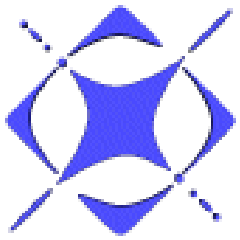


Architektur Muster

Strukturierungsprinzipien zur Organisation von Software-Systemen

Prof. Dr. Nikolaus Wulff

Architekturmuster

- Architekturmuster beschreiben die grundsätzliche Struktur und Organisation einer Anwendung und liegen auf der höchsten Abstraktionsebene.
- Ein Architekturmuster beschreibt die Menge vordefinierter Subsysteme, spezifiziert deren Zuständigkeit und enthält Regeln zu Organisation der Beziehungen zwischen Ihnen.
- Architekturmuster sind prinzipiell sprach neutral und Plattform unabhängig.
  - Konkrete Ausprägungen sind an ein bestimmtes OS oder eine Sprache gebunden, z. B. EJB- oder Dot.Net.

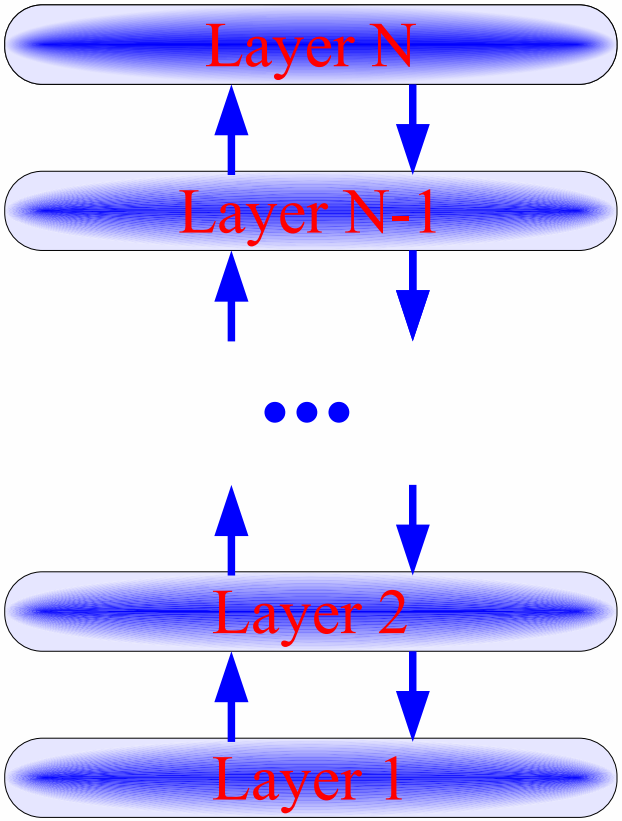
Architekturmuster Kategorien

- Vom Chaos zur Struktur
  - Layers
  - Pipes-and-Filter
  - Blackboard-Muster
- Verteilte Systeme
  - Broker-Muster
- Interaktive Systeme
  - Model-View-Controller
  - Presentation-Abstraction-Control
- Adaptierbare Systeme
  - Reflection-Muster
  - Microkernel

Layers

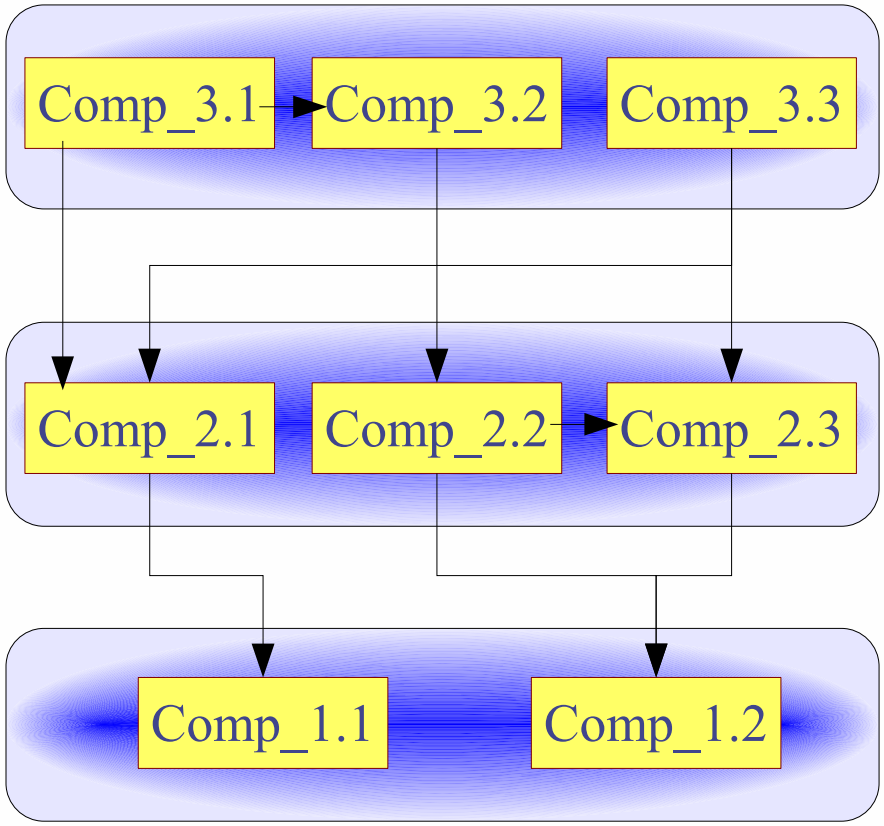
- Größere Softwaresysteme werden in Schichten (engl. Layers) organisiert.
- Jede Schicht stellt eine höhere Abstraktionsebene als die darunterliegende Schicht da.
- Dies führt zu einer ***vertikalen*** Struktur.
- Innerhalb einer Schicht gibt es viele verschiedene Komponenten, die ihrerseits nach bestimmten Aufgaben und Diensten organisiert werden, dies führt zu einer ***horizontalen*** Strukturierung.

