


Softwaretechnik

Kapitel 4




1. Wieso Software Engineering?
2. Vom Einzelkämpfer zum Großprojekt
- Systematische Softwareentwicklung**
3. *Anforderungen und Test: Basis des Projekts*
- 4. Entwurf: Strukturen und nicht-funktionale Eigenschaften**
5. Entwürfe notieren mit UML: *Modelle im SE*
6. Design Patterns: *Entwurfserfahrungen nutzen*
7. Management: *Technik und Projektmanagement*

Leibniz Universität Hannover

SWT 2015/16 - 149

Softwaretechnik

Inhalt von Kapitel 4



Systematische Softwareentwicklung

4. Entwurf: Strukturen und nicht-funktionale Eigenschaften


- Kommunikation – nicht nur mit Rechnern
- Schnittstellen und APIs
- Prinzipien Kohäsion und Kopplung
- Architektur und Entwurfsebenen
- Erfahrungen und Design Patterns
- Anforderungen – Entwurf – Codierung

Leibniz Universität Hannover

SWT 2015/16 - 150


Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Kommunikation über große Systeme

- Worüber kommunizieren?
 - Auch über zukünftigen Code
 - Nicht zu viele Details
 - Nur die Strukturen
- Zu welchem Zweck kommunizieren?
 - Zur Orientierung
 - Zum Erklären
 - Arbeitsaufteilung
 - Diskussion
 - Dokumentation




```
public Entry collapseEntries() {
    Entry first, second;
    HelperEntry merged;
    while (entries.size() > 1) {
        first = entries.removeFirst();
        second = entries.removeFirst();
        merged = new HelperEntry();
        merged.setFrequency(first.getFrequency() + second.getFrequency());
        merged.setOne(first.getOne() + second.getOne());
        merged.setZero(second.getZero());
        entries.add(merged);
    }
    return entries.getFirst();
}

public Entry collapseEntries() {
    Entry first, second;
    HelperEntry merged;
    while (entries.size() > 1) {
        first = entries.removeFirst();
        second = entries.removeFirst();
        merged = new HelperEntry();
        merged.setFrequency(first.getFrequency() + second.getFrequency());
        merged.setOne(first.getOne() + second.getOne());
        merged.setZero(second.getZero());
        entries.add(merged);
    }
    return entries.getFirst();
}
```

Kurt Schneider

SWT 2015/16 - 151

Das soll illustrieren: Sehr große Programme strukturiert man besser, statt sich alle zig-tausend Codezeilen anzusehen.




Entwurf

- Einem neuen Softwaresystem Struktur geben
 - also: strukturieren, was es noch nicht gibt
 - für die vielen Entwickler, die es bauen sollen: Vorgabe
 - nur Strukturen, Details meiden (info. Hiding)
- Entwurf aufschreiben
 - Oft mit graphischen Modellen (UML, folgt später)
 - Überlegungen und Entscheidungen textuell erklären
 - Zielgruppe: Entwickler
 - Aufgabe: ohne Missverständnisse in Code umsetzen
- Ziele
 - Code verschiedener Entwickler muss zusammenpassen
 - Qualitätsanforderungen strukturell unterstützen
 - Entwicklung und Wartung eine Richtung geben
 - Struktur auf Dauer verständlich machen und dadurch erhalten

Modelltheorie

- Original?
- für wen?
- Zu welchem Zweck?
- Was ist relevant?

Kurt Schneider Leibniz Universität Hannover SWT 2015/16 - 152

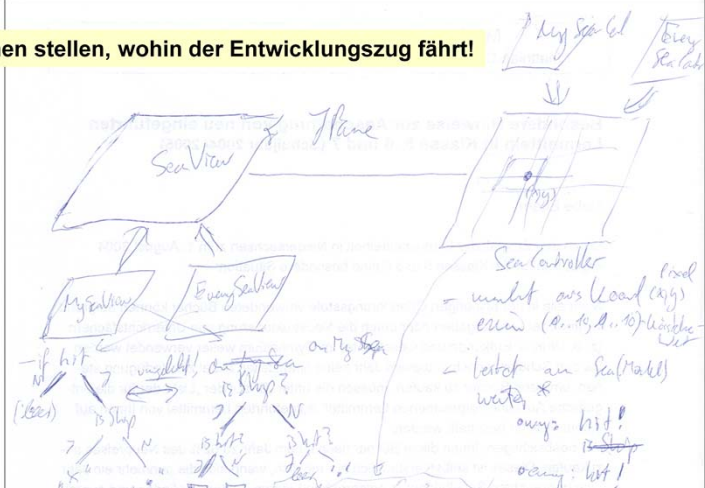


SOFTWARE
ENGINEERING

Zwischenziele des Entwurfs

- Entwurfsideen kommunizieren
- Alternativen vergleichen und diskutieren
- Kreativität nutzen, aber Ergebnisse sichern
- Erfahrungen einfließen lassen

Weichen stellen, wohin der Entwicklungszug fährt!



Kurt Schneider

SWT 2015/16 - 153


Auch unbewusst wird entworfen: Diese kaum leserliche Handskizze ist ein „echter Entwurf“ für ein Schiffe-Versenkien-Spiel.

Man sieht nur wenige Strukturen, Vererbungspfeile und praktisch keinerlei Details.

So ein Entwurf kann schon als Übersicht dienen, um sich in der Implementierung nicht zu verirren.

Wir streben aber professionelle Lösungen für große Projekte an, und die werden nicht mit Handskizzen auskommen.

Dafür braucht man dann die UML. Aber die Idee ist dieselbe: Struktur vor Detail festlegen.



Entwurfstufen

- **Top-Down Entwurf**
 - **Architektur/Grobentwurf:** Komponenten und Grobstruktur
 - **Feinentwurf:** Datenstrukturen, Algorithmen und Schnittstellen
 - **Code:** Ausfüllen der Strukturen, Erfüllen der Schnittstellen
- **Reifestufen der Entwurfsdokumentation**
 1. Skizze an der Tafel
 2. Abgezeichnet in grafischer Notation (meist UML): „Draft“
 3. Überarbeitet
 4. Freigegeben, verbindlich
- **Überlegenswert**
 - Wird Entwurf aktualisiert oder weggeworfen?
 - Wozu dient ein UML-Modell genau?
 - Was tut man, wenn man bei Codierung Entwurfsmängel sieht?

