

# Teil IV: Objektorientierter Entwurf

## 41 Grundlegende Architekturprinzipien

1

Prof. Dr. rer. nat. habil. Uwe  
Aßmann

Institut für Software- und  
Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

TU Dresden

Version 13-1.0, 08.07.13

- 1) Architekturprinzipien
- 2) Flexible Evolution mit Modularität und Geheimnisprinzip
- 3) Geschichtete Architekturen



# Obligatorische Literatur

- 2
- ▶ Zuser Kap 10.
  - ▶ Ghezzi 4.1-4.2
  - ▶ Pfleeger 5.1-5.3
  - ▶ ST für Einsteiger 5.3, 8

# Sekundäre Literatur

3

- ▶ David J. Parnas. On a buzzword: hierarchical structure. Proceedings IFIP Congress 1974, North-Holland, Amsterdam.
- ▶ Johannes Siedersleben. Moderne Softwarearchitektur. Umsichtig planen, robust bauen mit Quasar. dpunkt-Verlag, 2004.

# Teil IV - Objektorientierter Entwurf (Object-Oriented Design, OOD)

4

- 1) 41: Einführung in die objektorientierte Softwarearchitektur
  - 1) Modularität und Geheimnisprinzip
  - 2) Entwurfsmuster für Modularität
  - 3) BCD-Architekturstil (3-tier architectures)
- 2) 42: Verfeinerung des Entwurfsmodells zum Implementierungsmodell (Anreicherung von Klassendiagrammen)
- 3) 43: Verfeinerung von Lebenszyklen
  - 1) Verfeinerung von verschiedenen Steuerungsmaschinen
- 4) 44: Verfeinerung mit querschneidender Objektorichnung
- 5) 45 Wiederverwendung
  - 1) Objektorientierte Rahmenwerke (frameworks)
  - 2) Softwarearchitektur mit dem Quasar-Architekturstil

# 41.1. Herstellung Großer Softwaresysteme

5

*software*: computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.

(IEEE Standard Glossary of Software Engineering)

# Was heißt hier "groß"?

6

- ▶ Klassifikation nach W. Hesse:

Klasse	Anzahl Code-Zeilen	Personenjahre zur Entwicklung
sehr klein	bis 1.000	bis 0,2
klein	1.000 - 10.000	0,2 - 2
mittel	10.000 - 100.000	2 - 20
groß	100.000 - 1 Mio.	20 - 200
sehr groß	über 1 Mio.	über 200

# Riesige Systeme

7

- ▶ Telefonvermittlungssoftware EWSD (Version 8.1):
  - 12,5 Mio. Code-Zeilen
  - ca. 6000 Personenjahre
- ▶ ERP-Software SAP R/3 (Version 4.0)
  - ca. 50 Mio. Code-Zeilen
- ▶ Umfang der verwendeten Software (Anfang 2000):
  - Credit Suisse 25 Mio. Code-Zeilen
  - Chase Manhattan Bank: 200 Mio. Code-Zeilen
  - Citicorp Bank: 400 Mio. Code-Zeilen
  - AT&T: 500 Mio. Code-Zeilen
  - General Motors: 2 Mrd. Code-Zeilen

## Abkürzungen:

EWSD = Elektronisches Wählsystem Digital (Siemens-Projekt)

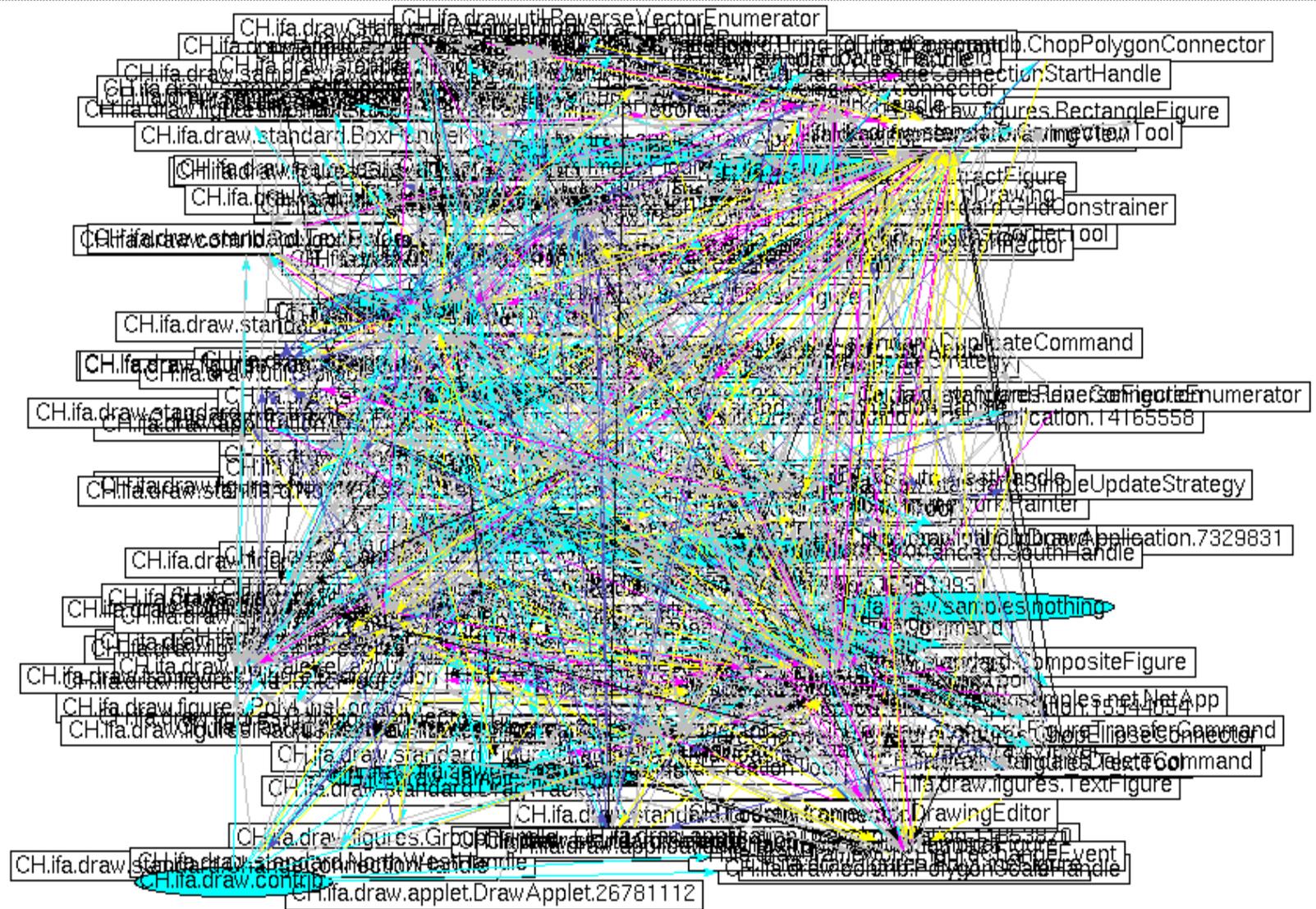
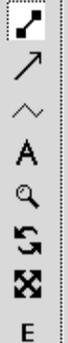
ERP = Enterprise Resource Planning

SAP: Deutscher Software-Konzern

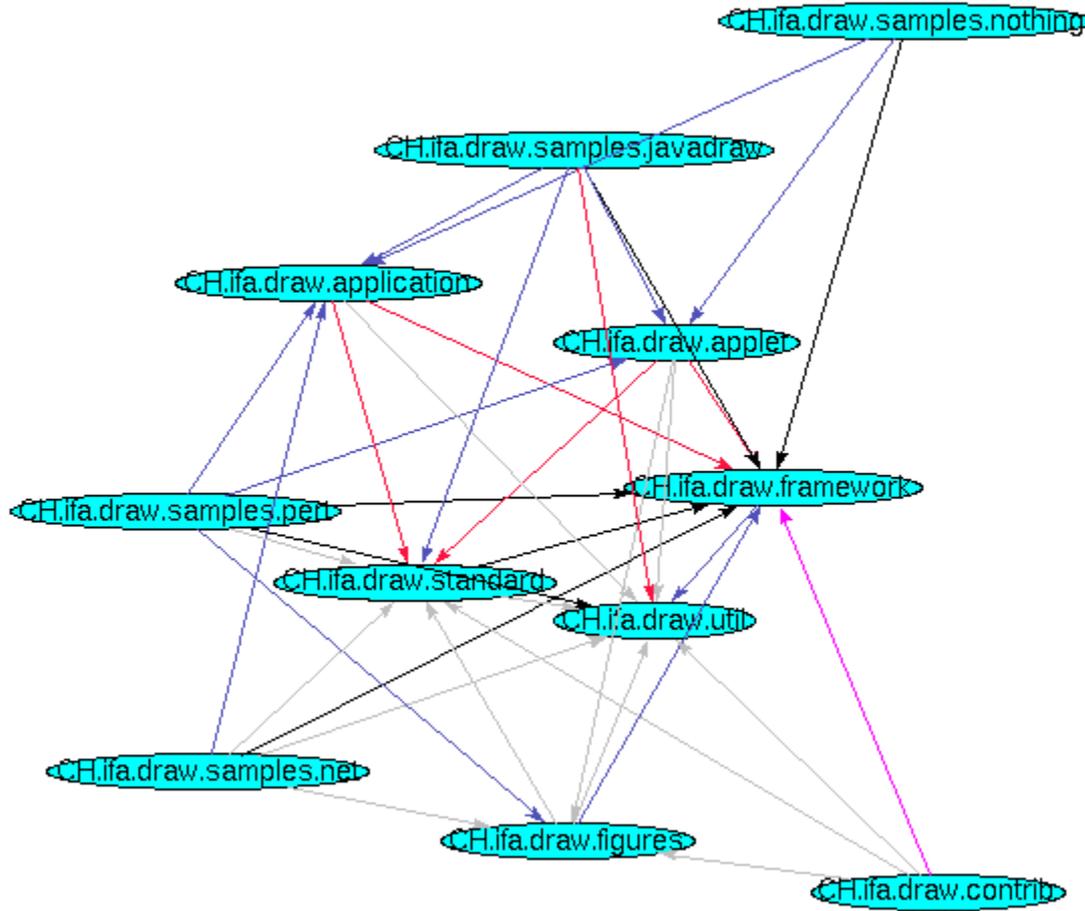
# Strukturprobleme

9

- ▶ The following figures are taken from the Goose Reengineering Tool, analysing a Java class system [Goose, FZI Karlsruhe, <http://www.fzi.de>]







# Softwarearchitektur

14

- ▶ Softwarearchitektur ist der Schlüssel zum Erfolg des Softwareingenieurs und seiner Firma.

Ohne gute Softwarearchitektur keine  
Wiederverwendung, Evolution, Variabilität,  
Erweiterbarkeit

Mit guter Softwarearchitektur  
Softwareproduktlinien, schnell erstellte neue Produkte,  
vertikale Portierung auf andere Domänen,  
einfaches Dienstleistungsgeschäft.



# 41.2 Grundlegende Architekturprinzipien

---

---

15

# Wesentliche Aspekte und Bestandteile eines Softwaresystems

16

- ▶ Anwendungsspezifische Funktionen
- ▶ Benutzungsoberfläche
- ▶ Ablaufsteuerung
- ▶ Datenhaltung
- ▶ Infrastrukturdienste
  - Objektverwaltung
  - Interne Objekt- und Prozeßkommunikation
  - Verteilungsunterstützung
- ▶ Kommunikationsdienste
- ▶ Sicherheitsfunktionen
- ▶ Zuverlässigkeitsfunktionen
- ▶ Systemadministration
  - Installation, Anpassung
  - Systembeobachtung
- Vertragsprüfung
- Etc.

Aspekte

Anwendung  
(Essenz, spezifisch)

Architektur

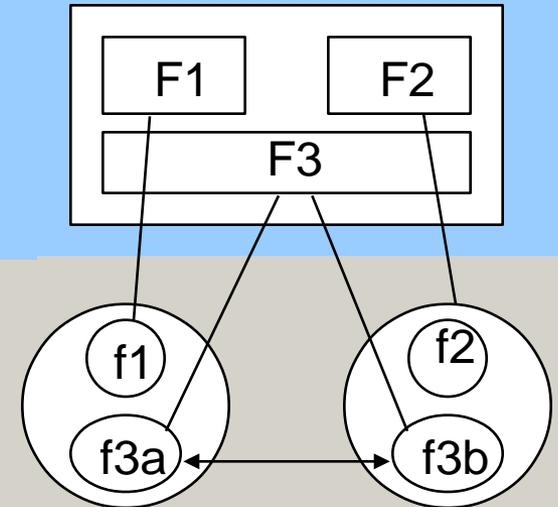
Technische  
Infrastruktur

Konsistenz-  
Administration

# Aspekte des Architekturentwurfs

17

- ▶ Strukturelle Zerlegung:
  - Blockdiagramme, Montagediagramme (UML-Komponentendiagramme)
  - Architekturstil: Schichten, Sichten, Dimensionen
- ▶ Struktur der physikalischen Verteilung:
  - Zentral oder verteilt?
  - Topologie

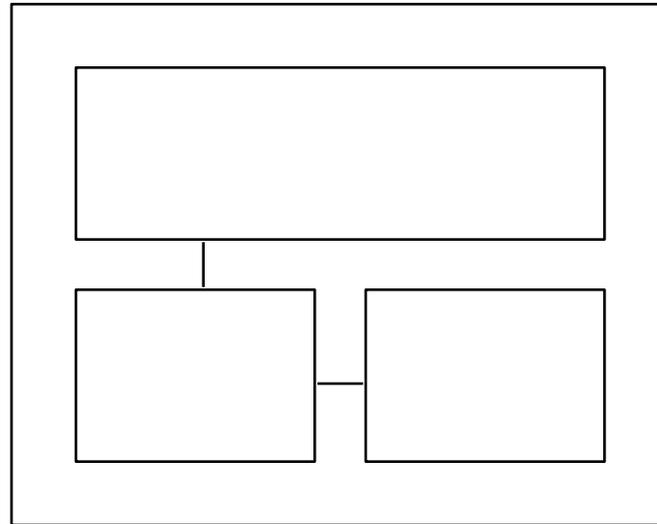


- ▶ Ablaufsicht
- ▶ Logischer Detail-Entwurf
- ▶ Einhaltung nichtfunktionaler Anforderungen:
  - Architekturbestimmende Eigenschaften (z.B. Realzeitsystem, eingebettetes System)
  - Effizienzanforderungen und Optimierung
  - Standardarchitekturen

# Blockdiagramme

18

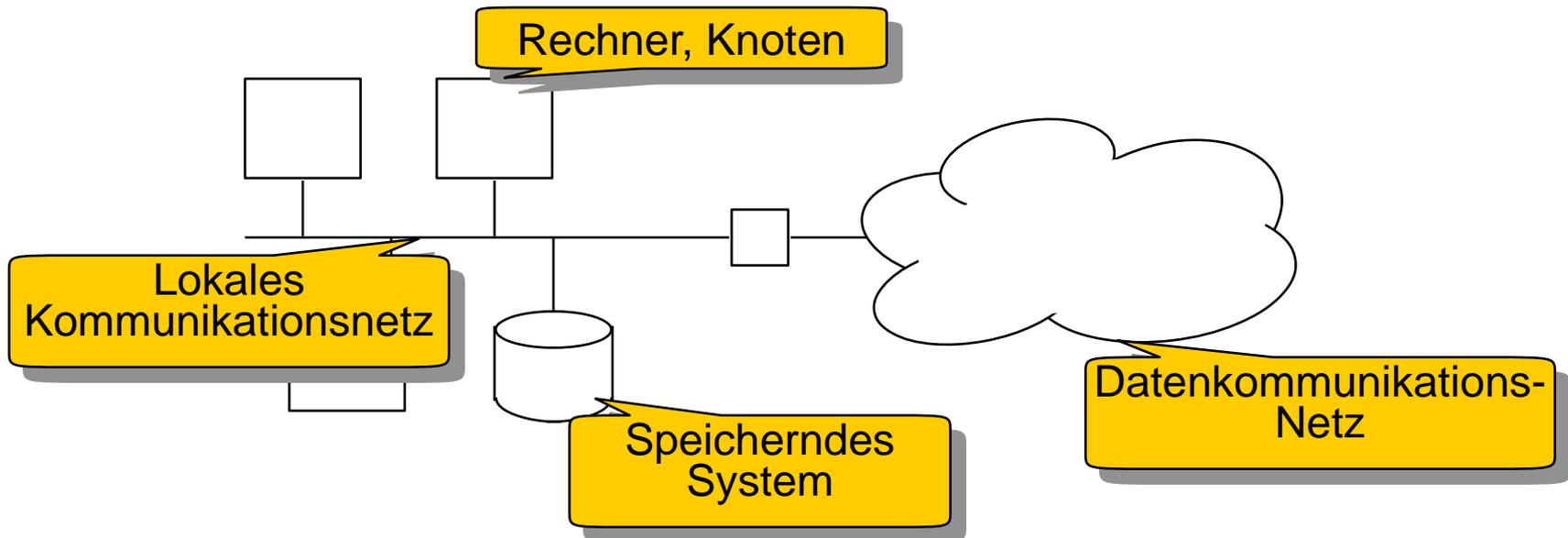
- ▶ **Blockdiagramme** sind das meistverbreitete, informelle Hilfsmittel zum Skizzieren der logischen **Struktur** einer Systemarchitektur.
  - Blockdiagramme sind kein Bestandteil von UML
  - **Blöcke** stellen UML-Komponenten *ohne* Anschlüsse dar



# Konfigurationsdiagramme

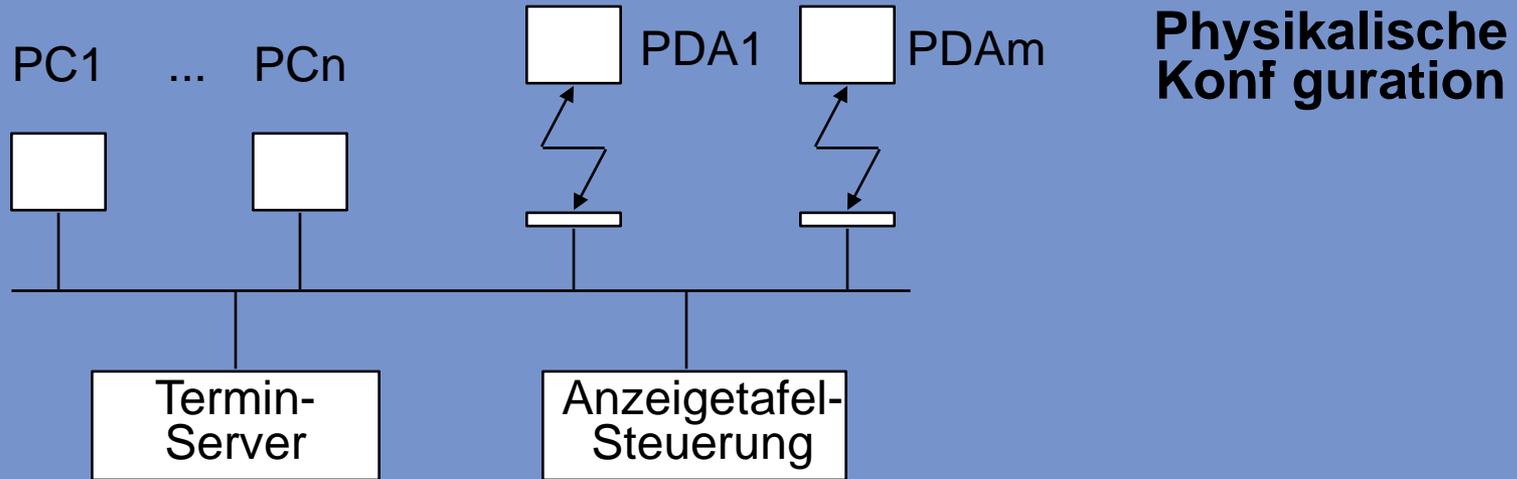
19

- ▶ Konfigurationsdiagramme sind Blockdiagramme mit “Bussen”
  - Konfigurationsdiagramme sind nicht Bestandteil von UML
  - ein verbreitetes Hilfsmittel zur Beschreibung der physikalischen Verteilung