Layers Struktur



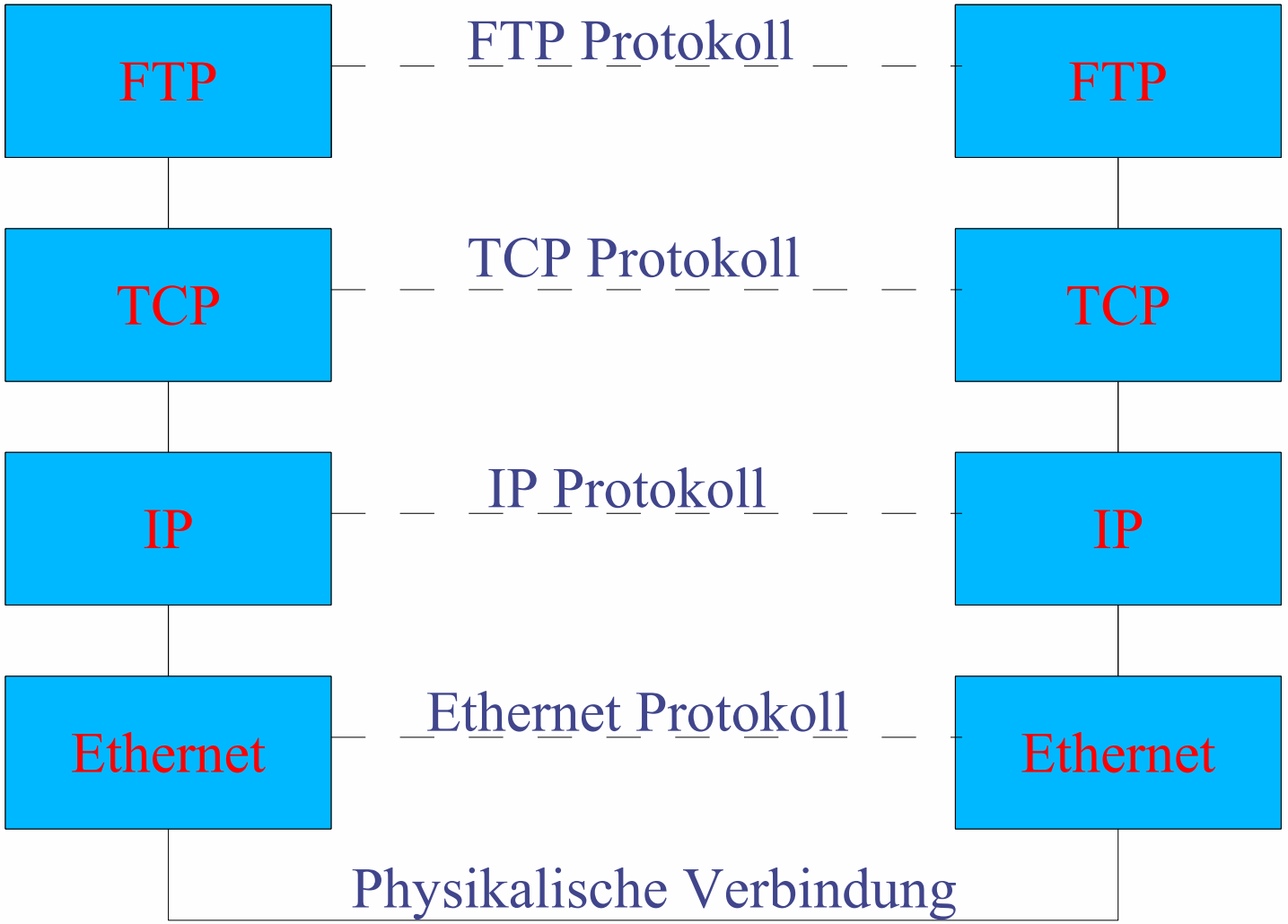
- Schicht J verwendet Dienste der Schicht J-1.
- Schicht J kommuniziert nur mit Schicht J-1.
- I.A. kennt die Schicht J die Schicht J+1 nicht.
- Abhängigkeiten zwischen den Schichten werden minimiert.

Layers und Partitionierung



- Komponenten einer Schicht kommunizieren untereinander und mit den Komponenten benachbarter Schichten.
- Kommunikation über mehrere Schichtgrenzen hinweg findet i.A. nicht statt.

