Einführung in Software Engineering

Barbara Paech, Marcus Seiler

Institute of Computer Science
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
http://se.ifi.uni-heidelberg.de
paech@informatik.uni-heidelberg.de

Klausurregelungen ISW

- Pro Vorlesung ISW gibt es 1 Prüfungsversuch
  - Prüfungsversuch ist Klausur am Ende der Vorlesungszeit
  - Nächster Prüfungsversuch erst in 1 Jahr (nochmalige Teilnahme an ISW sinnvoll)
  - Bei Krankheit *zeitnah* ein Attest bringen. Weiteres Vorgehen wird dann besprochen (typischerweise zeitnahe schriftliche Prüfung)

Klausurorganisation

- Termin Donnerstag,11.02.16, 09.00-11.00 (Zeit: 90 Minuten)
- Raum INF 308 HS 1
- Voraussetzungen
  - Klausurzulassung, Bekanntgabe in Moodle spätestens am 02.02.2016
  - Verbindliche Anmeldung
  - Ausweis mit Lichtbild und Studierendenausweis mitbringen
- Verbindliche Anmeldung bis 05.02.2016 in MÜSLI, d.h.
  - Wer keine Klausurzulassung besitzt, kann die Leistungspunkte nicht erwerben.
  - Wer an der Klausur teilnehmen will, muss sich anmelden.
  - Wer noch nicht sicher weiß, ob sie/er die Klausurzulassung erreicht, sollte sich trotzdem anmelden. Wir lehnen das bei fehlender Zulassung ab.
- Wer angemeldet und zugelassen ist und ohne Entschuldigung nicht erscheint, ist automatisch durchgefallen.
- Aushang der Noten vor dem Sekretariat Informatik (Gebäude 348, Raum 021)

Anfängerpraktikum

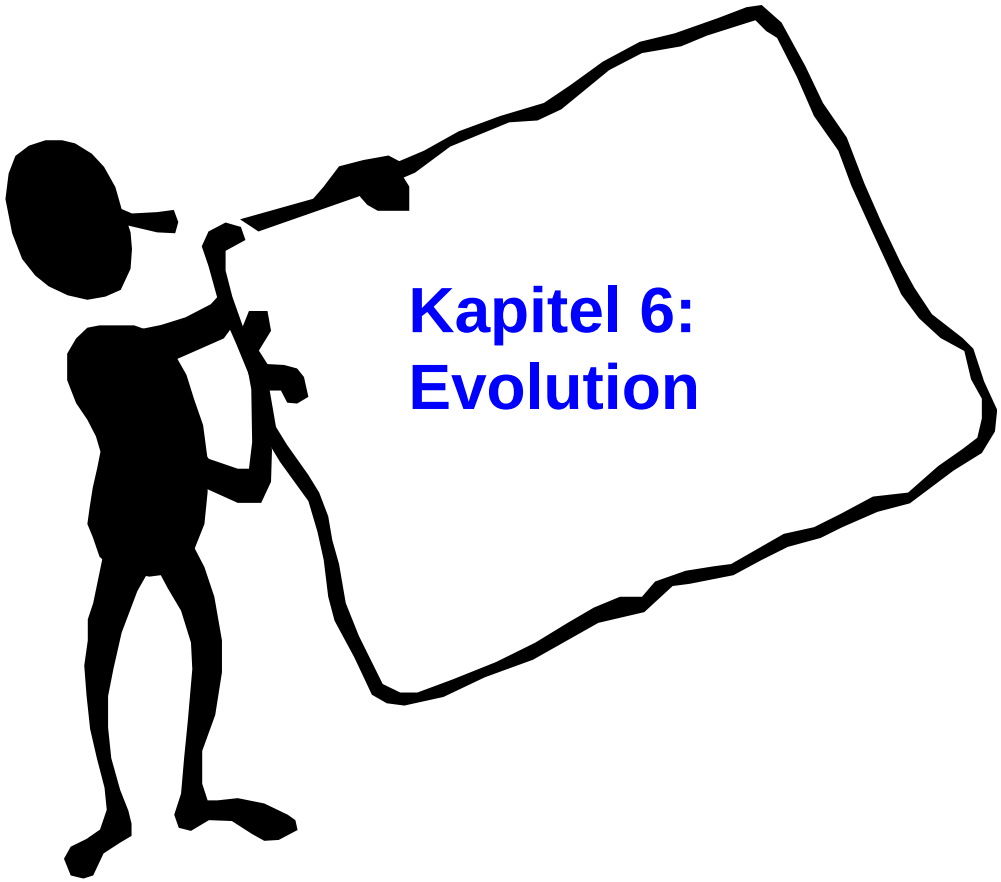
- Kenntnisse aus ISW können durch Anfängerpraktikum vertieft werden
- Entwicklung einer neuen Funktionalität in UNICASE in 3 Wochen in einem Team mit agilem Vorgehen SCRUM

- Findet als Block statt
  - Beginn MO 15.02.16
  - Ende DO 03.03.16

- Vorbesprechung HEUTE um 15:45 Uhr im Raum U012, INF350 (OMZ).
- Anmeldung spätestens bis ***05.02.16

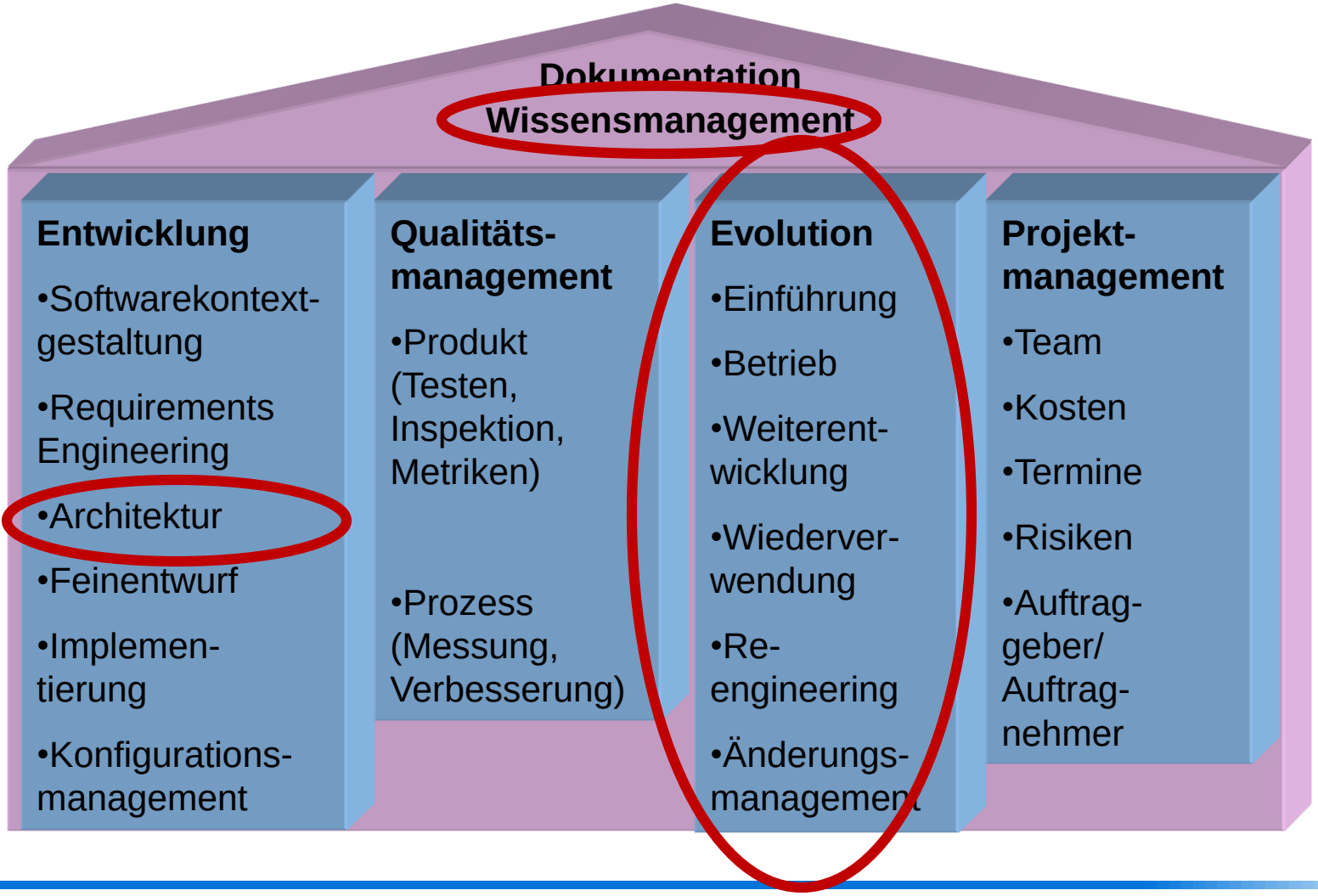
Kapitel Vorlesung

1. Einführung SWE
2. SWE im Kleinen (insbes. QS)
3. SWE im Großen (insbes. Anforderungen)
4. Kommunikation der EntwicklerInnen
5. Ergänzungen zu Anforderungen
6. Evolution (insbes. Wiederverwendung und Architektur)
7. SWE-Prozess (insbes. Zusammenfassung und Projektmanagement)



Kapitel 6: Evolution

Aufgabenbereiche des Software Engineering



6. Evolution

Evolution bezeichnet hier alle Aktivitäten, die eine Weiterentwicklung und Wiederverwendung unterstützen, bzw. die nach der Erstentwicklung stattfinden

- 6.1. Einführung
- 6.2. Architektur
- 6.3. Wiederverwendung
- 6.4. Weiterentwicklung und Änderungsmanagement
- 6.5. Betrieb und IT-Governance
- 6.6. Re-Engineering

6.1. Einführung Evolution

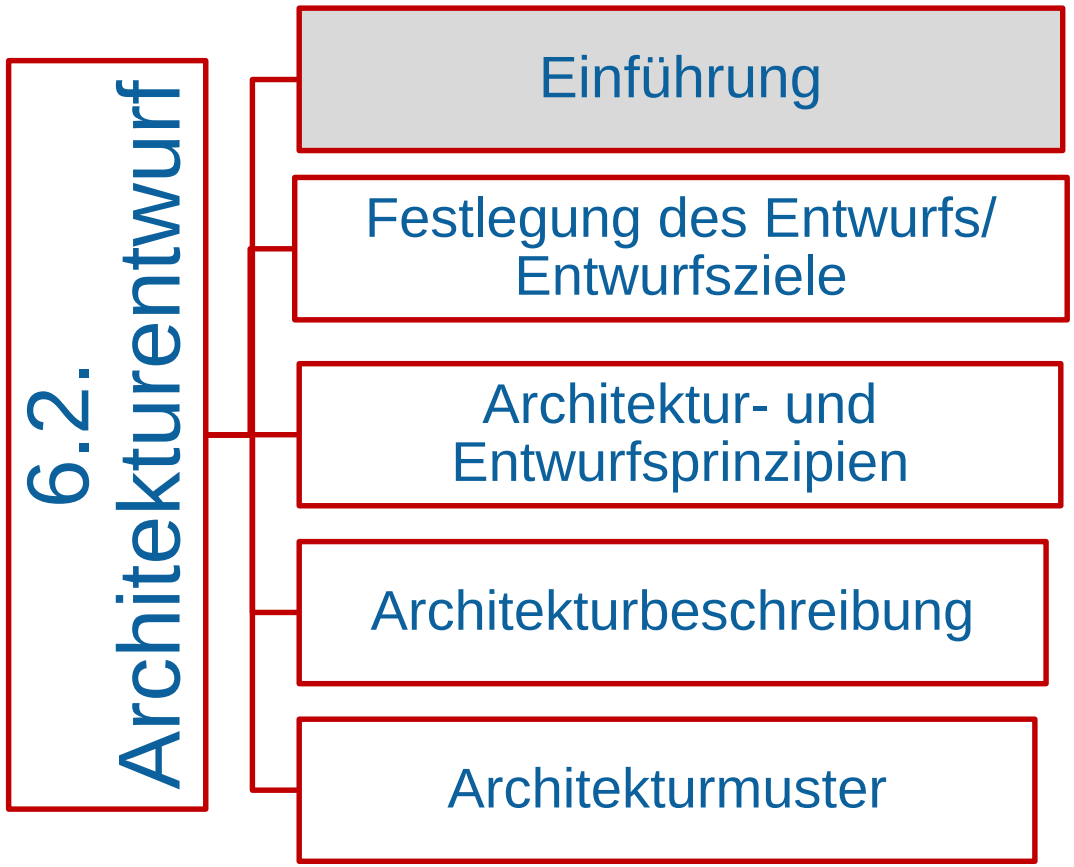
Wichtige Themen der Evolution (1)

- Software wird kontinuierlich weiterentwickelt => Evolution sollte vorbereitet und unterstützt werden
- Softwareentwicklung findet typischerweise nicht „auf der grünen Wiese“ statt (selten „Greenfield Development“)
- Bei der Entwicklung von Code ist darauf zu achten, dass
  - Code leicht weiterzuentwickeln ist
    - Entwurfsmuster (siehe 4.6.)
    - Architektur (6.2)
    - Wiederverwendung (6.3)
  - Und bei Weiterentwicklung der existierende Code und seine Prinzipien nicht unnötig verändert werden
    - Dokumentation der Prinzipien als Rationale (siehe 4.7.)

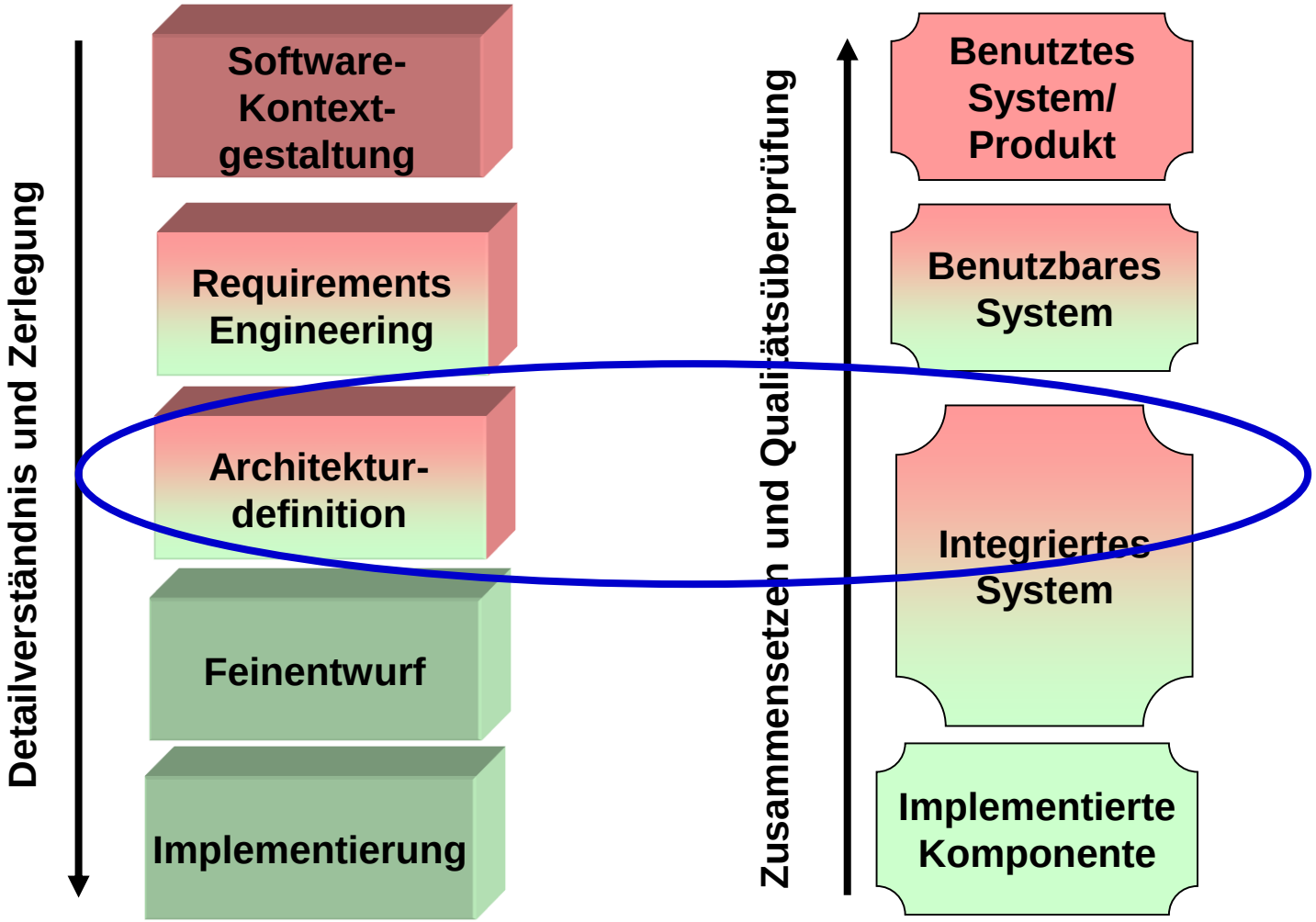
Wichtige Themen der Evolution (2)

- Das Management der Softwareentwicklung und der Betrieb sind auf Evolution auszurichten
  - Änderungsmanagement für Weiterentwicklung (6.4)
  - Geeigneter Betrieb und IT-Governance (6.5)

