DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Zielgruppe: **Wirtschaftsinformatik**

Modul: **Software Engineering**

Foliensatz: # Architektur- und Entwurfsmuster

Prof. Dr. Malte Prieß

Agenda (Lernziele)

I. Architekturmuster
   - Architekturmuster und Software-Schichten
   - Drei-Schichten-Architekturmuster
   - Model-View-Controller (MVC)
   - Plug-in-Architekturmuster

II. Entwurfsmuster

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Architekturmuster und Software-Schichten

Prof. Dr. Malte Prieß

- Software-Architektur ist zentral in den Phasen Analyse, Entwurf und Implementierung **komplexer Systeme**
- **Qualitätseigenschaften** von Softwaresystemen werden maßgeblich durch die gewählte Architektur bestimmt
- Für die **kontinuierliche Weiterentwicklung** ist die **Qualität** der Architektur selbst entscheidend
- Insb. zur **Integration und Migration** existierender Komponenten müssen adäquate Architekturen entworfen werden

Architektur- und Entwurfsmuster vermitteln Wissen über **bewährte Lösungen**.

Ziele

- Beherrschung von **Komplexität**
- **Unabhängige Konstruktion** von Software-Komponenten
- **Flexibilität** in Erweiterung, Änderung und Wiederverwendung
- **Architektur als Basis** für Reengineering, Refactoring und Migration

Wenn sich bestimmte **Lösungsstrukturen** bewährt haben, kann man sie dokumentieren und später erneut verwenden.

> *An **architectural pattern** expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.*
>
> Buschmann et al. (1996)

- Wenn wir ein **Architekturmuster** verwenden, legen wir damit die **Strukturen auf der obersten Ebene der Software-Architektur** fest
- Die Wahl eines Architekturmusters ist eine zentrale Entscheidung
- Ob das vorgesehene Architekturmuster geeignet ist, hängt davon ab, ob seine speziellen Randbedingungen und Voraussetzungen erfüllt sind

Quelle: Ludewig, Lichter, (2010), S.429

E.W. Dijkstra zerlegt in „The Structure of the THE Multiprogramming System“ (1968) ein Betriebssystem in fünf Schichten.

> *The **Layers architectural pattern** helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.*
>
> Buschmann et al. (1996)

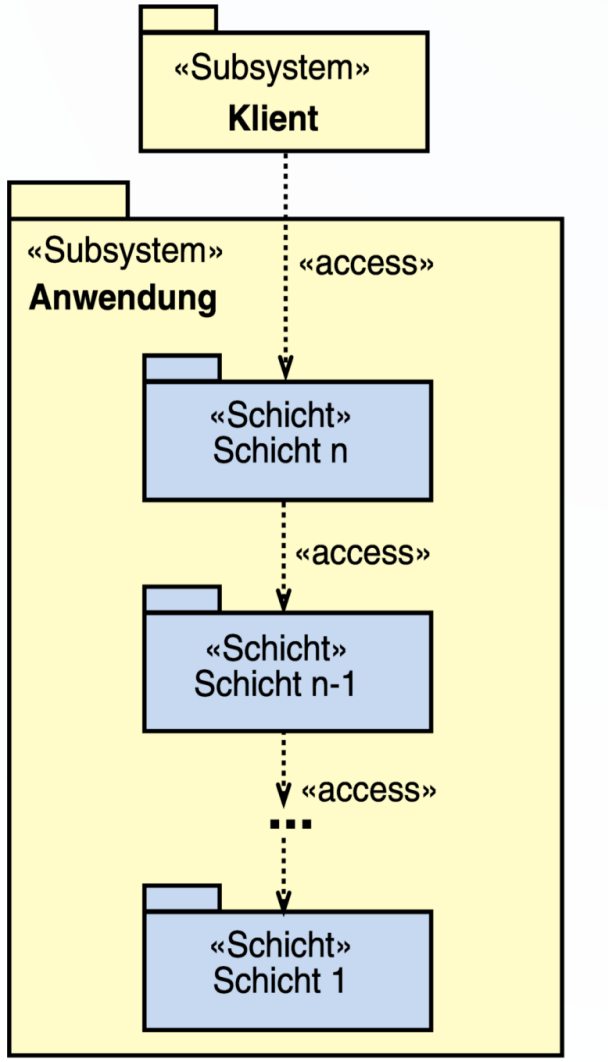
Schichten werden hiernach nach den folgenden Regeln festgelegt:

- Eine Schicht fasst logisch **zusammengehörende Komponenten** zusammen
- Eine Schicht stellt **Dienstleistungen** zur Verfügung, die an der (oberen) Schnittstelle der Schicht angeboten werden
- Die Dienstleistungen einer Schicht können nur von Komponenten der **direkt darüber liegenden Schicht** benutzt werden

Quelle: Ludewig, Lichter, (2010), S.430, 431

Eigenschaften von Schichten

- Schichten bauen (direkt) aufeinander auf, ein **Durchgriff** durch mehrere Ebenen **ist nicht erlaubt**
- Ein schichtenbasierter Entwurf hat folgende **Vorteile**:
  - Schichten sind nur gekoppelt, wenn sie benachbart sind
  - Aber auch diese Kopplung ist durch die Kapselung der Operationen gering
  - Änderungen wirken sich darum meist nur lokal (innerhalb einer Schicht) aus
  - Eine Schicht kann aus mehreren entkoppelten Teilen mit hohem Zusammenhalt bestehen



Quelle: Ludewig, Lichter, (2010), S.430

Protokollbasiert versus Objektorientiert

- **Protokollbasierte Schichten**

  Schicht, die nur mit ihren Nachbarschichten kommuniziert (Züllighoven (2005)).

- **Objektorientierte Schicht**

  Schicht, die für alle höheren Schichten sichtbar ist, damit dort die in der tieferen Schicht enthaltenen Komponenten (Klassen, Klassenbibliotheken, etc.) benutzt oder erweitert werden können.

- In großen objektorientierten Anwendungen können beide Schichtenarten vorkommen; dabei gilt:
  - Die Komponenten einer Schicht dürfen immer nur Komponenten der **nächsttieferen protokollbasierten** Schicht benutzen
  - Die Komponenten einer Schicht können die Komponenten **aller tieferen objektorientierten** Schichten nutzen

Quelle: Ludewig, Lichter, (2010), S.431

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Architekturmuster

Drei-Schichten-Architekturmuster

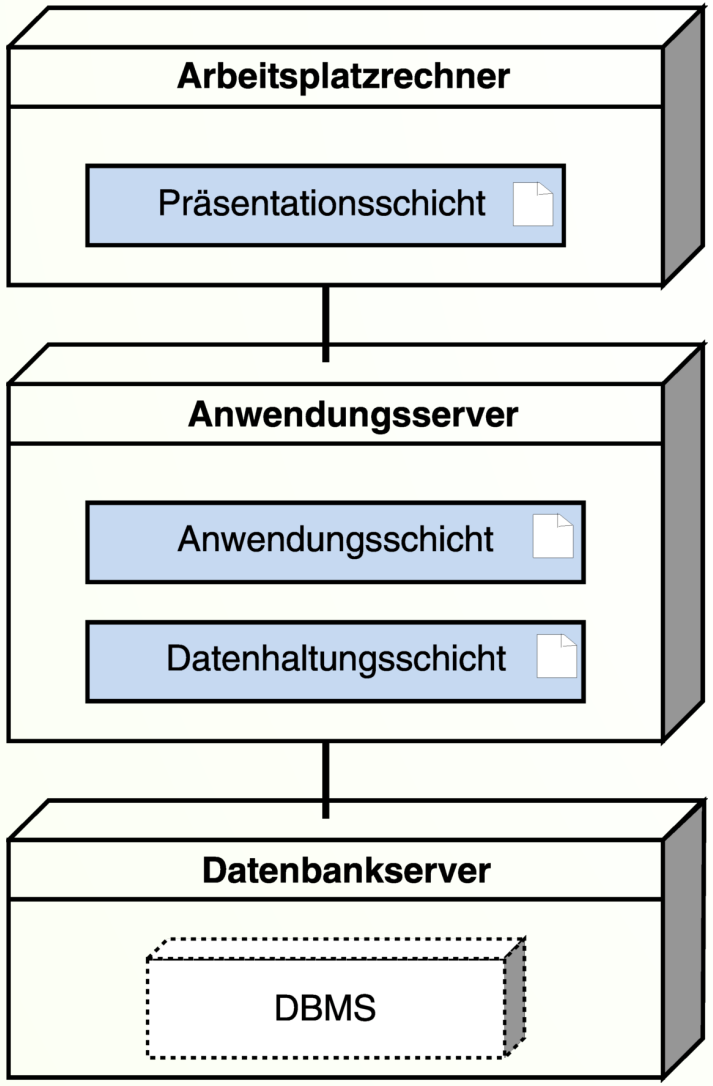
Prof. Dr. Malte Prieß

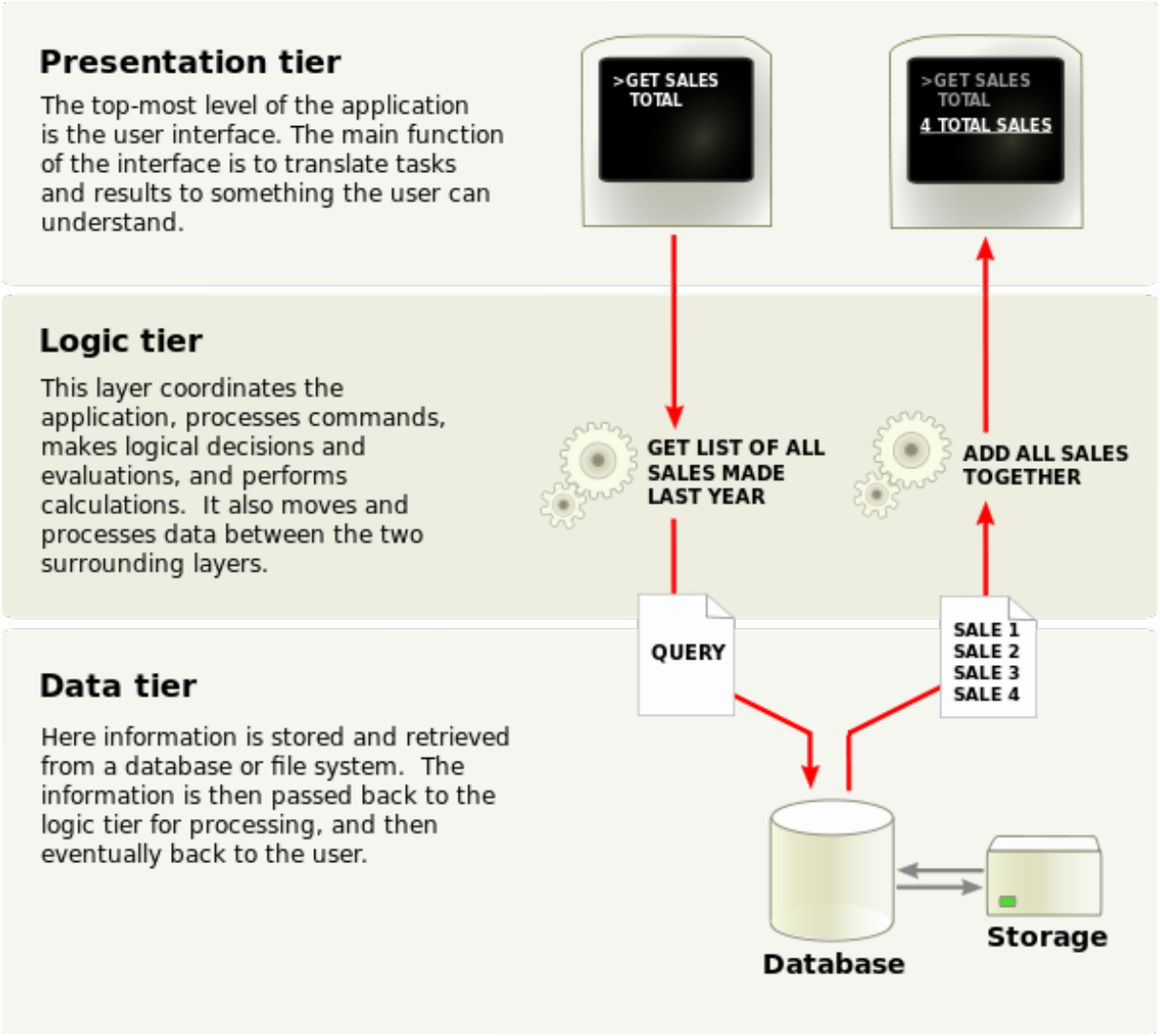
Das Drei-Schichten-Architekturmuster

- Viele interaktive Software-Systeme sind aus den folgenden drei **protokollbasierten Schichten** aufgebaut:
  - Präsentationsschicht
  - Anwendungsschicht
  - Datenhaltungsschicht
- Die **Präsentationsschicht** realisiert die Bedienoberfläche, sie stellt Informationen dar und steuert die Interaktion Benutzer – System. Dazu kommuniziert sie mit der Anwendungsschicht.
- In der **Anwendungsschicht** liegen alle Komponenten, die die fachliche Funktionalität realisieren. Sie sind so konstruiert, dass sie keine Information über die Präsentation enthalten.
- Darunter liegt die **Datenhaltungsschicht**. Sie sorgt für die dauerhafte Speicherung, meist in einer Datenbank. Details der Speicherung sind in dieser Schicht verborgen!

Quelle: Ludewig, Lichter, (2010), S.431, 432

Beispiel





Quelle: Ludewig, Lichter, (2010), S.432; http://en.wikipedia.org/wiki/File:Overview_of_a_three-tier_application_vectorVersion.svg

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Architekturmuster

Model-View-Controller (MVC)

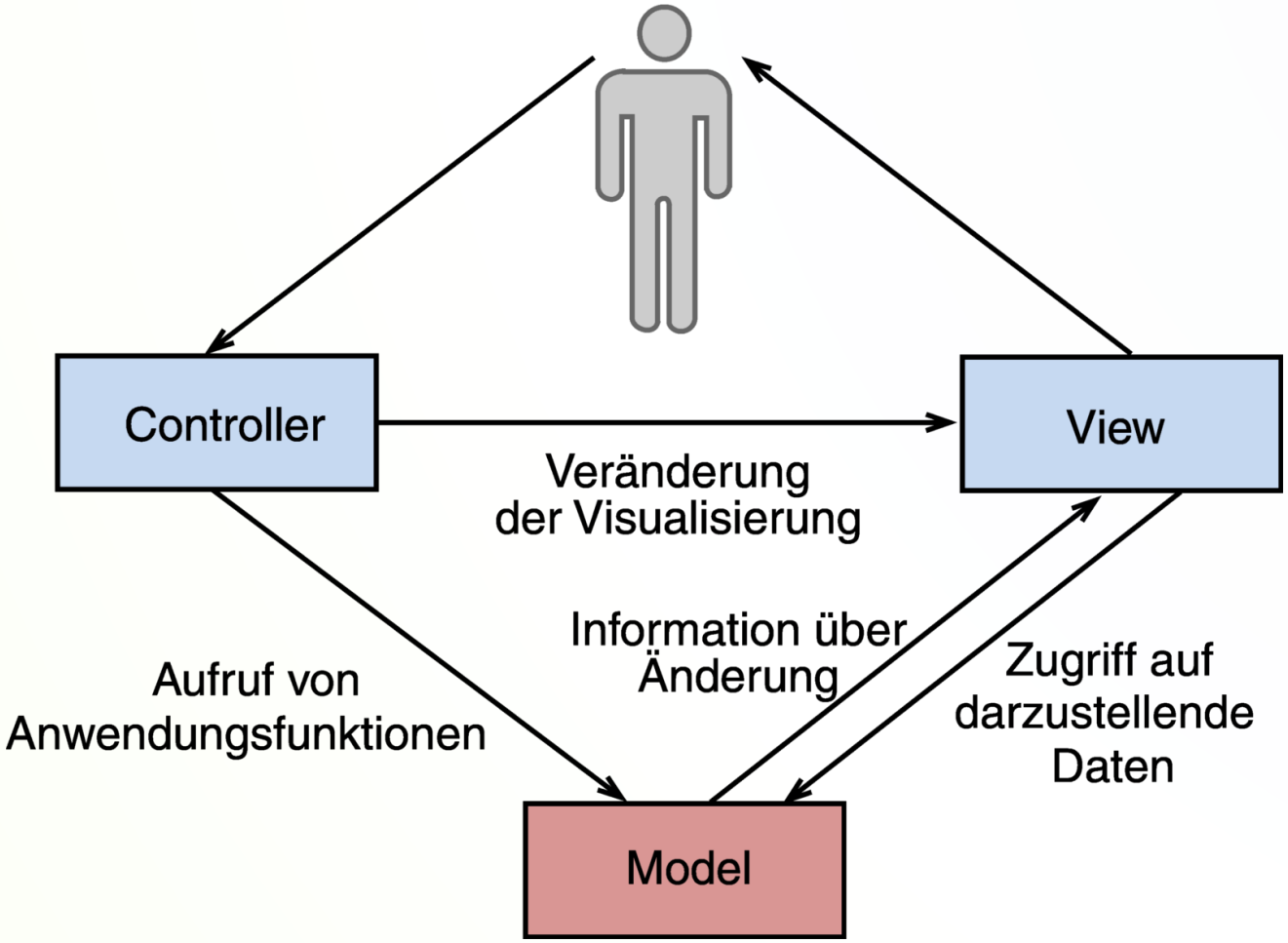
Prof. Dr. Malte Prieß

MVC gliedert eine interaktive Anwendung in drei Komponenten:

- Das **Model** realisiert die fachliche Funktionalität der Anwendung. Es kapselt die Daten der Anwendung, stellt jedoch Zugriffsoperationen zur Verfügung.
- Eine **View** (Ansicht) präsentiert dem Benutzer die Daten. Sie verwendet die Zugriffsoperationen des Models. Zu einem Model kann es für unterschiedliche Darstellungen beliebig viele Views geben.
- Jedem View ist ein **Controller** zugeordnet. Dieser empfängt die Eingaben des Benutzers und reagiert darauf. Wählt der Benutzer – beispielsweise über einen Menüeintrag – eine Funktion aus, dann ruft der Controller diese beim Model auf.

Quelle: Ludewig, Lichter, (2010), S.433, 434

Interaktion der MVC-Komponenten



Quelle: Ludewig, Lichter, (2010), S.434ff

Model-View-Controller (MVC)

- Wenn die Daten der Anwendung durch die Interaktion des Benutzers verändert werden, müssen die betroffenen Views darauf reagieren.
- Erfordert Mechanismus, mit dem Veränderungen des Models den Views mitgeteilt werden
- Das Model verwaltet dazu eine „Kundenliste“ (*Register*), in der sich alle Views, die das Model darstellen, eintragen müssen
- Ändert sich der Zustand des Models, so werden alle registrierten Views informiert; sie können dann reagieren und ihre Darstellung aktualisieren.

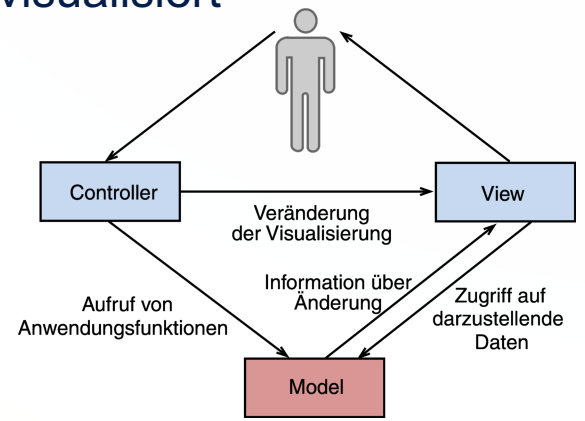
Dieser Ablauf wird auch als **Change-Update Mechanismus** bezeichnet (vgl. **Entwurfsmuster Observer**).

Quelle: Ludewig, Lichter, (2010), S.434ff

Model-View-Controller (MVC)

Vorteile?

