Verteilte Systeme
Entwurfsmuster

Erfolgreiche Architekturen ...

Die Cheops-Pyramide
Erbaut: 2551 bis 2528 v. Chr.

Das Reichstagsgebäude
Erbaut: 1884 – 1894

Die Glaskuppel
Erbaut: 1995 – 1999

… und weniger erfolgreiche

Der Schiefe Turm zu Pisa
Erbaut: 12. Jahrhundert

Der Untergrund erwies sich als nicht tragfähig genug für den Turm.

Die Knickpyramide
Erbaut: 2570 – 2480 v.Chr

Die Steigung der unteren Pyramide erwies sich als zu steil. Um ein Abrutschen zu verhinden wurde der obere Teil deutlich flacher gebaut.

Die objektorientierte Denkweise

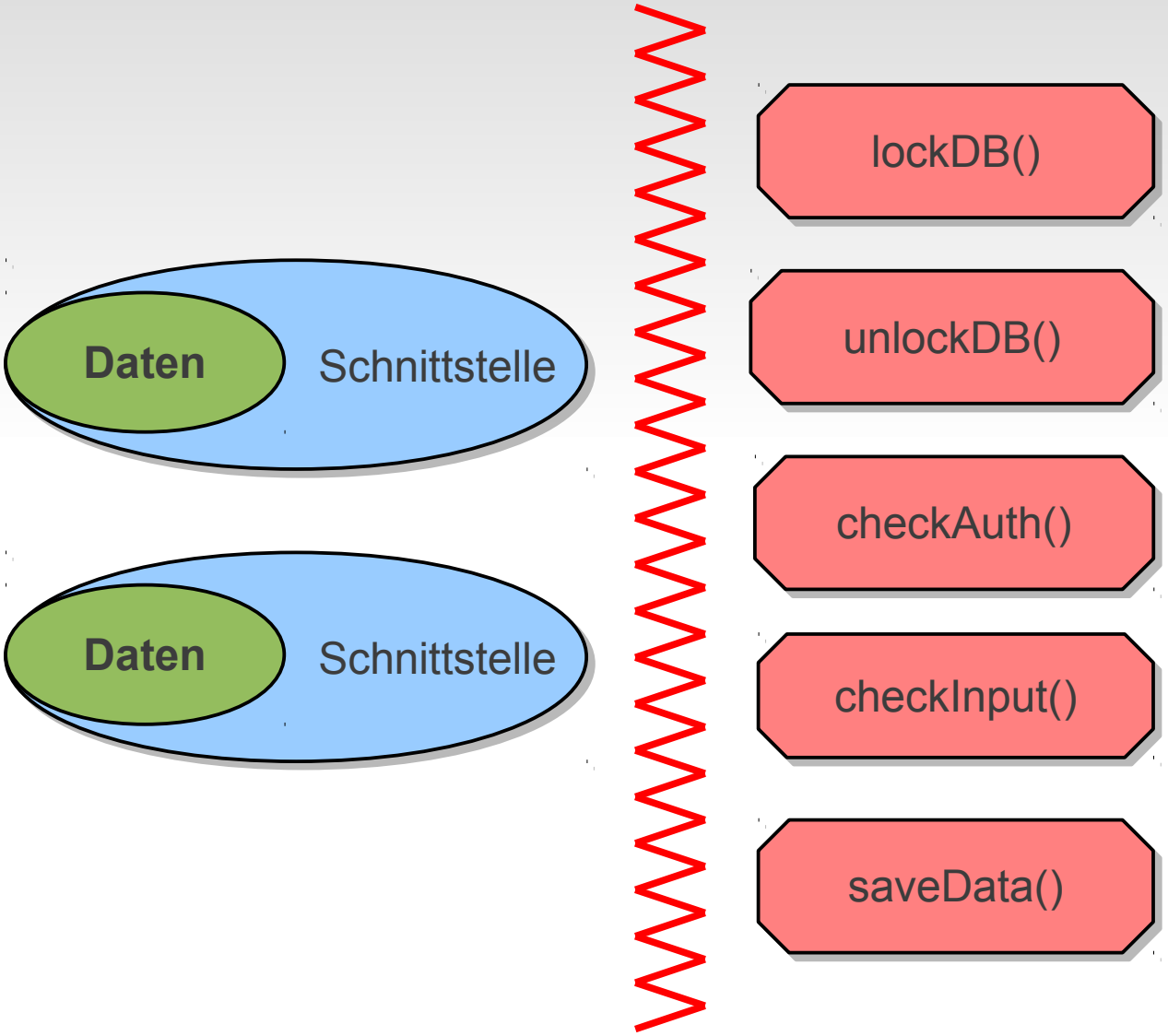
Kapselung

Abstraktion

Vererbung

Polymorphie

Kapselung von Daten



Objektorientierung bietet die **Möglichkeit**, die Daten von ihren Verwendern durch eine **Schnittstelle** zu entkoppeln, da der Direktzugriff auf fremde Daten unterbunden werden kann.

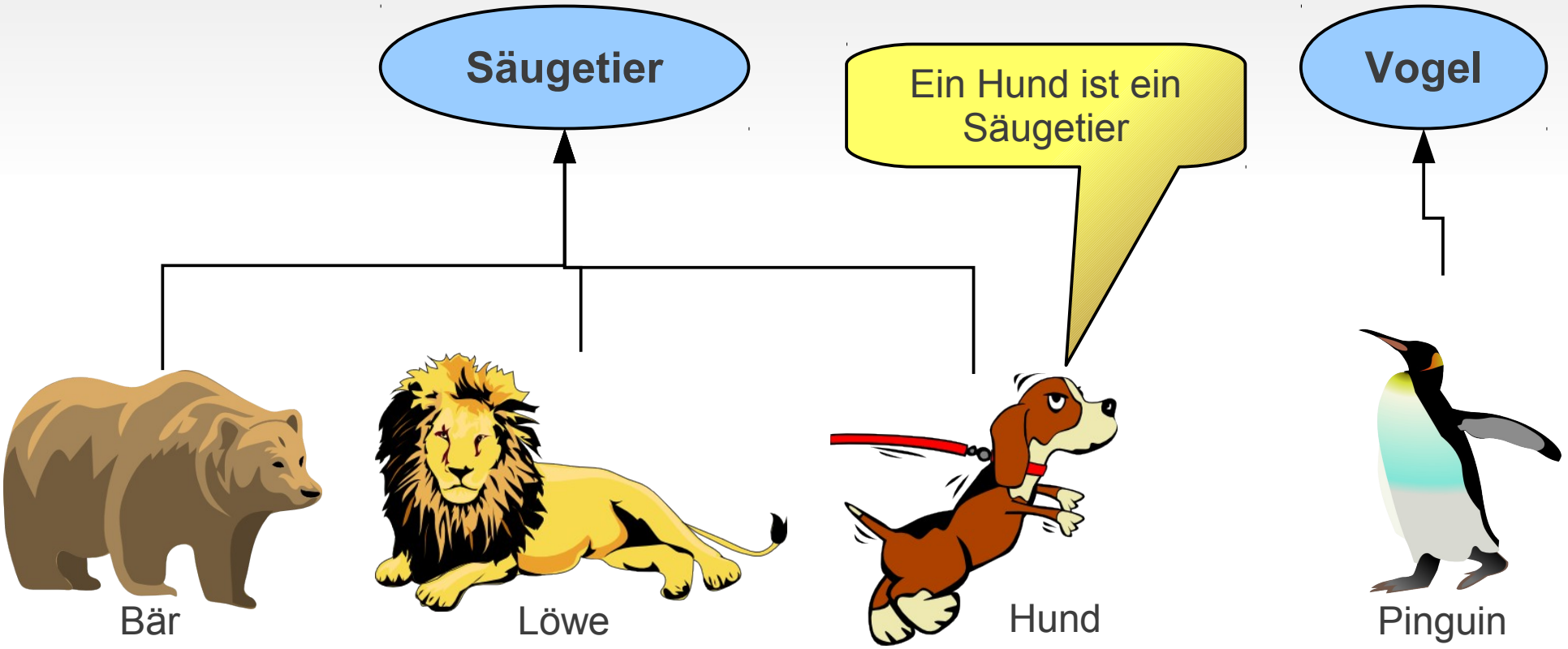
Die Zugriffsschnittstelle besteht aus ausführbaren Methoden. Somit kann sichergestellt werden, dass benötigte Prüfungen und Kontrollen nicht umgangen werden.

Kapselung von Daten

Wenn Sie nicht in Unterhosen durch die Straßen laufen würden, warum sollten es dann Ihre Daten tun?

