

# Teil IV: Objektorientierter Entwurf

## 41 Grundlegende Architekturprinzipien

Prof. Dr. rer. nat. Uwe Aßmann  
Institut für Software- und  
Multimediatechnik  
Lehrstuhl Softwaretechnologie  
Fakultät für Informatik  
TU Dresden  
Version 19-0.3, 06.07.19

- 1) Geschäftsmodelle
- 2) Architekturprinzipien
- 3) Architekturdiagramme
- 4) Architekturstile
- 1) Geschichtete Architekturen
- 5) Architektur mit Perspektivenmodellen



- ▶ Zuser Kap 10.
- ▶ Ghezzi 4.1-4.2
- ▶ Pfleeger 5.1-5.3
- ▶ ST für Einsteiger 5.3, 8
- ▶ Erste wissenschaftliche Papiere zur Lese:
  - Parnas, D.L. On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM 15 (12): 1053–58. December 1972, doi:10.1145/361598.361623
  - Phillipe B. Kruchten. The 4+1 view model of architecture. IEEE Software, Nov. 1995. doi:10.1109/52.469759

# Exkurs: Wie findet man ein Papier?

- ▶ <http://scholar.google.de>
- ▶ <http://iinwww.ira.uka.de/bibliography/index.html#search>

# Sekundäre Literatur

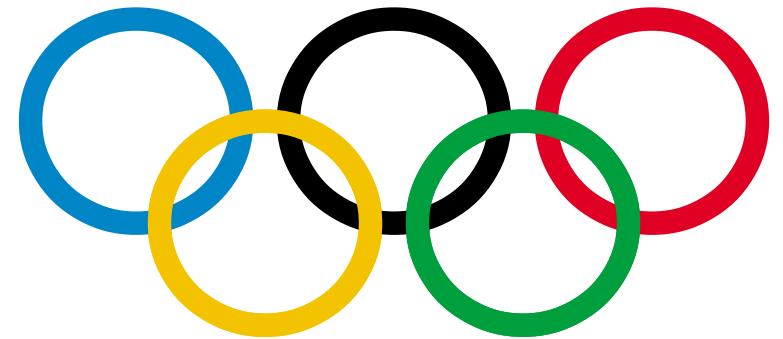
- ▶ David J. Parnas. On a buzzword: hierarchical structure. Proceedings IFIP Congress 1974, North-Holland, Amsterdam.
- ▶ Christine Hofmeister, Robert L. Nord, and Dilip Soni. Describing software architecture with UML. In Patrick Donohoe, editor, WICSA, volume 140 of IFIP Conference Proceedings, pages 145-160. Kluwer, 1999.
  - Christine Hofmeister, Robert Nord, and Dilip Soni. Applied Software Architecture. Addison-Wesley, Reading, MA, 2000.
- ▶ Johannes Siedersleben. Moderne Softwarearchitektur. Umsichtig planen, robust bauen mit Quasar. dpunkt-Verlag, 2004.

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

- 1) 40: Überblick
- 2) **41: Einführung in die objektorientierte Softwarearchitektur**
  - 1) Architekturprinzipien, Architekturstile, Perspektivenmodelle
  - 2) Modularität und Geheimnisprinzip
  - 3) BCD-Architekturstil (3-tier architectures)
- 3) **42: Verfeinerung mit querschneidender Objektorichung**
- 4) 43: Architektur interaktiver Systeme
- 5) 44: Punktweise Verfeinerung von Lebenszyklen
  - Verfeinerung von verschiedenen Steuerungsmaschinen



- ▶ Was ist der Unterschied zwischen Programmieren im Kleinen und *Programmieren im Großen*?
- ▶ Wie hilft Architektur, Produktlinien aufzubauen?
- ▶ Ringe sind keine Schichten
- ▶ Was sind die 5 olympischen Ringe der Software?



**OUR GOALS**

## 41.1 Geschäftsmodelle für Softwarefirmen

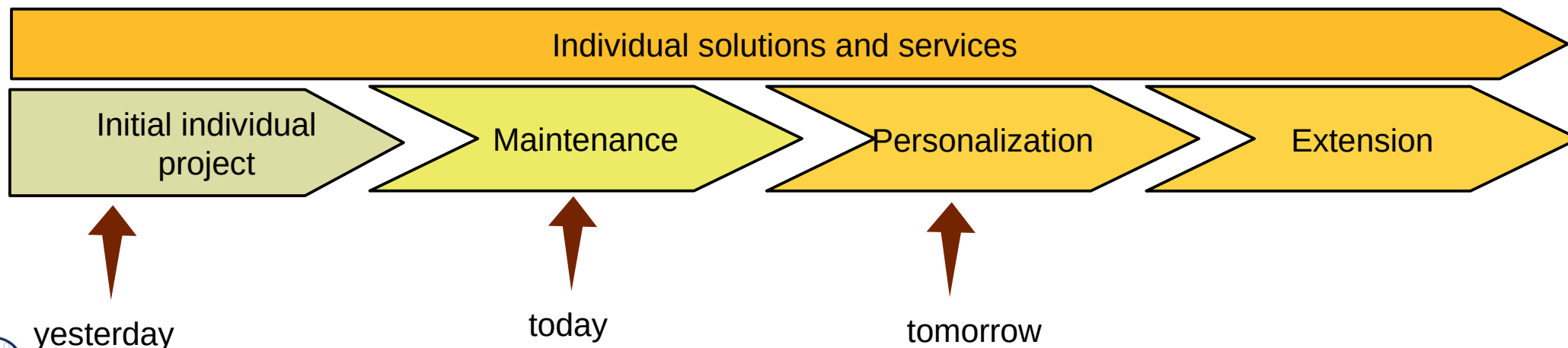
- ▶ → Kurs “Design Patterns and Frameworks (DPF)” in WS
- ▶ → Kurs “Software as a Business (SAAB)” in WS
- ▶ → Kurs “Component-Based Software Engineering (CBSE)” in WS



# Geschäftsmodell Individualprojekte und Dienstleistung (Individual Solutions and Services, iSaS)

Good software architecture simplifies the management of solutions and services (iSaS)

- ▶ **Individual project (solution to a problem of customer)**
- ▶ **Maintenance (Wartung)**
- ▶ **Configuration, Personalization**
- ▶ **Extension**





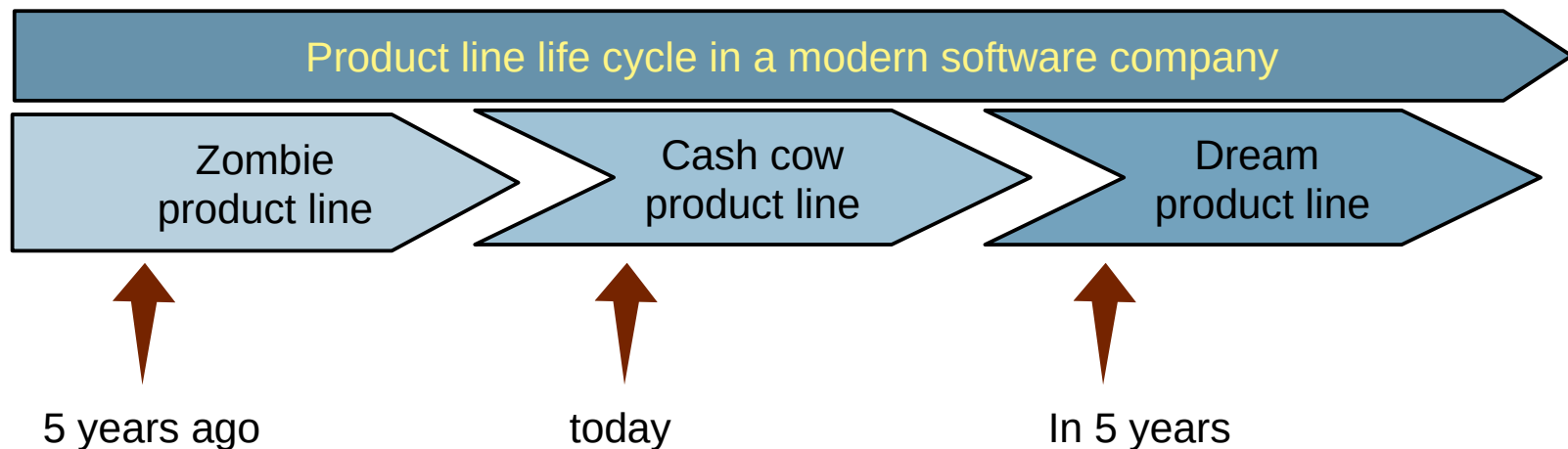
# Geschäftsmodell Produktfamilien (Product Families, Product Lines, SPL)

9

Softwaretechnologie (ST)

Good software architecture simplifies the management of SPL

- ▶ **Variability Points**
  - Exchanging parts easily
  - Variation, variability, complex parameterization
- ▶ **Extension Points**
  - Software must be extended
- ▶ **Glue Code** (adaptation overcoming architectural mismatches)
  - Coupling software that was not built for each other



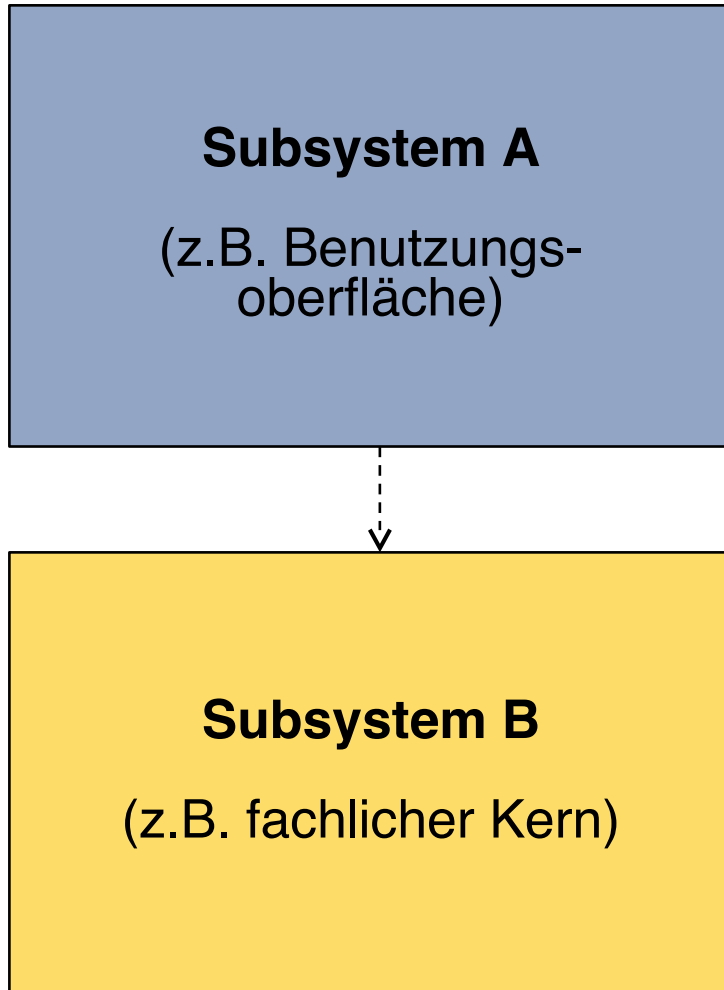
## 41.2 Architekturprinzipien

Architekturprinzipien bilden allgemeine Gestaltungsregeln (“best practices”) für die Architektur.

Entwurfsmuster realisieren diese Gestaltungsregeln in einem wiederverwendbaren Entwurf.



# 41.2.1 Architekturprinzip: Hohe Kohäsion + Niedrige Kopplung



- ▶ **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.

# 41.2.2 Architekturprinzip: Veränderungsorientierter Entwurf mit Modularität

- ▶ Zu ihrer besseren Wiederverwendbarkeit sollte Software in *Komponenten (Module)* eingeteilt werden (*Modularität*)
- ▶ Eine **Komponente (Modul)** im allgemeinen Sinne ist eine Wiederverwendungseinheit:
  - die Funktionalität mit hoher Kohäsion gruppiert
  - die *angebotene* und *benötigte* Schnittstellen besitzt, um lose Kopplung zu unterstützen:
    - keine impliziten, nur explizit in der Schnittstelle angegebene Abhängigkeiten zu anderen Komponenten
- ▶ Vorteile der **Modularität**:
  - Unabhängigkeit im Software-Entwicklungsprozess:
    - Komponente kann unabhängig von anderen entwickelt werden
    - Komponenten können einzeln getestet werden (Einheitstest, unit test)
    - Fehler können zu individuellen Komponenten verfolgt werden
  - Komponenten können ausgetauscht werden, ohne dass das System zusammenbricht (Ersetzbarkeit)
    - weil angebotene und benötigte Schnittstellen unterschieden werden

# Bemerk.: Modularität mit Komponentenmodellen und Kompositionssystemen

- ▶ Es gibt nicht nur die UML-Komponente...sondern viele verschiedene *Komponentenmodelle*:
  - Module einer modularen Programmiersprache (Modula, Ada)
  - *Binäre Module*, z.B. class-Files oder .o-Files
  - Klassen, Kollaborationen und Konnektoren in objektorientierten Sprachen
  - UML-Komponenten
  - Ganze Schichten eines Systems, insofern sie in eine Komponente gekapselt werden können (wie z.B. die TLA)
  - 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)

## 41.2.3. Architekturprinzip: Flexible Evolution mit dem Geheimnisprinzip

Parnas' Prinzip des Entwurfs mit dem **Geheimnisprinzip** gilt für alle Komponentenmodelle (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 (**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
- 4) Definiere *angebotene* und *benötigte* Schnittstellen

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

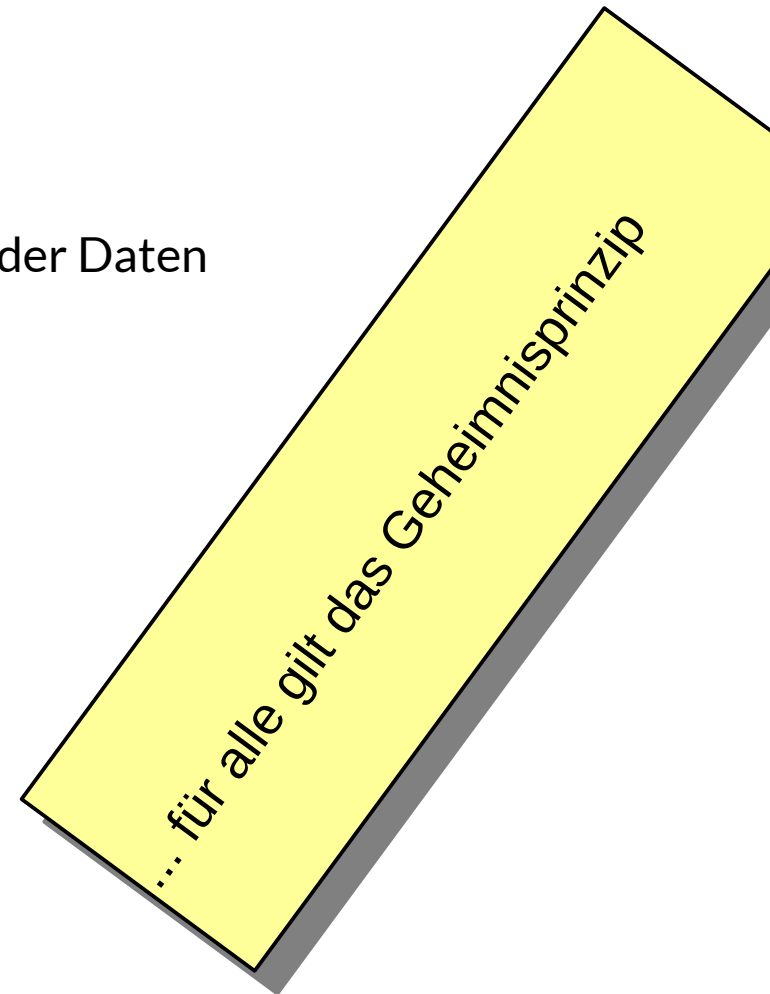
- ▶ Arbeitsweise von Algorithmen
- ▶ Datenformate
  - Texte, Dokumente, Bilder
- ▶ Datentypen
  - Abstrakte Datentypen und ihre konkrete Implementierung
- ▶ Benutzerschnittstellenbibliotheken
- ▶ Bearbeitungsreihenfolgen
- ▶ Verteilung
- ▶ Persistenz

# Verschiedene Arten von Komponenten/Modulen (in verschiedenen Sprachen)

16

Softwaretechnologie (ST)

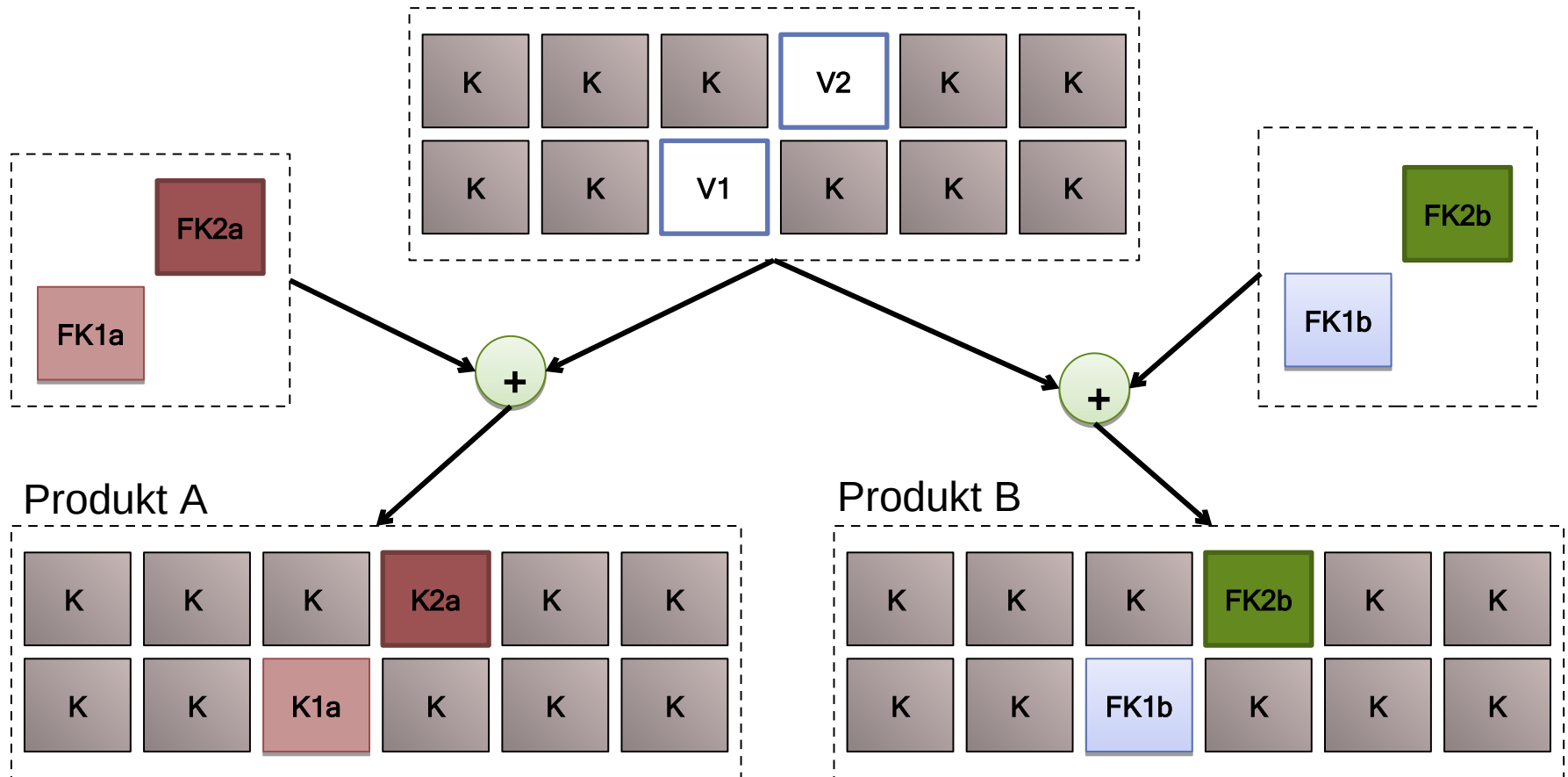
- ▶ Funktionale Module ohne Zustand
  - sin, cos, BCD arithmetic, gnu mp,...
- ▶ Daten-Repositoryen
  - Verbergen Repräsentation, Zugriff und Zustand der Daten
  - Symboltabellen, Materialcontainer, ...
- ▶ Abstrakte Datentypen
- ▶ Singletons (Konfigurationskomponenten)
  - Klassen mit einer einzigen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





# 4.2.4 Architekturprinzip Gemeinsames vs Spezifisches (Variabilitätspunkte)

- ▶ Für eine Produktlinie müssen die Komponenten, die allen Produkten gemeinsam sind, von denen getrennt werden, die spezifisch sind
- ▶ Auszutauschende Komponenten im *Framework* nennt man *Variabilitätspunkte (V)*, die mit jeweils unterschiedlichen *Füllselkomponenten (FK)* aufzufüllen sind