- Berücksichtigung des Prinzips **Separation of Concerns**
- **Lose Kopplung** zwischen den Komponenten, **hohe Kohäsion** innerhalb
- **Wiederverwendung** von Komponenten/Code
- Zu einem Modell mehrere View-Controller-Paare möglich
- Saubere **Entkopplung vom Model**: View-Controller-Paare können zur Laufzeit hinzugefügt/weggenommen werden
- **Change-Update Mechanismus**: Model wird in allen Views immer aktuell visualisiert

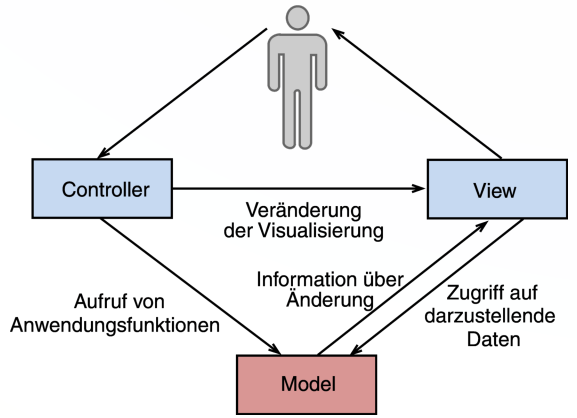


Quelle: Ludewig, Lichter, (2010), S.434ff

Model-View-Controller (MVC)

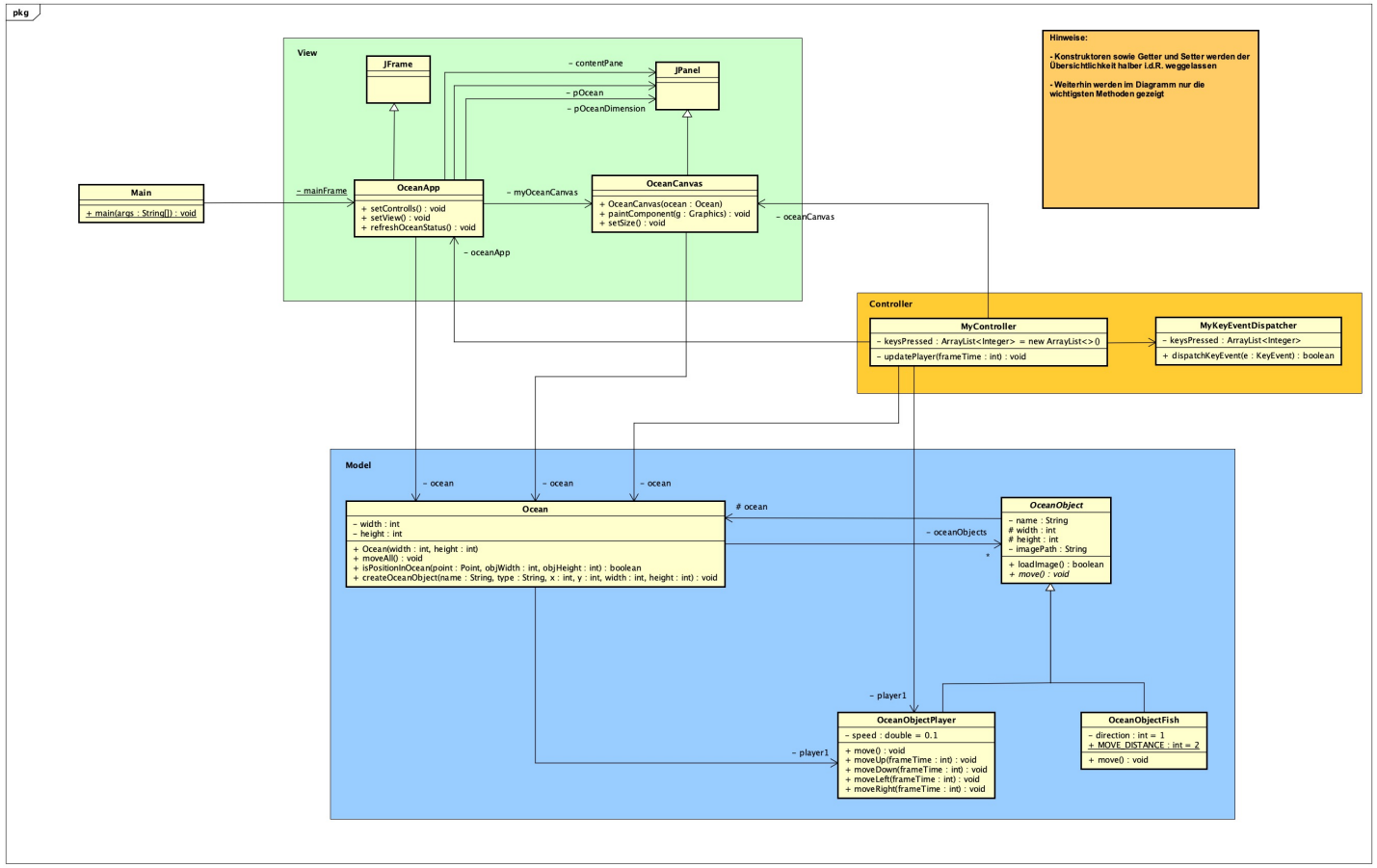
Nachteile?

» Wenn sich die Daten sehr oft und sehr schnell ändern, kann das dazu führen, dass die Views diese Veränderungen nicht mehr schnell genug anzeigen können (bei jeder Änderung müssen die Daten vom Model erfragt werden).



Quelle: Ludewig, Lichter, (2010), S.434ff

Im Rahmen der Spieleentwicklung haben wir die Trennung nach Model, View und Controller umgesetzt (kein Change-Update-Mechanismus implementiert, daher noch kein vollständiges MVC-Pattern):



Vgl. ProjectExampleOceanApp_MVC.asta

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Architekturmuster

Plug-in-Architekturmuster

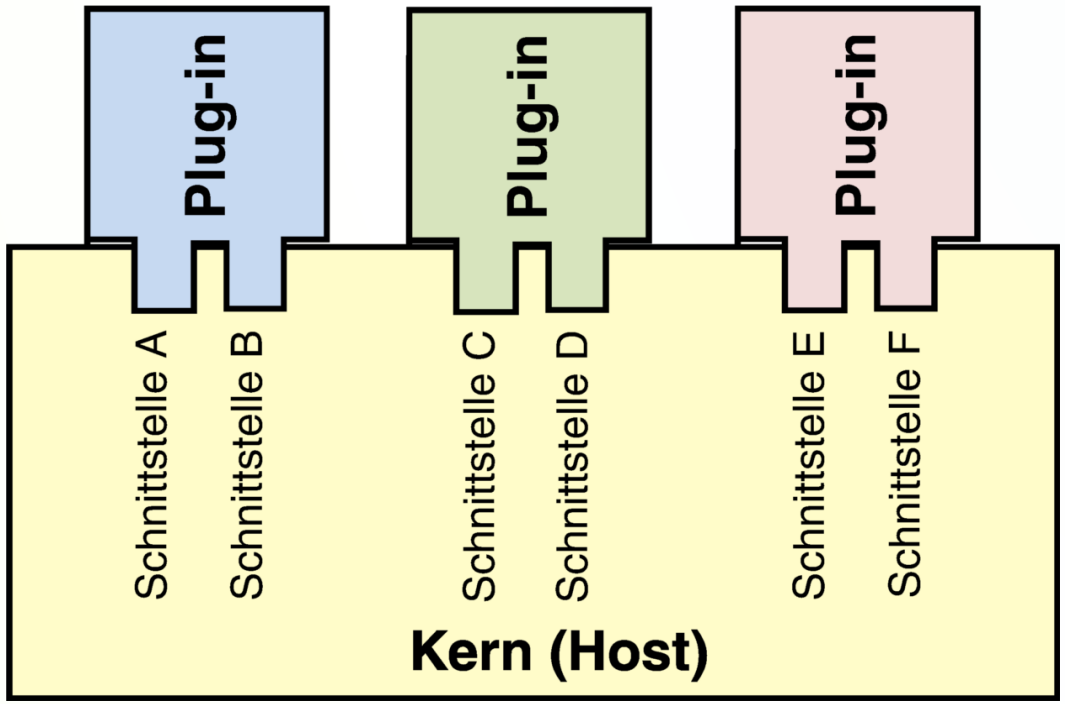
Prof. Dr. Malte Prieß

Das Plug-in-Architekturmuster

- Es bietet die Möglichkeit, ein System an dafür **vorgesehenen Punkten** zu erweitern, ohne es zu modifizieren (**Offen-geschlossen-Prinzip**)
- Eine Plug-in-Anwendung besteht aus einem **Kern** (auch **Host** = Wirt genannt), der durch sogenannte **Plug-ins** um neue Funktionen erweitert werden kann
  - Der Host definiert spezielle Schnittstellen, die sogenannten **Erweiterungspunkte**, auf die ein Plug-in Bezug nehmen kann
  - Damit ein Plug-in im Kontext des Hosts ausgeführt werden kann, muss es gemäß den vom Host **vorgegebenen technischen Konventionen** realisiert sein
  - Ein Plug-in kann selbst ein Host sein, d.h., **Plug-ins können Erweiterungspunkte für weitere Plug-ins definieren**

Quelle: Ludewig, Lichter, (2010), S.436

Schema einer Plug-in-Architektur

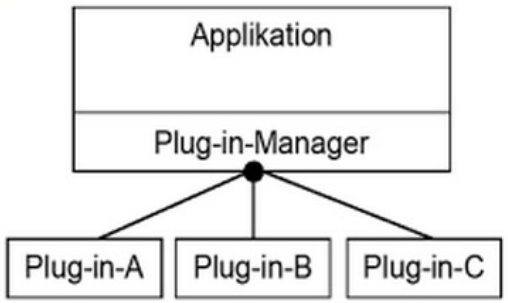


Quelle: Ludewig, Lichter, (2010), S.436

Das Plug-in-Architekturmuster

- Einfache Erweiterung der Anwendung durch Dritte möglich ohne Zugriff auf Quelltext der Anwendung
- Host ist ohne Plug-ins lauffähig, Plug-ins nicht ohne Host

Beispiele:



DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Software Engineering

Entwurfsmuster

Prof. Dr. Malte Prieß

Entwurfsmuster nach Gamma et al.

- Architekturmuster = Muster auf Ebene der Software-Architektur
- Entwurfsmuster = Muster auf Ebene der Komponenten
  - bieten Lösungen **für gängige Entwurfsprobleme** auf Ebene des Feinentwurfs von Komponenten an
  - werden **unabhängig** von einer bestimmten Sprache formuliert, greifen aber meist auf objektorientierte Konzepte zurück

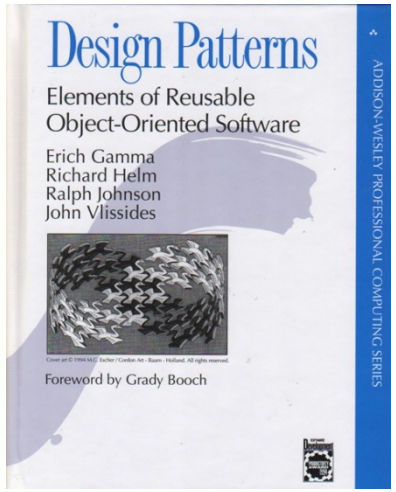
> **Design patterns** *... are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design*.
>
> Gamma et al. (1995)

- Um ein Entwurfsmuster zu verwenden, muss man wissen, **welches Problem es löst** und wie man das Entwurfsmuster im Kontext einer konkreten Architektur **ausprägen** kann

Quelle: Ludewig, Lichter, (2010), S.436

Entwurfsmuster nach Gamma et al.

- **Lösungsansätze für wiederkehrende Anforderungen** im objekt-orientierten Entwurf
- Erprobte Abbildung einer typischen Struktur oder Verhaltensweise
- Ausgerichtet auf **Anpassungsfähigkeit** und **Erweiterbarkeit** für zukünftige Anforderungen
- Erste Katalogisierung durch “Gang of Four”: Gamma et al. (1995)



DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

Kategorisierung der Entwurfsmuster

Prof. Dr. Malte Prieß

Kategorisierung der Entwurfsmuster nach Gamma et al. (1995)

1. Erzeugungsmuster → Erzeugung von Objekten
2. Strukturmuster → Modellierung von Datenstrukturen
3. Verhaltensmuster → Interaktion zwischen Objekten zur Laufzeit

Kategorisierung der Entwurfsmuster nach Gamma et al. (1995)

1. Erzeugungsmuster
   - Fabrikmethode *(Factory Method, Virtual Constructor)*
   - Abstrakte Fabrik *(Abstract Factory, Kit)*
   - Singleton *(Singleton)*
   - Erbauer *(Builder)*
   - Prototyp *(Prototype)*
2. Strukturmuster
3. Verhaltensmuster

Kategorisierung der Entwurfsmuster nach Gamma et al. (1995)

1. Erzeugungsmuster
2. Strukturmuster
   - Adapter *(Adapter)*
   - Brücke *(Bridge)*
   - Dekorierer *(Decorator, Wrapper)*
   - Fassade *(Facade)*
   - Fliegengewicht *(Flyweight)*
   - Kompositum *(Composite)*
   - Stellvertreter *(Proxy, Surrogate)*
3. Verhaltensmuster

Kategorisierung der Entwurfsmuster nach Gamma et al. (1995)

1. Erzeugungsmuster
2. Strukturmuster
3. Verhaltensmuster
   - Interpreter *(Interpreter)*
   - Schablonenmethode *(Template Method)*
   - Beobachter *(Observer, Publish-Subscribe, Listener)*
   - Besucher *(Visitor)*
   - Iterator *(Iterator, Cursor)*
   - Befehl *(Command, Action, Transaction)*
   - Memento *(Memento, Token)*
   - Strategie *(Strategy, Policy)*
   - Vermittler *(Mediator)*
   - Zustand *(State)*
   - Zuständigkeitskette *(Chain of Responsibility)*

Kategorisierung der Entwurfsmuster nach Ludewig, Lichter (2010)

1. Muster zum Verwalten on Objekten
2. Muster zum Anbinden vorh. Klassen
3. Muster zur Entkopplung von Klassen
4. Muster zur Trennung von unterschiedlichen und gemeinsamen Eigenschaften
5. Muster zur Trennung von unterschiedlichen und gemeinsamen Eigenschaften

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

Strategie (Strategy, Policy)

Prof. Dr. Malte Prieß

Strategie (Strategy)	
Problem	Objekte, die sich nur dadurch unterscheiden, dass sie gleiche Aufgaben teilweise durch verschiedene Algorithmen lösen, sollen durch eine Klasse, nicht durch eine Klassenhierarchie implementiert werden.
Beispiel	Objekte, die Eingabemasken implementieren, unterscheiden sich darin, wie die Eingaben geprüft werden.

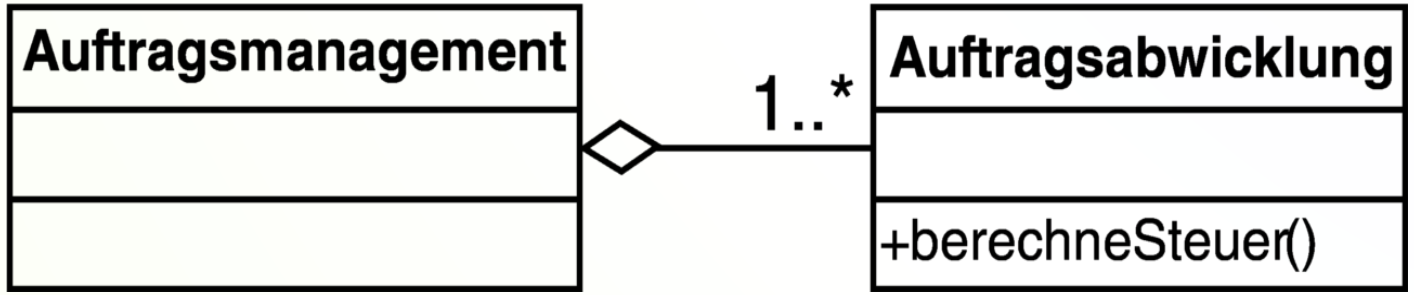
Kategorie

- Gamma et al.: **Verhaltensmuster**
- Ludewig, Lichter: Muster zur Trennung von unterschiedlichen und gemeinsamen Eigenschaften

Quelle: Ludewig, Lichter, (2010), S.441, 442

Anwendung zur elektronischen Verkaufsabwicklung

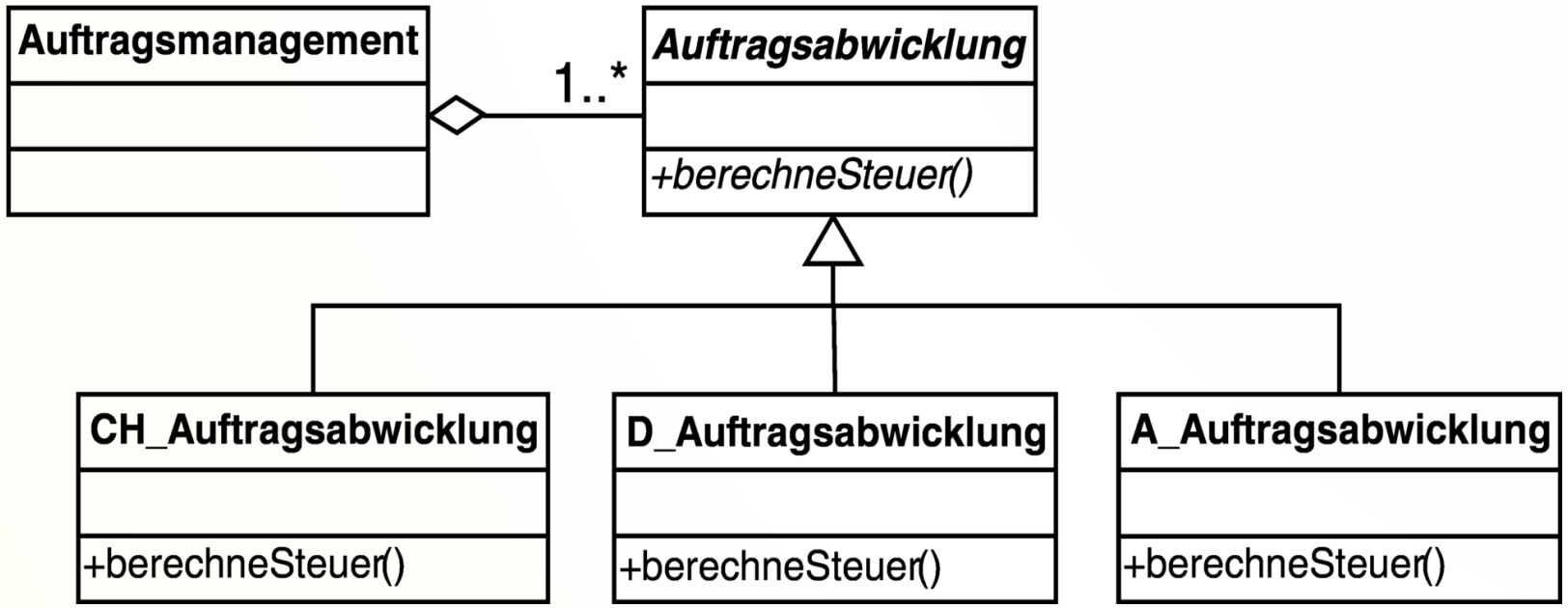
- `Auftragsmanagement` dient dazu, eingehende Aufträge anzunehmen, zu prüfen und zu verwalten
- Die Aufträge werden an Objekte der Klasse `Auftragsabwicklung` übergeben, die die Aufträge ausführen
- Um die anfallenden Steuern zu ermitteln, enthält diese Klasse die Methode `berechneSteuer()`



Quelle: Ludewig, Lichter, (2010), S.437, 438

Beispiel – Neue Anforderung

- Die Anwendung soll nun erweitert werden, damit auch Aufträge aus Österreich und der Schweiz abgewickelt werden können
- Naheliegender Entwurf:



Quelle: Ludewig, Lichter, (2010), S.437, 438

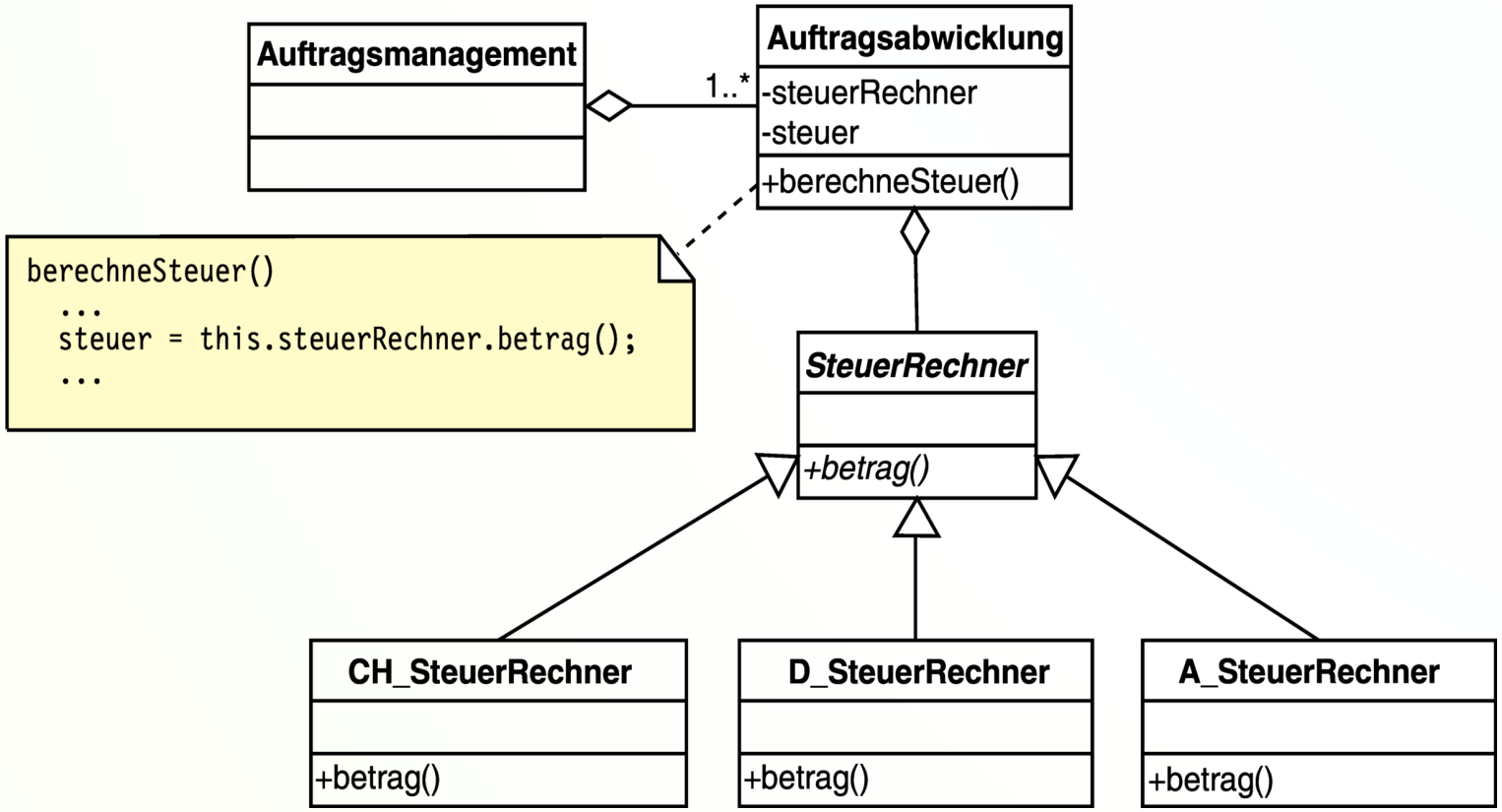
Betrifft Prinzipien: Separation of Concerns, Modularisierung

- Separation of Concerns:
  - Prinzip Separation of Concerns verletzt
  - Steuerberechnung und Auftragsabwicklung sind miteinander vermischt, schlechte Trennung
  - Variabler Aspekt – die Steuerberechnung – ist vermischt mit den statischen Funktionalitäten der Auftragsabwicklung
- Modularisierung, Kohäsion/Zusammenhalt
  - Kohäsion/Zusammenhalt im Modul Auftragsabwicklung schlecht
- **Wdh.: Eine wichtige Regeln** zur Trennung von Zuständigkeiten:
  - Ordne Funktionalitäten, die **variabel** sind oder später **erweitert** werden müssen, eigenen Komponenten zu

Quelle: Ludewig, Lichter, (2010), S.437, 438

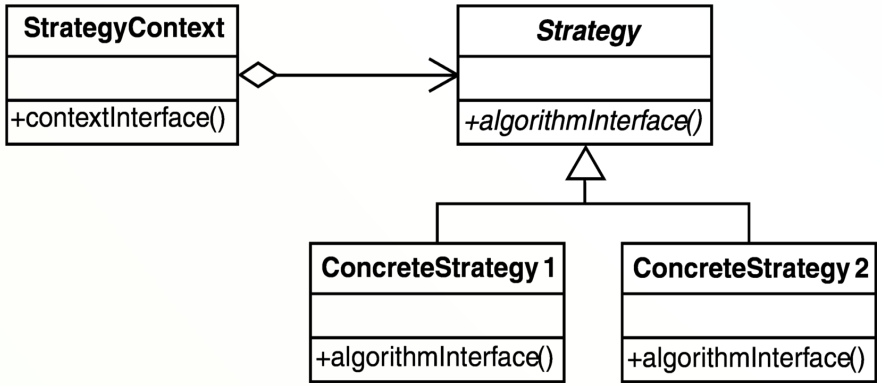
Lösung mit Entwurfsmuster Strategie (Strategy, Policy)

- Wir modellieren den **variablen Aspekt** (länderspezifische Steuerberechnung) explizit durch eine **eigene Klasse**
- Die verschiedenen Steuerberechnungsvorschriften werden in **Unterklassen** der Klasse `SteuerRechner` realisiert



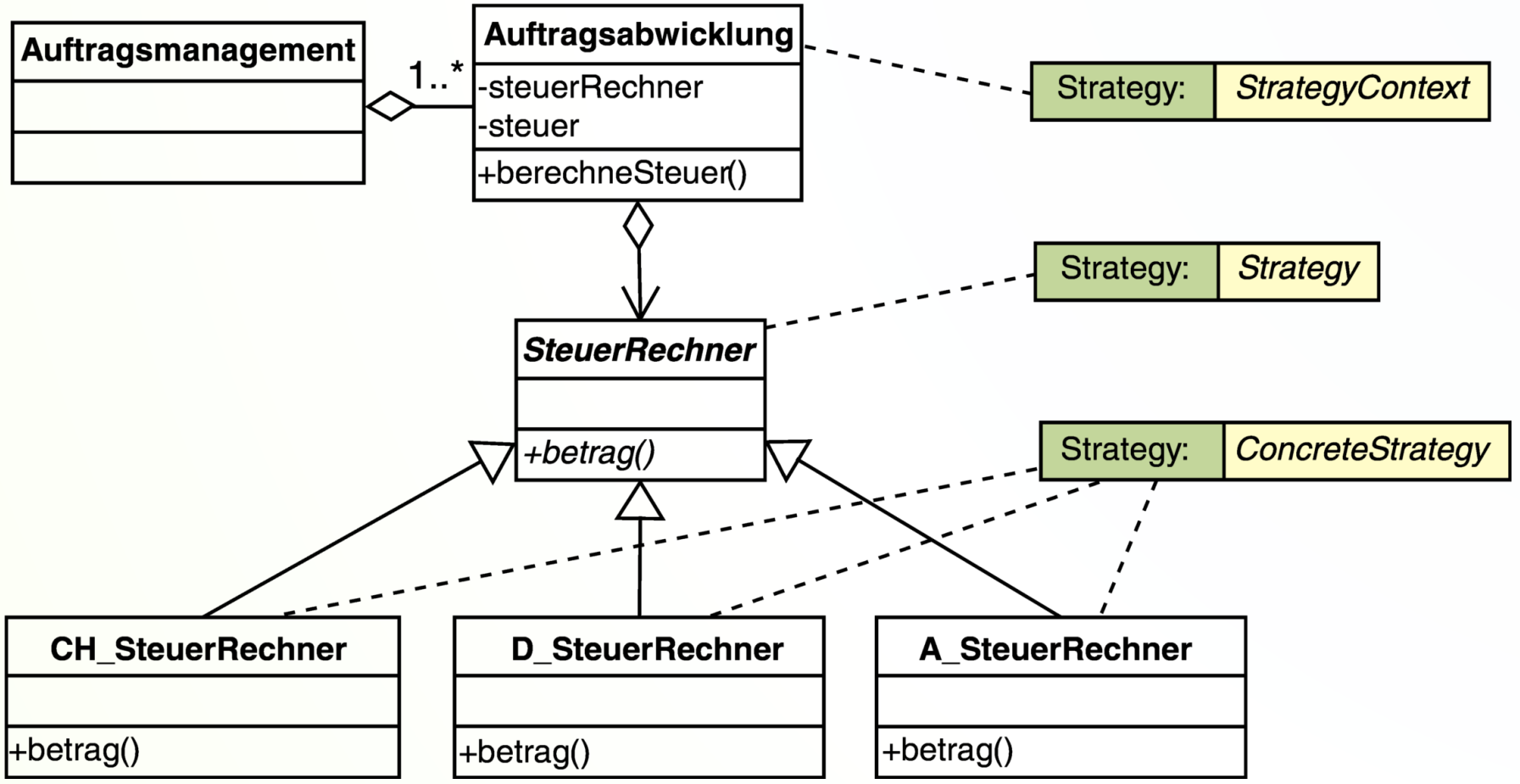
Quelle: Ludewig, Lichter, (2010), S.439

Strategie (Strategy, Policy)

Strategie (Strategy)	
Problem	Verwandte Klassen unterscheiden sich lediglich dadurch, dass sie gleiche Aufgaben teilweise durch verschiedene Algorithmen lösen.
Lösung	Die Klassen werden nicht – wie üblich – in einer Vererbungshierarchie angeordnet. Stattdessen wird eine Klasse erstellt, die alle gemeinsamen Operationen definiert (*StrategyContext*). Die Signaturen der Operationen, die unterschiedlich zu implementieren sind, werden in einer weiteren Klasse zusammengefasst (*Strategy*). Die Rolle Strategy legt somit fest, über welche Schnittstelle die verschiedenen Algorithmen genutzt werden. Von dieser Klasse wird für jede Implementierungsalternative eine konkrete Unterklasse abgeleitet (*Concrete-Strategy*). Die Klasse mit der Rolle StrategyContext benutzt konkrete Strategy-Objekte, um die unterschiedlich implementierten Operationen per Delegation auszuführen.
Struktur	

Quelle: Ludewig, Lichter, (2010), S.439

Beispiel – Zuordnung der Musterrollen und Klassen



Quelle: Ludewig, Lichter, (2010), S.440; Java-Implementierung: /DesignPattern/src/com/priess/strategy1/

Vorteile:

- Alternative zur Unterlassenbildung: Neue Auftragsabwicklungen können kreiert werden durch die Erstellung neuer Concrete Strategies oder durch die neue Kombination bestehender Strategy-Objekte
- **Besserer Zusammenhalt**, da Auftragsabwicklung und Steuerberechnung nicht mehr in einer Klasse gemischt werden
- Steuerberechnungen werden nach dem Prinzip der **Trennung von Zuständigkeiten** separat modelliert

  `Auftragsabwicklung` kennt nur noch die Schnittstelle `SteuerRechner` und nicht die konkreten `SteuerRechner`, ist also vom konkreten Verhalten entkoppelt.

  Diese ist damit implementierungsunabhängig und kann somit mit neuen Verhalten ausgestattet werden, ohne dass der Code dafür geändert werden muss.
- Der Client kann das **Verhalten** eines Contextobjekts sowohl zur **Designzeit** als auch zur **Laufzeit dynamisch ändern**
- Weiterhin können durch die Auslagerung des Verhaltens auch andere (verwandte) Contextklassen die Strategien **wiederverwenden** und müssen sie nicht selbst implementieren

Quelle: Ludewig, Lichter, (2010), S.438

*** Übung ***

1. Wie sehe für das Beispiel der Steuerberechnung eine entsprechende JAVA Implementierung mit einem Beispiel-Client aus?

2. Hunde-Beispiel auf der Seite von Philipp Hauer nachvollziehen

Lösungen:

- `/DesignPattern/src/com/priess/strategy1/`
- `/DesignPattern/src/com/priess/strategy2/`

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

Adapter

Prof. Dr. Malte Prieß

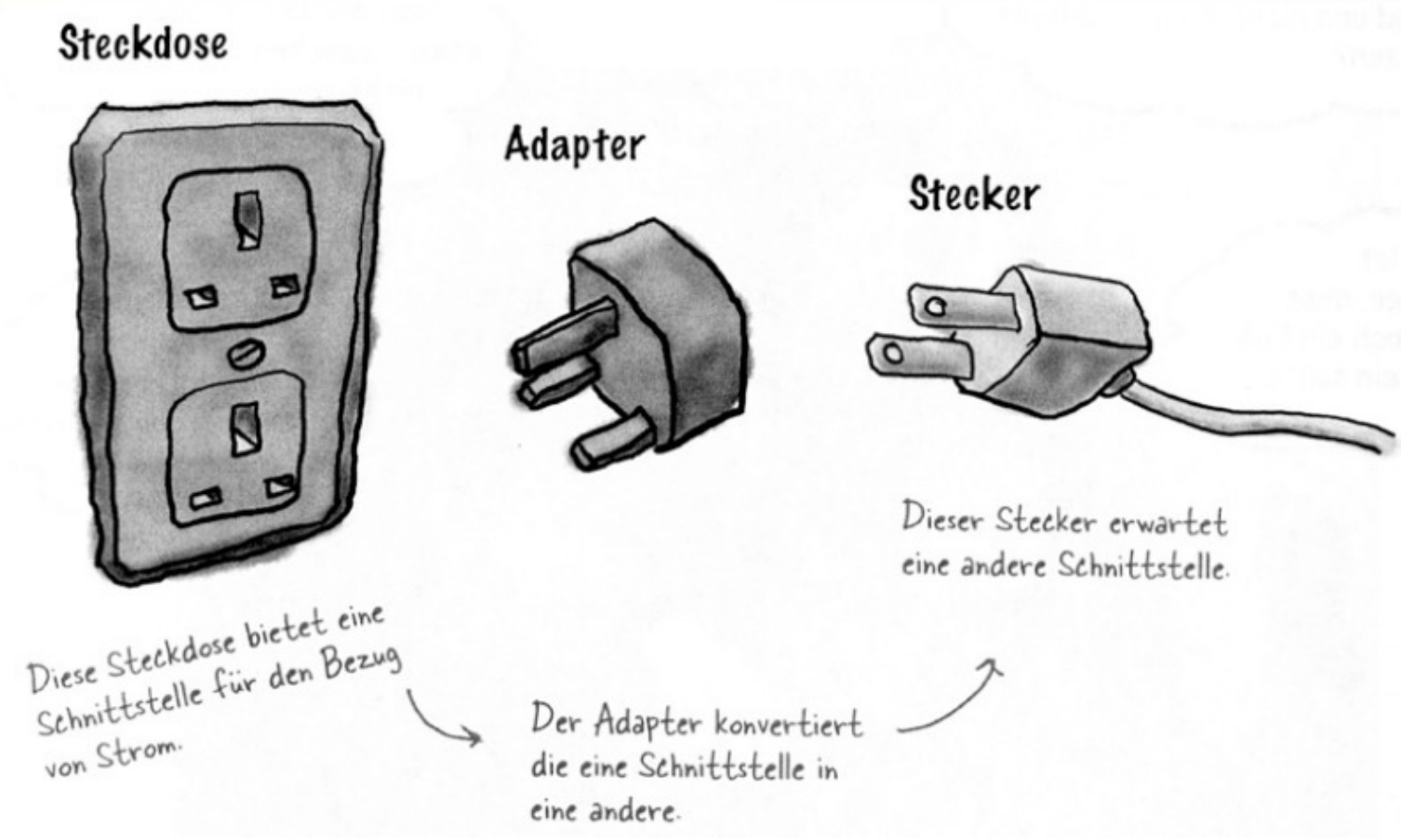
Adapter	
Problem	Eine Klasse soll verwendet werden, obwohl ihre Methoden-Signaturen nicht passen und auch nicht verändert werden können.
Beispiel	In einem Editor soll eine Klasse, die eine einfache Rechtschreibprüfung realisiert, durch eine zugekaufte Klasse ersetzt werden.

Kategorie

- Gamma et al.: **Strukturmuster**
- Ludewig, Lichter: Muster zum Anbinden vorh. Klassen

Quelle: Ludewig, Lichter, (2010), S.441, 442

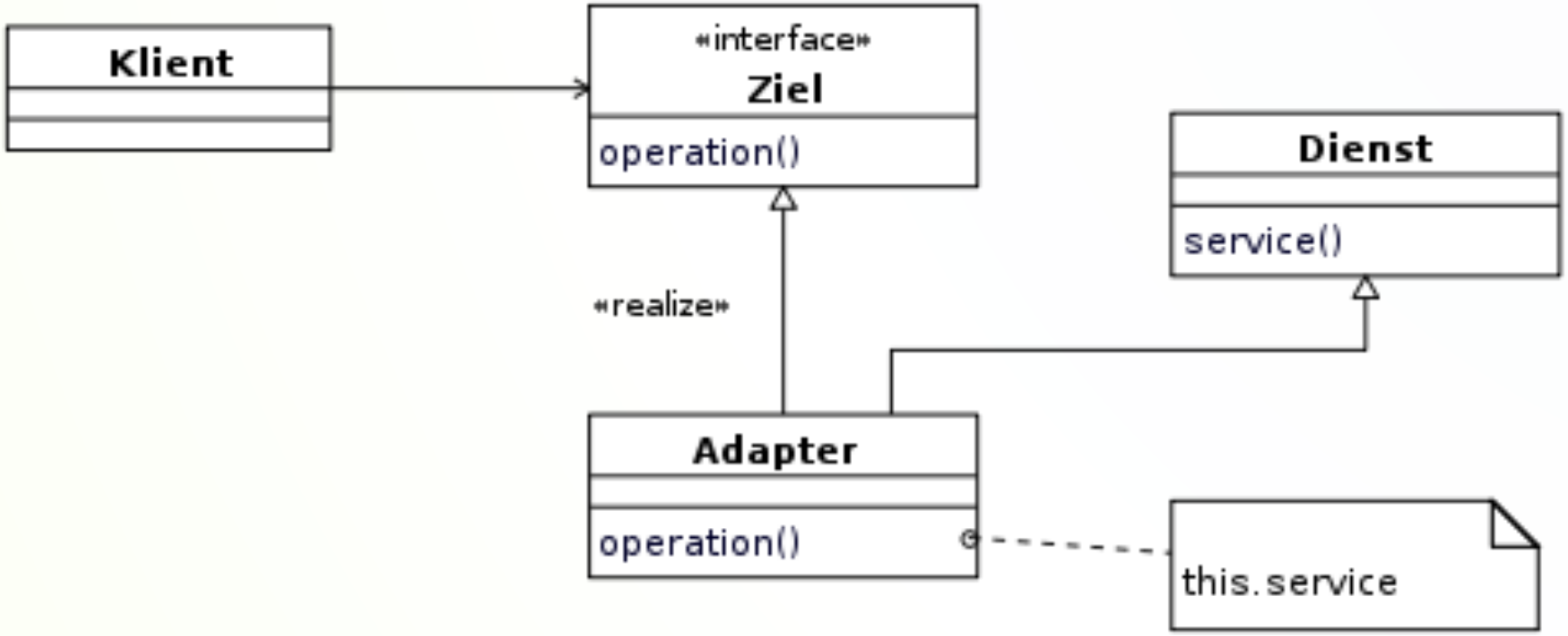
Adapter


Steckdose
Adapter
Stecker
Dieser Stecker erwartet eine andere Schnittstelle.
Diese Steckdose bietet eine Schnittstelle für den Bezug von Strom.
Der Adapter konvertiert die eine Schnittstelle in eine andere.