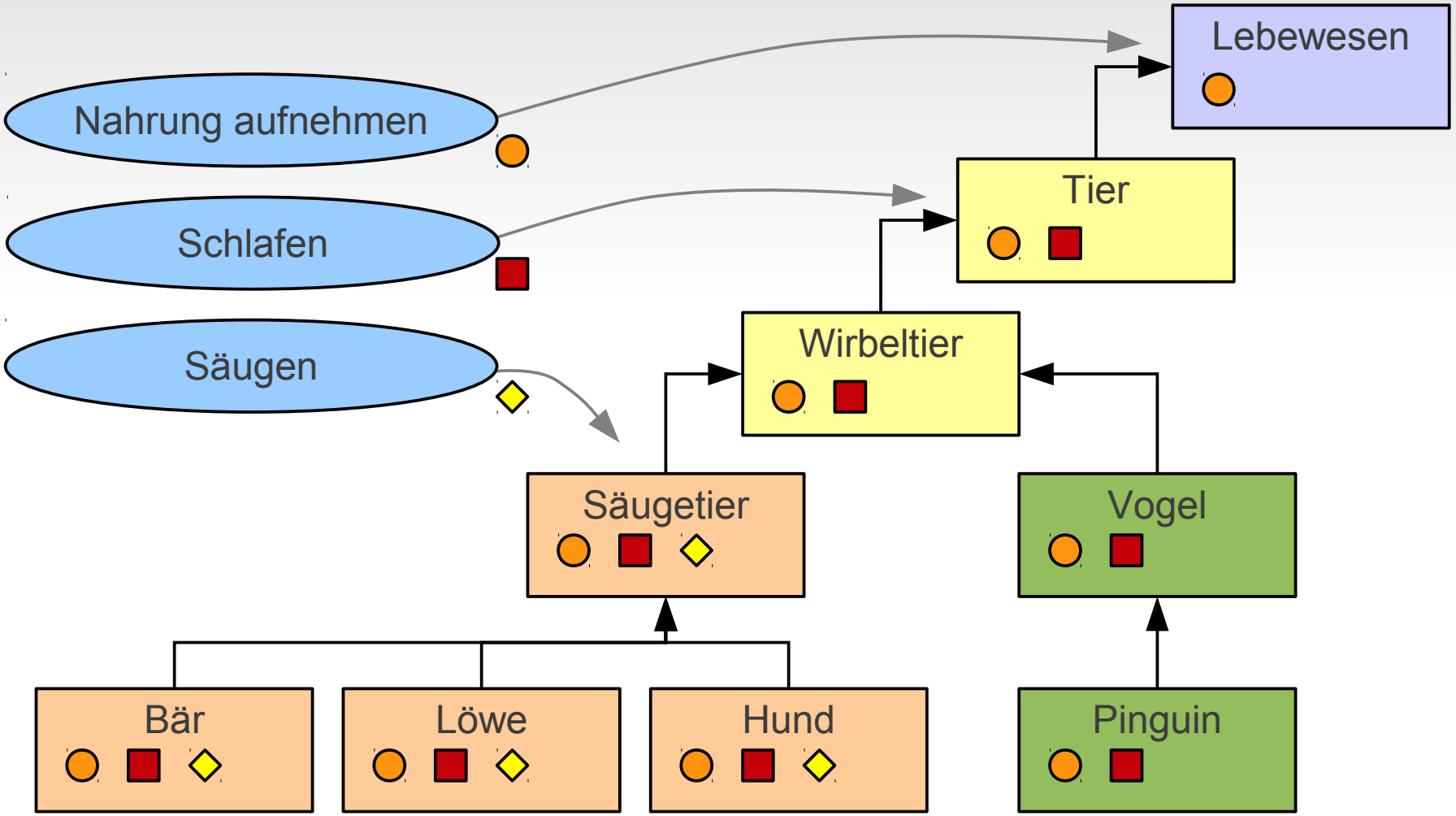
Abstraktion

Bildung von Klassenhierarchien durch Erkennung von *ist ein*-Beziehungen



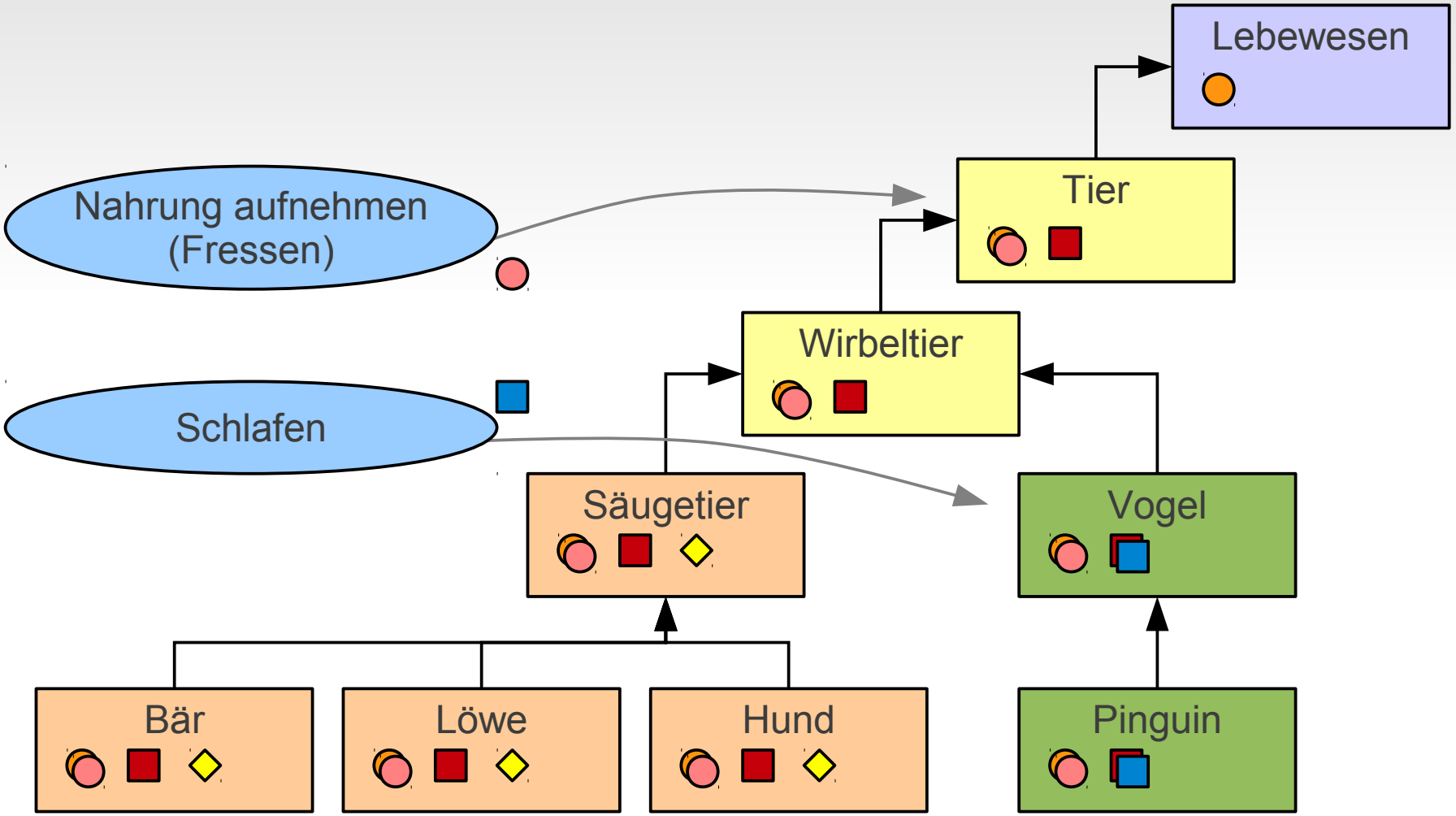
Vererbung

Merkmale und Verhalten in der allgemeinsten Klasse festlegen
Unterklassen erben diese dann



Polymorphie

Verhalten in Unterklassen individuell überschreiben
Namensgleiche Vorgänge mit unterschiedlichen Ausprägungen



Beziehungen zwischen Klassen

Ist X ein Y?

- Ist die eine Klasse ein Spezialfall der anderen?
- Oder ist die eine Klasse eine Verallgemeinerung der anderen?
- Beispiel: Ein Hund ist ein Wirbeltier oder ein Auto ist ein Transportmittel

Hat X ein Y?

- Besitzt ein Objekt ein anderes Objekt (Komposition)?
- Beispiel: Ein Auto hat Räder, aber ein Auto ist kein Rad und ein Rad ist auch kein Auto

Benutzt X ein Y?

- Benötigt ein Objekt ein anderes Objekt, um seine Aufgabe zu erfüllen?
- Beispiel: Ein Auto benutzt eine Straße. Es ist aber keine Straße und es besitzt auch keine. (Die Straße wird ja von mehreren Autos benutzt)

Beispiel: Entenhausen

```
public abstract class Ente {
    abstract public String quaken();

    public void schwimmen() {
        ...
    }
}
```

```
public class Stockente extends Ente {
    public String quaken() { return "Quak!"; }
}
```

```
public class Gummiente extends Ente {
    public String quaken() { return "Quietsch!" }

    public void schwimmen() {
        ...
    }
}
```

Implementieren der abstrakten Methode

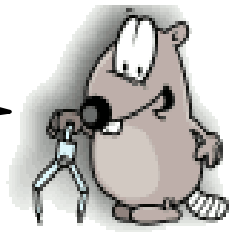
Überschreiben der geerbten Methode

Kleine Änderung ...

Führen wir schnell eine kleine Änderung durch ...

```
public abstract class Ente {
    abstract public String quaken();

    public void schwimmen() {
        // Alle Enten schwimmen gleich
    }

    public void fliegen() {
        // Natürlich können Enten fliegen
    }
}
```

Wir haben eine Methode hinzugefügt




Hey, hast du das gesehen?
Fliegende Gummienten ...

Entwurfsregel Nr. 1

Ein guter Entwurf
ist ein guter Entwurf
ist ein guter Entwurf.

Arbeiten Sie ihn daher
sorgfältig aus!

Entwurfsregel Nr. 2

Seien Sie sich bewusst, welche Verantwortlichkeiten in Ihren Programmen herrschen.

Beispielsweise:

- Anlegen und Suchen von Datensätzen
- Anzeige der Daten, Benutzerführung
- Validierung von Benutzereingaben
- Berechtigungsprüfungen
- Fachliche Berechnungen und Prüfungen
- Datenaustausch übers Netzwerk

Entwurfsregel Nr. 3

Teile und Herrsche:

Vermischen Sie niemals zwei Verantwortlichkeiten in einer Klasse

Darauf sollten Sie achten

Kapseln Sie, was sich verändert in einer eigenen Klasse

- Minimiert die Anpassung bestehender Methoden
- Denn es ist einfacher, eine Methode um einen Aufruf zu erweitern, als um einen neuen Algorithmus

Jede Klasse sollte für Erweiterungen offen aber gegen Veränderungen gesperrt sein

- Neue Methoden hinzuzufügen ist in Ordnung
- Änderungen an alten Methoden sind allerdings zu vermeiden

Ziel: Möglichst keine Änderungen an bestehendem Quellcode

Noch mehr Entwurfsprinzipien

Hinsichtlich zukünftiger Erweiterungen:

- Kapseln Sie, was sich verändert (in je einer eigenen Klasse)
- Eine Klasse sollte nur einen Grund für Änderungen haben
- Klassen sollten für Erweiterungen offen aber gegen Veränderungen gesperrt sein (Offen/Geschlossen-Prinzip)

Hinsichtlich dem Zusammenspiel der Objekte:

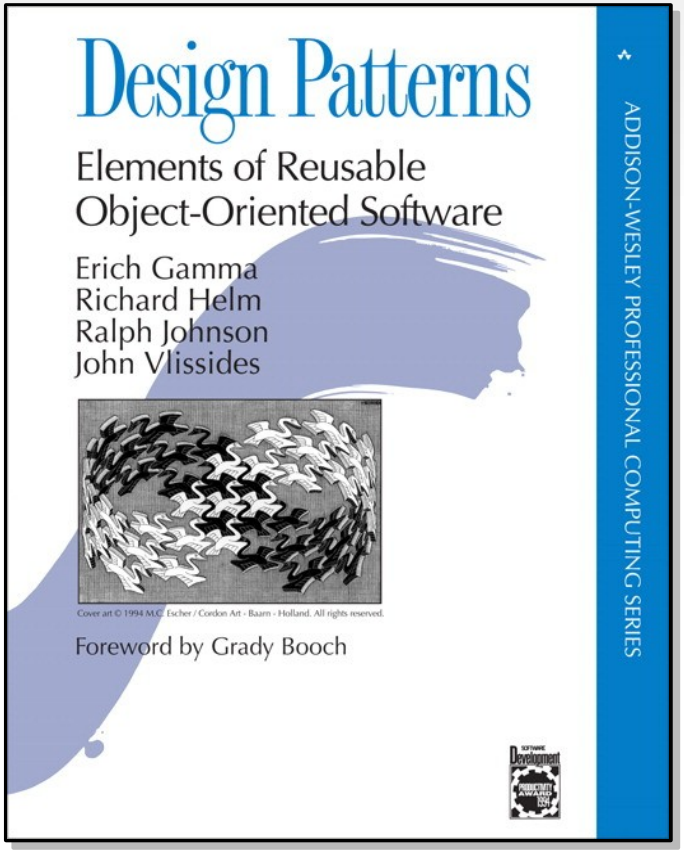
- Programmieren Sie gegen Schnittstellen und Abstraktionen, aber nicht gegen konkrete Implementierungen
- Ziehen Sie Komposition der Vererbung vor

Noch mehr Entwurfsprinzipien

- Vermeiden Sie zu enge Bindungen (zwischen den Klassen)
- Erzeugen Sie interagierende Objekte mit lockerer Kopplung

Noch mehr Entwurfsprinzipien

- Sprechen Sie nur mit Ihren Freunden
- Don't call us, we'll call you! (Hollywood-Prinzip)



Heuristik vs. Entwurfsmuster

Einfache Heuristiken

- Praxiserprobte Verhaltensweisen als allgemeine Richtlinie
- Verallgemeinerung der langjährig gesammelten Erfahrung
- Lassen sich oft in wenigen Sätzen beschreiben
- Werden häufig unbewusst eingesetzt (Vgl. Entwurfsregeln)

Entwurfsmuster

- Allgemeine Lösungen für wiederkehrende Aufgaben
- Ausführlich dokumentierte Feinentwürfe echter Probleme
- Werden in Musterkatalogen dokumentiert und publiziert
- Angelehnt an Patterns in der Architektur (z.B. 6 Foot Balcony)

Beispiel: Heuristiken

Separation of Concerns

Sorgen Sie für eine klare Trennung der Zuständigkeiten. Jedes Anwendungsmodul sollte nur eine Aufgabe haben.

Information Hiding Modules

Ein Modul sollte alle Details seiner Implementierung vor seinen Aufrufern verbergen. Die öffentliche Schnittstelle des Moduls sollte daher so wenig Parameter wie möglich besitzen.

