

# Seminar Software Design Patterns

## Sommersemester 09

### Paket Servicevariation



Template Method

State

Strategy

Vortragender: René Speck

# Seminar Software Design Patterns


## Sommersemester 09

### Inhalt

- ❖ Verhaltensmuster
- ❖ Template Method, State, Strategy
- ❖ Allgemein
- ❖ Problem, Lösung, Kontext
- ❖ Struktur, Beispiel
- ❖ Vor- und Nachteile

# Verhaltensmuster

## Behavioral Patterns

- ❖ beschreiben die Interaktion zwischen Objekten und komplexen Kontrollflüssen. Sie charakterisieren die Art und Weise, in der Klassen und Objekte zusammenarbeiten und Zuständigkeiten aufteilen.
- ❖ **Klassensmuster** ( Behavioral Class Pattern )  
teilen die Kontrolle auf verschiedene Klassen auf. Die Struktur liegt zur Übersetzungszeit fest vor.
  - ❖ Template Method
- ❖ **Objektmuster** ( Behavioral Object Pattern )  
nutzen Assoziation und Aggregation anstelle von Vererbung. Objektbeziehungen können zur Laufzeit geändert werden.
  - ❖ State
  - ❖ Strategy

Gleiche Klassendiagramme,  
aber unterschiedliche Absichten.

# Verhaltensmuster

## Behavioral Patterns

### ❖ Template Method (Schablonenmethode)

#### ❖ Allgemein :

Schablone als Strukturvorlage für die Reihenfolge der abzuarbeitenden Schritte. Einzelne spezifische Schritte sind noch unbekannt.



Hollywood-Prinzip : „Don't call us, we'll call you“<sup>[Swe85]</sup>

#### ❖ Beispiel:

Baupläne für Häuser können variieren, verschiedene Materialien, Größen, etc. Aber die Grundstruktur des Bauplanes bleibt bestehen: zuerst das Fundament, dann die Etagen, anschließend das Dach.

# Template Method

## Kontext :

- ❖ Viele ähnliche Algorithmen mit invariantem Code.
- ❖ Klassenbibliotheken

### ❖ Problem

1. Redundanter Code
2. Grundstruktur bekannt, genaues Verhalten unbekannt bzw. Algorithmus soll variabel bleiben.

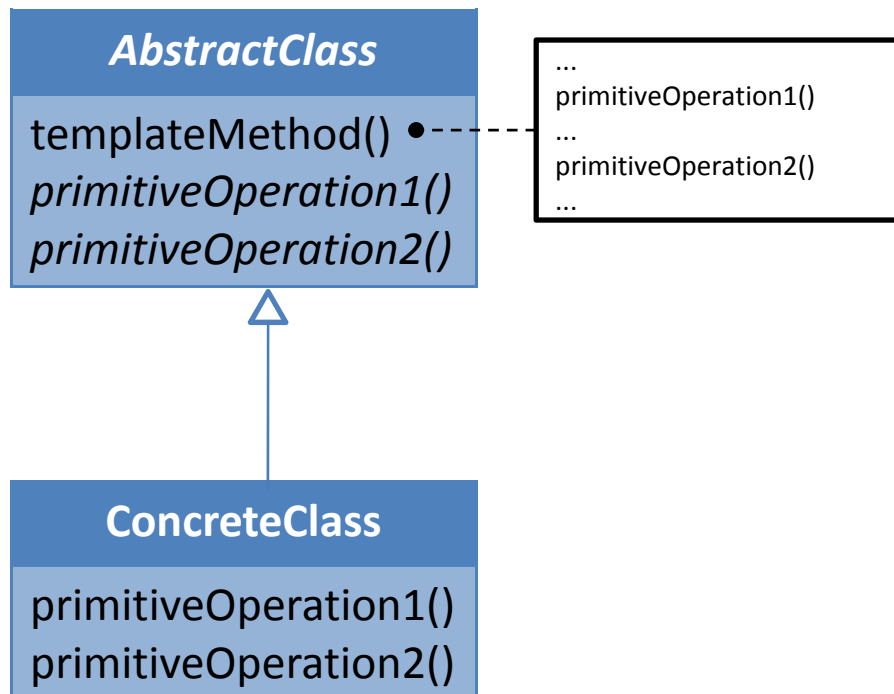
### ❖ Lösung

1. „Refaktorisierung zur Verallgemeinerung“ [Oj93]
2. Abstrakter Grundalgorithmus und Subklassen definieren konkretes Verhalten.

# Template Method

## Struktur:

- ❖ Definiert das Gerüst /die Struktur eines Algorithmus und überlässt einige Schritte den Unterklassen.
- ❖ Unterklassen können Methoden des Algorithmus überschreiben und somit sein Verhalten bestimmen, die Struktur bleibt unverändert.
- ❖ Die Templatemethode definiert den Algorithmus unter Verwendung von abstrakten Methoden die von Unterklassen überschrieben werden und sie legt die Reihenfolge der abzuarbeitenden Schritte fest.

