

# ZUSAMMENFASSUNG

## Was lernen Sie in diesem Modul?

- Anforderungsspezifikationen schreiben
  - o Use Cases (Anwendungsfälle)
- Objektorientierte Analyse – OOA
- Architektur und Design (OOD)
  - o Prinzipien und einige Patterns für guten objektorientierten Entwurf
  - o Architekturentwurf
- Testen
- Konfigurationsverwaltung
- Projektmanagement

## Prüfung

- Closed Book, ohne Unterlagen
- Wissensfragen
- Verständnisfragen
- Ausbildungsinhalte anwenden

<b>EINFÜHRUNG SOFTWARE ENGINEERING .....</b>	<b>11</b>
DEFINITION VON ENGINEERING .....	11
ELEMENTE DES SOFTWARE ENGINEERING.....	11
<i>Prozesse/Projektmanagement .....</i>	11
<i>Prinzipien und Pattern (Architektur und Design).....</i>	12
<i>Modelle.....</i>	12
<i>Notationen .....</i>	12
<i>Werkzeuge .....</i>	13
<b>OBJEKTOIENTIERTE DOMAIN-ANALYSE .....</b>	<b>13</b>
WAS IST OOA? – EINFÜHRUNG .....	13
ANALYSE 1 – ANFORDERUNGSANALYSE .....	13
ANALYSE 2 – DOMAIN-ANALYSE (OOA) .....	14
<i>OOA – „nach Larman“ .....</i>	14
<i>OOA – „klassisch“ .....</i>	15
<i>RUP Disciplines und OOA .....</i>	15
<i>Nutzen der Domain-Analyse .....</i>	16
<i>UML – Elemente für Domainmodell.....</i>	16
<b>DOMAIN-MODELLIERUNG .....</b>	<b>17</b>
ÜBERBLICK .....	17
DOMAIN-MODELL UND DESIGN-MODELL.....	17
IDENTIFIZIERUNG VON KONSEPTIONELLEN KLASSEN .....	17
<i>Strategien zum Finden von Konzepten.....</i>	17
<i>Checkliste mit Kategorien (siehe Larman S.140/141) .....</i>	18
<i>Tipps zum Bereinigen der Konzepte .....</i>	18
<i>Konzept oder Attribut .....</i>	18
<i>Modellierung von technischen Systemen .....</i>	18
<i>Beschreibungen als Konzepte .....</i>	18
<i>Geringer Representational Gap oder auch Semantic Gap.....</i>	19
HINZUFÜGEN VON ASSOZIATIONEN .....	19
<i>Definition Assoziation.....</i>	19
<i>UML für Assoziation im Domain-Modell.....</i>	19
<i>Arten von Assoziationen .....</i>	20
<i>Guidelines zum Modellieren von Assoziationen.....</i>	20
<i>Multiplizitäten .....</i>	20
<i>Namensgebung von Assoziationen.....</i>	20
<i>Assoziationen in Analyse und Design.....</i>	21
<i>Ordnen von Klassen im Domain-Modell gemäss Multiplizitäten.....</i>	21
<i>Achtung bei n-zu-m Assoziationen.....</i>	21
HINZUFÜGEN VON ATTRIBUTEN.....	22
<i>Definition Attribut .....</i>	22
<i>Gültige Attribut-Typen im Domain-Modell .....</i>	22
<i>Keine Attribute an Stelle einer Konzeptionelle Klassen.....</i>	22
<i>Im Domain-Modell: keine Attribute als Fremdschlüssel! .....</i>	22

<i>Mit Einheit behaftete Grössen als Datentypen von Attributen .....</i>	23
GENERALISIERUNG-SPEZIALISIERUNG.....	23
ZUSAMMENFASSUNG DOMAIN-MODELLIERUNG .....	24
<b>REQUIREMENTS – EINFÜHRUNG .....</b>	<b>25</b>
ANFORDERUNGEN UND DISCIPLINES .....	25
REQUIREMENTS: DISCIPLINE UND WORK PRODUCTS .....	25
REQUIREMENTS: 30% ALLER PROBLEME IN DER SOFTWARE-ENTWICKLUNG .....	25
REQUIREMENTS UND WASSERFALLMODELL.....	25
REQUIREMENTS MANAGEMENT .....	25
KLASSIFIKATION VON SOFTWAREANFORDERUNGEN .....	26
WIE (FUNKTIONALE) ANFORDERUNGEN BESCHRIEBEN WERDEN .....	26
<b>REQUIREMENTS – USE CASES.....</b>	<b>27</b>
USE CASE BEISPIEL .....	27
WIESO USE CASES?.....	27
FORMALE DEFINITION .....	27
BEGRIFFE.....	28
FORMAT VON USE CASES .....	28
BEISPIEL „FULLY DRESSED“ .....	28
EINSPALTIG ODER ZWEISPALTIG?.....	28
TEILE EINES USE CASES .....	29
USE CASE EBENEN NACH COCKBURN .....	29
ELEMENTARY BUSINESS PROCESS = EBP .....	29
VORGEHEN USE CASES.....	30
SYSTEM UND SYSTEMGRENZEN.....	30
AKTOREN UND ZIELE.....	30
REGELN FÜR USE CASES.....	30
AKTOREN.....	31
ESSENTIAL STYLE .....	31
USE CASE DIAGRAMM.....	31
<i>Include</i> .....	31
<i>extend</i> .....	32
UC-DIAGRAMM MIT BEZIEHUNGEN .....	32
GRAD DER AUSARBEITUNG .....	32
<b>REQUIREMENTS – NICHTFUNKTIONALE ANFORDERUNGEN, SOFTWARE REQUIREMENTS SPECIFICATION .....</b>	<b>33</b>
NICHTFUNKTIONALE ANFORDERUNGEN (NON FUNCTIONAL REQUIREMENTS) NF .....	33
<i>Klassifikation von Softwareanforderungen</i> .....	33
<i>Nichtfunktionale Anforderungen</i> .....	33
<i>Leistung, Mengen &amp; Randbedingungen</i> .....	33
<i>Qualitätsmerkmale</i> .....	33
<i>Qualitätsmodelle</i> .....	34
<i>Funktionale Anforderungen ↔ Qualitätsmerkmal Funktionalität.</i> .....	34
<i>Definition gemäss Larman</i> .....	34
SOFTWARE REQUIREMENTS SPECIFICATION .....	34

<b>MODELLING BEHAVIOUR – ZUSTANDS- UND ACTIVITY DIAGRAMME.....</b>	<b>35</b>
ZUSTANDSDIAGRAMME - STATE MACHINE DIAGRAMMS.....	35
<i>Zustandsdiagramme in UML.....</i>	35
<i>Zustandsdiagramme in Analyse.....</i>	35
<i>Ereignis (Event).....</i>	36
<i>Zustand (state).....</i>	36
<i>Zustandsübergang (transition) .....</i>	36
<i>Zustandsdiagramme für Use Case.....</i>	37
<i>System-Zustandsdiagramm .....</i>	37
<i>Zusätzliche Notationselemente.....</i>	37
<i>Typische Anwendungen von Zustandsdiagrammen .....</i>	37
<i>Vorgehen .....</i>	38
<i>Häufige Stolpersteine.....</i>	38
ACTIVITY DIAGRAMME.....	38
<i>Was sind UML Activity Diagramme und wofür sind Sie gut?.....</i>	38
<i>Basisnotation für Activity Diagramme.....</i>	39
<i>Zusätzliche Notationen für Activity Diagramme .....</i>	40
<i>Beispiel Activity Diagramm – Bestellabwicklung.....</i>	41
<b>SOFTWARE TESTING .....</b>	<b>42</b>
EINFÜHRUNG.....	42
<i>Definition.....</i>	42
<i>Anforderungen.....</i>	42
<i>Arten von Tests.....</i>	42
<i>Testprozess.....</i>	42
<i>Einordnung – Test und andere Qualitätsprüfungen .....</i>	43
<i>Testebenen und Tester.....</i>	43
<i>Begriffe – Verifikation und Validierung.....</i>	43
TESTMETHODEN .....	43
<i>Testumgebung.....</i>	43
<i>Black-Box-Tests und White-Box-Tests .....</i>	44
<i>Black-Box Testmethoden.....</i>	44
<i>White-Box Testmethoden .....</i>	46
<i>Testüberdeckung (test coverage).....</i>	46
TESTAUTOMATISIERUNG.....	47
<i>Testwerkzeuge.....</i>	47
<i>Unit Tests.....</i>	47
<i>JUnit.....</i>	47
TESTPRINZIPIEN .....	47
<i>Das wichtigste am Testen .....</i>	47
<i>„Test First“ Prinzip.....</i>	48
<i>Pragmatische Prinzipien beim Unit Testing .....</i>	48
<i>ATRIP – Gute Uni Tests .....</i>	48
<b>MICROTESTING (E-LEARNING) .....</b>	<b>49</b>

DREI DIMENSIONEN VON MICROTESTING.....	49
WHAT IS A MIRCOTEST?.....	49
<i>Testing in Isolation</i> .....	49
THE STANDARD MIRCOTEST .....	50
<i>Nameing and Scope</i> .....	50
<i>A Sample microtest</i> .....	50
<i>The Universal Strucure of Microtests</i> .....	51
TEST EVERYTHING INTERESTING?.....	51
ALL ABOUT XUNIT.....	51
<i>Principles of Writing xUnit Tests</i> .....	52
<i>Principles of Running XUnit Tests</i> .....	54
SMALL IS SUPERB!.....	54
PRECISE IS PERFECT .....	54
BOUNDARIES ARE BEAUTIFUL! .....	55
INDEPENDET, SIDE-EFFECT-FREE MIRCOTESTS .....	55
<i>Mainfestations Of Test Dependence</i> .....	55
MICROTESTS & EXCEPTIONS.....	55
<b>CLEAN CODE 1.....</b>	<b>56</b>
AUFGERÄUMTE BAUSTELLE.....	56
CODING GUIDELINES .....	56
DON'T REPEAT YOURSELF (DRY) .....	56
<i>Wartungs-Alptraum</i> .....	56
HANDHABbare GRÖSSEN .....	57
NO ERRORS, NO WARNINGS.....	57
KEEP IT SIMPLE .....	57
POSITIVE BEDINGUNGEN.....	57
<i>Schneller Ausstieg</i> .....	57
<i>Komplementäre Mengen</i> .....	58
KOPPLUNG UND KOHÄSION .....	58
SINGLE RESPONSIBILITY PRINCIPLE .....	59
<i>Beispiele für Zuständigkeiten</i> .....	59
<i>Nachteile, wenn diese Regel verletzt ist</i> .....	59
PROGRAM TO THE INTERFACE NOT TO AN IMPLEMENTATION.....	59
ISOLATE WHAT CHANGES.....	59
SMART DATA STRUCTURES .....	60
<i>Datenstrukturen vor Code</i> .....	60
<i>Warum das Datenmodell so wichtig ist</i> .....	60
PROGRAMMIERE FÜR MENSCHEN... NICHT FÜR DEN COMPILER.....	60
GUTE NAMEN .....	60
<i>Keine Zahlen in Namen</i> .....	61
<i>Keinen anonyme Konstanten</i> .....	61
NÜTZLICHE KOMMENTARE .....	61
<i>Mit Vorsicht und Umsicht kommentieren</i> .....	61
KIM GOODWIN .....	62
<b>DESIGN PATTERNS.....</b>	<b>63</b>

PROGRAM TO AN INTERFACE .....	63
STARBUZZ COFFEE .....	64
DECORATOR PATTERN.....	65
<i>Decorator um Decorator herum.....</i>	65
<i>Kostenrechnung.....</i>	65
<i>Definiton.....</i>	65
<i>Decorator für Starbuzz.....</i>	66
<i>Code.....</i>	66
<i>Real World Decorators – java.io.....</i>	66
COMPOSITE PATTERN .....	67
WIE IST EIN DESIGN PATTERN DEFINIERT? .....	67
FACTORY METHOD.....	68
COMMAND PATTERN .....	68
STATE PATTERN .....	69
TEMPLATE METHOD PATTERN .....	69
MVC PATTERN (MODEL-VIEW-CONTROLLER) .....	70
MVC MIT OBSERVER PATTERN .....	71
<i>Beispiele.....</i>	71
<i>MVC im Web-Zeitalter.....</i>	71
NULL OBJECT PATTERN .....	72
ÜBERSICHT .....	72
<b>SOFTWARE ARCHITEKTUR 1 – LAYERS, TIERS &amp; PARTITIONEN .....</b>	<b>73</b>
PACKAGES .....	73
<i>Aufteilungskriterien.....</i>	73
ZUSTÄNDIGKEITEN .....	74
<i>Beispiele für Zuständigkeiten .....</i>	74
SCHICHTEN.....	75
<i>Kunde und Dienstleistung .....</i>	75
<i>Abhängig und unabhängig.....</i>	75
<i>Oben und unten.....</i>	75
DREI-SCHICHTEN-MODELL.....	76
<i>Beispiel für vier bzw. sechs Schichten.....</i>	76
<i>Asymmetrie der Abhängigkeiten .....</i>	76
<i>Vererbung „oben – unten“.....</i>	77
<i>„oben – unten“ bei Datenklassen.....</i>	77
<i>Arten von Kopplung .....</i>	77
PARTITIONEN.....	78
<i>Partitionieren .....</i>	78
<i>Regeln für Abhängigkeiten.....</i>	78
<i>Abhängigkeiten = Kopplung .....</i>	79
LAYERS & TIERS .....	79
<i>Wo welcher Code läuft.....</i>	80
<i>Kommunikation in Layers &amp; Tiers .....</i>	80
PARTITIONEN IN EIGENER LAUFZEITUMGEBUNG .....	80
DATENMODELL KANN AUCH PARTITIONIERT WERDEN.....	81

<i>Abhängigkeiten beim Schneiden</i>	81
BEISPIEL POPPINS	82
ZUSAMMENFASSUNG	82
<b>DEPLOYMENT DIAGRAMME</b>	<b>83</b>
BEISPIEL	83
NOTATION	83
<i>Knoten</i>	83
<i>Nested Execution Environments</i>	84
<i>Assoziationen</i>	84
BEISPIEL LOAD BALANCER	84
ANWENDUNGSBEREICH & NUTZEN	84
MEHRERE DEPLOYMENT-VARIANTEN	85
<i>Architektur-Umbau</i>	85
<i>Neue Deployment-Variante</i>	85
<i>Spezielle Deployment-Variante</i>	86
VERSCHIEDENE STILE	86
<b>UNIFIED PROCESS – DIE MINIMALVERSION</b>	<b>87</b>
GESCHICHTE	87
<i>Vorgehensmodell «Wasserfall»</i>	87
<i>V-Modell (=geknickter Wasserfall)</i>	87
(RATIONAL/IBM) UNIFIED PROCESS	88
<i>Disciplines</i>	88
<i>Roles, Activities, Work Products</i>	89
<i>UP geht (zu) tief ins Detail</i>	89
<i>UP = Riesige Apotheke</i>	89
WAS BEI UP WICHTIG IST	89
<i>Immer iterativ vorgehen</i>	89
<i>Vier Phasen</i>	89
END OF ELABORATION	90
<i>Arbeitsteilung</i>	90
CHECKLISTE – END OF ELABORATION	90
ZUSAMMENFASSUNG	90
<b>SEQUENZDIAGRAMME</b>	<b>91</b>
UML SEQUENZDIAGRAMM	91
<i>Notation (einfach)</i>	91
<i>Notation (komplexeres Beispiel)</i>	91
<i>Beispiele</i>	92
<i>Einsatz in der SW-Entwicklung</i>	93
<i>Nutzen von Sequenzdiagrammen</i>	93
<i>Diagramme sind Kommunikation</i>	93
SYSTEM-SEQUENZDIAGRAMM (SSD)	93
<i>Abstrahiertes Systemverhalten</i>	94
<i>Mehr über SSD</i>	94
<i>Verwechslungsgefahr</i>	94

SEQUENZDIAGRAMM IM VERGLEICH.....	95
<i>Zusammenfassende Gegenüberstellung</i> .....	95
<b>SCRUM EINFÜHRUNG .....</b>	<b>96</b>
DEFINITION OF SCRUM.....	96
PRODUCT BACKLOG.....	96
USER STORIES.....	96
PRODUCT BACKLOG BOARD.....	97
USER STORY .....	97
<i>Kurzes Beispiel</i> .....	97
SPRINTS .....	98
EREIGNISSE.....	98
<i>Sprint Planning</i> .....	98
<i>Daily Standup</i> .....	99
<i>Sprint Review</i> .....	99
<i>Sprint Retrospective</i> .....	99
<i>Vorausplanung für den nächsten Sprint</i> .....	99
ROLLEN.....	99
<i>Product Owner</i> .....	100
<i>Scrum Master</i> .....	100
<i>Mitglieder des Entwicklungs-Teams</i> .....	100
HAUPT-TÄTIGKEITEN.....	100
<i>Das Schätz-Spiel</i> .....	100
<i>Weitere wichtige Tätigkeiten</i> .....	100
<i>Und Sie müssen auch noch...</i> .....	100
<i>Backlog Inhalte</i> .....	101
DEFINITION OF DONE .....	101
<b>SOFTWARE ARCHITEKTUR 2 – MODELLIERUNG IM UML .....</b>	<b>102</b>
VOM DOMAINMODELL ZUM DATENMODELL .....	102
<i>Domainmodell</i> .....	102
<i>Datenmodell</i> .....	102
<i>Architektur-Entwurf</i> .....	103
<i>Einfaches Architektur-Entwurfsmuster</i> .....	103
WAS DIE ARCHITEKTUR BEEINFLUSST .....	104
KEIN CODE-DESIGN IM UML.....	104
<i>Diagramme sind Kommunikationsmittel</i> .....	104
<i>UML vs. Code (Zu nah am Code?)</i> .....	104
<i>Objektorientierte Analyse mit UML</i> .....	106
<i>Objektorientiertes Design mit UML</i> .....	106
<i>Schwierigkeiten mit Diagrammen</i> .....	106
ÜBERSICHT ÜBER ALLE DIAGRAMME UND MODELLE .....	106
<b>FEINHEITEN IN DER DATENMODELLIERUNG .....</b>	<b>107</b>
N:M – BEZIEHUNGEN .....	107
<i>Zwischenlösung A</i> .....	107
<i>Zwischenlösung B</i> .....	107

Konsequenzen.....	108
Fazit für n:m Beziehungen.....	108
1:1 BEZIEHUNGEN .....	108
<i>Beispiel sinnvolle 1:1 Beziehung .....</i>	108
<i>Beispiele 1:0..1 Beziehung .....</i>	109
<i>Beispiel falsche 1:1 Beziehungen .....</i>	109
TEIL-GANZES BEZIEHUNGEN .....	109
<i>Nochmals 1:1 Beziehungen.....</i>	110
<i>Wann 1:1 Beziehungen? .....</i>	110
KLASSEN OHNE ATTRIBUTE.....	110
VERERBUNG OHNE GEMEINSAMKEITEN.....	111
<i>Beispiel .....</i>	111
VERERBUNG IN RELATIONALER DB ABBILDEN.....	111
<i>Table per Class? .....</i>	111
<i>Vererbung im Datenmodell? .....</i>	112
<i>Vererbungspfeile .....</i>	112
GERICHTETE/UNGERICHTETE BEZIEHUNGEN.....	112
<i>Richtung von Beziehungen in Datenmodell .....</i>	112
BEZIEHUNGEN UND ROLLEN ANSCHREIBEN? .....	113
<i>Beziehung mit Rollen .....</i>	113
<i>Wann Beziehungen anschreiben.....</i>	113
<b>SOFTWARE PROJEKT-MANAGEMENT, TEIL 1 .....</b>	<b>114</b>
1. NICHTS VERGESSEN DANK LISTEN .....	114
<i>Prioritäten .....</i>	114
2. PL IS THE KEEPER OF THE SCOPE .....	114
<i>Funktionsumfang (Eckpunkte für die Beobachtung).....</i>	114
<i>Scope Creep.....</i>	114
<i>Software ist Kommunikations-intensiv.....</i>	115
3. GEHEN SIE TÄGLICH AUF DIE BAUSTELLE .....	115
<i>Auf der Software-Baustelle.....</i>	115
4. ENTFERNUNG IST TEUER .....	115
5. ÜBERGABEN SIND TEUER .....	115
6. DIE HOHE KUNST: SICHTBAR MACHEN .....	116
<i>Nützliche Diagramme.....</i>	116
<i>Grafiken für den Fortschritt.....</i>	116
7. IMMER ITERATIV VORGEHEN.....	116
<i>Kurze Iterationen .....</i>	116
<i>Feedback vom Kunden .....</i>	117
8. INSPECT – ADAPT .....	117
<i>Projektkontrolle.....</i>	117
<i>Projekt Geschichtsschreibung .....</i>	117
<i>Projekt-Review nach Abschluss .....</i>	117
9. DAILY BUILD UND BRANCHES.....	118
10. VERSEHEN, WAS DER KUNDE WILL .....	118
<i>Anforderungen, Requirements .....</i>	118

<i>Frage die Endkunden, nicht einen Proxy .....</i>	118
<i>Partner sein, Vertrauen schaffen .....</i>	118
11. SO FRÜH WIE MÖGLICH, SO FORMAL WIE MÖGLICH .....	119
<i>Bilder können ein Projekt retten .....</i>	119
<i>Grad der Schriftlichkeit.....</i>	119
<i>Eingefrorene Spezifikation.....</i>	119
12. DIE VIER PROJEKT-VARIABLEN.....	120
<i>Was tun, wen überbestimmt?.....</i>	120
ZUSAMMENFASSUNG .....	120

# Einführung Software Engineering

Software Projekte laufen oft schief weil, ...

- Die Komplexität nicht beherrscht wird.
- Kosten oder Termin Überschreitungen stattfinden
- Aufgrund mangelnder Qualität
- Lösung des falschen Problems.

Diese Projekte führen in den meisten Fällen zu unzufriedenen Kunden. Es hat bereits viele Desaster gegeben.

## Definition von Engineering

Engineering is the

- Application of scientific, economic, social and practical knowledge in order to
- Invent, design, build, maintain, research and improve structures, machines, devices, systems, materials and processes.

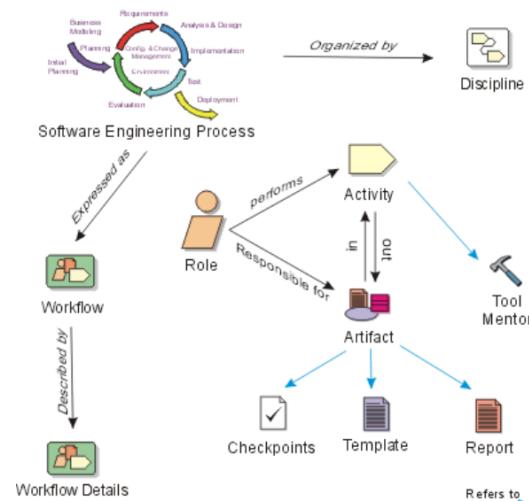
Eine Analogie bildet der Hausbau. Einen Unterstand für einen Rasenmäher kann ich in einem Nachmittag mit Holz, Säge, Hammer und Nägel problemlos aufbauen. Mit dem gleichen Material lassen sich aber nicht ein normales Haus oder ein richtig grosses Spital bauen. Dafür sind andere Materialien nötig.

Software Engineering bedeutet daher adäquate Mittel einzusetzen. Also auch nicht zu viel oder zu grosses Werkzeuge, Material, Techniken, Personal und Planung.

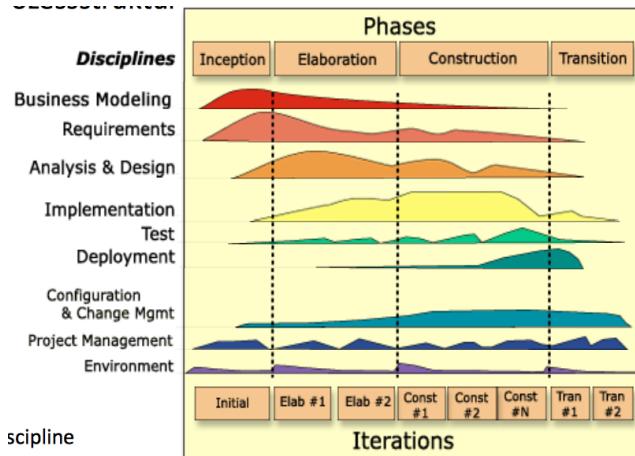
## Elemente des Software Engineering

### Prozesse/Projektmanagement

Für die Zusammenarbeit, Koordination und Planung.



Ein Prozess beschreibt, wer was wann und wie tut. Wer → Rollen, Wie → Activities gruppiert zu Disciplines, Was → Work Products und Wann → Workflows (Reihenfolge). Der Prozess ist der Rahmen der Softwareentwicklung.



Die Softwareentwicklung sollte zeitlich sowie inhaltlich gegliedert werden. Das ergibt eine zweidimensionale Prozessstruktur. Beispiele:

- RUP (Rational Unified Process)
- Vereinfacht OpenUp
- Prozesse auf Projekt anpassen
  - o D.h. Tailoring
  - o Development Case
- Agile Prozesse (z.B. Scrum)

Eine Disziplin (discipline) ist eine Menge von zusammengehöriger Aktivitäten. Die Reihenfolge der Aktivitäten ist der Workflow.

### Prinzipien und Pattern (Architektur und Design)

Prinzip Allgemeingültiger Grundsatz des Handelns

Als Beispiel das Information Hiding/Kapselung. Dabei werden die Details im eines Moduls oder einer Klasse versteckt und von aussen her ist eine einfache, abstrakte Schnittstelle sichtbar.

Prinzipien bilden die Basis des Software-Engineerings. Laram: GRASP-Patterns oder „Prinzipien guten objektorientierten Designs“ von R. C. Matrin.

### Modelle

Modelle sind Abstraktionen der realen Welt. Sie zeigen wesentliches, lassen aber unwesentliches weg.

Die Komplexität ist nur mit Modellen beherrschbar. Beispiele:

- Domain-Modell → abstrahiert den Problembereich
- Design-Modell → abstrakte Darstellung der Lösung

### Notationen

Eine Notation ist eine mehr oder weniger formale Sprache (textueller oder graphischer Art). Die Notationen werden zur Darstellung der Resultate der Softwareentwicklung verwendet. Beispiele:

- UML = Unified Modeling Language  
Standardisierte Notation für objektorientierte Softwareentwicklung
- Java = Notation für Software, die übersetzbare und ausführbar ist.

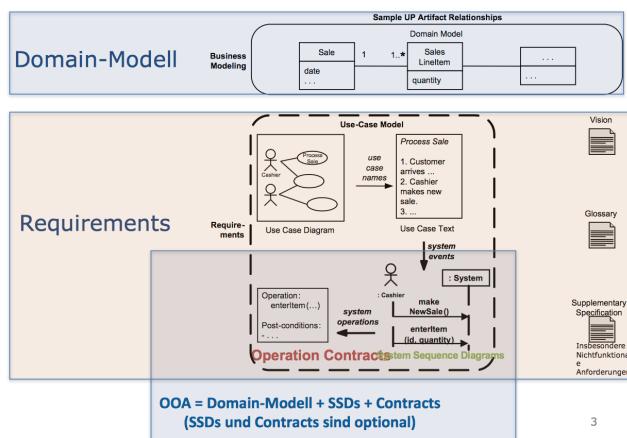
Die Notationen haben eine Syntax (Form) und eine Semantik (Bedeutung).

Sind rechnerunterstützende Mittel zur Darstellung Transformation von Notationen. Unterstützung von Activities, Workflows und automatischer Transformation von Work Products. Beispiele:

- C++ Compiler
  - o Transformiert C++ Sourcecode in Maschinencode
- Astah/StarUML
  - o Darstellung Analyse- und Design Modelle
  - o Z.T. Transformation von Designmodellen in Sourcecode
- Papier und Bleistift

## Objektorientierte Domain-Analyse

### Was ist OOA? – Einführung



### Analyse 1 – Anforderungsanalyse

**Ziel** Festlegen der Anforderungen

**Was ist zu realisieren?** Aus Black-box-Sicht (nicht wie)

**Wer** Auftraggeber oder Software-Entwicklungsorganisation in enger Zusammenarbeit mit dem Auftraggeber

### Resultate (work products)

- Vision
- Use Cases / User Stories
  - o Use Case Diagramm
  - o Textuelle Beschreibung der Use Cases
- Supplementary Specification
- Glossary

### Resultate SE1

Anforderungsspezifikation (Software Requirements Specification SRS)

## Analyse 2 – Domain-Analyse (OOA)

### Ziel

Methodische Analyse des Problembereiches

Was aus Black-Box- und White-Box-Sicht. Dabei handelt es sich um eine objektorientierte Domain-Analyse (Domänenanalyse) → Object-oriented Analysis (OOA).

Resultate (work products)

- Objektorientiertes Modell des Problembereichs/Systems
  - o **Statische Sicht:** Domainmodell (UML: Klassendiagramm)
  - o **Dynamische Sicht:** Use Case Szenarien (UML: Interaktionsdiagramm)

Dabei gibt es zwei verschiedene Varianten. „Nach Larman“ oder „klassisch“.

### OOA – „nach Larman“

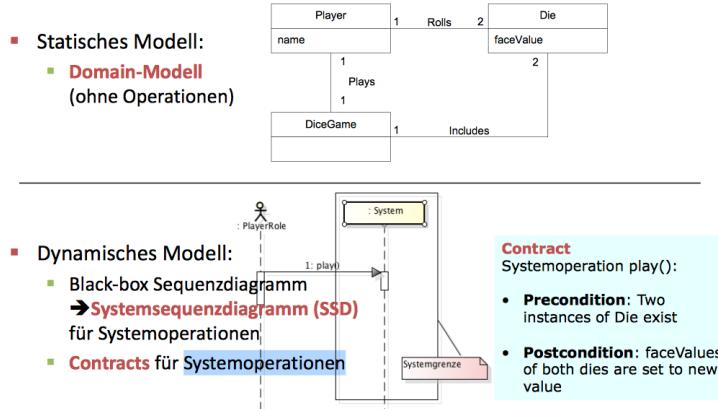
#### Statisches Modell

Gibt es ein Domain-Modell als Klassendiagramm. Dabei werden konzeptionelle Klasse mit Attributten ohne Operationen notiert. Im Klassendiagramm gibt es Beziehungen (Assoziationen) zwischen konzeptionellen Klassen.

#### Dynamisches Modell

Black-Box Interaktionsdiagramme für Use Case Szenarien bzw. Systemoperationen. Contracts für Systemoperationen. Eventuell Zustandsdiagramm für System oder einzelne konzeptionelle Klassen sowie Aktivitätsdiagramme für einzelne Use Cases.

### Beispiel



## Statisches Modell

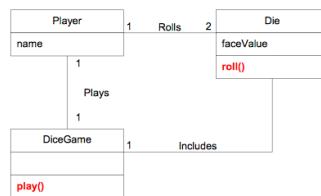
Domain-Modell = Klasendiagramm. Konzeptionelle Klassen (conceptual classes) mit Attributen und Operationen. Klassendiagramme mit Beziehungen (Assziationen) zwischen konzeptionellen Klassen.

## Dynamisches Modell

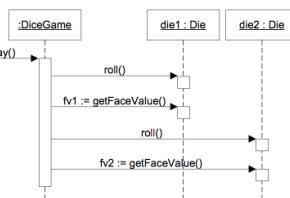
White-Box Interaktionsdiagramme für Use Case Szenarien oder Systemoperationen (elementarer Teil von UC). Eventuell: Zustandsdiagramme für System oder einzelne konzeptionelle Klassen

## Beispiel

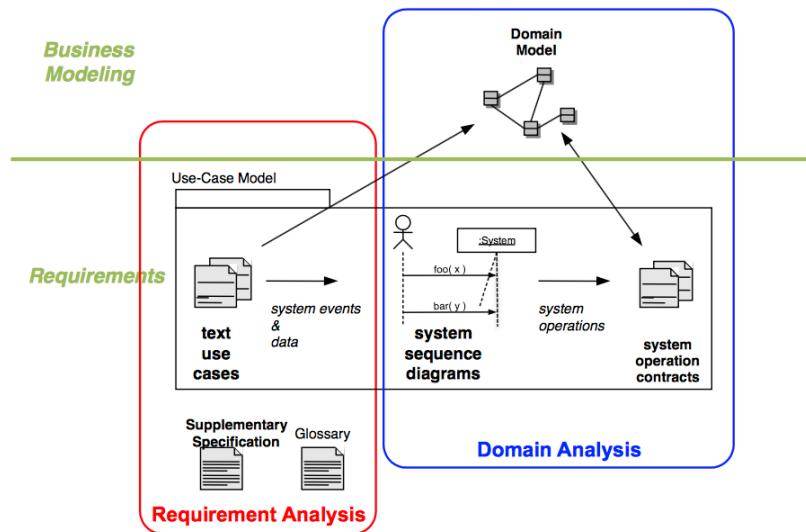
- Statisches Modell:
  - Domain-Modell (mit Operationen)



- Dynamisches Modell:
  - White-Box Interaktionsdiagramm für Use Case bzw. Systemoperation play()



## RUP Disciplines und OOA



## Statisches Domain-Modell

- Zerlegt Problem in verständliche Teile → Konzeptionelle Klassen, Konzepte
- Klärt Terminologie → visuelles Glossar
- Kommunikation zwischen verschiedenen Rollen in Entwicklung
- Basis für Entwurf
  - o Objektorientierter Entwurf bildet Konzeptionelle Klassen auf Software-Klassen ab
  - o Domain-Modell (oder Subset, der in Datenbank kommt) wird zu konzeptionellem DB-Modell

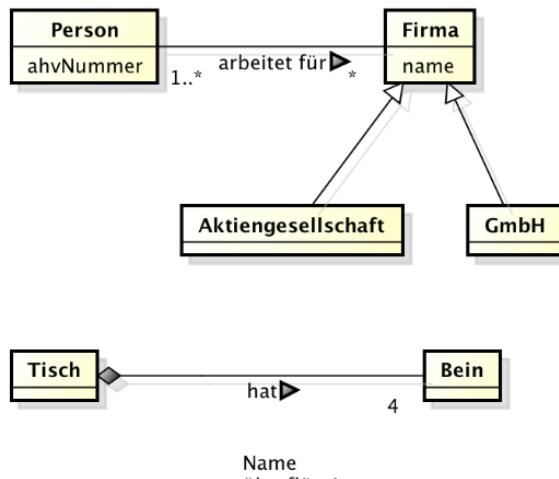
## Dynamisches Modell (Systemsequenzdiagramme, Contracts)

- Bietet systematisches Cross-Check zwischen Use Cases und Domain-Modell
- In der Praxis kaum je systematisch ausgeführt
- Aber wichtig, dass Cross-Check informal gemacht wird !

## UML – Elemente für Domainmodell

UML-Klassendiagramm mit wenigen UML-Elementen:

- **Klassen** für "Konzepte" der realen Welt
- **Attribute** für Informationen die zu Klasse gehören, evtl. mit **Typen** versehen
- **Assoziationen** mit **Multiplizitäten**, **Name** wenn nötig
  - Eventuell Ganzes-Teile Assoziationen: **Komposition**
- **Vererbung** für Generalisierung/Spezialisierung



# Domain-Modellierung

## Vorgehensvorschlag für die Erstellung eines Domain-Modells

1. 1 Mal grob durchlesen: Stichwörter notieren
2. 2-3 Mal genauer durchlesen: Klassen, Attribute & Assoziationen identifizieren
3. Domain-Modell verfeinern: Falsche oder redundante Assoziationen löschen
4. Überprüfen, ob alle notwendigen Punkte erfüllt sind.