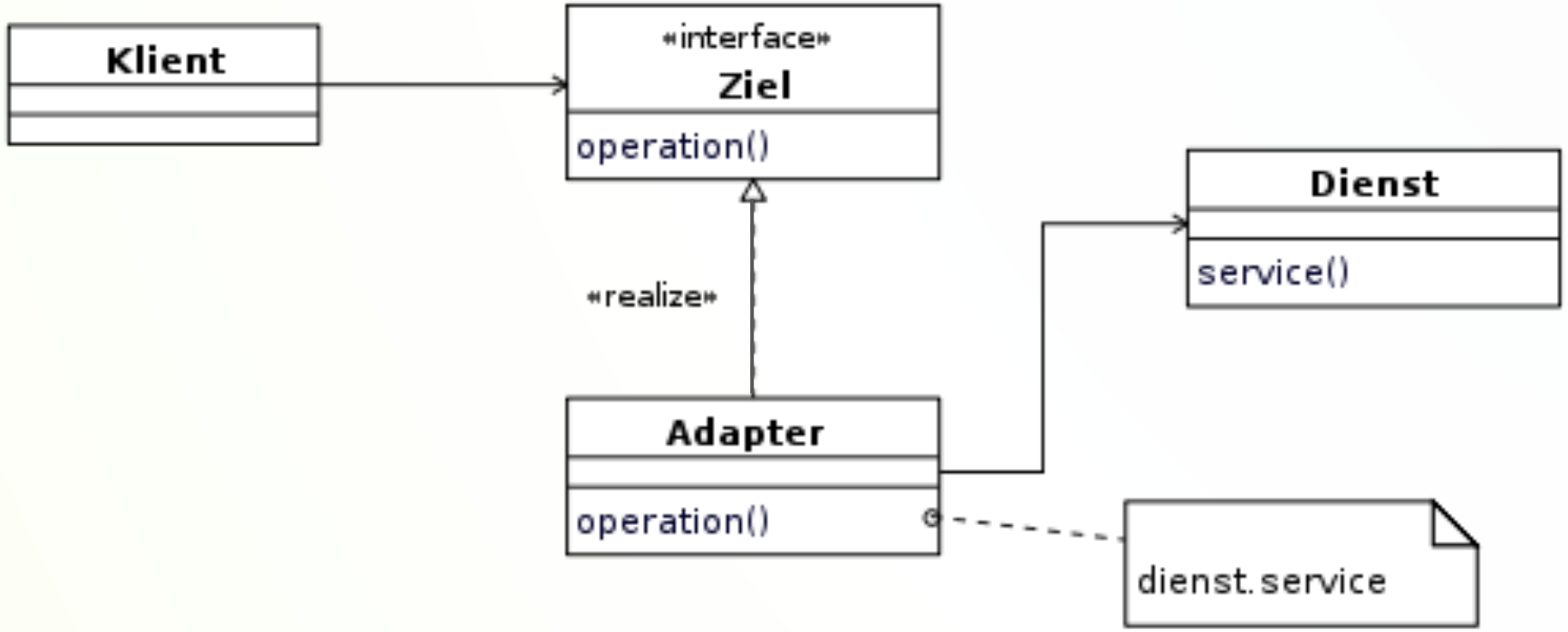
Strukturmuster Adapter

- Anpassung von Schnittstellen
- Bereitgestellte Schnittstelle eines Dienstes (z.B. einer Bibliothek) wird zur Verwendung im Client angepasst

Klassenadapter (Überschreiben)



Objektadapter (Delegation)



Eigenschaften Klassenadapter

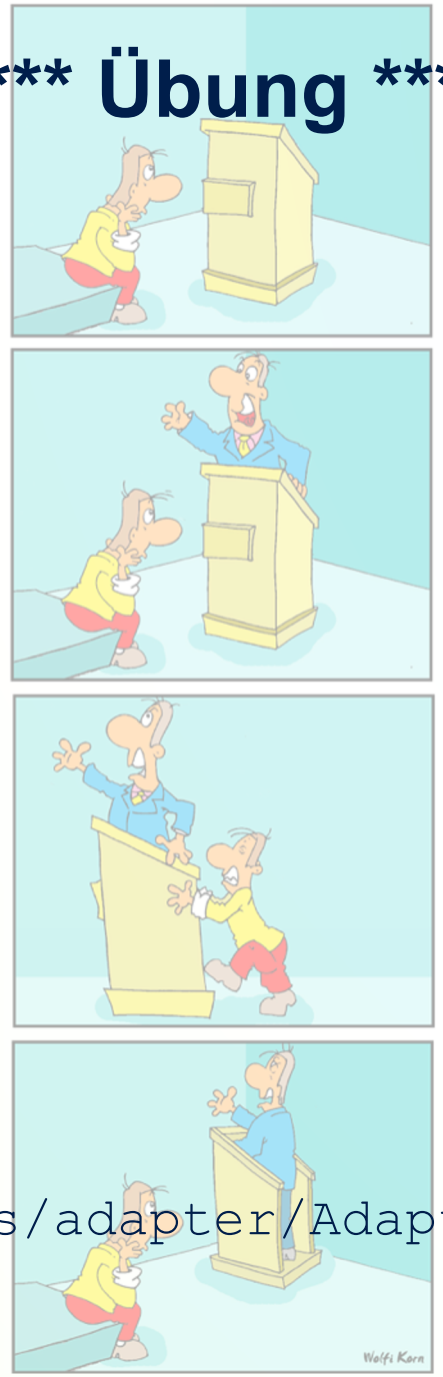
- Adapter kann nur eine Klasse anpassen
- Überschreiben: Adapter kann Teile der adaptierten Klasse modifizieren
- Nur 1 Objekt zur Laufzeit

Eigenschaften Objektadapter

- Adapter kann mehrere Klasse anpassen
- Delegation: Kein direktes Überschreiben von Teilen der adaptierten Klasse
- 2 Objekte (oder mehr) zur Laufzeit

*** Übung ***

Aufgabe 6 (Adapter)



Lösung:

`/DesignPattern/src/com/priess/adapter/AdapterPattern.java`

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

Einzelstück (Singleton)

Prof. Dr. Malte Prieß

Einzelstück (Singleton)	
Problem	Von einer Klasse darf nur genau ein global verfügbares Objekt erzeugt werden.
Beispiel	Es darf nur einen Drucker-Spooler im System geben.

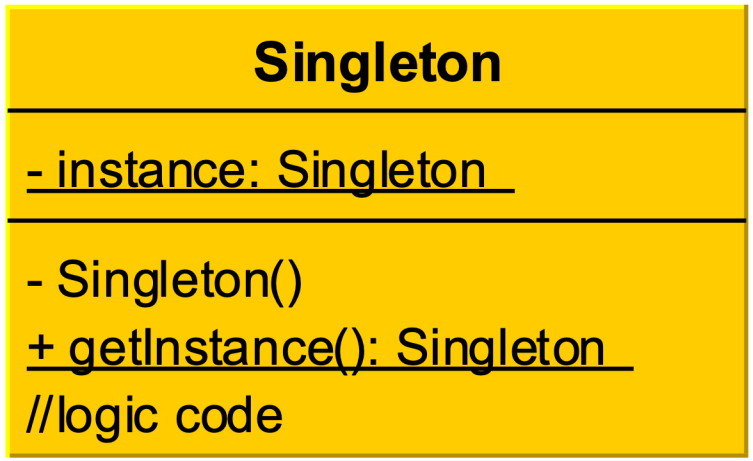
Kategorie

- Gamma et al.: **Erzeugungsmuster**
- Ludewig, Lichter: Muster zum Verwalten von Objekten

Quelle: Ludewig, Lichter, (2010), S.441, 442

Erzeugungsmuster Singleton

- Klasse mit **genau einem Objekt** (ähnlich globaler Variable)
- Erzeugung weiterer Objekte dieser Klasse nicht möglich
- Klasse bietet **einfachen Zugriff** auf einziges Objekt



Einzelstück (Singleton)

```
public class Singleton {

    //Field hält Referenz auf einzigartige Instanz
    private static Singleton instance;

    // Privater Konstruktur verhindert Instanziierung durch Client
    private Singleton(){
    }

    //Stellt Einzigartigkeit sicher. Liefert Exemplar an Client.
    //Hier: Unsynchronisierte Lazy-Loading-Variante
    public static Singleton getInstance(){
        if (instance == null){
            instance = new Singleton();
        }
        return instance;
    }

    //logic code
}
```

Variationen: Eager vs. Lazy Loading

Eager Loading: Instanziierung während die Klasse geladen wird

```
public class Singleton {

    private static final Singleton instance = new Singleton();

    private Singleton() {
    }

    public static Singleton getInstance(){
        return instance;
    }

}
```

Variationen: Eager vs. Lazy Loading

Lazy Loading: Instanziierung beim ersten Bedarf:

```
public class Singleton {

    private static Singleton instance;

    private Singleton(){
    }

    public static Singleton getInstance(){
        if (instance == null){
            instance = new Singleton();
        }
        return instance;
    }
}
```

Synchronisierung

Das **Lazy Loading** schafft allerdings ein neues Problem im Bereich des **Multithreadings**. So **kann es zur Erstellung zweier Singletons kommen**, wenn ein Thread nach der null-Prüfung - direkt vor der Instanziierung - den Fokus abgibt und ein anderer Thread getInstance() durchläuft. Dieser andere Thread erstellt dann ein Singleton. Der erste Thread erhält nun den Fokus wieder und weiß nicht, dass das Singleton bereits erzeugt wurde, und erstellt es noch einmal. **Die Einmaligkeit des Singletons ist verletzt**.

Daher bedarf es einer **Synchronisation**. Synchronisationen sind allerdings **teuer** und erzeugen einen **Overhead bei jedem Aufruf von getInstance()**. Bei einer performancekritischen Applikation mit zahlreichen Aufrufen von getInstance() sollte von dieser Variante Abstand genommen werden.

Lazy Loading mit einfach synchronisierter getInstance()-Methode:

```
public synchronized static Singleton getInstance(){
    if (instance == null){
        instance = new Singleton();
    }
    return instance;
}
```

Allerdings lässt sich auch dies **noch weiter optimieren**, sodass der normale `getInstance()`-Aufruf nicht mehr synchronisiert werden muss. Stattdessen wird nur die Erstellung synchronisiert und ein doppelter Null-Check verwendet. Dadurch wird (nach der einmaligen Erstellung) keine Performance im Normalbetrieb (`getInstance()`-Aufruf) verloren und trotzdem die Einmaligkeit der Singleton-Instanz gewährleistet.

Lazy Loading mit optimierter Synchronisierung:

```
public static Singleton getInstance() {
    if (instance == null) {
        synchronized (instance) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }
    return instance;
}
```

Können wir nicht einfach ein globales statisches Objekt der relevanten Klasse definieren? Also ...

```
public static final Singleton singleton = new Singleton();
```

- Wir können nicht garantieren, dass insgesamt nur ein Exemplar eines statischen Objekts erzeugt werden wird
- Wir verfügen möglicherweise nicht über genügend Informationen, jedes Singleton zur statischen Initialisierungszeit zu erzeugen. Ein Singleton benötigt möglicherweise Werte, die erst später während des Programmablaufs berechnet werden.
- Reihenfolge der Erzeugung der globalen Objekte nicht definiert. Abhängigkeiten zwischen Singletons würden hier zu Fehlern führen

Gamma et al. (2011), S.160, 161

Vorteile:

» Einfache Anwendung. Eine Singletonklasse ist schnell und unkompliziert geschrieben.

Gegenüber *globalen Variablen* ergeben sich eine Reihe von Vorteilen:

» **Zugriffskontrolle**. Das Singleton kapselt seine eigenen Erstellung und kann damit genau kontrollieren, wann und wie Zugriff auf das Singleton erlaubt wird. Setter und Getter können Plausibilitätsprüfungen beinhalten.

» **Sauberer Namensraum**. Der Namensraum wird nicht mit unzähligen globalen Variablen überfrachtet, sondern gekapselt in einem Singleton bereitgestellt.

» **Spezialisierung**. Ein Singleton kann abgeleitet werden, um ihm neue Funktionalität zuweisen zu können. Die Integration in bestehenden Code gestaltet sich einfach. Welche Unterklasse genutzt werden soll, kann dynamisch zur Laufzeit entschieden werden.

» **Lazy-Loading**. Singletons können erst erzeugt werden, wenn sie auch wirklich gebraucht werden.

Nachteile:

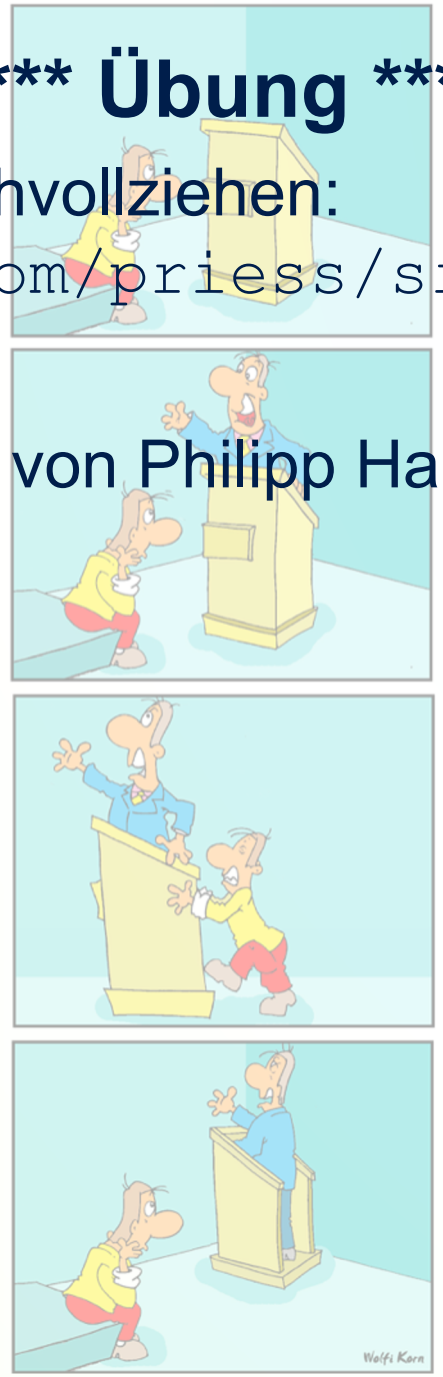
- **Prozedurales Programmieren**: Die ausgiebige Verwendung von Singletons führt zu einem ähnlich ungünstigen Zustand wie bei globalen Variablen. Dies entspricht der prozeduralen Programmierung und hat nichts mit Objektorientierung und Kapselung zu tun.
- **Globale Verfügbarkeit**: Durch die globale Verfügbarkeit wird das Singleton *überall* in der Applikation verfügbar. Enthält das Singleton Daten, so ist dies ein sehr fragwürdiges Design. Welche Daten können es sein, die in *allen* Schichten verfügbar sein sollen? Kann es nicht sogar gefährlich sein, bestimmte Daten oder Funktionalitäten überall frei verfügbar zu machen?

  Kann man diese nicht doch einer Schicht sauber zu ordnen und jede Schicht hinter wohldefinierten Schnittstellen und Datenaustausch kapseln? Singletons verleiten zu unsauberen und intransparenten Programmieren. **Die Notwendigkeit von Singletons sollte stets hinterfragt werden.**
- **Performance**: Besonders bei Mehrbenutzeranwendungen kann ein Singleton die Performance senken, da er - besonders in der synchronisierten Form - ein Flaschenhals darstellt.
- **Einmaligkeit** über physikalische Grenzen. Die Einzigartigkeit eines Singletons über physikalische Grenzen hinweg (JVM) zu gewährleisten, ist schwierig.
- Mehr s. bspw. `https://www.philipphauer.de/study/se/design-pattern/singleton.php`

*** Übung ***

1. JAVA-Implementierung nachvollziehen:
   `/DesignPattern/src/com/priess/singleton/`

2. Bank-Beispiel auf der Seite von Philipp Hauer nachvollziehen



DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

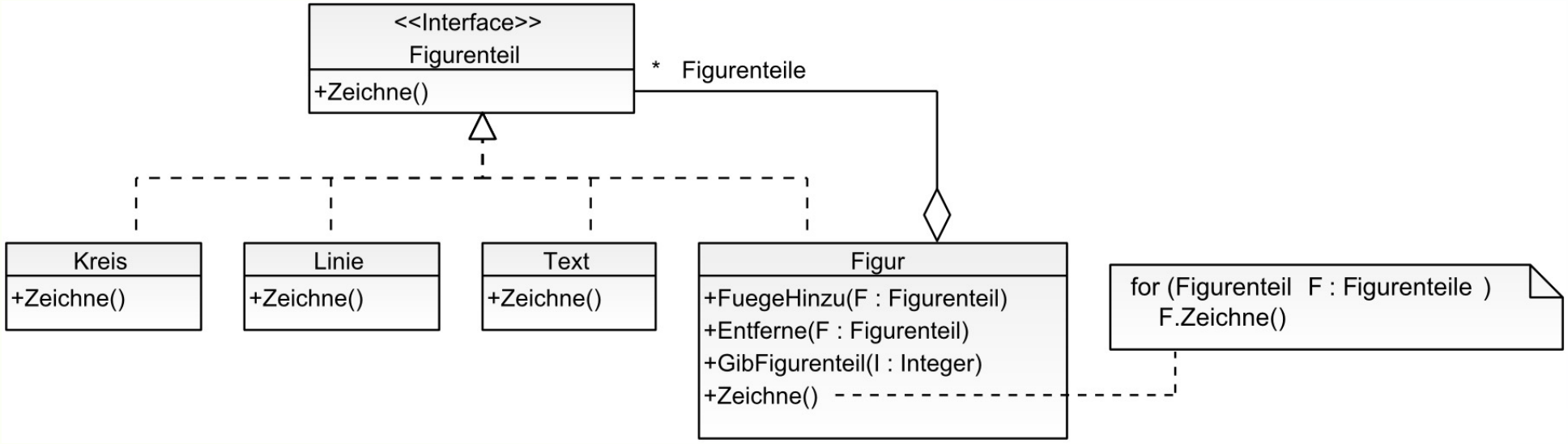
Kompositum (Composite)

Prof. Dr. Malte Prieß

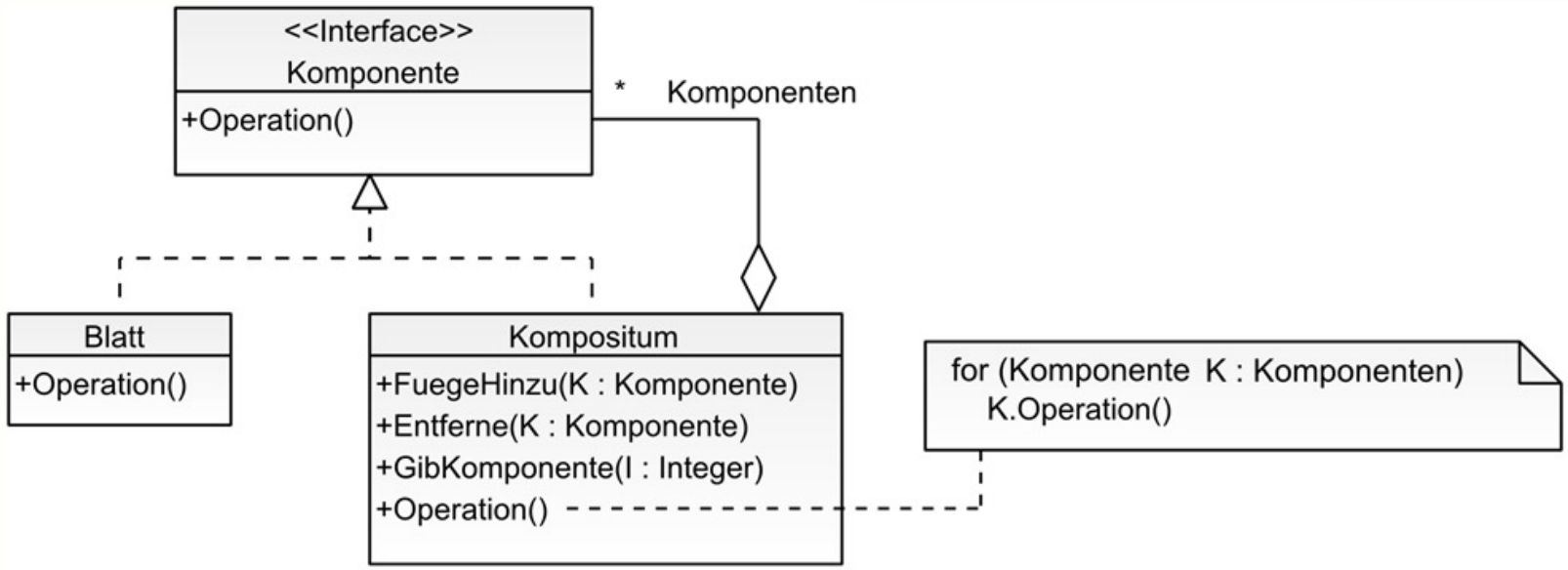
Strukturmuster Kompositum (*Composite*)

