universitätbonn

Kapitel 7b)
Systementwurf ▶ Architekturen

Stand: 21.11.2017

System-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 / Architektur
  - 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?
  - → 3. 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
- Pipes 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 beziehen

- Architekturanalysewerkzeuge
  - Identifikation von Schichten und Partitionen
  - Warnung vor Abhängigkeiten die der Schichtung entgegenlaufen

Schichten-Architekturen

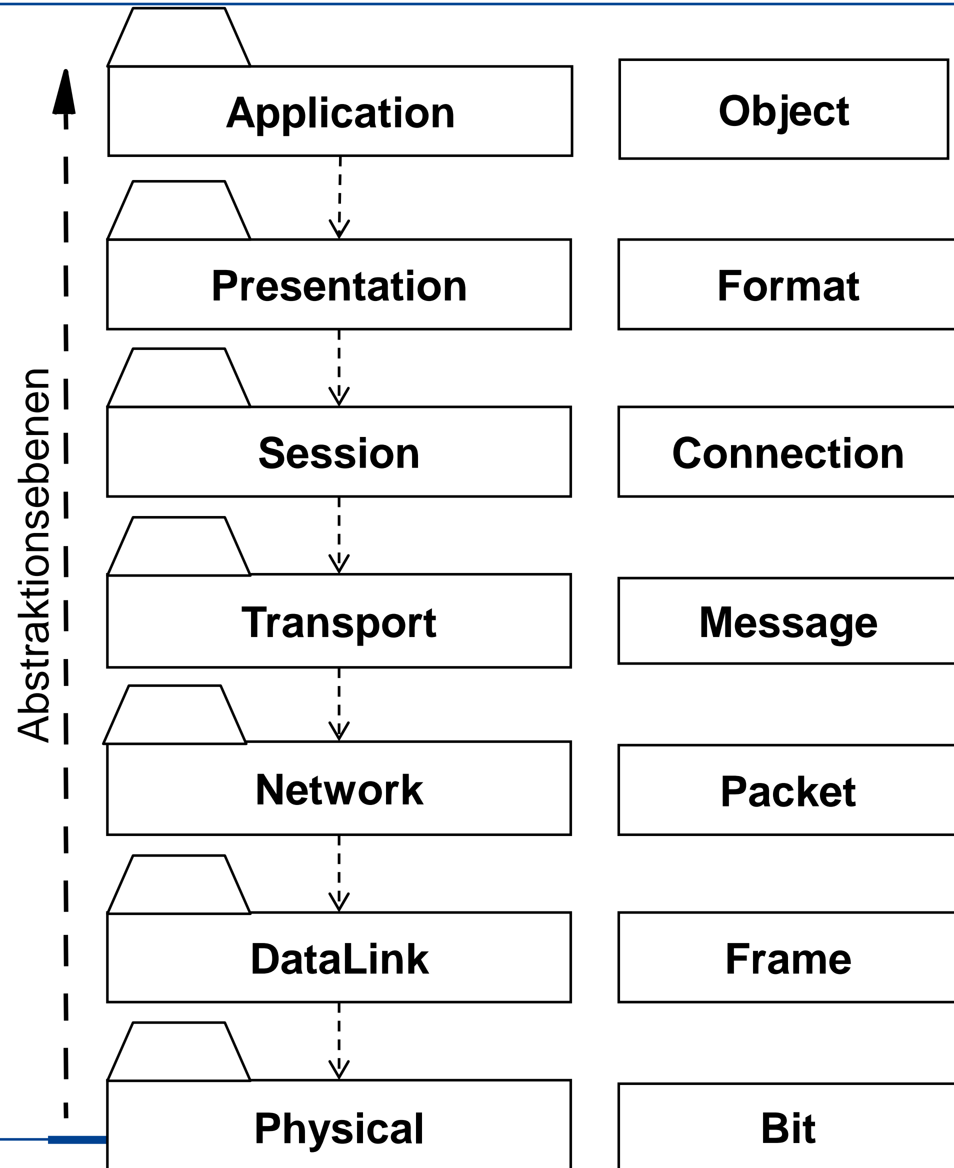
Geschichtete Systeme sind hierarchisch. Das ist wünschenswert, weil Hierarchie die Komplexität reduziert.

- 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.

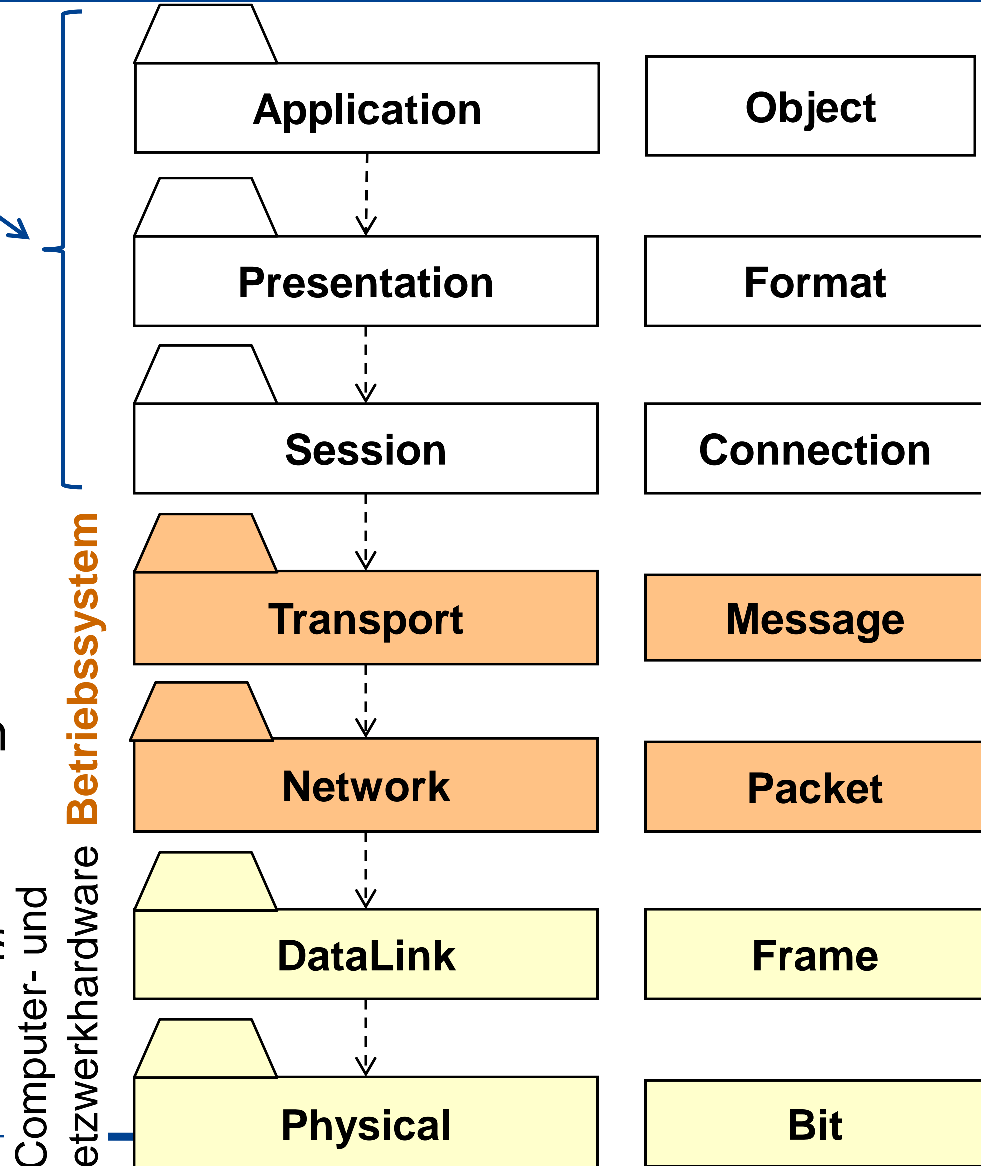


Geschlossene Schichten-Architektur
▸ Beispiel „Verteilte Kommunikation“

Verteilte Programmierung

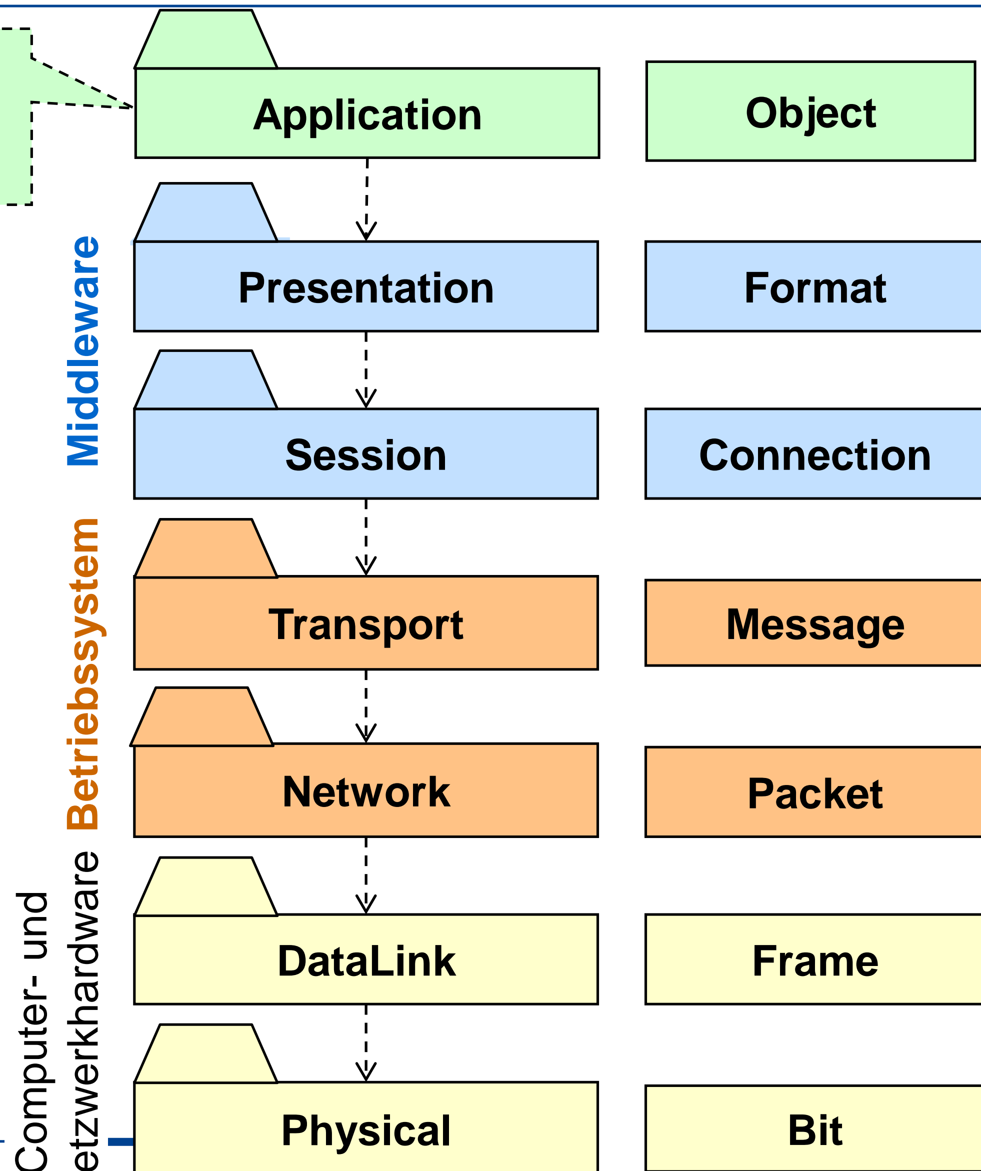
ist mühsam und fehleranfällig:

- *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



Middleware

Garantiert Transparenz der

- Verteilung
- Plattform

Plattform

Unterste Software- und Hardwareschichten

- 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

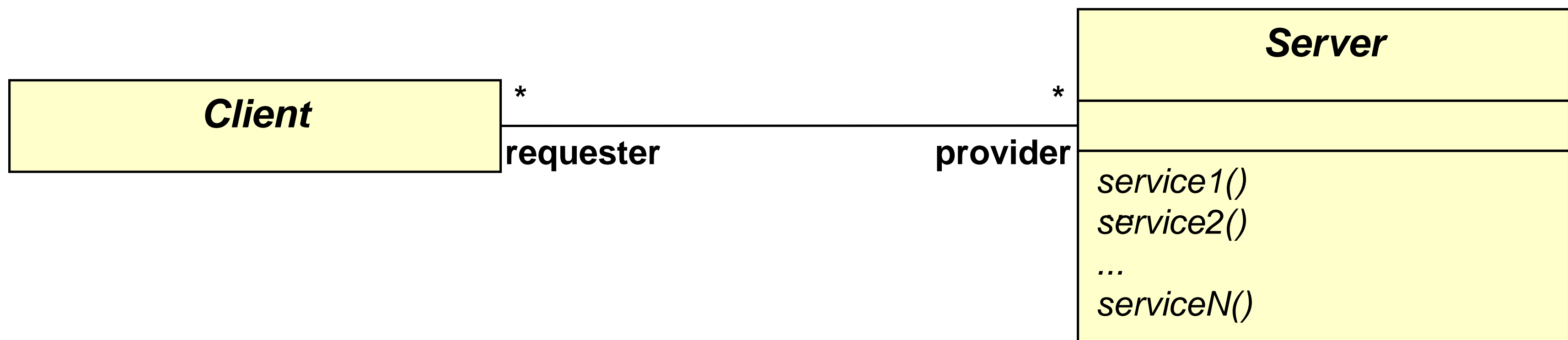
Applikationsserver

- 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

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)