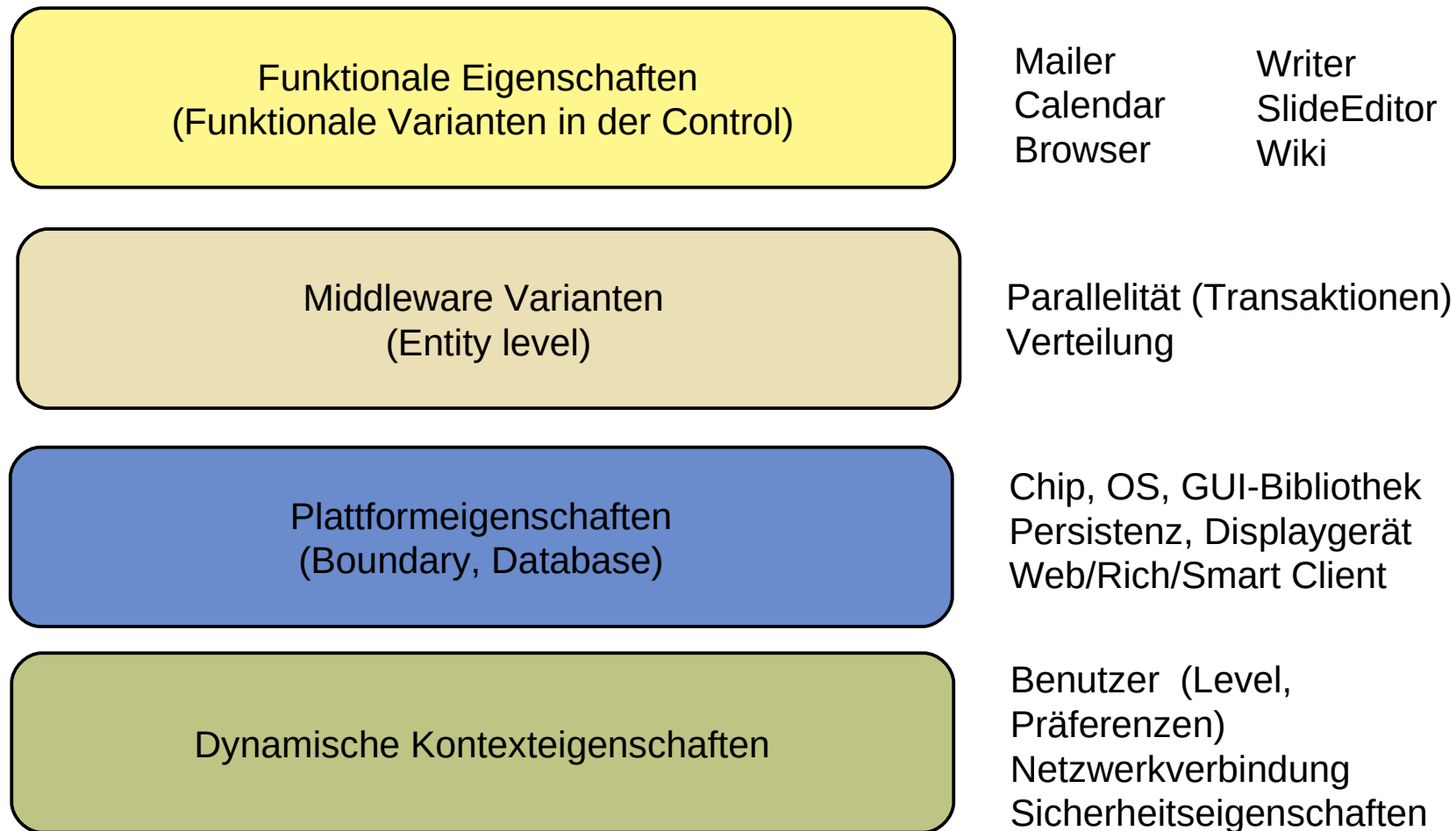
# Alle Variabilitätsmuster in objektor. Sprachen nutzen das Geheimnisprinzip

- ▶ Viele Entwurfsmuster (z.B. TemplateMethod) sind vom Parnas-Prinzip inspiriert.
- ▶ Sie sind **Variabilitätsmuster**, d.h., sie verbergen bestimmte Geheimnisse und erlauben dann, die Implementierungen auszutauschen (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

Variabilitätsmuster repräsentieren Variabilitätspunkte der Architektur

# 41.2.5. Architekturprinzip: Schichten von Variabilität

- ▶ **Software-Produktlinien** entstehen durch systematische Variation von Geheimnissen, wobei diese in *Schichten oder Aspekte* gruppiert werden

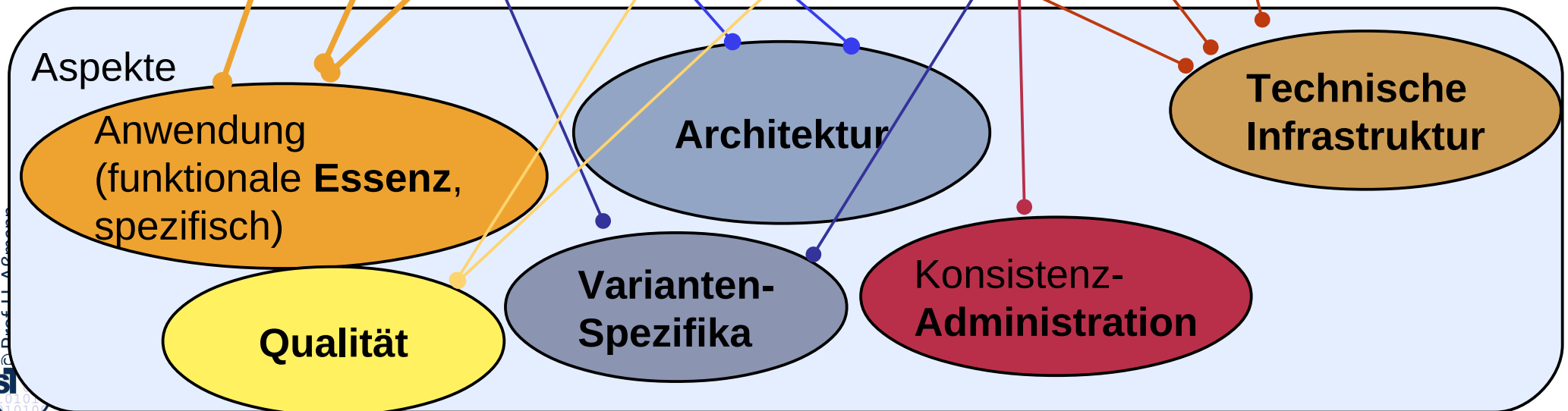


## 41.3 Architekturprinzip “Aspekttrennung”



# Wesentliche Aspekte eines Softwaresystems

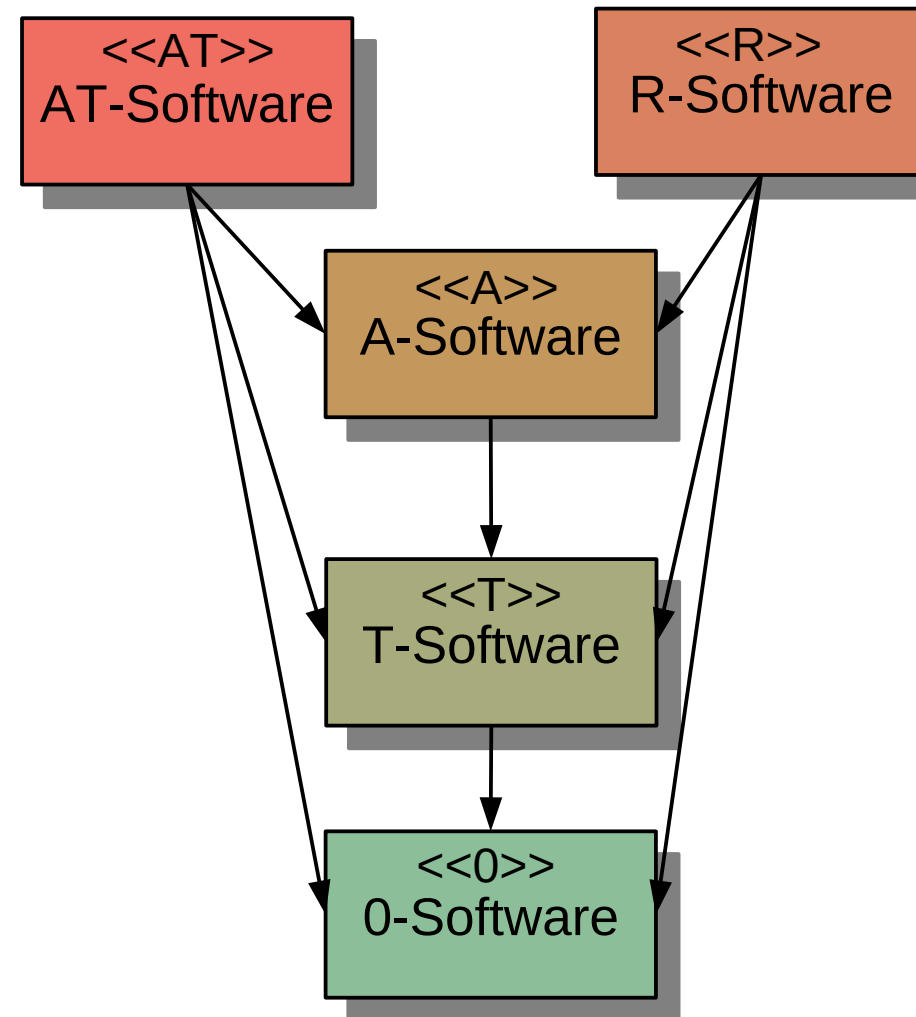
- ▶ Anwendungsspezifische Funktionen
- ▶ Benutzungsoberfläche
- ▶ Ablaufsteuerung
- ▶ Datenhaltung
- ▶ Infrastrukturdienste
  - Objektverwaltung
  - Interne Objekt- und Prozeßkommunikation
  - Verteilungsunterstützung
- ▶ Kommunikationsdienste
- ▶ Sicherheitsfunktionen
- ▶ Zuverlässigkeitsfunktionen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



## 4.3.1 Architekturprinzip Quasar: Trennung von Technik- und Anwendungskomponenten (Reuse Blood Groups, Blutgruppen)

Quasar definiert 4 Aspekte  
(Wiederverwendungs-Blutgruppen),  
Softwarekategorien für Komponenten, nach  
Wiederverwendbarkeit:

- ▶ 0: unabhängig von Anwendung und Technologie
  - JDK collections, C++ STL, GNU regexp
- ▶ A: anwendungs- oder domänenspezifisch.
  - Client, Customer, Account, Car, ...
- ▶ T: technologie-orientierte Schnittstelle, unabhängig von Anwendung
  - OSGI, JDBC, CORBA CosNaming
- ▶ AT: abhängig von Anwendung *und* Technologie
  - schwierig zu isolieren und wieder zu verwenden
- ▶ R: Repräsentationswechsel von Daten
  - Serialisierung, Deserialisierung, Verschlüsselung
  - Sprachwechsel (z.B. Java to Cobol)

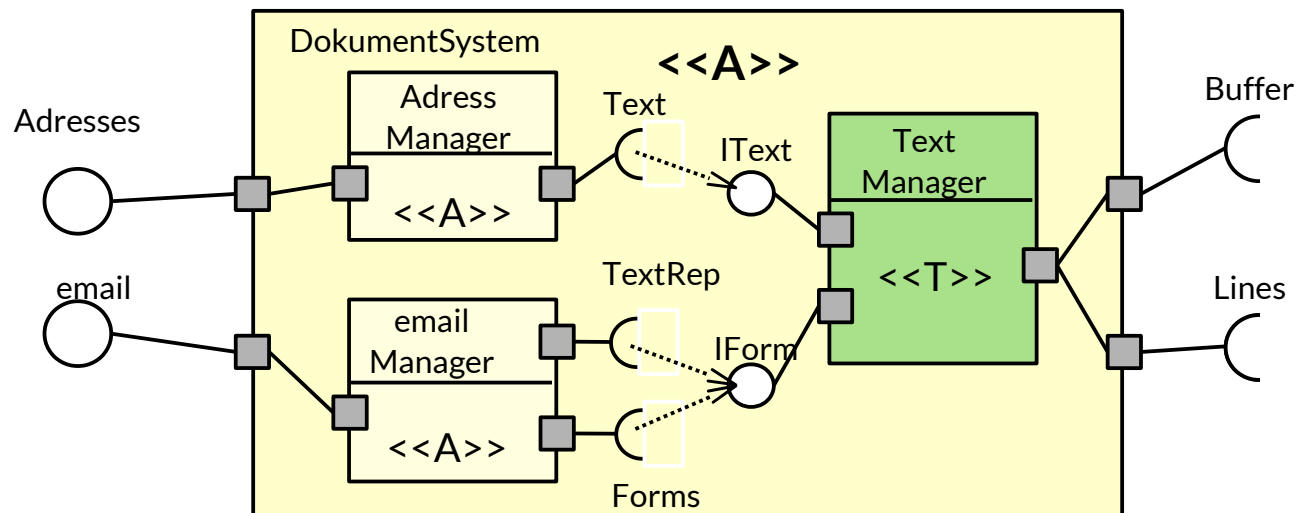


# Architekturprinzip Quasar: Trennung von Technik- und Anwendungskomponenten

23

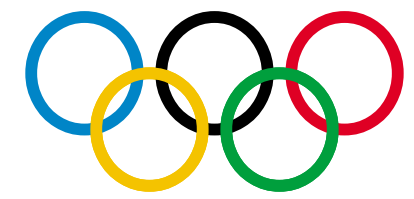
Softwaretechnologie (ST)

- ▶ Jede Komponente wird klassifiziert in Blutgruppen 0, T, A, AT, R
  - 0 – technologieunabhängige Algorithmen,
  - T – technologieabh. Komponenten,
  - A – Anwendungskomponenten,
  - R – Repräsentationwechselkomponenten

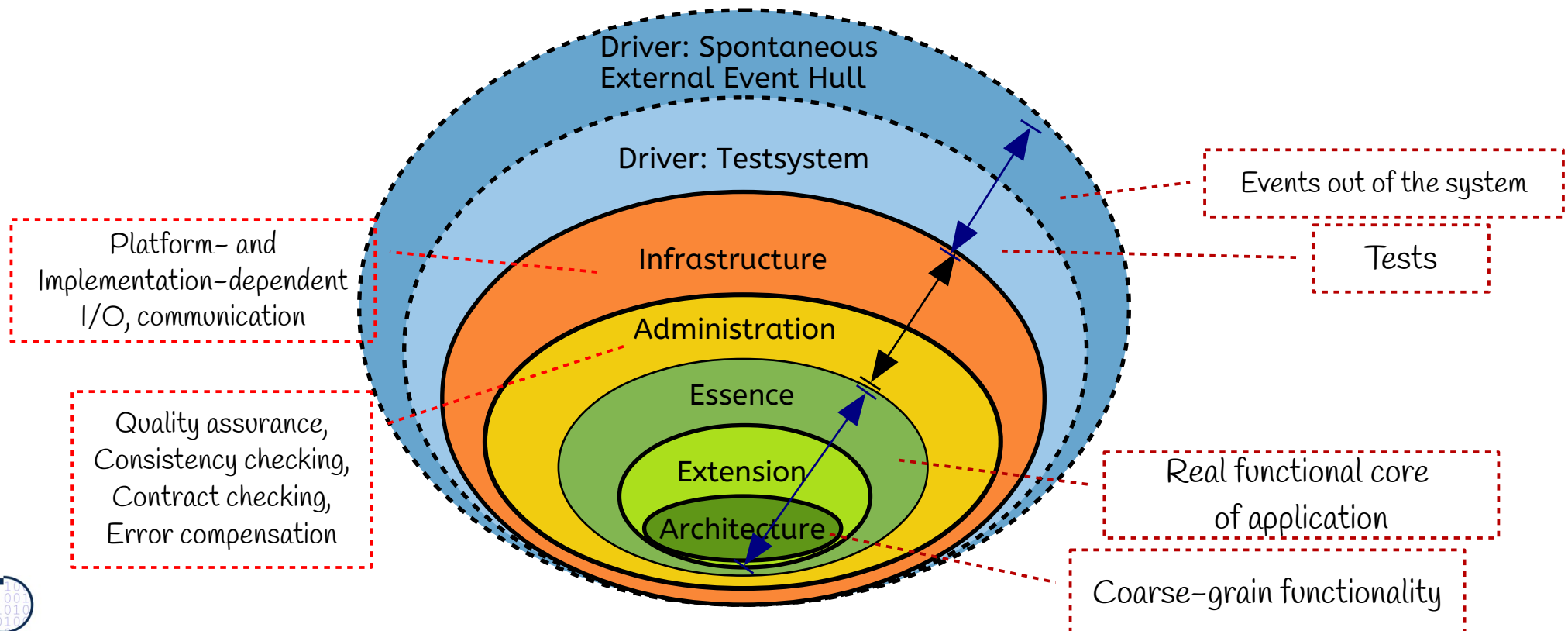


**[Siedersleben] Quasar-Wiederverwendungsgesetz:  
0- und T-Komponenten sind besser  
wiederverwendbar als Anwendungskomponenten.  
AT-Komponenten sind sehr schlecht wiederverwendbar.**

## 4.3.2. Olympische Dekomposition von Software in Ringe (Essentielle Dekomposition)



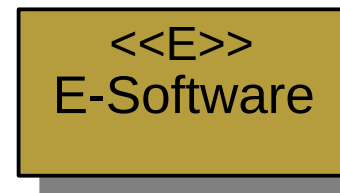
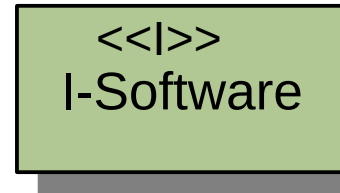
- ▶ **Software** hat 5 Ringe (*olympische* oder *essentielle Dekomposition* in 5 Aspekte):
  - **(Funktionale) Essenz** sind Funktionen unabhängig von der unterliegenden Technologie
    - **Architektur** ist ein Unter-Ring der Essenz, der grobkörnige Funktionalität liefert
  - **Administration** sichert die Qualität des Systems
  - **Infrastruktur (Middleware)** bietet die technologieabhängigen Funktionen an
  - **2 Externe Treiber.Ringe** treiben das System: entweder die Umgebung, die spontan Ereignisse und Eingabedaten generiert, oder das **Testsystem**





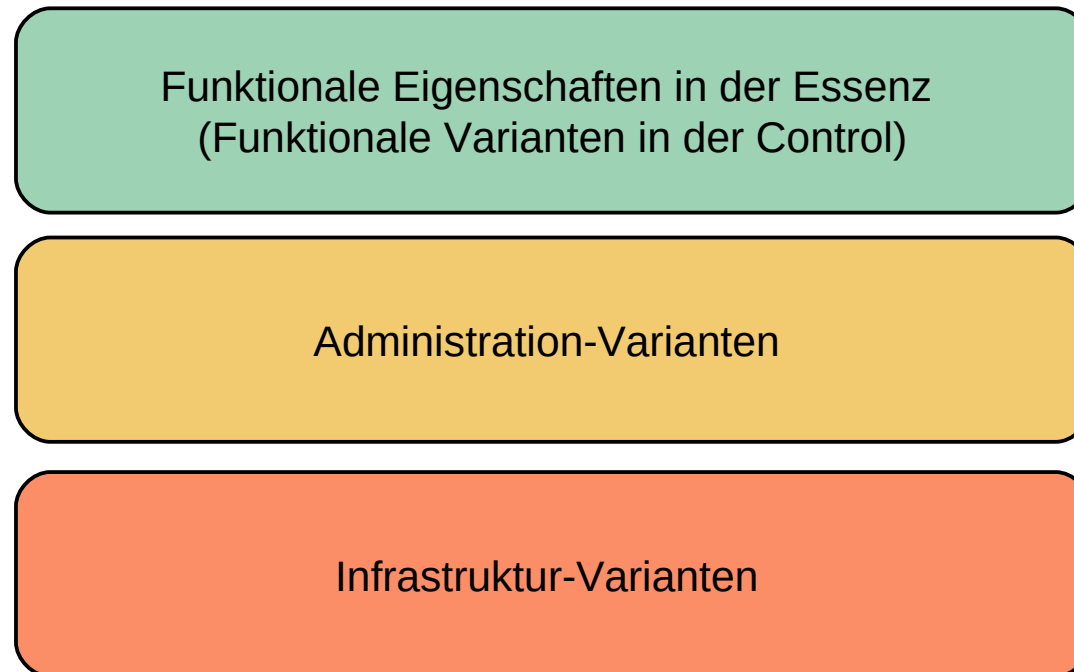
# Trennung von Ringen

- ▶ Olympische Dekomposition definiert 5 Aspekte, Softwarekategorien für Komponenten, nach Zweck:
  - E: Essenz
  - Arch: Architektur
  - Admin: Administration
  - I: Infrastruktur, Plattform
- ▶ Die Ringe durchdringen einander, d.h. bilden keine Schichten
- ▶ Jede Komponente wird **genau einem Ring** zugeordnet, ansonsten Mischmasch
- ▶ Ähnlich wie bei Quasar