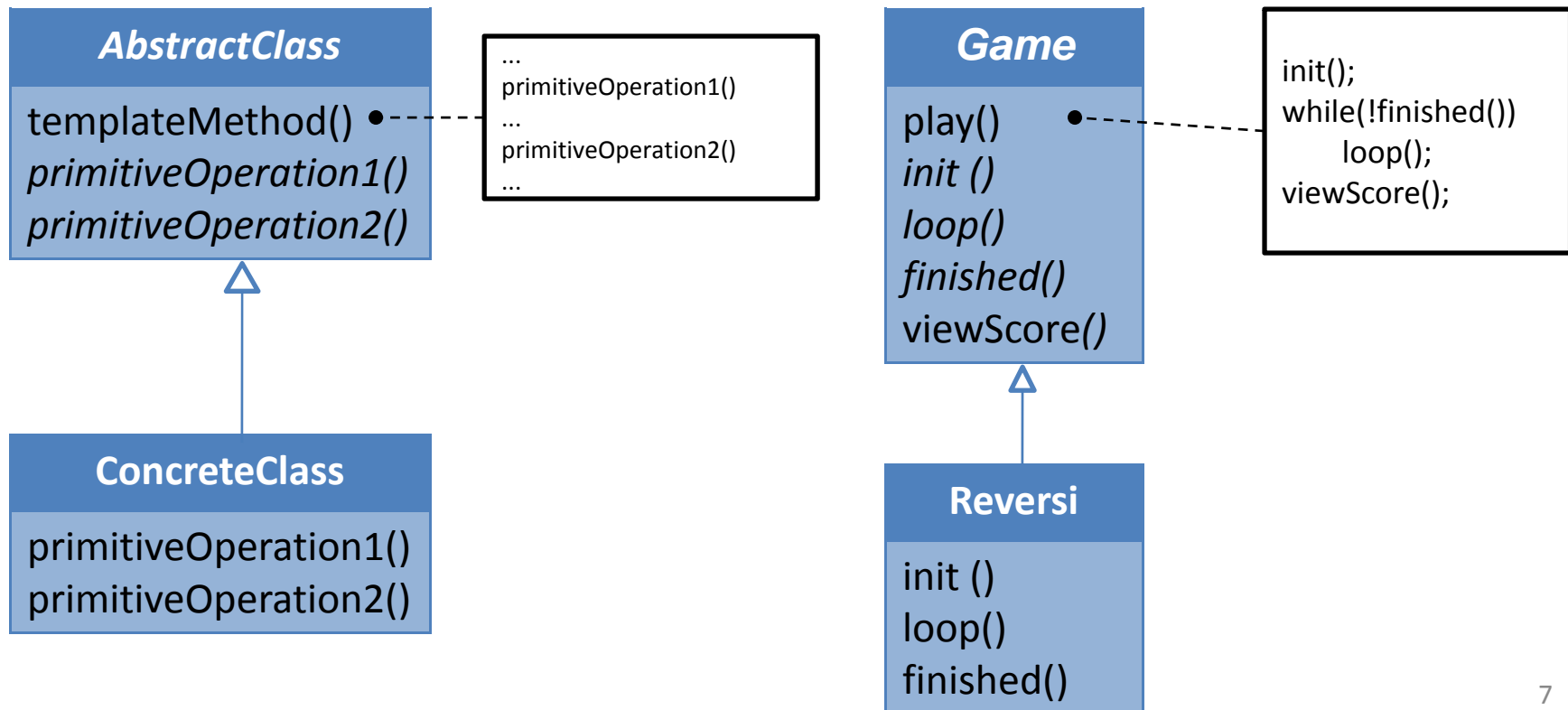


# Template Method

Struktur:

Beispiel:

- ❖ Definiert das Gerüst /die Struktur eines Algorithmus und überlässt einige Schritte den Unterklassen.
- ❖ Unterklassen können Methoden des Algorithmus überschreiben und somit sein Verhalten bestimmen, die Struktur bleibt unverändert.
- ❖ Die Templatemethode definiert den Algorithmus unter Verwendung von abstrakten Methoden die von Unterklassen überschrieben werden und sie legt die Reihenfolge der abzuarbeitenden Schritte fest.



# Template Method

## Vor- und Nachteile:

- ✓ grundlegende Technik zur Wiederverwendung von Code
- ✓ Entfernt redundanten Code in Unterklassen, indem der invariante Code in der Oberklasse gekapselt implementiert wird (verringert die Größe des Quelltextes)
- ✓ somit Zeitersparnis durch Verwendung bestehender Strukturen und schnellere Änderungen am Code
- ✓ Vereinfacht die Schritte des allgemeinen Algorithmus
- ✓ Unterklassen können den allgemeinen Algorithmus relativ einfach individualisieren
- ✗ fixe Struktur kann zum Nachteil werden
- ✗ verkompliziert das Design, wenn Unterklassen sehr viele Methoden implementieren müssen, um den Algorithmus zu konkretisieren



# Verhaltensmuster

## Behavioral Patterns

### ❖ State (Objects for States, Zustand)

#### ❖ Allgemein:

Die Art und Weise, wie etwas in einem bestimmten Moment ist.

#### ❖ Beispiel:

Der Aggregatzustand von Wasser (fest, flüssig, gasförmig) hängt nicht nur von der Temperatur ab, sondern auch vom Druck.

Das Verhalten von Wasser ist abhängig vom vorhandenen Aggregatzustand .



# State

## Kontext :

- ❖ Ermöglicht einem Objekt sein Verhalten zur Laufzeit zu ändern, wenn es seinen internen Zustand ändert.
- ❖ Nach außen sieht es so aus, als ob das Objekt seine Klasse gewechselt hätte.

