Teil IV: Objektorientierter Entwurf
41 Grundlegende Architekturprinzipien

1

Prof. Dr. rer. nat. Uwe Aßmann

Institut für Software- und Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

TU Dresden

Version 15-0.1, 11.07.15

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

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

Exkurs: Wie findet man ein Papier?

- http://scholar.google.de
- http://liinwww.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: Architektur interaktiver Systeme
4) 43: Punktweise Verfeinerung von Lebenszyklen
   - Verfeinerung von verschiedenen Steuerungsmaschinen
5) 44: Verfeinerung mit querschneidender Objektanreicherung

41.1 Grundlegende Architekturprinzipien

41.2.1 Aspekttrennung mit Perspektivenmodellen

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

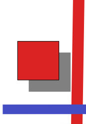
Ein **Perspektivenmodell (view model)** definiert eine Menge von Aspekten, von denen die Software aufgeteilt wird

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

Wesentliche Aspekte und Bestandteile eines Softwaresystems

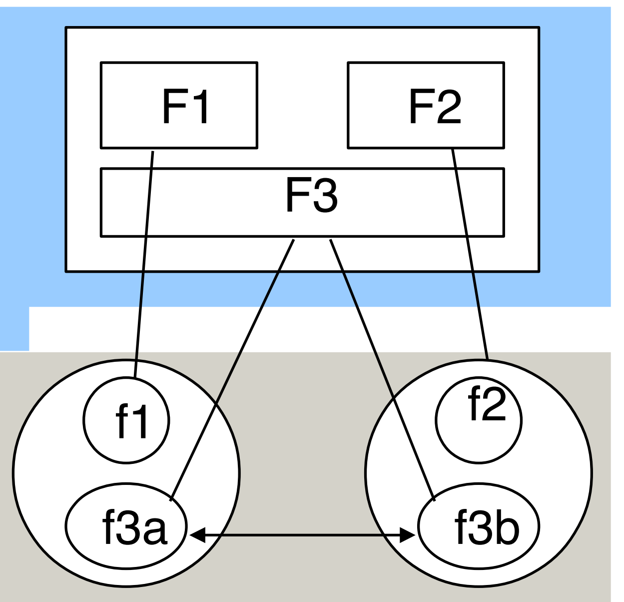
- Anwendungsspezifische Funktionen
- Benutzungsoberfläche
- Ablaufsteuerung
- Datenhaltung
- Infrastrukturdienste
  - Objektverwaltung
  - Interne Objekt- und Prozeßkommunikation
  - Verteilungsunterstützung
- Kommunikationsdienste
- Sicherheitsfunktionen
- Zuverlässigkeitsfunktionen
- Systemadministration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

- Ein **Aspekt** definiert eine Sicht auf ein System
- 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



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

- Folgende Aspekte werden in Perspektivenmodellen unterschieden:



- Struktursicht 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

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** that crosscut all other views

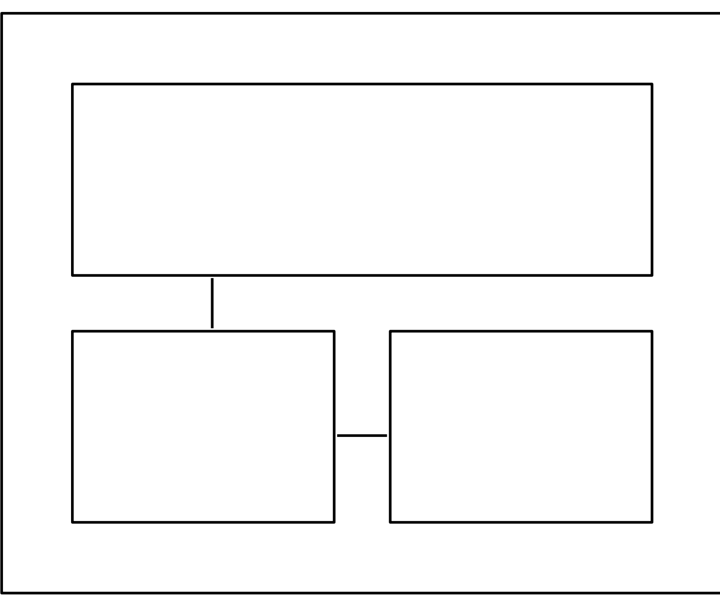
Hofmeister/Soni/Nord:

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

Blockdiagramme zur logischen Struktur

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



Konfigurationsdiagramme für physikalische Verteilung

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

Rechner, Knoten

Lokales Kommunikationsnetz

Speicherndes System

Datenkommunikations-Netz

Blockdiagramm → Konfigurations-Diagramm → Montagediagramme und Top-Level-Architektur in UML-Komponenten

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

Logische Struktur nicht-Interaktiver Anwendungen: Montagediagramme für die obersten Ebenen des Systems

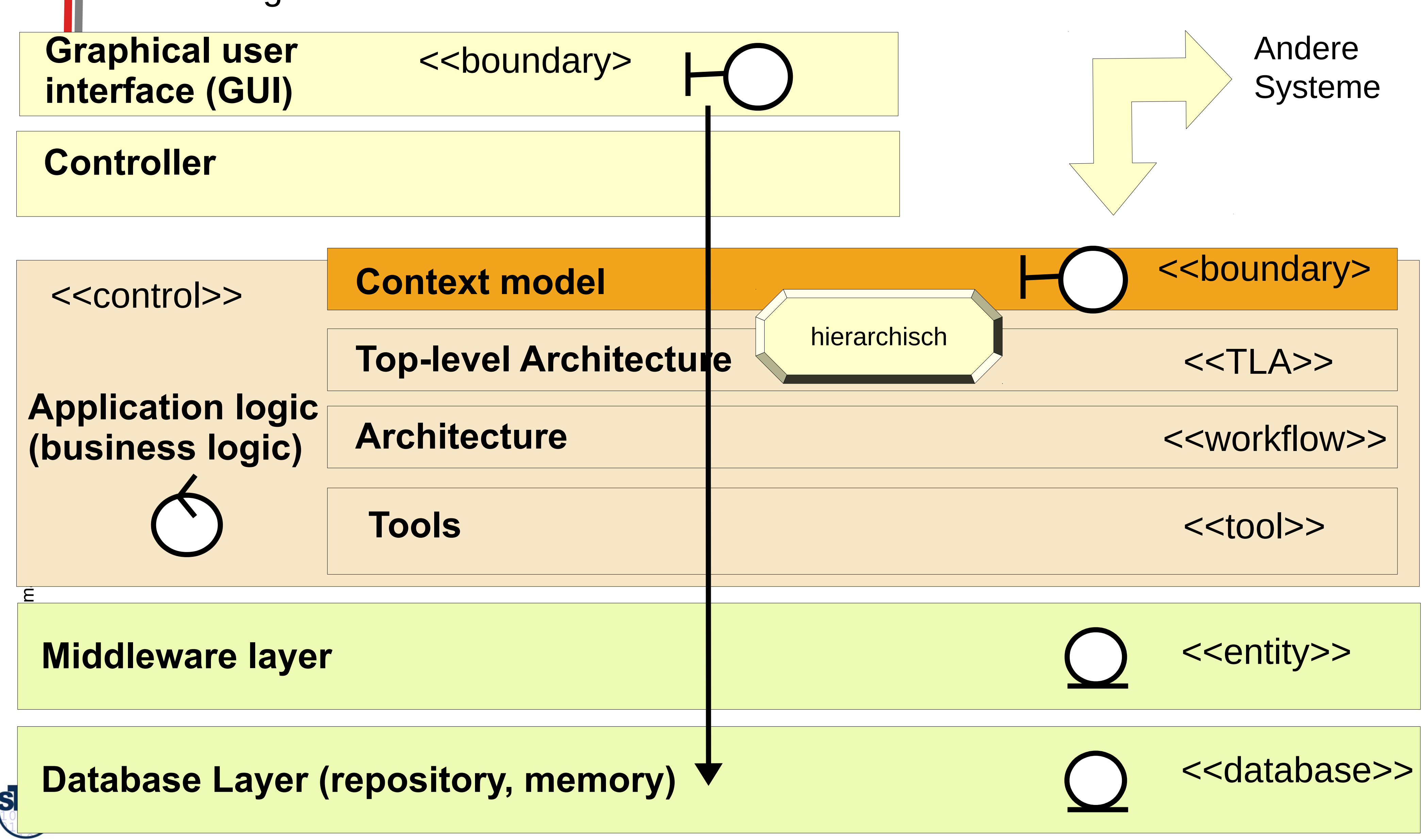
- Oberste Ebene des Systems ist meist hierarchisch und/oder geschichtet organisiert
  - Vermeide “wilde” objekt-orientierte Netzstrukturen
  - Damit die letzte Integration zum Gesamtsystem einfach verläuft: Integrationstests können dann bottom-up absolviert werden
- Hierarchien bilden Spezialfälle, denn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

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

- 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

Struktursicht: Weitere verfeinerte BCED-Schichtung eines Systems

- Im Folgenden verwenden wir 3-7 Schichten:



41.2 Architekturprinzipien

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

4.2.1 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.

