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 17-0.1, 17.06.17

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

Obligatorische Literatur

- 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

© Prof. U. Aßmann

Exkurs: Wie findet man ein Papier?

- http://scholar.google.de
- http://liinwww.ira.uka.de/bibliography/index.html#search

© Prof. U. Aßmann

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.

© Prof. U. Aßmann

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 Objektanreicherung**
4) 43: Architektur interaktiver Systeme
5) 44: Punktweise Verfeinerung von Lebenszyklen
   - Verfeinerung von verschiedenen Steuerungsmaschinen

© Prof. U. Aßmann

41.1 Grundlegende Architekturprinzipien

41.2.1 Aspekttrennung 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*, von denen die Software aufgeteilt wird

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

Wesentliche Aspekte und Bestandteile eines Softwaresystems

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

Aspekte

Anwendung (funktionale **Essenz**, spezifisch)

Architektur

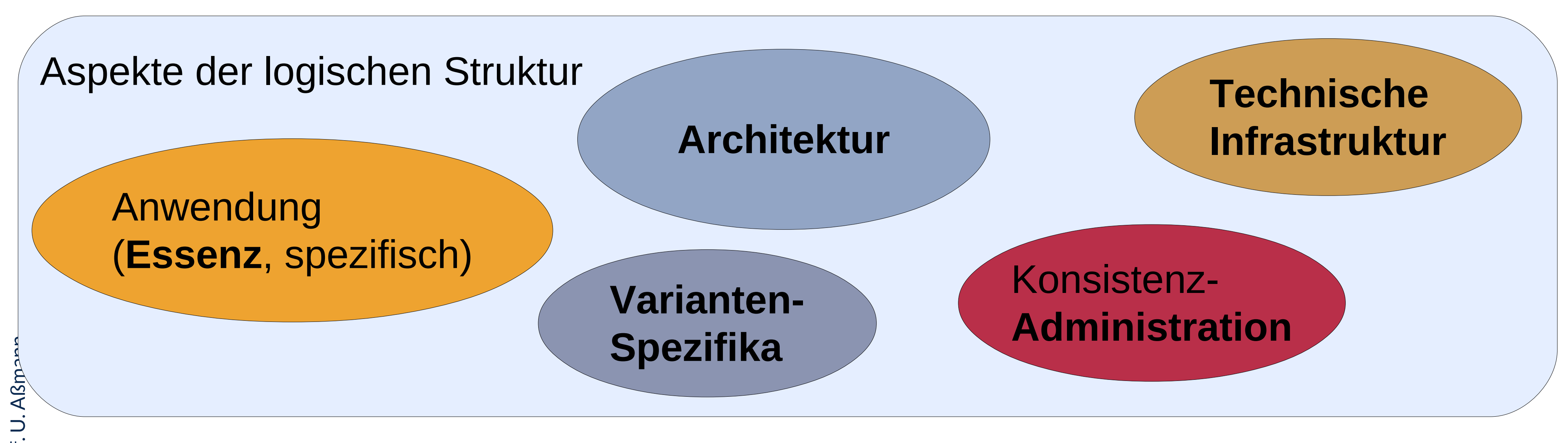
Technische Infrastruktur

Konsistenz-**Administration**

© Prof. U. Aßmann

Perspektive der logischen Struktur

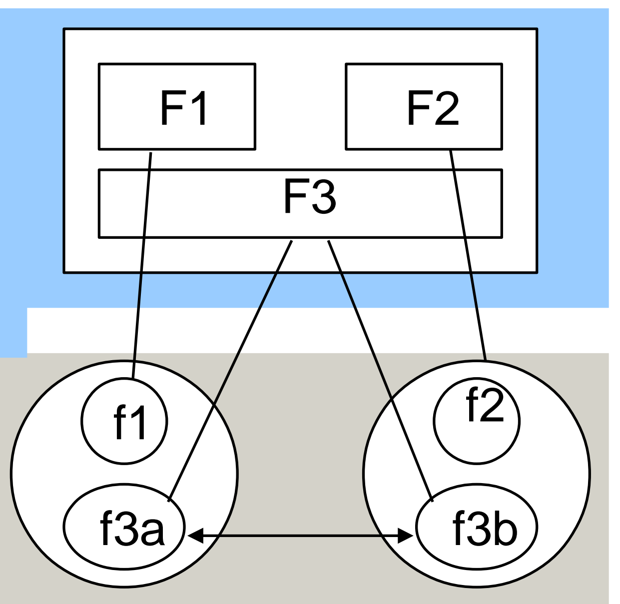
- Ein **Aspekt** definiert eine *Sicht* auf ein System, legt eine Teilmenge des Systems fest
- Eine **Perspektive** gruppiert eine Menge von Aspekten und deren Sichten
- Ein **Perspektivenmodell** definiert eine Menge von Perspektiven, die Aspekten und Sichten gruppieren
  - Zu jeder Perspektive wird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



© Prof. U. Aßmann

4.2.1.1 Perspektivenmodell “Programmieren im Großen I.G. zu Programmieren im Kleinen” Aspekt-Trennung im Architekturentwurf

- Folgende Aspekte 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

© Prof. U. Aßmann

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:

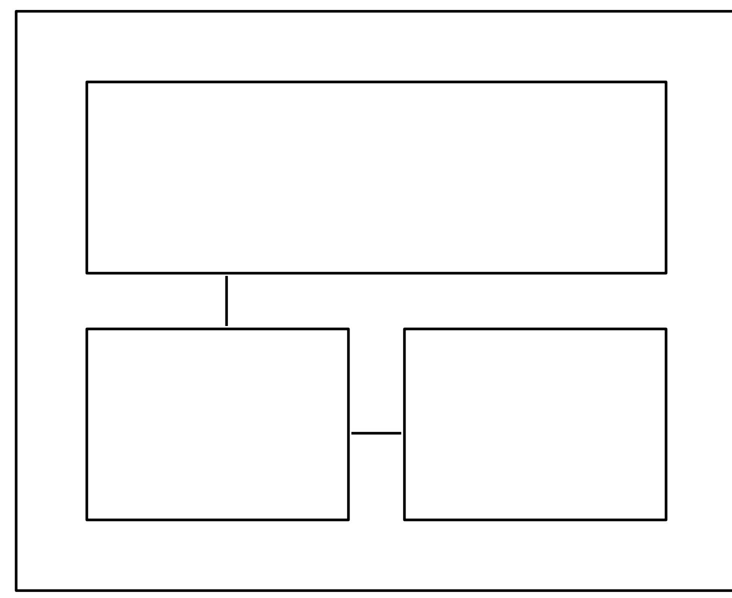
1) **Conceptual Architecture View:** logische Struktursicht aus Komponenten und Konnektoren
2) **Module Architecture View:** Struktursicht des Feinentwurfs mit Komponenten, Modulen und Klassen
3) **Execution Architecture View:** Prozess-Sicht
4) **Code Architecture View:** Arrangement der Code-Dateien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

- Siehe auch Wikipedia->View_Models

© Prof. U. Aßmann

Blockdiagramme zur logischen Struktur eines Systems

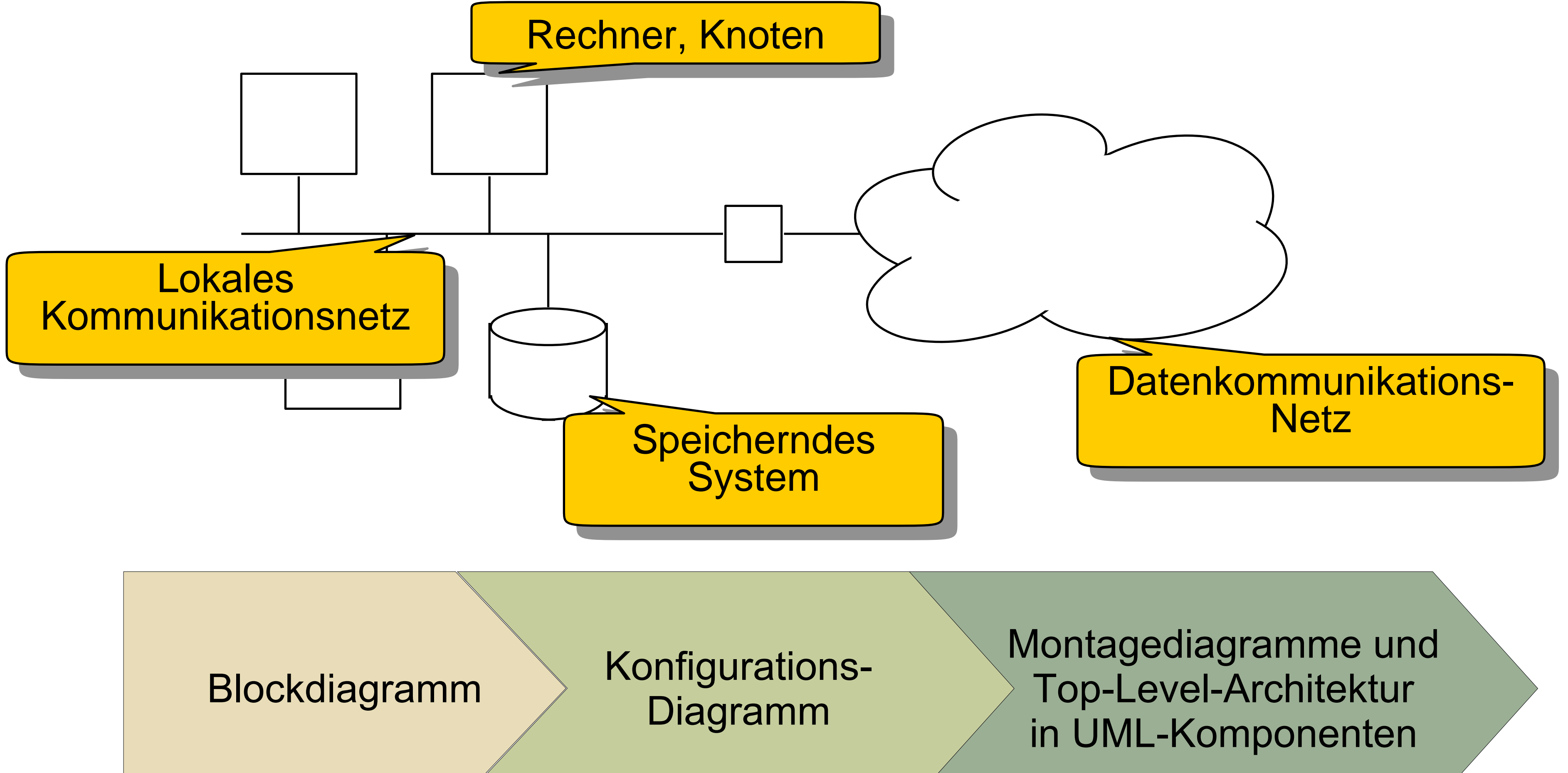
- Ein **Blockdiagramm** skizziert die logische Struktur einer Architektur
  - verbreitetes, informelles Hilfsmittel
  - Blockdiagramme sind kein Bestandteil von UML; Vorstufe von Montagediagrammen
  - **Blöcke** stellen UML-Komponenten *ohne Ports* dar



© Prof. U. Aßmann

Konfigurationsdiagramme für physikalische Verteilung

- Konfigurationsdiagramme sind Blockdiagramme mit “Bussen” zur Beschreibung der **physischen Sicht**
  - Konfigurationsdiagramme sind zwar nicht Bestandteil von UML, aber dennoch ein verbreitetes Hilfsmittel zur Beschreibung der physikalischen Verteilung



