



Universität Bremen

# Entwurfsmuster

Thomas Röfer

Wettbewerb

Motivation

Erzeugende Muster

Strukturelle Muster

Verhaltensmuster

## Mein Rückblick: „RoboCup“





# Euer Rückblick: „Textsuche“

Naive Suche

abrakadabra...

aber  
aber  
aber  
aber  
aber  
aber  
aber

Boyer-Moore

abrakadabra...

ababrakadabra...  
 ababrakadabra...  
 ababrakadabra...  
 ababrakadabra...

Knuth-Morris-Pratt

abrakadabra...

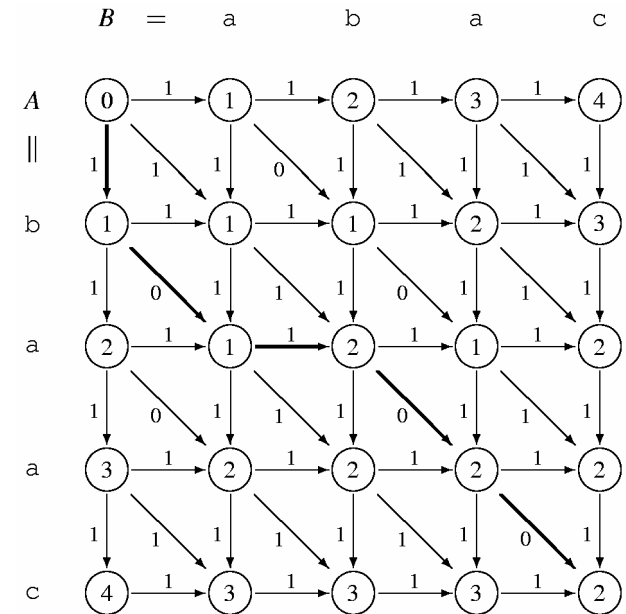
aber  
aber  
aber  
aber  
aber  
aber  
aber

Ähnlichkeitssuche

abrakadabra...

ababrakadabra...  
 ababrakadabra...  
 ababrakadabra...  
 ababrakadabra...  
 ababrakadabra...

Editierdistanz



# Fachgespräche

## ▶ **Umfang**

- ▶ ca. 20-25 Minuten pro 3er-Gruppe

## ▶ **Inhalt**

- ▶ Suchen (Kap. 11 + OW Kap. 3)
- ▶ Sortieren (Kap. 12 + OW Kap. 2)
- ▶ Bäume (Kap. 13 + OW Kap. 5)
- ▶ Hashing (Kap. 14 + OW Kap. 4)
- ▶ Manipulation von Mengen (OW Kap. 6)
- ▶ Geometrische Algorithmen (OW Kap. 7)
- ▶ Graphenalgorithmen (OW Kap. 8)
- ▶ Textsuche (OW Kap. 9)

# Wettbewerb

- ▶ **Aufgabe 1**
  - ▶ Das beste Programm für die Ameisen entwickeln
- ▶ **Aufgabe 2**
  - ▶ Die beste Ablaufumgebung für die Ameisensimulation schreiben
- ▶ **Organisation**
  - ▶ Selbstorganisiert über die PI-2-Newsgroup
- ▶ **Austragung**
  - ▶ Anfang des Wintersemesters