Kurt SchneiderLeibniz Universität HannoverSWT 2015/16 - 154

Reifestufen bedeuten: Man kann mit einer Handskizze anfangen, wird sie in einem echten Projekt aber nach und nach in einem UML-Werkzeug erfassen.

Schließlich muss jemand (der Projektleiter? Oder die Chef-Architektin?) entscheiden, dass der Entwurf so verwendet werden darf.

Das nennt man Freigabe. Danach kann der Entwurf in der Implementierung umgesetzt werden.

SOFTWARE
SE
ENGINEERING

Strukturen sind wichtiger als Details

- **Struktur legt Möglichkeiten fest**
 - Manches ist damit sehr einfach
 - Anderes sehr schwer
- **Es gibt kaum Patentrezepte**
 - Teile und Herrsche
 - Kohäsion und Kopplung
 - ...
 - Information Hiding




Struktur prägt
Verhaltensmöglichkeiten

Kurt SchneiderLeibniz Universität HannoverSWT 2015/16 - 155

Objektorientierung setzt auf Durchgängigkeit.

Dazu gehört, dass nicht nur in der Analyse, sondern auch im Entwurf wieder Klassen und Objekte auftauchen. In beiden Feldern wird UML verwendet.

Bisher sind wir bei den Klassen, aber die Modellierung des Verhaltens ist noch kaum vorgekommen (ein wenig bei den Use Cases). Das ist nun der nächste Punkt, bevor die UML-Notation auch für die Design Patterns eingesetzt wird.



Grundprinzip: Teile und herrsche

- **Großes Programm:**
 - Viele Aspekte, man kann nicht an alles denken
- **Lösung: Programme und damit Probleme zerlegen**
 1. Ein großes Programm wird in mehrere kleinere zerlegt
 - Dann werden die kleinen rekursiv weiter zerlegt
 - Bis die Programme/Aufgaben überschaubar sind
 - Dann sind Probleme lösbar
 2. Teil-Lösungen werden zusammengesetzt
 3. Damit sollte das große Problem gelöst sein
- **Frage dazu:**

Wie funktioniert „systematisches Zerlegen“ und wie beschreibt man das Ergebnis davon?

Kurt Schneider

Leibniz Universität Hannover

SWT 2015/16 - 156

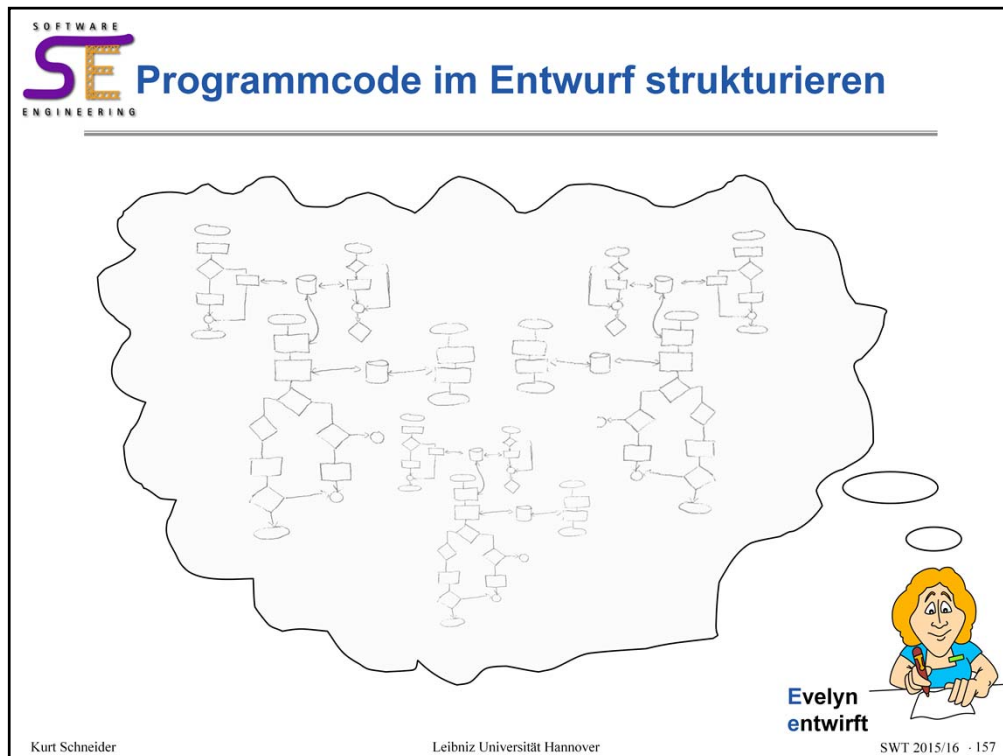
Eine der wenigen Hilfen beim Entwurf:

Große Probleme macht man einfacher, indem man sie zerlegt, die Teile löst, und dann wieder zusammensetzt.

Das war übrigens eine der Ingenieurspraktiken; hier taucht sie in der SW wieder auf.

Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



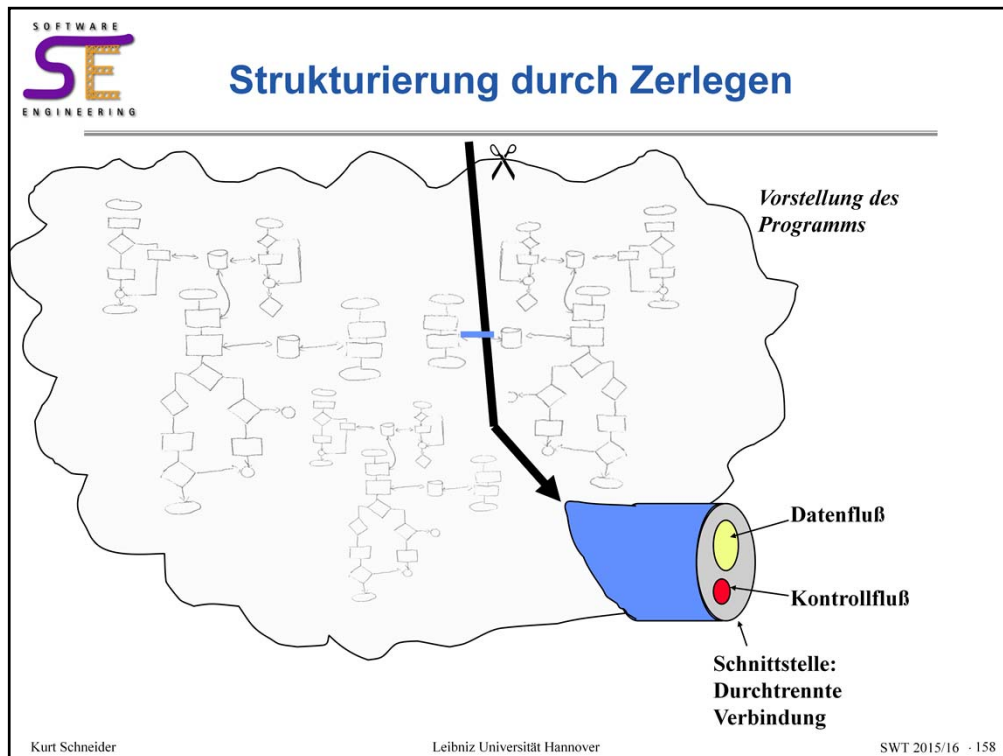
Und nun geht es um die Struktur des Code.