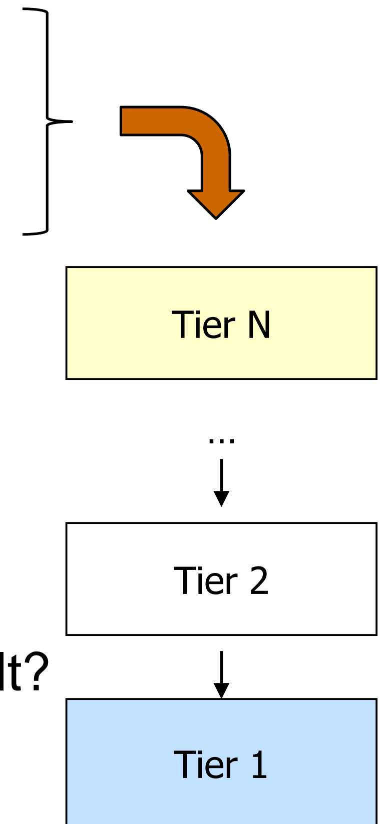
Entwurfsziele für Client/Server Systeme

- 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

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



universitätbonn

2-Tier Architektur

- Ältestes Verteilungsmodell: Client- und Server-Tier

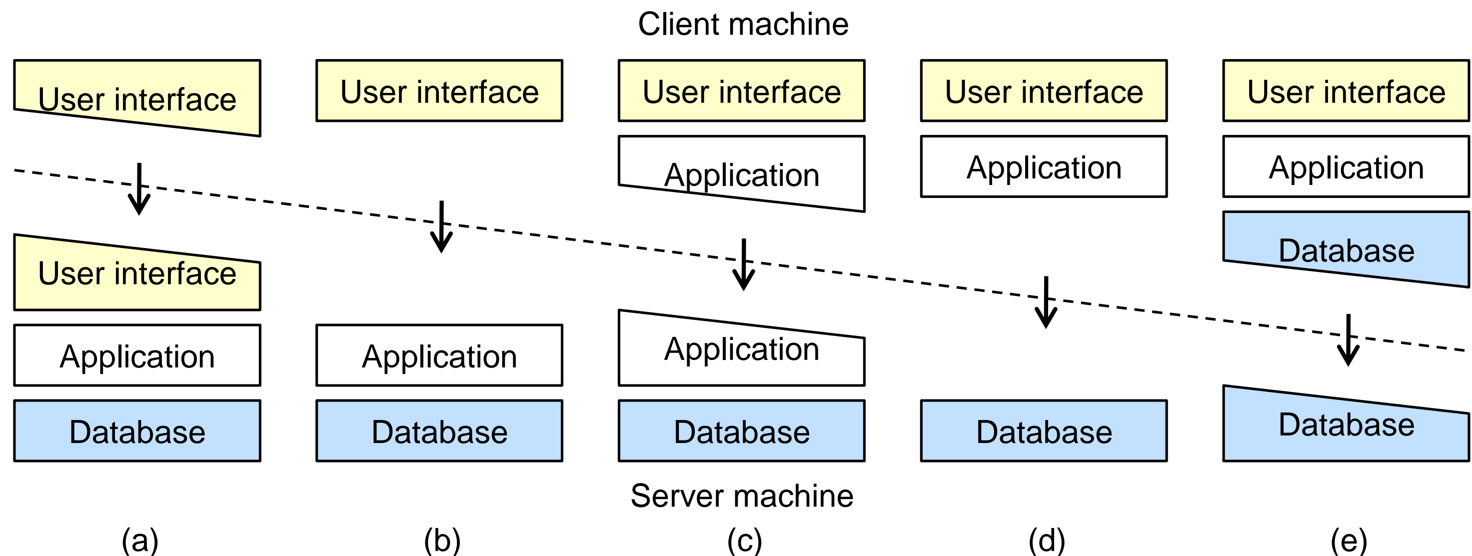
- Zuordnung von Aufgaben zu Tiers
  - Präsentation → Client
  - Anwendungslogik → Beliebig
  - Datenhaltung → Server

- Vorteile
  - Einfach und schnell umzusetzen
  - Performant

- Probleme
  - Schwer wartbar
  - Schwer skalierbar
  - Software-Update Problem

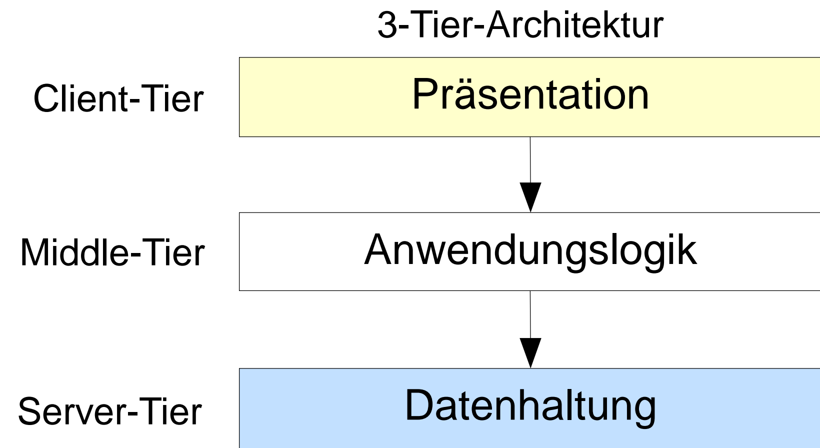
2-Tier Architektur ▸ Varianten

- 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

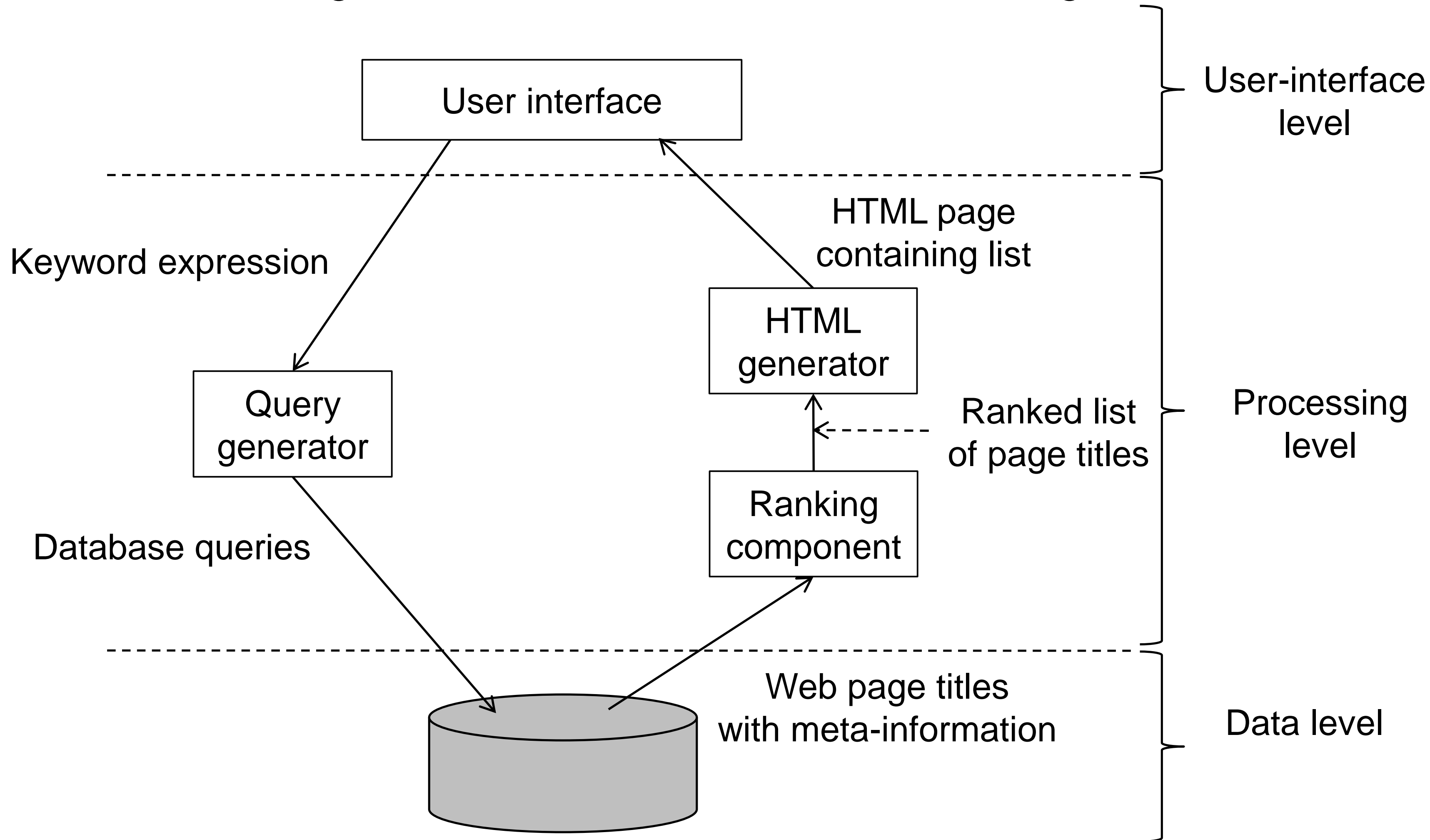
- Zuordnung von Aufgaben zu 3 Tiers
  - Präsentation ⇨ Client-Tier
  - Anwendungslogik ⇨ Middle-Tier
  - Datenhaltung ⇨ Server-Tier



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

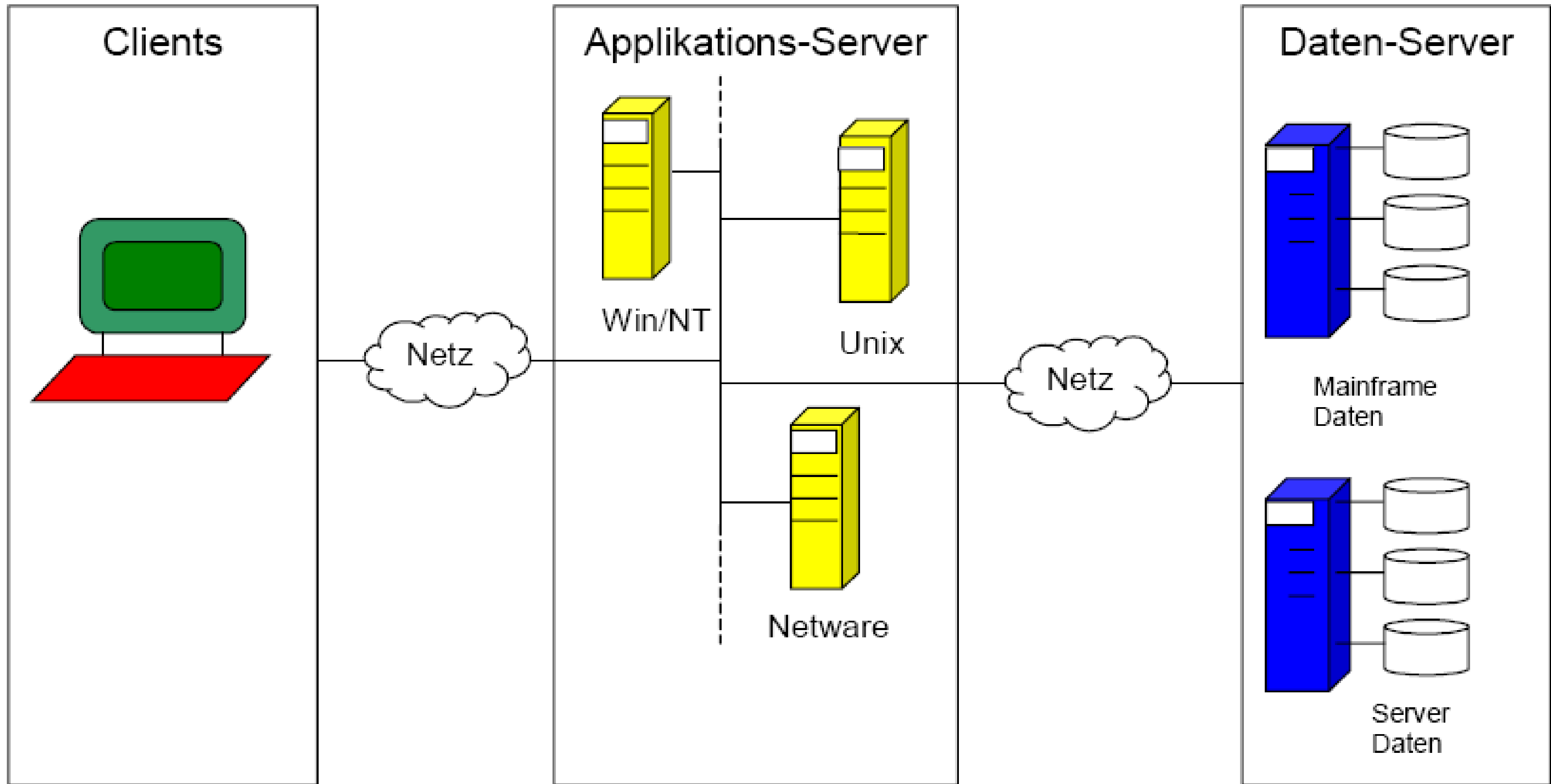
3–Tier Architekturen ▸ Beispiel Webanwendungen