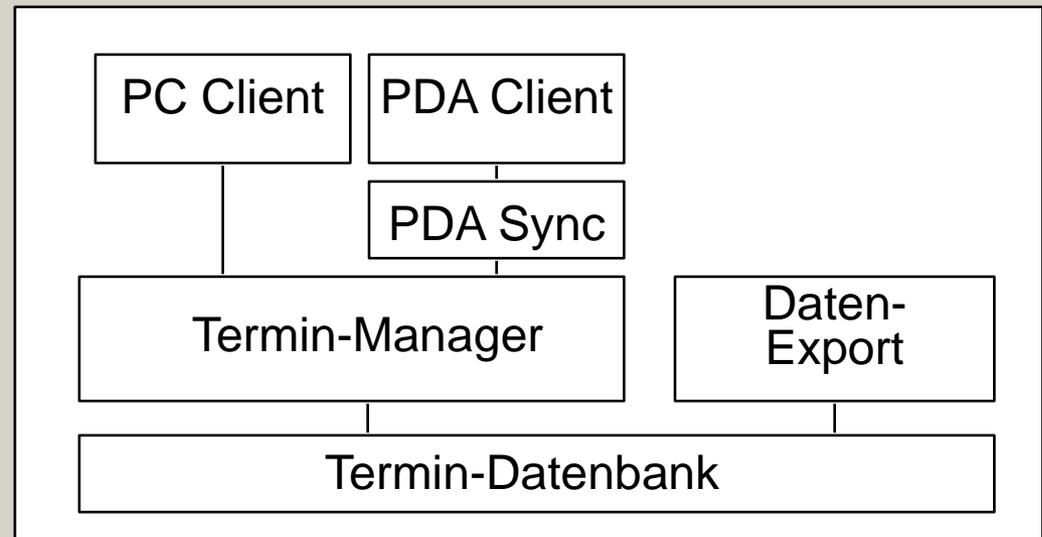


# Beispiel: Konfigurationsdiagramm für Terminverwaltung

20



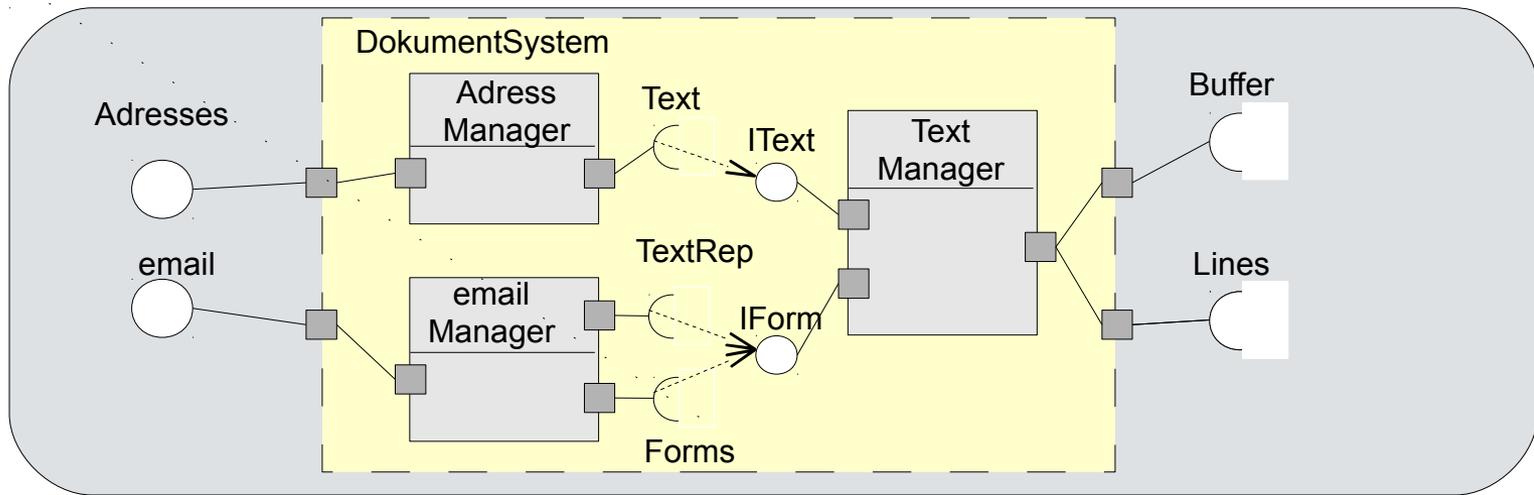
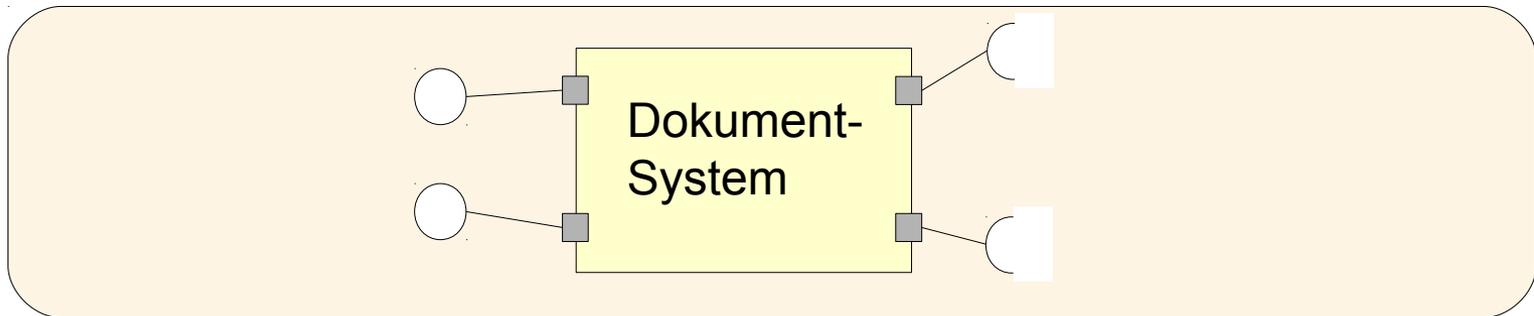
## Blockdiagramm



# Architekturstil für nicht-Interaktive Anwendungen: Hierarchische Organisation der obersten Ebenen

21

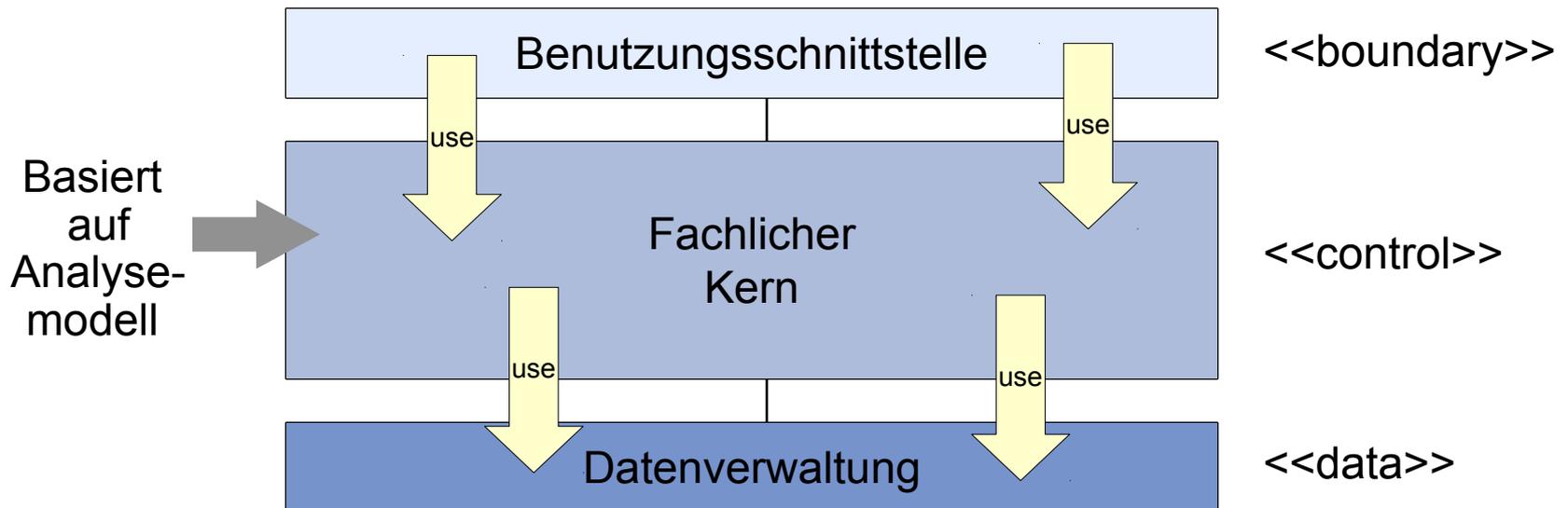
- ▶ Oberste Ebene des Systems ist meist hierarchisch und/oder geschichtet organisiert
  - Vermeide “wilde” objekt-orientierte Netzstrukturen
  - Damit die letzte Integration zum Gesamtsystem einfach verläuft: Integrationstests können dann bottom-up absolviert werden
- Hierarchien bilden Spezialfälle, denn sie können geschichtet werden



# Architekturstil für Interaktive Anwendungen: Drei-Schichten-Architektur (BCD)

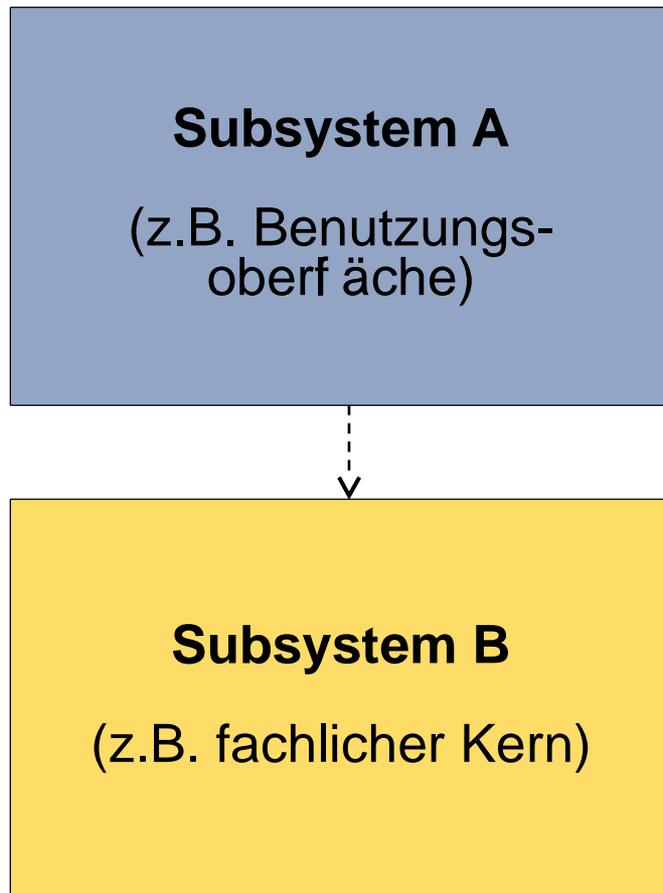
22

- ▶ Klassische Struktur eines interaktiven Anwendungssystems
- ▶ Integrationstest verläuft wegen azyklischer Benutzungsrelation (use) bottom-up: erst D, dann CD, dann BCD
- ▶ Fachlicher Kern (Anwendungslogik) kann weitere Schichten enthalten
  - Oft kapselt eine Facade eine Schicht nach oben ab, dann existieren bereits zwei Teil-Schichten



# Architekturprinzip: Hohe Kohäsion + Niedrige Kopplung

23

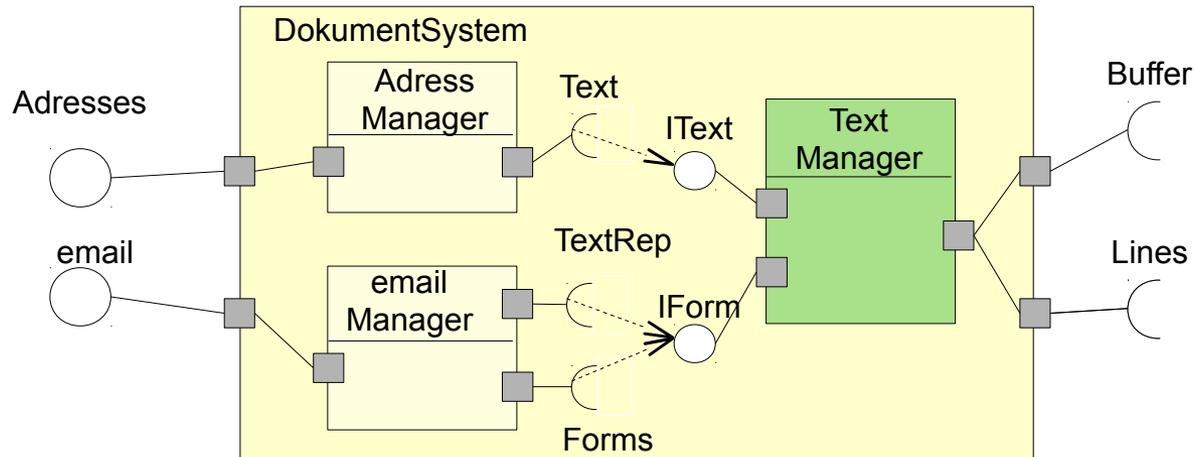


- ▶ **Hohe Kohäsion:**  
Subsystem B darf keine Information und Funktionalität enthalten, die zum Zuständigkeitsbereich von A gehört und umgekehrt.
- ▶ **Niedrige Kopplung:**  
Es muß möglich sein, Subsystem A weitgehend auszutauschen oder zu verändern, ohne Subsystem B zu verändern.  
  
Änderungen von Subsystem B sollten nur möglichst einfache Änderungen in Subsystem A nach sich ziehen.

# Architekturprinzip Quasar: Trennung von Technik- und Anwendungskomponenten

24

- ▶ Jede Komponente wird klassifiziert in Blutgruppen (0 – technologieunabhängige Algorithmen, T – technologieabh. Komponenten, A- Anwendungskomponenten)



**[Siedersleben] Quasar-Wiederverwendungsgesetz:  
0- und T-Komponenten sind besser  
wiederverwendbar als Anwendungskomponenten.**

# 41.2 Architekturprinzip: Veränderungsorientierter Entwurf mit dem Modul-Geheimnis

25



# Architekturprinzip: Einteilung in Komponenten (Module)

26

- ▶ Nach dem Teile-und-Herrsche-Prinzip sollte Software in *Komponenten (Module)* eingeteilt werden
- ▶ Eine **Komponente** im allgemeinden Sinne ist eine Wiederverwendungseinheit:
  - gruppiert Funktionalität und erzeugt Kohäsion
  - unterstützt lose Kopplung:
    - hat keine impliziten, nur explizit in der Schnittstelle angegebene Abhängigkeiten zu anderen Komponenten
    - kann unabhängig von anderen entwickelt werden
    - Komponenten können einzeln getestet werden (Einheitstest, unit test)
    - Fehler können zu individuellen Komponenten verfolgt werden
  - kann ausgetauscht werden, ohne dass das System zusammenbricht (Ersetzbarkeit)
  - kann wiederverwendet werden, weil angebotene und benötigte Schnittstellen unterschieden werden
- **Module** werden hier als Komponenten bezeichnet, die in binäre Form übersetzt werden können

# Bemerk.: Komponentenmodelle und Kompositionssysteme

27

- ▶ Es gibt nicht nur die UML-Komponente....
- ▶ sondern viele verschiedene *Komponentenmodelle*:
  - Module einer modularen Programmiersprache (Modula, Ada)
  - Klassen, Kollaborationen und Konnektoren in objektorientierten Sprachen
  - UML-Komponenten
  - Fragmentkomponenten, Schablonen (templates)
  - Dokumentkomponenten
  - Serverseitige Webkomponenten
- ▶ Ein *Kompositionssystem* definiert:
  - **Komponentenmodell**: Eigenschaften der Schnittstelle einer Komponente
  - **Kompositionstechnik**: Wie werden Komponenten komponiert?
  - **Kompositionssprache**: Wie wird die Architektur eines großen Systems beschrieben?
- ▶ --> Vorlesung CBSE (SS)

# Architekturprinzip: Flexible Evolution mit dem Geheimnisprinzip

28

Parnas' Prinzip des Entwurfs mit dem **Geheimnisprinzip**

(veränderungsorientierter Entwurf, *change-oriented modularization with information hiding*) [Parnas, CACM 1972]:

- 1) Bestimme alle Entwurfsfragen (-alternativen), die sich *ändern können*
- 2) Entwickle für jede Entwurfsfrage eine Komponente, die die Entscheidung bezüglich der Frage verbirgt
  - ▶ Die Entscheidung nennt man das **Komponenten-** oder **Modulgeheimnis (module secret)**
- 3) Entwerfe eine stabile Schnittstelle für die Komponente, die unverändert bleibt, wenn sich die Entwurfsentscheidung und somit die Implementierung des Modulgeheimnisses ändert

Das Geheimnisprinzip ermöglicht Austausch von Implementierungen hinter Schnittstellen und somit flexible Evolution

Das Geheimnisprinzip erniedrigt die externe Kopplung und erhöht die innere Kohäsion von Komponenten und Modulen

# Typische Geheimnisse von Modulen/Komponenten

29

- ▶ Arbeitsweise von Algorithmen
- ▶ Datenformate
  - Texte, Dokumente, Bilder
- ▶ Datentypen
  - Abstrakte Datentypen und ihre konkrete Implementierung
- ▶ Benutzerschnittstellenbibliotheken
- ▶ Bearbeitungsreihenfolgen
- ▶ Verteilung
- ▶ Persistenz
- ▶ Parallelität

# Verschiedene Arten von Komponenten/Modulen

30

- ▶ Funktionale Module ohne Zustand
  - sin, cos, BCD arithmetic, gnu mp,...
- ▶ Daten-Repositoryen
  - Verbergen Repräsentation, Zugriff und Zustand der Daten
  - Symboltabellen, Materialcontainer, ...
- ▶ Abstrakte Datentypen
- ▶ Singletons (Konfigurationskomponenten)
  - Klassen mit einer einzigen Instanz
- ▶ Prozesse (aktive Objekte)
- ▶ Klassen
  - Module, die ausgeprägt werden können
- ▶ Generische Klassen (Klassenschablonen)
- ▶ Komplexe Klassen (UML-Komponenten)
- ▶ Fragmentkomponenten

... für alle gilt das Geheimnisprinzip

# Variabilitätsmuster nutzen das Geheimnisprinzip

31

- ▶ Viele Entwurfsmuster (z.B. TemplateMethod) sind *Variabilitätsmuster*, d.h., sie lassen einem bestimmte Geheimnisse verbergen und dann die Implementierungen austauschen (variieren)
  - Fassade verbirgt ein ganzes Subsystem
  - Fabrikmethode verbirgt die Allokation von Produkten
  - TemplateMethod und Strategie verbergen einen Anteil eines Algorithmus
  - Singleton kapselt globale Konfigurationsdaten
- ▶ In UML kann man Entwurfsmuster als Komponenten (Wiederverwendungseinheiten) kapseln, indem man sie als Kollaborationen spezifiziert

# Was wollen wir eigentlich variieren?

32

- ▶ **Software-Produktlinien** entstehen durch systematische Variation von Geheimnissen

Dynamische Kontexteigenschaften

Benutzer (Level,  
Präferenzen)  
Netzwerkverbindung  
Sicherheitseigenschaften

Funktionale Eigenschaften  
(Funktionale Varianten in der Control)

Mailer            Writer  
Calendar        SlideEditor  
Browser          Wiki

MiddlewareVarianten  
(Entity level)

Parallelität (Transaktionen)  
Verteilung

Plattformeigenschaften  
(Boundary, Database)

Chip, OS, GUI-Bibliothek  
Persistenz, Displaygerät  
Web/Rich/Smart Client

# 41.3 Architekturprinzip Schichtung (Layered Architectural Styles)

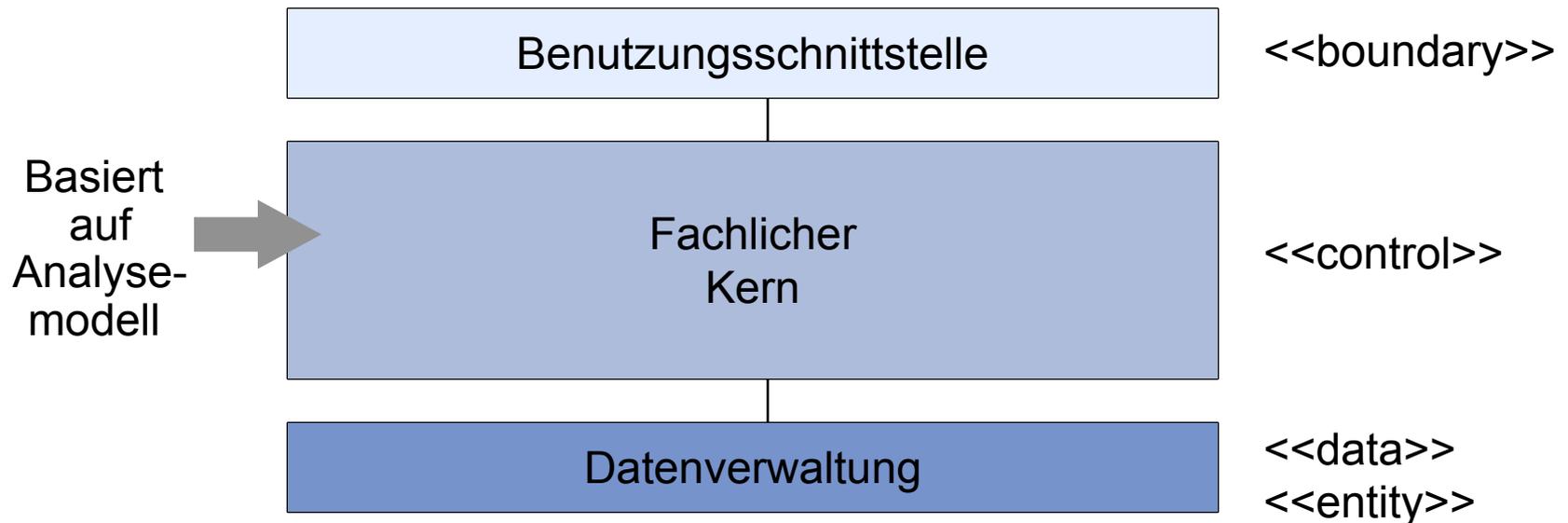
33

und die "benutzt"-Relation

# Drei-Schichten-Architekturstil (BCD) für interaktive Anwendungen

34

- ▶ Klassische Struktur eines interaktiven Anwendungssystems
- ▶ Schichten sind jeweils stark kohäsiv, und lose gekoppelt – warum?
- ▶ Oft kapselt eine Fassade eine Schicht, ein Einzelstück konfiguriert jede Schicht, Fabriken schneiden die Produkte der unteren Schichten zu, TemplateMethod/Class variieren Algorithmen der Produkte



# Verschiedene Relationen zwischen Komponenten

35

- ▶ Es gibt verschiedene Beziehungen zwischen den Komponenten eines Systems
- ▶ Ähnlichkeit
  - Vererbungsrelationen: is-a (set inheritance), behaves-like (Verhaltenskonformität), ...
- ▶ Zugriff
  - accesses-a (access relation)
  - Zugriffsrecht: is-privileged-to, owns-a (security)
  - Aufrufe: calls
    - is-called-by
    - delegates-to (delegation)
  - Senden und Empfangen von Nachrichten
- ▶ Die “Relies-On” Relation fasst alle diese zusammen

# Vertraut-auf-Relation (Relies-On, USES, Sees-A)

36

Komponente A **vertraut auf (relies-on, USES)** Komponente B  
gdw.  
A benötigt eine korrekte Implementierung von B für seine eigene  
korrekte Ausführung [Parnas]

- ▶ *benötigt eine korrekte Implementierung* beinhaltet:
  - A greift zu auf öffentliche Variable oder Objekt von B
  - A nutzt eine Ressource von B
  - A alloziert ein Objekt von B
  - A delegiert Arbeit auf B (A ruft auf B) or B delegiert Arbeit zurück auf A
  - A initiiert B durch Auslösen einer Ausnahme oder Ereignis

Ein Softwaresystem heißt **hierarchisch**, falls seine Komponenten eine  
hierarchische „vertraut-auf“-Relation besitzen  
Ein Softwaresystem heißt **geschichtet**, falls seine Komponenten eine  
geschichtete „vertraut-auf“-Relation besitzen

# Testen von hierarchischen und geschichteten Systemen

37

In einem hierarchischen oder geschichteten System erfolgt der Test bottom-up, d.h. aufwärts entlang der USES-Relation.

