Merkblatt Design Pattern- Strategy Pattern

Zweck

Das Strategy Pattern gehört zur Familie der Behavioral Patterns auch bekannt unter Verhaltensmuster.
Es beschreibt eine Lösung, wie verschiedene Implementierungen eines Algorithmus austauschbar gehalten werden können. Dadurch kann ein Programm mit verschiedenen Lösungsvarianten, genannt **Strategien**, ausgestattet werden, welche zur Laufzeit ausgetauscht werden können. Eine Strategie beinhaltet dabei eine konkrete Implementierung eines Algorithmus.
Die Lösung wird erreicht indem für den Algorithmus, welcher austauschbar sein soll, eine Schnittstelle definiert, welche die Lösung des Problems delegiert. Für alle unterstützte, konkrete Lösungen implementiert man diese Schnittstelle. In einer aufrufenden Klasse kann nun das Interface als Typendefinition benutzt werden und eine konkrete Implementierung kann nach Belieben zugewiesen werden. Im folgenden UML-Diagramm wird dies noch grafisch veranschaulicht:

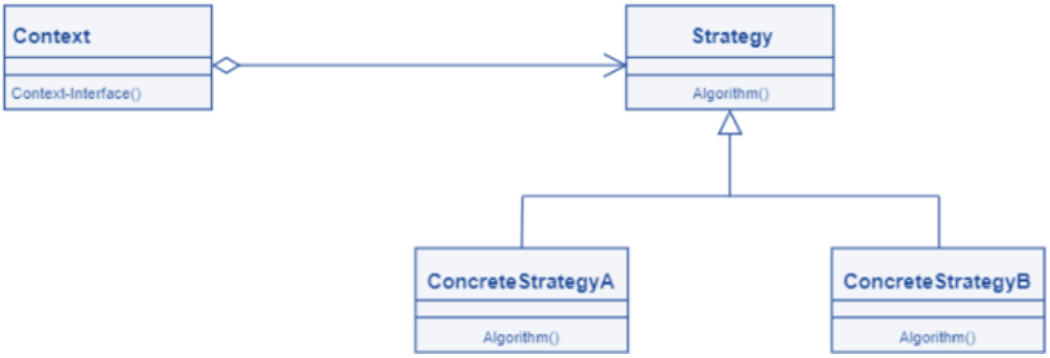

Abbildung 1: Veranschaulichung des Strategy Design-Patterns in UML

Problem

Sie möchten wissen, wie ein Objekt verschiedene Algorithmen verwenden kann, damit der zu verwendende Algorithmus dynamisch ausgesucht und geändert werden kann. Ein Algorithmus unabhängig von den Klassen, die ihn verwenden, implementiert und geändert werden.

Lösung

Strategy Pattern beschreibt, *wie* solche Probleme gelöst werden können:

- Definieren Sie eine gemeinsame Schnittstelle (Strategy), über die zur Laufzeit auf die Algorithmen zugegriffen und diese ausgetauscht werden können.

- Definieren Sie eine Familie von Algorithmen, kapseln Sie jede einzelne - definieren Sie Klassen (StrategyA, …), die verschiedene Algorithmen kapseln und austauschbar machen

Beispiel in Java

1. Erstellen Sie eine Schnittstelle für den austauschbaren Algorithmus. Hier wird beispielhaft eine numerische Operation mit zwei Ganzzahlen abgebildet.

```
public interface CalculatorOperation {
	public int calculate(int num1, int num2);
}
```

2. Erstellen Sie konkrete Klassen, die dieselbe Schnittstelle implementieren. Hier sind Beispiele für die mathematischen Grundoperationen Addition, Subtraktion, Muliplikation und Division

```
public class OperationAdd implements CalculatorOperation  {
	@Override
	public int calculate(int num1, int num2) {
		return num1 + num2;
	}
}
```

```
public class OperationSubstract implements CalculatorOperation  {
	@Override
	public int calculate(int num1, int num2) {
		return num1 - num2;
	}
}
```

```
public class OperationMultiply implements CalculatorOperation  {
	@Override
	public int calculate(int num1, int num2) {
		return num1 * num2;
	}
}
```

```
public class OperationDivide implements CalculatorOperation   {
	@Override
	public int calculate(int num1, int num2) {
		return num1 / num2;
	}
}
```

3. Eine Klasse erstellen, welche die soeben erstellte Schnittstelle nutzt. Hier wird eine Rechner-Klasse erstellt.

```
public class Calculator {

	private CalculatorOperation strategy;

	   public Calculator(CalculatorOperation strategy){
	      this.strategy = strategy;
	   }

	   public int calculate(int num1, int num2){
	      return strategy.calculate(num1, num2);
	   }
}
```

4. Verwenden Sie die erstellte Klasse mit einer beliebigen Strategie. Hier werden alle Grundoperationen durchlaufen.

```java
public class Main {

	public static void main(String[] args) {
		Calculator calculator = new Calculator(new OperationAdd());
		System.out.println("10 + 5 = " + calculator.calculate(10, 5));

		calculator = new Calculator(new OperationSubstract());
		System.out.println("10 - 5 = " + calculator.calculate(10, 5));

		calculator = new Calculator(new OperationMultiply());
		System.out.println("10 * 5 = " + calculator.calculate(10, 5));

		calculator = new Calculator(new OperationDivide());
		System.out.println("10 / 5 = " + calculator.calculate(10, 5));
	}
}
```