- Standardverteilungsmodell für einfache Webanwendungen:



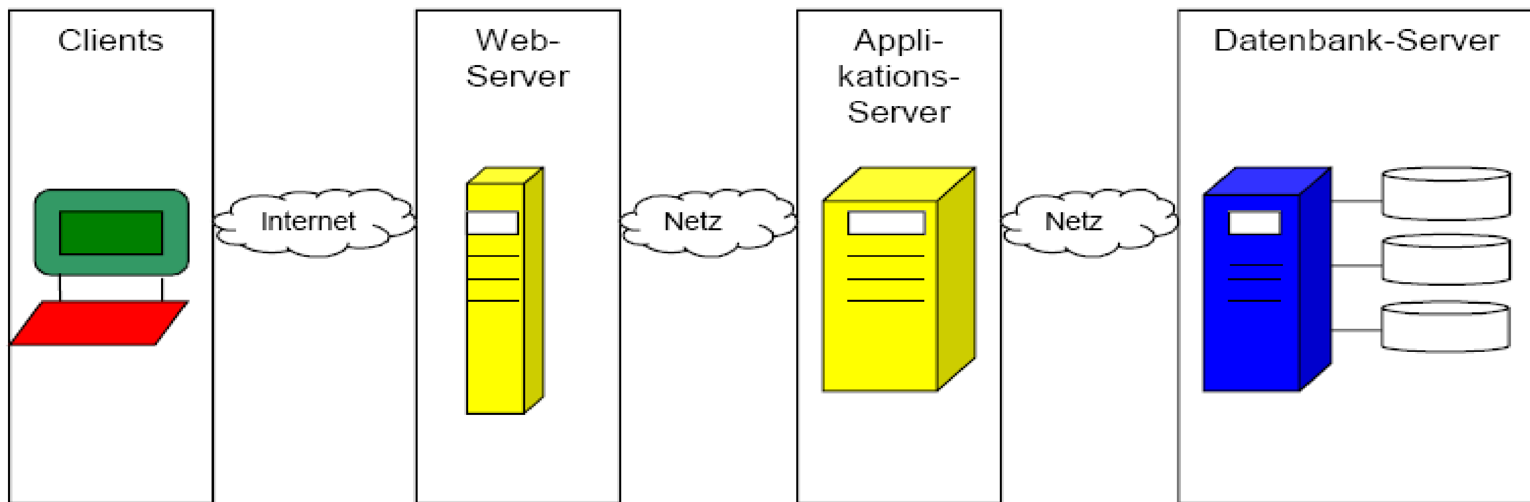
universitätbonn

3–Tier Architekturen ▸ Applications-Server (EJB)



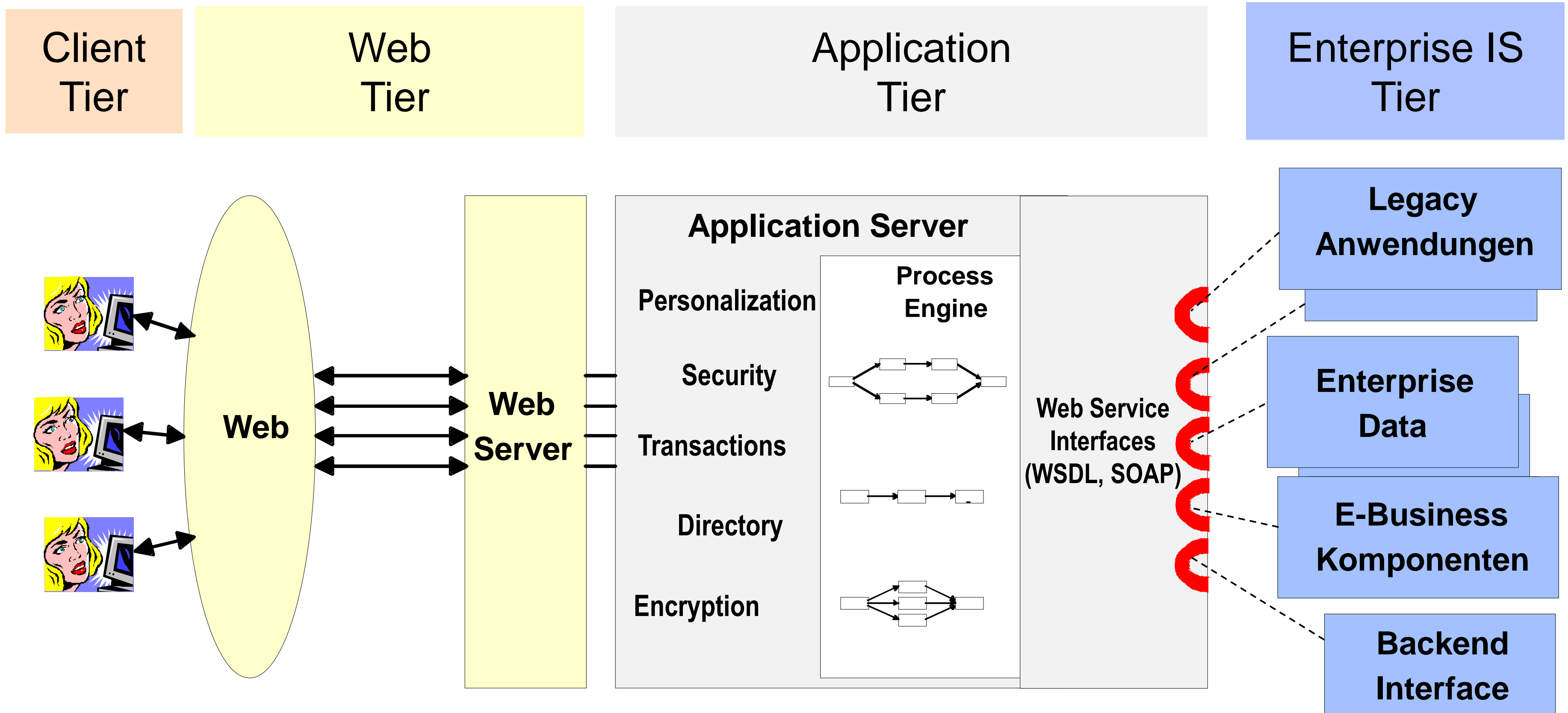
4- und Mehr-Tier Architekturen

- 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



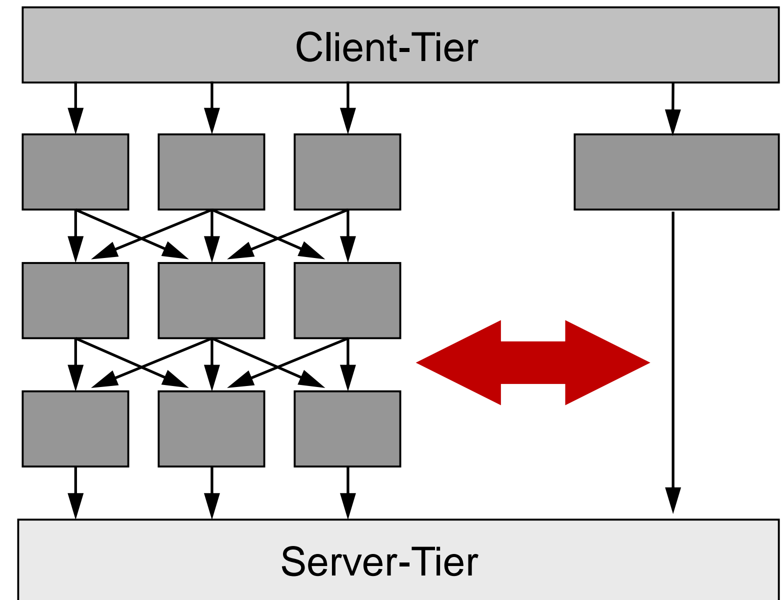
N-Tier-Architektur ▸ Abwägungen

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.

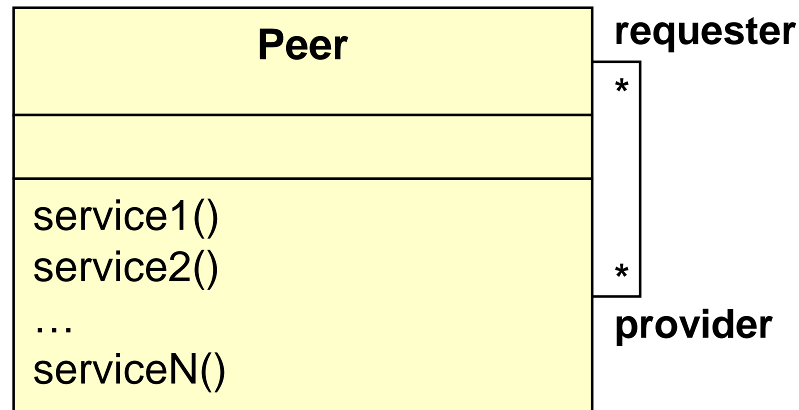
Probleme mit Client/Server Architekturen

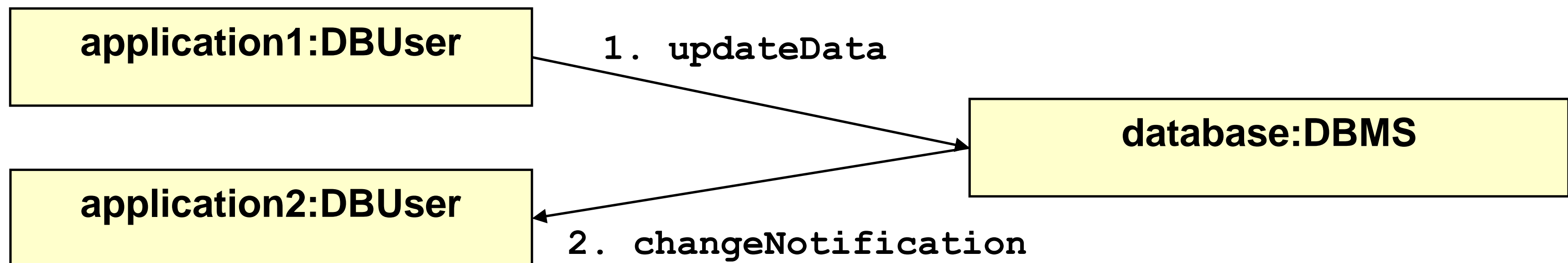
- Geschichtete Systeme unterstützen keine gleichberechtigte gegenseitige („Peer-to-peer“) Kommunikation

- „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.

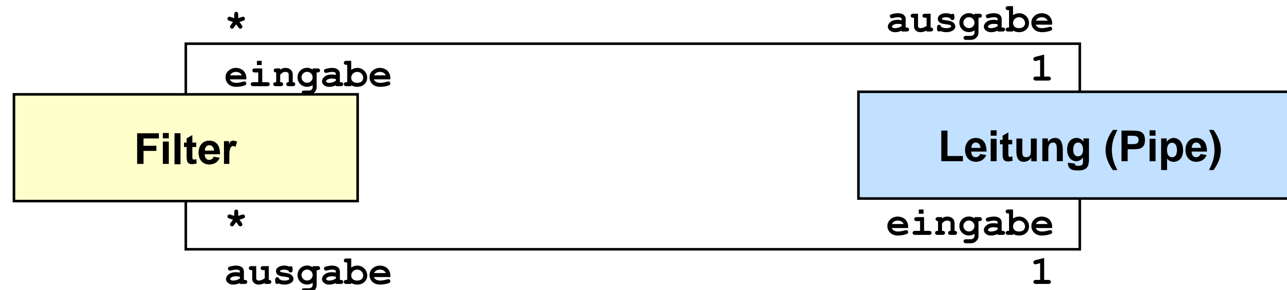
Peer-to-Peer Architektur

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





Pipe-and-Filter-Architecture

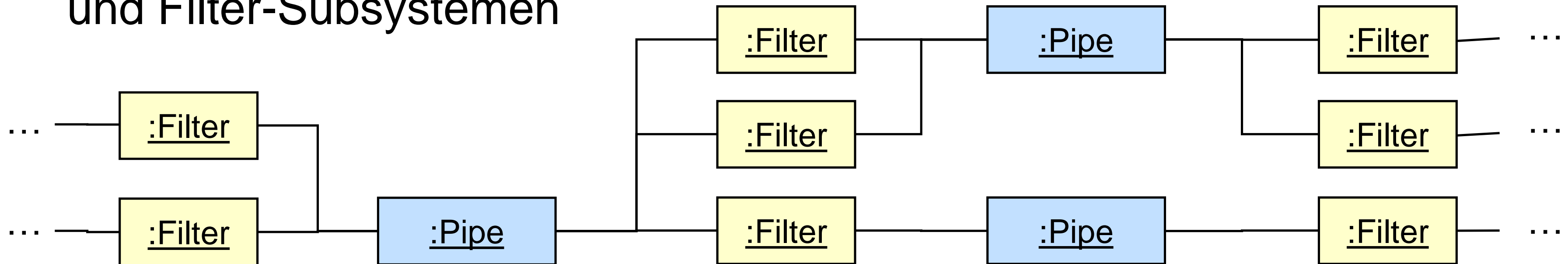


- Filter-Subsysteme bearbeiten Daten
  - Es sind reine Funktionen
  - Sie sind konzeptionell und implementierungstechnisch unabhängig von
    - ⇨ den Pipes die die Daten zu ihnen bzw. von ihnen Weg leiten
    - ⇨ den Erzeugern und Verbrauchern der Daten