# Example: USES Relation in 3- and 4-Tier Architectures (BCED)

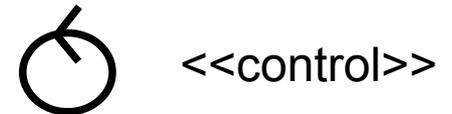
38

- ▶ 3- and 4-tier architectures have an acyclic USES relation, divided into 3 (resp. 4) layers that use each other in an acyclic relationship
  - Upper layers see lower layers, but not vice versa

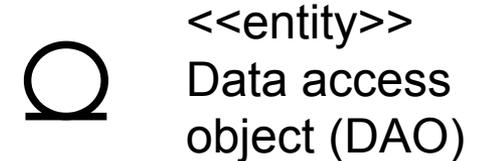
**Graphical user interface (GUI, Benutzerschnittstelle)**



**Application logic (business logic, Fachlicher Kern, Anwendungslogik)**



**Middleware (memory access, distribution)**



**Data Repository Layer (database, memory)**



# Example: 3- and 4-Tier Architectures (BCD/BCED)

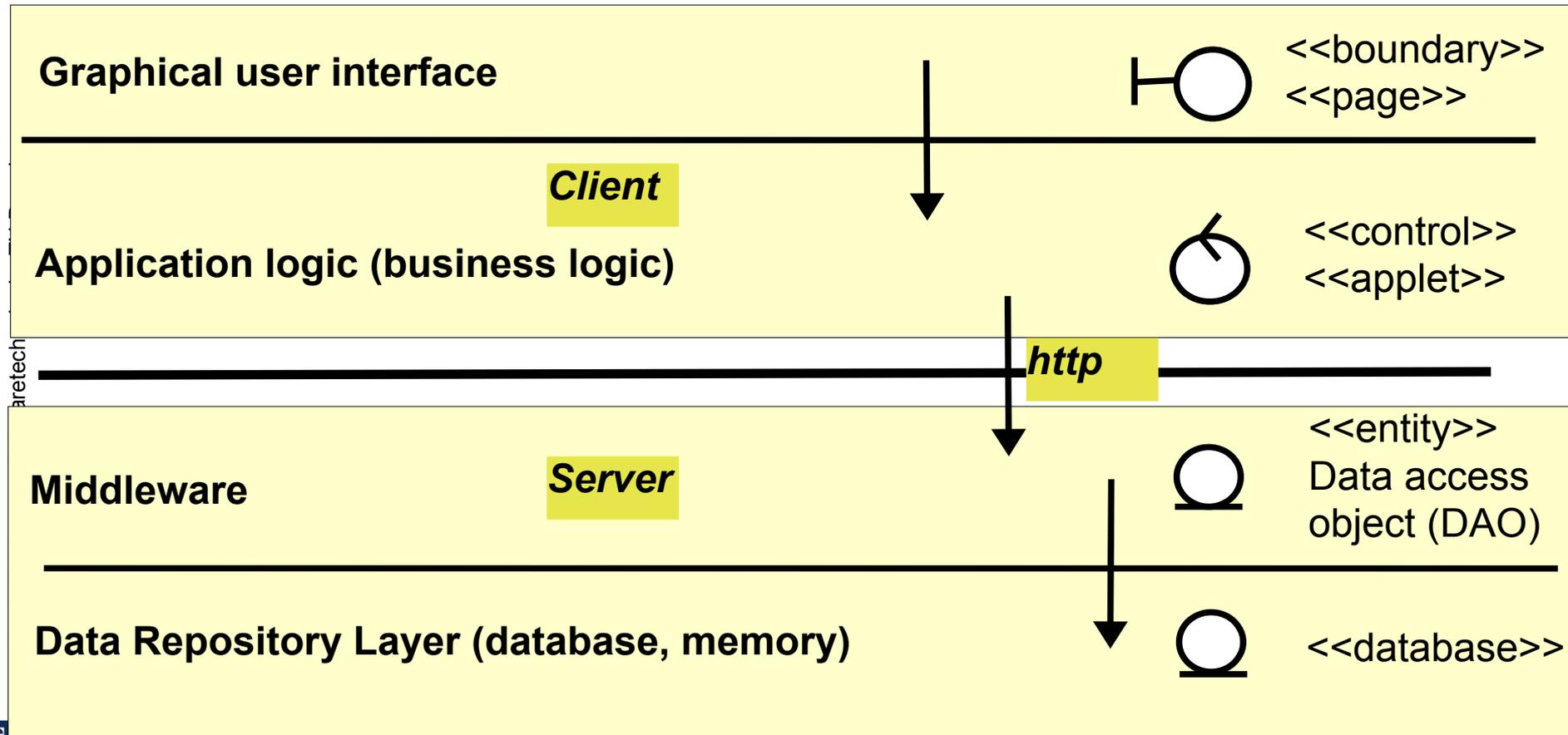
39

- ▶ Good encapsulation of cohesive knowledge in a layer
- ▶ Low coupling due to acyclic USES relationship
- ▶ Better exchange of subsystems of the application
  - GUI encapsulates user interactions and look
  - Data repository layer encapsulates how data is stored (database, transient, persistent component platforms such as Enterprise JavaBeans)
  - Middleware mediates between both
    - The middleware hides distribution
    - and deals with security
- ▶ The BCD/BCED architecture is *the* architecture for business-oriented software
- ▶ ... and for projects in the projects ...

# Example: 4-Tier Web System (Thick Client)

40

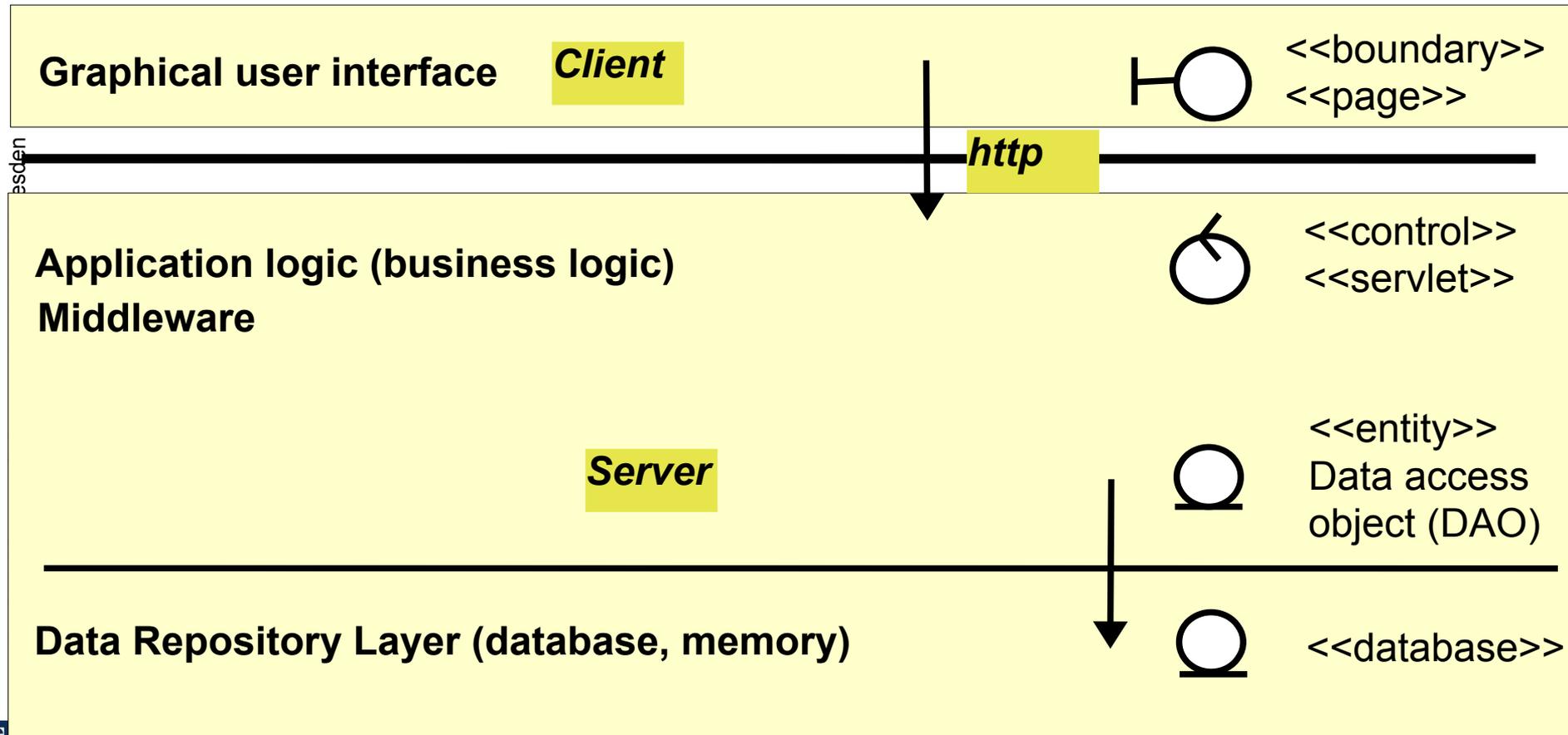
- ▶ Thick client Web Systems have a http-based middleware, in which GUI and application logic reside on the client, data is managed on the server



# Example: 4-Tier Web System (Thin Client)

41

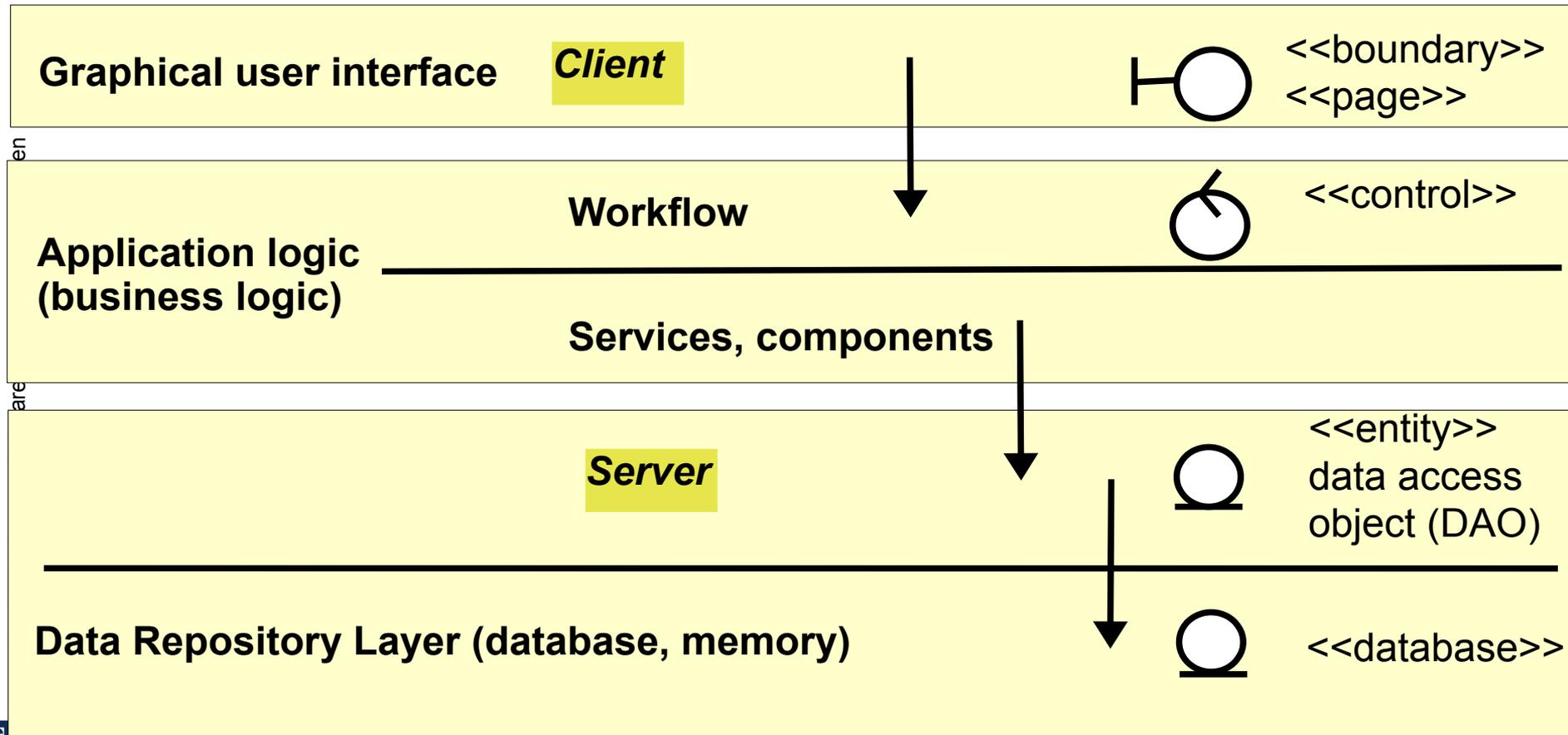
- ▶ Thin client Web Systems have a http-based middleware, in which GUI resides on the client, application logic and data is managed on the server



# Example: 5-Tier with Workflow Language

42

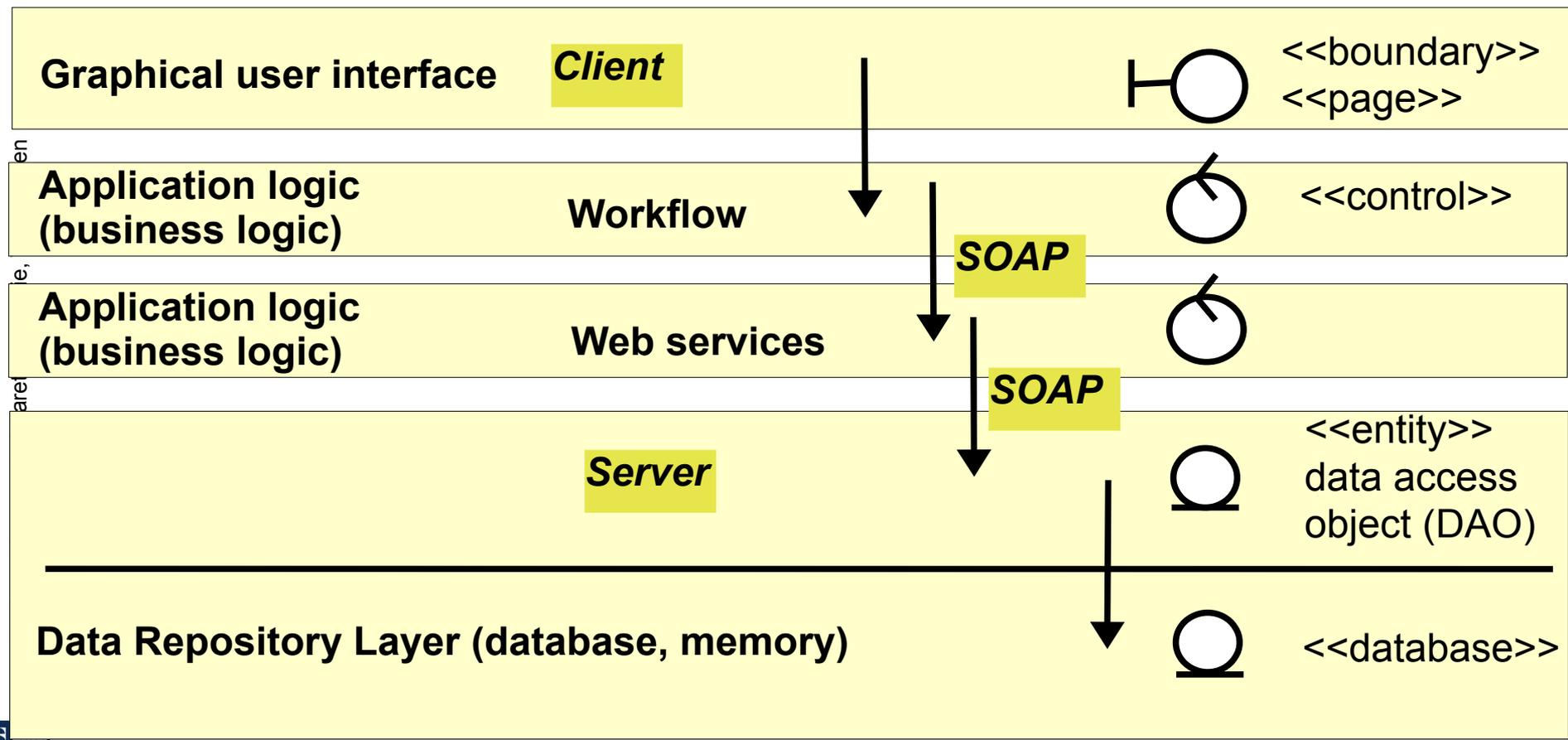
- ▶ Workflow languages (BPMN, BPEL) describe the top-level of the application architecture
  - Services and components are called by the workflow



# Example: 5-Tier with Workflow Language and Web Services

43

- ▶ Workflow languages (BPMN, BPEL) describe the top-level of the application architecture
  - Services and components are called by the workflow via SOAP protocol



# Why are Layered Architectures Successful?

44

- ▶ Layered architectures require an acyclic USES relationship
- ▶ They are successful,
  - Because the dependencies within the system are structured as a dag
  - System is structured
  - Internals of layers can be abstracted away

# What Have We Learned

45

- ▶ Designing the global *architectural style* of your application is important (Architekturentwurf)
- ▶ Layers play an important role
- ▶ The USES (relies-on) relation is different from is-a (inheritance) and part-of (aggregation)
  - It deals with prerequisites for correct execution
  - Can be used to layer systems, if it is acyclic
- ▶ Examples of architectural styles with acyclic USES relation:
  - The BCED 4-tier architecture
  - Layered abstract machines for interactive applications
  - Layered behavioral state machines
  - Both styles can be combined

# Conway's Law on Software Structure

46

Software is always structured in the same way as the organisation which built it.



# The End



# Anhang 41.A Entwurfsmuster Fassade zur Reduktion von Kopplung

---

48

... Ein Entwurfsmuster im Geiste Parnas  
(Wdh.)