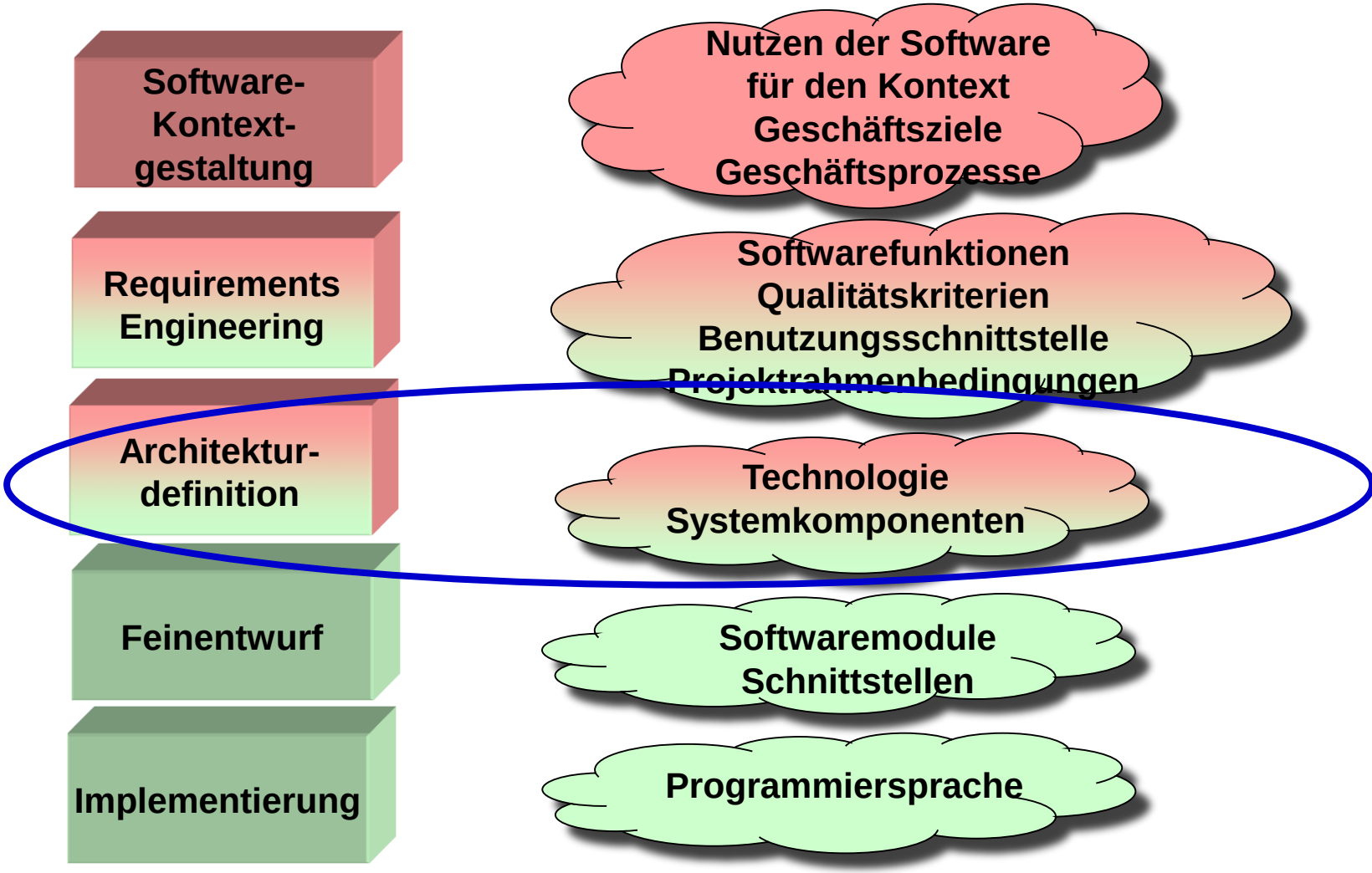
- Schlechte Erstentwicklung muss sonst nachträglich verbessert werden
  - Re-Engineering (6.6)



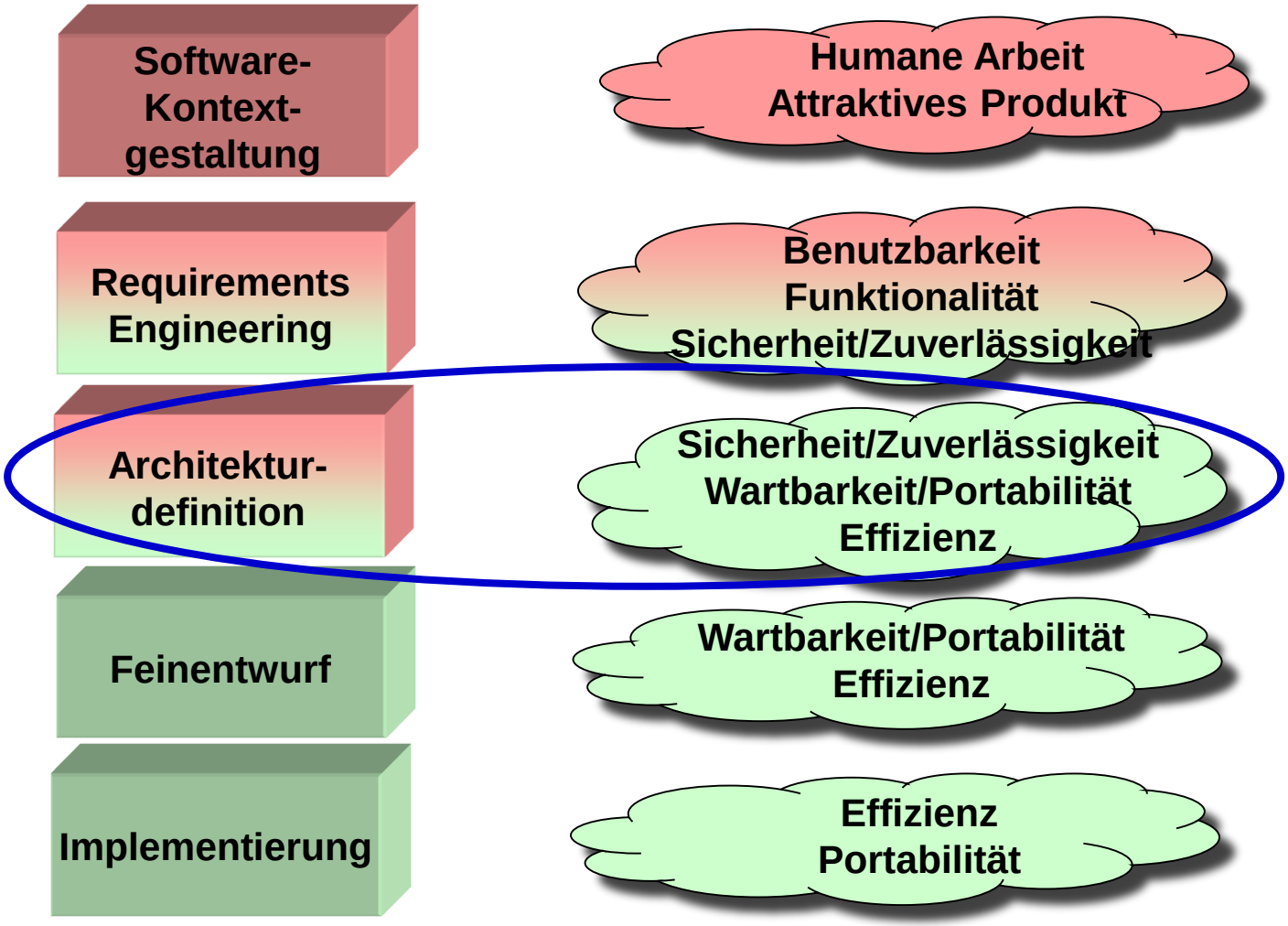
Wdh. Folie 01: Aktivitäten und Ergebnisse der Entwicklung



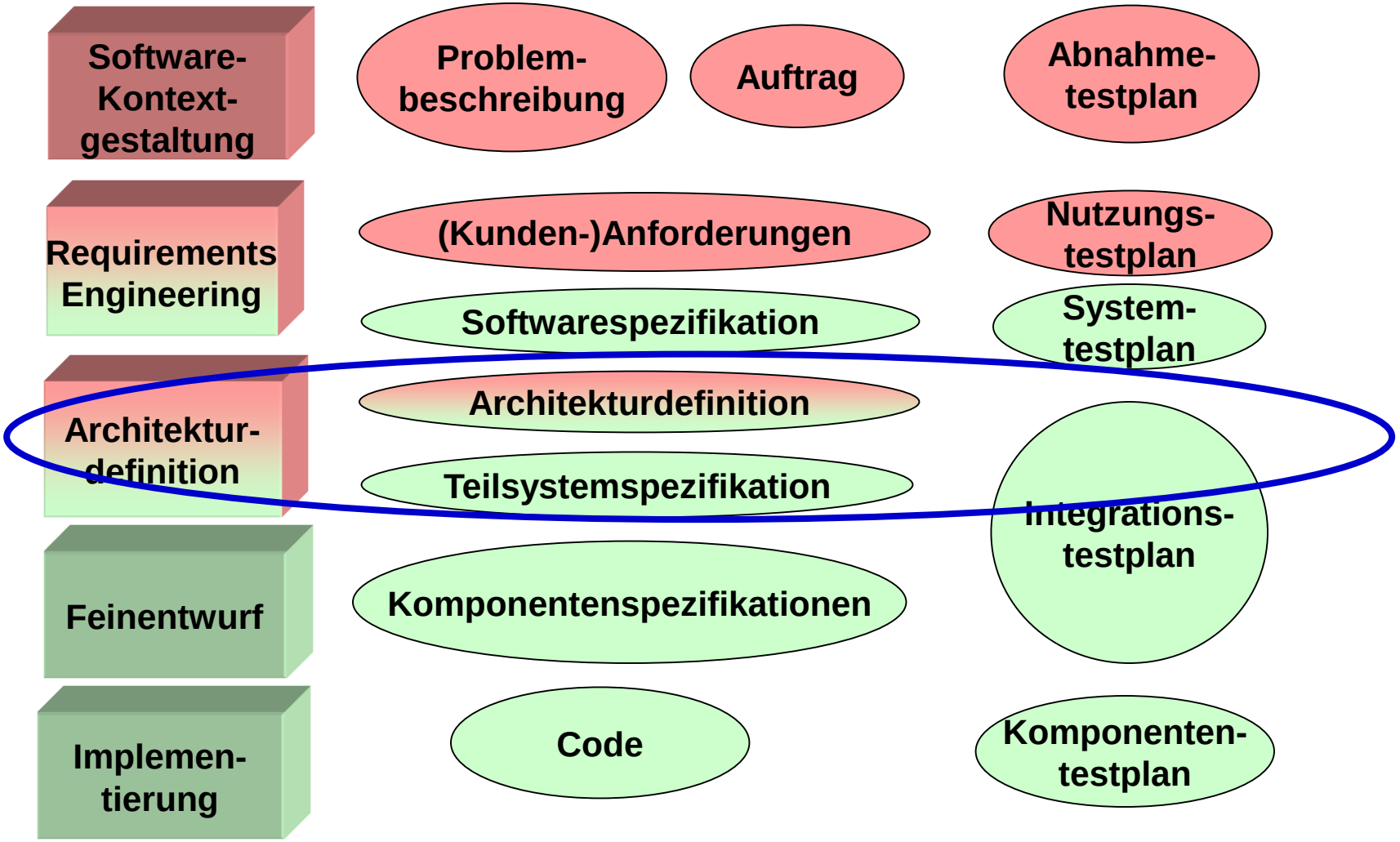
Wdh. Folien 01: Aktivitäten und Gestaltungsentscheidungen



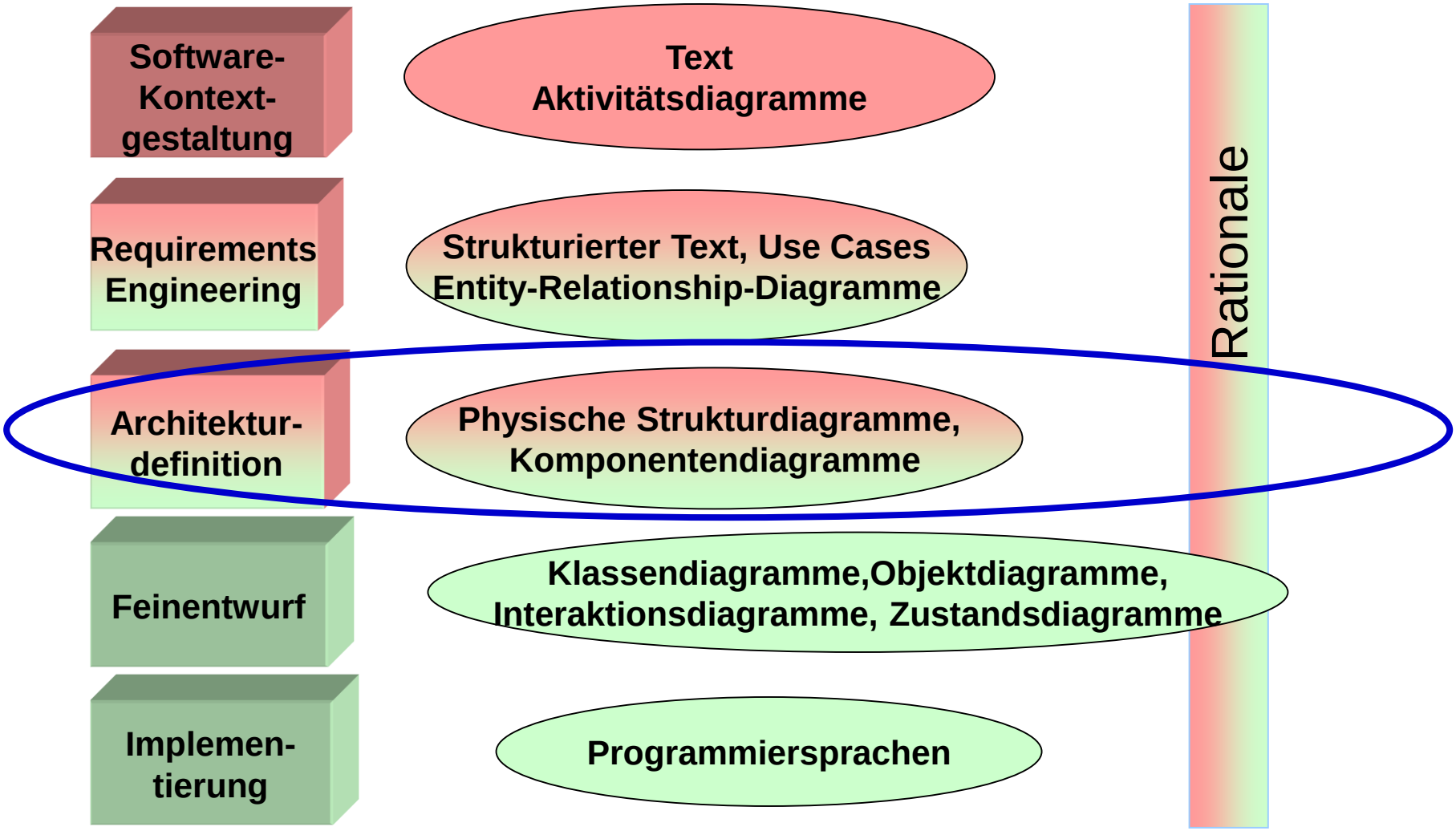
Wdh. Folien 11: SW-Produktqualität nach ISO/IEC 25010:2011



Wdh. Folien 06: Es gibt viele Dokumente für die Softwareentwicklung



Wdh. Folien 07: Beschreibungstechniken



Was ist eine Architektur?

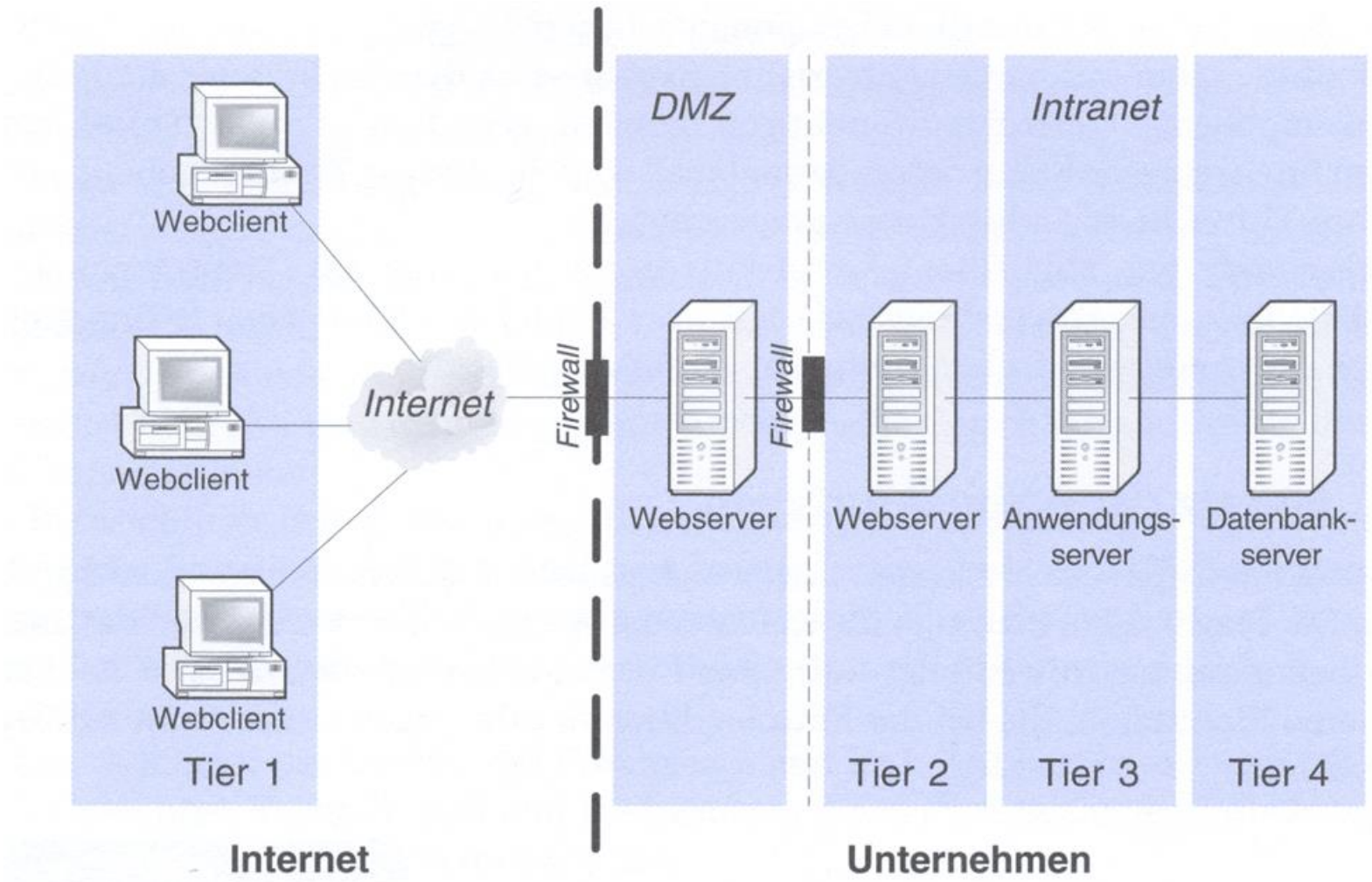
Definition: Architektur

- IEEE-Std. 1471-2011 Recommended Practice for Architectural Description of Software-Intensive Systems - Description (http://www.iso-architecture.org/ieee-1471/)