© Prof. U. Aßmann

Beispiel: Konfigurationsdiagramm für Terminverwaltung

Physikalische Konfiguration

PC1 ... PCn

PDA1

PDAm

Termin-Server

Anzeigetafel-Steuerung

Blockdiagramm

PC Client

PDA Client

PDA Sync

Termin-Manager

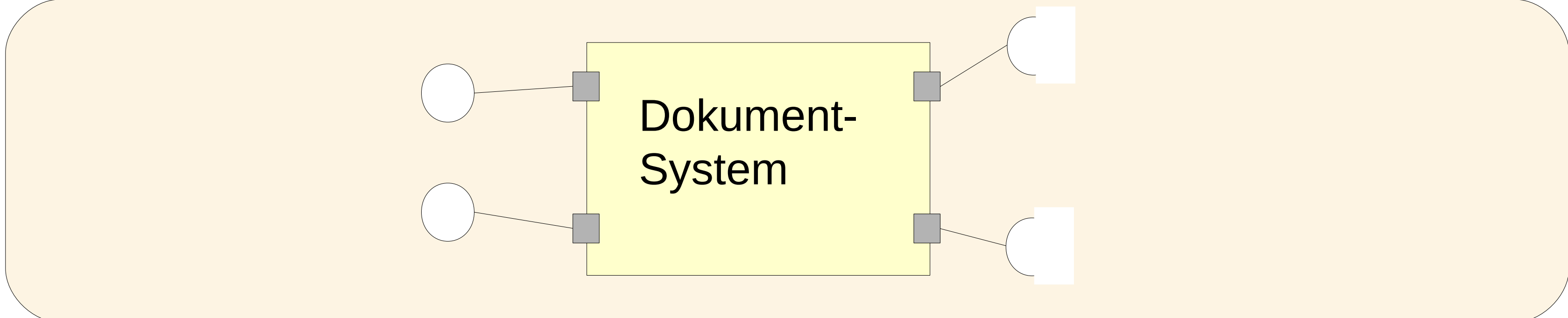
Daten-Export

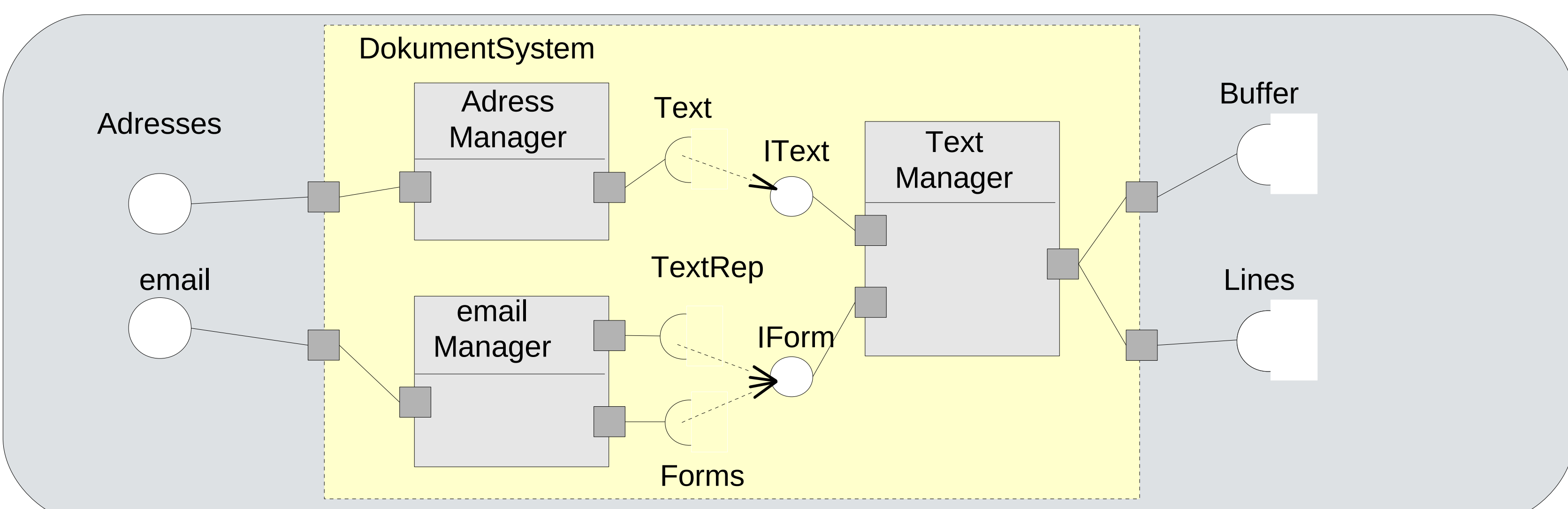
Termin-Datenbank

© Prof. U. Aßmann

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





© Prof. U. Aßmann

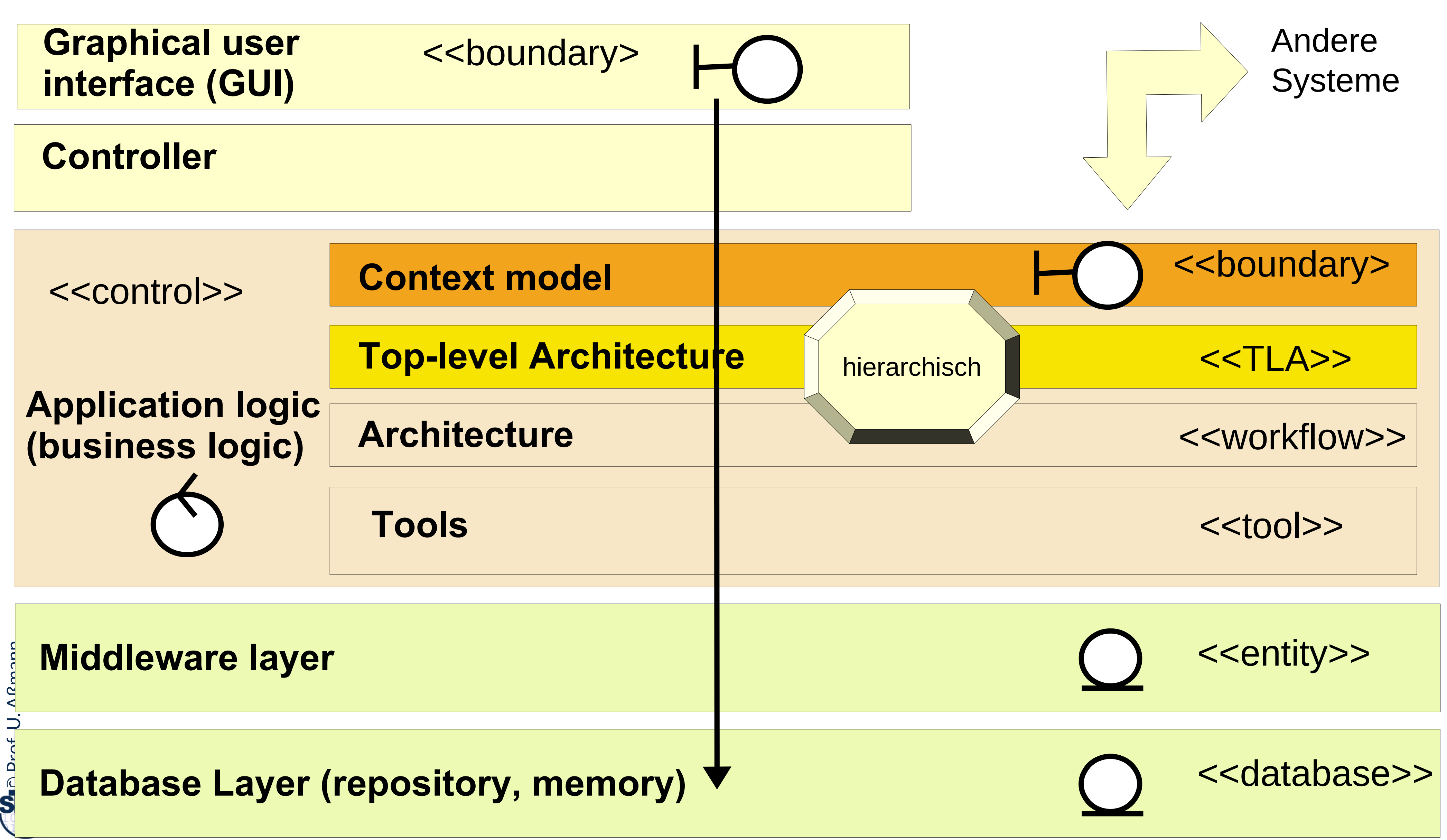
Architekturstil für Struktursicht Interaktiver Anwendungen: Vier-Schichten-Architektur (BCED)

- Ein **Schichtendiagramm** ist ein geschichtetes Blockdiagramm für die klassische Struktur eines interaktiven Anwendungssystems
- **Geschichteter Integrationstest** verläuft bottom-up
  - azyklischer Benutzungsrelation (use): erst D, dann ED, dann CED, dann BCED
- Fachlicher Kern (Anwendungslogik) kann weitere Schichten enthalten
  - Oft kapselt eine Facade eine Schicht nach oben ab, dann existieren bereits zwei Teil-Schichten

© Prof. U. Aßmann

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

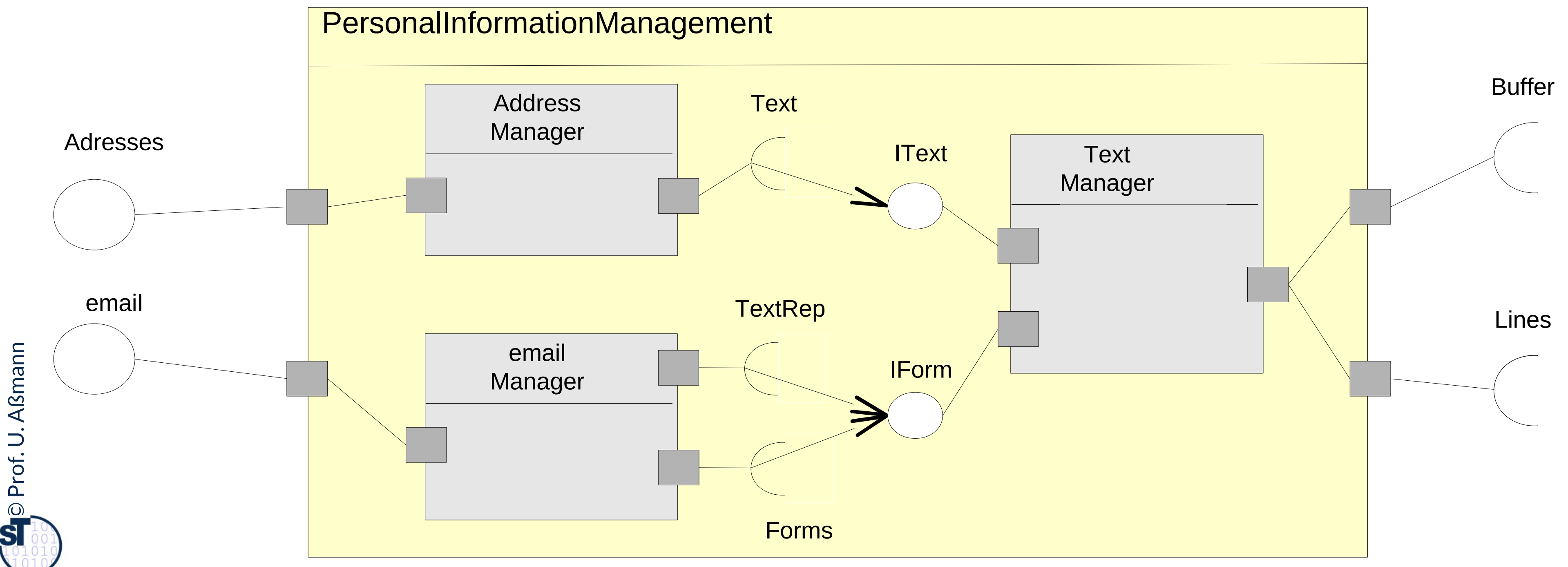
- Im Folgenden verwenden wir mehr als 8 Schichten:



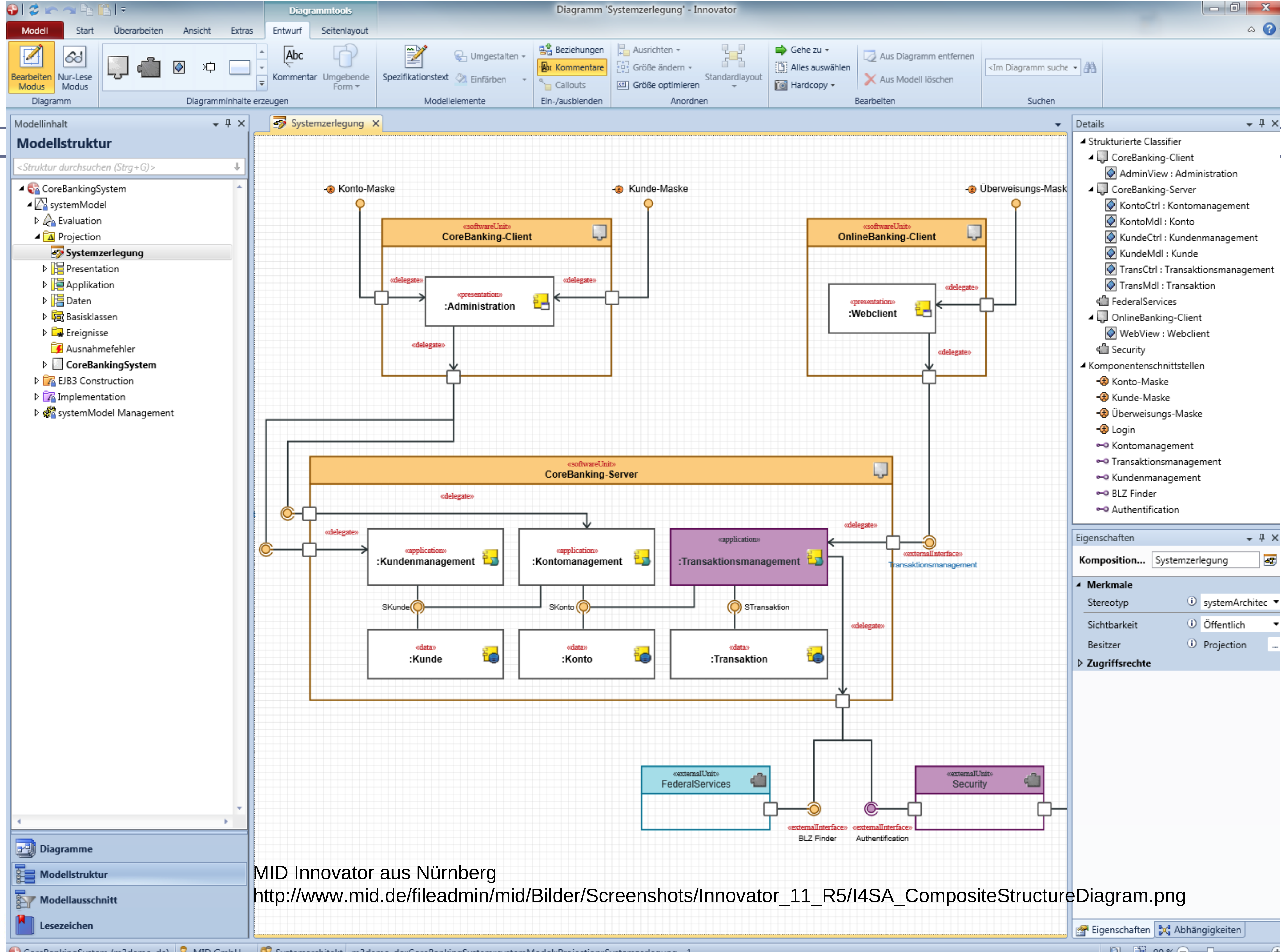
© Prof. U. Aßmann

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



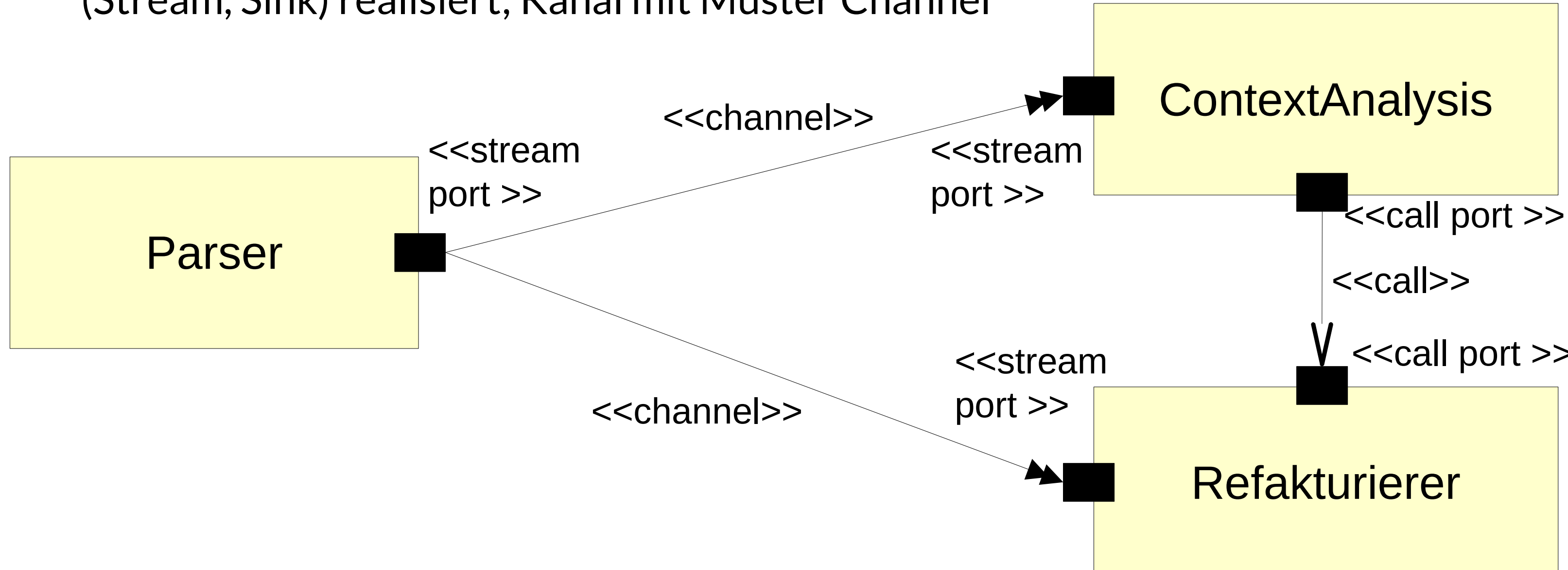
© Prof. U. Aßmann



MID Innovator aus Nürnberg
http://www.mid.de/fileadmin/mid/Bilder/Screenshots/Innovator_11_R5/I4SA_CompositeStructureDiagram.png

Strom-Anschlüsse von Komponentenklassen und Kanal-Konnektoren

- Ein ***Aufrufsanschluß (call port)*** ist eine Portschnittstelle, die angebotene oder benötigte Operationen enthält, über die es synchron aufgerufen wird oder synchron aufruft
- Ein ***Stromanschluss (stream port)*** ist eine Portschnittstelle, die einen Ein- oder Ausgabestrom enthält, über den kontinuierlich Daten ein und aus fließen (Iterator-Muster)
  - Ein Stromanschluß läßt auf ein aktives Objekt (Prozess) schließen, das den Strom bedient.
  - Daten können einfach oder strukturiert sein (große Objekte, Werte, Formulare, Webseiten)
  - Datentypen, die über den Stromanschluss fließen, stammen aus dem Domänenmodell
- Strom-Ports werden durch ***Strom-Kanäle (channels, pipes)*** zu Strömen verbunden
  - Es entstehen Pipe-und-Filter-Architekturen (Datenflussarchitekturen)
- Entwurfs- und Implementierungsmodell: Portschnittstellenklasse wird mit Muster Iterator (Stream, Sink) realisiert; Kanal mit Muster Channel



© Prof. U. Aßmann

41.2 Architekturprinzipien

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

41.2.1 Architekturprinzip: Hohe Kohäsion + Niedrige Kopplung

Subsystem A

(z.B. Benutzungs-oberfläche)

Subsystem B

(z.B. fachlicher Kern)

- **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.

© Prof. U. Aßmann

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 oder Portklassen besitzt, um lose Kopplung zu unterstützen:
    - keine impliziten, nur explizit in der Schnittstelle angegebene Abhängigkeiten zu anderen Komponenten
- Vorteile der **Modularität**:
  - Unabhängkeit im Softwre-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

© Prof. U. Aßmann

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)

© Prof. U. Aßmann

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

3b) 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

© Prof. U. Aßmann

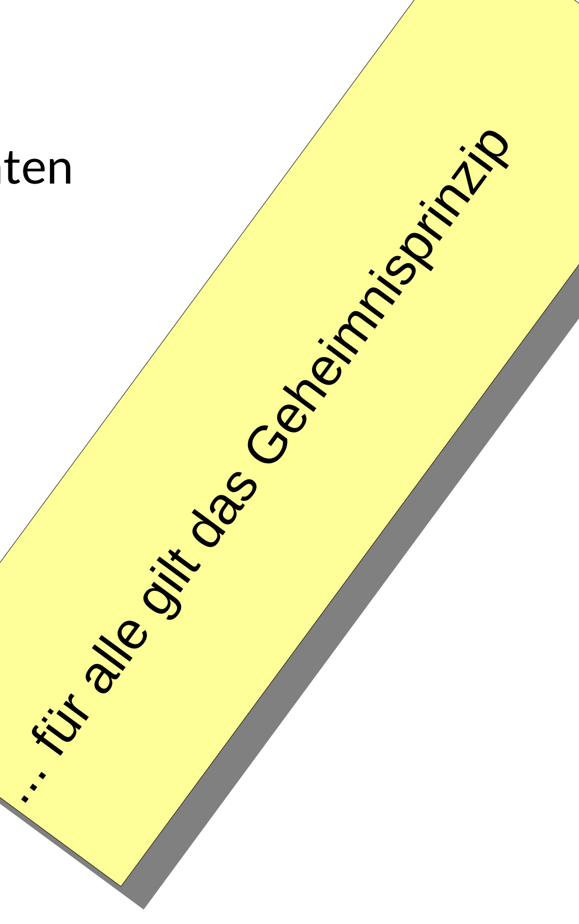
Typische Geheimnisse von Modulen/Komponenten

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

© Prof. U. Aßmann

Verschiedene Arten von Komponenten/Modulen (in verschiedenen Sprachen)

- 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



© Prof. U. Aßmann

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 (Wiederverwendungeinheiten) kapseln, indem man sie als Kollaborationen spezifiziert

© Prof. U. Aßmann

Architekturprinzip: Schichten von Variabilität

- **Software-Produktlinien** entstehen durch systematische Variation von Geheimnissen, wobei diese in *Schichten* bzw. ***Perspektiven*** gruppiert werden

Schicht	Beispiele
Funktionale Eigenschaften (Funktionale Varianten in der Control)	Mailer, Calendar, Browser, Writer, SlideEditor, Wiki
Middleware Varianten (Entity level)	Parallelität (Transaktionen), Verteilung
Plattformeigenschaften (Boundary, Database)	Chip, OS, GUI-Bibliothek, Persistenz, Displaygerät, Web/Rich/Smart Client
Dynamische Kontexteigenschaften	Benutzer (Level, Präferenzen), Netzwerkverbindung, Sicherheitseigenschaften

© Prof. U. Aßmann

41.3 Architekturprinzip Schichtung (Layered Architectural Style)

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

Vier-Schichten-Architekturstil (BCED) für interaktive Anwendungen

- 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

Benutzungsschnittstelle <<boundary>>

Basiert auf Analyse-modell

Fachlicher Kern

Kontextmodell

<<control>>

Middleware <<entity>>

Datenverwaltung <<data>>

© Prof. U. Aßmann

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 B auf, 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

© Prof. U. Aßmann

Stevens' Coupling Types (Ghezzi)

Content (tightest)	One module relies on the internal working of another. Chang- ing one module requires changes in the other as well.
Common	Two or more modules share some global state, e.g., a variable.
Control	One module controls the flow of another, e.g., passing infor- mation that determine how to execute.
External	Two or more modules share a common data format.
Stamp	Two or more modules share a common data format, but each of them uses a different part with no overlapping.
Data	Two or more modules share data through a typed interface, e.g., a function call.
Message (loosest)	Two or more modules share data through an untyped interface, e.g., via message passing.

