universitätbonn ROOTS

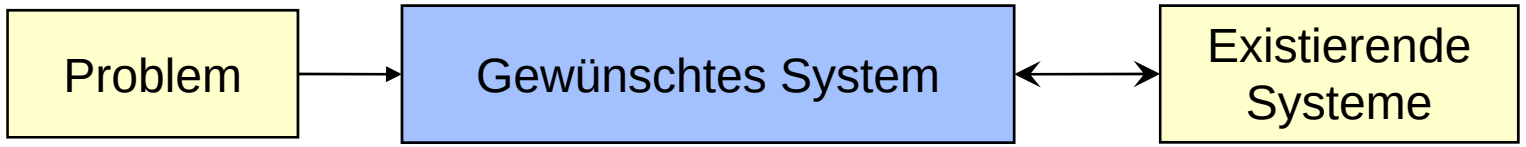
Kapitel 8
Systementwurf, Komponenten und Architekturen

Stand: 14.12.2010

Änderungen (14.12.2010):
Timing- & Deployment-Diagramme (nur Layout),
Implizite Kontrolle (Prolog-Beispiel eingefügt).

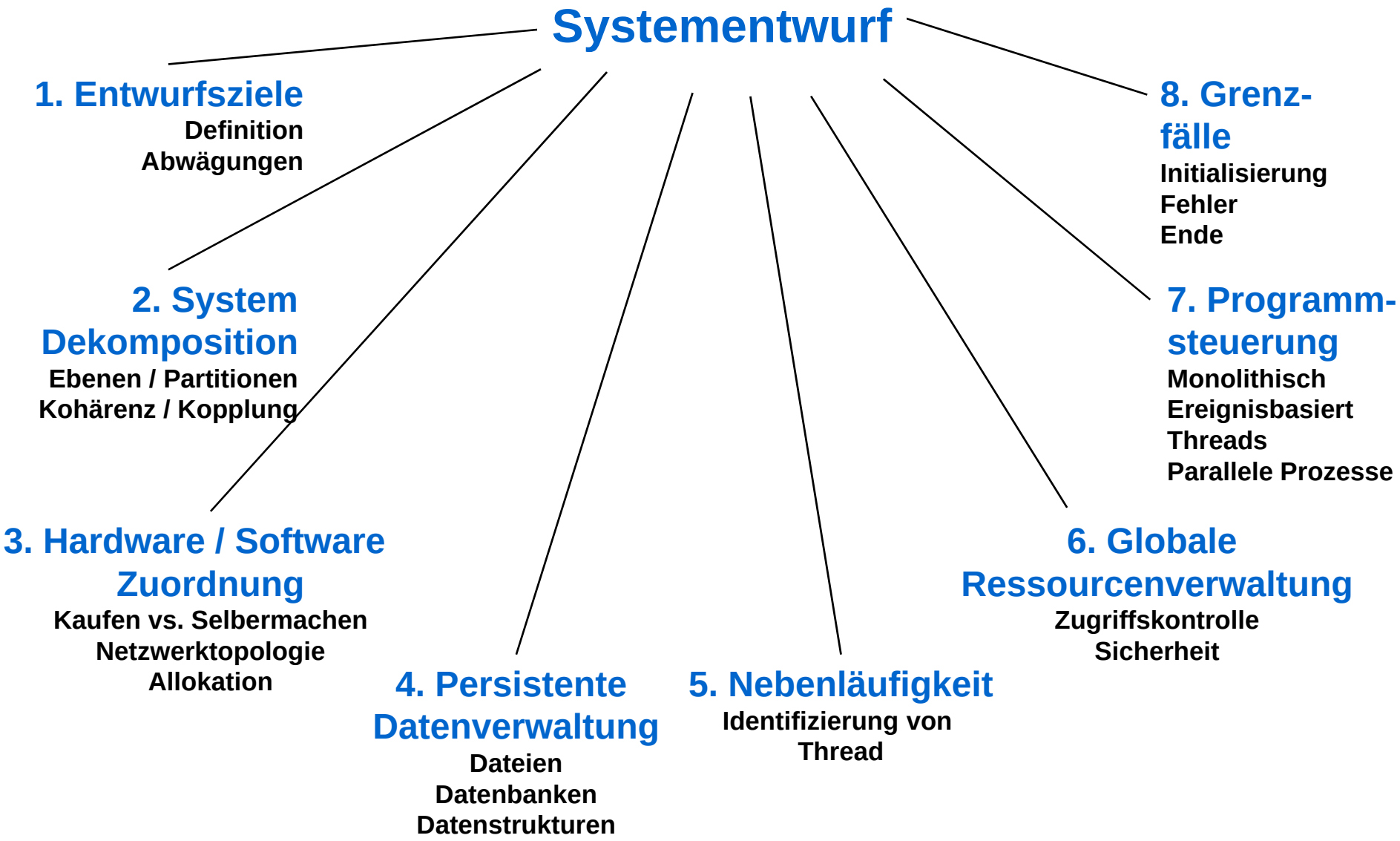
Systementwurf

- Ziel: Überbrücken der Lücke zwischen gewünschtem und existierendem System auf handhabbare Weise



- Idee: Anwendung des “Divide and Conquer”-Prinzips
  - Modellierung des neuen Systems als Menge von Subsystemen
- Folgeproblem: „Crosscutting concerns“ – Übergeordnete Belange die viele Subsysteme betreffen (z.B. Persistenz, Nebenläufigkeit, ...)
  - Erst wenn diese geklärt sind, kann man die Subsysteme unabhängig voneinander bearbeiten
- Weg: Zielbestimmung → Dekomposition → Klärung der übergeordneten Belangen
  - Danach erst Detailentwurf der Subsysteme

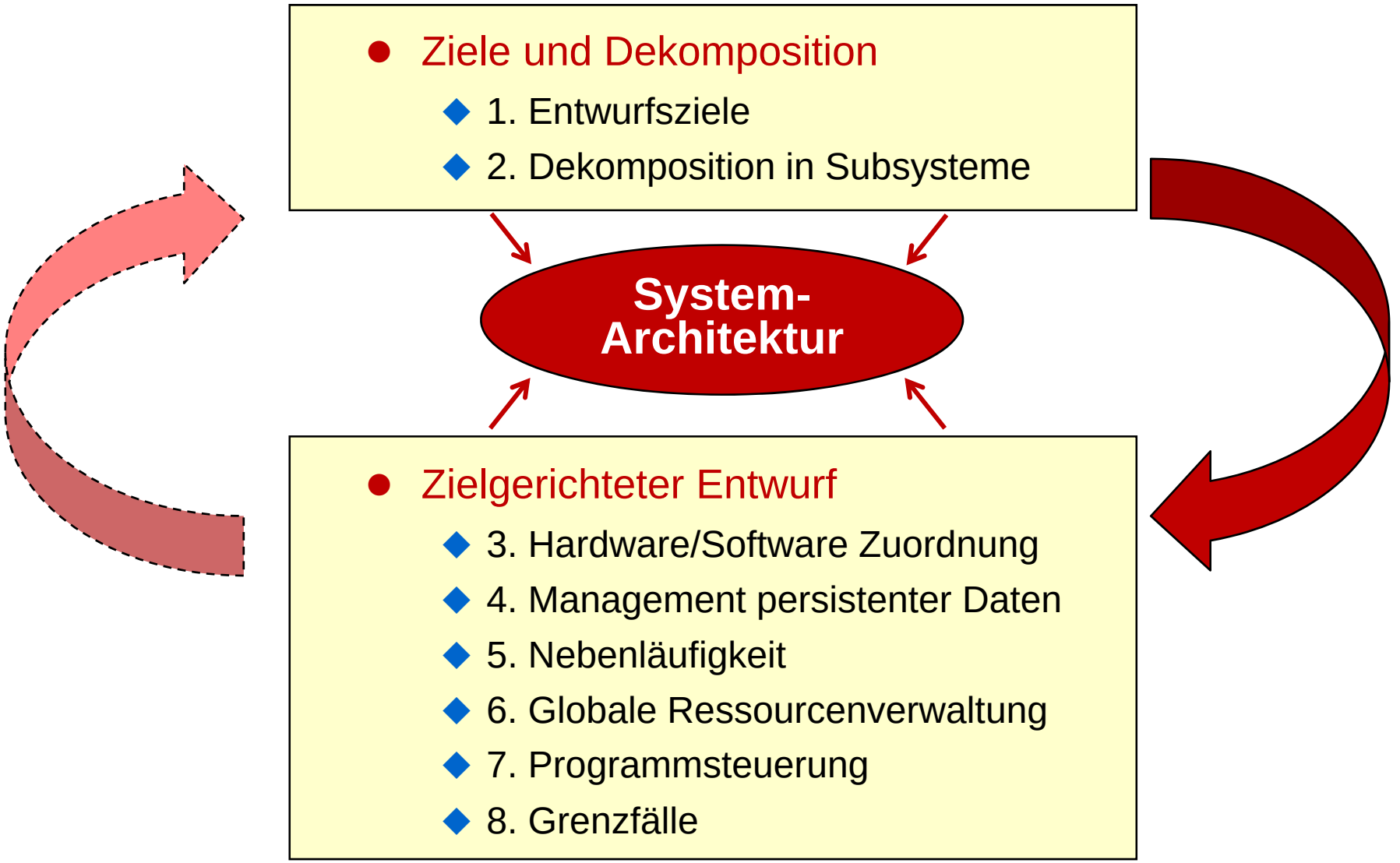
Systementwurf



Nutzung der Ergebnisse der Anforderungsanalyse für den Systementwurf

- Nichtfunktionale Anforderungen →
  - Aktivität 1: Definition der Entwurfsziele
- Use Case Modell, Objektmodell →
  - Aktivität 2: Systemdekomposition (Auswahl von Subsystemen nach funktionalen Anforderungen, Kohärenz und Kopplung)
  - Aktivität 3: Hardware/Software Zuordnung
  - Aktivität 4: Persistentes Datenmanagement
- Use Case Modell, Dynamisches Modell →
  - Aktivität 5: Nebenläufigkeit
  - Aktivität 6: Globale Ressourcenverwaltung
  - Aktivität 7: Programmsteuerung
  - Aktivität 8: Grenzfälle

Kapitel-Überblick



Ziele und Dekomposition

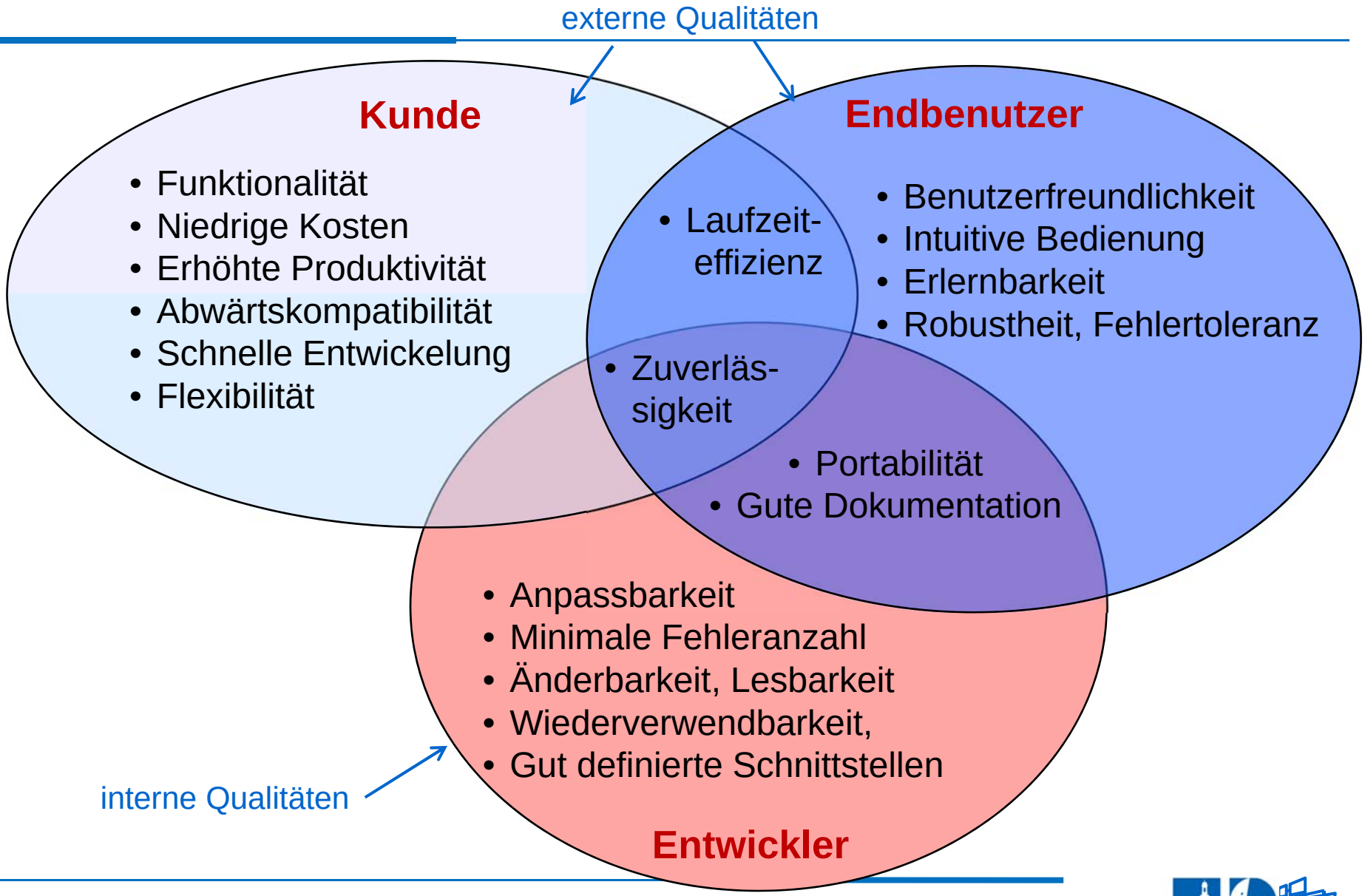
(→ Brügge & Dutoit, Kap. 6)

Entwurfsziele

Dienstidentifikation

Subsystemaufteilung

1. Entwurfsziele



Interessenskonflikte → Abwägungen

- Funktionalität vs. Benutzbarkeit
  - Je überladener, um so schwerer zu erlernen
- Funktionalität vs. schnelle Entwicklung
  - Viel Funktionalität zu implementieren braucht Zeit
- Kosten vs. Robustheit
  - Sparen an Qualitätssicherung
- Kosten vs. Wiederverwendbarkeit
  - Quick and dirty
- Effizienz vs. Portabilität
  - Effizienz durch Speziallösung für bestimmtes Betriebssystem, DBMS, ...
- Abwärtskompatibilität vs. Lesbarkeit
  - Viele Sonderfälle für Altversionen erschweren die Lesbarkeit

Bedeutung nichtfunktionaler Anforderungen für den Systementwurf

- Dilemma ▸ Zu viele Alternativen
  - Die gleiche Funktionalität ist auf verschiedenste Arten realisierbar
- Nutzen von NFA ▸ Auswahlkriterien
  - Nichtfunktionale Anforderungen dienen als Auswahlkriterien
  - Sie fokussieren die Entwurfsaktivitäten auf die relevanten Alternativen
- Beispiele (NF Anforderung → Lösungsmöglichkeiten)
  - „Hoher Durchsatz“ → Parallelität, optimistische Vorgehensweise, …
  - „Zuverlässigkeit“ → Einfache GUIs, Redundanz, …

2. Subsystem-Dekomposition

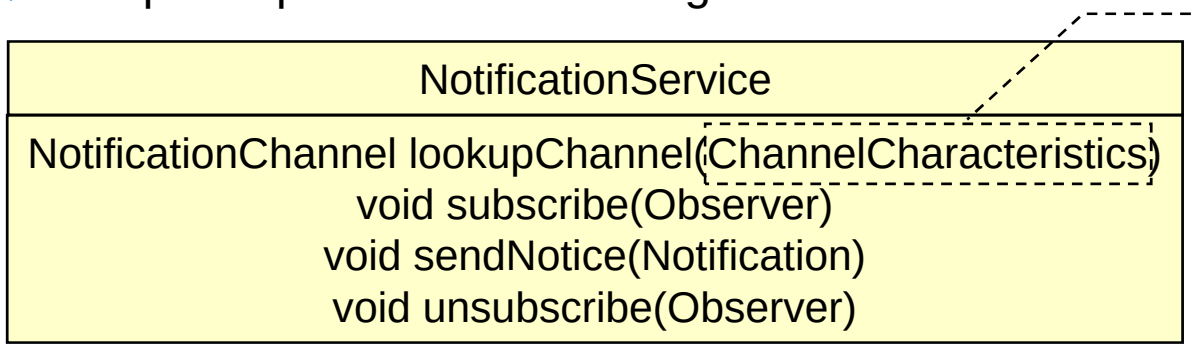
- Erster Schritt: Subsystem-Identifikation
  - Welche Dienste werden von dem Subsystemen zur Verfügung gestellt (Subsystem-Interface)?
  - →1. Gruppiere Operationen zu Diensten
  - →2. Gruppiere Typen die einen Dienst realisieren zu Subsystemen

- Zweiter Schritt: Subsystem-Anordnung
  - Wie kann die Menge von Subsystemen strukturiert werden?
  - Wie interagieren sie?
    - ⇨ Nutzt ein Subsystem einseitig den Dienst eines anderen?
    - ⇨ Welche der Subsysteme nutzen gegenseitig die Dienste der anderen?
  - → 1. Schichten und Partitionen
  - → 2. Software Architekturen

Subsystem-Identifikation: Dienste

- Dienst: Menge von Operationen mit gemeinsamem Zweck
  - Beispiel: Benachrichtigungsdienst
    - ⇨ lookupChannel(), subscribe(), sendNotice(), unsubscribe()
  - Dienste werden während des Systementwurfs identifiziert und spezifiziert

- Dienstspezifikation: Vollständig typisierte Menge von Operationen
  - In UML und Java würde das einem 'Interface' entsprechen
  - Beispiel: Spezifikation des obigem Dienstes



Entscheidung: Suche nach Channel der bestimmte Fähigkeiten hat soll möglich sein.

Alternative: Suche anhand von Namen

  - Verwendete Schnittstellen (Observer, ...) müssen natürlich auch spezifiziert werden

Subsystem-Identifikation: Subsystem-Schnittstelle

- Besteht aus einem oder mehreren zusammenhängenden Diensten
  - Vollständig typisiert → Parameter und Ergebnistypen
  - Zusammengehörig → Dienen gemeinsam einem bestimmten Zweck bzw. sinngemäß verwandten Funktionen
    - ⇨ Beispiel: Druckdienst (Druckerinstallation, -suche, -anmeldung, …, Drucken, Drucken anhalten, …)
- Spezifiziert Interaktion und Informationsfluss von/zu den Grenzen des Subsystems, aber nicht innerhalb des Subsystems
- Sollte wohldefiniert und schlank sein
- Wird auch API (Application Programmer's Interface) genannt
  - API wird aber oft undifferenziert für jegliche Schnittstellen genutzt.
  - Daher lieber den genaueren Begriff "Subsystem-Interface" verwenden.

Subsystem-Identifikation: Subsysteme

- Subsystem (UML: Package)
  - Stark kohärente Menge von Klassen, Assoziationen, Operationen, Events und Nebenbedingungen die einen Dienst realisieren
  - Wenn es gute Gründe gibt kann ein Subsystem mehr als einen Dienst anbieten

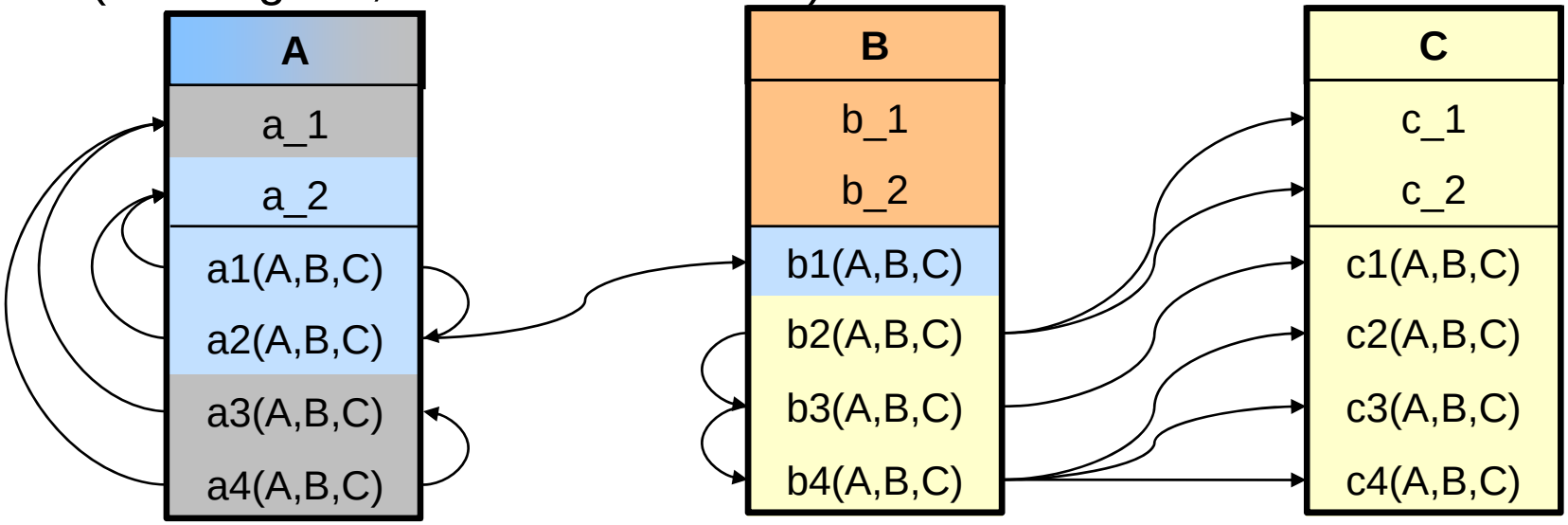
- Frage: Was ist Kohärenz?

Subsystem-Identifikation: Kopplung und Kohärenz

- Kohärenz = Maß der Abhängigkeiten innerhalb der Kapselungsgrenzen (hier: innerhalb eines Subsystems)
  - Starke Kohärenz: Die Klassen im Subsystem haben ähnliche Aufgaben und sind untereinander verknüpft (durch Assoziationen)
- Kopplung = Maß der Abhängigkeiten zwischen den Kapselungsgrenzen (hier: zwischen den Subsystemen)
  - Starker Kopplung: Modifikation eines Subsytems hat gravierende Auswirkungen auf die anderen (Wechsel des Modells, breite Neukompilierung, usw.)
- Ziel: Wartbarkeit
  - Die meisten Abhängigkeiten sollten innerhalb einzelner Subsysteme bestehen, nicht über die Subsystemgrenzen hinweg.
- Kriterien
  - Subsysteme sollten maximale Kohärenz und minimale Kopplung haben

Beispiel: Kopplung und Kohärenz

- Gegeben folgende drei Klassen. Die Pfeile zeigen Abhängikgeiten (Feldzugriffe, Methodeanaufrufe):



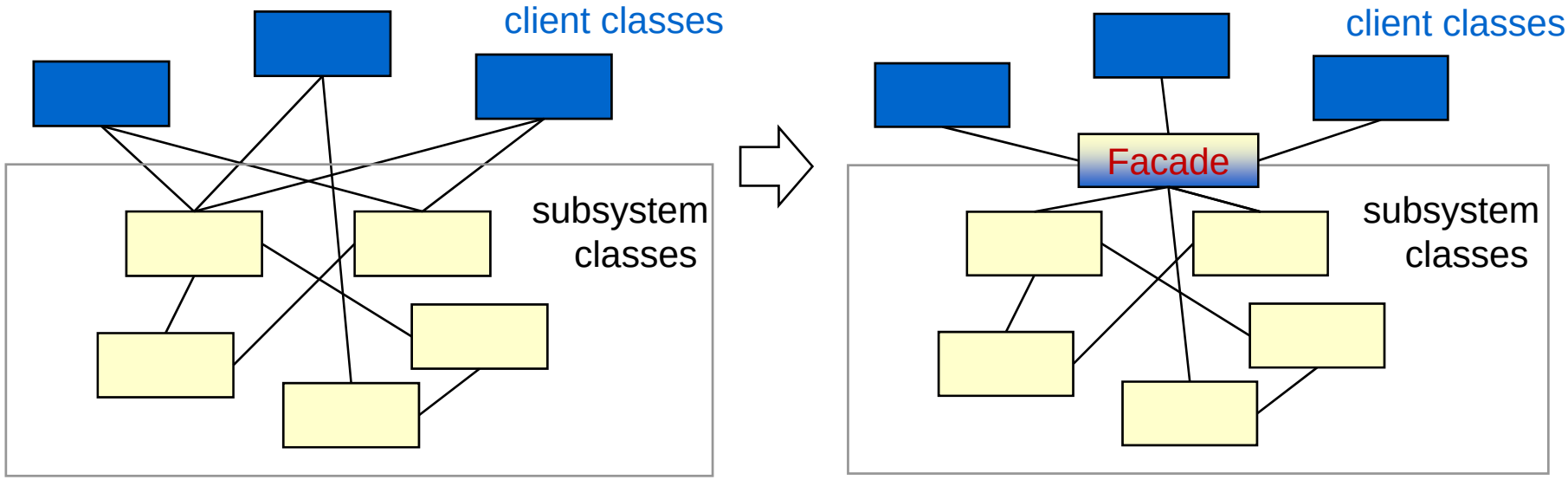
Klasse mit 2 unabhängigen Kohäsionseinheiten
→ aufsplitten in 2 Klassen!

Kaum Kohäsion.
→ b_1, b_2 ungenutzt
→b1 gehört nach A!
→ b2 - b4 gehören nach C!

Völlig unkohäsive Klasse.
B kümmert sich mehr um C-Elemente als C selbst!