Die Folie stellt eine „komplizierte Codestruktur“ dar, die man nur unscharf sieht.

Bevor das Programm erstellt ist, und bevor man etwas schriftlich entworfen hat, ist diese Struktur ja nur in den Köpfen der Beteiligten.

Hier ist es nur eine Entwicklerin, da ist es noch einfacher.

Sollen sich mehrere Entwerfer (Designer) über die Struktur unterhalten und sie gemeinsam bearbeiten, muss man sie irgendwie hinschreiben können.




Strukturierung in einem solch wolkigen Gebilde heißt nun zunächst: Das Gebilde zu zerteilen.

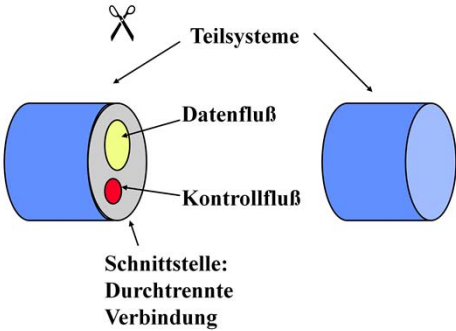
Dort, wo man es zerteilt oder zer-“schneidet“, entsteht eine „Schnittstelle“.



Schnittstellen sind für Software und Code ein wichtiges, elementares Strukturierungsmittel – jede Strukturierung führt zu Schnittstellen und umgekehrt.

Die Schnittstelle bezieht sich auf den Fluss von Daten und von Steuerung bzw. „Kontrolle“. Beides muss über die Schnittstelle, also die durchtrennte Verbindung zwischen den getrennten Teilen, hindurch.



Schnittstellen beschreiben



- Was gehört zu einer Schnittstellenbeschreibung?
 - Name der Funktion oder Methode
 - Sichtbarkeit: von wem aufrufbar?
 - Welcher Parametertyp
 - Welcher Rückgabotyp
- Zusatzinformationen
 - Sinn der Schnittstelle: wer nutzt sie für welchen Zweck?
 - Pre-/Postconditions
 - Über Schnitt-Stellen kann man *verhandeln und sie ändern.*


Kurt Schneider Leibniz Universität Hannover SWT 2015/16 - 159

Man kann sich genauer überlegen, was zu einer Schnittstelle gehört.

Die obigen Angaben sind schon technisch erforderlich. Ohne sie ist die Schnittstelle nicht eindeutig und kaum verwendbar. „Verwendbar“ heißt hier, dass jemand ein weiteres Teilsystem schreiben kann, das sich derselben Schnittstelle bedient.

Beachten Sie aber bitte auch die Zusatzinformationen. Sie werden kaum technisch ausgenutzt werden (allenfalls die Pre-/Postconditions), sind aber doch von großem Wert.

Und zwar erlauben Sie den menschlichen Weiter-Entwicklern der Software, die Hintergründe der Zerlegung (Strukturierung) zu erkennen. Angaben über den Grund von Entwurfsentscheidungen nennt man *Design Rationale*. Strukturierung und Schnittstellenbildung sind wichtige Entwurfs-entscheidungen.



Java-Signatur einer Methode

Schnittstellen-Beschreibung für den Compiler

DEFINITION Die **Signatur** einer Java-Methode umfasst :=_{Def} „Methodennamen und Parametertypen“

ACHTUNG: nicht den Rückgabewert (trotz Wikipedia)

Alles, was Java wissen muss, um die Methode eindeutig zu identifizieren

GENAUER „Two methods have the same **signature** [=_{Def}] if they have the same name and argument types:

KOMPLIZIERT bei Generics (z.B. List<Type>)

Two method or constructor declarations M and N have the **same argument types** if all of the following conditions hold:

- Have same numbers of formal and of type parameters
- Let $\langle A_1, \dots, A_n \rangle$ be the formal type parameters of M and
- let $\langle B_1, \dots, B_n \rangle$ be the formal type parameters of N.
- After renaming each occurrence of a B_i in N's type to A_i the **bounds of corresponding type variables** and the argument types of M and N **are the same.**“

Java Language Specification, 3rd Edition, 2005.
Download: <http://java.sun.com/docs/books/jls/>