### ❖ Problem

1. Unübersichtlich große bedingte Anweisungen die von Objektzuständen abhängen.
2. Objektverhalten ist Zustandsabhängig und ändert sich zur Laufzeit.

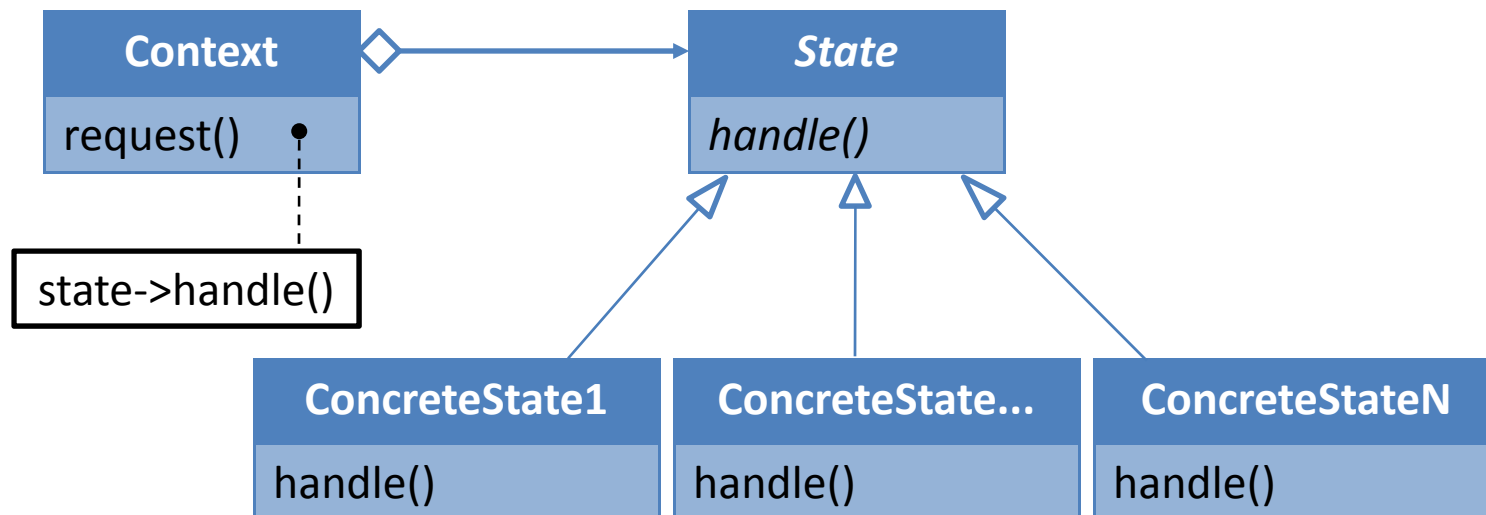
### ❖ Lösung

1. Kapseln des zustandsabhängigen Code in Zustandsklassen.
2. Komposition anstelle von Vererbung.

# State

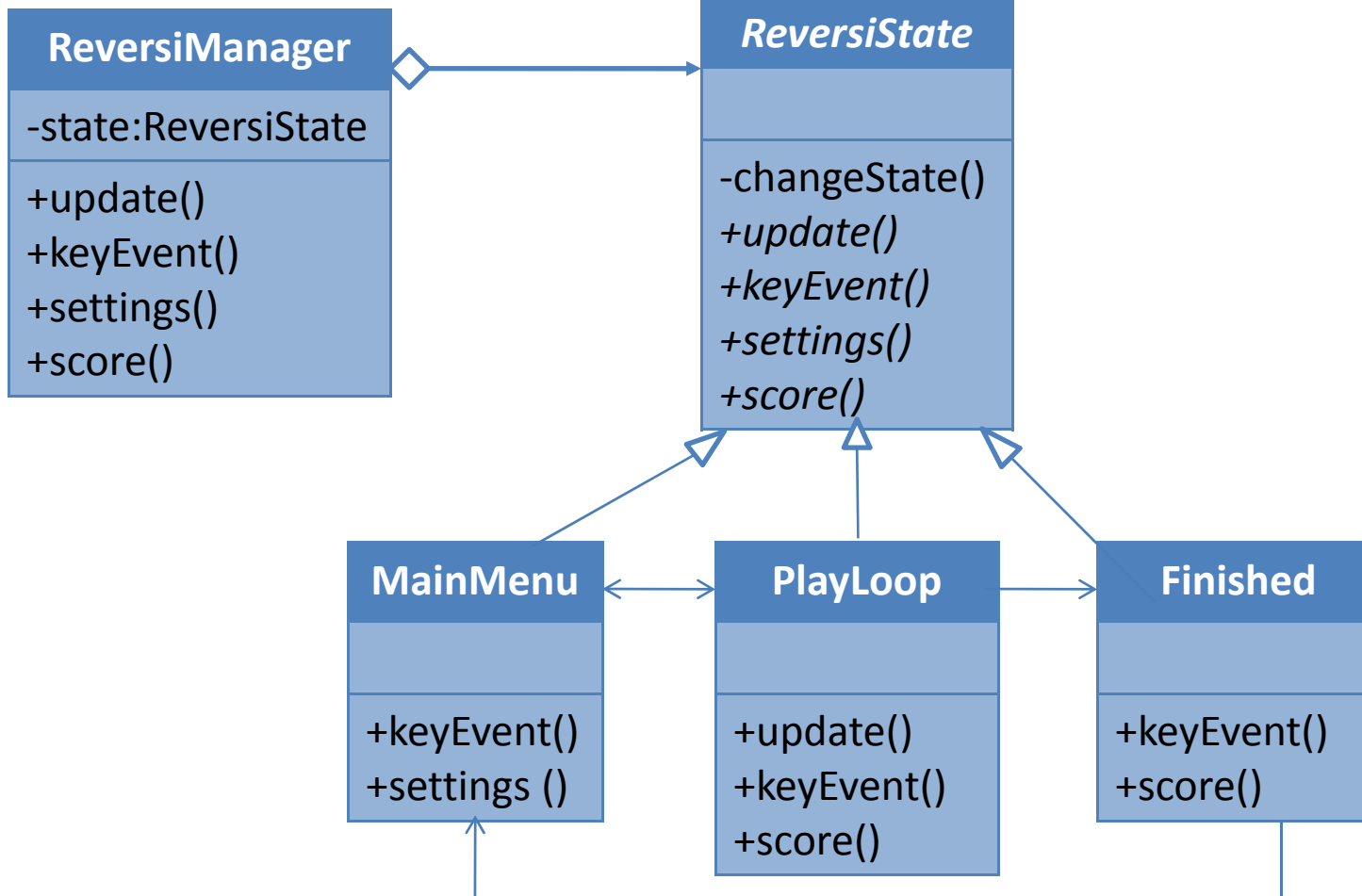
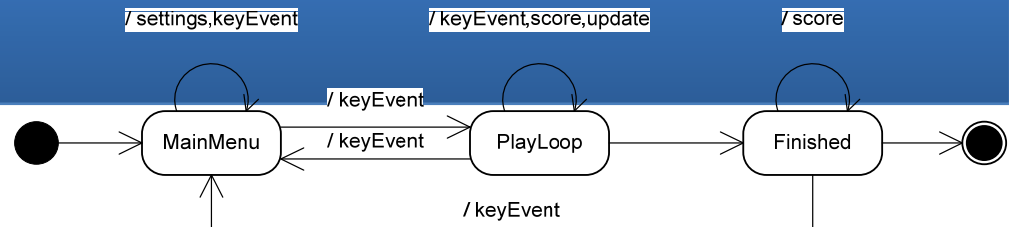
## Struktur:

- ❖ Abstrakte Klasse **State**, repräsentiert die Zustände der Klasse **Context** und ist somit eine Schnittstelle zur Kapselung des mit einem konkreten Zustand (**ConcreteStateA**,...,**ConcreteStateN**) verbundenen Verhaltens.
- ❖ Die konkreten Zustände implementieren zustandsspezifisches Verhalten.
- ❖ Die Klasse **Context** verwaltet ein Instanz eines konkreten Zustands, dieser repräsentiert den aktuellen Zustand.
- ❖ Sowohl **Context** als auch die konkreten Zustandsklassen können Zustands-übergänge hervorrufen.



# State

Beispiel:



# State

## Vor- und Nachteile:

- ✓ Verringert oder entfernt zustandsverändernde bedingte Logik und lagert diese in Klassen aus.
- ✓ Vereinfacht komplexe zustandsverändernde Logik.