## Resultate der Domain-Modellierung

1. Domain-Modell: Richtig & Verständlich
2. Eventuell:
  - a. Fragenkatalog und Annahmen bei Unklarheiten (Zwischenprodukt)
  - b. Glossar oder Begriffserklärung falls nötig

## Überblick

### Bis jetzt:

- Jeder kennt alle Elemente eines Domain-Modells auswendig
- Jeder hat mindestens 3 Domain-Modelle selber erstellt
- Probleme erkannt
  - o Elemente sind einfach. Anwendung braucht aber Übung
  - o Mehrere Varianten möglich. Welche Variante ist besser

In diesem Kapitel geht es darum die Stolpersteine zu identifizieren und soll zur Diskussion von Richtlinien und Best-Practices führen.

## Domain-Modell und Design-Modell

Ein **Design-Modell** zeigt „gespeicherte“ Entitäten und ihre Zusammenhänge für eine Realisation. Das Datenmodell als ER-Diagramm und das Klassenmodell als UML Klassendiagramm.

Das **Domain-Modell** zeigt wesentliche Konzeptionelle Klassen und ihre Zusammenhänge. Es beinhaltet auch Dinge, die nicht persistiert werden.

Das Domain-Modell kann eine „Inspiration“ für das Design-Modell sein!

## Identifizierung von konzeptionellen Klassen



## Strategien zum Finden von Konzepten

**Strategie 1** Von existierenden Modellen ausgehen.

**Strategie 2** Checkliste mit Kategorien von Konzeptionellen Klassen

**Strategie 3** Substantive suchen in textuelle Beschreibung („fully dressed UCs“). Aber Achtung vor Synonymen (z.B. Angestellter und Kassier), Heteronyme (z.B. Kurs für Kurstyp und Kursdruchführung) sowie Umschreibungen (Attribute, unnötige Details).

Die Strategien liefern Kandidatenliste für die Konzepte, welche es dann zu bereinigen oder zu ergänzen gilt.

## Checkliste mit Kategorien (siehe Larman S.140/141)

Business transactions	Description of things
Transaction line items	Catalogs
Product or service	Container (physical or information)
Where is transaction recorded	Thing in Container
Roles of people or organizations	Other collaborating systems
Place of transaction, service	Contracts
Events	Financial instruments
Physical objects	Schedules, manuals

### Tipps zum Bereinigen der Konzepte

- Suchen Sie die Namen aus dem Problembereich
- Konzepte beschreiben
- Implementationstechnische „Konzepte“ entfernen (kein Datenbank-Denken)
- Abgeleitete Konzepte entfernen
  - o Etwas, dass aus anderen Konzepten berechnet werden kann, zum Beispiel eine Receipt

### Konzept oder Attribut

Grundsätzlich, wenn es als Text oder Zahl ausgedrückt werden kann, soll es als Attribut abgebildet werden. Steckt mehr dahinter, soll ein Konzept/Klasse gewählt werden. In Zweifelsfalle ist ein Konzept zu wählen. Bei diesem Beispiel wäre eher die zweite Variante vorzuziehen.



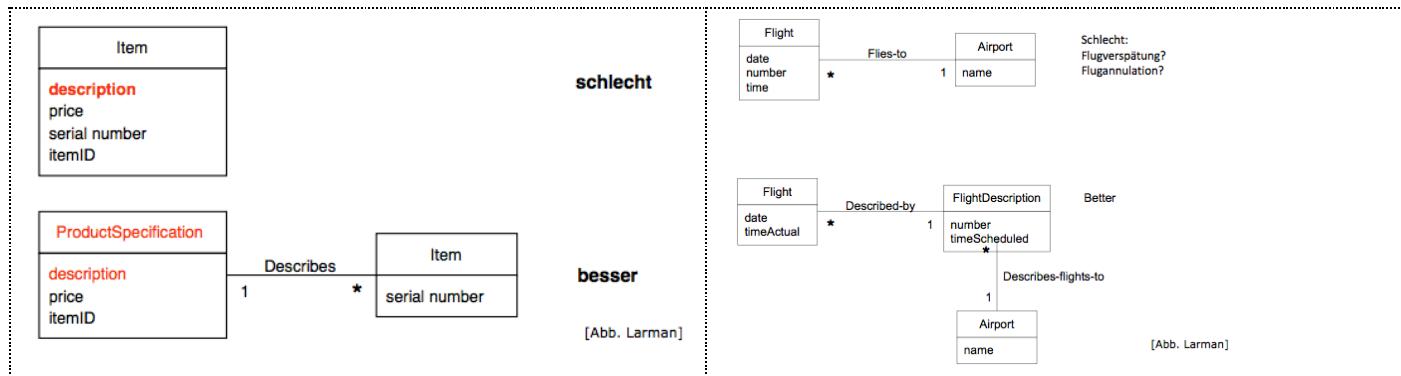
### Modellierung von technischen Systemen

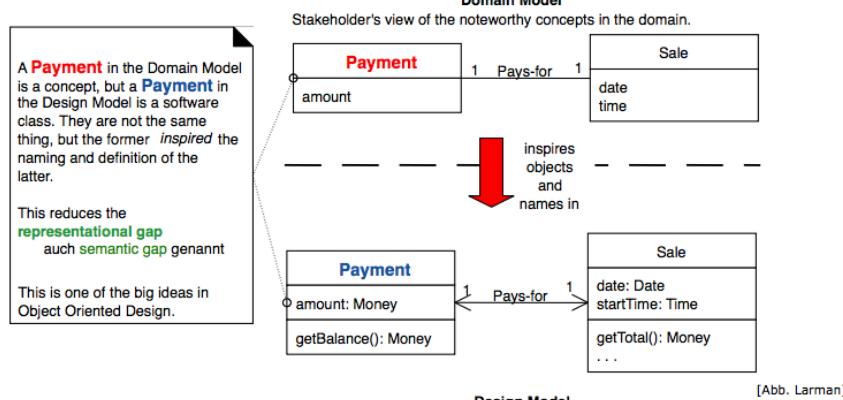
Solche Systeme sind anspruchsvoll zu modellieren, da Sie sehr abstrakt sind. Oft gibt es fehlende Entsprechungen in der realen Welt und die Begriffe sind meist implementationsspezifisch.

Als Beispiel ein Telekommunikationssystem mit den Konzepten Message, Connection, Dialog, Route und Protocol.

### Beschreibungen als Konzepte

Oft sind weitere Konzepte nötig, die andere Konzepte beschreiben. Ein Löschen aller gleichartigen Objekte würde zu einem Informationsverlust führen und zudem können mit weiteren Konzepten redundante Informationen vermieden werden.

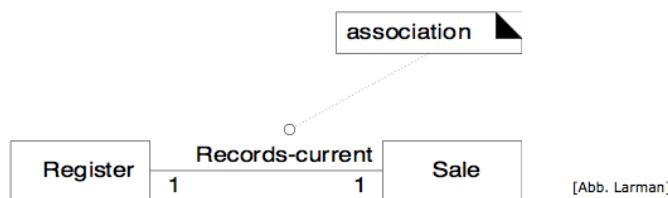




Therefore, the **representational gap** between how stakeholders conceive the domain, and its representation in software, has been **lowered**.

## Hinzufügen von Assoziationen

### Definition Assoziation



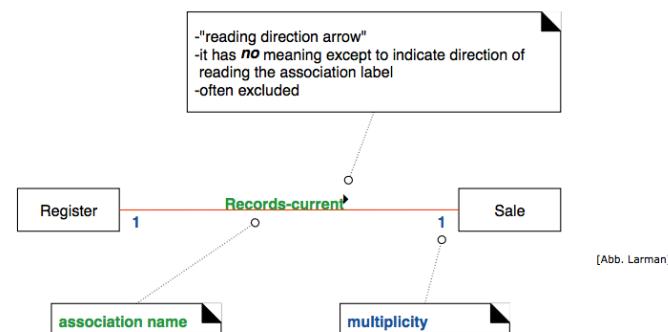
Gemäss Larman: „Eine Assoziation ist eine Beziehung zwischen Konzepten, welche eine bedeutungsvolle und interessierende Verbindung (zwischen ihren Instanzen) darstellen.“

Gemäss UML: „A structural relationship (between classes) that describes a set of links, in which a link is a connection among objects.“

Assoziationen sind grundsätzlich immer bidirektional.

### UML für Assoziation im Domain-Modell

In einem Domain-Modell sind folgende Modellierungselemente wesentlich:



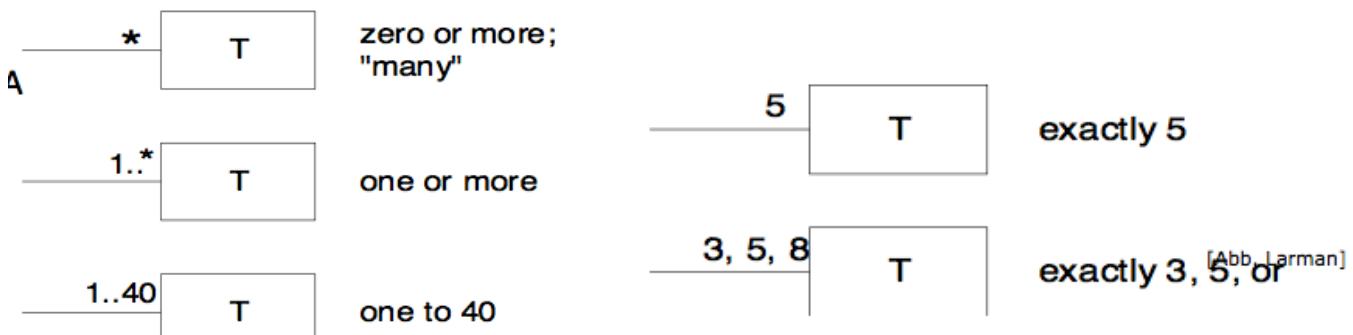
<b>A is a physical part of B</b>	Drawer – POST Wing - Airplane	<b>A is a logical part of B</b>	SalesLineItem – Sale Flight – FlightRoute
<b>A is physically contained in/on B</b>	Post – Store, Item – Shelf, Passenger – Airplane	<b>A is logically contained in B</b>	ItemDescription – Catalog, Flight – FlightSchedule
<b>A is a description for B</b>	ItemDescription – Item FlightDescription - Flight	<b>A is a line item of a transaction or report B</b>	SalesLineItem – Sale MaintenanceJob - MaintenanceLog
<b>A is known/logged/recorded/reported/captured d in B</b>	Sale - POST Reservation – FlightManifest		

### Guidelines zum Modellieren von Assoziationen

- Konzentration auf „Need-to-know“-Assoziationen
- Assoziationen müssen zum Verständnis des Modelles beitragen
- Redundante und abgeleitete Assoziationen sollen weggelassen werden
- Konzepte sind wichtiger als Assoziationen
- Zwischen zwei Konzepten sind auch mehrere Assoziationen möglich

### Multiplizitäten

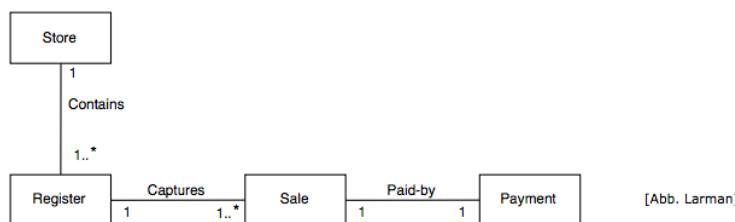
Die Multiplizität definiert wie viele Instanzen des Typs A mit einer Instanz des Typs B verbunden sein können. Aber Achtung: über welchen Zeitraum hängt vom Kontext ab. z.B. Auto-Besitzer.



### Namensgebung von Assoziationen

Typname → Satz-mit-Verb → Typname

Der Satz mit Verb soll mit einem Grossbuchstaben beginnen. Zwischen den Wörtern soll ein Bindestrich hinzugefügt werden, dass es einen Namen gibt. Am besten soll noch eine Leserichtung angegeben werden.

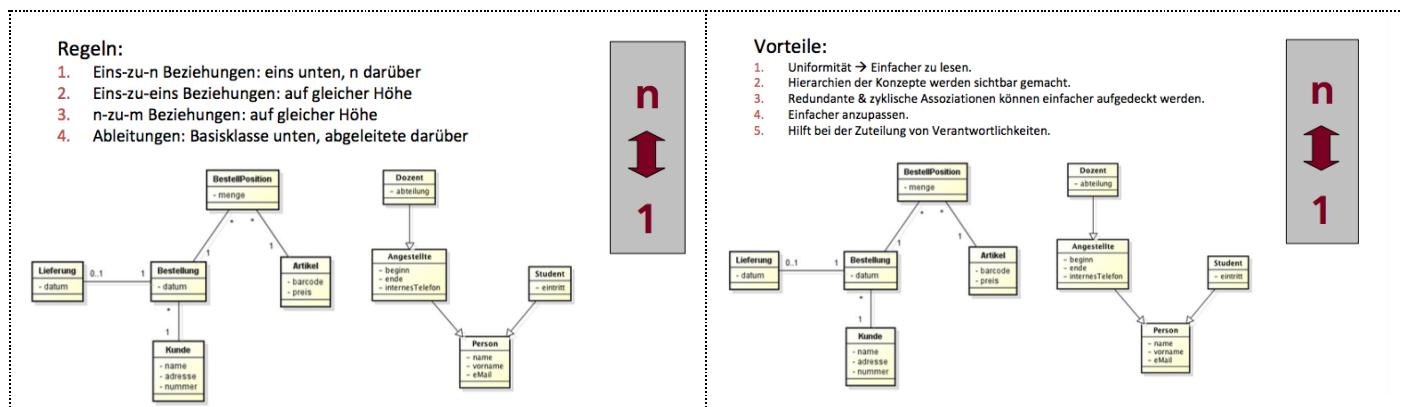


- Nur semantische Beziehung Konzepte der realen Welt
- Bidirektional

### Assoziation in Design, Implementation (Design-Modell)

- Navigationspfade von einem SW-Objekt zu einem anderen
- Meist unidirektional

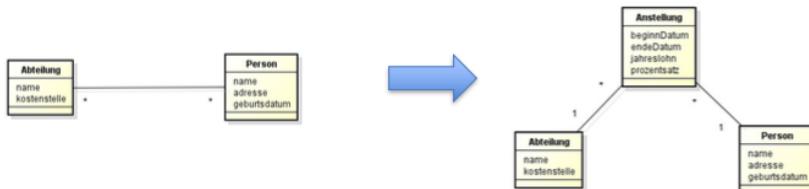
Nicht jede Assoziation in Analyse (Domain) entspricht genau einer Assoziation in Entwurf/Implementation (Design).



### Ordnen von Klassen im Domain-Modell gemäss Multiplizitäten

#### Achtung bei n-zu-m Assoziationen

Grundsätzlich sind diese Assoziationen in einem Domain-Modell zulässig. Sehr oft sind sie aber nicht richtig, da die Assoziation auch eigene Attribute hat und deshalb mit einer Zwischenklasse aufgelöst werden muss.



## Hinzufügen von Attributen

### Definition Attribut

Ein Attribut ist ein logisches Datenelement eines Objektes.

Im Domain-Modell tragen die Attribute zur Erfüllung der Anforderungen bei. Die Attributwerte müssen erinnert/gespeichert werden. Der Typ des Attributs ist optional.

### Gültige Attribut-Typen im Domain-Modell

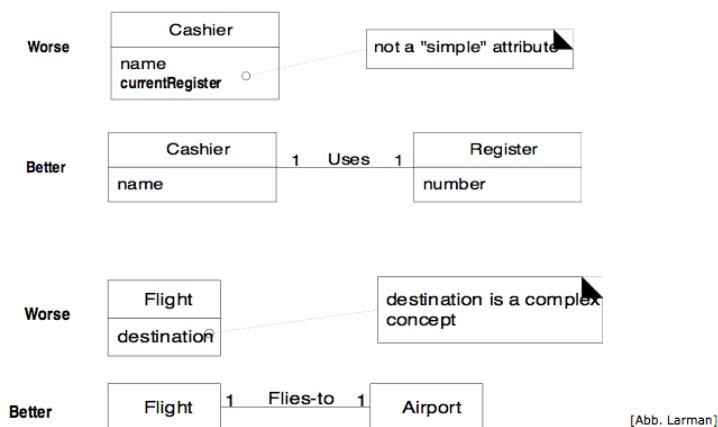
#### Einfache Datentypen

- d.h. keine weitere Struktur
- Beispiele: Boolean, Date, Number, String (Text), Time
- Achtung mit Strings → oft Konzeptionelle Klasse, Assoziationen
- Identität der Werte (==) nicht wesentlich,
- nur Wertevergleich (equals) wesentlich
- Beispiele: Telefonnummer, AHV-Nummer, Adresse (meistens)

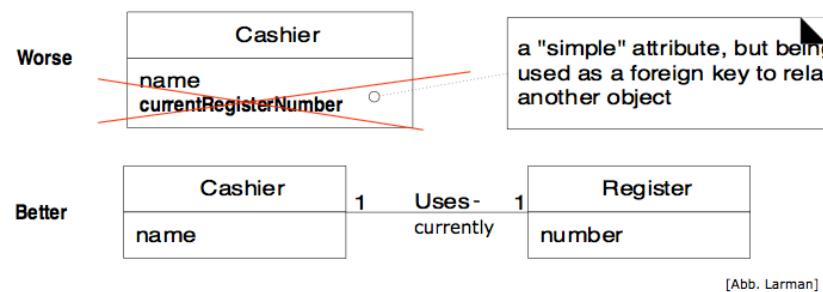
#### Achtung: Keine Design-Attribute

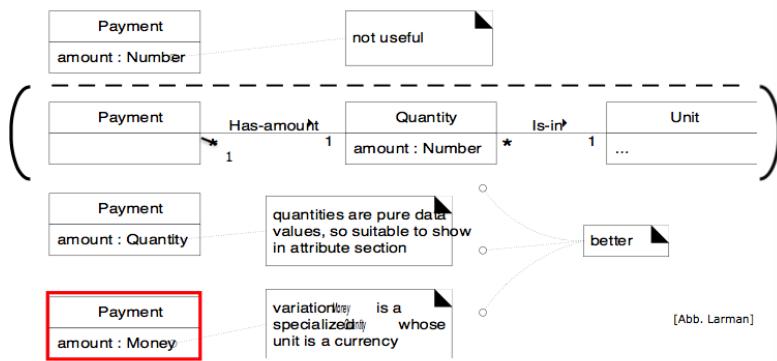
- keine Attribute, die Assoziation implementieren
- Keine Fremdschlüssel

### Keine Attribute an Stelle einer Konzeptionelle Klassen



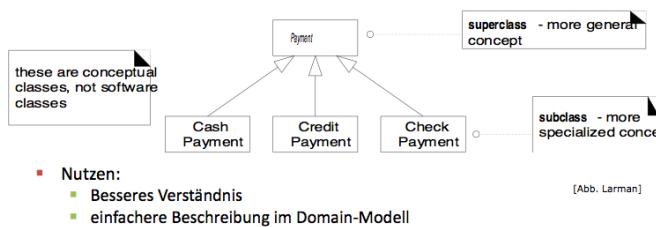
### Im Domain-Modell: keine Attribute als Fremdschlüssel!





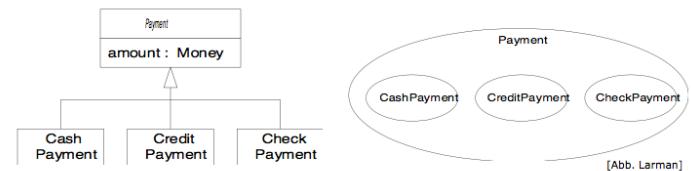
## Generalisierung-Spezialisierung

- Generalisierung:
  - Gemeinsamkeiten in Konzeptionellen Klassen  
→ Definition Supertyp



- Alle Elemente eines Subtyps gehören zur Menge der Elemente des Supertyps:

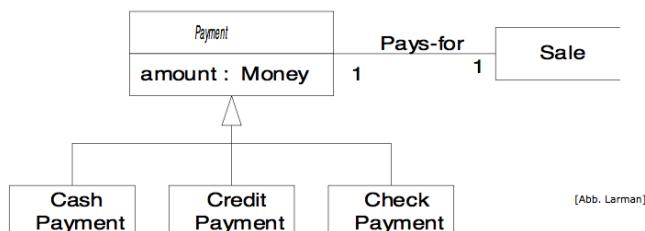
- Ein Subtyp-Objekt ist ein Supertyp-Objekt  
→ „is a“-Regel



- Nutzen:
  - Besseres Verständnis
  - einfachere Beschreibung im Domain-Modell

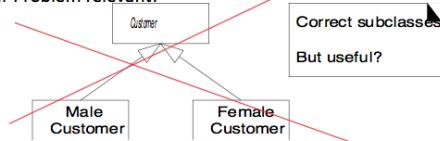
## 100%-Regel

- 100% der Supertyp-Definition gilt für Subtyp
- Gilt für alle Attribute und Assoziationen



## Wann definiert man Subtypen?

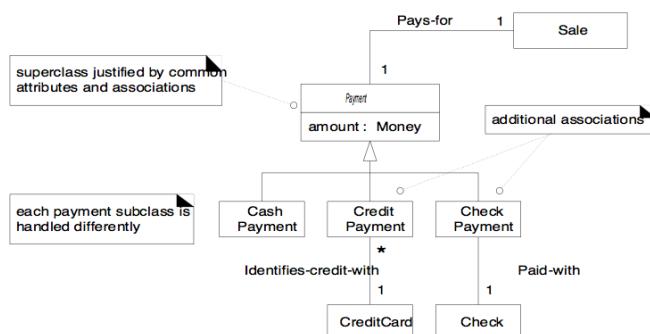
- Nur wenn für Problem relevant!



Bilde Subtyp, wenn

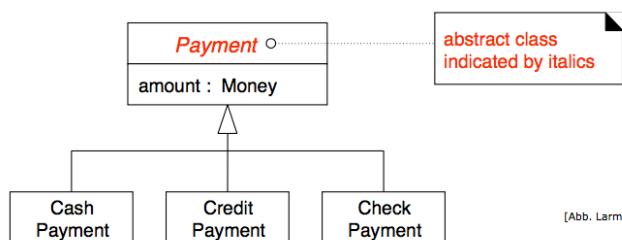
- zusätzliche relevante Attribute
- zusätzliche relevante Assoziationen
- anderes Verhalten
- Subtyp in anderer Weise verwendet

## Beispiel: Typenhierarchien im POS Domain-Modell

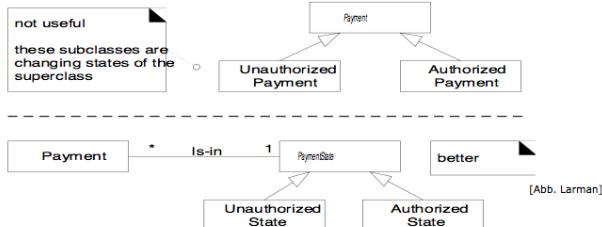


## Abstrakte Typen

- Jede Instanz vom Typ T ist auch Instanz eines Subtyps von T → T ist ein abstrakter Typ



- Modelliere Zustände eines Konzeptes nicht durch Subtypen!
- Zustandshierarchie einführen und mit Konzept assoziieren (→ State Pattern, aber in Domain-Modell übertrieben):

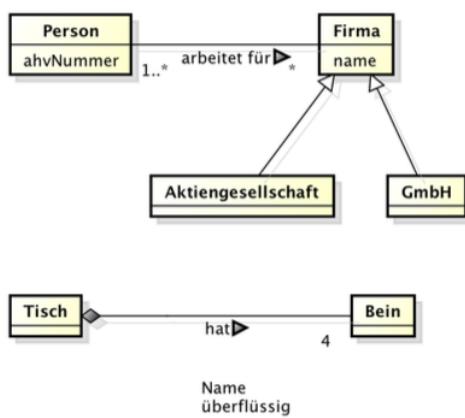


Besser: Separates Zustandsdiagramm für Konzept

## Klassenhierarchie und Vererbung

- **Domain-Modell:** Generalisierung/Spezialisierung von Konzepten (Konzeptionellen Klassen)
- **Im Design:** Typenhierarchien der Konzepte → Vererbungshierarchien von Software-Klassen

## Zusammenfassung Domain-Modellierung



UML-Klassendiagramm mit wenigen UML-Elementen:

**Klassen** für „Konzepte“ der realen Welt

**Attribute** für Informationen die zur Klasse gehören, evtl. mit Typen versehen

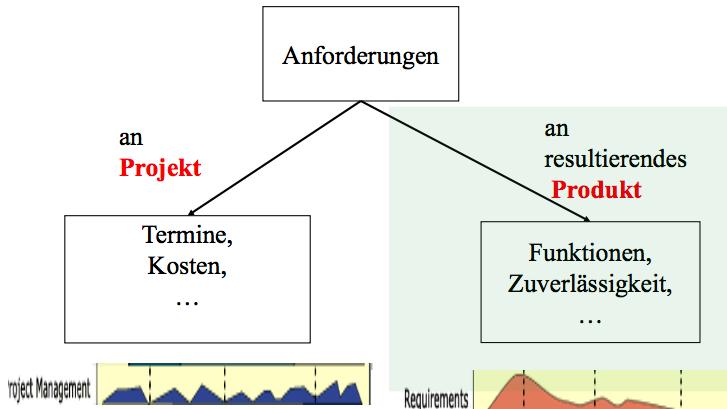
**Assoziationen** mit Multiplizitäten, Name wenn nötig (Eventuell Ganzes-Teile Assoziationen: Komposition)

**Vererbung** für Generalisierung/Spezialisierung

# Requirements – Einführung

Larman Übersicht, Material aus Kapitel 4,5 und 6

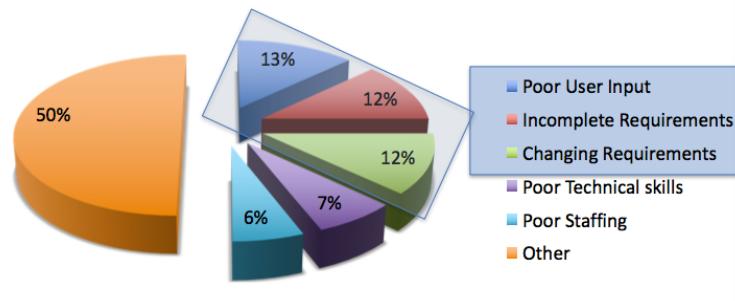
## Anforderungen und Disciplines



## Requirements: Discipline und Work Products

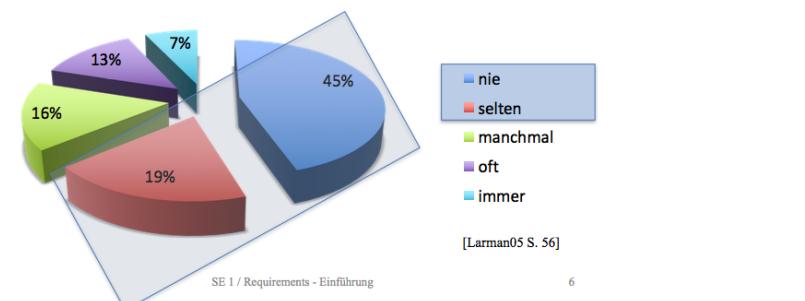
<b>Discipline</b>	Requirement Specification = Methodisches Vorgehen und Ablauf, um Fähigkeiten und Bedingungen festzulegen, die Software erfüllen muss.
<b>Work Product</b>	<p>Software Requirement Specification (SRS) = Fähigkeiten/Bedingungen die Software erfüllen muss.</p> <p>Ist das Resultat der Discipline Requirement Specification</p>

## Requirements: 30% aller Probleme in der Software-Entwicklung



## Requirements und Wasserfallmodell

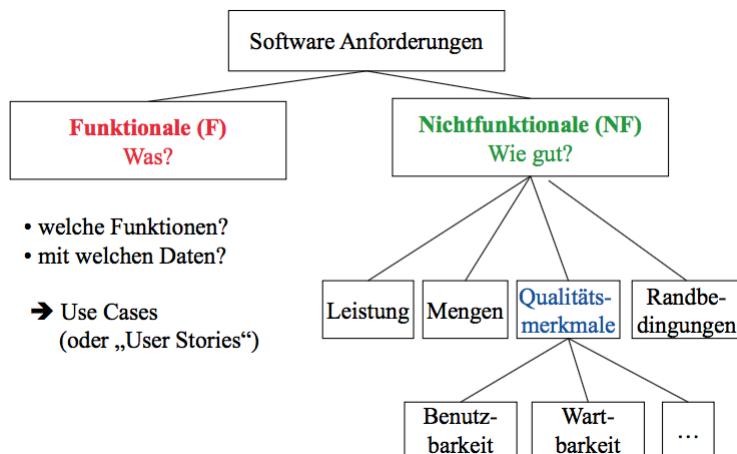
Requirements am Anfang umfassend zu spezifizieren ist sehr problematisch. Die Studien von Johnson zeigen die Verwendung von Funktionalitäten, die gemäss Wasserfallmodell spezifiziert wurden und ihre Nutzung.



6

## Requirements Management

Anforderungen ändern sich. Es sollten nicht alle Anforderungen am Anfang im Detail erfassen werden (iteratives Vorgehen). Sie sollen aber systematisch erfasst und verwaltet werden.



## Wie (funktionale) Anforderungen beschrieben werden

Wie (funktionale) Anforderungen beschrieben werden: User Stories und Use Cases		Wie (funktionale) Anforderungen beschrieben werden: Use Cases und Personas, Szenarien	
<p><b>User Stories</b></p> <ul style="list-style-type: none"> <li>Kurze Geschichte aus Benutzersicht           <ul style="list-style-type: none"> <li>Nur ein Szenario</li> </ul> </li> <li>Informal</li> <li>Ohne spezielle Systematik</li> <li>Gedankenstütze für Planung und Kommunikation</li> <li>Insbesondere bei <i>Agiler Softwareentwicklung</i> verwendet</li> </ul>	<p><b>Use Cases</b></p> <ul style="list-style-type: none"> <li>Geschichte aus Benutzersicht auf Ebene elementarer Geschäftsprozesse           <ul style="list-style-type: none"> <li>Oft mehrere Szenarien</li> </ul> </li> <li>Verschiedene Beschreibungsebenen           <ul style="list-style-type: none"> <li>Von „brief“ bis „fully dressed“</li> <li>Essential Style (losgelöst von Realisierung)</li> </ul> </li> <li>Teil einer systematischen Anforderungsspezifikation</li> </ul>	<p><b>Personas, Szenarien</b></p> <ul style="list-style-type: none"> <li>Personas sind fiktive Personen, die typische Benutzer beschreiben</li> <li>Szenarien sind informale, anschauliche (<i>d.h. nicht essential style</i>) Geschichten, wie eine Persona das System benutzen würde</li> <li>Insbesondere bei <i>User Centered Design</i> verwendet (siehe Ulnt1)</li> </ul>	<p><b>Use Cases</b></p> <ul style="list-style-type: none"> <li>Verwenden Actors (sind Benutzerrollen)</li> <li>Geschichte aus Benutzersicht auf Ebene elementarer Geschäftsprozesse           <ul style="list-style-type: none"> <li>Oft mehrere Szenarien</li> </ul> </li> <li>Verschiedene Beschreibungsebenen           <ul style="list-style-type: none"> <li>Von „brief“ bis „fully dressed“</li> <li>Essential Style (losgelöst von Realisierung)</li> </ul> </li> <li>Teil einer systematischen Anforderungsspezifikation</li> </ul>

# Requirements – Use Cases

## Lernziel

Jeder Teilnehmer kann die funktionalen Anforderungen eines Systems als Use Case in den Formen „brief“, „casual“ und „fully dressed“ beschrieben.

- Textuelle Ablaufbeschreibung mit Ziel und Zweck
- Kohärente Geschichte
- Benutzersicht "Actor tut das, dann macht das System das..."
- verschiedene Formate:  
mehr oder weniger formal und lang

- Geschichte:
  - 1986 populär gemacht von Ivar Jacobson
  - ab 1992 weiterentwickelt von Alistair Cockburn
  - Standard Vorgehen, Teil des Unified Processes 1997
  - UML Notation für Use Case Diagramme 1997

Use Case Diagramm ist kein  
Use Case, es bildet nur eine  
Übersicht ab.

## Use Case Beispiel

Wickle Einkauf ab: Ein Kunde kommt an eine Kasse mit Waren, die er kaufen möchte. Der Kassier verwendet das POS-System, um jeden eingekauften Artikel zu erfassen. Das System zeigt den Totalbetrag und Details der erfassten Artikel. Der Kunde gibt die Zahlungsinformationen ein, die vom System validiert und registriert wird. Das System führt nun das Inventar des Geschäfts nach und liefert eine Quittung für den Kunden. Dieser verlässt den Laden mit den eingekauften Waren.

## Wieso Use Cases?

- |   |   |
|---|---|
| <b>Use Cases</b>                                  | <b>Funktionslisten</b>                      |
| nicht lose Sätze                                  |   |
| ■ Stellen Anforderungen in Anwendungszusammenhang | ■ Ohne Anwendungszusammenhang               |
| ■ Gute Übersicht                                  |   |
| ■ Gut auf Vollständigkeit zu prüfen               |   |
| ■ Gute Ausgangslage für Realisierung              | Vergleich ob es so entwickelt wurde,        |
| ■ Gute Grundlage für Systemtest                   | Als Grundlage für das Suchen von Testcases. |
| ■ „Use Cases are Test Cases“                      |   |
|   | Ivar Jacobson                               |

## Formale Definition

"A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor.."

## Übersetzt:

- „Menge von Use-Case Instanzen“
- Jede Use-Case Instanz ist eine Folge von Aktionen, welche das System ausführt = Use Case Scenario / Flow und ein beobachtbares Resultat liefert.
- Das bestimmtem Aktor einen Nutzen bringt

Aktoren sind immer ausserhalb des Systems. Primary Actor lösen einen Use Case aus.



Person oder System, ausserhalb des zu entwickelnden Systems,  
Stackholders (Anspruchsgruppen, Interessegruppen)

## Use Case Instanz

Oder auch Szenario genannt, eine geschlossene Sequenz/Pfad durch die Geschichte (Erfolgs-Szenario/Basis-Szenario oder Alternativ-Szenarien)

## Format von Use Cases

### Brief (kurz)

Zusammenfassung in einem Abschnitt, beschreibt nur erfolgreiches Szenario

### Casual (ungezwungen)

Mehrere Abschnitte, informale Beschreibung mehrere Szenarien

### Fully dressed

Alle Szenarien im Detail beschrieben, mir Raster für ergänzende Abschnitte, wie Vorbedingungen und Nachbedingungen

## Beispiel „fully dressed“

### Use Case UC1: Process Sale

#### Primary Actor: Cashier

#### Stakeholders and Interests:

- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
  - Salesperson: Wants sales commissions updated.
  - Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.
  - Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
  - Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
  - Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.
- Was passieren soll, damit Use Case sein darf.**
- Preconditions:** Cashier is identified and authenticated.
- Success Guarantee (Postconditions):** Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded. **Was passiert, wenn der Use Case fertig ist.**

#### Main Success Scenario (or Basic Flow): Langwellig aber detailliert.

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.
- Cashier repeats steps 3-4 until indicates done.**

5. System presents total with taxes calculated.
  6. Cashier tells Customer the total, and asks for payment.
  7. Customer pays and System handles payment.
  8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
  9. System presents receipt.
  10. Customer leaves with receipt and goods (if any).
- und so weiter**

#### Extensions (or Alternative Flows): Was kann sonst passieren.

##### \* Stern kann immer auftreten.

- \* At any time, System fails.
- To support recovery and continue processing, and to maintain sensitive state, any event can be triggered from any point in the scenario:
  1. Cashier restarts System, logs in, and requests recovery of prior state.
  2. System reconstructs prior state.
  - 2a. System detects anomalies preventing recovery:
    1. System signals error to the Cashier, records the error, and enters a clean state.
    2. Cashier starts a new sale.
  - 3a. Invalid identifier:
    1. System signals error.
    2. Cashier enters item identifier for removal from sale.
    3. Cashier enters item identifier for removal from sale.
    4. System displays updated running total.
    5. Cashier cancels sale on system.
    6. Cashier suspends the sale.
  - 3b. Item identifier is available for reissue on any POS terminal:
    1. The system generated item price is not wanted (e.g., Customer complained about something and is offered a discount).
    2. Cashier enters item identifier.
    3. System presents new price.
  - 3c. System detects failure to communicate with external tax calculation system service:
    1. System generates price on the POS node, and continues.
    2. System signals error.
  - 3d. Customer says they are eligible for a discount (e.g., employee, preferred customer):
    1. Cashier signals discount request.
    2. Cashier enters discount identification.
    3. System presents discount total, based on discount rules.
  - 3e. Customer says they have credit in their account, to apply to the sale:
    1. Cashier enters account identification.
    2. Cashier enters Customer identification.
    3. System applies credit up to price=0, and requests remaining credit.
    4. Customer says they have credit but don't have enough cash:
      - 1a. Customer uses an alternate payment method.
      - 1b. Customer tells Cashier to cancel sale. Cashier cancels sale on System.

