STRATEGIE

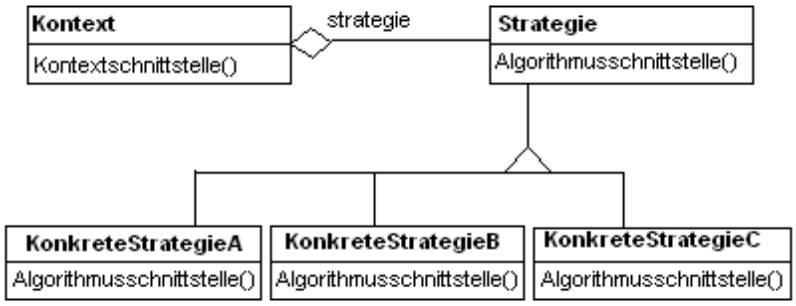
Klassifizierung: objektbasiertes Verhaltensmuster

1. Zweck:
- Familie von Algorithmen, jeden einzeln kapseln und austauschbar machen -> kann unabhängig vom nutzenden Klient ausgetauscht werden
- gekapselter Algorithmus = Strategie

2. Motivation:
- Klienten einfacher zu warten, wenn verwendete Algorithmus separat gespeichert
- durch Kapselung möglich nur wirklich benötigte Algorithmen einzubinden
- wenn Algorithmus direkt in Code -> schwieriger, Alg. zu ändern oder hinzuzuf.

3. Anwendbarkeit:
- wenn viele verwandte Klassen nur in Verhalten unterschiedlich
  -> mit Strategieobjekten, Klasse mit einer von mehreren Verhaltensm. konf.
- wenn Algorithmen verschiedene Vor-/Nachteile – je nach Situation
- wenn Algorithmus Daten verwendet, die Klienten nicht bekannt sein soll.
- wenn Klasse verschiedene Verhaltensweisen mit vielen mehrfachen Bedingungsanweisungen definiert -> Zweige von Bedingungsanweisungen in eigene Strategieklasse verlagern

4. Struktur:



5. Teilnehmer:
- Strategie: deklariert Schnittstelle -> wird von allen unterstützten Algs angeboten Kontextmenü nutzt Schnittstelle um Alg. einer Konkr.Stra. aufzurufen
- KonkreteStrategie: implementiert Algorithmus mit Strategieschnittstelle
- Kontext: - wird mit KonkreteStrategie-Objekt konfiguriert
  - verwaltet eine Referenz auf ein Strategieobjekt
  - kann Schnittstelle definieren, die Strategieobjekten Zugriff auf seine Daten ermöglicht

6. Interaktion:

- Kontextobjekt stellt Strategie benötigte Daten zur Verfügung, wenn Alg. ausgeführt oder Kontext übergibt sich selbst der Strategieoperation als Argument
- Kontextobjekt leitet Klientenanfragen an Strategie weiter, Klient erstellt konkreteStrategie-Objekt (eins aus Familie) -> Übergabe an Kontext

7. Konsequenzen (Vor-/Nachteile)

- Familien von verwandten Alg.:
  Hierarchien von Strategieklassen definieren jeweils Familien von Algs. und Verhalten -> können von Kontextobjekten wiederverwendet werden

- Eine Alternative zur Unterklassenbildung:
  -> anstatt viele verwandte Klassen mit Alg. fest im Kontext -> Algs. kapseln
  -> Strategieklassen ermöglichen Variation des Algorithmus unabhängig vom Kontext
- Strategien entfernen Bedingungsanweisungen:
  -> anstelle von Bed.anw. zur Auswahl des Vehaltens – Kapselung in Strategieklassen
- Auswahlmöglichkeit für Implementierungen:
  -> Klient kann passende Strategie wählen (z.B. bezügl. Zeit-/Speicherplatzverh.
- Klienten müssen Vor-/Nachteile der Strategien kennen um sie sinnvoll zu verwenden
- Kommunikationsaufwand zwischen Strategie und Kontext:
  -> durch gemeinsame Schnittstelle werden z.T. überflüssige Daten an Strat. übergeben
- Erhöhte Anzahl von Objekten durch Verwendung von Strategiemustern, deshalb:
  ->mögl. „zustandslose" Strat. verwenden, so das mehrere Kontextobj. eine Strat. nutzen

8. Implementierung

- Definition der Strategie- und Kontextschnittstelle
  Kontext übergibt benötigte Daten als Parameter oder Strategie erfragt Daten explizit und direkt vom Kontext (über Referenz)
- Strategien als Template-Parameter in C++, wenn:
  -> Strategie zur Übersetzungszeit auswählbar
  -> Strategie während Laufzeit beibehalten wird
- Strategieobjekte optional machen
  -> Kontext benötigt nicht zwingend Strategieobjekt, wenn vordefiniertes Standardverhalten ausreichend

9. Beispiel

Interface *Stepper* (Strategie):

```
package uebKW12;

public interface Stepper {
        public abstract String form();

        public abstract void step();
}
```

Klasse *Texter* (KonkreteStrategieA):

```
package uebKW12;

public class Texter implements Stepper {
                String foobar = "";
                public String form() {
                return foobar;
        }

        public void step() {
                foobar = foobar + "x";
        }
}
```

Klasse *Counter* (KonkreteStrategieB):

```
package uebKW12;

public class Counter implements Stepper {
        int state = 0;

        public String form () {
            return Integer.toString(state);

        }

        public int getState() {
                return state;
        }

        public void setState(int state) {
                this.state = state;
        }

        public void step () {
                setState(1 + getState());
        }
}
```

Klasse *Display* (Kontext):

```
package uebKW12;
import java...

public class Display extends Applet {
     Button b = new Button ("foobar-Schalter");
     Label  l = new Label  ("foobar-Label");

     public void init() {
        final Stepper c = new Counter ();
        add (b);
        add (l);

        b.addActionListener(new ActionListener() {
             public void actionPerformed
                       (ActionEvent e) {
                 c.step ();
                 l.setText(c.form());
             }
        });
    }
}
```