- ✓ Zustandsklassen lassen sich leichter erweitern .
- ✓ Stellt zu den Bedingungsanweisungen eine bessere Alternative zur Strukturierung von zustandsspezifischem Code dar. Verbessert also die Sicht auf die zustandsverändernde Logik.
- ✗ Wenn die Logik für Zustandsübergänge bereits einfach nachvollziehbar ist, wird das Design nur unnötig verkompliziert.
- ✗ Der Verwaltungsaufwand für die neu hinzugekommenen Klassen steigt und somit ist das auszuführende Programm langsamer.

# Verhaltensmuster

## Behavioral Patterns

### ❖ Strategy (Policy , Strategie)

### ❖ Allgemein:

Plan zur Durchführung eines kontextabhängigen Vorhabens.

### ❖ Beispiel:

Die Strategie im Mannschaftssport ändert sich während des Spiels, sie muss mit den Spielern abgesprochen und ausgewählt werden.



# Strategy

## Kontext :

- ❖ Definiert und kapselt eine Familie von Algorithmen und ermöglicht eine Variierung zur Laufzeit. Änderungen am Algorithmus sollen den Klienten nicht beeinflussen.
- ❖ Algorithmen können leicht ausgetauscht werden.

## ❖ Problem

1. Algorithmus definiert verschiedene Verhaltensweisen in bedingten Anweisungen.
2. Mehrere Varianten eines Algorithmus sind nötig.