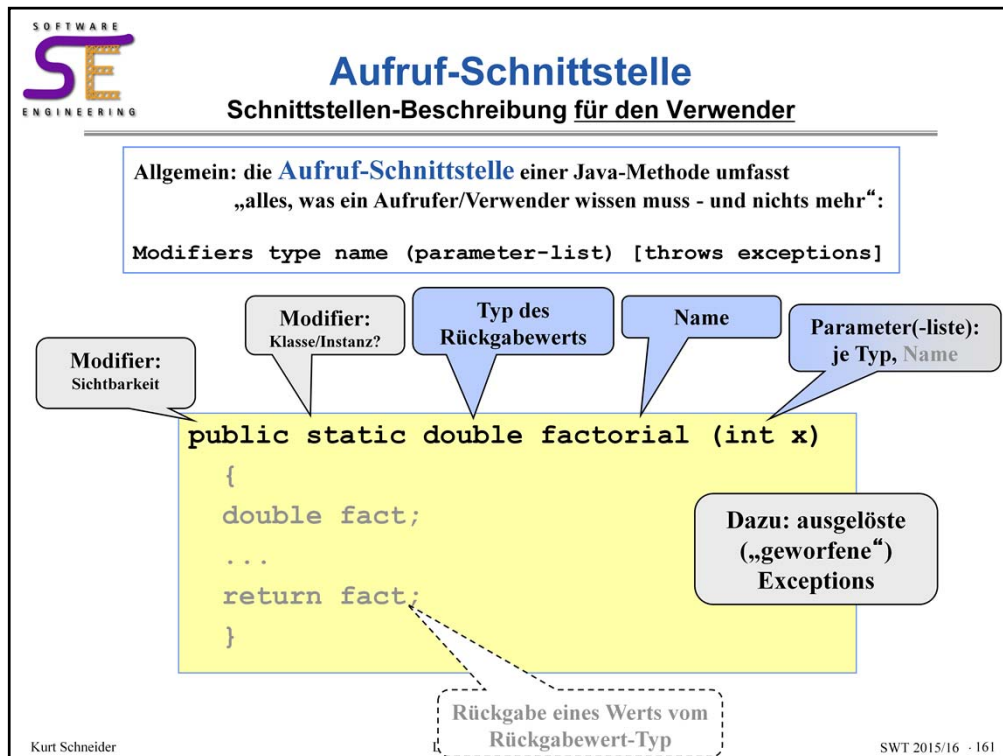
Kurt Schneider

Leibniz Universität Hannover

SWT 2015/16 - 160

Oben ist es genauer definiert, was eine Signatur in Java ist: Methodennamen und Parametertypen (ganz wie in der Mathematik).


Allerdings gibt es Komplikationen (unten).



Eine Aufrufsschnittstelle ist mehr als die Signatur.

-Die Signatur enthält nur, was der Compiler braucht, um herauszufinden, welche Methodenrealisierung beim Aufruf gemeint ist. Dazu braucht er die Parametertypen und den Operationsnamen, aber nicht den Rückgabewert; schließlich dürfen sich auch polymorphe (vererbte) Methoden nicht nur durch Rückgabewerten unterscheiden.

-Der Aufrufer/Verwender einer Methode möchte aber noch mehr wissen: Welcher Typ wird zurückgegeben? Welche Exceptions werden möglicherweise ausgelöst („geworfen“)? Und welche modifier gelten für die Methode (Sichtbarkeit usw.)?



API: Aufruf- und Bedienungsanleitung

Komfortable Schnittstellen-Beschreibung

Allgemein:
Application Programming Interface (API) =_{Def}
eine dokumentierte Software-Schnittstelle, mit deren Hilfe ein Programm bestimmte Funktionen eines anderen Programms nutzen kann