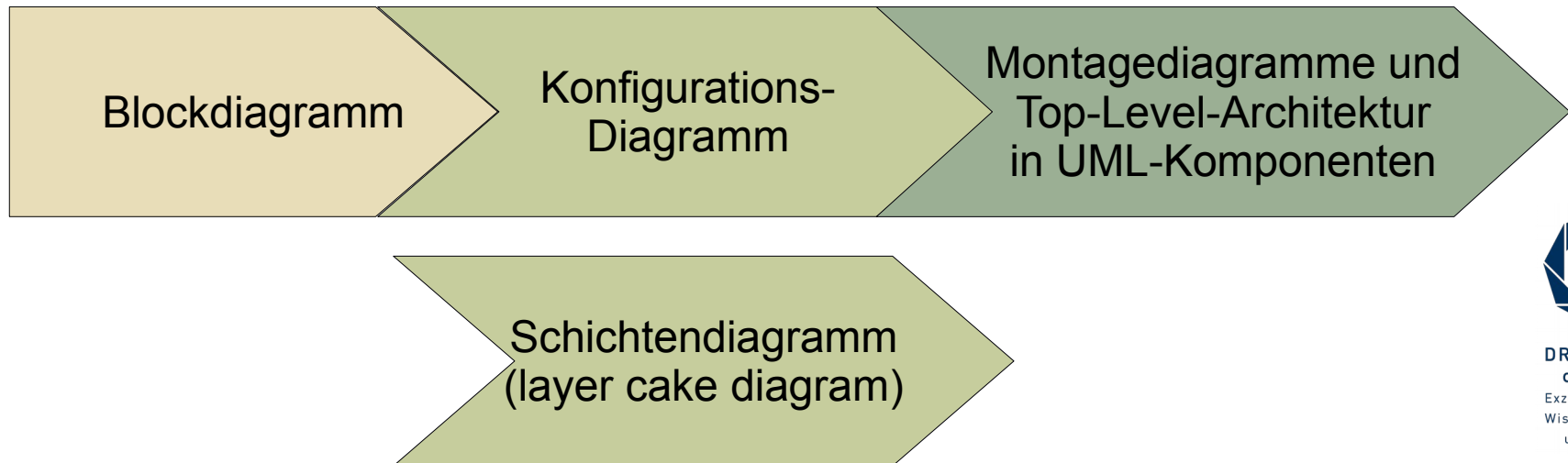


# 41.3.3. Architekturprinzip: Olympische Variabilität mit Ringen

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

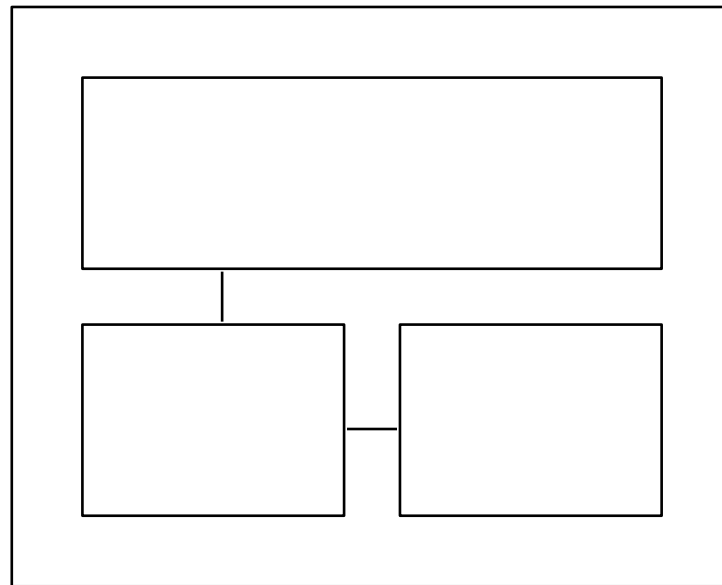


## 41.4 Diagrammarten für die logische Struktur



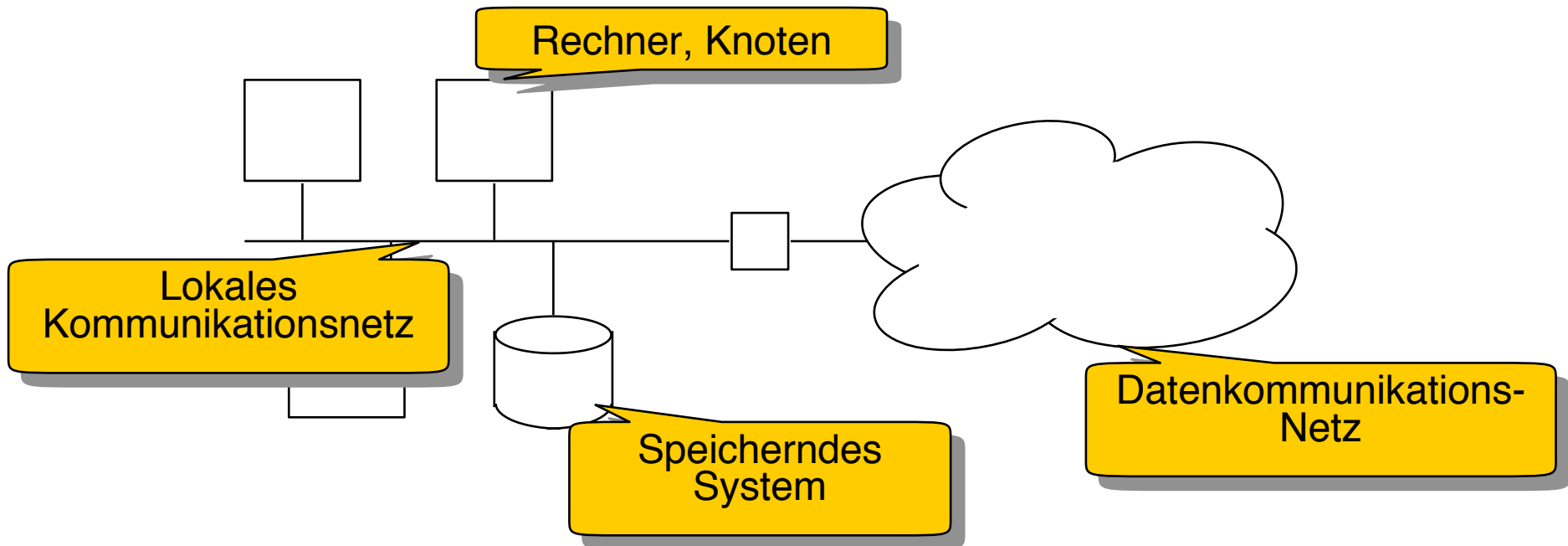
# Blockdiagramme zur logischen Struktur eines Systems

- ▶ Def.: Ein **Blockdiagramm** skizziert informell die logische Struktur einer Architektur
  - verbreitetes, informelles Hilfsmittel
  - Blockdiagramme sind kein Bestandteil von UML; Vorstufe von Montagediagrammen
  - **Blöcke** stellen UML-Komponenten *ohne Ports* dar

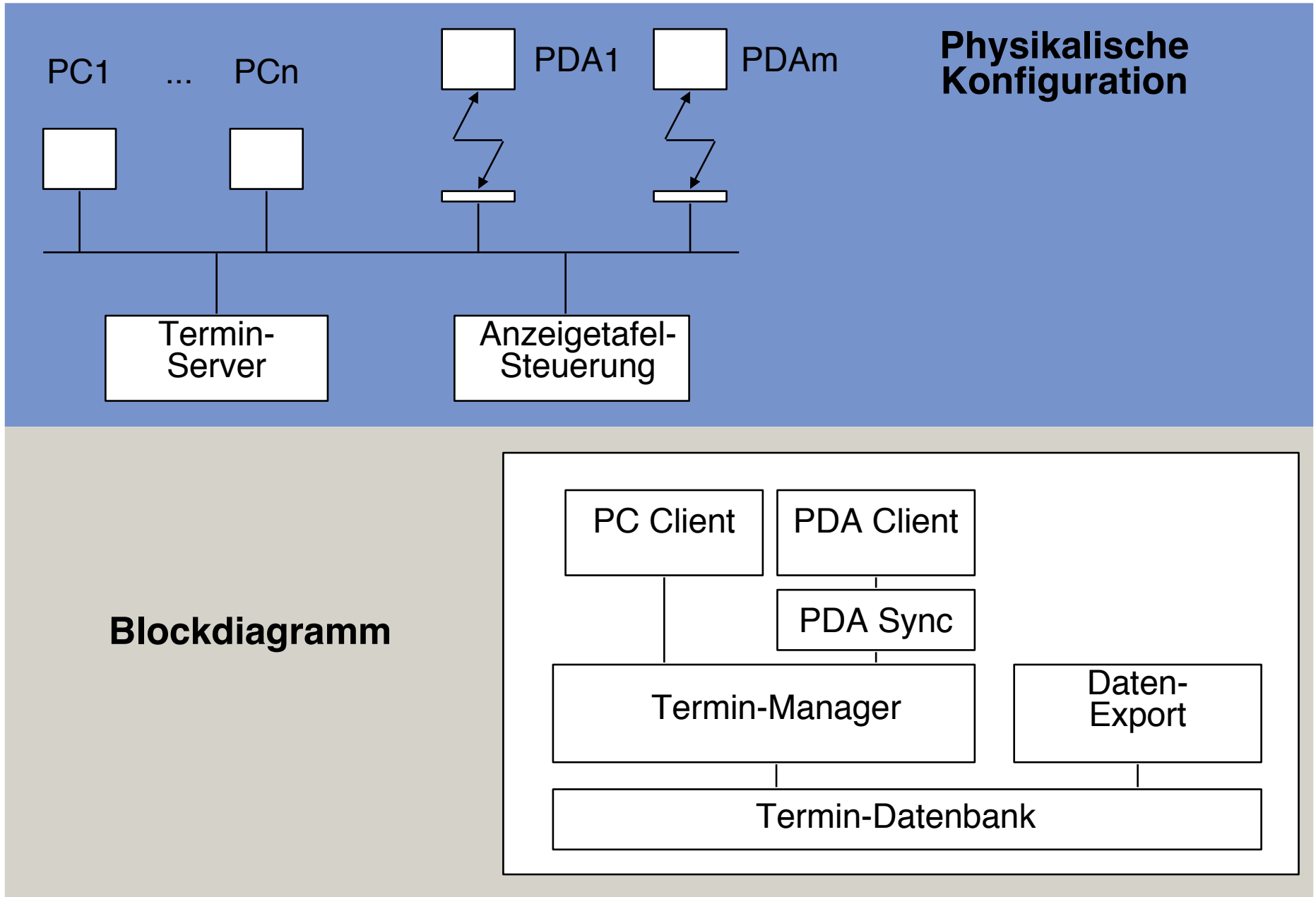


# Konfigurationsdiagramme für physikalische Verteilung

- ▶ Def.: Ein Konfigurationsdiagramm ist ein Blockdiagramm mit “Bussen” zur Beschreibung der **physischen Sicht (Verteilungssicht)**
  - Konfigurationsdiagramme sind zwar nicht Bestandteil von UML, aber dennoch ein verbreitetes Hilfsmittel zur Beschreibung der physikalischen Verteilung



# Beispiel: Konfigurationsdiagramm für Terminverwaltung

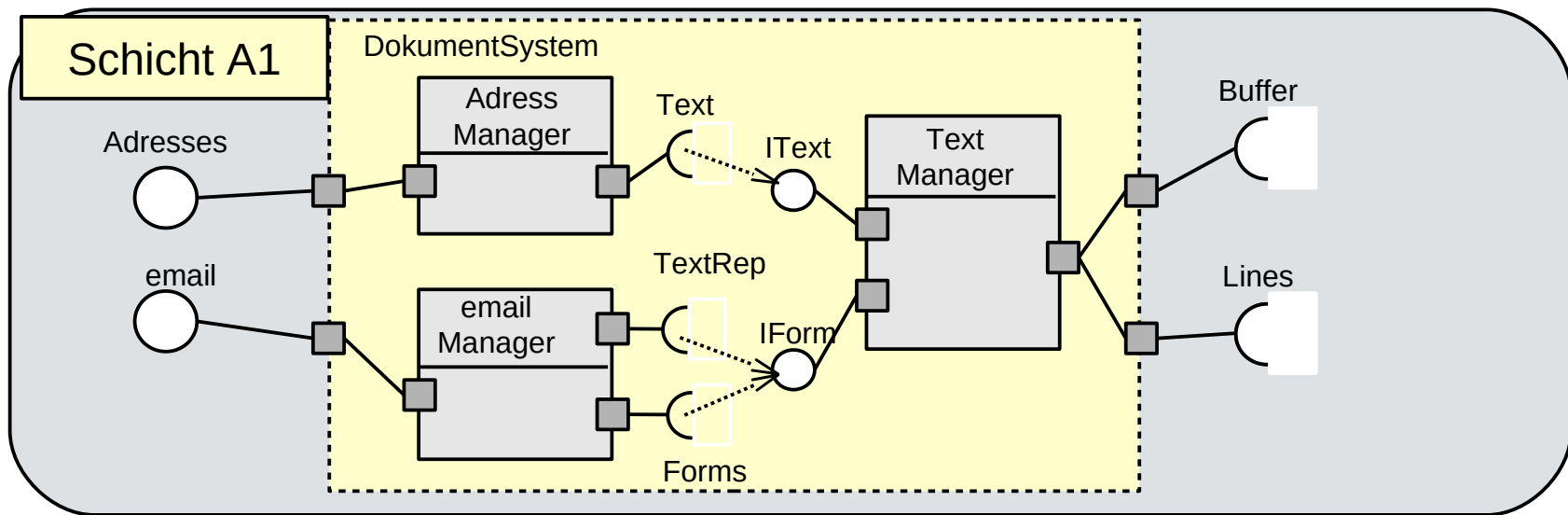
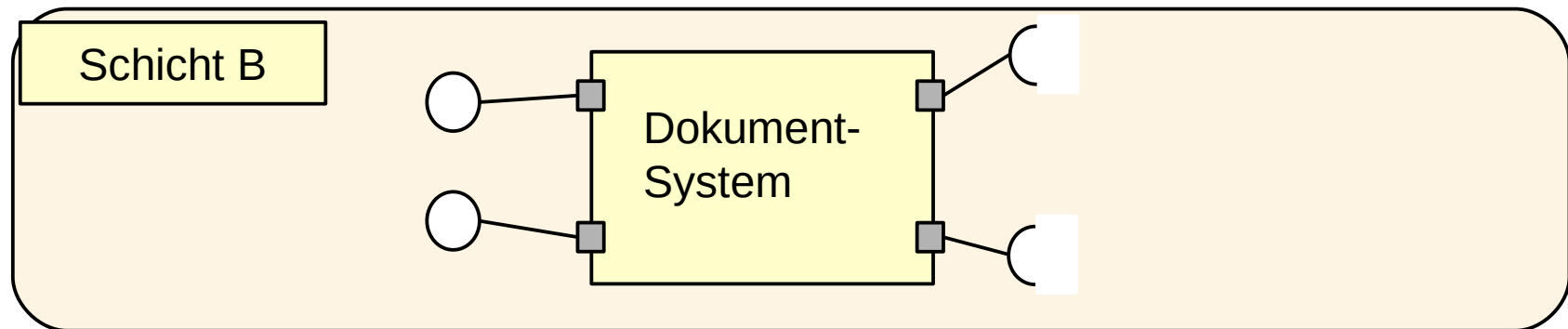


# Logische Struktur nicht-Interaktiver Anwendungen: Montagediagramme mit UML-Komponenten für die obersten Ebenen des Systems

31

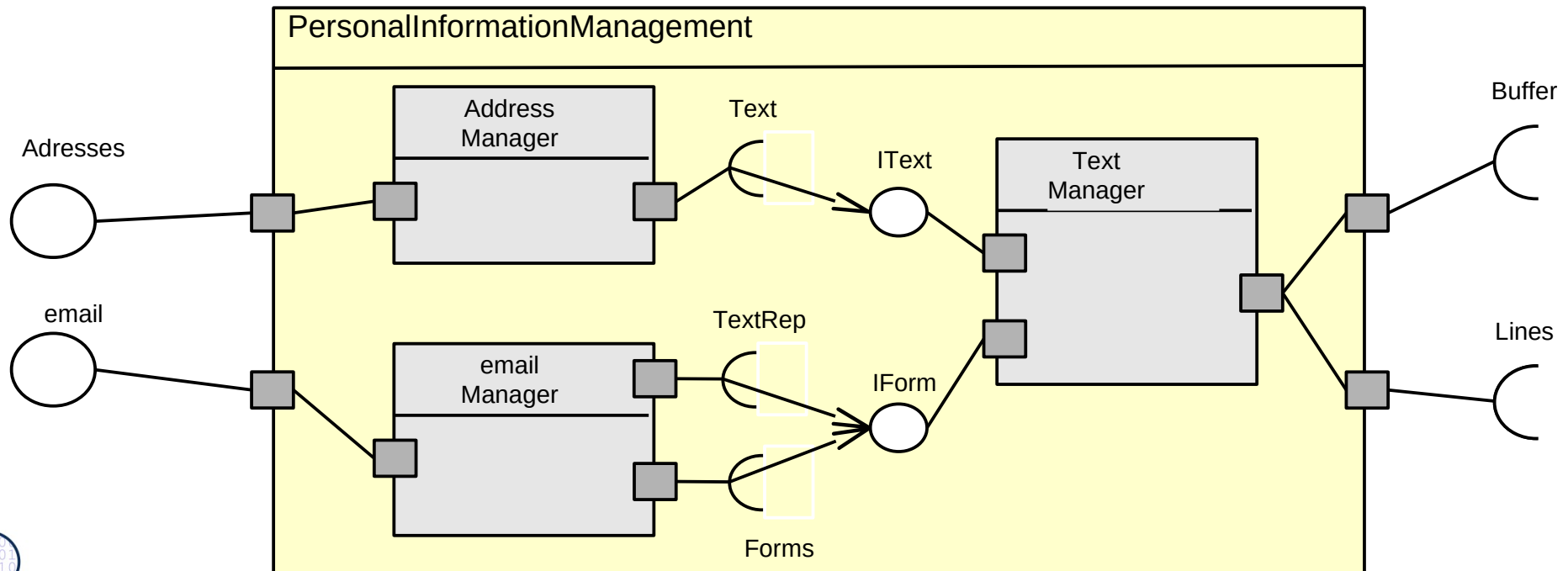
Softwaretechnologie (ST)

- ▶ Aus einem Blockdiagramm der Architektur des Systems wird ein Montagediagramm für Top-Level-Architektur entwickelt
  - 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



# Draufsicht auf die Schichten: Schachtelung von Klassen zu UML-Komponenten

- ▶ Die Schachtelung von Komponenten führt zu hierarchischen Systemen, die schichtbar sind, d.h. Jede Ebene bildet eine Schicht
- Implementierung mit **Facade Pattern**: Komponente spielt eine Facade für die Unterkomponenten einer Schicht
- Verfeinerung des Kontextmodells in die Top-Level-Architektur erzeugt eine weitere Schicht



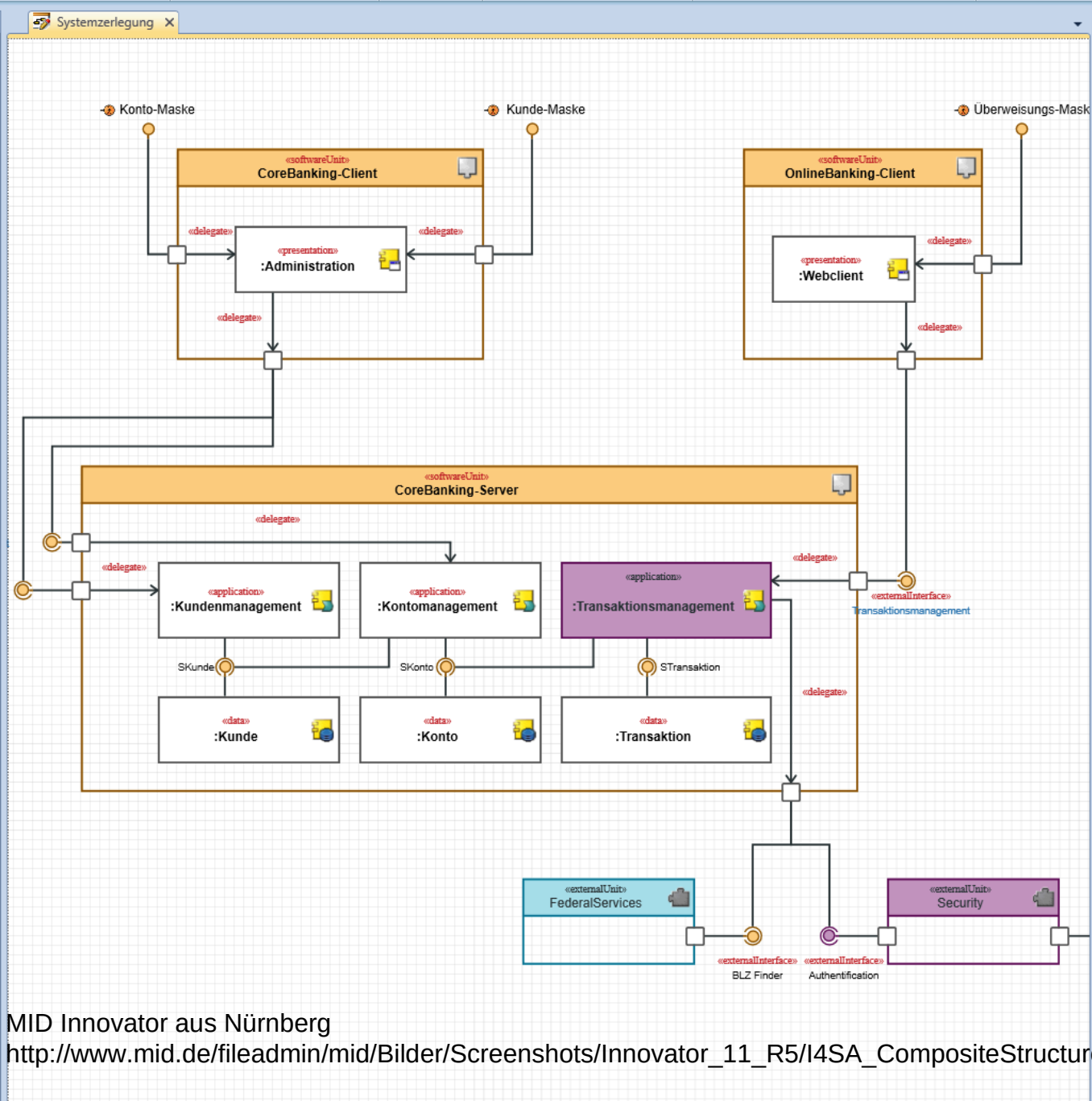


Modellinhalt

**Modellstruktur**

<Struktur durchsuchen (Strg+G)>

- CoreBankingSystem
  - systemModel
    - Evaluation
    - Projection
      - Systemzerlegung**
        - Presentation
        - Applikation
        - Daten
        - Basisklassen
        - Ereignisse
        - Ausnahmefehler
        - CoreBankingSystem
          - EJB3 Construction
          - Implementation
          - systemModel Management



Details

- Strukturierte Classifier
  - CoreBanking-Client
    - AdminView : Administration
  - CoreBanking-Server
    - KontoCtrl : Kontomanagement
    - KontoMdl : Konto
    - KundeCtrl : Kundenmanagement
    - KundeMdl : Kunde
    - TransCtrl : Transaktionsmanagement
    - TransMdl : Transaktion
  - FederalServices
  - OnlineBanking-Client
    - WebView : Webclient
  - Security
  - Komponentenschnittstellen
    - Konto-Maske
    - Kunde-Maske
    - Überweisungs-Maske
    - Login
    - Kontomanagement
    - Transaktionsmanagement
    - Kundenmanagement
    - BLZ Finder
    - Authentication

Eigenschaften

Komposition... Systemzerlegung

- Merkmale**
  - Stereotyp systemArchitec
  - Sichtbarkeit Öffentlich
  - Besitzer Projection
- Zugriffsrechte**

MID Innovator aus Nürnberg  
[http://www.mid.de/fileadmin/mid/Bilder/Screenshots/Innovator\\_11\\_R5/I4SA\\_CompositeStructureDiagram.png](http://www.mid.de/fileadmin/mid/Bilder/Screenshots/Innovator_11_R5/I4SA_CompositeStructureDiagram.png)

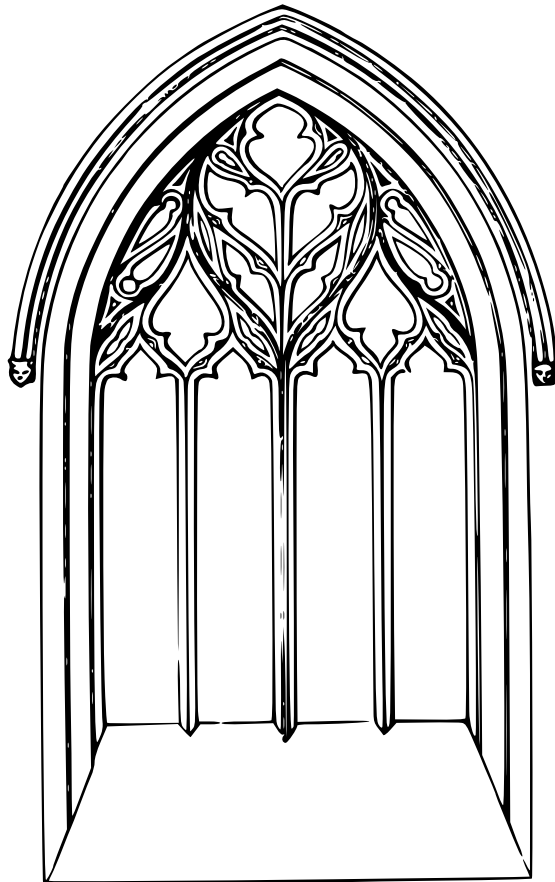
## 41.5 Architekturstile

Def.: Ein **Architekturstil** legt für alle Komponenten und Aspekte des Systems Randbedingungen und Einschränkungen fest.