TCP/IP Schichtenmodel

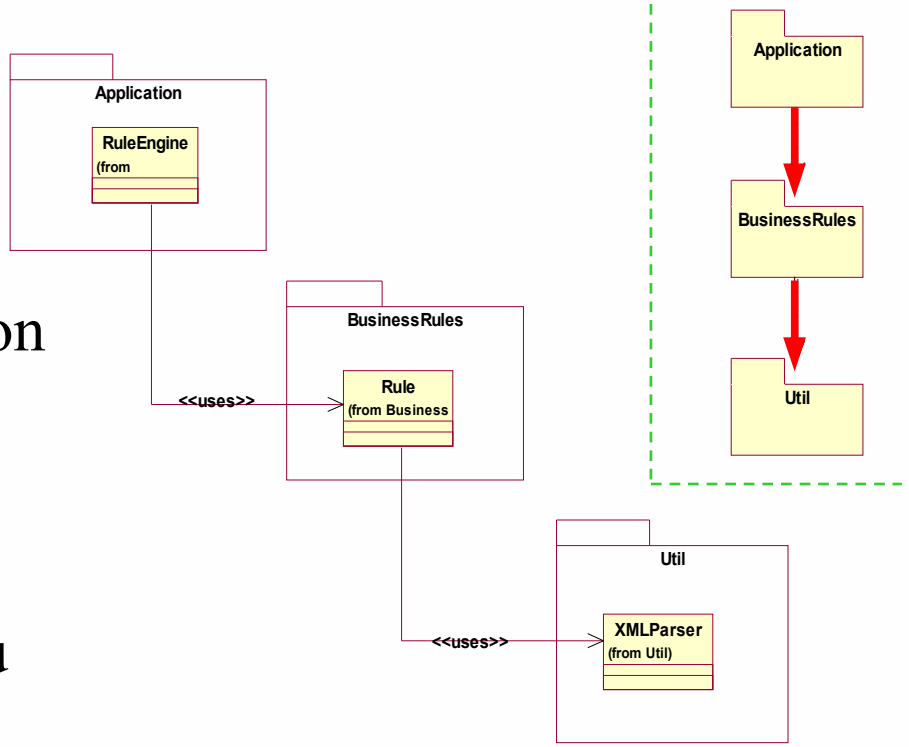


Layers cont.

- So wie bis lang beschrieben bauen die oberen Schichten auf den unteren auf. Die kann jedoch zu unerwünschten Abhängigkeiten führen.
  - Die abstrakten/höherwertigen oberen Schichten sind abhängig von den elementaren unteren Schichten.
  - Das **Dependency Inversion Principle** (DIP) stellt diese Abhängigkeitskette auf den Kopf.
  - Häufig wird dies von entsprechenden Frameworks nach dem **Inversion of Control** Muster (IoC) realisiert.
- Schichten kennen sich dann gar nicht mehr oder allenfalls von unten nach oben...

Top-Down Abhängigkeiten

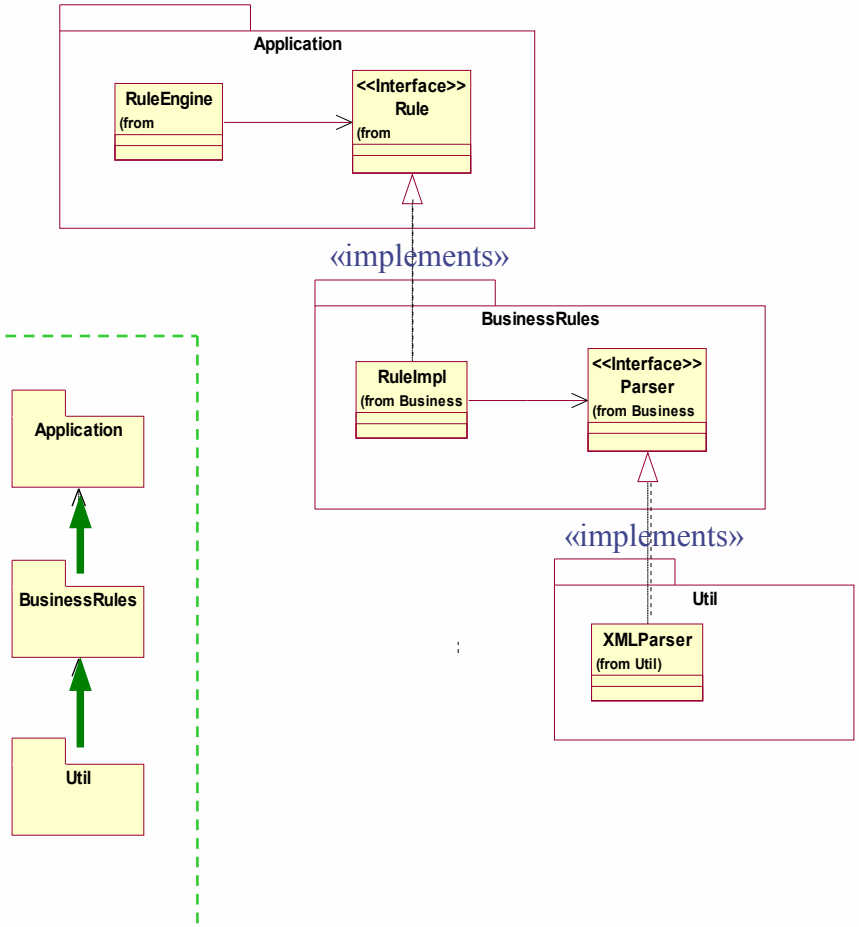
- Naiv modelliert verlaufen die Abhängigkeiten zwischen den Schichten Top-Down.
- Die High-Level Pakete sind von den Low-Level Paketen abhängig.
- Das Dependency Inversion Principle (DIP) verlangt genau das Gegenteil:



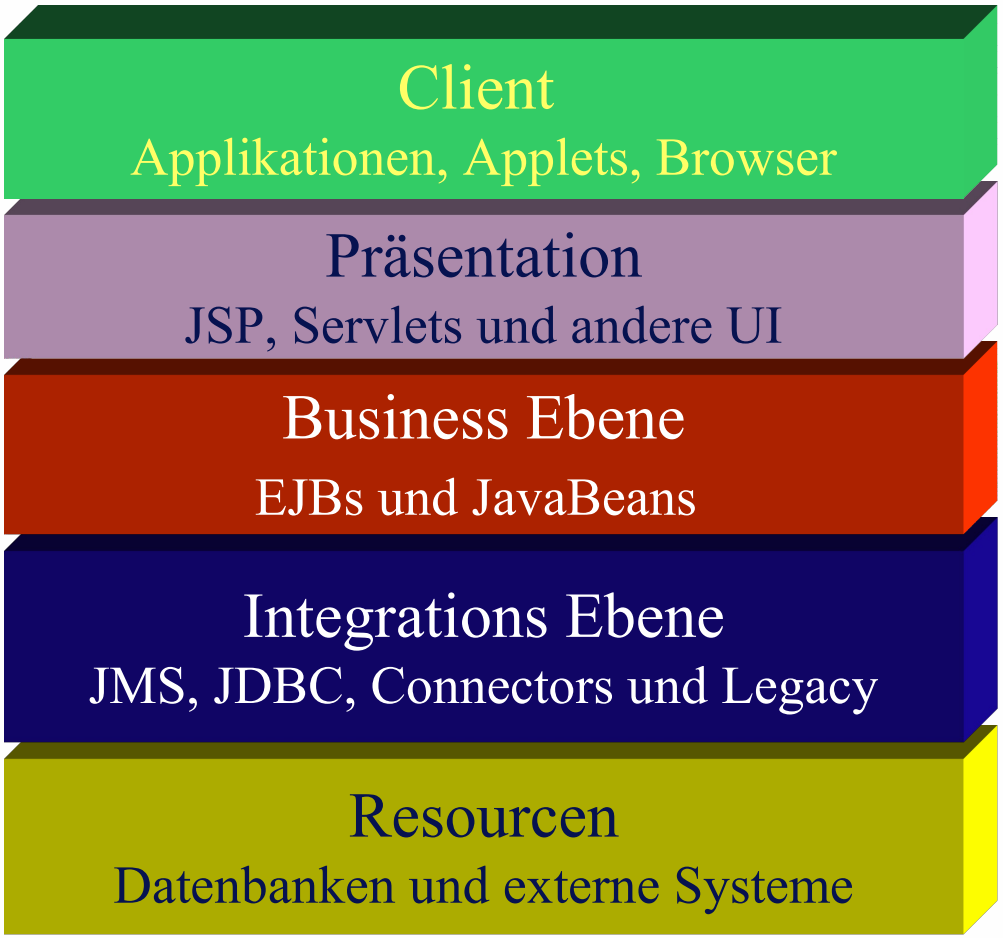
• High Level Module dürfen nie von Low Level Modulen abhängen.
• Low Level Module können von High Level Modulen abhängen.
• High Level Module müssen stabil sein.

Umkehren der Abhängigkeiten

- Nach dem Anwenden des DIP sind die High Level Module nicht mehr von den Low Level Modulen abhängig.
- Obwohl die Anwendung immer noch Top-Down designed ist verlaufen die Abhängigkeiten Bottum-Up.
- Anstatt ***uses*** wird eine ***implements*** Relation verwendet.
- Der Abhängigkeitsgraph ist auf den Kopf gestellt.
- => Dependency Inversion