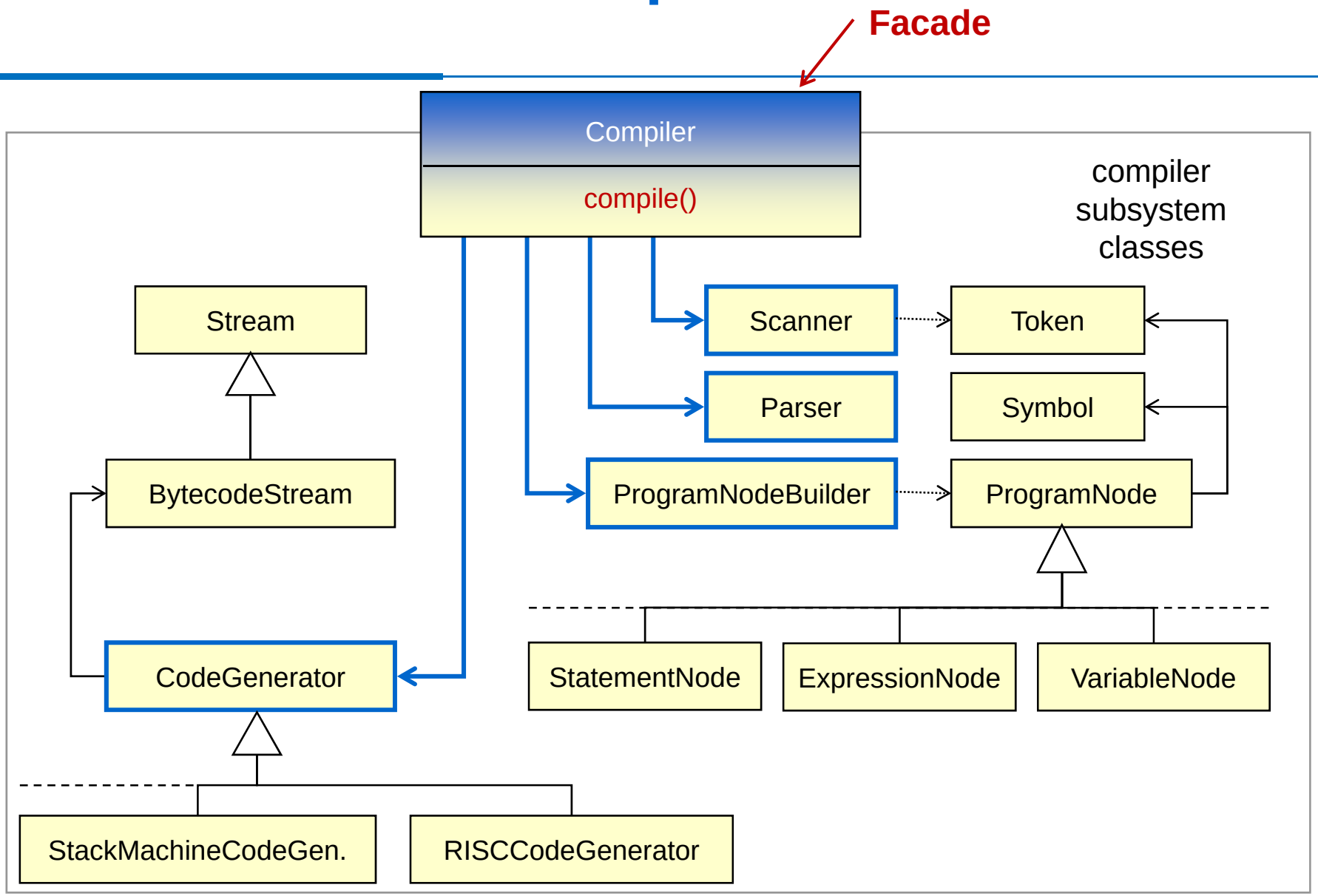
Subsystem-Implementierung mit „Façade Pattern“

- Absicht
  - Abhängigkeiten der Clients von der Struktur eines Subsystems reduzieren



- Idee: Façade = “Dienst”-Objekt
  - Funktionen einer Menge von Klassen eines Subsystems zu einem Dienst zusammenfassen
  - Objekt, das den Dienst eines Subsystems nach außen darstellt
  - Bietet alle Methoden des Dienstes
  - Vorteil: Clients müssen nichts über die Interna des Subsystems wissen

Facade Pattern: Beispiel



Facade Pattern: Anwendbarkeit

- Viele Abhängigkeiten zwischen Klassen
  - Reduzieren durch Facade-Objekte
- Einfaches Interface zu einem komplexen Subsystem
  - Einfache Dinge einfach realisierbar (aus Client-Sicht)
  - Anspruchsvolle Clients dürfen auch "hinter die Facade schauen"
    - ⇨ zB für seltene, komplexe Anpassungen des Standardverhaltens
- Hierarchische Strukturierung eines System
  - Eine Facade als Einstiegspunkt in jede Ebene

Facade Pattern: Konfigurierbarkeit

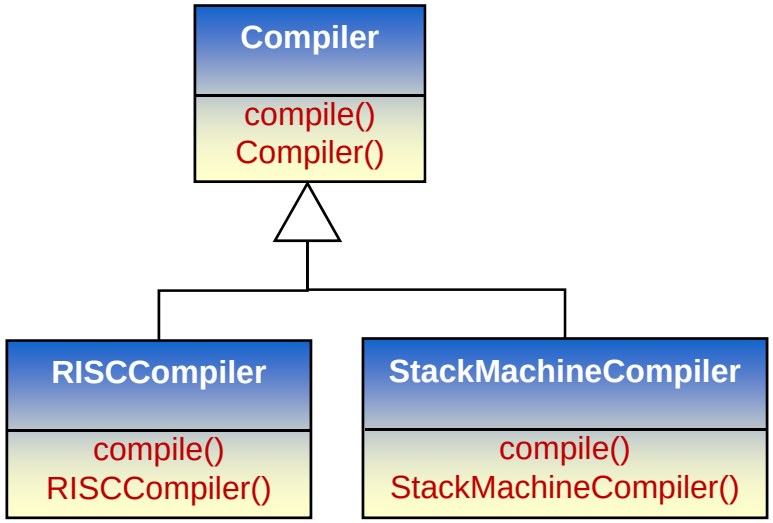
- Beispiel
  - Verwendung des „StackMachineCodeGenerator“ versus „RISCCodeGenerator“
- Realisierungsalternativen
  - Eigene Facade-Subklasse pro Konfiguration

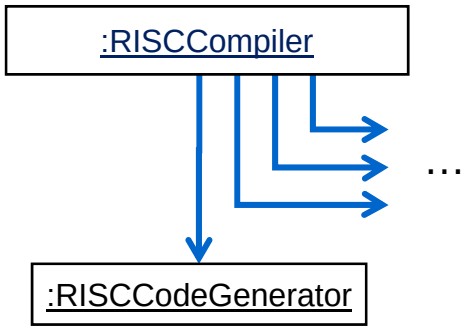
  oder
  - Nur eine Facade-Klasse deren Instanzen durch das explizite Setzen verschiedener Subsystem-Objekte konfiguriert werden

➢ Grafik hierzu siehe nächste Seite

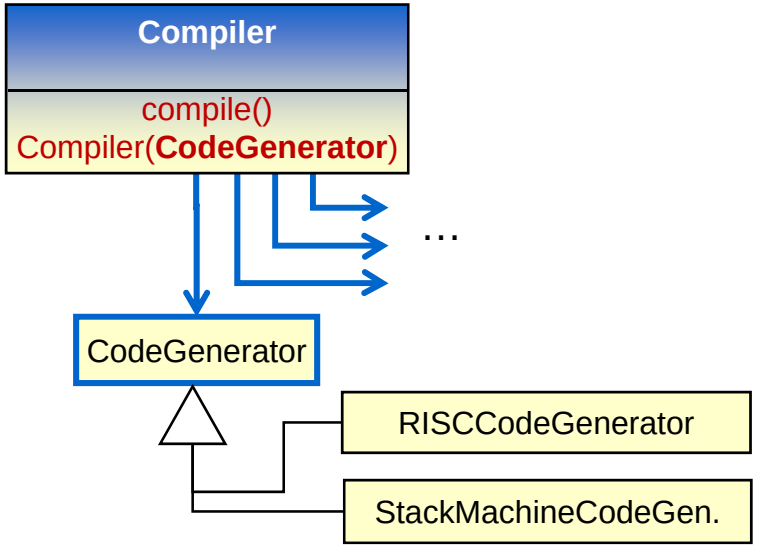
Facade Pattern: Konfigurierbarkeit

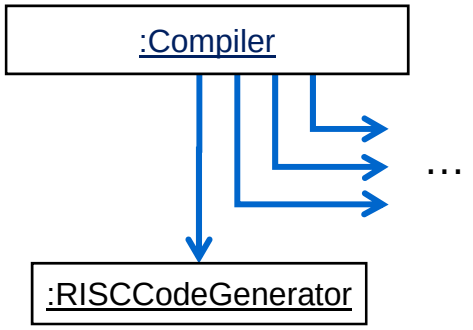
Konfiguration = Subklasse





Konfiguration = Parameter



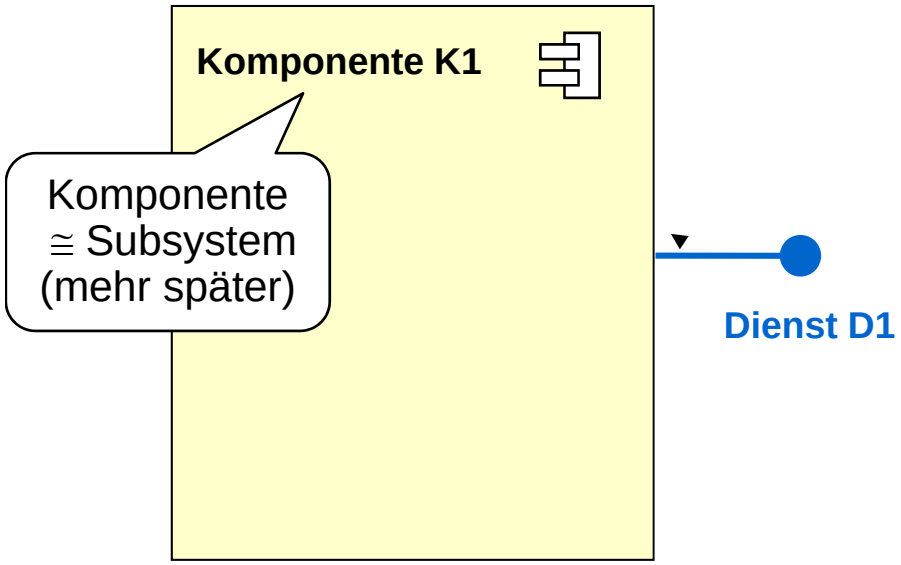


Façade als Realisierung eines Dienstes

Black-Box Sicht

Komponente bietet der Außenwelt einen Dienst

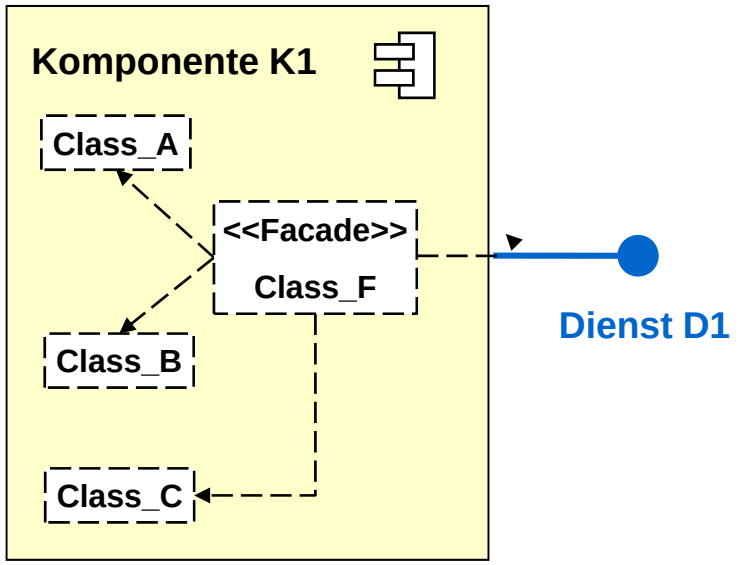
- K1 bietet D1
- Wie das geschieht ist egal



Interne Sicht

Dienst wird intern durch eine Façade implementiert

- Class_F implementiert D1 und agiert als Façade



Beispiel: Vom Analysemodell zur Systemdekomposition

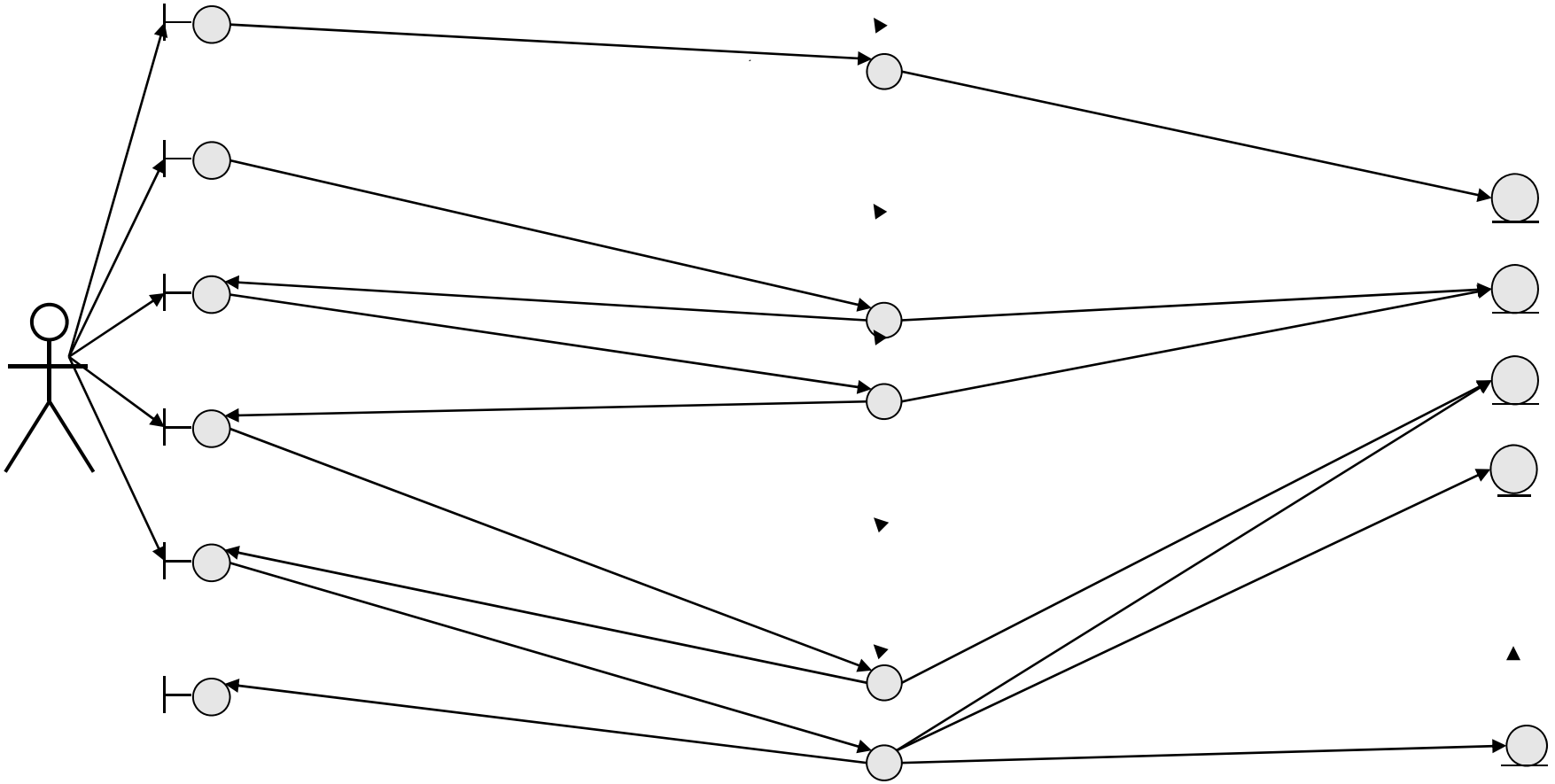
Gruppieren nach ähnlichen Funktionalitäten

Angebotene und genutzte Dienste identifizieren (Schnittstellen)

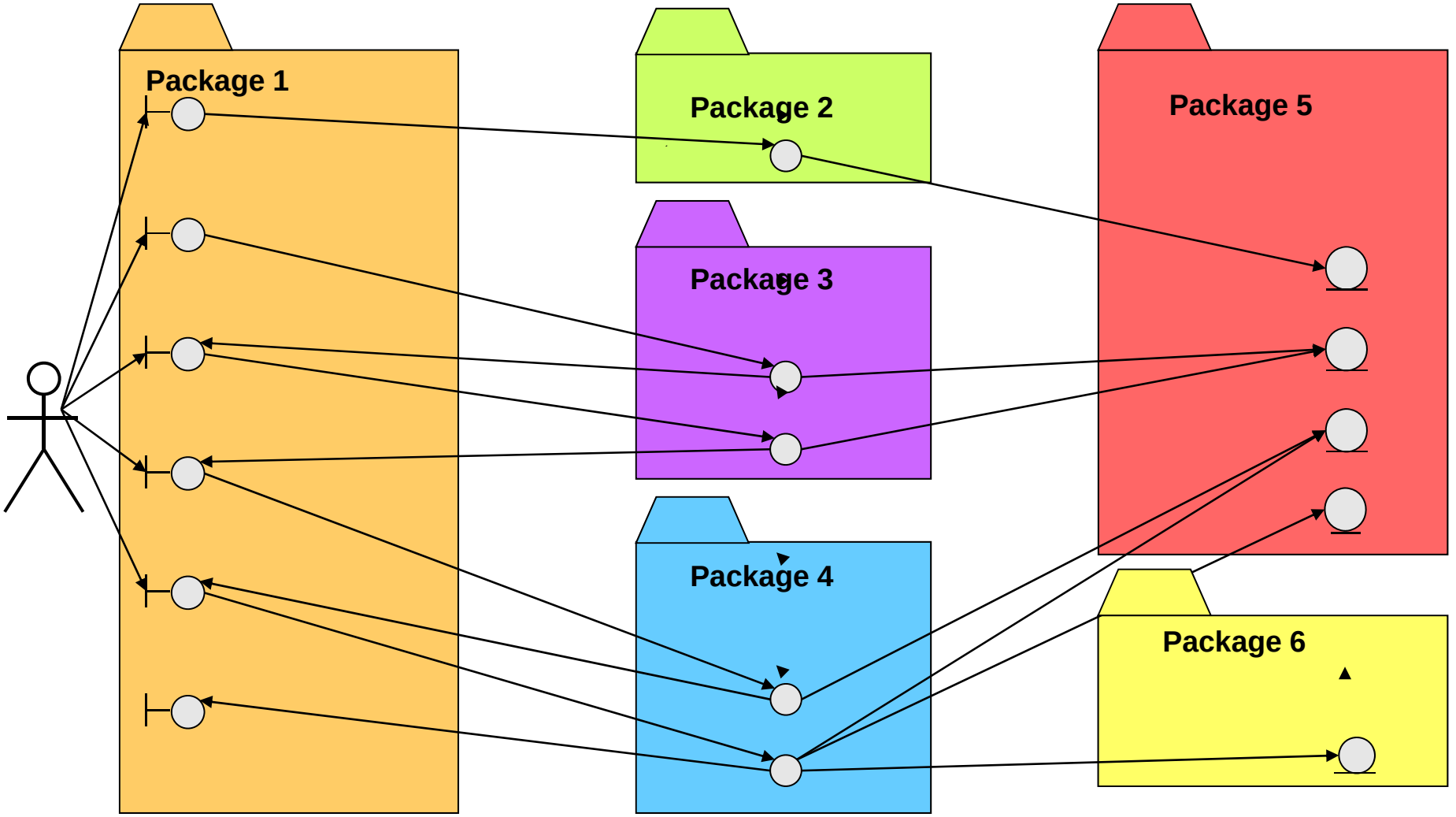
Subsysteme einführen (Komponenten)

Facades als Einstiegspunkte in die Subsysteme hinzufügen

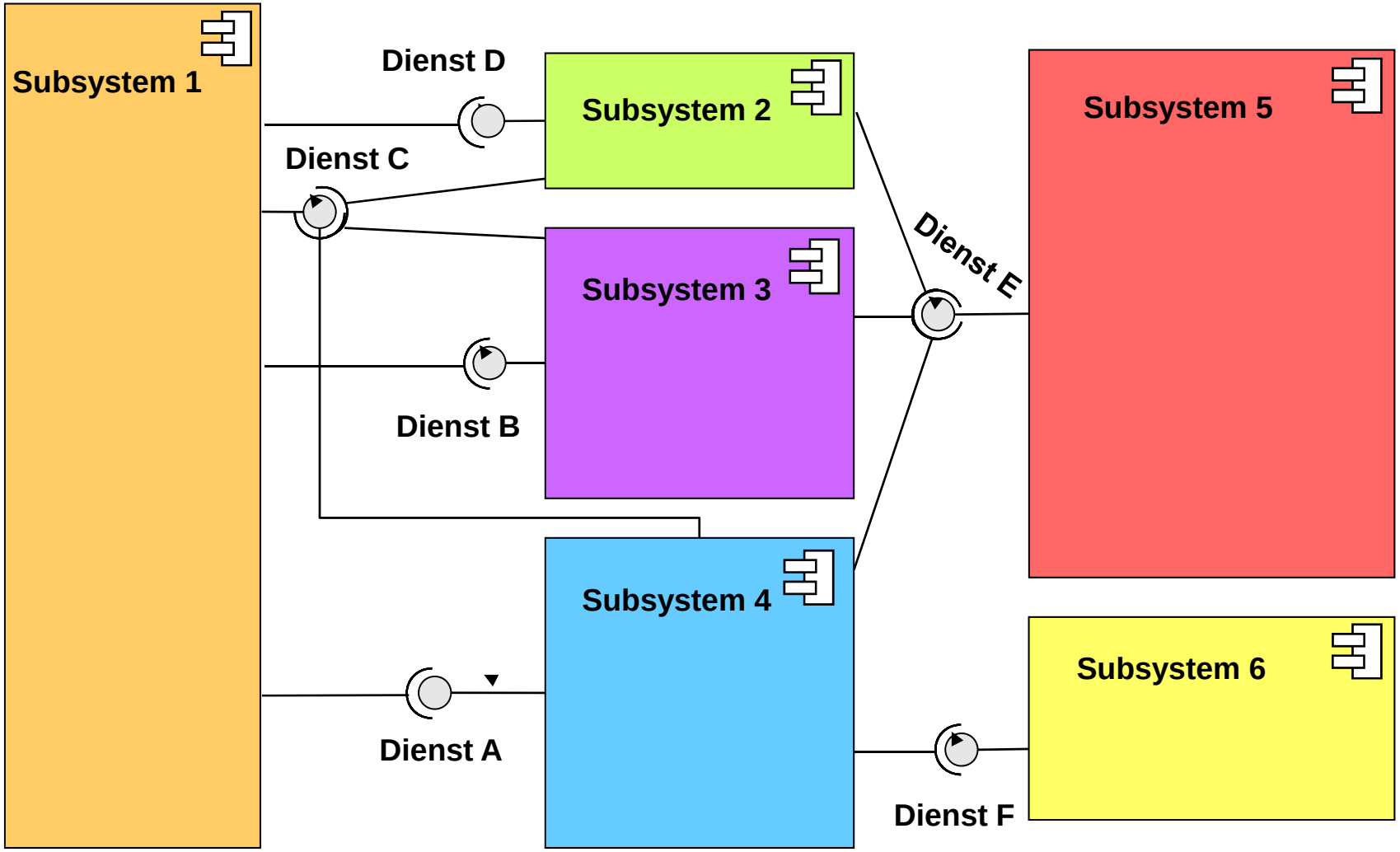
Ausgangspunkt: Objektmodell der Analyse



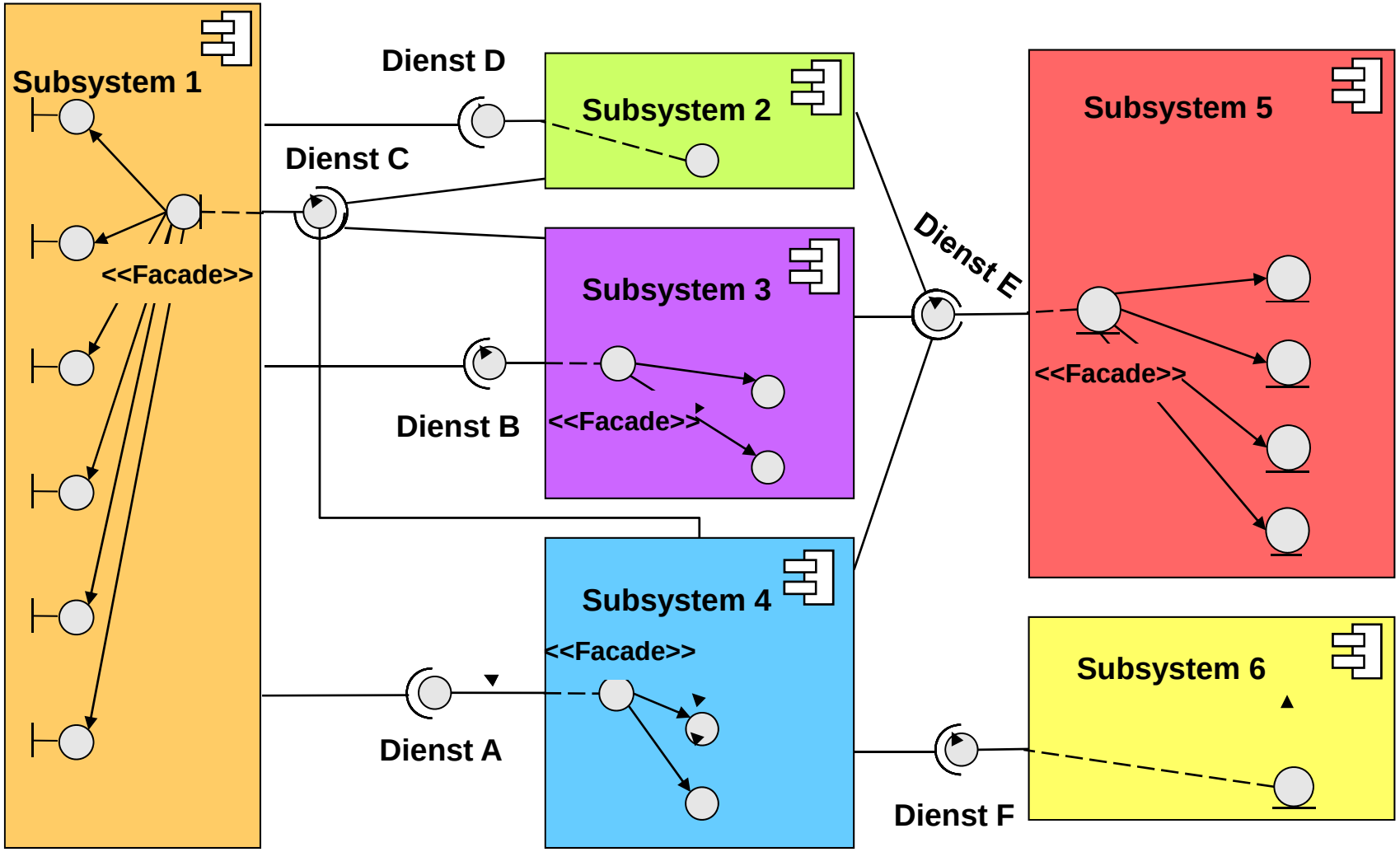
Gruppierung in Packages



System-Dekomposition: Komponenten bieten und nutzen Dienste



System-Dekomposition: Dienste-Realisierung mit Facades



Namensräume versus Subsysteme

Die zwei vorherigen Folien illustrieren, dass Subsysteme viel mehr sind als Packages

- Packages sind nur Namensräume, keine Kapselungseinheiten
  - Sie verhindern zufällige Namensgleichheit, erlauben aber Zugriff (via import-Mechanismus)
  - Sie haben keine eigene Kapselungsgrenze (keine Schnittstelle)
  - Sie reduzieren somit nicht die Abhängigkeiten (siehe vorvorherige Folien)
- Subsysteme werden als Komponenten (s. nächster Abschnitt) realisiert
  - Sie haben klar definierte Kapselungsgrenzen (Schnittstellen)
  - Sie begrenzen somit die möglichen Abhängigkeiten (da nur über die Schnittstellen zugegriffen werden kann)

Aufgabe (Diskussion mit Kollegen)

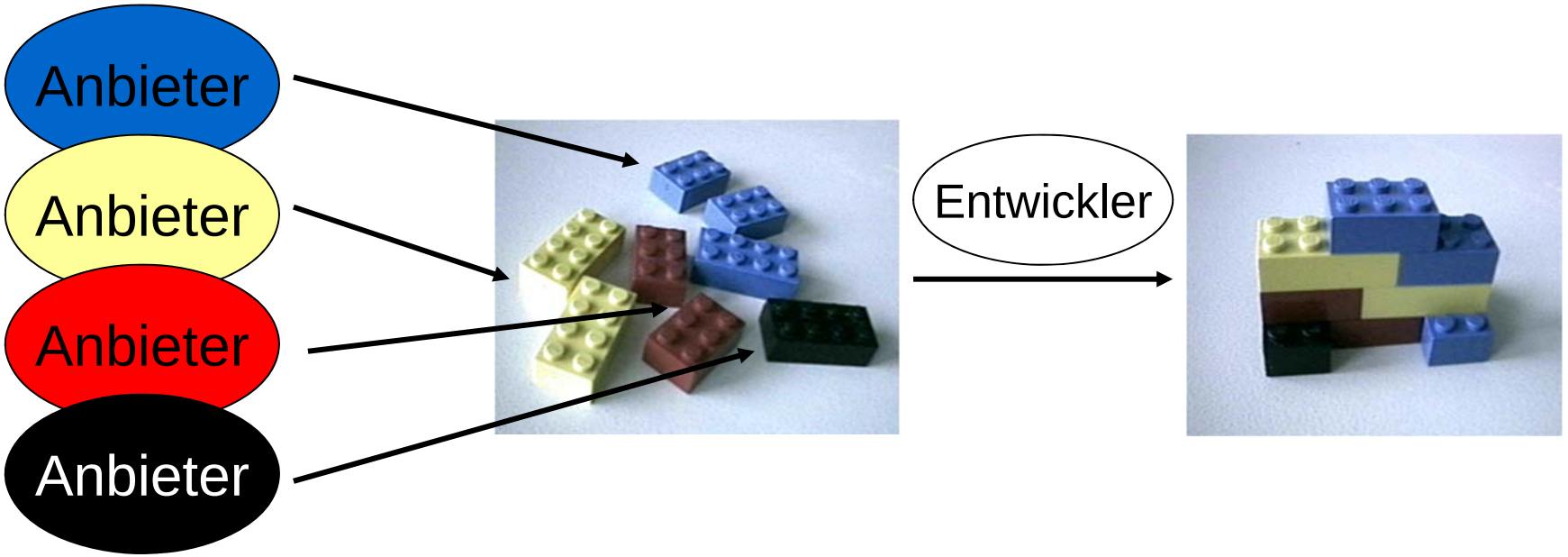
- Überlegen, Sie ob das vorherige Beispiel eine gute oder schlechte Dekomposition darstellt.
- Diskutieren Sie, was für eine Systemdekomposition „gut“ oder „schlecht“ ist.
- Kategorisieren Sie die Dekomposition aus dem Beispiel als eine der nachfolgend vorgestellten Software-Architekturen.
- Passt es genau? Brauchen Sie Änderungen damit es passt?