4.2.2 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

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)

Architekturprinzip: Flexible Evolution mit dem Geheimnisprinzip

Parnas' Prinzip des Entwurfs mit dem **Geheimnisprinzip** (veränderungsorientierter Entwurf, *change-oriented modularization with information hiding)* [Parnas, CACM 1972]:

1) Bestimme alle **Entwurfsfragen** (-alternativen), die sich *ändern können*

2) Entwickle für jede Entwurfsfrage eine Komponente, die die Entscheidung bezüglich der Frage verbirgt

- Die Entscheidung nennt man das **Komponenten**- oder **Modulgeheimnis (module secret)**

3) Entwerfe eine **stabile Schnittstelle** für die Komponente, die unverändert bleibt, wenn sich die Entwurfsentscheidung und somit die Implementierung des Modulgeheimnisses ändert

3b) Definiere *angebotene* und *benötigte* Schnittstellen

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

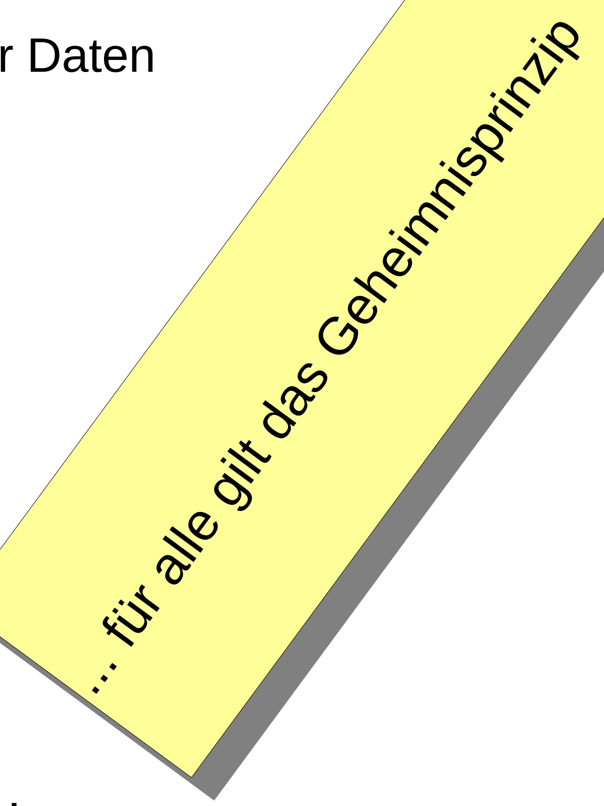
Das Geheimnisprinzip erniedrigt die externe Kopplung und erhöht die innere Kohäsion von Komponenten und Modulen

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

- 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



Variabilitätsmuster 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

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

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

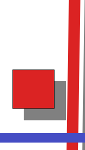
Komponente A **vertraut auf (relies-on,** USES) Komponente B gdw.
A benötigt eine korrekte Implementierung von B für seine eigene korrekte Ausführung [Parnas-Hierarchie]

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

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

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.



Beispiel: USES-Relation in 3- and 4-Tier Architekturen (BCED)

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

Graphical user interface (GUI, Benutzerschnittstelle) — <<boundary>>

Application logic (business logic, Fachlicher Kern, Anwendungslogik) — <<control>>

Middleware (memory access, distribution) — <<entity>> Data access object (DAO)