<http://www.cis.upenn.edu/group/proj/plclub/contest/>

# Motivation

## ▶ Idee

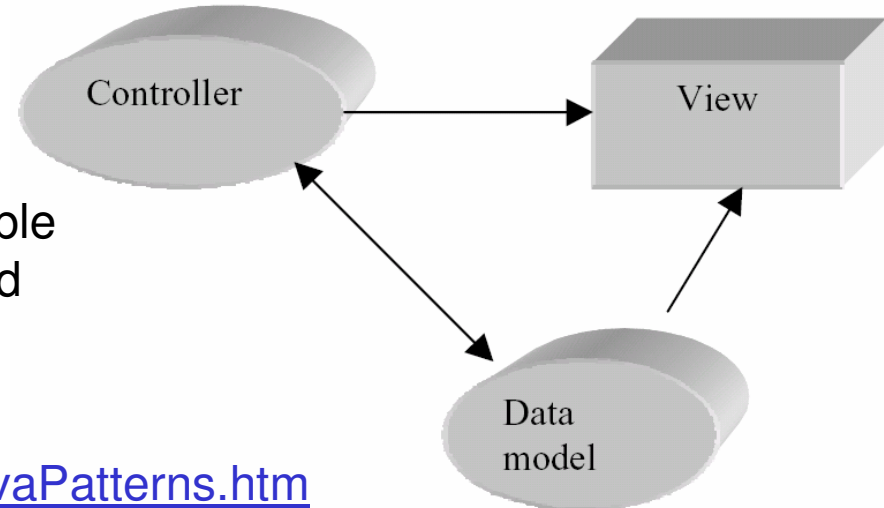
- ▶ Einige generelle Ansätze zur Lösung bestimmter Implementierungsprobleme haben sich als vorteilhaft erwiesen
- ▶ Diese werden in Form von *Entwurfsmustern* (*Design Patterns*) beschrieben, so dass das Rad nicht immer neu erfunden werden muss

## ▶ Historie

- ▶ Model-View-Controller Architektur für Smalltalk (Krasner und Pope, 1988)
- ▶ Design Patterns – Elements of Reusable Software (Gamma, Helm, Johnson und Vlissides, 1995)

## ▶ Design Patterns Java Companion

- ▶ <http://www.patterndepot.com/put/8/JavaPatterns.htm>





# Erzeugende Muster

- ▶ **Factory-Muster**
  - ▶ Erzeugt neue Objekte, wobei deren jeweilige Klasse von den übergebenen Parametern abhängt, aber immer eine Ableitung einer gemeinsamen abstrakten Klasse ist
- ▶ **Abstract-Factory-Muster**
  - ▶ Wie *Factory-Muster*, aber die Fabrik selbst ist auch abstrakt (also austauschbar)
- ▶ **Builder-Muster**
  - ▶ Trennt die Erzeugung von komplexen Objekten von ihrer Repräsentation, so dass verschiedene Repräsentationen erzeugt werden können, abhängig von den Anforderungen durch das Programm
- ▶ **Prototype-Muster**
  - ▶ Beginnt mit einer initialisierten Instanz eines komplexen Objekts und erzeugt weitere, indem es diesen Prototypen kloniert
- ▶ **Singleton-Muster**
  - ▶ Ist eine Klasse, die nur einmal instanziiert werden kann. Sie bietet einen globalen Zugriffspunkt auf diese Instanz

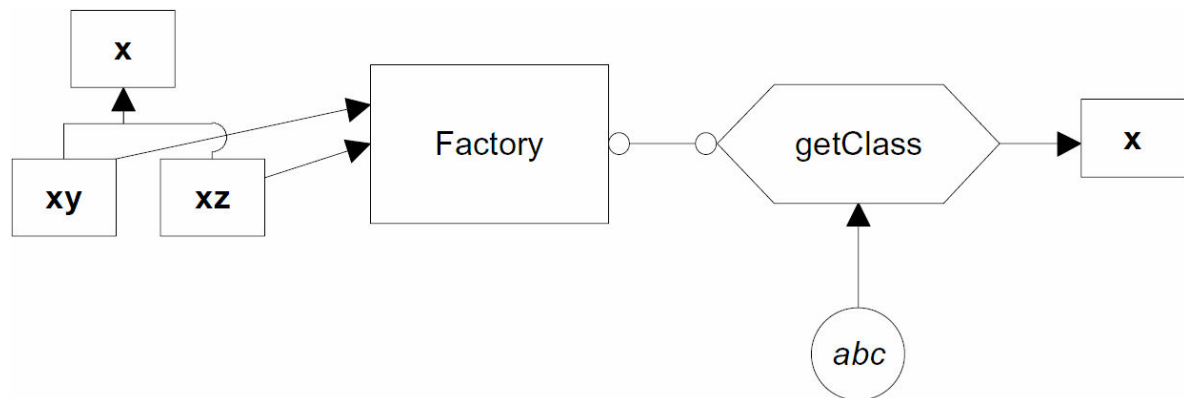
# Erzeugende Muster – Factory

## ▶ Ansatz

- ▶ Erzeugt neue Objekte unterschiedlicher Klassen
- ▶ Alle Klassen haben gemeinsamen, abstrakten Basistyp