Patterns für Subsysteme

- Facade
  - Subsystem abschirmen (gerade vorgeführt)
- Singleton
  - Nur eine einzige Facade-Instanz erzeugen
- Proxy
  - Stellvertreter für entferntes Subsystem
- Adapter
  - Anpassung der realen an die erwartete Schnittstelle
- Bridge
  - Entkopplung der Schnittstelle von der Implementierung

Software-Komponenten

Intuitive Vorstellung



Ziele

- Plattformunabhängige Wiederverwendung von Komponenten
  - günstigere,
  - bessere und
  - schnellere Softwareentwicklung.
- Unterstützung für flexibel anpassbare Geschäftsprozesse
  - Einfach existierende Dinge zu einem neuen Verbund zusammensetzen
- Fokus auf intelligente Anwendung anstatt der wiederholten (Neu-) Entwicklung des gleichen Basisfunktionalitäten
- Märkte für Komponenten
  - Möglichkeit Komponenten von Drittanbietern zu kaufen
  - Möglichkeit Komponenten an andere zu verkaufen

Komponenten-Definition

- Clemens Szyperski , WCOP 1996
  - „Eine Softwarekomponente ist eine Kompositionseinheit mit vertraglich spezifizierten Schnittstellen und nur expliziten Kontextabhängigkeiten."
  - „Eine Softwarekomponente kann unabhängig eingesetzt werden und wird von Dritten zusammengesetzt."

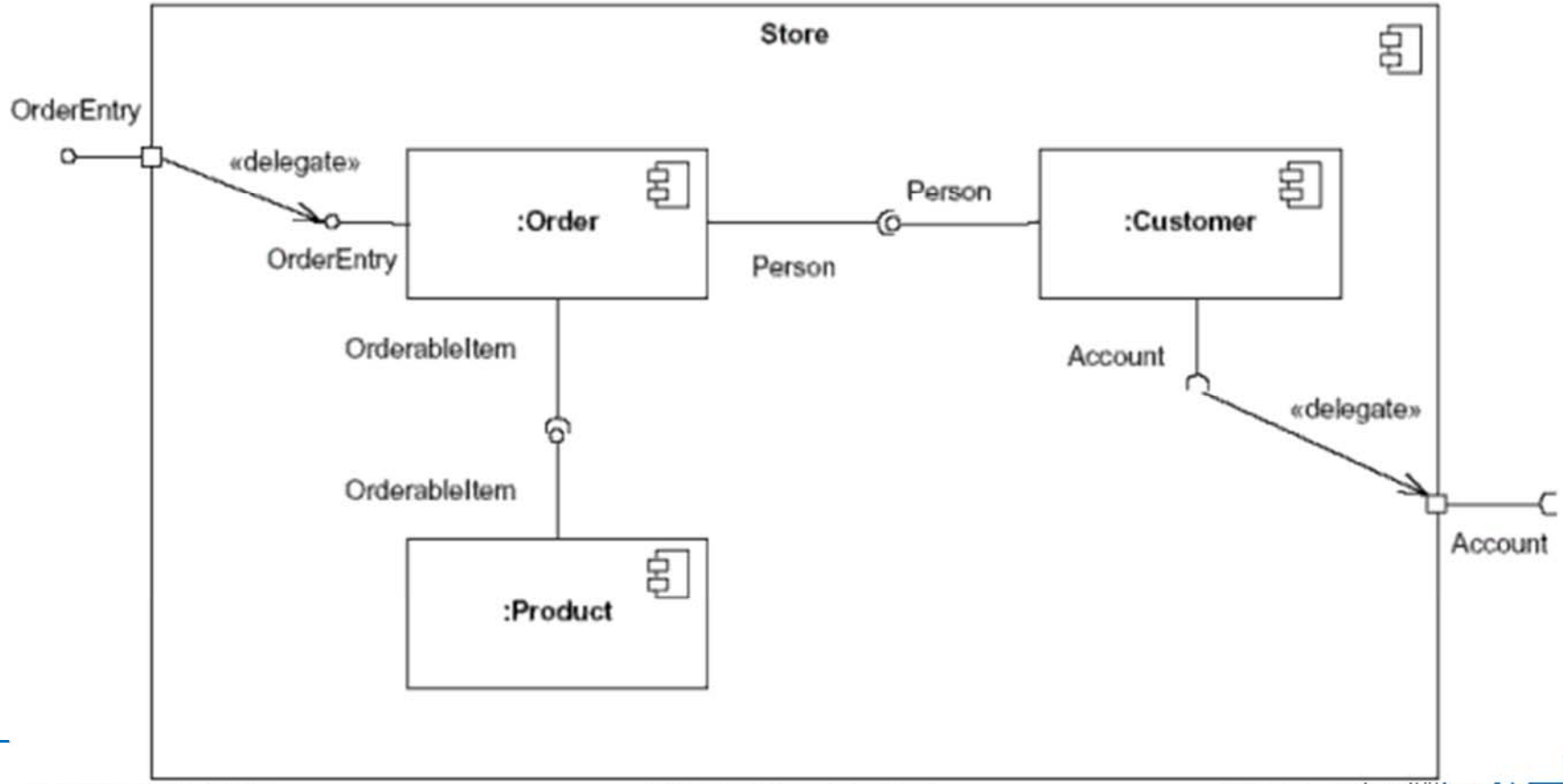
- Literatur
  - Workshop on Component-Based Programming (WCOP) 1996
  - Clemens Szyperski:
    „Component Software – Beyond Object-Oriented Programming“,
    Addison Wesley Longman, 1998.
  - Clemens Szyperski, Dominik Gruntz, Stephan Murer:
    „Component Software – Beyond Object-Oriented Programming“,
    Second Edition, Pearson Education, 2002.

Komponenten

- Kernidee → Nur explizit spezifizierte Kontextabhängigkeiten
  - Daher auch „benutzte Schnittstellen“ beschreiben!.
- Beispiel
  - Die Komponente „Bestellung“ braucht einen „Person“-Dienst um die eigenen Dienste anbieten zu können.

Komponenten: Beispiel

- Komposition
  - Die „Order“-Komponente nutzt den „Person“-Dienst von „Customer“
- Hierarchische Komponenten
  - Die „Store-Komponente besteht ihrerseits aus drei Unterkomponenten



Interaktionsspezifikation durch „Behaviour Protocoll“

- Gegeben
  - Folgende Schnittstelle

DB_Interface
open(DB_descr) : Connection close(Connection) query(Connection,SQL) : ResultSet getNext(ResultSet) : Result

- Problem
  - Wir wissen trotzdem nicht, wie das beabsichtigte Zusammenspiel der einzelnen Methoden ist.
  - Kann man die Methoden in jeder beliebigen Reihenfolge aufrufen?

- Lösung
  - Zu jedem Typ wird sein „Verhaltensprotokoll“ mit angegeben
  - Es ist ein regulärer Ausdruck der legale Aufrufsequenzen und Wiederholungen spezifiziert
- Beispiel
  - „Erst Verbindung zuer Datenbank erstellen, dann beliebig oft anfragen und in jedem Anfrageergebnis beliebig oft Teilergebnisse abfragen, dann Verbindung wieder schließen.“

```
protocoll DB_Interface_Use =
    open(DB_descr) ,
    ( query(Connection,SQL) : ResultSet ,
      ( getNext(ResultSet) : Result )*
    )* ,
    close(Connection)
```

SOFA (Software Appliances) Component Model

- SOFA Component (http://dsrg.mff.cuni.cz/sofa)
  - 1. provided and required interfaces
  - 2. frame (black-box view)
  - 3. architecture (gray-box view)
  - 4. connectors (abstract interaction)
  - 5. behavior protocols associated with 1., 2. ,3.

- Behaviour Protocol
  - incomming event (!)
  - outgoing event (?)
  - regular expression describing legal event sequences

- Example
  - !open, [!query, [!getNext]*]*, !close

- Behaviour protocols enable verification of composition

Weiterführende Literatur zu „Behaviour Protocolls“

„SOFA / DCUP“ Projekt an der Karls-Universität Prag, Prof. Plasil und Mitarbeiter

- http://dsrg.mff.cuni.cz/projects.phtml?p=sofa&q=0
- Hierachische Komponenten mit „angebotenen“ und „benutzten“ Schnittstellen
- Spezifikation von Behaviour Protocols für beide Arten von Schnittstellen
- Automatische Verifikation der Protokolleinhaltung bei der Komposition von Komponenten
  - Horizontale Verbindung von „frames“ untereinander
  - Vertikale Verbindung von „frame“ mit seiner „architecture“
- Das ganze sogar bei dynamischen Updates der Komposition (d.h. Ersetzung von Komponenten zur Laufzeit)

Übersicht über existierende Komponentenmodelle

	Modell	*IDL*	*Schnittstellen*	*Ereignisse*	*Konfiguration*	*Komposition*
Microsoft	**COM/DCOM**	ja	nur provided	durch Schnittstellen	nein	nein
	COM+	ja	nur provided	Ereignis-dienst	Kataloge	nein
	.NET	Gemeinsames Typsystem	nur provided	Nachrichten-server	Einsatz-beschreibung	nein
Java	**JavaRMI**	nein	nur provided	durch Schnittstellen	nein	nein
	JavaBeans	nein	nur provided	durch Schnittstellen	Binärdatei	nein
	EJB	nein	nur provided	durch Schnittstellen	Einsatz-beschreibung	nein
OMG	**CORBA**	ja	nur provided	Ereignis-dienst	nein	nein
	CCM	ja	provided und required	Quellen und Verbraucher	Komponenten-beschreibung	Kompositions beschreibung
	SOFA	ja	provided und required und **behaviour protocols**	Quellen und Verbraucher	Komponenten- und Einsatz-beschreibung	Kompositions beschreibung

Charakteristika Komponentenbasierter Softwareentwicklung

- Strikte Trennung zwischen Schnittstellen und Implementierung
  - Die Schnittstellenspezifikation enthält alle Informationen die ein potentieller Benutzer kennen muss. Es gibt keine anderen Kontextabhängigkeiten.
- Verfügbarkeit als Binärcode
  - Komponenten werden in ausführbarer, binärer Form für viele Plattformen geliefert. Quellcode ist nicht erforderlich.
- Plattformunabhängigkeit
  - Komponenten können auf einer Vielzahl von Rechnerumgebungen / Betriebssystemen eingesetzt werden.
- Ortstransparenz
  - Komponenten verwenden oft in Verbindung mit Middleware-Systemen eingesetzt, so dass man nicht wissen braucht, wo sich einzelne Komponenten zur Laufzeit befinden.

Charakteristika Komponentenbasierter Softwareentwicklung

- Wohldefinierter Zweck, der mehr als ein einzelnes Objekt umfasst
  - Eine Komponente ist auf ein spezifisches Problem spezialisiert
- Wiederverwendbarkeit
  - Als domänenspezifische Abstraktionen erlauben Komponenten Wiederverwendung auf Ebene von (Teil-)Anwendungen
- Kontextfreiheit
  - Die Integration von Komponenten sollte unabhängig von einschränkenden Randbedingungen sein.
- Portabilität und Sprachunabhängigkeit
  - Es sollte möglich sein, Komponenten in (fast) jeder Programmiersprache zu entwickeln.

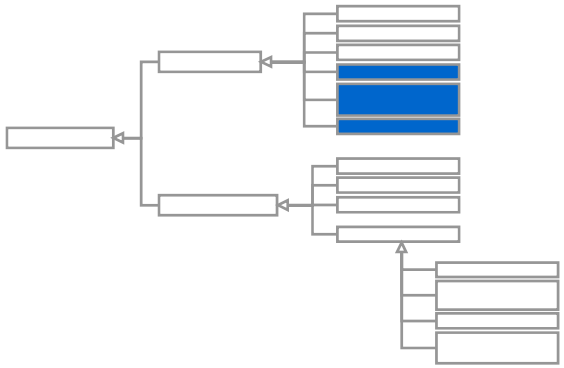
Charakteristika Komponentenbasierter Softwareentwicklung

- Reflektive Fähigkeiten
  - Komponenten sollten Reflektion unterstützen, so dass die von Ihnen angebotenen und benötigten Dienste durch Introspektion bestimmt werden können.
- Plug & Play
  - Komponenten sollten leicht einzusetzen sein.
- Konfiguration
  - Komponenten sollten parametrisierbar sein, damit sie leicht neuen Situationen angepasst werden können.
- Zuverlässigkeit
  - Komponenten sollten ausgiebig getestet werden.

Charakteristika Komponentenbasierter Softwareentwicklung

- Eignung für Integration / Komposition
  - Es sollte möglich sein, Komponenten zu komplexeren Komponenten zusammenzusetzen. Komponenten müssen miteinander interagieren können.
  - Unterstützung für visuelle Kompositionswerkzeuge

Komponentendiagramme

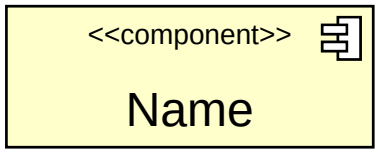


Komponentendiagramm: UML 2.0

- Komponentendiagramm zeigt Komponenten und deren Abhängigkeiten
- Komponenten sind gekapselte Teile eines Systems mit nach außen wohldefinierten Schnittstellen
  - Angebotene Schnittstellen (‚provided interfaces')
  - Benötigte/benutzte Schnittstellen (‚required interfaces')
- Komponenten kapseln beliebig komplexe Teilstrukturen
  - Klassen, Objekte, Beziehungen oder ganze Verbünde von Teilkomponenten (→ hierarchische Komposition)
  - Quellcode, Laufzeitbibliotheken, ausführbare Dateien, …
- Komponenten bieten ‚Ports'
  - Ein Port ist Name für eine Menge zusammengehöriger Schnittstellen
  - Verschiedene Ports (Namen) für mehrfach vorhandene gleiche Schnittstelle (z.B. mehrere USB-Schnittstellen am gleichen Gerät)

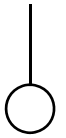
Komponentendiagramm: Elemente

- Komponente („component“)

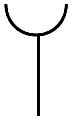


- Interface mit Stereotype

- angebotenes Interface



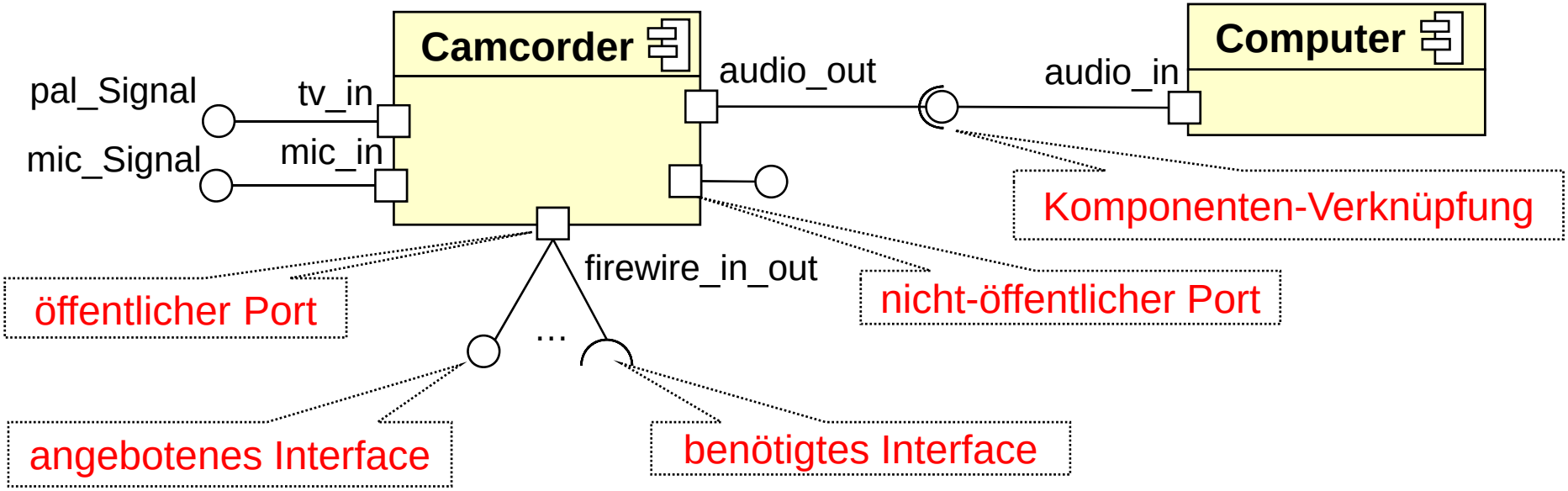
- benötigtes Interface



- Port

- Beziehung

Komponentendiagramm: Beispiel



Komponenten und Rollen (‚Parts‘)

Komponenten können in Klassendiagrammen verwendet werden und umgekehrt

- Rollen (‚Parts‘)
  - Der gleiche Typ (Interfaces oder Klasse) kann in verschiedenen Komponenten verschiedene Rollen spielen.
    - ⇨ „Das engl. Wort „Part“ heißt in diesem Kontext „Rolle“, nicht „Teil“!
  - Notation
    - ⇨ Rollen mit Multiplizität

      Rolle:Typ [Multiplizität]

      Mutiplizität
      Rolle:Typ
    - ⇨ Rolleninstanzen

      instanz/**Rolle**:Typ

      b1/**Benutzer**:Typ

universitätbonn ROOTS

Subsystem-Anordnung
→
Software-Architekturen

Subsystem Entwurf

- Erster Schritt: Subsystem-Dekomposition
  - Welche Dienste werden von dem Subsystemen zur Verfügung gestellt (Subsystem-Interface)?
  - →1. Gruppiere Operationen zu Diensten.
  - →2. Identifiziere Subsysteme als stark kohärente Menge von Klassen, Assoziationen, Operationen, Events und Nebenbedingungen die einen Dienst realisieren

- Zweiter Schritt: Subsystem-Anordnung
  - Wie kann die Menge von Subsystemen strukturiert werden?
    - ⇨ Nutzt ein Subsystem einseitig den Dienst eines anderen?
    - ⇨ Welche der Subsysteme nutzen gegenseitig die Dienste der anderen?
  - → 1. Schichten und Partitionen
  - → 2. Software Architekturen

Softwarearchitekturen

- Architektur = Subsysteme
  + Beziehungen der Subsysteme (statisch)
  + Interaktion der Subsysteme (dynamisch)

Architekturen

- Schichten-Architektur
  - Client/Server Architektur
  - N-tier
- Peer-To-Peer Architektur
- Repository Architektur
- Model/View/Controller Architektur
- Pips and Filter Architektur

Schichten und Partitionen

- Schicht (=Virtuelle Maschine)
  - Subsysteme, die Dienste für eine höhere Abstraktionsebene zur Verfügung stellt
  - Eine Schicht darf nur von tieferen Schichten abhängig sein
  - Eine Schicht weiß nichts von den darüber liegenden Schichten
- Partition
  - Subsysteme, die Dienste auf der selben Abstraktionsebene zur Verfügung stellen.
  - Subsysteme, die sich gegenseitig aufeinander neziehen
- Architekturanalysewerkzeuge
  - Identifikation von Schichten und Partitionen
  - Warnung vor Abhängigkeiten die der Schichtung entgegenlaufen

Schichten-Architekturen

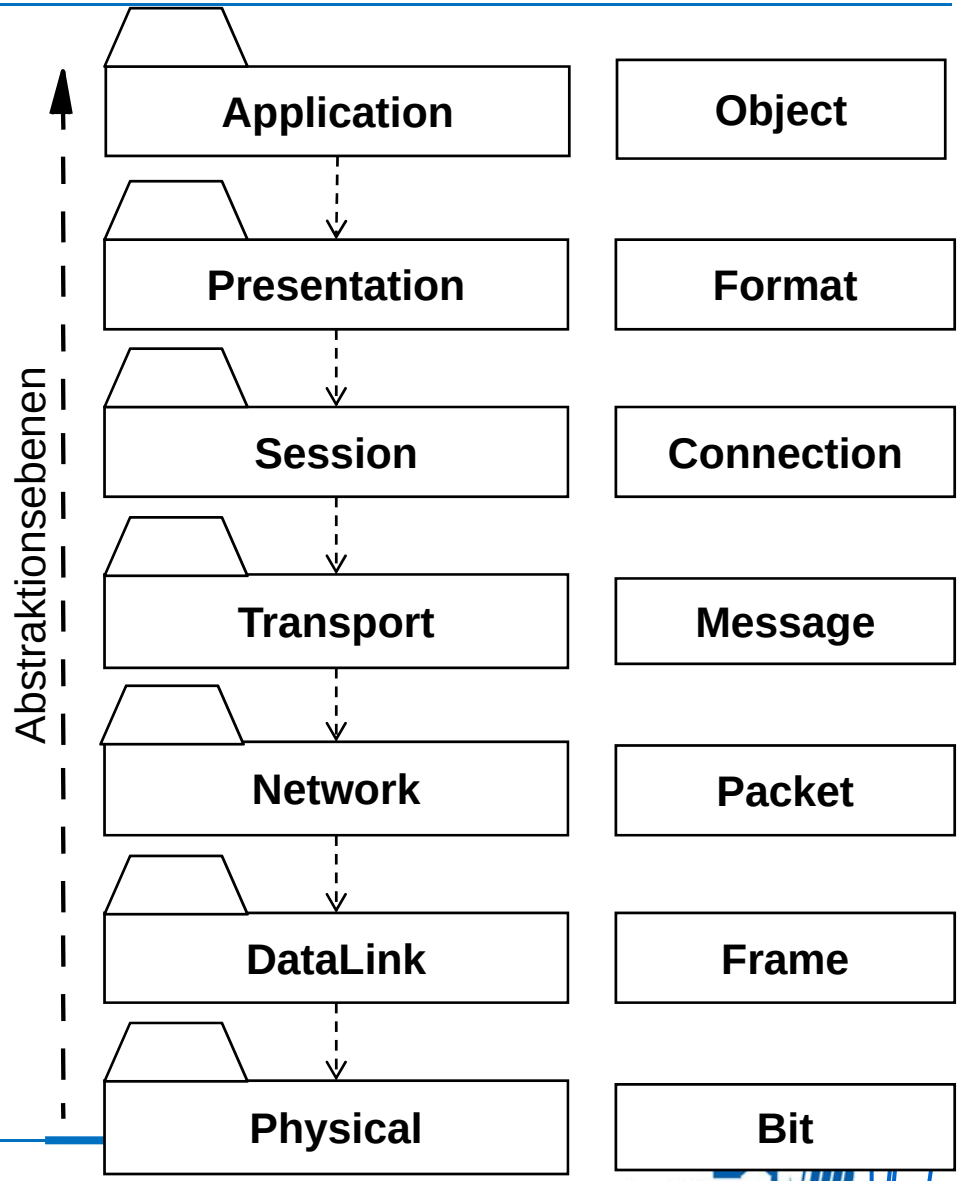
Geschichtete Systeme sind hierarchisch. Das ist wünschenswert, weil Hierarchie die Komplexität reduziert.

- Geschlossene Schichten-Architektur (Opaque Layering)
  - Jede Schicht kennt nur die nächsttiefere Schicht.
  - Geschlossene Schichten sind leichter zu pflegen.
- Offene Schichten-Architekturen (Transparent Layering)
  - Jede Schicht darf alle tiefer liegenden Schichten kennen / benutzen.
  - Offene Schichten sind effizienter.

Geschlossene Schichten-Architektur
Beispiel „Verteilte Kommunikation“

- ISO’s OSI Referenzmodell
  - ISO = International Organization for Standardization
  - OSI = Open System Interconnection

- Das Referenzmodell definiert Netzwerkprotokolle in 7 übereinander liegenden Schichten sowie strikte Regeln zu Kommunikation zwischen diesen Schichten.