W. Stevens et al. Classics in software engineering. chapter Structured Design. 1979.

© Prof. U. Aßmann

Geschichteter Test von hierarchischen und geschichteten Systemen

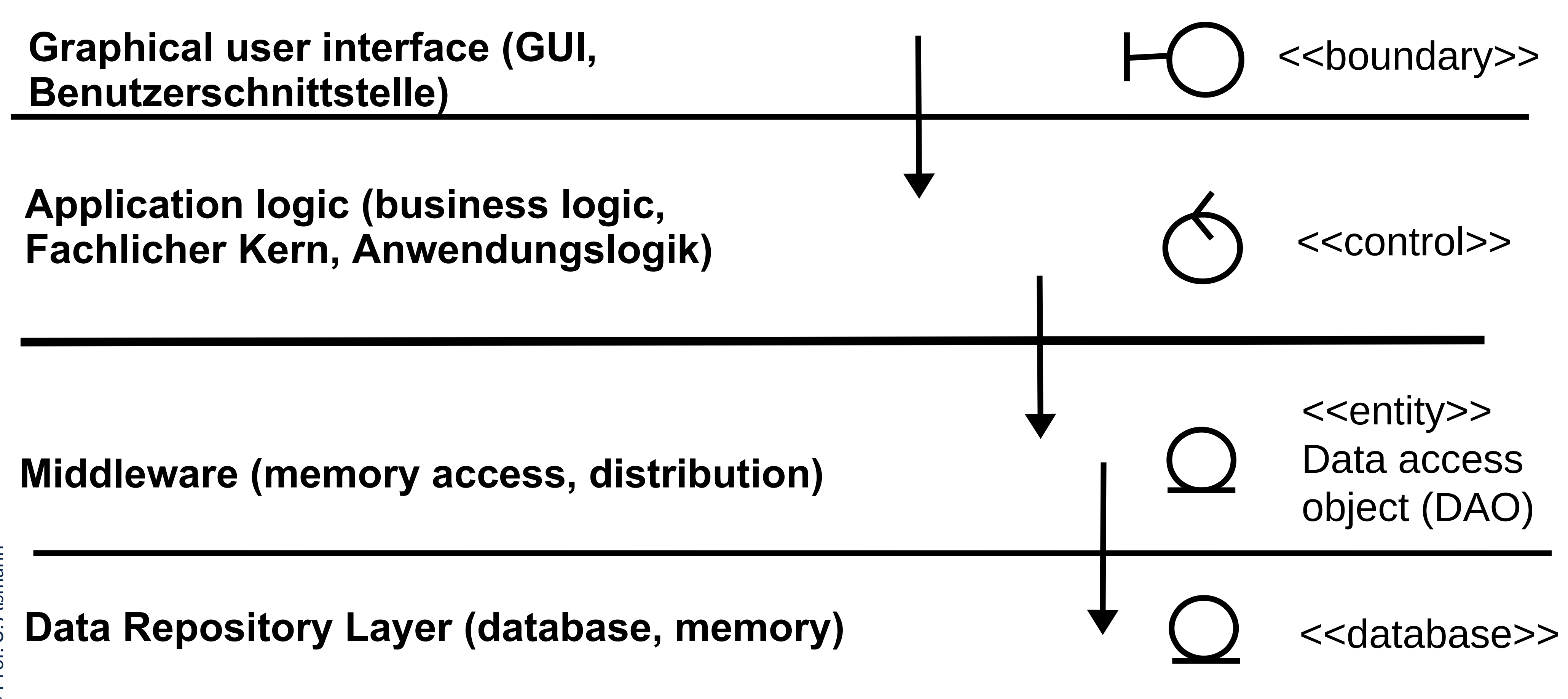
In einem hierarchischen oder geschichteten System erfolgt der Test bottom-up, d.h. aufwärts entlang der USES-Relation.

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

© Prof. U. Aßmann

Beispiel: USES-Relation in 4-Tier Architekturen (BCED)

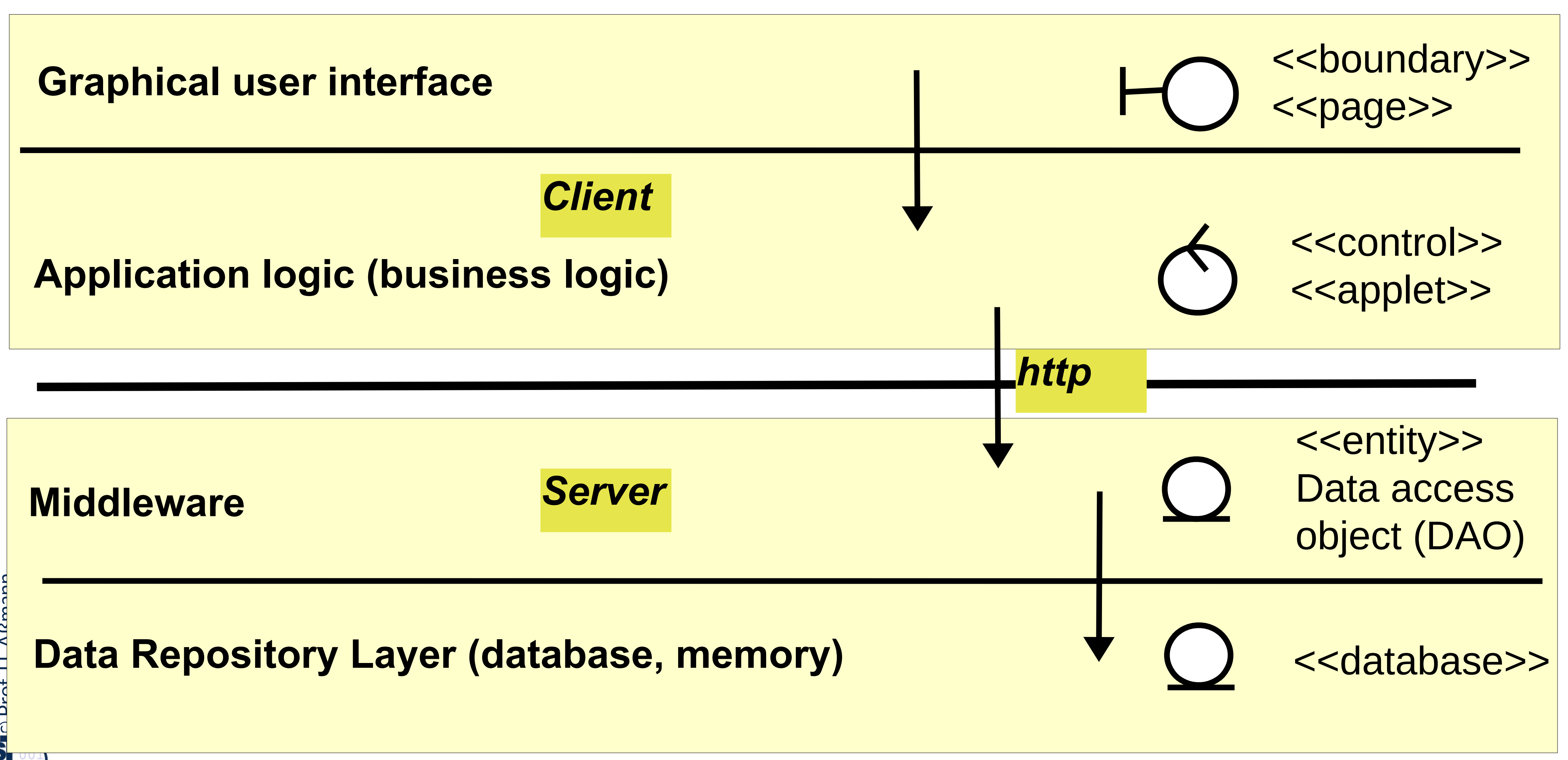
- 4-Schichtarchitekturen nutzen eine azyklische USES-Relation
  - Obere Schichten nutzen untere, aber nicht umgekehrt



© Prof. U. Aßmann

Beispiel: 4-Tier Web System (Thick Client)

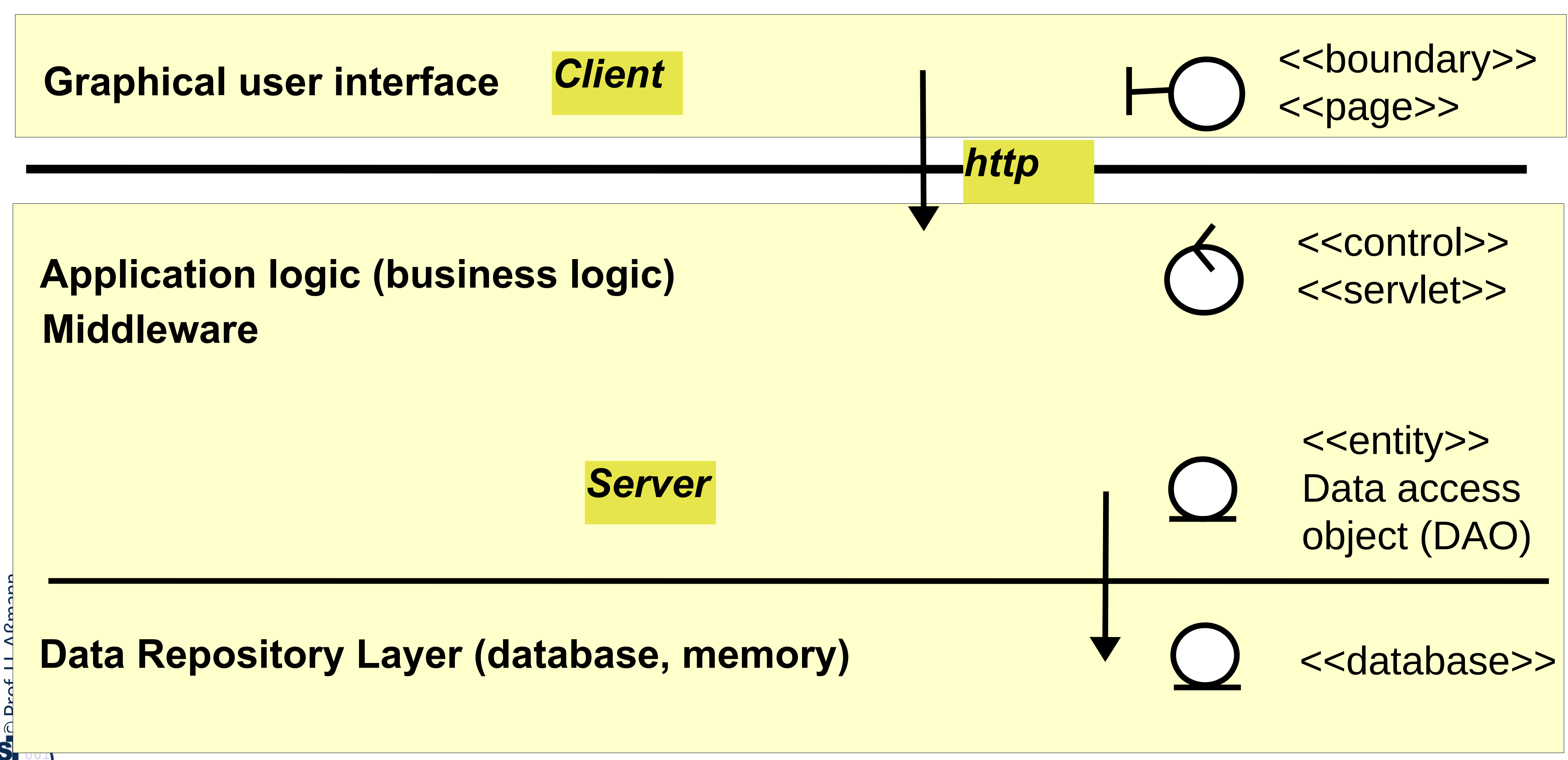
- “Thick client” Web-Systeme nutzen eine http-basierte Middleware
- GUI und AL verbleiben auf dem Client, die Daten werden auf dem Server verwaltet