J2EE Schichtenmodel



Benutzer Interaktion

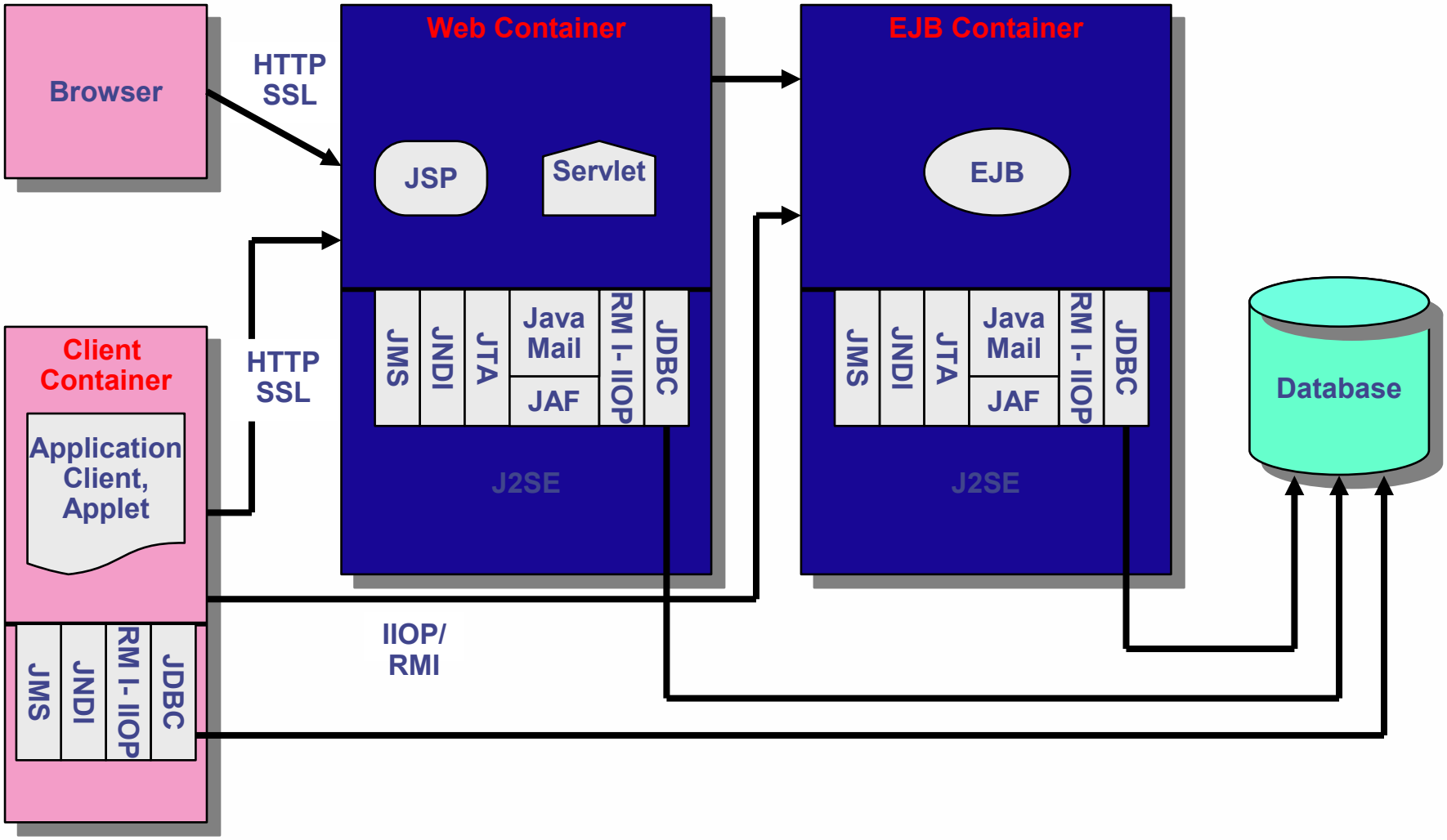
Single sign-on, Sessionverwaltung, Formatierung und Zustellung

Geschäftslogik, Transaktionen, Daten und Services

Resource adapters, rule engine und workflow, Altanwendungen

Daten und externe Dienste

J2EE Architektur



Layers: Vorteile

- Wiederverwendung von Schichten
  - Die API einer Schicht kann in verschiedenen Kontexten verwendet werden.
- Unterstützung von Standardisierung
  - Klar definierte Abstraktionen erleichtern die Entwicklung standardisierter APIs.
- Abhängigkeiten bleiben lokal
  - Änderungen z.B. eines Gerätetreibers betreffen nur eine Schicht, ohne die anderen zu tangieren.
- Austauschbarkeit
  - Eine Implementierung kann einfach ausgetauscht werden.

Layers: Nachteile

- Kaskaden veränderten Verhaltens
  - Ändert sich das Verhalten einer Schicht, so propagiert dies durch alle Schichten. (API nicht kompatibel)
- Geringe Effizienz
  - Zwischen den Schichten finden häufig Datentransformationen statt.
- Unnötige Arbeit
  - Niedrige Dienste verrichten oft redundante oder überflüssige Arbeit. Dies verringert die Performance
- Schwierigkeit die richtige Granularität zu finden

Pipes-and-Filters

- Eine Pipes-and-Filter Architektur eignet sich für Systeme, die Datenströme verarbeiten.
- Jeder Verarbeitungsschritt ist in einem Filter gekapselt.
- Daten werden durch Kanäle (engl. Pipes) weiter geleitet.
- Die Filter lassen sich beliebig neu anordnen, hintereinander schalten und austauschen.
- Dies ermöglicht es Familien von verwandten Systemen zu erzeugen.

