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*

Softwaretechnik

Inhalt von Kapitel 4

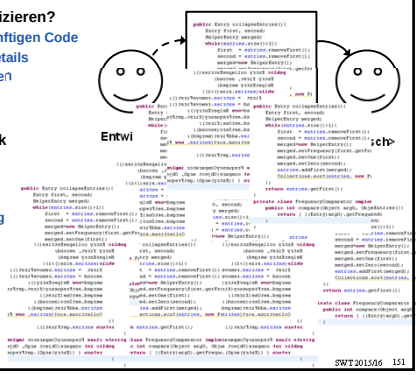
Systematische Softwareentwicklung

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

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



Entwurf

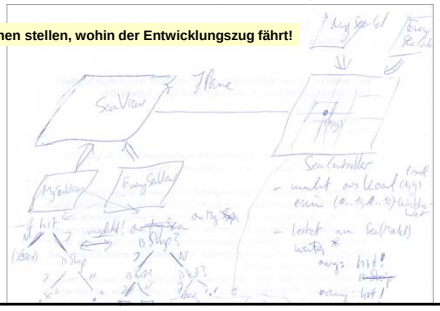
- Einem neuen Softwaresystem Struktur geben
  - also: strukturieren, was es noch nicht gibt
  - für die vielen Entwickler, die es bauen sollen: Vorgabe
  - nur Strukturen, Details meiden (info. Hiding)
- Entwurf aufschreiben
  - Oft mit graphischen Modellen (UML, folgt später)
  - Überlegungen und Entscheidungen 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?

Zwischenziele des Entwurfs

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

Weichen stellen, wohin der Entwicklungszug fährt!



Entwurfsstufen

- 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?

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

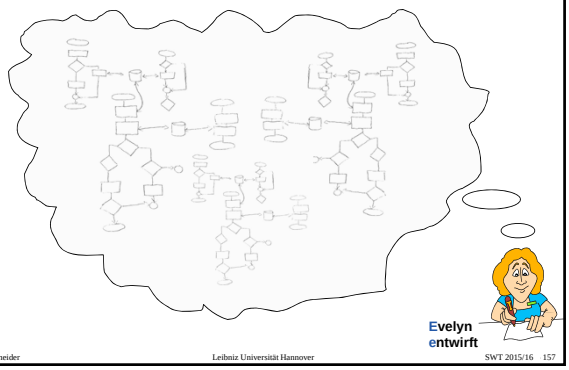
Grundprinzip: Teile und herrsche

- **Großes Programm:**
  - Viele Aspekte, man kann nicht an alles denken
- **Lösung: Programme und damit Probleme zerlegen**

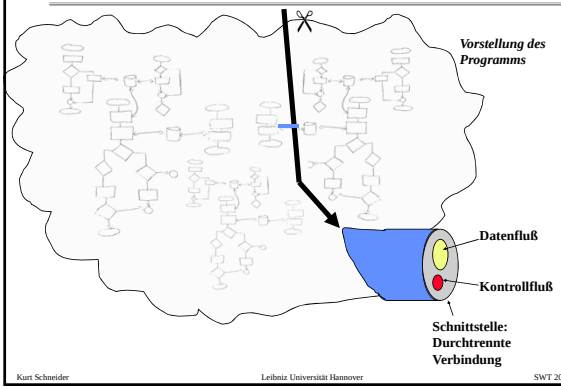
1. Ein großes Programm wird in mehrere kleinere zerlegt
   - Dann werden die kleinen rekursiv weiter zerlegt
     - *Bis die Programme/Aufgaben überschaubar sind*
     - *Dann sind Probleme lösbar*
2. Teil-Lösungen werden zusammengesetzt
3. Damit sollte das große Problem gelöst sein

- **Frage dazu:**
  Wie funktioniert „systematisches Zerlegen“ und wie beschreibt man das Ergebnis davon?

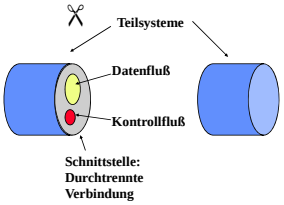
Programmcode im Entwurf strukturieren



Strukturierung durch Zerlegen



Schnittstellen beschreiben



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

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 <A1,...,An> be the formal type parameters of M and
- let <B1,...,Bn> be the formal type parameters of N.
- After renaming each occurrence of a Bi in N's type to Ai 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/

Aufruf-Schnittstelle

Schnittstellen-Beschreibung für den Verwender

Allgemein: die Aufruf-Schnittstelle einer Java-Methode umfasst „alles, was ein Aufrufer/Verwender wissen muss - und nichts mehr“:

```
Modifiers type name (parameter-list) [throws exceptions]
```

Modifier: Sichtbarkeit

Modifier: Klasse/Instanz?

Typ des Rückgabewerts

Name

Parameter(-liste): je Typ, Name

```
public static double factorial (int x)
  {
  double fact;
  ...
  return fact;
  }
```

Dazu: ausgelöste („geworfene“) Exceptions

Rückgabe eines Werts vom Rückgabewert-Typ

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. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

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`

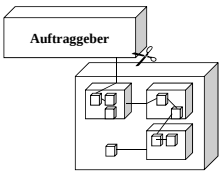
Dokumentation

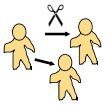
Was *tut* die Funktion?

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

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!