- 9a. There are product rebates:  
1. System presents the rebate forms and rebate receipts for each item with a rebate.
- 9b. Customer requests gift receipt (no prices visible):  
1. Cashier requests gift receipt and System presents it.
- Was aus Technologischer und Benutzersicht wichtig ist.**
- Special Requirements:** \* nicht funktional!
- \* Credit card number must be visible from 1 meter.
  - Credit authorization response within 30 seconds 90% of the time.
  - Somehow, we want robust recovery when access to remote services such the inventory system is lost.
  - Language internationalization on the text displayed.
  - Pluggable business rules to be insertable at steps 3 and 7.
  - ...

#### Technology and Data Variations List:

- 3a. Item identifier entered by car code reader scanner (if bar code is present) or key-in.
- 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- 7a. Credit account information entered by card reader.
- 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

#### Frequency of Occurrence:

#### Während der Entstehung....

- When creating the use case.
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader or does the cashier have to do it?

## Einspaltig oder zweispaltig?

Es spielt keine Rolle!. Einspaltig ist einfacher zu editieren und braucht weniger Platz.

## Allgemeine Informationen

UC ##: Name (Substantiv + Verb)	
Goal	Kurze Beschreibung des Ziels.
Level	Summary oder User Goal oder Subfunction.
Primary Actor	Actor, dessen Ziel erfüllt wird.
Trigger	Das Ereignis, welches den Use Case auslöst.
Stakeholders and Interests	Anspruchsgruppen und Ihre Ziele an den UC.
Preconditions	Bedingungen, die gelten müssen, bevor der UC ausgeführt wird.
Postconditions	Bedingungen, die gelten, nachdem Use Case ausgeführt wurde. Eventuell unterteilt in Success und Failure Postconditions
Abgeändert nach Larman05 und <a href="http://alistair.cockburn.us/Basic+use+case+template">http://alistair.cockburn.us/Basic+use+case+template</a>	

## Main Success Scenario

### Main Success Scenario:

1. Actor Sowieso tut ... (mit präziser Angabe der Schnittstellendaten)
2. System macht ...
3. Actor Sowieso tut ...
4. System macht ...
- Wiederhole 3 bis 4 bis ...
5. ...

Abgeändert nach Larman05 und <http://alistair.cockburn.us/Basic+use+case+template>

## Extensions

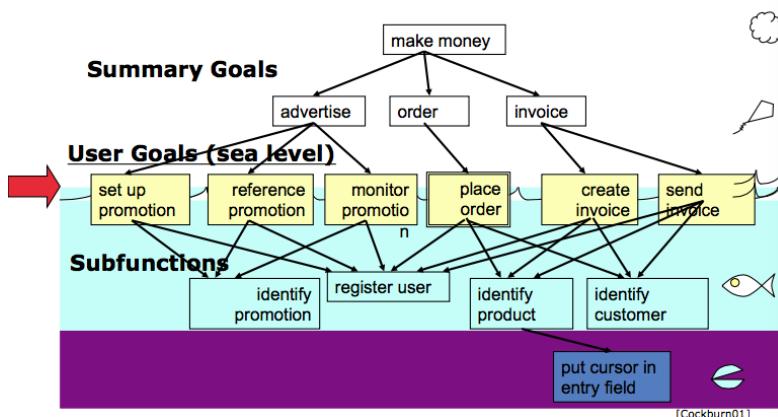
Extensions:	
2a. Wenn ...	Zur Nummerierung:  2: Bezug auf Schritt oder Schrittfolge z.B. 2-4 im Main Success Szenario.  a: erste Erweiterung zu Schritt 2, nächstes b usw.  Weiter bei Schritt 5
*a. Wenn Bedingung erfüllt	*: in beliebigem Schritt, wenn Bedingung erfüllt.
1. ...	

## Ergänzende Informationen

Special Requirements	(Nichtfunktionale Anforderungen), die spezifisch für UC sind.
Technology and Data Variation Lists	Randbedingungen für die Realisierung bezüglich Technologie und Datenformaten.
Frequency of Occurrence	Z.B. wöchentlich, mehrmals täglich, alle 200 Millisekunden usw.
Open Issues	Offene Punkte, die noch abzuklären sind.

Abgeändert nach Larman05 und <http://alistair.cockburn.us/Basic+use+case+template>

## Use Case Ebenen nach Cockburn



## Elementary Business Process = EBP

Es ist sinnvoll Use Cases auf der Ebene von EBP.

**EBP** = Elementary Business Process, Ebene der Benutzerziele

- Eine Aufgabe durch eine Person an einem Ort zu einem Zeitpunkt durchgeführt.
  - o Nicht immer alles erfüllt → Session
- Als Reaktion auf Business-Event
- Hat messbaren Nutzen für Business (Boss Test)
- Hinterlässt Daten des Systems in konsistentem Zustand

Beispiele dafür sind „Bestellungen aufgeben“ oder eine „Rechnung bezahlen“.

## Vorgehen Use Cases

- Systemgrenzen festlegen
  - o Software, HW/SW-System, Organisation (mit HW/SW-System)
- Primäre Akteure identifizieren
  - o Ihre Ziele als Systembenutzer werden durch System erfüllt
- Ziele jedes Akteuren identifizieren
  - o So abstrakt wie möglich auf Ebene EBP
- Use Cases schreiben – zuerst nur im „brief-format“
  - o Erfüllen Benutzerziele
  - o 1 Use Case pro Benutzerziel
  - o UI weglassen (essential style)
  - o Namen gemäss Ziel meist aus Substantiv und Verb

## System und Systemgrenzen

### Akteuren und Ziele

Erst primäre Akteure finden (relativ einfach) und dann „versteckte“ Akteure suchen:

- Wer macht Security oder Benutzerverwaltung?
- Systemüberwachung?
- Zeitgesteuerte Aktivitäten?
- Evaluation von Systemaktivität? Syslogs etc?

### Aktor-Ziel-Liste als Hilfe

Actor	Goal
Cashier	process sales process rentals handle returns cash in cash out ...
Manager	start up shut down ...

## Regeln für Use Cases

### Griffige Namen sind wichtig

- o Zusammenfassung des Ziels/Zwecks
- o Namen Verb + Objekt z.B. process sale

### 1 Use Case pro Benutzerziel

- o Ausnahme von CRUD (create, retrieve, update und delete) können in einem Mitglied bearbeiten zusammengefasst werden.

## Aktoren

### Primary Actors (primäre Aktoren)

- Services des Systems erfüllen Zeile der primären Aktoren
- Helfen Ziele der Benutzer zu identifizieren
- Helfen Use Cases zu finden

### Supporting Actors (unterstützende Aktoren)

- Stellen einen Service für das System zur Verfügung
- Helfen Interfaces des Systems zu klären

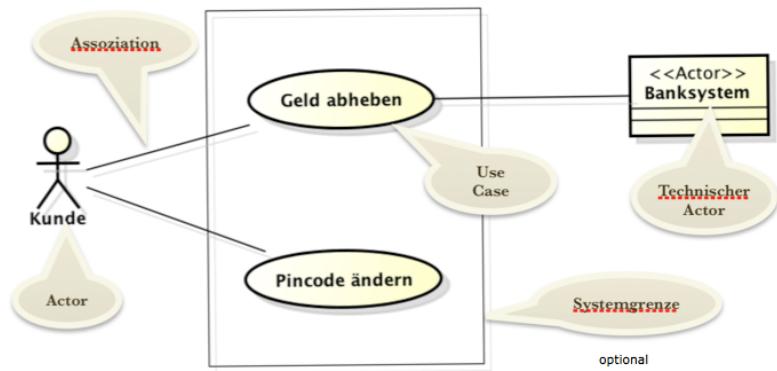
## Essential Style

Use Cases sollten im Essential Style geschrieben werden. Dieser Style ist auf Ziele und Absichten ausgerichtet und losgelöst von der Realisierung. Es ist sicher ohne GUI Beschreibung.

## Wieso?

Die Anforderungen (was) und das Design (wie) sollten getrennt sein. Das Use Cases beschrieben was, ist der Fall ja klar.

## Use Case Diagramm

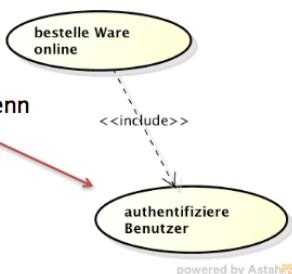


## Sinn

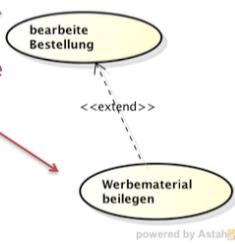
Übersicht über das Use Case Model. Es ist ein Kontextdiagramm. Es zeigt die Akteure, Systemgrenzen, Use Cases und Beziehungen zwischen den Akteuren und Use Cases. Es zeigt also den Umfang (Scope) des Systems. Es ist aber nicht zu detailliert.

## Include

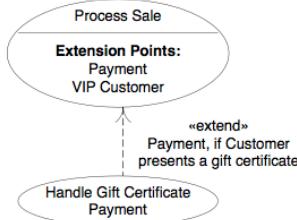
- Teil von Use Case in separaten Subfunction Use Case ausgelagert
- Teil(e) mit «include» "ausklammern", wenn
  - Teil in mehreren Use Cases auftritt
  - Use Case sehr komplex und lang
- In Use Case Beschreibung:
  - Level: Subfunction
- «include» einfachste und häufigste Beziehung zwischen Use Cases



- Erweitert (bestehenden) Basis Use Case
- Basis Use Case sollte auch ohne Extension Use Case funktionieren
- Könnte auch in Extension-Abschnitt des Basis Use Cases beschrieben werden
- Regel: Zurückhaltend verwenden
- Meist sinnvoll nur wenn Basis Use Case nicht modifiziert werden darf



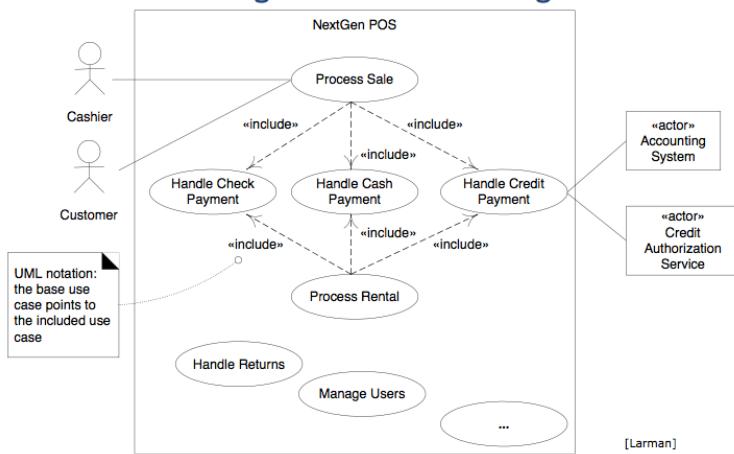
- Basis Use Case wird nicht verändert
  - nennt Extension Points
- Extension Use Case
  - nennt Basis Use Case und Extension Point
  - Trigger (Bedingung wann ausgeführt)



UML notation:  
1. The extending use case points to the base use case.  
2. The condition and extension point can be shown on the line.

[Larman]

## UC-Diagramm mit Beziehungen



## Grad der Ausarbeitung

### Inception

- 90% der Use Cases im „brief“ Format
- Eventuelle kritische Use Cases „fully dressed“

### Elaboration

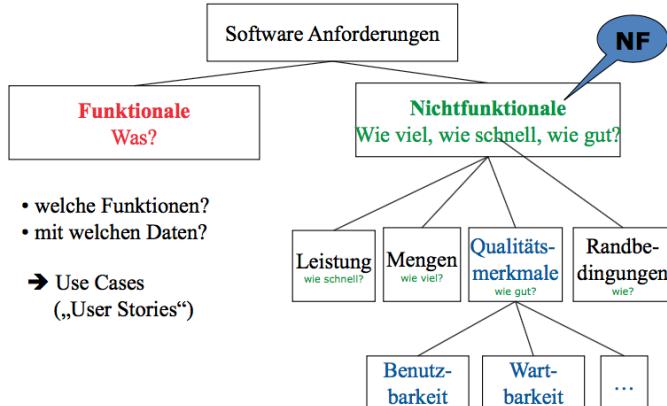
- Iterationen liefern Feedback um Anforderungen weiter zu erfassen, präzisieren, z.B. nach RUP
  - o Ende Iteration 1: 30% UCs „Fully dressed“
  - o Ende Iteration 2: 50% UCs „Fully dressed“
  - o Ende Elaboration 80% UCs „Fully dressed“

In den agilen Prozessen sind grundsätzlich alles im „Brief“ Format und beinhalteten ev. Allgemeine Informationen. Die Szenarien sollen als User Stories in Back-Log Items.

# Requirements – Nichtfunktionale Anforderungen, Software Requirements Specification

## Nichtfunktionale Anforderungen (Non functional requirements) NF

### Klassifikation von Softwareanforderungen



### Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen gehen oft vergessen. Dies ist einerseits darauf zu führen, dass wir meist nur an die Implementation denken und uns zusätzlich in die Kundensicht versetzen müssten.

Sie sind aber dennoch sehr wichtig. Sie haben einen sehr grossen Einfluss auf die Architektur sowie die Benutzbarkeit, welches beide sehr wichtige Aspekte sind. Das Thema Sicherheit sollte ihr auch noch erwähnt sein. Grundsätzlich muss man dabei aber aufpassen, da es schwierig ist sie überprüfbar zu spezifizieren.

### Leistung, Mengen & Randbedingungen

#### Leistungsanforderungen (Performance)

Darunter zählen Antwortzeiten, Durchsatzraten, .... Aber Achtung: Sie sind von der Hardwareumgebung sowie von der Systembenutzung abhängig.

#### Mengenanforderungen

Anzahl von Kundendatensätzen, Anzahl gleichzeitiger Benutzer

#### Randbedingungen (environment)

Welche Einschränkungen sind bezüglich der Realisierung zur Erfassen. z.B. Programmiersprache, Verwendung von bestimmten Datenbankservern. Man sollte aber aufpassen, dass man sich nicht unnötige Einschränkungen macht.

#### Qualitätsmerkmale

Grundsätzlich wie gut ist es.

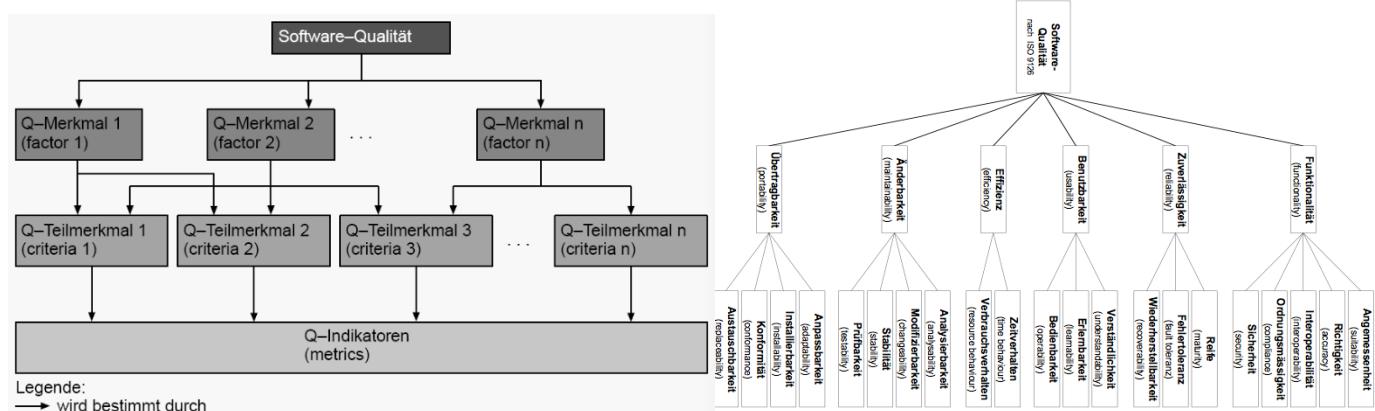
Unter Softwarequalität versteht man die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukt, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

#### Qualitätsmerkmale von Software

Eine Menge von Merkmalen eines Softwareprodukts, anhand derer seine Qualität beschrieben und beurteilt wird. Ein Software-Qualitätsmerkmal kann über mehrere Stufen in Teilmerkmale verfeinert werden.

Ein Qualitätsmodell setzt sich aus Merkmalen und Teilmerkmalen zusammen.

## Modell für Software Qualität sowie nach ISO 9126



## Funktionale Anforderungen ↔ Qualitätsmerkmal Funktionalität

Die funktionalen Anforderungen beschreiben was gemacht wird, während das Qualitätsmerkmal Funktionalität beschreibt wie gut etwas gemacht werden kann.

### Definition gemäss Larman

Diese werden in der Literatur „Other Requirements“ genannt und mit FURPS+ angekürzt.

<b>Functionality:</b>	Nur Sachen, welche über mehrere Use Cases vorhanden sind (Log, Error Handling, Authentication)
<b>Usability</b>	Text von einem Meter sichtbar, Farben in den Symbolen
<b>Reliability</b>	Kann auch ohne einen extern Server genutzt werden.
<b>Performance</b>	Einloggen dauert unter 10 Sekunden.
<b>Supportability</b>	Anpassbare Business Regeln
<b>+</b>	Lizenziierung, Hardware & Software, Packetierung

Wichtig ist dabei, dass die Formulierung konkret ist und auch getestet werden kann. Dokumentiert werden diesen in der „Supplimentary Specification“

## Software Requirements Specification

### Dokumentation von Software Anforderungen

Document	Captures	Goal
Vision	Big Ideas; Executive summary	Introduction; Basis for management decisions
Use Cases	Functional Requirements	Basis for development
Supplementary Specification	Requirements not captured by use cases (FURPS+)	
Glossary	Terms and Definitions	
Business Rules	Rules and policies that transcend the current application	

# Modelling Behaviour - Zustands- und Activity Diagramme

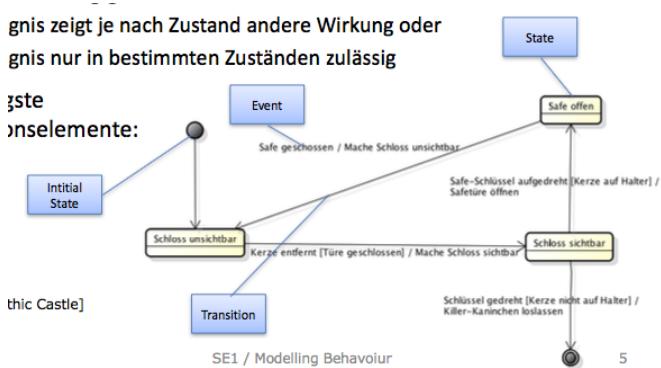
**Zustandsdiagramme** zeigen ein zustandsabhängiges Verhalten von Systemen und Objekten. Es findet Anwendung in der Analyse und im Design.

**Activitydiagramme** sind UML-konforme Flussdiagramme, aber mit zusätzlichen Parallelität. Es findet ebenfalls Verwendung in der Analyse sowie im Design. Es hilft dabei Geschäftsprozesse zu modellieren, kann bei Use Cases angewendet werden und zeigt auch Designaspekte auf.

## Zustandsdiagramme - State Machine Diagramms

Grundsätzlich sind es „Objekte“ mit einem zustandsabhängigem Verhalten. Ein Objekt ist ein ganze System, eine konzeptionelle Klasse bzw. eine Design-Klasse.

Zustandsabhängiges Verhalten heisst, dass das Ereignis je nach Zustand eine andere Wirkung zeigt oder das ein Ereignis nur in bestimmten Zuständen zulässig ist.



## Zustandsdiagramme in UML

Zustandsdiagramme gibt es in unterschiedlichen Varianten. Sie haben Unterschiede in der graphischen Darstellung, feine Unterschiede in der Semantik oder sind mehr oder weniger aufgeteilt.

Das UML basiert auf der Notation von David Harel (1987). Es werden hier nicht alle Feinheiten der UML-Notation behandelt.

## Zustandsdiagramme in Analyse

In der Analyse werden Zustandsdiagramme für das ganze System oder eventuell auch nur für einzelne Konzeptionelle Klassen verwendet.

Es gibt zwei Arten von Zustandsdiagrammen für das System:

- Use-Case-Zustandsdiagramme zeigen die Systemereignisse und Systemzustände sowie die Zustandsübergänge für einen Use Case.
- System-Zustandsdiagramme zeigen alle Systemereignisse und Systemzustände sowie Zustandsübergänge des Systems.

Ein System-Zustandsdiagramm enthält alle Use-Case Zustandsdiagramme.

Zustandsdiagramme modellieren das dynamische Verhalten. Sie geben einen Mehrwert, falls das zustandsabhängige Verhalten nicht trivial ist. In Studentenprojekten wird es tendenziell zu wenig verwendet.

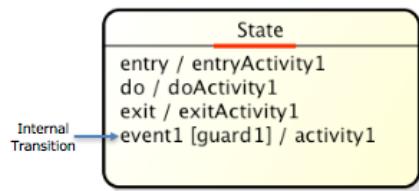


Ein Event passiert zu einem Zeitpunkt. Sie sind in der Regel ausserhalb des betrachteten Systems (bzw. Objekts). Das System wird darüber benachrichtigt und reagiert auf das Ereignis.

### Kategorien von Ereignissen

- Externe Ereignisse – Normalfall
- Interne Ereignisse
  - o Aktivität abgeschlossen
  - o Data Condition
    - Bsp. Temperatur sinkt unter Schwellenwert.
- Zeitliche Ereignisse (temporal event)

### Zustand (state)



Allgemein kann man sagen, dass es ein Zustand eines Objekt ist mit allen Attributwerten. Bei einem Zustandsdiagramm sollten nur jene Zustände unterschieden werden, welche ein unterschiedliches Verhalten aufweisen.

### In einem Zustand....

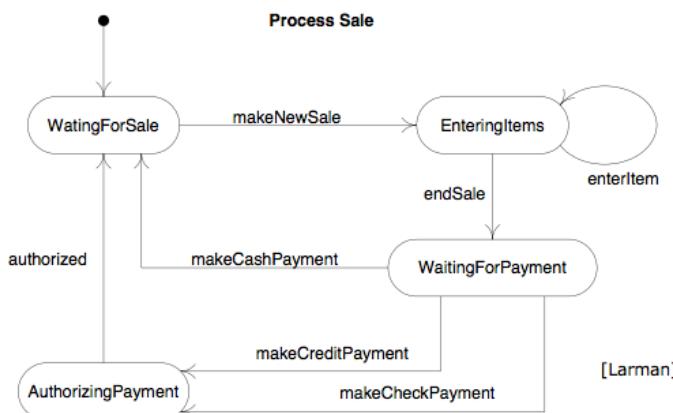
- Kann ein Objekt einfach auf ein Ereignis warten
- Oder eine Aktivität (do/activity) ausführen (Aktivitäten können durch Ereignisse unterbrochen werden)
- Beim Eintritt in den Zustand kann eine entry/activity ausgeführt werden.
- Beim Verlassen eines Zustands kann eine exit/activity ausgeführt werden
- Ein Event, der nicht zu einem Zustandsübergang führt, kann durch eine Internal Transition beschrieben werden.

### Zustandsübergang (transition)

Grundsätzlich bildet der „Übergang von einem Zustand in einen anderen“ ab. Zustandsübergänge werden durch Ereignisse ausgelöst. Bei einer Zustandsübergang können Aktivitäten (activities) ausgeführt werden. Die Zustandsübergänge erfolgen „augenblicklich“.



Aktivitäten sind für die Ausführung von Operationen zuständig. Es sind atomar und können nicht durch ein Ereignis unterbrochen werden. Übergänge auf den gleichen Zustand sind möglich.

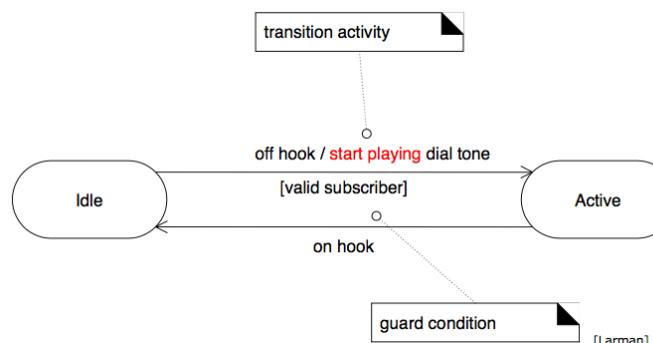


## System-Zustandsdiagramm

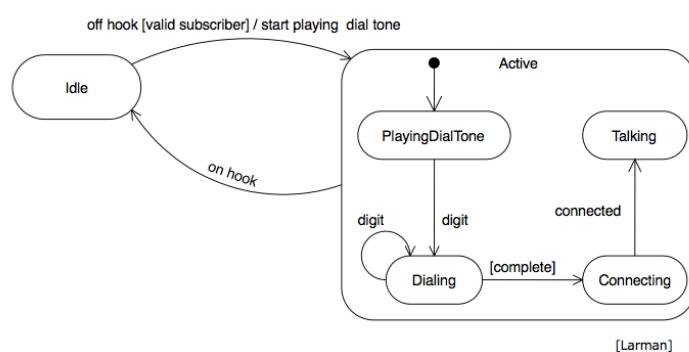
Es zeigt eine zulässige Reihenfolge aller System-Ereignisse. Dabei ist es die Vereinigung aller Use Case Zustandsdiagramme. In komplexen Systemen kann es sehr unübersichtlich werden.

## Zusätzliche Notationselemente

### Activities und Guards



### Verschachtelte Zustände (Nested States)



## Typische Anwendungen von Zustandsdiagrammen

Meist wird es für Echtzeit-Systeme eingesetzt, da dort ein zustandsabhängiges Verhalten meist komplex ist. Dazu zählen Steuerungen oder auch Telekommunikationssysteme.

Seltener findet es Verwendung bei EDV-Systemen. Wenn das Verhalten sehr einfach ist (2-3 Zustände) sind Zustandsdiagramme von begrenztem Nutzen. Es kann aber in Teilbereichen (UI Navigation) die Verständlichkeit erhöhen.

- **Beschreibung grob lesen:**

- o Relevantes System und Abstraktionsebene festlegen

- **Beschreibung mehrmals detailliert lesen und:**

- o Relevante Zustände identifizieren (Pflicht)
- o Relevante Ereignisse (Events) identifizieren (Pflicht)
- o Transitionen erfassen (Pflicht)
- o Guards identifizieren und am richtigen Ort erfassen (Nicht immer sinnvoll)
- o Aktivitäten identifizieren und am richtigen Ort erfassen (Nicht immer sinnvoll)
- o Diagramm evtl. mittels Nesting vereinfachen. (Nicht immer sinnvoll)

## Häufige Stolpersteine

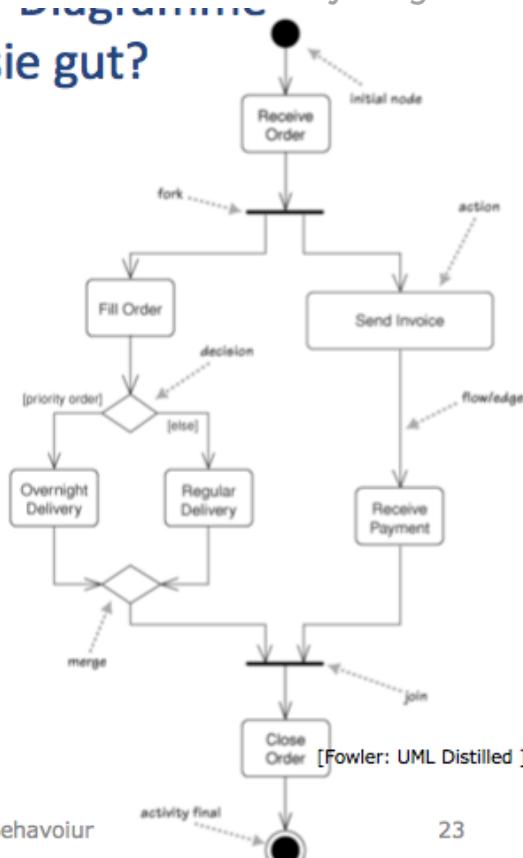
- Falsche Abstraktionsebene (zu abstrakt oder zu konkret)
- Aktivitäten statt Zustände identifiziert (Zustandsdiagramm ist kein Aktivitätsdiagramm auch wenn es manchmal ähnlich aussieht!)
- Unterschiede zwischen Ereignissen, Aktivitäten und Guards unklar:
  - o Ereignisse (Event): Externe Stimuli – Empfangen einer Nachricht,
  - o Aktivität (Activity): Reaktion des Systems – Imperative Anweisung
  - o Guard: Vorbedingung – Boolesche Formel
- Start- und End-Zustand fehlen (End-Zustand darf teilweise fehlen)

## Activity Diagramme

Larman im Kapitel 28.

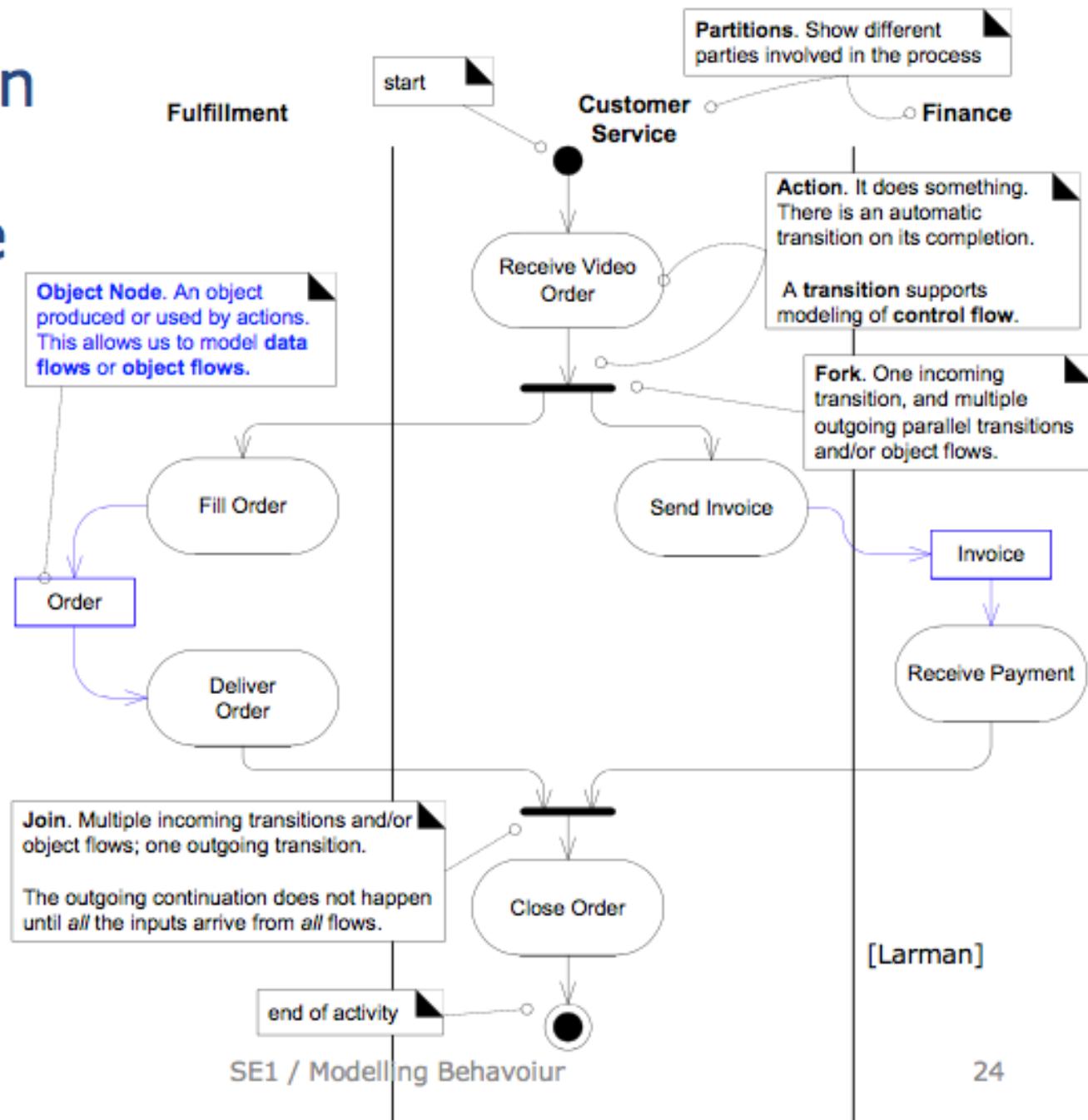
### Was sind UML Activity Diagramme und wofür sind Sie gut?

 **sie gut?**



UML Activity Diagramme zeigen sequentielle und parallele Aktivitäten. Es wird verwendet für Geschäftsprozesse, Use Cases mit parallelen Aktivitäten und im Design, speziell bei paralleler Verarbeitung.

Ein Activity Diagramm zeigt kurzgesagt eine Abfolge von Actions.

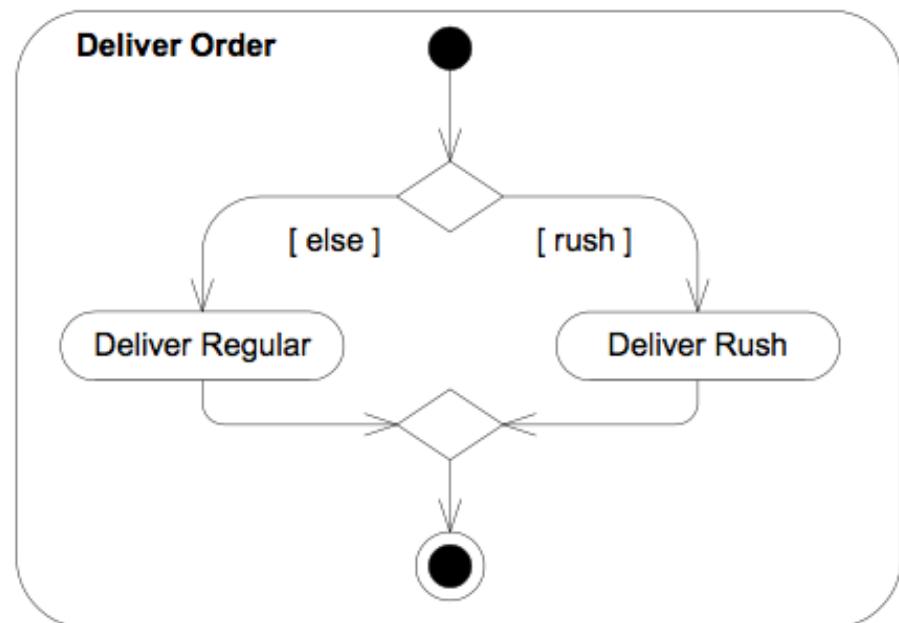


## Unterschied zwischen Join und Merge

Bei einem Join treffen sich zwei parallele Abläufe wieder, welche sich zuvor mit einem Fork getrennt haben. In jedem Fall werden aber beide Abläufe durchlaufen.