- Architektur beschreibt die grundlegende Organisation eines Systems verkörpert durch seine Komponenten, ihre Beziehungen zueinander und zur Umgebung

- Architektur beschreibt die Prinzipien des Entwurfs (oft durch eine Referenzarchitektur)

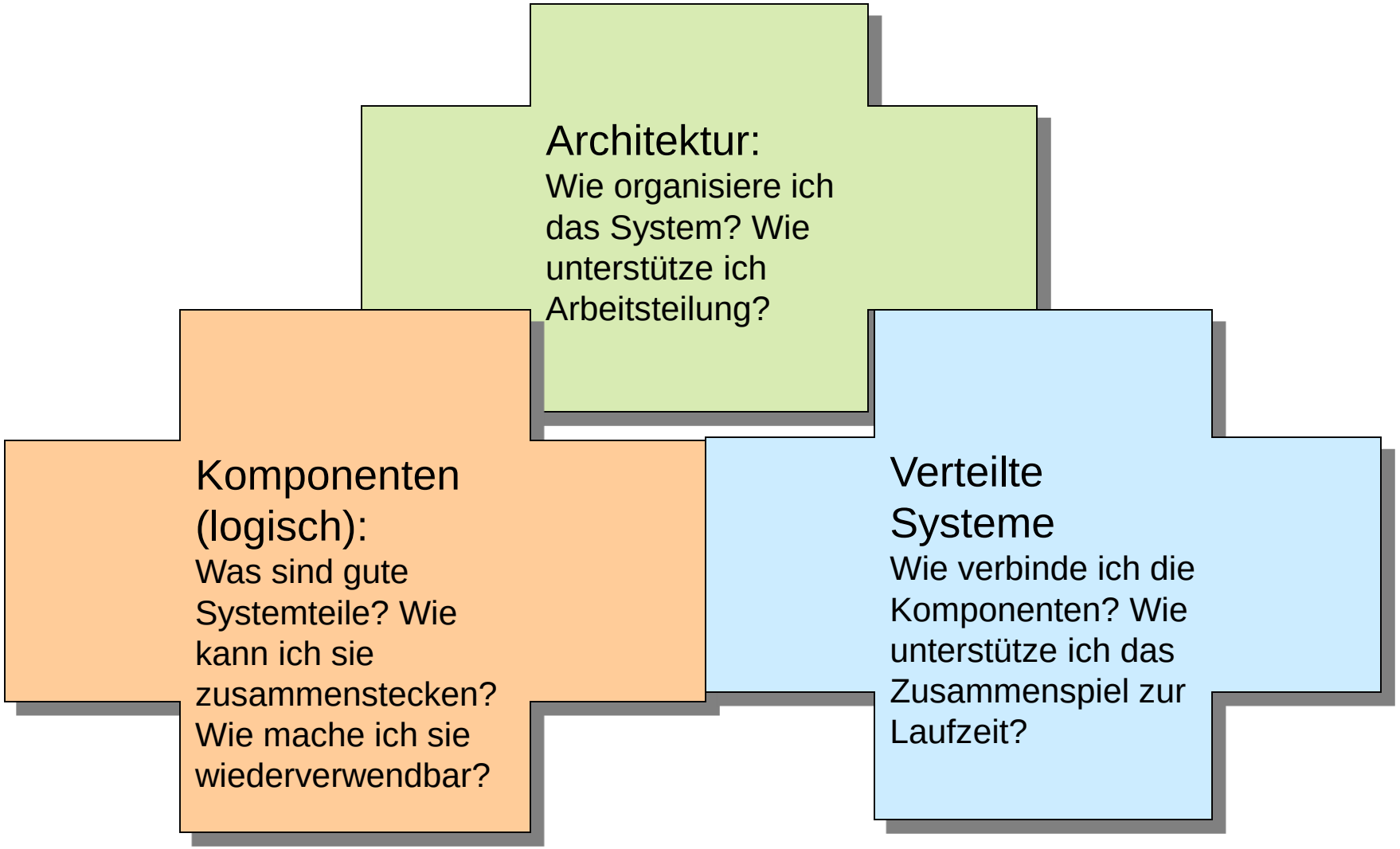
Beispiel: 4+Tier-Architektur



Motivation

- Eine gute Software-Architektur zu erstellen, ist schwierig, weil
  - Eine Struktur festzulegen ist, die langfristig trägt
  - Die Realisierbarkeit vieler Anforderungen, insbesondere von Qualitätseigenschaften, von der Architektur abhängt
  - Der technologische Fortschritt sehr schnell ist, insbesondere im Bereich Middleware
  - Wenig Standards existieren
  - Theoretisch klare Konzepte (z.B. Schnittstellen) in der Praxis oft nicht durchzuhalten sind
  - Architektur oft auch die Arbeitsteilung im Projekt bestimmt

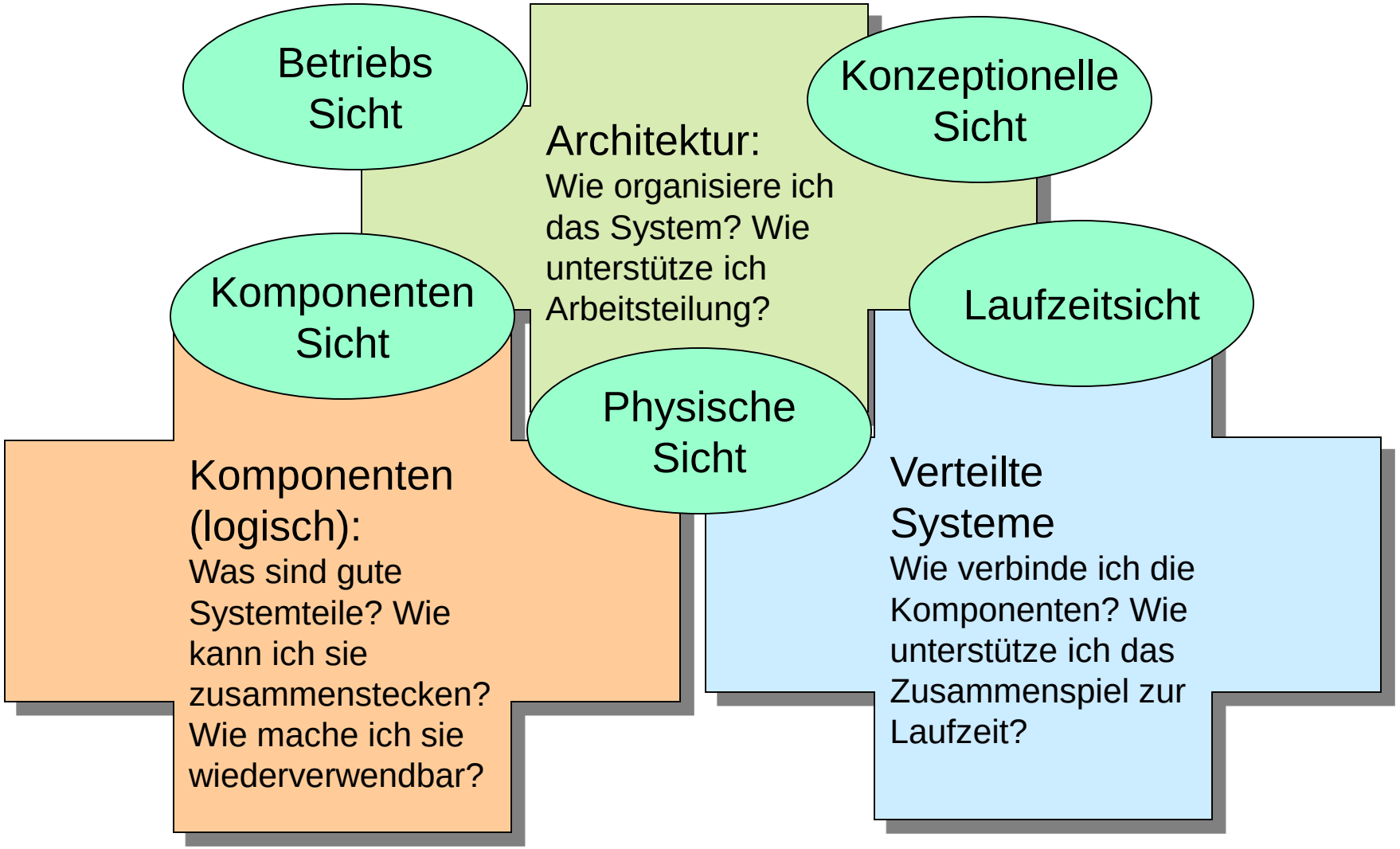
Wissensgebiete der SoftwarearchitektInnen



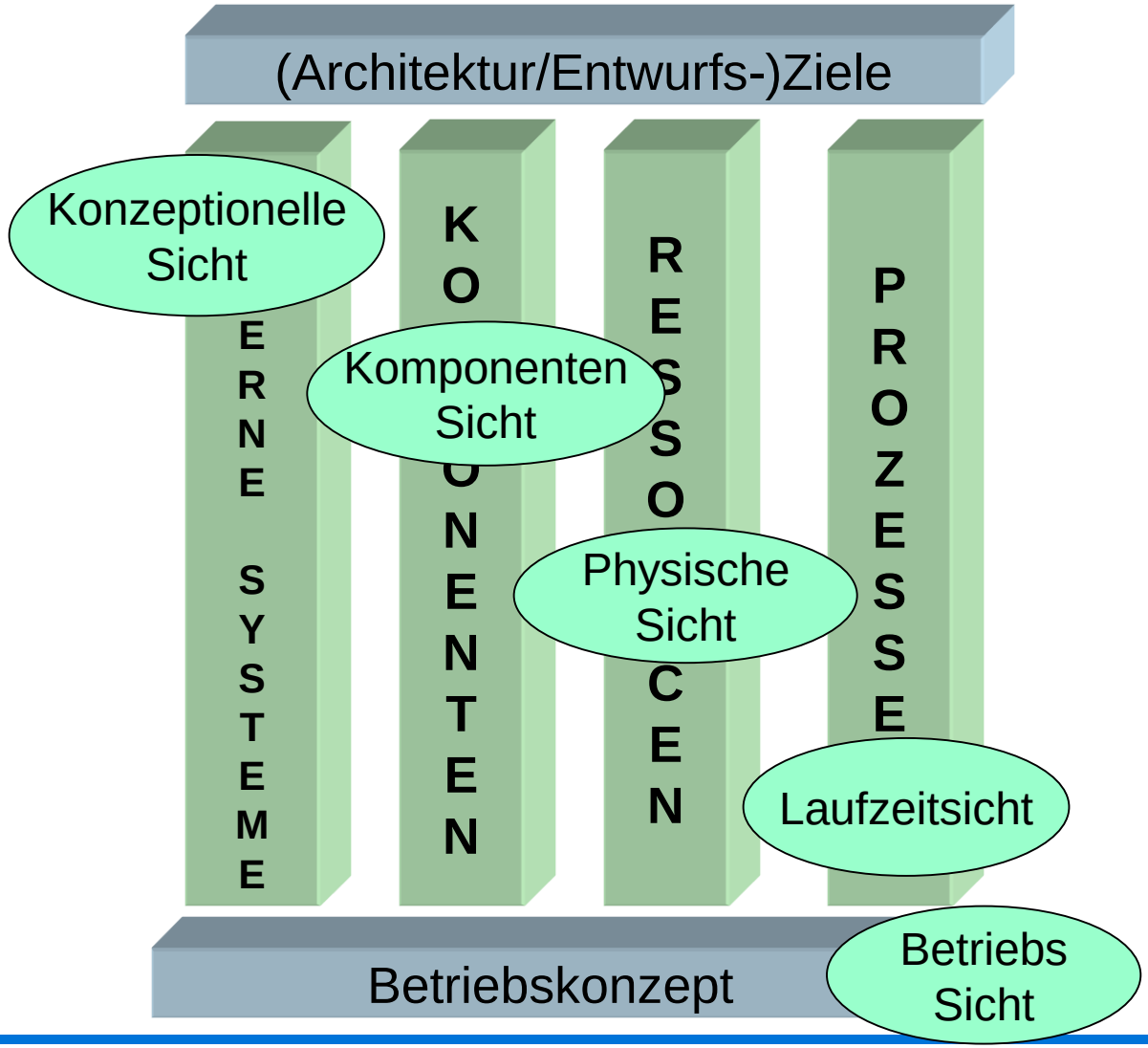
Sichten auf Software-Architektur

- **Konzeptionelle Sicht** für AuftraggeberIn, NutzerIn
  - Grobüberblick (inkl. Externer Systeme), meist mit „Konfigurationsdiagrammen“
- **Komponentensicht** für EntwicklerIn
  - Komponenten, Schnittstellen und innere Struktur (Bündelung und Konkretisierung der Analyseklassen), statische Qualitätsziele (z.B. Wartbarkeit)
- **Physische Sicht** für AuftraggeberIn, ProgrammiererIn
  - Ressourcen und Verteilung der Komponenten auf die Ressourcen, dynamische Qualitätsziele (z.B. Effizienz, Zuverlässigkeit)
- **Laufzeitsicht** für IntegratorIn, ProgrammiererIn
  - Prozesse (die beim Programmablauf entstehen), dynamische Qualitätsziele
- **Betriebssicht** für AuftraggeberIn (OperatorIn)
  - Betriebskonzept

Wissensgebiete der SoftwarearchitektInnen



Welche Inhalte umfasst der Architekturentwurf?



Architektur/Entwurfsziele

- Architektur/Entwurfsziele
  - Beschreiben die durch die Architektur und den Feinentwurf zu erreichenden Systemeigenschaften
  - Konkretisieren die Qualitätsanforderungen durch Betrachtung der Komponenten und Ressourcen sowie weiterer Rahmenbedingungen

- Beispiele für Architektur/Entwurfsziele
  - Flexibilität (Konfigurierbarkeit, Wartbarkeit, Erweiterbarkeit)
  - Betriebssicherheit (Sicherheit und Zuverlässigkeit, z.B. Fehlertoleranz, Ausfallsicherheit, Verfügbarkeit)
  - Effizienz

Externe Systeme

- Externe Systeme sind die Nachbarsysteme (d.h. existierende Anwendungssoftware), mit denen das System zusammenarbeitet

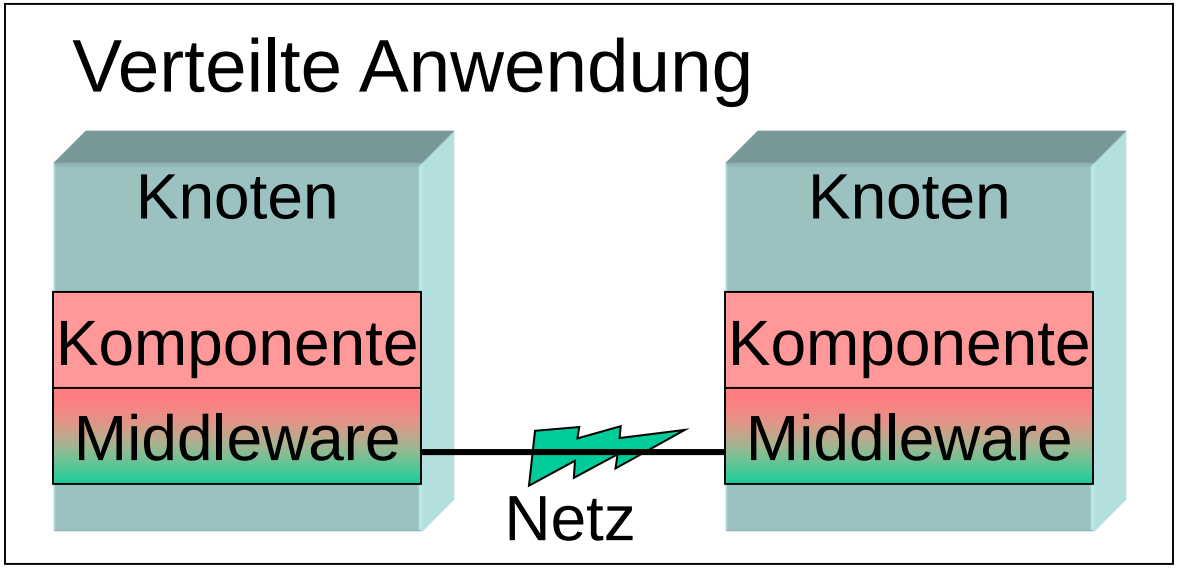
- Achtung: Abgrenzung zu Ressourcen manchmal nicht klar, wenn das andere System vor allem als Server arbeitet

Ressourcen

- Ressourcen
  - Umfassen die benötigte Software und Hardware
- Beispiele für Hardwareressourcen:
  - Rechner (Prozessoren)
  - Netzwerk (Kommunikation)
  - Bildschirme (Ein/Ausgabe)
  - Drucker (Ausgabe)
- Beispiele für Softwareressourcen:
  - Middleware (z.B. Datenbankschicht),
  - Systemsoftware (z.B. Betriebssystem)

Middleware für verteilte Anwendungen

- Dienste des Betriebssystems sind für die meisten Anwendungen zu rudimentär. Eine **Middleware** ist eine intelligente Programmierschnittstelle zur Nutzung eines verteilten Systems durch eine Anwendung.