Fügt Objekte zu **Baumstrukturen** zusammen, um **Teil-Ganzes-Hierarchien** abzubilden und Kompositionen von Objekten **einheitlich** zu behandeln

Beispiel: Zeichnen von Figuren



Kompositum (Composite)




```
interface Komponente {
    void operation();
}

class Blatt implements Komponent

    public void operation() {
        // ...
    }
}
```

```
class Kompositum implements Komponente {

    List<Komponente> komponenten;

    public void operation() {
        for (Komponente k : komponenten)
            k.operation();
    }

    void fuegeHinzu(Komponente k) { komponenten.add(k); }

    void entferne(Komponente k) { komponenten.remove(k); }

    Komponente gibKomponente(Integer i) { return komponenten.get(i); }
}
```

Anwendung

- Wenn **rekursive** Datenstrukturen vorliegen, d.h.:

  Ein Kompositum kann sowohl aus einfachen Blättern als auch aus weiteren zusammengesetzten Komposita bestehen

Vorteile

- Code der **Clients unabhängig** vom Detailgrad der Komponente (elementar oder zusammengesetzt)
- Neue Kompositions- und Blattklassen **leicht erweiterbar**

*** Übung ***

1. Aufgabe 7 (Composite)

2. Mitarbeiter-Beispiel auf der Seite von Philipp Hauer nachvollziehen

Lösungen:

- UML Diagramm
  - `Dateimanager.asta`
  - `DateimanagerErw.asta`
- JAVA Beispiel-Implementierung
  - `/DesignPattern/src/com/priess/kompositum2/`
  - `/DesignPattern/src/com/priess/kompositum3/`

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

Beobachter (Observer)

Prof. Dr. Malte Prieß

Beobachter (Observer)	
Problem	Mehrere Objekte müssen ihren Zustand anpassen, wenn sich ein bestimmtes Objekt ändert.
Beispiel	Alle GUI-Objekte, die ein Dateisystem darstellen, müssen ihre Darstellung anpassen, wenn Dateien eingefügt oder gelöscht werden.

Kategorie

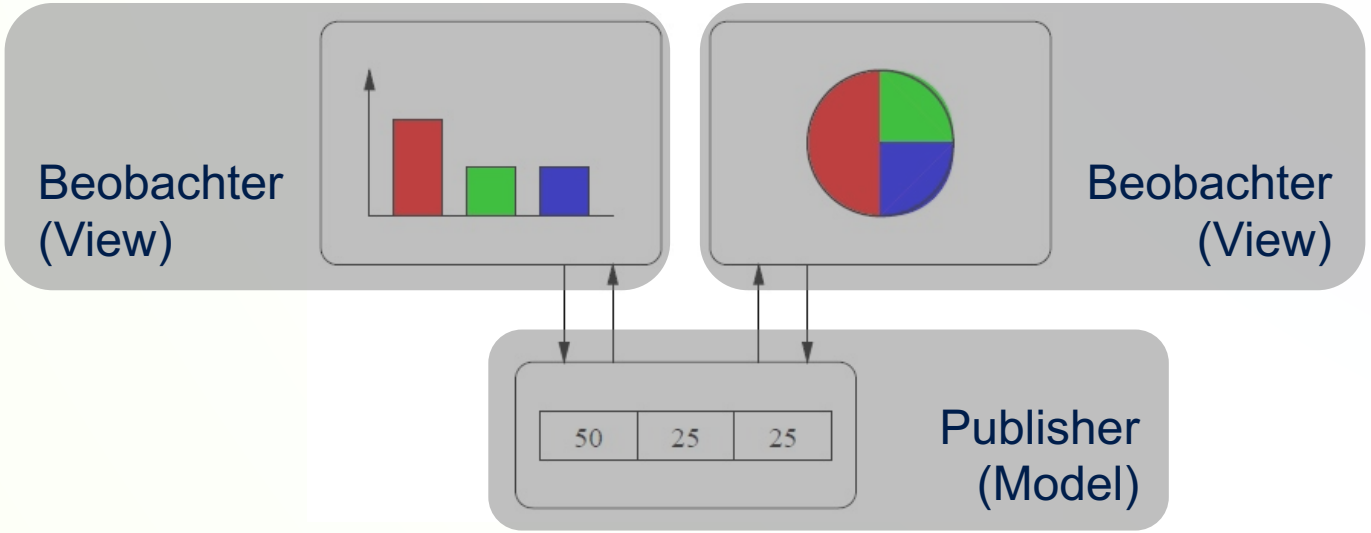
- Gamma et al.: **Verhaltensmuster**
- Ludewig, Lichter: Muster zur Entkopplung von Klassen

Quelle: Ludewig, Lichter, (2010), S.441, 442

Verhaltensmuster Beobachter (*Observer, Publish-Subscribe, Listener*)

- 1:n-Abhängigkeit zwischen Objekten verwalten
- Bei Änderung des beobachteten Objekts (= *Publisher*) werden alle abhängigen Objekte (= *Observer, Subscriber, Listener*) informiert und automatisch aktualisiert (vgl. Change-Update Mechanismus im MVC-Architekturmuster)

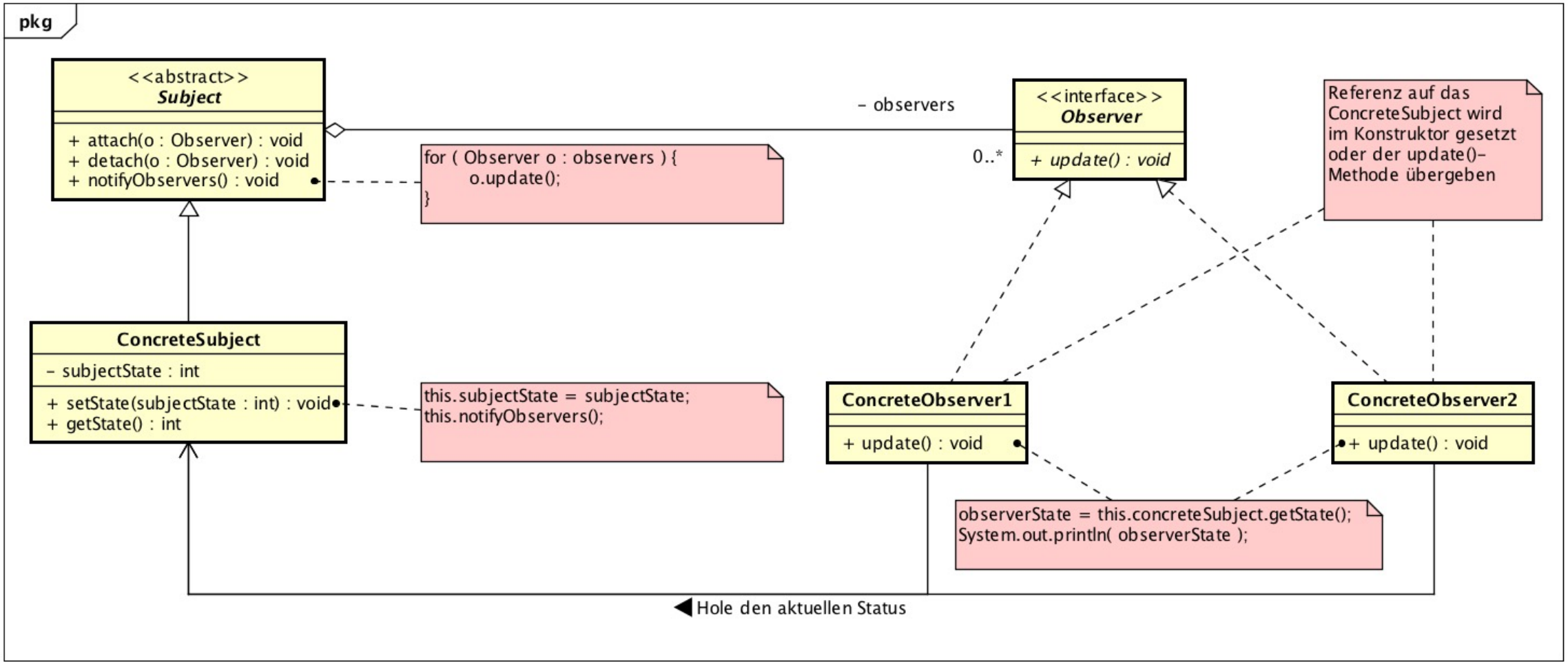
Beispiel: Konsistente Sichten auf Datenmodell in verschiedenen Dashboards



Anwendungsfälle:

- In Fällen in denen die Veränderung eines Objekts die Modifikation eines oder mehrerer Objekte nötig macht, wobei keine Aussage über die Anzahl der zu ändernden Objekte gemacht werden kann.
  - GUIs (User verändert Daten, neue Daten müssen in allen GUI-Komponenten aktualisiert werden).
  - Ein Datensatz (Key-Value-Paare) mit mannigfaltigen Visualisierungen (Tabelle, Balkendiagramm, Tortendiagramm etc.).
  - Im MVC-Pattern (Model-View-Controller) bei der View-Model-Kommunikation.
  - Bei jedem Sekundentakt eines Zeitgeber muss sowohl die Digitaluhr als auch die Analoguhr aktualisiert werden.
- Szenarien, in denen Objekte andere Objekte benachrichtigen sollen, ohne dabei Näheres über das zu benachrichtigende Objekt zu wissen. Das Observer Pattern ermöglicht diese gewünschte lose Kopplung der Objekte.

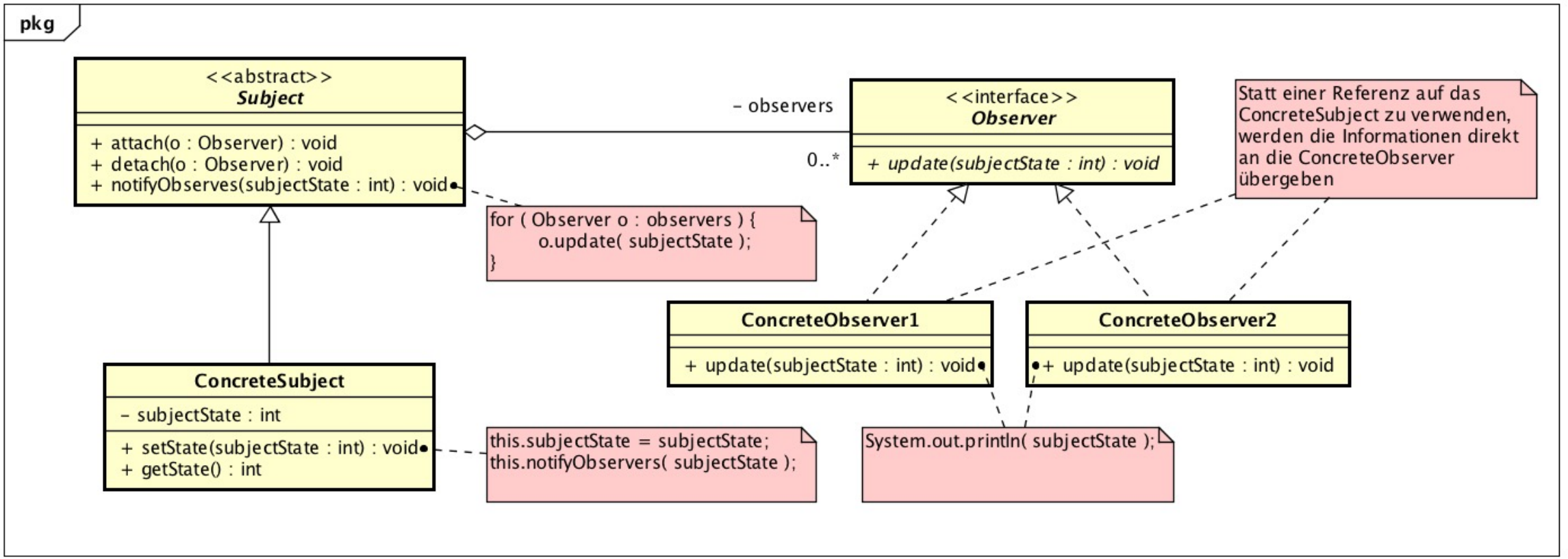
Beobachter (Observer) – Pull-Modell



powered by Astah

Quelle: DesignPattern/com/priess/observerFormalPull/*; ObserverPull.asta

Beobachter (Observer) – Push-Modell



Quelle: DesignPattern/com/priess/observerFormalPush/*; ObserverPush.asta

Vorteile:

- Subject und Observer sind **lose und abstrakt gekoppelt**, da das Subject keine konkreten Observer kennt, sondern **nur die Observerschnittstelle**.
- **Zustandskonsistenz**: Die Daten im Gesamtsystem bleiben konsistent, da die Observer ihren Zustand automatisch bei Änderung des Subjects anpassen.
- **Flexibilität und Modularität**: Das System erlaubt es, dass mehrere verschiedene Observer ein einziges Subject beobachten, aber auch dass ein Observer mehrere Subjects beobachtet.
- **Wiederverwendbarkeit**, **Erweiterbarkeit**:
  - Neue `ConcreteObserver` können leicht hinzugefügt werden, ohne, dass der Code von `Subject` und `ConcreteSubject` angepasst werden muss
  - Neuer `ConcreteObserver` muss lediglich das Interface Observer implementieren und beim `ConcreteSubject` angemeldet werden
  - Durch das Observer Pattern lassen sich Subject und Observer unabhängig voneinander variieren. Somit kann das Subject wiederverwendet werden, ohne seine abhängigen Observer verwenden zu müssen und umgekehrt.
- **Dynamik**: `Observer` können zur Laufzeit hinzugefügt und wieder entfernt werden

Nachteile:

- **Aktualisierungskaskaden und –zyklen**: Bei umfangreichen Systemen mit vielen Subjects und Observer kann es schnell zu ganzen Aktualisierungskaskaden kommen, da die Observer nichts von einander wissen und nicht abschätzen können, welche Folgen eine einzige Modifikation an einem Subject hat.
- **Abmeldung von Observer**: Schnell vergisst man ein Observer beim Subject abzumelden, wenn man es nicht mehr braucht. Dies kann in Fällen von Mehrfachanmeldung (Mehrfachbenachrichtigung) merkwürdige Effekte zur Folge haben und verhindert die automatische Speicherfreisetzung (Garbage Collection in Java und C#), da das Subject immer noch eine Referenz auf ein nicht mehr gebrauchtes Objekt hält.

Für die Art und Weise, wie Observer die benötigten Informationen erhält, gibt es zwei Varianten, Push- und Pull-Modell (Dabei gibt es kein richtig und falsch, sondern nur ein zweckmäßig und nicht zweckmäßig im bestimmten Fall)**:**

- Beim **Pull-Modell** erhält der `Observer` nur eine minimale Benachrichtigung und muss sich die benötigten Informationen selber aus dem `Subject` holen
- Dazu erhält der `Observer` eine Referenz auf das konkrete `Subject` (via Konstruktor oder via Argument der `update()`-Methode)

```
public interface Observer2 {

    public void update(ConcreteSubject concreteSubject);

}
```

+ Jeder `Observer` erhält nur die benötigten Informationen
+ Klar, von welchem `Subject` die Informationen stammt
+ `update()`-Schnittstelle stabil: Mehr Informationen über weiteren Getter beim `Subject`; andere `Observer` bleiben unverändert
– `Observer` muss selbstständig herausfinden, was sich geändert hat

Für die Art und Weise, wie Observer die benötigten Informationen erhält, gibt es zwei Varianten, Push- und Pull-Modell:

» Im **Push-Modell** übergibt das Subject der update()-Methode detaillierte Informationen über die Änderung als Parameter

```
public interface Observer1 {
    public void update(int length, int width, boolean visible, String name);
}
```

\+ Stärkere Entkopplung (Kopplung geringer) von `Observer` und `Subject`: `Observer` benötigt hierbei keine Informationen über das `Subject`

\+ Das `Subject` kann den `Observern` konkret mitteilen, was sich geändert hat

– Untersch. Bedürfnisse: Jedoch benötigt nicht jeder `Observer` alle ihm übergebenen Informationen. Es kann unnötiger Datentransfer entstehen.

– Erweiterungen schwierig: Ergänzung um weiteren Parameter erfordert Anpassung aller konkreter `Observer`. Abhilfe könnte in diesem Fall die Nutzung eines speziellen (Event-)Objektes als Paramter statt der Übergabe der einzelnen Parameter: Dieses Objekt kann alle notwendigen Daten kapseln. Dieses Vorgehen ist der klassische Ansatz von zahlreichen GUI-Bibliotheken (wie Swing): Eventobjekte kapseln die Änderungsinformation.

Voraussetzung ...

Push-Modell

» Wenn das Subject Aussagen über die Bedürfnisse seiner Observer treffen kann (beispielsweise, wenn nur ein Observer existiert oder einige gleichartige), dann ist das Push-Modell zu bevorzugen.

Pull-Modell

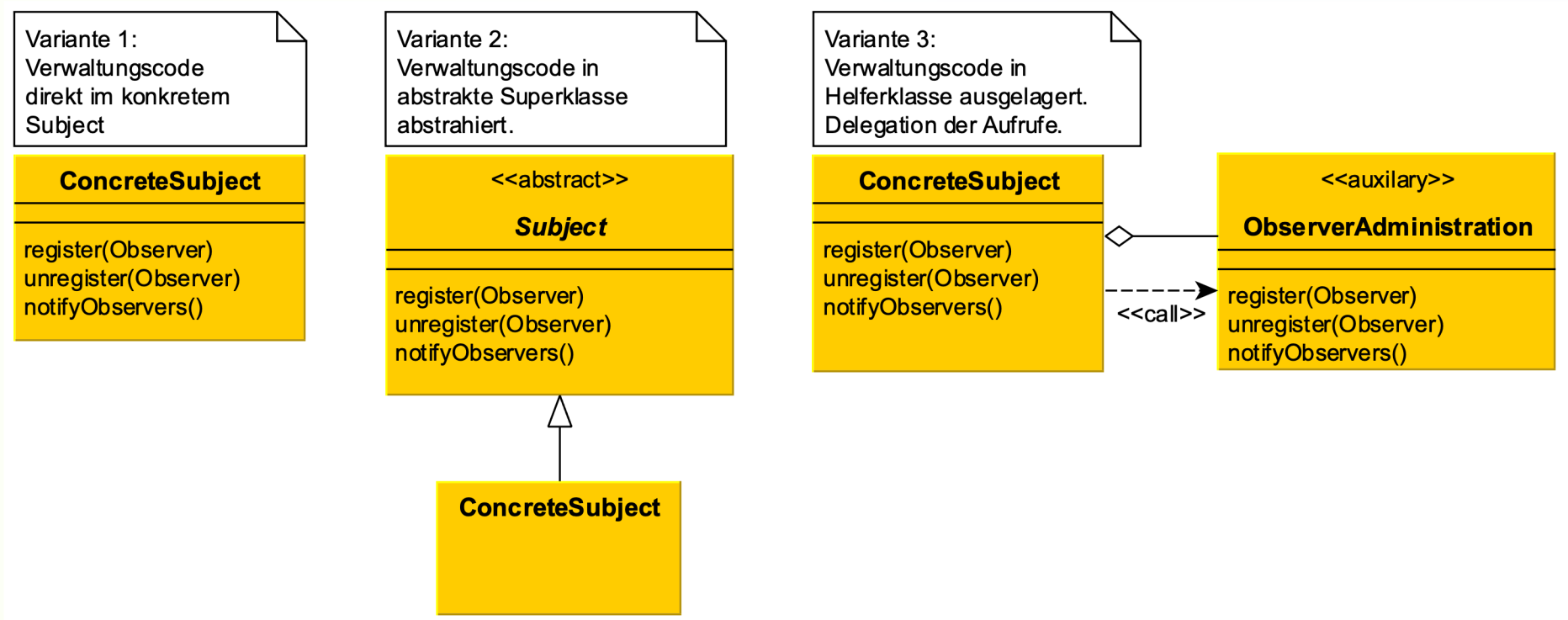
» Weiß das Subject aber nichts über seine Observer (beispielweise, wenn es viele verschiedenartige Observer sind), dann sollte das Pull-Modell realisiert werden.

