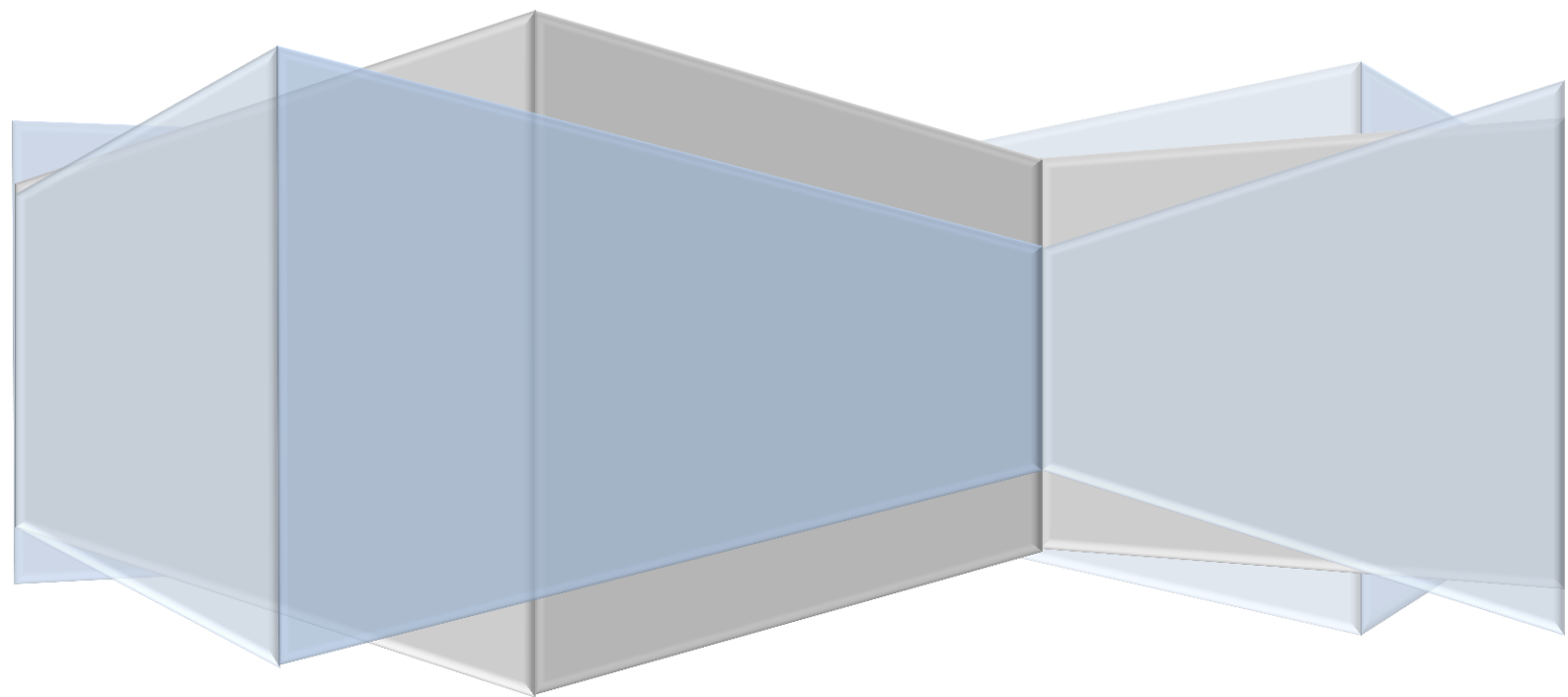


TGM Wien

S03 - Calculator

SEW 2014/2015 | Stand: 12.11.2014

Michael Weinberger



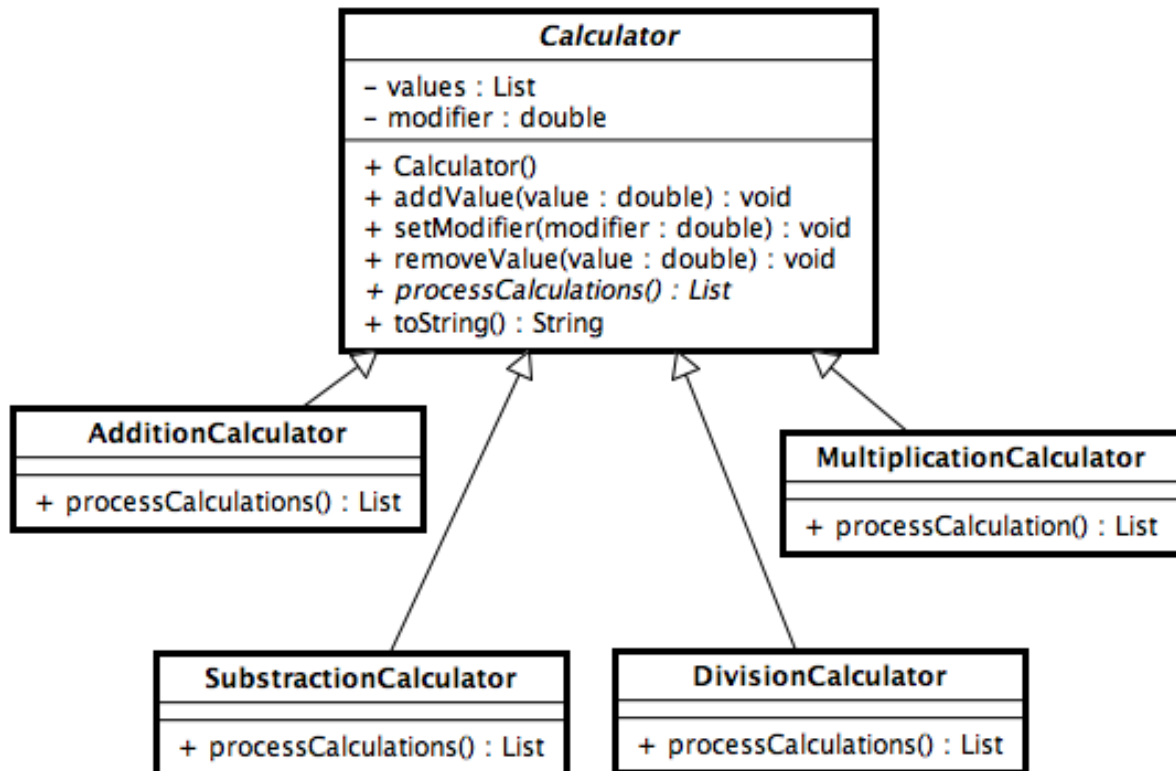
Inhalt

Aufgabenstellung.....	2
Beschreibung auf Moodle	2
Designüberlegung.....	3
Erster Ansatz.....	3
Konkrete Idee/Umsetzung	4
Detaillierte Arbeitsaufteilung (Aufwandsabschätzung, Endzeitaufteilung)	5
Aufgabentrennung	5
Aufwandabschätzung	5
Endzeitaufteilung	5
Fazit	5
Detaillierte Arbeitsaufteilung (Aufwandsabschätzung, Endzeitaufteilung)	6
Resultate.....	6
Niederlagen	6
Testbericht.....	7
Coverage.....	7
Beschreibung.....	7
Quellenangaben:	8

Aufgabenstellung