- Beispiele sind CORBA, JAVA-EE, .NET

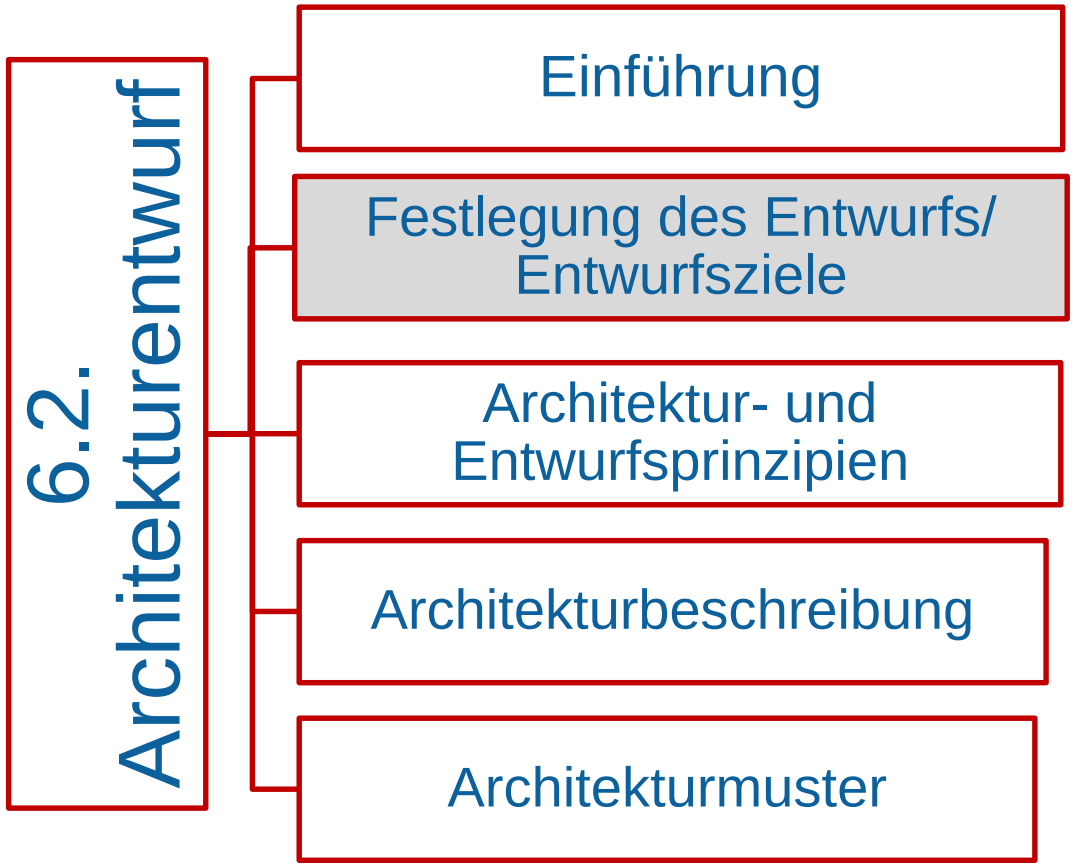
Prozesse

- Prozesse
  - Repräsentieren das System zur **Laufzeit**
  - Müssen **koordiniert** werden

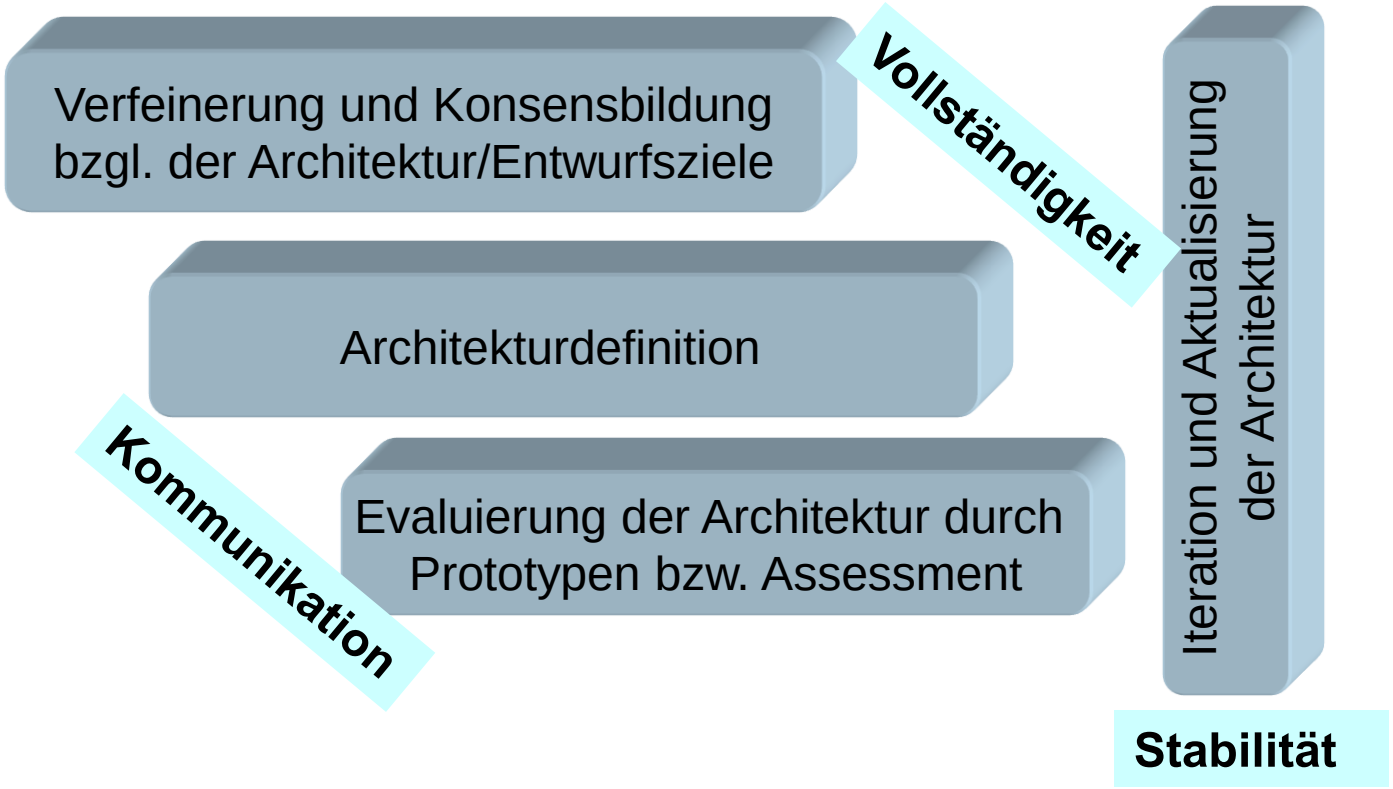
- Beispiele für Prozesse
  - Dialogprozesse, reagieren auf Benutzereingaben
  - Batchprozesse
  - JAVA-Threads

Betriebskonzept

- Das Betriebskonzept
  - Beschreibt die **Grenzfälle der Nutzung**, d.h.
    - Installation, De-Installation
    - Hochfahren, Herunterfahren
    - Ressourcen-Ausfall
    - Komponenten-Ausfall



Was ist beim Architekturentwurf zu tun?



Festlegung von Architektur/Entwurfszielen

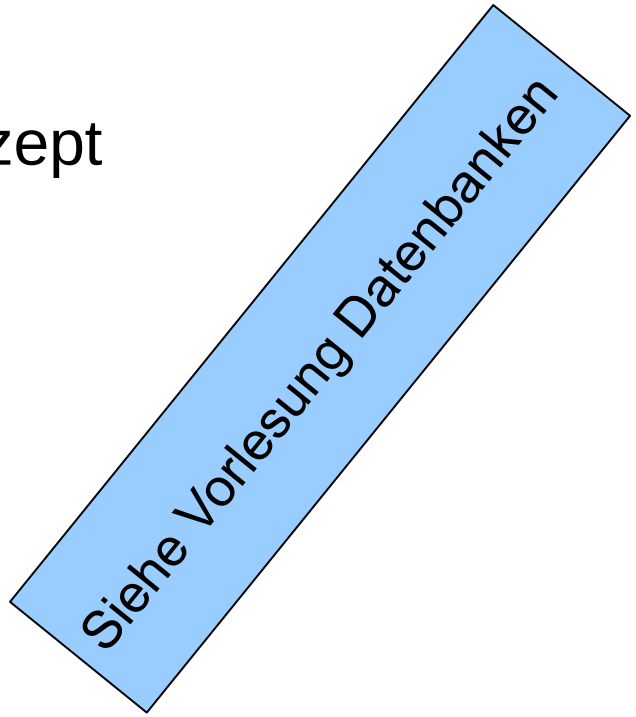
- Typische Ziele in der Praxis
  - Allgemein: hohe Effizienz, hohe Flexibilität, hohe Wartbarkeit
  - Sicherheit
  - Globale funktionale Anforderungen
    - Verteilung
    - Persistenz der Daten
    - Integration von Altsystemen und zugekauften Systemen
    - Protokollierung
    - Internationalisierung

Verteilung

- Ermöglicht
  - Lösung größerer Probleme
  - Einbindung von Altsystemen
  - Ausfallsicherheit
- Verursacht
  - Overhead in Kommunikation
  - Probleme bei der Entwicklung (z.B. Test)
- Benötigt komplexe Basismechanismen (Middleware)
  - Zur Bearbeitung verteilter Objekte
  - Für Datenkonvertierung und –transport
  - Für Fehlerbehandlung über Systemgrenzen hinweg

Persistenz

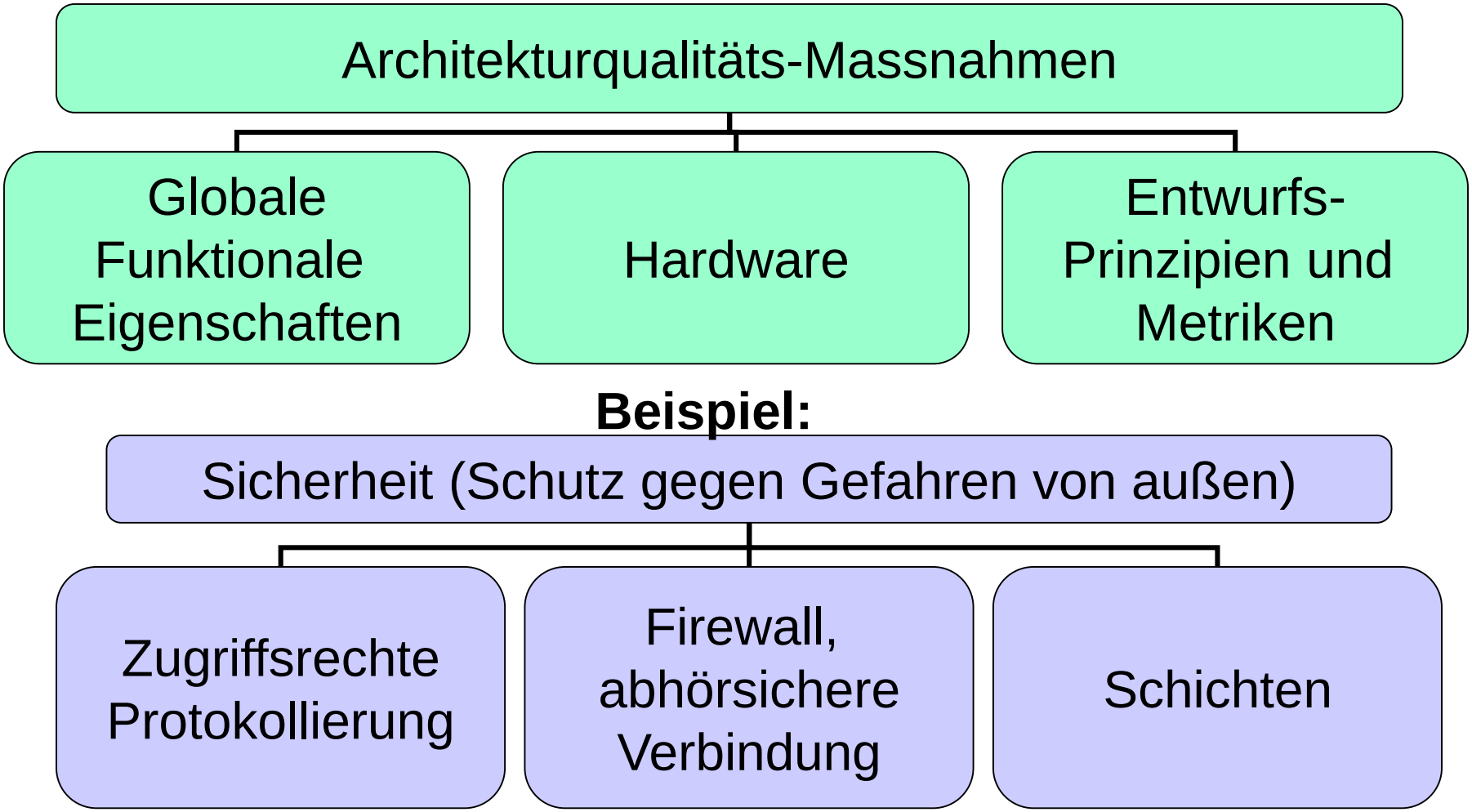
- Sicherstellen der Daten über die Programmlaufzeit hinaus
- Persistenzmechanismus sollte gekapselt sein, d.h. Zugriffe zu Datenbank oder Filesystem
- Oft verbunden mit Transaktionskonzept
- Beispiele:
  - Enterprise JavaBeans

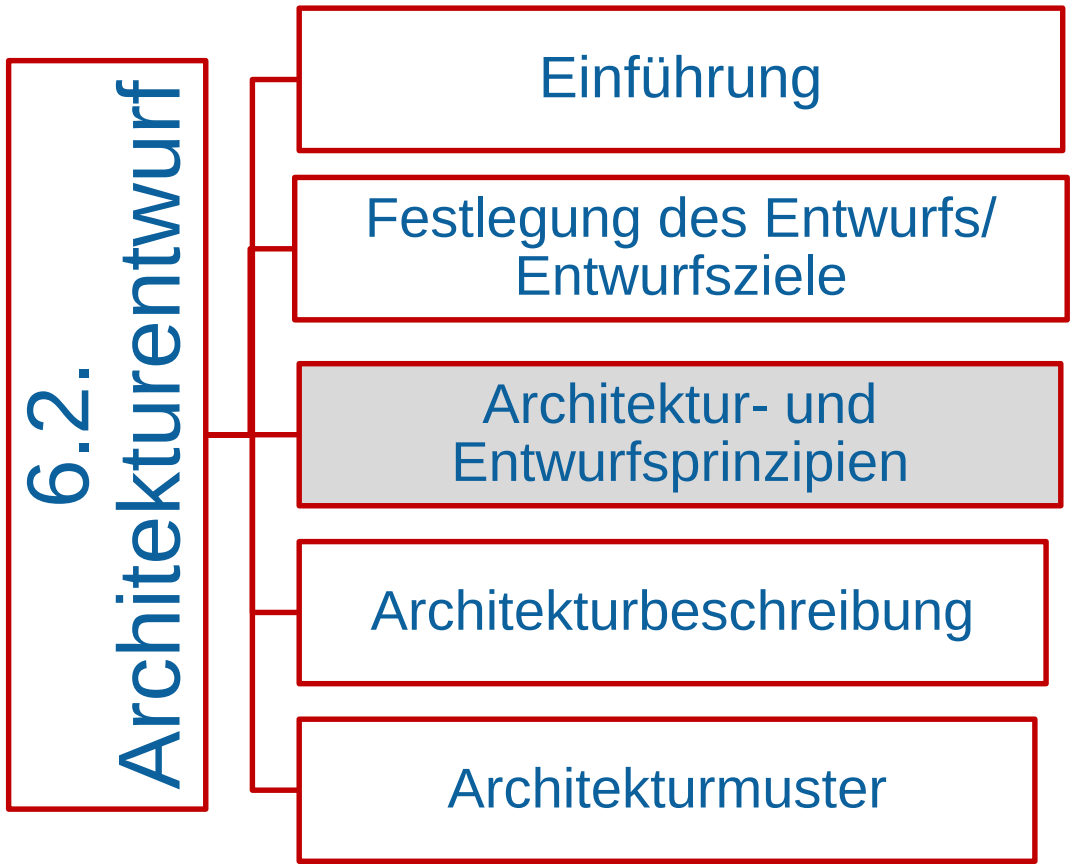


Integration mit anderen Systemen

- Einbezug von Altsystemen und zugekauften Systemen
- Sollte möglichst wenig Änderungen an vorhandenen Systemen bewirken
- Insbesondere schwierig bei übergreifenden Transaktionen
- Typisches Beispiel:
  - Wrapper Konzept

Umsetzung von Architekturqualität





Architektur- und Entwurfsprinzipien

- Es ist wichtig, beim Architekturentwurf die Abhängigkeiten zwischen den Komponenten zu reduzieren, da diese
  - teuer (wg. Kommunikation) und
  - fehleranfällig sind
- Andererseits dürfen die Komponenten nicht zu groß sein, da dann wenig Zusammenhang der Elemente in der Komponente, so dass die Wartbarkeit erschwert ist.