- 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

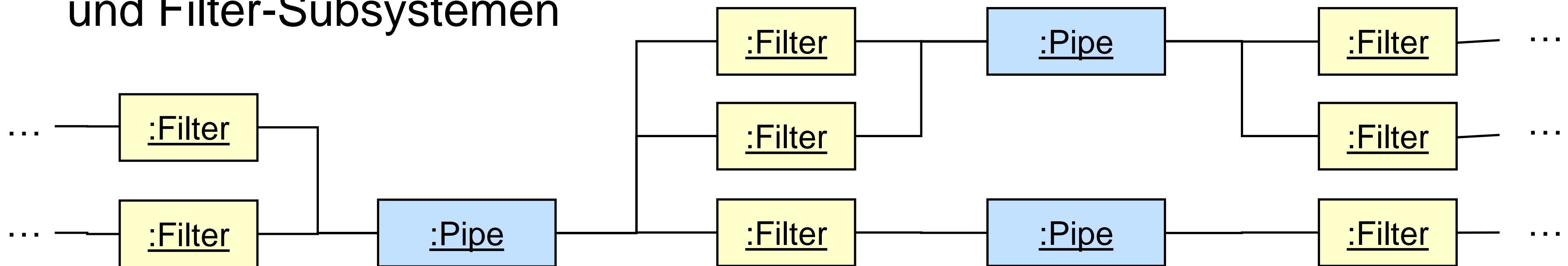
- Das Gesamtsystem entsteht einfach durch die Verknüpfung von Pipe- und Filter-Subsystemen



- 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

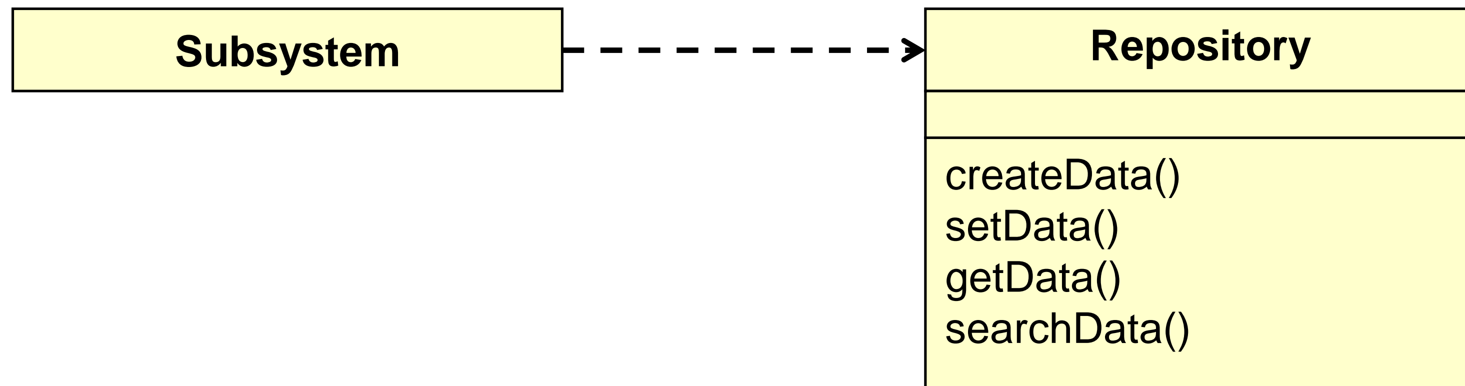
- Das Gesamtsystem entsteht einfach durch die Verknüpfung von Pipe- und Filter-Subsystemen



- 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“

Compiler

SemanticAnalyzer

SyntacticAnalyzer

Optimizer

LexicalAnalyzer

CodeGenerator

Repository

ParseTree

SymbolTable

Type Info

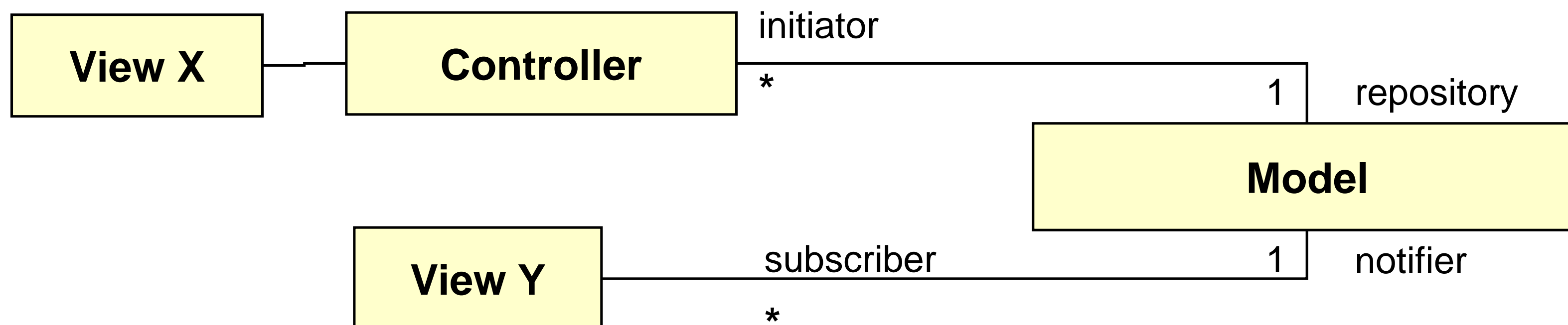
SourceLevelDebugger

Editor

© 2000-2017 Dr. G. Kniesel

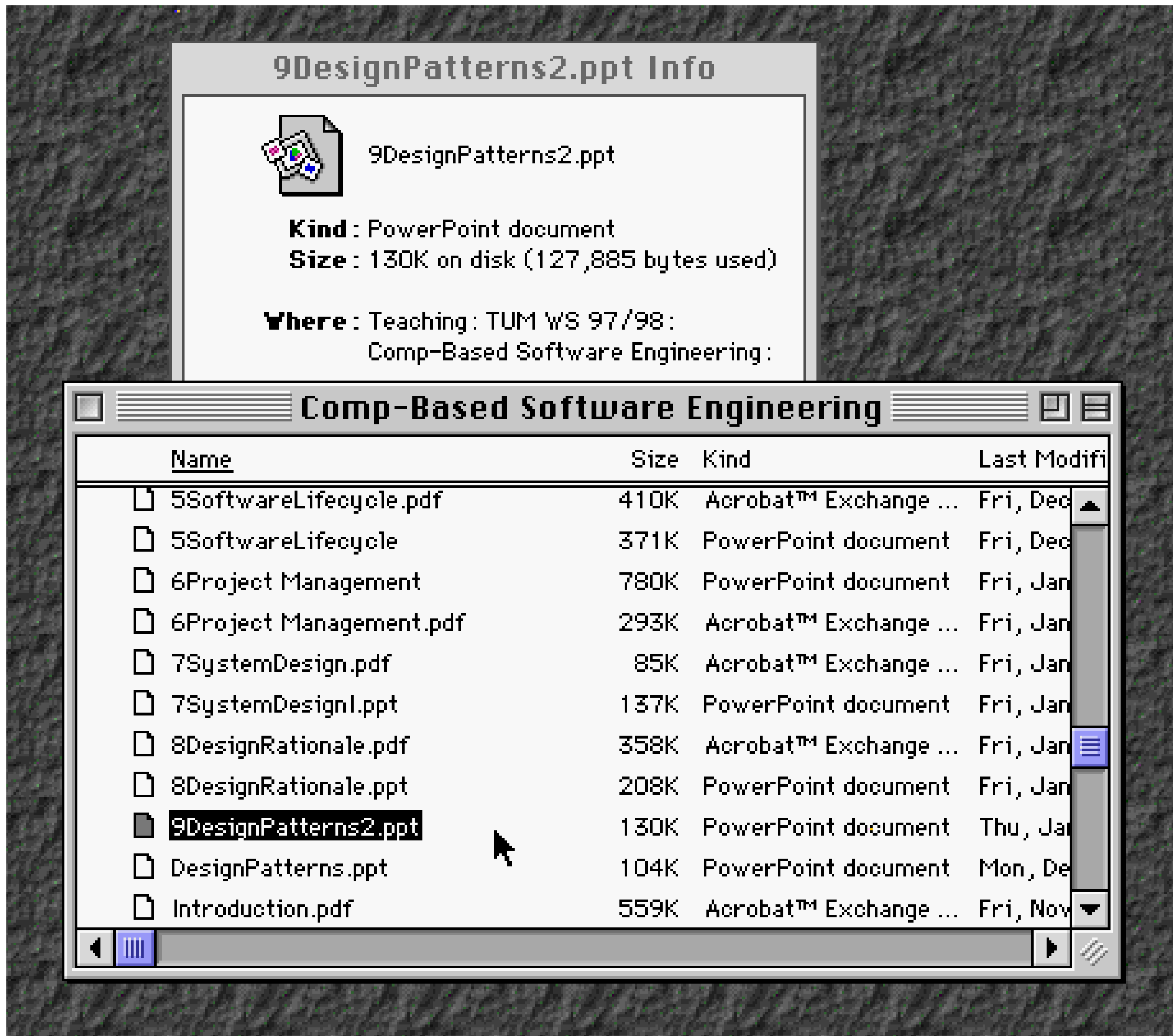
Model/View/Controller

- 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



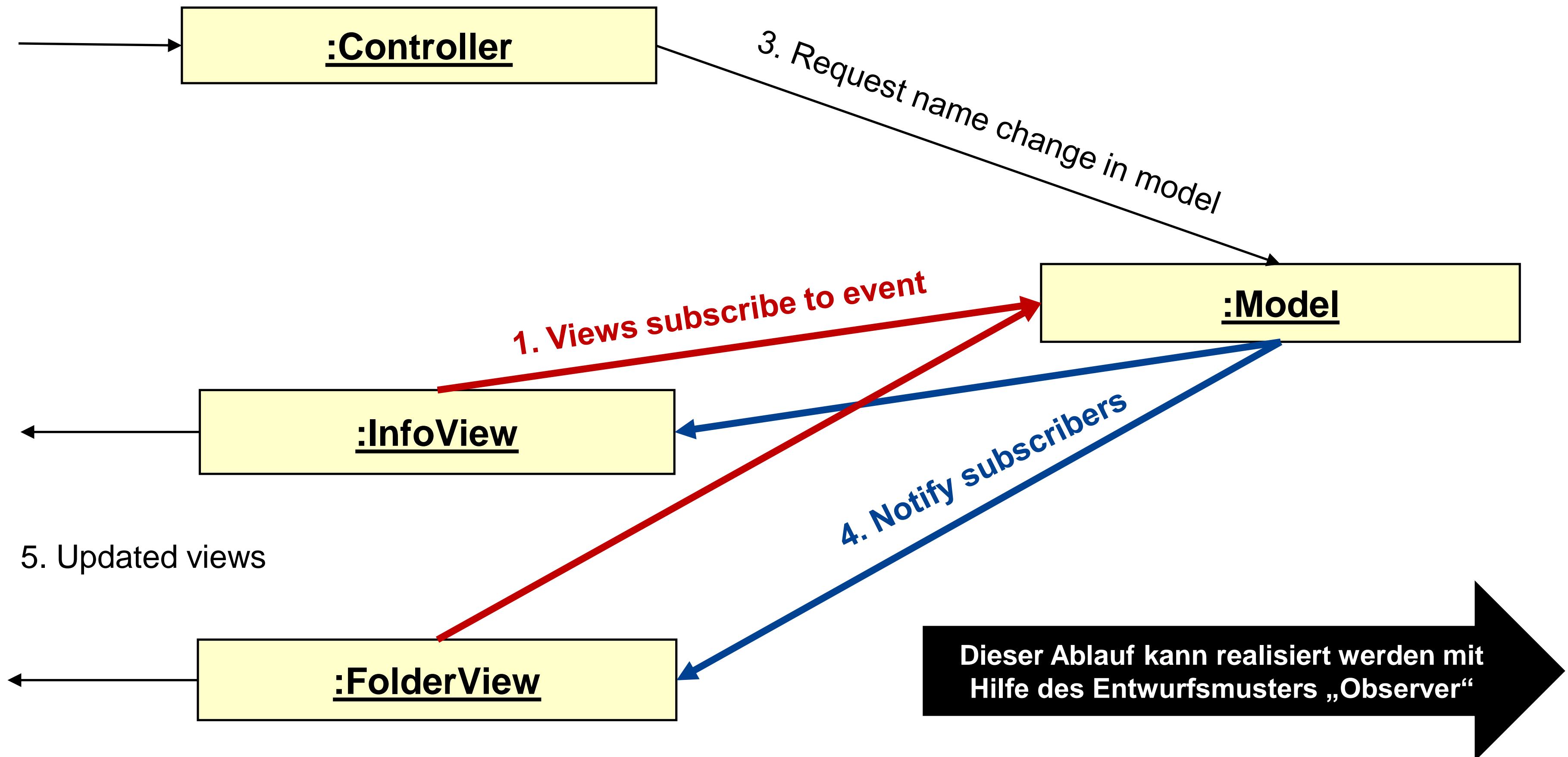
© 2000-2017 Dr. G. Kniesel

Beispiel einer auf der MVC Architektur basierenden Dateiverwaltung



universitätbonn

Abfolge von Events



Das Observer Pattern

Das Observer Pattern: Einführung

- Absicht
  - Stellt eine 1-zu-n Beziehung zwischen Objekten her
  - Wenn das eine Objekt seinen Zustand ändert, werden die davon abhängigen Objekte benachrichtigt und entsprechend aktualisiert