# Architekturstil

- ▶ Ein Architekturstil legt für alle Komponenten und Aspekte des Systems Randbedingungen und Einschränkungen fest.



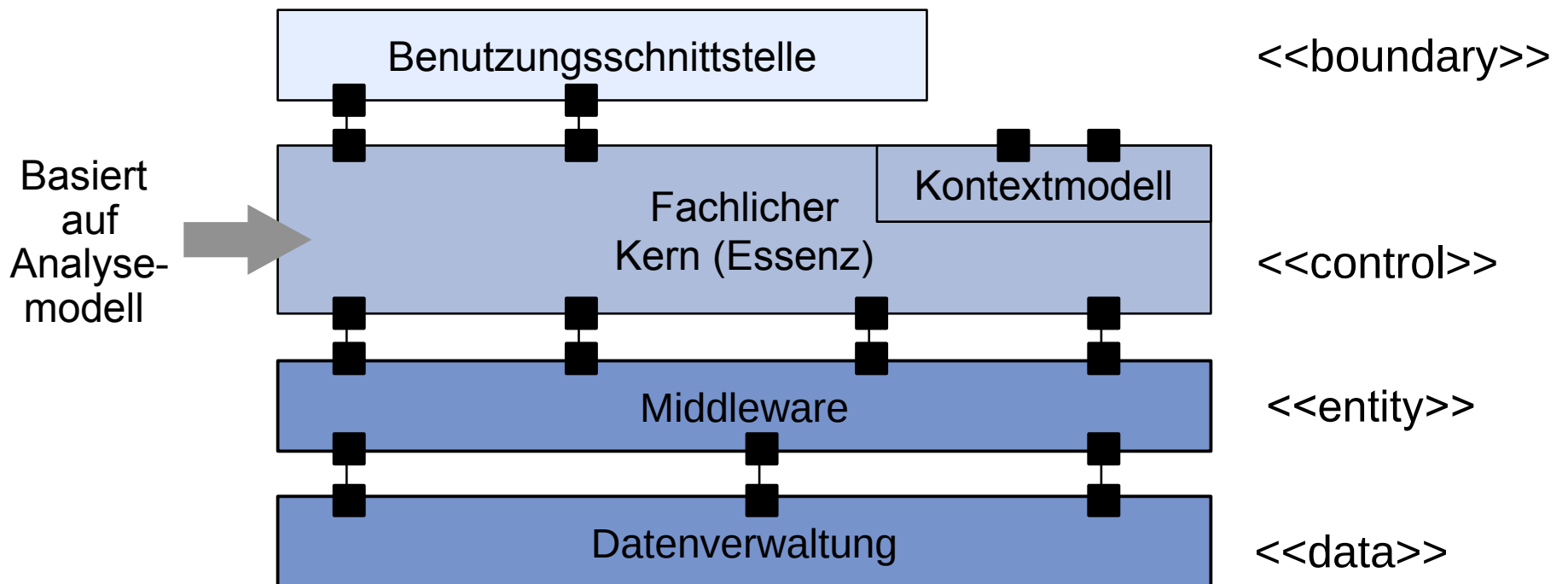
## 41.4.1 Architekturprinzip Schichtung und Architekturstil “Geschichtete Systeme” (Layered Architectural Style)

Schichten kapseln kohärentes Wissen und erzeugen lose  
Kopplung durch die “vertraut-auf”-Relation



# Architekturstil für interaktiver Anwendungen: Vier-Schichten-Architektur (BCED)

- ▶ Klassische Struktur eines interaktiven Anwendungssystems
- ▶ Schichten sind jeweils stark kohäsiv, und lose gekoppelt – warum?
  - Schichten haben angebotene Schnittstellen nach oben und benötigte Schnittstellen nach unten
  - 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

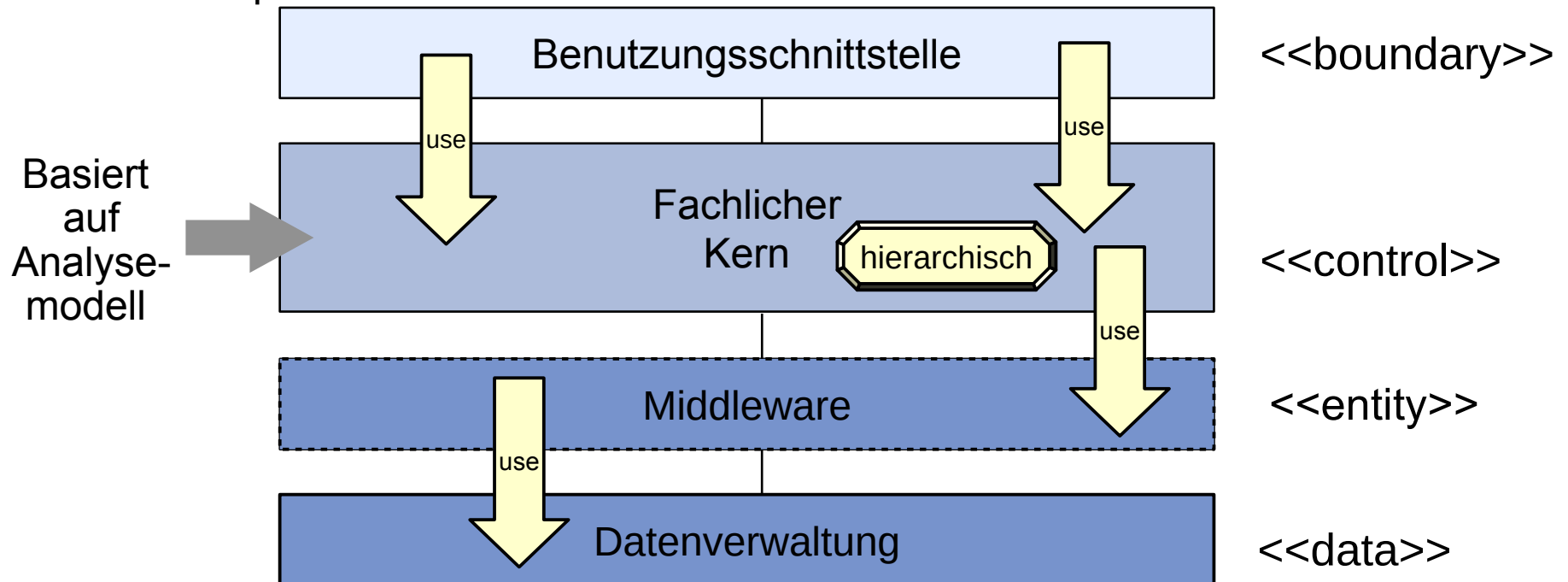


# Architekturstil für interaktiver Anwendungen: Vier-Schichten-Architektur (BCED)

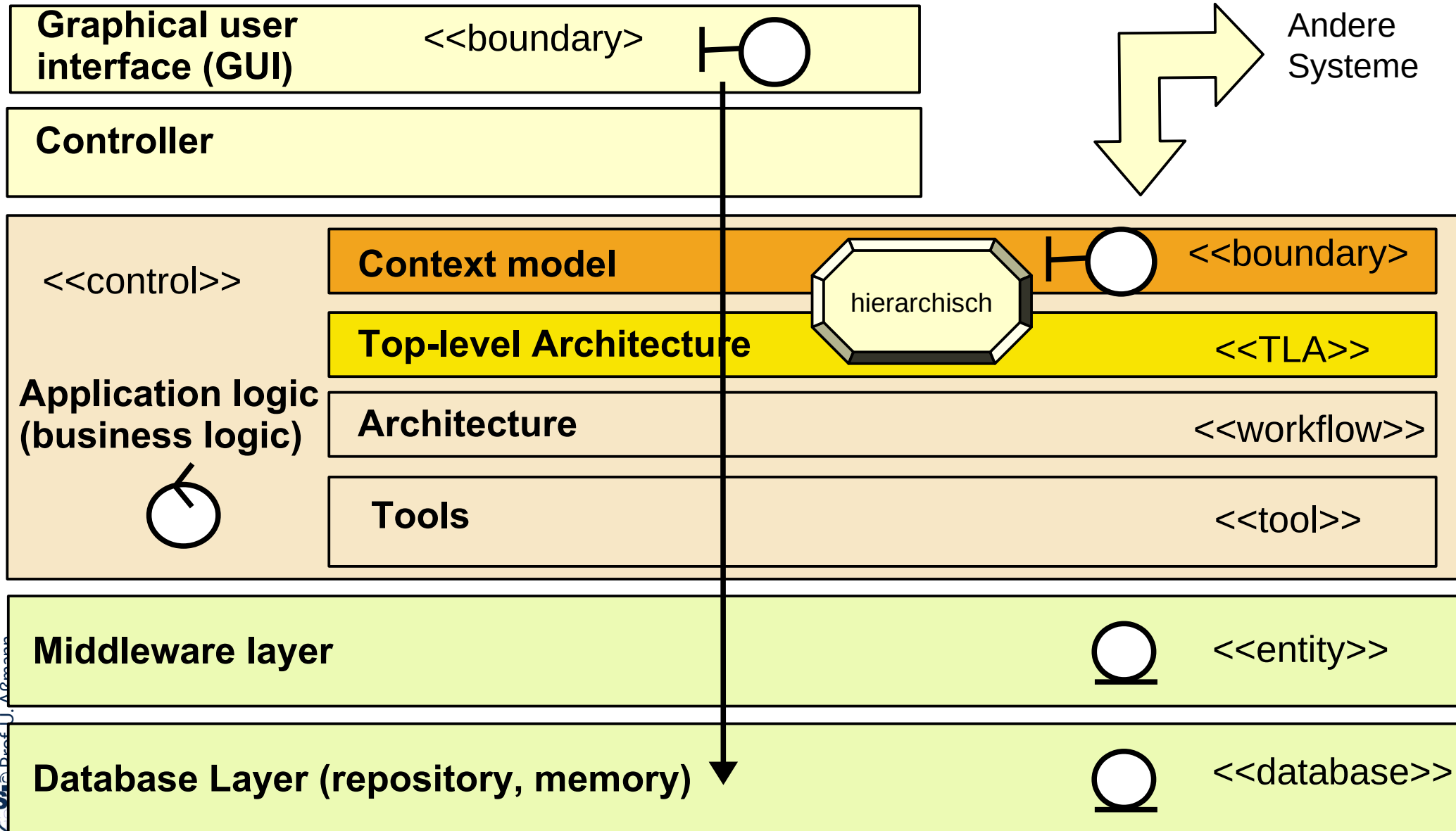
38

Softwaretechnologie (ST)

- ▶ Def.: Ein **Schichtendiagramm** ist ein geschichtetes Blockdiagramm mit Benutzungsrelation
  - verwendet für die klassische Struktur eines interaktiven Anwendungssystems
- **Einschränkung des Architekturstils:** azyklische Benutzungsrelation (use)
- ▶ **Geschichteter Integrationstest** verläuft bottom-up: erst D, dann ED, dann CED, dann BCED
- ▶ Fachlicher Kern (Anwendungslogik) kann weitere Schichten enthalten
  - Oft kapselt eine Facade eine Schicht nach oben ab



# Struktursicht: 8-Schichtenmodell, eine Verfeinerung des BCED-Schichtung eines Systems



# Schichtung beruht auf der “Vertraut-auf”-Relation (Relies-On, USES, Sees-A)

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

- ▶ *benötigt eine korrekte Implementierung* beinhaltet:
  - A ruft auf B, d.h. A delegiert Arbeit auf B oder B delegiert Arbeit zurück auf A
  - A greift zu auf öffentliche Variable oder Objekt von B
  - A nutzt eine Ressource von B
  - A alloziert ein Objekt von B
  - 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



# Geschichteter Test von hierarchischen und geschichteten Systemen

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

Integrationstests sollten bottom-up, aufwärts entlang der USES-Relation geschehen

# Frage: wie testet man ein interaktives Informationssystem?

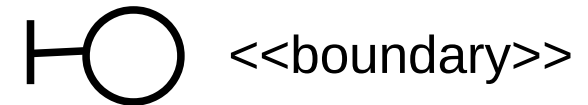
- ▶ .. wie es im Projekt-Praktikum im WS vorkommt

# Antwort: Bottom-Up!

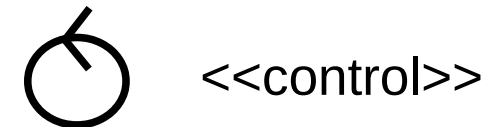
## USES-Relation in 4-Tier Architekturen (BCED)

- ▶ 4-Schichtarchitekturen nutzen eine azyklische USES-Relation
  - Obere Schichten nutzen untere, aber nicht umgekehrt

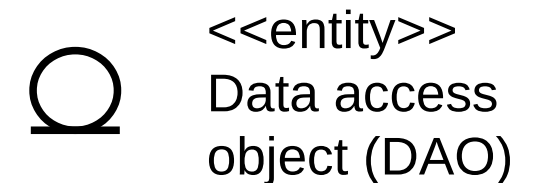
**Graphical user interface (GUI,  
Benutzerschnittstelle)**



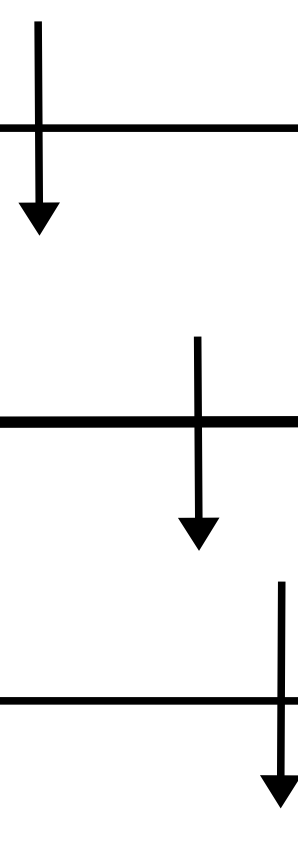
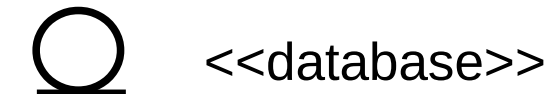
**Application logic (business logic,  
Fachlicher Kern, Anwendungslogik)**



**Middleware (memory access, distribution)**



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

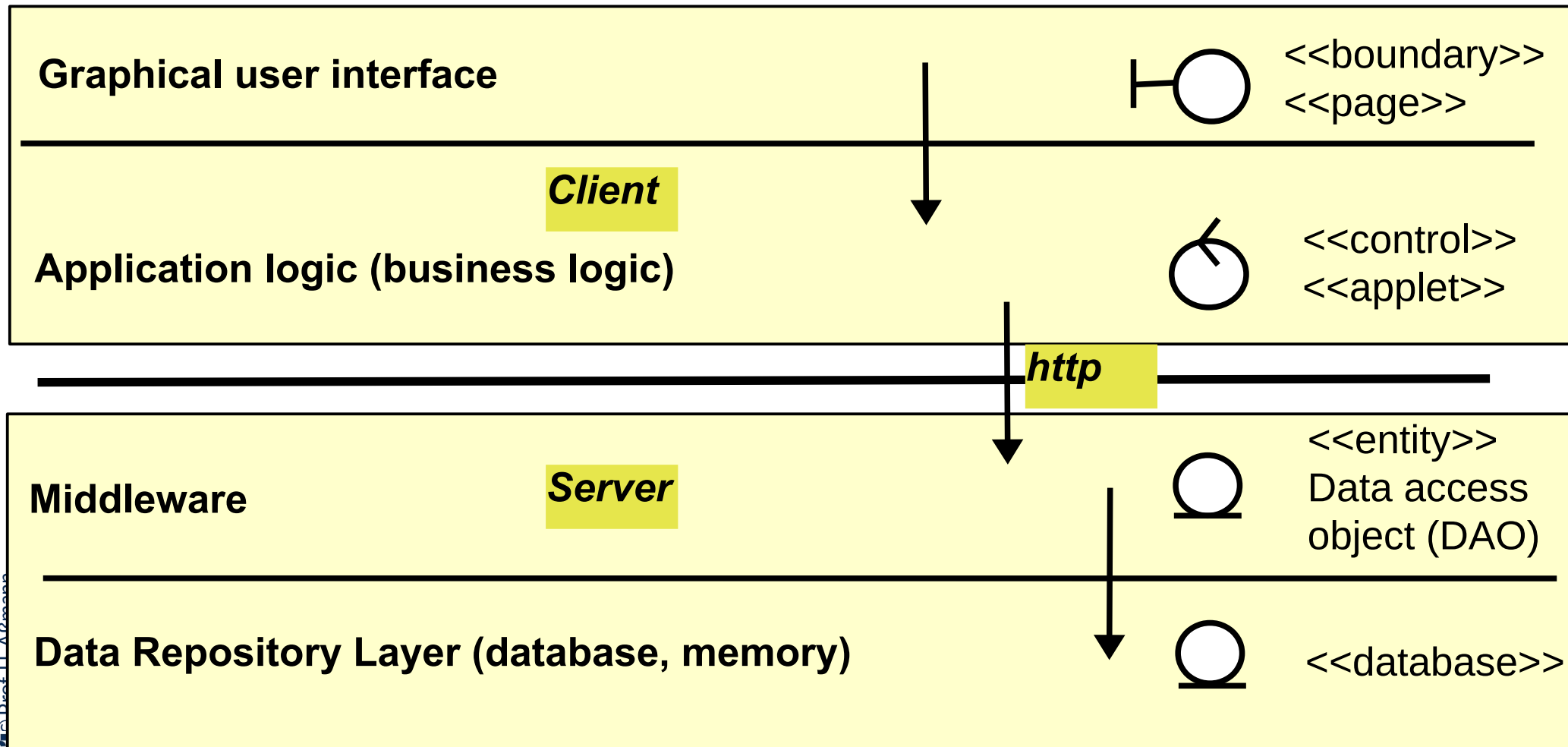


## 41.5.2 Andere geschichtete Systeme



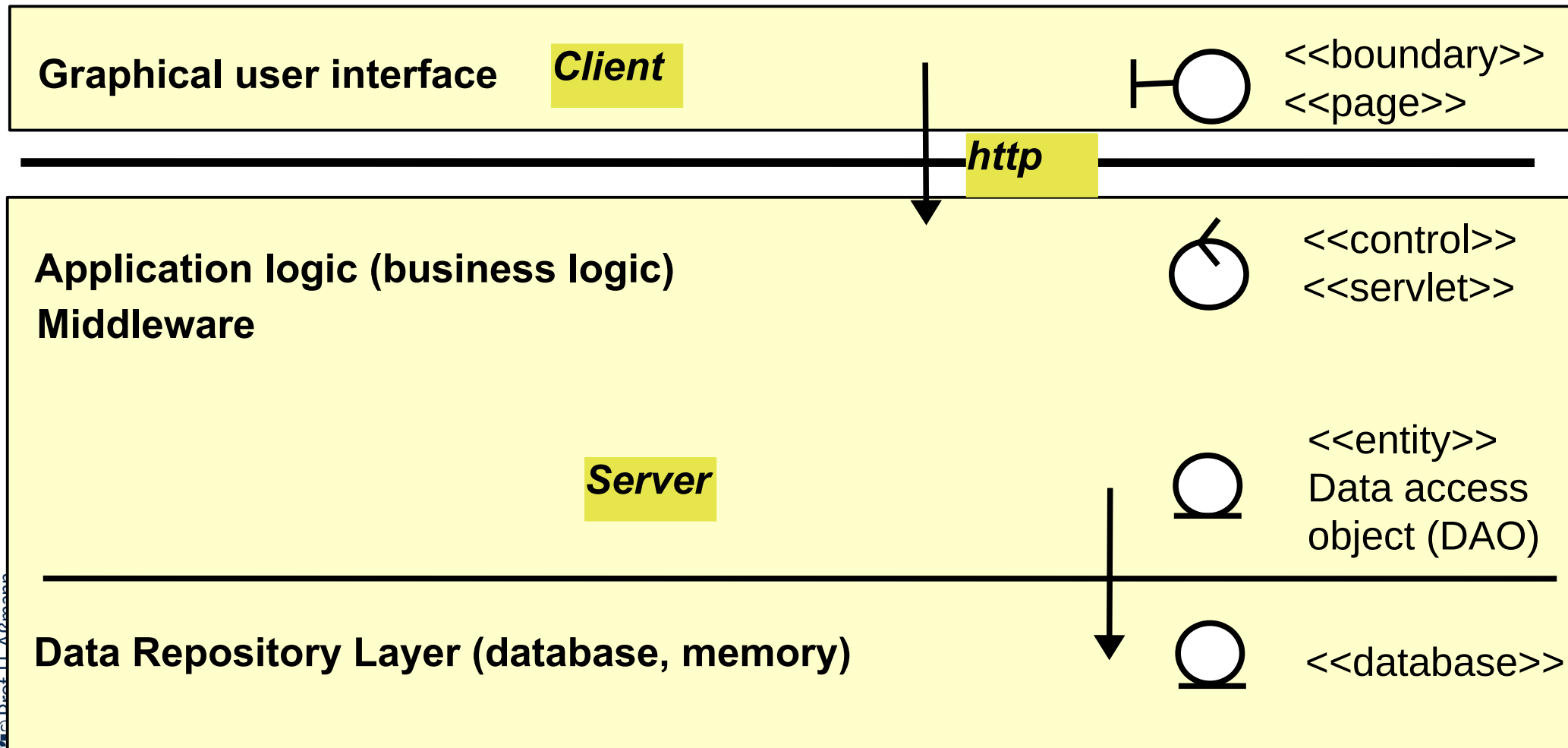
# Beispiel: 4-Tier Web System (Thick Client)

- ▶ “Thick client” Web-Systeme nutzen eine http-basierte Middleware
- ▶ GUI und AL verbleiben auf dem Client, die Daten werden auf dem Server verwaltet