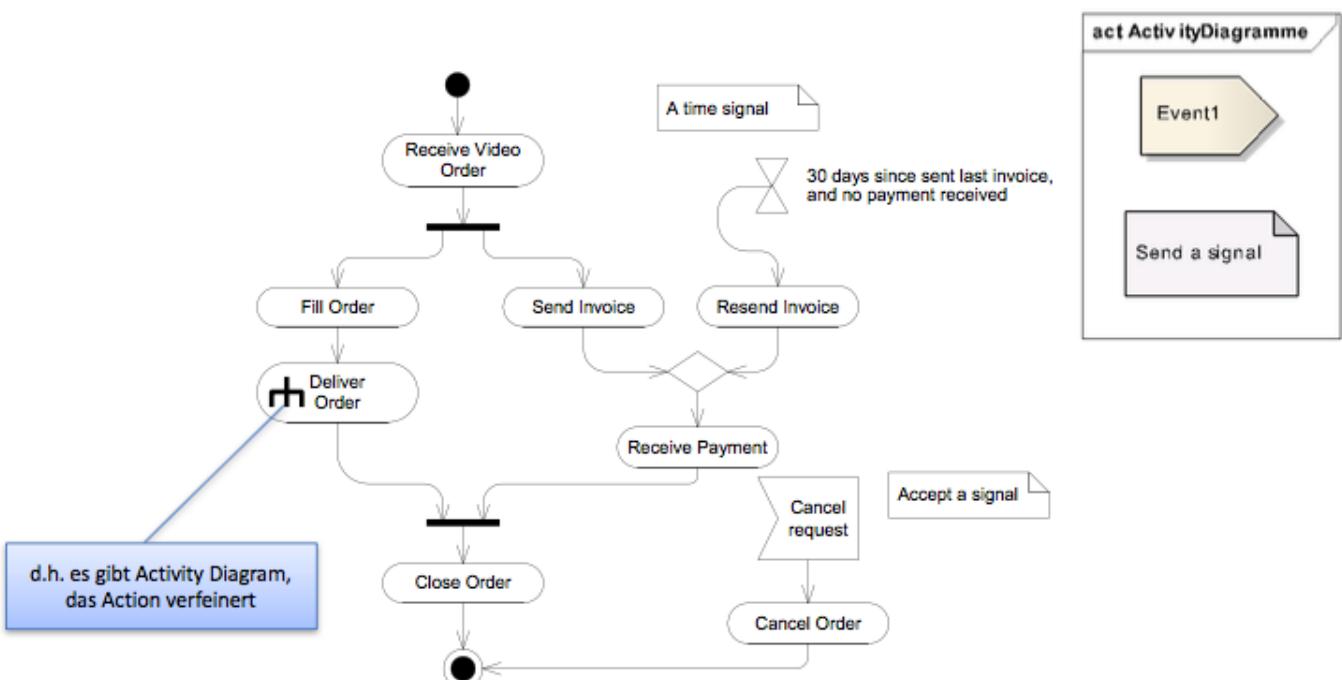
Bei einem Merge treffen sich zwei Abläufe wieder, welche zuvor durch eine Decision getrennt wurden. Je nach Input wird der eine oder der andere Weg durchlaufen, aber nicht beide.

**Decision:** Any branch happens.  
Mutual exclusion



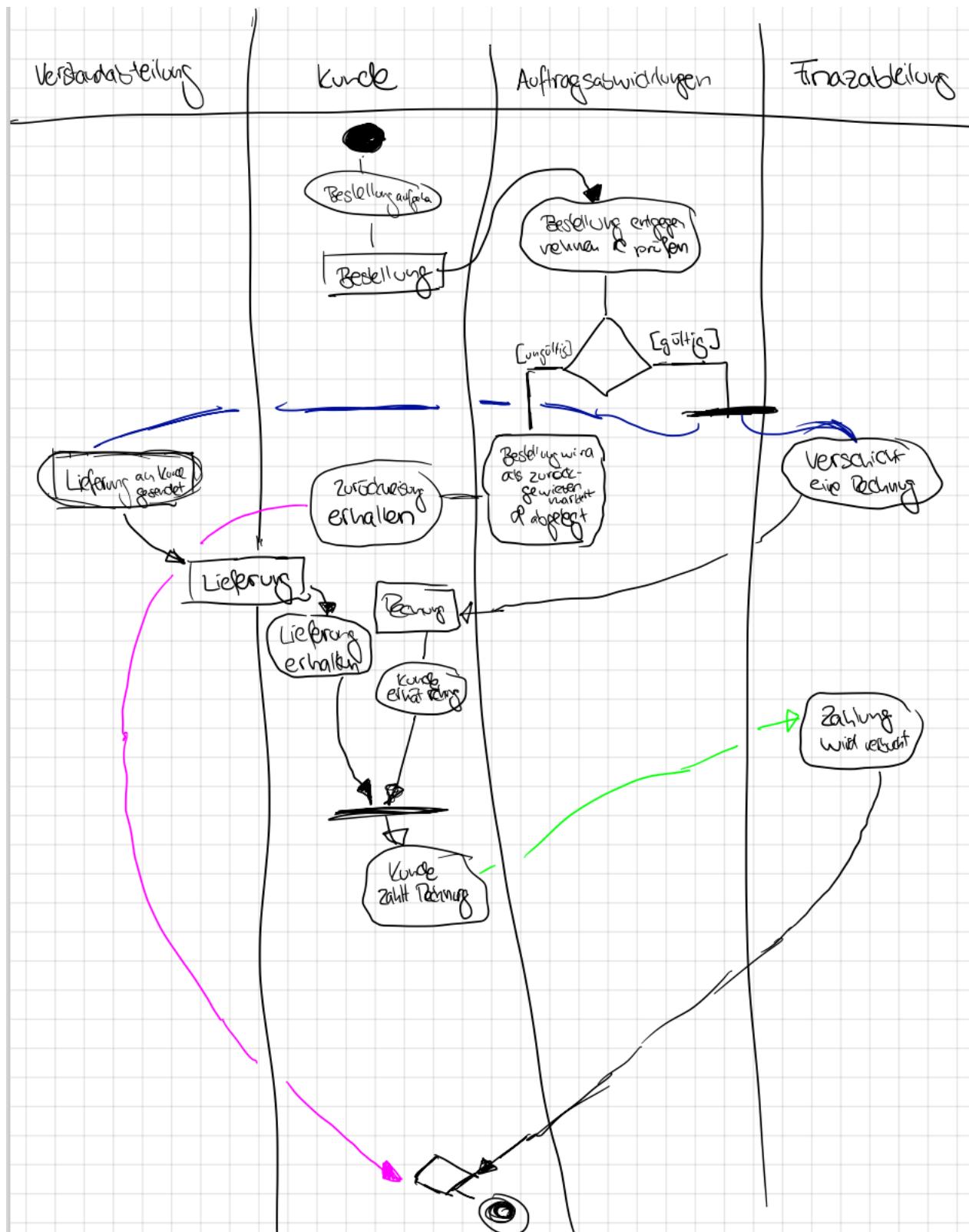
**Merge:** Any input leads to continuation. This is in contrast to a *join*, in which case *all* the inputs have to arrive before it continues.

[Fowler]



## Beispiel Activity Diagramm – Bestellabwicklung

Die Auftragsabwicklung nimmt Bestellungen entgegen und prüft diese. Ist eine Bestellung nicht gültig, so wird dem Kunden eine entsprechende Antwort geschickt und die Bestellung wird als „zurückgewiesene“ Bestellung abgelegt. Ist die Bestellung gültig, so verschickt die Versandabteilung die gewünschten Artikel in einer Lieferung an den Kunden. Gleichzeitig verschickt die Finanzabteilung die Rechnung an den Kunden. Der Kunde zahlt natürlich erst, wenn er auch die Lieferung erhalten hat. Trifft die Zahlung vom Kunden bei der Finanzabteilung ein, so wird sie verbucht und als „erledigt“ abgelegt.



# Software Testing

Dieser Teil ist nicht im Larman enthalten.

## Einführung

### Definition

Die Ausführung von Software mit dem Ziel Fehler zu finden und sicherzustellen, dass die Vorgaben erfüllt sind. Testen kann aber keine Fehlerfreiheit garantieren. Vollständiges Testen ist bei nichtrivialer Software unmöglich.

Ad-Hoc etwas „Herumprobieren“ ist nicht Testen. Testen muss systematisch und wiederholbar sein.

### Speziell

Test Driven Development (TDD). Dies ist einen Entwicklungs- bzw. eine Codiermethodik. Dabei werden die Tests vor dem Code geschrieben.

## Anforderungen

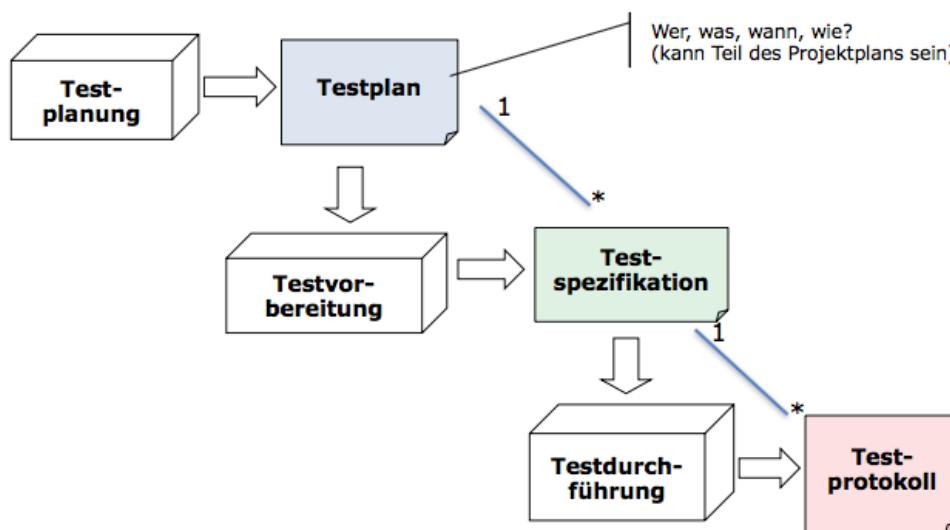
- Geplant → Testplanung (wer, was, wann, wie?)
- Systematisch spezifiziert → Testspezifikation
- Resultate festgehalten → Testprotokoll

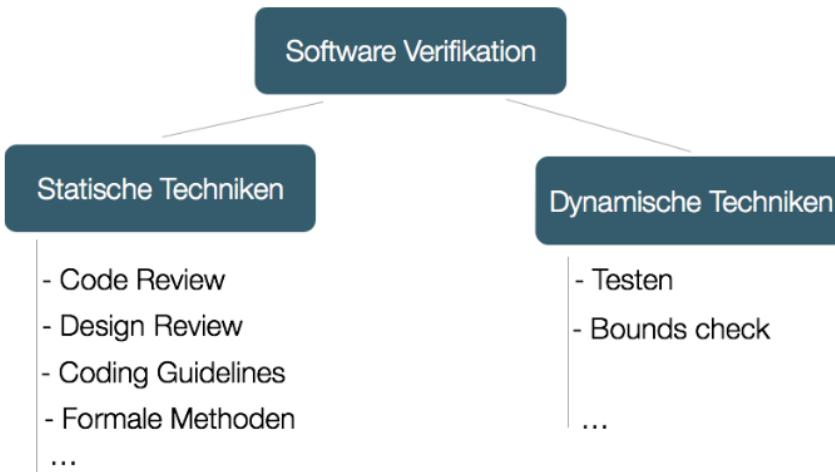
Dies ist es das es reproduzierbar sowie nachvollziehbar ist. Wenn immer möglich auch automatisch.

## Arten von Tests

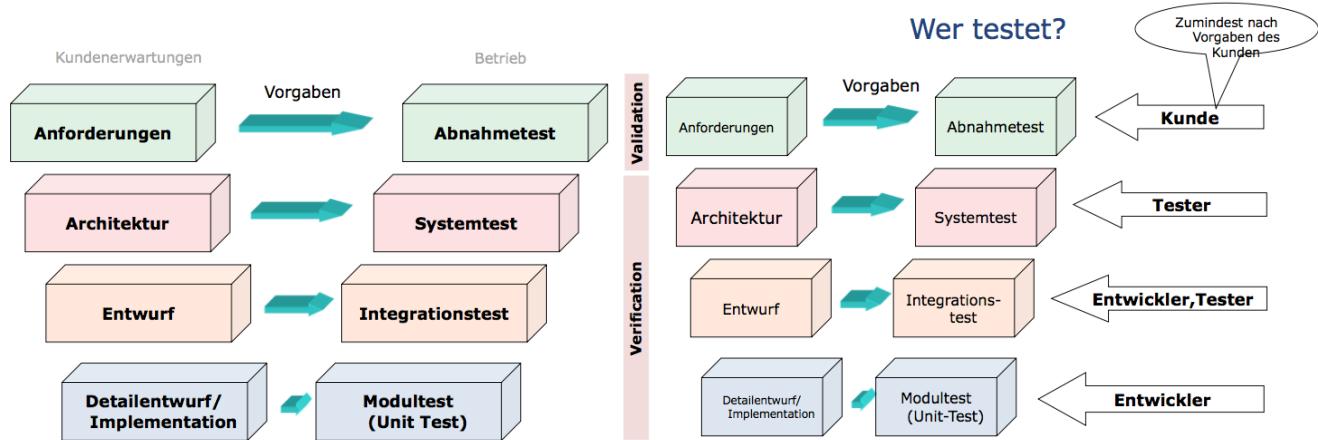
- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>▪ Anforderungen kategorisiert:           <ul style="list-style-type: none"> <li>▪ <b>Funktionale Anforderungen</b></li> <li>▪ <b>Nichtfunktionale Anforderungen</b> <ul style="list-style-type: none"> <li>▪ <b>Leistung</b></li> <li>▪ <b>Usability</b></li> <li>▪ ...</li> </ul> </li> </ul> </li> </ul> | <ul style="list-style-type: none"> <li>▪ Tests entsprechend kategorisiert:           <ul style="list-style-type: none"> <li>▪ <b>Funktionale Tests</b></li> <li>▪ <b>Nichtfunktionale Tests</b> <ul style="list-style-type: none"> <li>▪ <b>Leistungstests</b></li> <li>▪ <b>Usability Tests</b></li> <li>▪ ...</li> </ul> </li> </ul> </li> </ul> |
|---|--|

## Testprozess





## Testebenen und Tester



## Begriffe – Verifikation und Validierung

**Verifikation – „Are we building the product right?“**

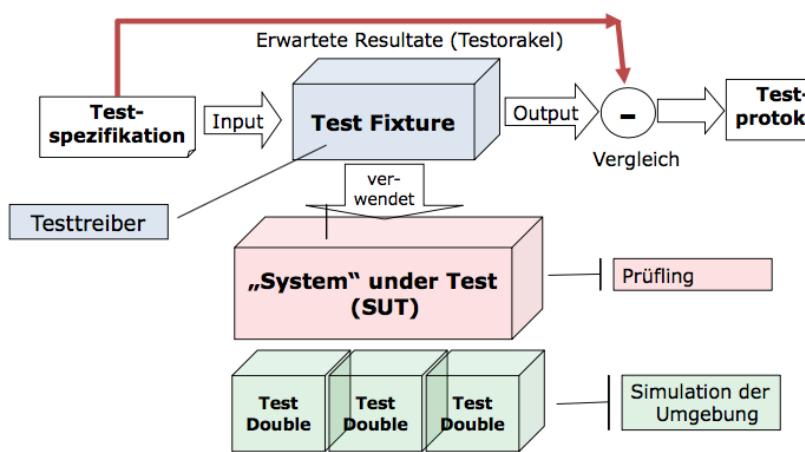
Überprüft Work Products während der Entwicklung, ob Sie ihre Vorgaben erfüllen.

**Validierung – „Are we building the right product?“**

Überprüft das Softwareprodukt, ob es die Anforderungen des Auftraggebers erfüllt.

## Testmethoden

### Testumgebung



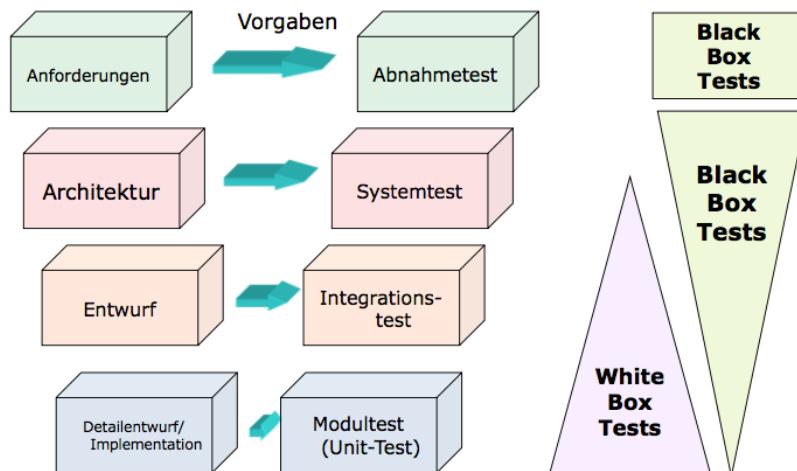
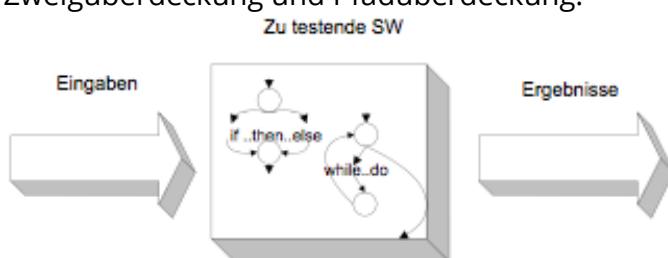
## Black-Box Tests

Testfälle ohne Kenntnis der inneren Struktur. Zu den Methoden zählen Äquivalenzklassen, Grenzwertanalyse und Zustand-basiertes Testen.



## White-Box Tests

Testfälle mit Kenntnis der inneren Struktur. Dabei wird meist kontrollfluss-orientiert oder datenfluss-orientiert getestet werden. Zum Kontrollfluss-orientierten Testen gehören Anweisungsüberdeckung, Zweigüberdeckung und Pfadüberdeckung.



## Black-Box Testmethoden

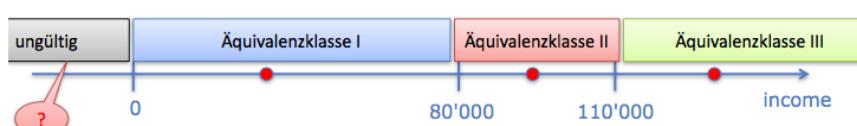
### Äquivalenzklassen

**Äquivalenzklasse** = Wertebereich von Eingabegrößen, für welche der Prüfling voraussichtlich das gleiche Verhalten zeigt.

### Beispiel der Steuerberechnung long calculateTaxes(long income)

„Für die Einkommensklasse von 80.000 bis 110.000 sei der Steuersatz 25 %, darunter 20% und darüber 30%. Negative Einkommen gibt es nicht.“

**Testdaten:** Aus jeder Äquivalenzklasse einen Wert

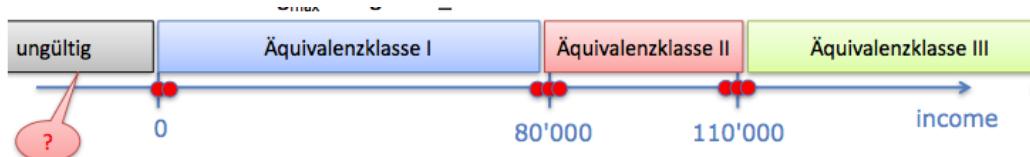


**Grenzwertanalyse** = Wahl der Testdaten an den Grenzen der Äquivalenzklassen. Genau auf der Grenze, knapp darüber oder knapp darunter.

### Beispiel der Steuerberechnung long calculateTaxes(long income)

„Für die Einkommensklasse von 80.000 bis 110.00 bis 110.00 sei der Steuersatz 25 %, darunter 20% und darüber 30%. Negative Einkommen gibt es nicht.“

**Testdaten:** 0, 1, 79999, 80000, 80001, 109999, 110000, 111001 und long.MAX\_VALUE



### Was machen wir mit ungültigen Wertebereichen?

Beispiel der Steuerberechnung: „negative Einkommen gibt es nicht“.

#### Variante A

Man könnte bei negativen Einkommen 0 zurückgeben (optimistische Behandlung) oder eine Exception werden (defensive Programmierung). Damit verändert, erweitert man die Anforderungen an die Methode. Ungültige Wertebereiche gibt es nicht mehr und damit braucht es auch dafür entsprechende Testfälle.

#### Variante B

Man gibt dem Aufrufer die Verantwortung die Methode nur mit einem Einkommen von  $\geq 0$  aufzurufen (Design by Contract, Fehler Barrikaden). Stammen die Einkommensdaten aus externen Schnittstellen (UI, Kommunikationsschnittstellen, Flies, externen Datenbanken) muss vor dem Aufruf die Gültigkeit überprüft werden (Validierung). Testen muss man die Methode calculateTaxes(long income) aber nicht mit negativen, d.h. ungültigen Werten.

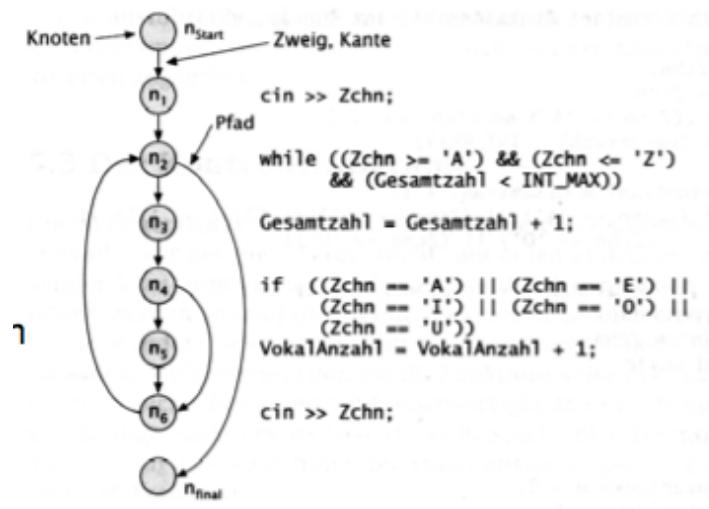
#### Welche ist vorzuziehen?

Keine klare Antwort. Diese Wahl hat grossen Einfluss nicht nur auf das Testing, sondern das Fehlerverhalten des Gesamtsystems. → Kritikalität des Gesamtsystems muss betrachtet werden.

#### Zustandsbasiertes Testen

Beispiel eines Stacks. Die Zustände sind eine Art von Äquivalenzklassen.

Testfall	Zustand	Stack leer	Stack halbvoll	Stack voll
Element hinzufügen				
Element entfernen				



Die Kenntnis der Kontrollstrukturen ist die Basis für die Testfälle.

### Kontrollflussgraph

Jedes Statement als Knoten bzw. sequentielle zu einem Knoten zusammenfassen. Die Kanten verbinden Knoten. Entsprechend können so auch Verzweigungen und Schleifen abgebildet werden.

Die Tests müssen so erstellt werden, so dass eine gute Überdeckung (test coverage) vorhanden ist. Die Testüberdeckung lässt sich mit einem Dynamic Analyzer messen.

## Testüberdeckung (test coverage)

Hauptsächlich bezogen auf das White-Box Testing.

### Anweisungsüberdeckung (statement coverage)

Dies ist der prozentuale Anteil der Anweisungen, die in Test ausgeführt werden/wurden. Im Prinzip ist 100% das Ziel, aber ohne Getter und Setter sowie trivialen Code.

### Zweigüberdeckung (branch coverage)

Prozentualer Anteil der Kanten (Zweige), die in Tests durchlaufen werden. Eine 100% Zweigüberdeckung enthält auch eine 100% Anweisungsüberdeckung. Zusätzlich werden auch alle „leeren Zweige“ durchlaufen.

### Pfadüberdeckung (path coverage)

Prozentualer Anteil der Pfade, die in Tests durchlaufen werden. Ein Pfad ist ein möglicher Weg durch den Kontrollgraph. Kleine Programme mit Schleifen resultieren in riesigen Anzahl von Pfade. Eine 100% Pfadüberdeckung ist kaum zu erreichen.

Dabei sollten die Testfälle reduziert werden. → Äquivalenzklassen/Grenzwertanalyse auf die Kontrollstrukturen übertragen.

### Bedingungsüberdeckung (condition coverage)

Bei den zusammengesetzten Bedingungen werden die verschiedenen Kombinationen getestet.

Eine einfache Bedingungsüberdeckung ist vorhanden, wenn alle atomaren Bedingungen mind. Einmal true und einmal false waren. Dies garantiert aber keine 100 %ige Anweisungsüberdeckung.

Wenn man auch alle nichtatomaren Bedingungen hinzunimmt kommt es zu einer minimalen Mehrfach-Bedingungsüberdeckung.

### Funktionsüberdeckung

Spezifikationsorientiertes Testen

- Basis sind die Use Cases „tut's das, was der Kunde spezifiziert hat?“
- Ein Szenario pro (Sub-) Use Case erstellen (Testdaten plus erwarteter Output)
- Gut durch Kunden ausführbar
- Black-Box Test auf Systemebene

## Vorteile

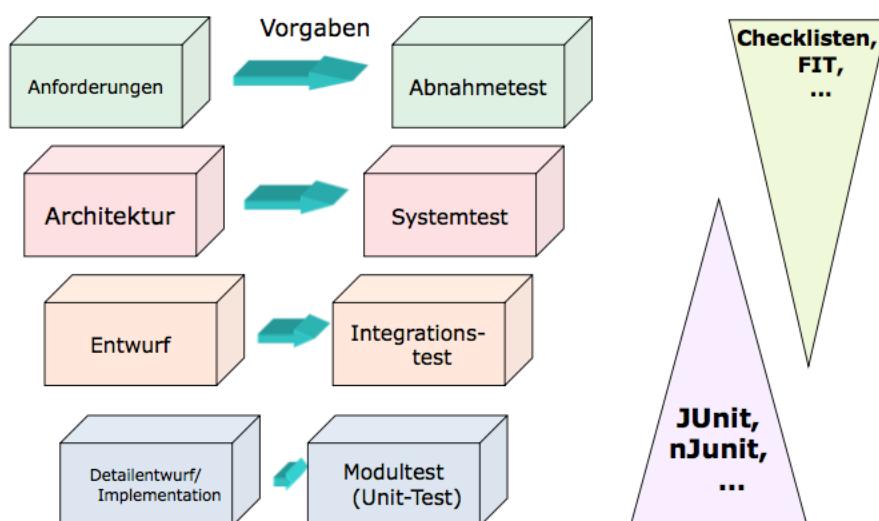
- Wiederholbarkeit → Regressionstests
  - o Absicherung bei Änderung, Portierung, Erweiterung
  - o Geringe/keine Kosten bei einer Wiederholung
- Eindeutige Spezifikation
  - o Testcode ist Programmcode und damit eindeutig

## Nachteile

- Es gibt mehr Code zu schreiben und zu pflegen
- Testcode ist auch Programmcode
  - o Werden wirklich die richtigen Anforderungen getestet?
  - o Was muss überhaupt getestet werden?

In der Regel gilt: Je mehr automatisiert, desto besser.

## Testwerkzeuge



## Unit Tests

Testen eines Moduls (Übersetzungseinheit, Quell-Datei oder Klasse). Der Test findet durch den Programmierer selbst statt. Es ist automatisch und daher auch wiederholbar. Vorausgesetzte Ergebnisse werden mit Assertions geprüft. Heute findet dies meist mit einem Unit Testing Framework statt. Zum Beispiel JUnit, oder CUTE.

## JUnit

Wie detailliert im Kapitel Microtesting behandelt.

## Testprinzipien

### Das wichtigste am Testen

- Früh testen
- Häufig testen
- Systematisch testen
- Automatisiert testen

„Erst den Test schreiben, dann den Code dazu.

Dabei gibt es mehrere Vorteile. Es zeigt, dass die Aufgabe verstanden worden ist. Dies kann man sozusagen eine „ausführbare Spezifikation“ nennen. Die Tests sind wiederholbar. Das Refactoring steht sich leicht daher und es entsteht kein unnötiger Code.

### Pragmatische Prinzipien beim Unit Testing

- Test anything that might break → Keine Tests für „funktionslosen“ Code
- Test everything that does break → bei Fehler zuerst einen Test schreiben, der den Fehler reproduziert und dann korrigieren
- New code is guilty until proven innocent
- Write at least as much test code as production code
- Run local test with each compile → nicht weitermachen bevor Tests laufen
- Run all tests before check-in to repository

### ATRIP – Gute Uni Tests

#### Automatic

- nicht nur von Hand, sondern auch bei check-ins, daily-builds oder fortlaufend

#### Thorough

- sorgfältig, bug → test → fix

#### Repeatable

- gleiche Ergebnisse unabhängig wie oft aufgerufen (aufräumen nicht vergessen)

#### Independent

- keine (zeitlichen) Abhängigkeiten zwischen einzelnen Tests

#### Professional

- Auch die Tests überprüfen: Bedingungen negieren und prüfen, ob Test fehlschlägt. Absichtlich kurzfristig Fehler in Code einbauen.

# Microtesting (e-Learning)

Mircotesting geht über das Schreiben von automatischen Unit Tests bis auf den Objectlevel hinunter.

Verifiziert mit Unit Test, dass dies aller richtig funktioniert. Es erlaubt uns Designänderungen an der Software vorzunehmen, ohne uns gross darum zu kümmern ob es nachher immernoch gleich funktioniert. Laufen immernoch alle Tests durch, so läuft auch die Software wie gewünscht.

## Drei Dimensionen von Microtesting

### Complexity

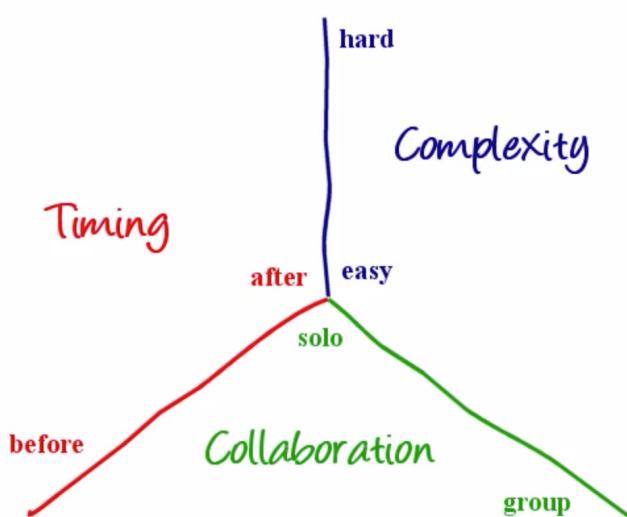
Dies sie die Complexity des Code, mit welchem wir arbeiten (Hard Code, Easy Code)

### Collaboration

Möchten nur wie alleine Testen oder möchten wir innerhalb von Gruppen(Objekten/Abhängigkeiten) testen.

### Timing

Wann schreiben wir unsere Tests, vorher oder nachher?



Dieses e-Learning deckt nur die Basics von Microtesting ab. Daher geht es nur um Easy/Solo/After. Easy steht dafür, dass die Klassen keine schweren Probleme darstellen. Solo steht dafür, dass die Klasse hauptsächlich alleine arbeiten kann und keine Abhängigkeiten hat. Mit After ist gemeint, dass die Test nach dem Code geschrieben werden.

### What Is a mircotest?

„A mircotest is a short, simple, automated test that probes a single behavior of a single class“.

### Geschichte

Die Geschichte ist schon länger vorhanden. Der Anfang fand in den 90iger Jahren statt. Zum Anfang empfanden die meisten TDD (Test Driven Development) als mühsam, als das es Sie unterstützt hätte. Die Coaches gingen um die Welt und haben dies versucht zu vermitteln.

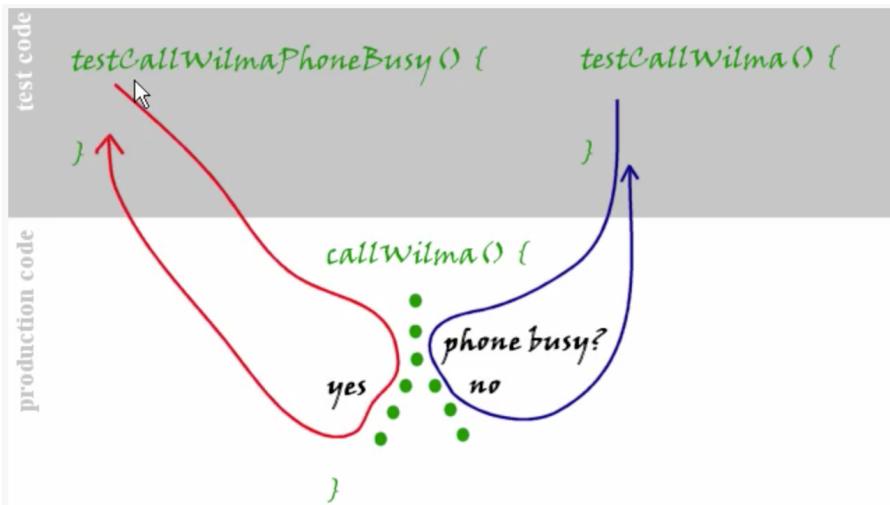
### Testing in Isolation

Ein guter Microtest setzt ein einfaches neues einzelnes Objekt ein und testet das Verhalten davon, ob es diesem entspricht was der Entwickler machen wollte.

## The Standard Mircotest

Für den Testcode wir eine eigene Klasse geschrieben. Innerhalb der Klasse werden verschiedene Methoden erstellt, welche das Verhalten zum Beispiel der Klasse Fred testen. Am besten immer test im Namen angeben. Um testen gehört natürlich auch die Instanzierung. Für eine Methode in der Klasse „Fred“ kann es natürlich auch mehrere Testmehoden geben, welche diese testen.

Wenn entsprechende Verzweigungen vorhanden sind, müssen natürlich alle Wege getestet werden.



### Nameing and Scope

Wie zu sehen ist, ist der Scope einer Mircotest sehr klein. Es muss nicht unbedingt nur einer Methode entsprechen, aber es soll nur eine kleine Funktion oder Verhalten testen.

Für die Tests sollten möglich einfache, sprechende Namen verwendet werden. Hier ein Beispiel für eine Seitennavigation.

The microtest	Verifies the behavior of
<code>nextPageNavigation()</code>	basic forward navigation
<code>previousPageNavigation()</code>	basic backward navigation
<code>nextFromAlbumTitleIsFirstTrackTitle()</code>	Next on an album boundary (the very first page)
<code>nextFromTrackTitleIsFirstContentPage()</code>	Next on one track boundary
<code>nextFromLastContentPageInTrackIsNextTrackTitle()</code>	Next on the other track boundary
<code>previousFromTrackTitleIsLastContentPageInPreviousTrack()</code>	Previous on one track boundary
<code>previousFromFirstContentPageIsTrackTitle()</code>	Previous on the other track boundary
<code>previousFromFirstTrackTitleIsAlbumTitle()</code>	Previous that reaches the album's first page

### A Sample microtest

Hier ein kleines Beispiel. Das System im Tests hat Benutzer, davon hat jeder sein Profile. Das Profile trackt verschiedene Dinge. Zum Beispiel ob er die EULA akzeptiert hat.