Subject: Abstrakte Superklasse oder Interface?

- Die Subjectschnittstelle kann auch ein Interface statt einer abstrakten Klasse sein. Dadurch ist man gezwungen, den oft generischen Administrations- und Aktualisierungscode in jeder Subjectimplementation neu zu schreiben, statt den entsprechenden Code von einer abstrakten Superklasse zu erben. Obwohl dies Kohäsionsverlust bedeutet, kann solch ein Vorgehen in Fällen sinnvoll sein, in den kein allgemeiner Administrations- und Aktualisierungscode von den konkreten Subjects abstrahiert werden kann, da sie zu unterschiedlich sind.
- Natürlich kann auch keine Schnittstelle oder abstrakte Subjectsuperklasse definiert werden und Observer können gleich gegen ein konkretes Subject arbeiten. Dies ist dann zweckmäßig, wenn das Subject nicht ausgetauscht werden muss.

Subject: Ort des Verwaltungscodes

Wohl überlegt sollte auch sein, in welcher Klasse die Verwaltungsmethoden (registerObserver(), unregisterObserver(), notifyOberserver() etc.) für die Observer sein sollen. Drei Möglichkeiten wären denkbar:



Subject: Ort des Verwaltungscodes

- **Direkt im konkretem Subject**:
  - + Einfach und verständlich.
  - + Platz der Superklasse verfügbar (bei Einfachvererbung)
  - - Kohäsionsverlust (Vermischung von Verwaltungscode und Subjectlogik)
- **Abstrakte Subjectsuperklasse**:
  - + Schlanker Code der konkreten Subjects
  - + Wiederverwendbarkeit des Verwaltungscodes
  - - Platz der Superklasse vergeben (bei Einfachvererbung)
  - - Probleme beim Pull-Modell. Update()-Methode ist mit der abstrakten Subjectsuperklasse parametrisiert. Observer müssen Fallunterscheidungen und Downcasts durchführen.

Subject: Ort des Verwaltungscodes

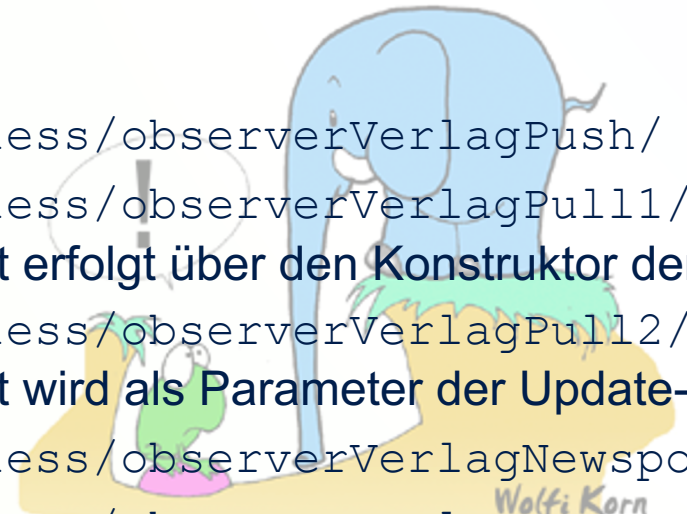
- **Extra-Helferklasse**, die von dem Subject aggregiert wird. Das Subject delegiert Verwaltungsaufrufe an die Helferklasse, die die gesamte Verwaltungslogik kapselt.
  - + Hohe Kohäsion des Subjects, dank Delegation.
  - + Wiederverwendbarkeit des Verwaltungscodes.
  - + In Programmiersprachen ohne Mehrfachvererbung vergibt man sich damit nicht den Platz der Superklasse.

*** Übung ***

1. Aufgabe 9 (Observer)
2. Verlags-Beispiel auf der Seite von Philipp Hauer im Detail nachvollziehen; hier auch die beispielhafte Erweiterung um ein weiteres Subject betrachten

Lösungen:

- `VerlagPush.asta`
- `VerlagPull.asta`
- `/DesignPattern/src/com/priess/observerVerlagPush/`
- `/DesignPattern/src/com/priess/observerVerlagPull1/`
  (Referenz auf das ConcreteSubject erfolgt über den Konstruktor der ConcreteObserver)
- `/DesignPattern/src/com/priess/observerVerlagPull2/`
  (Referenz auf das ConcreteSubject wird als Parameter der Update-Methode übergeben)
- `/DesignPattern/src/com/priess/observerVerlagNewsportalPush/`
- `/DesignPattern/src/com/priess/observerVerlagNewsportalPull1/`
- `/DesignPattern/src/com/priess/observerVerlagNewsportalPull2/`



DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

Abstrakte Fabrik (Abstract Factory)

Prof. Dr. Malte Prieß

Abstrakte Fabrik (Abstract Factory)	
Problem	In einer Anwendung müssen kontextabhängig Objekte unterschiedlicher Klassen erzeugt werden.
Beispiel	Ein grafischer Editor muss abhängig vom eingesetzten Monitortyp unterschiedliche grafische Objekte erzeugen.

Kategorie

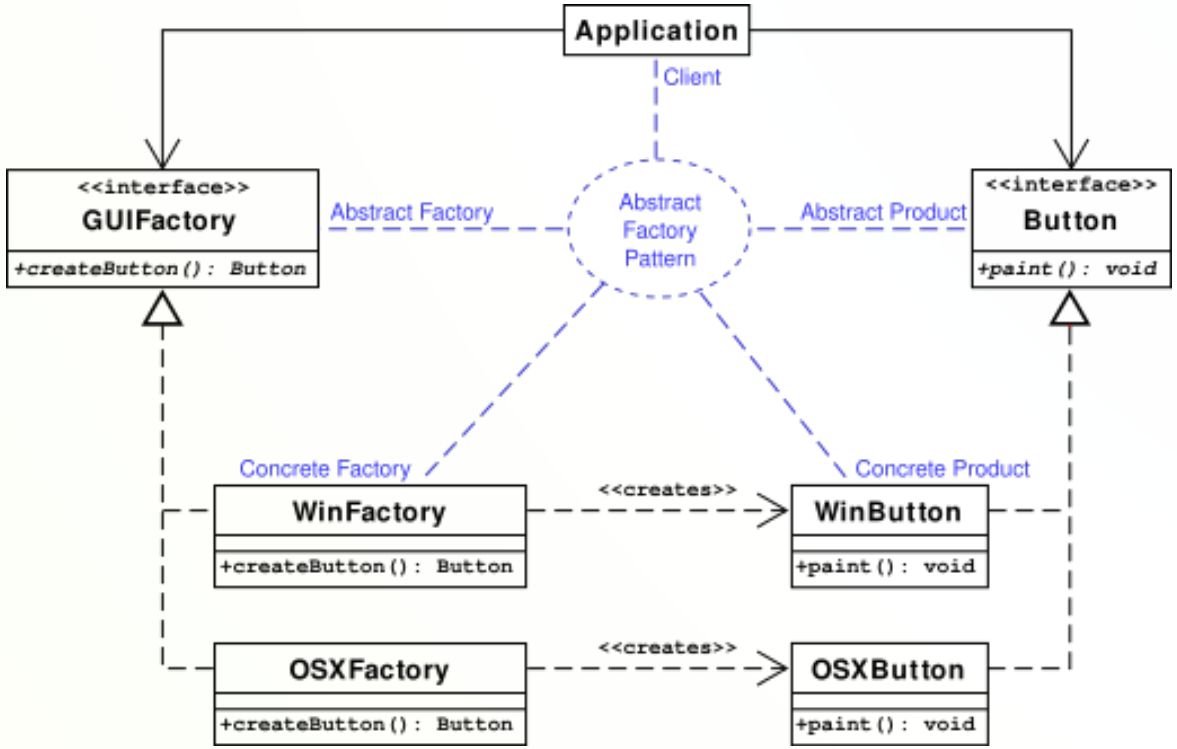
- Gamma et al.: **Erzeugungsmuster**
- Ludewig, Lichter: Muster zur Trennung von unterschiedlichen und gemeinsamen Eigenschaften

Quelle: Ludewig, Lichter, (2010), S.441, 442

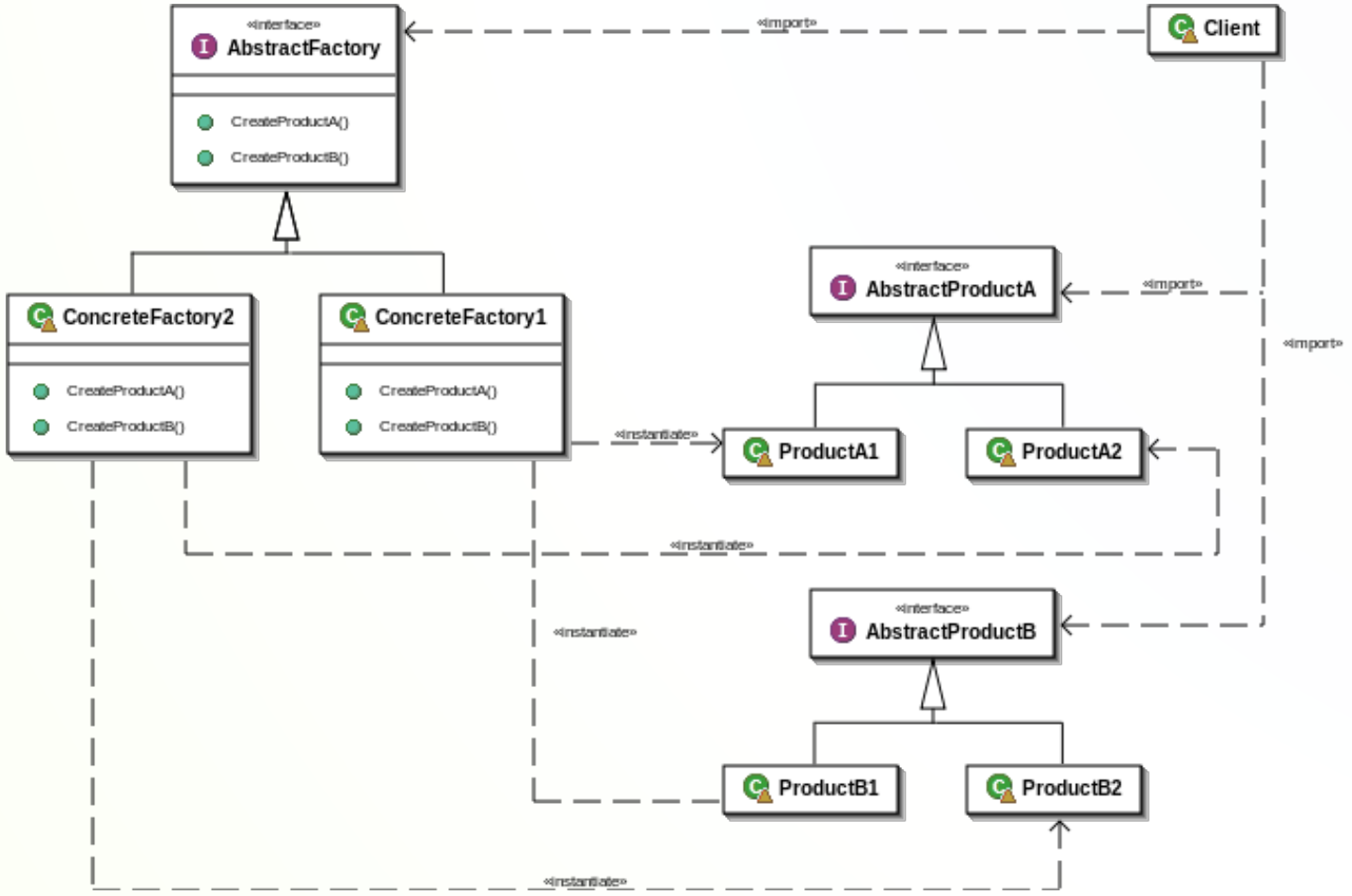
Erzeugungsmuster Abstrakte Fabrik (*Abstract Factory, Kit*)

» Schnittstelle zum Erzeugen von Familien verwandter Objekte, ohne ihre konkreten Klassen zu benennen

Beispiel: GUI-Bibliothek mit mehreren Themes (Look & Feel)



Erzeugungsmuster Abstrakte Fabrik (*Abstract Factory, Kit*)



Weiteres Beispiel: Spielesammlung (vgl. Übungsaufgabe)

- Abstraktes Produkt 1
  `Spielbrett`, auf das Spielfiguren platziert werden und das eine Methode besitzt, um gezeichnet zu werden. Konkrete, abgeleitete Produkte sind `Mühlebrett`, `Damebrett`, `Halmabrett`, …
- Abstraktes Produkt 2
  `Spielfigur`, die auf ein Spielbrett gesetzt wird. Konkrete, abgeleitete Produkte sind `Holzstein`, `Hütchen`, …
- Abstrakte Fabrik
  `Spielfabrik`, die zusammengehörige Teile (Spielbrett und Spielfiguren) eines Gesellschaftsspiels erstellt. Konkrete, davon abgeleitete Fabriken sind `Mühlefabrik`, `Damefabrik`, `Halmafabrik`, …

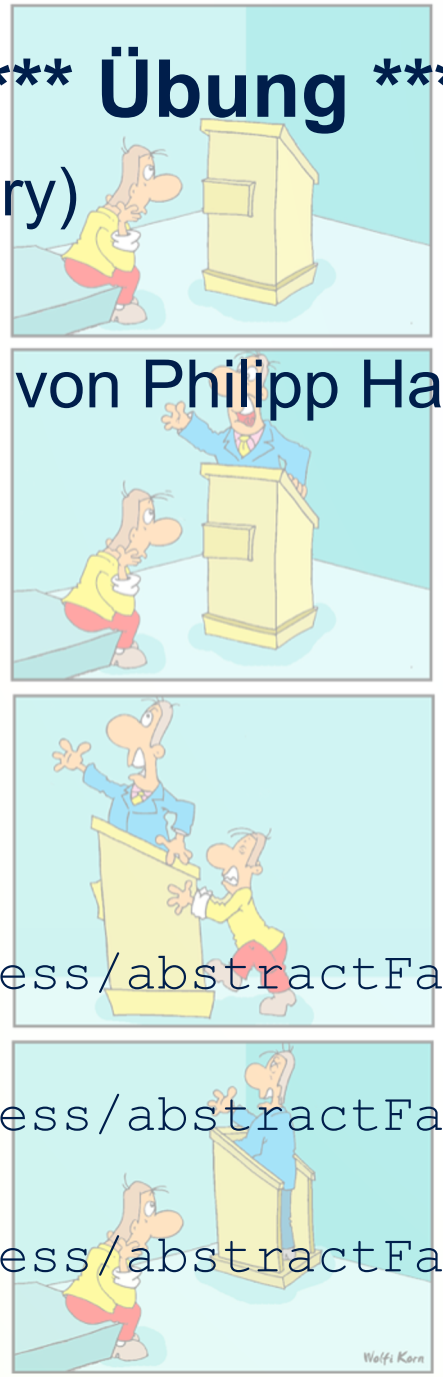
Anwendung

- Wenn Unabhängigkeit der Anwendung von konkreten Produkten erforderlich ist
- Anwendung soll verschiedene Produktfamilien unterstützen und zur Laufzeit austauschen können

Vorteile

- **Flexibilität**, **Allgemeingültigkeit**:
  - Client ist von konkreten Produkten unabhängig (allgemeingültiger Code)
  - Einfacher Austausch von Produktfamilien: Einzige Änderung ist die Instanziierung der konkreten Fabrik an einer zentralen Stelle im Client
- **Erweiterbarkeit** und **Wartbarkeit**
  - Einfache Erweiterung durch neue Fabriken/Produktfamilien
  - Leichte Änderungen an bestehenden Fabriken möglich (lediglich Änderung der `createProduct()`-Methode)
- **Konsistenz**
  - Es werden nur solche Produkte erzeugt, die zueinander passen (spezifiziert in der Implementierung der konkreten Fabrik)

*** Übung ***



1. Aufgabe 10 (Abstract Factory)
2. Shop-Beispiel auf der Seite von Philipp Hauer nachvollziehen

Lösungen:

- Basis-Ansatz:
  `/DesignPattern/src/com/priess/abstractFactorySpielBasic/`
- Etwas mehr:
  `/DesignPattern/src/com/priess/abstractFactorySpielFeature1/`
- Noch etwas mehr:
  `/DesignPattern/src/com/priess/abstractFactorySpielFeature2/`

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

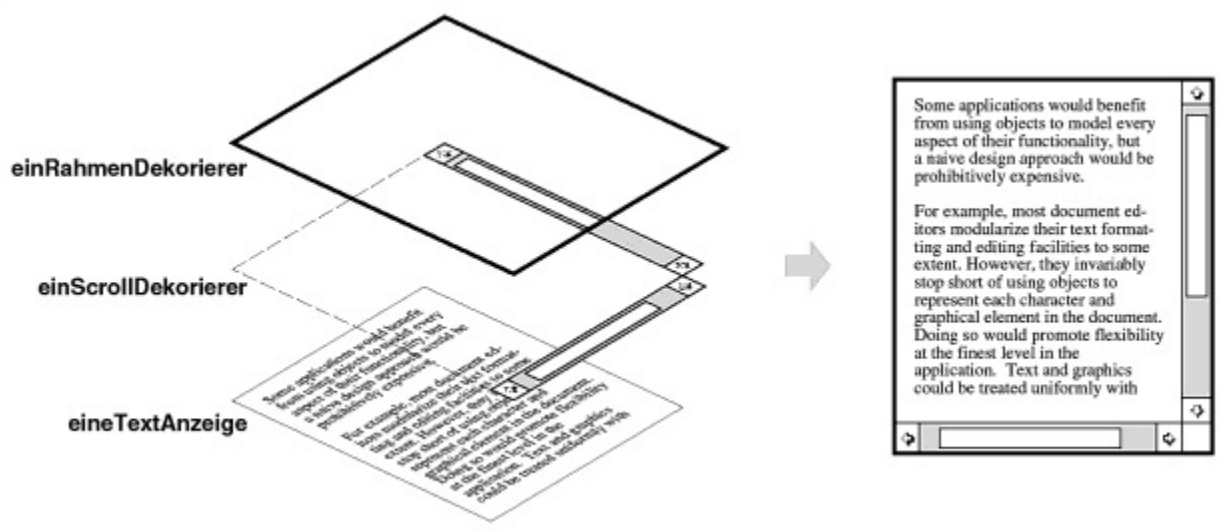
Dekorierer (Decorator)

