SEW S03 / CALCULATOR

Implementierung des Strategy Patterns

12. NOVEMBER 2014

STEFAN ERCEG
4AHITT

Inhalt

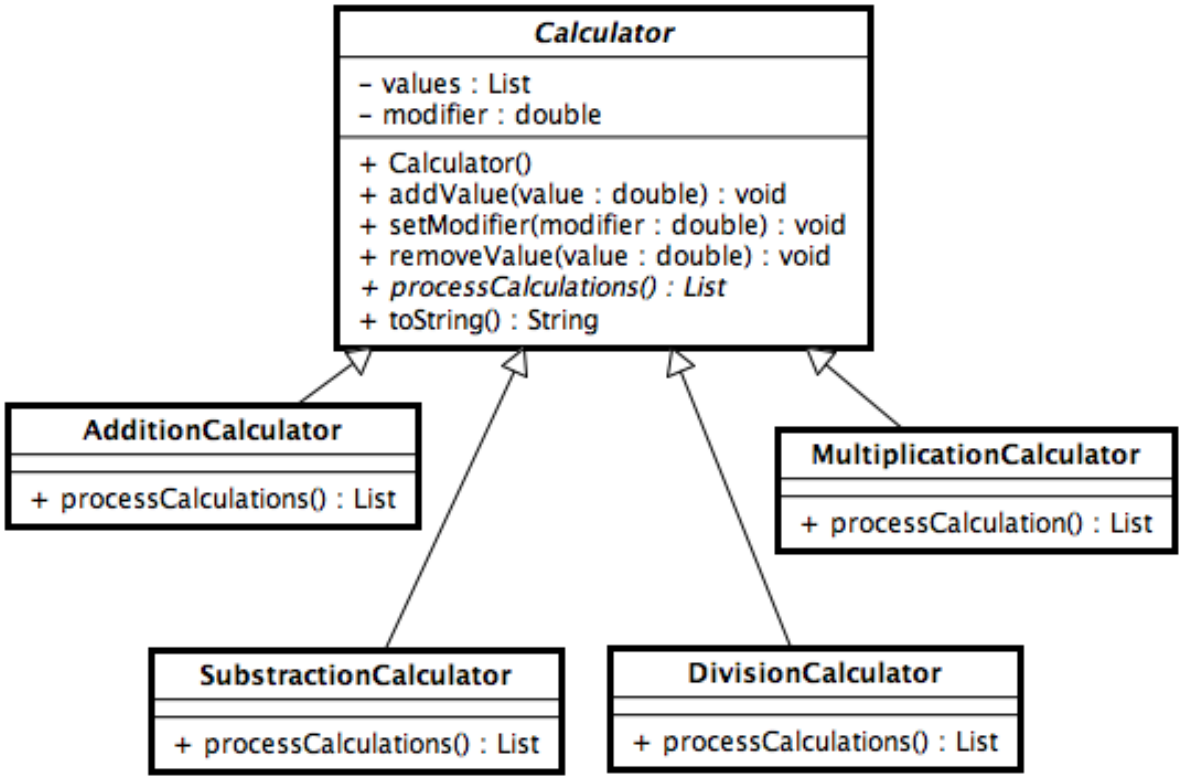
1. Github

Link: https://github.com/serceg-tgm/4AHITT-SEW/tree/master/S03

Tag: erceg_s03

2. Aufgabenstellung

Folgendes UML-Diagramm soll so geändert werden, dass es dem Strategy-Pattern entspricht:



Die abstrakte Klasse Calculator hat die Aufgabe, Werte aus einer Liste mit einem modifier zu verändern und das Ergebnis als neue Liste zurück zu geben. Dazu dient die abstrakte Methode processCalculations, die in den konkreten Subklassen so überschrieben wurde, dass sie je nach Klasse die Werte aus der Liste mit dem modifier addiert, subtrahiert, multipliziert oder dividiert.

Das veränderte UML-Diagramm wird implementiert.