Der Tests erstellt ein leeres Profile, akzeptiert die EULA und überprüft nachher ob die Methode acceptEULA auch richtig funktioniert hat.

```
@Test
public void acceptEULA() {
    PropMgr propMgr = new PropMgr();
    Profile profile = new Profile(propMgr);

    profile.acceptEULA(DateUtil.createDate(2005, 12, 18, 11, 56));

    assertTrue(profile.acceptedEULA());
    assertEquals("Time of EULA acceptance", "200512181156", profile.eulaAcceptance());
}
```

Jeder Mircotest hat etwa die gleiche Struktur. In dem meisten Fällen wir da auf A3 oder AAA referenziert.

### Arrange

Hier sollten die Objekte für die Tests instanziert werden, sowie weitere Objekte welche wir dazu brauchen. Allfällige Datenwerte sollen gesetzt werden, dass wir auf den für uns interessanten Ausführungsteil kommen.

### Act

Führe eine oder mehrere Methoden auf unserem unter dem Test stehenden Objekt aus.

### Assert

Prüfe die Rückgabewerte oder die Variablen, ob Sie diesem entsprechen was wir erwarten. Die Assert-Bedingungen sollten true sein nach der Ausführung.

### (Teardown), nur wenn gebraucht

Löschen oder zerstören von Objekte, welche wir während den Tests gebraucht haben. Meist ist dies aber nicht nötig, da an vielen Orten eine automatische Garbage Collection vorhanden ist.

## Test Everything Interesting?

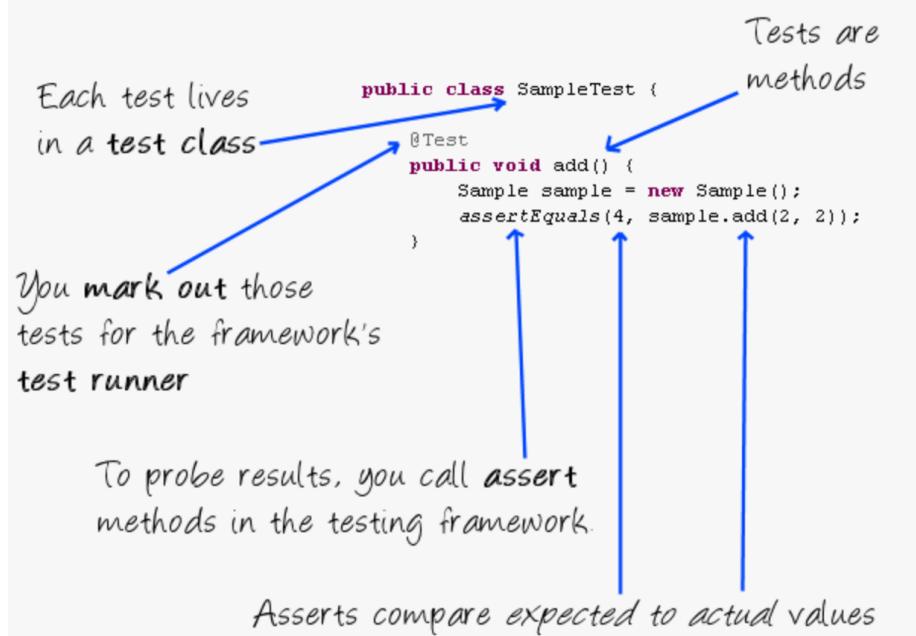
Unser Ziel ist es alle interessanten Pfade innerhalb der Klasse zu testen und nicht 100 % Coverage zu erreichen. Folgendes sind nicht unsere Ziele

- **100 % Coverage** (Gibt nicht an, wie viele das interessant sind und sie geben nicht an ob es mit anderen Daten immernoch gleich aussieht)
- **One-to-One Mapping** (Wir möchten nicht jede einzelne Methode testen, sondern ob das Verhalten und die Funktionalität unserer Klasse stimmt.)
- **Catch every possible error** (Sehr zeitaufwändig, man soll ich klar dafür entscheiden, was man testen will).

Wenn wir verschiedene Executions Paths testen müssen wir uns bewusst sein, dass wir nur mit Beispieldaten arbeiten. Daher sollte diese mehrmals mit verschiedenen Daten getestet werden.

## All about xUnit

JUnit ist das meist verbreitete Tool auf dem Plant um Java Code zu testen. Die Anfänge gehen bis in die Mitte-90iger zurück.



### Test Case

Die Testmethoden leben in einem Testcase, einer Klasse oder einem File, welches nur das Testing da ist. Die Testklassen werden meist mit XTest bezeichnet, wobei das X für den Klassennamen der zu testenden Klasse steht.

### Test Methods

Jede Testmethode ist public void und nimmt keine Argumente entgegen. Optional wirft Exceptions, welche das Testframework dann abfangen muss. Die Methode kann irgendeinen Namen haben. Wichtig ist die Notation am Anfang mit @Test.

### Assertions

In jedem Assert wird angegeben, was du erwartest. JUnit vergleicht dann den erhaltenen Wert mit dem angegebenen Wert.

#### Check that two **Objects** are equal:

```

@Test
public void stringEquality() {
    String expected = "Something";
    String actual   = "Something";
    assertEquals(expected, actual);
}

```

#### Check that two integers are equal:

```

@Test
public void intEquality() {
    int expected = 10;
    int actual   = 10;
    assertEquals(expected, actual);
}

@Test
public void integerEquality() {
    Integer expected = new Integer(11);
    Integer actual   = new Integer(11);
    assertEquals(expected, actual);
}

```

## Software Engineering 1

Due to their internal representation, it is tricky to say when two `floats` or two `doubles` are equal. The canonical solution is to assert equality to within a certain precision, or tolerance:

```
@Test
public void doubleEquality() {
    double expected = 10.01;
    double actual = 10.01;
    double precision = 0.00001;
    assertEquals(expected, actual, precision);
}

@Test
public void floatEquality() {
    float expected = 10.0001F;
    float actual = 10.00009F;
    float precision = 0.0001F;
    assertEquals(expected, actual, precision);
}
```

Check that `Object` references (like `Integer` or `String`) point to the same object, or whether they are `null` or not.

```
@Test
public void objectsSame() {
    Integer anObject = new Integer(1);
    Integer sameObject = anObject;
    assertEquals(anObject, sameObject);
}

@Test
public void objectsNotSame() {
    Integer anObject = new Integer(1);
    Integer anotherObject = new Integer(1);
    assertEquals(anObject, anotherObject);
}

@Test
public void nullObject() {
    String object = null;
    assertNull(object);
}

@Test
public void nonNullObject() {
    String object = "Hello";
    assertNotNull(object);
}
```

### Check that a boolean expression is true or false:

```
@Test
public void booleanTrue() {
    boolean result = true;
    assertTrue(result);
}

@Test
public void booleanFalse() {
    boolean result = false;
    assertFalse(result);
}
```

### Merke

Die `assertEquals` greift auf die `equals`-Methode des Objektes zurück. Diese muss daher in jedem Fall implementiert werden.

## Failure Messages

Man kann zusätzlich eine Fehlermeldung mitgegeben als Argument, um den Mismatch zwischen den beiden Werten zu erklären.

```
@Test
public void add() {
    assertEquals("2+2 should equal 4 for most values of 2", 4, add(2, 2));
}
```

Wenn der Test nicht erfolgreich ist, wird zusammen mit dem Testnamen und der Zeilennummer die Fehlermeldung ausgegeben.

## Testing Exceptions

Wenn der Code möglicherweise eine Exception werfen kann, muss dies der Test speziell handeln.

### The easy way

```
@Test(expected=NumberFormatException.class)
public void parseIntThrowsException() {
    Integer.parseInt("trying to parse text instead of numbers");
}
```

### The full-control solution

```
public void testparseIntThrowsException() {
    try {
        Integer.parseInt("trying to parse letters instead of numbers");
        fail("Should have thrown NumberFormatException");
    } catch (NumberFormatException expectedException) {
        // if we get here it means the test has passed
    }
}
```

Note: Take care to catch the exact type of exception.

Note: Do not omit the failure call; if you do, code that fails to throw an exception will incorrectly pass.

## Duplicated Setup Code

Dazu kann es kommen, wenn man immer wieder ähnliche Testsfälle erstellt, wo zum Beispiel Objekte initialisiert. Wenn immer möglich sollte redundanter Code entfernt werden und ein spezielle Setupmethode ausgelagert werden.

## Teardown

Wie bei der gemeinsamen Datenkonfiguration entfernt die After-Methode die Duplizierung in der Datenbereinigung. Es ist leichter zu lesen als konvolutierte try / catch-Blöcke zum Aufräumen von Daten und ist besonders nützlich, wenn ein Test abnormal beendet werden könnte.

```

public class UserCreationTest extends UserManagementTests {
    private Group group;
    private User user;

    @Before
    public void createGroupAndUser() {
        group = createGroup(groupname);
        user = createUser(username);
    }

    @After
    public void removeGroupAndUser() {
        removeUser(user);
        removeGroup(group);
    }
}
  
```

## One-Time Setup and Teardown

In einen Fällen ist es doof, das Setup und der Teardown für jeden Tests immer zu wiederholen (vor allem bei zeitaufwändigen Operationen). JUnit erlaubt ein Spezialsetup- und Teardown, welche nur einmal für die Klasse durchlaufen werden.

```

public class HeavyDatabaseUserTest...
@BeforeClass
public static void openSession() {
    openDBConnection();
}

@AfterClass
public static void closeSession() {
    closeDBConnection();
}
  
```

## Principles of Running XUnit Tests

Jeder Test muss natürlich bestanden werden. Wenn es zu einem Error kommt, dann hat der Code terminiert, weil es ihm nicht möglich war es korrekt zu beenden. Kommt es zu einem Fehler, dann entsprechen die erwarteten Werte nicht den aktuellen Werten.

Die Testausführung kann wie folgt beschrieben werden. Ein Testrunner startet und wir geben ihm an, welche Tests das er machen soll. Der Runner durchläuft jeden Test. Am Schluss sagt er, welche Tests bestanden sind und welche nicht.

## Small is Superb!

Der Code sollte wenn immer möglich aufgeteilt werden. Grosse Test verschlechtern die Übersicht, es sind viele Wege vorhanden, man weiss nicht genau wo ein Fehler sein wird und es sind zu viele Abhängigkeiten im Spiel.

## Precise is perfect

Ein Chirurg und Software haben nicht viel gemeinsam, aber sie brauchen beide präzise Tools um zu arbeiten. Egal ob am TDD machen will, eine einfache Tests schreiben oder Fälle machen um Bugs zu finden, in jedem Falle brauche ist Mircotest, welche präzise sind.

## Genaue Namen

Ich sollte wissen was jeder Test macht, ohne dass ich den Inhalt der Methode gelesen habe.

## Genaue Assertions

Ich möchte den Testassert lesen können und verstehen was der Test macht/will.

## Genaue Couverage

Ich möchte das jeder Test mit einem Minimum an Code ausführen um den Test zu bewerkstelligen

## Boundaries are Beautiful!

Gute Programmier nutzen Encapsulation um den Code abzugrenzen. Dies heisst soviel, wie Teile des Codes von aussen her nicht zugriffbar machen. Eine Fernbedienung ist dafür ein gutes Beispiel. Im Testing nutzen wir sozusagen immer die Fernbedienung und gehen nie direkt auf den Fernseher.

## Independet, Side-Effect-Free Mircotests

Die Tests sollten nicht auf Daten aus anderen Tests beruhen. Gleichzeitig will man ja das Data Setup geshared haben. Dies sollte man wenn immer möglich machen, aber eben aufpassen das die Daten für jeden Tests unabhängig bleiben.

## Mainfestations Of Test Dependence

Zwei Beispiele, welche Abhängigkeiten/Seiteneffekte aufweisen.

### Expectations: Test A is responsible for setting up some of test B's data context.

In the following example, test B will only run successfully if test A runs successfully first:

```
@Test
public void a() {
    SystemConfiguration.setInstance(new TestSystemConfiguration());
}

@Test
public void b() {
    ... // exercise code that calls SystemConfiguration.getProperty(...)
    ... // assert on the code, assuming test properties were read
}
```

### Side effects: Test A could affect test B's data context in ways test B isn't built to handle.

In the following example, test B verifies `countUsers()`: It creates two users, asserts that the count yields 2, then removes the users. Test A also creates a user (for unrelated purposes) but doesn't clean it up. When test B runs, it will fail because 3 users will be in the system, not 2:

```
@Test
public void a() {
    UserManager.createUser("Yoda");
    ...
    // Yoda isn't removed from the system
}

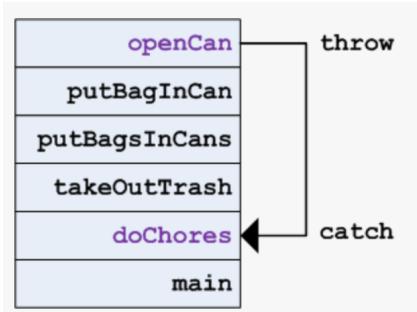
@Test
public void b() {
    UserManager.createUser("Luke");
    UserManager.createUser("Darth");
    assertEquals(2, UserManager.countUsers());
    UserManager.removeUser("Darth");
    UserManager.removeUser("Luke");
}
```

Wenn man folgenden Grundsatz einhält ist obiges relativ einfach einzuhalten. Schreiben Sie jeden Tests, als wäre es der einzige der läuft.

Vermeiden sie persistente Zustände ganz. Wenn dies nicht möglich ist, schauen dies das Sie die persistenten Zustände wie Globale und statische Variablen wieder aufräumen. Alles was auf den nächsten Test Einfluss haben kann, soll auf den Ursprungszustand gebracht werden.

## Microtests & Exceptions

In einer Exception springen wir von einem Ort im Stack zu einem anderen Ort. Exceptions sind nicht normal. Dinge die nicht normal sind, sind interessant. Dinge die interessant sind, müssen getestet werden.



## Aufgeräumte Baustelle

```
for (int i = 0; i <= tableModel.getColumnCount() - 1; i++) {
    TableColumn column = table.getGridColumn().getColumn(i);
    // column.setCellRenderer(new TableCellRenderer());

    if (i == 0)
        column.setPreferredWidth(100);
    else if (i == 1)
        column.setPreferredWidth(400);
    else
        column.setPreferredWidth(100);
}
```

Konsistentes Layout, keine TODOs, kein auskommentierter Code und zudem sollte klar ersichtlich sein, was fertig ist und was nicht. Wenn möglich mit Grenzen entlang der Übersetzungseinheiten.

## Coding Guidelines

Wir fahren auf der rechten Strassenseite. Es gibt Länger, da fährt man auf der linken Strassenseite.

Diese ist eine Konvention, die man einhalten muss, weil man nicht alleine unterwegs ist.

Das Team muss sich daher immer auf Codier-Richtlinien einigen. Kein Projekt ohne Codier-Richtlinien. Regelmässig dies mit Reviews überprüfen.

## Don't Repeat Yourself (DRY)

```
$form->get('fieldorder')->setValue($fieldOrder); $form->get('fieldorder')->setValue($fieldOrder);

if ($request->isGet()) {
    $this->viewModel->setVariables(array(
        'form'  => $form,
        'hash'  => $hashEntity->getId(),
        'id'    => $insertAtPosition,
    ));
    return $this->viewModel;
} elseif ($request->isPost()) {
    $field = new Field();
}

if ($request->isGet()) {
    $this->viewModel->setVariables(array(
        'form'  => $form,
        'hash'  => $hashEntity->getId(),
        'id'    => $insertAtPosition,
    ));
    return $this->viewModel;
} elseif ($request->isPost()) {
    $field = new Field();
```

Keine Duplikation von Code und von Informationen in der Dokumentation. Insbesondere ist Copy/Paste schlecht. Der Code bläht dadurch auf → Mehr Code == mehr Arbeit, mehr Fehler.

## Wartungs-Alptraum

Programmierer kopieren gerne mal was und vergessen, ein Detail zu ändern (Subtiler Bug). Manche Programmierer kopieren etwas, ohne es vollständig zu verstehen. Aber es funktioniert von Beginn an. Wenn es der eigene Code ist und du musst was ändern, dann musst du vermutlich alle Instanzen des kopierten Codes ändern. Wenn es nicht dein eigener Code ist und du musst was ändern, dann musst du alle Instanzen des kopierten Codes ändern. Zudem stellt sich die Frage welche ist die neuste Version des Codes bzw. die beste Version.

- Design-Diagramme (Übersichten) max. A3 (wegen Drucker und Beamer)
- Code-Zeilen: max. 120 Zeichen lang (unter anderem wegen Beamer)
- Max. Einrückungs-/Verschachtelungstiefe: 5
- Methoden (Funktionen): max. 1 Bildschirm, d.h. max. ca. 30 Zeilen
- Klassen: max. ca. 300 Zeilen
- Packages; keine Angabe, da abhängig von Funktion/Zuständigkeit
- Schichten: max. 7
- Tiers: max. 4 (Performance, Security)

Klar gibt es immer auch Ausnahmen.

## No Errors, No Warnings

Seien Sie hart: Compiler-Warnungen dürfen nur ausnahmsweise ignoriert werden. Jeder build muss ohne Compiler-Fehler und ohne Compiler- Warnings durchlaufen!

Falls Sie z.B. die „serializable UID“ Warnung von Java ignorieren wollen - und das kann in vielen Projekten sinnvoll sein - dann konfigurieren Sie die Compiler-Switches so. Aber sonst dürfen keine Warnungen auftauchen. Und wenn, dann muss man sie ernst nehmen.

Allzuschnell gewöhnt man sich an das Auftauchen von Warnungen und nimmt sie nicht mehr ernst. Und dann geht eines Tages eine wichtige Warnung unter - weil es immer wieder Warnungen gibt, und die schaut man sich ja nicht alle an, man ist ja nicht doof...

## Keep it Simple

*'There are two ways of constructing a software design:  
One way is to make it so simple that there are obviously no deficiencies,  
and the other way is to make it so complicated that there are no obvious  
deficiencies.  
The first method is far more difficult.'*

– C.A.R. Hoare

## Positive Bedingungen

### Schneller Ausstieg

Sogenannte „guard clauses“ am Anfang können helfen, die Logik klarer zu machen und das Einrücken weniger Tief zu halten.

```
public boolean Contains(String str, String substr) {
    if (str == null || substr == null) return false;
    if (substr.equals("")) return true;

    // ...
}
```

Ein schneller Ausstieg ist dann OK, wenn die Entscheidungs-Mengen sich genau ergänzen, daher wenn das Gegenteil der Entscheidung genau die ergänzende Menge beschreibt.

```
if (x > 7) then
    machA();
else
    tueB();
        if (x <= 7) then
            tueB();
        else
            machA();
```

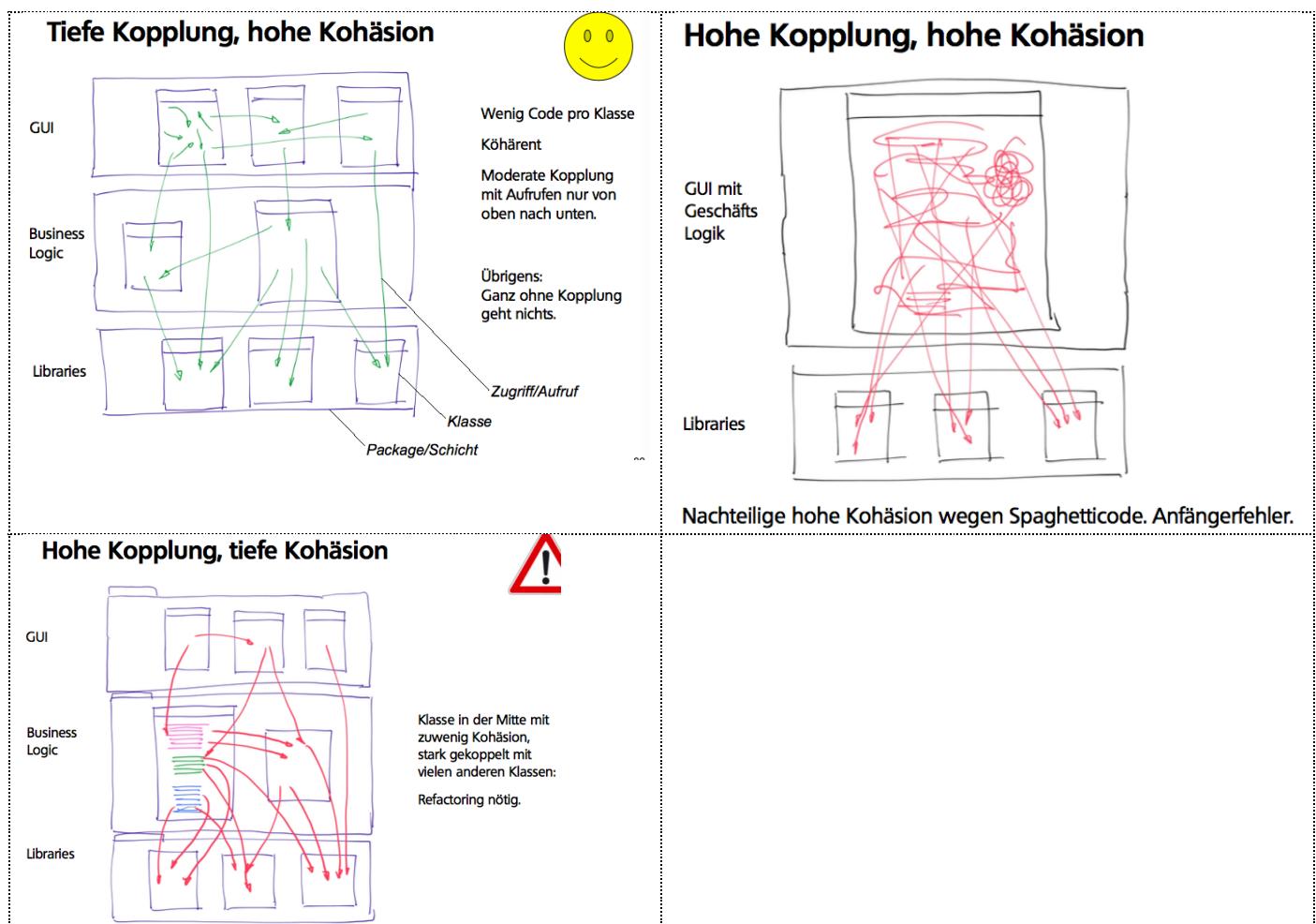
## Kopplung und Kohäsion

**Kopplung:** Aufrufe von einer Klasse zur anderen (z.B. „feature envy“), von einem Package zum anderen

**Kohäsion:** Zusammenhalt innerhalb einer Klasse „warum sind wir zusammen, was ist der gemeinsame Zweck?“ weniger häufig/wichtig: Zusammenhalt innerhalb eines Packages

### Regeln

- Ganz ohne Koppelung geht es nicht, aber Koppelung (=Abhängigkeiten) sollte minimiert werden.
- Kohäsion soll gegeben sein, sonst gehören die Dinge nicht in eine Klasse, in ein Package.



## Single Responsibility Principle

Auch bekannt unter Assignment of Responsibilities, Responsibility Driven Design oder Separation of Concerns.

Zuordnung von Verantwortlichkeiten nach der Regel „Immer nur für etwas verantwortlich sein, nicht für mehrere Dinge“.

### Beispiele für Zuständigkeiten

**Ausgabe** (GUI, z.B. Warnungs-Popup, Datenreihe als Kurve darstellen, Ausgabe- Fenster öffnen...)

**Formatierung der Ausgaben** (z.B. länderspezifische Sprach-Strings, HTML/XML- Tags hinzufügen...)

**GUI-Logik** (welcher Knopf/Menu-Eintrag ist enabled/disabled, was ist der nächste Schritt/Dialog)

**Benutzer-Eingaben** ab Tastatur, Scanner oder GUI lesen (reine HW-Kapselung, Daten-Weiterleitung)

**Eingabe-Validierung** nach Lesen (z.B. Parsing, gültige Wertebereiche prüfen "muss eine Zahl zwischen 0.01 und 100 sein" ...)

**Programmlogik/Ablauflogik**, Business Logic (Schritte, wann passiert was, wie oft wird wiederholt, wer hat welche Rechte, wann wird gelöscht...)

**Algorithmen**, Mathematische Berechnungen (z.B. Sortieren, Matrizenrechnung, Numerik, Trigonometrie...)

**Kapselung von Hardware** (z.B. Ansteuerung eines IR-Sensors)

**Persistenz** (dauerhafte Speicherung von Werten, z.B. in DB, als XML-Datei, ini-Datei, serialisiertes Objekt...)

### Nachteile, wenn diese Regel verletzt ist

Die Klasse wird zu gross. Das hat Folgen:

- Mehrere Entwickler: häufige Kollisionen bei SVN/git check-in.
- Schwer zu testen: Unit Tests werden komplex oder fast unmöglich.
- Fehlerbehebung führt leichter zu Nebeneffekten
- Klasse ist ständig in Bearbeitung: → sehr häufig Regressionstests nötig.
- Kopplung zu anderen Klassen steigt an.

## Program to the Interface not to an Implementation

Typisch in Java List<MyDatastrucutre> zu Beispiel List<Person> ohne zu spezifizieren, welcher Typ List es ist. Damit ist die Implementation entkoppelt vom Aufrufenden. Dependency Inversion Principle.

Es gibt Fälle, wo es überflüssig ist, zu einer Implementation noch ein Interface zu machen, vor allem dann, wenn die Implementation wahrscheinlich nie ändern wird -- aber normalerweise ist "program towards an interface" eine sehr gute Taktik.

## Isolate What Changes

Eigentlich besser nicht fragen „was ändert sich“, sondern, „was bleibt gleich?“ und dann das in einem Interface modellieren.

**Beispiel** Datenspeicherung eines Gruppenkalenders: File lokal, DB lokal, DB remote, Cloud service via REST ... Frage: „was brauche ich denn an Funktionen?“

**Antwort:**

- „Gib mir alle Einträge von Mitarbeiter X in Woche Y“
- „Gib mir alle Einträge aller Mitarbeiter in Woche Y“
- „Speichere (neu/geändert) Eintrag Z“
- „Suche einen freien Slot von Länge A in Zeitraum B-C für MA-Set S“
- „CRUD Mitarbeiter“

(und dann das Ganze noch für die Monatsansicht)

**Smart Data Structures**

**“Smart data structures and dumb code works a lot better than the other way around.”**

— Eric Raymond, *The Cathedral and the Bazaar*, Chapter 5

**Datenstrukturen vor Code**

**“I’m a huge proponent of designing your code around the data, rather than the other way around, and I think it’s one of the reasons git has been fairly successful [...] I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important”**

– Linus Torvalds

**Warum das Datenmodell so wichtig ist**

In einer Schweizer Grossbank gilt schon lange die Faustregel: **2:5:20**

2 Jahre hält das GUI, 5 Jahre hält die Geschäftslogik und 20 Jahre halten die Daten.

Änderungen in der zentralen Datenhaltung (in der Datenstruktur) führen immer zu sehr weit gestreuten Änderungen im Code, von der Geschäftslogik bis GUI. Alle stöhnen. Zu Recht.

**Programmiere für Menschen... nicht für den Compiler**

- Sich mit „Hurra, es läuft!“ zufrieden zu geben ist ein Anfängerfehler.
- Code wird häufiger gelesen als geschrieben.
- Oft sollte Code erweitert werden können, wird aber weggeworfen, weil „das Teil ist nicht wartbar“.
- Schreiben Sie Code so, wie sie ihn selbst anzutreffen wünschen.
- Ein Review in der Gruppe zeigt, ob der Code gut lesbar ist.

**Gute Namen**

Aussprechbar, mit Bedeutung, Konsistent, Methoden mit Verb, boolean mit „is“, Kurze Namen nur bei kleinem Scope, Nicht abkürzen, wenn man nur 3 Buchstaben spart, Keine Präfixe, Nur 7 bit ASCII Zeichen.

<b>groupID</b>	<b>ist besser als</b>	<b>grpID</b>
<b>nameLength</b>	<b>ist besser als</b>	<b>namLn</b>
<b>powersOfTwo</b>	<b>ist besser als</b>	<b>pwrsOf2</b>
<b>resetPrinter</b>	<b>ist besser als</b>	<b>rstprt</b>

Eine Zahl kann leicht mit einem Buchstaben verwechselt werden: 0 und o, 1 und l, 2 und Z, S und 5, 8 und B. Wenn Sie in einem Programm drei Pointer brauchen p1, p2, p3 sollte die dann am besten mit previosPtr, currentPtr und nextPtr heißen, wenn das ihre Bedeutung ist.

```
String text, text1, text2, text3, text4;
```

## Was die wirkliche Bedeutung war:

```
String titleText;
String pageHeaderText;
String bodyText;
String pageFooterText;
String endOfReportText;
```

Keinen anonyme Konstanten

## Sondern:

### Nie so was:

```
for (i = 1; i < 27; i++)
    for (i = 1; i < MAX_OPEN_FILES; i++)
```

```
const int MAX_OPEN_FILES = 27;
```

## Nützliche Kommentare

Der Methodename sollte sagen, was gemacht wird. Die Statements im Code zeigen, wie es gemacht wurde. Ein (eventueller) Kommentar sollte erklären warum genau so.

## Mit Vorsicht und Umsicht kommentieren

Ein Kommentar sollte nie das Offensichtliche kommentieren („die Variable name erhält den Namen der Person“). Kommentare sollten selten geschrieben werden. Besser die Bedeutung in den Namen klar machen. Die Java Doc Kommentare sind nur nützlich für lower-Level Funktionen (meist in einer Bibliothek), die man sinnvoll aufrufen können sollte, ohne den Code zu lesen, nur anhand der JavaDoc.

```
public class Repository {

    /** The Constant repositoryXml. */
    private static final String repositoryXml = "repository.xml";

    private static final int MAXDISPLAYNAMELENGTH = 20;
    private static final int MAXDISPLAYADDRESSLENGTH = 50;

    /** The foldername. */
    private String name, address, foldername;

    /** The web. */
    private boolean web;

    /** The plugins. */
    private ArrayList<Plugin> plugins = new ArrayList<Plugin>();
}
```

Aufzupassen ist auch mit irreführenden Kommentaren. Dies kann durch nicht aktualisierte Kommentare auftreten. Nun ein Beispiel eines nützlichen Kommentars.

```
const int OK_BUTTON = 1; // the creation order of the buttons is
const int CANCEL_BUTTON = 2; // important!! Check it in the "Layout"
                           // tool with "Edit/Creation Order...".
                           // Must be "OK" ... "Cancel" ... , then
                           // all others
```

Allen grossen Handwerks-Künsten ist gemeinsam: das Wesentliche kann mit wenigen Aphorismen (ein einzelner Gedanke, der aus nur einem Satz oder wenigen Sätzen selbstständig bestehen kann) ausgedrückt werden. Aber es dauert Jahre, bis man gelernt hat, diese Aphorismen in die Praxis umzusetzen. Die Meisterschaft darin zu erreichen braucht Jahrzehnte.

Das Design ist keine Wissenschaft – und wird es nie sein. Es wird auch nie ein Betty Bossi Rezept sein, das jeder nachkochen kann. Die Methode macht das Design nicht aus – der Designer macht's.

# Design Patterns

## Program to an Interface

Beispiel eines Enten-Simulator.

**Beispiel Enten-Simulator (aus Head First Design Patterns)**  
Abstrakte Klasse, da nur Teile neu implementiert sind.

```

classDiagram
    class Duck {
        quack()
        swim()
        display()
        fly()
        // OTHER duck-like methods...
    }
    class MallardDuck {
        display() {
            // looks like a mallard
        }
    }
    class RedheadDuck {
        display() {
            // looks like a redhead
        }
    }
    MallardDuck --> Duck
    RedheadDuck --> Duck
    Note over Duck: Lots of other types of ducks inherit from the Duck class.
  
```

**Jetzt sollen Enten auch fliegen**

```

classDiagram
    class Duck {
        quack()
        swim()
        display()
        fly()
        // OTHER duck-like methods...
    }
    class MallardDuck {
        display() {
            // looks like a mallard
        }
    }
    class RedheadDuck {
        display() {
            // looks like a redhead
        }
    }
    Note over Duck: All subclasses inherit fly()
    Note over Duck: What Joe added.
    Note over Duck: Other Duck types...
  
```

## Die wollen auch noch Gummienten

```

classDiagram
    class Duck {
        quack()
        swim()
        display()
        fly()
        // OTHER duck-like methods...
    }
    class MallardDuck {
        display() {
            // looks like a mallard
        }
    }
    class RedheadDuck {
        display() {
            // looks like a redhead
        }
    }
    class RubberDuck {
        quack() {
            // overridden to Squeak
        }
        display() {
            // looks like a rubberduck
        }
    }
    MallardDuck --> Duck
    RedheadDuck --> Duck
    RubberDuck --> Duck
  
```

By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.

Aber die Quaken nicht, die quietschen, und fliegen tun sie auch nicht.

## Wie wär's mit einem Interface?

```

classDiagram
    interface Flyable {
        fly()
    }
    interface Quackable {
        quack()
    }
    class Duck {
        swim()
        display()
        // OTHER duck-like methods...
    }
    class MallardDuck {
        fly()
        quack()
    }
    class RedheadDuck {
        fly()
        quack()
    }
    class RubberDuck {
        quack()
    }
    class DecoyDuck {
        display()
    }
    MallardDuck --> Flyable
    MallardDuck --> Quackable
    RedheadDuck --> Flyable
    RedheadDuck --> Quackable
    RubberDuck --> Quackable
    DecoyDuck --> Quackable
  
```

(Gummiente)      (Lockvogel-Ente)

## Neu auch noch eine Holzente

I could always just override the fly() method in rubber duck, the way I am with the quack() method...

RubberDuck

```

quack() {
    // squeak
}
display() {
    // rubber duck
}
fly() {
    // override to do nothing
}
  
```

But then what happens when we add wooden decoy ducks to the program? They aren't supposed to fly or quack...

DecoyDuck

```

quack() {
    // override to do nothing
}
display() {
    // decoy duck
}
fly() {
    // override to do nothing
}
  
```

Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

## Verhalten ausgelagert

```

classDiagram
    interface FlyBehavior {
        fly()
    }
    interface QuackBehavior {
        quack()
    }
    class FlyWithWings {
        fly() {
            // implements duck flying
        }
    }
    class FlyNoWay {
        fly() {
            // do nothing - can't fly!
        }
    }
    class Quack {
        quack() {
            // implements duck quacking
        }
    }
    class Squeak {
        quack() {
            // rubber ducky squeak
        }
    }
    class MuteQuack {
        quack() {
            // do nothing - can't quack!
        }
    }
    FlyWithWings --> FlyBehavior
    FlyNoWay --> FlyBehavior
    Quack --> QuackBehavior
    Squeak --> QuackBehavior
    MuteQuack --> QuackBehavior
  
```

And here's the implementation for flying for all ducks that have wings.

Quacks that really quack.

Quacks that squeak.

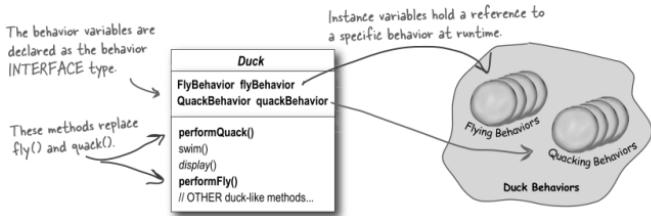
Quacks that make no sound at all.