getImage

```
public Image getImage(URL url,  
                     String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument is a specifier that is relative to the `url` argument. This method always returns immediately, whether or not the image exists. If the image does not exist, the data will be loaded. The graphics primitives that draw the image will then use the loaded image.

Parameters:
`url` - an absolute URL giving the base location of the image
`name` - the location of the image, relative to the `url` argument

Returns:
the image at the specified URL

See Also:
[Image](#)

Dokumentation

Was tut die Funktion?

Contract:
Voraussetzungen?
Zusicherungen?
Worauf kann sich Nutzer verlassen?

Kurt Schneider

Leibniz Universität Hannover

SWT 2015/16 - 162

Noch weiter gefasst: API


Das, was man aus Javadoc erzeugen kann, ist oft die API.

Allerdings muss man dann darauf achten, dass nur Informationen wiedergegeben sind, die zur Verwendung nötig sind – eventuell auch Beschreibungen.

Dagegen muss man verhindern, dass irgend ein interner Entwicklerkommentar als Javadoc gekennzeichnet wird und dann mitgeneriert wird.

Im Gegensatz zur Aufrufschnittstelle stehen hier auch inhaltliche Angaben, meist ist alles schön formatiert und verlinkt.

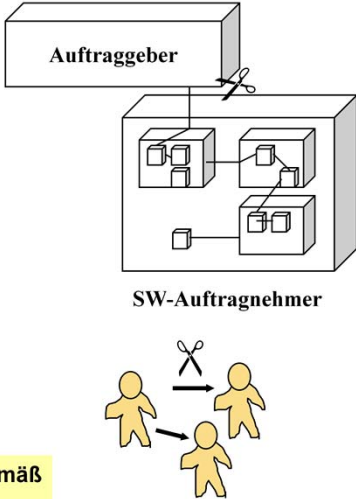
Die API ist praktisch die Schnittstelle einer ganzen Klasse oder eines Systems, so wie ein Java-Interface für eine Klasse steht.



Organisatorische Schnittstellen

- Zwischen Auftraggeber und Auftragnehmer
- Zwischen einer Business Unit und der anderen
- Zwischen einem Team und dem anderen
- Zwischen einzelnen Bearbeitern

Auch für diese Schnittstellen gelten sinngemäß die selben Regeln und Erfahrungen!



Kurt Schneider

Leibniz Universität Hannover

SWT 2015/16 - 163

Eine Bemerkung am Rande:

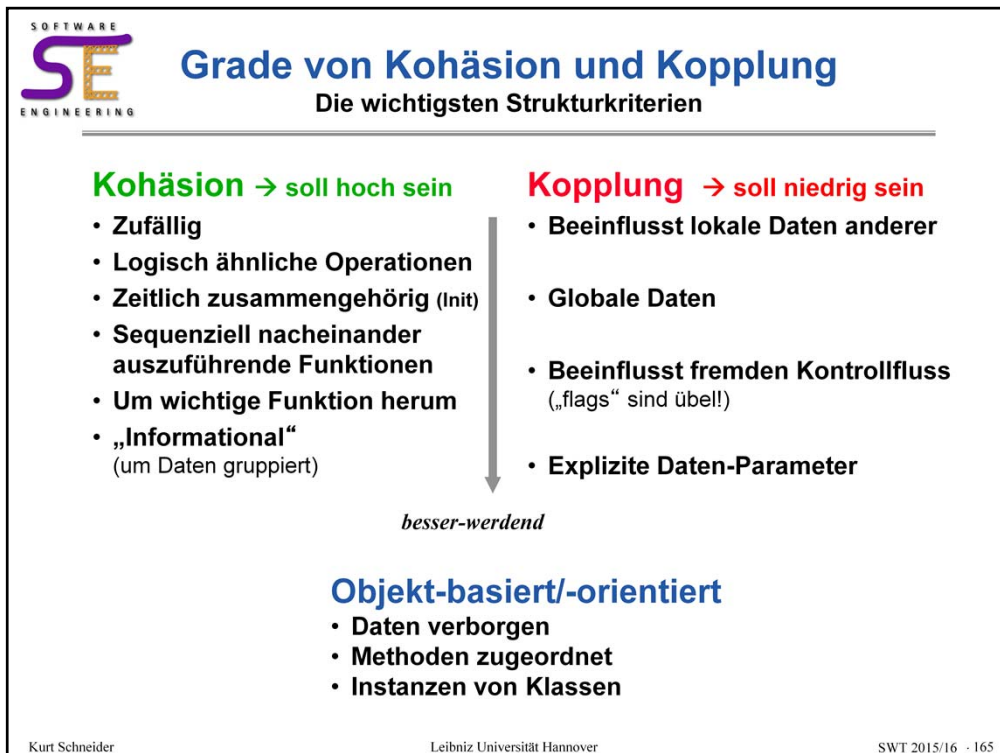
Komplizierte Strukturen treten nicht nur in Code oder den „zugehörigen Dokumenten“ auf, sondern auch in Organisationen (Aufbauorganisation oder Ablauforganisation).

In erster Näherung lassen sich die Aussagen über Schnittstellen analog auf Organisationseinheiten übertragen. Auch hier fließen „Daten“ und „Kontrolle“. Und hier wie dort lassen sich (ähnliche) Regeln aufstellen, wo man sinnvollerweise schneiden sollte.