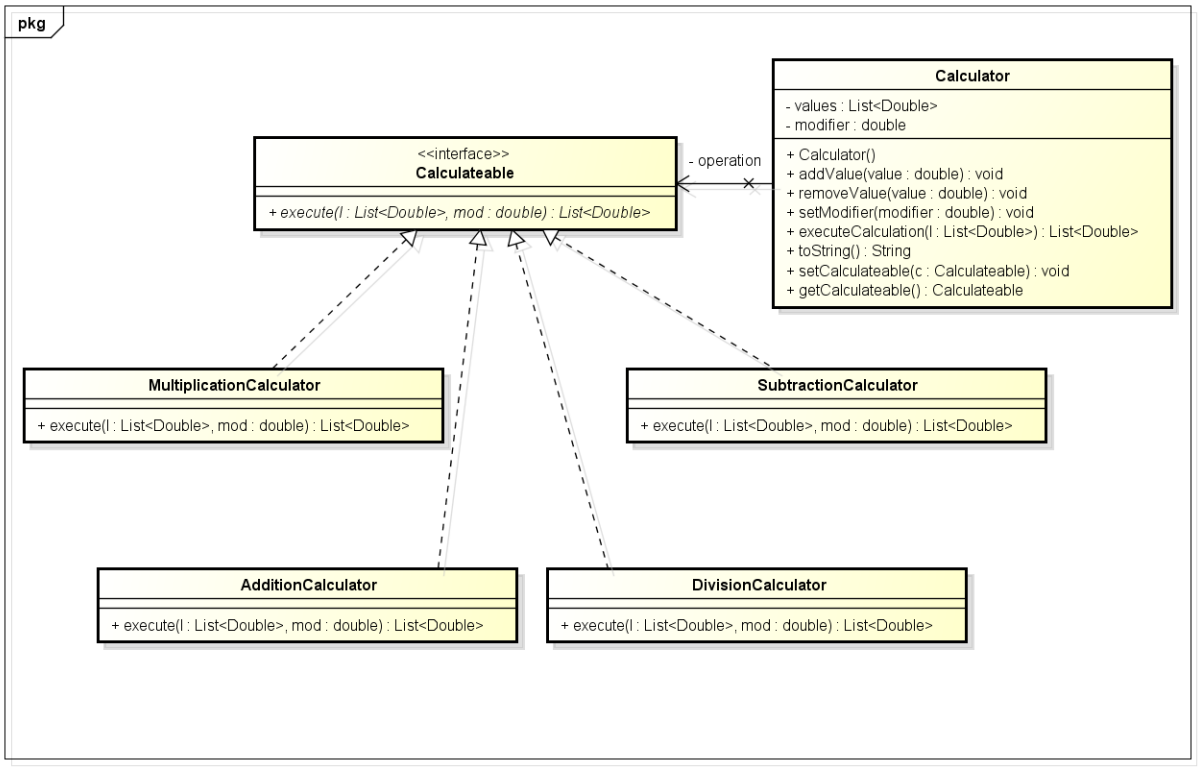
3. Zeitabschätzung

Teilaufgabe	Zeit
UML-Diagramm	20 Minuten
Codeerstellung inkl. Javadoc	60 Minuten
Programmtestung mittels JUnit	120 Minuten
Protokoll	60 Minuten
Gesamt	**260 Minuten (4 h 20 min)**

4. Tatsächlicher Zeitaufwand

Teilaufgabe	Zeit
UML-Diagramm	30 Minuten
Codeerstellung inkl. Javadoc	55 Minuten
Programmtestung mittels JUnit	150 Minuten
Protokoll	100 Minuten
Gesamt	**335 Minuten (5 h 35 min)**

5. Designüberlegung



Das UML-Diagramm wurde nun so geändert, dass es einem Strategy-Pattern entspricht.

Strategy-Patterns kommen prinzipiell zum Einsatz, wenn man mehrere Algorithmen für einen bestimmten Task besitzt und der Client während der Laufzeit entscheiden kann, welche Implementierung er verwenden möchte.

Bei dieser Aufgabe haben wir im unveränderten UML-Diagramm mehrere Klassen gehabt, die die Methode „processCalculations()“ von der Superklasse überschrieben haben. Im UML-Diagramm hätte die Methode bei den Subklassen überhaupt nicht stehen müssen, da die Vererbung schon darauf hingewiesen hat, dass alle Methoden auch in den Subklassen vorhanden sind.

Um die stark variablen Teile des Programms von den statischen zu trennen, wurde ein Interface „Calculateable“ erstellt, welches die Methode „execute“ besitzt, bei der die Liste mit den enthaltenden Zahlen und der Modifier als Parameter übergeben werden. Je nach Klasse, die das Interface implementiert, werden die Werte aus der Liste addiert, subtrahiert, dividiert oder multipliziert.