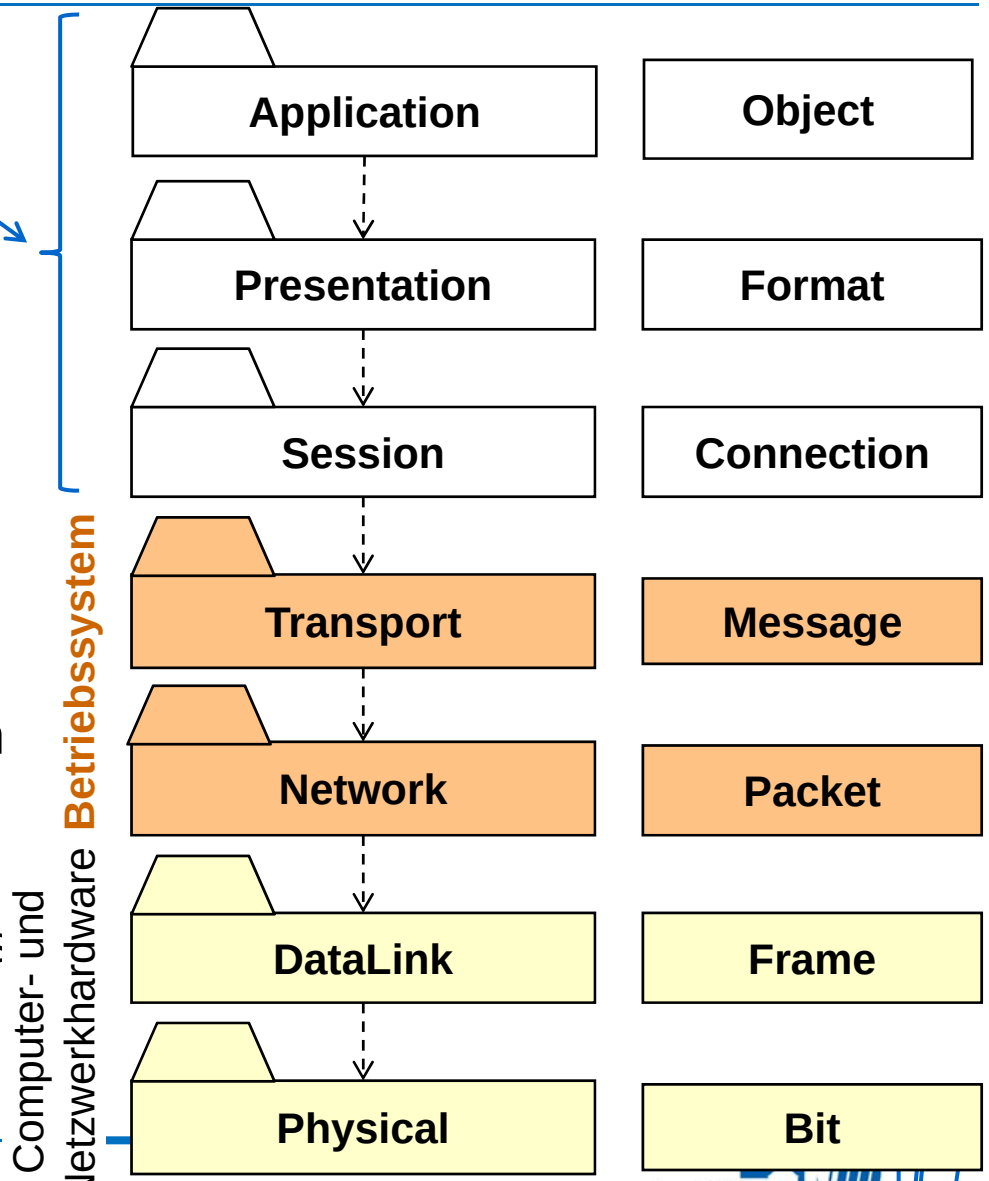


Geschlossene Schichten-Architektur
Beispiel „Verteilte Kommunikation“

Verteilte Programmierung

ist mühsam und fehleranfällig:

- *Verbindungsherstellung* zwischen Prozessen
- *Kommunikation* zwischen Prozessen statt Objekten
- *Packen/Entpacken* von Informationen in Nachrichten statt Parameterübergabe
- *Umcodierung* der Informationen wegen heterogener Platformen
- *Berücksichtigung technischer Spezifika* des Transportsystems



Geschlossene Schichten-Architektur

Middleware erlaubt Konzentration auf Anwendungsschicht

- Middlewareabhängig
- Plattformunabhängig

Middleware

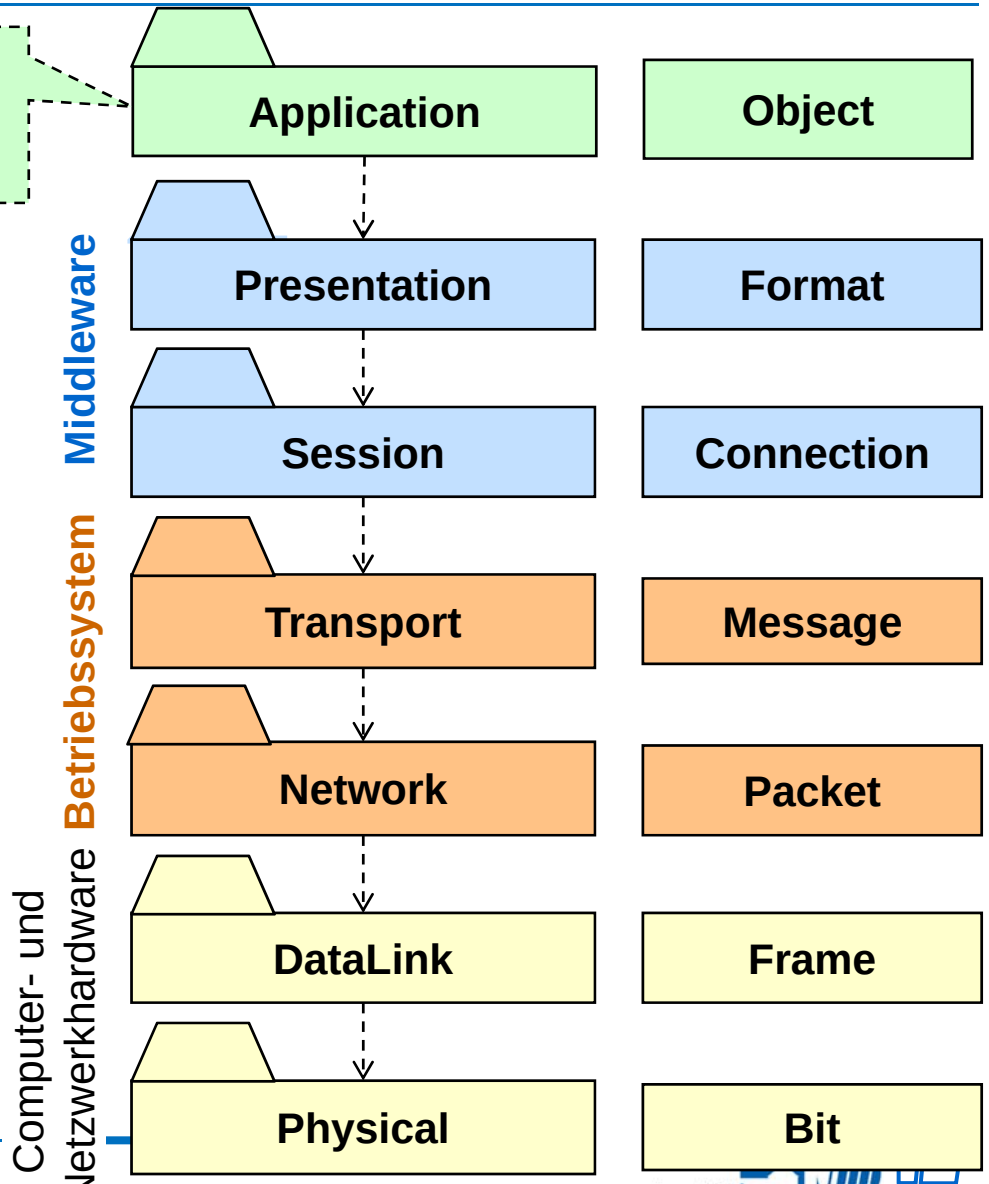
Garantiert Transparenz der

- Verteilung
- Platform

Plattform

Unterste Hardware- und Softwareschichten

- Betriebssystem
- Computer- und Netzwerkhardware



Middleware

- Definition: Middleware
  - Softwaresystem auf Basis standardisierter Schnittstellen und Protokolle, die Dienste bietet, die zwischen der Plattform (Betriebssystem + Hardware) und den Anwendungen angesiedelt sind und deren Verteilung unterstützen
- Bekannte Ansätze
  - Remote Procedure Calls
  - Java RMI (Remote Method Invocation)
  - CORBA (Common Object Request Broker Architecture)
- Wünschenswerte Eigenschaften
  - Gemeinsame Ressourcennutzung
  - Nebenläufigkeit
  - Skalierbarkeit
  - Fehlertoleranz
  - Sicherheit
  - Offenheit
- ➢ Vertiefung: Vorlesung „Verteilte Systeme“ (Dr. Serge Schumilov)

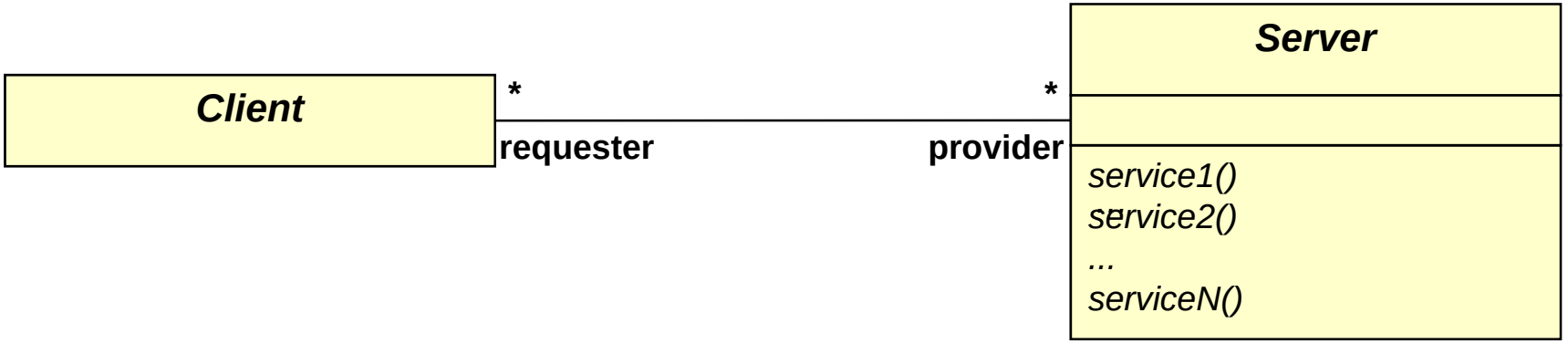
Applikationsserver

- Definition: Applicationsserver
  - Softwaresystem das als Laufzeitumgebung für Anwendungen dient und dabei über Middleware-Funktionen hinausgehende Fähigkeiten bietet
    - ⇨ Transparenz der Datenquellen
    - ⇨ Objekt-Relationales Mapping
    - ⇨ Transaktionsverwaltung
    - ⇨ Lebenszyklusmanagement („Deployment“, Updates, Start)
    - ⇨ Verwaltung zur Laufzeit (Monitoring, Kalibrierung, Logging, ...)
- Beispiel-Systeme (kommerziell)
  - IBM WebSphere
  - Oracle WebLogic
- Beispiel-Systeme (open source)
  - JBoss
  - Sun Glassfish
  - Apache Tomcat

Schichten-Architektur ▸ Client / Server

Abbildung von Schichten auf Rechner im Netzwerk

- Server bieten Dienste für Clients
- Clients ruft Operation eines Dienstes auf; die wird ausgeführt und gibt ein Ergebnis zurück
  - Client kennt das Interface des Servers (seinen Dienst)
  - Server braucht das Interface des Client nicht zu kennen
- Nutzer interagieren nur mit dem Client



Schichten-Architektur ▸ Client / Server

- Oft bei Datenbanksystemen genutzt
  - Front-End: Nutzeranwendung (Client)
  - Back-End: Datenbankzugriff und Datenmanipulation (Server)
- Vom Client ausgeführte Funktionen
  - Maßgeschneiderte Benutzerschnittstelle
  - Front-end-Verarbeitung der Daten
  - Aufruf serverseitiger RPCs (Remote Procedure Call)
  - Zugang zum Datenbankserver über das Netzwerk
- Vom Datenbankserver ausgeführte Funktionen
  - Zentrales Datenmanagement
  - Datenintegrität und Datenbankkonsistenz
  - Datenbanksicherheit
  - Nebenläufige Operationen (multiple user access)
  - Zentrale Verarbeitung (zum Beispiel Archivierung)

Entwurfsziele für Client/Server Systeme

- Portabilität
  - Server kann auf vielen unterschiedlichen Maschinen und Betriebssystemen installiert werden und funktioniert in vielen Netzwerkumgebungen
- Transparenz
  - Der Server könnte selbst verteilt sein (warum?), sollte dem Nutzer aber einen einzigen “logischen” Dienst bieten
- Performance
  - Client sollte für interaktive, UI-lastig Aufgaben maßgefertigt sein
  - Server sollte CPU-intensive Operationen bieten
- Skalierbarkeit
  - Server hat genug Kapazität, um eine größere Anzahl Clients zu bedienen
- Flexibilität
  - Server sollte für viele Front-Ends nutzbar sein
- Zuverlässigkeit
  - System sollte individuelle Knoten-/Verbindungsprobleme überleben

Probleme mit Client/Server Architekturen

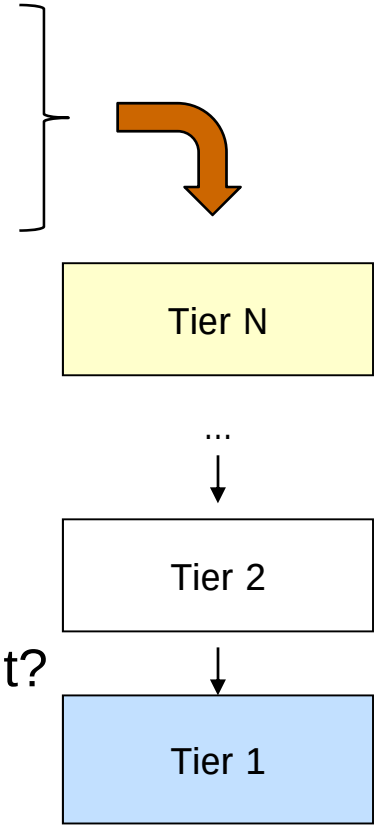
- Geschichtete Systeme unterstützen keine gleichberechtigte gegenseitige („Peer-to-peer“) Kommunikation

- „Peer-to-peer“ Kommunikation wird oft benötigt
  - Beispiel: Eine Datenbank empfängt Abfragen von einer Anwendung, schickt aber auch Benachrichtigungen an die Anwendung wenn der Datenbestand sich geändert hat.

Schichten-Architektur ▸ Von der einfachen Client/Server- zur N-Tier-Architektur

Entwurfsentscheidungen verteilter Client-Server-Anwendung

- Wie werden die Aufgaben der Anwendung auf Komponenten verteilt?
  - Typische Aufgabenteilung
    - ⇨ *Präsentation* – Schnittstelle zum Anwender
    - ⇨ *Anwendungslogik* – Bearbeitung der Anfragen
    - ⇨ *Datenhaltung* – Speicherung der Daten in einer Datenbank
- Wie viele Prozessräume gibt es?
  - Eine Stufe / Schicht (engl. *‚tier‘*) kennzeichnet einen Prozessraum innerhalb einer verteilten Anwendung
  - Das *N* legt fest, wie viele Prozessräume es gibt
  - Ein Prozessraum kann, muss jedoch nicht(!), einem physikalischen Rechner entsprechen
- Wie werden die Komponenten auf Prozessräume verteilt?
  - Die Art der Zuordnung der Aufgaben zu den tiers macht den Unterschied der verschiedenen n-tier Architekturen aus

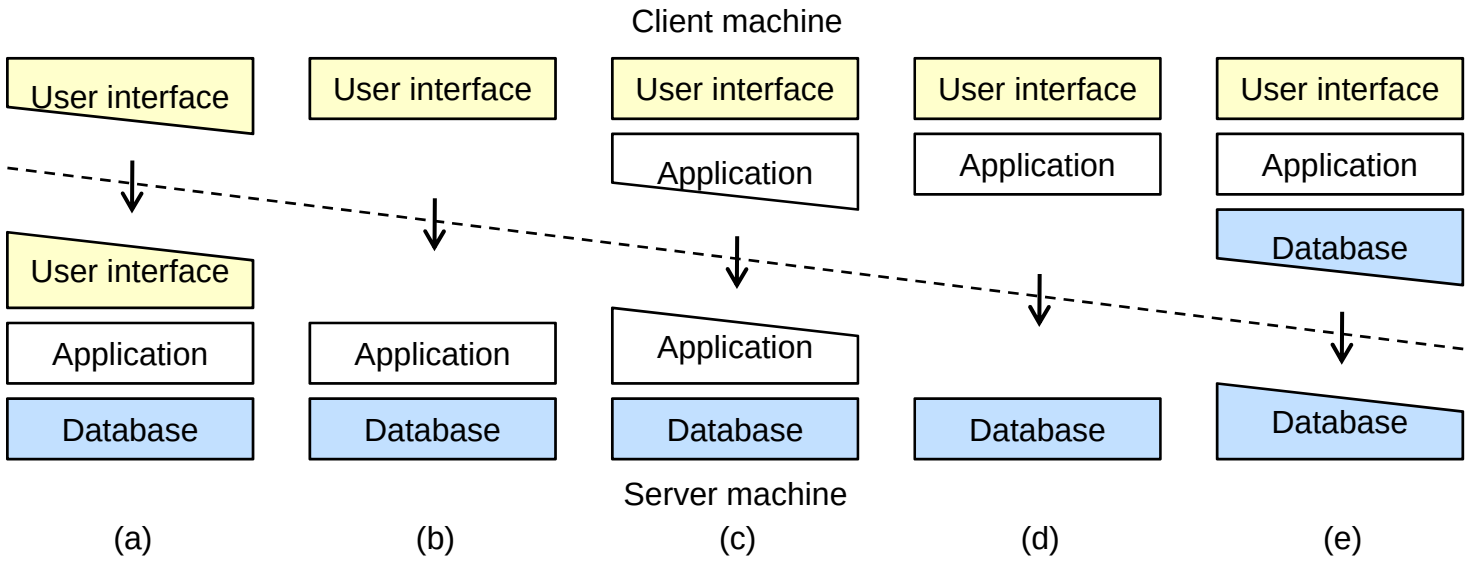


2-Tier Architektur

- Ältestes Verteilungsmodell: Client- und Server-Tier
- Zuordnung von Aufgaben zu Tiers
  - Präsentation → Client
  - Anwendungslogik → Beliebig
  - Datenhaltung → Server
- Vorteile
  - Einfach und schnell umzusetzen
  - Performant
- Probleme
  - Schwer wartbar
  - Schwer skalierbar
  - Software-Update Problem

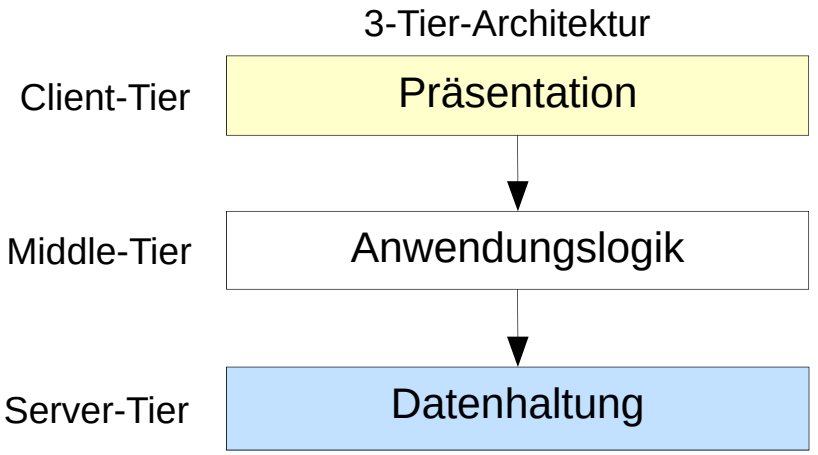
2-Tier Architektur ▸ Varianten

- Ultra-Thin-Client Architekturen (a):
  - ⇨ Die Client-Tier beschränkt sich auf Anzeige von Dialogen in einem Browser.
- Thin-Client Architekturen (a,b):
  - ⇨ Die Client-Tier beschränkt sich auf Anzeige von Dialogen und die Aufbereitung der Daten zur Anzeige.
- Fat-Client Architekturen (c,d,e):
  - ⇨ Teile der Anwendungslogik liegen zusammen mit der Präsentation auf der Client-Tier.



3–Tier Architekturen

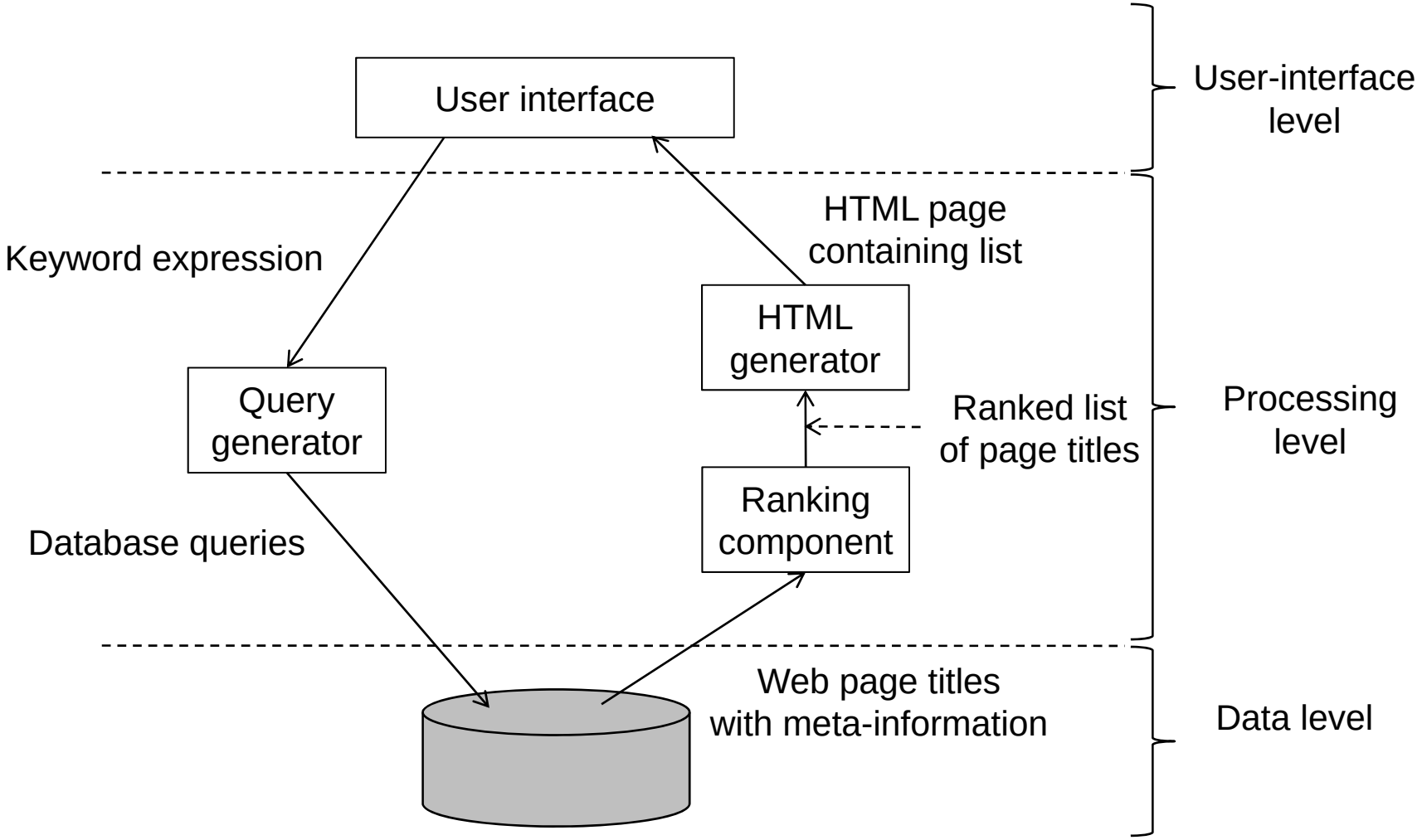
- Zuordnung von Aufgaben zu 3 Tiers
  - Präsentation ⇨ Client-Tier
  - Anwendungslogik ⇨ Middle-Tier
  - Datenhaltung ⇨ Server-Tier



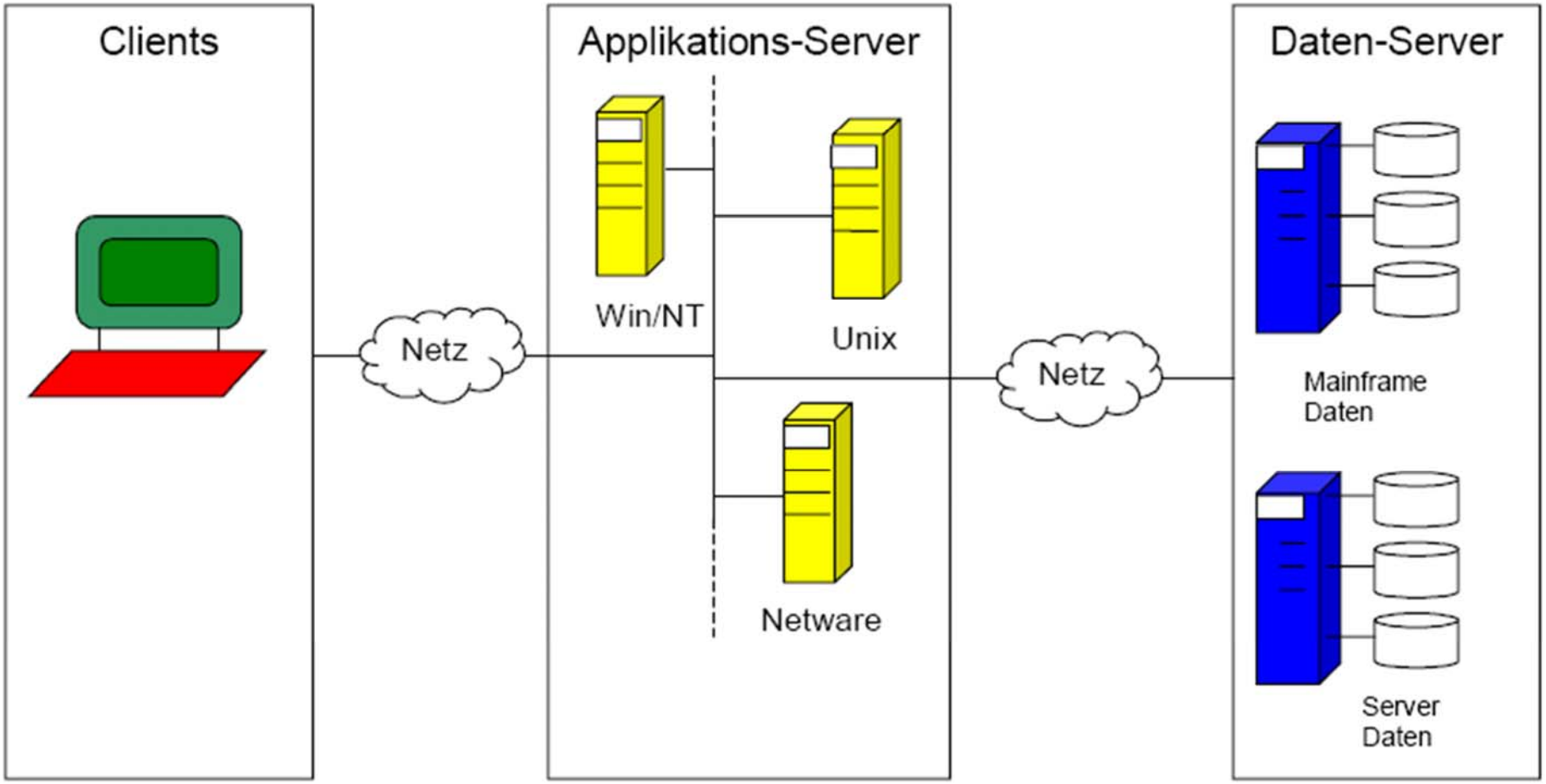
- Standardverteilungsmodell für einfache Webanwendungen
  - Client-Tier = Browser zur Anzeige
  - Middle-Tier = Webserver mit Servlets / ASP / Anwendung
  - Server-Tier = Datenbankserver

3–Tier Architekturen ▸ Beispiel Webanwendungen

- Standardverteilungsmodell für einfache Webanwendungen:

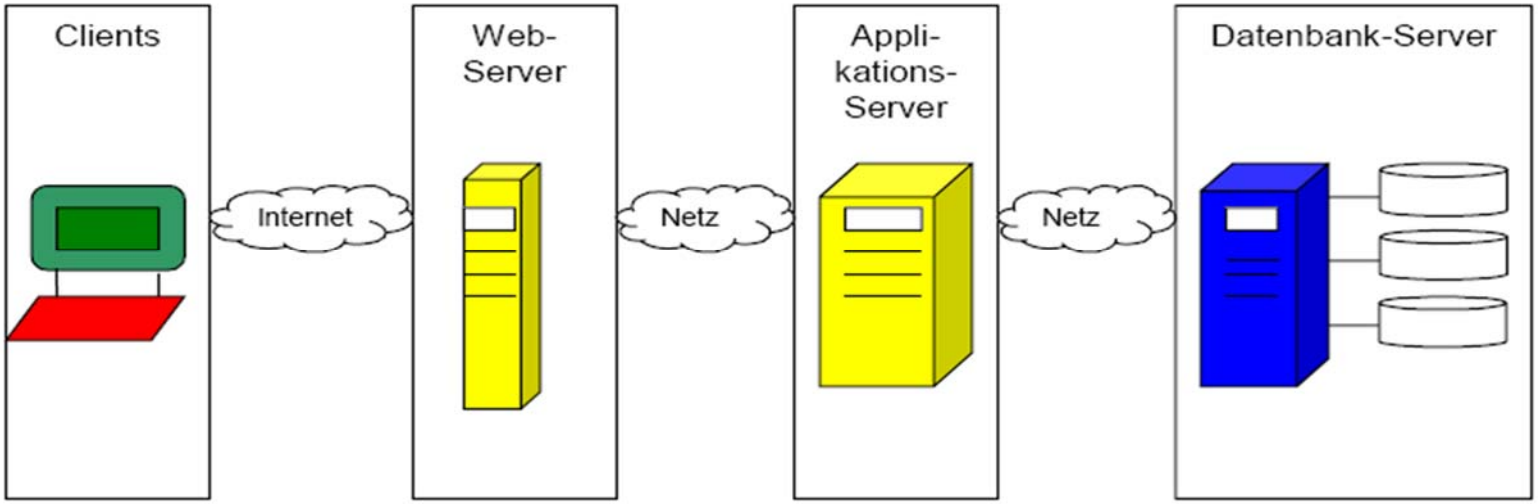


3–Tier Architekturen ▸ Applications-Server (EJB)