Beschreibung auf Moodle

Ändere folgendes UML-Diagramm so um, dass es dem Strategy-Pattern entspricht und implementiere dann dieses.

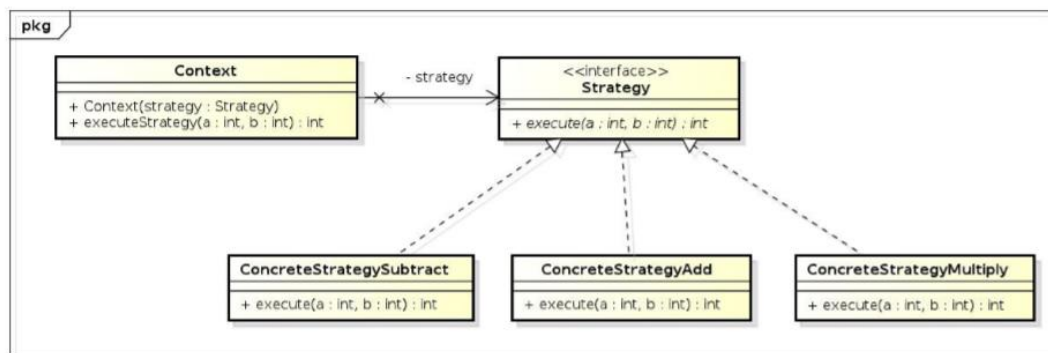


Die abstrakte Klasse Calculator hat die Aufgabe, Werte aus einer Liste mit einem modifier zu verändern und das Ergebnis als neue Liste zurück zu geben. Dazu dient die abstrakte Methode processCalculations, die in den konkreten Subklassen so überschrieben wurde, dass sie je nach Klasse die Werte aus der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

Die Abgabe ist den Meta-Regeln entsprechend, bis Mittwoch, 12. November 2014, 23:55 abzugeben.