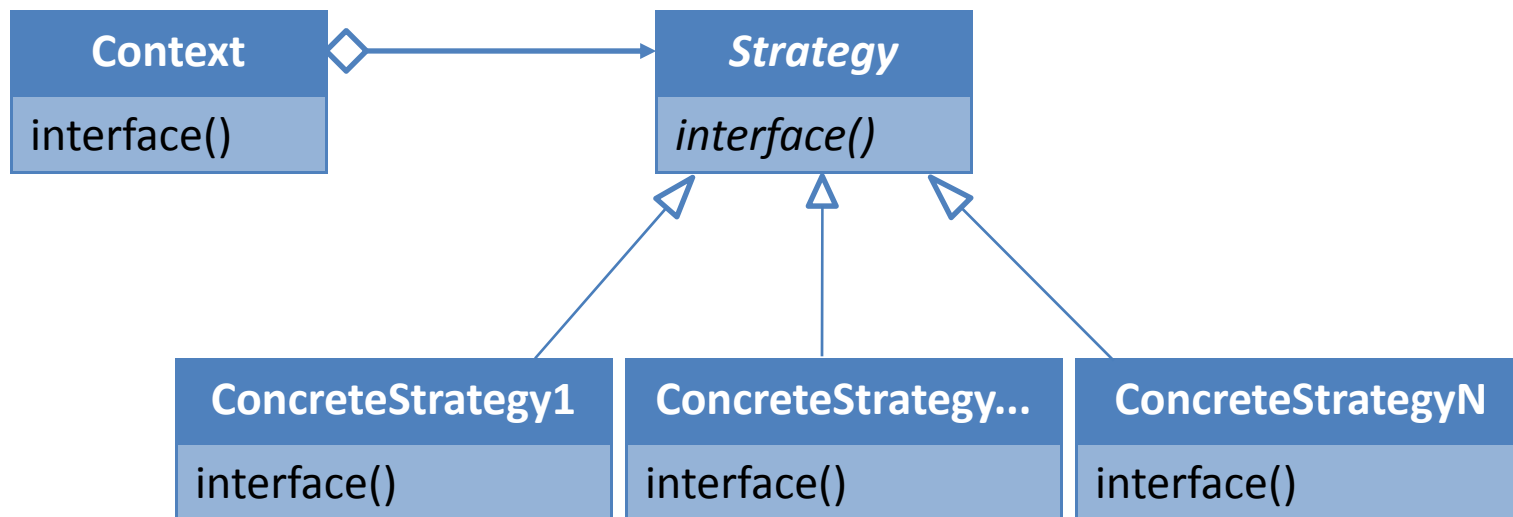
## ❖ Lösung

1. Zusammenhängende bedingten Anweisungen verlagern.
2. Varianten als eigenständige Klassenhierarchie implementieren.

# Strategy

## Struktur:

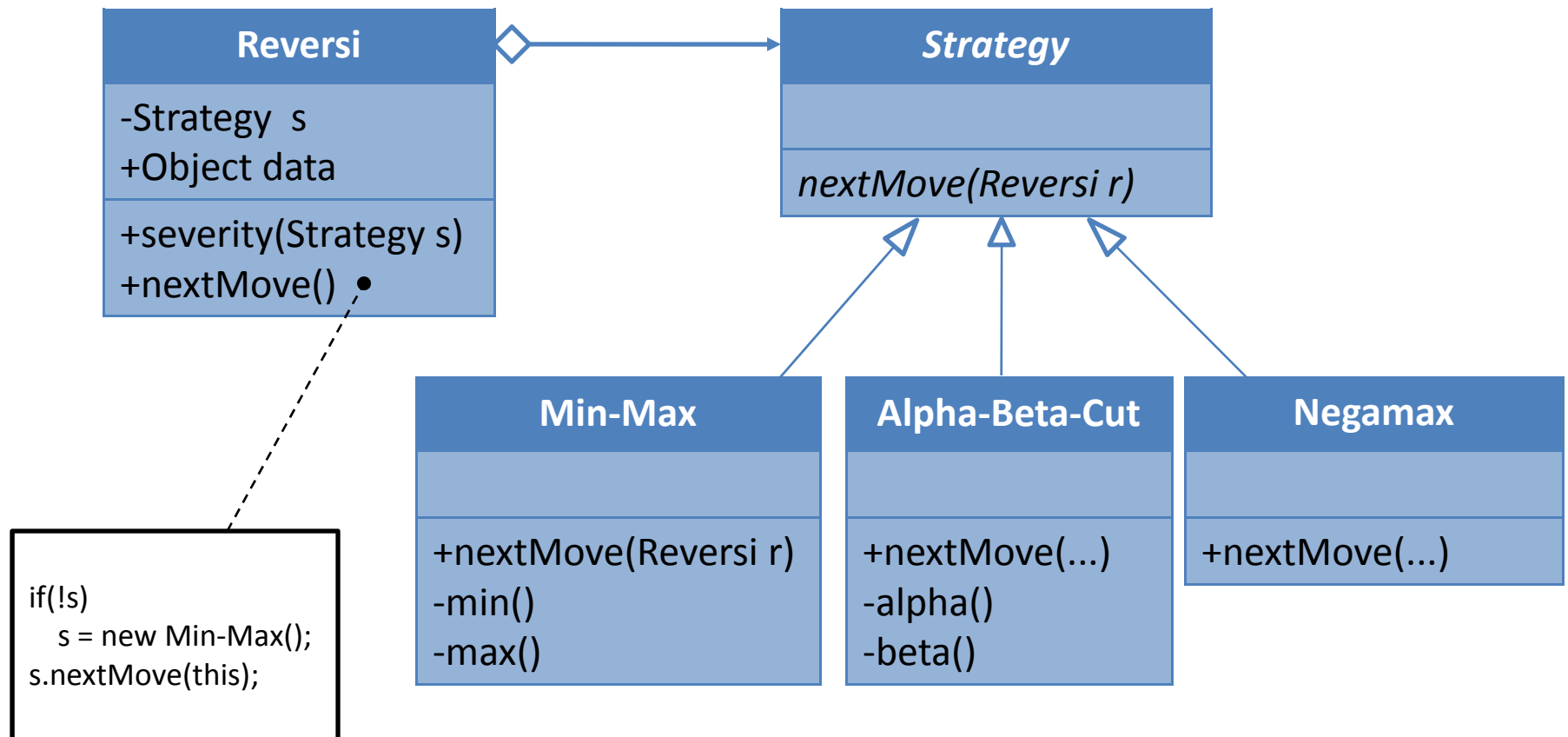
- ❖ Abstrakte Klasse **Strategy** , stellt eine Schnittstelle zur Kapselung einer Familie von Algorithmen (**ConcreteStrategyA**,..., **ConcreteStrategyN**) zur Verfügung.
- ❖ Der konkrete Algorithmus wird extern über eine Methode oder von der Klasse **Context** ausgewählt.
- ❖ Die Klasse **Context** stellt alle nötigen Daten für den konkret ausgewählten Algorithmus bereit.
- ❖ Der Benutzer interagiert ausschließlich mit dem **Context** Objekt.