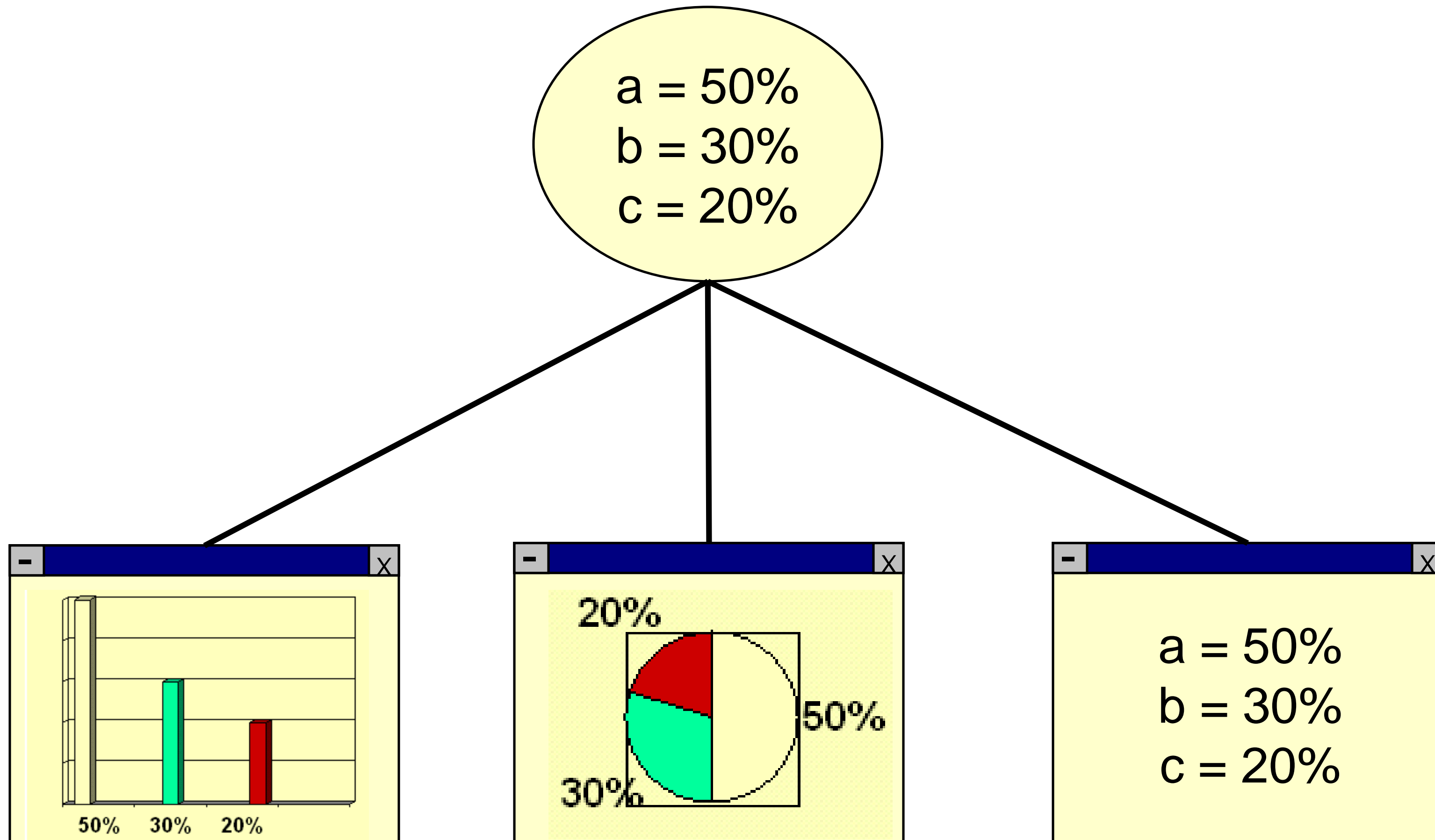
- Andere Namen
  - "Dependents", "Publish-Subscribe", "Listener"

- Motivation
  - Verschiedene Objekte sollen zueinander konsistent gehalten werden
  - Andererseits sollen sie dennoch nicht eng miteinander gekoppelt sein. (bessere Wiederverwendbarkeit)

  ➢ Diese Ziele stehen in einem gewissen Konflikt zueinander.
  Man spricht von *„conflicting forces“*, gegenläufig wirkenden Kräften.

Das Observer Pattern: Beispiel

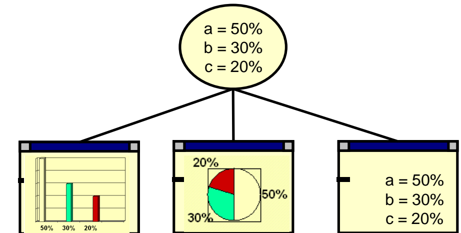
- Trennung von Daten und Darstellung
  - Wenn in einer Sicht Änderungen vorgenommen werden, werden alle anderen Sichten aktualisiert – Sichten sind aber unabhängig voneinander.



© 2000-2017 Dr. G. Kniesel

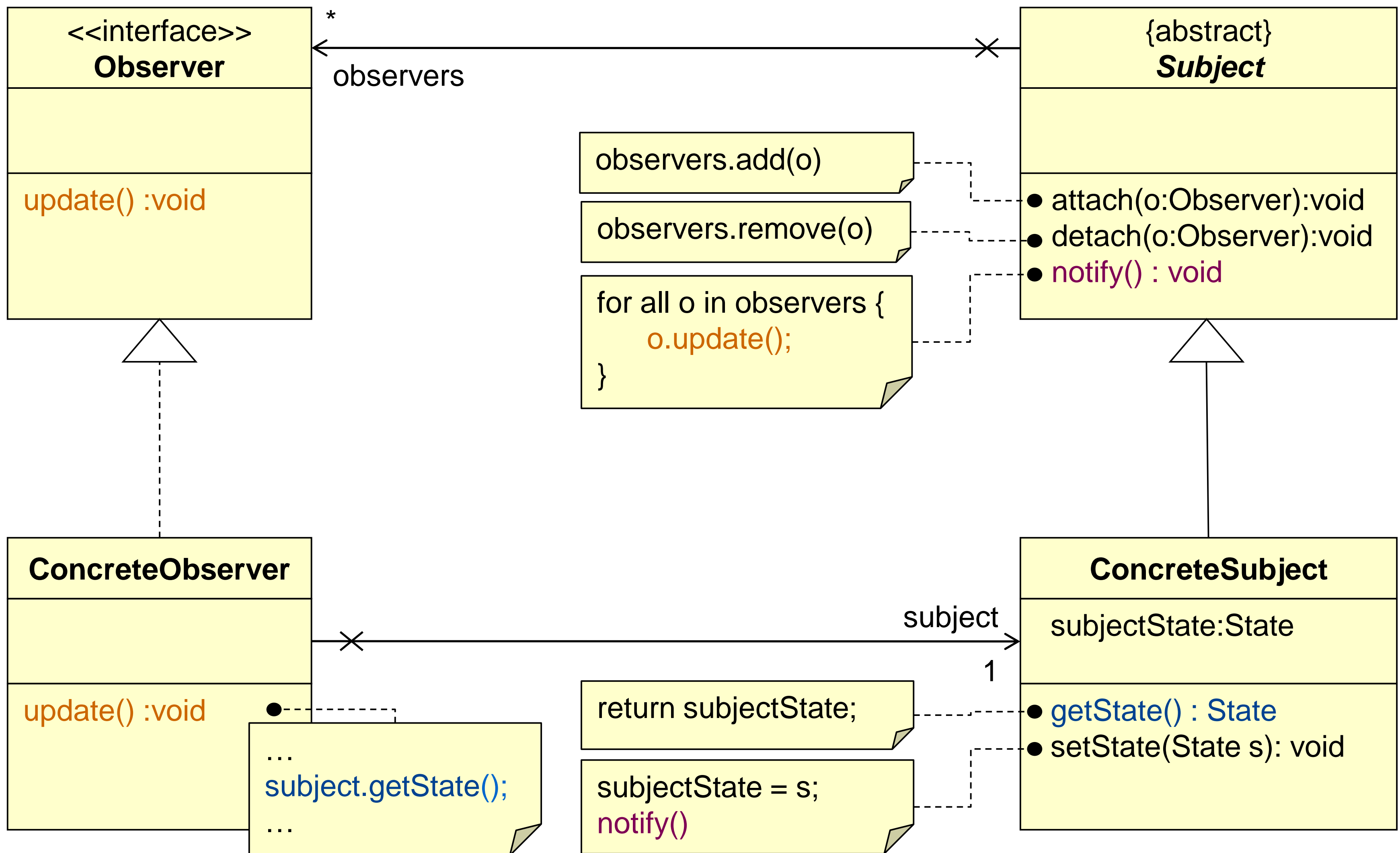
Das Observer Pattern: Anwendbarkeit

Das Pattern ist in folgenden Kontexten anwendbar:



- Abhängigkeiten
  - Ein Aspekt einer Abstraktion ist abhängig von einem anderen Aspekt.
  - Aufteilung dieser Aspekte in verschiedene Objekte erhöht Variationsmöglichkeit und Wiederverwendbarkeit.
- Folgeänderungen
  - Änderungen an einem Objekt erfordert Änderungen an anderen Objekten.
  - Es ist nicht bekannt, wie viele Objekte geändert werden müssen.
- Lose Kopplung
  - Objekte sollen andere Objekte benachrichtigen können, ohne Annahmen über die Beschaffenheit dieser Objekte machen zu müssen.

Das Observer Pattern: Struktur (N:1, Pull-Modell)



Rollenaufteilung / Verantwortlichkeiten (Pull Modell)

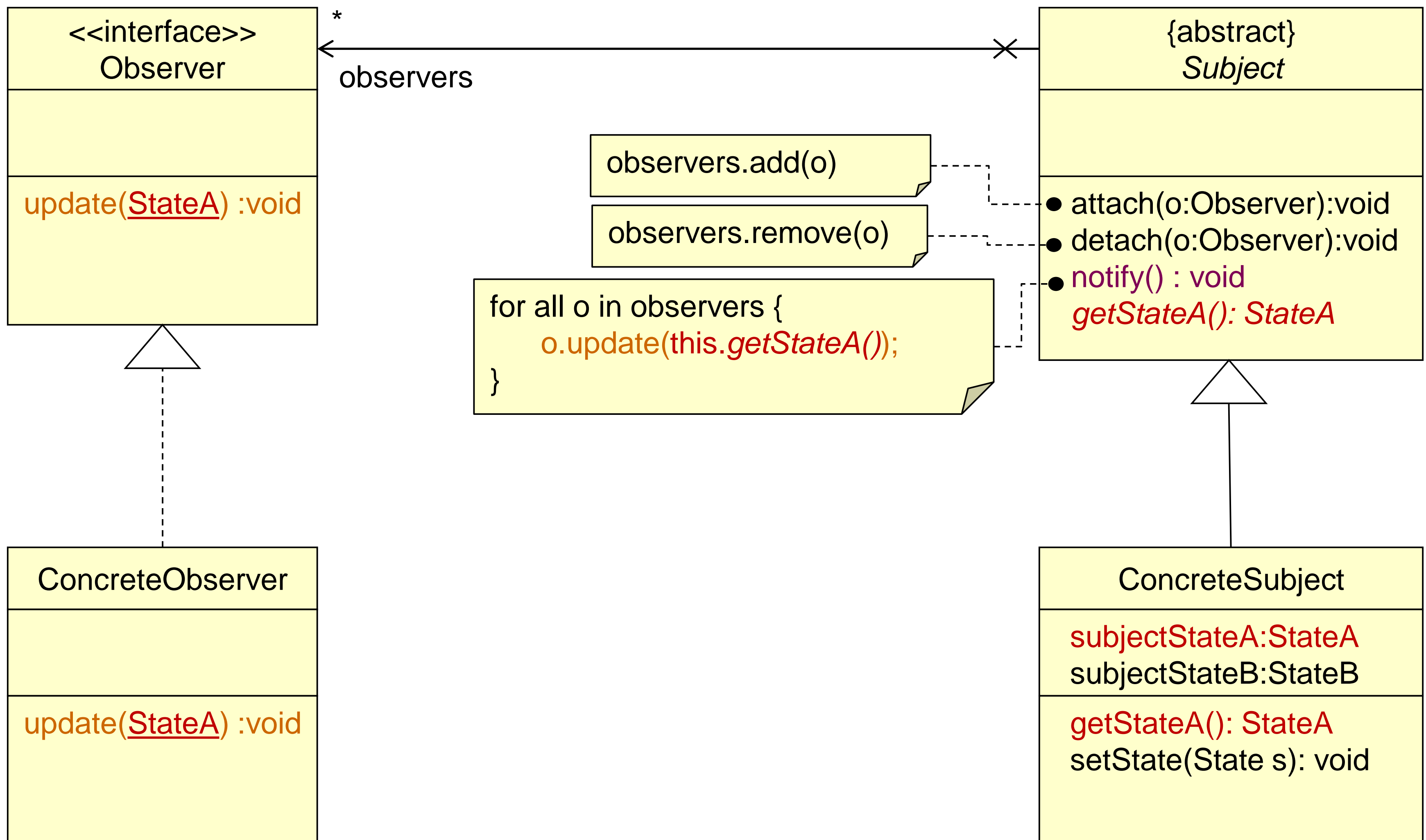
- Observer („Beobachter“) -- auch: Subscriber, Listener
  - update() -- auch: handleEvent
    - ⇨ Reaktion auf Zustandsänderung des Subjects
- Subject („Subjekt“) -- auch: Publisher
  - attach(Observer o) -- auch: register, addListener
    - ⇨ Observer registrieren
  - detach(Observer o) -- auch: unregister, removeListener
    - ⇨ registrierte Observer entfernen
  - notify()
    - ⇨ update-Methoden aller registrierten Observer aufrufen
  - setState(…)
    - ⇨ zustandsändernde Operation(en)
    - ⇨ für internen Gebrauch und beliebige Clients
  - getState()
    - ⇨ abfrage des aktuellen Zustands
    - ⇨ damit Observer feststellen können was sich wie geändert hat