# Entwurfsmuster Fassade (Facade)

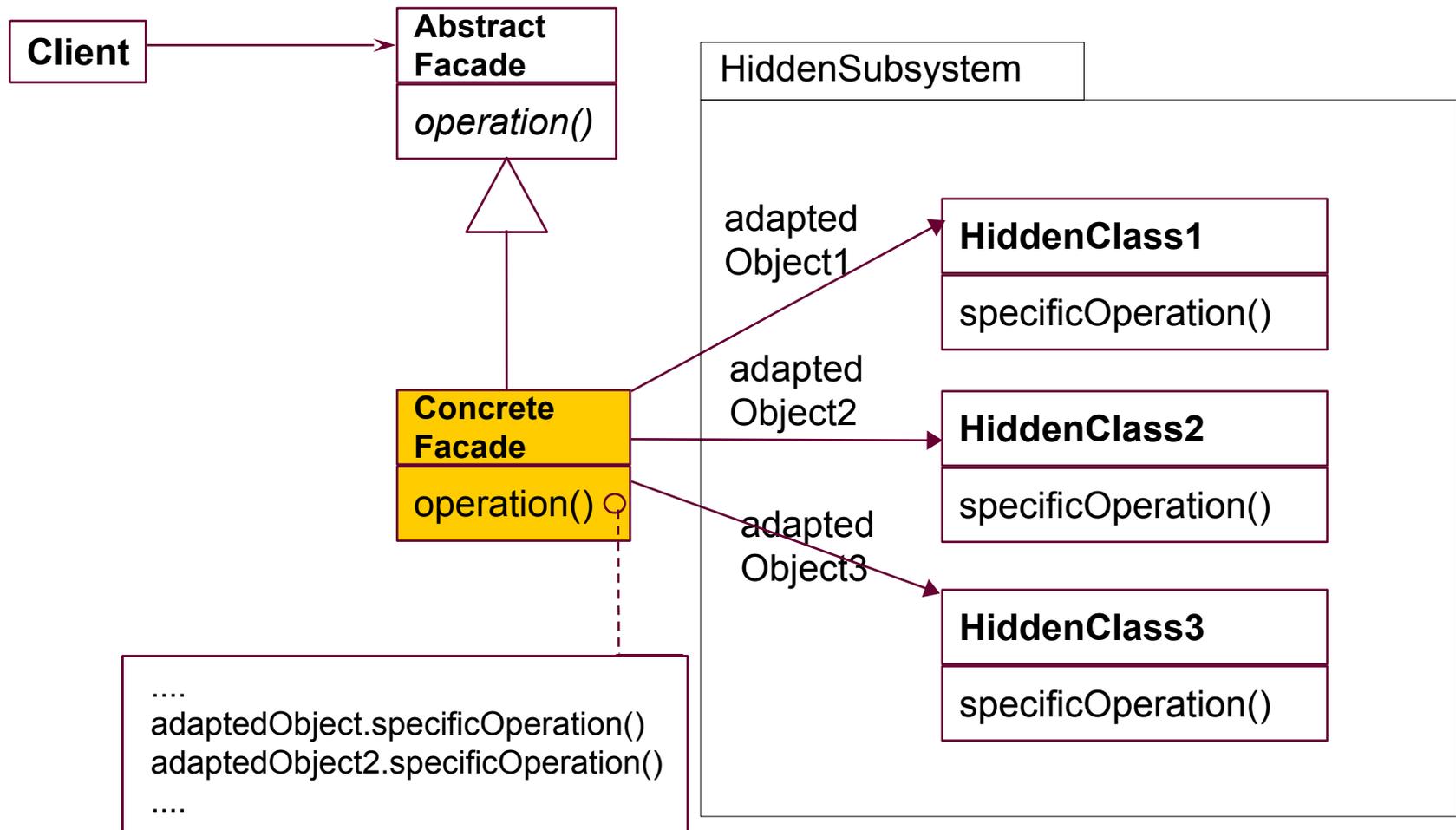
49

- ▶ Eine *Fassade (Facade)* ist ein Objektadapter, der ein komplettes Subsystem verbirgt
  - Die Fassade bildet die eigene Schnittstelle auf die Schnittstellen der verkapselten Objekte ab
  - Eine UML-Komponente ist gleichzeitig eine (einfache) Fassade. Die Delegationskonnektoren werden 1:1 an innere Komponenten delegiert; interne Adapter können adaptieren

# Fassaden verbergen Subsysteme

50

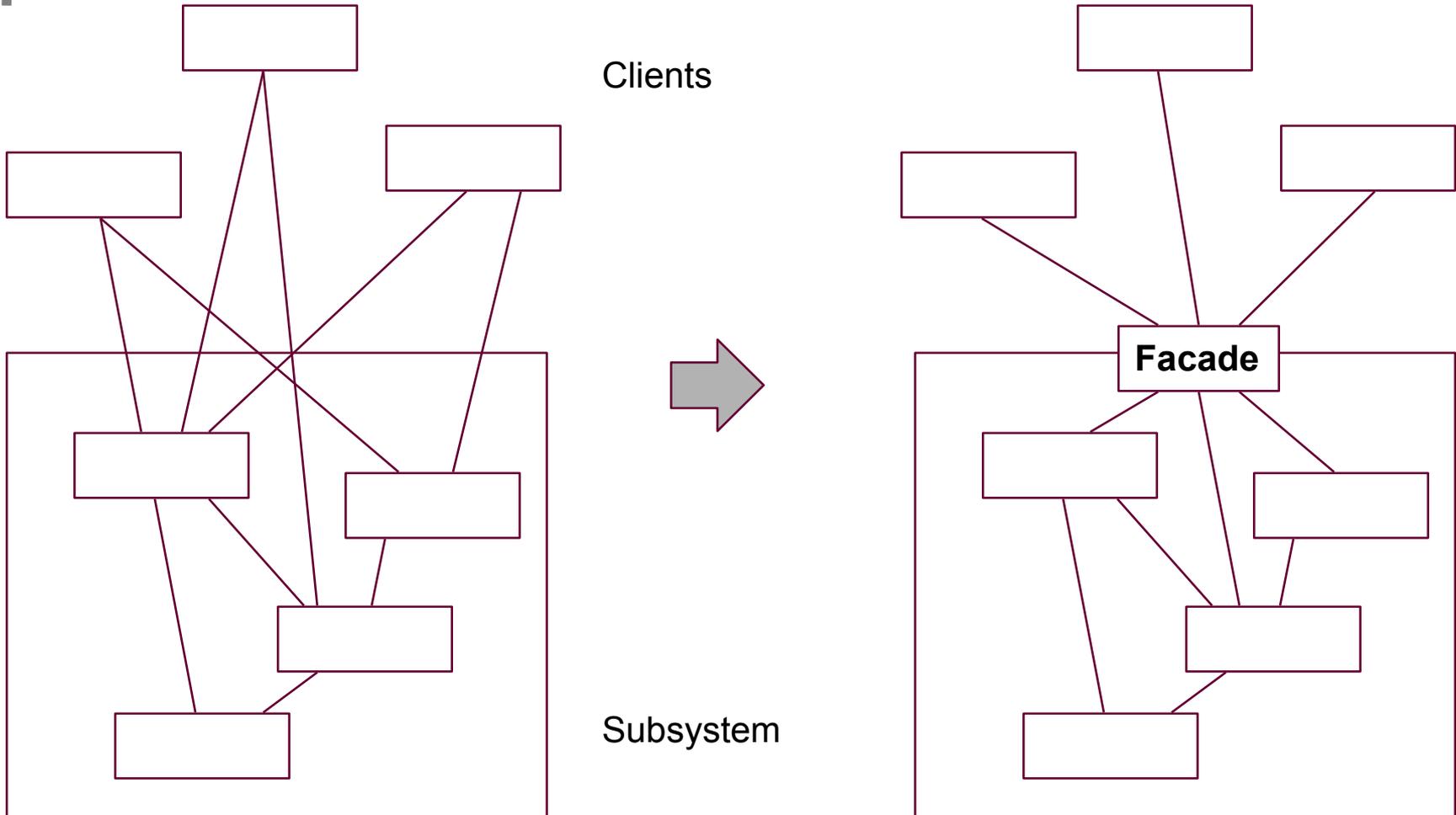
- ▶ Eine Fassade bietet eine *Sicht* auf ein Subsystem an. Es darf mehrere Sichten geben, nur keinen direkten Zugriff auf die inneren Objekte



# Restrukturierung hin zur Fassade

51

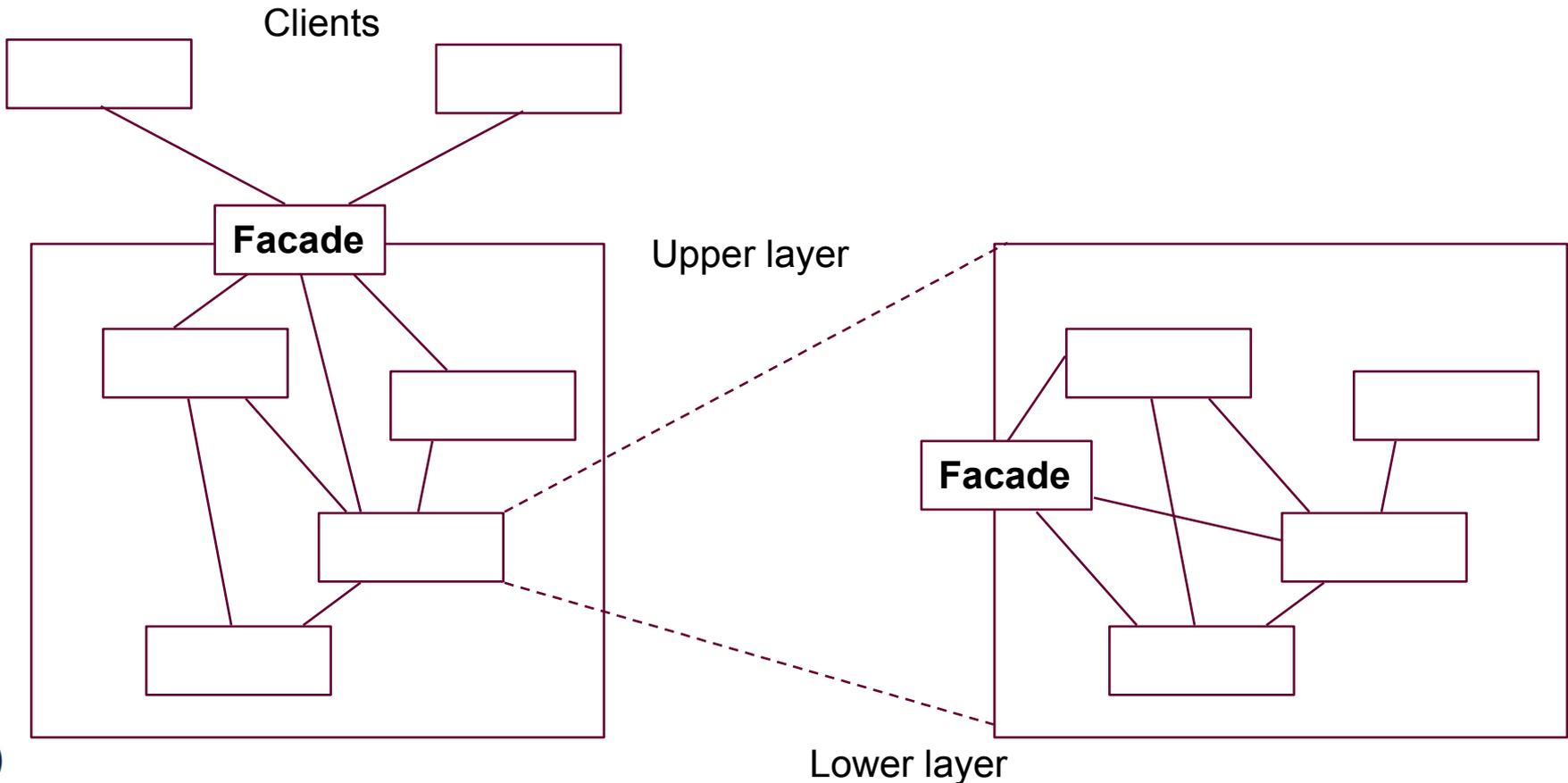
- ▶ Fassaden entkoppeln; Subsysteme können leichter ausgetauscht werden (Variabilitätsmuster)



# Fassaden und Schichten

52

- ▶ Falls einzelne Klassen eines Subsystems wieder Fassaden sind, entstehen *fassadengeschützte Schichten*





# 41.B Entwurfsmuster

## Fabrikmethode (FactoryMethod)

---

---

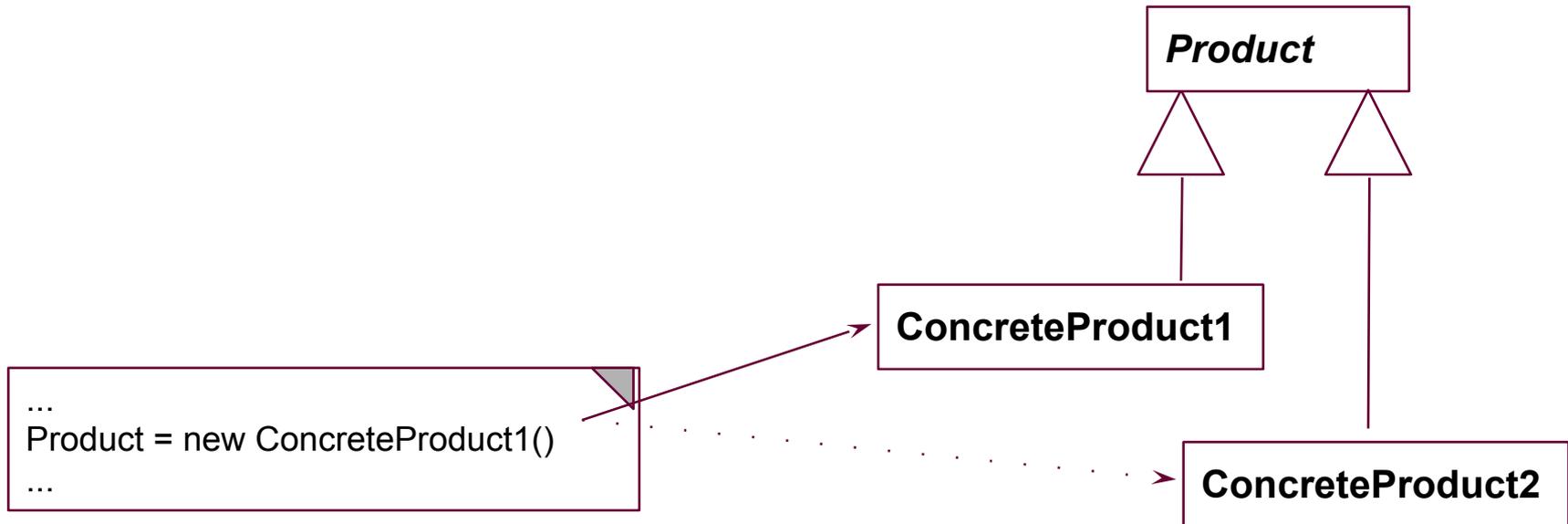
53

(Wdh.)  
zur polymorphen Variation von Komponenten (Produkte)  
und zum Verbergen von Produkt-Arten

# Problem der Fabrikmethode

54

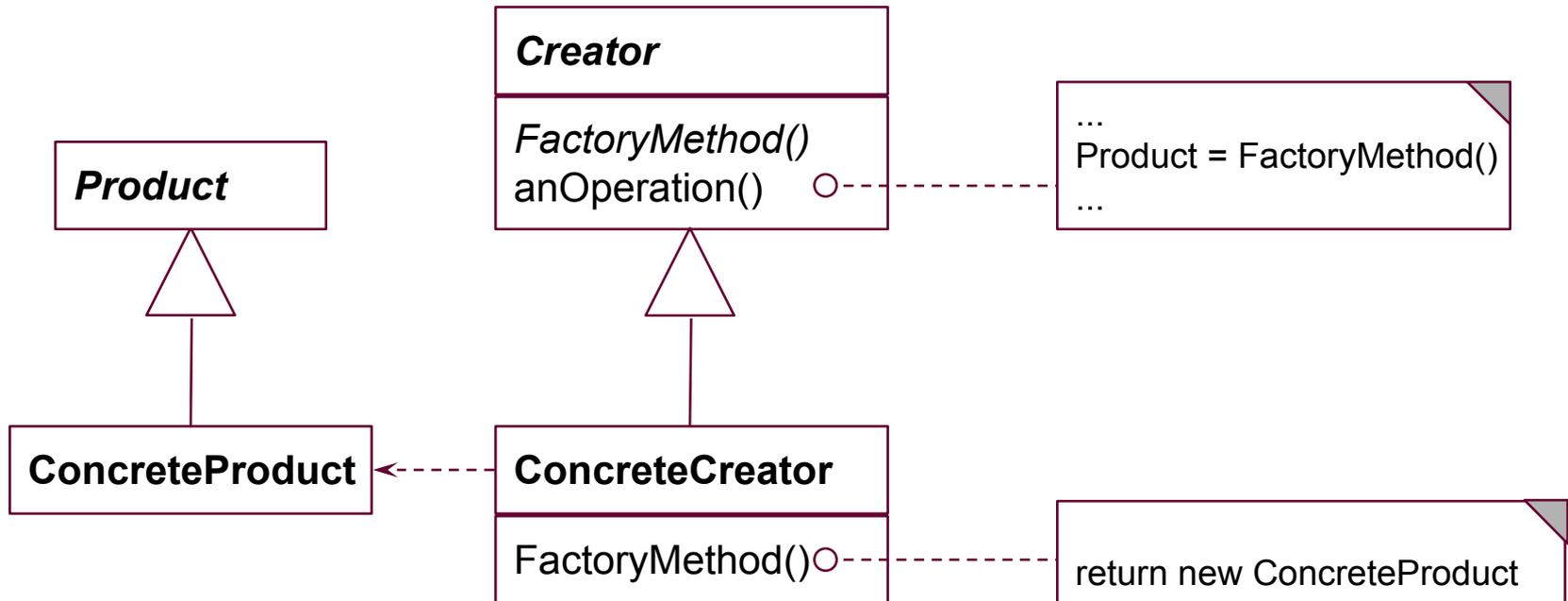
- ▶ Wie variiert man die Erzeugung für eine polymorphe Hierarchie von Produkten?
- ▶ Problem: Konstruktoren sind nicht polymorph!



# Struktur Fabrikmethode

55

- ▶ FactoryMethod ist eine Variante von TemplateMethod, zur Produkterzeugung



# Factory Method (Polymorphic Constructor)

56

- ▶ Abstract creator classes offer abstract constructors (polymorphic constructors)
  - Concrete subclasses can specialize the constructor
  - Constructor implementation is changed with allocation of concrete Creator

```
// Abstract creator class
public abstract class Creator {
    // factory method
    public abstract Set createSet(int n);
}
```

```
public class Client {
... Creator cr = [.. subclass ]..
    public void collect() {
        Set mySet = Creator.createSet(10);
        ....
    }
}
```

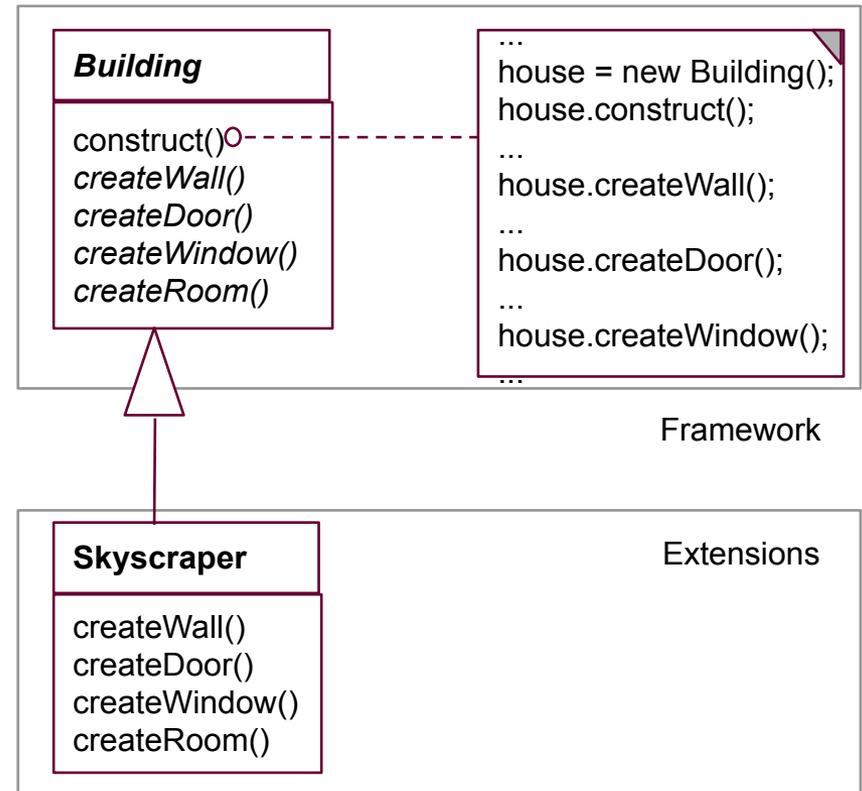


```
// Concrete creator class
public class ConcreteCreator extends Creator {
    public Set createSet(int n) {
        return new ListBasedSet(n);
    }
    ...
}
```

# Beispiel FactoryMethod

57

- ▶ Rahmenwerk für Gebäudeautomation
  - Klasse Building hat eine Schablonenmethode zur Planung von Gebäuden
  - Abstrakte Methoden: createWall, createRoom, createDoor, createWindow
- ▶ Benutzer können Art des Gebäudes verfeinern
- ▶ Wie kann das Rahmenwerk neue Arten von Gebäuden behandeln?