Urs Forrer, HSR Rapperswil

Seite 63 von 120

26. Dezember 2017

## Die bessere Enten-Klasse

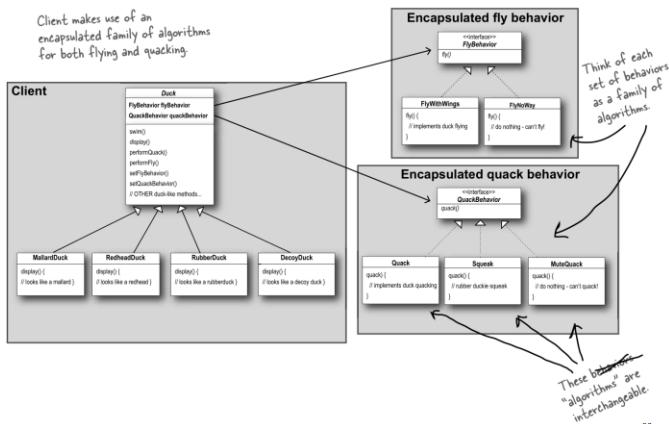


## Zwei Setter zur Verhaltensänderung

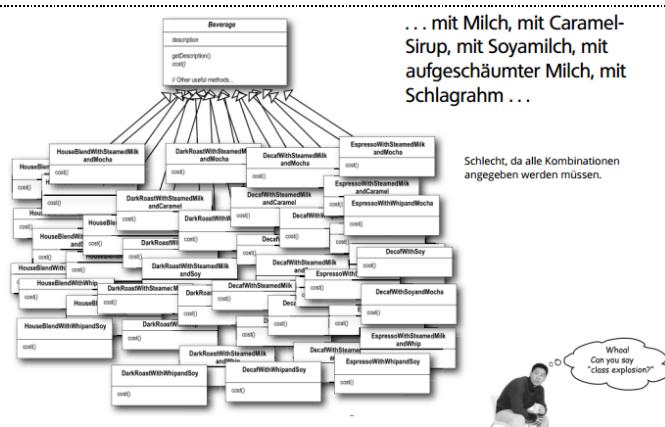
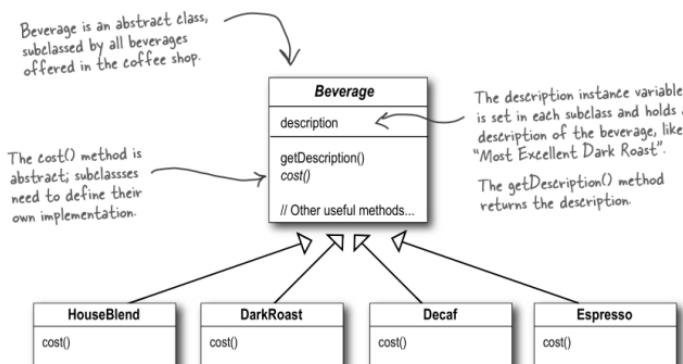
```
public void setFlyBehavior(FlyBehavior fb) {  
    flyBehavior = fb;  
}  
  
public void setQuackBehavior(QuackBehavior qb) {  
    quackBehavior = qb;  
}
```

... und los geht's!

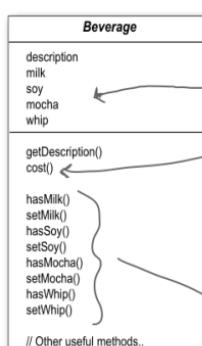
## Zusammenspiel



# Starbuzz Coffee



## Alternative: Instanz-Variablen



New boolean values for each condiment.

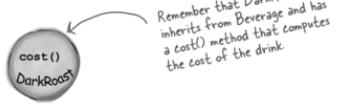
Now we'll implement `cost()` in `Beverage` (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override `cost()`, but they will also invoke the super version so that they can calculate the total cost of the basic `HouseSpecial` plus the costs of the added condiments.

These get and set the boolean  
values for the condiments.

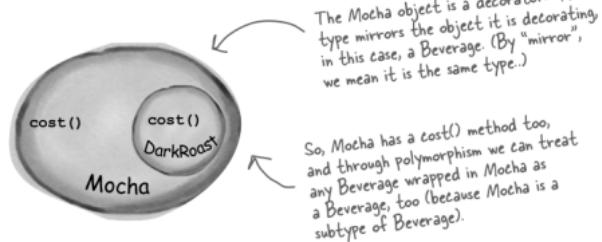
- Komplizierte Berechnung
  - Setter und Getter müssen erweiterte werden.

In was für Probleme laufen wir mit dieser Lösung?

① We start with our DarkRoast object.



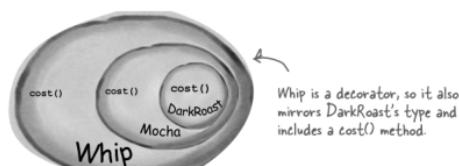
② The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.



## Decorator um Decorator herum

Dinge über die anderen Verpacken

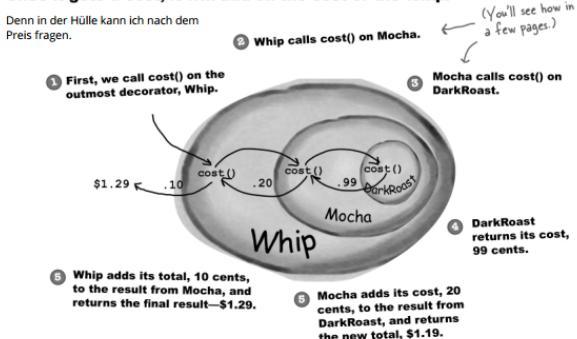
③ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



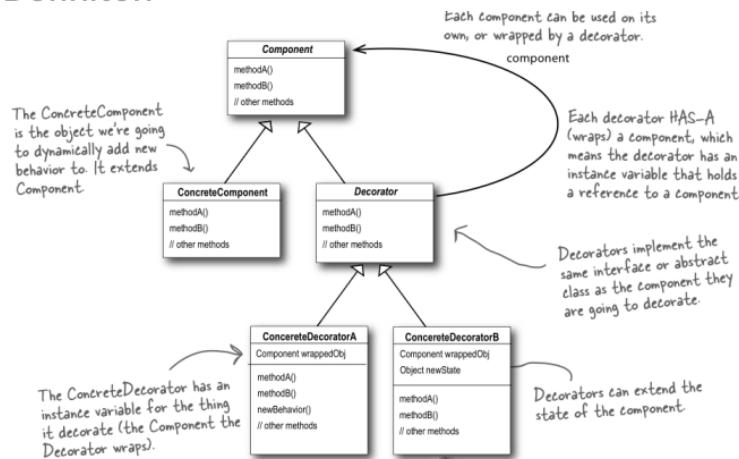
So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

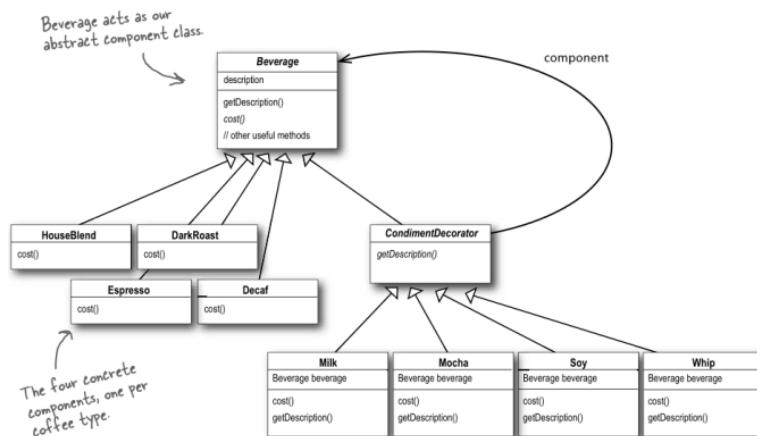
## Kostenrechnung

④ Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.



## Definition





## Code

```

public class Espresso extends Beverage {
    public Espresso() {
        description = "Espresso";
    }
    public double cost() {
        return 1.99;
    }
}
  
```

Das machen wir vier mal für alle vier Basisgetränke

```

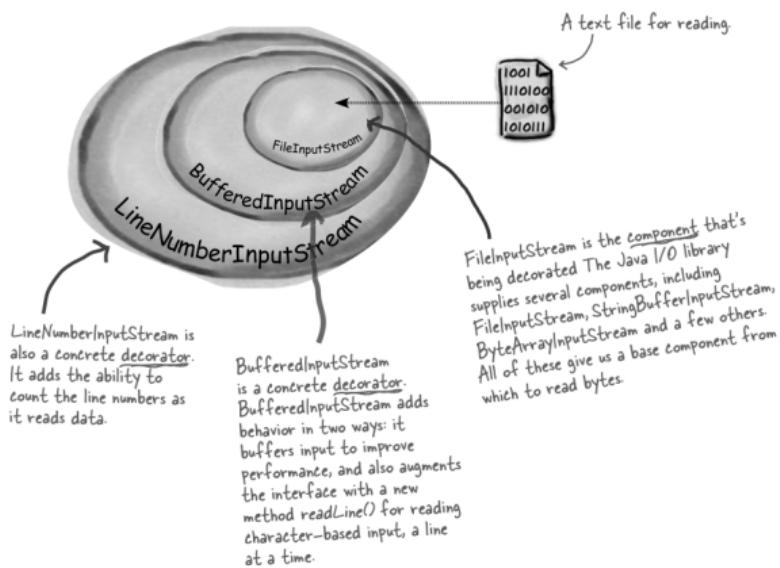
public class StarbuzzCoffee {
    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription() + " $" + beverage.cost());
        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription() + " $" + beverage2.cost());
        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription() + " $" + beverage3.cost());
    }
}
  
```

```

public class Mocha extends CondimentDecorator {
    Beverage beverage;
    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }
    public double cost() {
        return .20 + beverage.cost();
    }
}
  
```

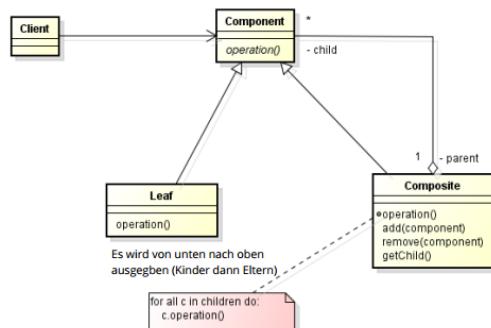
We want our description to not only include the beverage - say "Dark Roast" - but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

## Real World Decorators - java.io



## Problem

Wir wollen «Einheiten» immer gleich behandeln können, egal ob es zusammengesetzte Einheiten sind oder einfache Einheiten. Oft benutzt in Baum-Strukturen, denn dann können wir innere Knoten gleich behandeln wie Endknoten.

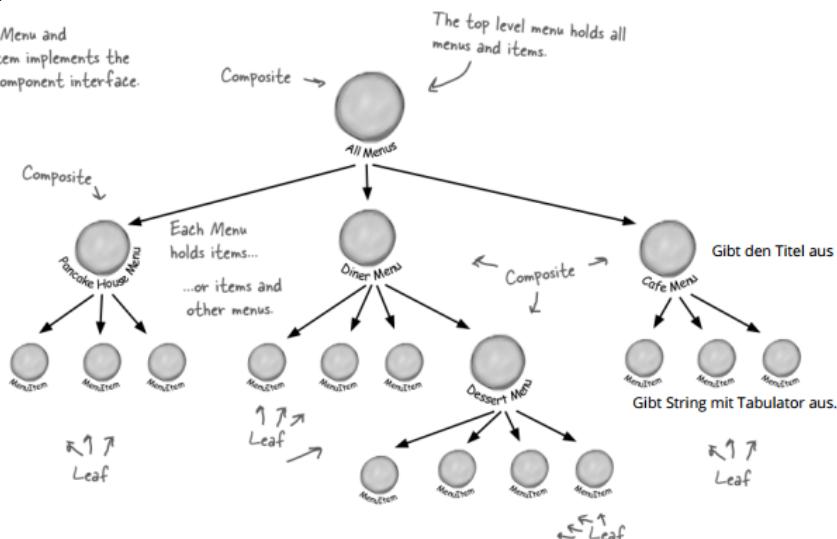


Mit dieser Struktur können wir einfach eine Schleife über alle Components machen, ohne uns darum zu kümmern, ob die Component zusammengesetzt ist oder nicht.

Weitere Beispiele sind JavaSwing oder Menus mit Untermenüs.  
Zudem Mitarbeiter und Vorgesetzte.

## Beispiel

Every Menu and MenuItem implements the **MenuComponent** interface.



## Wie ist ein Design Pattern definiert?

Der Ursprung ist in der Architektur bei Christopher Alexander. Bei einem Pattern sind folgende Dinge definiert:

- Beschreibung (UML, Naming) mit Struktur und Funktionalität
- Einsatzbereich / Problem
- Abgrenzungen, Diskussion, alternative Lösungen

Eine Vollbeschreibung, wie im Buch, beinhaltet folgende Teile:

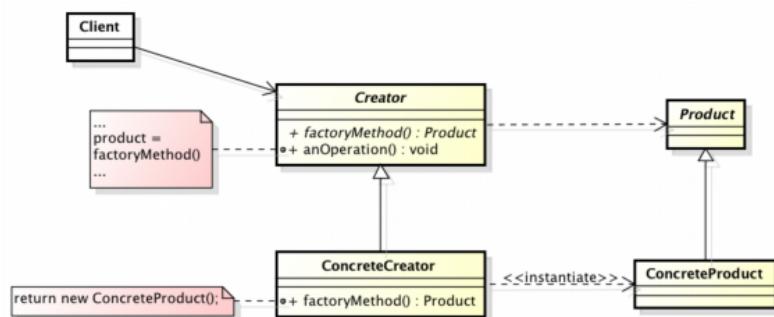
Intent	Participants	Implementation / Sample Code
Motivation	Collaborations	Known Uses
Applicability	Consequences	Related Patterns
Structure		

**Problem**

Wer ist verantwortlich Objekte zu erzeugen, wenn spezielle Bedingungen gelten, wie komplexe Erzeugungslogik?

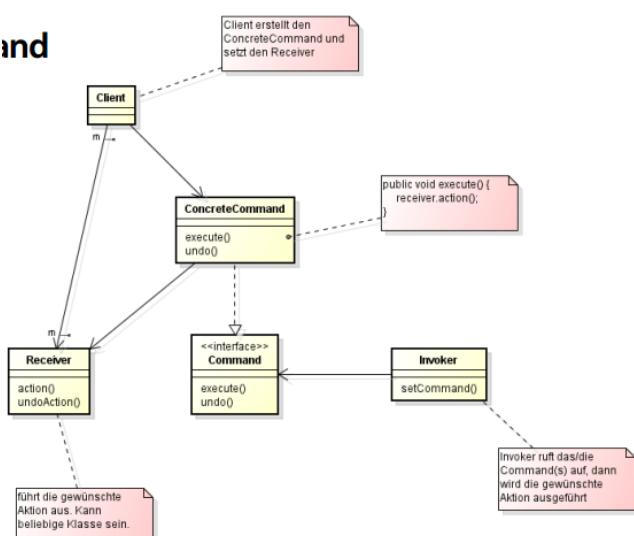
**Lösung**

Kapsle Objekterzeugung in separater Factory Klasse. Entscheide dort anhand von Parametern oder Umgebungsvariablen, welche konkrete Klasse instanziert werden soll. Factory erzeugt die Klasse zu einem bestimmten Interface. Typische Namen der erzeugenden Methode sind `createSomething()`, `makeSomething()` oder `new Something()`.

**Struktur**

Definiere eine Factory Methode als abstrakte Methode in Basisklasse, um Objekte zu erzeugen, aber Sub-Klassen entscheiden, welche Klasse instanziert wird. Beispiel: Factory Methode erzeugt korrektes Sprachbundle, abhängig von Browser-Sprach-Einstellungen des Aufrufendens. Es gibt die folgenden

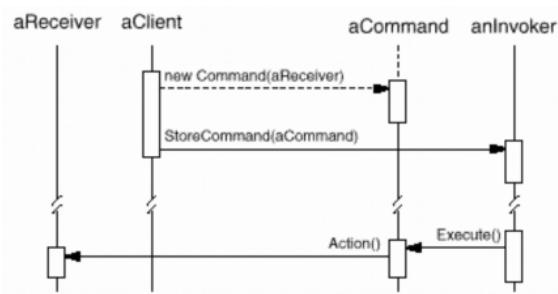
Fabriken: Concrete Factory, Abstract Factory und Factory Method. Behandlung und Diskussion in SE2.

**Command Pattern****and****Problem**

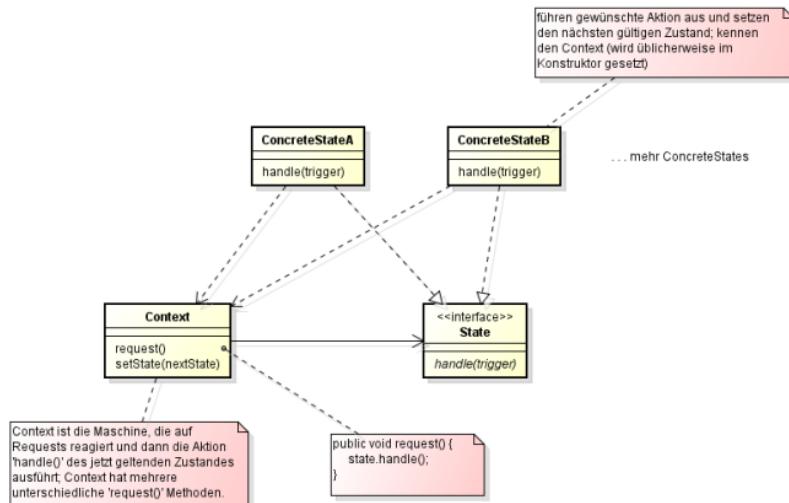
Wie kann man die Ausführung einer Aktion von der Aktivierung der Aktion trennen (Entkopplung von Aufrufenden und Ausführenden) und auch die Aktion ggf. später durchführen, rückgängig machen oder protokollieren?

**Lösung**

Kapsle die Aktion als Command Objekt. Gib allen Command Objekten eine gemeinsame Schnittstelle, meist heisst diese `execute()`. Statt `action()` direkt aufzurufen, erzeuge das entsprechende Command Objekt, auf dem dann die Aktion ausgeführt wird.

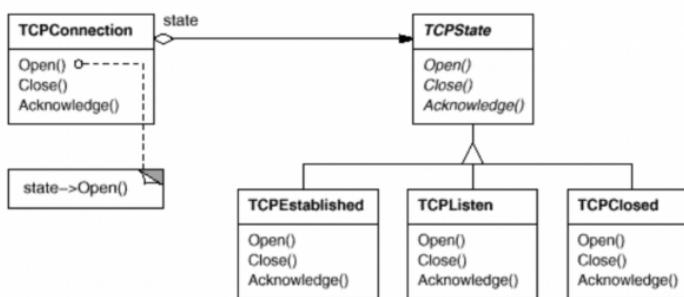
**Interaktion**

Anwendung zur Entkopplung, für Callbacks, Für Queue-Mechanismen oder für Undo und Change-log Operationen.



Klassendiagramm wie bei Strategy.

### Beispiel aus GoF



## Template Method Pattern

### Problem

Die Struktur eines Algorithmus ist die gleiche in verschiedenen Unterklassen, aber die Verarbeitungsdetails variieren von Unterklasse zu Unterklasse.

### Lösung

Definiere die Struktur des Algorithmus (Skelett) in der Basisklasse. Diese Template Method ruft andere Methoden für die variierenden Details auf. Diese anderen Methoden (Hook methods) werden in den Unterklassen entsprechend überschrieben.

### Beispiel Game

```

/**
 * An abstract class that is common to several games in
 * which players play against the others, but only one is
 * playing at a given time.
 */
abstract class Game {
    /* Hook methods. Concrete implementation may differ in each subclass*/
    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    /* A template method : */
    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}
  
```

```

class Monopoly extends Game {
    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Initialize players
        // Initialize money
    }
    void makePlay(int player) {
        // Process one turn of player
    }
    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }
    void printWinner() {
        // Display who won
    }
    /* Specific declarations for the Monopoly game. */
    ...
}

class Chess: Game {
    private var gameEnded: Bool = false

    init() {
    }

    func initializeGame() {
        println("Initialize 2 players.")
        println("Put the pieces on the board.")
    }

    func makePlay() {
        println("Process one turn of Chess player.")
    }

    func endOfGame() -> Bool {
        println("Has Checkmate or Stalemate been reached? \(gameEnded)")
        return gameEnded
    }

    func printWinner() {
        println("Display who won the Chess.")
    }
    /* Specific declarations for the chess game. */
}

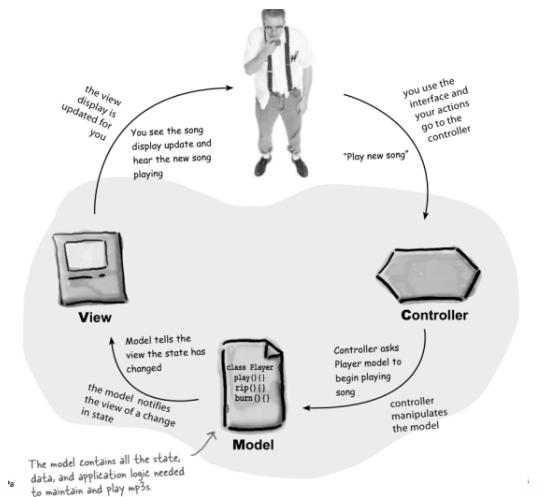
```

## The Hollywood Principle / Inversion of Control

“Don’t call us, we’ll call you”. Das Framework übernimmt meiste Arbeit und stellt sogenannte “Hook Methods” zur Verfügung, die man überschreiben kann (manchmal muss). Die «Einhak-Methoden» sind oft mit einem voreingestellten Standardverhalten versehen, z.B. «male den Hintergrund des Panels weiß».

## MVC Pattern (Model-View-Controller)

Am Beispiel eines MP3-Players.



### View

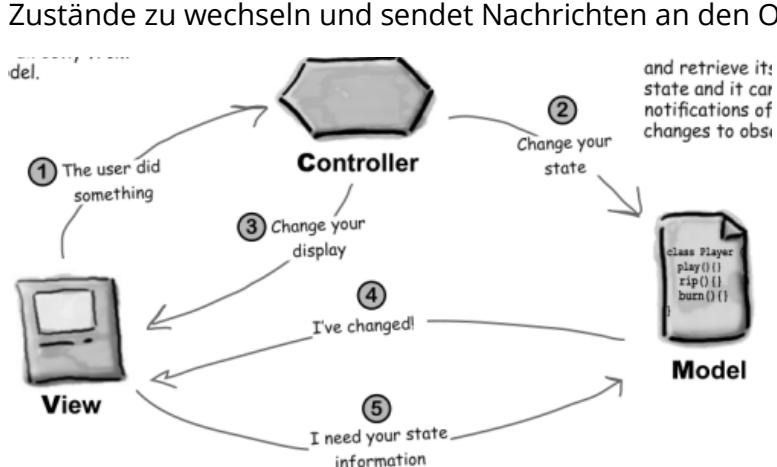
Gibt eine Präsentation des Models. Die View bekommt die Daten und Zustände, welche es braucht normalerweise direkt vom Model

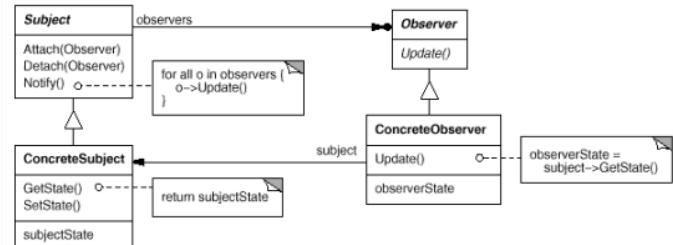
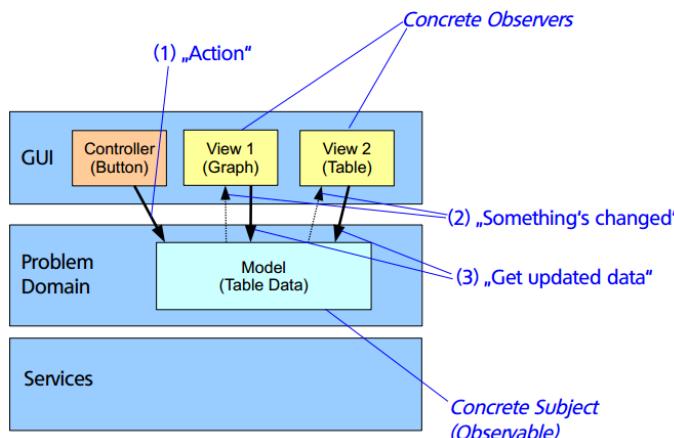
### Controller

Nimmt den Benutzerinput und findet heraus, was es für das Model bedeutet.

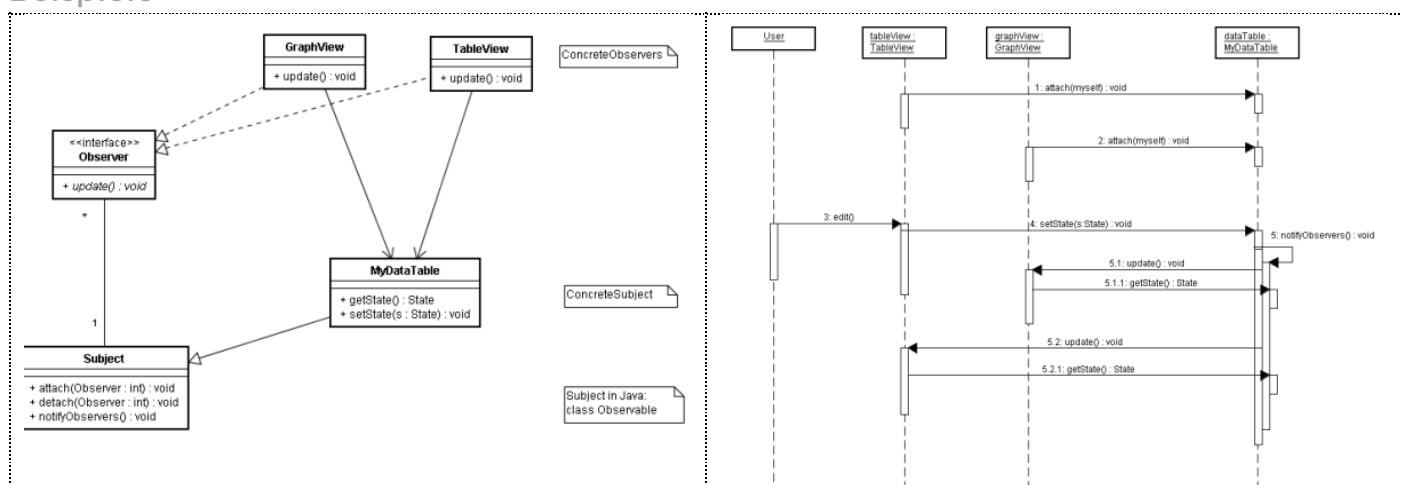
### Model

Das Model hält alle Daten und die Applikationslogik. Das Model ist gegenüber dem Controller und der View ahnungslos. Es bietet zudem eine Schnittstelle um die Zustände zu wechseln und sendet Nachrichten an den Observer über geändert Zustände.

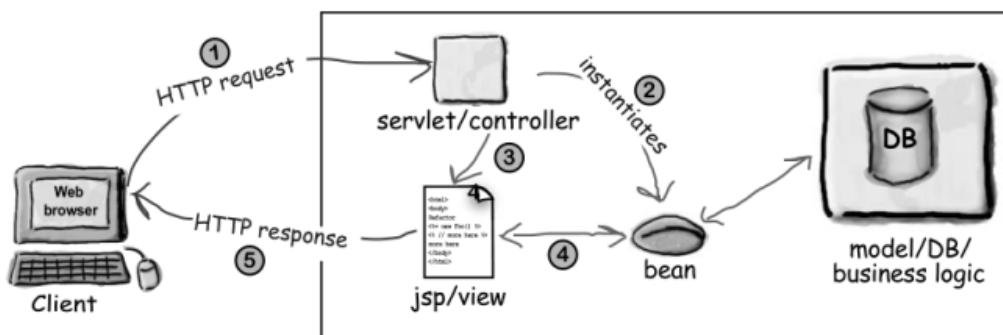




## Beispiele

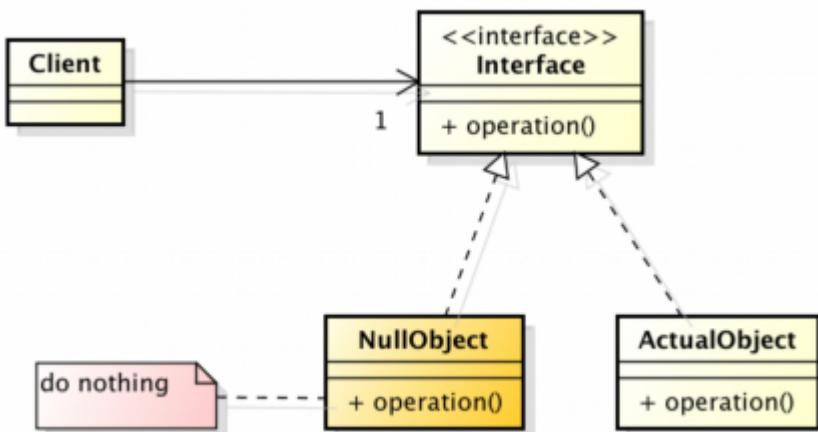


## MVC im Web-Zeitalter



Stelle etwas für nichts zur Verfügung.

## Übersicht



Pattern Name	GoF	Head First	Larman (+) = zu knapp	SE1 Wissen	SE1 Vorlesung
Facade (Facade Controller in Larman)	*	x	+	ja	
Strategy	*	x	+	ja	
Factory Method (~ Concrete Factory in Larman)	*	x	(+)	ja	✓
Adapter	*	x	+	ja	
Proxy (virtual, remote, caching...)	*	x			
Observer	*	x	+	ja	
Composite	*	x	(+)	ja	✓
Singleton	*	x	+	ja	
Use Case/Session/Application Controller			+	ja	
Abstract Factory	*	x			
Command	*	x		ja	✓
State	*	x		ja	✓
Template Method	*	x		ja	✓
Decorator	*	x		ja	✓
Iterator	*	x			
Null Object		x	(+)	ja	✓
MVC		x	(+)	ja	✓

# Software Architektur 1 – Layers, Tiers & Partitionen

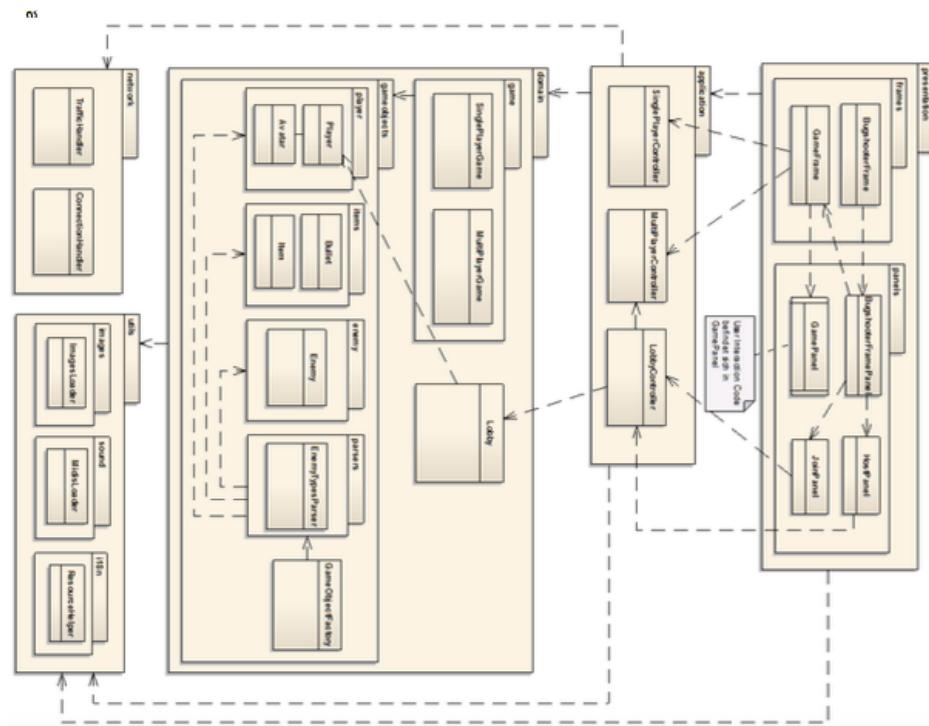
## Packages

Hier mit den wichtigsten Klassen eingezeichnet (meist Interface-Klassen nach oben). Beispiel: EPJ Projekt BugShooter.

## 4 (5) Layers

### 24 Klassen (von 125)

Man sieht: das ist schon grenzwertig für einen Beamer.



## Aufteilungskriterien

### Aufteilungskriterien

Kohäsion

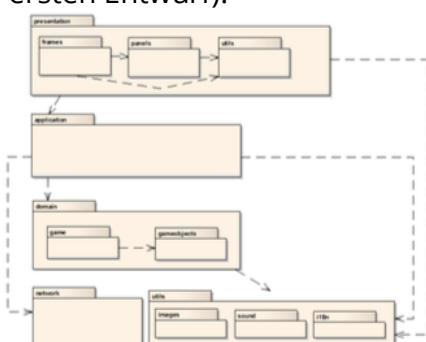
entsteht durch

Single Responsibility

Zusammengehörigkeit

Zuständigkeit / Arbeitsteilung

Im obigen Beispiel des EPJ Projektes (auf der obersten Abstraktionsebene, d.h. Schichten/Packages im ersten Entwurf):



- Views, UI
- Application Controllers
- GameObjects (Domain)
- Network
- Utilities, Helper Classes

## Zuständigkeiten

Visuelle Darstellung

(Use Case) Controllers

Business (Game) Logic

Parser (Entscheider)

Domain Objects  
(Datenmodellierung)

Speicherung

Hilfsklassen

Netzwerk

### Beispiele für Zuständigkeiten

**Ausgabe** (GUI, z.B. Warnungs-Popup, Datenreihe als Kurve darstellen, Ausgabe- Fenster öffnen...)

**Formatierung der Ausgaben** (z.B. länderspezifische Sprach-Strings, HTML/XML- Tags hinzufügen...)

**GUI-Logik** (welcher Knopf/Menu-Eintrag ist enabled/disabled, was ist der nächste Schritt/Dialog)

**Benutzer-Eingaben** ab Tastatur, Scanner oder GUI lesen (reine HW-Kapselung, Daten-Weiterleitung)

**Eingabe-Validierung** nach Lesen (z.B. Parsing, gültige Wertebereiche prüfen "muss eine Zahl zwischen 0.01 und 100 sein"...)

**Programmlogik/Ablauflogik**, Business Logic (Schritte, wann passiert was, wie oft wird wiederholt, wer hat welche Rechte, wann wird gelöscht...)