Prof. Dr. Malte Prieß

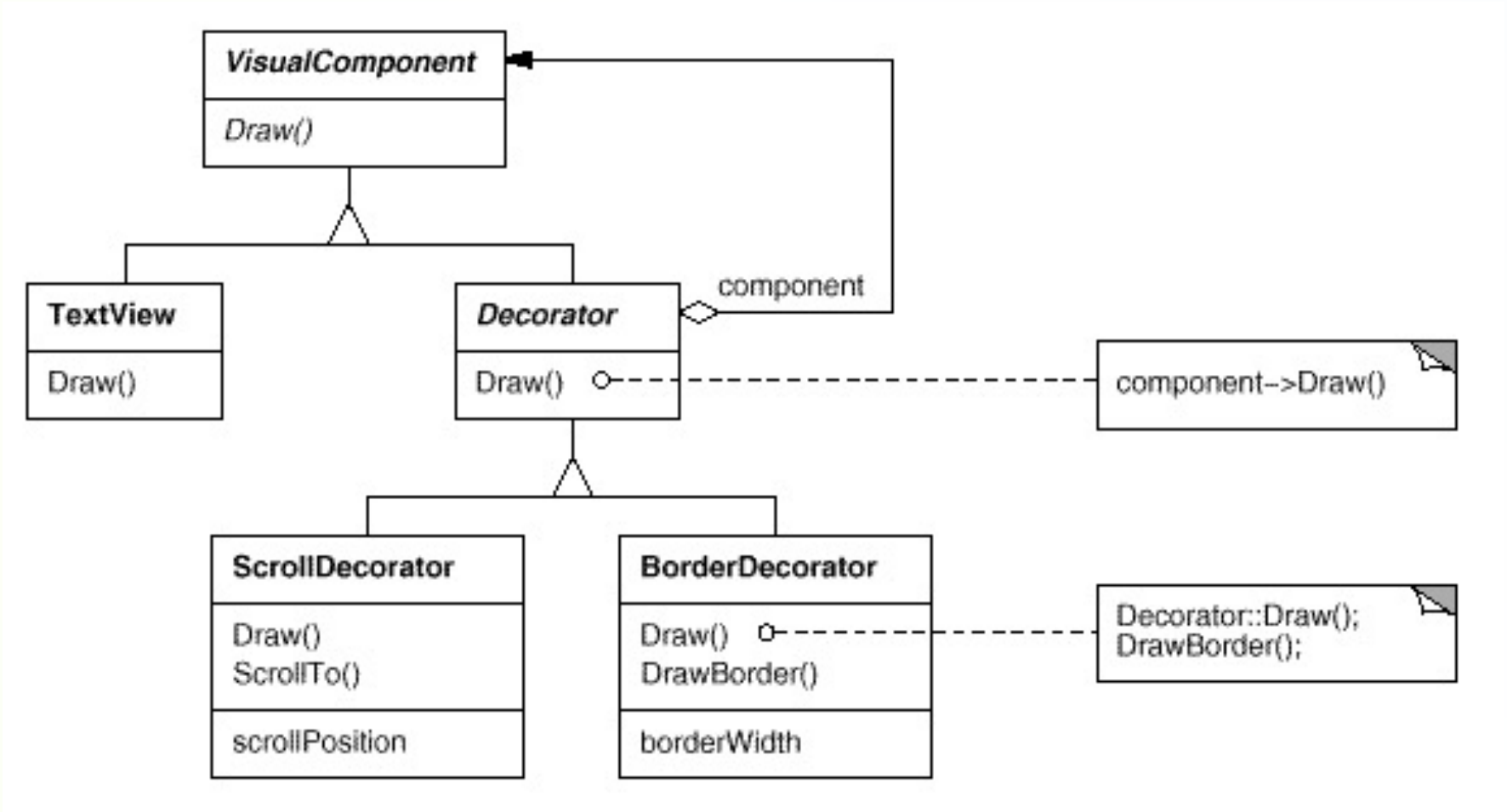
Strukturmuster Dekorierer (*Decorator, Wrapper*)

- Erweitert Objekte dynamisch um Funktionalität
- Flexible Alternative zur Unterklassenbildung

Beispiel: Dekoration einer Textanzeige

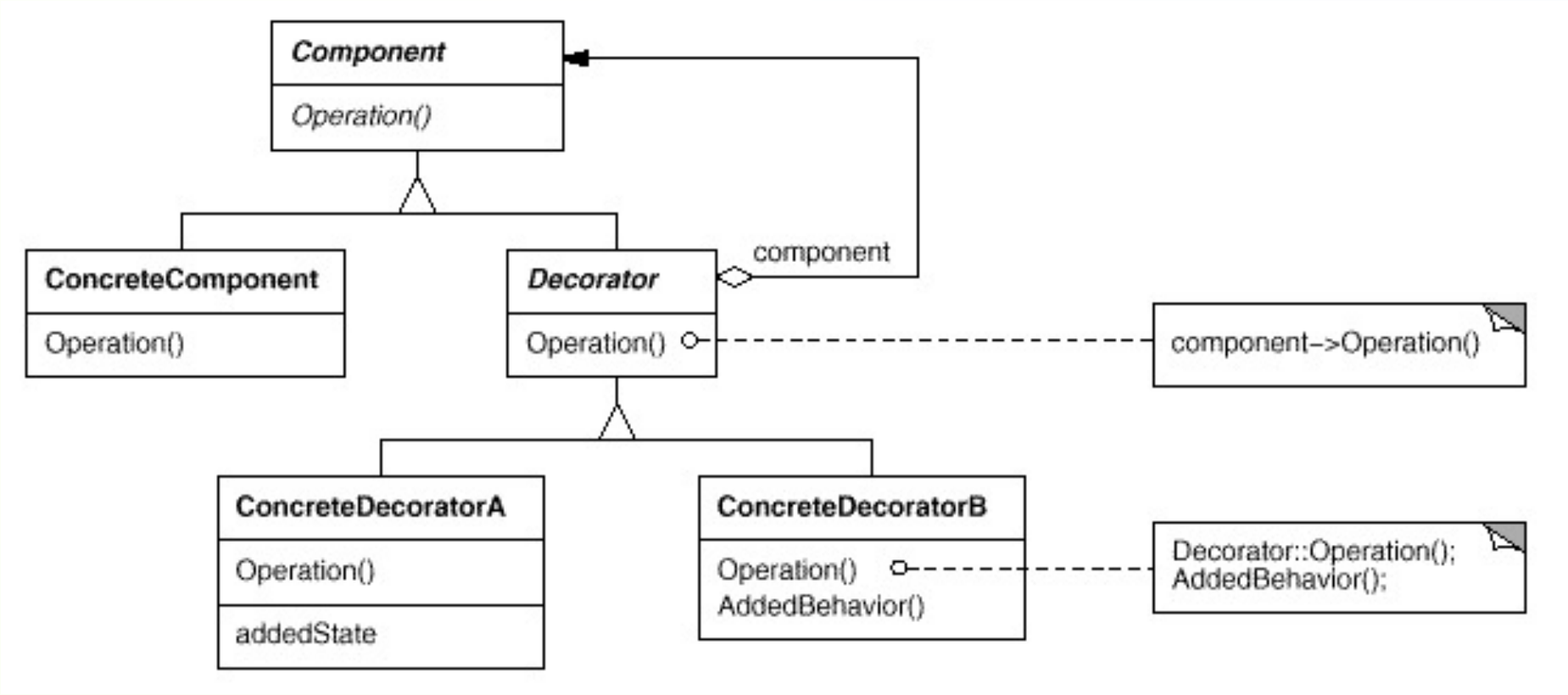


Beispiel: Dekoration einer Textanzeige

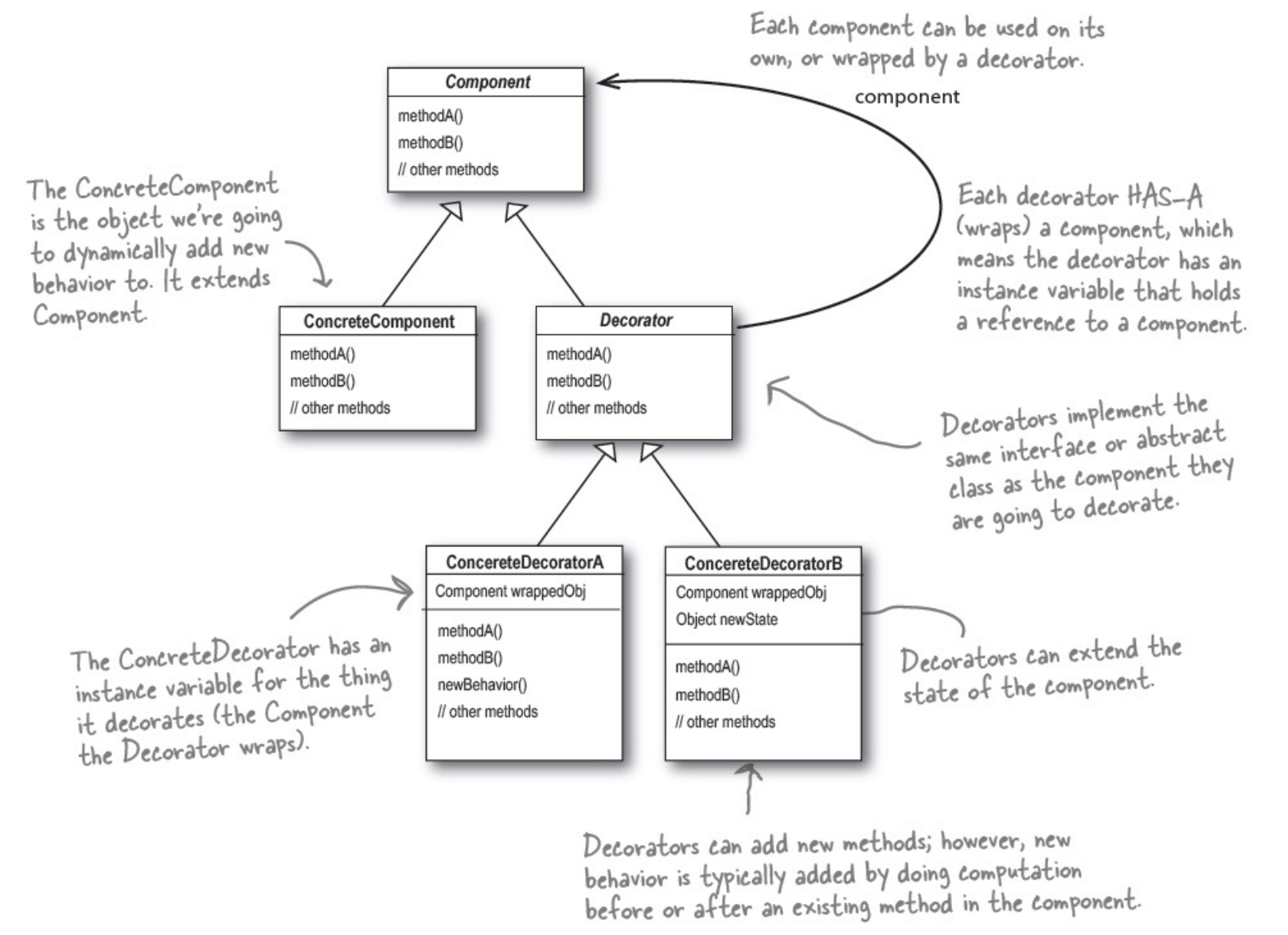


Gamma et al. (2011), S.201

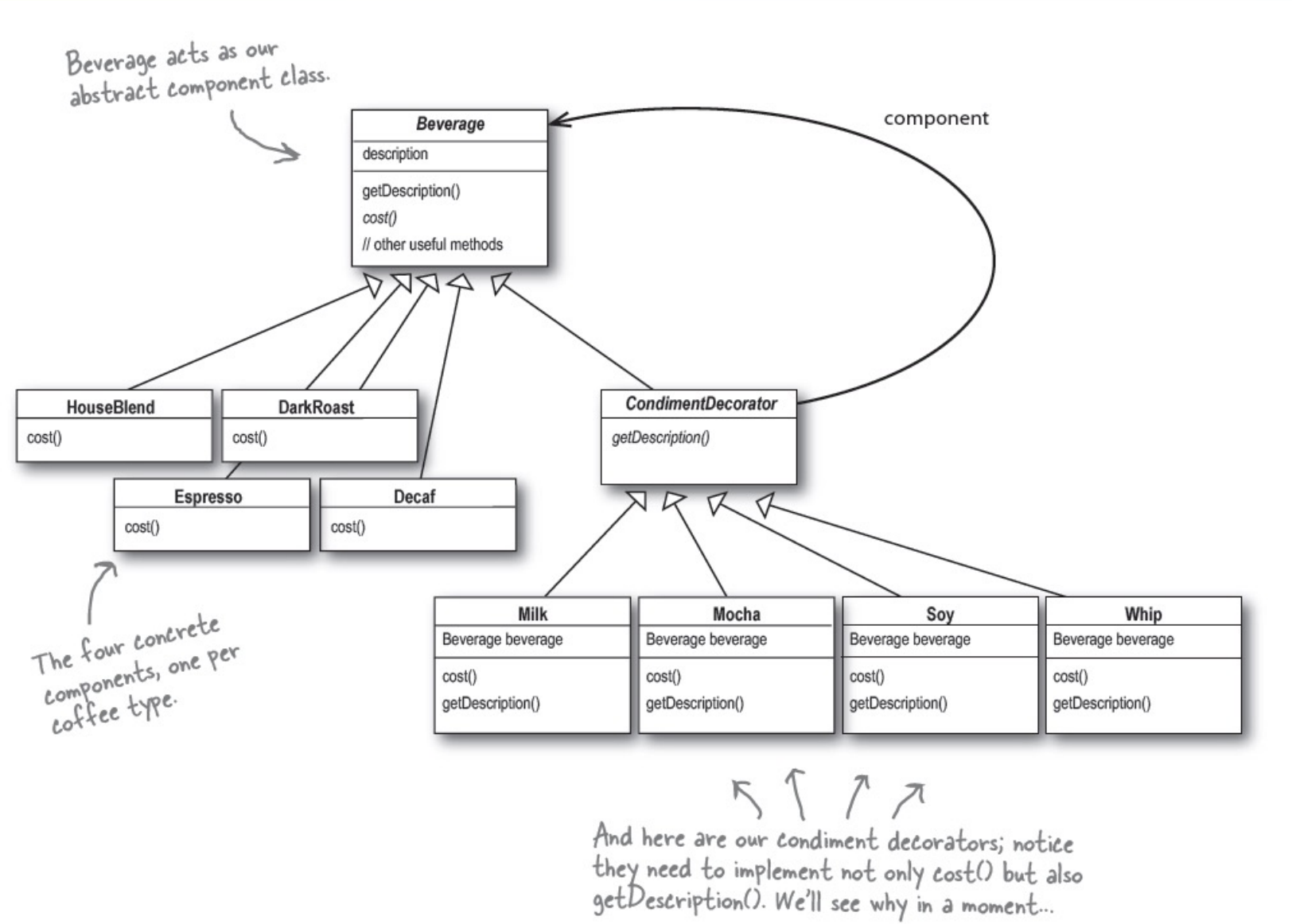
Strukturmuster Dekorierer



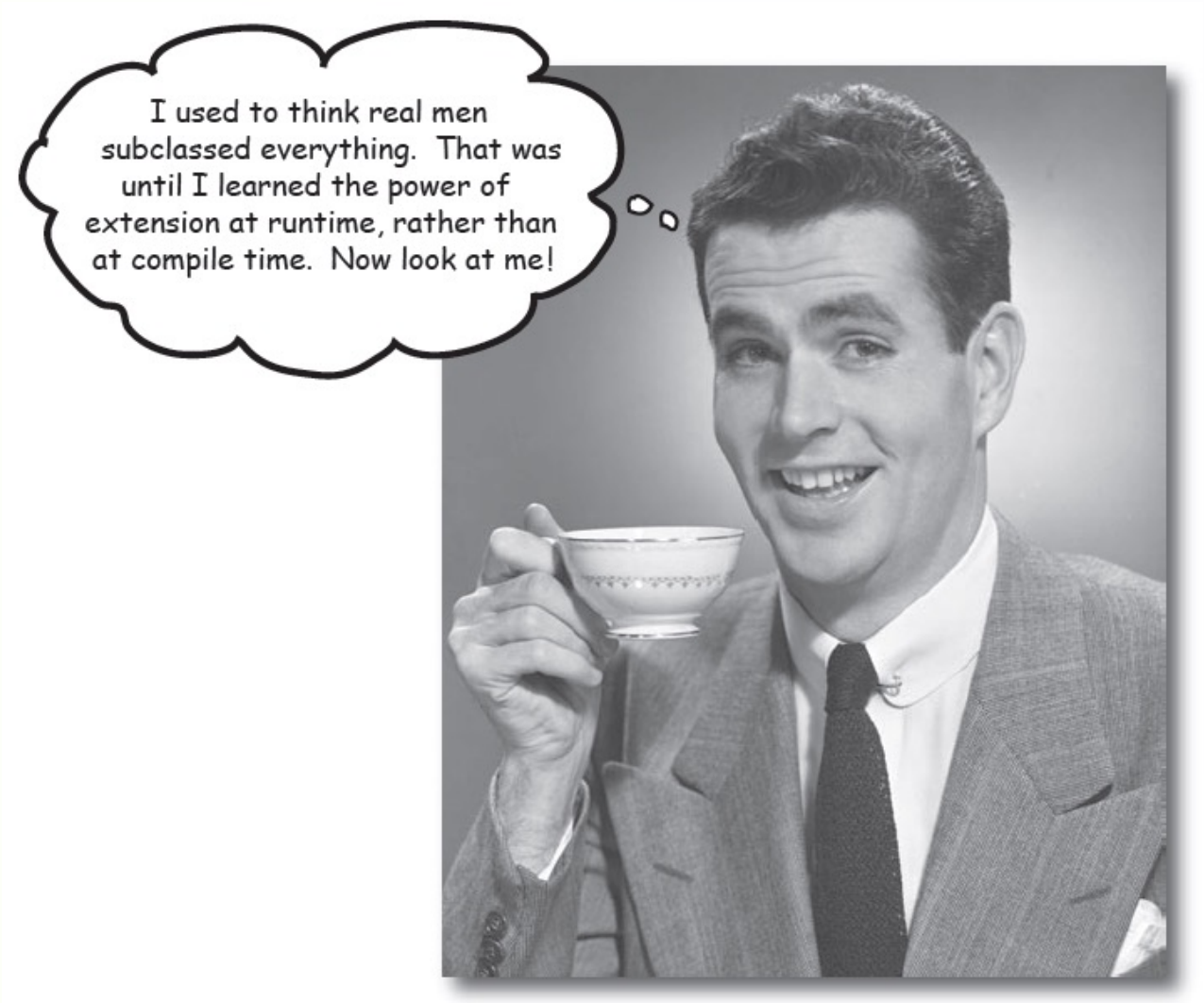
Dekorierer (Decorator)



Dekorierer (Decorator)



Dekorierer (Decorator)



Quelle: [FFS+05]

Anwendung

- Wenn Objekte dynamisch (zur Laufzeit) um Funktionalität erweitert bzw. reduziert werden sollen
- Rekursives Schachteln von Funktionalitäten
- Wenn Erweiterung durch Unterklassen nicht sinnvoll, insb. wegen Explosion der Klassenanzahl

Vor- und Nachteile

+ Flexibilität gegenüber statischer Vererbung bzgl. Erweiterungs-reihenfolge (neue Klasse für jede zusätzliche Funktionalität)
+ Allgemeine (in der Hierarchie oben stehende) Klassen werden nicht mit Funktionalität überlastet
– Viele kleine Objekte → Debugging unübersichtlich
– Vorsicht bei Vergleich zur Laufzeit:
Dekorierer und dekoriertes Objekt sind nicht identisch

*** Übung ***

1. Aufgabe 8 (Decorator)

2. Restaurant-Beispiel auf der Seite von Philipp Hauer nachvollziehen

Lösungen:

- `Cafe.asta`
- `/DesignPattern/src/com/priess/decoratorExample/`

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

Fassade (Fascade)

Prof. Dr. Malte Prieß

Fassade (Facade)	
Problem	Eine Komponente soll nicht den gesamten, sondern nur einen eingeschränkten Funktionsumfang anderer Komponenten kennen.
Beispiel	Eine Komponente zur statistischen Auswertung benötigt nur wenige Funktionen eines Kundeninformationssystems.