Das Observer Patterns: Implementierung

Wie werden die Informationen über eine Änderung weitergegeben: „push“ versus „pull“

- Pull: Subjekt übergibt in „update()“ keinerlei Informationen, aber die Beobachter müssen sich die Informationen vom Subjekt holen
  - + Geringere Kopplung zwischen Subjekt und Beobachter.
  - – Berechnungen werden häufiger durchgeführt.
- Push: Subjekt übergibt in Parametern von „update()“ detaillierte Informationen über Änderungen.
  - + Rückaufrufe werden seltener durchgeführt.
  - – Beobachter sind weniger wiederverwendbar (Abhängig von den Parametertypen)
- Zwischenformen sind möglich

Das Observer Pattern: Struktur (N:1, Push-Modell)



Das Observer Pattern: Implementierung

- Vermeidung irrelevanter Notifikationen durch Differenzierung von Ereignissen
  - Bisher: Notifikation bei *jeder* Änderung
  - Alternative: Beobachter-Registrierung und Notifikation nur für spezielle Ereignisse
  - Realisierung: Differenzierung von attach(), detach(), update() und notify() in jeweils ereignisspezifische Varianten
  - Vorteile:
    - ⇨ Notifikation nur für relevante Ereignisse → höhere Effizienz
    - ⇨ Weniger „Rückfragen" pro Ereignis → höhere Effizienz
  - Nachteil: Mehr Programmieraufwand, wenn man sich für viele Ereignistypen interessiert
    - ⇨ Aber: Werkzeugunterstützung möglich

Das Observer Pattern: Implementierung

- Speicherung der Beziehung zwischen Subjekt und Beobachter
  - Instanzvariable im Subjekt oder ...
  - globale (Hash-)Tabelle
- Beobachtung mehrerer Subjekte
  - Beispiel Tabellenkalkulation: Jede Zelle muss evtl. viele Andere beobachten um sich bei deren Änderung neu zu berechnen.
  - Realisierung: Die „update()"-Methode muss einen Parameter haben, der das Subjekt angibt, das sich gerade geändert hat.
  - Querbezug zu Pull-Modell: Der Parameter kann für pull-Rückfragen an das Modell genutzt werden (keine feste Speicherung / Assoziation nötig)
  - Problem: Welchen Typ hat der Parameter?
    - ⇨ „Object": Zu unspezifisch, damit kann man nicht viel anfangen
    - ⇨ „Subject": Zu unspezifisch, damit kann man nicht viel anfangen
    - ⇨ „Concrete Subject": Zu spezifisch für Observer anderer Concrete Subjects

Das Observer Patterns: Implementierung

- ChangeManager
  - Verwaltet Beziehungen zwischen Subjekt und Beobachter. (Speicherung in Subjekt und Beobachter kann entfallen.)
  - Definiert die Aktualisierungsstrategie
  - Benachrichtigt alle Beobachter. Verzögerte Benachrichtigung möglich
    - ⇨ Insbesondere wenn mehrere Subjekte verändert werden müssen, bevor die Aktualisierungen der Beobachter Sinn macht

- Wer ruft notify() auf?
  - a) "setState()"-Methode des Subjekts:
    - ⇨ + Klienten können Aufruf von "notify()" nicht vergessen.
    - ⇨ – Aufeinanderfolgende Aufrufe von "setState()" führen zu evtl. überflüssigen Aktualisierungen.
  - b) Klienten:
    - ⇨ + Mehrere Änderungen können akkumuliert werden.
    - ⇨ – Klienten vergessen möglicherweise Aufruf von "notify()".

Das Observer Pattern: Implementierung

- Konsistenz-Problem
  - Zustand eines Subjekts muss vor Aufruf von "notify()" konsistent sein.
  - Vorsicht bei Vererbung bei Aufruf jeglicher geerbter Methoden die möglicherweise „notify()“ -aufrufen :

```
public class MySubject extends SubjectSuperclass {
    public void doSomething(State newState) {
      super.doSomething(newState);  // ruft "notify()" auf
      this.modifyMyState(newState); // zu spät!
    }
}
```

- Lösung
  - Dokumentation von „notify()“-Aufrufen erforderlich (Schnittstelle!)
  - Besser: In Oberklasse „Template-Method Pattern“ anwenden um sicherzustellen, dass „notify()“-Aufrufe immer am Schluss einer Methode stattfinden → s. nächste Folie.

Das Observer Pattern: Implementierung

Verwendung des „Template Method Pattern“

```
public class SubjectSuperclass {
    ...
    final public void doSomething(State newState) {
      this.doItReally(newState);
      this.notify();  // notify immer am Schluß
    }
    public void doItReally(State newState) {
    }
}
```

Template Method → `doSomething`

Hook Method → `doItReally`

```
public class MySubject extends SubjectSuperclass {
    public void doItReally(State newState) {
        this.modifyMyState(newState);
    }
}
```

Template Method Pattern

- Anwendbarkeit
  - Feste Reihenfolge von Aktionen muss garantiert werden
  - … aber genaue Ausgestaltung der Aktionen soll anpassbar sein

- Beispiel
  - s. vorherige Folie

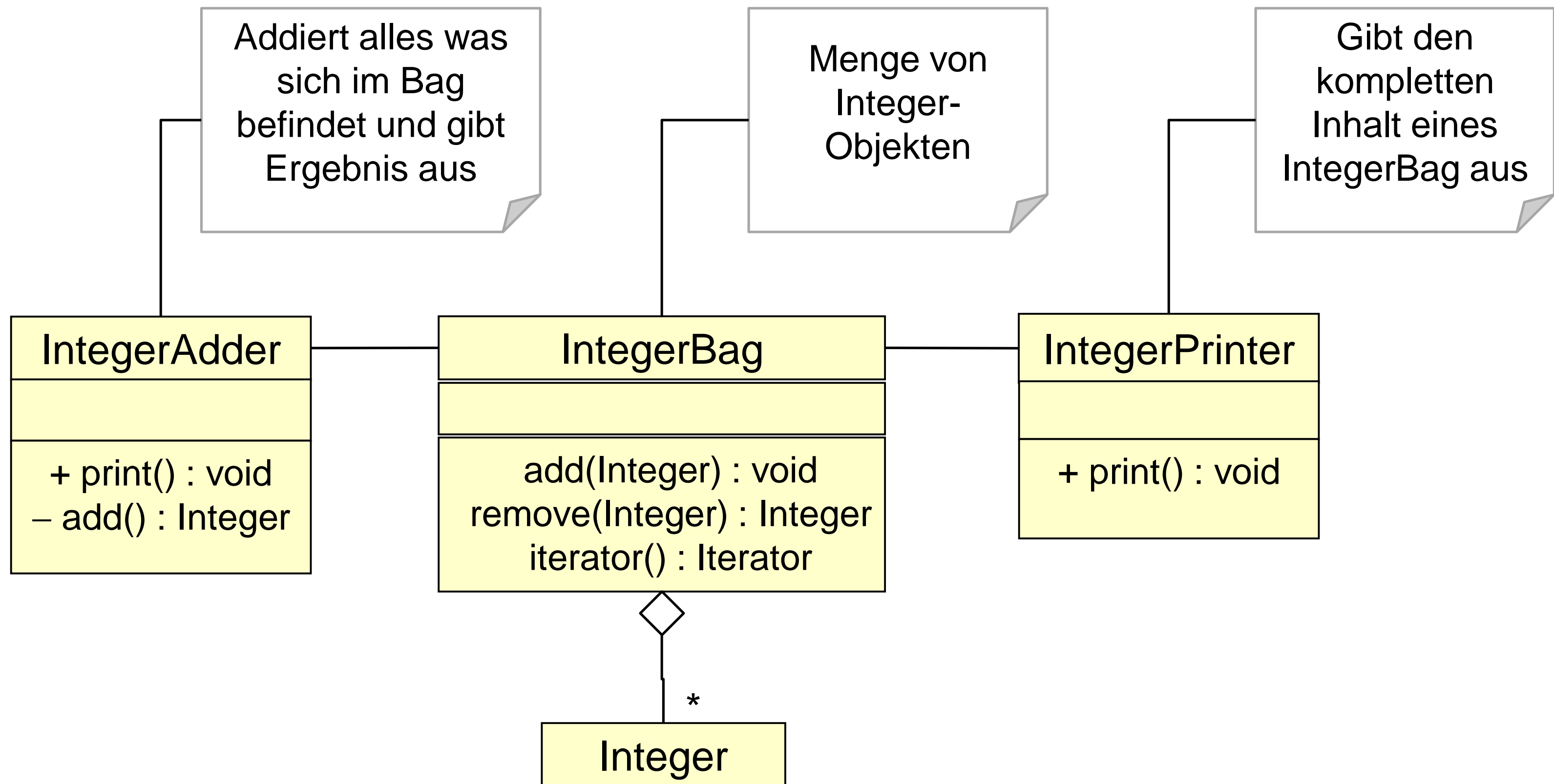
- Schema
  - Feste Reihenfolge □ festgelegt durch nicht überschreibbare Methode („final“ Schlüsselwort in Java)
    - ⇨ Gelb unterlegte Methode auf vorheriger Seite
  - Anpassbare Aktionen □ überschreibbare Methode die in der Template Method aufgerufen wird
    - ⇨ Blau unterlegte Methoden auf vorheriger Seite

Das Observer Pattern: Konsequenzen

- Unabhängigkeit
  - Konkrete Beobachter können hinzugefügt werden, ohne konkrete Subjekte oder andere konkrete Beobachter zu ändern.
  - Konkrete Subjekte können unabhängig voneinander und von konkreten Beobachtern variiert werden.
  - Konkrete Subjekte können unabhängig voneinander und von konkreten Beobachtern wiederverwendet werden.
- „Broadcast"-Nachrichten
  - Subjekt benachrichtigt alle angemeldeten Beobachter
  - Beobachter entscheiden, ob sie Nachrichten behandeln oder ignorieren
- Unerwartete Aktualisierungen
  - Kleine Zustandsänderungen des Subjekts können komplexe Folgen haben.
  - Auch uninteressante Zwischenzustände können unnötige Aktualisierungen auslösen.

Let's play patterns!

Aufgabe A



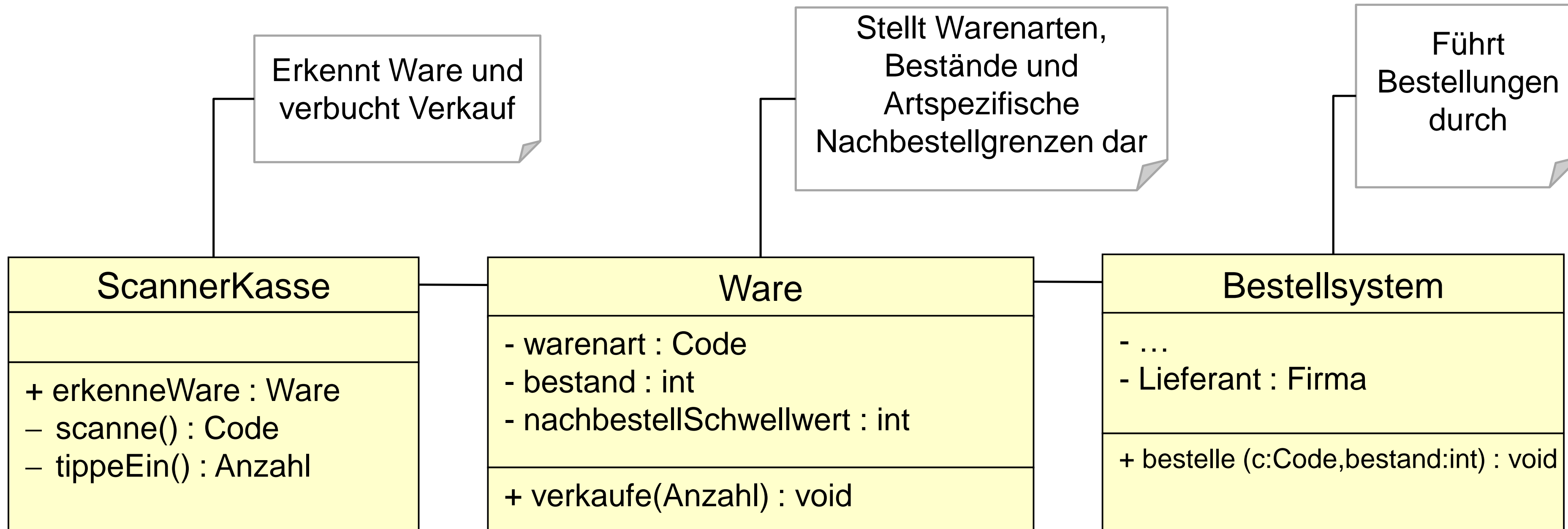
- Jedes mal wenn sich der Inhalt des IntegerBag ändert
  - aktualisieren IntegerAdder die zugehörige Summe
  - gibt IntegerPrinter den Gesamtinhalt erneut aus

Mögliche Lösung

- Siehe http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html

universitätbonn

Aufgabe B



- Jedes mal wenn der Bestand einer Warenart unter den jeweiligen Nachbestellschwellwert sinkt, wird das Bestellsystem informiert mit Angabe der Warenart und des Restbestands.