## ▶ Anwendungen

- ▶ Wenn eine Klasse nicht vorhersehen kann, ein Objekt welcher Klasse erzeugt werden muss
- ▶ Wenn eine Klasse durch die Wahl ihrer Unterklassen spezifiziert, welche Objekte erzeugt werden sollen
- ▶ Falls die Entscheidung, von welcher Klasse Objekte erzeugt werden sollen, zentralisiert werden soll





# Erzeugende Muster – Factory

```
class Namer
{
    protected String first,
                    last;

    String getFirst() {return first;}
    String getLast() {return last;}
}

class FirstFirst extends Namer
{
    FirstFirst(String s)
    {
        int i = s.lastIndexOf(" ");
        first = i > 0
            ? s.substring(0, i).trim()
            : "";
        last = s.substring(i + 1).trim();
    }
}
```

```
class LastFirst extends Namer
{
    LastFirst(String s)
    {
        int i = (s + ",").indexOf(",");
        last = s.substring(0, i).trim();
        first = (s + " ")
            .substring(i + 1).trim();
    }
}

class NameFactory
{
    Namer getNamer(String s)
    {
        if(s.indexOf(",") > 0)
            return new LastFirst(s);
        else
            return new FirstFirst(s);
    }
}
```



# Erzeugende Muster – Singleton

## ▶ **Ansatz**

- ▶ Manchmal wird eine Klasse benötigt, die man nur einmal instanziiieren kann

## ▶ **Mögliche Umsetzung**

- ▶ Verhinderung der mehrfachen Instanziierung
  - ▶ *Markierung in statischem Attribut*
  - ▶ *Konstruktor erzeugt Exception, falls mehrfach konstruiert*
  - ▶ *Statische Methode liefert neue Instanz oder null, falls mehrfach konstruiert*
- ▶ Alles statisch
  - ▶ *Spätere Umstellung auf mehrere Instanzen schwierig*

## ▶ **Anwendungen**

- ▶ Anwendungsobjekt
- ▶ Hauptfenster
- ▶ zentrale Verwaltungsaufgaben



# Erzeugende Muster – Prototype

## ▶ Ansatz

- ▶ Teilweise wird ein Objekt mehrfach benötigt
- ▶ Anstatt es neu zu konstruieren, kann man eine bereits bestehende Instanz klonen

## ▶ Verfahren

- ▶ Im Allgemeinen muss man dazu eine tiefe Kopie erstellen
- ▶ Achtung: Javas *clone()* erstellt standardmäßig eine flache Kopie

```
class CloneTest
{
    static int cloneTest()
    {
        int[][] a = new int[1][1],
                b = (int[][]) a.clone();
        a[0][0] = 17;
        return b[0][0];
    }
}
```



# Strukturelle Muster

- ▶ **Adapter-Muster**
  - ▶ Kann benutzt werden, um eine Klassensignatur einer anderen nachzubilden
- ▶ **Composite-Muster**
  - ▶ Ist eine Komposition von Objekten, wobei jedes davon wiederum eine weitere Komposition sein kann
- ▶ **Proxy-Muster**
  - ▶ Ein Stellvertreter für ein anderes Objekt, welches z.B. auf einem anderen Rechner läuft
- ▶ **Flyweight-Muster**
  - ▶ Hierbei enthalten die Objekte ihren Zustand nicht selbst, sondern speichern ihn extern
- ▶ **Façade-Muster**
  - ▶ Wird benutzt, um ein ganzes Subsystem durch eine einzige Klasse zu repräsentieren
- ▶ **Bridge-Muster**
  - ▶ Trennt die Schnittstelle eines Objekts von seiner Implementierung
- ▶ **Decorator-Muster**
  - ▶ Erlaubt das dynamische Hinzufügen von Erweiterungen eines Objekts