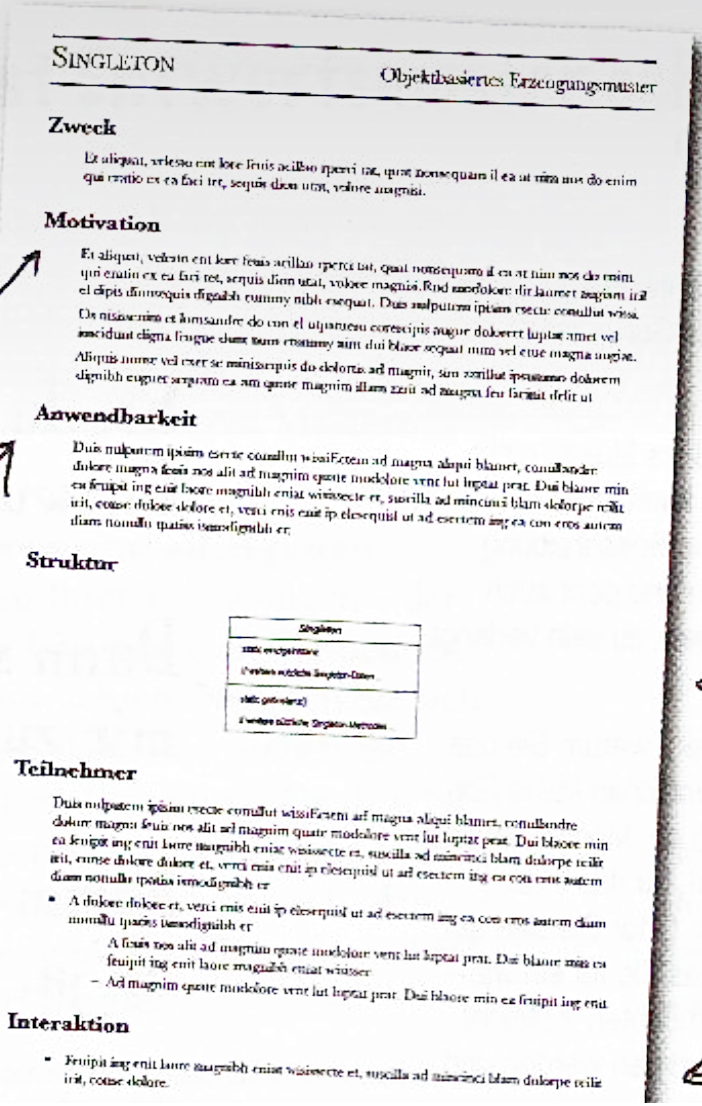
Beispiel: Entwurfsmuster

Alle Muster in einem Katalog beginnen mit einem Namen. Der Name ist ein lebenswichtiger Teil des Patterns – ohne einen guten Namen wird ein Muster nicht Teil des Vokabulars, das Sie mit anderen Entwicklern teilen. (Daher verwenden wir in dieser Ausgabe auch die englischen Originalnamen, auf die kann man sich einigen, während es zu viele (schlechte) deutsche Übertragungen gibt.)

Die Motivation liefert Ihnen ein konkretes Szenario, das das Problem und die Art der Problemlösung beschreibt.

Unter Anwendbarkeit werden Situationen beschrieben, in denen das Muster angewendet werden kann.

Die Teilnehmer sind die Klassen und Objekte in dem Design. Dieser Abschnitt beschreibt ihre Zuständigkeiten und Rollen in dem Entwurfsmuster.



Besser leben mit Mustern

SINGLETON — Objektbasiertes Erzeugungsmuster

Zweck

Motivation

Anwendbarkeit

Struktur

Teilnehmer

Interaktion

Dies ist die Klassifikation oder Kategorie des Musters. Ein paar Seiten weiter unten gehen wir darauf noch ein.

Der Zweck beschreibt in einem kurzen Satz, was das Muster macht. Sie können das auch als die Definition des jeweiligen Musters betrachten (so wie wir es in diesem Buch bereits gemacht haben).

Unter Struktur finden Sie ein Diagramm, das die Beziehungen zwischen den an dem Muster beteiligten Klassen wiedergibt.

Inhalte eines Entwurfsmusters

Name
Jedes Muster besitzt einen eindeutigen Namen

Zweck, Motivation
Kurzbeschreibung des Problems und Erläuterung desselben an einem Beispiel

Anwendbarkeit
Aufzählung, wann das Muster sinnvoll ist und wann es besser nicht verwendet werden sollte

Verwandte Muster
Verweise auf ähnliche oder häufig zusammen genutzte Muster

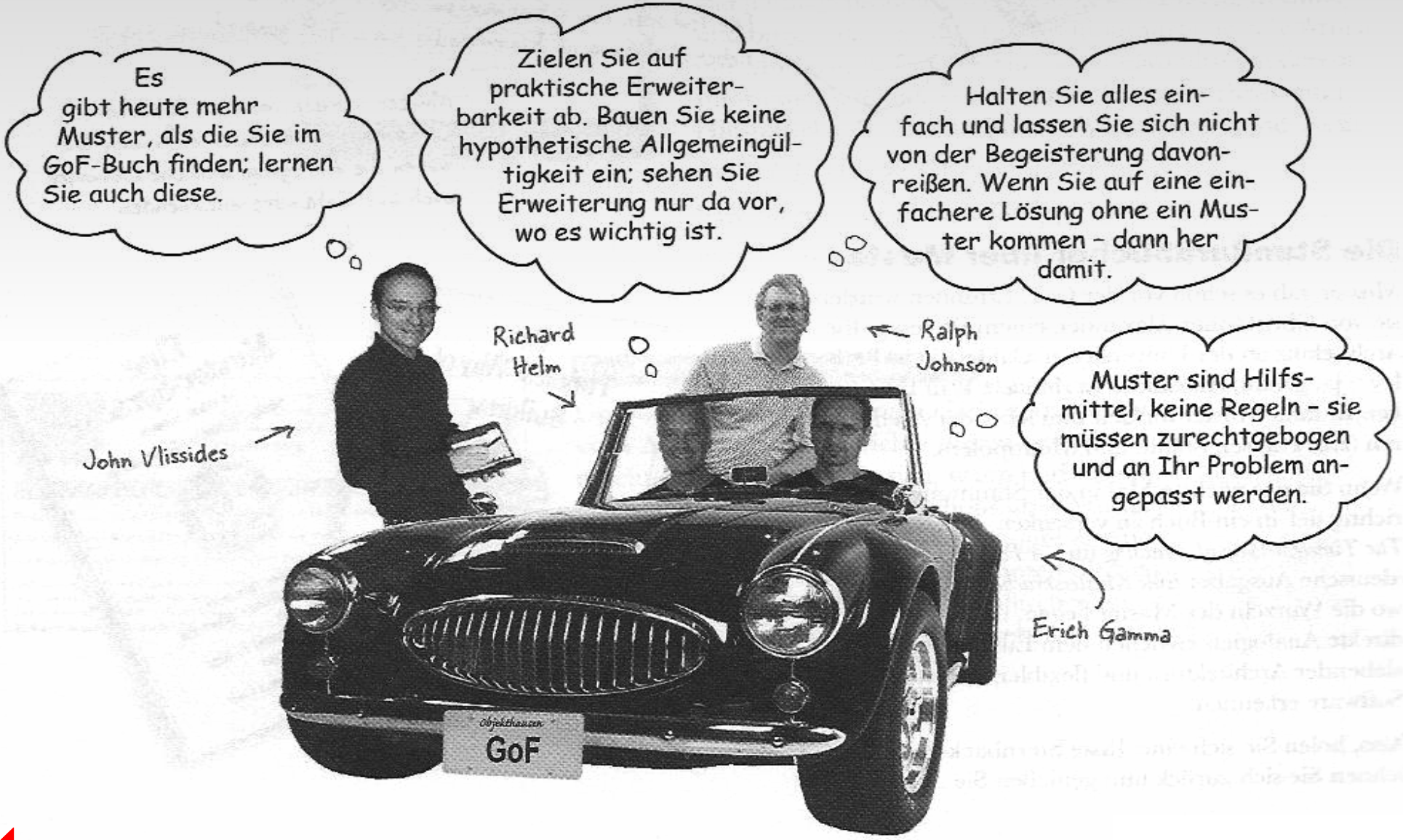
Struktur, Teilnehmer
Graphische Darstellung und Beschreibung aller Klassen und Objekte

Interaktion, Konsequenzen
Zusammenarbeit der Teilnehmer und Auswirkungen auf den Entwurf

Beispielcode
Ausführliche Diskussion anhand einer Beispielimplementierung

Bekannte Verwendungen
Beispiele für den erfolgreichen Einsatz des Musters

The Gang of Four ...



... und ihre Muster

	Erzeugungsmuster	Strukturmuster	Verhaltensmuster
Klassenbasiert	Fabrikmethode	Adapter (Nur mit Mehrfachvererbung)	Interpreter Schablonenmethode
Objektbasiert	Abstrakte Fabrik Erbauer Prototyp Singleton	Adapter (Objektbasiert) Brücke Dekorierer Fasade Fliegengewicht Kompositum Proxy	Befehl Beobachter Besucher Iterator Memento Strategie Vermittler Zustand Zuständigkeitskette

POSA/Siemens-Patterns

Bekannte Muster aus folgenden Bereichen:

Idiome

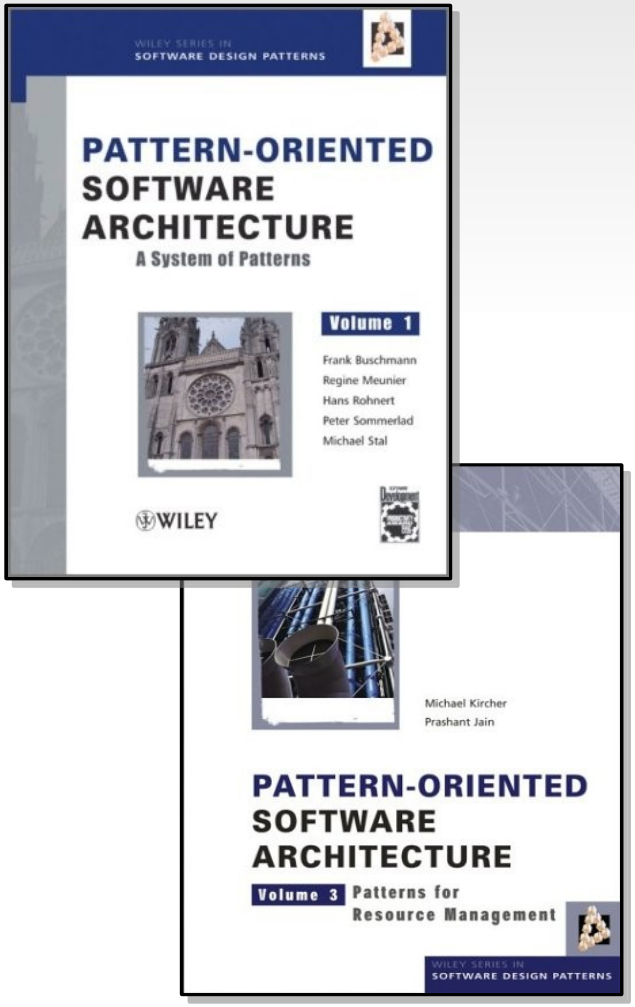
- Unterste Abstraktionsebene
- Sprachspezifisch
- Zum Beispiel Smart Pointer Idiom (C++)