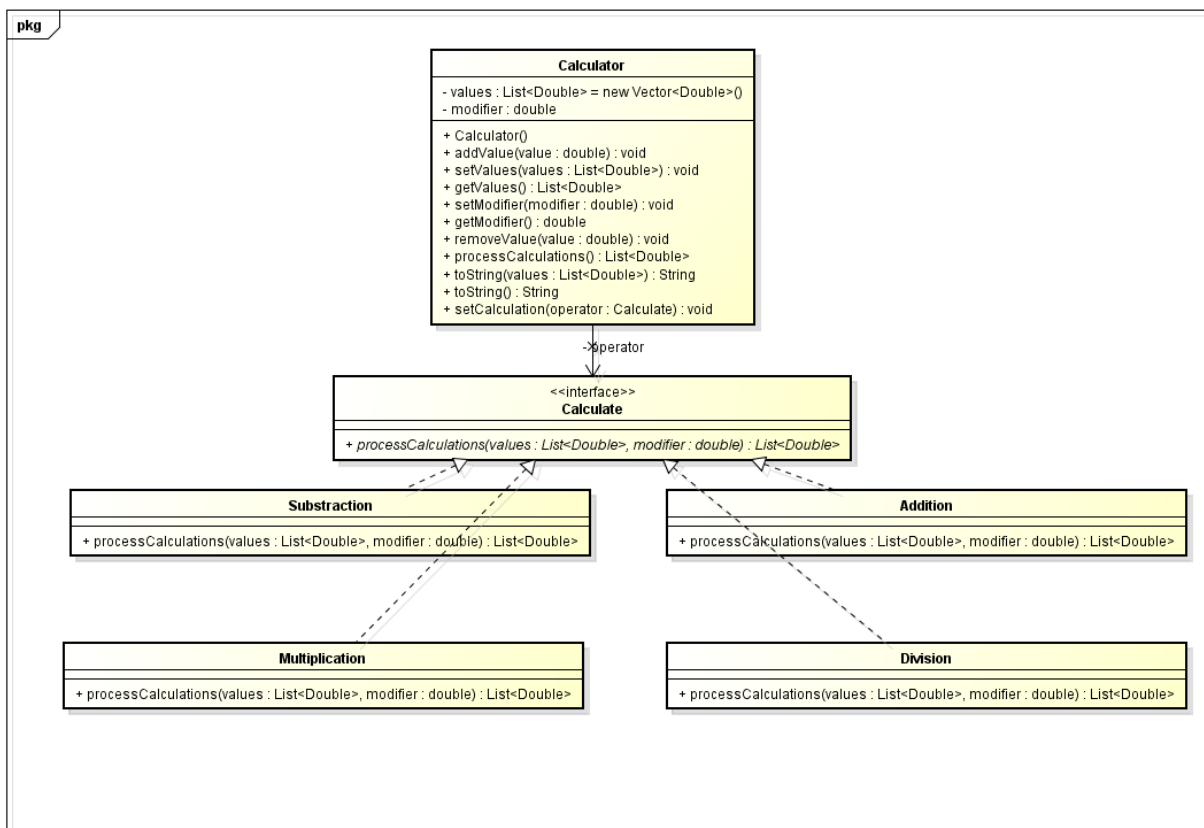
Das erste Entwurfsmuster

- Das **Strategy-Pattern** definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar.
- Algorithmus unabhängig von Clients variierbar



Dieses Beispiel aus 'Entwurfsmuster von Kopf bis Fuß' gab bereits gut erklärt (Duck!) vor, wie dieses Pattern umzusetzen ist und welchen Nutzen es bringen kann.

Konkrete Idee/Umsetzung



Anhand des oben genannten Beispiels konnte ich dieses UML-Diagramm erstellen. Durch Benutzen des Strategy-Patterns können nun theoretisch beliebig viele andere Rechenoperationen ohne Codeveränderung in anderen Klassen hinzugefügt werden dank des modularen Aufbaus.