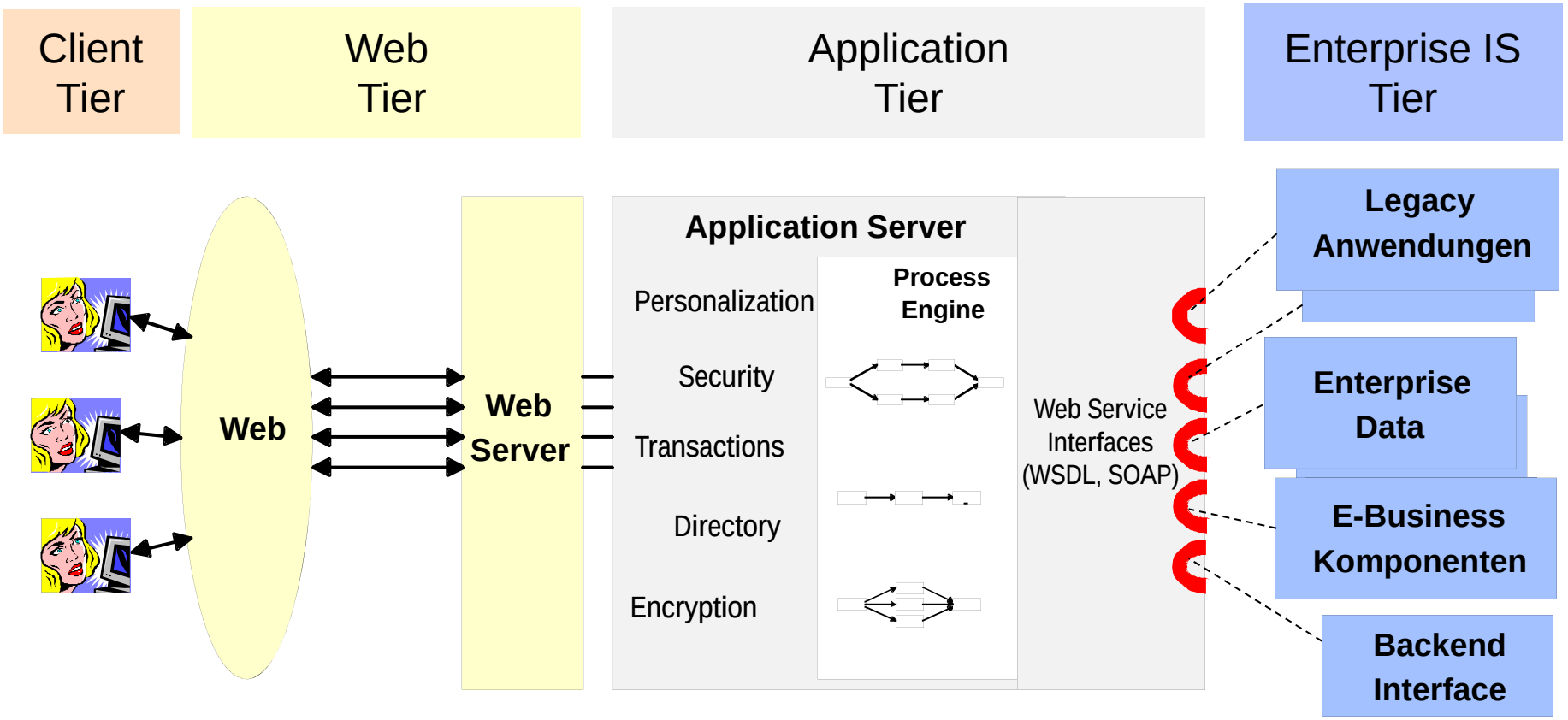
4- und Mehr-Tier Architekturen

- Unterschied zu 3-Schichten-Architekturen
  - Die Anwendungslogik wird auf mehrere Schichten verteilt (Webserver, Application Server)



- Motivation
  - Minimierung der Komplexität („Divide and Conquer“)
  - Besserer Schutz einzelner Anwendungsteile
- Grundlage für die meisten Applikationen im E-Bereich
  - E-Business, E-Commerce, E-Government

4-Tier-Architektur eines Informationssystems



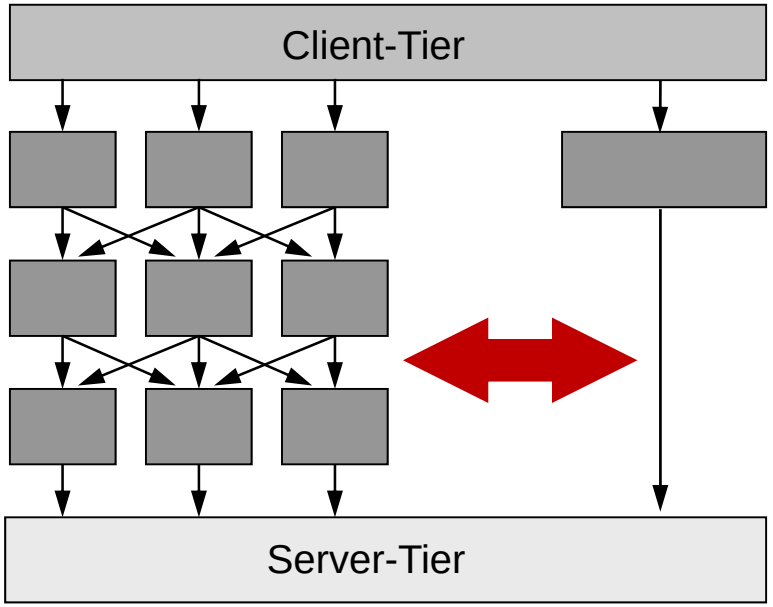
N-Tier-Architektur ▸ Abwägungen

Box = Komponente des Systems

Pfeil = Kommunikationsverbindung

- Mehr Boxen
  ⇒ mehr Verteilung + Parallelität
  ⇒ mehr *Kapselung* + *Wiederverwendung*
- Mehr Boxen
  ⇒ mehr Pfeile
  ⇒ Verbindungen zu verwalten
  ⇒ *mehr Koordination* + *Komplexität*
- Mehr Boxen
  ⇒ mehr Vermittlung
  ⇒ mehr Datentransformationen
  ⇒ *schlechte Performanz*

Entwickler einer Architektur versuchen deswegen immer ein *Kompromiss* zu finden

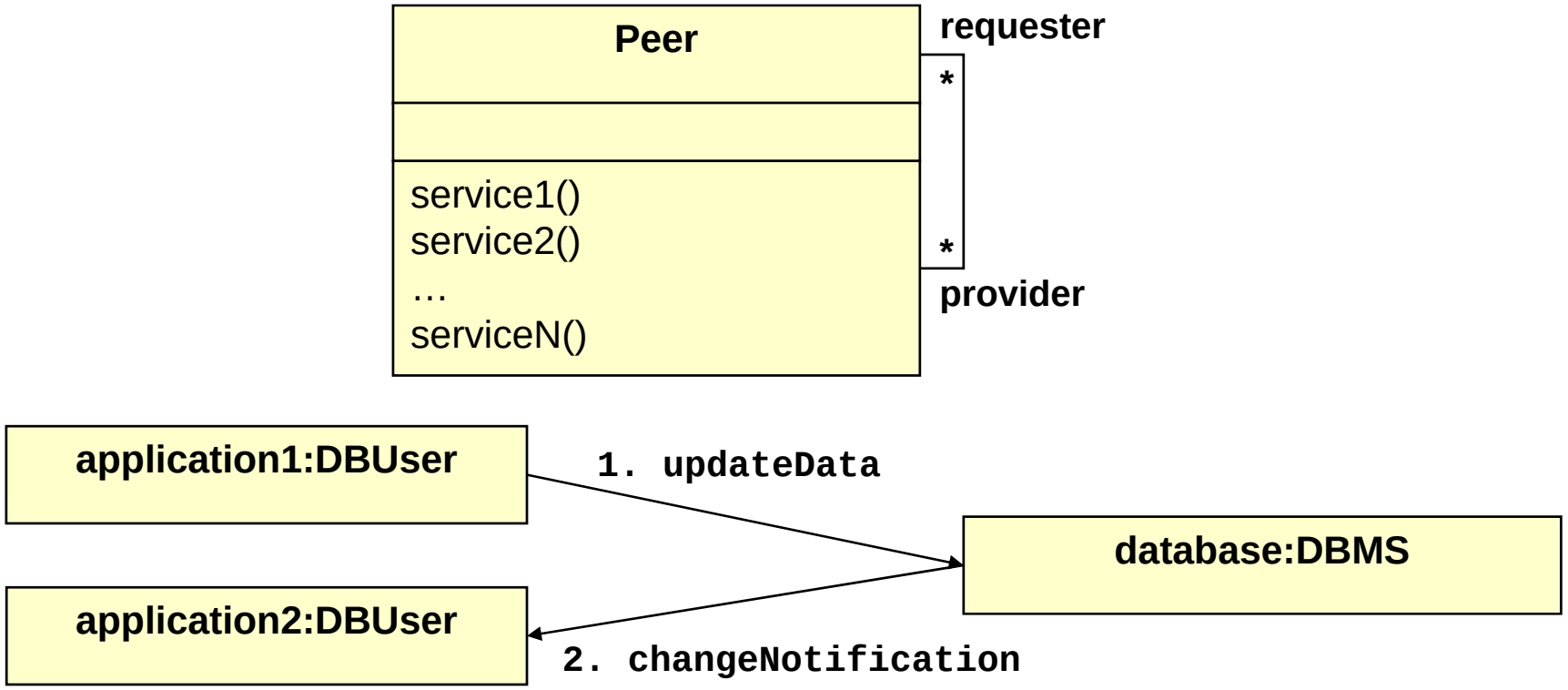


Es gibt kein Designproblem, das man durch Einführung einer zusätzlichen Vermittlungsschicht nicht lösen kann.

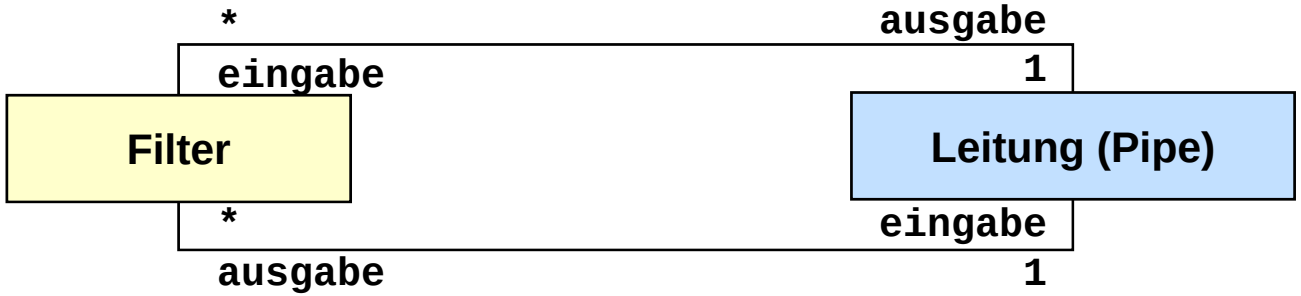
Es gibt kein Performanzproblem, das man durch Entfernung einer zusätzlichen Vermittlungsschicht nicht lösen kann.

Peer-to-Peer Architektur

- Generalisierung der Client/Server Architektur
- Clients können Server sein und umgekehrt
- Schwieriger wegen möglicher Deadlocks



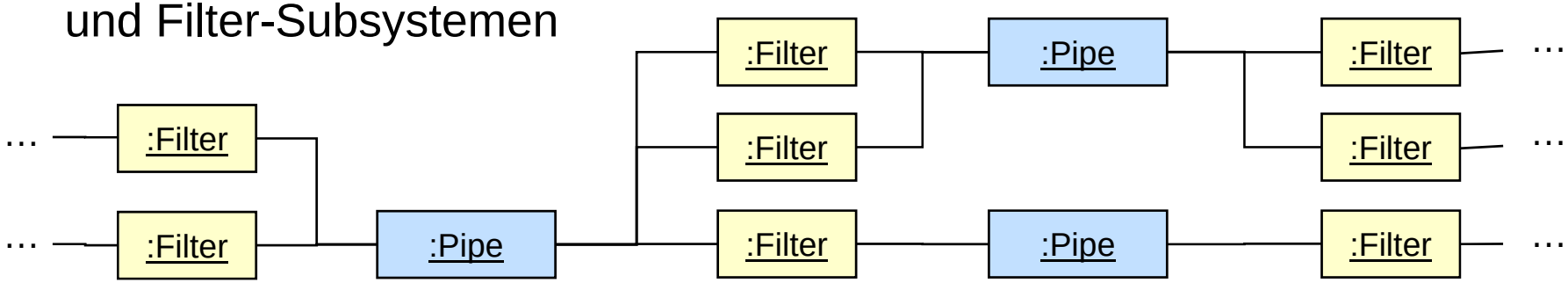
Pipe-and-Filter-Architecture



- Filter-Subsysteme bearbeiten Daten
  - Es sind reine Funktionen
  - Sie sind konzeptionell und implementierungstechnisch unabhängig von
    - ⇨ den Pipes die die Daten zu ihnen bzw. von ihnen Weg leiten
    - ⇨ den Erzeugern und Verbrauchern der Daten
- Leitungs-Subsysteme leiten Daten weiter
  - Sie sammeln Daten von einem oder mehreren Filtern
  - Sie leiten Daten an einen oder mehrere Filter weiter
  - Sie dienen der Synchronisation paralleler Filteraktivitäten
  - Sie sind von allen anderen Subsystemen völlig unabhängig (genau wie die Filter)

Pipe-and-Filter-Architektur: Ausschnitt aus möglicher Systemkonfiguration

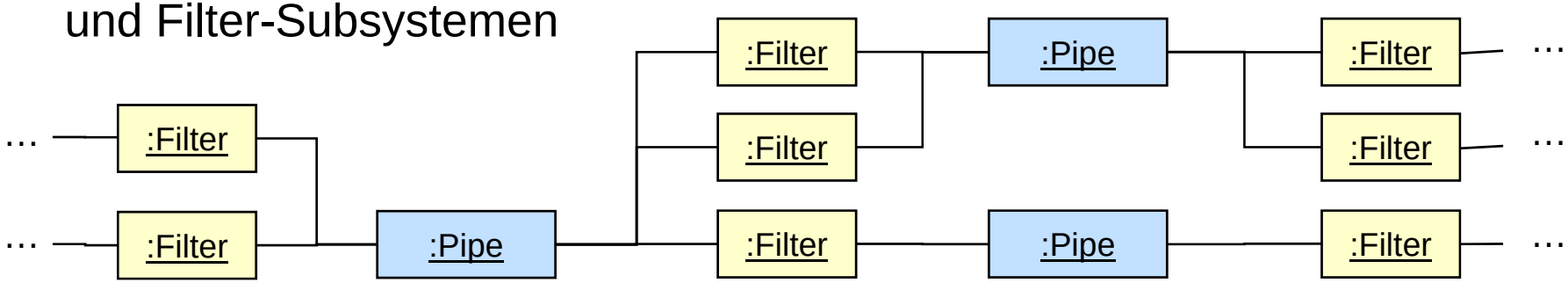
- Das Gesamtsystem entsteht einfach durch die Verknüpfung von Pipe- und Filter-Subsystemen



- Vorteile
  - Flexibilität: leichter Austausch von Filtern, Leichte Rekonfiguration der Verbindungen über Pipes
  - Effizienz: Hoher Grad an Parallelität (alle Filter können Parallel arbeiten!)
  - Gut geeignet für automatisierte Transformationen auf Datenströmen
    - ⇨ Beispiel: Sattelitendatenbearbeitung

Pipe-and-Filter-Architektur: Ausschnitt aus möglicher Systemkonfiguration

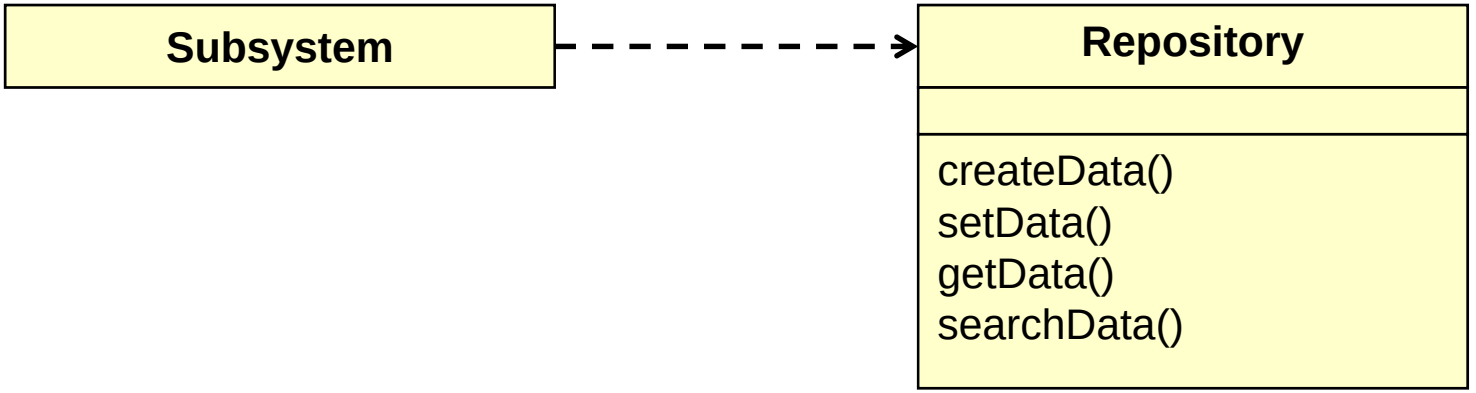
- Das Gesamtsystem entsteht einfach durch die Verknüpfung von Pipe- und Filter-Subsystemen



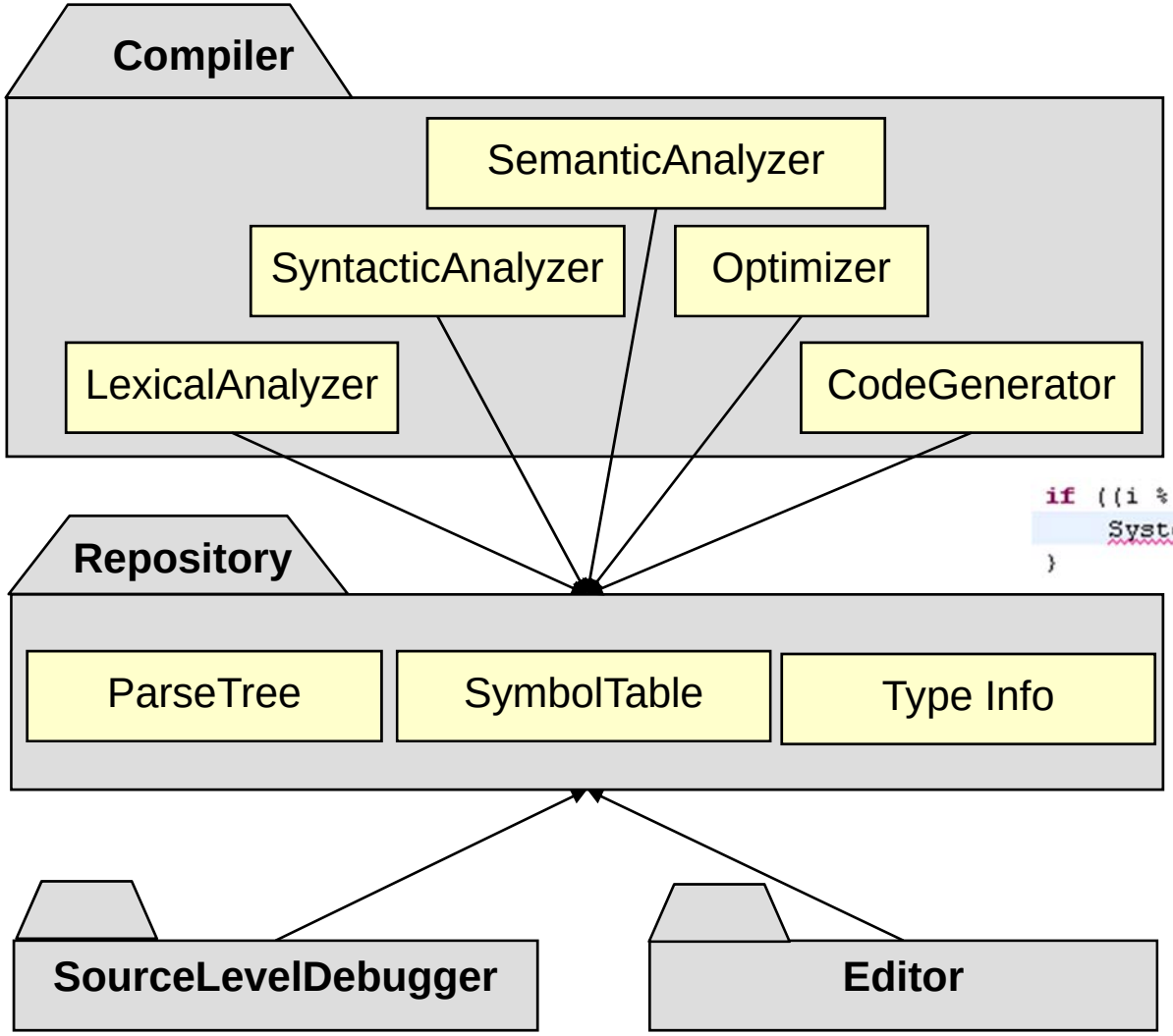
- Weniger geeignet für
  - Hochinteraktive Aufgaben
    - ⇨ Benutzerinteraktion macht die potentielle Parallelität zunichte
  - Aufgaben, wo die Daten sich nicht bzw. nur wenig ändern, da sich dann der Aufwand die Daten ständig zu kopieren nicht lohnt
    - ⇨ In diesem Fall ist eine Repository-Architektur vorteilhafter

Repository Architektur

- Subsysteme lesen und schreiben Daten einer einzigen, gemeinsamen Datenstruktur
- Subsysteme sind lose gekoppelt (Interaktion nur über das Repository)
- Kontrollfluss wird entweder zentral vom Repository diktiert (Trigger) oder von den Subsystemen bestimmt (locks, synchronization primitives)



Repository Architektur: Beispiel „Eclipse“



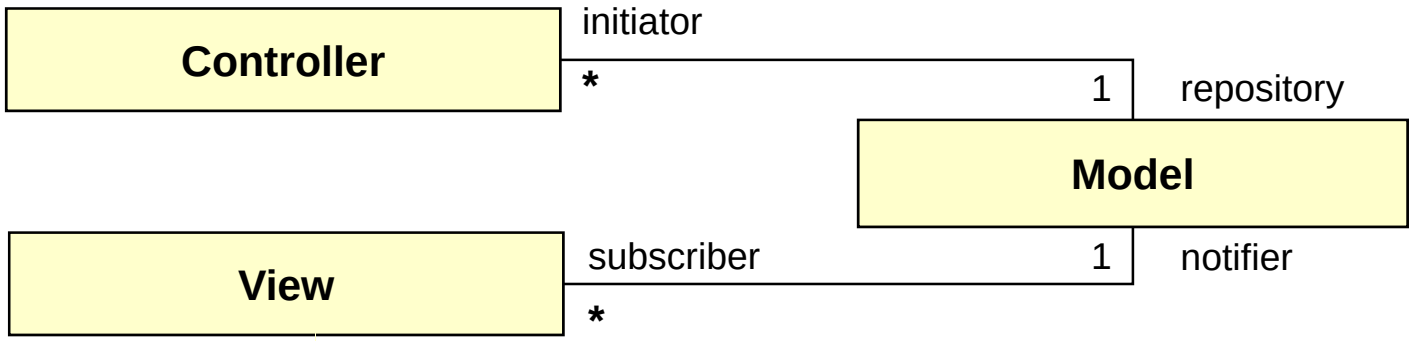

```
if ((i % 3 == 0) || (i % 5 == 0)) {
    System.out.println
}
```

println() : void - PrintStream
println(boolean x) : void - PrintStream
println(char x) : void - PrintStream
println(char[] x) : void - PrintStream
println(double x) : void - PrintStream
println(float x) : void - PrintStream
println(int x) : void - PrintStream
println(long x) : void - PrintStream
println(Object x) : void - PrintStream

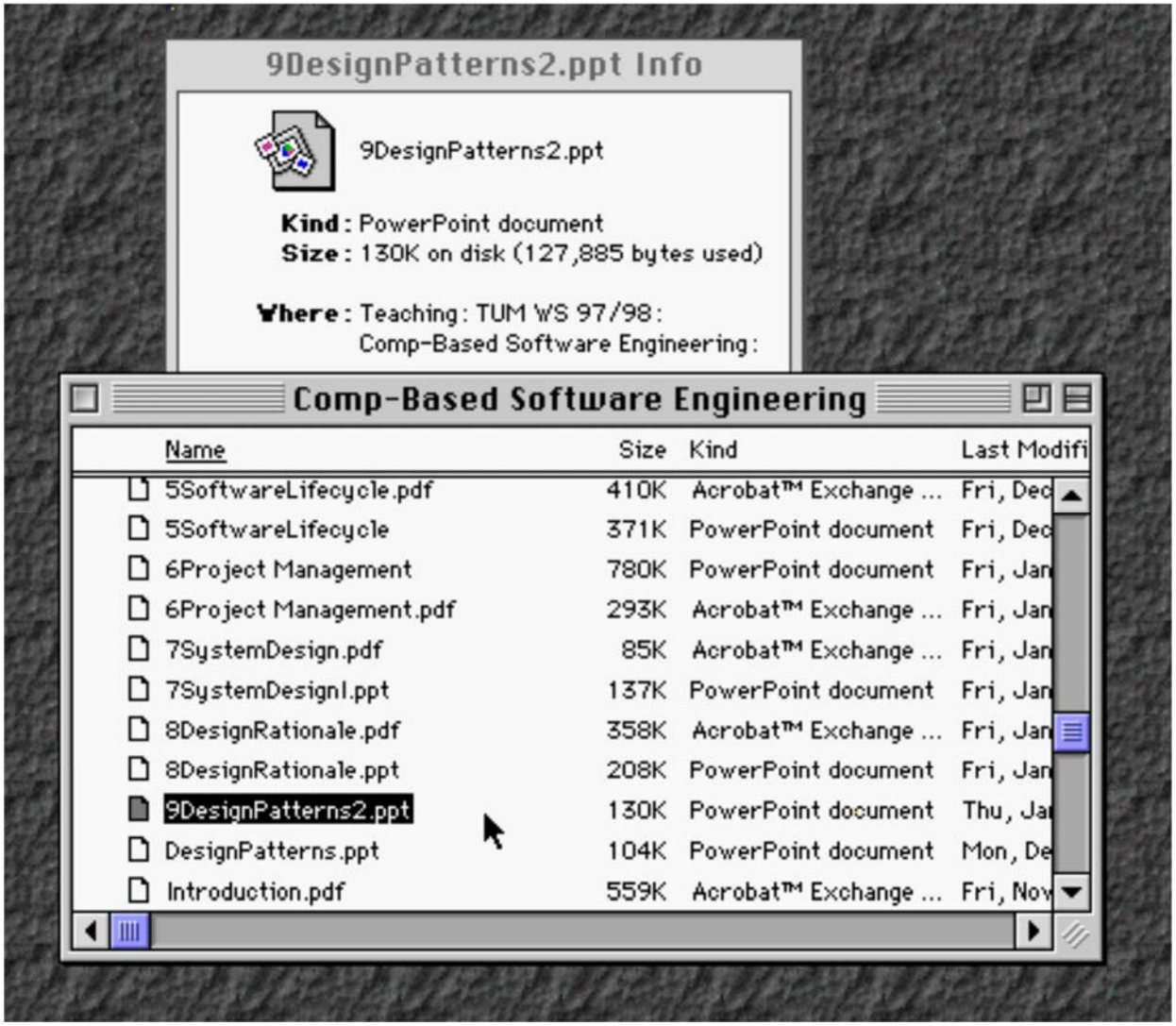
Press 'Ctrl+Space' to show Template Proposals