© Prof. U. Aßmann

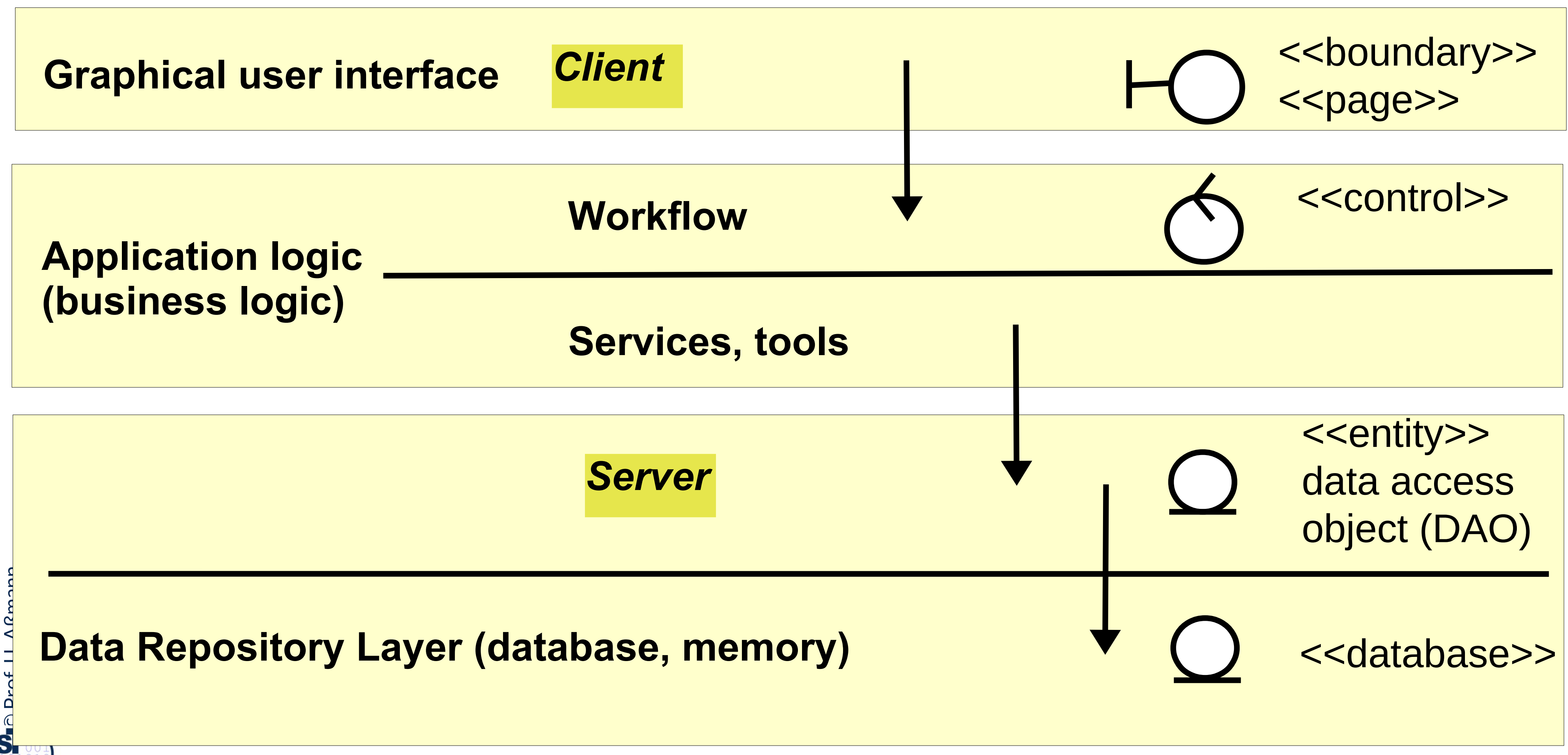
Beispiel: 4-Tier Web System (Thin Client)

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



Beispiel: 5-Tier mit Workflow Language

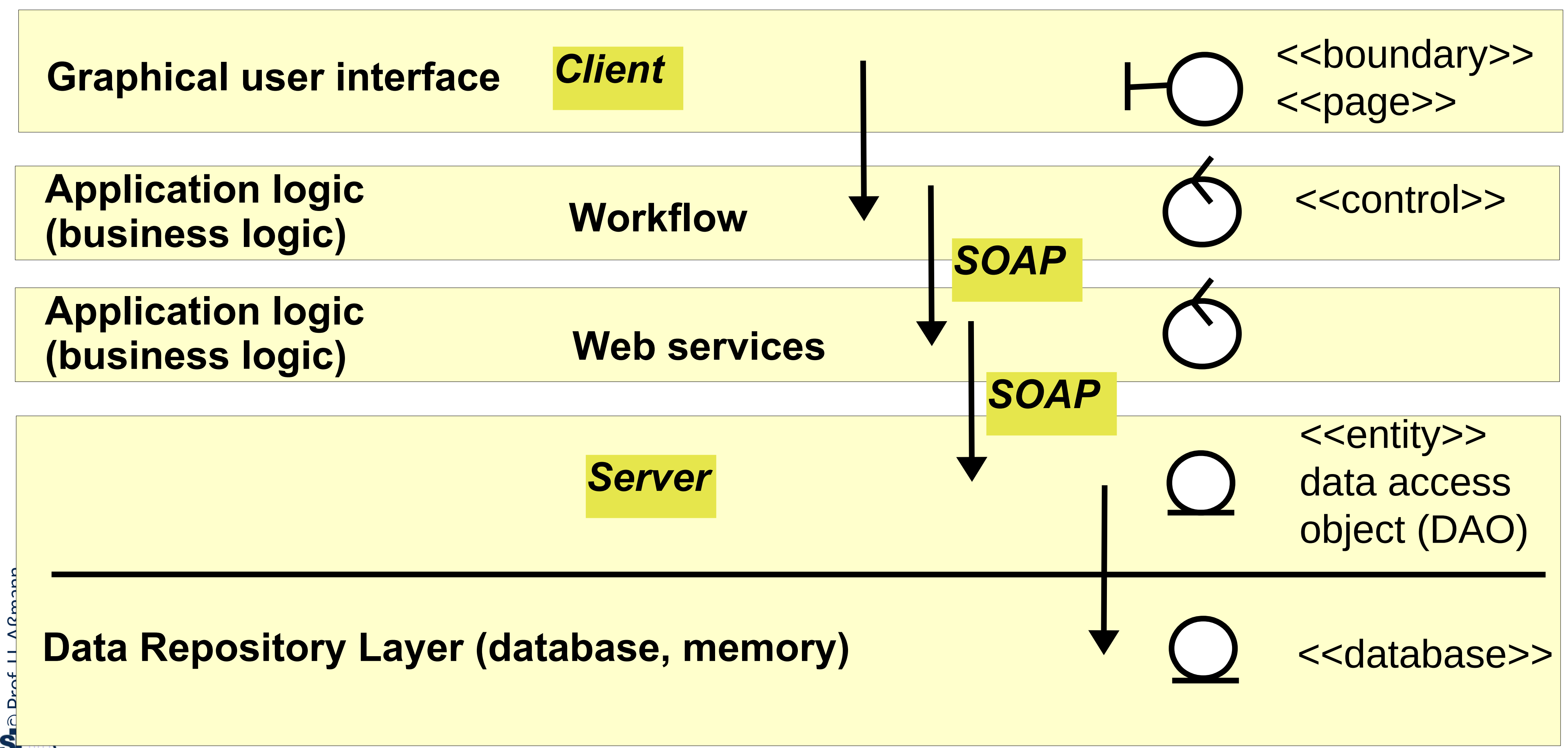
- Arbeitsfluss-Sprachen (Workflow languages) wie BPMN, BPEL definieren die Arbeitsfluss-Schicht der AL, unterhalb der Top-Level-Architektur
  - Services und Tools arbeiten auf den Daten



© Prof. U. Aßmann

Beispiel: 5-Tier mit Workflow Language und Web Services

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



© Prof. U. Aßmann

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

- Jede Schicht enthält ein Stromobjekt für bidirektionale Kanalkommunikation

Presentation Layer
Session Layer
..
..
..
..
Data Transport Layer
..

Presentation Layer
Session Layer
..
..
..
..
Data Transport Layer
..

© Prof. U. Aßmann

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)

© Prof. U. Aßmann

Beispiel: Datenbanksysteme

SQL compiler

transaction manager

lock manager

table manager

physical storage management

record management layer

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 start, Kopplung gering
  - Veränderungsorientierter Entwurf, Austausch von Schichten möglich ⍰ Evolution einfach
  - Verantwortlichkeiten für Testmanagement können klar definiert werden: Für jede Schicht werden separate Testsuites entwickelt (Bottum-up tests)

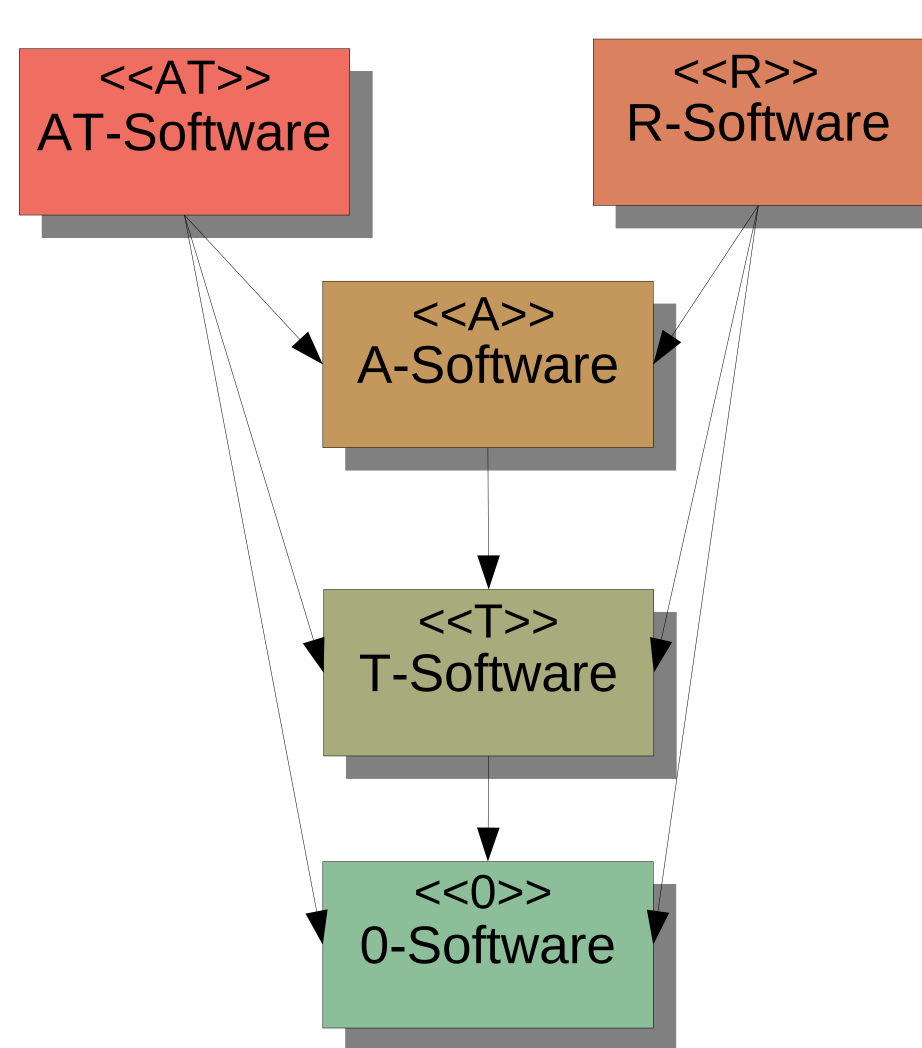
© Prof. U. Aßmann

41.4 Architekturprinzip Trennung von Anwendung undTechnik

Perspektivenmodell Quasar: Trennung von Technik- und Anwendungskomponenten (Reuse Blood Groups, Blutgruppen)