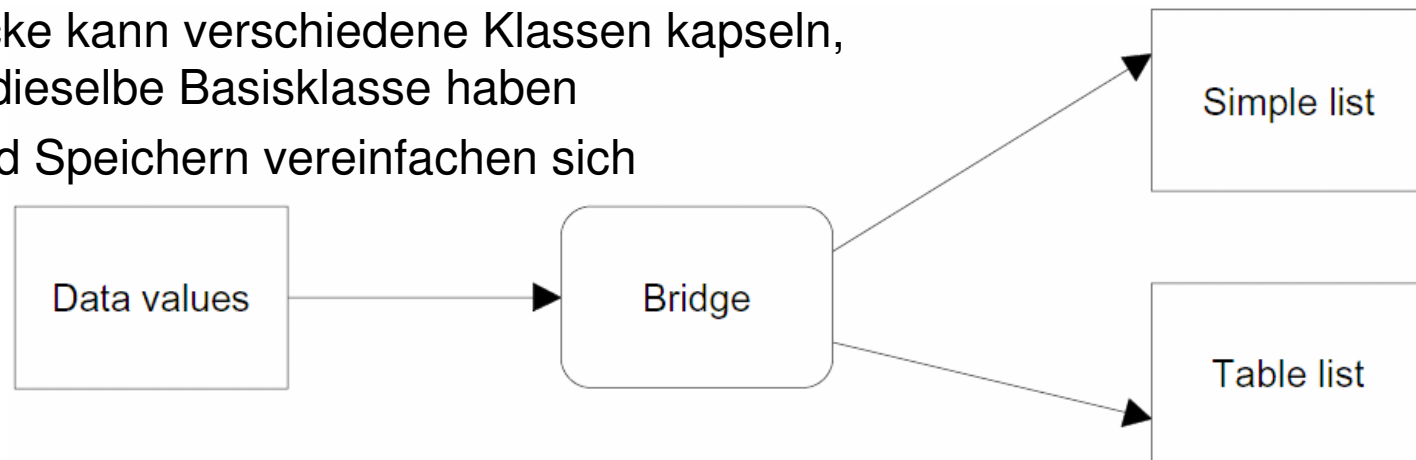
# Strukturelle Muster – Bridge

## ▶ Ansatz

- ▶ Eine *Bridge* stellt eine einheitliche Schnittstelle für verschiedene Instanzen von Klassen zur Verfügung
- ▶ Bei der Konstruktion kann angegeben werden, welche dieser Klassen instanziiert werden soll

## ▶ Vorteile

- ▶ Schnittstelle und Implementierung sind stärker entkoppelt
- ▶ Eine Brücke kann verschiedene Klassen kapseln, die nicht dieselbe Basisklasse haben
- ▶ Laden und Speichern vereinfachen sich