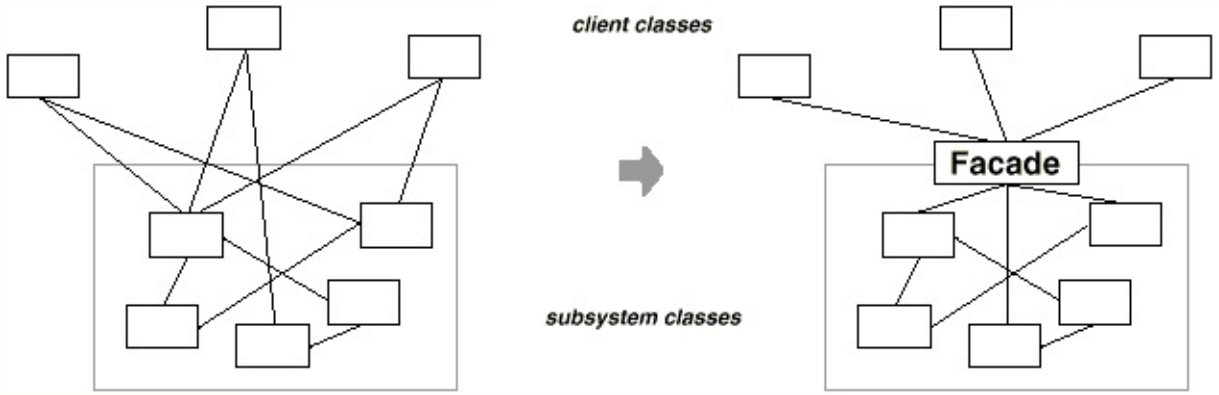
Kategorie

- Gamma et al.: **Strukturmuster**
- Ludewig, Lichter: Muster zur Entkopplung von Klassen

Quelle: Ludewig, Lichter, (2010), S.441, 442

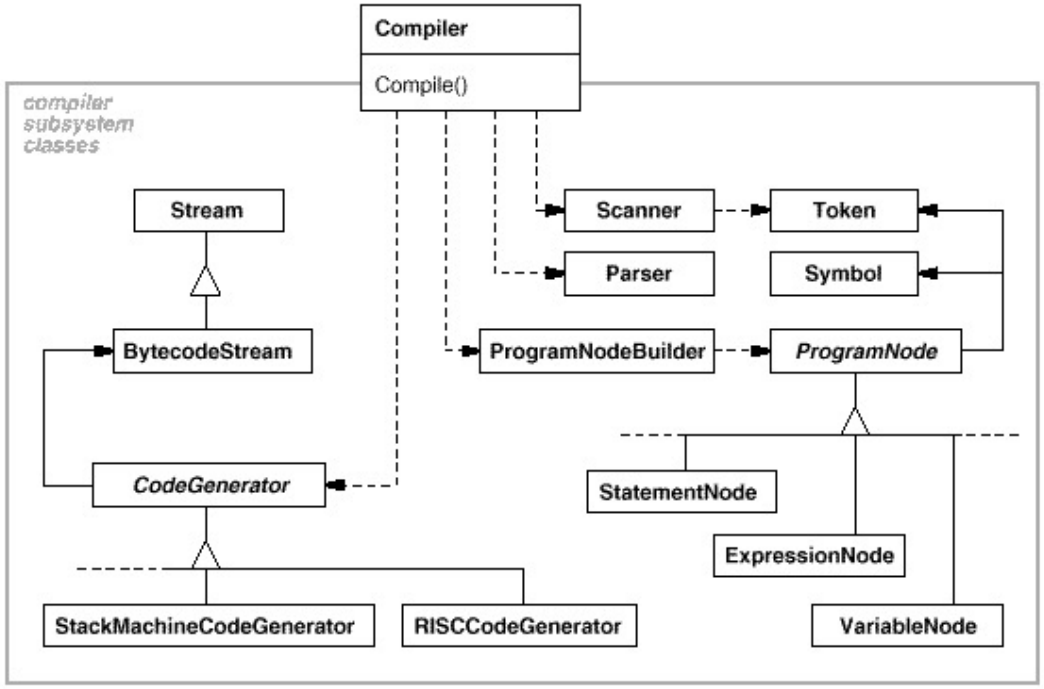
Strukturmuster Fassade (*Facade*)

- Konsolidieren der wichtigsten Schnittstellen eines Subsystems zu einer **einheitlichen übersichtlichen Schnittstelle**
- Bietet häufig auch **typische Aufruf-Sequenzen** von Subsystem-Operationen als **kombinierte Operation**
- **Weniger Abhängigkeiten** zwischen Subsystemen, wenn Zugriff ausschließlich über Fassade (→ Schichtenarchitektur)
- Bei Bedarf bleibt Durchgriff auf ursprüngliche Schnittstellen möglich



Beispiel: Compiler

- Compiler-Subsystem enthält Klassen *Scanner*, *Parser*, *CodeGenerator*, … jede mit diversen öffentlichen Operationen → API ist komplex
- Fassade kombiniert eine typische Aufruf-Sequenz der einzelnen Operationen in der Operation *Compile()*



Anwendung

- Wenn einfache Schnittstelle zu komplexem Subsystem hilfreich

Vorteile

- Clients verwenden nur 1 Fassaden-Objekt statt vieler Subsystem-Objekte
- Änderungen innerhalb des Subsystems wirken sich nicht auf Clients aus, wenn Zugriffe nur über Fassade erfolgen (Kapselung)
  → bessere Änderbarkeit

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

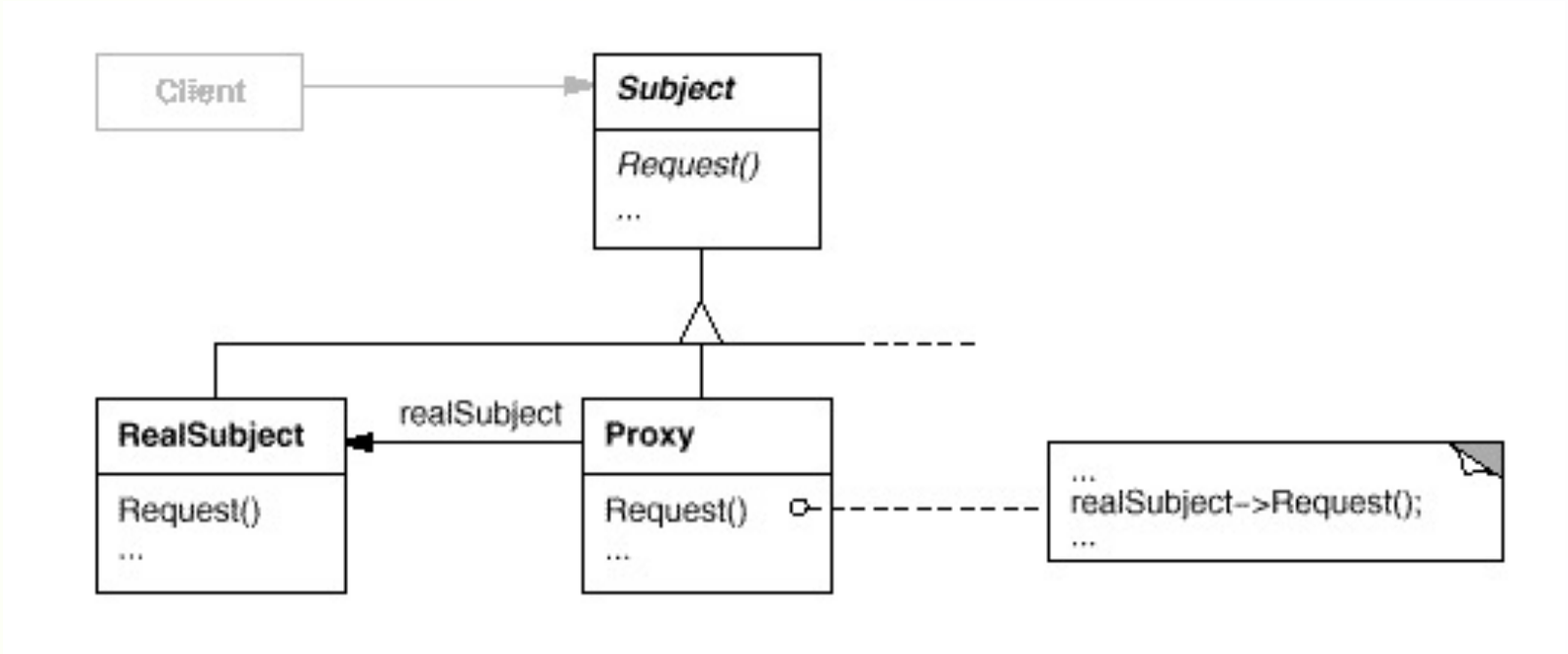
Entwurfsmuster

Proxy

Prof. Dr. Malte Prieß

Strukturmuster Proxy

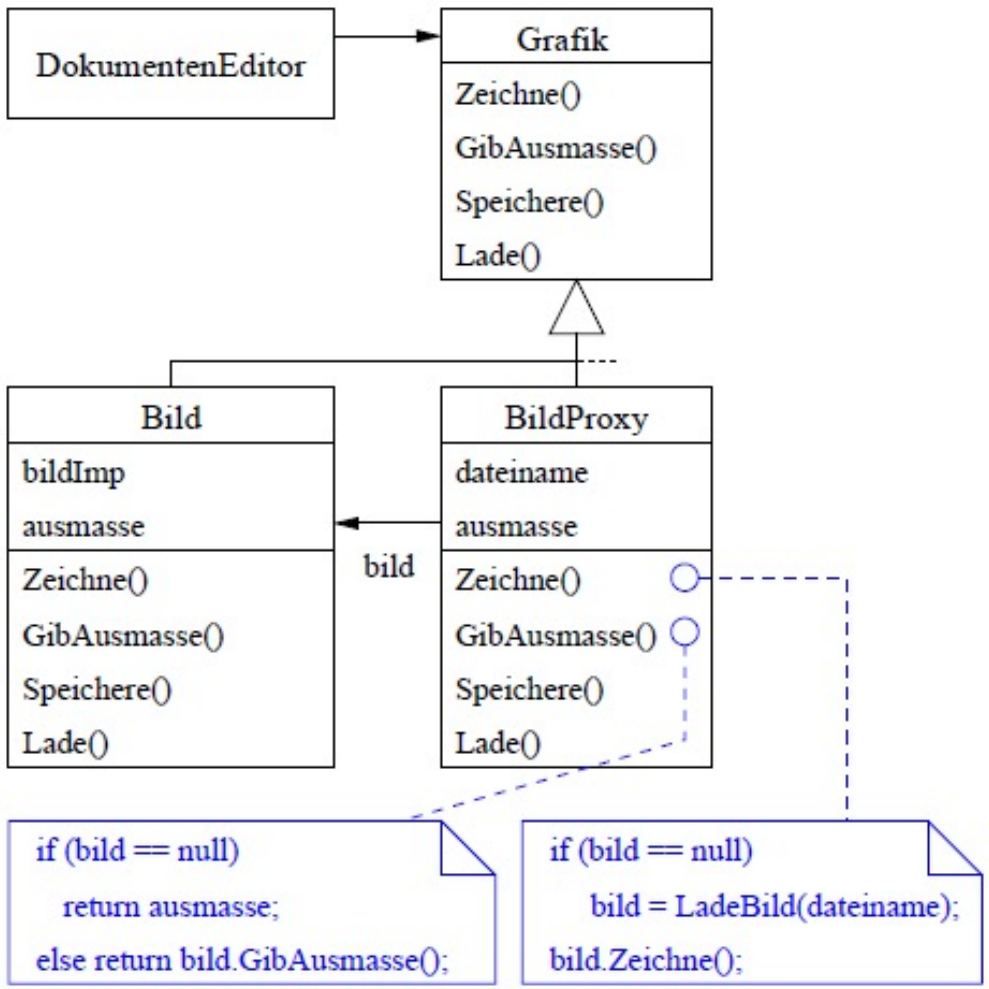
Zugriff auf Objekt wird durch vorgelagerten Stellvertreter gesteuert



Anwendung

1. Virtuelles Proxy
   - Erzeugt komplexe Objekte erst bei Bedarf, z.B. Laden von Bildern, Videos
   - Verzögert Aufwand der Initialisierung
2. Remote-Proxy
   - Verteilte Kommunikation: Proxy auf Client-Seite vertritt Server-Objekt, z.B. RMI, Web Services
3. Schutz-Proxy
   - Kontrolliert den Zugriff auf das ursprüngliche Objekt (Authentifizierung, Autorisierung)
4. Smart Reference
   - Delegation mit zusätzlichen Operationen, z.B. Reference-Counting, Sperren, Copy-on-Write

Beispiel: Darstellung von Bildern



DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Software-Engineering

Praktische Anwendung der Entwurfsmuster

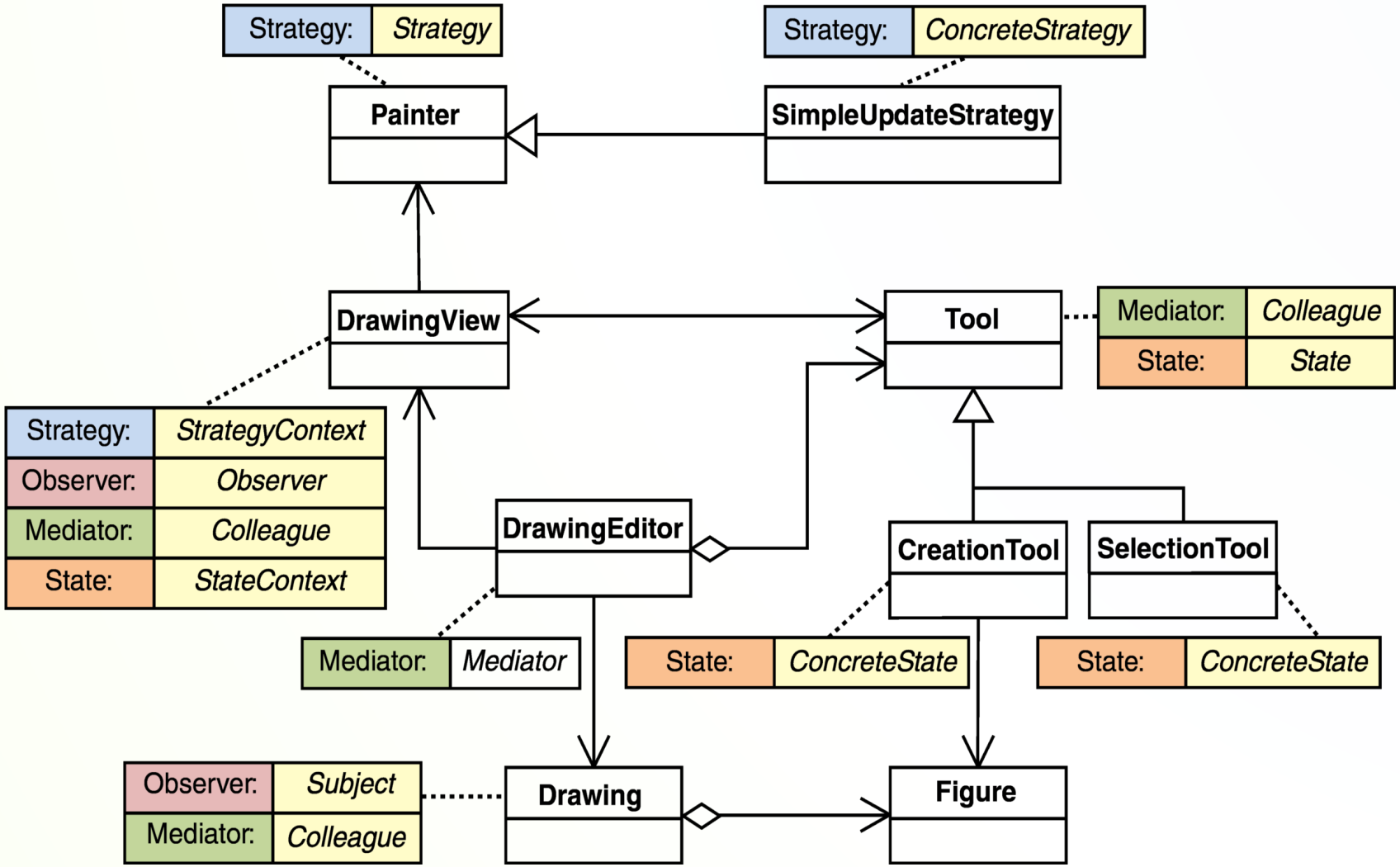
Prof. Dr. Malte Prieß

Anwendung am Beispiel

- JHotDraw-Rahmenwerk
  - Dieses objektorientierte Rahmenwerk definiert und implementiert die gemeinsame Architektur für **interaktive grafische Editoren**
- Anforderungen
  - Die Reaktion auf eine Benutzerinteraktion (z. B. einen Mausklick in der Zeichenfläche) muss sich **leicht ändern lassen**
  - Eine Grafik soll auf **beliebig vielen Zeichenflächen** darstellbar sein.
  - Der Algorithmus, der die Zeichenfläche nach einer grafischen Manipulation aktualisiert, soll **einfach austauschbar** sein

Quelle: Ludewig, Lichter, (2010)

Rollen im JHotDraw Rahmenwerk



Quelle: Ludewig, Lichter, (2010)

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Entwurfsmuster

Bewertung

Bewertung

- Die Entwurfsmuster geben uns die Möglichkeit, die **Erfahrungen** anderer zu nutzen und erprobte Lösungen einzusetzen
- Entwurfsmuster erfüllen fundamentale Prinzipien wie **Flexibilität**, **Generalität**, **Abstraktion**, **Wartbarkeit**
- Sie berücksichtigen bspw. die Prinzipien/Grundsätze **Separation of Concerncs**, **geringe Kopplung**, **hohe Kohäsion** und tragen damit zur **Modularisierung** bei
- Sie unterstützen uns, **nichtfunktionale Anforderungen**, beispielsweise Änderbarkeit oder Wiederverwendbarkeit, beim Architekturentwurf zu berücksichtigen
- Sie schaffen ein **Vokabular** für den Entwurf und erleichtern die Dokumentation und die Kommunikation über Architekturen
- Sie können beim **Reengineering** vorhandener Software als Hilfsmittel zur Analyse dienen

Aber:

Ein Entwurfsmuster zu verstehen ist relativ einfach. Man braucht jedoch viel Entwurfserfahrung, um die Muster sinnvoll einzusetzen.

Quelle: Ludewig, Lichter, (2010)

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

Software-Engineering

Gesammelte Entwurfsprinzipien

Prof. Dr. Malte Prieß

Gesammelte Entwurfsprinzipien

Curtis' law	Good designs require deep application domain knowledge.
Simon's law	Hierarchical structures reduce complexity.
Constantine' law	A structure is stable if cohesion is strong and coupling low.
Parnas' law	Only what is hidden can be changed without risk.
Denert's law	Separation of concerns leads to standard architectures.
DeRemer's law	What applies to small systems does not apply to large ones.
Dijkstra-Mills-Wirth law	Well structured programs have fewer errors and are easier to maintain.
Lanergan's law	The larger and more decentralized an organization, the more likely it is that it has reuse potential.
McIlroy's law	Software reuse reduces cycle time and increases productivity and quality.
Conway's law	A system reflects the organizational structure that built it.

Quelle: Endres, Rombach. A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories. Addison Wesley, 2003.

Gesammelte Entwurfsprinzipien

Bauer-Zemanek hypothesis	Formal methods significantly reduce design errors, or eliminate them early.
Beck-Fowler hypothesis	Agile programming methods reduce the impact of requirement changes.
Basili-Boehm COTS hypothesis	COTS-based software does not eliminate the key development risks.
Dahl-Goldberg hypothesis	Object oriented programming reduces errors and encourages reuse.
Booch' hypothesis	Object-oriented designs reduce errors and encourage reuse.
Gamma's hypothesis	Reusing designs through patterns yields faster and better maintenance.

Quelle: Endres, Rombach. A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories. Addison Wesley, 2003.

DUALE HOCHSCHULE SH

in Trägerschaft der Wirtschaftsakademie Schleswig-Holstein

★ Vielen Dank für die Aufmerksamkeit ★