Zusammenfassung: Schnittstellen

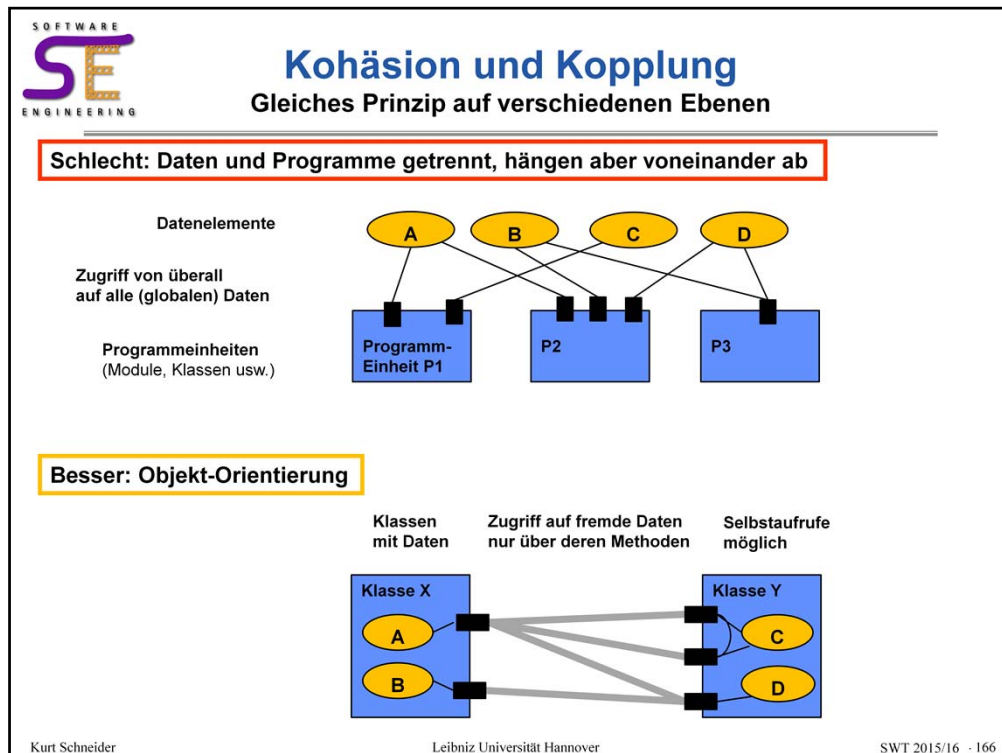
- Technische Schnittstellen sind exakt festzulegen
 - Programmiersprachen-spezifisch (Signatur, Aufruf, API)
 - Paketzugehörigkeit, Aufrufart usw.
 - Schnittstellen-Beschreibungen beachten!
- Organisatorische Schnittstellen so genau wie möglich
 - Organisationseinheiten und Personen
 - Aufgaben, Kompetenzen, Verantwortlichkeiten (AKV)
 - Genaue Regeln bei der Übergabe von einem zum anderen
- Das reicht noch nicht: Zusatzinformationen nicht vergessen
 - Sinn der Schnittstelle: *wieso und wieso hier geschnitten?*
 - Pre-/Postconditions
 - Ergebnis der Schnittstellen-Verhandlungen



Die Grade von Kohäsion und Kopplung werden nach unten immer „besser“. Die Objektorientierung ist praktisch angetreten, beide Stränge fortzusetzen und durch ihre Kombination noch weiter zu verbessern.

Oder anders gesagt: Wenn Sie „richtig“ objekt-orientiert programmieren und auch schon im Entwurf die Konzepte der Objektorientierung einsetzen, dann führt das zu guter Kohäsion und Kopplung. Im Detail müssen Sie immer noch einmal nachkontrollieren und trotzdem noch selbst darauf achten. Beispielsweise, indem Sie die Paketstruktur und die Methodenaufteilung nach Kohäsion und Kopplung verbessern.

„Schlecht objektorientiert“ würde heißen, dass Sie zwar die Syntax von Java oder C++ verwenden, aber eigentlich prozedural (wie in C, Fortran oder Pascal) programmieren. Die Syntax alleine hilft nicht viel.

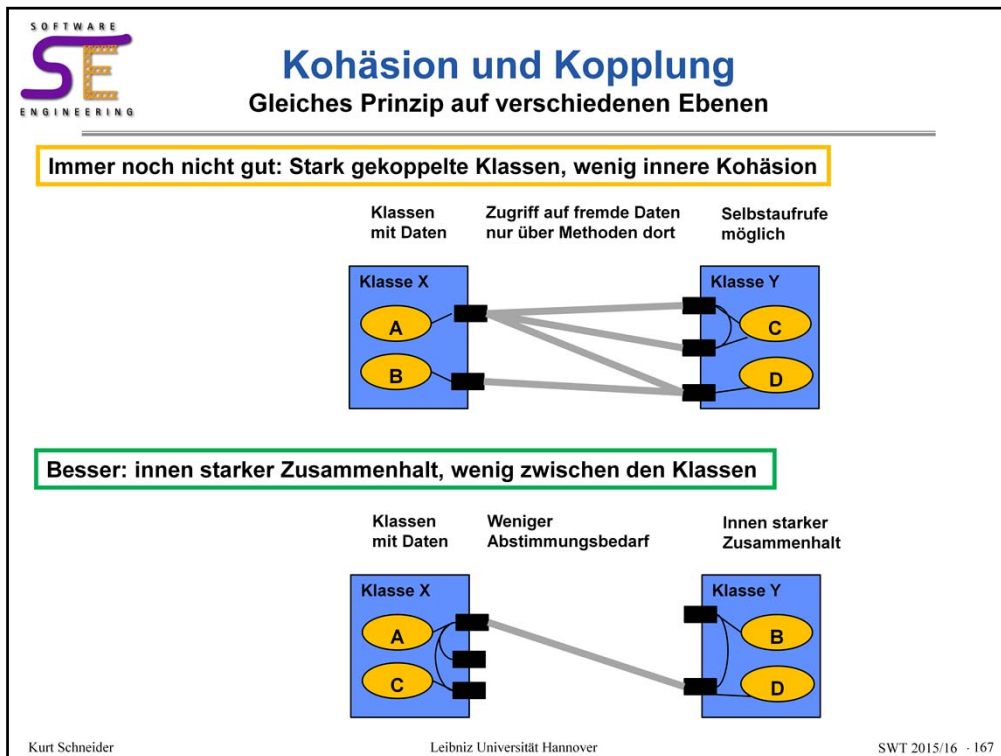


Hier noch einmal die wesentliche Idee hinter den Entwurfskriterien Kohäsion und Kopplung.

Ganz oben einige (globale oder gemeinsame) Datenelemente. Alle Funktionen greifen darauf zu. Wenn einer etwas daran ändert, wird das Verhalten der anderen Operation dadurch beeinflusst. Jeder muss eigentlich jeden kennen.

Schon besser in der Objektorientierung, wo die Daten immer in Objekten gekapselt sind. Nur mit den eigenen Operationen kann man darauf zugreifen. Wenn ein anderes Objekt (der selben oder einer anderen Klasse) etwas tun möchte, dann kann es sich nur an die Operationen wenden. Es kann sein, dass das auch die Daten verändert, muss aber nicht. Es ist verborgen. Information Hiding.