# Lösung mit FactoryMethod

58

- ▶ Bilde createBuilding() als Fabrikmethode aus

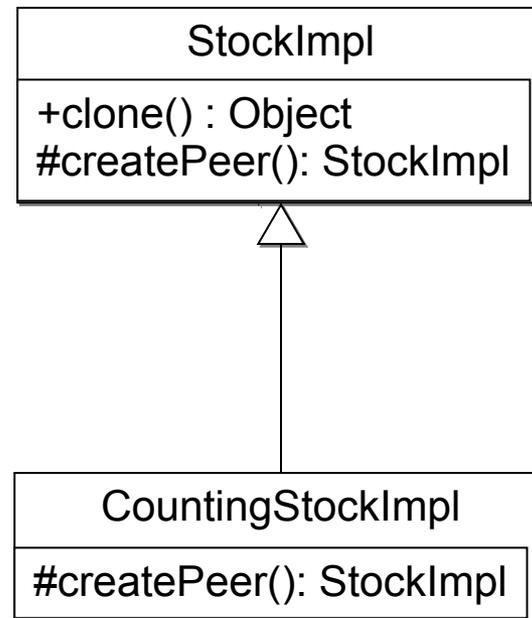
```
// abstract creator class
public abstract class Building {
    public abstract
        Building createBuilding();
    ...
}
```

```
// concrete creator class
public class Skyscraper extends Building {
    Skyscraper() {
        ...
    }
    public Building createBuilding() {
        ... fill in more info ...
        return new Skyscraper();
    }
    ...
}
```

# Factory Method im SalesPoint-Rahmenwerk

59

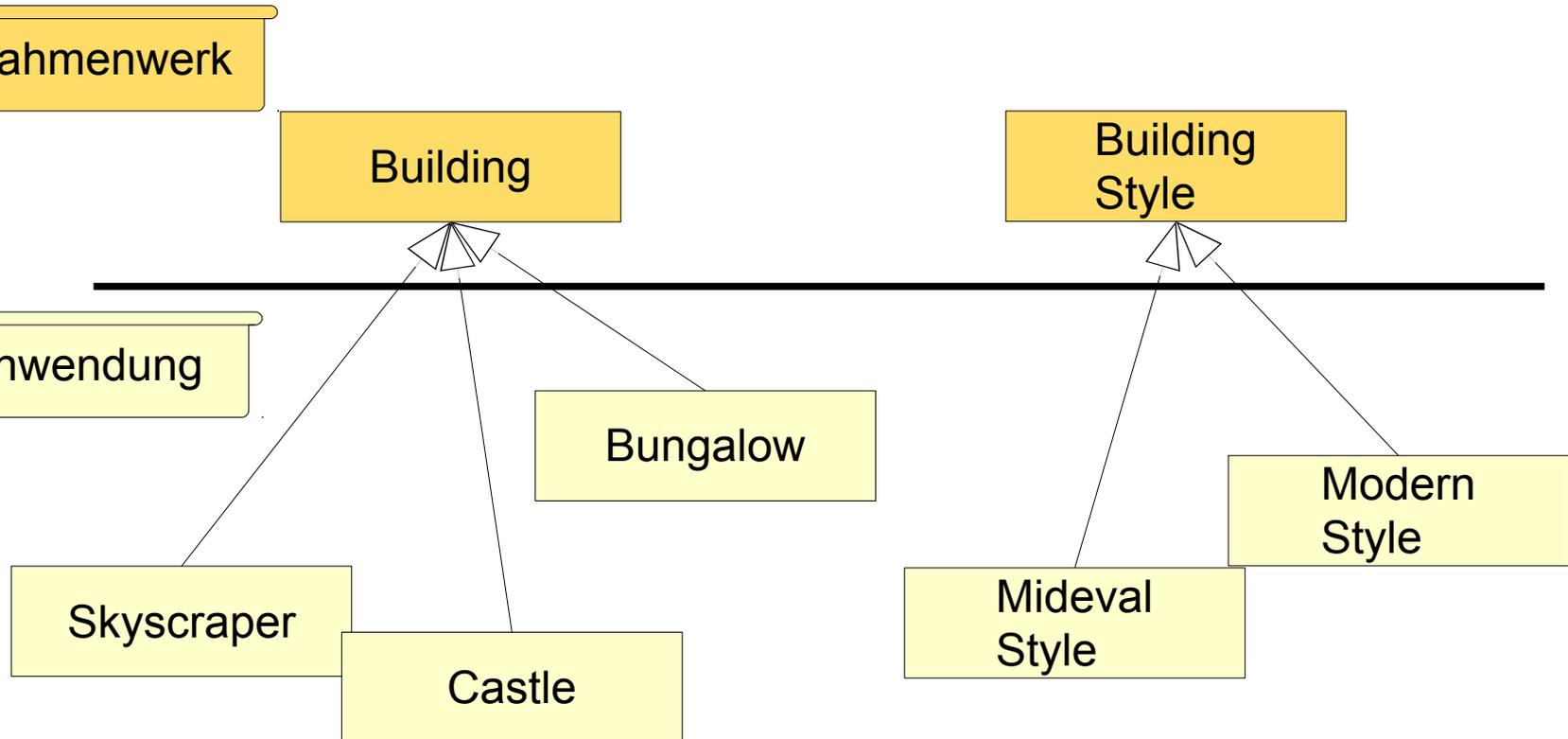
- ▶ Anwender von SalesPoint verfeinern die StockImpl-Klasse, die ein Produkt des Warenhauses im Lager repräsentiert
  - z.B. mit einem CountingStockImpl, der weiß, wieviele Produkte noch da sind



# Einsatz in Komponentenarchitekturen

60

- ▶ In Rahmenwerk-Architekturen wird die Fabrikmethode eingesetzt, um von oberen Schichten (Anwendungsschichten) aus die Rahmenwerkschicht zu konfigurieren:



# 41.C

## Strategie (Strategy, Template Class)

61

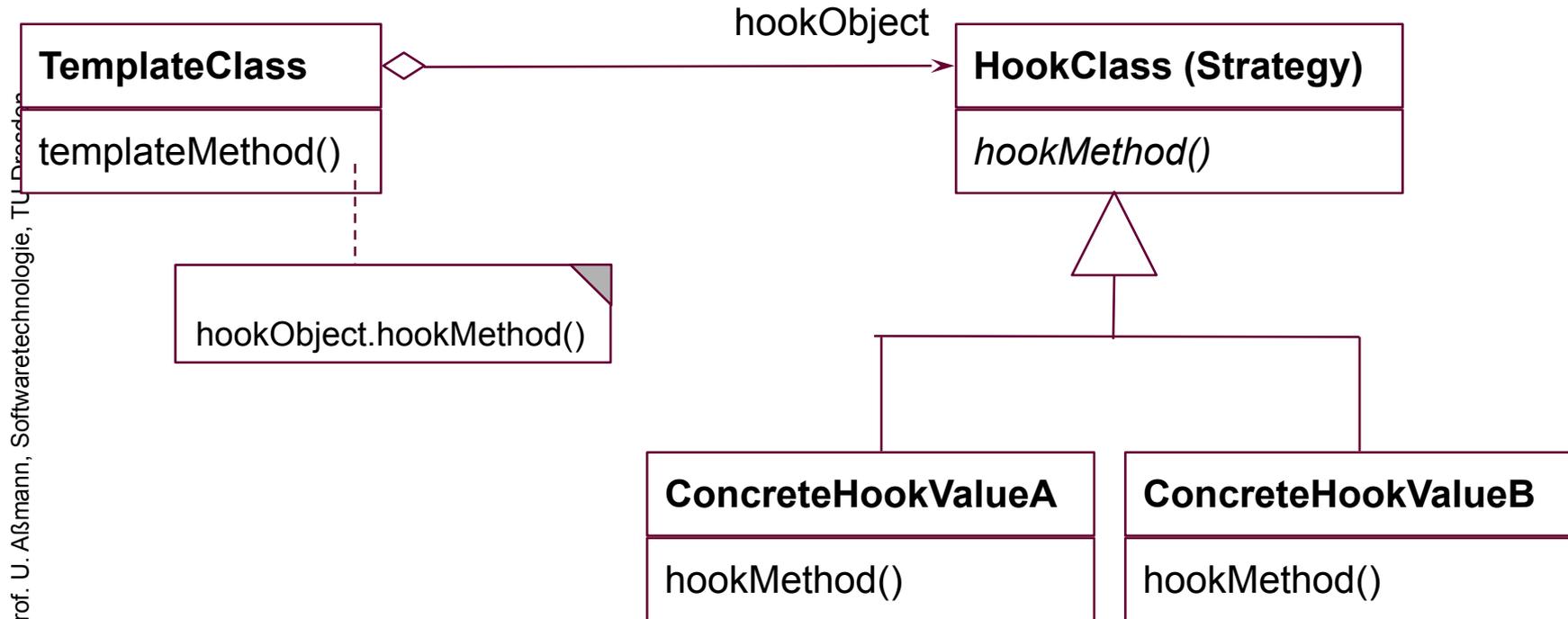
(Wdh.)



# Strategy (also called Template Class)

62

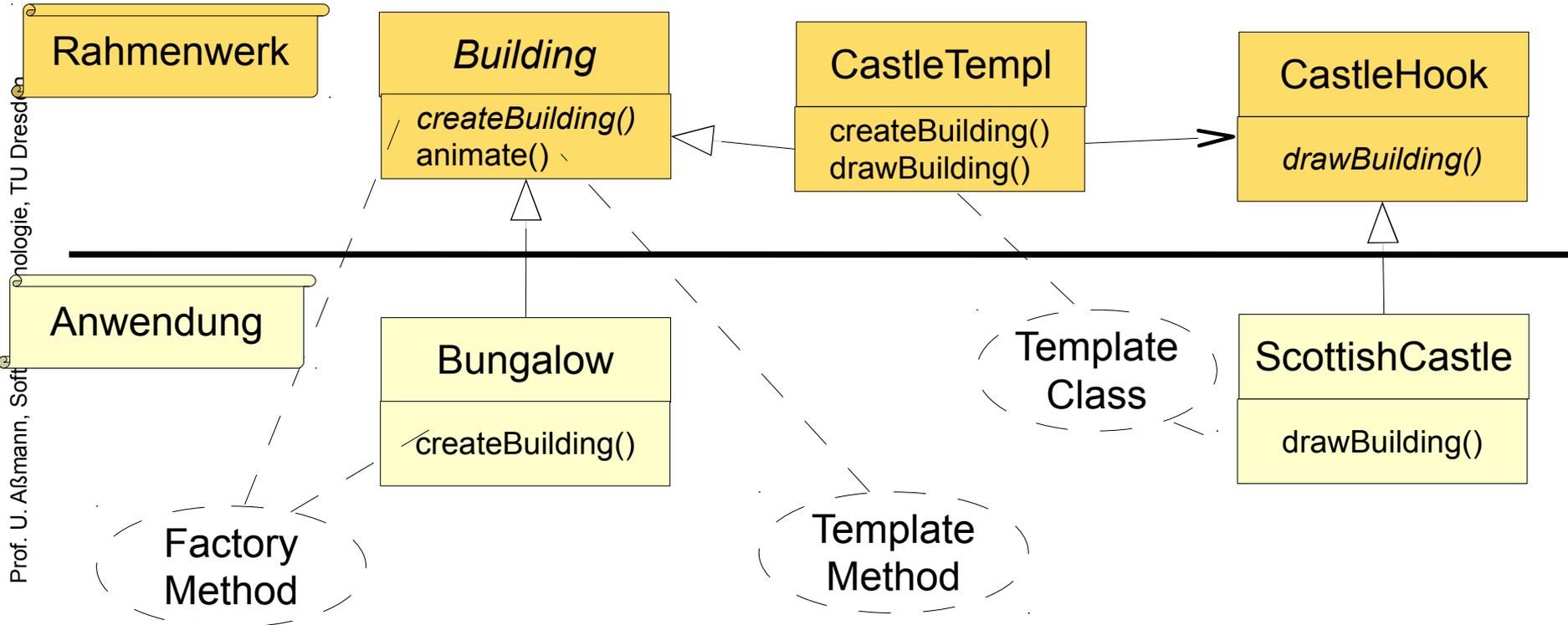
- ▶ Strategy wirkt wie TemplateMethod, nur wird die Hakenmethode in eine separate Klasse ausgelagert
- ▶ Zur Variation der Hakenklasse (und -methode)



# Kombinierter Einsatz in Rahmenwerken

63

- ▶ FactoryMethod variiert den Konstruktor
- ▶ TemplateMethod oder Strategy (TemplateClass) variiert die Hookmethode
- ▶ Bridge (s. später) variiert die TemplateMethode





# 41.D Entwurfsmuster Einzelstück (Singleton)

---

---

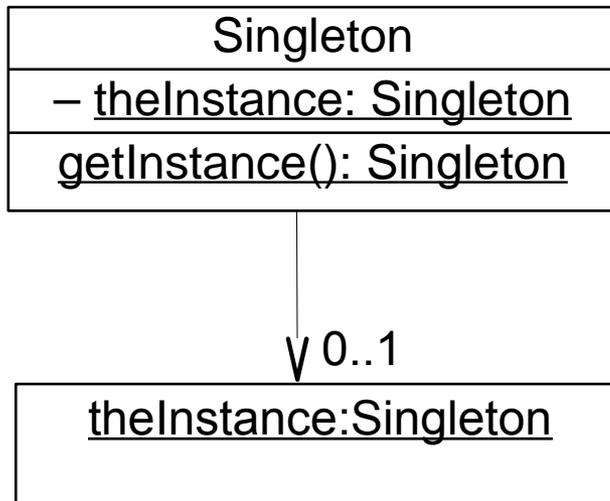
64

zur globalen Konfiguration einer Komponente  
oder Schicht  
(Wdh)

# Entwurfsmuster Einzelstück (Singleton)

65

- ▶ Gesucht: globales Objekt, das global oder innerhalb einer Laufzeitkomponente (z.B. Schicht) Daten, z.B. Konfigurationsdaten, vorhält
- ▶ Idee:
  - Erstelle eine Klasse, von der genau ein Objekt existiert (Invariante)
  - Erstelle einen artifiziellen Konstruktor (Fabrikmethode), der oft aufgerufen werden kann, aber die Invariante sicherstellt
  - Eigentlicher Konstruktor wird *verborgen* (*private*)
  - Austausch der Konfiguration durch Unterklassenbildung (Variabilität)



```
class Singleton {
    private static Singleton theInstance =
    null;
    private Singleton () {}
    public static Singleton getInstance () {
        if (theInstance == null)
            theInstance = new Singleton();
        return theInstance;
    }
}
```

# Singleton im SalesPoint – the Shop

66

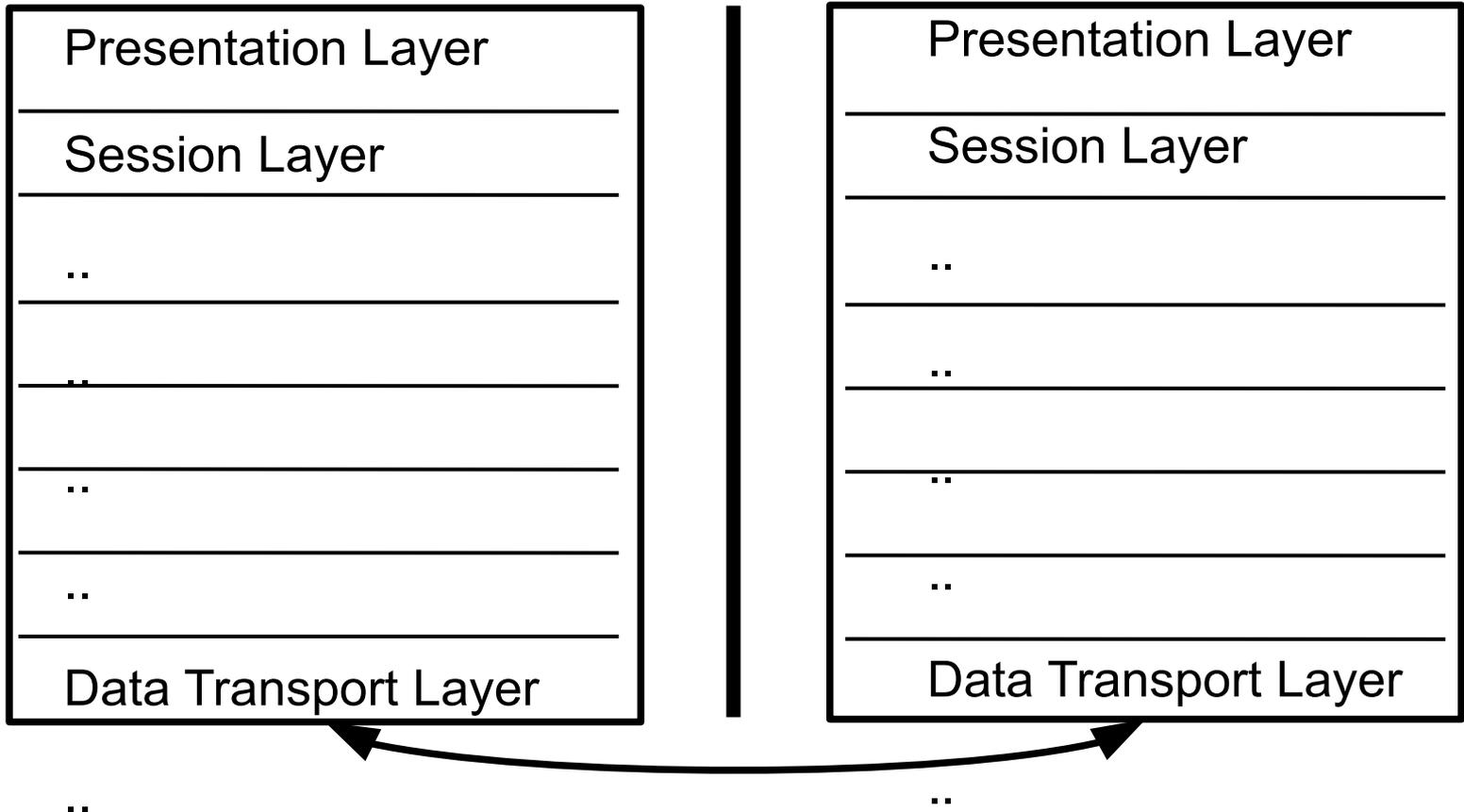
- ▶ Der Shop im SalesPoint-Rahmenwerk ist ein Einzelstück (die Firma). Dagegen gibt es viele Verkaufsstellen (sales points)
- ▶ Austausch der Eigenschaften des Shops durch Unterklassenbildung



# Example: ISO-OSI 7 Layers Network Architecture

67

- ▶ Every layer contains an abstract machine (set of operations)



# Example: Operating Systems

68

UNIX,  
Linux,  
Android

User Space

---

Kernel

---

Apple-  
UNIX,  
iOS

User Space

---

Kernel

---

Microkernel (Mach)

---

Windows NT/XP:

User Space

---

Kernel

---

Hardware Abstraction Layer (HAL)

---

# Example: Database Systems

69

SQL compiler

transaction manager

lock manager

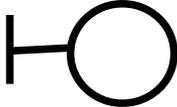
table manager

physical storage management

record management layer

# Repet.: BCD/BCED Classification

70

- ▶ Boundary classes:  <<boundary>>
  - Represent an interface item that talks with the user
  - May persist beyond a run
- ▶ Control class:  <<control>>
  - Controls the execution of a process, workflow, or business rules
  - Does not persist
- ▶ Entity class:  <<entity>>
  - Describes persistent knowledge. Caches a persistent object from a database (data access object, DAO)
- ▶ Database class  <<database>>
  - Adapter class for the database
  - Often, Entity and Database classes are unified
- ▶ **BCD/BCED is linked with the 3-tier architecture**

# Teil IV: Objektorientierter Entwurf

## 41 Grundlegende Architekturprinzipien

1

Prof. Dr. rer. nat. habil. Uwe  
Aßmann  
Institut für Software- und  
Multimediatechnik  
Lehrstuhl Softwaretechnologie  
Fakultät für Informatik  
TU Dresden  
Version 13-1.0, 08.07.13

- 1) Architekturprinzipien
- 2) Flexible Evolution mit Modularität und Geheimnisprinzip
- 3) Geschichtete Architekturen



- Parallelen zum Fachgebiet der Architektur:
  - Architekten sind an der Nahtstelle zwischen Kunde und Baufirma.
  - Schlechter Architekturentwurf kann nicht durch gute Bauqualität kompensiert werden.
  - Es gibt Architektur-Spezialisten für bestimmte Anwendungsgebiete.
  - Es gibt "Schulen", die bestimmte Grundprinzipien vertreten.
  - Es gibt bestimmte Standard-Muster und Grundregeln, die immer wieder angewendet werden.