**Algorithmen**, Mathematische Berechnungen (z.B. Sortieren, Matrizenrechnung, Numerik, Trigonometrie...)

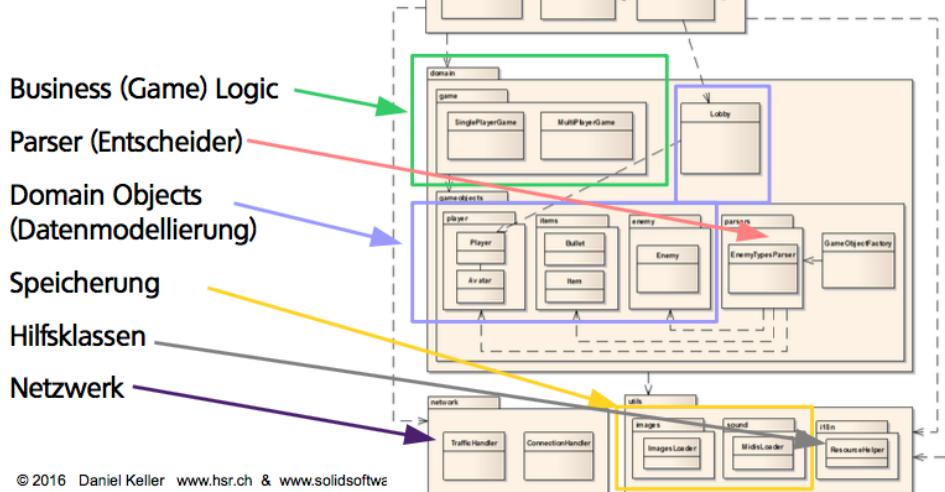
**Kapselung von Hardware** (z.B. Ansteuerung eines IR-Sensors)

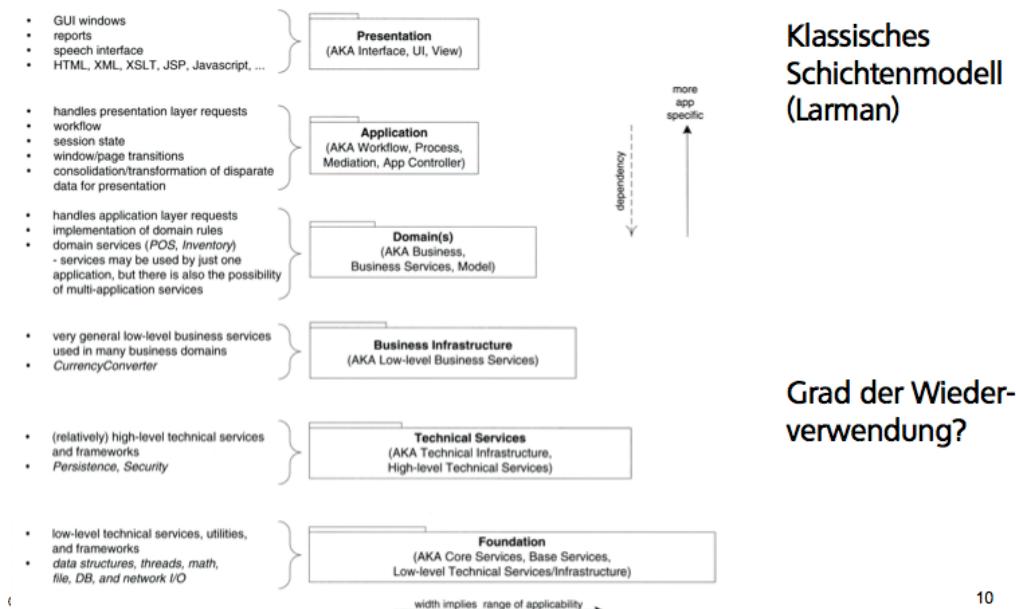
**Persistenz** (dauerhafte Speicherung von Werten, z.B. in DB, als XML-Datei, ini-Datei, serialisiertes Objekt...)

Wenn die Zuständigkeiten gut (Single Responsibility) zugeteilt sind, ist die Kohäsion – wie gewünscht – hoch.

## Ziele

- **Hohe Kohäsion** (Guter Zusammenhalt innerhalb einer Klasse)
- Und **Tiefe Kopplung** (Minimierte Abhängigkeiten von anderen Klassen)





## Kunde und Dienstleistung

Kunde

```
import services.DateHelper;
```

```
...
```

```
workDays = DateHelper.calculateDiffInWorkingDays( beginDate, endDate );
```

```
...
```

```
public class DateHelper {
```

```
...
```

```
    public static int calculateDiffInWorkingDays( Date first, Date second ) {
```

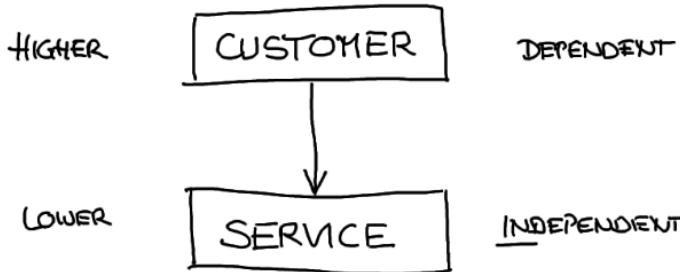
```
        ...
```

```
        return workingDays;
```

```
}
```

Dienstleistun

# Abhangig und unabhangig



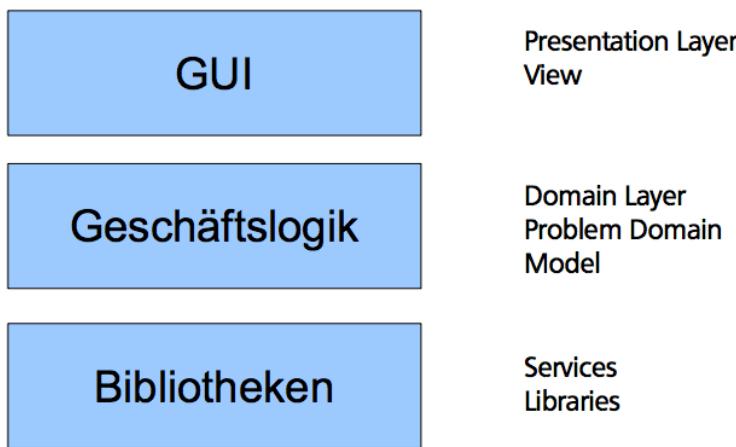
Schichten sind **asymmetrisch!** Abhangigkeiten beachten.

# Oben und unten

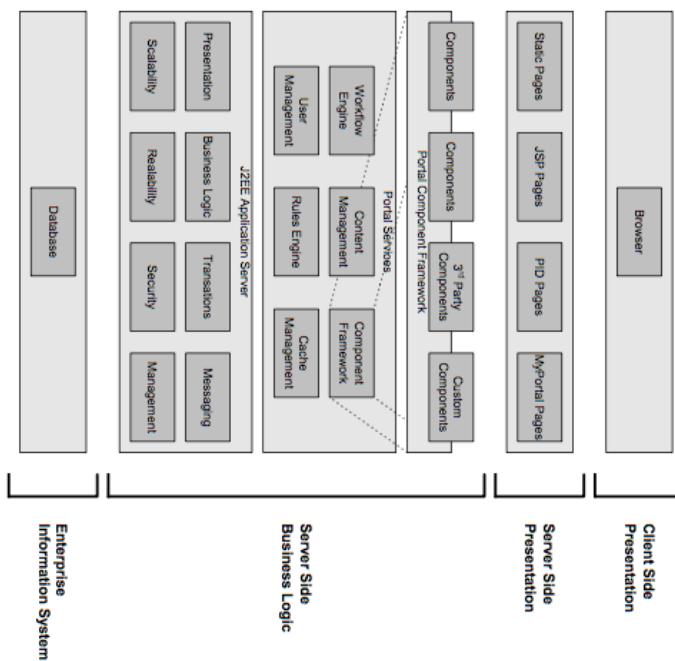
Oben wird etwas gebraucht, ausgewählt – unten definiert (Vitrine); wartet darauf ausgewählt /gebraucht zu werden,

Oben App spezifisch – Unten allgemeiner nutzbar (Wiederverwendung eher möglich)

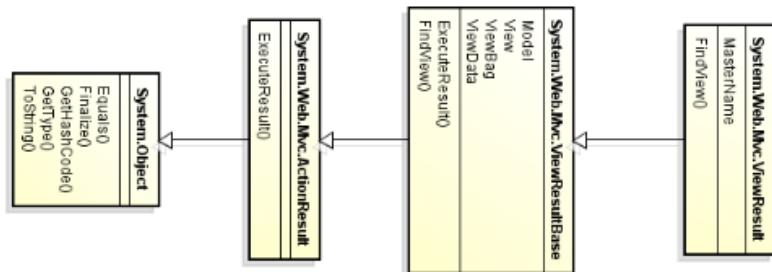
Oben eher von Hand testen – unten eher automatisch testbar.



## Beispiel für vier bzw. sechs Schichten



## Asymmetrie der Abhangigkeiten

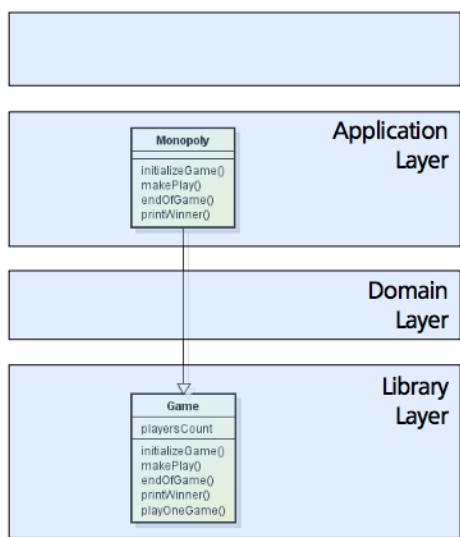


z.B. bei Vererbung:

Eine abgeleitete Klasse ist abhangig von allen Basisklassen/Mutterklassen in der Vererbungshierarchie

Problem der 'Brittle Base Class': eine Änderung weit unten in der Hierarchie hat potentiell sehr weitreichende Konsequenzen. Also darf ich später an einer Basisklasse kaum etwas ändern.

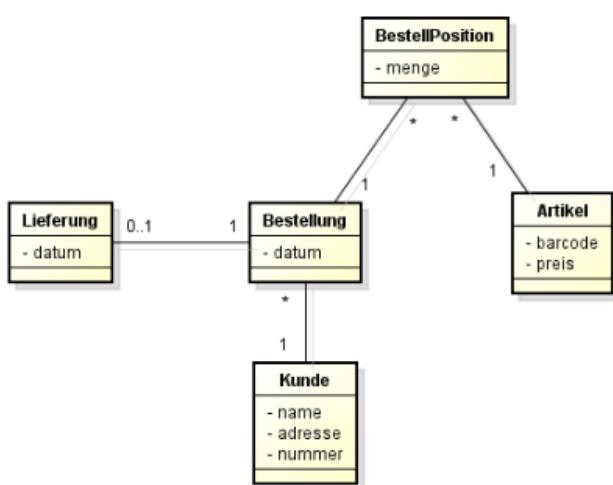
(s. Metrik 'Depth of Inheritance Tree')



Warum zeichne ich bei der Vererbung die Mutterklasse (Basisklasse) unten und die abgeleitete (Tochter-) Klasse oben?

Weil das dem Einsatz- Szenario entspricht: Gegeben ist die Basisklasse aus einer Bibliothek (unten), und ich leite davon ab, in einer höheren Schicht.

### „oben – unten“ bei Datenklassen



Beispiel: Bestellung hat den Foreign Key „kunden\_ID“ und referenziert damit den Primary Key „ID in der Tabelle „Kunde.“

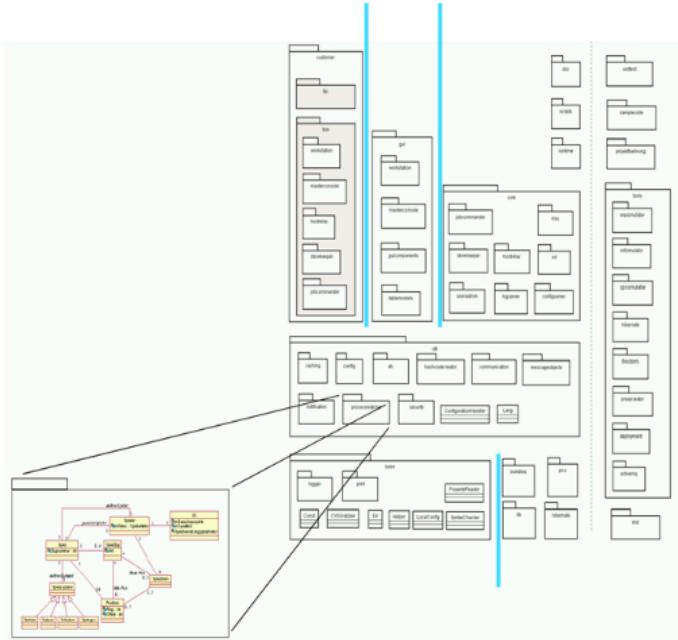
Warum zeichne ich bei 1:n Beziehungen bei reinen Datenklassen die 1-Klasse unten und die n-Klasse oben.

Weil das der Abhängigkeit entspricht: Der Foreign Key in der n-Klasse (oben) referenziert den Primary Key in der 1-Klasse (unten). Nicht umgekehrt.

### Arten von Kopplung

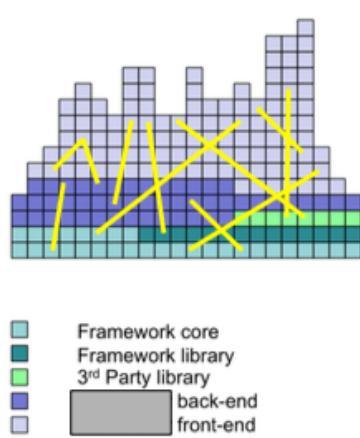
#### Kopplung

stark (schlecht)	Direktzugriff auf Datenstrukturen einer anderen Klasse Sharing a global variable Viele verschiedene Methodenaufrufe einer anderen Klasse Hohe Anzahl komplexer Parameter bei Aufrufen Vererbung Kreieren eines Objektes und Halten der Referenz Halten einer Referenz auf ein anderes Objekt Implementation eines Interfaces Typ-Abhängigkeit
schwach (gut)	Einfache Aufrufe mit wenig Parametern



Vertikale Unterteilung d.h. gleiche Höhe der Pakete/Klassen, aber (möglichst) unabhängige Zuständigkeiten.

## Partitionieren



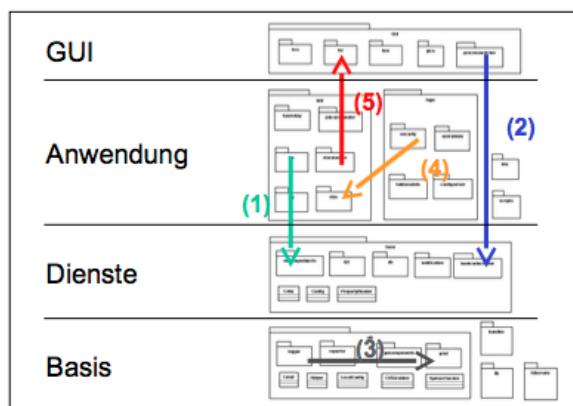
Layering allein genügt nicht.

**Beispiel:** Grösseres System ohne Partitionierung (120'000 LOC, 20 Personen-Jahre), alles ist potentiell mit allem verknüpft.

Auswirkungen:

- Änderungen können unerwartet weitreichende Folgen haben: verlängerte Entwicklungszeiten
- Updates der Basissoftware (Linux, Apache, PHP, Zend Framework, MySQL): alles oder nichts - leider bis 6 Jahre alt.
- Fehlersuche ist erschwert: potentiell überall
- Testen nach Fehlerbehebung: ALLES!

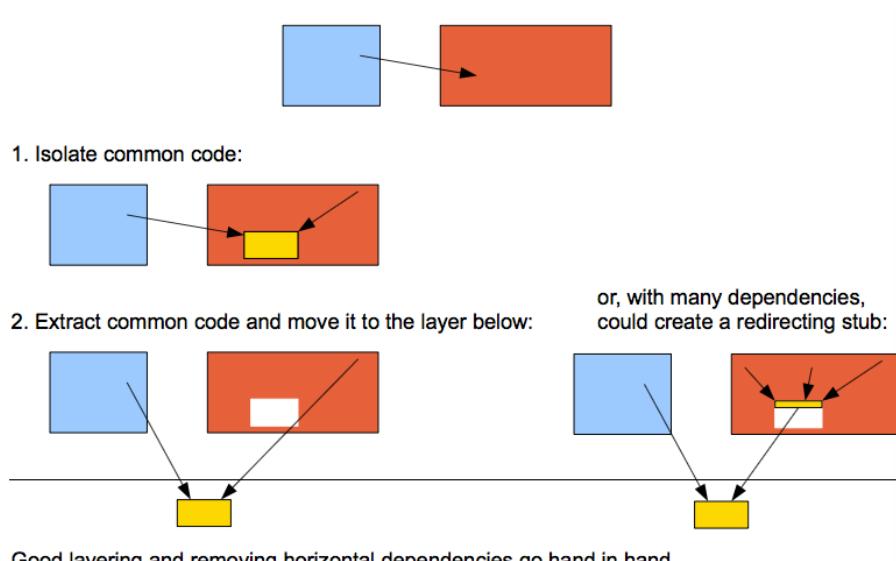
## Regeln für Abhängigkeiten



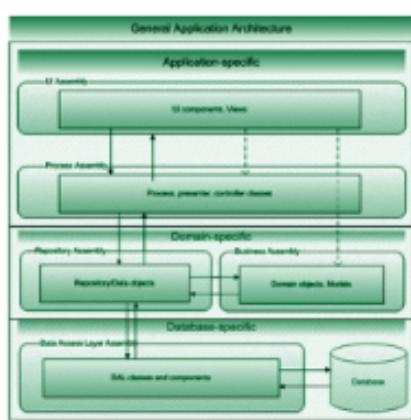
**Aufrufe**

1. ... von unten auf die nächste darunterliegende Schicht sind immer ok
2. ... nach unten, die eine Schicht über hüpfen sind manchmal auch ok
3. ... innerhalb einer Schicht und Partition sind OK, sollten aber minimiert werden
4. ... in einer Schicht quer zu einer anderen Partition sollten dringend vermieden werden.
5. ... Nie von unten nach oben, ausser callbacks (z.B. Observer Pattern)

**Abhängigkeiten** zwischen Klassen in zwei Partitionen derselben Schicht sollten vermieden werden  
(4.). Wie behebt man das am besten?



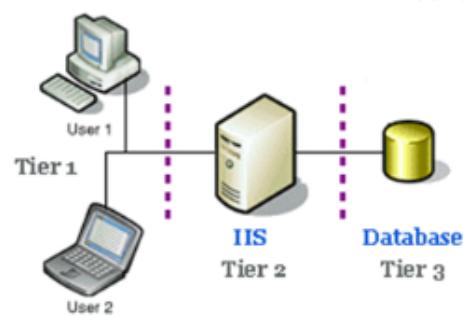
Abhängigkeiten = Kopplung

**Layers & Tiers****Schichten (Layers, horizontal)**

„Wie der Code strukturiert ist“ hierarchische Abhängigkeiten, klar, wer den Takt angibt.

**Tiers (vertikal)**

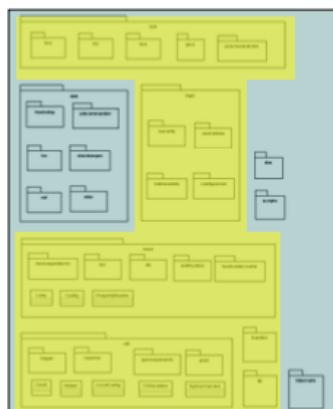
Bilder: [davidhayden.com](http://davidhayden.com)



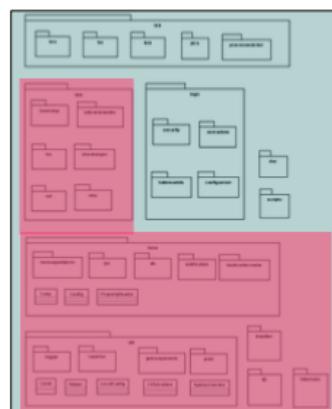
„Wo welcher Teil läuft“, in der Regel gleichberechtigte Partner, kein diktierter Takt.

In einem System werden n Schichten üblicherweise auf n-x Tiers abgebildet (selten n+x). Beides sind Ansichten derselben Architektur: Klassen/Schichtendiagramm und Deploymentdiagramm.

Geteilter Code – als typisches Beispiel: Libraries- kann auf mehreren Tiers (=Maschinen) laufen.

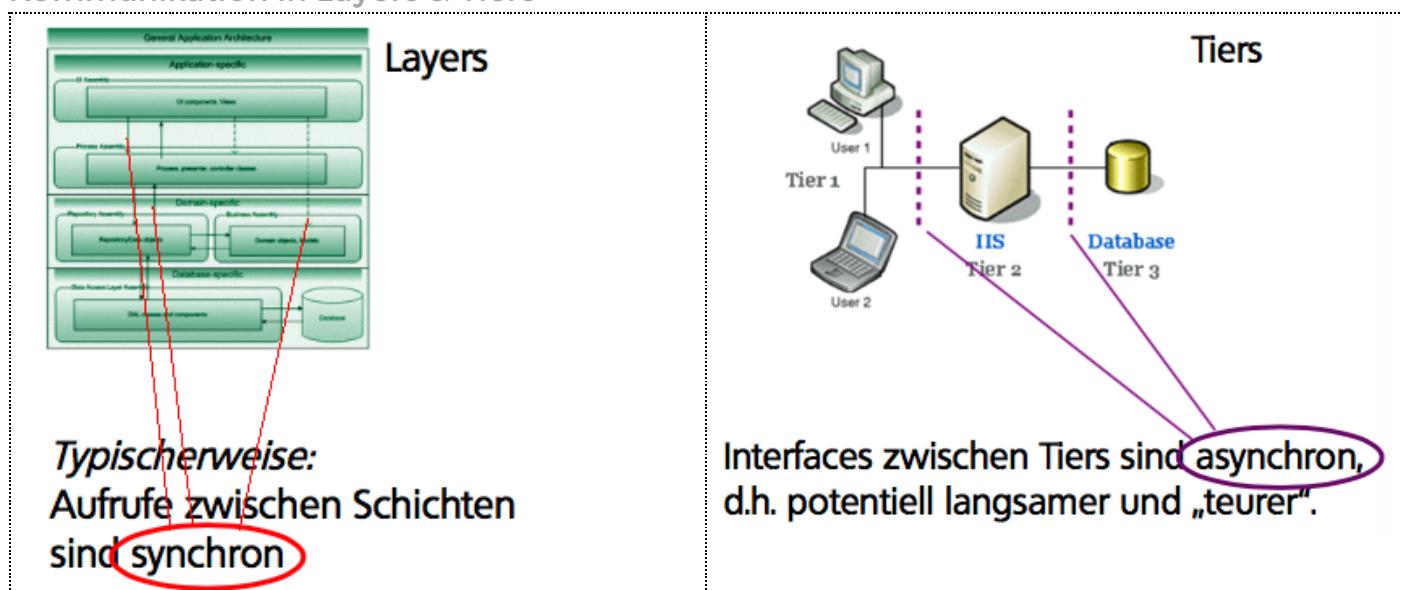


(Fat) Clients



Server

## Kommunikation in Layers & Tiers



## Partitionen in eigener Laufzeitumgebung

Auch Partitionen können evtl. auf eigenen Tiers gelagert werden.

### Vorteile

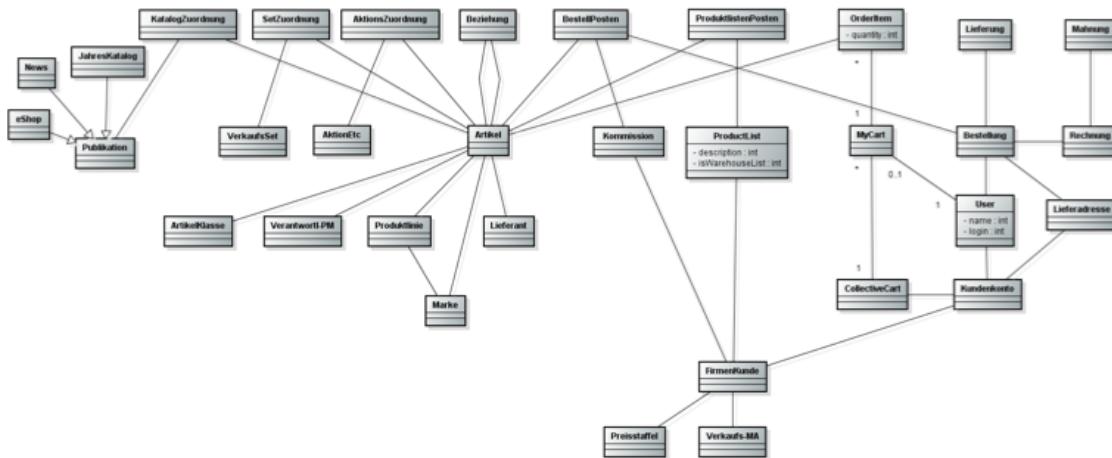
- Man hat Freiheiten in der Implementation (z.B. das meiste in PHP, „Data Receiver“ in Java aus Performance-Gründen)
- Man kann die Update-Falle umgehen
- Weniger Nebenwirkungen bei Erweiterungen
- Leichter (Unit) testbar
- Schnellere Fehlersuche

### Nachteile

- Potentielle Performance-Einbusse (Lag)
- Aufweniger im Deployment und der Wartung

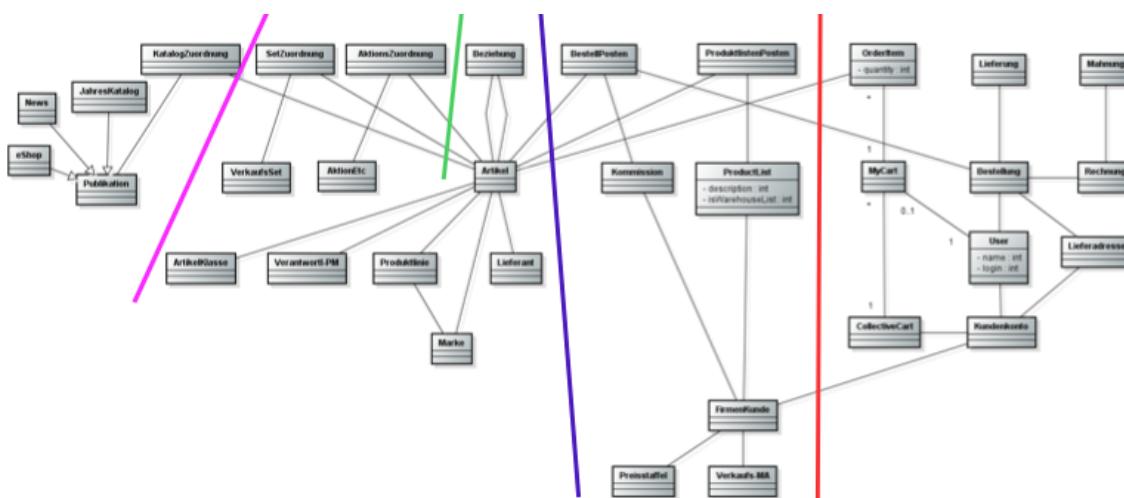
## Datenmodell kann auch partitioniert werden

... und damit auch die Verantwortlichkeiten. Beispiel: e-Commerce System. Eine Schnittstelle sollte möglichst wenig Beziehungen durchschneiden. Hier ist das Datenmodell 1:N unten:oben geordnet.



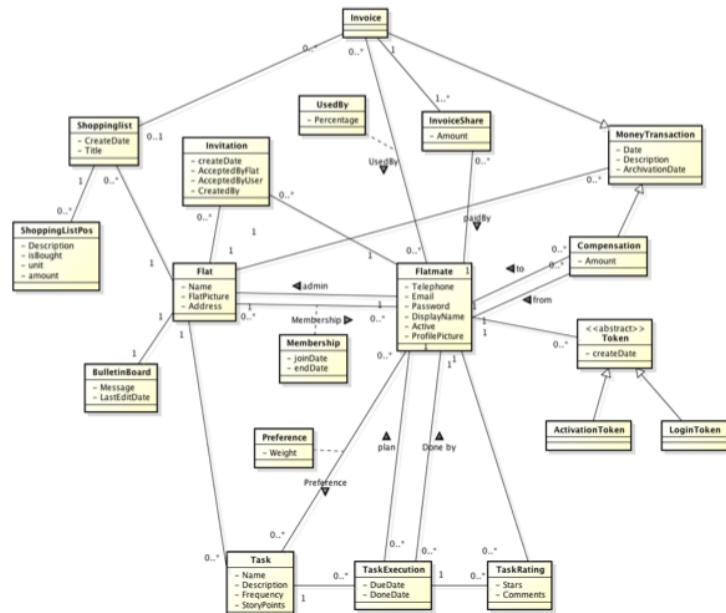
### Abhängigkeiten beim Schneiden

Eine Schnittstelle sollte nur Beziehungen in einer Richtung durchschneiden. Ist beim Schnitt ganz rechts nicht der Fall.

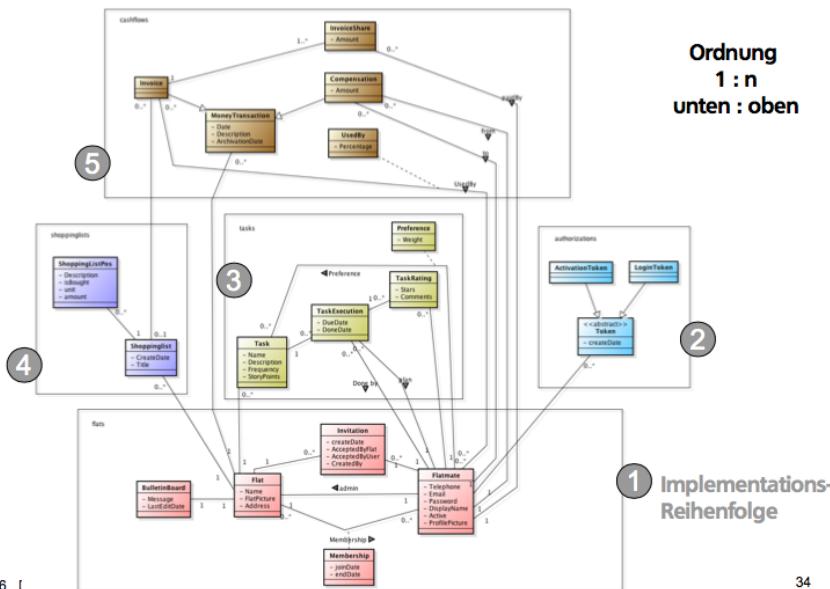


**Beispiel PopPins****WG-Software** mit Ämtli, Bewertungen, Einkaufsliste und Geld verrechnen

Dieses Beispiel soll zeigen, wie wichtig das eine Ordnung sowie eine Partitionierung des Datenmodells es. Der Inhalt der beiden Modelle ist dabei genau der selbe.



© 2016



© 2016

34

**Zusammenfassung**

Sie sollten das folgende jetzt kennen und einordnen können:

- Schichten (Layers) mit verschiedenen Bezeichnungen
- Packages (als Ordnungsmittel, als Schichten & Partitionen)
- Partitionen (Code und Daten)
- Tiers (auch wenn viele Leute keinen Unterschied zu Layers machen)
- Wo synchrone und wo asynchrone Kommunikation
- Regel für Abhängigkeiten (oben – unten – quer)
- Tiefe Kopplung und hohe Kohäsion
- Es ist nicht egal, wie ich die Klassen oben/unten anordne (Code und Daten)

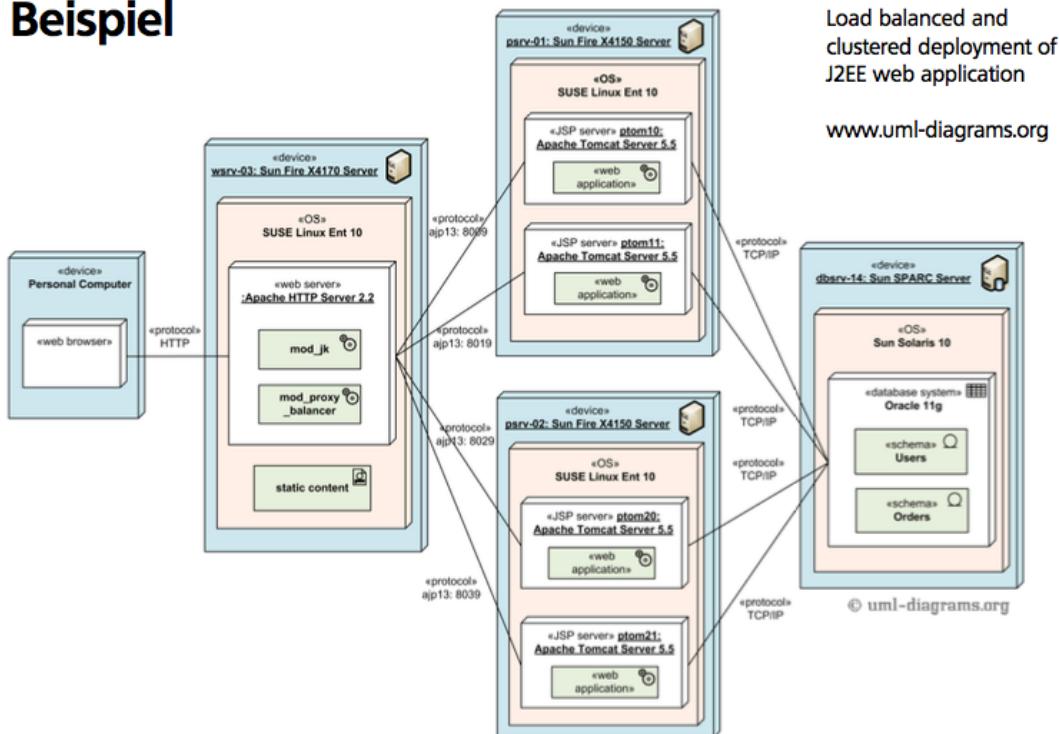
# Deployment Diagramme

Auf Deutsch: Einsatz- und Verteilungsdiagramm

„Verteilungsdiagramme zeigen, welche Software (Komponenten, Objekte) auf welcher Hardware (Knoten) laufen, daher wie diese konfiguriert sind und welche Kommunikationsbeziehungen dort bestehen.“

## Beispiel

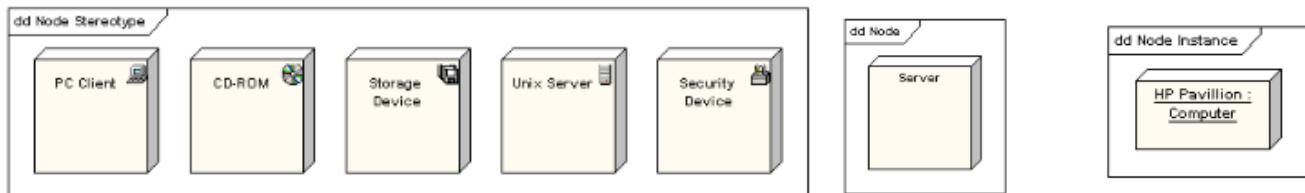
### Beispiel

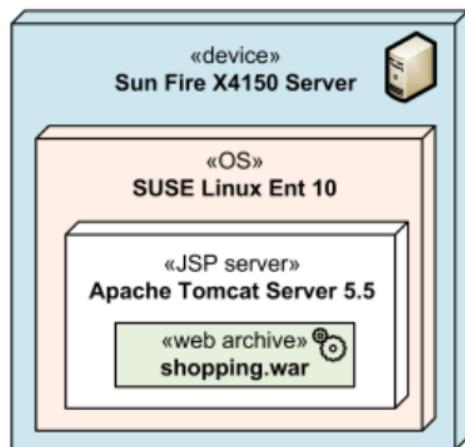


## Notation

### Knoten

Knoten sind Hardware (heute meist virtuell) und können etwas ausführen.