- Beispiel: z.B. „Gottklasse“, die sehr viele Operationen und Attribute hat
  - Wenige Abhängigkeiten
  - Aber schlecht wartbar, weil unklar ist, was wirklich zusammengehört und konsistent geändert werden muss

Arten der Abhängigkeit

- Kommunikation
  - Eine Komponente verwendet eine Operation (Dienst) einer anderen Komponente.
- Gemeinsame Daten
  - Zwei Komponenten greifen auf die gleichen Daten (z.B. Variablen, Datei, Datenbank) zu oder nutzen die gleiche Daten als Kommunikationsmedium, um Daten auszutauschen. Dieser Zugriff ist immer asynchron.
- Gemeinsame Systemressourcen
  - Zwei Komponenten verwenden die gleichen Systemressourcen (z.B. CPU, Speicher, Festplatte).
- Vererbung (nur bei Klassen, nicht bei Komponenten)
  - Eine Unterklasse erbt alle Eigenschaften (Attribute und Operationen) einer oder mehrerer Oberklassen.

Guter Entwurf (1)

- Grundprinzip Modularisierung
  - Separation of Concerns (Zerlegung des Systems in handhabbare Einzelteile)
    - Typisches Beispiel: Klassen der Objektorientierung
  - Information Hiding (Nur wesentliche Informationen einer Komponente nach außen bekannt machen)
    - Typisches Beispiel: Schnittstellenkonzept, Facade-Muster

- Weiterführende Prinzipien
  - Hohe Kohäsion (siehe auch Folien 08)
  - Niedrige Kopplung (siehe auch Folien 08)
  - Abstraktion (Schichtenbildung)
  - Wiederverwendung

Guter Entwurf (2)

- Hohe Kohäsion innerhalb einer Komponente
  - erleichtert Verständnis, Wartung und Anpassung, da alle betroffenen Elemente (und keine anderen!) in einer Komponente zu finden sind
- Niedrige Kopplung
  - erleichtert die Wartung und macht Systeme stabiler, da wenig Komponentenübergreifende Kommunikation (auch gut für Arbeitsteilung)
- Hierarchische Zerlegung, Schichtung
  - Spezielle Form von hoher Kohäsion und niedriger Kopplung
- Wiederverwendung
  - Verringert Redundanz, erhöht Stabilität und Handhabbarkeit

Wdh. Folien 08: Prinzip Kohäsion

- Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente
- Hohe Kohäsion: starke Abhängigkeit zwischen den Elementen einer Komponente
  - Es gibt keine Zerlegung in Untergruppen von zusammengehörigen Elementen,
    - z.B. Menge der Attribute einer Klasse nicht so zerlegbar, dass ein Teil der Operationen nur auf einem Teil der Attribute arbeitet
- Mechanismen zu Erreichung von Kohäsion:
  - Früher: ähnliche Funktionalitäten zusammenfassen
    - Schlecht, wenn Daten verstreut
  - Prinzipien der Objektorientierung (Datenkapselung)
    - Operationen der Klasse zuordnen, auf deren Attribute sie zugreifen
  - Verwendung geeigneter Muster zu Kopplung und Entkopplung

Wdh. Folien 08: Prinzip Kopplung

- Kopplung ist ein Maß für die Abhängigkeiten zwischen Komponenten.
- Niedrige Kopplung: geringe Abhängigkeiten zwischen Komponenten
- Mechanismen zur Reduktion der Kopplung:
  - **Schnittstellenkopplung**, d.h. Austausch nur über Schnittstellen
    - z.B. get/set-Operationen statt Attributzugriff (also kein direkter Zugriff auf Attribute von außen)
  - Möglichst wenig Aufrufe zwischen den Komponenten
- Datenkopplung, d.h. gemeinsame Daten von Komponenten vermeiden!
- Strukturkopplung, d.h. gemeinsame Strukturanteile vermeiden !
  - z.B. keine Vererbung über Komponentengrenzen hinweg

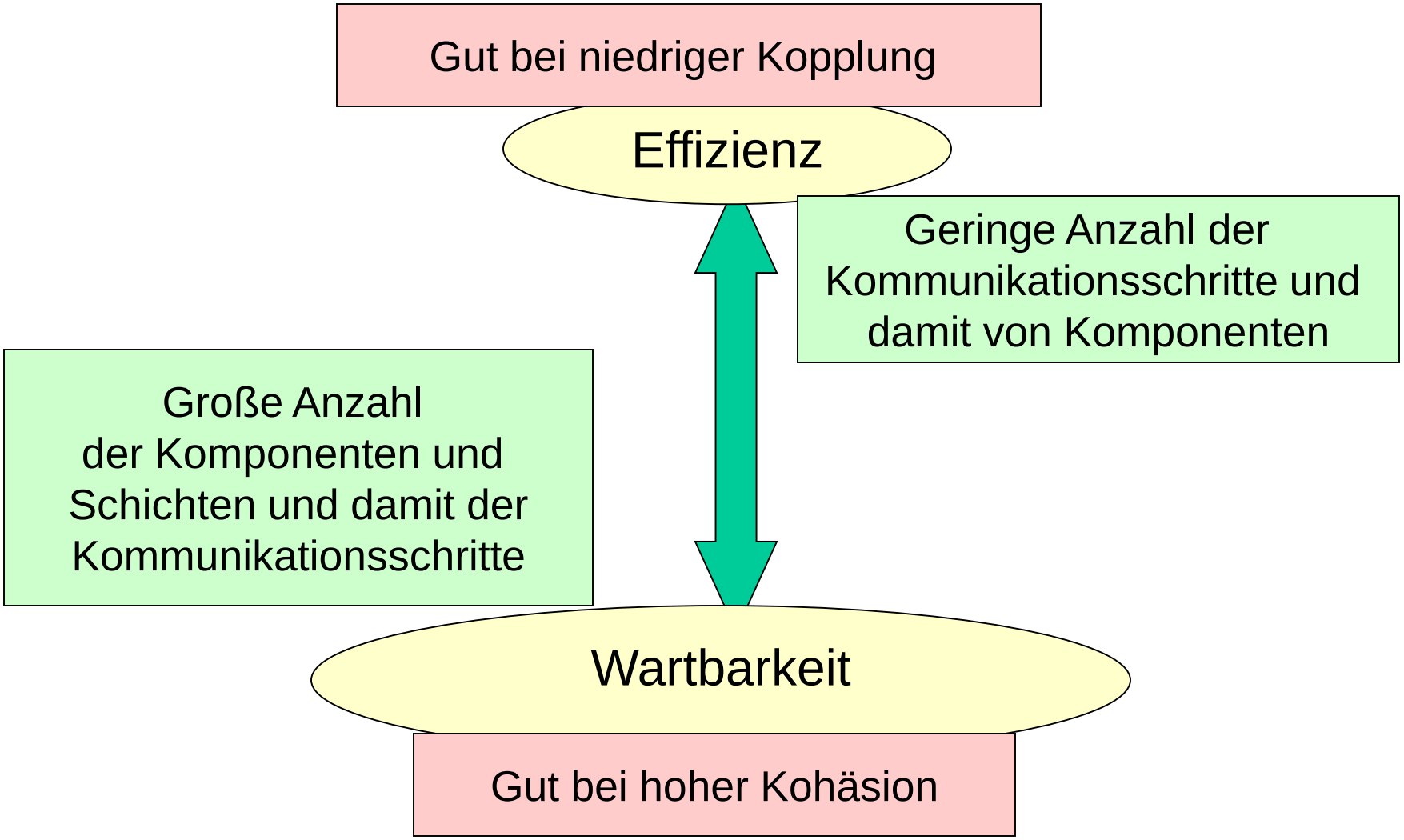
Übung: Wartbarkeit vs. Effizienz

- Vergleich der Architekturziele (Qualitätseigenschaften) Wartbarkeit und Effizienz
- Effizienzgruppe bzw. Wartbarkeitsgruppe
  - Durch welche Mechanismen erreicht man Wartbarkeit bzw. Effizienz?
  - Welchen Zusammenhang gibt es zu Kohäsion und Kopplung?

Wartbarkeit vs. Effizienz

	Wartbarkeit	Effizienz
Mechanismen		
Bezug zu Kopplung		
Bezug zu Kohäsion		

Architekturkonflikte



Hohe Effizienz

- Wird erreicht durch
  - Stärkere Hardware
  - Parallelität, Cache
  - Verringern der Kommunikation, große Komponenten (Niedrige Kopplung)

  ABER bewirkt auch
  - Niedrige Kohäsion durch großer Komponenten
  - Verringern der Verteilung durch Zentralisierung
  - Einführung von Redundanzen und damit Verringerung der Wartbarkeit

- Verringert Wartbarkeit und Kohäsion

Hohe Wartbarkeit

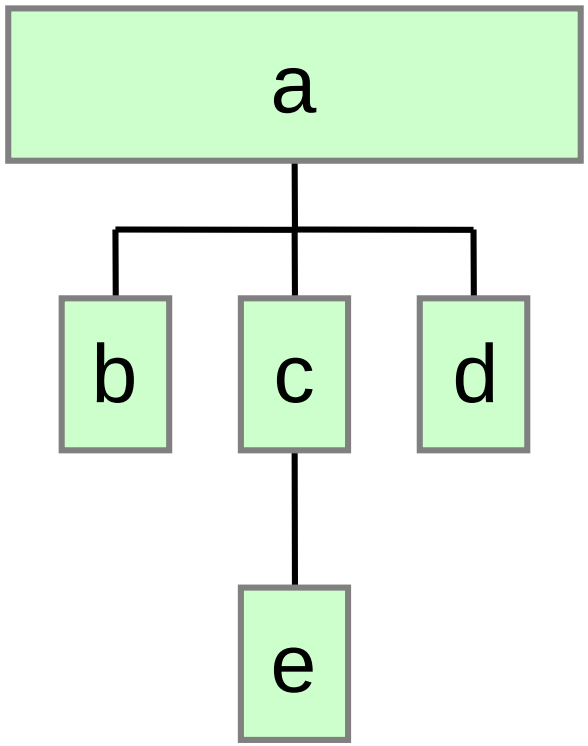
- D.h. leichte Änderbarkeit von
  - Funktionalität, Datenstrukturen
  - Schnittstellen zu anderen Systemen
  - Benutzungsschnittstellen, Plattform

- Wird erreicht durch
  - Abstraktion
  - Kapselung (Verstecken interner Details)
  - viele kleine Komponenten (hohe Kohäsion)

  ABER bewirkt auch
  - Hohe Kopplung durch Kommunikation zwischen vielen Komponenten
  - damit Verringerung der Effizienz

- Verringert Effizienz und erhöht Kopplung

Prinzip: Hierarchische Zerlegung



Elemente jeder Ebene bilden eine Schicht:
Geschlossene Architektur:
Kommunikation nur innerhalb einer Schicht
und zu angrenzenden Schichten

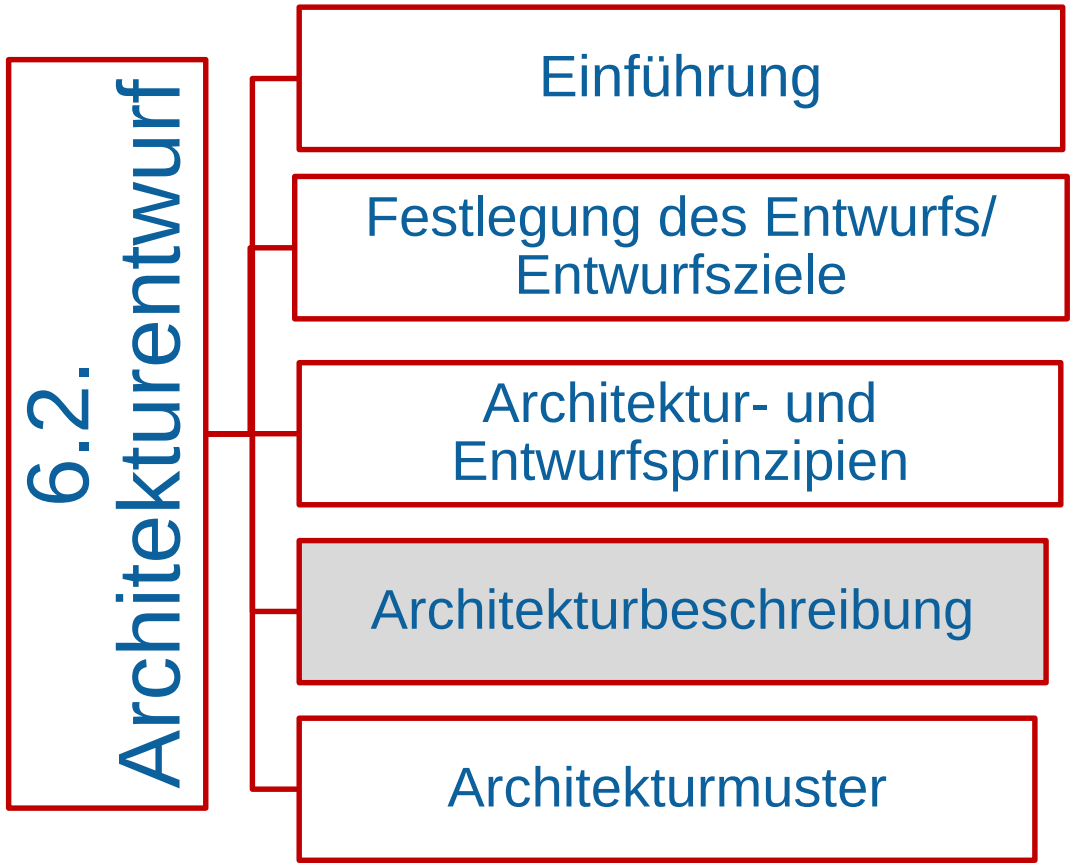
⇒ **Niedrige Kopplung und hohe Kohäsion**

ABER oft nicht gut anwendbar

Prinzip: Interne Wiederverwendung

- Interne Wiederverwendung ist ein Maß für die Ausnutzung von Gemeinsamkeiten zwischen Komponenten

- Hohe Wiederverwendung: Komponenten wird in vielen Varianten oder in vielen Kontexten verwendet
- Mechanismen für Wiederverwendung:
  - Vererbung, Parameterisierung
  - Module/Objekte mit allgemeinen Schnittstellen
- Aber: Wiederverwendung kann die Kopplung erhöhen:
  - Problem Strukturkopplung (z.B. bei Vererbung)



Architekturdokumentation

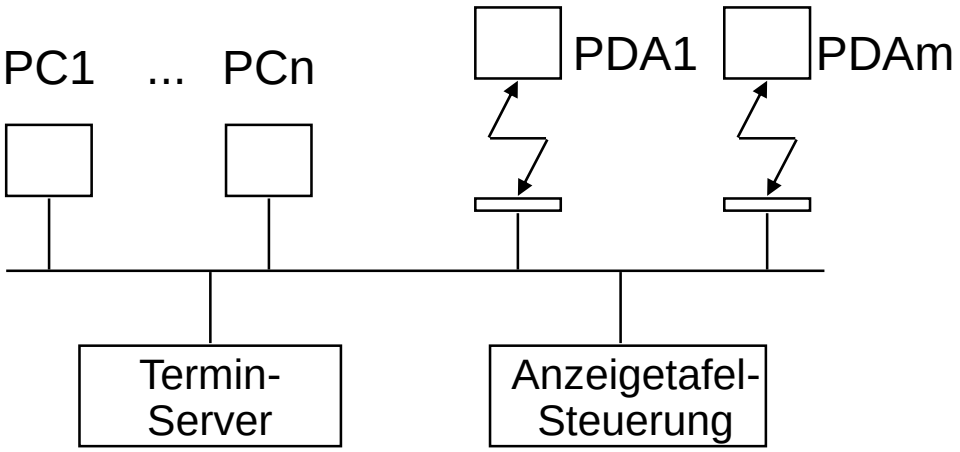
- Einleitung (analog zu RE-Dokumenten)
  - Zweck, Umfang des Systems, Abkürzungen, Referenzen, Überblick, insbesondere Architektur/Entwurfsziele
- SOLL-Architektur (ggf. aufbauend auf IST-Architektur) unter Angabe von
  - Überblick
  - Komponenten und Schnittstellen
  - Externe Systeme (Umgebung)
  - Ressourcen und Abbildung der Komponenten auf die Ressourcen
  - Betriebskonzept
- Hervorgehobene Aspekte
  - Datenverwaltung (Persistenz)
  - Zugangskontrolle und Sicherheit
  - Allgemeiner Kontrollfluss (Prozesse)
- Komponentenbeschreibung
- Annahmen (siehe RE-Dokumente)

[aufbauend auf Bruegge, Dutoit 2004]

Architekturmodellierung

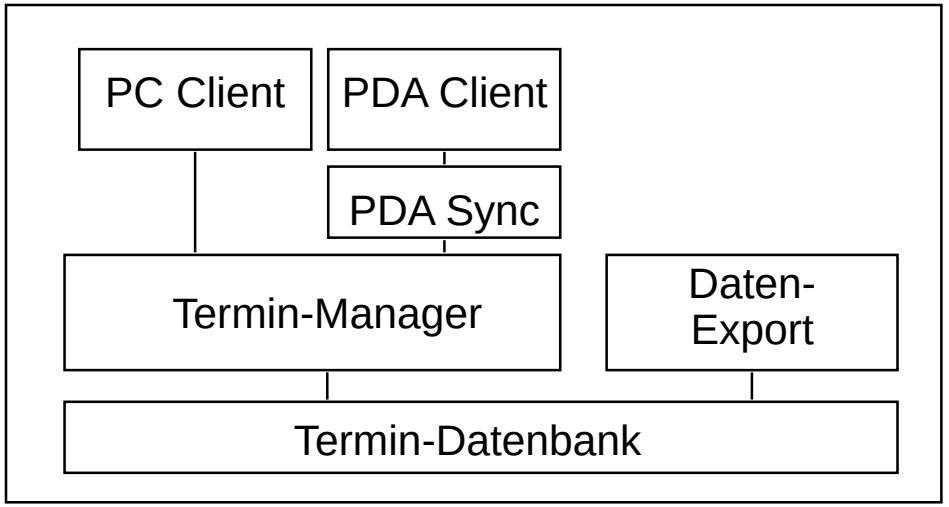
- Komponentenstruktur
  - Paketdiagramm: Überblick über System (rein logische Strukturierung, oft Codepakete)
  - Kompositionsstrukturdiagramm (statisch): Wie ist mein System strukturiert und wie spielen (technische) Komponenten zusammen (Innenleben)?
  - Komponentendiagramm (dynamisch): Welche (technischen) Komponenten entstehen zur Laufzeit und wie sind sie strukturiert (Beziehungen)?
- Physische Struktur
  - Verteilungsdiagramm: Wie sehen die Ressourcen (Hardware, Server, Datenbanken, …) des Systems aus? Wie werden die Komponenten zur Laufzeit wohin verteilt?