2

- ▶ Zuser Kap 10.
- ▶ Ghezzi 4.1-4.2
- ▶ Pfleeger 5.1-5.3
- ▶ ST für Einsteiger 5.3, 8

- ▶ David J. Parnas. On a buzzword: hierarchical structure. Proceedings IFIP Congress 1974, North-Holland, Amsterdam.
- ▶ Johannes Siedersleben. Moderne Softwarearchitektur. Umsichtig planen, robust bauen mit Quasar. dpunkt-Verlag, 2004.

# Teil IV - Objektorientierter Entwurf (Object-Oriented Design, OOD)

4

- 1) 41: Einführung in die objektorientierte Softwarearchitektur
  - 1) Modularität und Geheimnisprinzip
  - 2) Entwurfsmuster für Modularität
  - 3) BCD-Architekturstil (3-tier architectures)
- 2) 42: Verfeinerung des Entwurfsmodells zum Implementierungsmodell (Anreicherung von Klassendiagrammen)
- 3) 43: Verfeinerung von Lebenszyklen
  - 1) Verfeinerung von verschiedenen Steuerungsmaschinen
- 4) 44: Verfeinerung mit querschneidender Objektorichnung
- 5) 45 Wiederverwendung
  - 1) Objektorientierte Rahmenwerke (frameworks)
  - 2) Softwarearchitektur mit dem Quasar-Architekturstil



## 41.1. Herstellung Großer Softwaresysteme

5

*software*: computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.

(IEEE Standard Glossary of Software Engineering)



***Softwaresystem***: Ein System (oder Teilsystem), dessen Komponenten aus Software bestehen

Klassifikation von Software:

- Generisches Produkt oder Einzelanfertigung (vereinbartes Produkt)?
- Systemsoftware oder Anwendungssoftware?
- eingebettete (produktintegriert) oder eigenständig?
- Realzeitanforderungen oder

## Was heißt hier "groß"?

6

- Klassifikation nach W. Hesse:

Klasse	Anzahl Code-Zeilen	Personenjahre zur Entwicklung
sehr klein	bis 1.000	bis 0,2
klein	1.000 - 10.000	0,2 - 2
mittel	10.000 - 100.000	2 - 20
groß	100.000 - 1 Mio.	20 - 200
sehr groß	über 1 Mio.	über 200



Quelle: Gumm/Sommer,  
Einführung in die  
Informatik, 4. Auflage,  
2000, S. 639

Annahme: eine Person  
produziert pro Tag ca. 10  
bis 100 Zeilen  
Programmcode, d.h. ca.  
5000 Zeilen im Jahr.

## Riesige Systeme

7

- ▶ Telefonvermittlungssoftware EWSD (Version 8.1):
  - 12,5 Mio. Code-Zeilen
  - ca. 6000 Personenjahre
- ▶ ERP-Software SAP R/3 (Version 4.0)
  - ca. 50 Mio. Code-Zeilen
- ▶ Umfang der verwendeten Software (Anfang 2000):
  - Credit Suisse 25 Mio. Code-Zeilen
  - Chase Manhattan Bank: 200 Mio. Code-Zeilen
  - Citicorp Bank: 400 Mio. Code-Zeilen
  - AT&T: 500 Mio. Code-Zeilen
  - General Motors: 2 Mrd. Code-Zeilen

### Abkürzungen:

EWSD = Elektronisches Wählsystem Digital (Siemens-Pro)  
ERP = Enterprise Resource Planning  
SAP: Deutscher Software-Konzern



Quellen für die Zahlen:  
EWSD: Siemens-interne  
Information  
SAP: [www.ct-software.com](http://www.ct-software.com)  
Jahr 2000: Gary North's  
Y2K Links and Forums,  
basierend auf  
Presseberichten zwischen  
1996 und 1999  
Quellcode von Windows  
2000 (gerichtetweise): ca

# Strukturprobleme

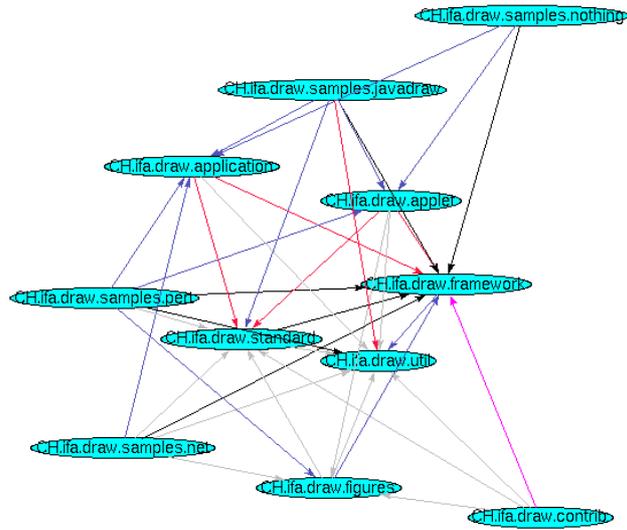
9

- ▶ The following figures are taken from the Goose Reengineering Tool, analysing a Java class system [Goose, FZI Karlsruhe, <http://www.fzi.de>]









- ▶ Softwarearchitektur ist der Schlüssel zum Erfolg des Softwareingenieurs und seiner Firma.

Ohne gute Softwarearchitektur keine  
Wiederverwendung, Evolution, Variabilität,  
Erweiterbarkeit

Mit guter Softwarearchitektur  
Softwareproduktlinien, schnell erstellte neue Produkte,  
vertikale Portierung auf andere Domänen,  
einfaches Dienstleistungsgeschäft.

# 41.2 Grundlegende Architekturprinzipien

15

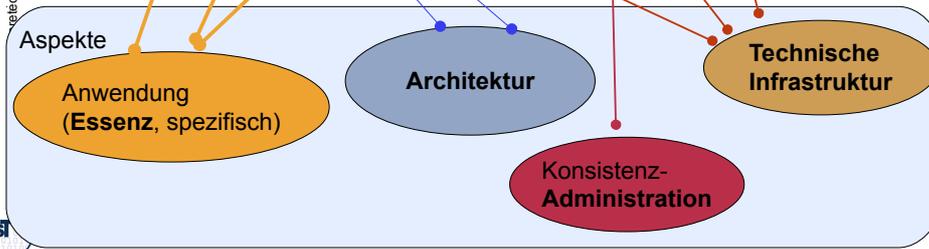


# Wesentliche Aspekte und Bestandteile eines Softwaresystems

16

- ▶ Anwendungsspezifische Funktionen
- ▶ Benutzungsoberfläche
- ▶ Ablaufsteuerung
- ▶ Datenhaltung
- ▶ Infrastrukturdienste
  - Objektverwaltung
  - Interne Objekt- und Prozesskommunikation
  - Verteilungsunterstützung
- ▶ Kommunikationsdienste
- ▶ Sicherheitsfunktionen
- ▶ Zuverlässigkeitsfunktionen
- ▶ Systemadministration
  - Installation, Anpassung
  - Systembeobachtung
- ▶ Vertragsprüfung
- ▶ Etc.

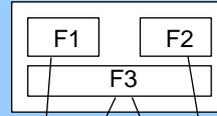
Informationstechnologie, TU Dresden



## Aspekte des Architekturentwurfs

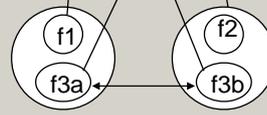
### ▶ Strukturelle Zerlegung:

- Blockdiagramme, Montagediagramme (UML-Komponentendiagramme)
- Architekturstil: Schichten, Sichten, Dimensionen



### ▶ Struktur der physikalischen Verteilung:

- Zentral oder verteilt?
- Topologie

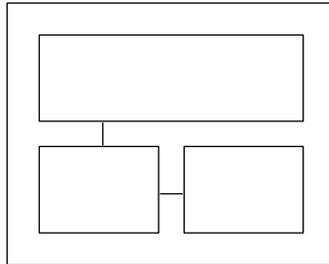


- ▶ Ablaufsicht
- ▶ Logischer Detail-Entwurf
- ▶ Einhaltung nichtfunktionaler Anforderungen:
  - Architekturbestimmende Eigenschaften (z.B. Realzeitsystem, eingebettetes System)
  - Effizienzanforderungen und Optimierung
  - Standardarchitekturen

## Blockdiagramme

18

- ▶ **Blockdiagramme** sind das meistverbreitete, informelle Hilfsmittel zum Skizzieren der logischen **Struktur** einer Systemarchitektur.
  - Blockdiagramme sind kein Bestandteil von UML
  - **Blöcke** stellen UML-Komponenten *ohne* Anschlüsse dar



Prof. U. Abmann, Softwaretechnologie, TU Dresden



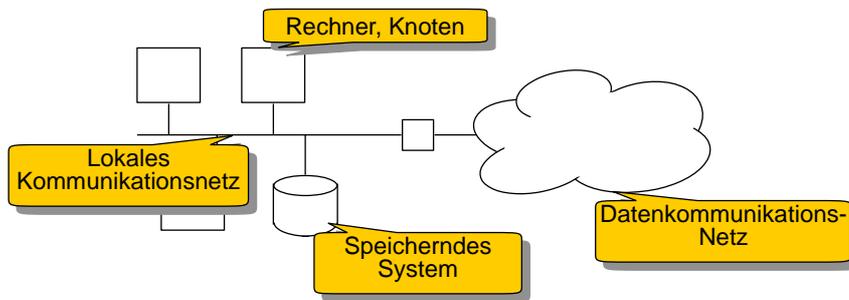
Die Aufteilung eines Systems in Teilsysteme sollte grundsätzlich immer zum Ziel haben:

- Hohe Kohäsion, d.h. hoher Zusammenhalt innerhalb der Teilsysteme
- Niedrige Kopplung, d.h. relativ einfache ("schlanke") Schnittstellen zwischen den Teilsystemen

# Konfigurationsdiagramme

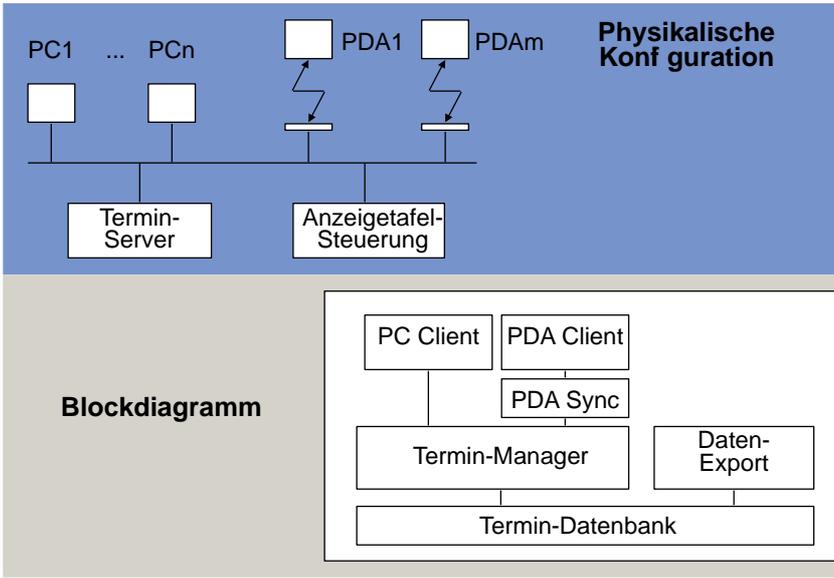
19

- ▶ Konfigurationsdiagramme sind Blockdiagramme mit "Bussen"
  - Konfigurationsdiagramme sind nicht Bestandteil von UML
  - ein verbreitetes Hilfsmittel zur Beschreibung der physikalischen Verteilung



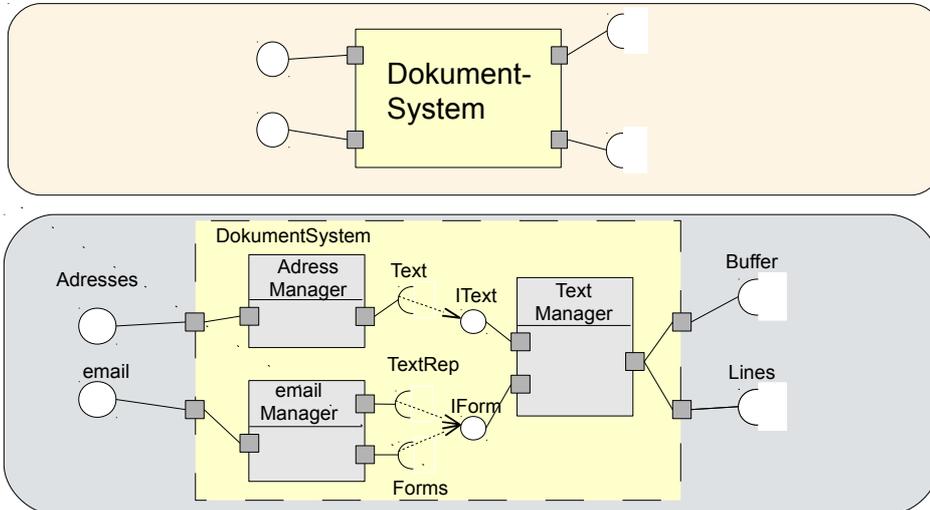
# Beispiel: Konfigurationsdiagramm für Terminverwaltung

20



# Architekturstil für nicht-Interaktive Anwendungen: Hierarchische Organisation der obersten Ebenen

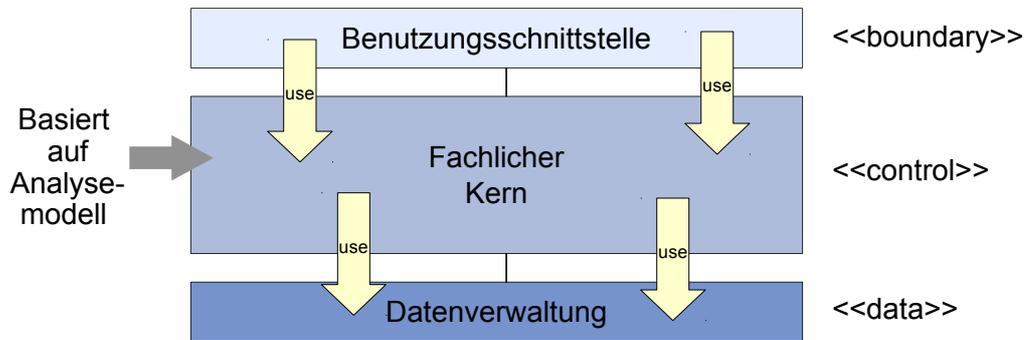
- 21
- ▶ Oberste Ebene des Systems ist meist hierarchisch und/oder geschichtet organisiert
    - Vermeide "wilde" objekt-orientierte Netzstrukturen
    - Damit die letzte Integration zum Gesamtsystem einfach verläuft: Integrationstests können dann bottom-up absolviert werden
  - Hierarchien bilden Spezialfälle, denn sie können geschichtet werden



# Architekturstil für Interaktive Anwendungen: Drei-Schichten-Architektur (BCD)

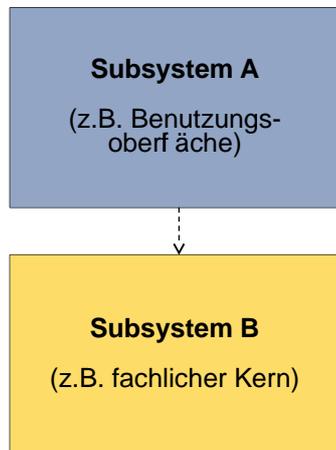
22

- ▶ Klassische Struktur eines interaktiven Anwendungssystems
- ▶ Integrationstest verläuft wegen azyklischer Benutzungsrelation (use) bottom-up: erst D, dann CD, dann BCD
- ▶ Fachlicher Kern (Anwendungslogik) kann weitere Schichten enthalten
  - Oft kapselt eine Facade eine Schicht nach oben ab, dann existieren bereits zwei Teil-Schichten



## Architekturprinzip: Hohe Kohäsion + Niedrige Kopplung

23



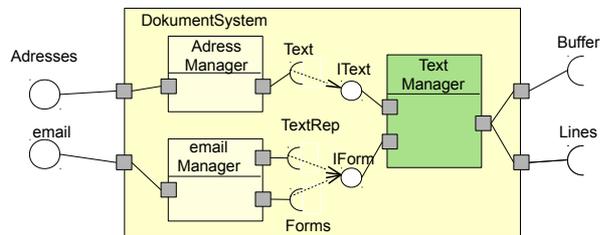
- ▶ **Hohe Kohäsion:**  
Subsystem B darf keine Information und Funktionalität enthalten, die zum Zuständigkeitsbereich von A gehört und umgekehrt.
- ▶ **Niedrige Kopplung:**  
Es muß möglich sein, Subsystem A weitgehend auszutauschen oder zu verändern, ohne Subsystem B zu verändern.  
Änderungen von Subsystem B sollten nur möglichst einfache Änderungen in Subsystem A nach sich ziehen.



## Architekturprinzip Quasar: Trennung von Technik- und Anwendungskomponenten

24

- ▶ Jede Komponente wird klassifiziert in Blutgruppen (0 – technologieunabhängige Algorithmen, T – technologieabh. Komponenten, A- Anwendungskomponenten)



**[Siedersleben] Quasar-Wiederverwendungsgesetz:  
0- und T-Komponenten sind besser  
wiederverwendbar als Anwendungskomponenten.**

## 41.2 Architekturprinzip: Veränderungsorientierter Entwurf mit dem Modul-Geheimnis



25



# Verteilung in (e)

---

Prinzip sollte Software in *Komponenten*

den Sinne ist eine

führt Kohäsion

explizit in der Schnittstelle angegebene  
Komponenten

entwickelt werden

getestet werden (Einheitstest, unit test)

den Komponenten verfolgt werden

so dass das System zusammenbricht

weil angebotene und benötigte Schnittstellen

explizit bezeichnet, die in binäre Form

# Bemerk.: Komponentenmodelle und Kompositionssysteme

27

- ▶ Es gibt nicht nur die UML-Komponente...
- ▶ sondern viele verschiedene *Komponentenmodelle*:
  - Module einer modularen Programmiersprache (Modula, Ada)
  - Klassen, Kollaborationen und Konnektoren in objektorientierten Sprachen
  - UML-Komponenten
  - Fragmentkomponenten, Schablonen (templates)
  - Dokumentkomponenten
  - Serverseitige Webkomponenten
- ▶ Ein *Kompositionssystem* definiert:
  - **Komponentenmodell**: Eigenschaften der Schnittstelle einer Komponente
  - **Kompositionstechnik**: Wie werden Komponenten komponiert?
  - **Kompositionssprache**: Wie wird die Architektur eines großen Systems beschrieben?
- ▶ --> Vorlesung CBSE (SS)



# flexible Evolution mit p

---

## in Geheimnisprinzip

f, *change-oriented modularization with*  
[M 1972]:

alternativen), die sich *ändern können*  
eine Komponente, die die Entscheidung

das **Komponenten-** oder  
**secret**)

für die Komponente, die unverändert  
eidung und somit die Implementierung

Austausch von Implementierungen  
somit flexible Evolution

# se von ten

---

konkrete Implementierung  
n

# Verschiedene Arten von Komponenten/Modulen

30

- ▶ Funktionale Module ohne Zustand
  - sin, cos, BCD arithmetic, gnu mp,...
- ▶ Daten-Repositoryen
  - Verbergen Repräsentation, Zugriff und Zustand der Daten
  - Symboltabellen, Materialcontainer, ...
- ▶ Abstrakte Datentypen
- ▶ Singletons (Konfigurationskomponenten)
  - Klassen mit einer einzigen Instanz
- ▶ Prozesse (aktive Objekte)
- ▶ Klassen
  - Module, die ausgeprägt werden können
- ▶ Generische Klassen (Klassenschablonen)
- ▶ Komplexe Klassen (UML-Komponenten)
- ▶ Fragmentkomponenten

... für alle gilt das Geheimnisprinzip



# Variabilitätsmuster nutzen das Geheimnisprinzip

31

- ▶ Viele Entwurfsmuster (z.B. TemplateMethod) sind *Variabilitätsmuster*, d.h., sie lassen einem bestimmte Geheimnisse verbergen und dann die Implementierungen austauschen (variieren)
  - Fassade verbirgt ein ganzes Subsystem
  - Fabrikmethode verbirgt die Allokation von Produkten
  - TemplateMethod und Strategie verbergen einen Anteil eines Algorithmus
  - Singleton kapselt globale Konfigurationsdaten
- ▶ In UML kann man Entwurfsmuster als Komponenten (Wiederverwendungseinheiten) kapseln, indem man sie als Kollaborationen spezifiziert



# Was wollen wir eigentlich variieren?

32

- ▶ **Software-Produktlinien** entstehen durch systematische Variation von Geheimnissen

Dynamische Kontexteigenschaften

Benutzer (Level,  
Präferenzen)  
Netzwerkverbindung  
Sicherheitseigenschaften

Funktionale Eigenschaften  
(Funktionale Varianten in der Control)

Mailer      Writer  
Calendar    SlideEditor  
Browser     Wiki

MiddlewareVarianten  
(Entity level)

Parallelität (Transaktionen)  
Verteilung

Plattformeigenschaften  
(Boundary, Database)

Chip, OS, GUI-Bibliothek  
Persistenz, Displaygerät  
Web/Rich/Smart Client

# 41.3 Architekturprinzip Schichtung (Layered Architectural Styles)



33

und die "benutzt"-Relation

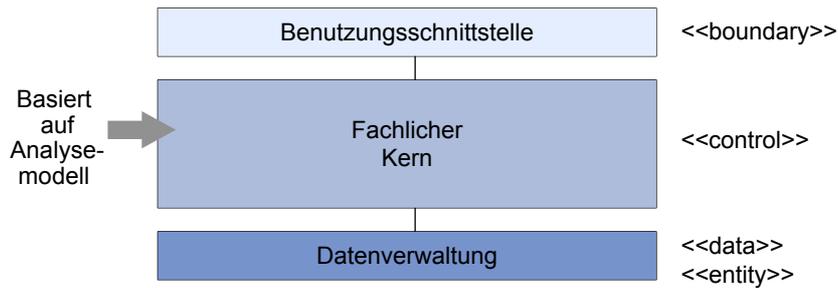


Split off from 3-design-intro March 2003, t  
better to PUM-I.