Entwurfsmuster

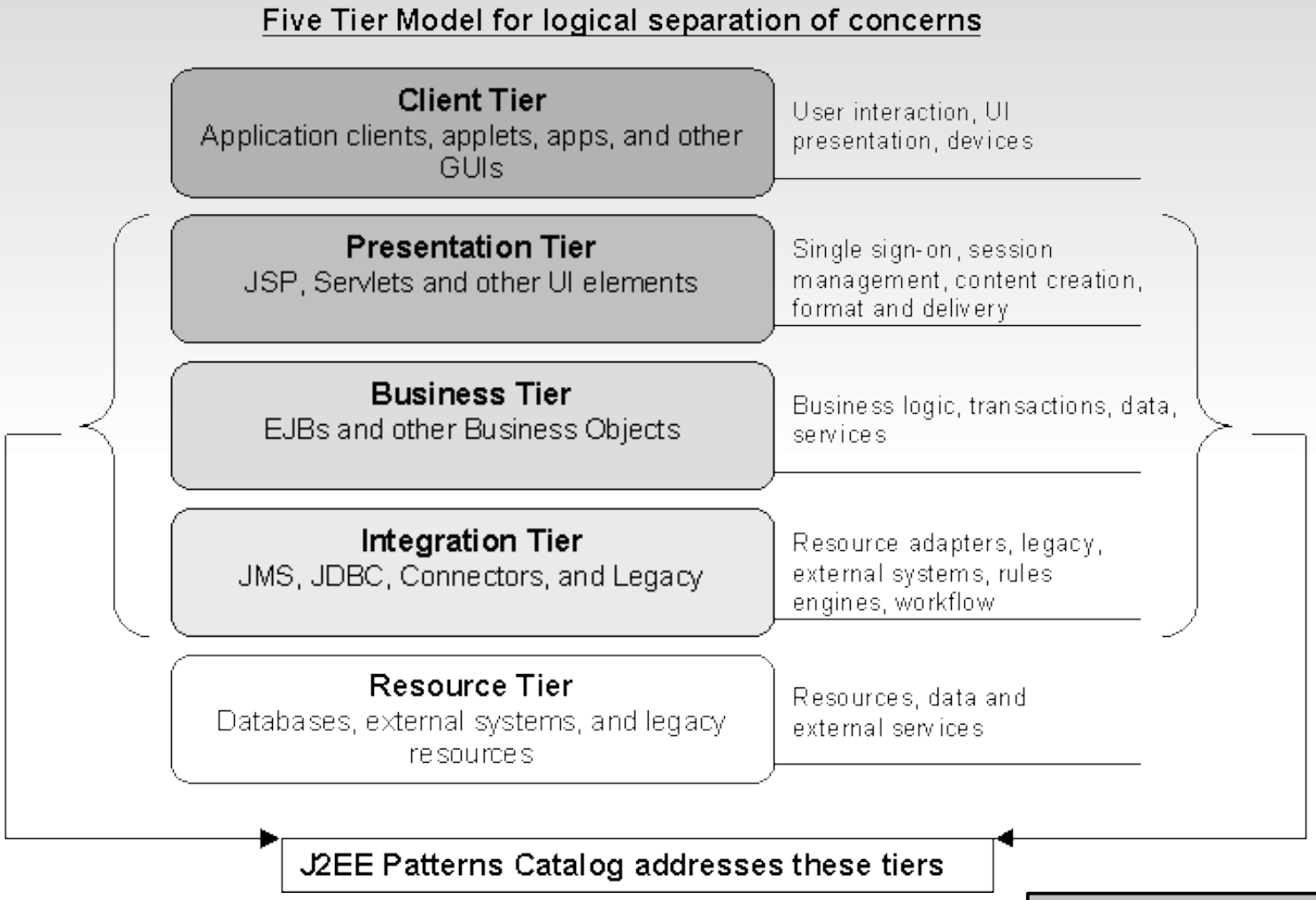
- Vergleichbar mit den GoF-Mustern
- Aber weniger umfangreich

Architekturmuster

- Der eigentliche Mehrwert von POSA
- Zum Beispiel MVC oder Layer-Pattern



Java EE-Patterns



Patterns für Java Enterprise
Von Sun/Oracle und der Community
Nach Schichten gruppiert

Vorteile von Design Patterns

Spielen Sie Schach? Dann kennen Sie sicher den Unterschied zwischen einer Kurzeinführung, die erklärt, wie die einzelnen Figuren ziehen, und einem Lehrbuch, das zeigt, wie man gewinnt.

"Entwurfsmuster" ist das Buch, das zeigt, wie man mit objektorientierter Programmierung (OOP) gewinnt. Es gibt viele Programmiersprachen, und noch mehr Bücher, die sie beschreiben und die auch die OOP erklären. Aber was fange ich mit den Objekten an? Kapriziere ich mich auf Vererbung und leite alles und jedes von "Thing" ab - und halse mir dabei bucklige Verwandschaften und kaum zu durchschauende Seiteneffekte ein? Erstelle ich "eierlegende Wollmilchmonster" oder beschränken sich meine Objekte auf nur leicht verkomplizierte Datentypen, die ich dann doch wieder, im Grunde prozedural, versuche miteinander zur Zusammenarbeit zu bewegen?

Wenn ich weiß, wie die Schachfiguren ziehen, so hat mir noch keiner gesagt, warum das Pferdchen am Rand ungünstig steht, was eine italienische Eröffnung ist oder wie man eine Königsfestung angreift. Die Muster fehlen.

-- Kundenrezession auf Amazon zum GoF-Buch Entwurfsmuster

Vorteile von Design Patterns

Entwurfsmuster …

- Bieten erprobte Lösungsansätze
- Ermöglichen dadurch bessere Entwürfe
- Erhöhen somit auch die Codequalität
- Können jedoch beliebig angepasst werden
- Bieten dennoch ein gemeinsames Vokabular
- Verbessern somit die Kommunikation unter Entwicklern
- Finden sich überall in den Java APIs
- Werden entdeckt und nicht erfunden
- Sollten jedoch sparsam eingesetzt werden

Beispiel: Vokabular

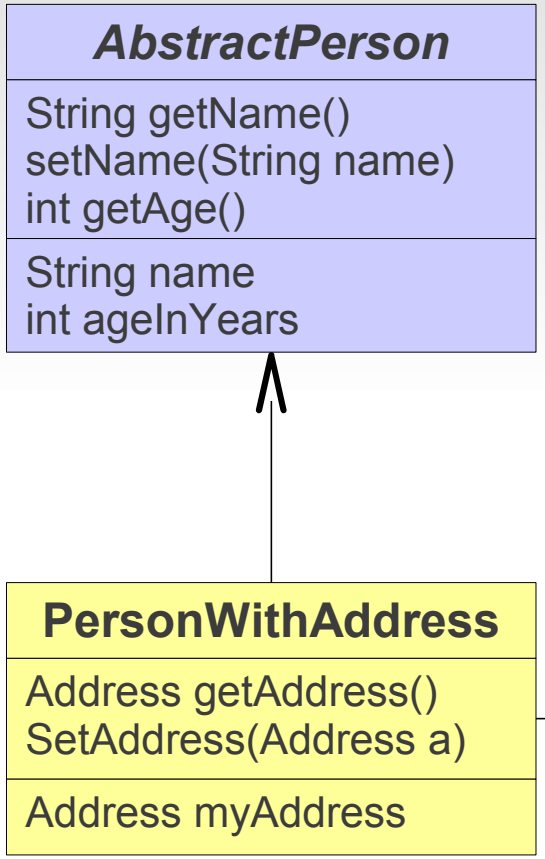
Ohne Design Patterns:
(Unvollständig, Verwirrend, Zeit verschwendend)

Also, ich habe da diese Radioklasse erstellt. Sie behält alle Objekte im Auge, die ihr lauschen, und jedes mal, wenn ein neues Stückchen Information ankommt, benachrichtigt sie jeden dieser Zuhörer. Toll daran ist, dass die Zuhörer sich jederzeit an das Radio anschließen können und sich auch selbst wieder entfernen können. Und die Radioklasse selbst weiß gar nichts über die Zuhörer; jedes Objekt, das das richtige Interface implementiert, kann sich registrieren.

Mit Design Patterns:
(Kurz, Präzise, Vollständig)

Observer

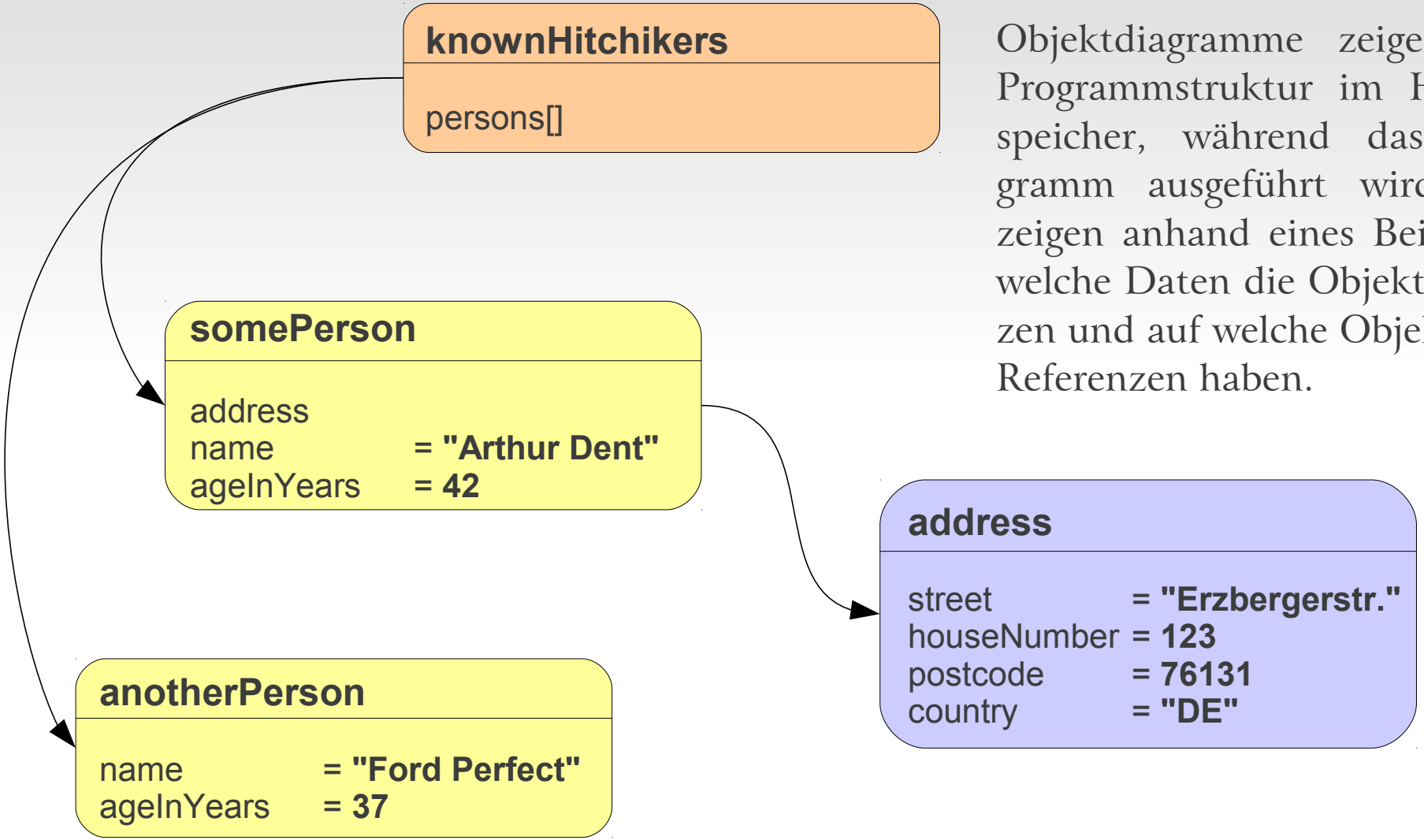
Notation: Klassendiagramme




```
public abstract class AbstractPerson {
    private String name;
    private int ageInYears;

    public String getName() { ... }
    public void setName(String name) { ... }
    public int getAge() { ... }
}

public class PersonWithAddress
extends AbstractPerson {
    private Address myAddress;

    public Address getAddress() { ... }
    public void setAddress(Address a) { ... }
}
```

Klassendiagramme zeigen die Klassen und ihre Beziehungen untereinander. Der offene Pfeil beschreibt die Vererbung, der geschlossene Pfeil eine Referenz (Komposition).