Model / View / Controller

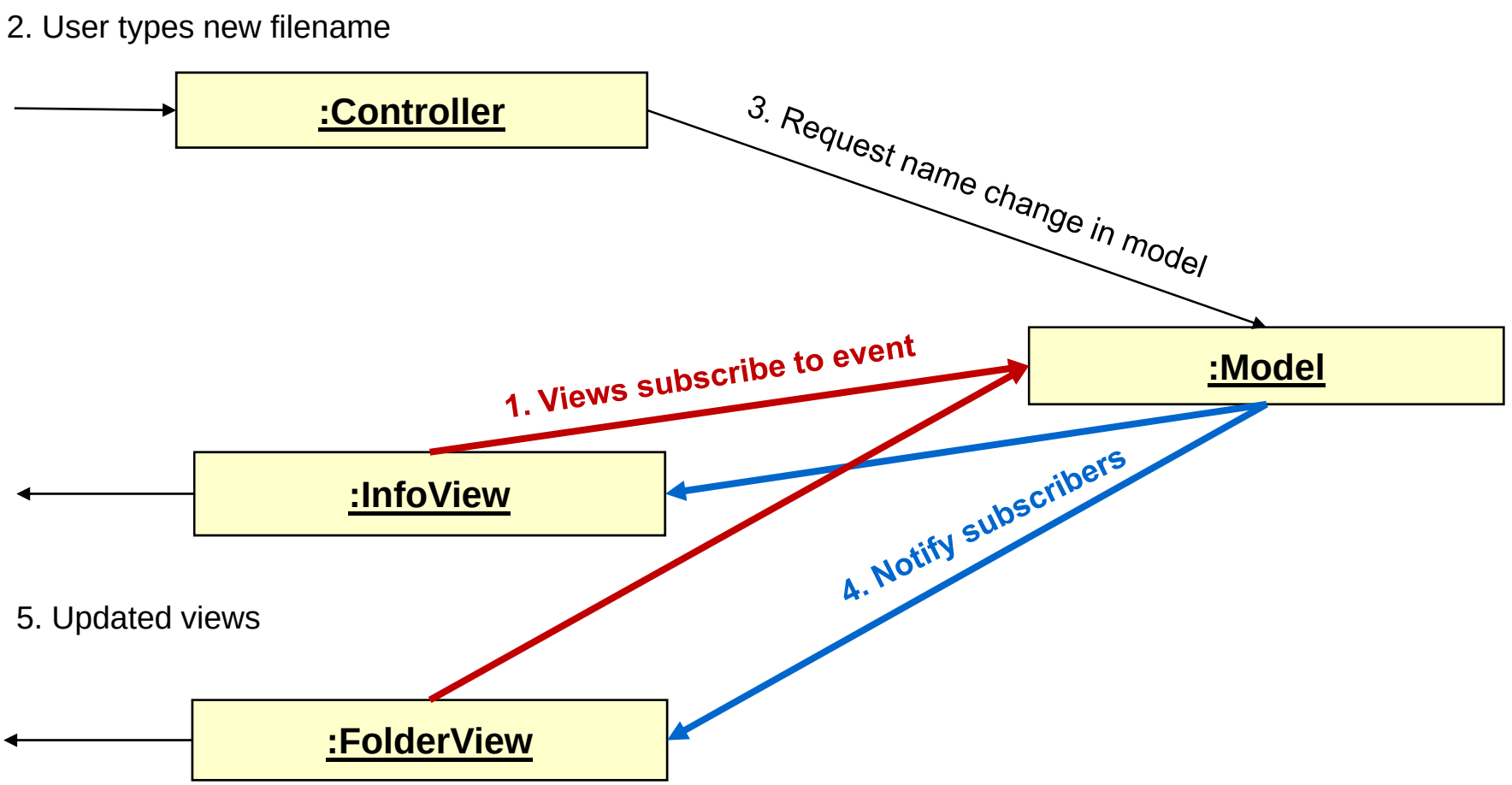
- Subsysteme werden in drei verschiedene Typen unterteilt
  - Model Subsystem: Zuständig für das Wissen der Anwendungsdomäne und Benachrichtigung der Views bei Änderungen im Model.
  - View Subsystem: Stellt die Objekte der Anwendungsdomäne für den Nutzer dar
  - Controller Subsystem: Verantwortlich für die Abfolge der Interaktionen mit dem Nutzer.
- MVC ist eine Verallgemeinerung der Repository Architektur:
  - Das Model Subsystem implementiert die zentrale Datenstruktur
  - Das Controller Subsystem schreibt explizit den Kontrollfluss vor



Beispiel einer auf der MVC Architektur basierenden Dateiverwaltung



Abfolge von Events

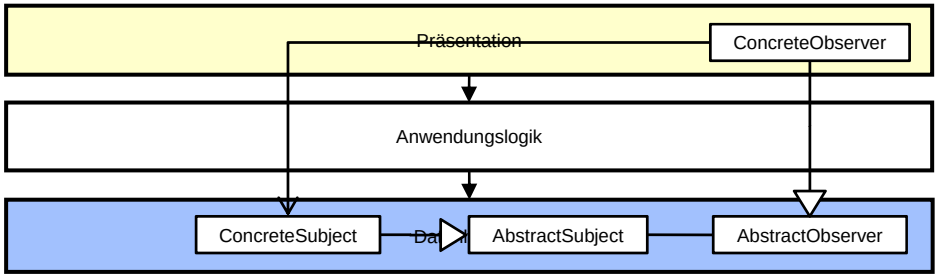


Software-Architekturen und das Observer-Pattern

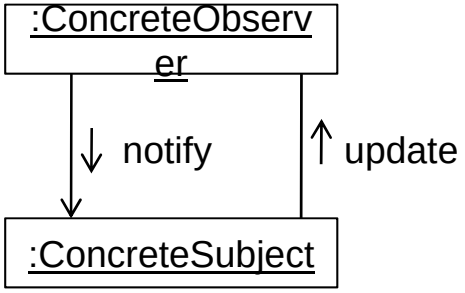
Das Observer Pattern: Konsequenzen

- Abstrakte Kopplung
  - Subjekte aus tieferen Schichten eines Systems können mit Beobachtern aus höheren Schichten kommunizieren, ohne die Schichten zu verletzen.

Statische Abhängigkeit: Hierarchisch

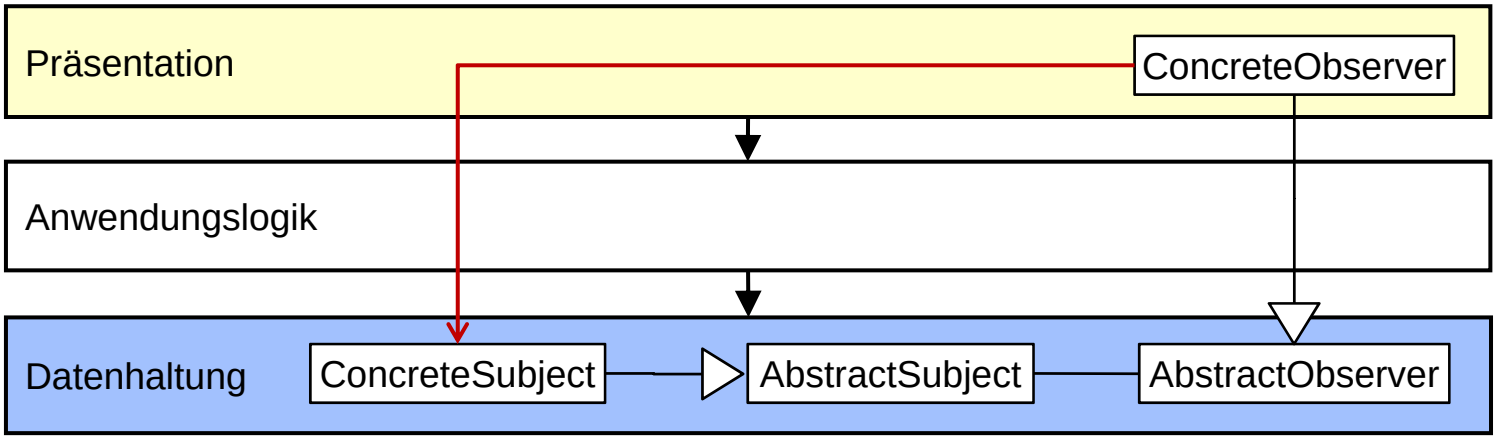


Dynamische Interaktion/Referenzen: Bidirektional



Auswirkungen von Observer auf „Presentation-Application-Data“-Ebenen

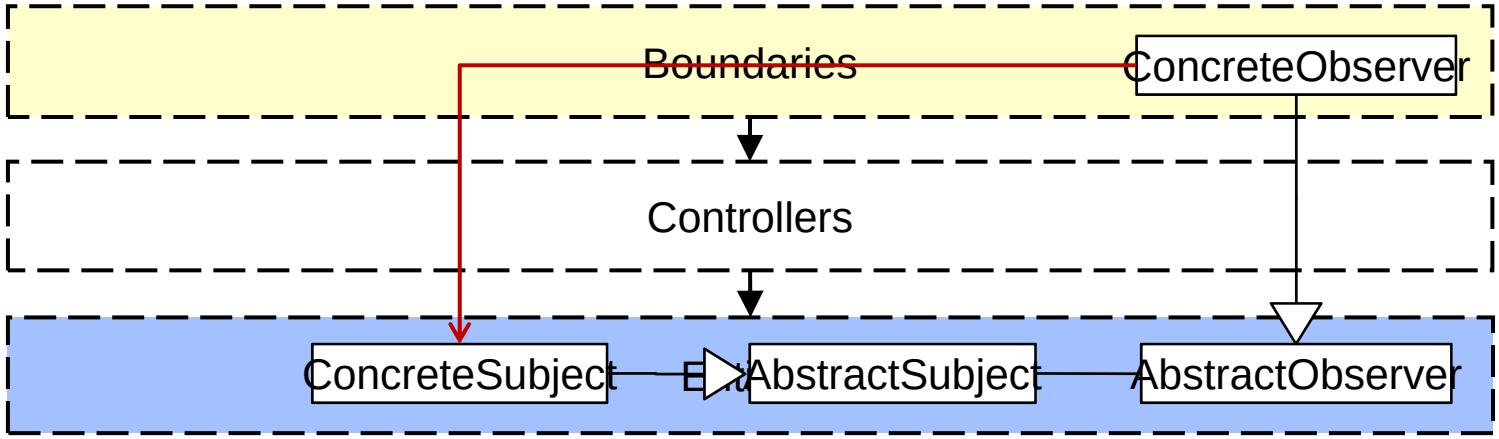
- n-tier-Architekturen basieren auf der rein hierarchische Anordnung von Presentation, Anwendungslogik und Datenhaltung



- Die Aktualisierung der Präsentation bei Änderung der Daten ist durch das Observer-Pattern möglich, ohne dass die Daten von der Präsentation wissen müssen.
  - Sie sind nur von dem AbstractObserver abhängig.
  - Wenn dessen Definition in der Datenschicht angesiedelt ist, wird die Ebenenanordnung nicht verletzt

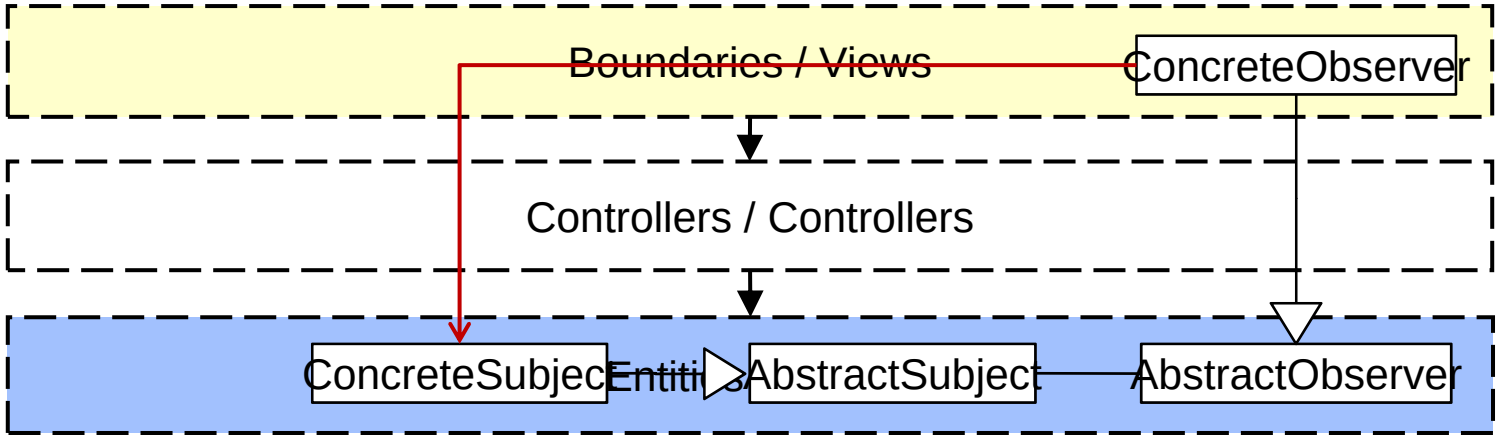
Auswirkungen von Observer für „Boundary-Controller-Entity“ Stereotypen

- Die Abhängigkeitsreduzierung ist die gleiche wie bei den Presentation-Application-Data Ebenen der N-Ebenen-Architekturen



- Der einzige Unterschied zwischen BCE und PAD ist die Gruppierung:
  - BCE beschreibt lediglich die Funktionen einzelner Objekttypen (Es sagt nichts über ihre Gruppierung in Ebenen aus)
  - PAD sagt etwas über die Gruppierung von Objekttypen gleicher Funktion:
    - ⇨ Alle Boundaries mit GUI-Funktionalität in der Präsentationsschicht
    - ⇨ Alle Controller in der Anwendungslogik-Schicht
    - ⇨ Alle Entities in der Datenhaltungs-Schicht

Auswirkungen von Observer für Model-View-Controller



- Views sind immer Boundaries und Observer
  - Sonst könnten Sie ihre Funktion nicht erfüllen
  - Das schließt nicht aus, dass sie eventuell noch andere Rollen spielen
- Boundaries sind nicht immer Views
  - Beispiel: Tastatur

Auswirkungen von Observer für Model-View-Controller

- Observer sind nicht immer Views!
  - Auch Kontroller können Observer sein!
  - Sie können sich bei einer Menge von Modellelementen als Observer registrieren und deren Veränderungen sammeln, bewerten und gefiltert oder kummuliert weitergeben.
    - ⇨ Aktive Weitergabe: Aufruf / Steuerung von Aktionen anderer Objekte („Kontroller“-Rolle)
    - ⇨ Passive Weitergabe: Beachrichtigung von eigenen Observern („Modell“-Rolle)
    - ⇨ Siehe auch „Mediator-Pattern“ („Vermittler“)
  - Das gleiche gilt auch für andere Modellelemente / Entities: auch sie können Observer von anderen Objekten sein.

Zusammenfassung

- Systementwurf
  - Verkleinert die Lücke zwischen Anforderungen und der Implementierung
  - Zerteilt das Gesamtsystem in handhabbare Stücke
- Definition der Entwurfsziele
  - Beschreibt und priorisiert die für das System wichtigen Qualitäten
  - Definiert das Wertesystem anhand dessen Optionen überprüft werden
- Subsystemdekomposition
  - Führt zu einer Menge lose gekoppelter Teile, die zusammen das System bilden
- Softwarearchitektur
  - Beschreibt die Beziehungen / Interaktionen der Subsysteme

universitätbonn ROOTS

Zielgerichteter Entwurf
(→ Brügge & Dutoit, Kap. 7)

Überblick

- Dekomposition (vorhergehender Abschnitt)
  - 0. Überblick über das Systementwurf
  - 1. Entwurfsziele
  - 2. Dekomposition in Subsysteme

- Zielgerichteter Entwurf
  - 3. Nebenläufigkeit
  - 4. Hardware/Software Zuordnung
  - 5. Management persistenter Daten
  - 6. Globale Ressourcenverwaltung und Zugangskontrolle
  - 7. Programmsteuerung
  - 8. Grenzfälle

3. Nebenläufigkeit

- Identifizieren nebenläufiger Ausführungsstränge und Behandeln von Fragen zur Nebenläufigkeit.
- Entwurfsziel: Reaktionszeit, Performance.
- Threads („Fäden“, „Stränge“)
  - Ein Thread ist ein Pfad durch eine Menge von Zustanddiagramme, wobei stets genau ein Objekt zur selben Zeit aktiv ist.
  - Ein Thread bleibt in einem Zustandsdiagramm bis ein Objekt einen Event an ein anderes Objekt sendet und auf einen anderen Event wartet
  - Thread (ab-)spaltung: Ein Objekt sendet ein asynchrones Event

Nebenläufigkeit (Fortsetzung)

- Zwei Objekte heißen inhärent nebenläufig, wenn sie zur gleichen Zeit Events empfangen können ohne zu interagieren
- Inhärent nebenläufige Objekte sollten verschiedenen Threads zugeordnet werden
- Objekte mit sich wechselseitig ausschließenden Aktivitäten sollten demselben Thread zugeordnet werden (Warum?)

Fragen zur Nebenläufigkeit

- Welche Objekte des Objektmodells sind unabhängig?
- Welche Arten von Threads sind identifizierbar?
- Bietet das System vielen Nutzern Zugriff?
- Kann eine einzelne Anfrage an das System in mehrere Teilanfragen zerlegt werden?
- Können diese Teilanfragen parallel abgearbeitet werden?

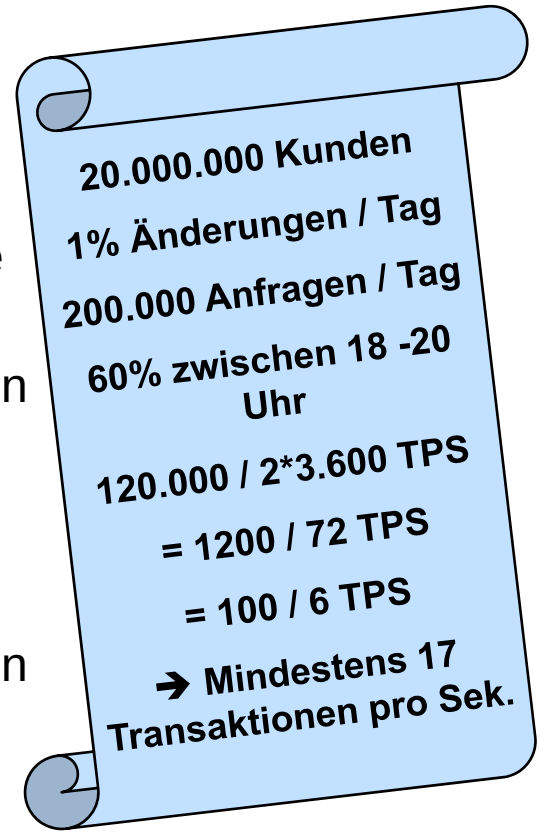
4. Hardware/Software Zuordnung

- Diese Aktivität befasst sich mit drei Fragen:
  - Wie sollen wir jedes einzelne Subsystem realisieren
    - ⇨ In Hardware oder in Software?
  - Welche Hard- / Software ist schon verfügbar?
    - ⇨ Komponenten von Drittanbietern die man nutzen kann
    - ⇨ Altsysteme die man integrieren muss
  - Wie wird das Objektmodell auf die gewählte Hard- und Software abgebildet?
    - ⇨ Objekte in die Realität abbilden → auf Rechner / Prozessor, Speicher, I/O
    - ⇨ Assoziationen in die Realität abbilden → auf Bus-/Netzwerktopologie

- Viele Schwierigkeiten beim Entwurf eines Systems sind die Folge von Kunden-Vorgaben bzgl. Hard- und Software.
  - „Wir haben gerade erst X Millionen für System Y ausgegeben…“
  - „Aufgabe X muss von Hard- /Software Y gelöst werden.“

Zuordnung von Objekten

- Auf Rechner / Prozessor
  - Ist die geforderte Berechnungsgeschwindigkeit zu hoch für einen einzelnen Prozessor?
  - Bringt die Verteilung der Aufgaben auf verschiedene Prozessoren einen Geschwindigkeitsgewinn?
  - Wie viele Prozessoren sind für den dauerhaft stabilen Betrieb unter Dauerlast notwendig?
- Auf Speicher
  - Ist genug Speicher vorhanden, um Belastungsspitzen abzufangen?
- Auf I/O
  - Reicht die Kommunikationsbandbreite zwischen den Hardware-Einheiten, auf denen Subsysteme eingesetzt werden, um die gewünschte Reaktionszeit zu garantieren?



Zuordnung der Assoziationen: Kommunikationstopologie

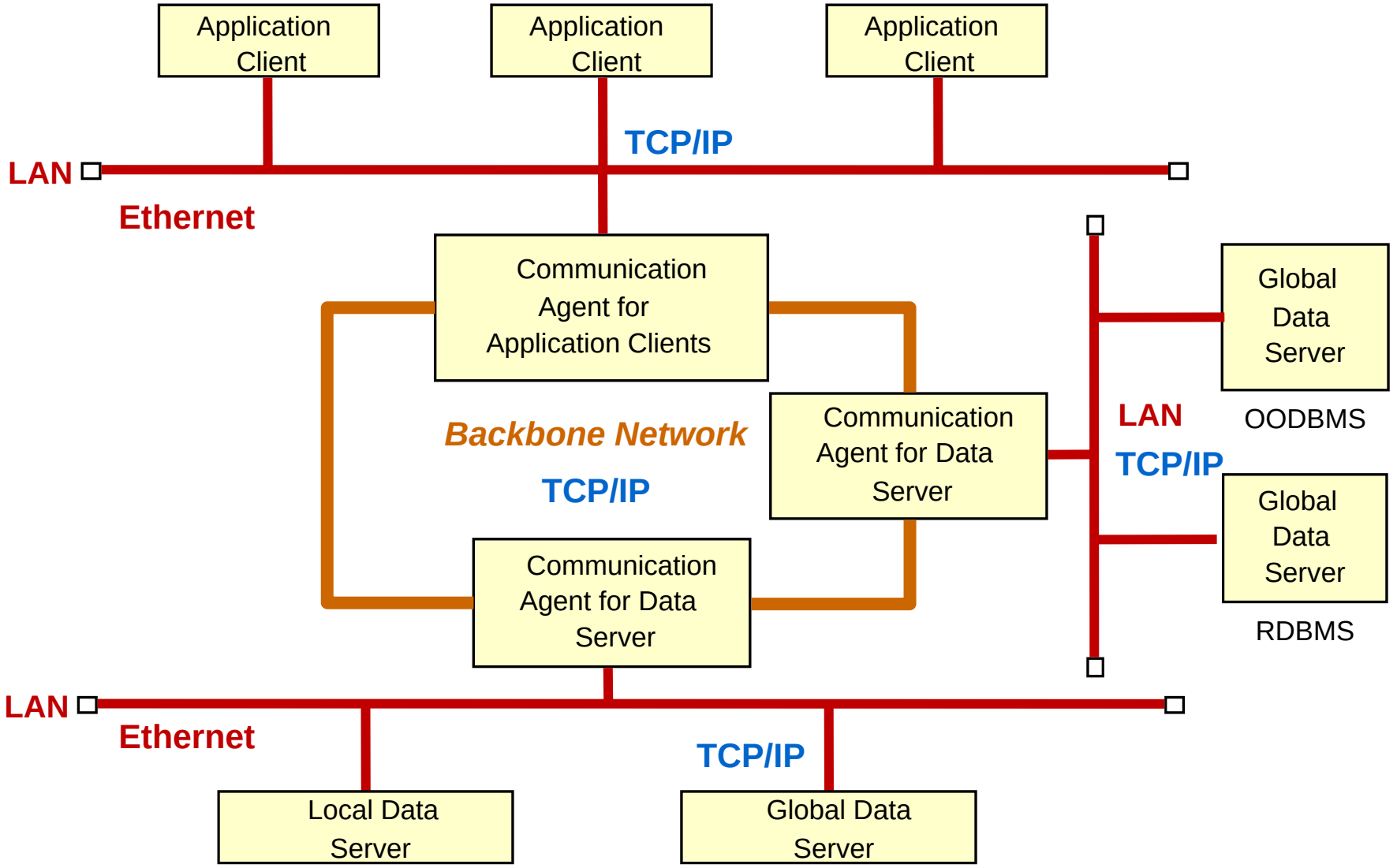
- Beschreibe die physikalische Topologie der Hardware
  - Entspricht oft der physikalischen Schicht in ISO's OSI Referenzmodell
    - ⇨ Welche Assoziationen werden auf physikalische Verbindungen abgebildet?
    - ⇨ Welche <<benutzt>>-Beziehungen aus dem Analyse-/Entwurfsmodell korrespondieren mit physikalischen Verbindungen?

- Beschreibe die logische Topologie (Assoziationen zwischen den Subsystemen)
  - Identifiziere Assoziationen, die nicht direkt auf physikalische Verbindungen abzubilden sind:
    - ⇨ Wie sollen diese Assoziationen implementiert werden?

Netzwerktopologie bei verteilten Systemen

- Wenn die Architektur verteilt ist, müssen wir auch die Netzwerkarchitektur beschreiben (Kommunikationssubsystem).
- Zu stellende Fragen
  - Was ist das Übertragungsmedium? (Ethernet, Wireless)
  - Welche “Quality of Service” (QOS) ist vorhanden/erforderlich? Welche Art von Kommunikationsprotokollen können genutzt werden?
  - Sollen Interaktion asynchron, synchron oder sperrend sein?
  - Welche Anforderungen gibt es an die Bandbreite zwischen den Subsystemen?
    - ⇨ Stock Price Change -> Broker
    - ⇨ Icy Road Detector -> ABS System

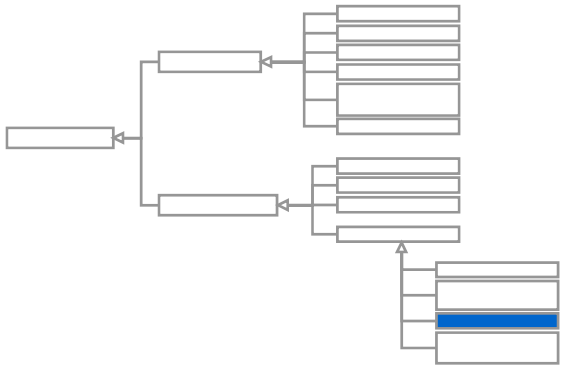
Typisches Beispiel für physikalische Topologie



Fragen zu Hardware/Software Zuordnung

- Ist bestimmte Funktionalität schon verfügbar (in Hard- oder Software)?
- Welche Hard-/ Softwaresysteme existieren
  - ... und können oder müssen genutzt werden
- Müssen Aufgaben auf bestimmter Hardware ausgeführt werden, um andere Hardware zu steuern oder nebenläufige Operationen zu erlauben?
  - Das ist für Embedded Systems oft der Fall.
- Welche Vernetzungstopologie besteht zwischen den physikalischen Einheiten?
  - Baum, Stern, Matrix, Ring
- Was ist das passende Kommunikationsprotokoll zwischen den Subsystemen?
  - Abhängig von Bandbreite, Latenz und gewünschter Sicherheit
- Generelle Frage zur Systemperformance:
  - Was ist die gewünschte Reaktionszeit?