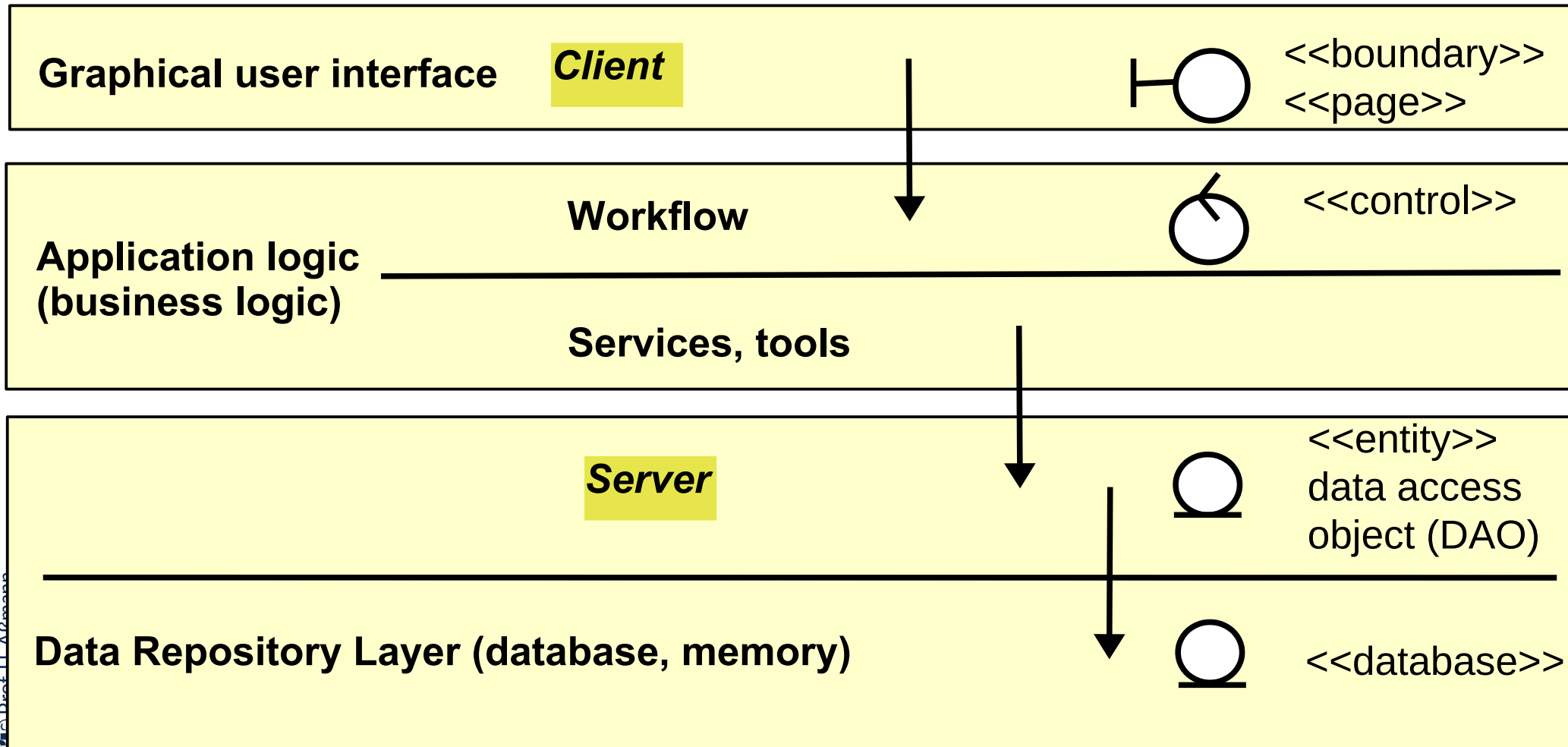
# Beispiel: 4-Tier Web System (Thin Client)

- ▶ “Thin client” Web-Systeme verwalten außer dem GUI alles auf dem Server



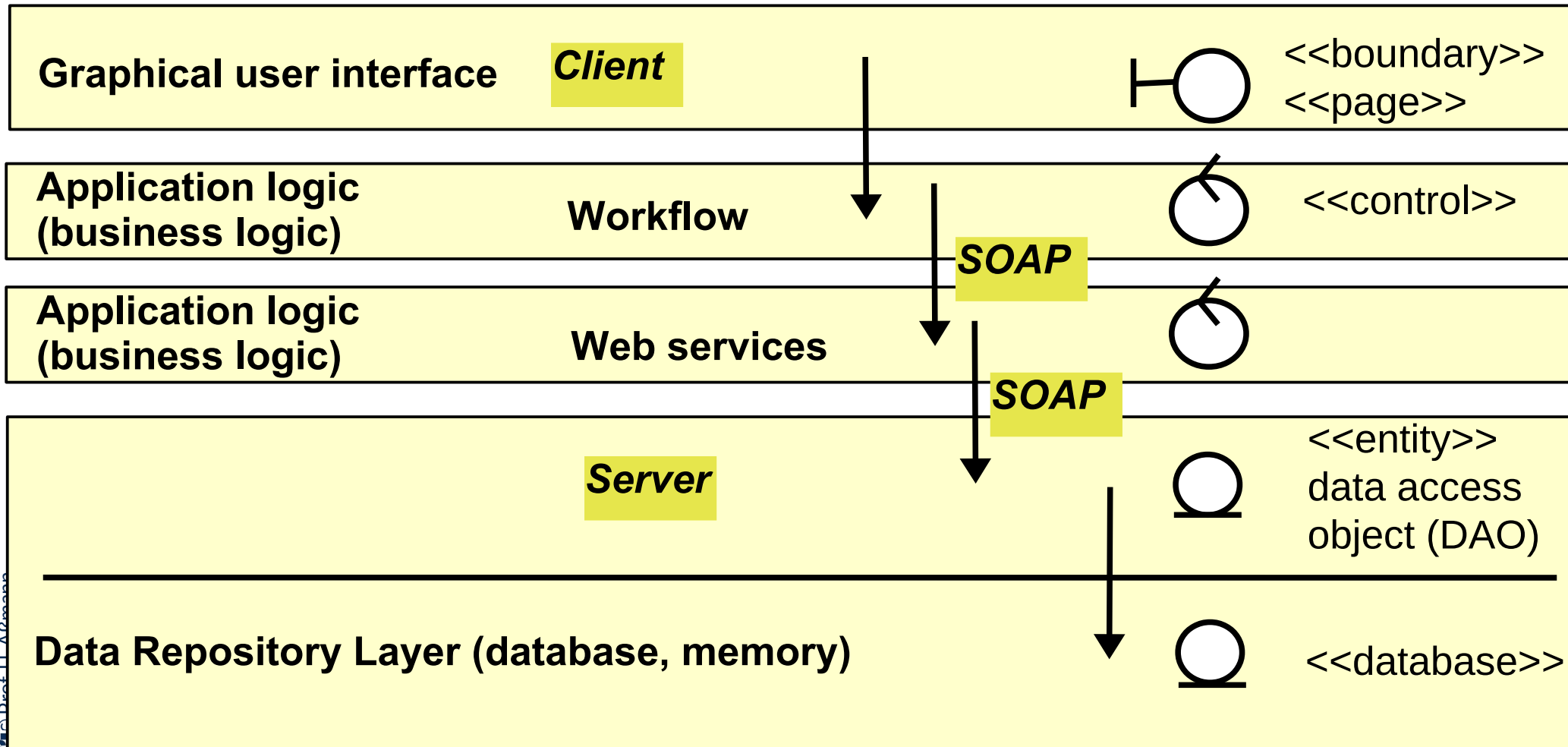
# Beispiel: 5-Tier mit Workflow Language

- ▶ Arbeitsfluss-Sprachen (Workflow languages) wie BPMN, BPEL definieren die Arbeitsfluss-Schicht der AL, unterhalb der Top-Level-Architektur
  - Services und Tools arbeiten auf den Daten



# Beispiel: 5-Tier mit Workflow Language und Web Services

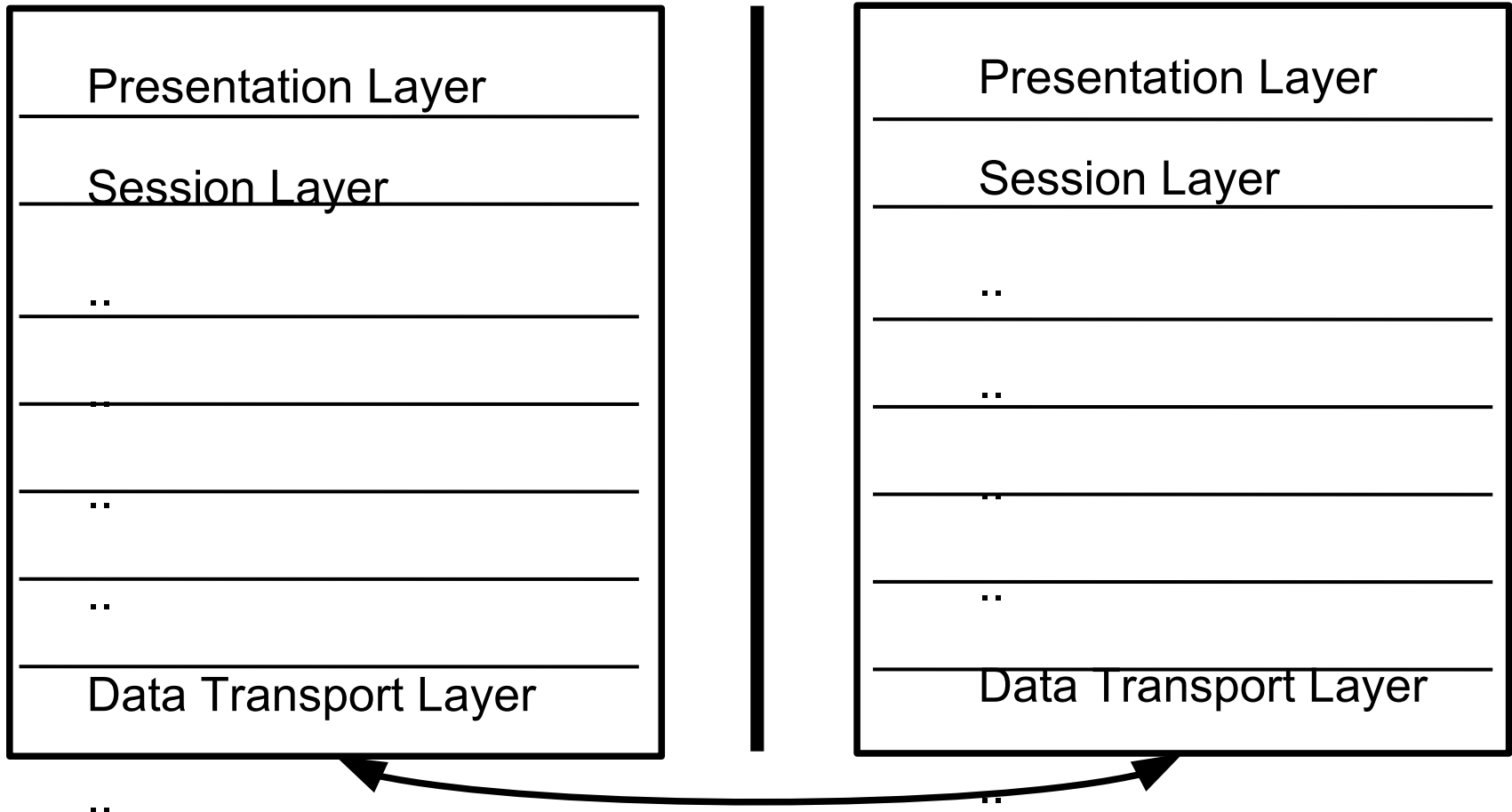
- ▶ Arbeitsfluss-Sprachen können auch *Web services* ansteuern, dann ist die Anwendung verteilt





# Weiteres Beispiel: ISO-OSI 7-Schichten Netzwerk-Architektur

- ▶ Jede Schicht enthält ein Stromobjekt für bidirektionale Kanalkommunikation



# Beispiel: Betriebssysteme

UNIX,  
Linux,  
Android

User Space

---

Kernel

---

Apple-  
UNIX,  
iOS

User Space

---

Kernel

---

Microkernel (Mach)

---

Windows NT/XP:

User Space

---

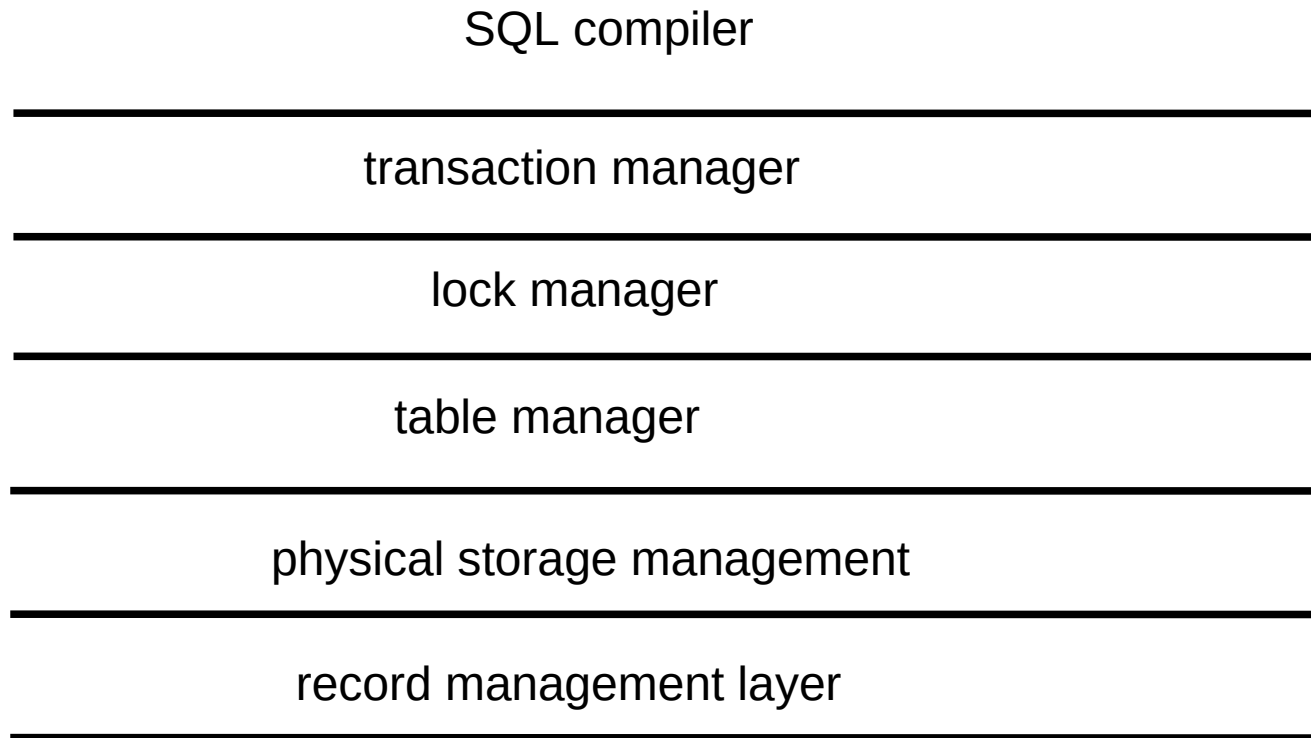
Kernel

---

Hardware Abstraction Layer (HAL)

---

# Beispiel: Datenbanksysteme



# Warum sind geschichtete Architekturen wichtig?

- ▶ Der “Layered architecture style” benötigt eine azyklische USES-Relation
- ▶ Vorteile:
  - Jede Schicht wirkt von außen wie eine einzige Komponente mit angebotenen und benötigten Schnittstellen (Entwurfsmuster Facade)
  - Kohäsion stark, Kopplung gering
  - Veränderungsorientierter Entwurf, Austausch von Schichten möglich → Evolution einfach
  - Verantwortlichkeiten für Testmanagement können klar definiert werden: Für jede Schicht werden separate Testsuites entwickelt (Bottom-up tests)

## 41.6. Architektur mit Perspektivenmodellen

Einer Architektur liegt eine Perspektive zugrunde, die mehrere Sichten auf das System definiert.

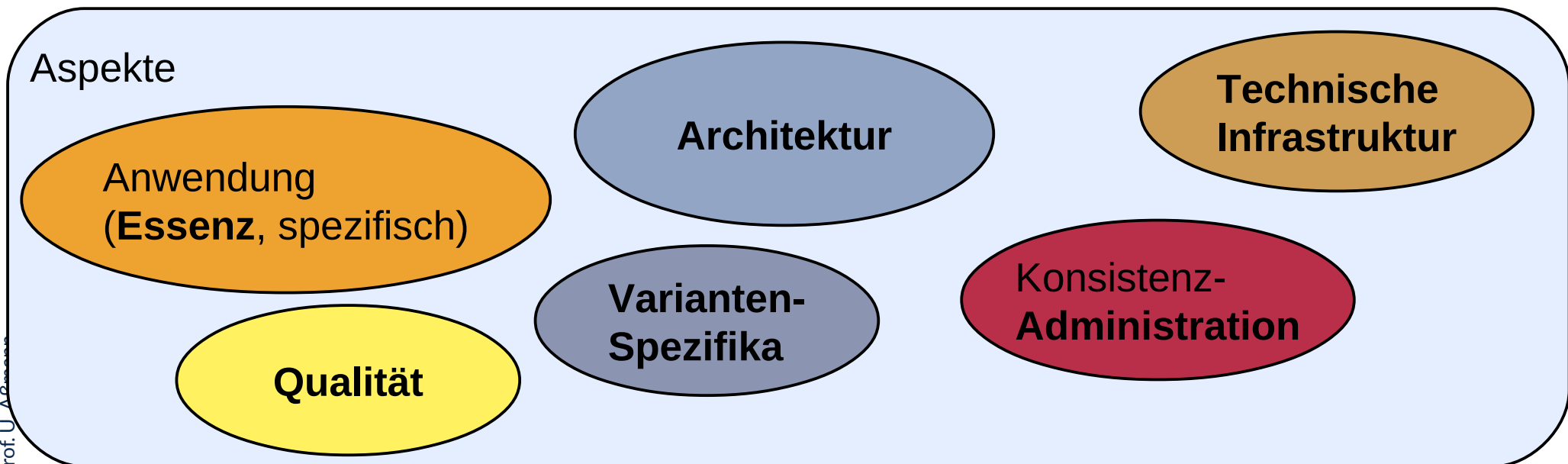
Ein **Perspektivenmodell (view model)** definiert eine Menge von *Perspektiven mit Aspekten und ihren Sichten*, von denen die Software aufgeteilt wird

In UML: Ein Perspektivenmodell definiert ein Profil von Stereotypen, die auf Fragmente eines UML-Modells aufgeklebt werden können



# Die Perspektive und die Aspekte der logischen Struktur

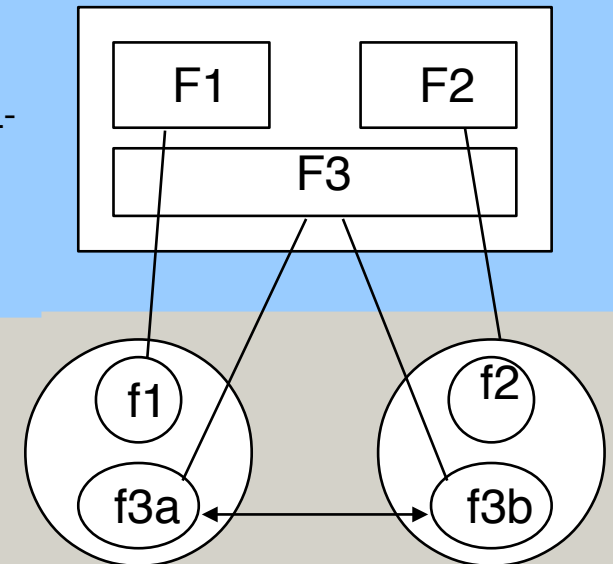
- ▶ Eine **Sicht** bildet eine Teilmenge der Komponenten eines Systems.
- ▶ Ein **Aspekt** definiert eine Sicht auf ein System, legt eine Teilmenge des Systems fest.
- ▶ Eine **Perspektive** gruppiert eine Menge von Aspekten und deren Sichten.
- ▶ Ein **Perspektivenmodell** definiert eine Menge von Perspektiven, die Aspekten und Sichten gruppieren.
  - Zu jeder Perspektive wird mindestens ein Modell erstellt



- ▶ Folgende Perspektiven (Gruppen von Aspekten und ihren Sichten) werden in Perspektivenmodellen unterschieden:

- ▶ **Logische Sicht** mit struktureller Zerlegung:

- Struktur im Großen: Blockdiagramme, Montagediagramme (UML-Komponentendiagramme)
- Architekturstil: Schichten, Sichten, Dimensionen
- Logischer Detail-Entwurf



- ▶ **Verteilungssicht:** Struktur der physikalischen Verteilung:

- Zentral oder verteilt? Topologie

- ▶ **Dynamische Sicht** (Ablaufsicht)

- Prozesse, Synchronisation
- Flüchtigkeit und Persistenz von Daten

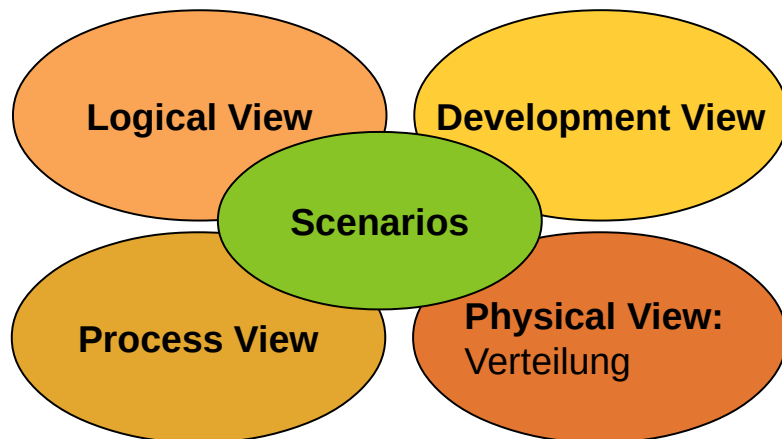
- ▶ **Qualitätssicht:** Einhaltung nichtfunktionaler Anforderungen

- Architekturbestimmende Eigenschaften (z.B. Realzeitsystem, eingebettetes System)
- Effizienzanforderungen und Optimierung
- Standardarchitekturen

# Weitere Beispiele für Perspektivenmodelle (View Models)

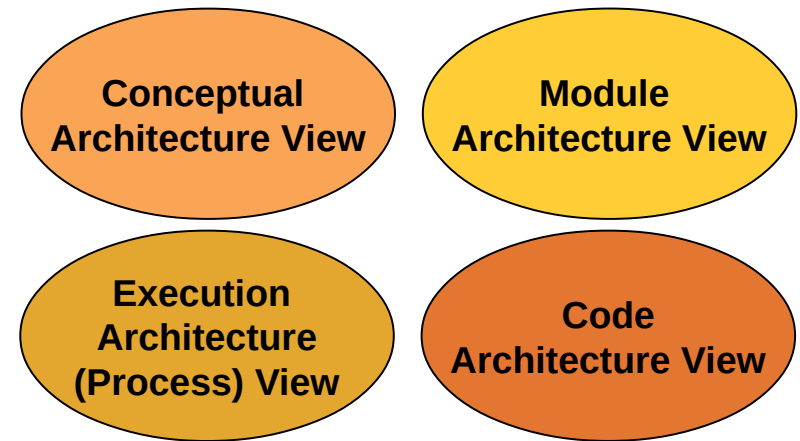
## 4+1 Views von Kruchten:

- 1) **Logical View:** logische Struktursicht in Klassen, Komponenten
- 2) **Development View:** Entwicklungssicht
- 3) **Process View:** Prozess-Sicht
- 4) **Physical View:** Verteilung, nicht-funktionale Eigenschaften
- 5) + **Scenarios**, die alle anderen Sichten querschneiden



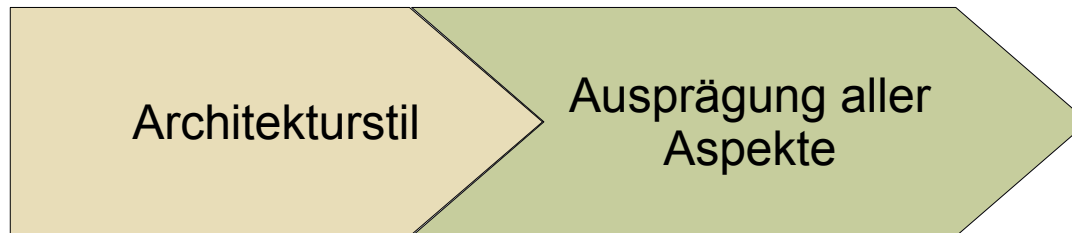
## Hofmeister/Soni/Nord:

- 1) **Conceptual Architecture View:** logische Struktursicht aus Komponenten und Konnektoren
- 2) **Module Architecture View:** Struktursicht des Feinentwurfs mit Komponenten und Klassen
- 3) **Execution Architecture View:** Prozess-Sicht
- 4) **Code Architecture View:** Arrangement der Code-Dateien im Filesystem



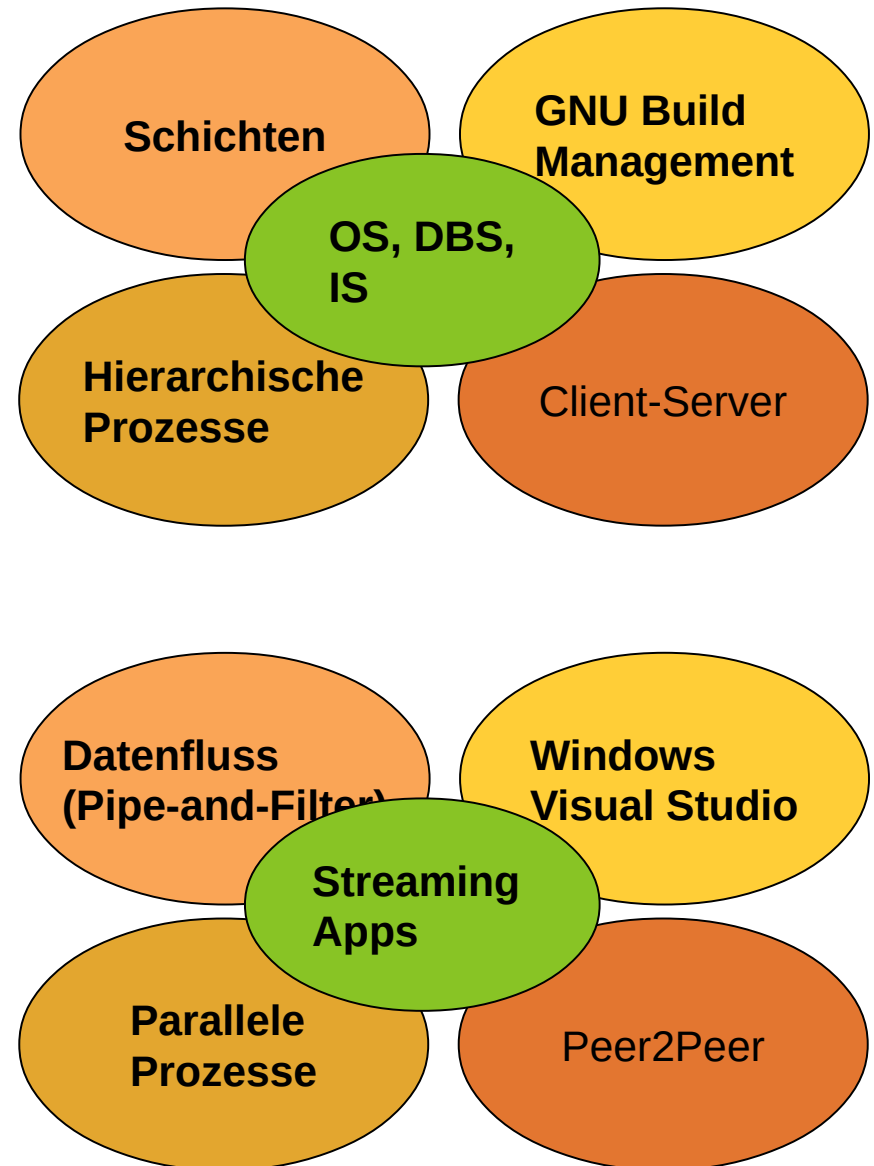
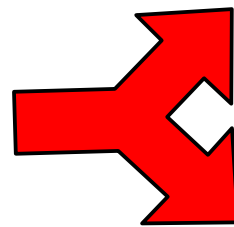
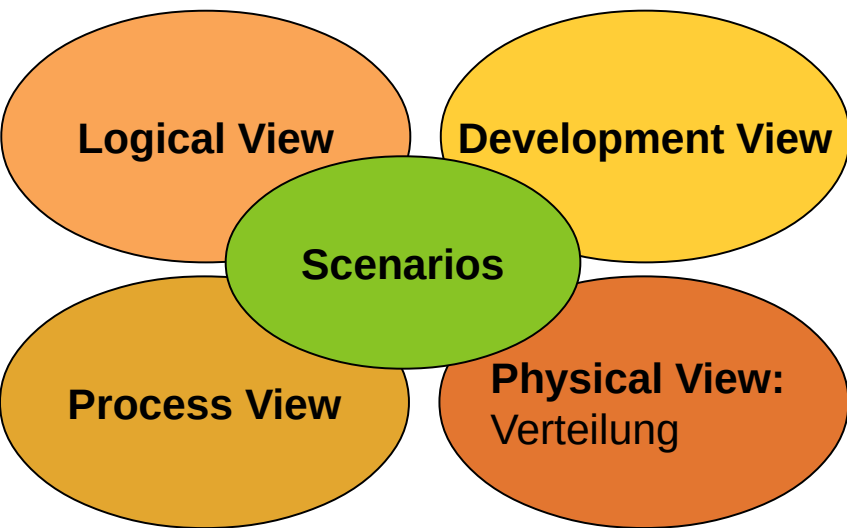


## 41.6.1 Perspektivenmodelle und Architekturstile



# Architekturstile im Perspektivenmodell "4+1 Views"

- ▶ Def.: Ein **Architekturstil** legt für jeden Aspekt eines Perspektivenmodells Randbedingungen fest.
- ▶ Beispiele für Ausprägungen von 4+1V:



# Was haben wir gelernt?

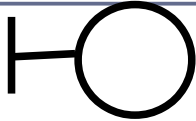
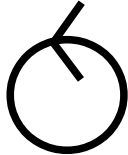


- ▶ Architektur trennt die Aspekte des Programmierens im Großen vom Programmieren im Kleinen
- ▶ Der Architekturstil der Anwendung muss festgelegt werden und spielt eine große Rolle im Architekturentwurf
- ▶ Ein Perspektivenmodell hilft uns, Software zu in Aspekte und Sichten zu gliedern
- ▶ Schichtenbasierte Architekturen sind wichtig
  - Azyklische USES (relies-on) Relation

# Vorsicht: Conway's Law über Software-Strukturen

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

- ▶ Erklären Sie, warum das Kontextmodell aus hierarchischen Komponenten besteht.
- ▶ Was ist der Unterschied zwischen Blockdiagramm, Montagediagramm und Verteilungsdiagramm?
- ▶ Welche architektonische Sichten auf ein System kennen Sie?
- ▶ Erklären Sie Kruchten's 4+1 Perspektivenmodell
- ▶ Warum bildet eine Komponente eine Instanz des Entwurfsmusters Facade?
- ▶ Erklären Sie, wann ein System eine Schichtung besitzt. Warum sind Schichten wichtig?
- ▶ Erklären Sie das 4-Schichtenmodell und vergleichen Sie es mit seiner Verfeinerung, dem 8-Schichtenmodell.
- ▶ Erklären Sie den Unterschied zwischen Schichten und Ringen.
- ▶ Warum brauchen wir Sichten für eine gute Softwarearchitektur?

# Repet.: BCD/BCED Classification

- ▶ 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

Prof. Dr. rer. nat. Uwe Aßmann  
Institut für Software- und  
Multimediatechnik  
Lehrstuhl Softwaretechnologie  
Fakultät für Informatik  
TU Dresden  
Version 19-0.3, 06.07.19

- 1) Geschäftsmodelle
- 2) Architekturprinzipien
- 3) Architekturdiagramme
- 4) Architekturstile
- 1) Geschichtete Architekturen
- 5) Architektur mit Perspektivenmodellen



- ▶ Zuser Kap 10.
- ▶ Ghezzi 4.1-4.2
- ▶ Pfleeger 5.1-5.3
- ▶ ST für Einsteiger 5.3, 8
- ▶ Erste wissenschaftliche Papiere zur Lese:
  - Parnas, D.L. On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM 15 (12): 1053–58. December 1972, doi:10.1145/361598.361623
  - Phillipe B. Kruchten. The 4+1 view model of architecture. IEEE Software, Nov. 1995. doi:10.1109/52.469759



## Exkurs: Wie findet man ein Papier?

- ▶ <http://scholar.google.de>
- ▶ <http://iinwww.ira.uka.de/bibliography/index.html#search>

Bitte tippen Sie diese URL in Ihren Browser ein.  
Suchen Sie nach den Papieren von Parnas und Kruchten.  
Können Sie pdf-Dateien finden?

- ▶ David J. Parnas. On a buzzword: hierarchical structure. Proceedings IFIP Congress 1974, North-Holland, Amsterdam.
- ▶ Christine Hofmeister, Robert L. Nord, and Dilip Soni. Describing software architecture with UML. In Patrick Donohoe, editor, WICSA, volume 140 of IFIP Conference Proceedings, pages 145-160. Kluwer, 1999.
  - Christine Hofmeister, Robert Nord, and Dilip Soni. Applied Software Architecture. Addison-Wesley, Reading, MA, 2000.
- ▶ Johannes Siedersleben. Moderne Softwarearchitektur. Umsichtig planen, robust bauen mit Quasar. dpunkt-Verlag, 2004.

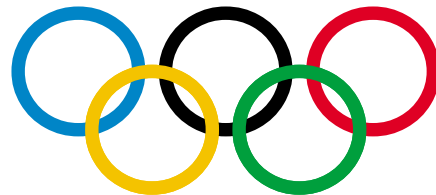
## Teil IV - Objektorientierter Entwurf (Object-Oriented Design, OOD)

5 Softwaretechnologie (ST)

- 1) 40: Überblick
- 2) 41: Einführung in die objektorientierte Softwarearchitektur
  - 1) Architekturprinzipien, Architekturstile, Perspektivenmodelle
  - 2) Modularität und Geheimnisprinzip
  - 3) BCD-Architekturstil (3-tier architectures)
- 3) 42: Verfeinerung mit querschneidender Objektorichung
- 4) 43: Architektur interaktiver Systeme
- 5) 44: Punktweise Verfeinerung von Lebenszyklen
  - Verfeinerung von verschiedenen Steuerungsmaschinen



- ▶ Was ist der Unterschied zwischen Programmieren im Kleinen und *Programmieren im Großen*?
- ▶ Wie hilft Architektur, Produktlinien aufzubauen?
- ▶ Ringe sind keine Schichten
- ▶ Was sind die 5 olympischen Ringe der Software?



**OUR GOALS**

Olympische Dekomposition in 5 Ringe hilft, den sozialen Reifegrad von Software zu heben.



## 41.1 Geschäftsmodelle für Softwarefirmen

- ▶ → Kurs "Design Patterns and Frameworks (DPF)" in WS
- ▶ → Kurs "Software as a Business (SAAB)" in WS
- ▶ → Kurs "Component-Based Software Engineering (CBSE)" in WS

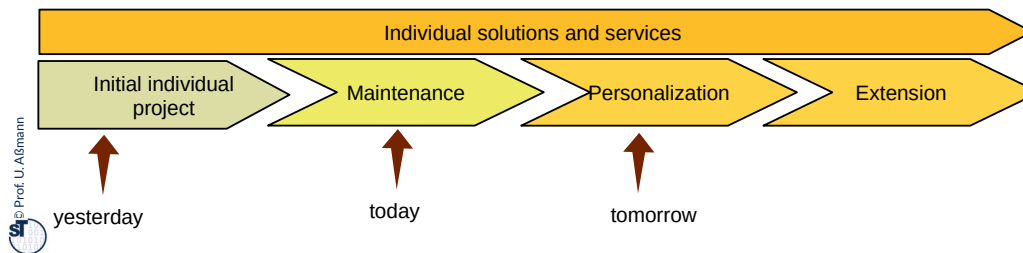


## Geschäftsmodell Individualprojekte und Dienstleistung (Individual Solutions and Services, iSaS)

8 Softwaretechnologie (ST)

Good software architecture simplifies the management of solutions and services (iSaS)

- ▶ Individual project (solution to a problem of customer)
- ▶ Maintenance (Wartung)
- ▶ Configuration, Personalization
- ▶ Extension



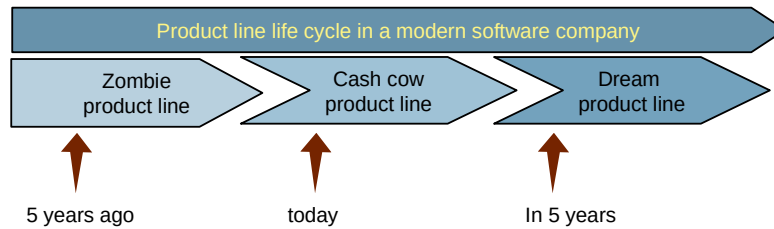
Producing a product family (product line) is a successful business model for companies. Therefore, a systematic design towards product lines can be a decisive economic factor in the life of a company.

In a company, usually 2-3 product lines are active at the same time:

- The “zombie” product line is the one of the past, from which no new products are created, but old products are in use and must be maintained. Therefore the

Good software architecture simplifies the management of SPL

- ▶ **Variability Points**
  - Exchanging parts easily
  - Variation, variability, complex parameterization
- ▶ **Extension Points**
  - Software must be extended
- ▶ **Glue Code** (adaptation overcoming architectural mismatches)
  - Coupling software that was not built for each other



Producing a product family (product line) is a successful business model for companies. Therefore, a systematic design towards product lines can be a decisive economic factor in the life of a company.

In a company, usually 2-3 product lines are active at the same time:

- The “zombie” product line is the one of the past, from which no new products are created, but old products are in use and must be maintained. Therefore the



## 41.2 Architekturprinzipien

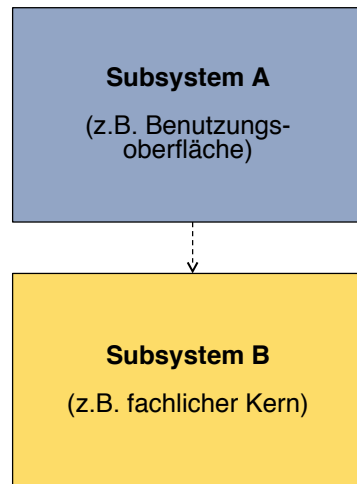
Architekturprinzipien bilden allgemeine Gestaltungsregeln ("best practices") für die Architektur.

Entwurfsmuster realisieren diese Gestaltungsregeln in einem wiederverwendbaren Entwurf.





## 41.2.1 Architekturprinzip: Hohe Kohäsion + Niedrige Kopplung



- ▶ **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.



## 41.2.2 Architekturprinzip: Veränderungsorientierter Entwurf mit Modularität

- ▶ Zu ihrer besseren Wiederverwendbarkeit sollte Software in *Komponenten (Module)* eingeteilt werden (*Modularität*)
- ▶ Eine **Komponente (Modul)** im allgemeinen Sinne ist eine Wiederverwendungseinheit:
  - die Funktionalität mit hoher Kohäsion gruppiert
  - die *angebotene* und *benötigte* Schnittstellen besitzt, um lose Kopplung zu unterstützen:
    - keine impliziten, nur explizit in der Schnittstelle angegebene Abhängigkeiten zu anderen Komponenten
- ▶ Vorteile der **Modularität**:
  - Unabhängigkeit im Software-Entwicklungsprozess:
    - Komponente kann unabhängig von anderen entwickelt werden
    - Komponenten können einzeln getestet werden (Einheitstest, unit test)
    - Fehler können zu individuellen Komponenten verfolgt werden
  - Komponenten können ausgetauscht werden, ohne dass das System zusammenbricht (Ersetzbarkeit)
    - weil angebotene und benötigte Schnittstellen unterschieden werden



# Bemerk.: Modularität mit Komponentenmodellen und Kompositionssystemen

- ▶ Es gibt nicht nur die UML-Komponente....sondern viele verschiedene *Komponentenmodelle*:
  - Module einer modularen Programmiersprache (Modula, Ada)
  - *Binäre Module*, z.B. class-Files oder .o-Files
  - Klassen, Kollaborationen und Konnektoren in objektorientierten Sprachen
  - UML-Komponenten
  - Ganze Schichten eines Systems, insofern sie in eine Komponente gekapselt werden können (wie z.B. die TLA)
  - 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)



### 41.2.3. Architekturprinzip: Flexible Evolution mit dem Geheimnisprinzip

Parnas' Prinzip des Entwurfs mit dem **Geheimnisprinzip** gilt für alle Komponentenmodelle (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 (**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
- 4) Definiere *angebotene* und *benötigte* Schnittstellen

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



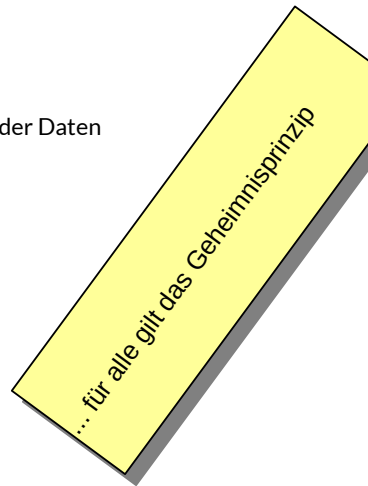
# Typische Geheimnisse von Modulen/Komponenten

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

# Verschiedene Arten von Komponenten/Modulen (in verschiedenen Sprachen)

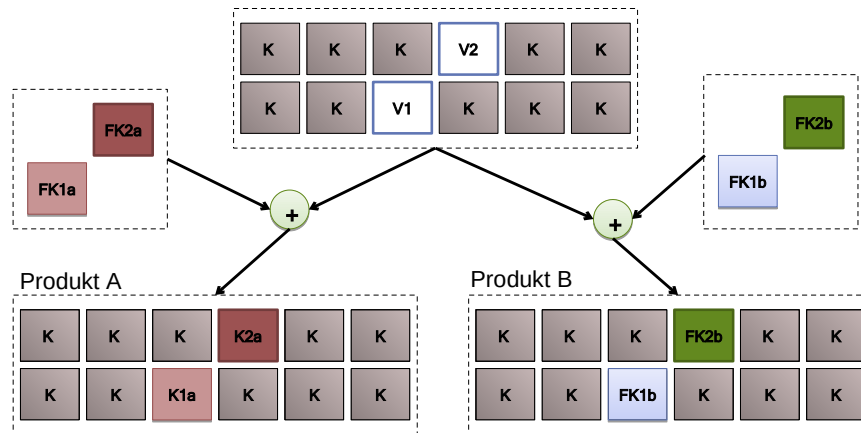
16 Softwaretechnologie (ST)

- ▶ Funktionale Module ohne Zustand
  - sin, cos, BCD arithmetic, gnu mp,...
- ▶ Daten-Repositorien
  - 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)
- ▶ Entwurfsmuster Facade zur Kapselung von Schichten
- ▶ Schichten eines Systems
- ▶ Fragmentkomponenten



## 4.2.4 Architekturprinzip Gemeinsames vs Spezifisches (Variabilitätspunkte)

- ▶ Für eine Produktlinie müssen die Komponenten, die allen Produkten gemeinsam sind, von denen getrennt werden, die spezifisch sind
- ▶ Auszutauschende Komponenten im *Framework* nennt man *Variabilitätspunkte (V)*, die mit jeweils unterschiedlichen *Füllselkomponenten (FK)* aufzufüllen sind



## Alle Variabilitätsmuster in objektor. Sprachen nutzen das Geheimnisprinzip

- ▶ Viele Entwurfsmuster (z.B. TemplateMethod) sind vom Parnas-Prinzip inspiriert.
- ▶ Sie sind **Variabilitätsmuster**, d.h., sie verbergen bestimmte Geheimnisse und erlauben dann, die Implementierungen auszutauschen (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

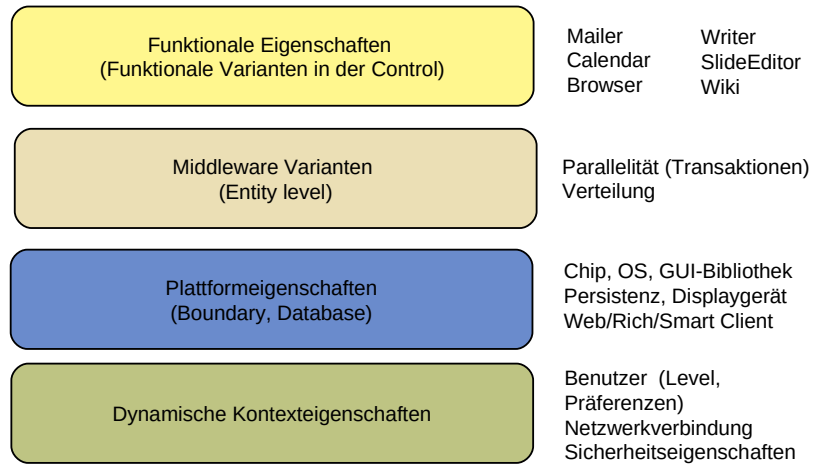
Variabilitätsmuster repräsentieren Variabilitätspunkte der Architektur





## 41.2.5. Architekturprinzip: Schichten von Variabilität

- ▶ **Software-Produktlinien** entstehen durch systematische Variation von Geheimnissen, wobei diese in *Schichten oder Aspekte* gruppiert werden





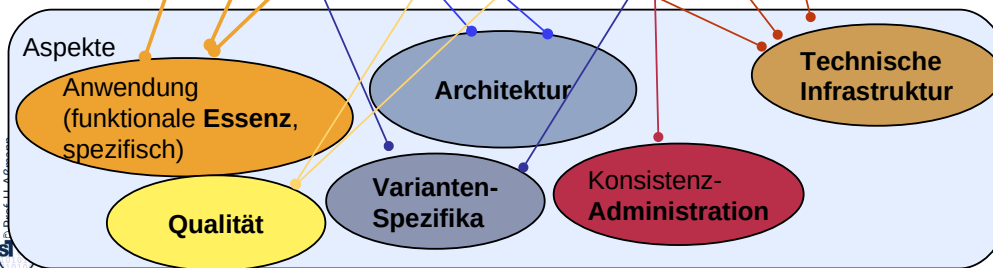
## 41.3 Architekturprinzip “Aspekttrennung”



## Wesentliche Aspekte eines Softwaresystems

21 Softwaretechnologie (ST)

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

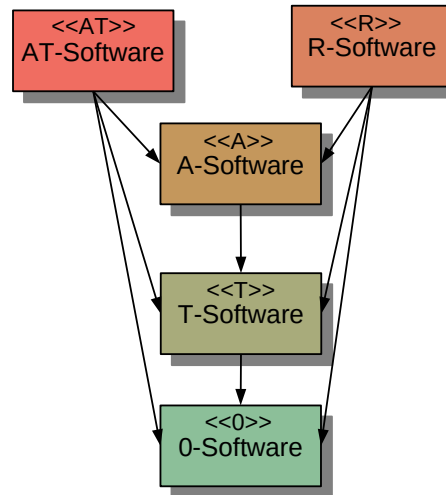


Programme dienen vielen Zwecken, d.h. sind zu vielen Aspekten korreliert, die in verschiedenen Perspektiven zusammengefasst werden können.