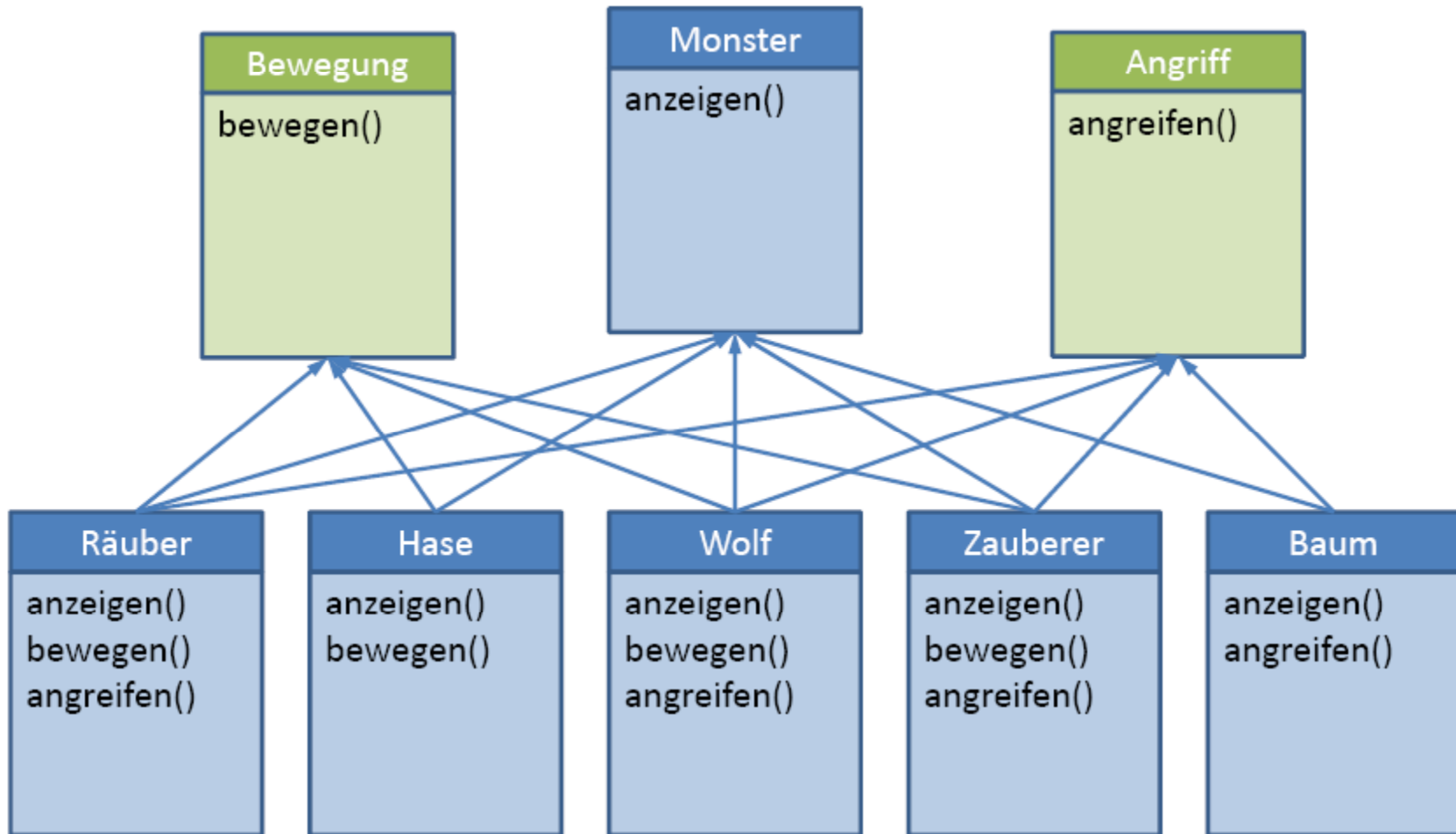
# Strategy

Beispiel:



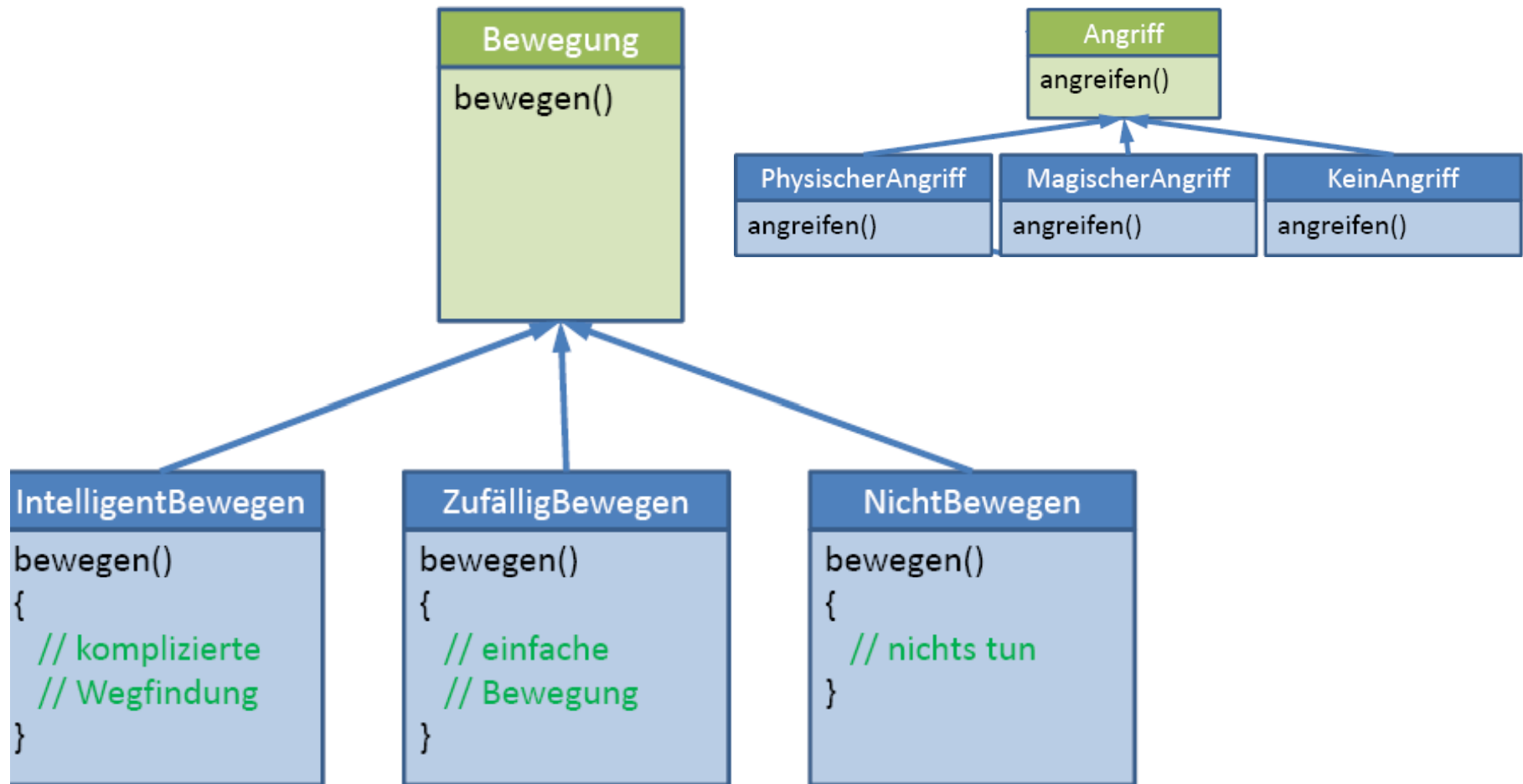
# Strategy

## 2. Beispiel<sup>1</sup>: Monstervererbung (Wiederholung)



# Strategy

## 2. Beispiel<sup>1</sup> : Kapseln



# Strategy

## 2. Beispiel<sup>1</sup>: dynamischer Monsterhase

```
public class Hase : public Monster
{
    public Hase() // Konstruktor
    {
        myBewegung = new ZufaeligBewegen();
        myAngriff = new KeinAngriff();
    }
    public void anzeigen()
    {
        std::cout << "Mein Name ist Hase";
    }
    //Verhalten dynamisch ändern:
    public void setAngriffsVerhalten(Angriff *neuerAngriff)
    {
        myAngriff = neuerAngriff;
    }
}
```

Monster
Bewegung myBewegung; Angriff myAngriff;
anzeigen() doBewegen() doAngriff() { myAngriff.angreifen() }