Timing-Diagramme für Realzeitanforderungen

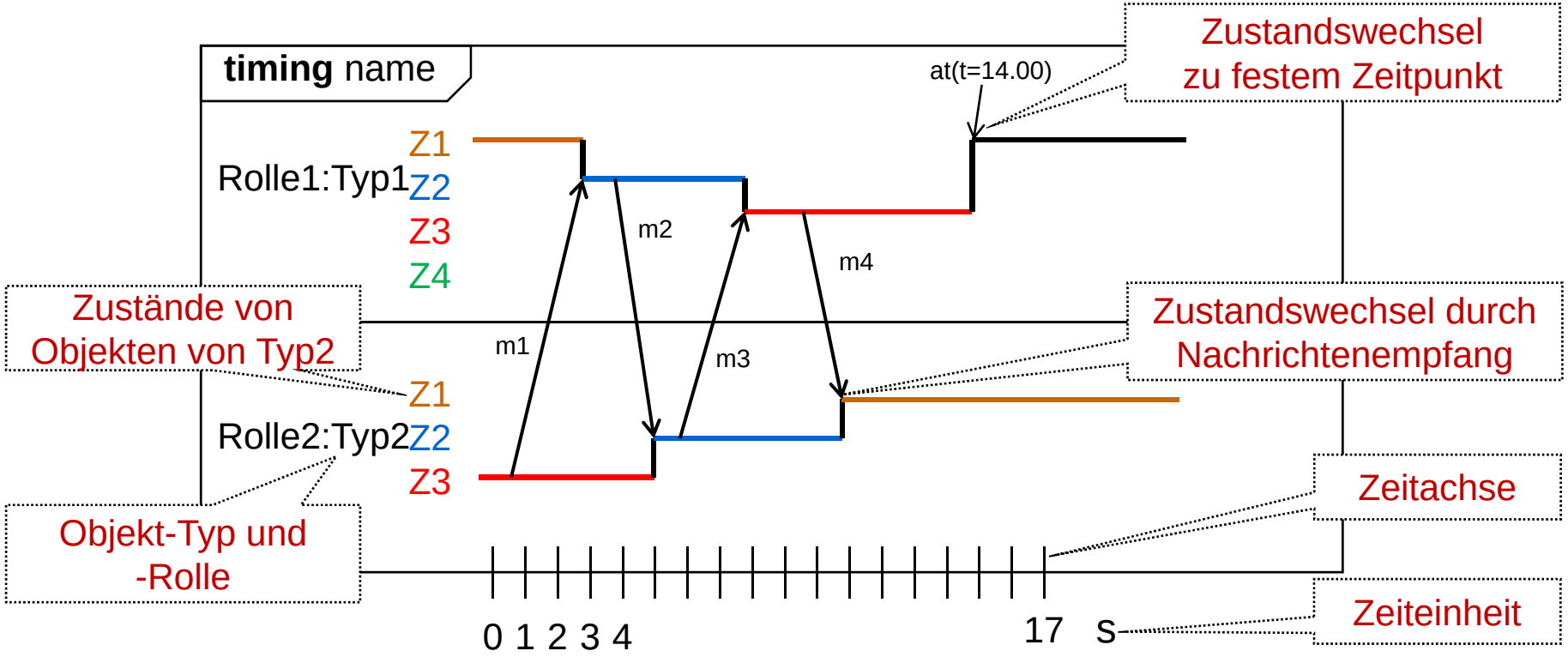


3a. Realzeitanforderungen

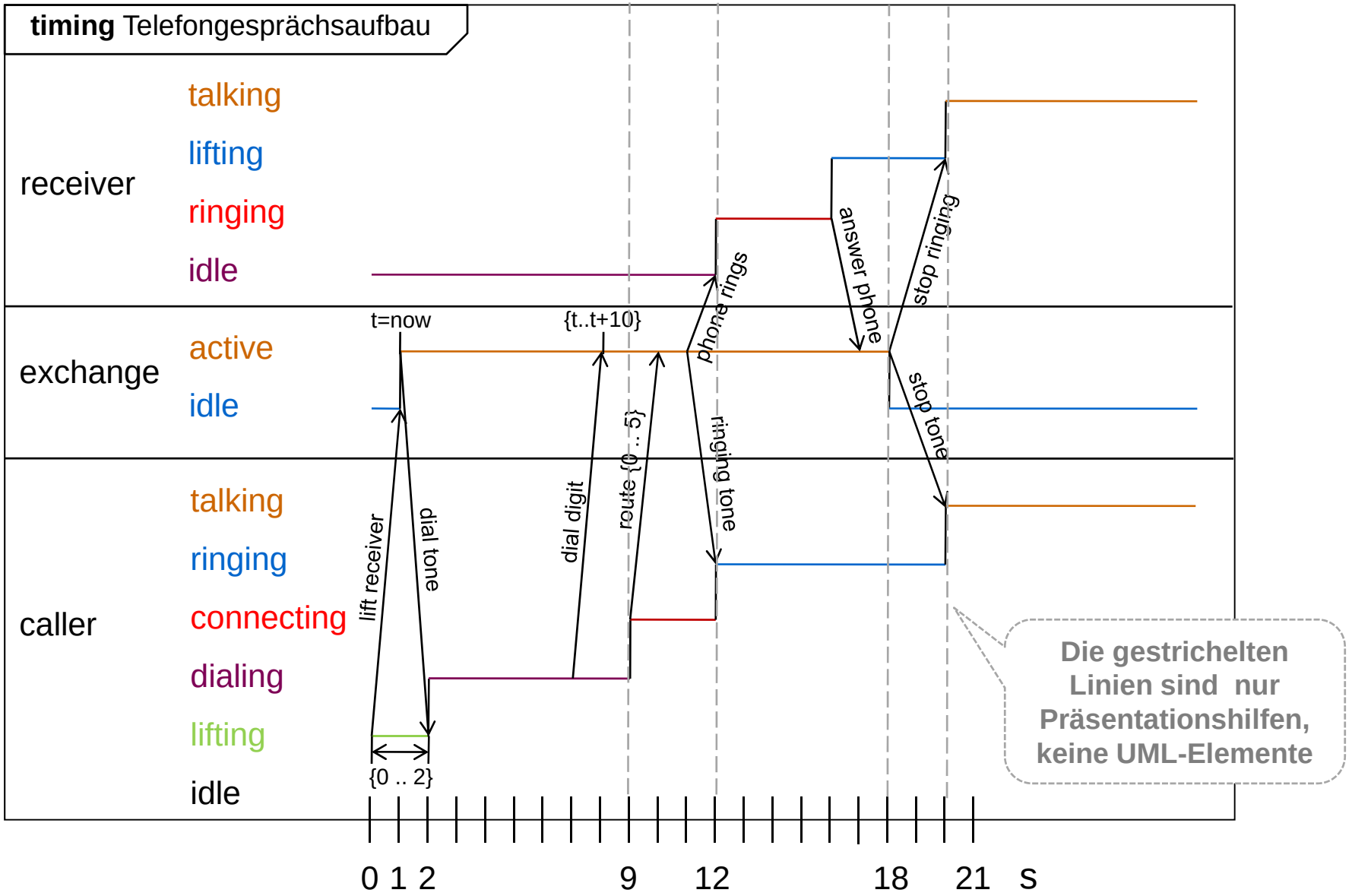
- Realzeitanforderungen bezeichnen Anforderungen an Software in einer bestimmten Zeitspanne zu reagieren oder etwas zu einem bestimmten Zeitpunkt zu tun
- Meistens geht es um sehr kurze Zeitspannen (Sekunden und alles darunter)
- Typischerweise bei "Eingebetteten Systemen" (Mischung aus Hard- und Software)
  - Im Fahrzeug- und Maschinenbau, Telekommunikation, Anlagen, …
- Dilemma
  - Software ist flexibler (leichter austauschbar und wartbar) und kostengünstiger
  - Aussagen über die absolute Zeit, die ein bestimmter Aufruf dauert sind aber sehr schwer zu treffen
    - ⇨ Problem "Dynamisches Binden": Was wird denn nun aufgerufen?
    - ⇨ Problem "Garbage Collection": Wann unterbricht sie evtl. einen Aufruf?
    - ⇨ …

Zeitverlaufsdiagramm (Timing Diagram)

- Modelliert zeitabhängige Zustandsübergänge der Interaktionspartner
  - Auslöser ist fester Zeitpunkt oder Nachrichtenaustausch
- Nützlich bei Interkationen mit zeitkritischem Verhalten (Realzeitanforderungen)
- Stellt exemplarischen Ablauf da (keine Kontrollstrukturen o.ä.)



Zeitdiagramm ▸ „Telefongespräch“

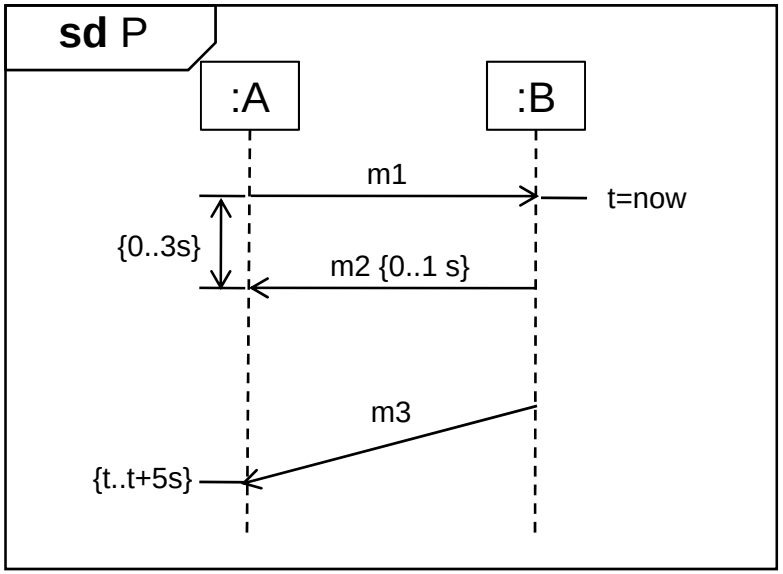


Sequenz- versus Zeitverlaufsdiagramm

Gleiche Zeitinformationen darstellbar (Zeitpunkt, relativer und absoluter Zeitraum)

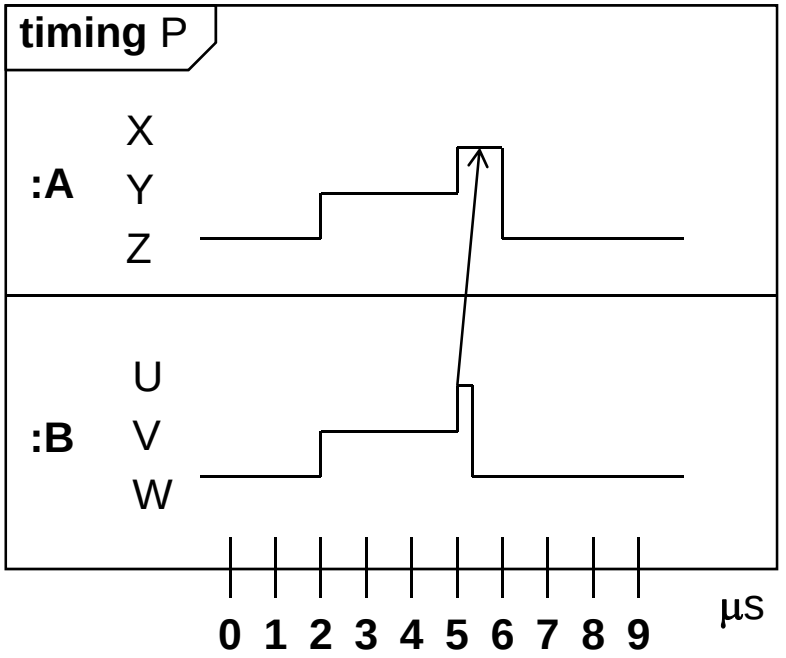
Sequenzdiaramm mit Zeit

- Kontrollstrukturen („Fragmente“ für Bedingung, Wiederholung, Abbruch, …)

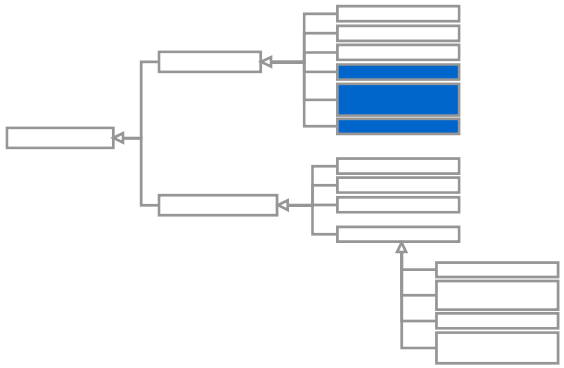


Timing Diagramm

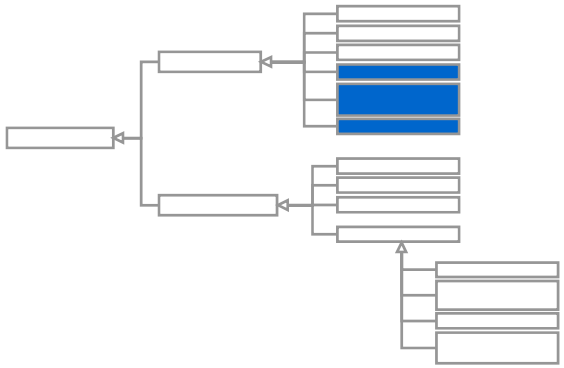
- Zusammenhang von Nachrichten und Zustandsübergängen



Verteilungsdiagramme (Deployment Diagrams)



Verteilungsdiagramme (Deployment Diagrams)



Verteilungsdiagramm (Einsatzdiagramm / Deployment Diagram)

- Zeigt
  - Ausführungsknoten (Rechner und Laufzeitumgebungen),
  - Kommunikationsbeziehungen zwischen Ausführungsknoten,
  - Manifestation (=Realisierung) von Komponenten durch Artefakte,
  - Einsatz von Artefakten auf Ausführungsknoten,
  - Konfiguration des Einsatzes,
  - sonstige Beziehungen (Abhängigkeits-Pfeile zeigen von der abhängigen Komponente weg)

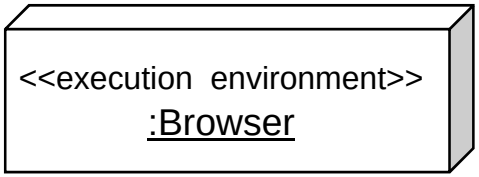
- Nutzen: Spezifikation der
  - Hardware/Software Zuordnung
  - Subsystemdekomposition
  - Verteilung im Netzwerk
  - Einsatz zur Laufzeit

Verteilungsdiagramm ▸ Knoten

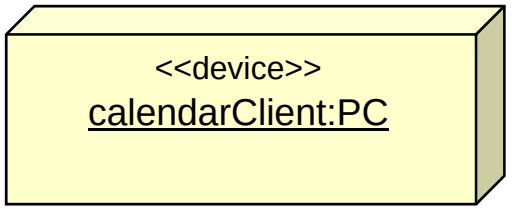
- Komponente
  - Logische Einheit mit expliziten Abhängigkeiten
  - Wie im Komponentendiagramm definiert

- Artefakt
  - Physische Einheit, z.B. Modell, Hilfetext, Quellcode, ausführbarer Code (class-file, jar-Archiv, o-file).
  - Realisierung einer oder mehrerer Komponenten

- Laufzeitumgebung („execution environment“)
  - Softwaresystem in dem Artefakte zum Einsatz kommen
  - Z.B. Java Virtual Machine, Applikationsserver, …

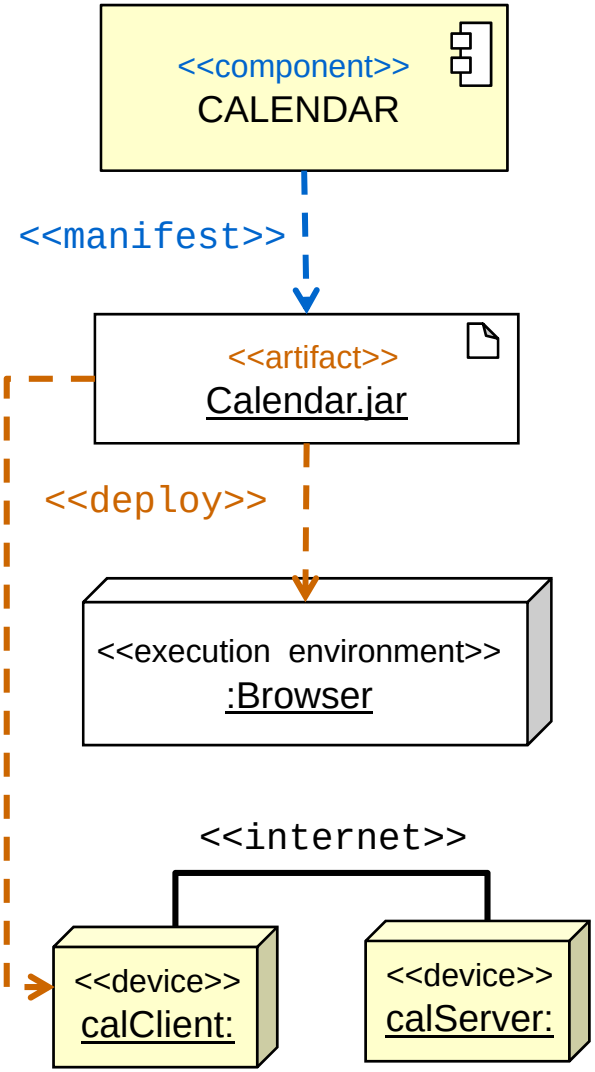


- Gerät („device“)
  - Physikalisches Gerät auf dem Artefakte zum Einsatz kommen (Rechner)

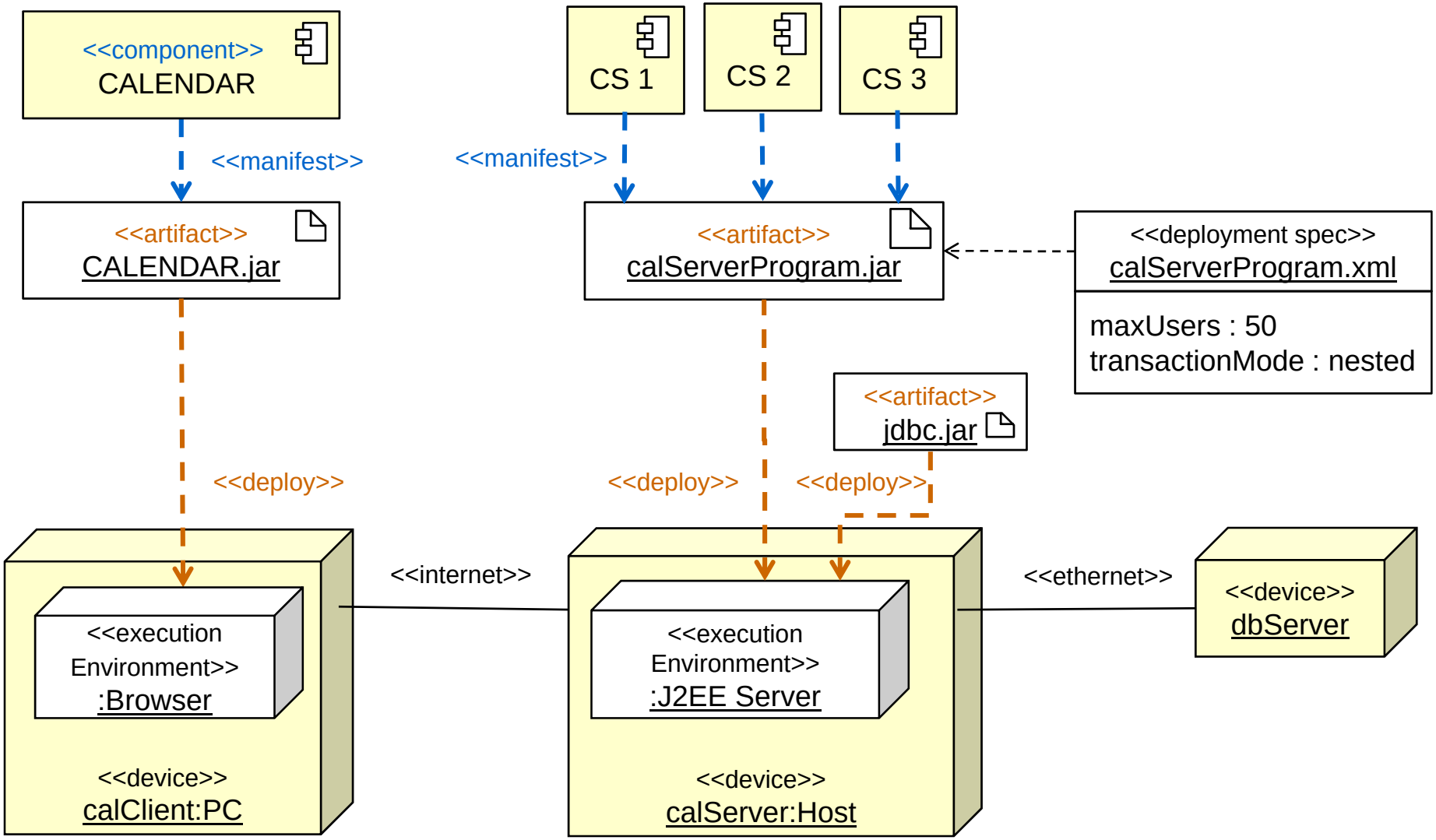


Verteilungsdiagramm ▸ Kanten

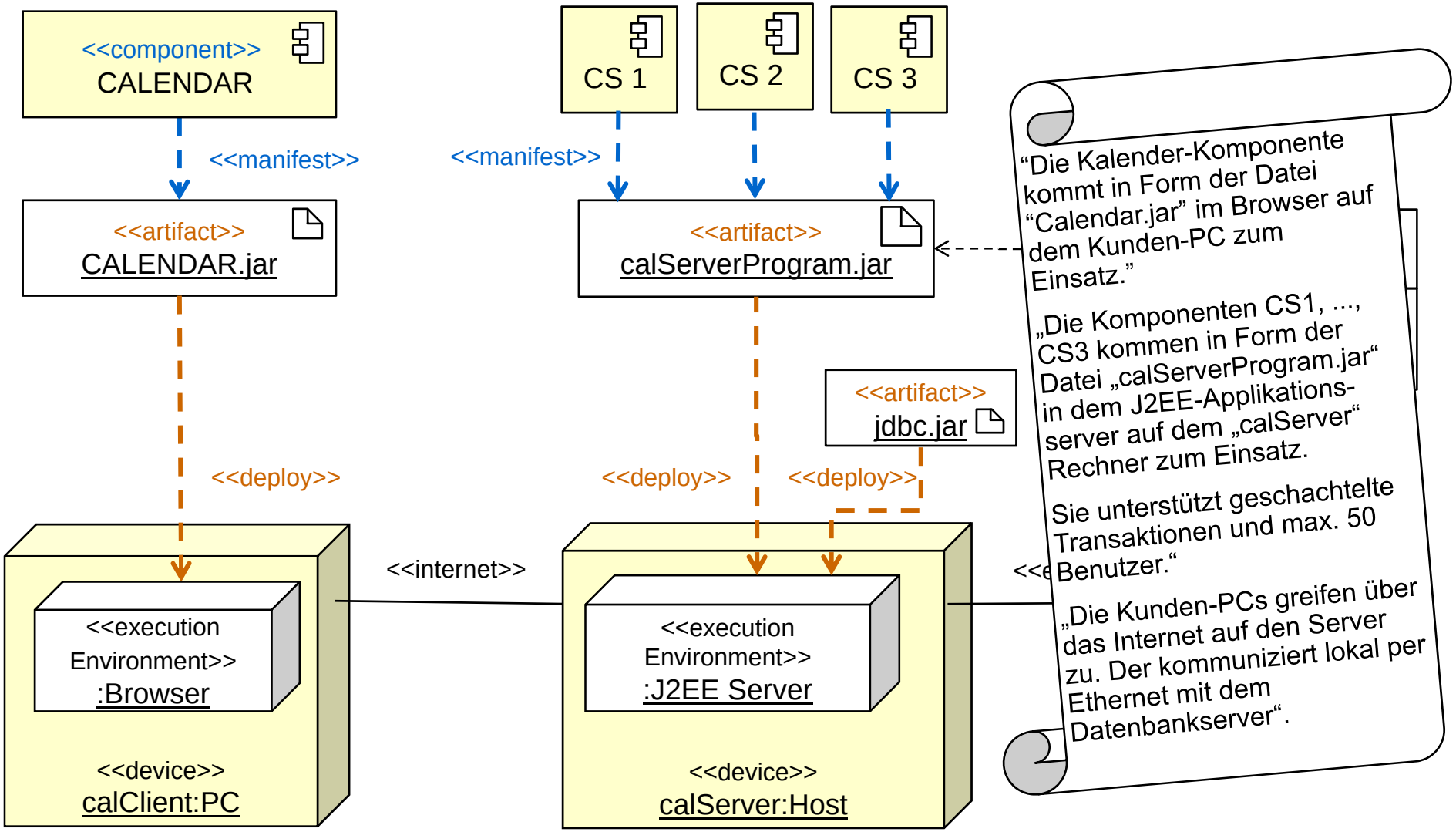
- Manifestation (<<manifest>>)
  - Komponente ist durch Artefakt realisiert
- Einsatzbeziehung (<<deploy>>)
  - Artefakt wird auf Ausführungsumgebung oder Gerät eingesetzt
- Kommunikationsbeziehung
  - Physische Verbindung über die Ausführungsumgebungen kommunizieren
  - Art kann als Stereotyp angegeben werden, z.B. <<internet>>, <<ethernet>>, ...