Konsequenzen von Observer für Abhängigkeiten

Für Abhängigkeitsreduzierung allgemein

Für Presentation-Application-Data (PAD)

Für Boundary-Controller-Entity (BCE)

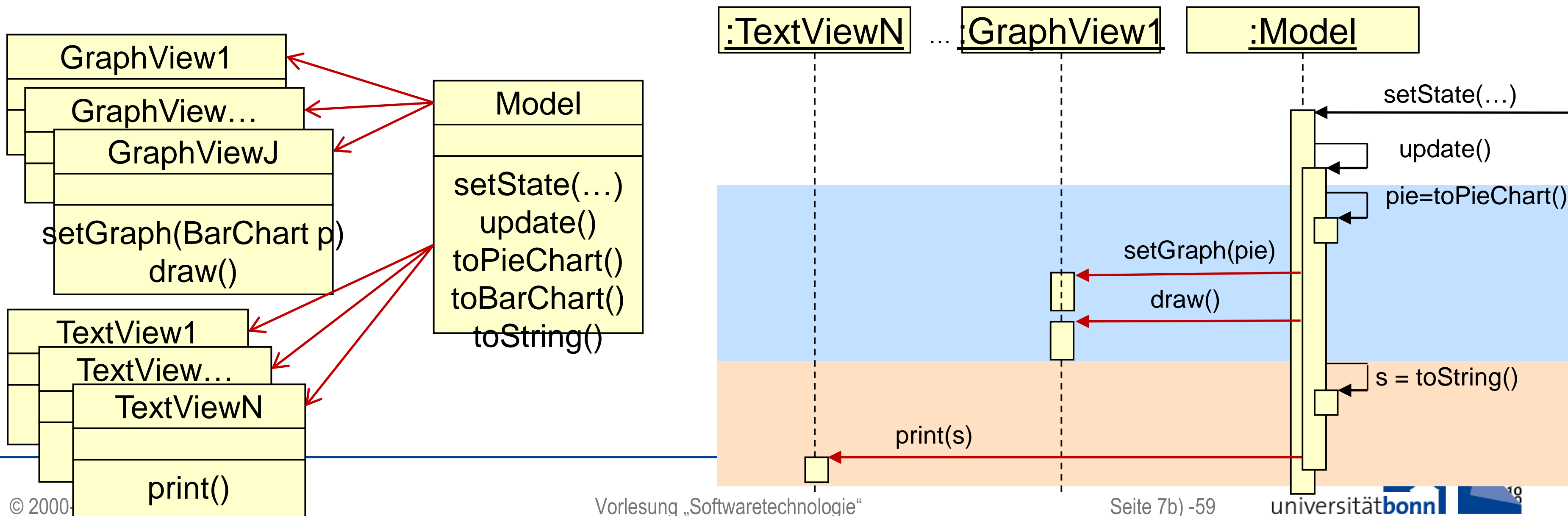
Für Model-View-Kontroller (MVC)

PAD versus BCE versus MVC

Abhängigkeiten mit und ohne Observer

Ohne Observer: Abhängigkeit View ← Model

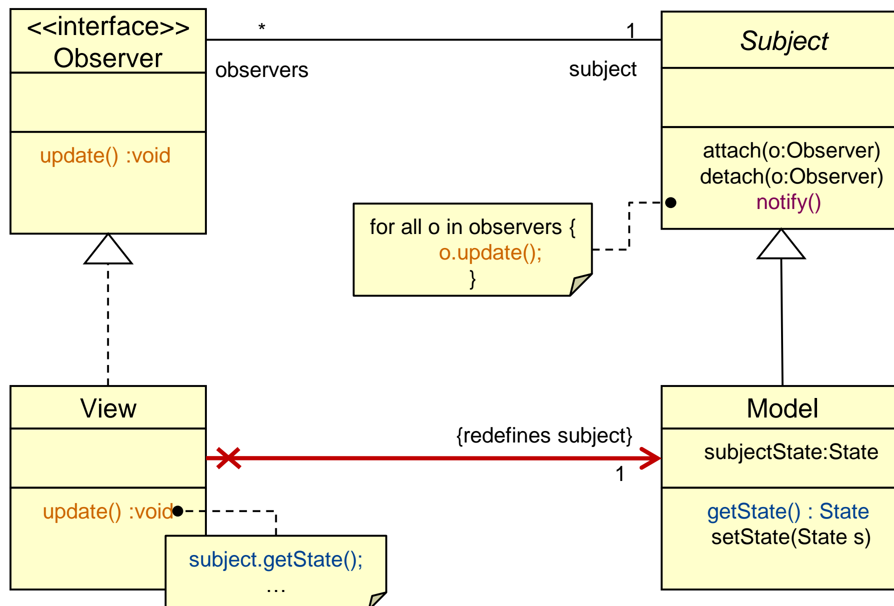
- Ein konkretes Model kennt das Interface eines jeden konkreten View und steuert was genau jeder der Views nach einem update tun soll:
  - myGraphView1.setGraph(this.toPieChart()); myGraphView1.draw();
  - myGraphView2.setGraph(this.toBarChart()); myGraphView2.draw();
  - …
  - myTextViewN.print(myState.toString());

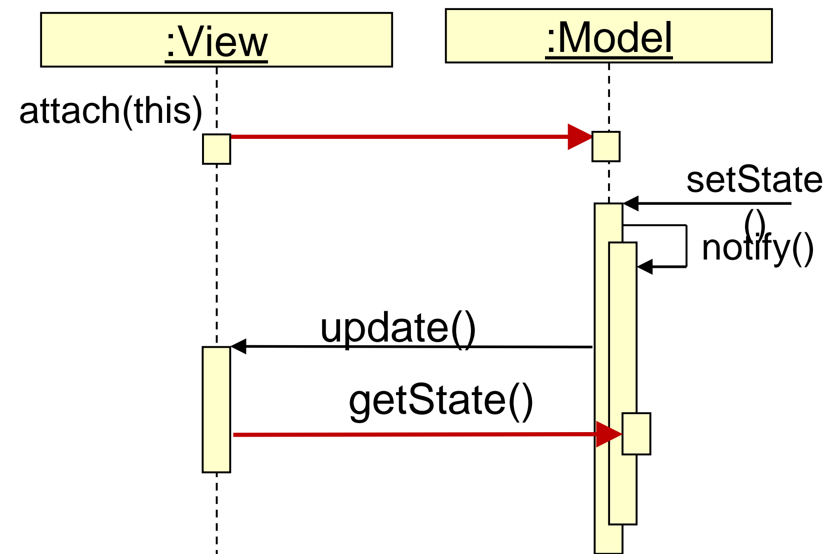


Abhängigkeiten mit und ohne Observer

Mit Observer: Abhängigkeit View → Model

- Ein konkretes Model kennt nur das abstrakte Observer-Interface
- Ein konkreter View kennt die zustandsabfragenden Methoden des konkreten Models
  - model.getState1();
  - model.getState2();





universitätbonn

Nettoeffekt: Abhängigkeits- und Kontrollumkehrung

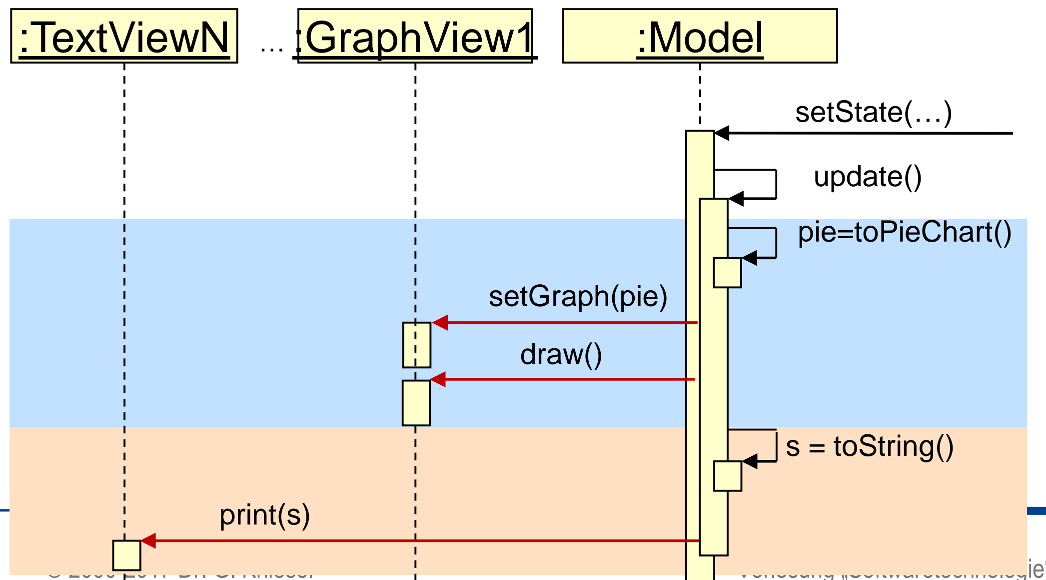
Abhängigkeitsumkehrung („Dependency Inversion“)

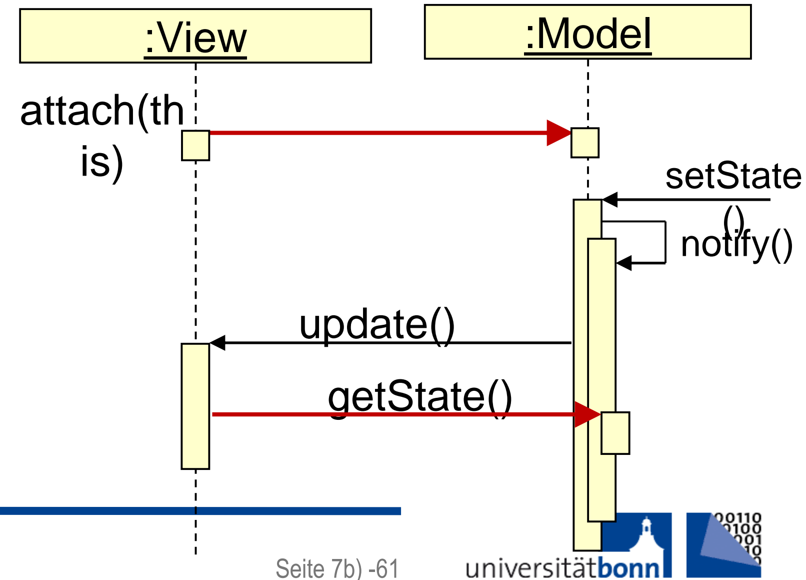
- Ohne Observer
  - Abhängigkeit Model → Views
- Mit Observer
  - Abhängigkeit Model ← Views

Kontrollumkehrung („Inversion of Control“)

- Ohne Observer
  - Model steuert alle updates
- Mit Observer
  - Jeder View steuert sein update

Vergleich Ablauf ohne / mit Observer





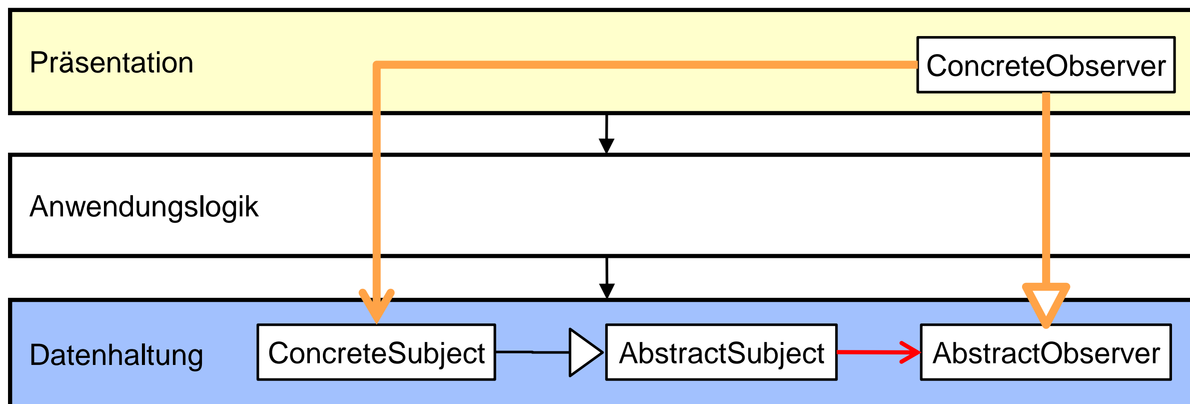
universitätbonn

Software-Architekturen und das Observer-Pattern

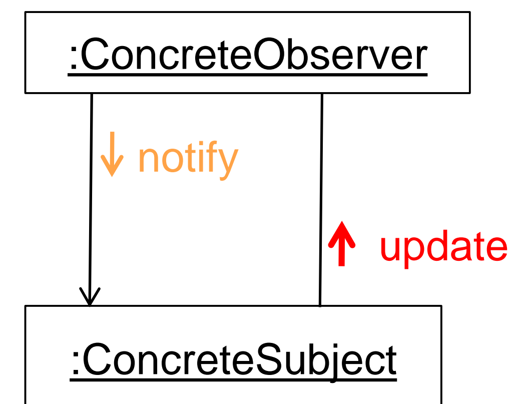
Das Observer Pattern: Konsequenzen

- Abstrakte Kopplung
  - Subjekte aus tieferen Schichten eines Systems können mit Beobachtern aus höheren Schichten zur Laufzeit kommunizieren (update()-Aufruf), …
  - … ohne dass die statischen Abhängigkeiten zwischen den Klassen die Ebenenarchitektur verletzen.

