Strategy Pattern

Ziel:

Auswechselbarkeit von Algorithmen, Vereinheitlichen von Algorithmen. Beispiele:

- Algorithmus für Zeilenumbruch
- Algorithmus für Suche in Fahrplan (schnellste, billigste, wenig Umsteigen, etc…)
- Spelling Checker (D / E / F…)
- Spielstärke eines Schachalgorithmus
- Algorithmus zur Angabe von Feiertagen in einem Kalender (länderspezifisch)

Motivation:

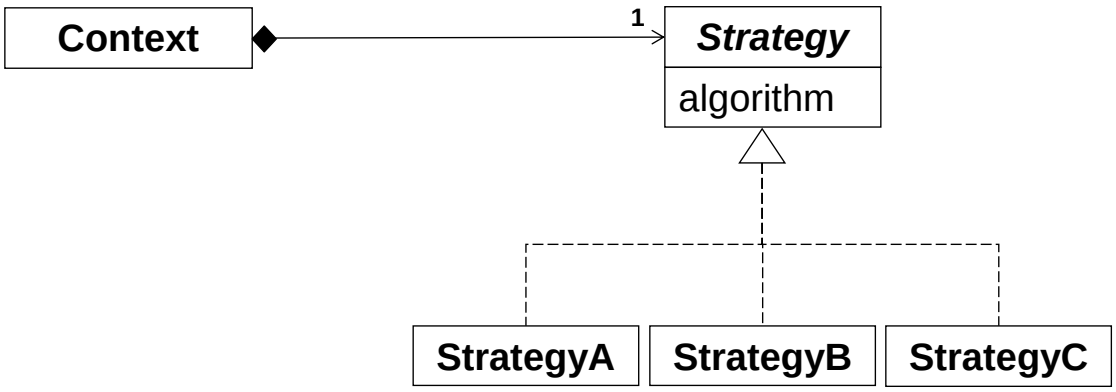
Häufig kommt es vor, dass das Verhalten eines Objektes je nach Kontext angepasst werden muss. Erste einfache Lösung:

```
if (option == 'A') {
   /* Code for option A */
} else if (option == 'B') {
   /* Code for option B */
}
```

Nachteile:

- Bei Änderungen müssen viele Stellen gesucht und geändert werden!
- Code für die einzelnen Optionen in einem Codeblock / einer Klasse;
  Gefahr, dass eine Änderung einer Option Nebeneffekte auf andere Optionen hat.

Struktur:



- Das Kontext-Objekt verwendet eine konkrete Strategy-Instanz (die Strategie wird dabei entweder beim Erzeugen des Kontexts gesetzt oder der Kontext wählt eine geeignete Strategie je nach Situation – in diesem Fall hat er Referenzen auf mehrere Strategie-Objekte).
- Schnittstelle von Strategy muss mächtig genug sein, so dass alle erdenklichen Algorithmen abgedeckt sind.
- Kontext kann Schnittstelle definieren, welche der Strategie Zugriff auf den Kontext erlaubt.

Beispiel:

```
public class Date {
	int day, month, year;
	private final PrintDate p; // algorithm which defines how to format a date

	public Date(PrintDate p) { this.p = p; }

	public void print() {
		p.print(this);
	}

	public static void main(String[] args) throws Exception {
		PrintDate printer = (PrintDate) Class.forName(args[0])
					.getDeclared uctor().newInstance();
		Date d = new Date(printer);

		LocalDate today = LocalDate.now();
		d.day = today.getDayOfMonth();
		d.month = today.getMonthValue();
		d.year = today.getYear();

		d.print();
	}
}

public interface PrintDate {
	void print(Date d);
}

public class StdPrintDate implements PrintDate {
	@Override
	public void print(Date d) {
		System.out.println("Date: " + d.day + "." + d.month + "." + d.year);
	}
}

public class USPrintDate implements PrintDate {
	@Override
	public void print(Date d) {
		System.out.println("Date: " + d.month + "/" + d.day + "/" + d.year);
	}
}
```

Bemerkungen:

- Schnittstelle der Strategie kann auch gerade den Kontext enthalten.
- Konkrete Implementierungen können von verschiedenen Kontexten genutzt werden, aber dann dürfen sie keinen kontextspezifischen Zustand speichern (d.h. sie müssen stateless sein).
- Alternative zu Spezialisierung der Kontextklasse!
- Unter Umständen kann die Schnittstelle sehr umfangreich werden um alle Strategien abzudecken. Für eine einzelne Strategie bedeutet dies dann oft, dass ein grosser Teil der Parameter ignoriert werden muss.
- Um umfangreichen Schnittstellen entgegenzuwirken wird oft eine sehr generische Schnittstelle verwendet, d.h. Verwendung von Typ Object oder String. Folge: Typsicherheit wird aufgeweicht.

Verwendung:

- Als Designer / Architekt planen Sie sich von Anfang an den Einsatz eines Strategy-Pattern. Dann haben Sie die Schwierigkeit eine geeignete Schnittstelle zu definieren, welche möglichst einfach und dennoch mächtig genug ist, um alle möglichen Strategien zu bedienen.
- Refactoring: Als Programmierer fallen Ihnen die if-else if-Leitern im Code negativ auf. Sie beschliessen sie durch ein State- oder Strategy Pattern zu ersetzen. Auch hier kann die Schnittstellendefinition eine Herausforderung sein, besonders dann, wenn noch nicht alle Strategien oder Zustände bekannt sind.
- Implementation einer Strategy: Am häufigsten kommt es vor, dass das Strategy-Pattern in einer Software-Library schon vorgegeben ist. Sie müssen dann nur noch eine bestimmte Strategie implementieren (Beispiel: AWT/Swing-LayoutManager).
- Anwenden falls:
    - Kombination: Viele Klassen unterscheiden sich nur in ihrem Verhalten
      → gemeinsames Verhalten als Kontext „herausfaktorisieren“.
    - Separation: Eine Klasse enthält verschiedene Verhalten, welche durch mehrere bedingte Anweisungen getrennt werden (if / switch)
      → zusammengehörige if-Zweige in ihre eigenen konkreten Strategieklassen verschieben.
    - Erweiterbarkeit: Falls verschiedene Varianten eines Algorithmus benötigt werden.

State versus Strategy:

State- oder Strategy-Pattern sind sehr ähnlich und es ist oft schwierig festzulegen, ob es sich um eine State- oder Stragety-Implementierung handelt. Typischerweise gilt – wobei Abweichungen davon auch typisch sind:

State	Strategy
• Zustandsänderung zur Laufzeit (Zustandsmaschine)	• Strategie wird bei der Erzeugung des Kontextes gesetzt und danach typischerweise nicht mehr geändert (AWT/Swing LayoutManager)
• Definiert zustandsspezifisches Verhalten	• Algorithmus hat häufig eine „compute“-Methode
• Der Zustand wird von aussen gesetzt (setState) oder durch den Zustand selbst (setNextState)	• Die Strategie wird von aussen gesetzt (setStrategy) oder durch den Kontext (abhängig von Parametern)
• Mehrere public-Methoden	• Nur eine public-Methode (und zusätzliche private-Methoden)
• Kein Zustand in einer Stateklasse, aber die States können auf Zustand im Kontext zugreifen.	• Strategy kann Algorithmus-spezifischen Zustand enthalten
• Macht (zustandsabhängig) callbacks im Kontext	• Erledigt die Arbeit autonom (Algorithmus)