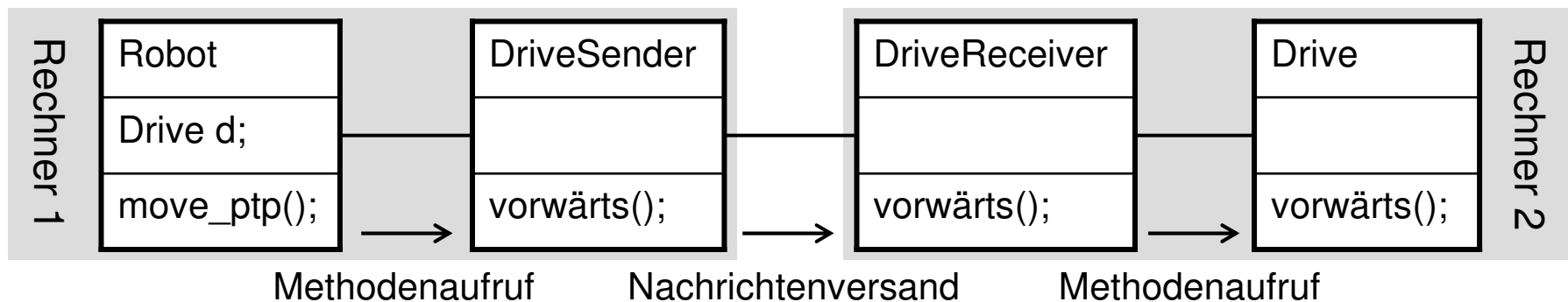
# Strukturelle Muster – Proxy

## ▶ Ansatz

- ▶ Ein *Proxy* steht für ein Objekt, das in einem anderen Zeitrahmen ausgeführt wird, also z.B. in einem anderen *Thread* oder auf einem anderen Rechner
- ▶ Ein Proxy bildet die Schnittstelle zu diesem Objekt (Stellvertreter)

## ▶ Anwendungen

- ▶ Inter-Prozess-Kommunikation
- ▶ Dynamisches (Nach-)Laden von Graphiken





# Verhaltensmuster

- ▶ Das *Observer*-Muster definiert einen Weg, wie eine Anzahl von Klassen über eine Änderung informiert werden kann.
- ▶ Der *Mediator* definiert, wie die Kommunikation zwischen Klassen vereinfacht werden kann. Dazu wird eine weitere Klasse eingeführt, die den anderen Klassen erspart, voneinander zu wissen.
- ▶ Die *Chain of Responsibility* erlaubt eine weitere Entkopplung von Klassen, indem Anfragen solange weitergeleitet werden, bis ein Zuständiger gefunden wird.
- ▶ Das *Template*-Muster bietet eine abstrakte Definition eines Algorithmus.
- ▶ Der *Interpreter* bietet eine Definition, wie Sprachelemente in ein Programm integriert werden können.
- ▶ Das *Strategy*-Muster kapselt einen Algorithmus innerhalb einer Klasse.
- ▶ Das *Visitor*-Muster fasst Operationen verschiedener Klassen an einer Stelle zusammen.
- ▶ Das *State*-Muster bietet Speicher für die Attribute einer Klasse.
- ▶ Das *Memento*-Muster speichert den Zustand einer Klasse bzw. stellt ihn wieder her.
- ▶ Das *Command*-Muster bietet einen einfachen Weg, um die Ausführung eines Kommandos von der Umgebung zu trennen, die es erzeugt hat.
- ▶ Das *Iterator*-Muster formalisiert das Durchlaufen der Daten in einer Liste.

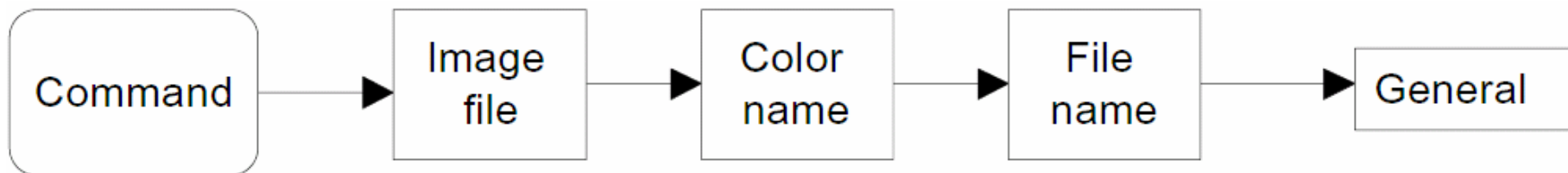
## Verhaltensmuster – Chain of Responsibility

### ▶ Ansatz

- ▶ Ein Kommando (Ereignis, Aufgabe) wird der Reihe nach an verschiedene Objekte in einer Liste geschickt, solange, bis eines der Objekte das Kommando verarbeitet hat