Statische Abhängigkeiten: Hierarchisch

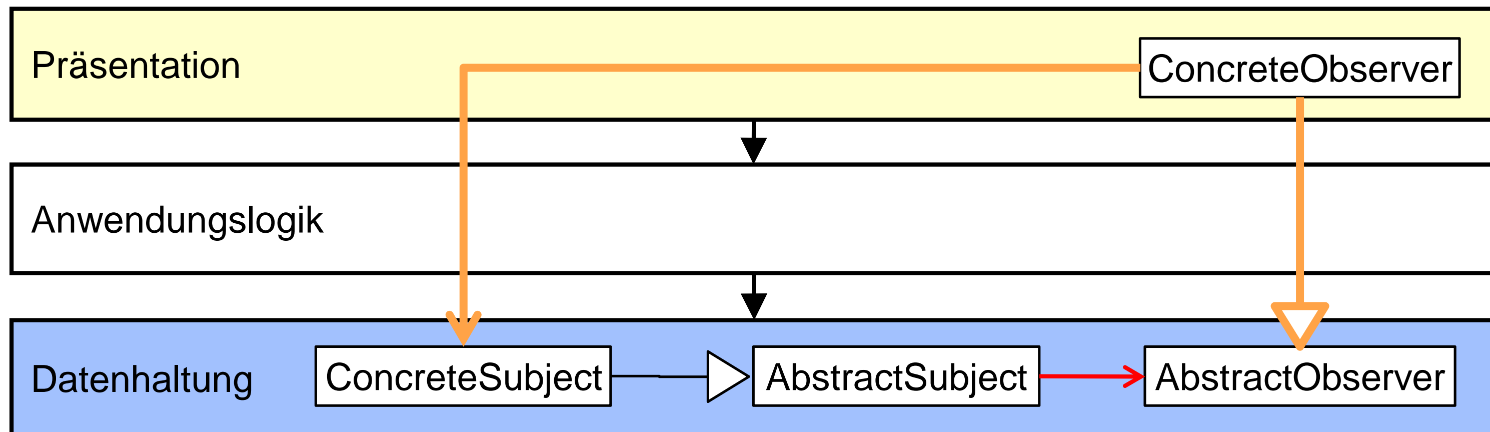


Dynamische Interaktionen: Bidirektional



Auswirkungen von Observer auf Ebenen: „Presentation-Application-Data“ (1)

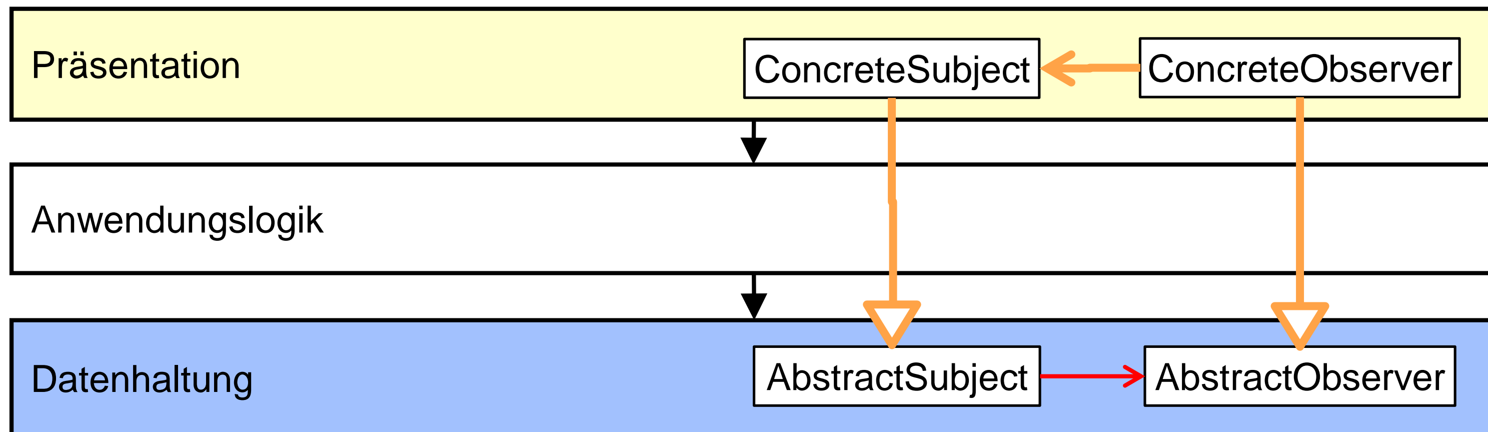
- N-tier-Architekturen basieren auf der rein hierarchische Anordnung von Presentation, Anwendungslogik und Datenhaltung



- 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 als AbstractSubjects nur von dem AbstractObserver abhängig.
  - Da dessen Definition auch in der Datenschicht angesiedelt ist, wird die Ebenenanordnung nicht verletzt

Auswirkungen von Observer auf Ebenen: „Presentation-Application-Data“ (2)

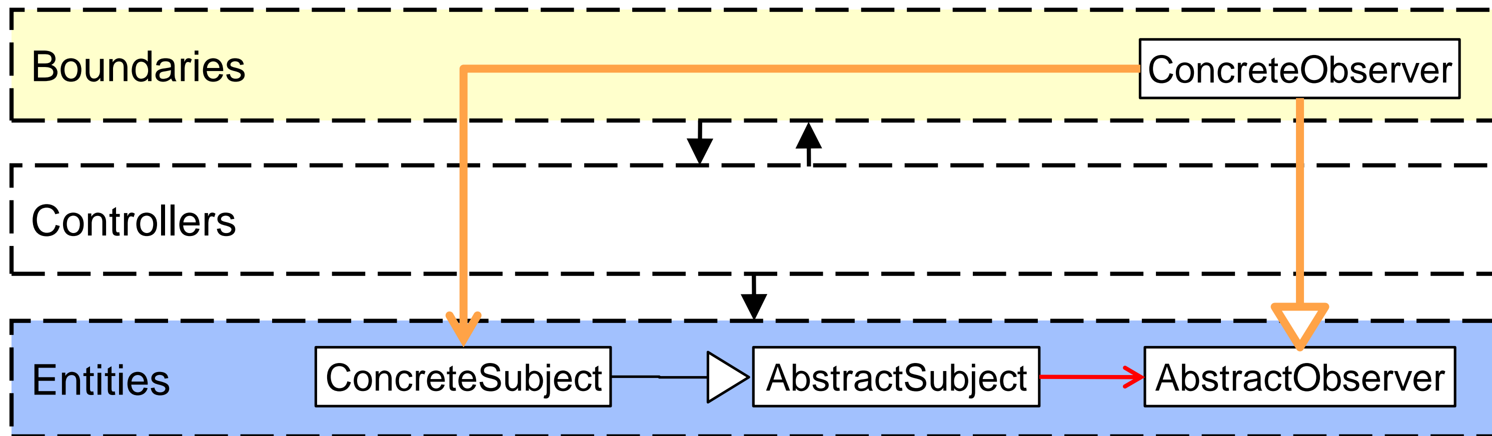
- ConcreteSubjects können in der Präsentations-Ebene angesiedelt sein, ohne die Ebenenarchitektur zu verletzen:



- So können manche Präsentationsobjekte andere Präsentationsobjekte beobachten und sich bei deren Änderung automatisch anpassen
- Der ConcreteSubject und/oder ConcreteObserver kann auch in der Anwendungslogik liegen
  - Nur der Fall, dass der ConcreteObserver in einer Ebene unter dem ConcreteSubject liegt darf nicht auftreten

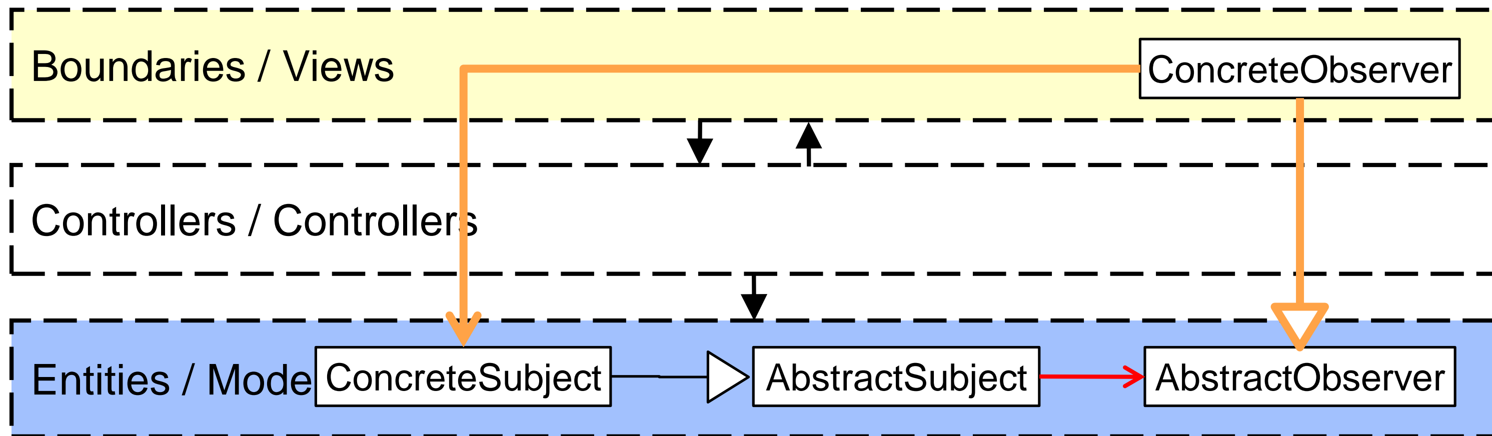
Auswirkungen von Observer für „Boundary-Controller-Entity“ Stereotypen

- Die Abhängigkeitsreduzierung ist die Gleiche wie bei 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
    - ⇨ Controller primär in der Anwendungslogik-Schicht
    - ⇨ Alle Entities in der Datenhaltungs-Schicht

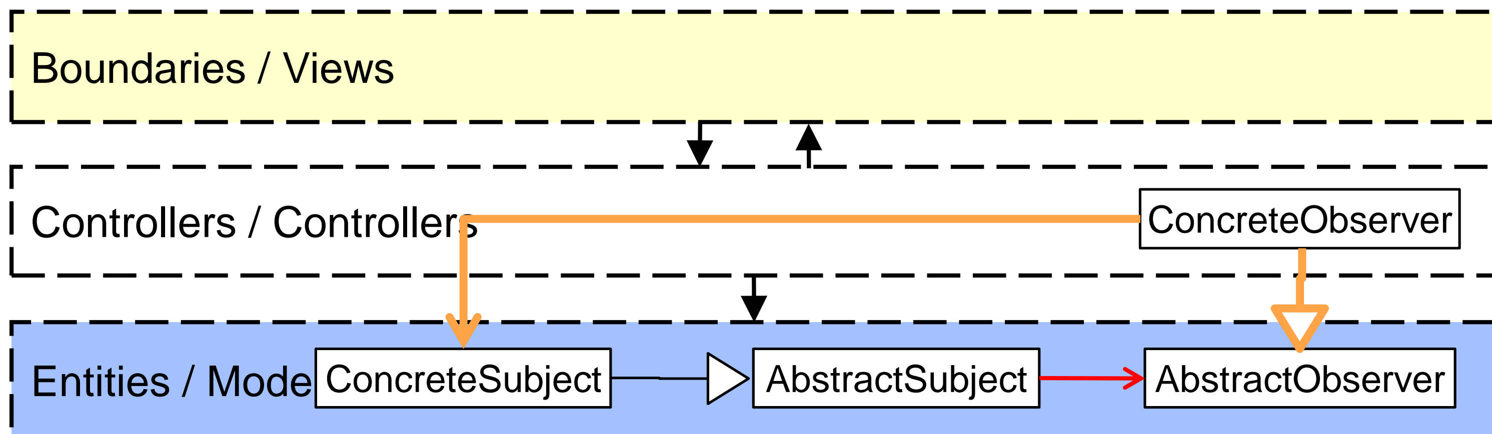
Auswirkungen von Observer für Model-View-Controller



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

Auswirkungen von Observer für Model-View-Controller

- Observer sind nicht immer Views! Sie können auch Controller sein!



- Sie können sich bei Modellelementen als Observer registrieren, deren Veränderungen sammeln und gefiltert oder kummuliert weitergeben.
  - Aktive Weitergabe: Aufruf / Steuerung von Aktionen anderer Objekte („Controller"-Rolle)
  - Passive Weitergabe: Benachrichtigung von eigenen Observern („Modell"-Rolle)
  - Siehe auch „Mediator-Pattern" („Vermittler")