## Drei-Schichten-Architekturstil (BCD) für interaktive Anwendungen

34

- ▶ Klassische Struktur eines interaktiven Anwendungssystems
- ▶ Schichten sind jeweils stark kohäsiv, und lose gekoppelt – warum?
- ▶ Oft kapselt eine Fassade eine Schicht, ein Einzelstück konfiguriert jede Schicht, Fabriken schneiden die Produkte der unteren Schichten zu, TemplateMethod/Class variieren Algorithmen der Produkte



# onen zwischen

---

en zwischen den Komponenten eines

heritance), behaves-like

ns-a (security)

richten

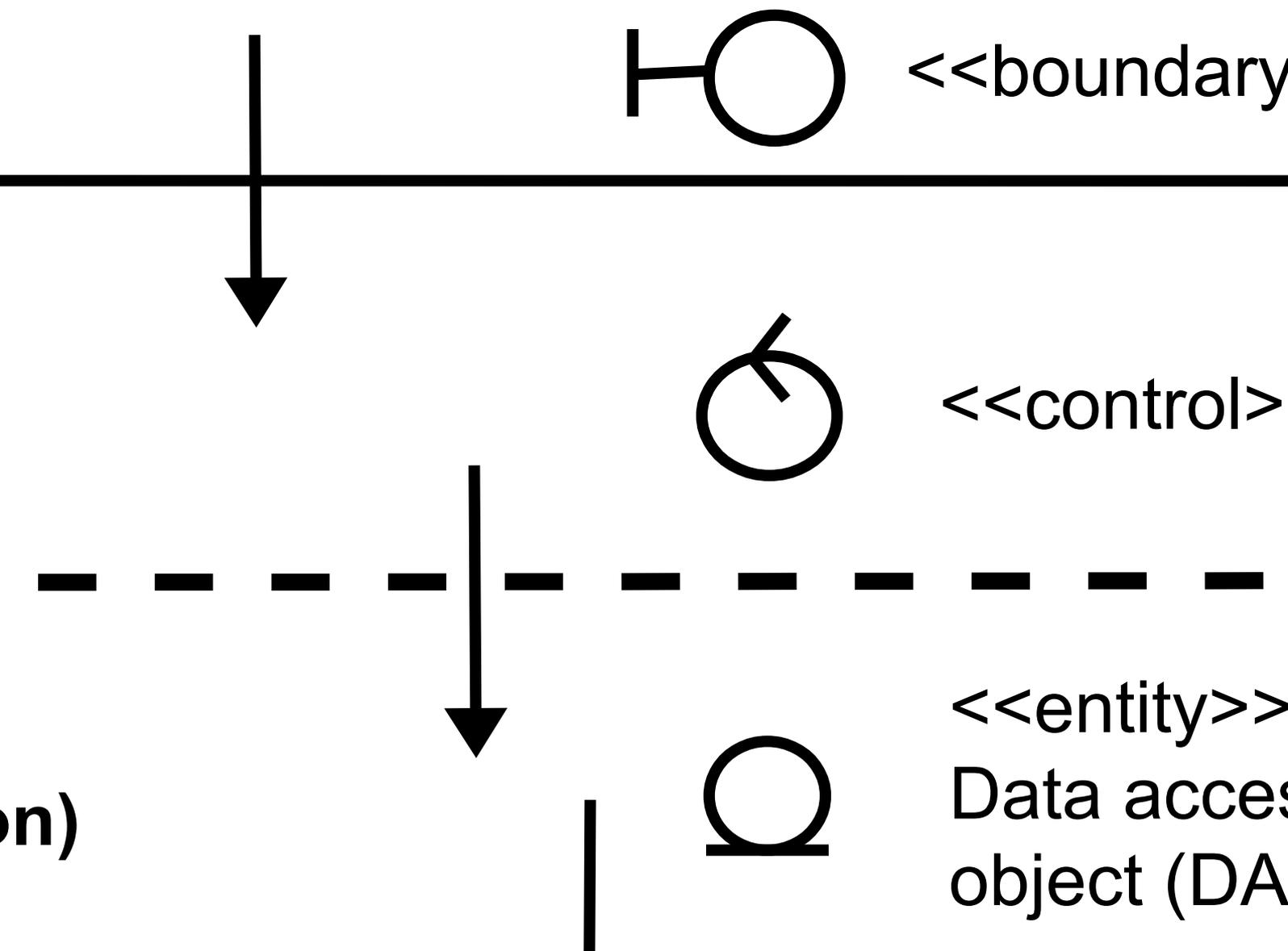
e diese zusammen





# tion in 3- and 4-Tier

an acyclic USES relation, divided into 3  
er in an acyclic relationship  
at not vice versa



# er Architectures

---

knowledge in a layer

S relationship

of the application

ns and look

es how data is stored (database, transient,  
such as Enterprise JavaBeans)

oth

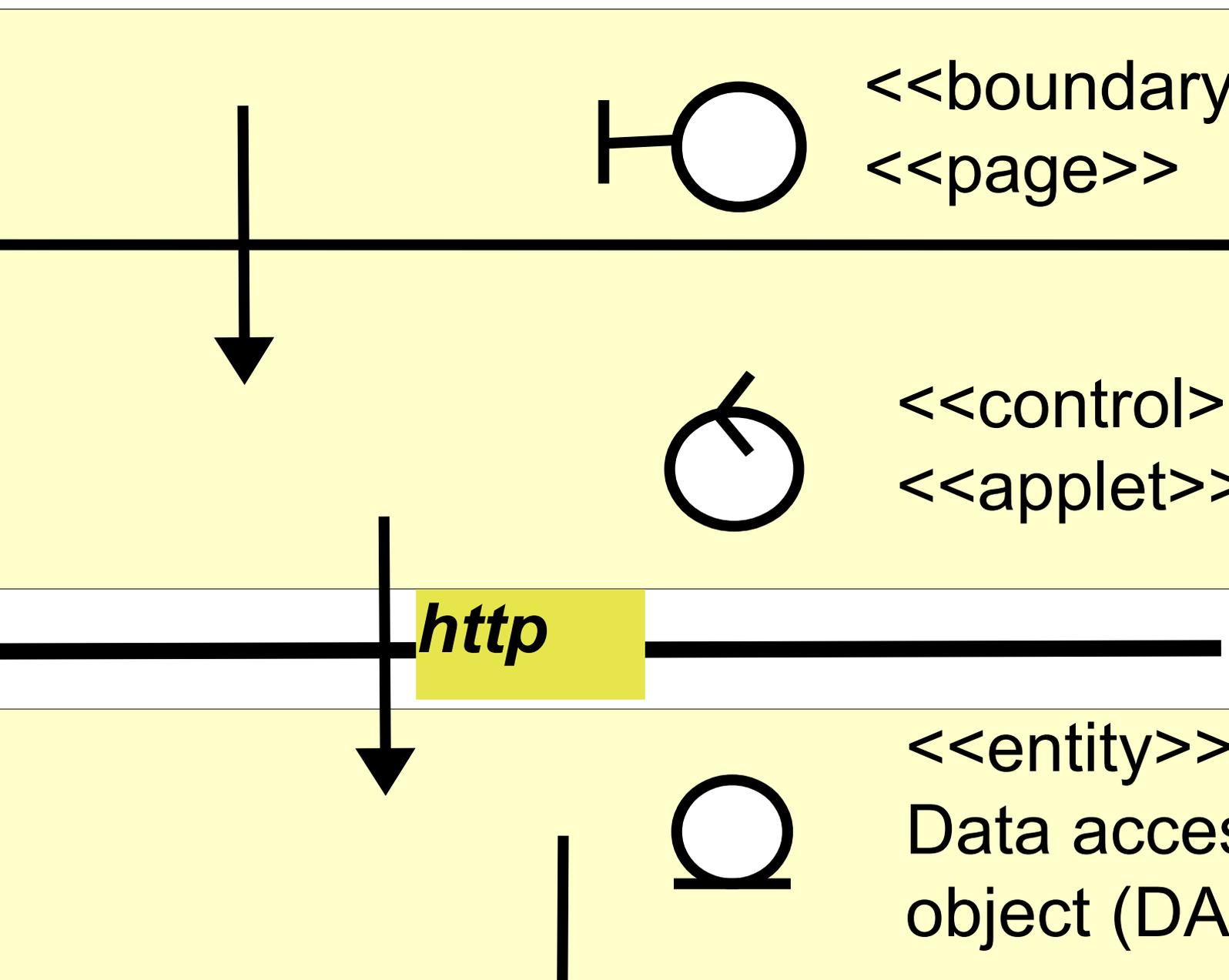
tion

e architecture for business-oriented

..

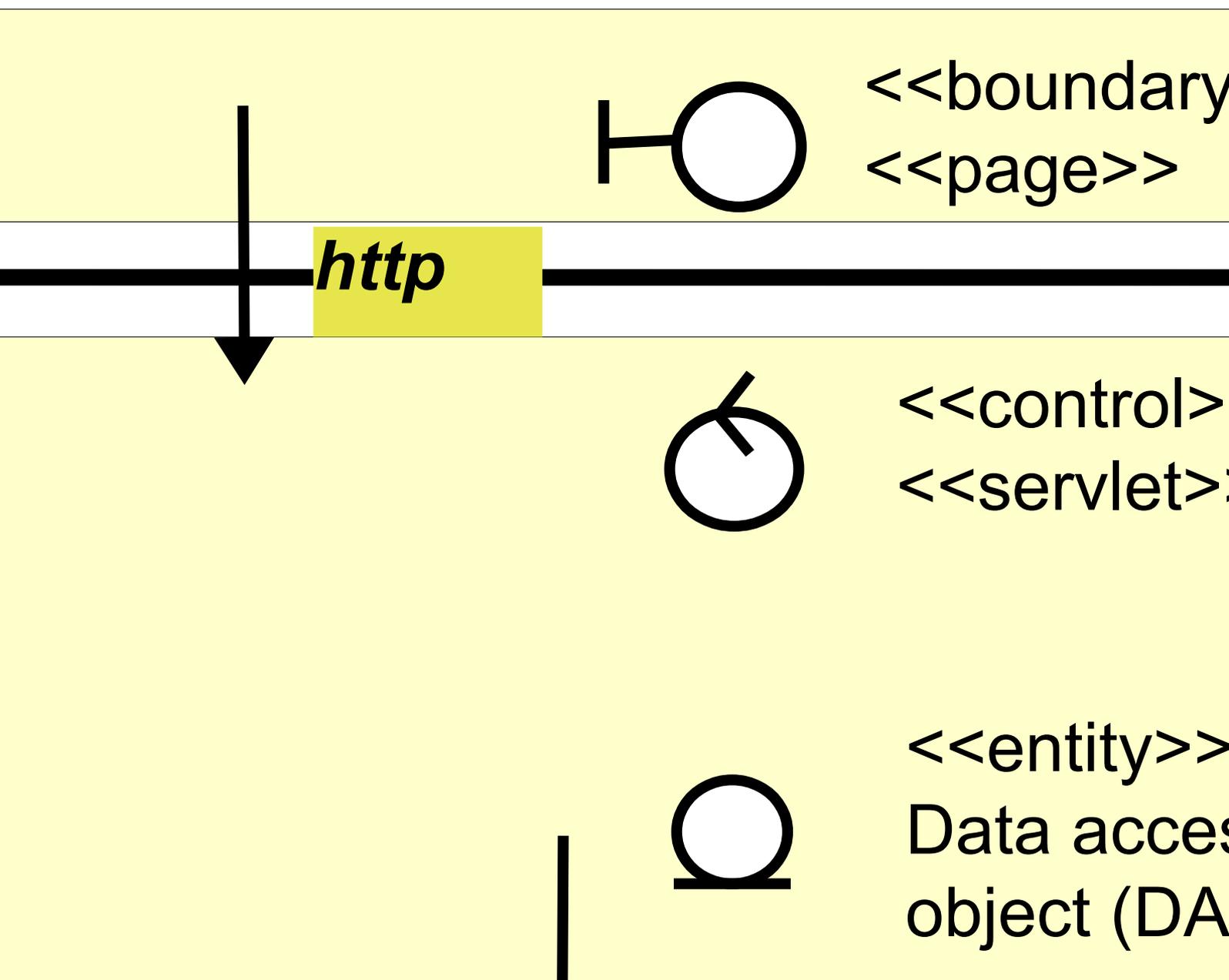
# System (Thick Client)

http-based middleware, in which GUI and control are on the client, data is managed on the server



# System (Thin Client)

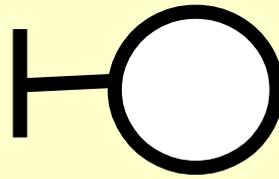
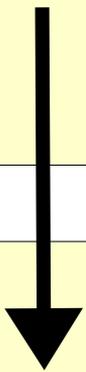
http-based middleware, in which GUI logic and data is managed on the server



# Workflow Language

L) describe the top-level of the application

led by the workflow

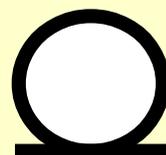
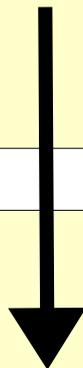


<<boundary>>  
<<page>>



<<control>>

components

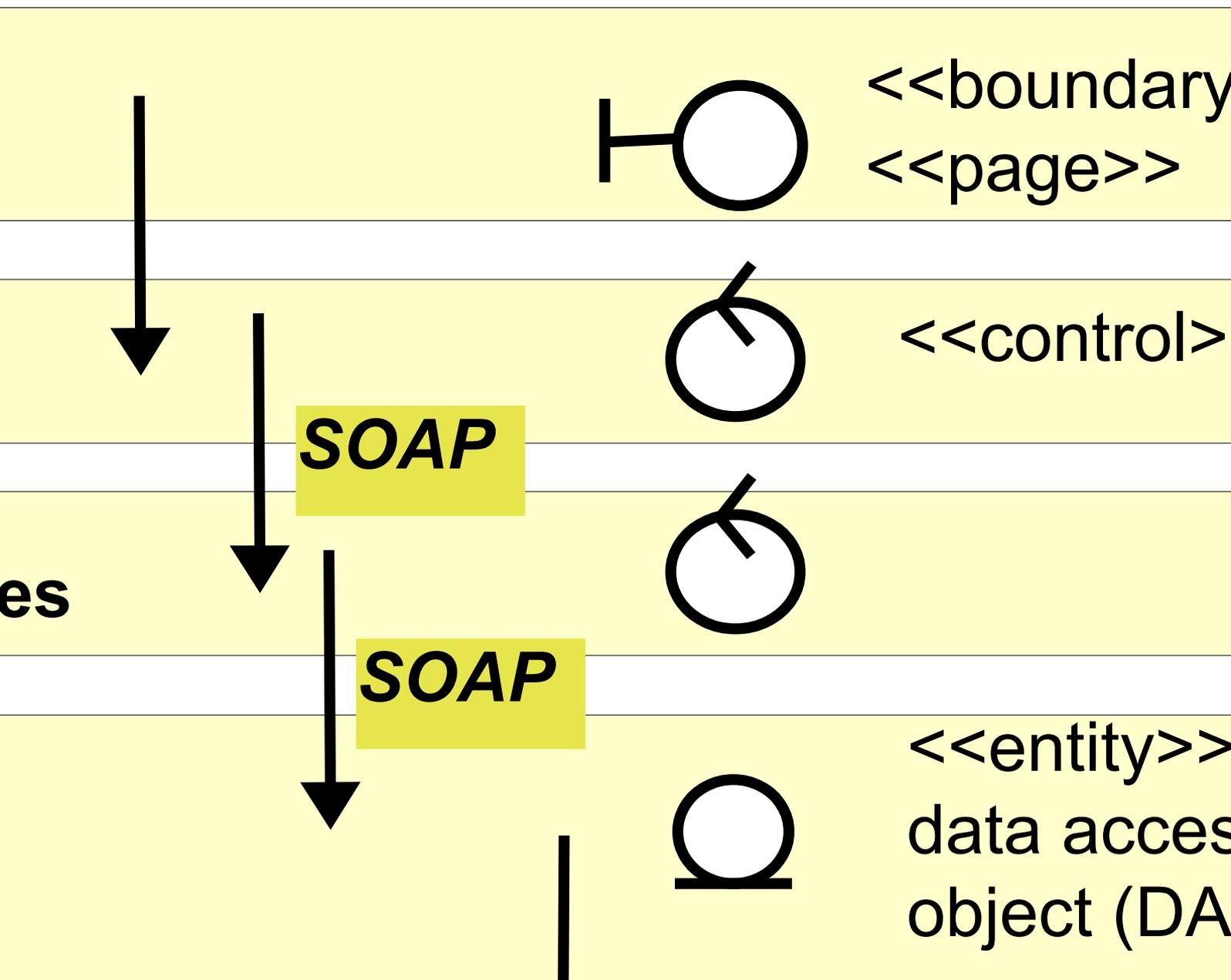


<<entity>>  
data access  
object (DAO)

# Workflow Language

L) describe the top-level of the application

led by the workflow via SOAP protocol



# Why are Layered Architectures Successful?

44

- ▶ Layered architectures require an acyclic USES relationship
- ▶ They are successful,
  - Because the dependencies within the system are structured as a dag
  - System is structured
  - Internals of layers can be abstracted away



# What Have We Learned

45

- ▶ Designing the global *architectural style* of your application is important (Architekturentwurf)
- ▶ Layers play an important role
- ▶ The USES (relies-on) relation is different from is-a (inheritance) and part-of (aggregation)
  - It deals with prerequisites for correct execution
  - Can be used to layer systems, if it is acyclic
- ▶ Examples of architectural styles with acyclic USES relation:
  - The BCED 4-tier architecture
  - Layered abstract machines for interactive applications
  - Layered behavioral state machines
  - Both styles can be combined



Software is always  
way as the organ

# The End

47

Prof. U. Alsmann, Softwaretechnologie, TU Dresden



# Anhang 41.A Entwurfsmuster Fassade zur Reduktion von Kopplung

48

... Ein Entwurfsmuster im Geiste Parnas  
(Wdh.)