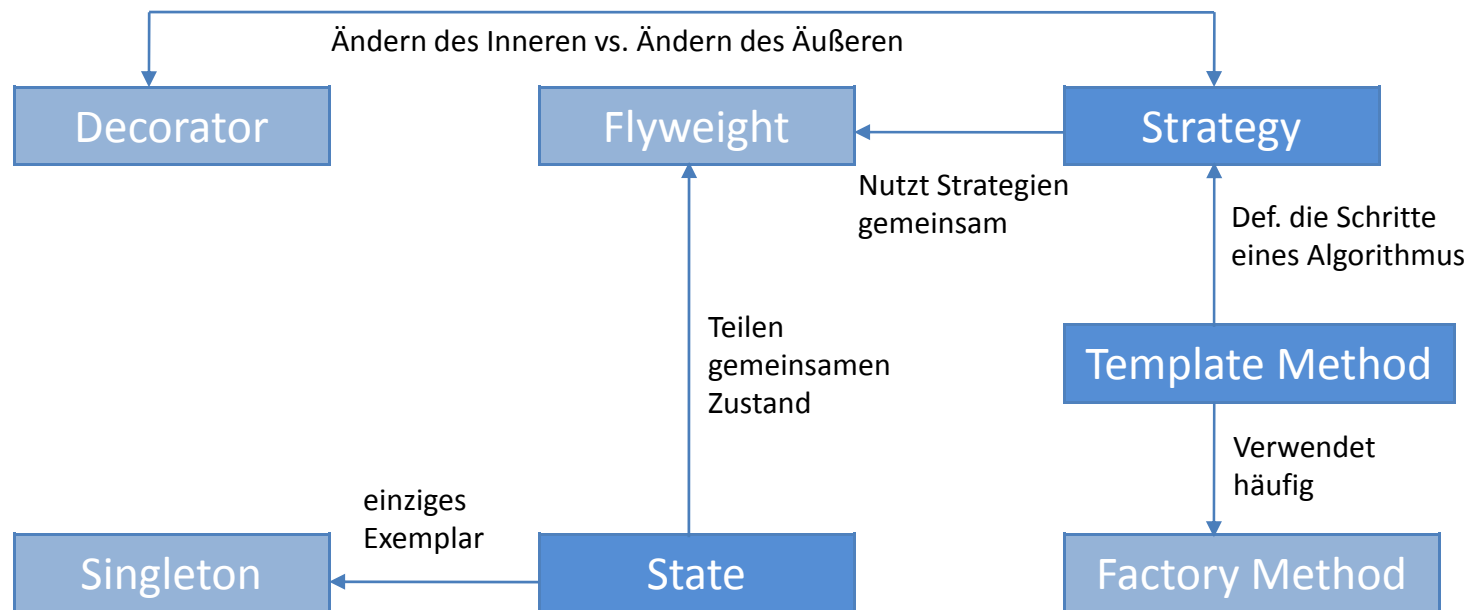
# Strategy

## Vor- und Nachteile:

- ✓ Hohe Wiederverwendung der Familie von verwandten Algorithmen
- ✓ Algorithmen können schneller ausgetauscht, gewartet, erweitert,... werden.
- ✓ Verringert Bedingungsanweisungen
- ✓ Kann einheitliches Interface für den Klienten bereitstellen, der aus verschiedenen Algorithmen desselben Verhaltens wählen kann
- ✗ Klient muss die einzelnen Strategieobjekte und deren Unterschiede kennen  
(Lösung: default Verhalten)
- ✗ Höherer Kommunikationsaufwand, da Kontextparameter von einfachen Strategieobjekte nicht genutzt werden  
(Lösung: Kontext wird Strategieobjekte bekannt gemacht)
- ✗ Erhöhte Anzahl an Objekten in einer Anwendung  
(Lösung: Flyweight)

# Paket Servicevariation

Beziehung zu anderen Pattern:



# Paket Servicevariation

Beziehung zu anderen Pattern:

Template Method	Strategy	State
Def. ein Gerüst eines Alg. behält dabei die Kontrolle über den Alg. da Vererbung verwendet wird.	Def. eine Familie von Alg. und macht sie austauschbar, da Komposition verwendet wird.	Kapselt Verhaltensweisen in Zustandsobjekten. Zustände können unabhängig vom Klienten gewechselt werden.
Benötigt weniger Objekte als Strategy bzw. State	Ist flexibler als Template Method.	Ist flexibler als Template Method.
Klient wählt feste Struktur.	Klient wählt das konkrete Strategy Objekt aus und kann dieses austauschen. Er muss evtl. alle kennen.	Klient legt nur den Startzustand fest. Zustandsübergänge sind unabhängig vom Klient.  Zustände bleiben dem Klienten verborgen.

# Paket Servicevariation

## Quellen:

### ❖ Literaturverzeichnis

- [GoF] E. Gamma, R. Helm, R. Johnson , J. Vlissides: Entwurfsmuster
- [Ker04] Joshua Kerievsky: Refactoring to Patterns
- [Swe85] R. E. Sweet. The Mesa programming environment
- [Oj93] W. F. Opdeyke und R. E. Johnson: Creating abstract superclasses by refactoring
- [Fsb08] E. Freeman, E. Freeman, K. Sierra, B. Bates: Entwurfsmuster von Kopf bis Fuß

### ❖ Abbildungen

<http://www.patterntesting.com/images/pattern-training.jpg>  
[http://www.probau-immobilien.com/images/bauplanung\\_grafik.gif](http://www.probau-immobilien.com/images/bauplanung_grafik.gif)  
[http://www.waterspender.de/uploads/image/Fotolia\\_103764\\_XS.jpg](http://www.waterspender.de/uploads/image/Fotolia_103764_XS.jpg)  
<http://www.immer-wieder-jim.de/lexikon/football.jpg>  
[http://www.typen.ch/hausbau/page22/files/page22\\_5.jpg](http://www.typen.ch/hausbau/page22/files/page22_5.jpg)



# Seminar Software Design Patterns

## Sommersemester 09