Notation: Objektdiagramme



Objektdiagramme zeigen die Programmstruktur im Hauptspeicher, während das Programm ausgeführt wird. Sie zeigen anhand eines Beispiels, welche Daten die Objekt besitzen und auf welche Objekte sie Referenzen haben.

Unser erstes Muster

Muster: Fassade

Zweck

- Vereinfachung einer sehr komplexen Schnittstelle
- Mehrere, komplizierte Objekte vor dem Aufrufer verstecken

Motivation

- Eine Heimkinoanlage besteht aus sehr vielen Komponenten
- Einen Film anzuschauen ist daher viel zu kompliziert!

Blueray-Player
Automatische Leinwand
Surround-Anlage
Endstufe und Boxen
Radio-Tuner
Popcorn-Maschine
Dimmbare Beleuchtung

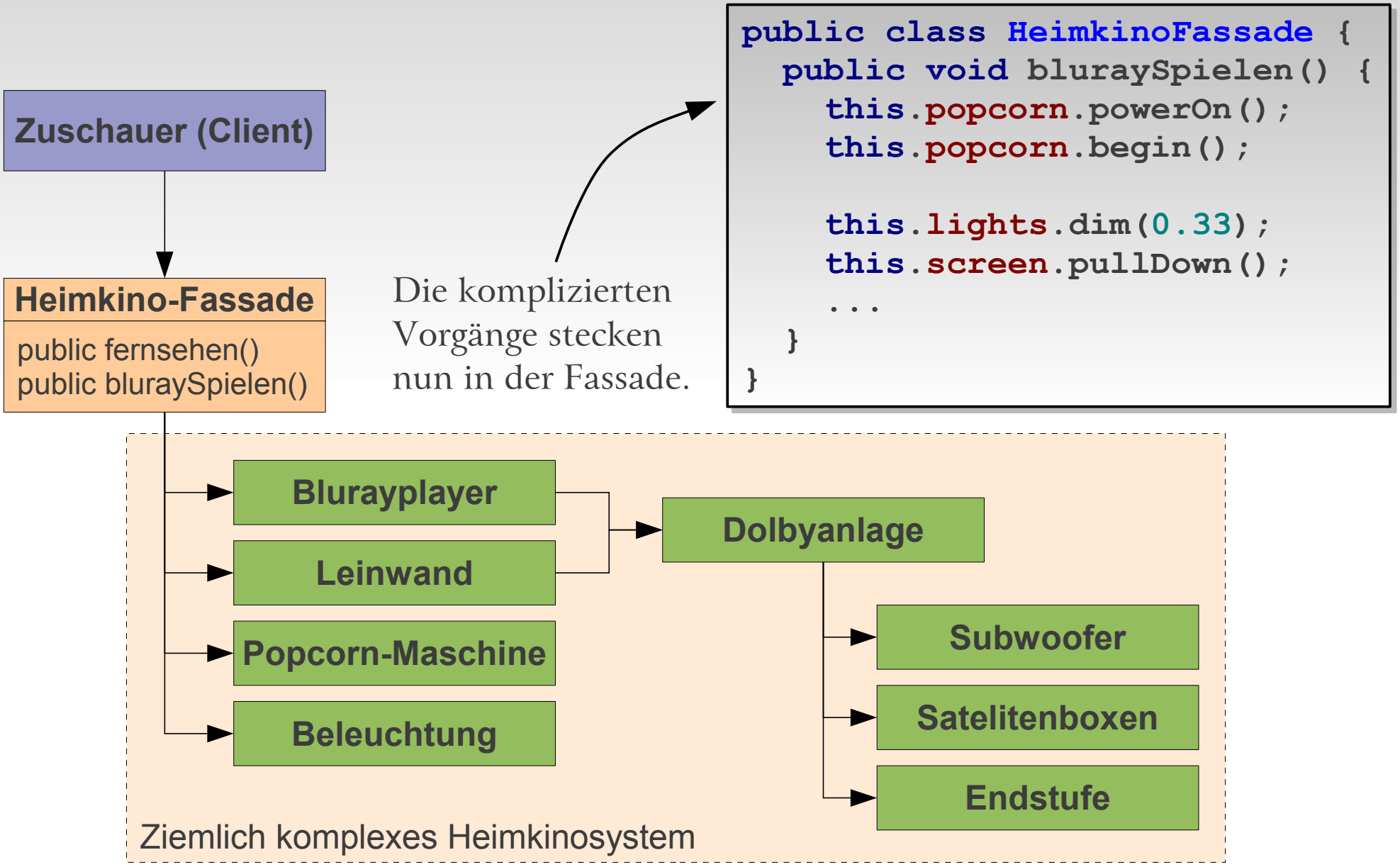
Einen Film anschauen (auf die harte Tour)

Wählen Sie eine DVD aus, entspannen Sie sich und machen Sie sich für den Kinozauber bereit. Ach, eine Sache hätten wir noch vergessen – um einen Film zu schauen, müssen Sie ein paar einfache Aufgaben erledigen:

1. Die Popcorn-Maschine einschalten
2. Die Popcorn-Maschine starten
3. Die Beleuchtung dimmen
4. Die Leinwand herunterfahren
5. Den Beamer einschalten
6. Den Beamer-Eingang auf DVD schalten
7. Den Beamer in den Breitwandmodus versetzen
8. Den Verstärker einschalten
9. Den Verstärkereingang auf DVD setzen
10. Den Verstärker auf Surround-Sound setzen
11. Die Verstärkerlautstärke auf Mittel (5) setzen
12. Den DVD-Spieler einschalten
13. Den DVD-Spieler starten

```
this.popcorn.powerOn();
this.popcorn.begin();

this.lights.dim(0.33);
this.screen.pullDown();

this.beamer.switchOn();
this.beamer.setMode(DVD);
this.beamer.setMode(WIDE_SCR);

...
```

Muster: Fassade



Muster: Singleton

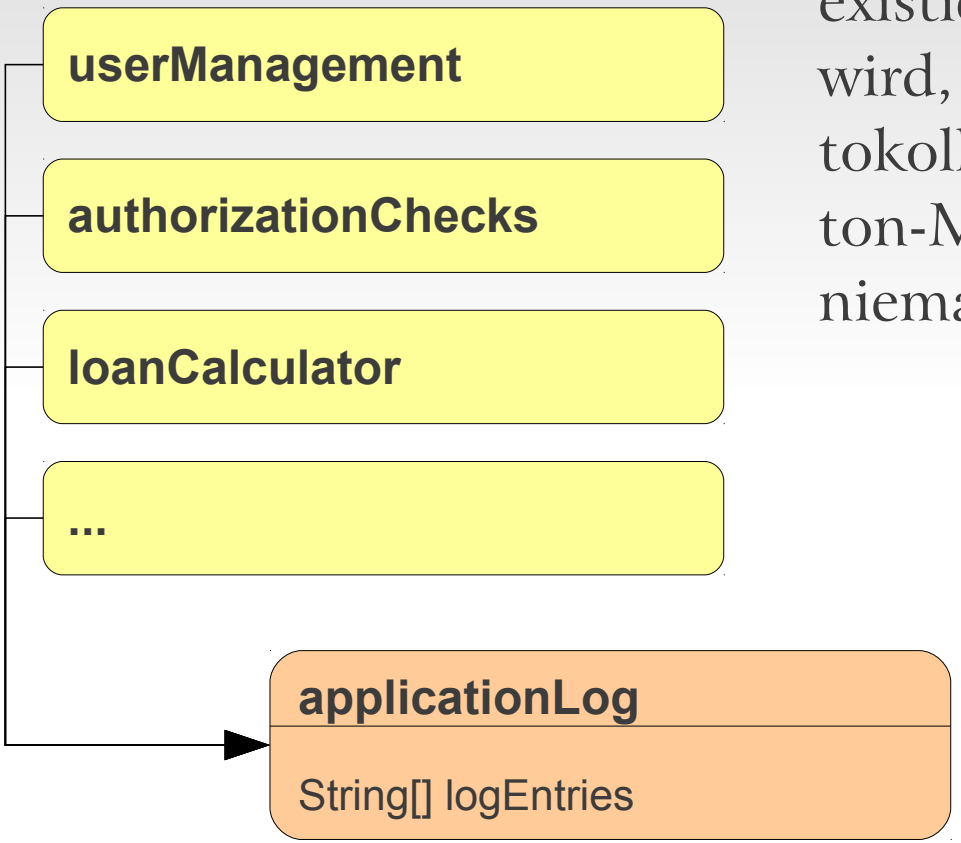
Zweck

- Sicherstellen, dass es von einer Klasse nur ein Objekt gibt
- Zentrale Objekte nicht in globalen Variablen ablegen
- Kontrollieren, ob ein Objekt erzeugt werden darf oder nicht

Motivation

- Eine große Anwendung soll verschiedene Aktionen protokollieren
- Das Protokoll wird an vielen Stellen im Programm fortgeschrieben
- Zum Schreiben des Protokolls existiert eine spezielle Klasse
- Da es nur ein Protokoll gibt, kann es aber nur ein Objekt geben

Muster: Singleton



Innerhalb der gesamten Anwendung existiert nur ein Objekt, welches genutzt wird, um Einträge ins Anwendungsprotokoll zu schreiben. Durch das Singleton-Muster wird sichergestellt, dass es niemals zwei solche Objekte geben kann.

```
log = ApplicationLog.getInstance();
log.warning("Invalid username!");
```

Muster: Singleton

Singleton
public static Singleton getInstance() private Singleton()
private static Singleton instance

Dadurch dass der Konstruktor **private** ist, können außerhalb der Klasse keine Objekte von ihr erzeugt werden. Um ein Objekt zu erhalten muss immer **getInstance()** aufgerufen werden.

Nicht threadsichere Variante:

```
public class Singleton {
  private static Singleton instance;
  private Singleton() { ... }

  public Singleton getInstance() {
    if (Singleton.instance == null) {
      Singleton.instance = new Singleton();
    }

    return Singleton.instance;
  }
}
```

```
log = ApplicationLog.getInstance();
log.warning("Invalid username!");
```

Muster: Singleton

Singleton
public static Singleton getInstance() private Singleton()
private static Singleton instance

In der Methode **getInstance()** können beliebige, weitere Prüfungen untergebracht werden. Zum Beispiel können hier Berechtigungen geprüft werden, ob ein Objekt überhaupt erzeugt werden darf.