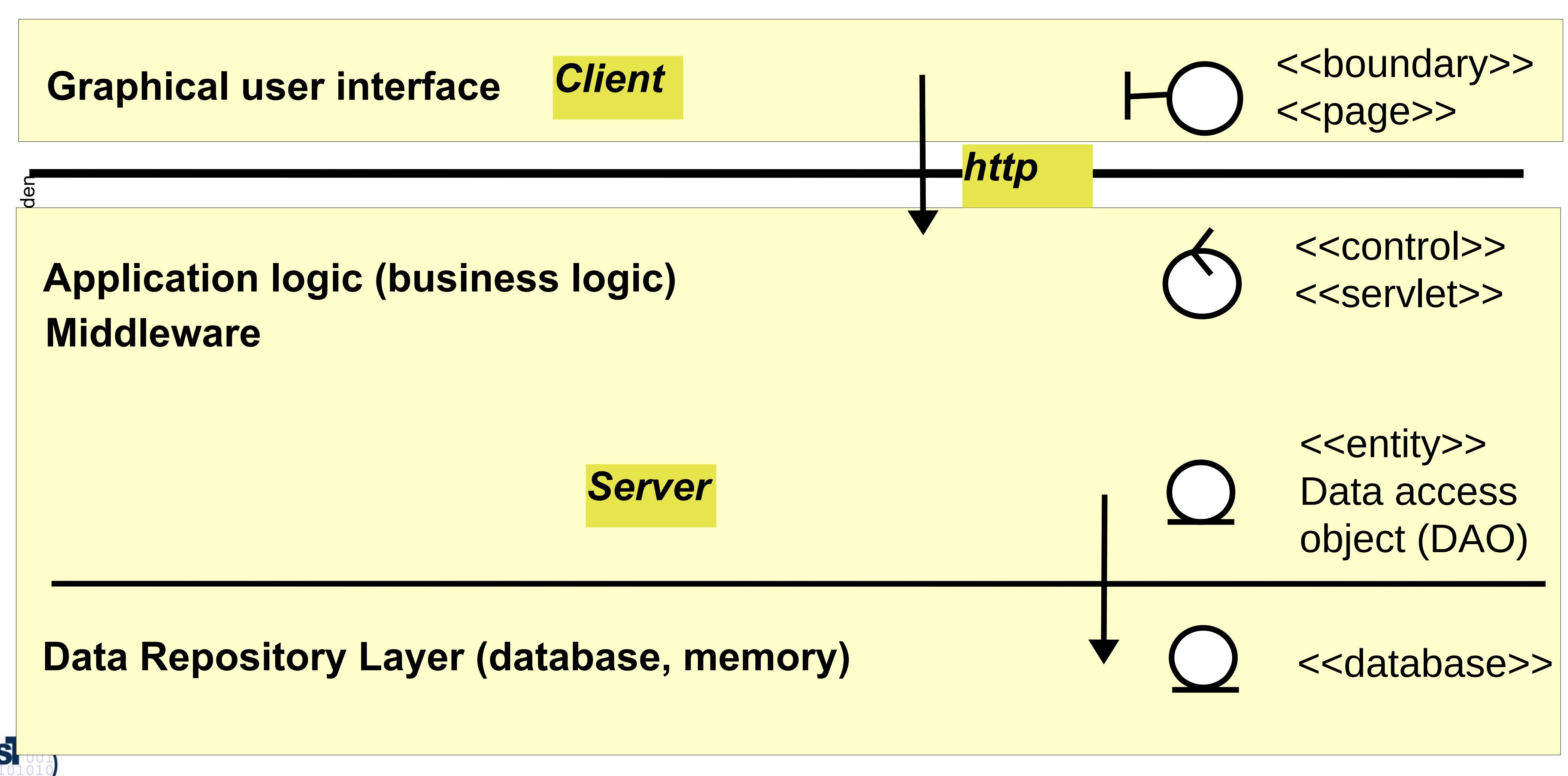
Data Repository Layer (database, memory) — <<database>>