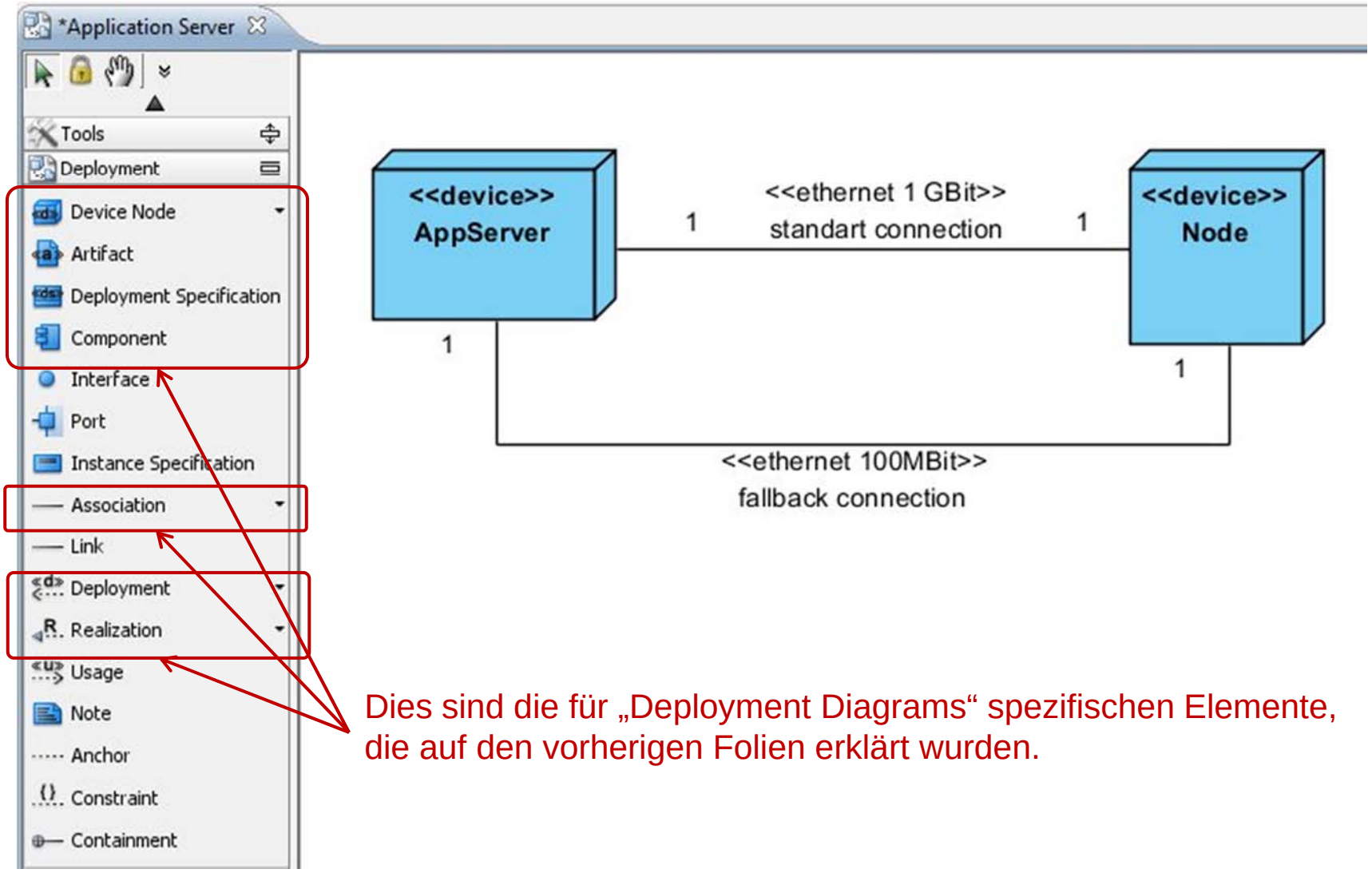
Verteilungsdiagramm ▸ Beispiel



Verteilungsdiagramm ▸ Beispiel ▸ Erläuterung



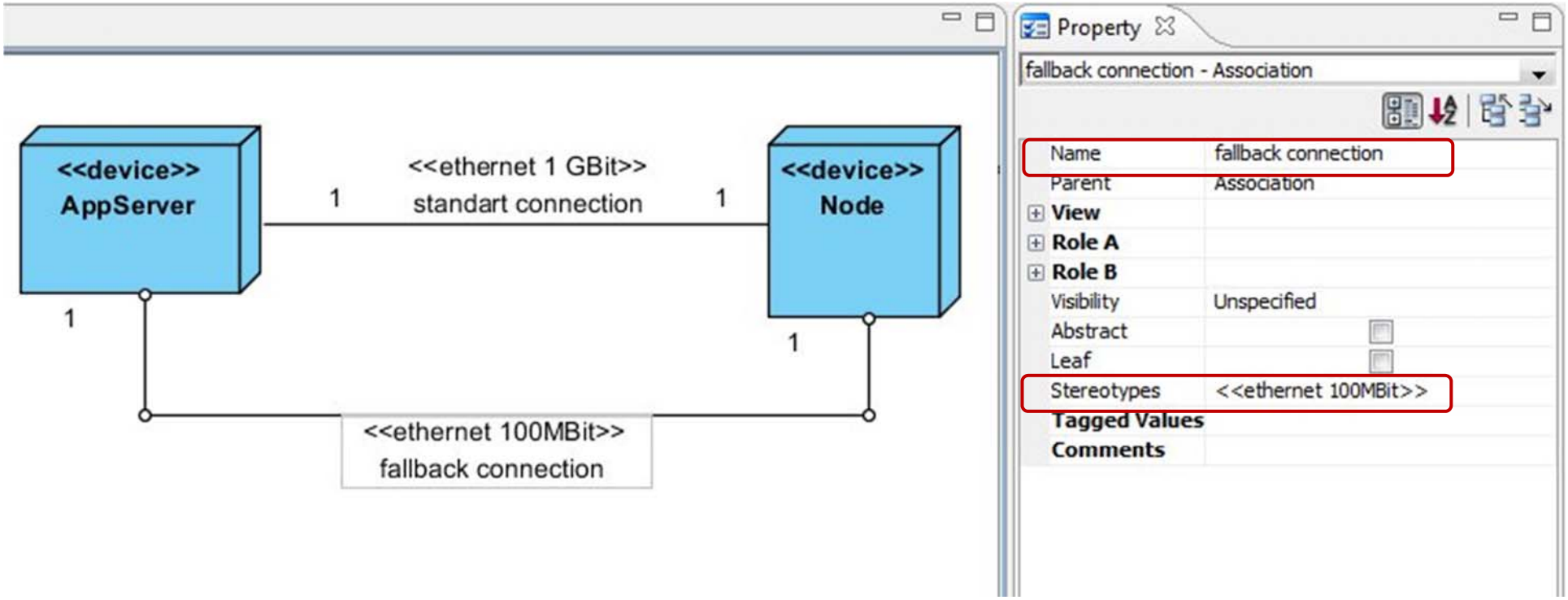
Visual Paradigm ▸ Tips I



Dies sind die für „Deployment Diagrams“ spezifischen Elemente, die auf den vorherigen Folien erklärt wurden.

Visual Paradigm ▸ Tips II

- Generell ist es wichtig zwischen Namen und Stereotyp einer Beziehung zu unterscheiden. Der Stereotyp gibt die Art einer Beziehung an. Der Name ist lediglich ein Bezeichner.
- Das gilt auch für Kommunikationsbeziehungen! Also schreiben Sie bitte nicht z.B. „ethernet“ in den Namen, sondern machen Sie es wie hier:



universitätbonn ROOTS

Datenmanagement

Datenmanagement

- Einige Objekte in den Modellen müssen persistent sein
  - Trenne sauber zwischen den Subsystemen, die Persistenz-Dienste anbieten und denen, die sie nutzen.
  - Definiere klare Schnittstellen.
- Ein nicht persistentes Objekt kann durch (interne) Datenstrukturen realisiert werden
- Ein persistentes Objekt kann folgendermaßen realisiert werden
  - Dateien
    - ⇨ Billig, einfach, permanente Speicherung
    - ⇨ Low level (Lese-/Schreiboperationen)
    - ⇨ Der Anwendung muss gegebenenfalls Code hinzugefügt werden, um eine angemessene Abstraktion zu realisieren
  - Datenbank
    - ⇨ Mächtig, leicht zu portieren
    - ⇨ Unterstützt mehrere Schreiber und Leser

Datei oder Datenbank?

- Persistenz via Dateien benutzt man für
  - Große, unstrukturierte Daten (Bitmaps, Core Dumps, Event Traces)
  - Daten mit geringer Informationsdichte (Archivdateien, Logdateien)
  - Daten, die nur kurzzeitig zu speichern sind

- Persistenz via Datenbanken benutzt man für
  - Strukturierte Daten (→ Relationen),
    die in verschiedenen Detailstufen (→ Sichten)
    von vielen Nutzern (→ Transaktionsmanagement)
    zugreifbar sein müssen
  - Daten, die von vielen Anwendungen (→ Transaktionsmanagement)
    benutzt werden
  - Daten, die auf verschiedenen Plattforme (→ Datenabstraktion)
    zur Verfügung stehen müssen

Was muss bei Benutzung einer Datenbank beachtet werden?

- Speicherplatz
  - Die Datenbank benötigt in etwa die dreifache Größe der Daten (→ Indices)
- Antwortzeit
  - Datenbanken sind I/O- oder kommunikationsabhängig (verteilte Datenbanken). Die Antwortzeit wird auch von der CPU Zeit, Locks und Verzögerungen durch häufige Bildschirmausgaben beeinflusst.
- Lock Arten
  - Pessimistisches Locking: Lock wird vor dem Zugriff auf ein Objekt gesetzt und erst wieder aufgehoben wenn der Zugriff beendet wurde.
  - Optimistisches Locking: Häufiger Lese- / Schreibzugriff (hohe Parallelität!) Wenn die Aktivität beendet wurde, prüft die Datenbank ob Konflikte bestehen; wenn ja werden alle Änderungen verworfen.
- Administration
  - Große Datenbanken benötigen ausgebildete Support Mitarbeiter, um Sicherheits-Mechanismen, Datenträgerplatz und Backups zu verwalten, die Leistung zu überwachen und Einstellungen anzupassen.

Fragen zum Datenmanagement

- Sollte die Datenbank verteilt sein?
- Sollte die Datenbank erweiterbar sein?
- Wie oft wird auf die Datenbank zugegriffen?
- Was ist die erwartete Anfragefrequenz? Im schlimmsten Fall (worst case)?
- Was ist die Größe einer typischen Anfrage und einer im worst case?
- Müssen die Daten archiviert werden?
- Muss die physikalische Lage der Datenbanken versteckt werden (Ortstransparenz)?
- Wird ein explizites Interface zum Zugriff auf die Daten benötigt?
- Was ist das Anfrageformat?
- Sollte die Datenbank relational oder objektorientiert sein?

Abbildung eines Objektmodells auf eine relationale Datenbank

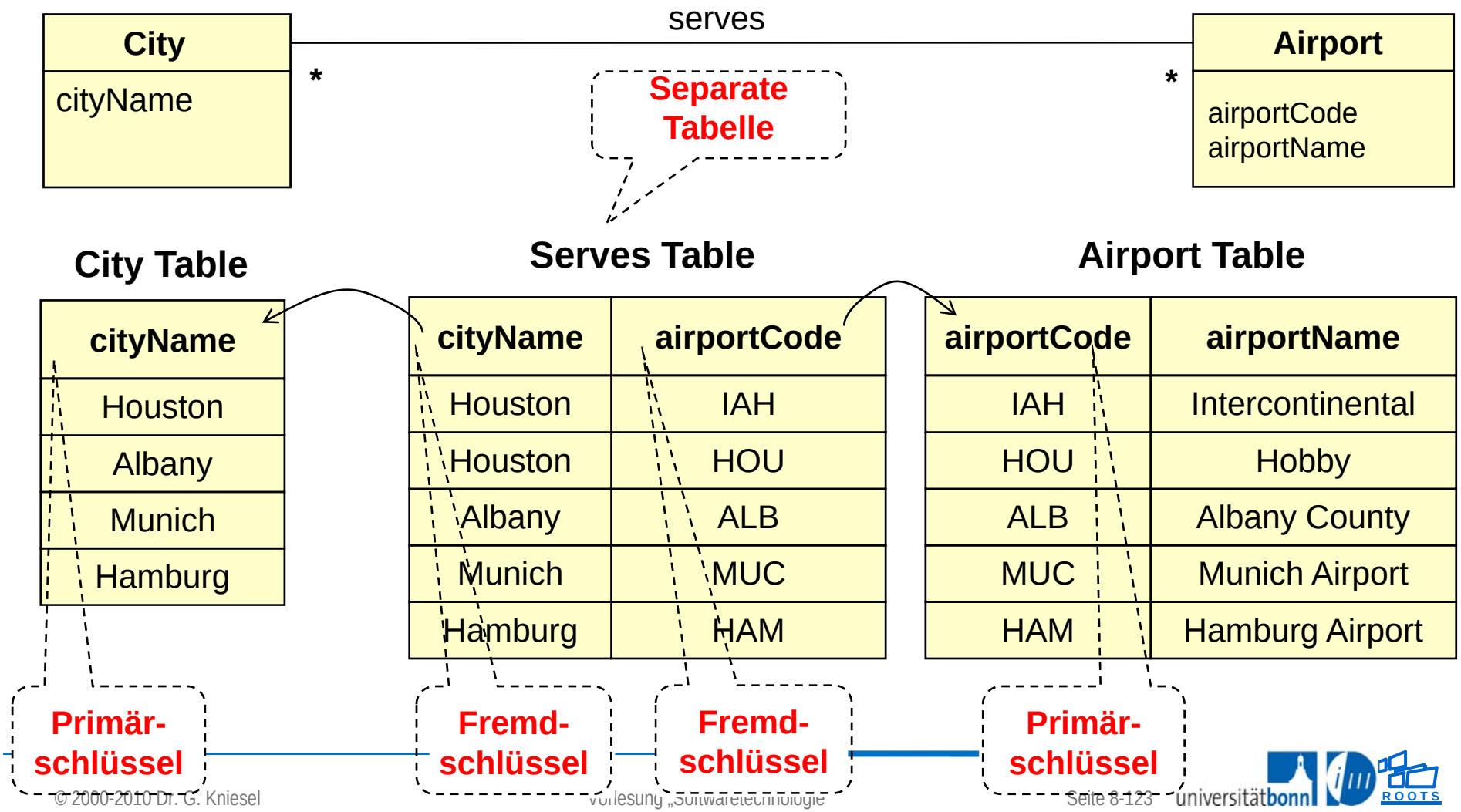
- UML Objektmodelle können auf relationale Datenbanken abgebildet werden:
  - Etwas Verlust entsteht, weil alle UML-Konstrukte auf ein einziges relationales Datenbankkonstrukt abgebildet werden – die Tabelle.

- UML Zuordnungen
  - Jede Klasse wird auf eine Tabelle abgebildet
  - Jedes Attribut einer Klasse wird auf eine Spalte abgebildet
  - Eine Instanz einer Klasse repräsentiert eine Zeile in der Tabelle
  - Eine n-zu-m Beziehung wird in eine eigene Tabelle abgebildet
  - Eine 1-zu-n Beziehung wird als Fremdschlüssel implementiert

- Methoden werden nicht abgebildet ☹
  - „Stored procedures“ können nicht einzelnen Relationen zugeordnet werden (keine Kapselung, kein dynamisches Binden, …)

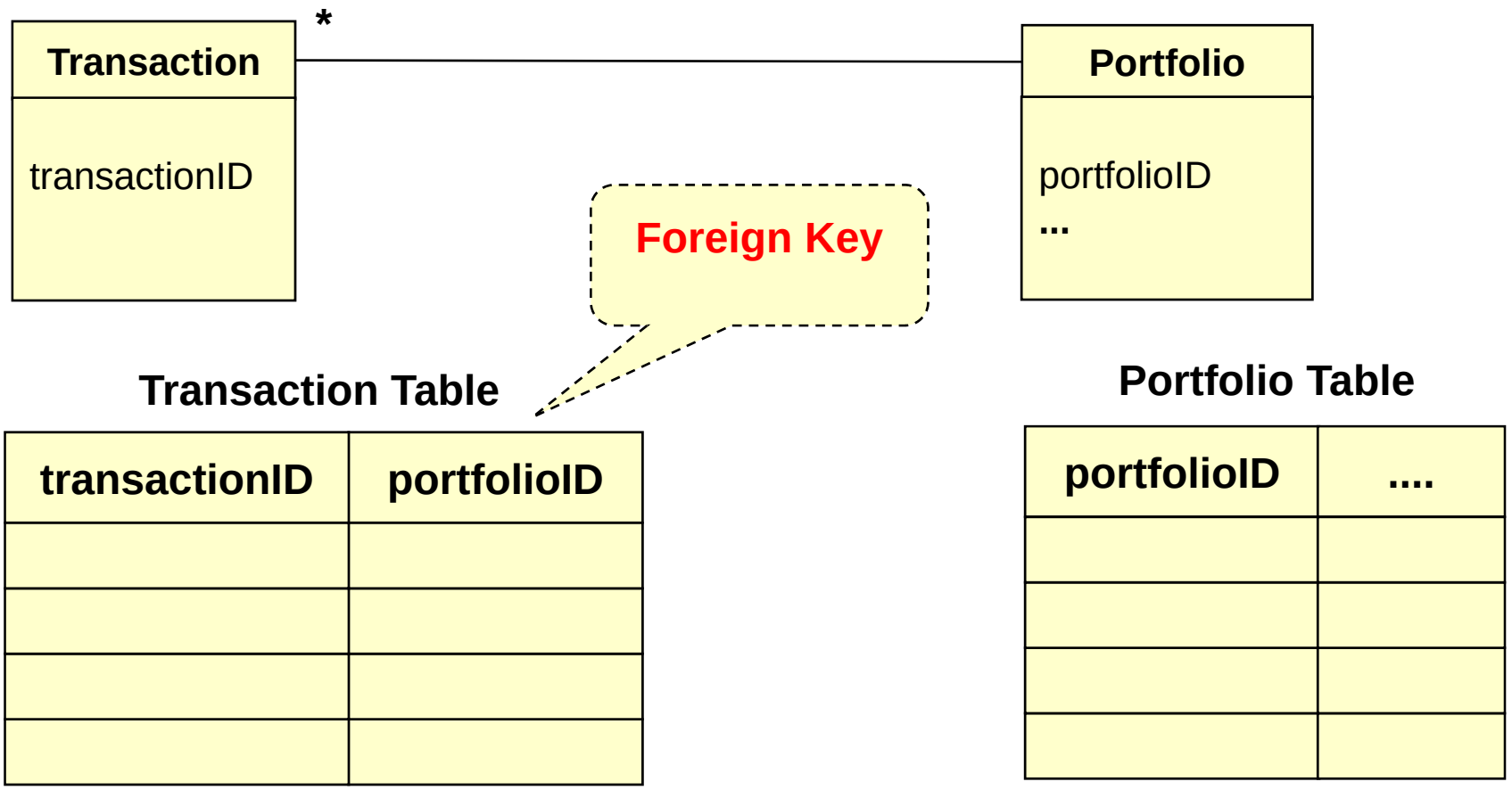
Von Objektmodellen zu Tabellen I

- N-zu-M Assoziation: Eigene Tabelle für die Assoziation



Von Objektmodellen zu Tabellen II

- 1-zu-n oder n-zu-1 Assoziationen: Verdeckte Fremdschlüssel



Von Objektmodellen zu Tabellen ▸ Werkzeuge

- Applikationsserver und ähnliche Werkzeuge erledigen die Abbildung eines Objektmodells auf ein relationales Schema („object relational mapping“) automatisch

- Gängige Systeme / Frameworks
  - Java Data Objects (JDO) – Teil der Java Enterprise APIs
  - Hybernate – Teil des Applikationsservers JBoss
  - ... Google nach „object relational mapping“ ...

universitätbonn ROOTS

Globale Ressourcenverwaltung

Globale Ressourcenverwaltung

- Resourcenverwaltung befasst sich mit Zugriffskontrolle
  - Sie beschreibt die Zugriffsrechte für verschiedene Akteure
  - Sie beschreibt, wie Objekte sich vor unberechtigtem Zugriff schützen

- Zugriffskontrollmatrix
  - Zeilen = Akteure
  - Spalten = Objekte
  - Inhalt der Felder = zulässige Operationen

	Objekttyp1	**Objekttyp 2**	**Objekttyp 3**
Actor A	Op1.1, Op1.2	---	Op3.1
Actor B	Op1.2	Op2.2, Op2.3	Op3.2
Actor C	---	Op2.1	Op3.3

Bsp: Actor C darf Operation Op2.1 auf Objekten des Typs Objekttyp2 ausführen.

Globale Ressourcenverwaltung: Realisierung der Zugriffskontrollmatrix

- Spaltenweise Aufteilung = Jedes Objekt weiss wer, was damit tun darf
  - Access Control Lists (Beispiel: „Unix“)

- Zeilenweise Aufteilung = Jeder Actor besitzt ein „Ticket“ das besagt, welche Operationen er ausführen darf
  - Capabilities (Beispiel: „Amoeba“)
  - Synonyme: „Capability “ / „Ticket“ / „Ausweis“/ „Schlüssel“

	Objekttyp1	**Objekttyp 2**	**Objekttyp 3**
Actor A	Op1.1, Op1.2	---	Op3.1
Actor B	Op1.2	Op2.2, Op2.3	Op3.2
Actor C	---	Op2.1	Op3.3

Bsp: Actor C darf Operation Op2.1 auf Objekten des Typs Objekttyp2 ausführen.

Fragen zu globalen Ressourcen

- Benötigt das System eine Authentifizierung?
- Wenn ja, welches Authentifizierungsschema?
  - Nutzername und Passwort? → Zugriffskontrollliste (ACL)
  - Tickets? → Capability-based
- Welche Benutzerschnittstelle für die Authentifizierung?
- Wann und wie wird ein Dienst dem Rest des Systems bekannt gemacht?
  - Zur Laufzeit?
  - Beim Kompilieren?
  - Über einen TCP-IP-Port?
  - Durch einen Namen?
- Benötigt das System einen netzweiten „Name Server“?

universitätbonn ROOTS

Bestimmung des Kontrollparadigmas (Programmsteuerung)

Bestimmung des Kontrollparadigmas (Programmsteuerung)

A) Implizite Kontrolle (deklarative Sprachen)

- Regelbasierte Systeme
- Logische Programmierung (Prolog)
- Datenbankabfragesprachen (SQL)

Prinzip: Sie programmieren Sachverhalte, nicht Algorithmen.

- Beispiel: Verwandschaftsbeziehungen in „Prolog“

Fakten

```
kurt  istVaterVon peter.
peter istVaterVon paul.
peter istVaterVon hans.
```

Regel

```
G istGrossvaterVon C :- G istVaterVon F, F istVaterVon C.
```

Implikation

Konjunktion

Anfragen

```
?- kurt istGrossvaterVon paul.
?- kurt istGrossvaterVon Wem.
?- Wer istGrossvaterVon paul.
```

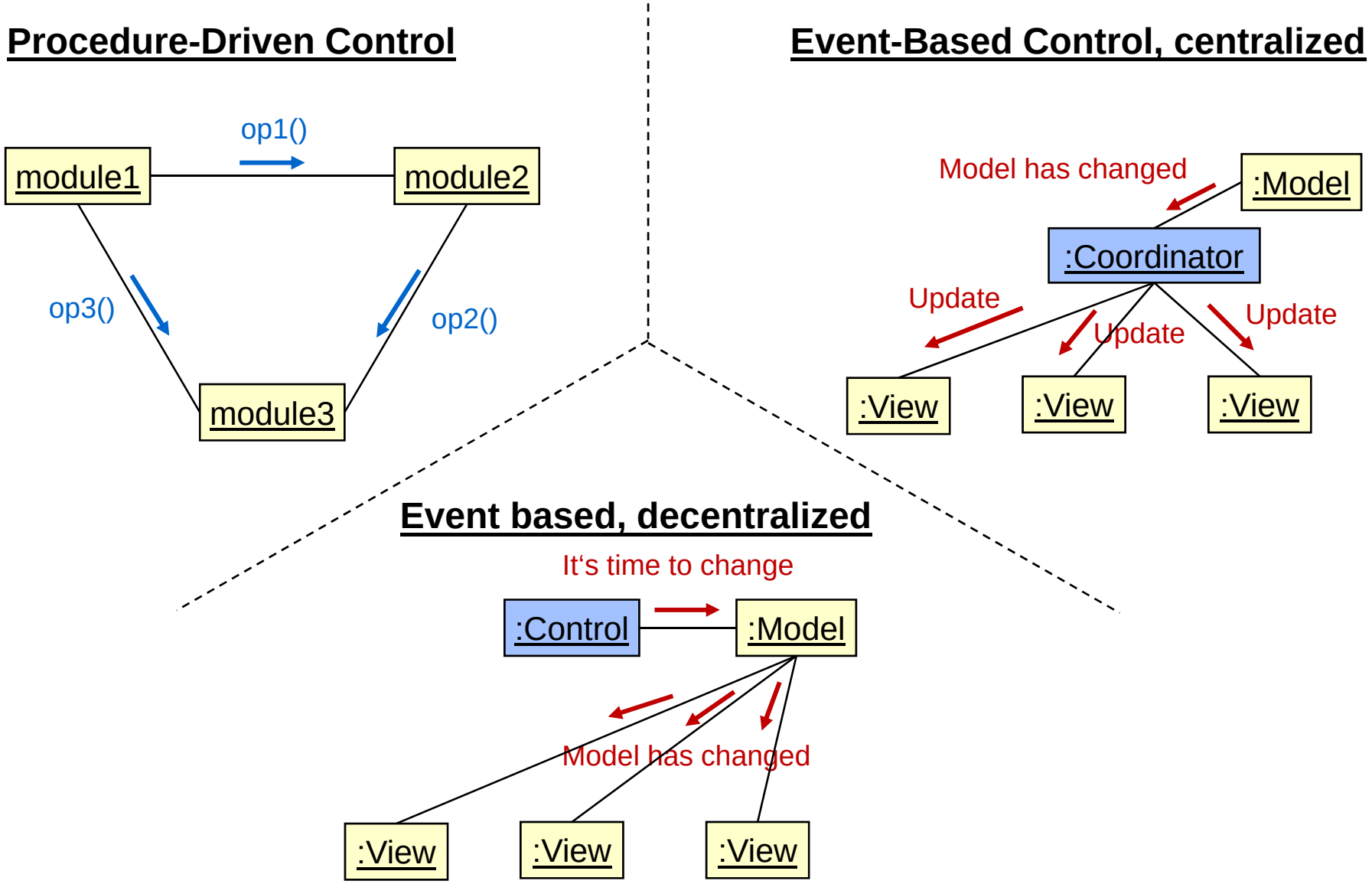
```
→ ja.
→ Wem = paul oder hans
→ Wer = kurt
```

Bestimmung des Kontrollparadigmas (Programmsteuerung)

B. Explizite Kontrolle (prozedurale und objektorientierte Sprachen)

- Zentrale Kontrolle
  - ⇨ Kontrolle befindet sich in einem Objekt / einer Komponente
- Dezentrale Kontrolle
  - ⇨ Kontrolle befindet sich in verschiedenen unabhängigen Objekten
  - ⇨ Geschwindigkeitsgewinn durch Parallelität versus mehr Kommunikation.
  - ⇨ Beispiel: Nachrichten-basiertes System
- Prozedurgesteuerte Kontrolle
  - ⇨ Kontrolle befindet sich im Programmcode.
  - ⇨ Beispiel: Hauptprogramm ruft Prozeduren in Subsystemen auf.
  - ⇨ Einfach, leicht zu bauen
- Eventgesteuerte Kontrolle
  - ⇨ Kontrolle sitzt in einem Dispatcher, der Funktionen von Subsystemen durch Rückfragen aufruft.
  - ⇨ Flexibel, gut für Benutzerschnittstellen

Prozedurgesteuerte vs. eventgesteuerte Kontrolle



Zentrale vs. dezentrale Kontrolle

- Welches von beiden soll man benutzen?
- Zentrale Kontrolle
  - Ein Kontrollobjekt oder Subsystem kontrolliert alles (“Spinne im Netz”)
  - Änderungen in der Kontrollstruktur sehr einfach durchzuführen
  - Möglicher Flaschenhals für die Performance
  - Mögliche Vermischung verschiedener Kontrollaufgaben („tangling“)
- Dezentrale Kontrolle
  - Kontrolle ist verteilt
  - Streut die Verantwortung
  - Passt gut in die objektorientierte Entwicklung
  - Übersicht geht eventuell verloren
  - Koordination schwierig

Grenzfälle

Grenzfälle

- Die meiste Zeit beschäftigt man sich beim Systementwurf mit dem Verhalten im Betriebszustand.

- Abschliessend muss man sich aber auch mit Grenzfällen befassen.
  - Initialisierung
    - ⇨ Beschreibt, wie das System aus einem nicht initialisierten Zustand in einen Betriebszustand gebracht wird ("startup use cases”).
  - Terminierung
    - ⇨ Beschreibt, welche Ressourcen vor der Beendigung aufgeräumt werden und welche Systeme benachrichtigt werden (“Terminierungs-Use Cases").
  - Fehler
    - ⇨ Viele mögliche Gründe: Programmierfehler, externe Probleme (Stromversorgung).
    - ⇨ Guter Systementwurf sieht fatale Fehler voraus (“Fehler-Use Cases”).

Fragen zu den Grenzfällen

- Initialisierung
  - Wie startet das System?
    - ⇨ Auf welche Daten muss beim Hochfahren zugegriffen werden?
    - ⇨ Welche Dienste müssen registriert werden?
  - Was tut die Benutzerschnittstelle beim Startvorgang?
    - ⇨ Wie präsentiert sie sich dem Nutzer?
- Terminierung
  - Dürfen einzelne Subsystemen terminieren?
  - Werden andere Subsysteme benachrichtigt, wenn ein einzelnes terminiert?
  - Wie werden lokale Updates der Datenbank mitgeteilt?
- Fehler
  - Wie verhält sich das System, wenn ein Knoten oder eine Kommunikationsverbindung ausfällt? Gibt es dafür Backupverbindungen?
  - Wie stellt sich das System nach einem Fehler wieder her? Unterscheidet dieser Vorgang sich von der Initialisierung?

Rückblick ▸ Zielgerichteter Systementwurf

Aktivitäten

- Identifikation von Nebenläufigkeit
- Hardware/Software Zuordnung
- Management persistenter Daten
- Globale Ressourcenverwaltung
- Wahl des Programmsteuerung
- Grenzfälle

Nutzen

- Jede Aktivität überprüft die gewählte Architektur in Hinsicht auf eine bestimmte Frage.
- Nach Beendigung dieser Aktivitäten können die Schnittstellen der Subsysteme abschließend definiert werden.
  - ➢ Start frei für den Objektentwurf der Subsysteme!

Rückblick ▸ Systementwurf (Gesamt)