Threadsichere Variante:

```
public class Singleton {
 private static Singleton instance;
 private Singleton() { ... }

 public Singleton getInstance() {
  synchronized(Singleton.class) {
   if (Singleton.instance == null) {
    Singleton.instance = new Singleton();
   }
  }

  return Singleton.instance;
 }
}
```

Muster: Proxy

Zweck

- Zugriffe auf ein Objekt durch einen Stellvertreter kontrollieren

Anwendungsbeispiele

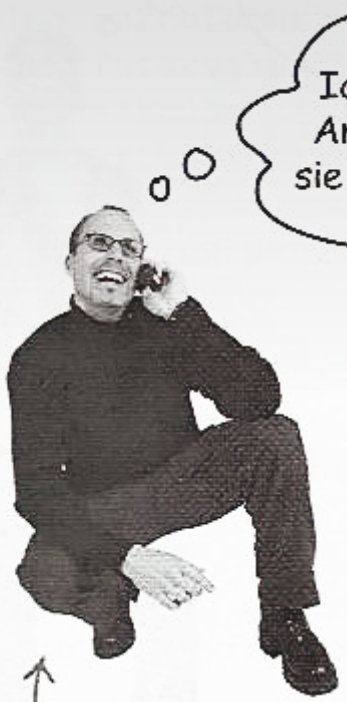
- Aufruf von Methoden auf einem entfernten Computer (RMI)
- Möglichst späte Erzeugung von speicherintensiven Objekten
- Methodenaufrufe mit Berechtigungsprüfungen versehen

Motivation

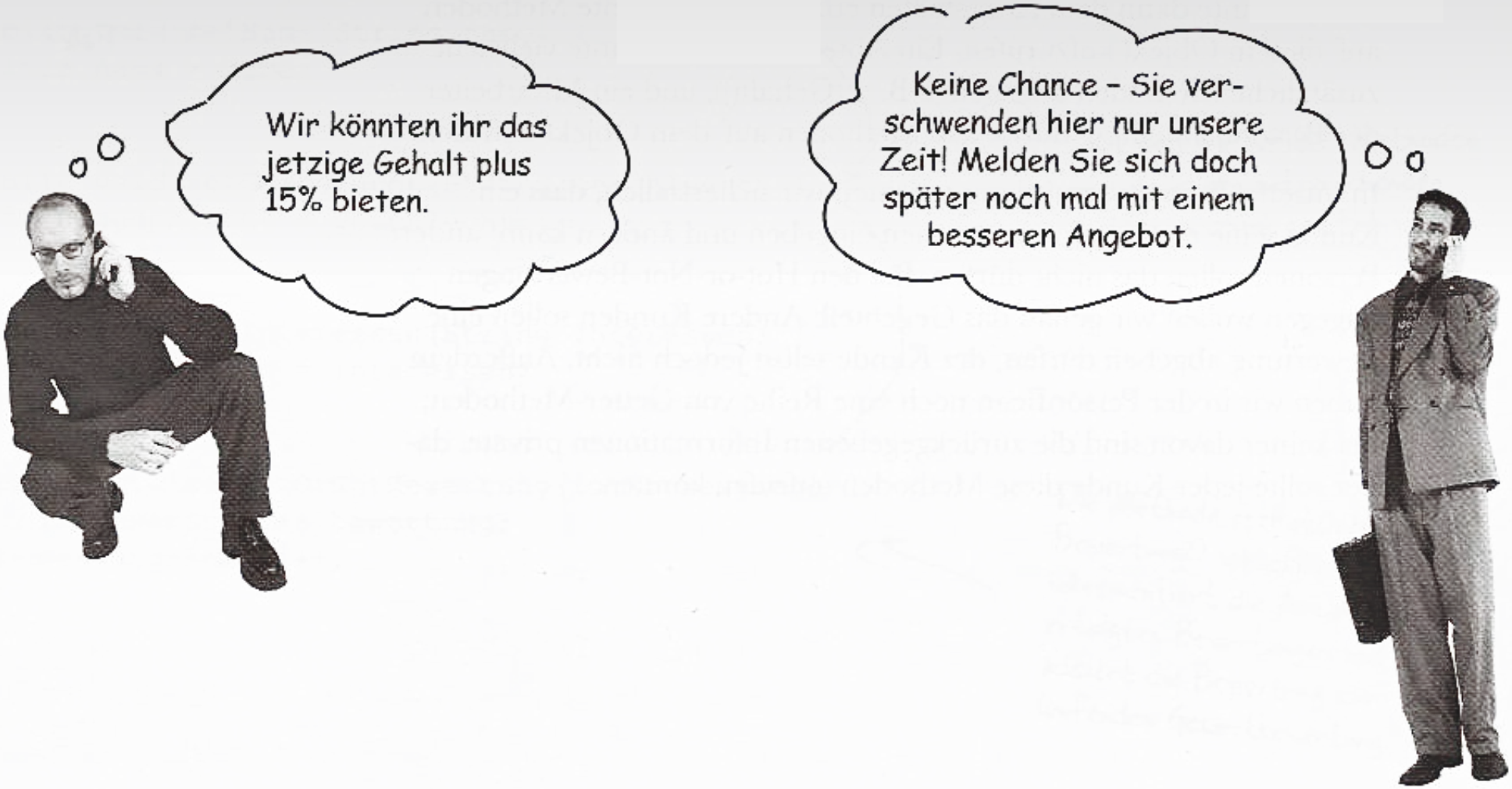
- Eine Textverarbeitung erlaubt es, Bilder im Text zu verwenden
- Die Ladezeit eines Dokuments soll sehr schnell sein
- Jedes Bild soll daher erst geladen werden, wenn es benötigt wird
- Anstelle eines ungeladenen Bilds soll ein Platzhalter erscheinen

Kurzdrama: Objektschutz

Die Internetblase ist geplatzt und nur noch eine blasse Erinnerung. In jenen fernen Tagen brauchten Sie nur über die Straße zu gehen, um einen besseren, höher bezahlten Job zu finden. Es waren sogar Agenten für Software-Entwickler in Mode ...



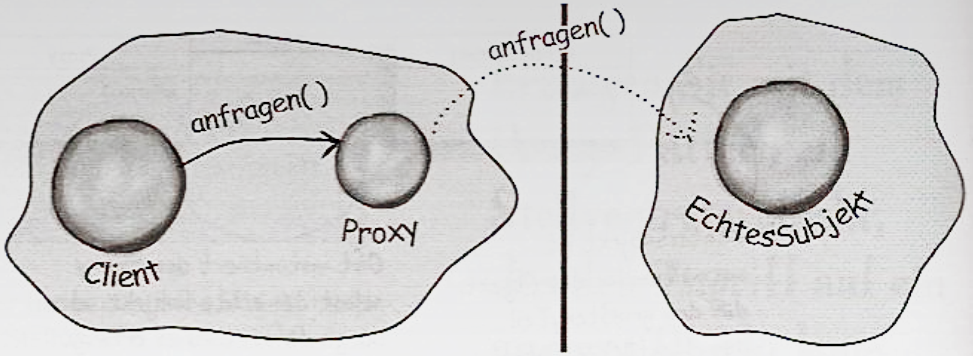
Muster: Proxy



Muster: Proxy

Remote-Proxy

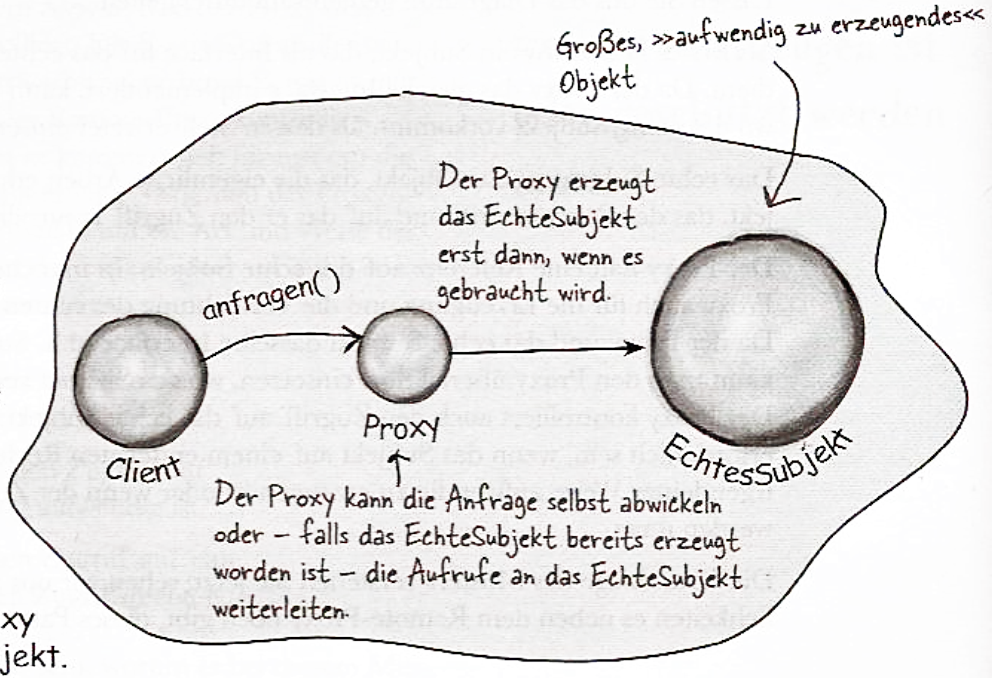
Beim Remote-Proxy fungiert der Proxy als lokaler Vertreter für ein Objekt, das in einer anderen JVM existiert. Ein Methodenaufruf auf dem Proxy bewirkt, dass der Aufruf über das Netz übertragen wird, d.h. ein Fernaufruf stattfindet. Das Ergebnis wird an den Proxy zurückgegeben und von diesem an den Client weitergeleitet.



Dieses Diagramm kennen wir mittlerweile schon ziemlich gut ...

Virtueller Proxy

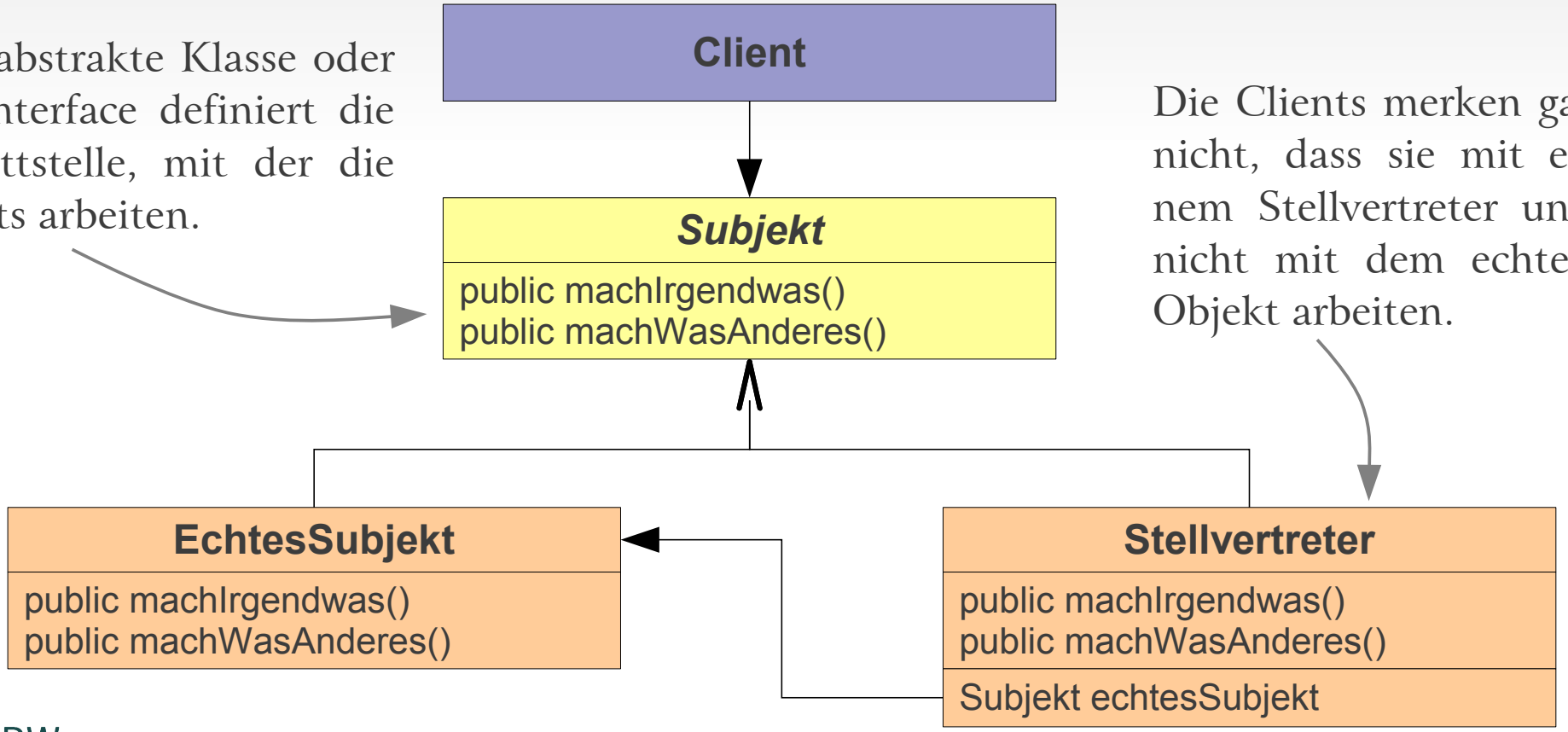
Der virtuelle Proxy fungiert als Vertreter für ein Objekt, dessen Erzeugung möglicherweise aufwendig ist. Häufig schiebt der virtuelle Proxy die Erzeugung des Objekts so lange hinaus, bis es tatsächlich benötigt wird; vor und während der Erzeugung fungiert der virtuelle Proxy außerdem als Ersatz für das Objekt. Danach delegiert der Proxy Anfragen direkt an das EchteSubjekt.



