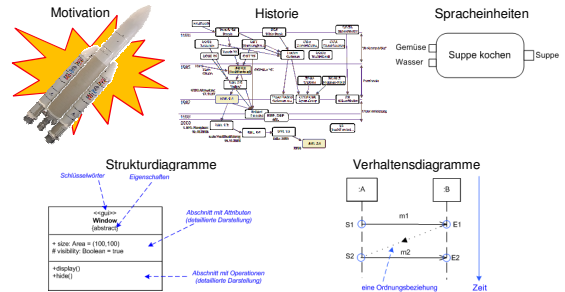


# Entwurfsmuster

Thomas Röfer

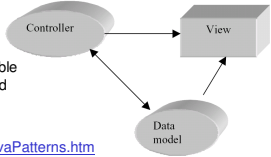
- Motivation
- Erzeugende Muster
- Strukturelle Muster
- Verhaltensmuster

# Rückblick „UML“



# 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>

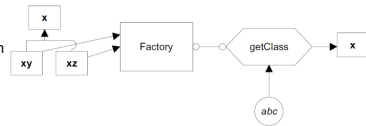


# Erzeugende Muster

- 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 – 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];
    }
}
```

## Erzeugende Muster – Singleton

- **Ansatz**
  - Manchmal wird eine Klasse benötigt, die man nur einmal instanzieren kann
- **Mögliche Umsetzung**
  - Verhinderung der mehrfachen Instanzierung
    - Markierung in statischem Attribut
    - Konstruktor erzeugt *Exception*, falls mehrfach konstruiert
    - Statische Methode liefert neue Instanz oder null, falls mehrfach konstruiert
    - Privater Konstruktor
  - Alles statisch
    - Spätere Umstellung auf mehrere Instanzen schwierig
- **Anwendungen**
  - Anwendungsobjekt
  - Hauptfenster
  - zentrale Verwaltungsaufgaben

```
class Singleton
{
    public static Singleton theInstance
    = new Singleton();
    private Singleton() {}
}
```

## 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
- **Facade-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 – Adapter

- **Ansatz**
  - Ein Adapter kapselt eine existierende Klasse so, dass sie von anderen existierenden Klassen genutzt werden kann
- **Anwendungen**
  - Kombination von existierenden Software-Systemen, z.B. Klassenbibliotheken
  - Dadurch kann das neu schreiben von großen Mengen funktionierender Software vermieden werden

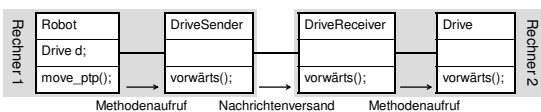
```
class MyIntArray
{
    int a[] = new int[10];
}

interface IntArray
{
    void set (int i, int value);
    int get (int i);
}

class Adapter implements IntArray
{
    MyIntArray m = new MyIntArray ();
    public void set (int i, int value)
    { m.a[i] = value; }
    public int get (int i)
    { return m.a[i]; }
}
```

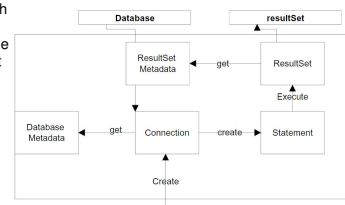
## Strukturelle Muster – Proxy

- **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



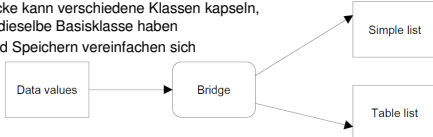
## Strukturelle Muster – Façade

- **Ansatz**
  - Software-Subsysteme, die aus einer großen Menge an Klassen bestehen, sind für Benutzer unübersichtlich
  - Daher kann man sie durch eine *Façade* kapseln, die die öffentliche Schnittstelle an zentraler Stelle enthält



## 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



Pl-2: Entwurfsmuster

13

## 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 *State*-Muster wählt abhängig von einem Zustand unterschiedliche Subklassen für die Ausführung von Methodenaufrufen aus
- Das *Strategy*-Muster kapselt einen Algorithmus innerhalb einer Klasse
- Das *Visitor*-Muster fasst Operationen verschiedener Klassen an einer Stelle zusammen
- Das *Memento*-Muster speichert den Zustand einer Klasse bzw. stellt ihn wieder her
- Das *Iterator*-Muster formalisiert das Durchlaufen der Daten in einer Liste

Pl-2: Entwurfsmuster

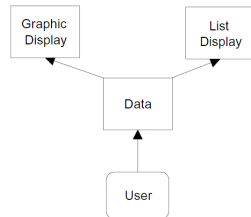
14

## 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);
}
```

