83     void exec() {
84         State.getInstance().operandSquare();
85     }
86 }
87
88 public class OneOverXOperator extends OneOperandCalculOperator {
89     void exec() {
90         State.getInstance().operandOver();
91     }
92 }
93
94 public abstract class TwoOperandCalculOperator extends CalculOperator {
95     abstract void exec();
96
97     boolean checkShouldExecute() {
98         // checks that a two-operands-operator could be executed
99         return State.getInstance().beforeTwoOperands();
100     }
101 }
102
103 public class DivOperator extends TwoOperandCalculOperator {
104     void exec() {
105         State.getInstance().operandDiv();
106     }
107 }
108
```

```
109 public class MinusOperator extends TwoOperandCalculOperator {
110     void exec() {
111         State.getInstance().operandMinus();
112     }
113 }
114
115 public class PlusOperator extends TwoOperandCalculOperator {
116     void exec() {
117         State.getInstance().operandPlus();
118     }
119 }
120
121 public class TimesOperator extends TwoOperandCalculOperator {
122     void exec() {
123         State.getInstance().operandTimes();
124     }
125 }
126
127 public abstract class ControlOperator extends Operator {
128     abstract void exec();
129
130     boolean checkShouldExecute() {
131         // they are always allowed!
132         return true;
133     }
134 }
135
136 public class BackSpaceOperator extends ControlOperator {
137     void exec() {
138         State.getInstance().controlBackSpace();
139     }
140 }
141
142 public class CEOperator extends ControlOperator {
143     void exec() {
144         State.getInstance().controlClearError();
145     }
146 }
147
148 public class COperator extends ControlOperator {
149     void exec() {
150         State.getInstance().controlClear();
151     }
152 }
153
154 public class EnterOperator extends ControlOperator {
155     void exec() {
156         State.getInstance().controlEnter();
157     }
158 }
159
160 public class MROperator extends ControlOperator {
161     void exec() {
162         State.getInstance().controlMemoryRecall();
```

```
163     }
164 }
165
166 public class MSOperator extends ControlOperator {
167     void exec() {
168         State.getInstance().controlMemoryStore();
169     }
170 }
171
172 public abstract class NumberOperator extends Operator {
173     abstract void exec();
174
175     boolean checkShouldExecute() {
176         // checks that a numerical modifier could be executed
177         return State.getInstance().checkNumericalOperator();
178     }
179 }
180
181 public class DotOperator extends NumberOperator {
182
183     void exec() {
184         State.getInstance().addDot();
185     }
186 }
187
188 public class DigitOperator extends NumberOperator {
189
190     private int myValue = 0;
191
192     public DigitOperator(int value) {
193         myValue = value;
194     }
195
196     void exec() {
197         State.getInstance().addDigit(myValue);
198     }
199 }
200
```