Oben gibt es aber recht viele Aufrufe zwischen den Klassen. Das ist fast so schlimm wie vorher die Zugriffe auf die gemeinsamen Daten. Alles hängt eng zusammen und kann sich nicht unabhängig voneinander verändern.



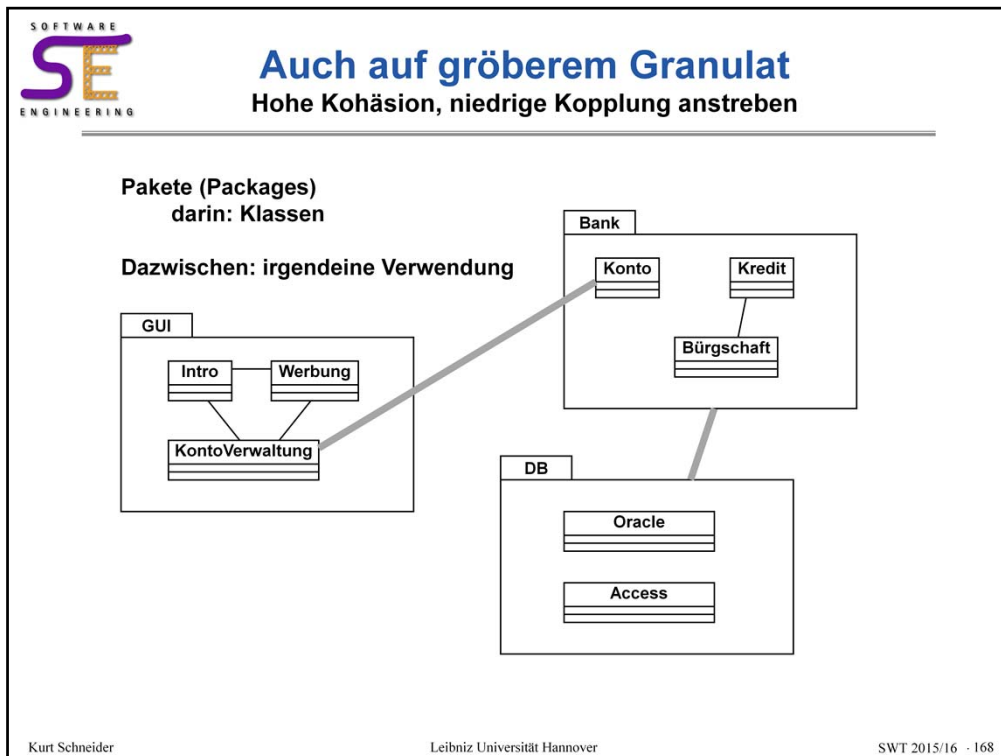
Das sieht man hier.

Daher werden die Klassen anders angeordnet (oder „geschnitten“, wie man sagt). Nun hat man die Dienste so angeordnet, dass die meisten Aufrufe innerhalb einer Klasse bleiben: starke Kohärenz.

Zwischen den Klassen bzw. deren Objekten kommt es aber kaum zu Aufrufen. Damit müssen die Klassen auch nicht so viel voneinander wissen: Schwache Kopplung.

Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Hier noch ein Beispiel auf Paket-Klassenebene.


Hier gilt dasselbe.

Die DB zeigt nur verschiedene Varianten, das ist aber schlechte Kohäsion, denn die brauchen einander nicht.

Dagegen ist die GUI eng verknüpft, das ist gute Kohäsion.

Insgesamt erkennt man die drei Schichten GUI-Bank-DB (später als „Drei-Schichten-Architektur“ eingeführt).

Deren Idee ist ja, die Kopplung zwischen den drei Bereichen („Schichten“, hier in Paketen gefasst) zu verringern. Das ist ganz gut gelungen.




Weitere Entwurfsregeln

genauer hingesehen

- Für schwache Kopplung
 - Wenige Schnittstellen: nicht viele Verbindungen
 - Kleine Schnittstellen: wenig Informationen austauschen
 - (Wenig Interaktion über Schnittstellen) – das ist nicht so wichtig!
- Klare Verhältnisse schaffen
 - Nur explizite Schnittstellen, keine impliziten Nebenwirkungen
 - Dokumentieren, ob sich Einheit noch ändert (Stabilität der Schnittstellen)

Kurt Schneider Leibniz Universität Hannover SWT 2015/16 - 169

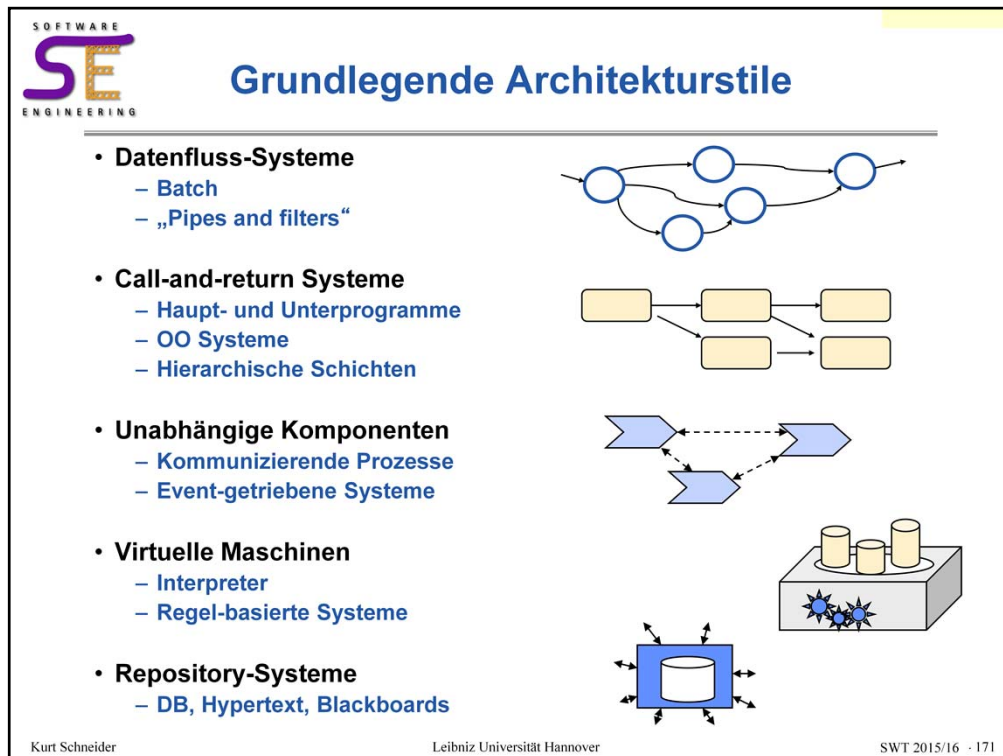


Weitere Entwurfsregeln

genauer hingesehen

- **Once and only once: Keine Doppelungen**
 - Algorithmen doppelt
 - Daten doppelt gehalten
 - Struktur dupliziert
 - ... führt zu Inkonsistenzen und Änderungsanomalien
- **Kognitive Grenzen beachten**
 - Nutzer der Schnittstellen soll möglichst wenig wissen-müssen
 - Interna „geheimhalten“ und verstecken