Beispiel informelle Diagramme

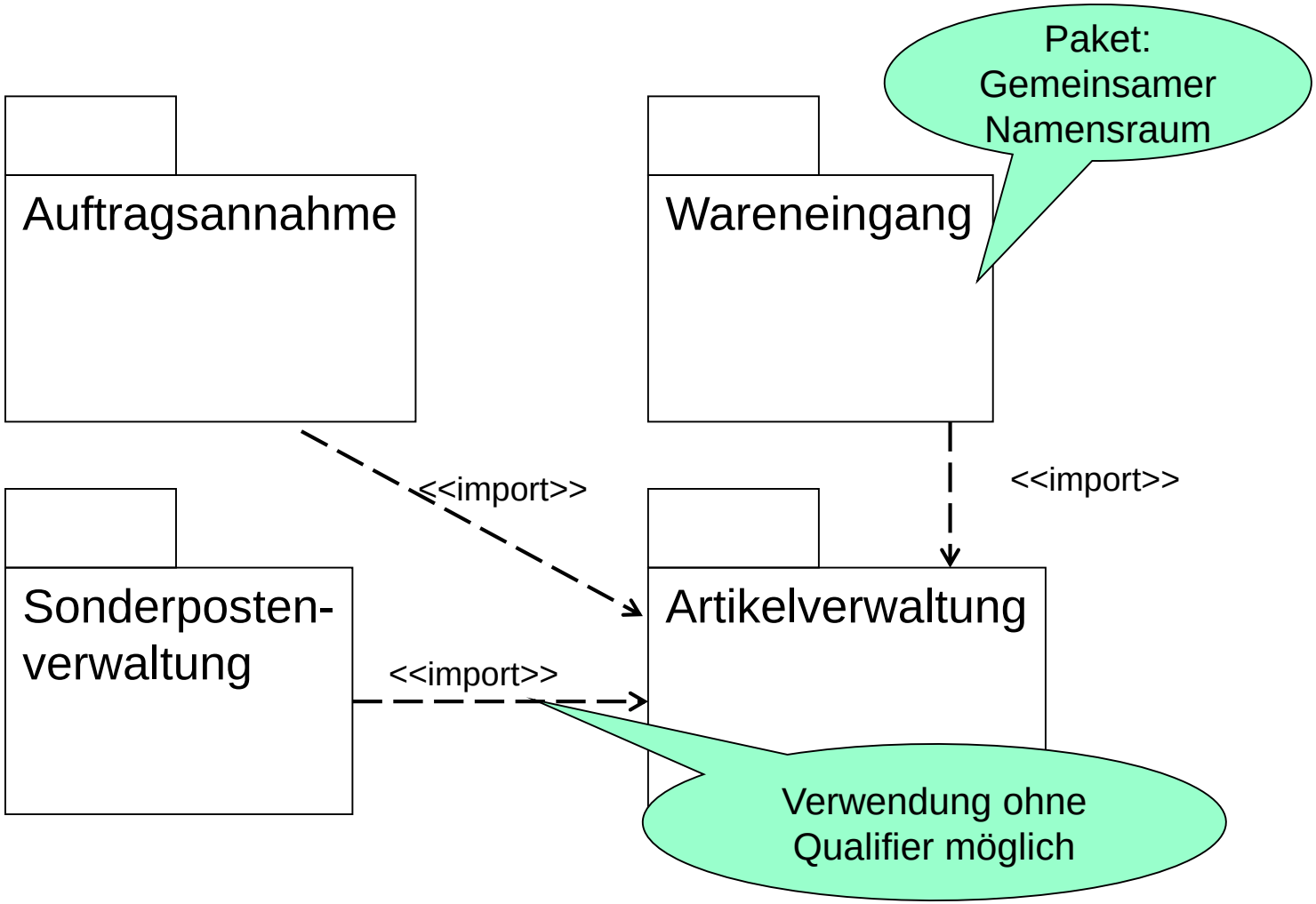


Physikalische Konfiguration

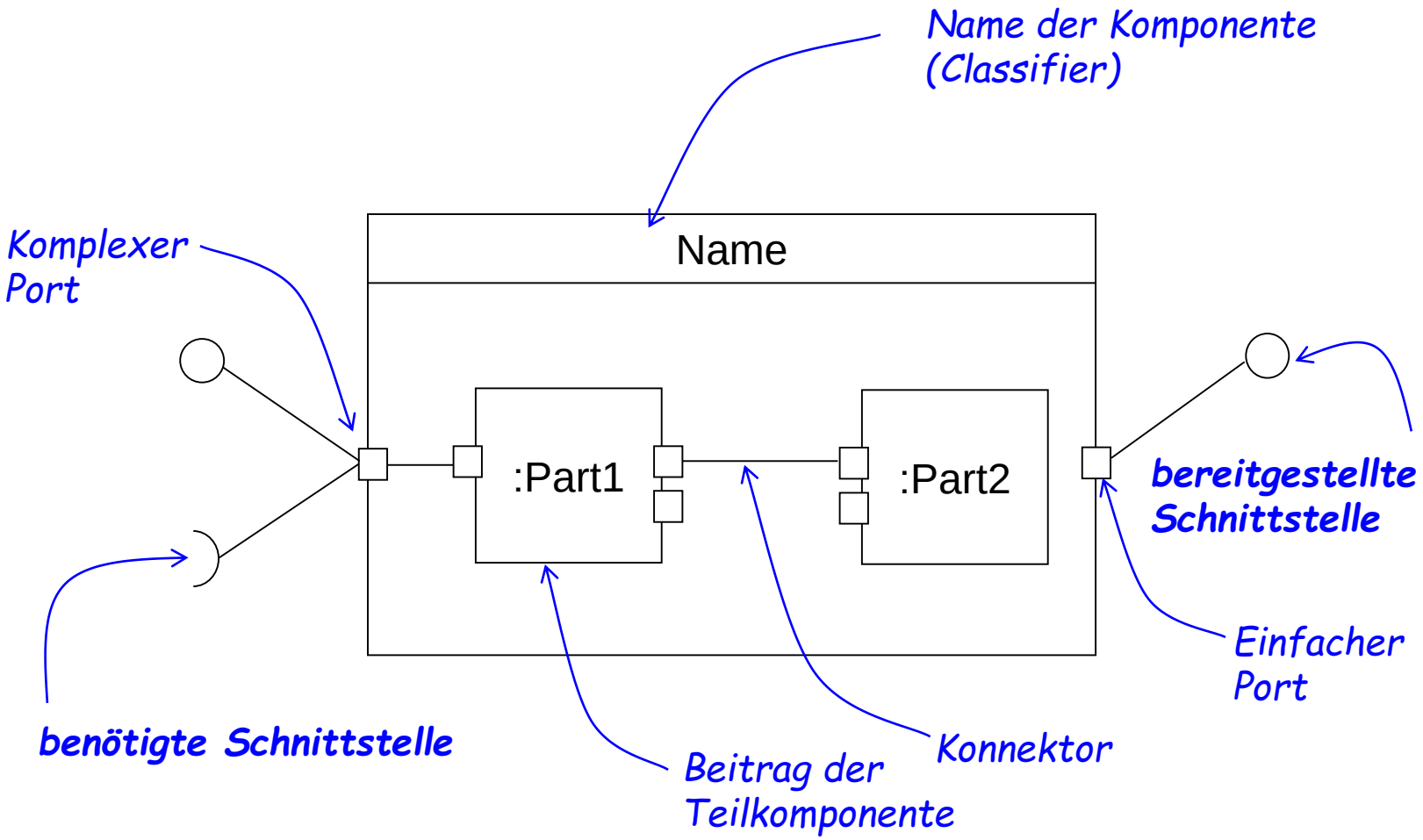
Komponenten



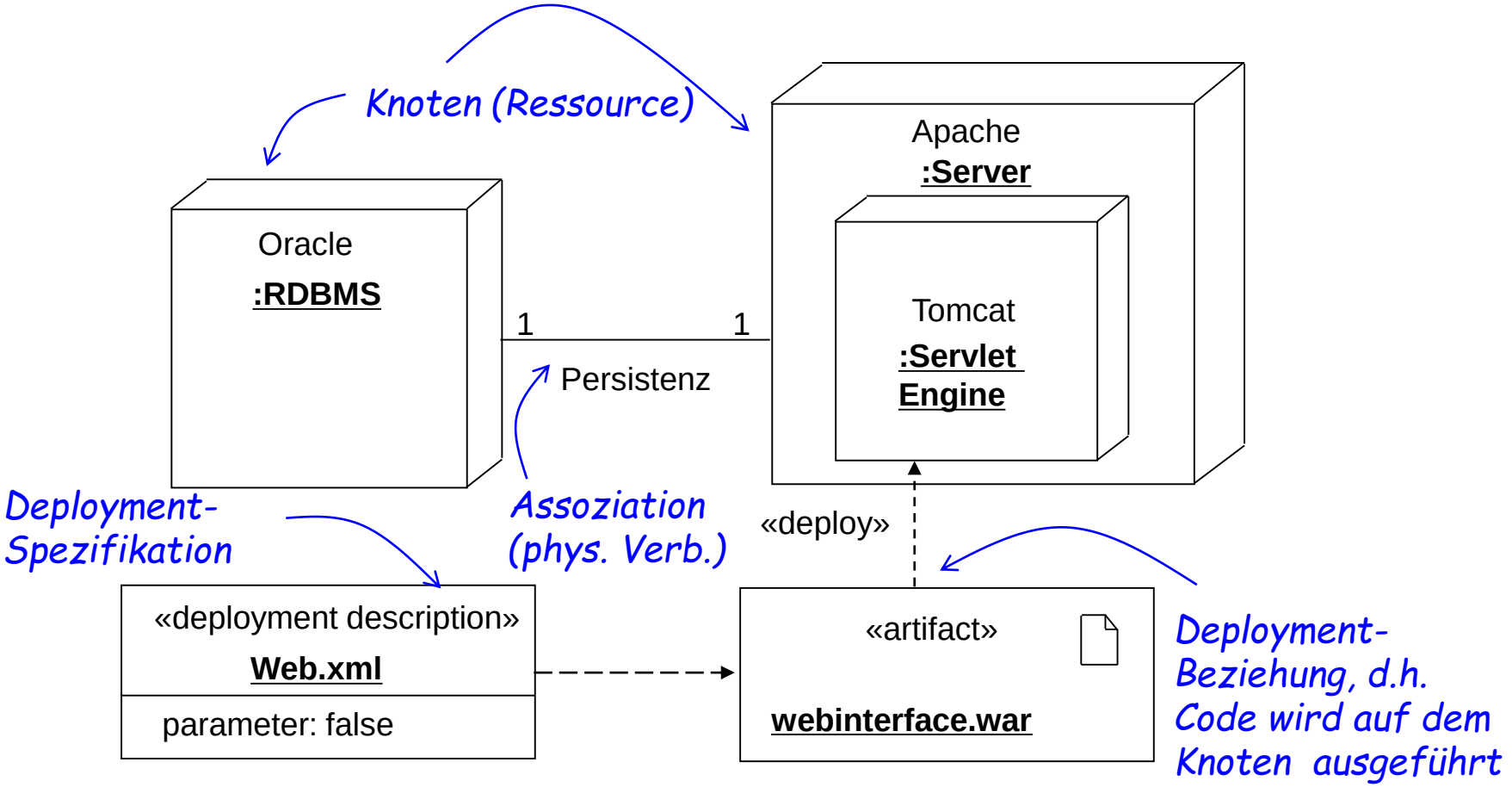
UML Paketdiagramm

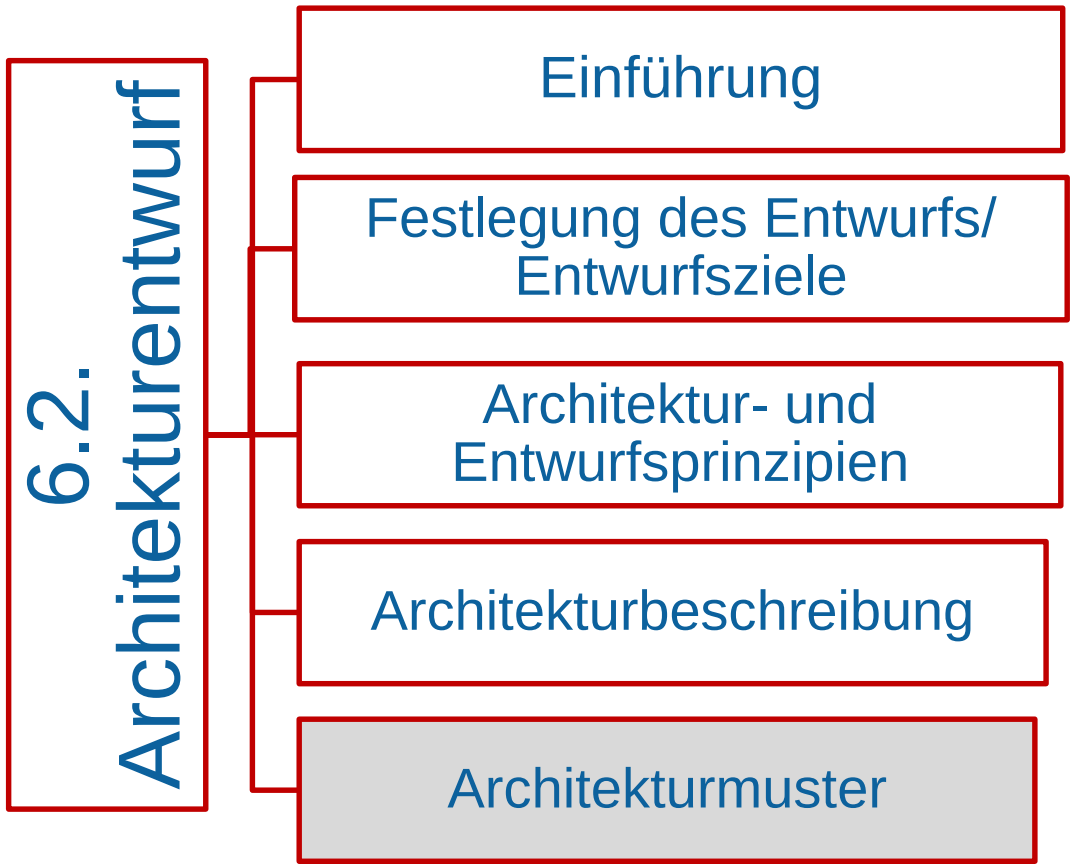


UML Komponentendiagramm



UML Verteilungsdiagramm





Architekturmuster

- Architekturmuster dienen (wie die Entwurfsmuster) dazu, eine bewährte Lösung für eine Klasse verwandter Probleme schematisch zu beschreiben

- Sind zu unterscheiden nach der entsprechenden Sicht
  - Komponentensicht
  - Physische Sicht
  - Laufzeitsicht

Wichtige Beispiele

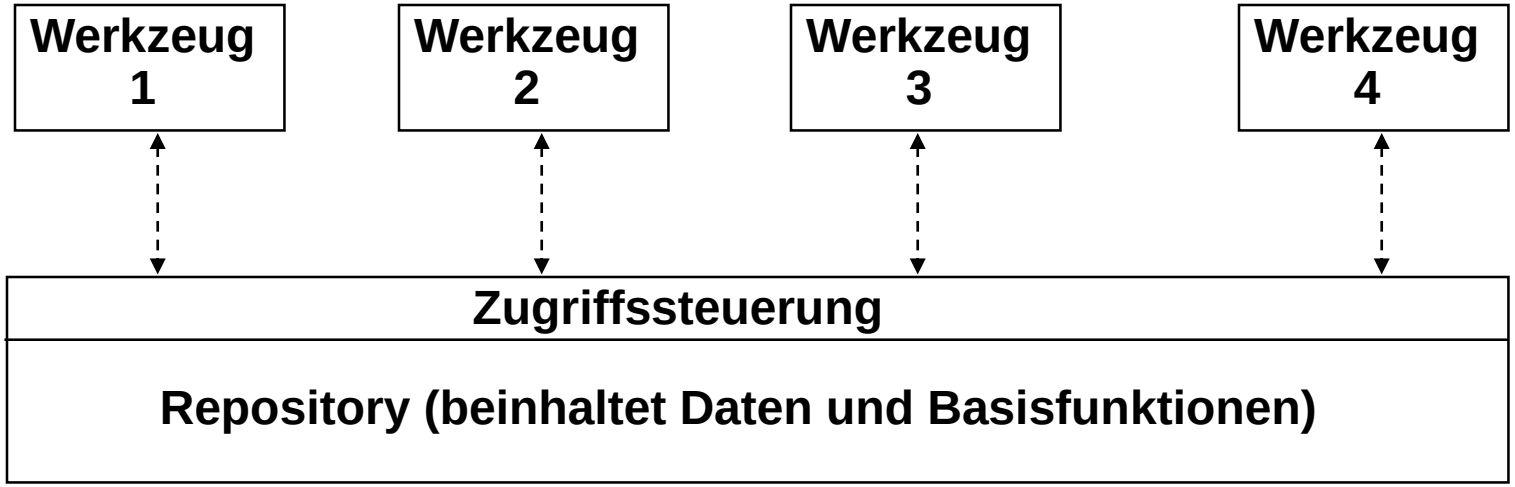
- Komponentensicht
  - Repository
  - Schichten
  - Model-View-Controller
- Physische Sicht (Verteilungsmuster)
  - Client/Server:
    - Two-Tier
    - Three-Tier
- Laufzeitsicht (Steuerungsmuster)
  - Zentrale Steuerung
    - Call-Return
    - Master-Slave