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*

Grade von Kohäsion und Kopplung

Die wichtigsten Strukturkriterien

Kohäsion → soll hoch sein
- Zufällig
- Logisch ähnliche Operationen
- Zeitlich zusammengehörig (Init)
- Sequenziell nacheinander auszuführende Funktionen
- Um wichtige Funktion herum
- „Informational“ (um Daten gruppiert)

Kopplung → soll niedrig sein
- Beeinflusst lokale Daten anderer
- Globale Daten
- Beeinflusst fremden Kontrollfluss („flags“ sind übel!)
- Explizite Daten-Parameter

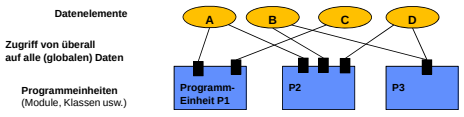
besser-werdend

Objekt-basiert/-orientiert
- Daten verborgen
- Methoden zugeordnet
- Instanzen von Klassen

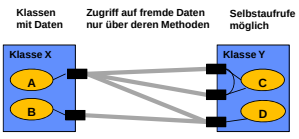
Kohäsion und Kopplung

Gleiches Prinzip auf verschiedenen Ebenen

Schlecht: Daten und Programme getrennt, hängen aber voneinander ab



Besser: Objekt-Orientierung

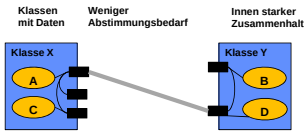


Kohäsion und Kopplung

Gleiches Prinzip auf verschiedenen Ebenen

Immer noch nicht gut: Stark gekoppelte Klassen, wenig innere Kohäsion

Besser: innen starker Zusammenhalt, wenig zwischen den Klassen

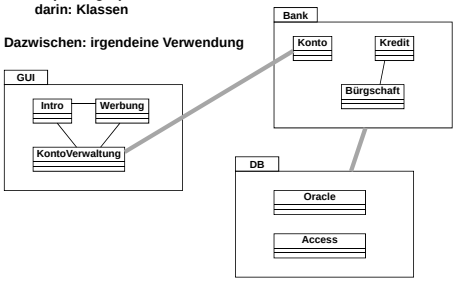


Auch auf gröberem Granulat

Hohe Kohäsion, niedrige Kopplung anstreben

Pakete (Packages)
darin: Klassen

Dazwischen: irgendeine Verwendung



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)

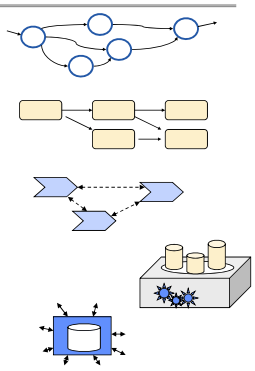
Weitere Entwurfsregeln

genauer hingesehen

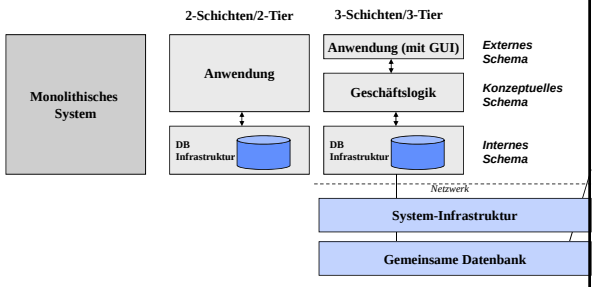
- **Once and only once: Keine Doppelungen**
  - Algorithmen doppelt
  - Daten doppelt gehalten
  - Struktur dupliziert
  - … führt zu Inkonsistenzen und Änderungsanomalien
- **Kognitive Grenzen beachten**
  - Nutzer der Schnittstellen soll möglichst wenig wissen-müssen
  - Interna „geheimhalten“ und 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**

Grundlegende Architekturstile

- **Datenfluss-Systeme**
  - Batch
  - „Pipes and filters“
- **Call-and-return Systeme**
  - Haupt- und Unterprogramme
  - OO Systeme
  - Hierarchische Schichten
- **Unabhängige Komponenten**
  - Kommunizierende Prozesse
  - Event-getriebene Systeme
- **Virtuelle Maschinen**
  - Interpreter
  - Regel-basierte Systeme
- **Repository-Systeme**
  - DB, Hypertext, Blackboards



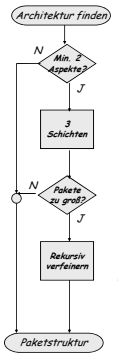
Schichten-Architekturen



Vgl.: Zuser et al. (2004): Software Engineering mit UML und dem Unified Process. Pearson Studium

Praxis-Zusammenfassung

Schichten-Architektur und Pakete



WENN mindestens zwei der folgenden Aspekte vorkommen:

- GUI
- Nicht-triviale Logik
- Datenverwaltung/Verteilung

DANN verwenden Sie eine Drei-Schichten-Architektur!

Erstellen Sie je ein Paket (in Java: Package) **für jeden Aspekt.**

- Stecken Sie Klassen in diese Pakete, sortiert nach GUI/Logik/Daten.
- Pakete soll hohe Kohäsion und niedrige Kopplung haben
- Jedes Paket kann rekursiv Unterpakete enthalten:
  - *feinere Schichten oder*
  - *viele Klassen, die man noch einmal aufteilt.*

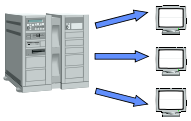
Vorteile

- Sie können gezielt auf einen Aspekt zugreifen, ohne andere anzufassen.
- Sie brauchen (fast) nichts über die anderen zu wissen (Information Hiding)

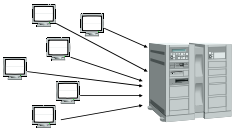
Kurt Schneider — Leibniz Universität Hannover — SWT 2015/16 — 173

Architekturen für verteilte Systeme

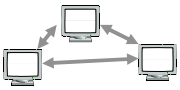
- **Master-Slave**



- **Client-Server**



- **Peer-to-Peer**



Kurt Schneider — Leibniz Universität Hannover — SWT 2015/16 — 174