Ebenfalls wurde bei der Klasse „Calculator“ die Komposition bevorzugt, d.h. es wurde ein Attribut Calculateable erstellt, welches dann mittels der Methoden „setCalculateable“ und „getCalculateable“ gesetzt bzw. herausgelesen werden kann.

6. Arbeitsdurchführung

6.1. Implementierung einer ArrayList

Bei der Initialisierung des List-Attributs in der Klasse „Calculator“ musste im Konstruktor definiert werden, welche Liste nun genau implementiert werden soll. Ich entschied mich dabei für eine Array-List.

Genauso wurden in den Methoden „execute“ Klassen, die das Interface „Calculateable“ implementieren, Array-Lists erstellt, in die die neuen Werte hinzugefügt werden soll. Der Code schaut dabei folgendermaßen aus:

```
public List<Double> execute(List<Double> l, double mod) {

      /* eine neue ArrayList wird erstellt, in die alle neuen
      veränderten Werte hinzugefügt werden */

      ArrayList<Double> temp = new ArrayList<Double>();

      /* alle Werte aus der Liste werden durchgegangen und addiert */

      for (int i = 0; i < l.size(); i++) {

            /* die Werte werden in ein double-Datentyp gecastet */
            temp.add(i, ((double) l.get(i)) + mod);

      }

  return temp;

}
```

6.2. Aufhebung der abstrakten Klasse

Die Klasse „Calculator“ wird ebenfalls nicht mehr als abstrakte Klasse angesehen, da in der Methode „executeCalculation“ die Methode „execute“ von den jeweiligen Operationstypen-Klassen ausgeführt wird:

```
public void executeCalculation() {

      this.values = this.operation.execute(values, modifier);

}
```

6.3. getter- und setter-Methoden für den Operationstypen

Es wird eine setter-Methode benötigt, um den gewünschten Operationstyp zu setzen und ihn somit verwenden zu können. Dabei wird als Parameter der Operationstyp übergeben und auf das in der Calculator-Klasse vorhandene Attribut „Calculateable operation“ gesetzt:

```
public void setCalculateable(Calculateable c) {
	this.operation = c;
}
```

Gleich hinzu habe ich auch eine getter-Methode eingebaut, die einfach den aktuell verwendeten Operationstypen herausliest und zurückgibt.

```
public Calculateable getCalculateable() {
	return this.operation;
}
```

7. Testbericht

7.1. TestCalculator

In dieser Testklasse wurden die Methoden, die in der Klasse Calculator vorhanden sind, getestet.

Folgende Testfälle wurden durchgeführt:

- constructor:
  Es wurde überprüft, ob der Konstruktor nicht null ist, d.h. ob bestimmte Initialisierungen im Konstruktor durchgeführt wurden.
- addValue:
  Ein bestimmter Wert wurde zur Liste hinzugefügt und es wurde überprüft, ob der Wert nun in der Liste vorhanden ist.
- removeValue (Test 1):
  Bei diesem Testfall wurden 3 einzigartige Werte hinzugefügt und einer von diesen entfernt.
- removeValue (Test 2):
  Bei diesem Testfall wurden 3 Werte hinzugefügt, es befand sich jedoch ein doppeltvorkommender Wert in der Liste und dieser wurde entfernt. Gelöscht wurde dabei nur ein Element, nämlich das erstvorkommende.
- setCalculateable & getCalculateable:
  Ein Operationstyp wurde mittels der Methode „setCalculateable“ gesetzt und danach mittels „getCalculateable“ herausgelesen.
- setModifier:
  Hier wurde für den Modifier ein Wert gesetzt und überprüft, ob der Wert für das Attribut „modifier“ korrekt gesetzt wurde.

toString:
  Ein Wert wurde zur Liste hinzugefügt und der Modifier und die in der Liste verfügbaren Werte als String zurückgegeben.

executeCalculation:
  Der Modifier und 3 Werte wurden hinzugefügt. Als Operationstyp wurde die Addition ausgewählt und mittels der Methode „toString“ überprüft, ob die Werte sich entsprechend und korrekt verändert haben.