Muster: Proxy

Eine abstrakte Klasse oder ein Interface definiert die Schnittstelle, mit der die Clients arbeiten.

Die Clients merken gar nicht, dass sie mit einem Stellvertreter und nicht mit dem echten Objekt arbeiten.



Muster: Proxy

```
public interface Grafik {
    int getWidth();
    int getHeight();
    void draw();
}
```

```
public class EchteGrafik
implements Grafik {
    ...
}
```

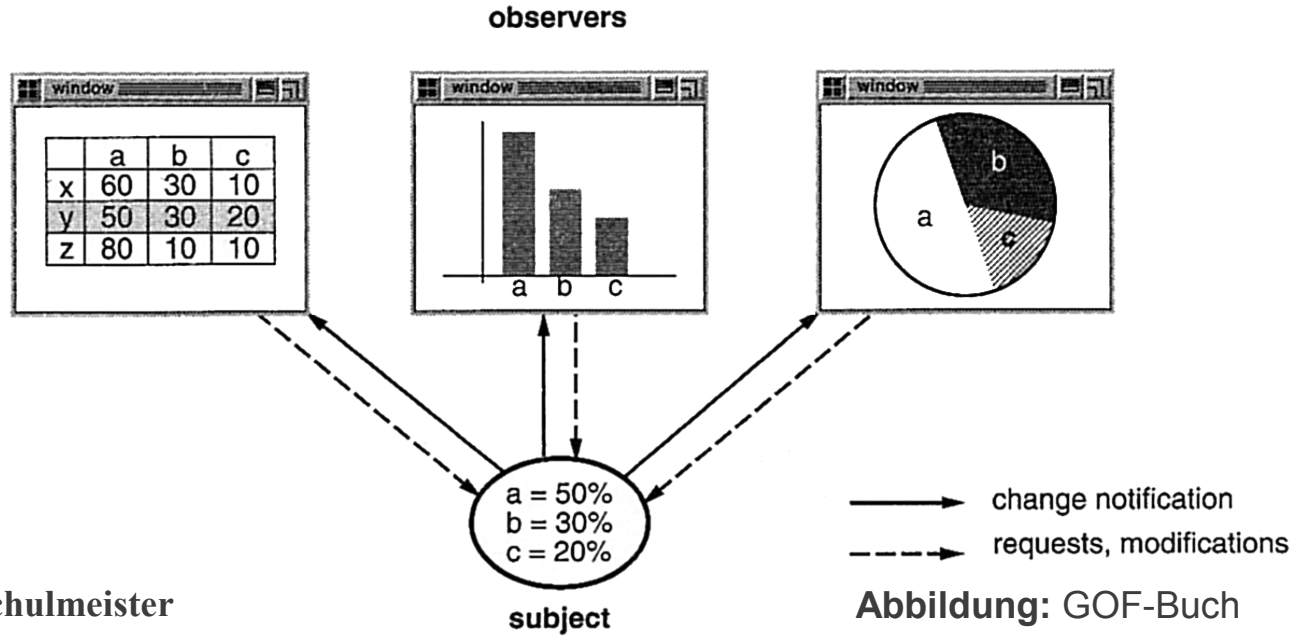
```
public class GrafikProxy implements Grafik {
    Grafik echteGrafik;

    public int getWidth() {
        if (this.echteGrafik == null) { return 100; }
        else { return this.echteGrafik.getWidth(); }
    }

    public void draw() {
        if (this.echteGrafik == null) {
            this.echteGrafik = new EchteGrafik();
        } ...
    }
}
```

Muster: Beobachter/Observer

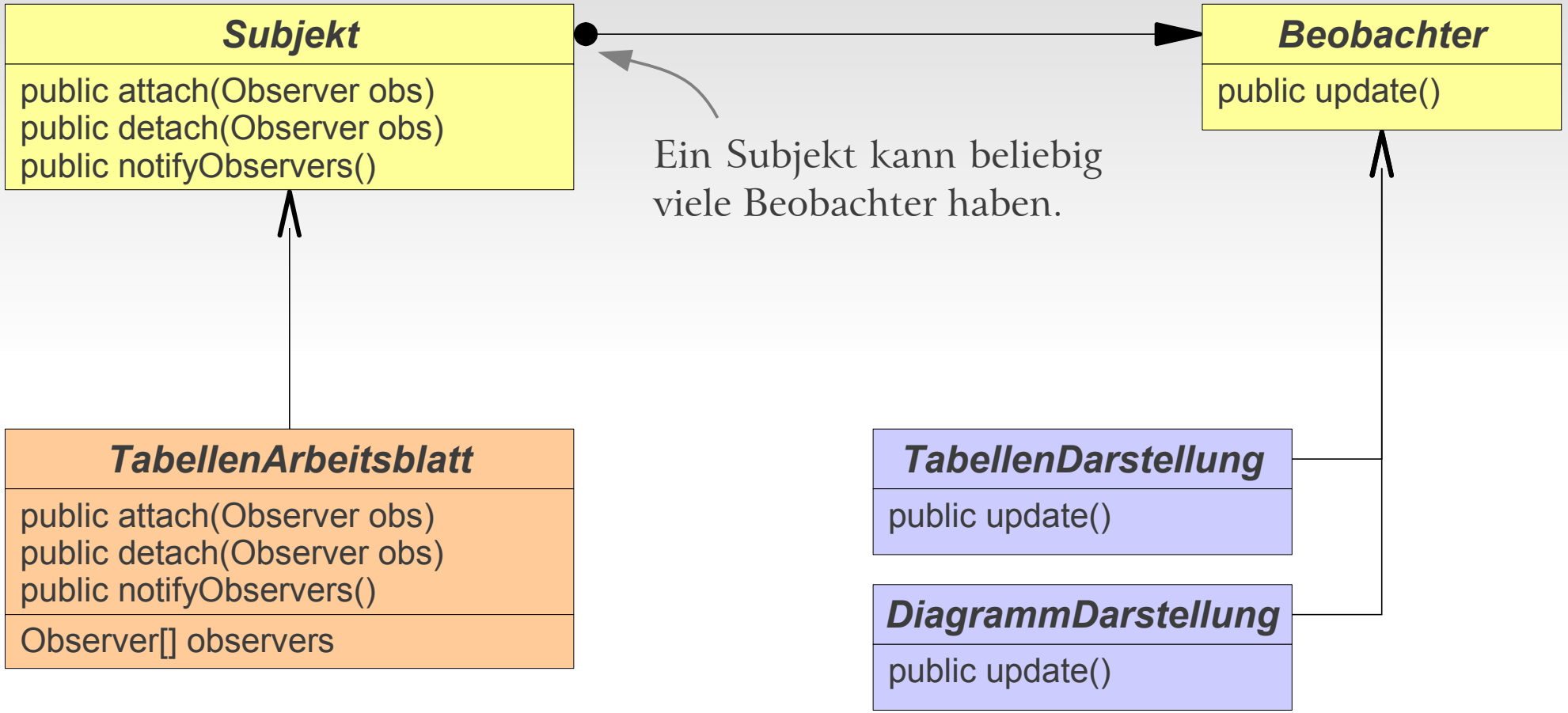
Zweck: Benachrichtigung, wenn sich ein Wert ändert

- Definition einer 1:n-Beziehung zwischen mehreren Objekten
- Mehrere abhängige Objekte beobachten ein anderes Objekt
- Ändert sich das Objekt, werden alle Beobachter benachrichtigt
- Beobachter können zur Laufzeit hinzugefügt oder entfernt werden

Motivation: Tabellenkalkulation



Muster: Beobachter/Observer



Dank der abstrakten Basisklassen oder Interfaces kennen sich die Klassen für das Arbeitsblatt und die Darstellungen nicht!

Muster: Beobachter/Observer

```
public interface Subject {
    void attach(Observer obs);
    void detach(Observer obs);
    void notifyObservers();
}
```

```
public interface Observer {
    void update();
}
```

```
public class Worksheet implements Subject {
    private ArrayList<Observer> observers =
        new ArrayList<Observer>();

    public void attach(Observer obs) { this.observers.add(obs); }
    public void detach(Observer obs) { this.observers.remove(obs); }

    public void notifyObservers() {
        for (Observer observer : this.observers) {
            observer.update();
        }
    }

    // Sonstige Methoden eines Arbeitsblatts
}
```

Model-View-Controller

Zweck

- Saubere Gliederung einer graphischen Anwendung
- Keine Abhängigkeiten vom Backend zur Benutzungsoberfläche

Über das MVC-Muster

- Sehr viel älter als das Musterbuch der Viererbande
- Beinhaltet aber viele Muster davon (je nach Variation)

Model-View-Controller

Funktioniert wie ein Doppelkeks:

- Viewobjekte zeichnen die Programmoberfläche
- Modelobjekte erbringen die eigentliche Funktionalität
- Der Controller sitzt dazwischen und hält beide zusammen

Mögliche Variationen:

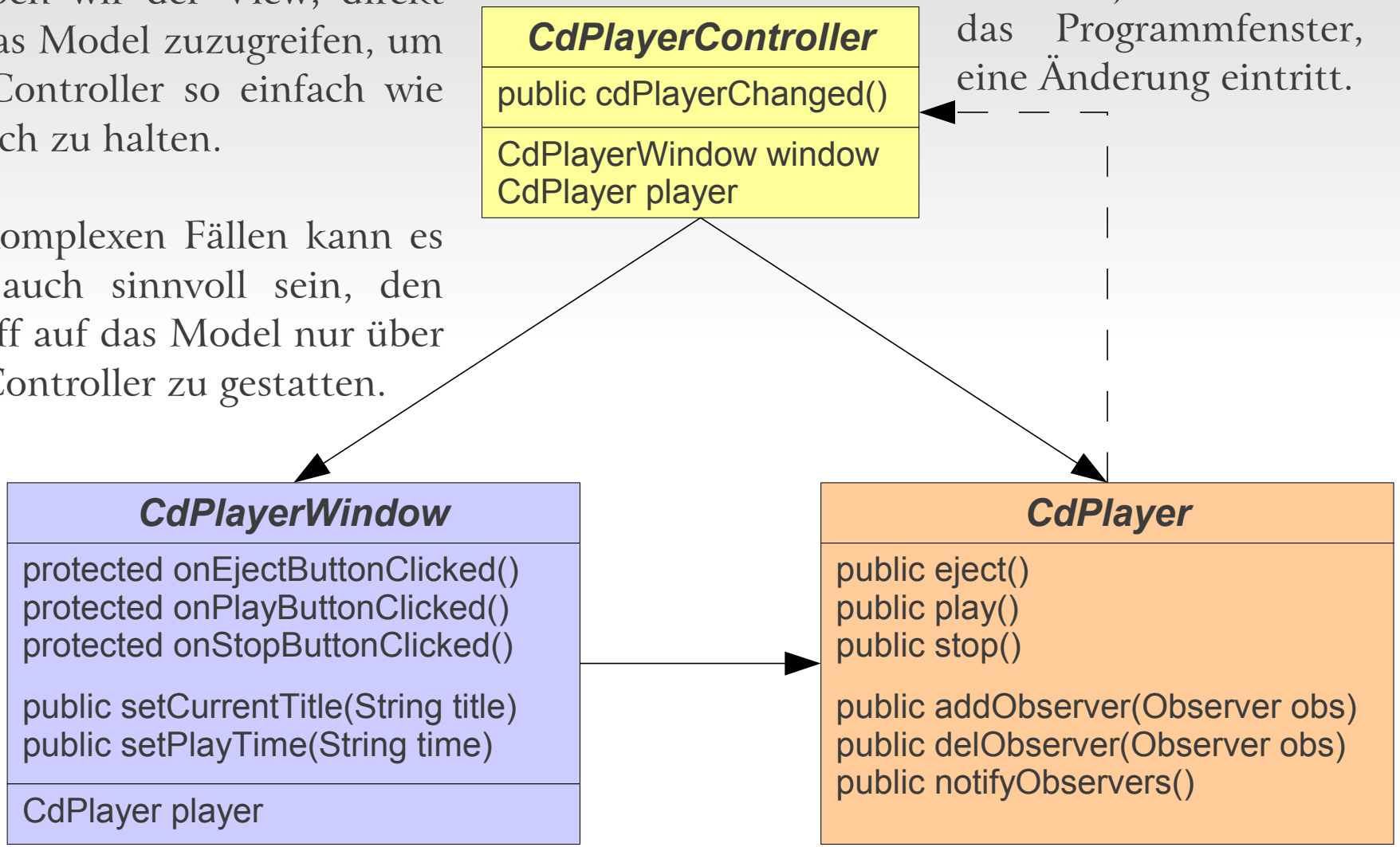
- Austauschbare Controller für dieselbe View
- Ein Controller, der mehrere Views steuert
- Zusammenarbeitende oder geschachtelte Controller

Model-View-Controller

In diesem einfachen Beispiel erlauben wir der View, direkt auf das Model zuzugreifen, um den Controller so einfach wie möglich zu halten.