Architekturmuster Repository (1)

- **Problem:**
  - Vorhandene Menge an Basisfunktionen um Daten zu bearbeiten
  - Basisfunktionen sollen über verschiedene Wege aufrufbar sein
  - Zugriff auf Basisfunktionen und Daten soll gesichert sein

- **Lösung:**
  - Einführung einer zentralen Stelle für die Basisfunktion und Daten
  - Zugriff auf zentrale Stelle nur nach Zugangsüberprüfung

Architekturmuster Repository (2)



- Varianten in der Zugriffskoordination und Benachrichtigung
- Anderer Name: Blackboard
- Beispiele:
  - KI-Anwendungen, z.B. Bild- und Spracherkennung
  - Integrierte Entwicklungsumgebungen (Z.B. UNICASE)
  - Anwendungen zum Zugriff auf zentrale Datenbank
  - Management-Informationssysteme, Data Warehousing

Architekturmuster Repository (3)

- **Vorteile:**
  - Gesamtsicht des Systemzustands
  - Entkopplung der Werkzeuge, damit hohe Erweiterbarkeit
  - Wiederverwendung des Repository und seiner Struktur

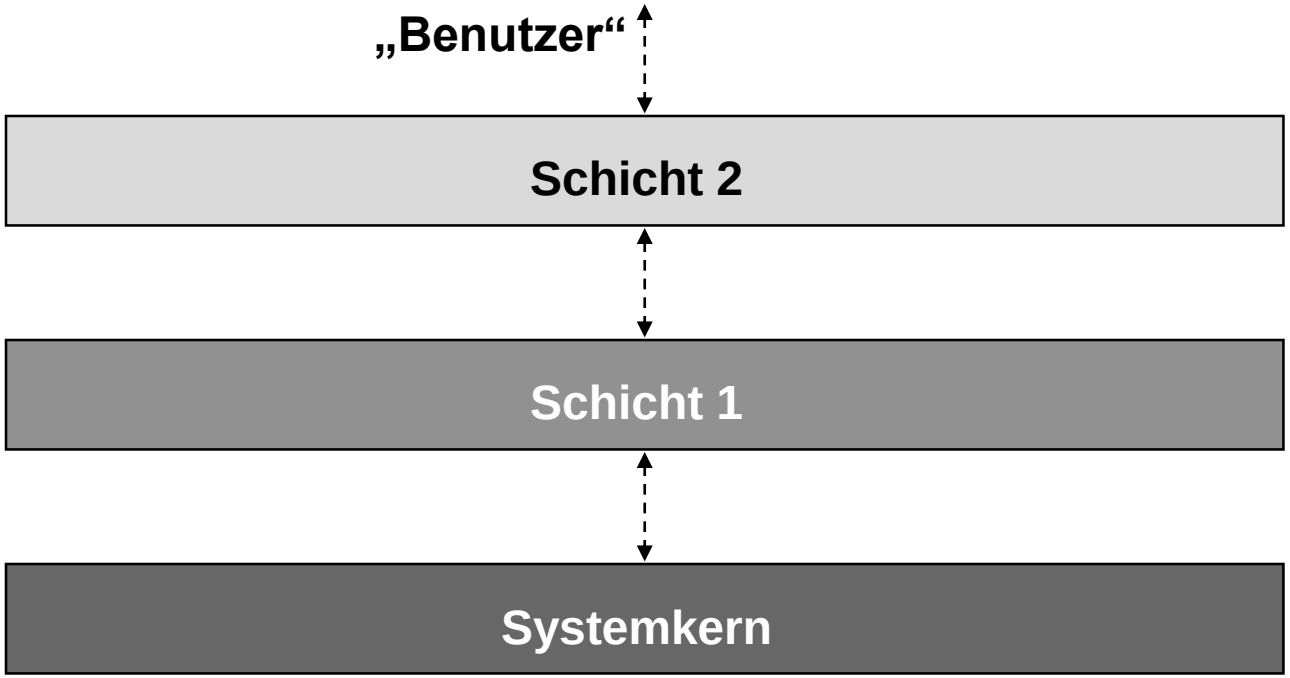
- **Nachteile:**
  - Zugriffskoordination kann zu "Flaschenhals" werden oder Inkonsistenzen verursachen
  - Einheitliche Schnittstelle für alle Werkzeuge nötig
  - Lokale Datenhaltung in den Werkzeugen birgt Gefahr der Inkonsistenz und benötigt aufwendige Benachrichtigungsmechanismen

Architekturmuster Schichten (1)

- **Problem:**
  - Einfacher Aufbau eines Systems
  - Funktionen sollen möglichst getrennt voneinander sein
  - Aufruf von Funktionen nur in eine Richtung
  - Ermöglichung der sequentiellen und parallelen Entwicklung der Komponenten eines Systems

- **Lösung:**
  - Aufbau des Systems in Schichten
  - Festlegung von fest definierten Schnittstellen

Prinzip "Schichten"



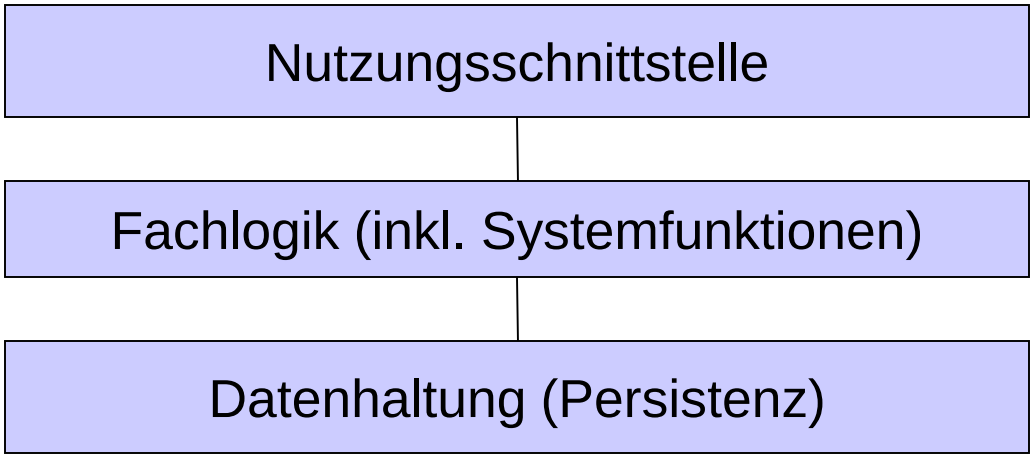
- Jede Schicht bietet Dienste (nach oben) und nutzt Dienste (von unten)
- Beispiele:
  - Kommunikationsprotokolle
  - Datenbanksysteme, Betriebssysteme

Architekturmuster Schichten (3)

- **Vorteile:**
  - Entwurf in Abstraktionsebenen
  - Erweiterungen möglich durch Einschieben von Schichten
  - Schichten-Implementierung austauschbar

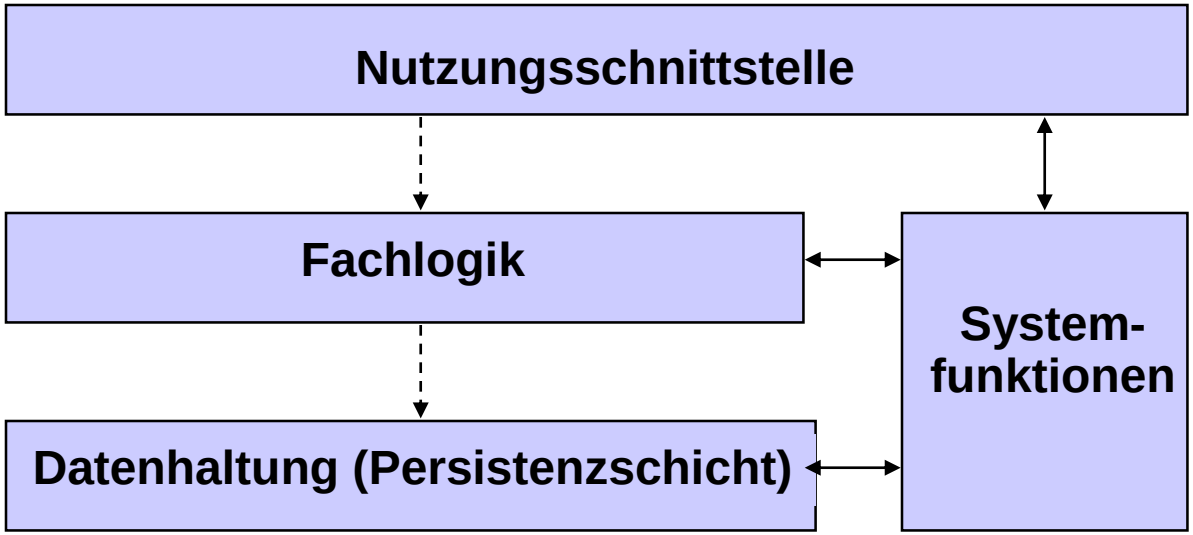
- **Nachteile:**
  - Schichtenstruktur nicht immer geeignet
  - Schichten nicht einfach zu definieren (z.B. OSI vs. TCP/IP)
  - Kopplungen über mehrere Schichten aus Effizienz-Gründen schlecht

3-Schichten-Referenzarchitektur



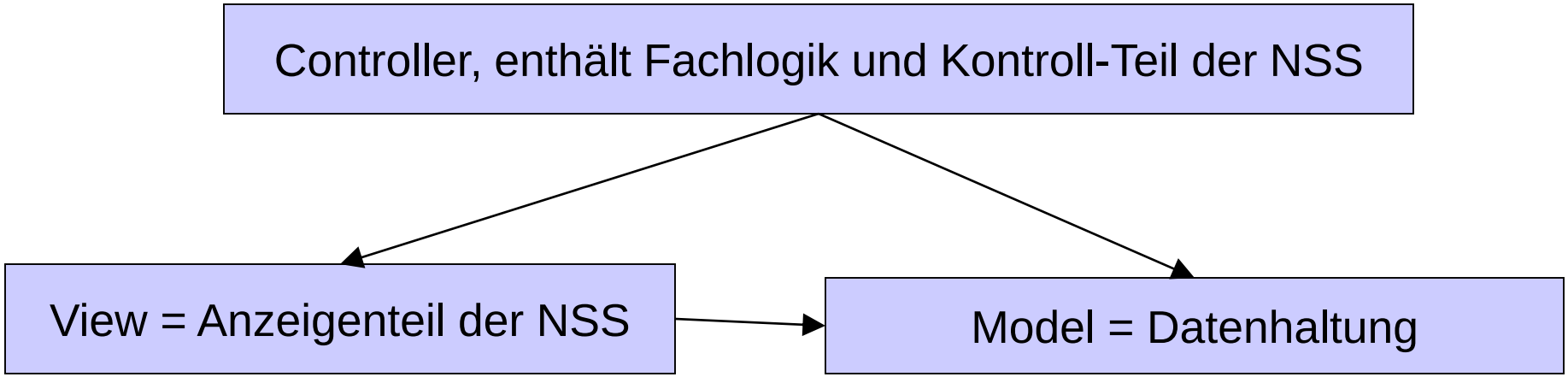
- Entwurfsregeln:
  - Nutzungsschnittstelle greift ***nie*** direkt auf Datenhaltung zu.
  - Persistenzschicht verkapselt Zugriff auf Datenhaltung, ist aber nicht identisch mit dem Mechanismus der Datenhaltung (z.B. Datenbank).
  - Fachlogik basiert auf dem Analyse-Klassenmodell
- Erlaubt das Aufsetzen von mehreren (interaktiven, batch, etc.) Nutzungsschnittstellen und den Austausch von Datenbanken

Variante: 4-Schichten-Referenzarchitektur



- Trennung von Fachlogik und grundlegenden Systemfunktionen
- Beispiele für grundlegende Systemfunktionen:
  - Verkapselung von plattformspezifischen Funktionen
  - Schnittstellen zu Fremdsystemen

Variante: Model-View-Controller



- Nutzungsschnittstelle (NSS) holt sich Daten direkt vom Model (als Observer)
- Controller übernimmt Verarbeitung der Eingaben und entscheidet, welche Datenänderungen durchzuführen
- Fachlogik manchmal auch im Modell
- Detailliertes Zusammenspiel in vielen Varianten möglich

Wichtige Beispiele

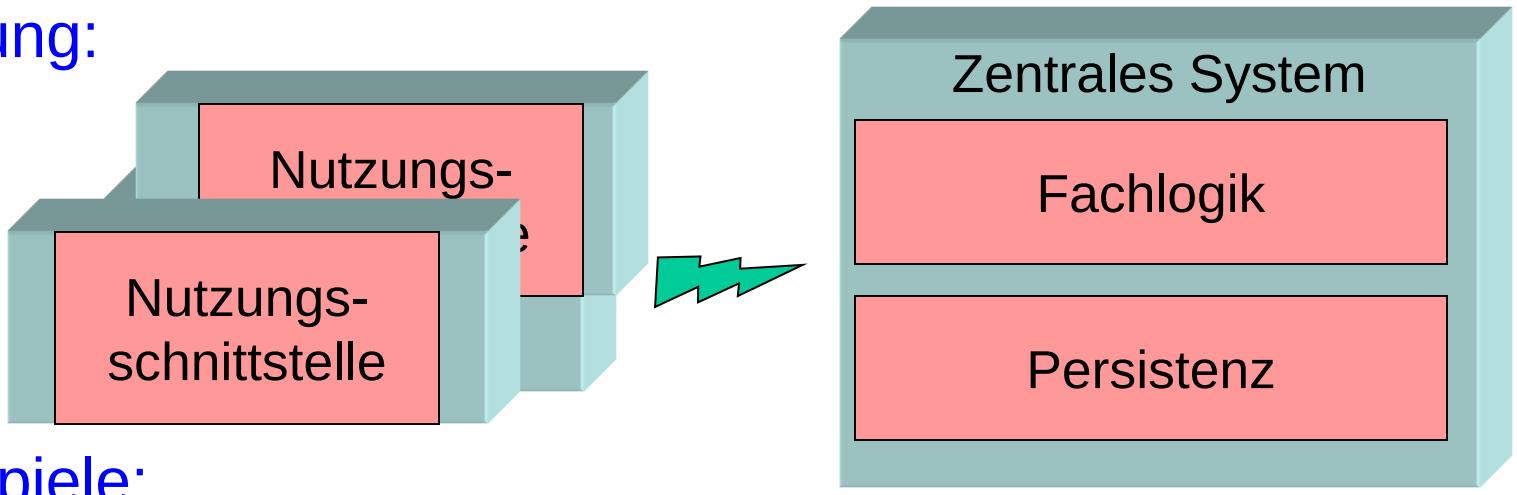
- Komponentensicht
  - Repository
  - Schichten
  - Model-View-Controller

- Physische Sicht (Verteilungsmuster)
  - Client/Server:
    - Two-Tier
    - Three-Tier

- Laufzeitsicht (Steuerungsmuster)
  - Zentrale Steuerung
    - Call-Return
    - Master-Slave

Verteilungsmuster "Zentrales System"

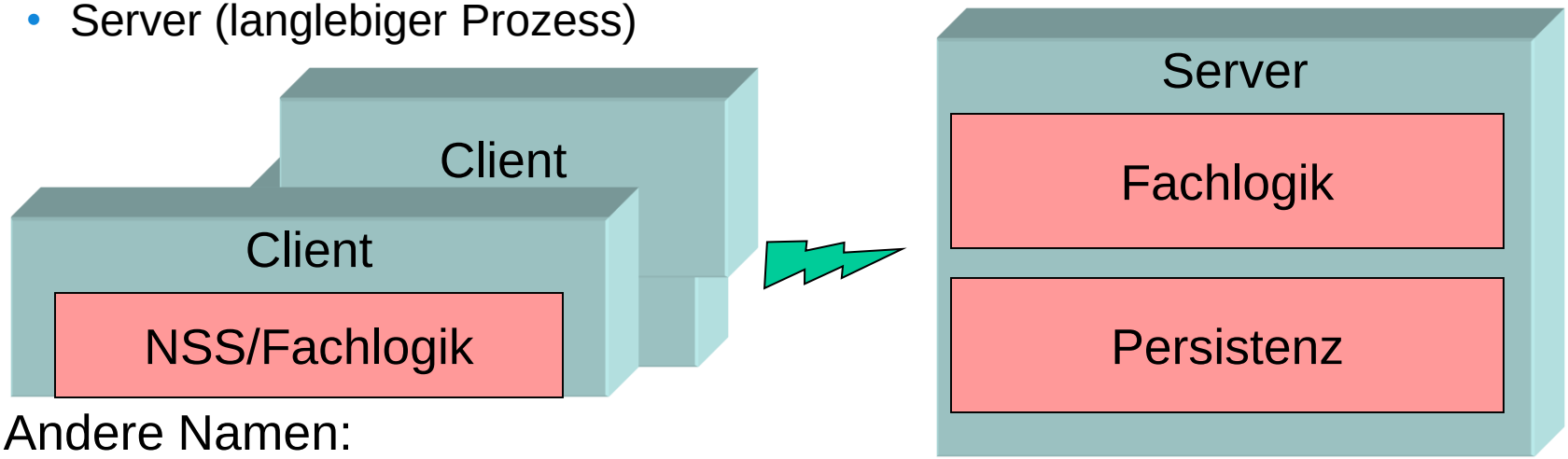
- Problem:
  - Fachlogik und Daten sollen an einem zentralen Ort liegen
  - Nur Berechtigten soll Zugriff ermöglicht werden
  - Keine wichtigen Prozesse außerhalb des zentralen Ort
- Lösung:



- Beispiele:
  - Klassische Großrechner-("Mainframe"-) Anwendungen
  - Noch einfachere Variante:
    Lokale PC-Anwendungen (identifizieren Zentrale und Terminal)

Verteilungsmuster "Client/Server"

- Problem:
  - Zentrale Datenhaltung
  - Fachlogik nicht nur zentral (wg. Flaschenhals)
- Lösung:
  - Sogenannte "Two-Tier" Client/server-Architektur
  - Client (kurzlebiger Prozess für eine Aufgabe)
  - Server (langlebiger Prozess)



- Andere Namen:
  - "Front-end" für "Client", "Back-end" für "Server"

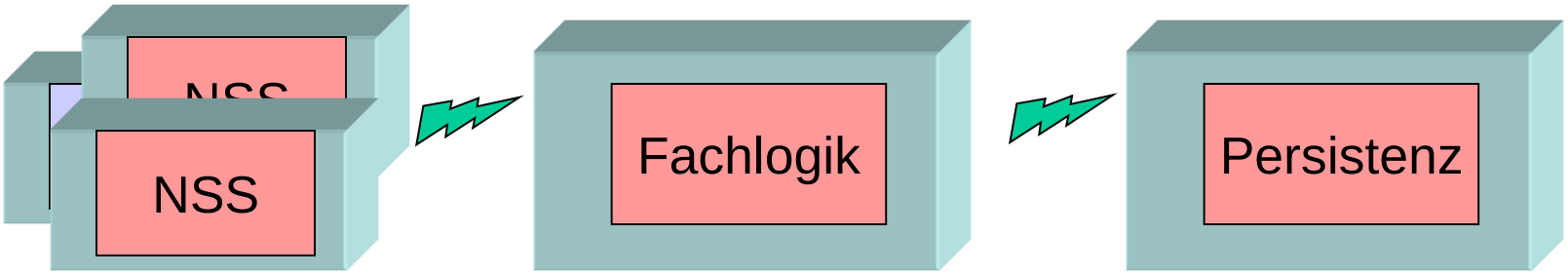
"Thin-Client" und "Fat-Client"

- Thin-Client:
  - Nur die Nutzungsschnittstelle auf dem Client-System
  - Ähnlich zu Zentralem System, aber oft Download-Mechanismen für Client-Software (einfachere Wartung)
  - Heute oft als abgespeckte Rechner

- Fat-Client:
  - Teile der Fachlogik (oder gesamte Fachlogik) auf dem Client-System
  - Hauptfunktion des Servers: Datenhaltung
  - Entlastung des Servers
  - Zusätzliche Anforderungen an Clients (z.B. Installation von Software)

Verteilungsmuster "Three-Tier Client/Server"

- Problem:
  - Trennung Fachlogik von Daten
  - Zentraler Ort für Fachlogik
  - Zentraler Ort für Daten
- Lösung:

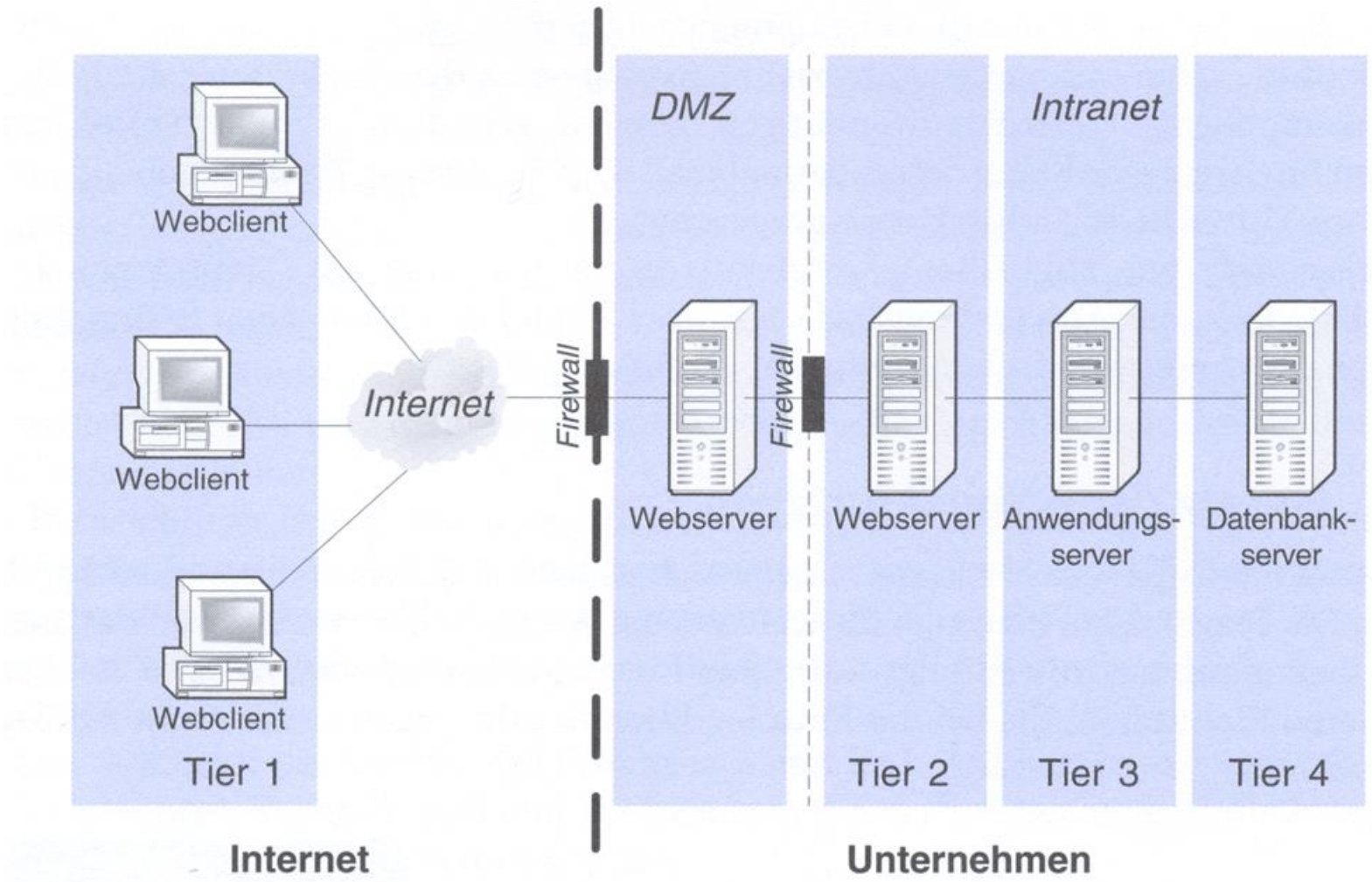


Client-Tier:
- Nutzungsschnitt-stelle
- evtl. Fachlogik

Middle-Tier (Anwendungsserver):
- Fachlogik
- evtl. Verteilung von Anfragen auf verschiedene Server

Server:
- Datenhaltung, Rechenleistung etc.
- Kommunikation unter Servern meist breitbandig.

4+Tier-Architektur



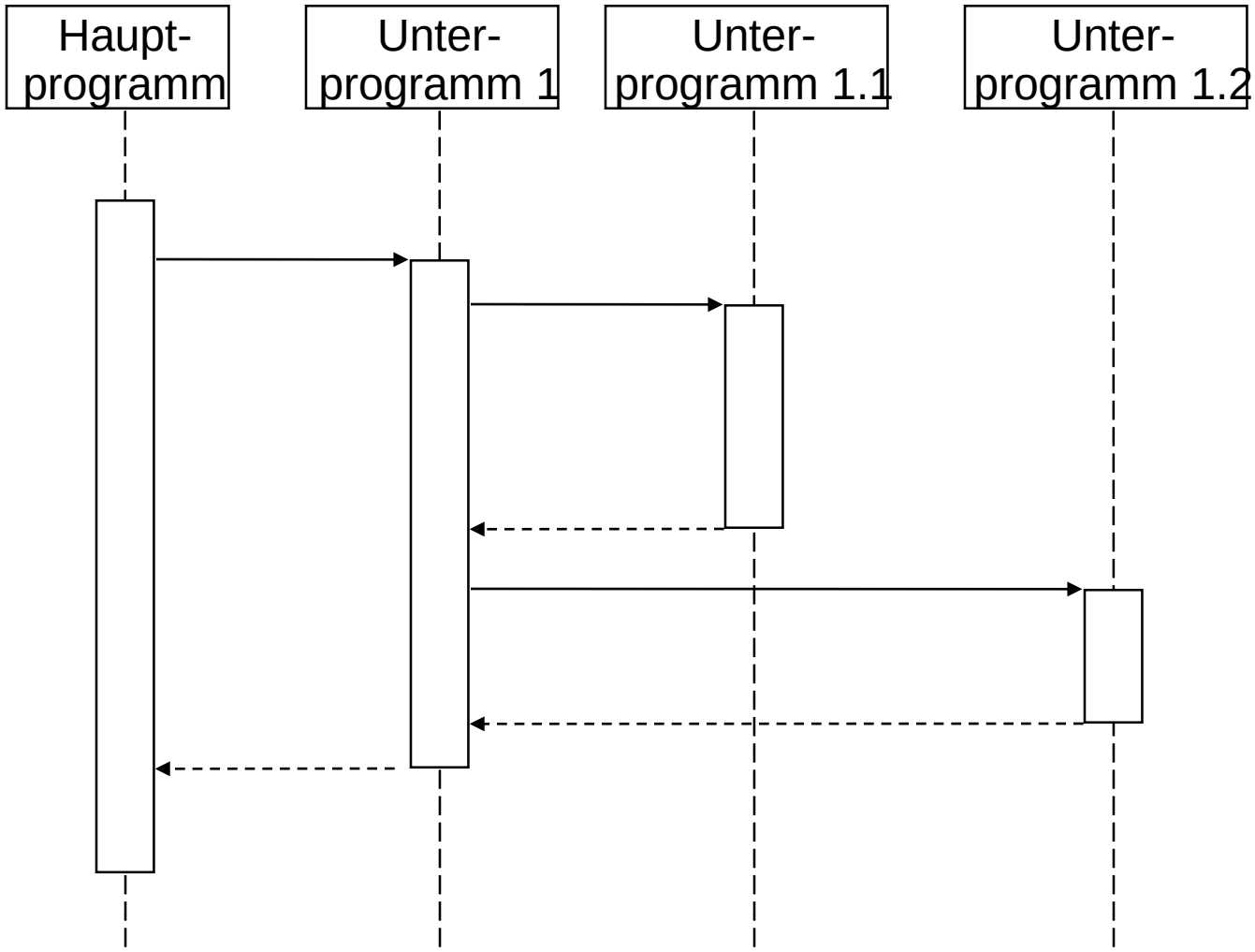
Wichtige Beispiele

- Komponentensicht
  - Repository
  - Schichten
  - Model-View-Controller
- Physische Sicht (Verteilungsmuster)
  - Client/Server:
    - Two-Tier
    - Three-Tier
- Laufzeitsicht (Steuerungsmuster)
  - Zentrale Steuerung
    - Call-Return
    - Master-Slave

Steuerungsmuster Call-Return

- **Problem:**
  - Steuerung und Aufruf von Unterprogrammen
  - Immer nur ein Unterprogramm aktiv
  - Zentrales Programm für Aufrufe
- **Lösung:**
  - Klassische Ablaufstruktur von prozeduralen Systemen
  - Basis ist ein ständig laufendes Hauptprogramm
  - Hauptprogramm ruft benötigte Unterprogramme auf
  - Während Ausführung des Unterprogramm läuft Hauptprogramm weiter (im Wartemodus)
  - Unterprogramme können weitere Unterprogramme aufrufen
- **Vorteile:**
  - Einfach und leicht verständlich
- **Nachteile:**
  - Schlechte Unterstützung von Parallelität
  - Ausnahmesituationen sind schwierig zu behandeln

Visualisierung Call-Return



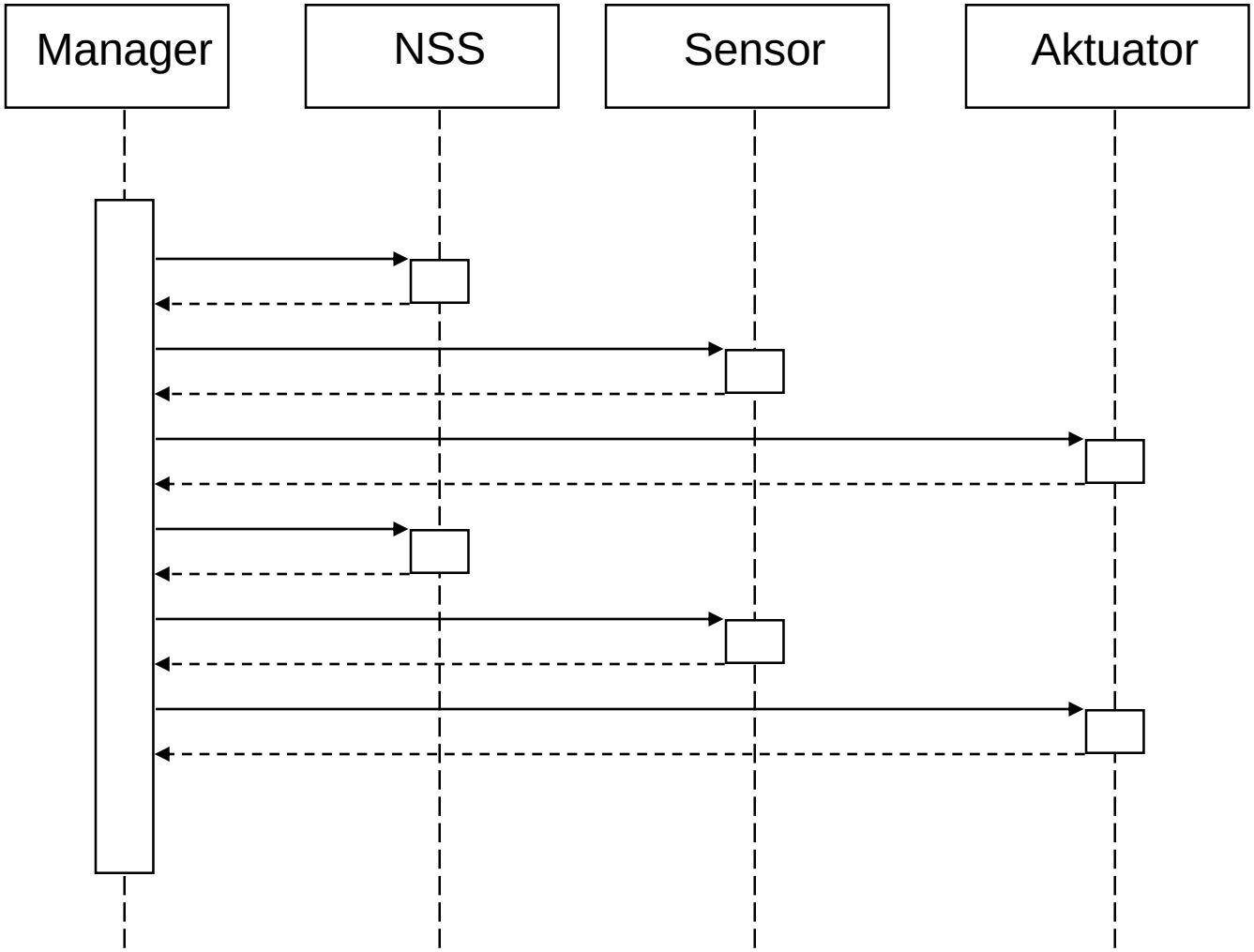
Steuerungsmuster Master-Slave

- **Problem:**
  - Zentrale periodische Abfrage von Zustandsänderungen
  - Reaktion auf Ereignisse durch zentralen Programmteil

- **Lösung:**
  - Verwendung einer zentralen Endlosschleife (Master) zur Ereignisbehandlung
  - Master prüft, ob Ereignisse vorliegen oder Zustandsänderungen erfolgt sind
    - Führt dann entsprechende Aktivitäten durch
    - Master reagiert auf entsprechende Zustandsänderungen oder Ereignisse

- **Vorteil:**
  - Gut um Realzeitsysteme zu konstruieren
    - Aber: jedes aufgerufene Programm darf bestimmte Schranke an Zeit nicht überschreiten

Visualisierung Beispiel Master-Slave



Zusammenfassung Architekturmuster

- Architekturmuster machen Erfahrungswissen für andere zugänglich.

- Architekturmuster sind eine gute Grundlage, um verschiedene Optionen zu diskutieren.

Zusammenfassung Architektur

- Architekturentwicklung basiert heute immer noch meist auf Erfahrung, es gibt wenig generelle Prinzipien

- Methoden zur klaren Definition von Architektur/Entwurfszielen (insbesondere Lösung von Konflikten zwischen Zielen) und zur Evaluation von Architekturen sind noch Forschungsthemen

Wird in Vorlesungen zu verteilten Systemen vertieft

Literatur

- Bruegge B, Dutoit A (2004) Object-oriented Software engineering, Pearson, 2004
- IEEE Std. 1471-2011, Recommended Practice for architectural description of software-intensive systems
- Posch T, Birken K, Gerdom M (2004) Basiswissen Softwarearchitketur, dpunkt Verlag
- Reussner R, Hasselbring (eds.) (2006) Handbuch der Software-Architektur, dpunkt Verlag
- Rupp Ch, Queins S, Zengler B (2012) UML 2 glasklar, Hanser Verlag
- Siedersleben J (2004) Moderne Softwarearchitektur, dpunkt Verlag
- Sommerville I (2012) Software Engineering, Addison Wesley
- Störrle H (2005) UML 2 für Studenten, Pearson Studium