 - Struktur des Systems begründen und dokumentieren:
 - Hinweis auf ähnliche Strukturen im Anwendungsfeld
 - Hinweis auf ähnliche Teamstruktur (schlechter, da vergänglich)
 - Oder andere, explizit genannte Gründe
- **Architektur := die Lehre von der Struktur und der Strukturierung**

Kurt Schneider Leibniz Universität Hannover SWT 2015/16 - 170



Dies sind fünf grundlegende Architekturmuster.

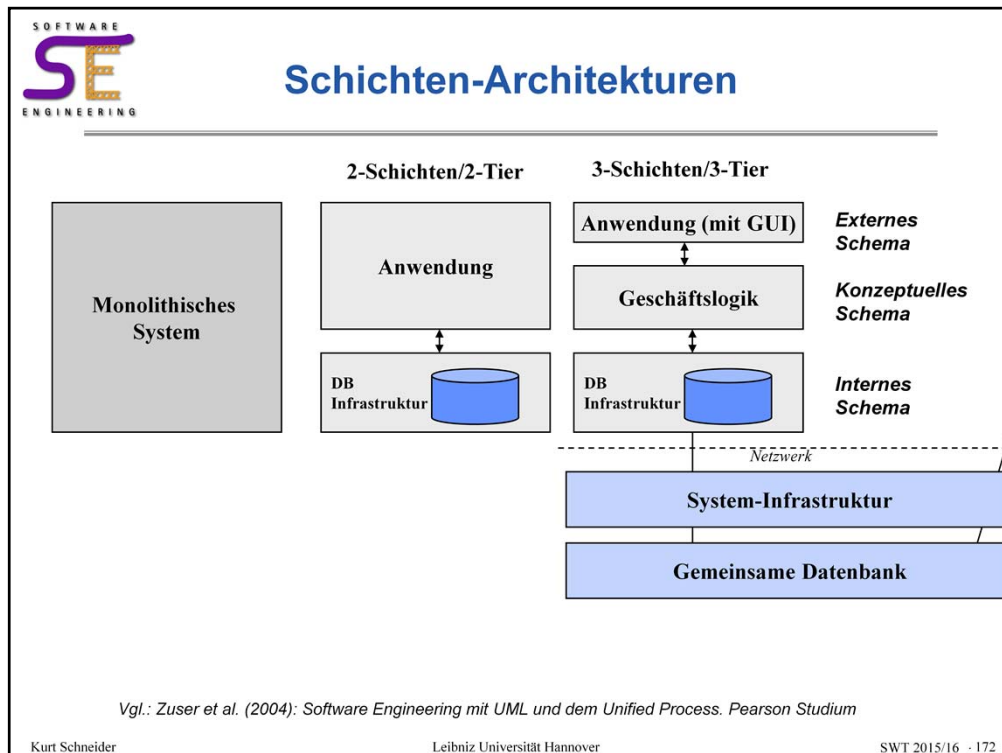
Natürlich werden sie sehr unterschiedlich konkret ausgeprägt. Aber in der Regel fügt sich eine Architektur in eines dieser Muster ein.

Wenn man aber eine Komponente für einen Architekturtyp entwickelt hat, wird sie nicht ohne weiteres in einen anderen passen.

Die Architekturtypen prägen als ihre Komponenten.

Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Links ein Monolithisches System, ohne erkennbare Struktur.

Oft kann man die Persistenz- oder Speicherschicht abtrennen und mit einer klaren Schnittstelle versehen (mit den bekannten Vorteilen).

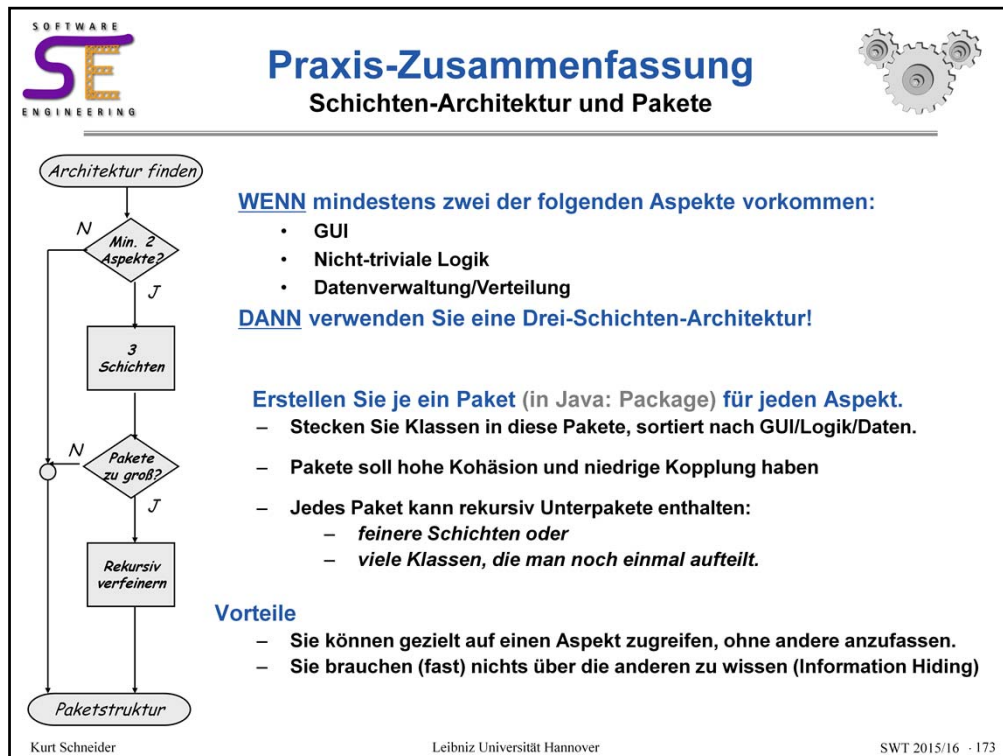
Dann kommt die sehr bekannte Drei-Schichten-Architektur, in der man auch noch die GUI von der „Geschäftslogik“ trennt.

Rechts daneben die entsprechenden Schema-Bezeichnungen für die Schichten.

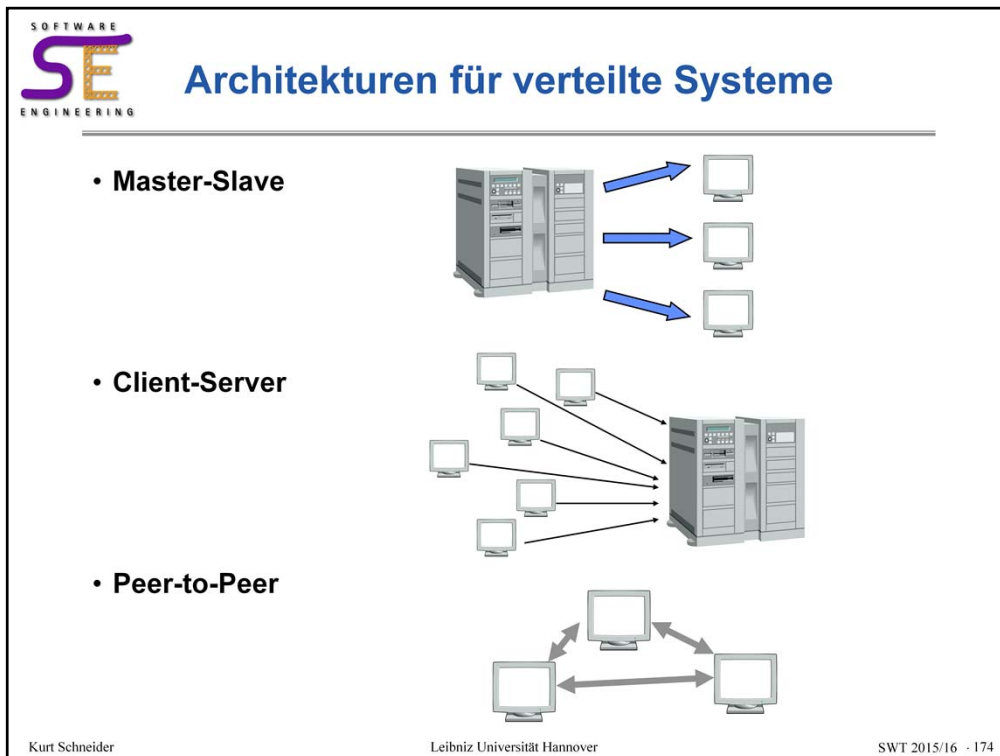
Es kann sein, dass die Speicher- bzw. Persistenzschicht auch eine verteilte Infrastruktur „kapselt“ (d.h.: versteckt im Sinne des Information Hiding).

Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Die Daumenregeln sollen die Zusammenhang herstellen, das Gesagte so zusammenfassen, dass man es praktisch leicht einsetzen kann.



Beim Master-Slave-System herrscht der Master. Die größte Gefahr ist, dass ein Slave verhungert, also gar nicht mehr vom Master versorgt wird.

Bei Client-Server-Systemen erheben eher die Clients hohe Ansprüche an den Server, der nicht unbedingt immer nachkommt.

Im Peer-to-Peer-Netzwerk gibt es keine hervorgehobenen Master oder Server. Jeder serviert dem anderen, was er hat – und nimmt dafür dessen Dienste auf anderem Gebiet in Anspruch. Hier fehlt es an sehr performanten Servern, diese Architektur ist eher für kleine, homogene Anwendungen geeignet.