Ein komplexen Fällen kann es aber auch sinnvoll sein, den Zugriff auf das Model nur über den Controller zu gestatten.

Der Controller beobachtet das Modelobjekt und aktualisiert das Programmfenster, wenn eine Änderung eintritt.



Enthaltene Entwurfsmuster

Observer / Beobachter

Controller und View können Beobachter des Models sein.

Mediator

Wenn die View nicht direkt auf das Model zugreifen darf, ist der Controller ein Mediator, da er sämtliche Anfragen zwischen Model und View vermittelt.

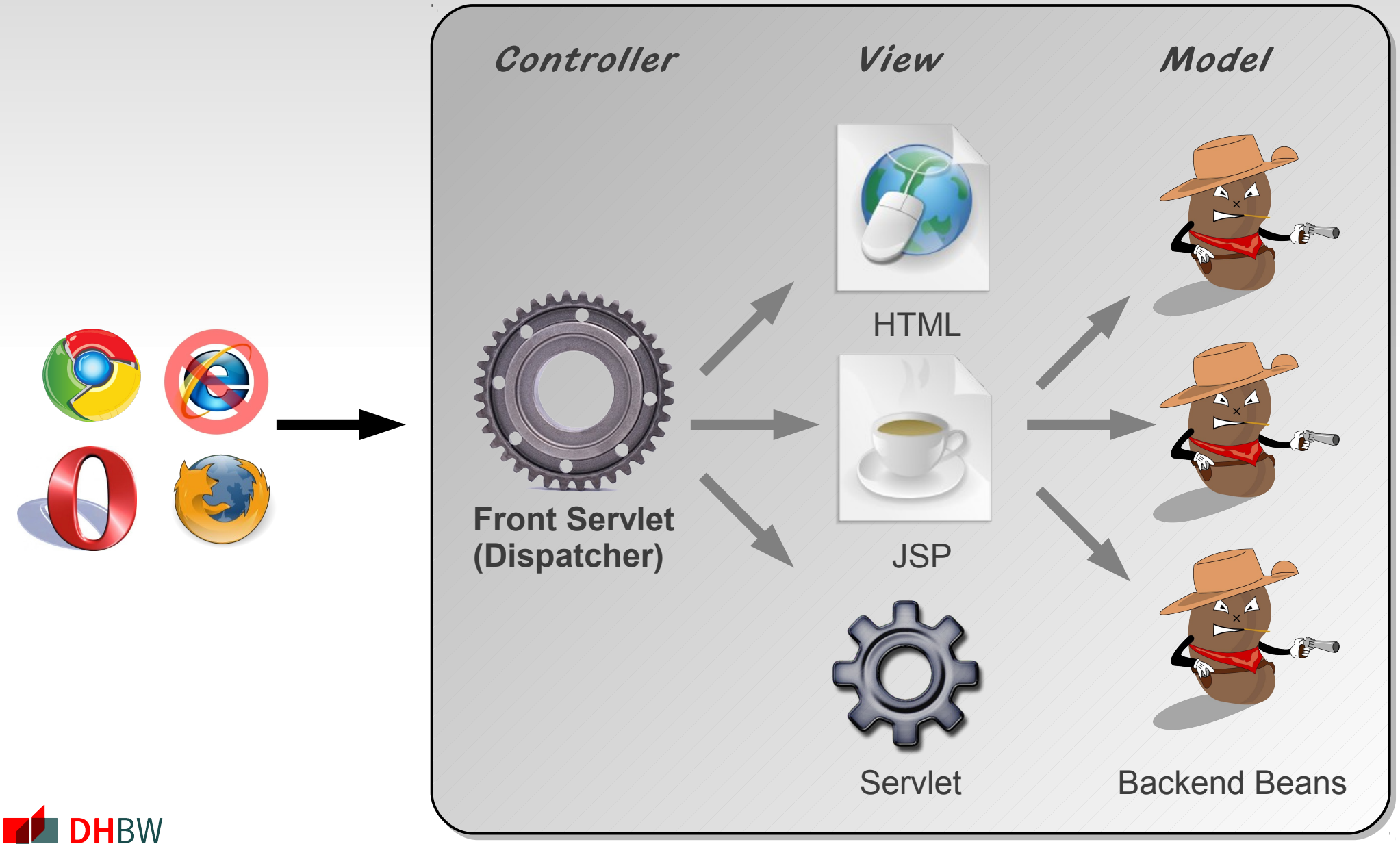
Kompositum

In vielen Frameworks werden Controller in Teil/Ganzes-Beziehungen geschachtelt, wenn ein Fenster Subscreens, Tabreiter usw. enthält.

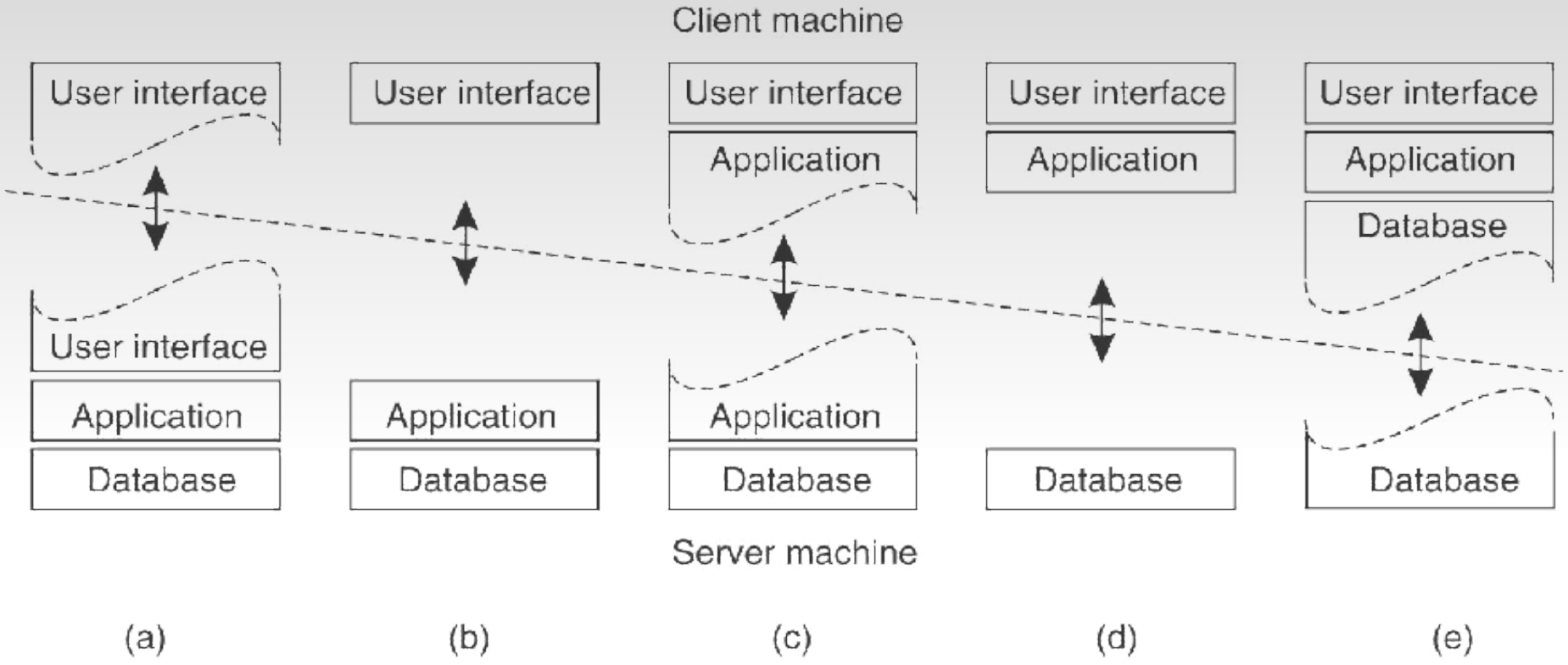
Zuständigkeitskette

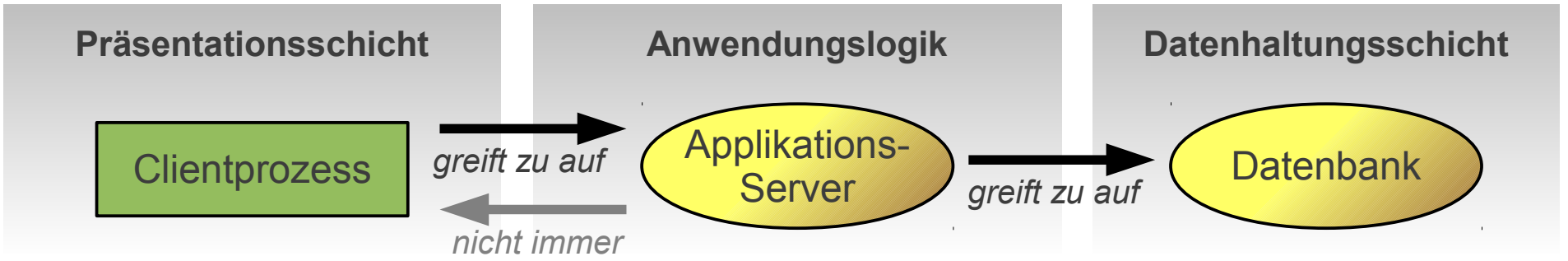
Wenn mehrere Controller zusammenarbeiten, wird eine Anfrage vom Benutzer an alle Controller weitergeleitet, bis sie von einem bearbeitet wird.

Webanwendungen und MVC



Verteilte Systeme und MVC





Interessante Links und Bücher

http://c2.com/doc/oopsla87.html
Einer der ersten Artikel zu Entwurfsmustern von Beck & Cunningham

http://wwwswt.informatik.uni-rostock.de/deutsch/Infothek/Entwurfsmuster/patterns/index.html
Deutsche Beschreibung der meisten GoF-Entwurfsmuster mit Beispieldiagrammen

http://www.patterndepot.com/put/8/JavaPatterns.htm
Online verfügbares Buch über die GoF-Entwurfsmuster (Englisch)
Finden Sie auch auf Moodle

http://www.cs.helsinki.fi/u/salaakso/patterns/
User Interface Design Patterns

Adam Bien, J2EE Patterns (2002), Addison-Wesley, ISBN: 3-8273-1903-X
Nicht mehr auf der Höhe der Zeit, hilft aber das aktuelle Java EE zu verstehen

Freeman & Freeman, Entwurfsmuster von Kopf bis Fuß, O'Reilly, ISBN: 3-89721-421-0 Must Read für alle Javaentwickler!