### 4.3.1 Architekturprinzip Quasar: Trennung von Technik- und Anwendungskomponenten (Reuse Blood Groups, Blutgruppen)

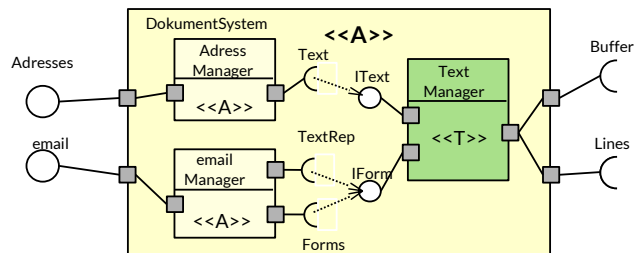
Quasar definiert 4 Aspekte (Wiederverwendungs-Blutgruppen), Softwarekategorien für Komponenten, nach Wiederverwendbarkeit:

- ▶ 0: unabhängig von Anwendung und Technologie
  - JDK collections, C++ STL, GNU regex
- ▶ A: anwendungs- oder domänenspezifisch.
  - Client, Customer, Account, Car, ...
- ▶ T: technologie-orientierte Schnittstelle, unabhängig von Anwendung
  - OSGI, JDBC, CORBA CosNaming
- ▶ AT: abhängig von Anwendung *und* Technologie
  - schwierig zu isolieren und wieder zu verwenden
- ▶ R: Repräsentationswechsel von Daten
  - Serialisierung, Deserialisierung, Verschlüsselung
  - Sprachwechsel (z.B. Java to Cobol)



# Architekturprinzip Quasar: Trennung von Technik- und Anwendungskomponenten

- ▶ Jede Komponente wird klassifiziert in Blutgruppen O, T, A, AT, R
  - O – technologieunabhängige Algorithmen,
  - T – technologieabh. Komponenten,
  - A – Anwendungskomponenten,
  - R – Repräsentationwechselkomponenten

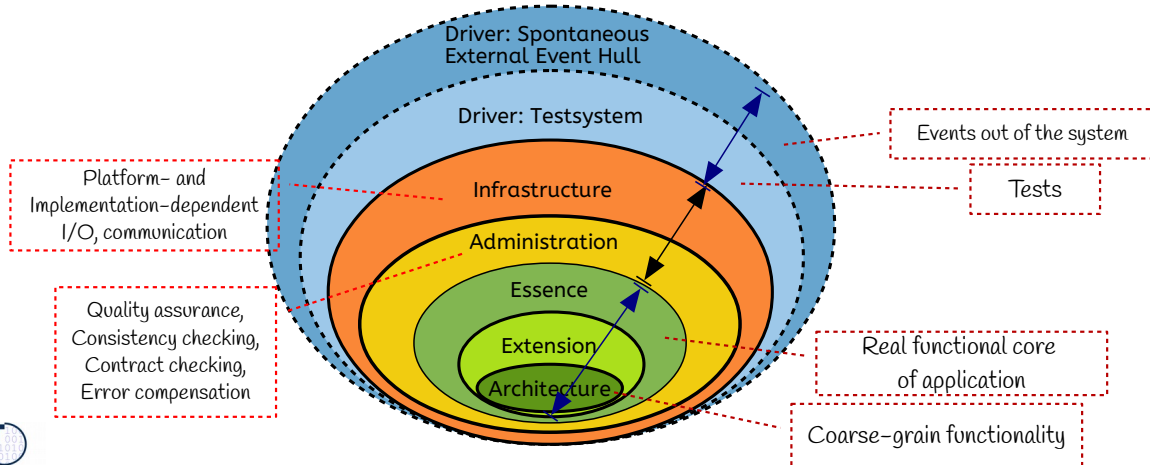


**[Siedersleben] Quasar-Wiederverwendungsgesetz:**  
O- und T-Komponenten sind besser wiederverwendbar als Anwendungskomponenten.  
AT-Komponenten sind sehr schlecht wiederverwendbar.

## 4.3.2. Olympische Dekomposition von Software in Ringe (Essentielle Dekomposition)



- ▶ **Software** hat 5 Ringe (*olympische* oder *essentielle Dekomposition* in 5 Aspekte):
  - **(Funktionale) Essenz** sind Funktionen unabhängig von der unterliegenden Technologie
    - **Architektur** ist ein Unter-Ring der Essenz, der grobkörnige Funktionalität liefert
  - **Administration** sichert die Qualität des Systems
  - **Infrastruktur (Middleware)** bietet die technologieabhängigen Funktionen an
  - **2 Externe Treiber-Ringe** treiben das System: entweder die Umgebung, die spontan Ereignisse und Eingabedaten generiert, oder das **Testsystem**



**Software** hat 5 Ringe (*olympische* oder *essentielle Dekomposition* in 5 Aspekte):

**(Funktionale) Essenz** sind Funktionen unabhängig von der unterliegenden Technologie

Essenz nimmt **perfekte Technologie** an, z.B. Prozesse ohne Zeit, unendlichen Speicher, unendliche Bandbreite

**Architektur** ist ein Unter-Ring der Essenz, der grobkörnige Funktionalität liefert

**Administration** sichert die Qualität des Systems (Vertragsprüfung, Ausnahmen, Datenkonsistenz).

**Infrastruktur (Middleware)** bietet die technologieabhängigen Funktionen an

**Treiber** treiben das System: entweder die Umgebung, die spontan Ereignisse und Eingabedaten generiert, oder das Testsystem

Administration und Infrastruktur bilden die **physikalischen Ringe**; Treiber, Essenz und Architektur die **logischen Ringe**

Warum braucht man zur Wechsel auf eine neue Plattform eine andere Implementierung des Infrastruktur-Rings?

## Trennung von Ringen

- ▶ Olympische Dekomposition definiert 5 Aspekte, Softwarekategorien für Komponenten, nach Zweck:
  - E: Essenz
  - Arch: Architektur
  - Admin: Administration
  - I: Infrastruktur, Plattform
- ▶ Die Ringe durchdringen einander, d.h. bilden keine Schichten
- ▶ Jede Komponente wird **genau einem Ring** zugeordnet, ansonsten Mischmasch
- ▶ Ähnlich wie bei Quasar

<<I>>  
I-Software

<<Admin>>  
Admin-  
Software

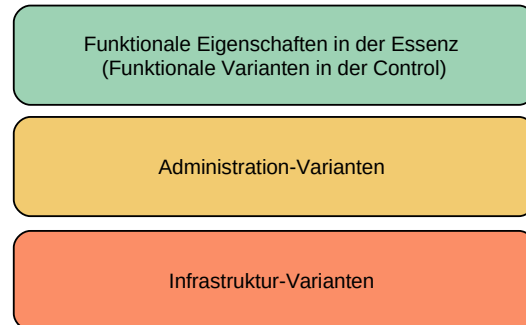
<<Arch>>  
Arch-Software

<<E>>  
E-Software



### 41.3.3. Architekturprinzip: Olympische Variabilität mit Ringen

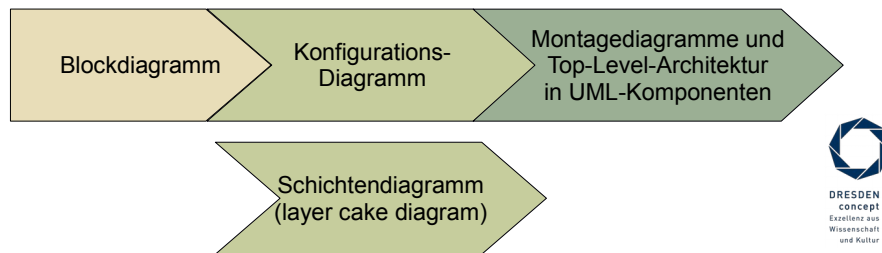
- ▶ **Software-Produktlinien** entstehen durch systematische Variation von *Ringen*



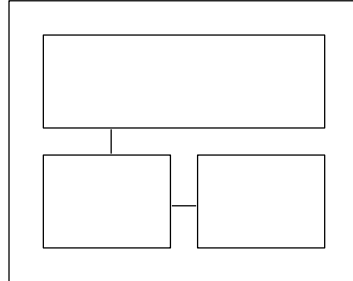




## 41.4 Diagrammarten für die logische Struktur



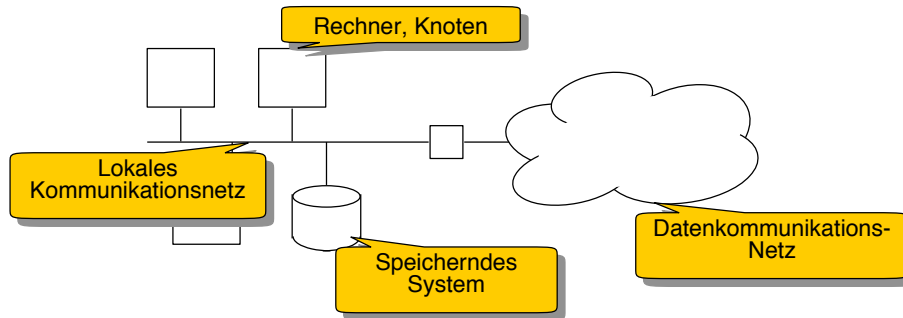
- ▶ Def.: Ein **Blockdiagramm** skizziert informell die logische Struktur einer Architektur
  - verbreitetes, informelles Hilfsmittel
  - Blockdiagramme sind kein Bestandteil von UML; Vorstufe von Montagediagrammen
  - **Blöcke** stellen UML-Komponenten *ohne Ports* dar



# Strukturperspektive

## Konfigurationsdiagramme für physikalische Verteilung

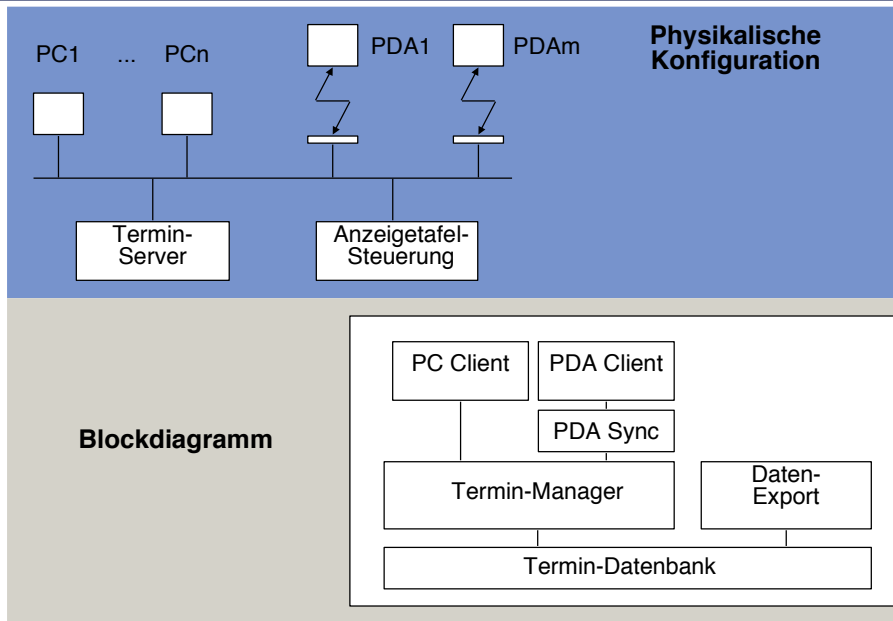
- ▶ Def.: Ein Konfigurationsdiagramm ist ein Blockdiagramm mit "Bussen" zur Beschreibung der **physischen Sicht (Verteilungssicht)**
  - Konfigurationsdiagramme sind zwar nicht Bestandteil von UML, aber dennoch ein verbreitetes Hilfsmittel zur Beschreibung der physikalischen Verteilung



# Verteilungsperspektive

## Beispiel: Konfigurationsdiagramm für Terminverwaltung

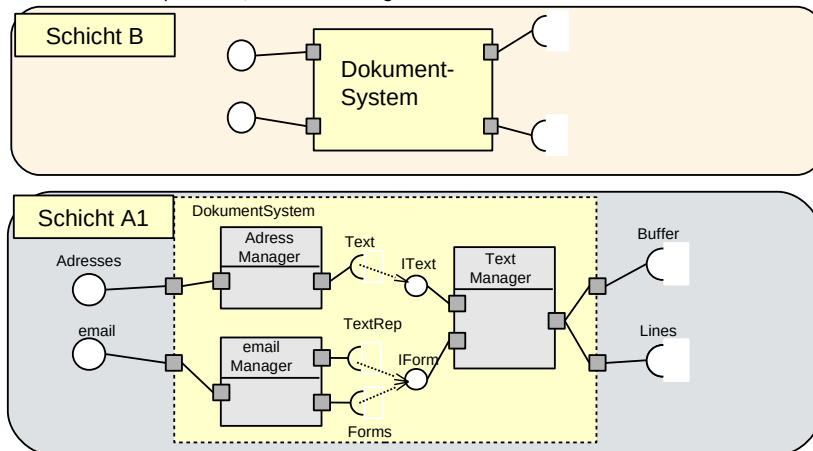
30 Softwaretechnologie (ST)



# Verteilungsperspektive

# Logische Struktur nicht-Interaktiver Anwendungen: Montagediagramme mit UML-Komponenten für die obersten Ebenen des Systems

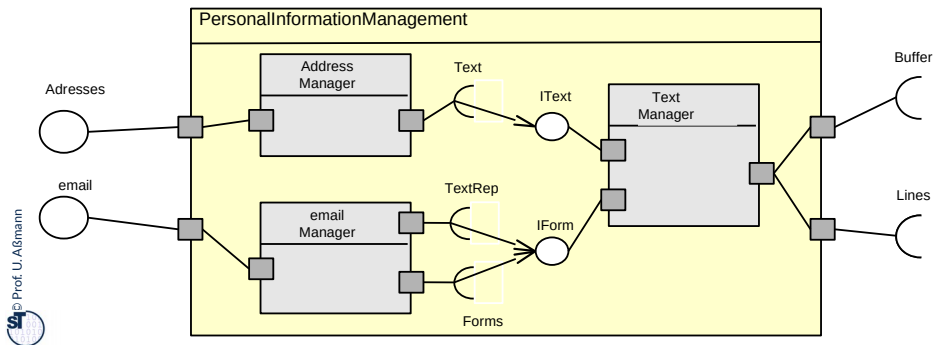
- ▶ Aus einem Blockdiagramm der Architektur des Systems wird ein Montagediagramm für Top-Level-Architektur entwickelt
  - Oberste Ebene des Systems ist meist hierarchisch und/oder geschichtet organisiert
  - Vermeide "wilde" objekt-orientierte Netzstrukturen
  - Damit die letzte Integration zum Gesamtsystem einfach verläuft: Integrationstests können dann bottom-up absolviert werden
- Hierarchien bilden Spezialfälle, denn sie können geschichtet werden

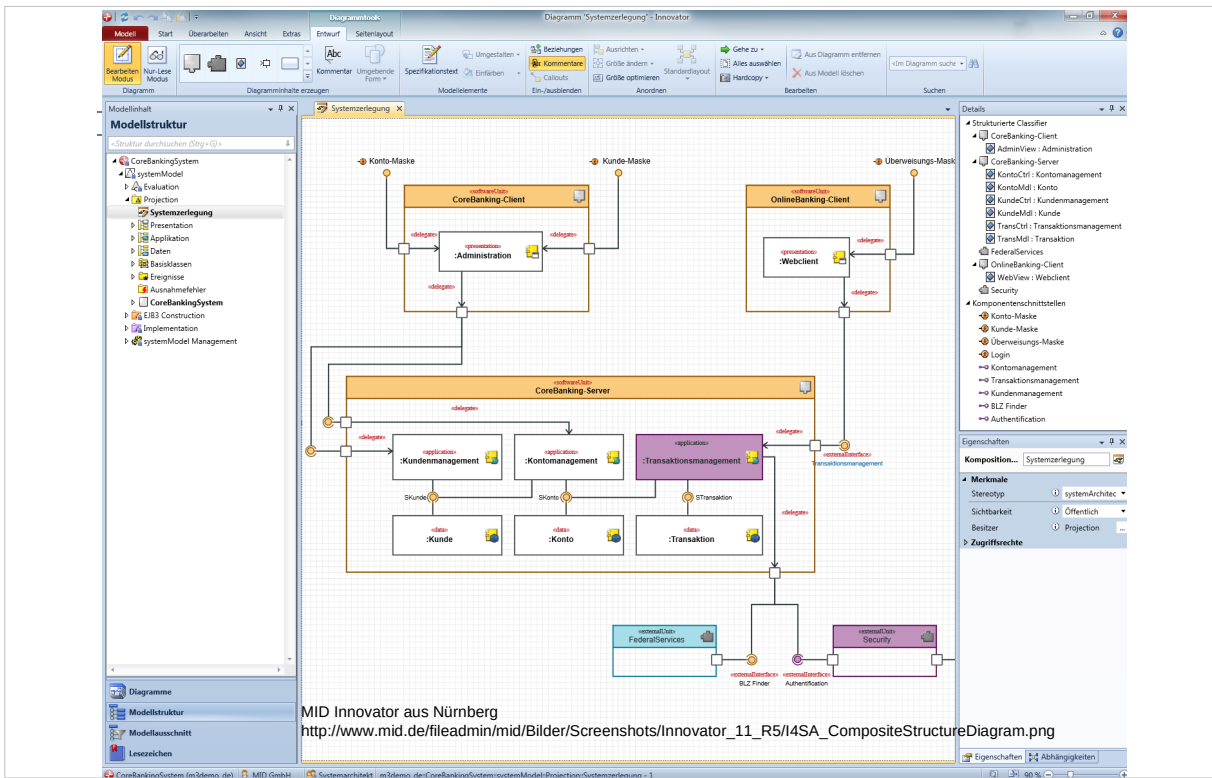


## Strukturperspektive

## Draufsicht auf die Schichten: Schachtelung von Klassen zu UML-Komponenten

- ▶ Die Schachtelung von Komponenten führt zu hierarchischen Systemen, die schichtbar sind, d.h. Jede Ebene bildet eine Schicht
- Implementierung mit **Facade Pattern**: Komponente spielt eine Facade für die Unterkomponenten einer Schicht
- Verfeinerung des Kontextmodells in die Top-Level-Architektur erzeugt eine weitere Schicht







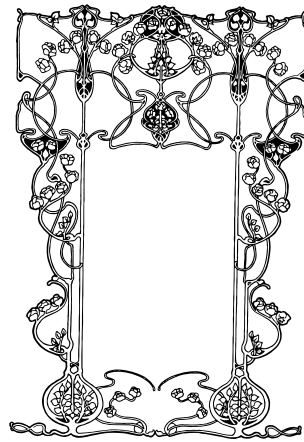
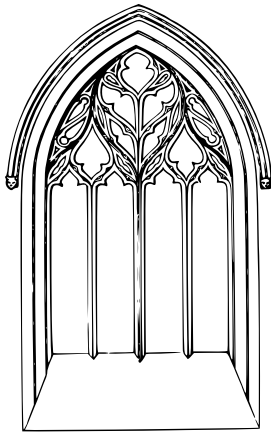
## 41.5 Architekturstile

Def.: Ein **Architekturstil** legt für alle Komponenten und Aspekte des Systems Randbedingungen und Einschränkungen fest.





- ▶ Ein Architekturstil legt für alle Komponenten und Aspekte des Systems Randbedingungen und Einschränkungen fest.