Pipes-and-Filter Beispiele

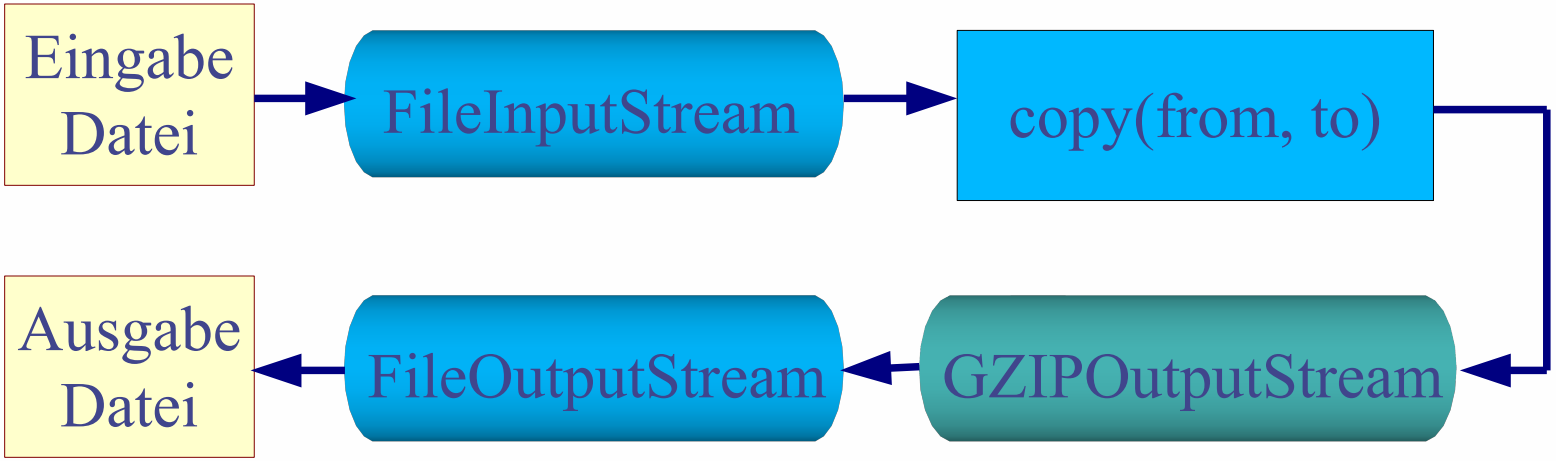
- Der Kommandointerpreter sowie viele Werkzeuge des Unix Betriebssystems sind nach dem Pipes-and-Filter Muster gebaut. Die Ausgabe des einen dient als Eingabe für das nächste Werkzeug:

```
cat file | sort -u | grep name
```

- Die Klassen des `java.io`-Pakets sind nach dem P&F Muster designed.
- Die `java.awt.image`- ImageProducer, ImageSource und ImageFilter Abstraktionen bilden eine P&F Architektur, die sich beliebig erweitern lässt, durch hinzufügen neuer Quellen und Senken.

Ströme verketten

- Die Java IO-Ströme lassen sich wie bei der Unix Pipe Architektur verketten.
- Dadurch ist es möglich Filter dazwischen zu schalten oder Ströme mit neuen Eigenschaften zu dekorieren.



Ein ZIP Strom

- Die Verkettung von Strömen ist recht einfach:

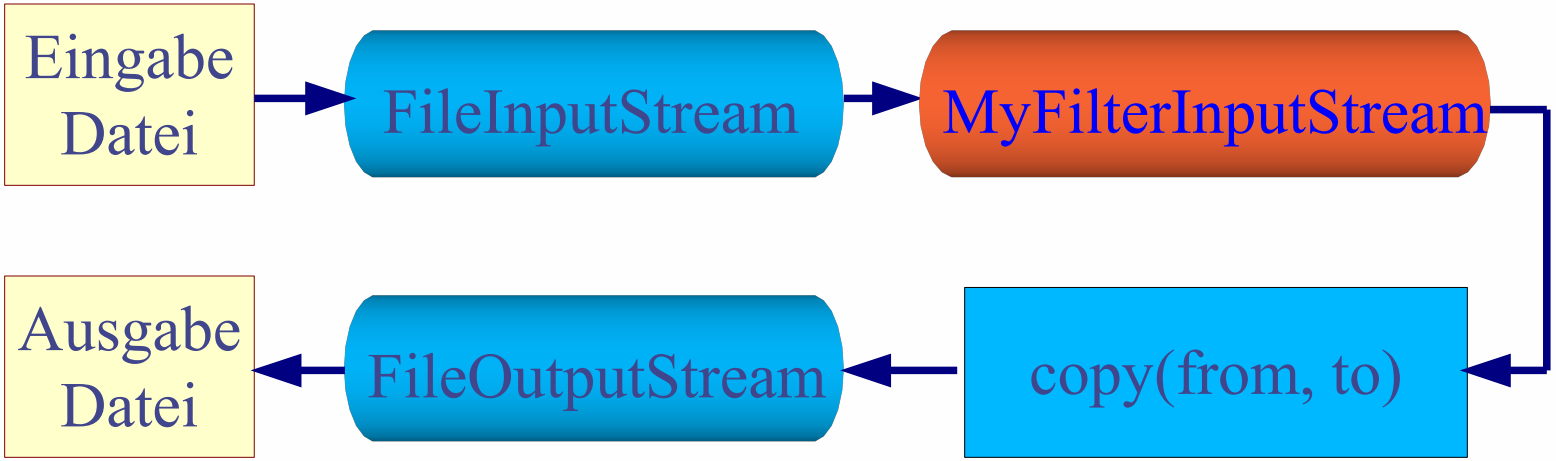
```
OutputStream out;
out = new FileOutputStream(destination);
out = new BufferedOutputStream(out);
out = new GZIPOutputStream(out);
```

- und hier die generische Kopiermethode:

```
protected void copy(InputStream in,
        OutputStream out) throws IOException
{
    byte[] buf = new byte[4096];
    int read, EOF = -1;
    while( (read = in.read(buf)) != EOF) {
        out.write(buf, 0, read);
    }
}
```

Filter als Ströme

- Durch die Verkettung von Eingabe- oder Ausgabeströmen lassen sich sehr einfach und modular Filter entwickeln.
- Es muss lediglich ein zusätzlicher Filter in die Ein- oder Ausgabe geschaltet werden:



Pipes-and-Filter: Vorteile

- Flexibilität durch Austausch und Hinzufügen neuen von Filtern.
- Flexibilität durch Neuanordnung.
- Wiederverwendung einzelner Filter.
- Rapid Prototyping von Pipeline Prototypen.
- Zwischendateien sind nicht notwendig aber so gewünscht möglich.