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

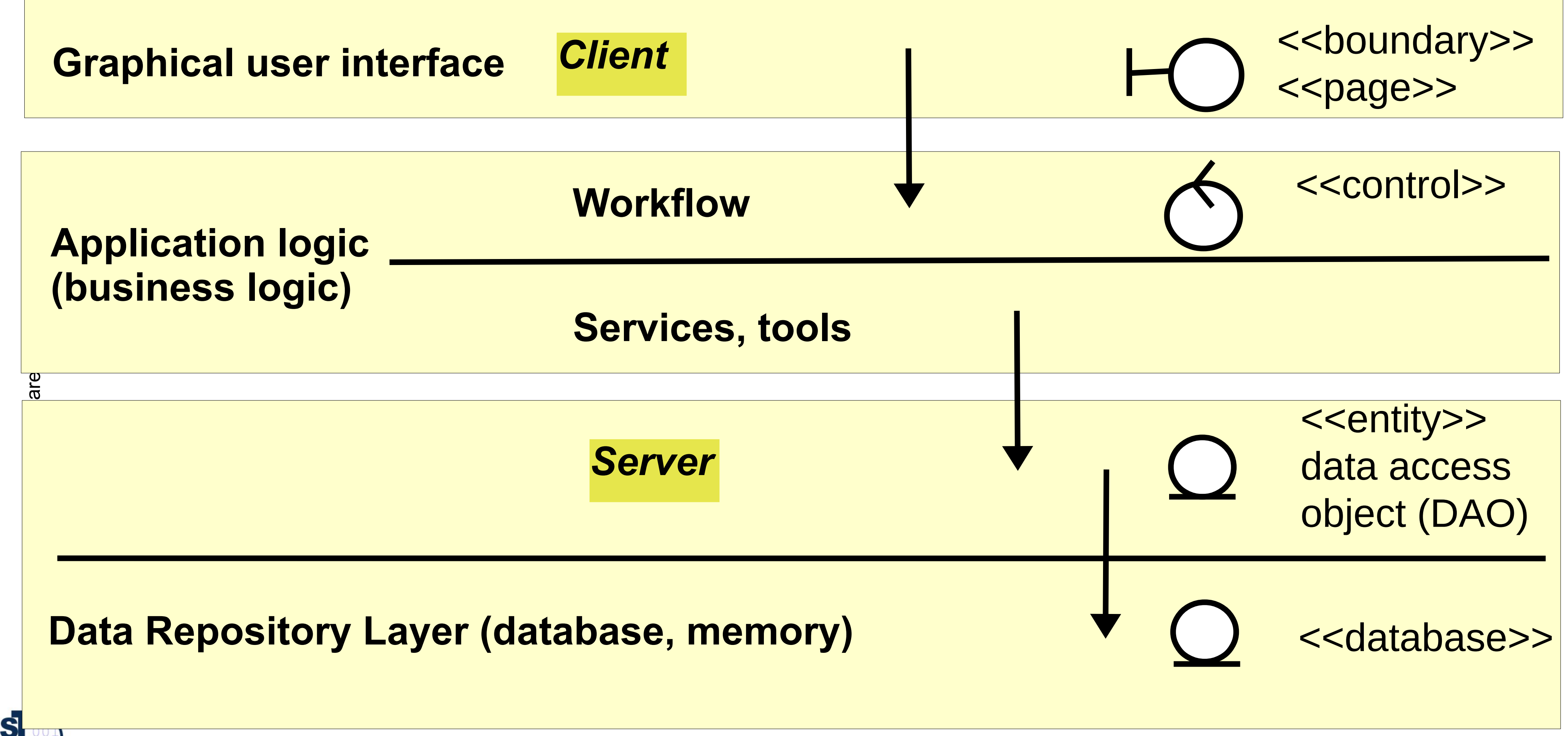
Beispiel: 4-Tier Web System (Thin Client)

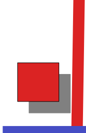
- "Thin client" Web-Systeme verwalten außer dem GUI alles auf dem Server



Beispiel: 5-Tier with 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 with Workflow Language and Web Services

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

Graphical user interface — ***Client*** — <<boundary>> <<page>>

Application logic (business logic) — **Workflow** — <<control>>

SOAP

Application logic (business logic) — **Web services**

SOAP

Server — <<entity>> data access object (DAO)

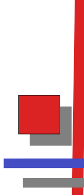
Data Repository Layer (database, memory) — <<database>>

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)

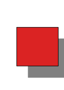
Was haben wir gelernt?

- 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



Vorsicht: Conway's Law über Software-Strukturen

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



The End

Example: ISO-OSI 7 Layers Network Architecture

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

Example: Operating Systems

UNIX,
Linux,
Android

User Space

Kernel

Apple-
UNIX,
iOS

User Space

Kernel

Microkernel (Mach)

Windows NT/XP:

User Space

Kernel

Hardware Abstraction Layer (HAL)

Example: Database Systems

SQL compiler

transaction manager

lock manager

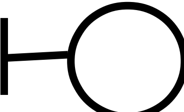
table manager

physical storage management

record management layer

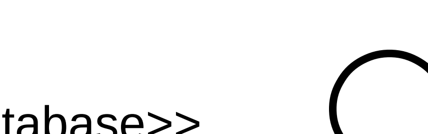
Repet.: BCD/BCED Classification

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



- 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**