Die 3 Prinzipien des Patterns lauten:

- Umsetzung auf Interfaces und nicht auf Implementierung
- Veränderbaren Code trennen von Gleichbleibenden (sog. statischen)
- Komposition vor Vererbung

Patterns im Allgemeinen können nicht als ‚Wunderwaffe‘ oder auch ‚Eierlegende Wollmilchsau‘ bezeichnet werden. Ohne einen guten Entwurf ist das Pattern kein alleiniger Garant für Qualität. In manchen Fällen sind spezielle Anwendungen dieser sogar unerwünscht/ineffizient.

Das Strategy-Pattern hat sowohl Vor- als auch Nachteile, u.a. bessere Übersichtlichkeit des Codes dank dem Aufbau, aber auch eine erhöhte Zahl an Objekten im Arbeitsspeicher.

Detaillierte Arbeitsaufteilung (Aufwandsabschätzung, Endzeitaufteilung)

Aufgabentrennung

Dies ist eine Einzelarbeit, dieser Punkt erübrigt sich.

Aufwandsabschätzung

Diese Aufgabe sollte (inkl. Testen und Protokollieren) in 2 *Stunden* fertiggestellt werden.

Endzeitaufteilung

Es gab keine größeren, zeitintensiven Probleme bei der Umsetzung, jedoch habe ich etwas mehr Zeit investiert um das Programm möglichst vollständig zu machen.

Design	Implementierung	Dokumentieren	Protokoll	Testen/Debuggen	Endergebnis
0,25	1	0,5	0,25	0,5	2,5

Fazit

Es kam nur zu einer kleinen, halbstündigen Abweichung gegenüber der veranschlagten Zeit von 2 Stunden, also gesamt investiert wurden 2,5 *Stunden*.

Detaillierte Arbeitsaufteilung (Aufwandsabschätzung, Endzeitaufteilung)

Resultate

Wie vielleicht schon erwähnt, hat das Pattern Vor- und Nachteile. In diesem Fall aber geht der Plan vollkommen auf: Die Anwendung ist nun modular sowie flexibel erweiterbar, ohne Code umschreiben zu müssen in anderen Klassen.

Niederlagen

Bei der Umsetzung kam es nur zu kleineren Fehlern, wie etwa der unsachgemäße Aufruf der `List<Double>`, der regelmäßig in `NullPointerExceptions` mündete. Diese waren nach einem kurzen Blick in die Java API geklärt, da diese nur richtig instanziiert werden mussten.

```
private List<Double> values = new Vector<Double>();
```

Des Weiteren gab es anfangs einen einfachen Tippfehler zu korrigieren, der die Konsolenausgabe auf eine, nämlich die letzte Zahl beschränkt hat.

```
public String toString(List<Double> values) {
    String out = "";
    Iterator<Double> in = values.iterator();
    while (in.hasNext()) {
        out = "" + in.next() + "\n";
    }
    return out;
}
```










statt:

```
public String toString(List<Double> values) {
    String out = "";
    Iterator<Double> in = values.iterator();
    while (in.hasNext()) {
        out = out + in.next() + "\n";
    }
    return out;
}
```

Es gab ansonsten keinen weiteren größeren Zeitverlust, ebenso bei den oben genannten, wo der Lösungsansatz schon fast von selbst kam.

Testbericht

Coverage

▲ S03		100,0 %	404	0	404
▲ src		100,0 %	404	0	404
▲ weinberger.s03_calculator		100,0 %	404	0	404
▶ Addition.java		100,0 %	26	0	26
▶ Calculator.java		100,0 %	103	0	103
▶ Division.java		100,0 %	26	0	26
▶ Multiplication.java		100,0 %	26	0	26
▶ Substraction.java		100,0 %	26	0	26
▶ TestS03.java		100,0 %	197	0	197

Es wurden 100% der Befehle in diesem Projekt abgedeckt.

Beschreibung

Mithilfe von JUnit 4 konnte in 9 (10 inkl. @Before) Testfällen jede Funktion abgedeckt werden.

Quellenangaben:

<http://docs.oracle.com/javase/8/docs/api/>

„Entwurfsmuster von Kopf bis Fuß“ – Mitschrift sowie PDF