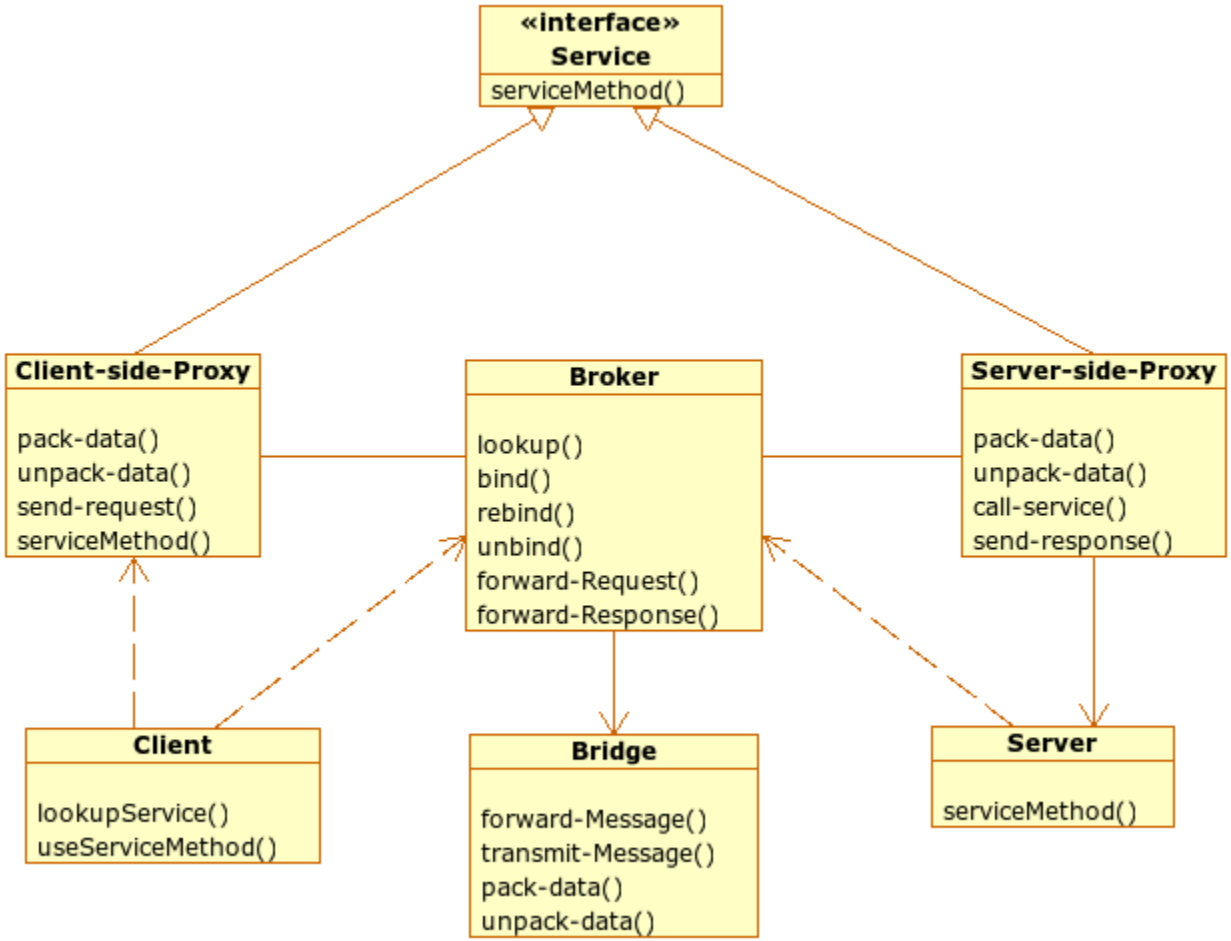
Pipes-and-Filter: Nachteile

- Die Kosten der Datenübertragung zwischen den Filtern können je nach Pipe sehr hoch sein.
- Häufig überflüssige Datentransformationen zwischen den einzelnen Filterstufen.
- Fehlerbehandlung über Filterstufen hinweg ist teilweise schwierig.

Broker

- Das Broker Muster dient zur Strukturierung verteilter Software-Systeme mit entkoppelten, entfernten Komponenten.
- Ein Vermittler (engl. Broker) dient zur Koordination der Kommunikation.
- Eine Broker Architektur bietet Dienste für das Hinzufügen, Entfernen, Auswechseln, Aktivieren und Suchen von Komponenten.
- Aus Sicht eines Klienten ist es nicht erkennbar, ob er mit einer lokalen oder entfernten Komponente kommuniziert.

Broker Beteiligte



Broker Beispiele

- Die Broker Architektur war das grundlegende Muster für die Entwicklung von CORBA. Der Common-Object-Request-Broker-Architecture.
- Eine Abwandlung ist in der J2EE EJB und der .Net Plattform zu finden.
- Auch Java RMI beinhaltet Grundprinzipien des Broker Musters. Registry und Naming Service sind Teile des Brokers. Stub- und Skeleton-Klassen werden als die Client- und Server-Proxys generiert.