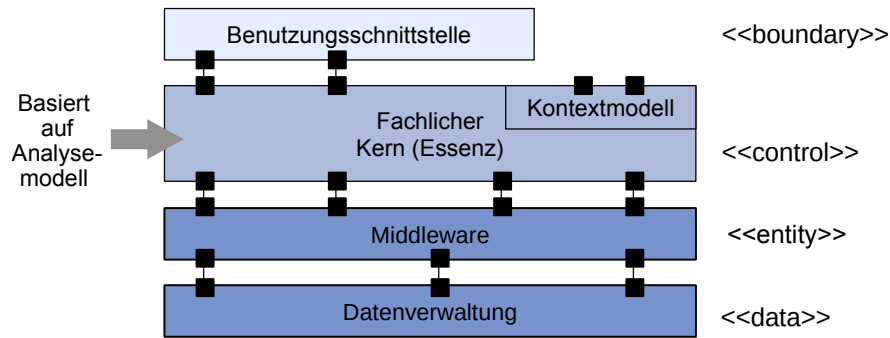
## 41.4.1 Architekturprinzip Schichtung und Architekturstil “Geschichtete Systeme” (Layered Architectural Style)

Schichten kapseln kohärentes Wissen und erzeugen lose  
Kopplung durch die “vertraut-auf”-Relation



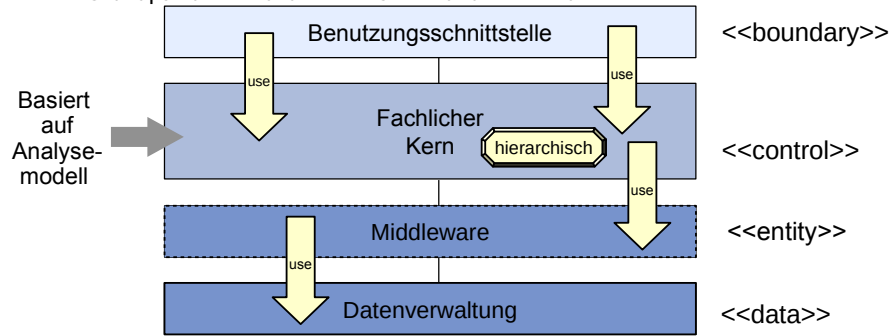
# Architekturstil für interaktiver Anwendungen: Vier-Schichten-Architektur (BCED)

- ▶ Klassische Struktur eines interaktiven Anwendungssystems
- ▶ Schichten sind jeweils stark kohäsiv, und lose gekoppelt – warum?
  - Schichten haben angebotene Schnittstellen nach oben und benötigte Schnittstellen nach unten
  - 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



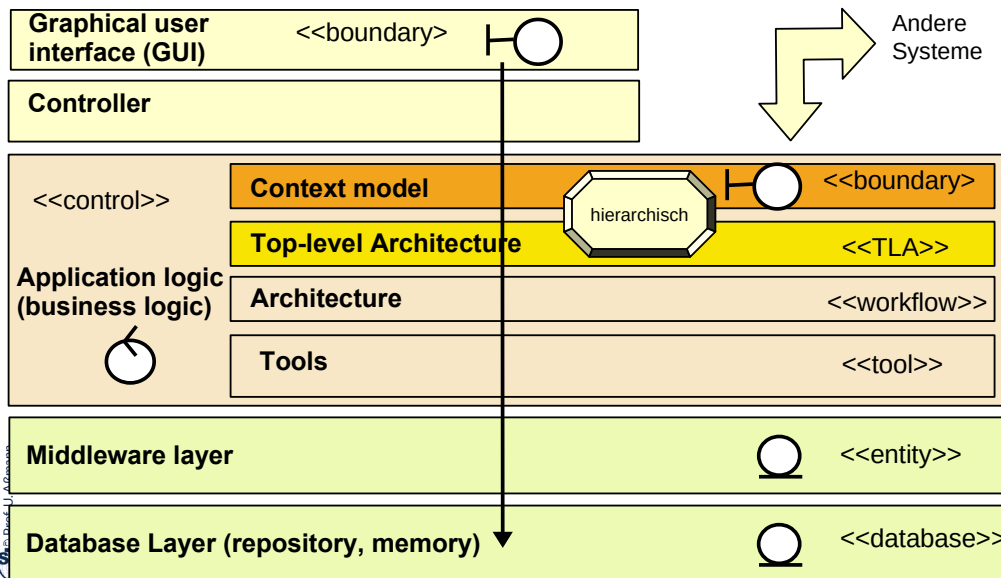
# Architekturstil für interaktiver Anwendungen: Vier-Schichten-Architektur (BCED)

- ▶ Def.: Ein **Schichtendiagramm** ist ein geschichtetes Blockdiagramm mit Benutzungsrelation
  - verwendet für die klassische Struktur eines interaktiven Anwendungssystems
- **Einschränkung des Architekturstils:** azyklische Benutzungsrelation (use)
- ▶ **Geschichteter Integrationstest** verläuft bottom-up: erst D, dann ED, dann CED, dann BCED
- ▶ Fachlicher Kern (Anwendungslogik) kann weitere Schichten enthalten
  - Oft kapselt eine Facade eine Schicht nach oben ab



## Struktursicht: 8-Schichtenmodell, eine Verfeinerung des BCED-Schichtung eines Systems

39 Softwaretechnologie (ST)



Das Schichtendiagramm einer klassischen interaktiven Anwendung

## Schichtung beruht auf der "Vertraut-auf"-Relation (Relies-On, USES, Sees-A)

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

- ▶ *benötigt eine korrekte Implementierung* beinhaltet:
  - A ruft auf B, d.h. A delegiert Arbeit auf B oder B delegiert Arbeit zurück auf A
  - A greift zu auf öffentliche Variable oder Objekt von B
  - A nutzt eine Ressource von B
  - A alloziert ein Objekt von B
  - 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

## Verschiedene Relationen zwischen Komponenten

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

## Geschichteter Test von hierarchischen und geschichteten Systemen

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

Integrationstests sollten bottom-up, aufwärts entlang der USES-Relation geschehen

## Frage: wie testet man ein interaktives Informationssystem?

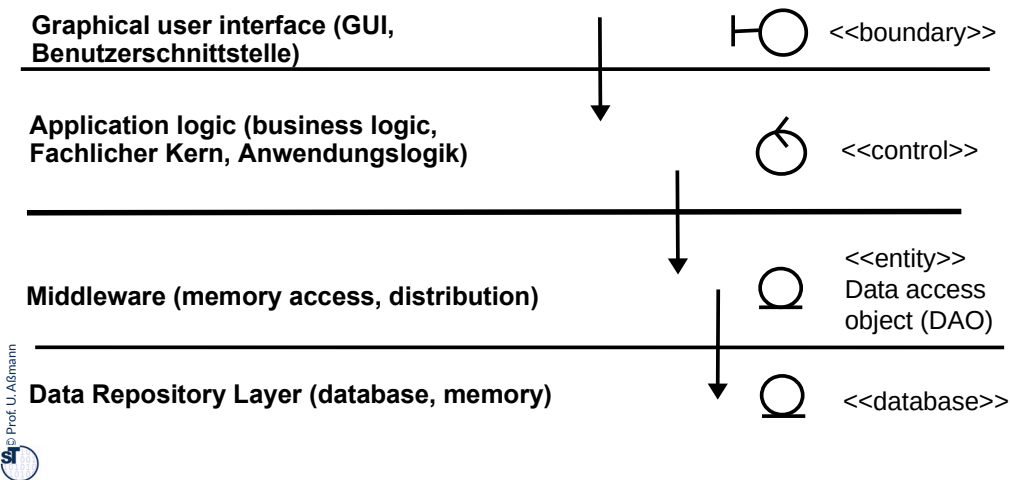
- ▶ .. wie es im Projekt-Praktikum im WS vorkommt



## Antwort: Bottom-Up! USES-Relation in 4-Tier Architekturen (BCED)

44 Softwaretechnologie (ST)

- ▶ 4-Schichtarchitekturen nutzen eine azyklische USES-Relation
  - Obere Schichten nutzen untere, aber nicht umgekehrt



### Vorteile der Schichtenarchitektur:

- Kohäsion in einer Schicht
- Niedrige Kopplung durch azyklische USES-Beziehung
- Gute Austauschbarkeit der Schichten
  - GUI kapselt user interface
  - Data repository layer kapselt die Speicherung der Daten und die Persistenz
  - Middleware behandelt die Verteilung
- Der BCD/BCED Architekturstil ist der Haupt-Architekturstil für interaktive Anwendungen
- ... auch für Ihr Projekt im Praktikum

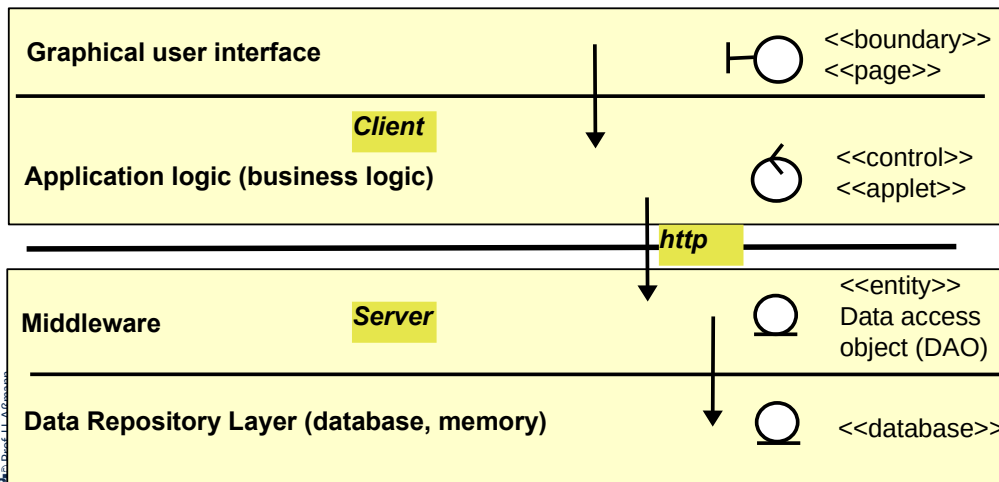


## 41.5.2 Andere geschichtete Systeme



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

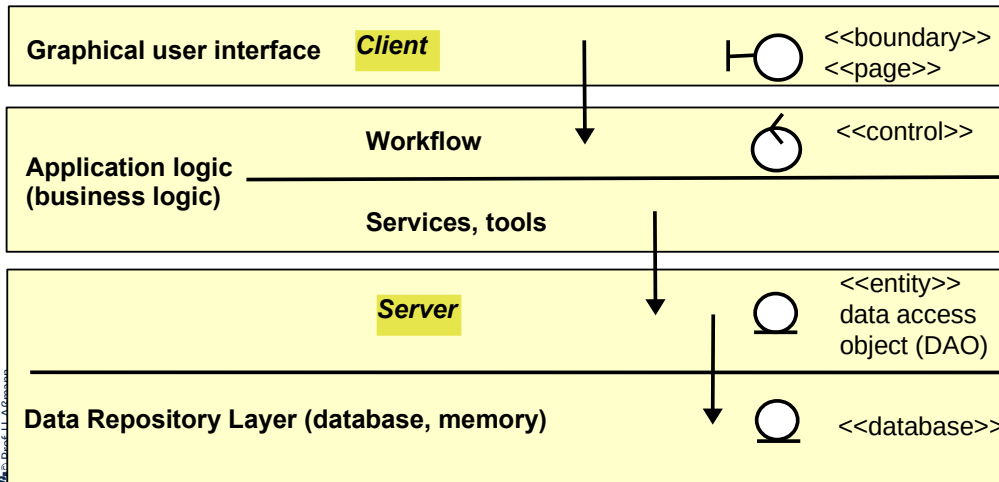
- ▶ "Thick client" Web-Systeme nutzen eine http-basierte Middleware
- ▶ GUI und AL verbleiben auf dem Client, die Daten werden auf dem Server verwaltet





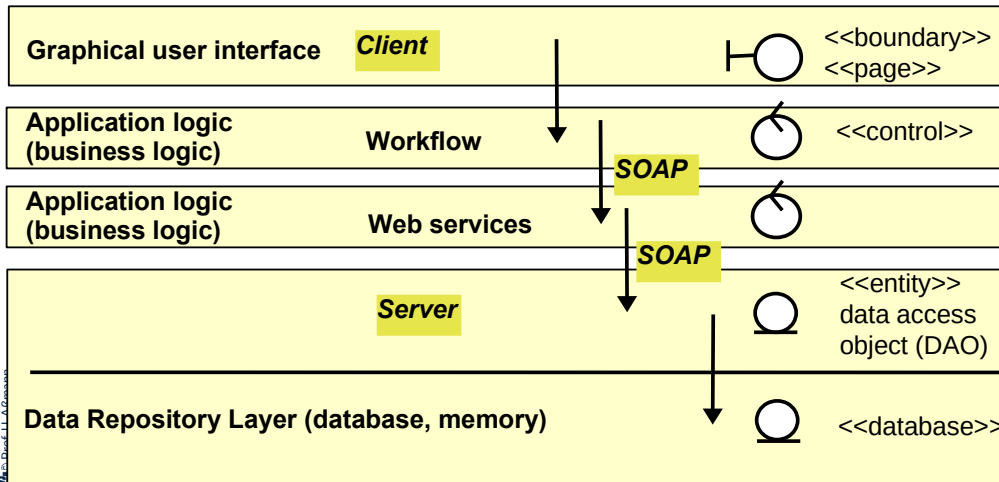
## Beispiel: 5-Tier mit Workflow Language

- ▶ Arbeitsfluss-Sprachen (Workflow languages) wie BPMN, BPEL definieren die Arbeitsfluss-Schicht der AL, unterhalb der Top-Level-Architektur
  - Services und Tools arbeiten auf den Daten



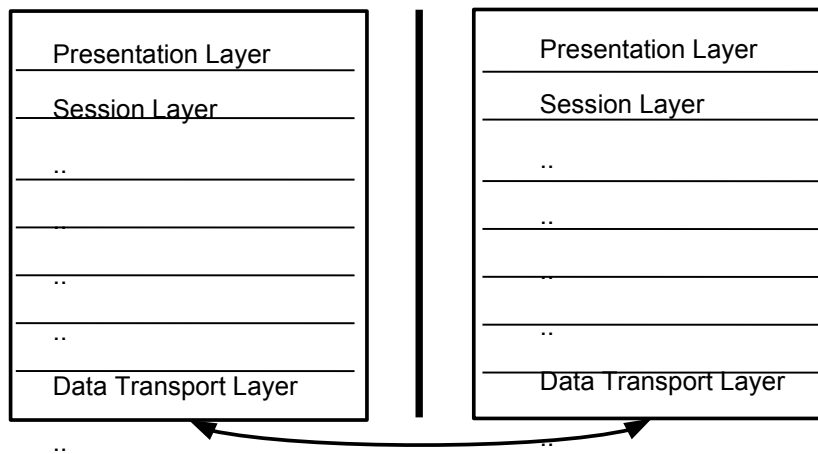
## Beispiel: 5-Tier mit Workflow Language und Web Services

- ▶ Arbeitsfluss-Sprachen können auch *Web services* ansteuern, dann ist die Anwendung verteilt



## Weiteres Beispiel: ISO-OSI 7-Schichten Netzwerk-Architektur

- ▶ Jede Schicht enthält ein Stromobjekt für bidirektionale Kanalkommunikation



# Beispiel: Betriebssysteme

UNIX,  
Linux,  
Android

User Space

Kernel

Apple-  
UNIX,  
iOS

User Space

Kernel

Microkernel (Mach)

Windows NT/XP:

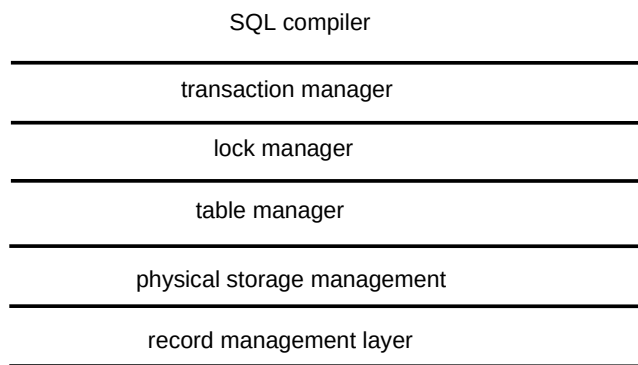
User Space

Kernel

Hardware Abstraction Layer (HAL)







## Warum sind geschichtete Architekturen wichtig?

- ▶ Der “Layered architecture style” benötigt eine azyklische USES-Relation
- ▶ Vorteile:
  - Jede Schicht wirkt von außen wie eine einzige Komponente mit angebotenen und benötigten Schnittstellen (Entwurfsmuster Facade)
  - Kohäsion stark, Kopplung gering
  - Veränderungsorientierter Entwurf, Austausch von Schichten möglich → Evolution einfach
  - Verantwortlichkeiten für Testmanagement können klar definiert werden: Für jede Schicht werden separate Testsuites entwickelt (Bottom-up tests)



## 41.6. Architektur mit Perspektivenmodellen

Einer Architektur liegt eine Perspektive zugrunde, die mehrere Sichten auf das System definiert.

Ein **Perspektivenmodell (view model)** definiert eine Menge von *Perspektiven mit Aspekten und ihren Sichten*, von denen die Software aufgeteilt wird

In UML: Ein Perspektivenmodell definiert ein Profil von Stereotypen, die auf Fragmente eines UML-Modells aufgeklebt werden können





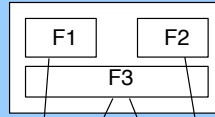
## Inhalte von Perspektivenmodellen

57 Softwaretechnologie (ST)

- ▶ Folgende Perspektiven (Gruppen von Aspekten und ihren Sichten) werden in Perspektivenmodellen unterschieden:

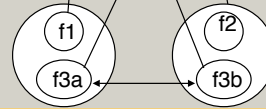
- ▶ **Logische Sicht mit struktureller Zerlegung:**

- Struktur im Großen: Blockdiagramme, Montagediagramme (UML-Komponentendiagramme)
- Architekturstil: Schichten, Sichten, Dimensionen
- Logischer Detail-Entwurf



- ▶ **Verteilungssicht:** Struktur der physikalischen Verteilung:

- Zentral oder verteilt? Topologie



- ▶ **Dynamische Sicht (Ablaufsicht)**

- Prozesse, Synchronisation
- Flüchtigkeit und Persistenz von Daten

- ▶ **Qualitätssicht:** Einhaltung nichtfunktionaler Anforderungen

- Architekturbestimmende Eigenschaften (z.B. Realzeitsystem, eingebettetes System)
- Effizienzanforderungen und Optimierung
- Standardarchitekturen

Programmieren im Großen ist nicht dasselbe wie Programmieren im Kleinen.

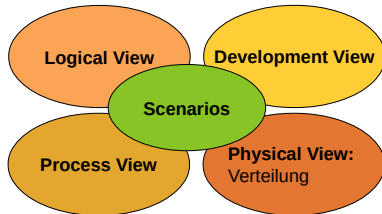
Für beides müssen spezifische Sprachen benutzt werden:

- Programmiersprache (z.B. Java)
- Architektursprache (z.B. UML-Komponentendiagramme)

## Weitere Beispiele für Perspektivenmodelle (View Models)

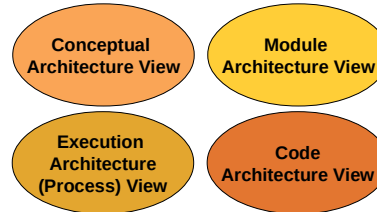
### 4+1 Views von Kruchten:

- 1) **Logical View:** logische Struktursicht in Klassen, Komponenten
- 2) **Development View:** Entwicklungssicht
- 3) **Process View:** Prozess-Sicht
- 4) **Physical View:** Verteilung, nicht-funktionale Eigenschaften
- 5) + **Scenarios**, die alle anderen Sichten querschneiden



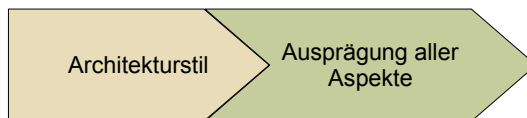
### Hofmeister/Soni/Nord:

- 1) **Conceptual Architecture View:** logische Struktursicht aus Komponenten und Konnektoren
- 2) **Module Architecture View:** Struktursicht des Feinentwurfs mit Komponenten und Klassen
- 3) **Execution Architecture View:** Prozess-Sicht
- 4) **Code Architecture View:** Arrangement der Code-Dateien im Filesystem



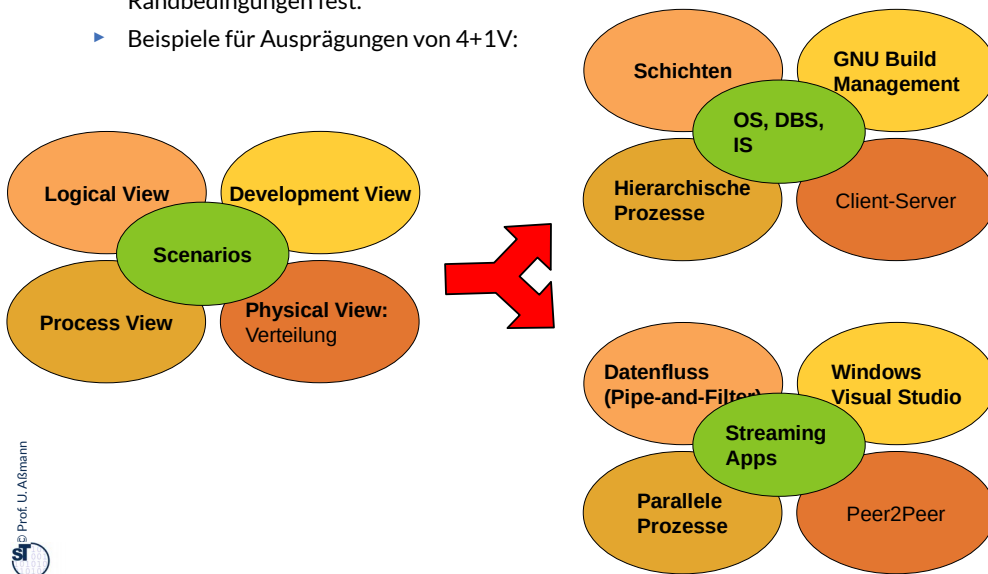


## 41.6.1 Perspektivenmodelle und Architekturstile



# Architekturstile im Perspektivenmodell "4+1 Views"

- ▶ Def.: Ein *Architekturstil* legt für jeden Aspekt eines Perspektivenmodells Randbedingungen fest.
- ▶ Beispiele für Ausprägungen von 4+1V:





## Was haben wir gelernt?

- ▶ Architektur trennt die Aspekte des Programmierens im Großen vom Programmieren im Kleinen
- ▶ Der Architekturstil der Anwendung muss festgelegt werden und spielt eine große Rolle im Architekturentwurf
- ▶ Ein Perspektivenmodell hilft uns, Software zu in Aspekte und Sichten zu gliedern
- ▶ Schichtenbasierte Architekturen sind wichtig
  - Azyklische USES (relies-on) Relation





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



- ▶ Erklären Sie, warum das Kontextmodell aus hierarchischen Komponenten besteht.
- ▶ Was ist der Unterschied zwischen Blockdiagramm, Montagediagramm und Verteilungsdiagramm?
- ▶ Welche architektonische Sichten auf ein System kennen Sie?
- ▶ Erklären Sie Kruchten's 4+1 Perspektivenmodell
- ▶ Warum bildet eine Komponente eine Instanz des Entwurfsmusters Facade?
- ▶ Erklären Sie, wann ein System eine Schichtung besitzt. Warum sind Schichten wichtig?
- ▶ Erklären Sie das 4-Schichtenmodell und vergleichen Sie es mit seiner Verfeinerung, dem 8-Schichtenmodell.
- ▶ Erklären Sie den Unterschied zwischen Schichten und Ringen.
- ▶ Warum brauchen wir Sichten für eine gute Softwarearchitektur?



## Repet.: BCD/BCED Classification

- ▶ 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**