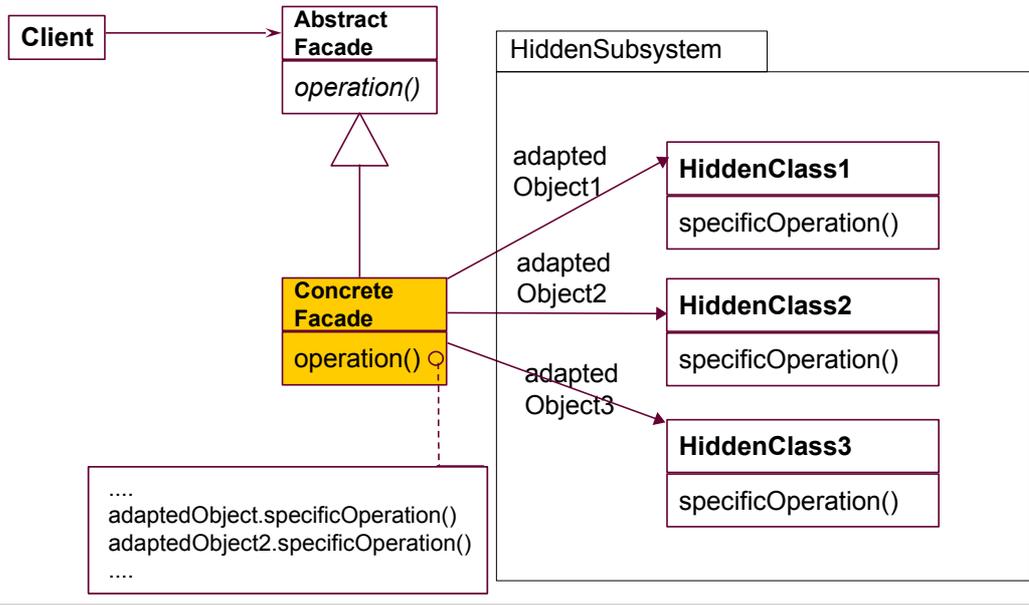


- ▶ Eine *Fassade (Facade)* ist ein Objektadapter, der ein komplettes Subsystem verbirgt
  - Die Fassade bildet die eigene Schnittstelle auf die Schnittstellen der verkapselten Objekte ab
  - Eine UML-Komponente ist gleichzeitig eine (einfache) Fassade. Die Delegationskonnektoren werden 1:1 an innere Komponenten delegiert; interne Adapter können adaptieren

# Fassaden verbergen Subsysteme

50

- ▶ Eine Fassade bietet eine *Sicht* auf ein Subsystem an. Es darf mehrere Sichten geben, nur keinen direkten Zugriff auf die inneren Objekte

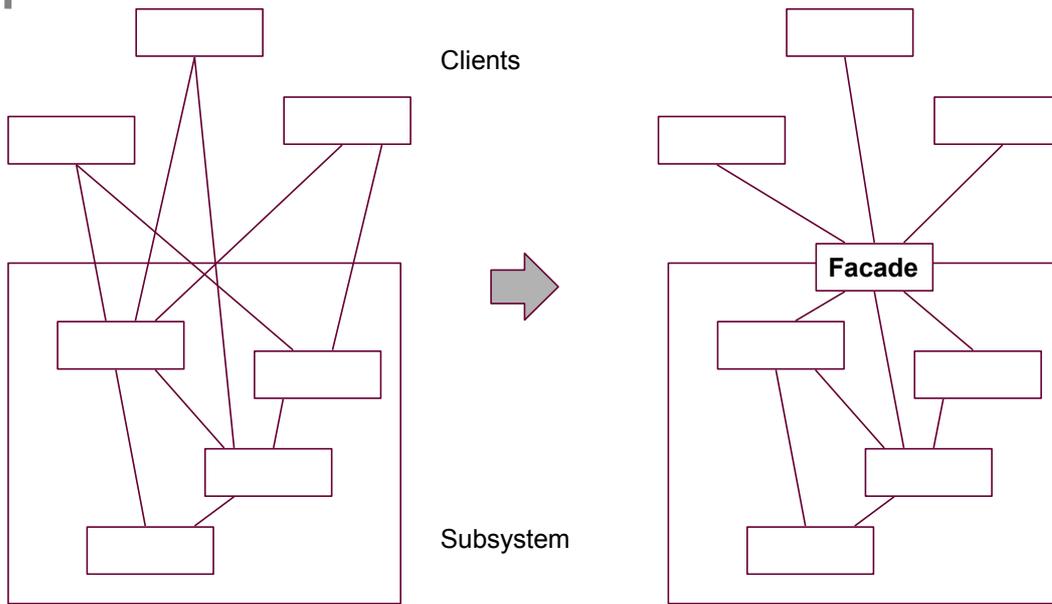


# Restrukturierung hin zur Fassade

51

- ▶ Fassaden entkoppeln; Subsysteme können leichter ausgetauscht werden (Variabilitätsmuster)

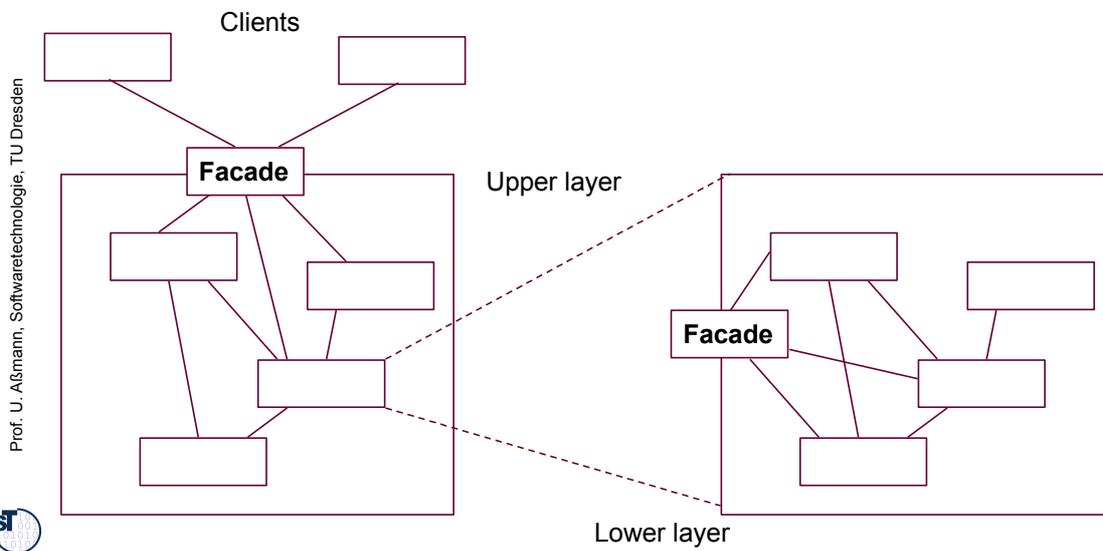
Prof. U. Alßmann, Softwaretechnologie, TU Dresden



# Fassaden und Schichten

52

- ▶ Falls einzelne Klassen eines Subsystems wieder Fassaden sind, entstehen *fassadengeschützte Schichten*



# 41.B Entwurfsmuster

## Fabrikmethode (FactoryMethod)

53

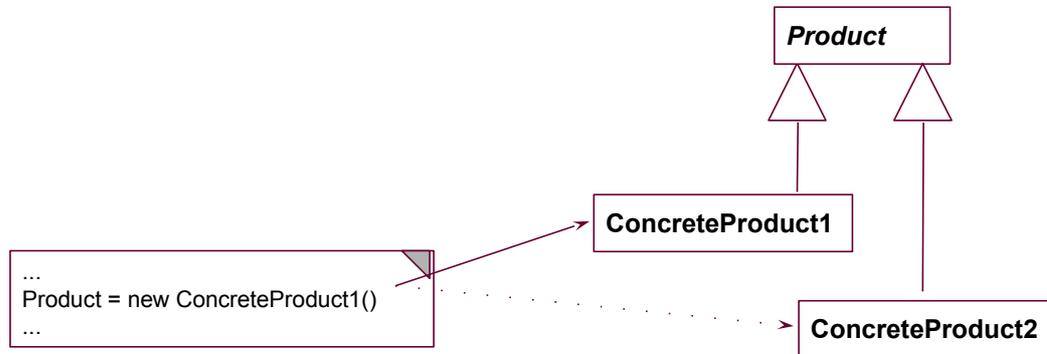
(Wdh.)  
zur polymorphen Variation von Komponenten (Produkte)  
und zum Verbergen von Produkt-Arten



# Problem der Fabrikmethode

54

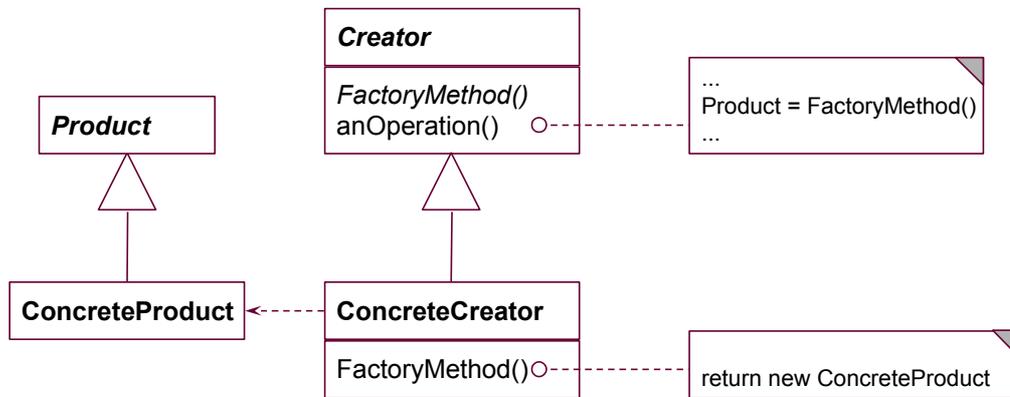
- ▶ Wie variiert man die Erzeugung für eine polymorphe Hierarchie von Produkten?
- ▶ Problem: Konstruktoren sind nicht polymorph!



# Struktur Fabrikmethode

55

- ▶ FactoryMethod ist eine Variante von TemplateMethod, zur Produkterzeugung



# Factory Method (Polymorphic Constructor)

56

- ▶ Abstract creator classes offer abstract constructors (polymorphic constructors)
  - Concrete subclasses can specialize the constructor
  - Constructor implementation is changed with allocation of concrete Creator

```
// Abstract creator class
public abstract class Creator {
    // factory method
    public abstract Set createSet(int n);
}
```

```
public class Client {
... Creator cr = [... subclass]..
    public void collect() {
        Set mySet = Creator.createSet(10);
        ....
    }
}
```

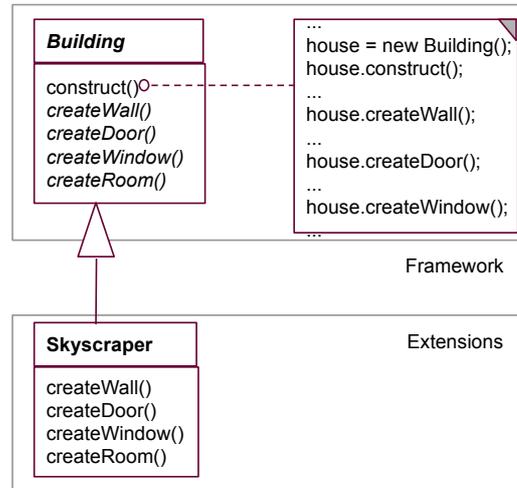
```
// Concrete creator class
public class ConcreteCreator extends Creator {
    public Set createSet(int n) {
        return new ListBasedSet(n);
    }
    ...
}
```



# Beispiel FactoryMethod

57

- ▶ Rahmenwerk für Gebäudeautomation
  - Klasse Building hat eine Schablonenmethode zur Planung von Gebäuden
  - Abstrakte Methoden: createWall, createRoom, createDoor, createWindow
- ▶ Benutzer können Art des Gebäudes verfeinern
- ▶ Wie kann das Rahmenwerk neue Arten von Gebäuden behandeln?



- ▶ Bilde createBuilding() als Fabrikmethode aus

```
// abstract creator class
public abstract class Building {
    public abstract
        Building createBuilding();
    ...
}
```

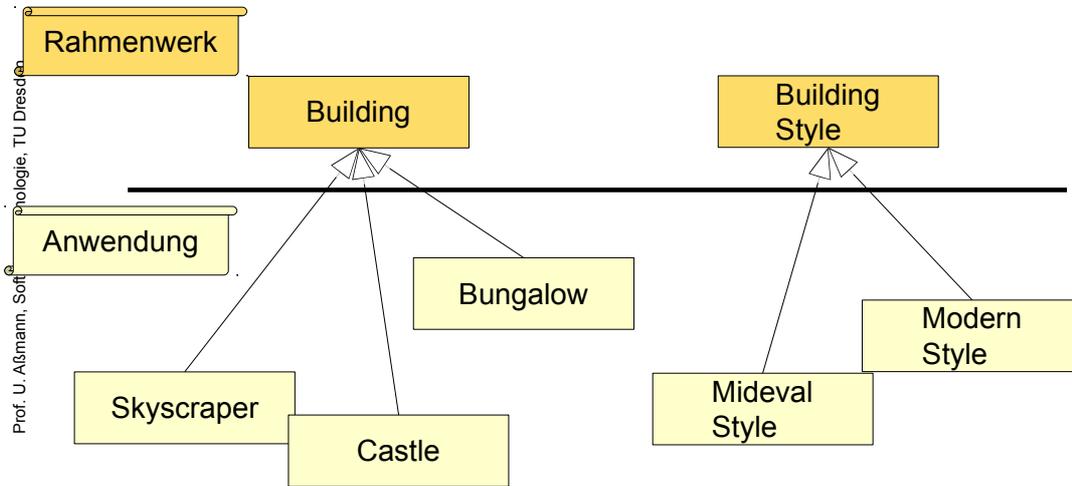
```
// concrete creator class
public class Skyscraper extends Building {
    Skyscraper() {
        ...
    }
    public Building createBuilding() {
        ... fill in more info ...
        return new Skyscraper();
    }
    ...
}
```



# Einsatz in Komponentenarchitekturen

60

- ▶ In Rahmenwerk-Architekturen wird die Fabrikmethode eingesetzt, um von oberen Schichten (Anwendungsschichten) aus die Rahmenschicht zu konfigurieren:



# 41.C

## Strategie (Strategy, Template Class)

61

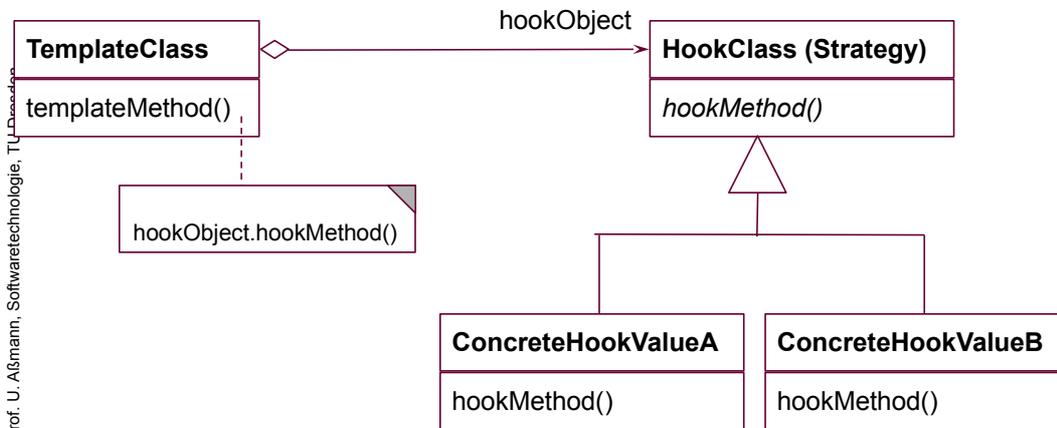
(Wdh.)



# Strategy (also called Template Class)

62

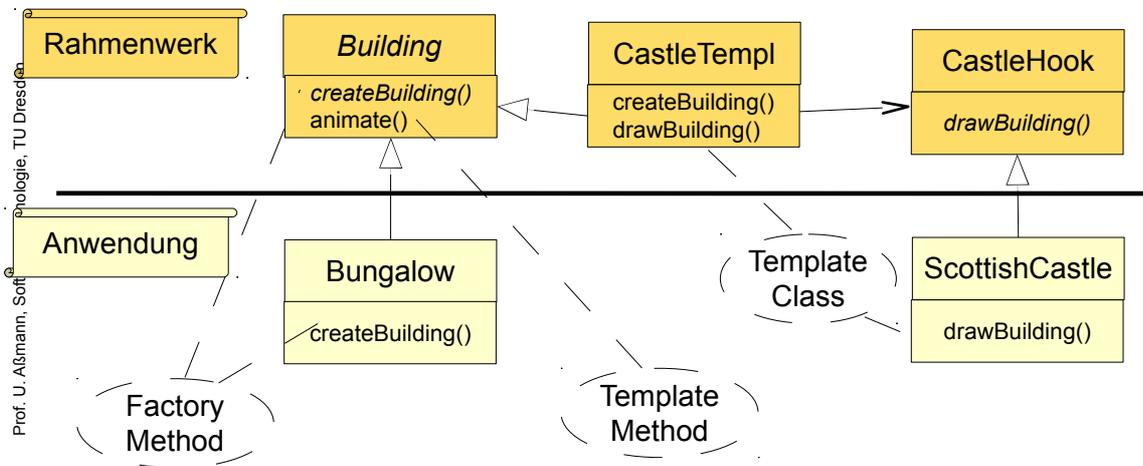
- ▶ Strategy wirkt wie TemplateMethod, nur wird die Hakenmethode in eine separate Klasse ausgelagert
- ▶ Zur Variation der Hakenklasse (und -methode)



# Kombinierter Einsatz in Rahmenwerken

63

- ▶ FactoryMethod variiert den Konstruktor
- ▶ TemplateMethod oder Strategy (TemplateClass) variiert die Hookmethode
- ▶ Bridge (s. später) variiert die TemplateMethode



Prof. U. Alsmann, Software-Technologie, TU Dresden





## 41.D Entwurfsmuster Einzelstück (Singleton)

---

64

zur globalen Konfiguration einer Komponente  
oder Schicht  
(Wdh)



Softwaretechnologie, © Prof. Uwe Aßmann  
Technische Universität Dresden, Fakultät Informatik

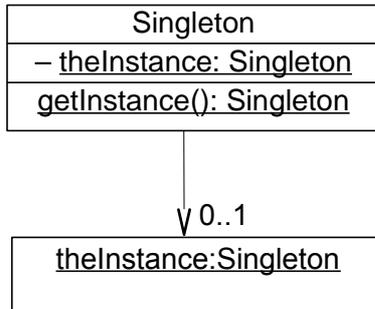
Split off from 3-design-intro March 2003, to fit  
to PUM-I.

# Entwurfsmuster Einzelstück (Singleton)

65

- ▶ Gesucht: globales Objekt, das global oder innerhalb einer Laufzeitkomponente (z.B. Schicht) Daten, z.B. Konfigurationsdaten, vorhält
- ▶ Idee:
  - Erstelle eine Klasse, von der genau ein Objekt existiert (Invariante)
  - Erstelle einen artifizienten Konstruktor (Fabrikmethode), der oft aufgerufen werden kann, aber die Invariante sicherstellt
  - Eigentlicher Konstruktor wird *verborgen* (*private*)
  - Austausch der Konfiguration durch Unterklassenbildung (Variabilität)

Prof. U. Alsmann, Softwaretechnologie, TU Dresden



```
class Singleton {
    private static Singleton theInstance =
    null;
    private Singleton () {}
    public static Singleton getInstance() {
        if (theInstance == null)
            theInstance = new Singleton();
        return theInstance;
    }
}
```

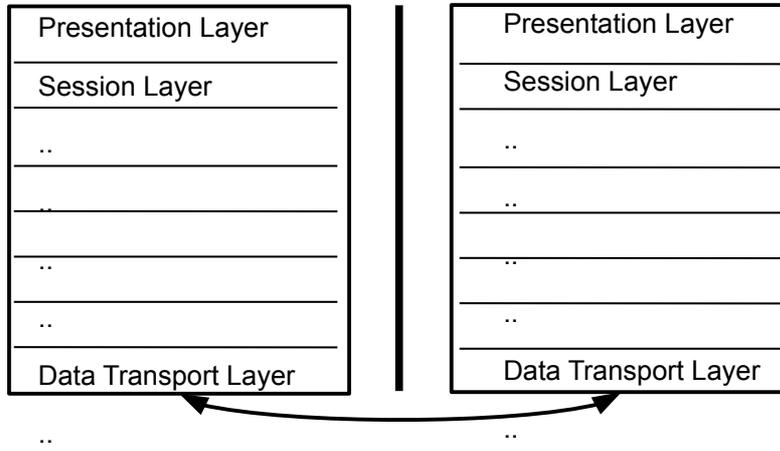




# Example: ISO-OSI 7 Layers Network Architecture

67

- ▶ Every layer contains an abstract machine (set of operations)



# Example: Operating Systems

68

UNIX,  
Linux,  
Android

User Space

Kernel

Apple-  
UNIX,  
iOS

User Space

Kernel

Microkernel (Mach)

Prof. U. Asmann, Software-Technologie, TU Dresden

Windows NT/XP:

User Space

Kernel

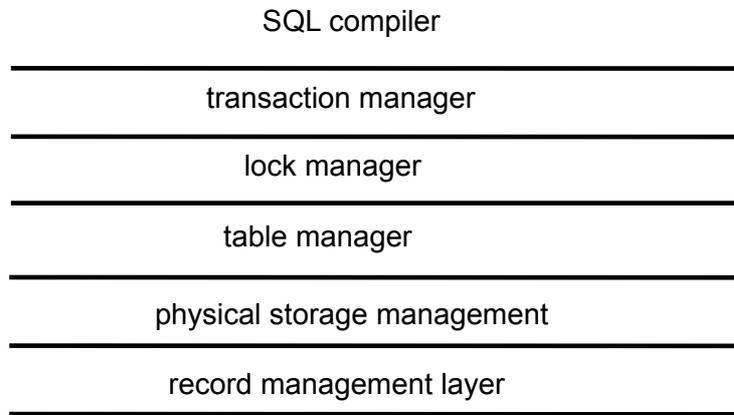
Hardware Abstraction Layer (HAL)



# Example: Database Systems

69

Prof. U. Alsmann, Softwaretechnologie, TU Dresden



# Repet.: BCD/BCED Classification

70

- ▶ Boundary classes: <<boundary>>  

  - Represent an interface item that talks with the user
  - May persist beyond a run
- ▶ Control class: <<control>>  

  - Controls the execution of a process, workflow, or business rules
  - Does not persist
- ▶ Entity class: <<entity>>  

  - Describes persistent knowledge. Caches a persistent object from a database (data access object, DAO)
- ▶ Database class <<database>>  

  - Adapter class for the database
  - Often, Entity and Database classes are unified
- ▶ **BCD/BCED is linked with the 3-tier architecture**