Quasar ist ein Perspektivenmodell, das 4 Aspekte definiert (**Wiederverwendungs-Blutgruppen)**, Softwarekategorien für Komponenten, nach Wiederverwendbarkeit:

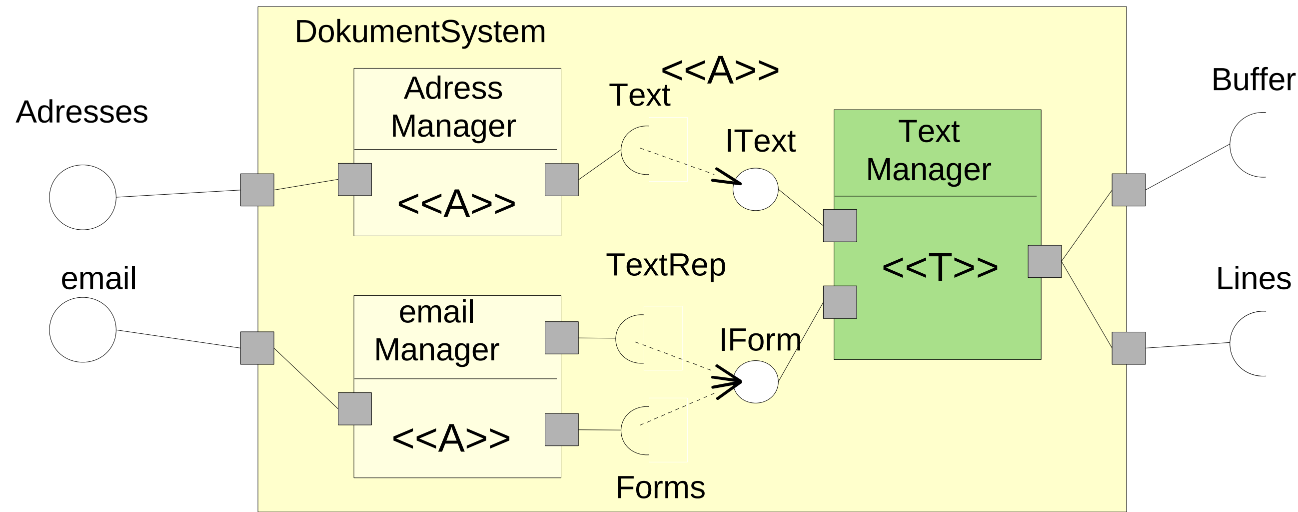
- 0: unabhängig von Anwendung und Technologie
  - JDK collections, C++ STL, GNU regexp
- A: anwendungs- oder domänenspezifisch.
  - Client, Customer, Account, Car, ...
- T: technologie-orientierte Schnittstelle, unabhängig von Anwendung
  - OSGI, JDBC, CORBA CosNaming
- AT: abhängig von Anwendung *und*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



© Prof. U. Aßmann

Perspektivenmodell Quasar: Trennung von Technik- und Anwendungskomponenten

- 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.**

© Prof. U. Aßmann

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
- Schichtenbasierte Architekturen sind wichtig
  - Azyklische USES (relies-on) Relation

© Prof. U. Aßmann

Vorsicht: Conway's Law über Software-Strukturen

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

© Prof. U. Aßmann

The End

- 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 Kruchtens 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 9-Schichtenmodell.

Anhang 41.A.1 Entwurfsmuster Fassade zur Reduktion von Kopplung

... Ein Entwurfsmuster im Geiste Parnas

(Wdh.)

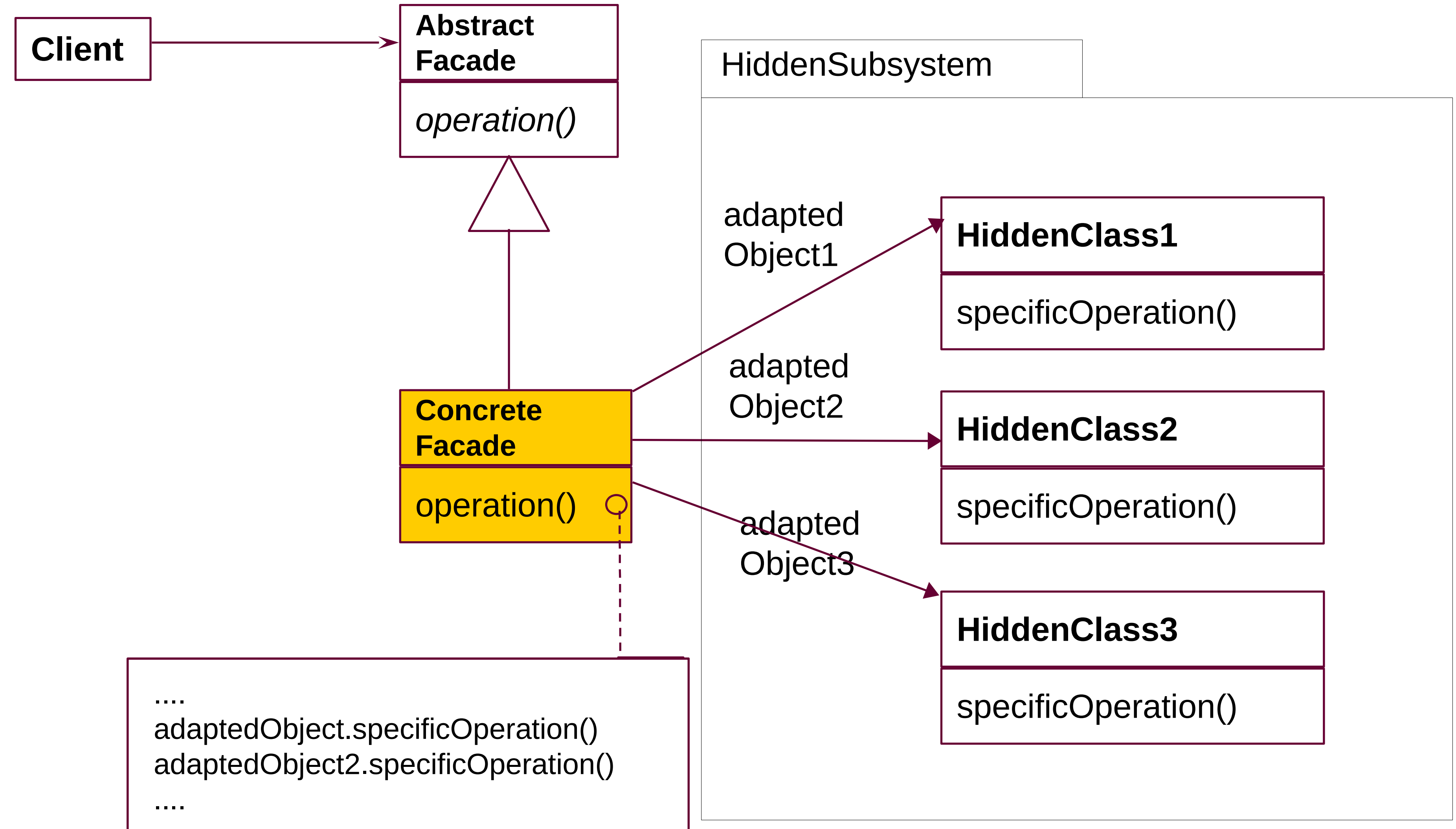
Entwurfsmuster Fassade (Facade)

- 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

© Prof. U. Aßmann

Fassaden verbergen Subsysteme

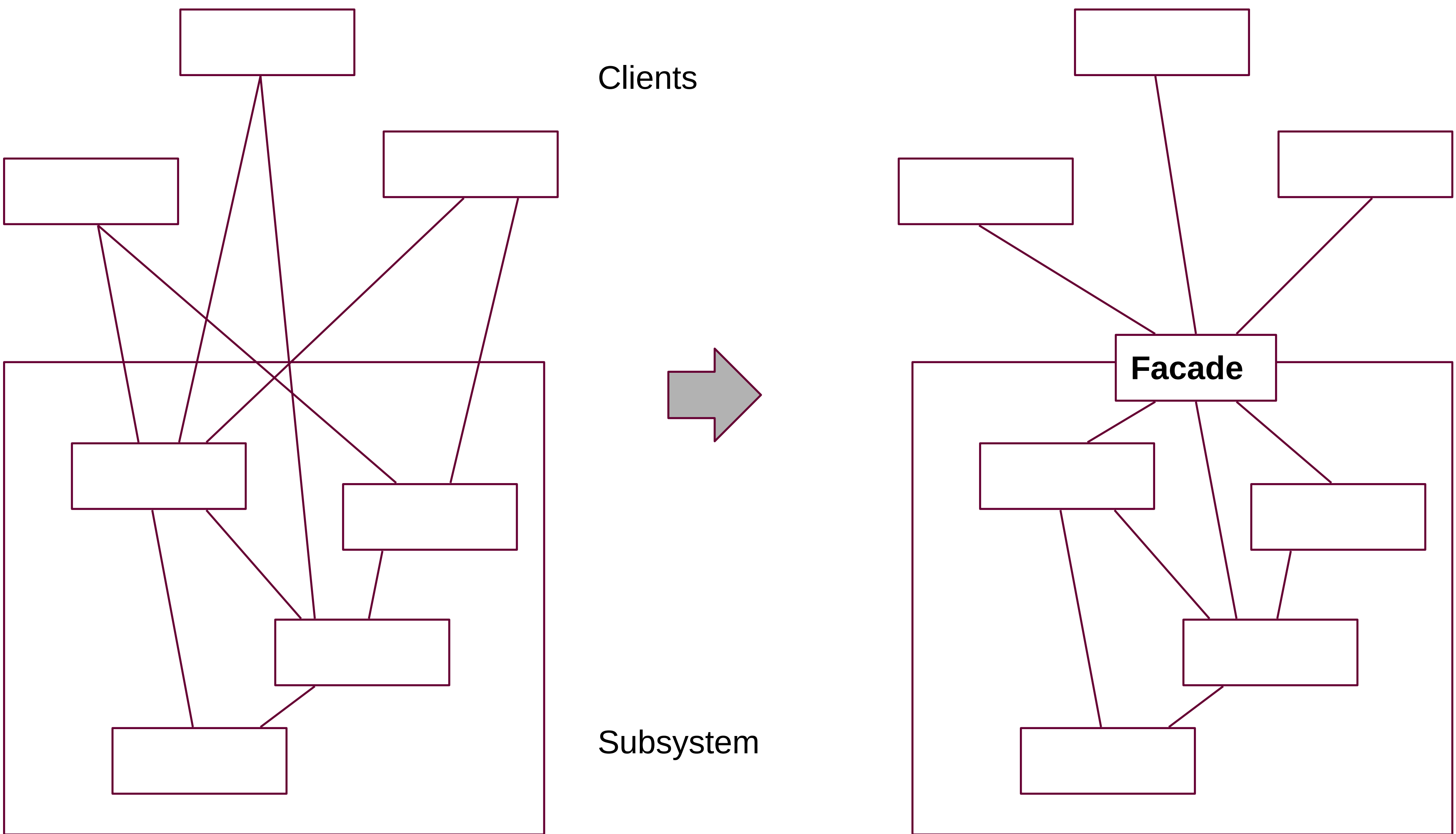
- Eine Fassade bietet eine *Sicht* auf ein Subsystem an. Es darf mehrere Sichten geben, nur keinen direkten Zugriff auf die inneren Objekte