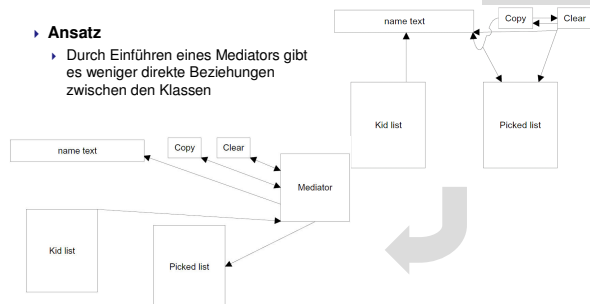


Pl-2: Entwurfsmuster

15

## Verhaltensmuster – Mediator

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

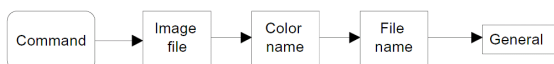


Pl-2: Entwurfsmuster

16

## 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 Benutzeroberflächen
    - Button → Dialog → Dokument → Hauptfenster → Applikation
  - Anzeigeprogramm für verschiedene Graphikformate
    - JPEG → GIF → PNG → TIF → RAW

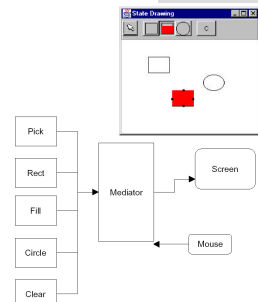


Pl-2: Entwurfsmuster

17

## Verhaltensmuster – State/Strategy

- **Ansatz**
  - Oft befindet sich das Programm in einem (wechselbaren) Zustand, der festlegt, was bei bestimmten Operationen passiert
  - Das *State*-Muster wählt abhängig von diesem Zustand unterschiedliche Subklassen für die Ausführung von Methodenaufrufen aus
  - Das *Strategy*-Muster kommt zur Anwendung, wenn der Benutzer direkt eine von mehreren Alternativen auswählen kann
- **Anwendungen**
  - Interaktive Manipulation von Objekten
  - Modul/Lösungskonzept des GermanTeams



Pl-2: Entwurfsmuster

18

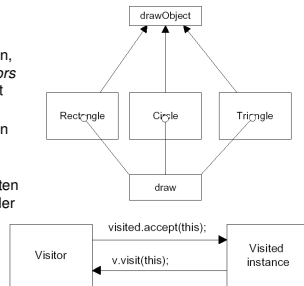
## Verhaltensmuster – Visitor

### Ansatz

- Wenn verschiedene Klassen ähnliche Operationen ausführen, kann diese in Form eines *Visitors* außerhalb der Klassen definiert werden
- Der Visitor ruft bei allen Klassen die Methode *accept* auf, die wiederum ihn aufrufen
- Im *Visitor* wird die zum besuchten Objekt passende Überladung der Methode *visit* aufgerufen

### Anwendungen

- Zusammenfassen von Code



## Verhaltensmuster – Visitor

```

interface Visitor {
    void visit(MyClass m);
}

class MyClass {
    int data;
    void accept(Visitor v) {
        v.visit(this);
    }
}
    
```

```

class MyPrinter implements Visitor {
    void print(MyClass[] ms) {
        for(MyClass m : ms)
            m.accept(this);
    }
    public void visit(MyClass m) {
        System.out.println(m.data);
    }
}
    
```

## Wie geht's weiter?

### Größere Softwaresysteme

- Software-Projekt
- Hauptstudiumsprojekt

### Andere Programmiersprachen

- C++ (TI-2)
- Haskell (PI-3)

### Nach PI-2: Rad nicht neu erfinden

- Vordefinierte Klassen nutzen, z.B. `java.util.*`

### BlueJ und was dann?

- „Richtige“ Entwicklungsumgebungen
  - Eclipse, Visual Studio...*
- Direkte Anbindung an Online-Hilfe
- „IntelliSense“
- Mächtige Debugger
  - Breakpoints, Watchpoints, Edit & Continue...*
- Enge Integration zwischen Entwicklungsumgebung und Laufzeitbibliothek