Broker: Vorteile

- Standort unabhängig: Der Broker kümmert sich um das Auffinden von Services.
- Einfache Änderbarkeit und Erweiterung von Komponenten, falls die Schnittstellen stabil sind.
- Portierbarkeit eines Broker Systems.
- Interoperabilität zwischen Broker Systemen.
- Wiederverwendung der Komponenten.

Broker: Nachteile

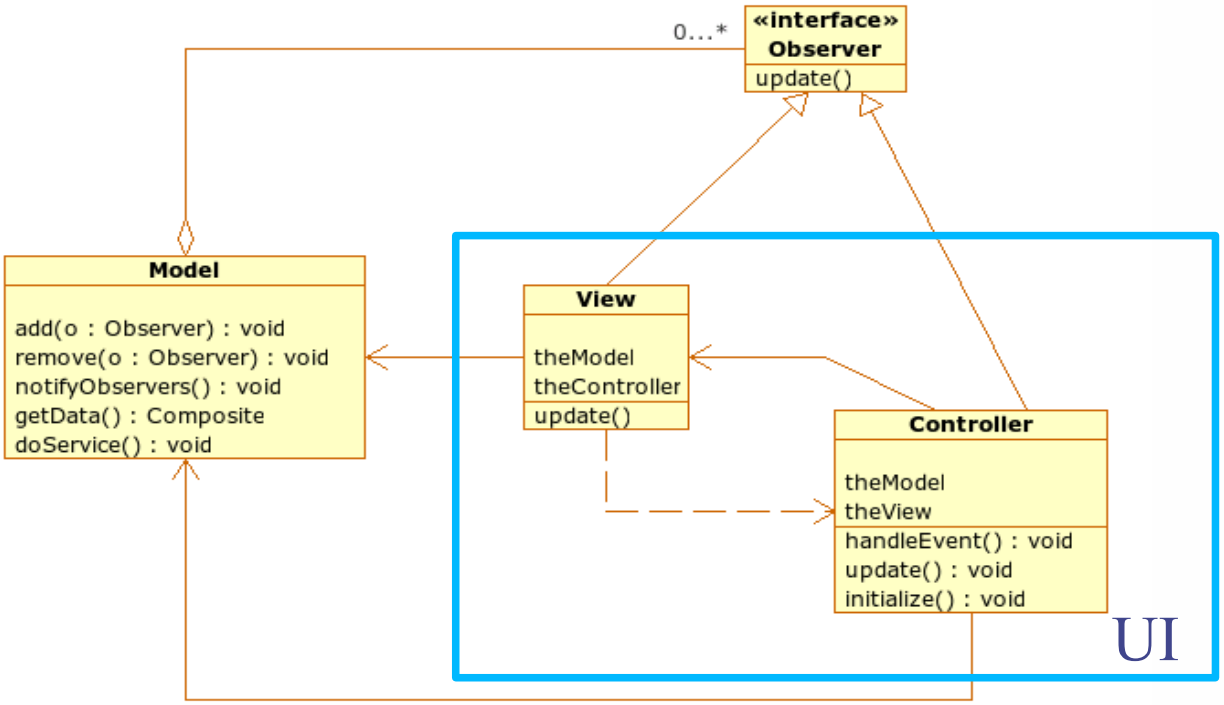
- Eingeschränkte Effizienz, hohe Netzwerklast.
- Niedrige Fehlertoleranz
- Schwieriges Testen und Debuggen.

- Die versprochene Interoperabilität zwischen den Broker Systemen wurde nie erreicht!
- Wiederverwendung funktioniert meist nur effizient innerhalb eines Servers.

Model-View-Controller

- Das Model-View-Controller Muster unterteilt eine interaktive Anwendung in drei Komponenten:
- Das Modell enthält die Kernfunktionalität und die Daten.
- Die View präsentiert dem Anwender die graphische Repräsentation der Daten.
- Der Controller behandelt die Benutzereingaben.
- View und Controller zusammen bilden die Benutzerschnittstelle. Ein Mechanismus zur Benachrichtigung über Änderungen sichert die Konsistenz mit dem Modell. => Observer-Pattern

MVC statische Sicht



- Der Controller akzeptiert Eingaben und ruft die service-Methode des Modells.
- Das Modell ändert seinen Status und informiert alle Observer.
- Alle Views passen ihre Ansicht entsprechend den neuen Daten an.
- Der Controller schaltet entsprechende Elemente der View an oder aus.

MVC: Vorteile

- Mehrere Ansichten auf das selbe Modell.
- Automatische Synchronisation aller Ansichten.
- Austauschbarkeit von Ansicht und Controller.
- Gute Trennung von Modell und VC.
- Potential für vorgefertigte Frameworks.

MVC: Nachteile

- Größere Komplexität
- Potential für unnötige häufige Aktualisierungen
- Enge Verbindung zwischen View und Controller
- Enge Kopplung von VC an das Modell.
- Häufig ineffizienter Datenzugriff auf das Modell.
- View und Controller sind schwer zu portieren.

Zusammenfassung

- Architekturmuster bilden gegenwärtig die oberste Abstraktionsebene eines Software-Designs.
- Sie erleichtern die Paketierung eines Frameworks oder einer Anwendung.
- Sie erleichtern das Entkoppeln von Klassen, indem hochfrequente Kommunikationsanteile innerhalb eines Paketes liegen.
- Das DIP und IoC Prinzip hat in Frameworks wie z.B. Spring in den letzten Jahren Einzug in alle Software Entwürfe gefunden.