7.2. TestOperationtypes

In dieser Testklasse wurden jene Klassen getestet, die das Interface „Calculateable“ implementiert haben. Dazu zählen „AdditionCalculator“, „SubtractionCalculator“, „MultiplicationCalculator“ und „DivisionCalculator“. Bei diesen Klassen konnte nur die implementierte Methode „execute“ getestet werden und das Verhalten der jeweiligen Operationstypen bei bestimmten Einstellungen für den Modifier und die Werte in der Liste.

Testfälle für AdditionCalculator:

1.) Der Modifier und 3 Werte wurden hinzugefügt. Als Operationstyp wurde die Addition ausgewählt und mittels der Methode „toString“ überprüft, ob die Werte sich entsprechend und korrekt verändert haben.
2.) Der größte Double-Wert wurde als Modifier eingesetzt und auch in die Liste als Wert eingespeichert. Als Ergebnis ist eine unendliche Zahl herausgekommen.
3.) Der größte Double-Wert * -1 wurde als Modifier eingesetzt und der größte Double-Wert in die Liste als Wert eingespeichert. Das Ergebnis betrug 0.0.
4.) Der größte Double-Wert * -1 wurde als Modifier eingesetzt und auch in die Liste als Wert eingespeichert. Als Ergebnis ist eine unendliche (negative) Zahl herausgekommen.

Testfälle für SubtractionCalculator:

1.) Der Modifier und 3 Werte wurden hinzugefügt. Als Operationstyp wurde die Subtraktion ausgewählt und mittels der Methode „toString“ überprüft, ob die Werte sich entsprechend und korrekt verändert haben.
2.) Der größte Double-Wert wurde als Modifier eingesetzt und auch in die Liste als Wert eingespeichert. Das Ergebnis betrug 0.0.
3.) Der größte Double-Wert * -1 wurde als Modifier eingesetzt und der größte Double-Wert in die Liste als Wert eingespeichert. Als Ergebnis ist eine unendliche (negative) Zahl herausgekommen.
4.) Der größte Double-Wert * -1 wurde als Modifier eingesetzt und auch in die Liste als Wert eingespeichert. Das Ergebnis betrug 0.0.

Testfälle für MultiplicationCalculator:

1.) Der Modifier und 3 Werte wurden hinzugefügt. Als Operationstyp wurde die Multiplikation ausgewählt und mittels der Methode „toString“ überprüft, ob die Werte sich entsprechend und korrekt verändert haben.
2.) Der größte Double-Wert wurde als Modifier eingesetzt und auch in die Liste als Wert eingespeichert. Als Ergebnis ist eine unendliche Zahl herausgekommen.
3.) Der größte Double-Wert * -1 wurde als Modifier eingesetzt und der größte Double-Wert in die Liste als Wert eingespeichert. Als Ergebnis ist eine unendliche (negative) Zahl herausgekommen.

Testfälle für DivisionCalculator:

1.) Der Modifier und 3 Werte wurden hinzugefügt. Als Operationstyp wurde die Division ausgewählt und mittels der Methode „toString“ überprüft, ob die Werte sich entsprechend und korrekt verändert haben.
2.) Der größte Double-Wert wurde als Modifier eingesetzt und auch in die Liste als Wert eingespeichert. Das Ergebnis betrug 1.0.
3.) Der größte Double-Wert * -1 wurde als Modifier eingesetzt und der größte Double-Wert in die Liste als Wert eingespeichert. Das Ergebnis betrug -1.0.
4.) Als Modifier und als Wert wurden 0.0 eingetragen. NaN (Not a Number) wurde zurückgegeben.

7.3. TestRunner

Diese Klasse wurde implementiert, damit alle Testklassen und somit deren Testfälle beim Ausführen des erstellten Runnable-Jar-Files ausgeführt werden.

Dabei wurde eine SuiteClass definiert, in der alle class-Dateien der jeweiligen Testklassen definiert wurden.

In der main-Methode wurde mittels JUnitCore als Main-Klasse diese erstellte Klasse ausgewählt.

8. Lessons learned

Folgende Skills wurden bei dieser Aufgabe erlernt bzw. erweitert:

1.) Verwendung eines Strategy-Patterns
2.) Vertiefung der 3 Designprinzipien „auf Interfaces statt auf Implementierungen programmieren“, „die Komposition der Vererbung bevorzugen“ und „statische Teile des Programmcodes von stark variablen Teilen trennen“
3.) Verwendung von Collections (ArrayList)
4.) kennengelernt, wie die 4 Operationstypen bei bestimmten Eingaben reagieren