© Prof. U. Aßmann

Restrukturierung hin zur Fassade

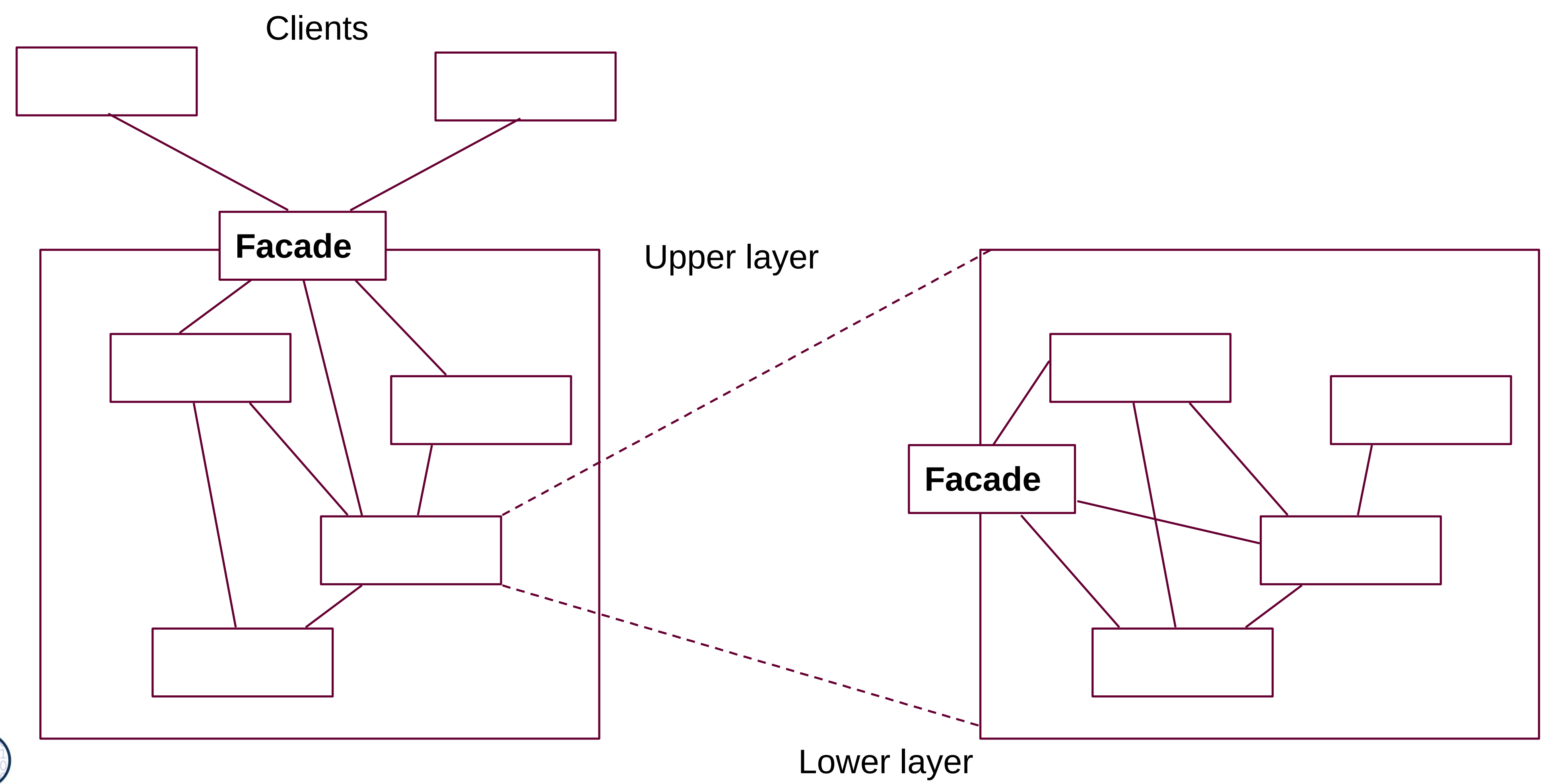
- Fassaden entkoppeln; Subsysteme können leichter ausgetauscht werden (Variabilitätsmuster)



© Prof. U. Aßmann

Fassaden und Schichten

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



© Prof. U. Aßmann

41.A.2 Entwurfsmuster Fabrikmethode (FactoryMethod)

(Wdh.)

zur polymorphen Variation von Komponenten (Produkte)

und zum Verbergen von Produkt-Arten

Problem der Fabrikmethode

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

```
...
Product = new ConcreteProduct1()
...
```

Product

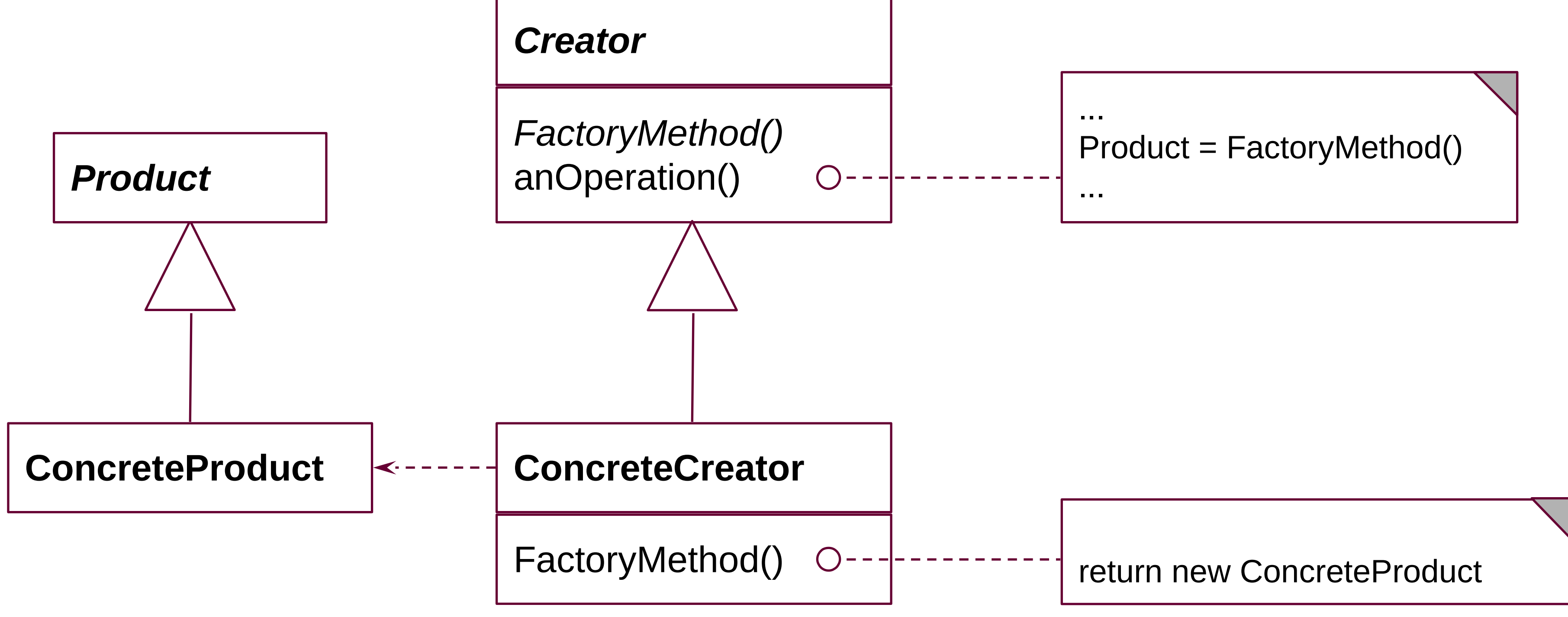
ConcreteProduct1

ConcreteProduct2

© Prof. U. Aßmann

Struktur Fabrikmethode

- FactoryMethod ist eine Variante von TemplateMethod, zur Produkterzeugung



© Prof. U. Aßmann

Factory Method (Polymorphic Constructor)