### ▶ Anwendungen

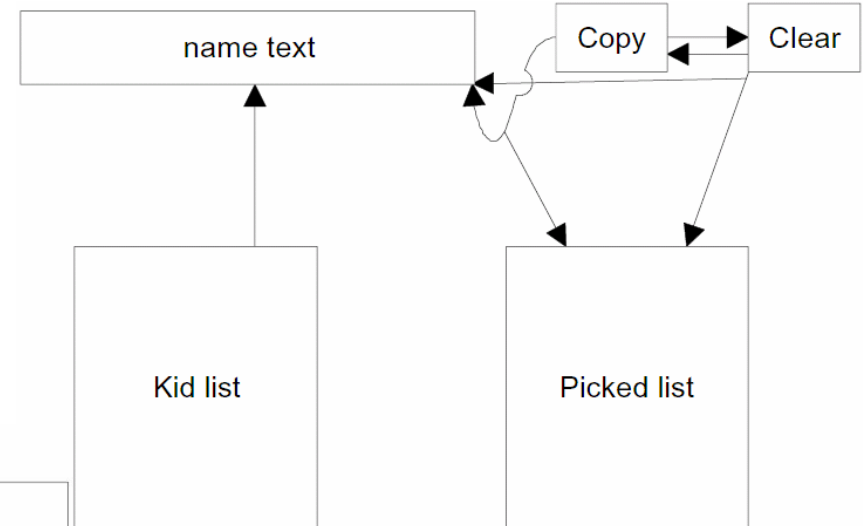
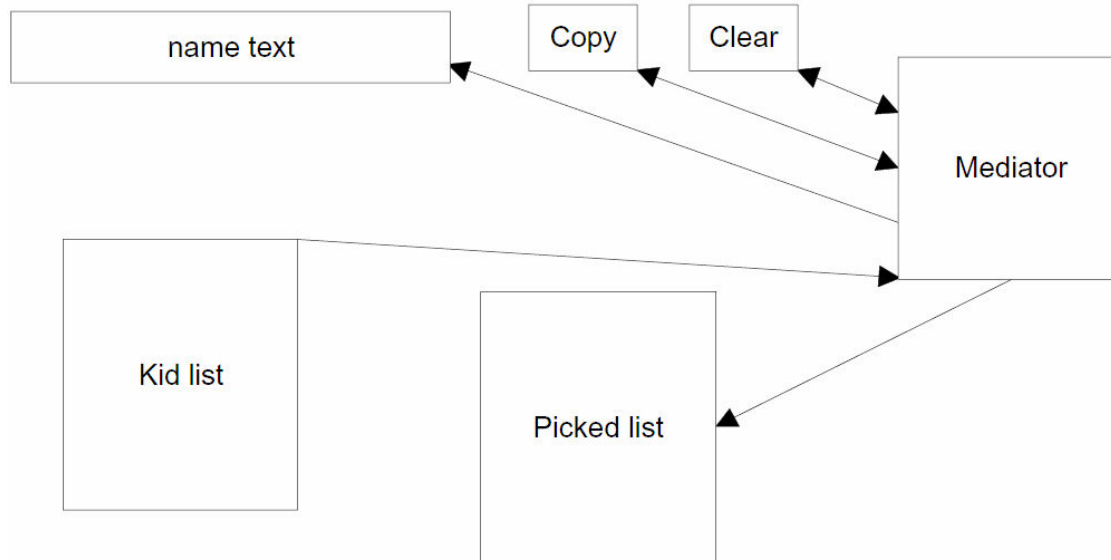
- ▶ Ereignisse in graphischen Benutzungsoberflächen
  - ▶ *Button → Dialog → Dokument → Hauptfenster → Applikation*
- ▶ Anzeigeprogramm für verschiedene Graphikformate
  - ▶ *JPEG → GIF → PNG → TIF → RAW*



# Verhaltensmuster – Mediator

▶ **Ansatz**

- ▶ Durch Einführen eines Mediators gibt es weniger direkte Beziehungen zwischen den Klassen



# Verhaltensmuster – Observer

## ▶ Ansatz

- ▶ Eine Klasse teilt Beobachtern mit, dass sich ihr Zustand geändert hat
- ▶ Diese können dann darauf reagieren

## ▶ Anwendungen

- ▶ Visualisierung von Daten

```
interface Observer
{
    void sendNotify(Notification n);
}

interface Subject
{
    void registerInterest(Observer o);
}
```