- 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);
}
```

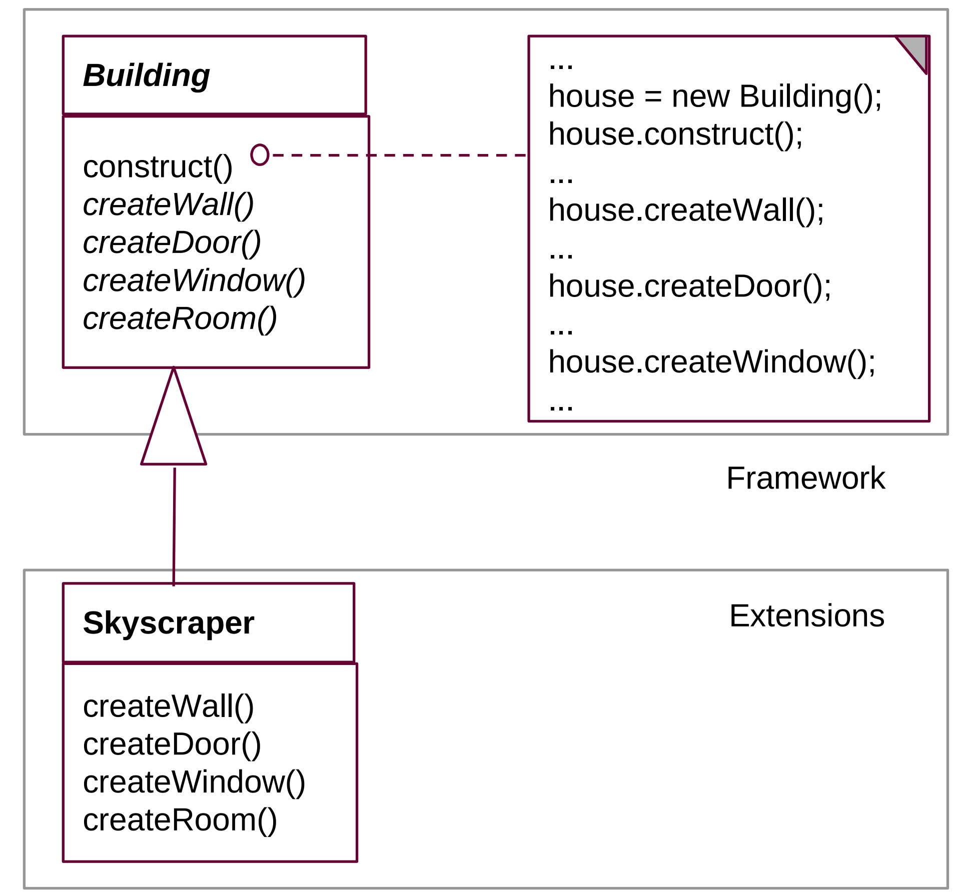
```
public class Client {
... Creator cr = [.. subclass ]..
  public void collect() {
    Set mySet = Creator.createSet(10);
    ....
  }
}
```

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

© Prof. U. Aßmann

Beispiel FactoryMethod

- 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?



© Prof. U. Aßmann

Lösung mit FactoryMethod

- Bilde createBuilding() als Fabrikmethode aus

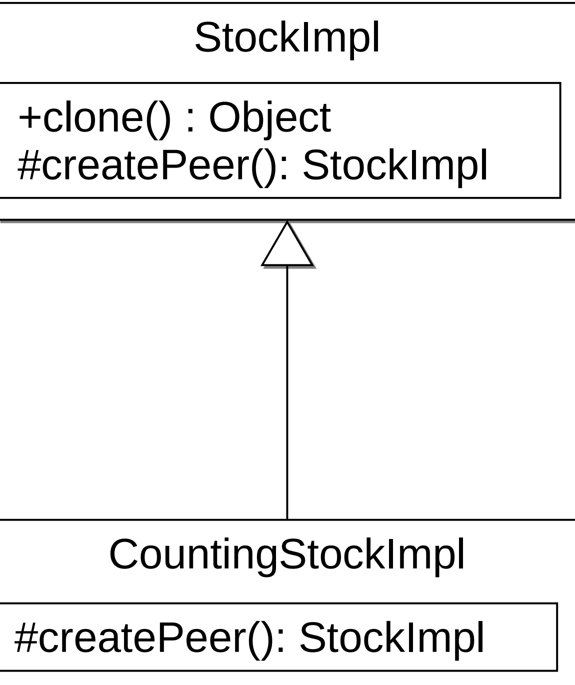
```
// abstract creator class
public abstract class Building  {
   public abstract
        Building createBuilding();
   ...
}
```

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

© Prof. U. Aßmann

Factory Method im SalesPoint-Rahmenwerk

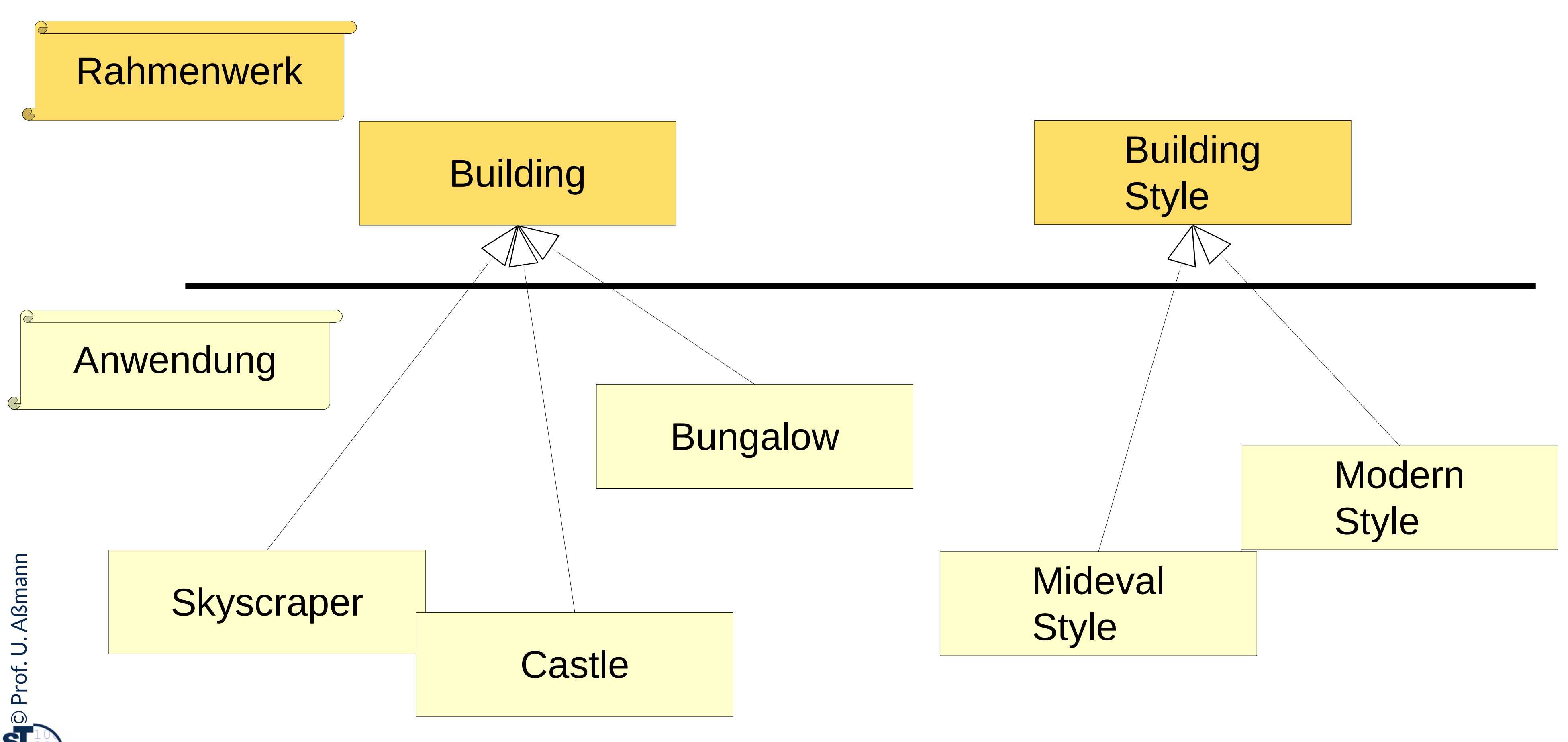
- Anwender von SalesPoint verfeinern die StockImpl-Klasse, die ein Produkt des Warenhauses im Lager repräsentiert
  - z.B. mit einem CountingStockImpl, der weiß, wieviele Produkte noch da sind



© Prof. U. Aßmann

Einsatz in Komponentenarchitekturen

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



© Prof. U. Aßmann

41.A.3
Strategie (Strategy, Template Class)

(Wdh.)

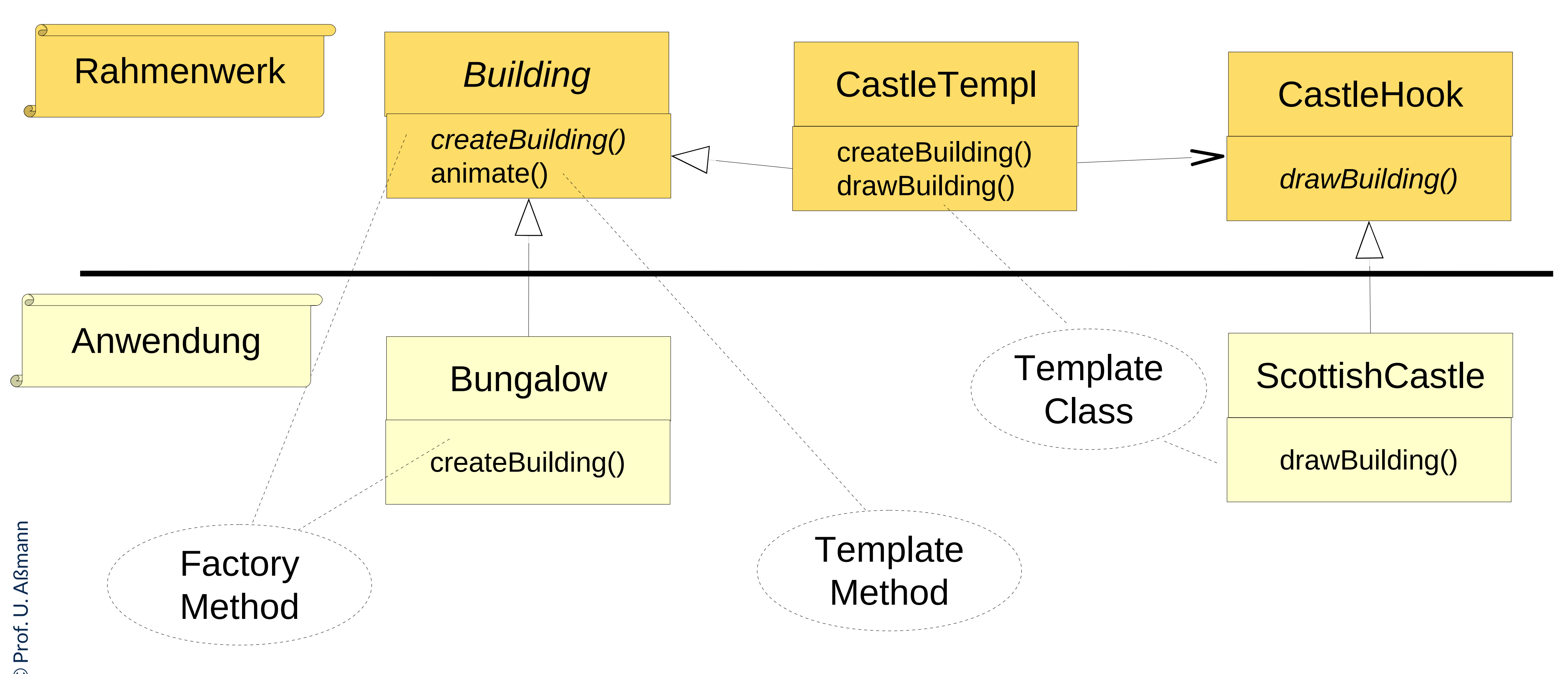
Strategy (also called Template Class)

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

© Prof. U. Aßmann

Kombinierter Einsatz in Rahmenwerken

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



© Prof. U. Aßmann

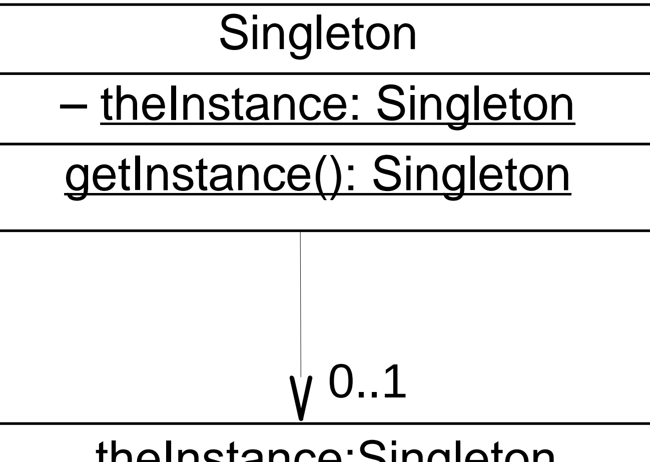
41.A.4 Entwurfsmuster Einzelstück (Singleton)

zur globalen Konfiguration einer Komponente oder Schicht

(Wdh)

Entwurfsmuster Einzelstück (Singleton)

- 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;
  }
}
```

© Prof. U. Aßmann

Singleton im SalesPoint – the Shop

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

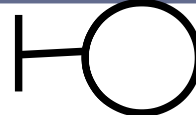
Shop
#Shop() $getTheShop(): Shop $setTheShop (Shop sh)

© Prof. U. Aßmann

Repet.: BCD/BCED Classification

- Boundary classes:

<<boundary>>



  - Represent an interface item that talks with the user
  - May persist beyond a run

<<control>>

- Control class:

  - Controls the execution of a process, workflow, or business rules
  - Does not persist

- Entity class:

  - Describes persistent knowledge. Caches a persistent object from a database (data access object, DAO)

- Database class

  - Adapter class for the database
  - Often, Entity and Database classes are unified

- **BCD/BCED is linked with the 3-tier architecture**

© Prof. U. Aßmann