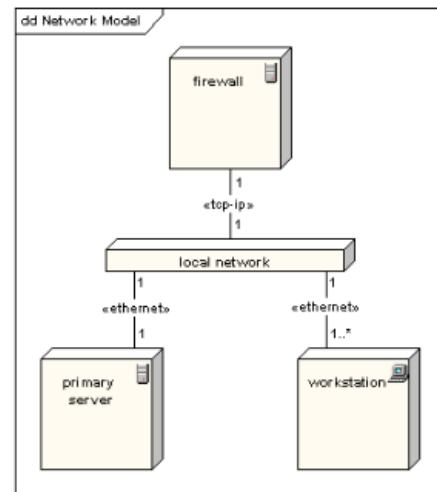
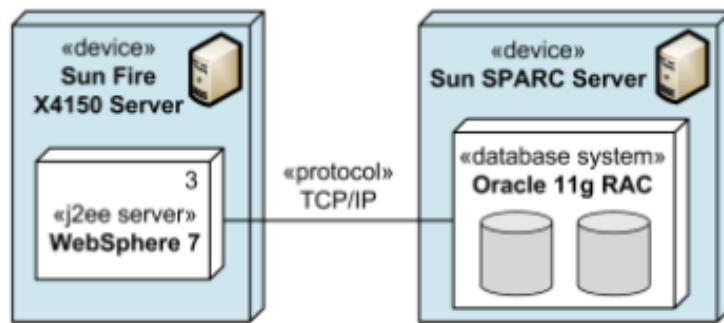




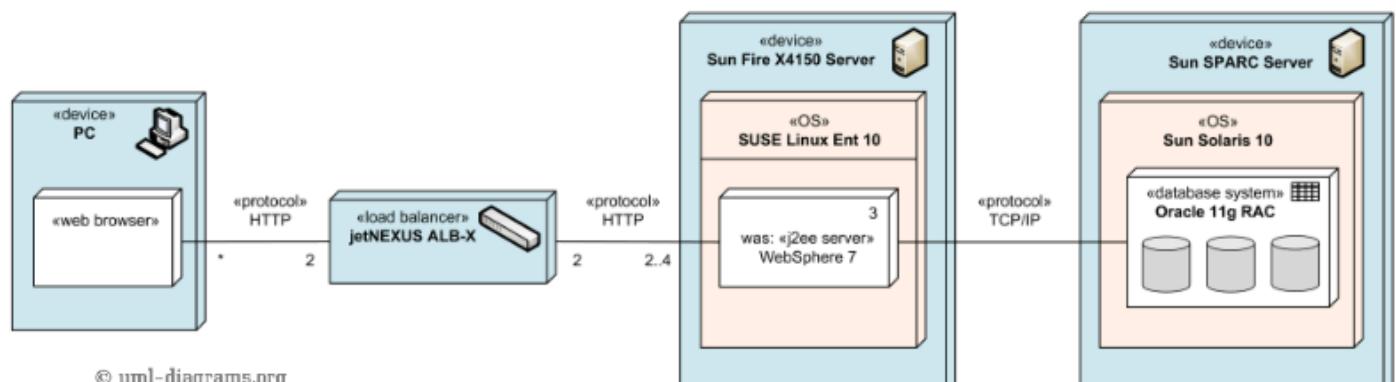
Verschachtelte Ausführungs-Umgebungen innerhalb eines Knotens. Typisch sind es „OS“, „workflow engine“, „database system“, „J2EE Container“, „web server“ oder „web browser“.

## Assoziationen

Verbindungen mit Angaben zum Protokoll und evtl. Multiplizitäten.



## Beispiel Load Balancer



## Anwendungsbereich & Nutzen

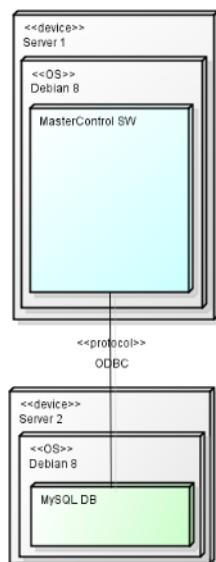
Beschreibung für Software Architekturen

- Einsatz-Szenarien sichtbar (oft mehrere Szenarien)
- Abstraktion für z.B. Performance-Diskussion
- Visualisierung Security: Angriffspunkte
- UML ← Vagrant, Chef, Puppet (für J2EE auch →)

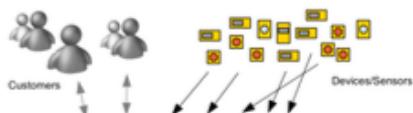
## Mehrere Deployment-Varianten



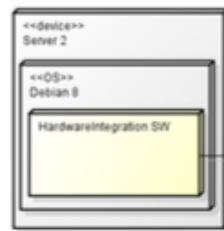
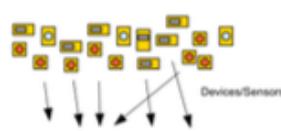
Einfach umzusetzen:  
so oder so



## Architektur-Umbau

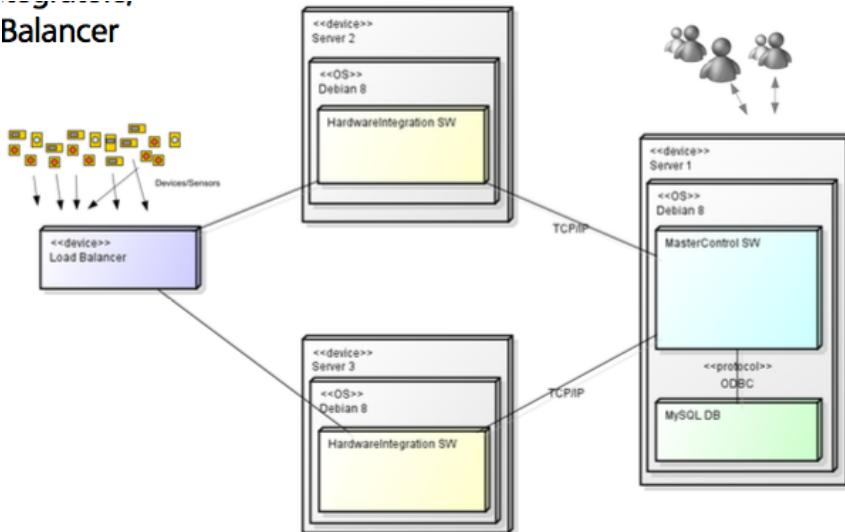
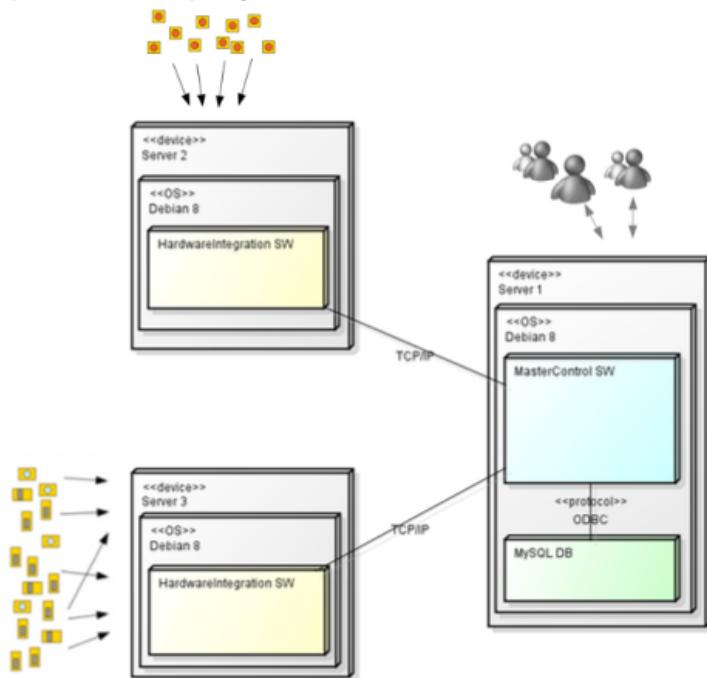


Herauslösen der Hardware-Abhängigkeiten:  
Empfangen der Sensor-Daten wird an separaten Prozess ausgelagert



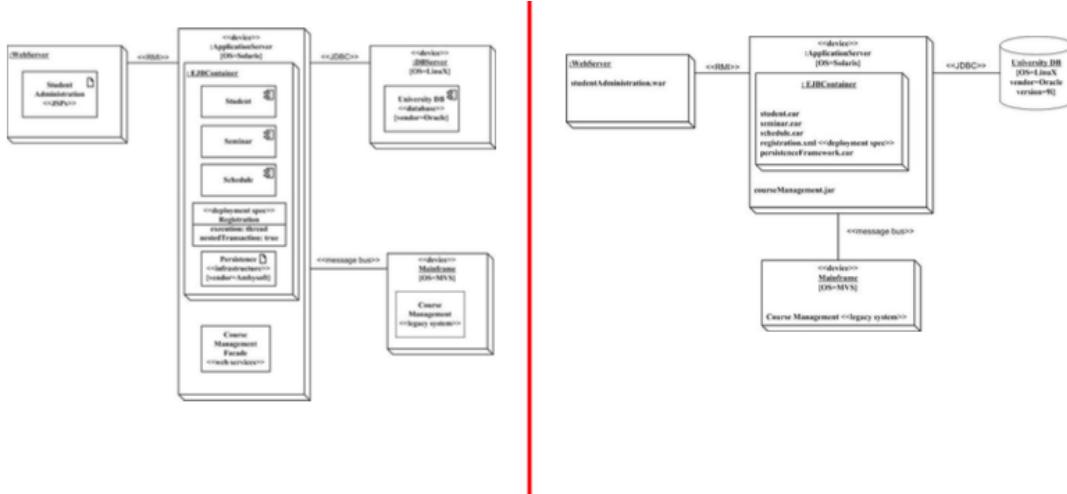
## Neue Deployment-Variante

Mehrere Instanzen des HWIntegrators mit Load Balancer.

**Balancer****Spezielle Deployment-Variante**

Ein Sensortyp (rot-gelb) mit sehr hohen Performance-Anforderungen kriegt einen eigenen, dafür getunten HWIntegrator.

Alle anderen Sensoren melden sich wie vorher beim HWIntegrator 1.

**Verschiedene Stile**

# Unified Process – Die Minimalversion

## Geschichte

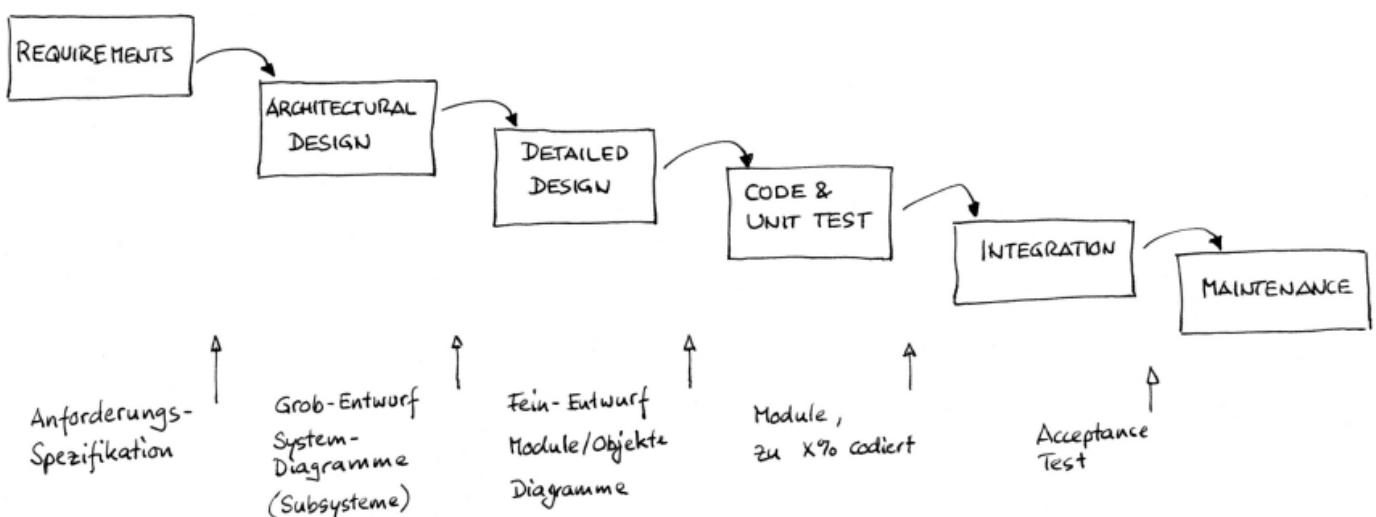
Das Wasserfall-Modell, das übliche Vorgehen seit Urzeiten (bis ca. 1990). Da gab es IBM und Oracle, Informatik in Grossfirmen und sonst nichts

## Analogie

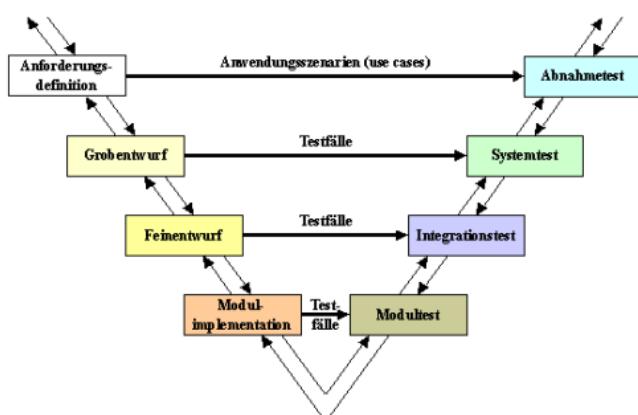
Wie baut man eine Brücke? (Engineering = Planung!) und da meinten viele im Ernst, Software bauen sei wie Brücken bauen. Nur genügend Ingenieure und Arbeiter anstellen, dann klappt das mit guter Planung.

## Vorgehensmodell «Wasserfall»

Zeit, Budget und Funktionalität vorgegeben.



## V-Modell (=geknickter Wasserfall)



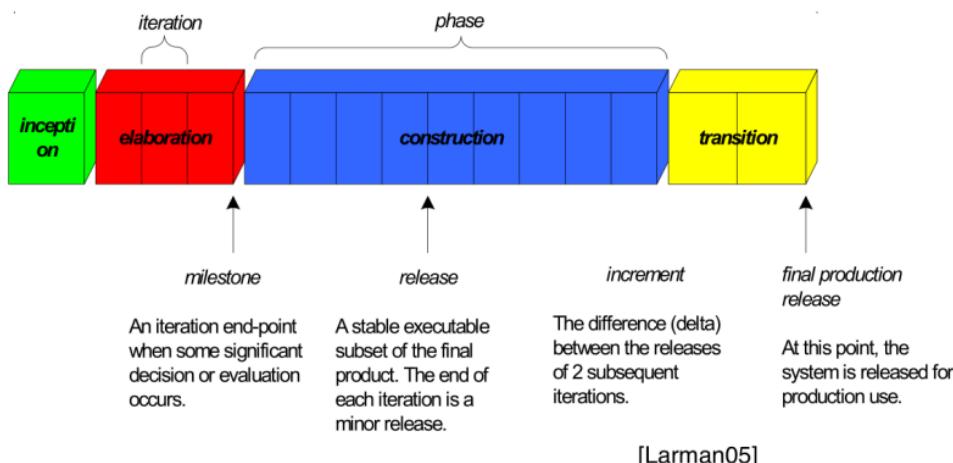
Bei grossen Firmen: immer fixes Budget und fixe Termine. Typisch für Banken, Versicherungen, Militär, Pharma und Verwaltungen. Das Vorgehen ist OK, wenn man schon einmal so etwas gemacht hat.

## Nachteile

Schwerfälliger Tanker, Illusion der Planbarkeit

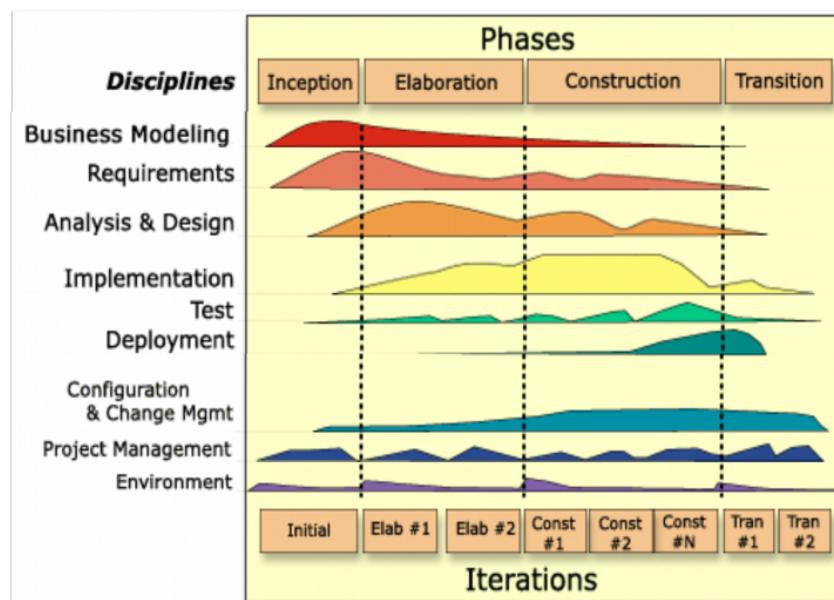
Versuch, besser zu sein als das Wasserfall-Modell (ca. 1997 ... 2002). Definiert ein teils strukturiertes, teils agiles Vorgen.:

- 4 Phasen
- Immer Iterativ
- Disciplines
- Roles
- Work Products (Artefacts)



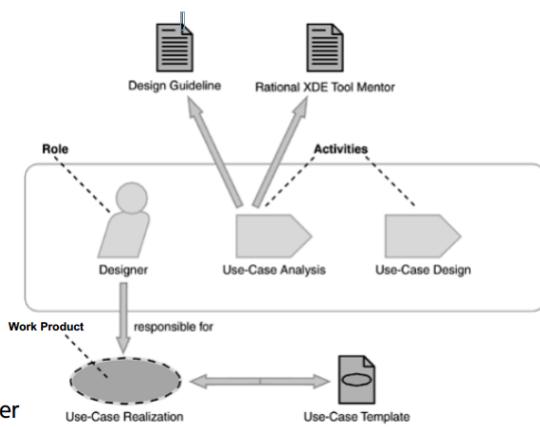
Wie gesagt immer iterativ, in diesem Fall sind es 15 Iterationen. Phasen können unterschiedlich lang sein, z.B. bei hoch innovativen Projekten ist die Elaboration deutlich länger, bei bekannten Themen kann die Elaboration kürzer sein.

## Disciplines



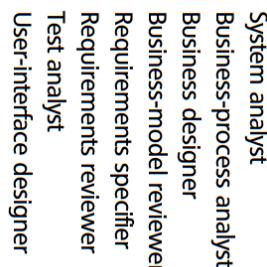
Roles:

- Analyst
- Designer
- Architect
- Developer
- Tester
- Project Manager
- ...



### UP geht (zu) tief ins Detail

UP definiert 150 Aktivitäten, 40 Rollen und 80 Major Work Products. z.B. wird die Rolle des Analysten – falls gewünscht – noch weiter unterteilt. Daher ist es für kleinere Sache nicht unbedingt geeignet.



### UP = Riesige Apotheke

Das heisst flexibel und vernünftig handeln. Daher nicht alle Medikamente auf einmal nehmen. Im UP nennen Sie das «to tailor the process» («Zuscheiden» zu Beginn).

Der Unified Process ist nicht per Definition ein Wasserfall. UP Kann aber wie ein Wasserfall genutzt werden, aber das ist eine (ungeschickte) Interpretation. Richtig eingesetzt, ist UP nur zu Beginn eine Art Wasserfall (aber auch da iterativ), nach End of Elaboration kann der Prozess dann voll agil sein.

### Was bei UP wichtig ist....

#### Immer iterativ vorgehen

Jede Iteration hat ein lauffähiges Produkt zum Ziel. Immer ein Stück anbliefern (Demo, Freude, Feedback für Team). Das Kunden-Feedback ist wichtig. Nach jeder Iteration kann neu geplant/ausgerichtet werden. Der Kunde sieht früh, wenn etwas nicht wie gewünscht läuft.

#### Vier Phasen

Die Inception nur sehr kurz (< 5%), nur Vision und Eckwerte. Elaboration ca. 20- 30% der Zeit und eher chaotisch. Die Construction-Phase hoffentlich ohne Überraschungen zudem teuer und viel Personal. Die Transition stark projektabhängig, aber immer da.



## End of Elaboration

Wichtiger Meilenstein mit einer Checkliste (siehe nächster Punkt). Der Abbruch ist hier noch relativ günstig möglich. Aber hier genaue Schätzungen. Es bildet der Wendepunkt vom Ausprobieren zum Produzieren.

### Arbeitsteilung

Bei grösseren Projekten (mehr als ca. 8 Personen, mehr als ein Standort) gilt:

*Die Arbeit aufteilen ist erst möglich, nachdem der Kunden verstanden wurde (Scope definiert), die Architektur steht und die Tool Chain steht.*

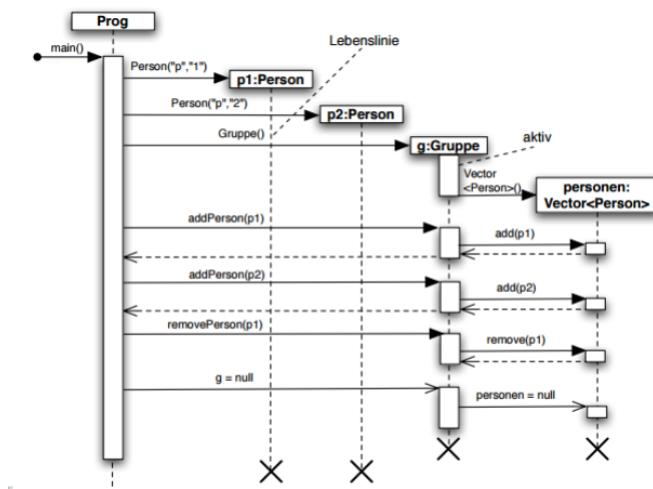
Und denn dieses Punkt, ab wann man das Aufteilen kann ist das End of Elaboration.

### Checkliste – End of Elaboration

- Wir haben den Kunden verstanden (Requirements so vollständig wie nur möglich). Der Scope (=grober Funktionsumfang ist vereinbart).
- Wir haben alle Werkzeuge im Griff (DIE, Versionskontrolle, Build Server, Deployment, Unit Testing, Workflow Tools, Wiki, ...)
- Wir wissen, wie die Architektur aussehen wird (Architektur skizziert, die grossen Interfaces festgelegt, Architektur-Prototypen gemacht).
- Für das User Interface gibt es einen ersten Entwurf (Grafiken, Wireframes, klickbarer Prototyp) und dem Kunden gefällt es.
- Wir haben die Detailplanung für die nächsten zwei Iterationen gemacht und wir haben dem Kunden eine genaue Zeitschätzung geliefert.
- Alle grossen Risiken, alle grossen Fragezeichen sind weg.

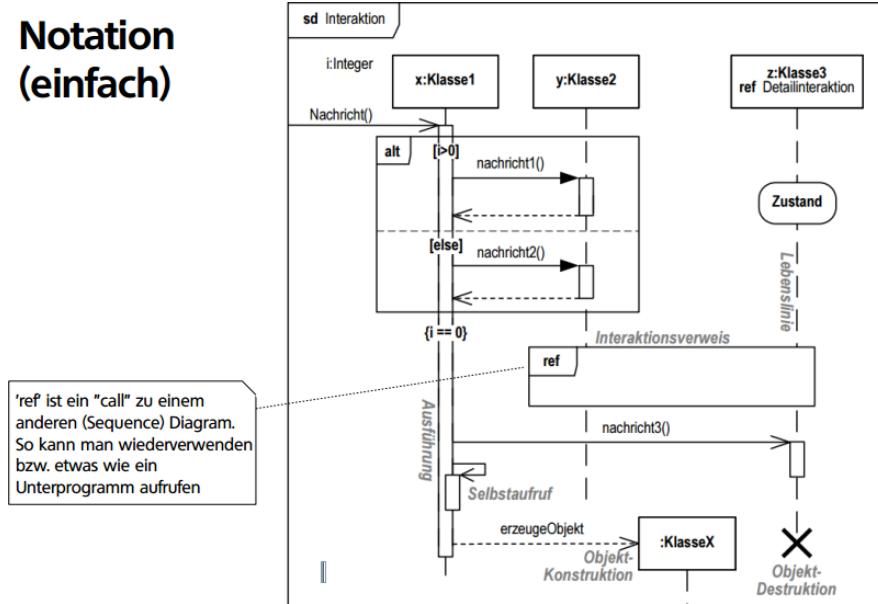
### Zusammenfassung

Der Unified Process kann sehr gut auf wenige Elemente eingedampft werden, und dann ist er so agil wie Scrum mit etwas Vorausplanung. 4 Phasen. Kurze Iterationen. Sehr wichtig und hilfreich der Checkpoint «End of Elaboration». Nach «End of Elaboration» hoch agil weiterfahren.



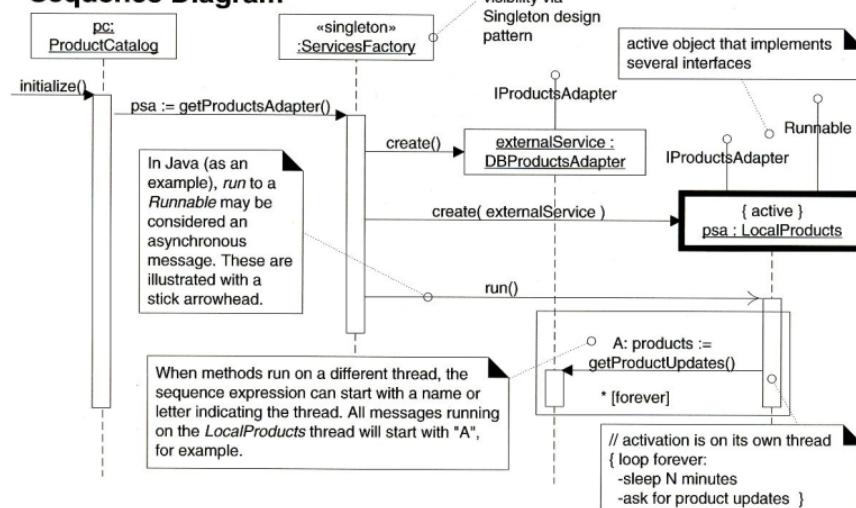
## Notation (einfach)

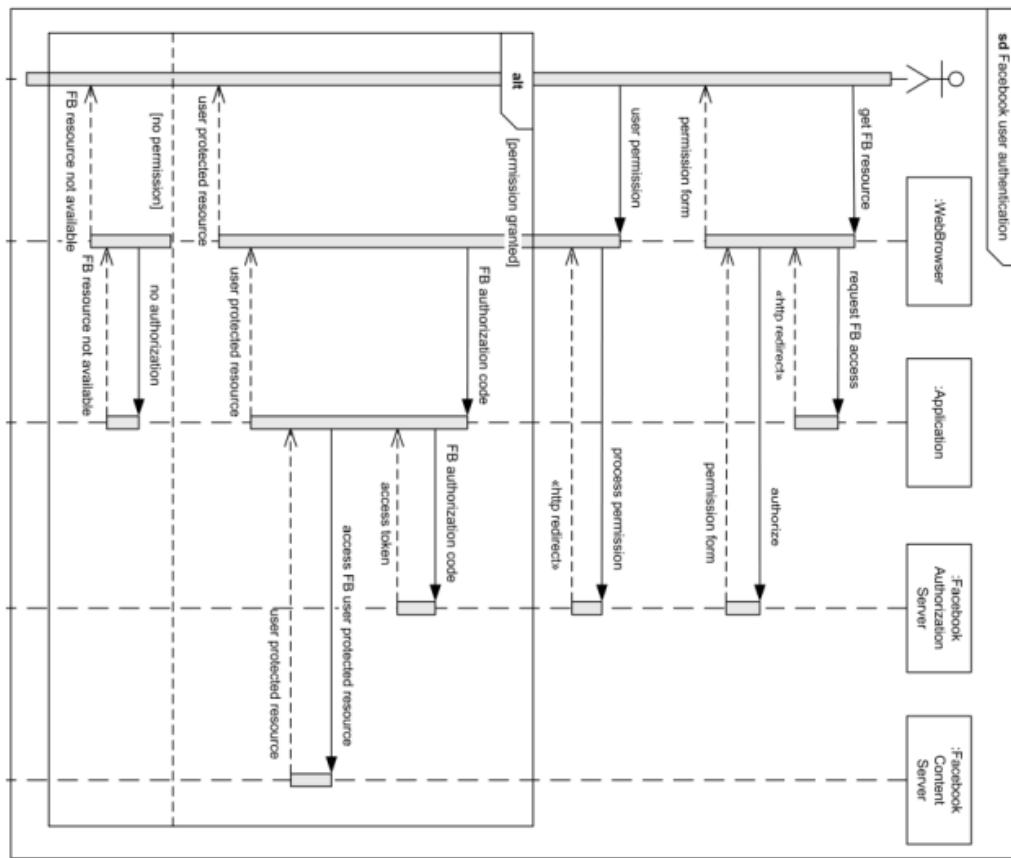
## Notation (einfach)



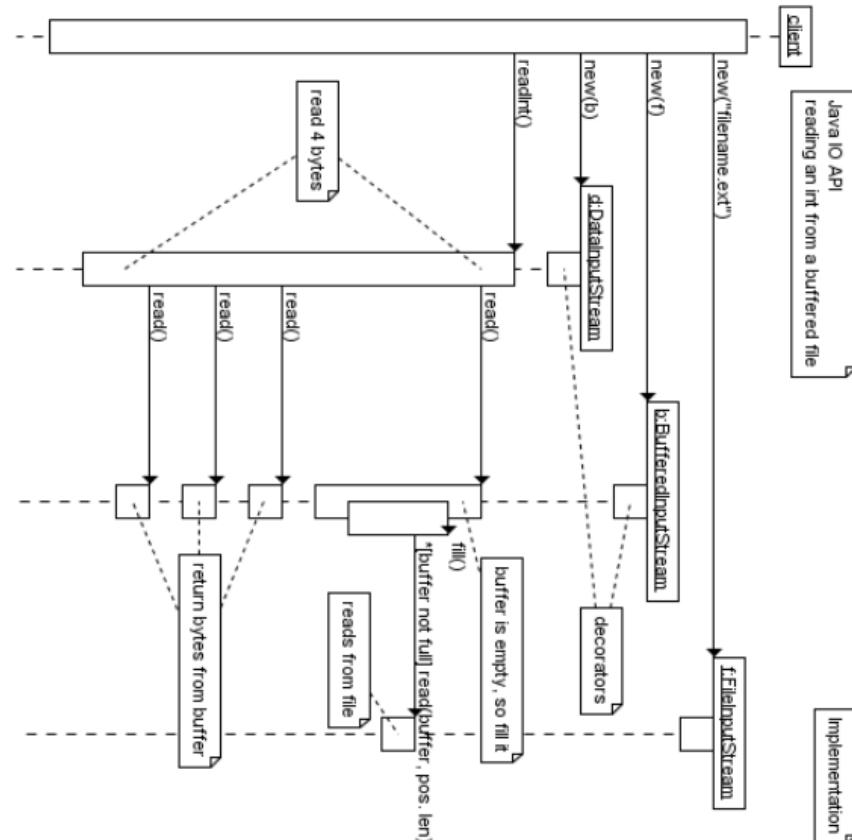
## Notation (komplexeres Beispiel)

## Sequence Diagram





Java.io

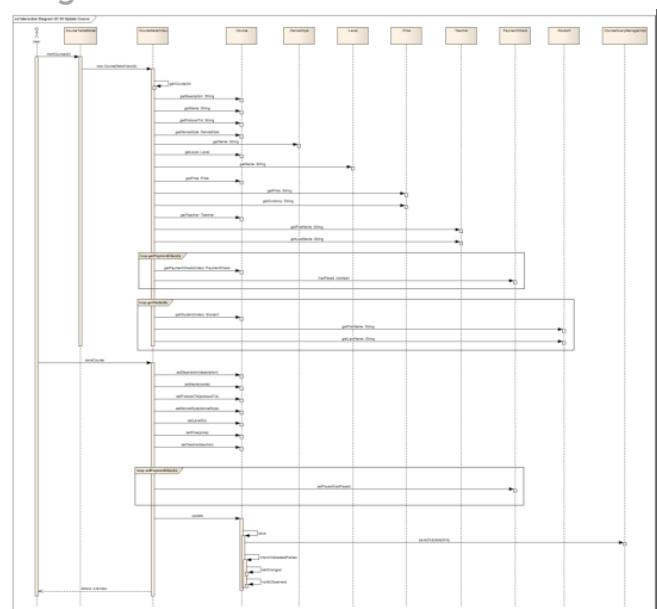


- Zeigt detailliert das Zusammenspiel von mehreren Klassen
- Ist ein Werkzeug für SW-Entwickler (nicht für Req. Spezifikation)
- Ist nah am Code, von einigen Tools sogar zur Laufzeit generierbar
- Hat seinen Einsatz im objekt-orientierten Design
- Entweder beim Entwurf oder als nachträgliche Dokumentation

### Nutzen von Sequenzdiagrammen

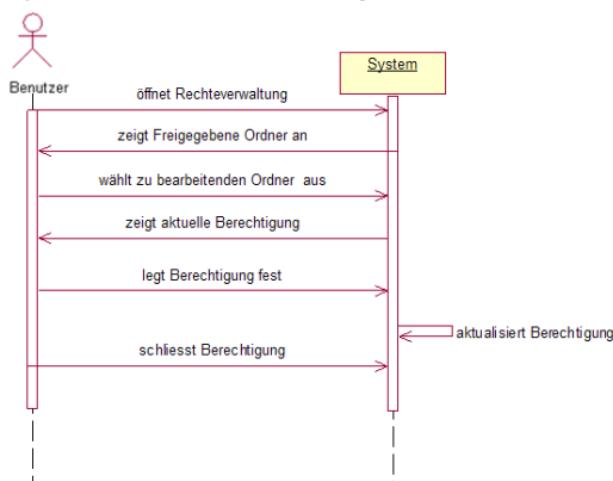
Das Zusammenspiel von mehreren Klassen ist im Sequenzdiagramm besser sichtbar als im Code. Komplexe Sequenzen (z.B. das Aufstarten von mehreren Prozessen) können gut erklärt und diskutiert werden. Besonderer Wert beim Zusammenspiel von Teilen, die von unterschiedlichen Teams/Personen entwickelt werden.

### Diagramme sind Kommunikation



Ein Sequenzdiagramm kann schnell einmal sehr gross werden. Dann wird es schwierig zu kommunizieren (Druck, Beamer). Wichtig, die A3-Regel beachten.

### System-Sequenzdiagramm (SSD)



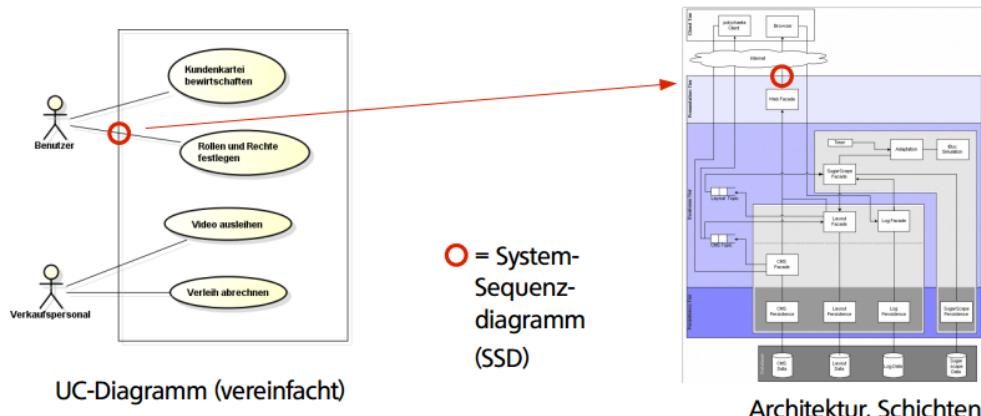
Immer nur zwei Handelspartner: ein Aktor und ein System (es gibt Ausnahmen z.B. Multiplayer Games).

System → Blackbox

Erfindung von Larman unter anderem weil Domainmodell ohne Methoden.

Schlaue Idee, aber nicht weitherum bekannt.

Aber warum eine «Schlaue Idee von Larman». Die Oberste (Unit-)testbare Schnittstelle (Methodenaufrufe ohne GUI) könnte z.B auch eine REST Schnittstelle definieren.



## Mehr über SSD

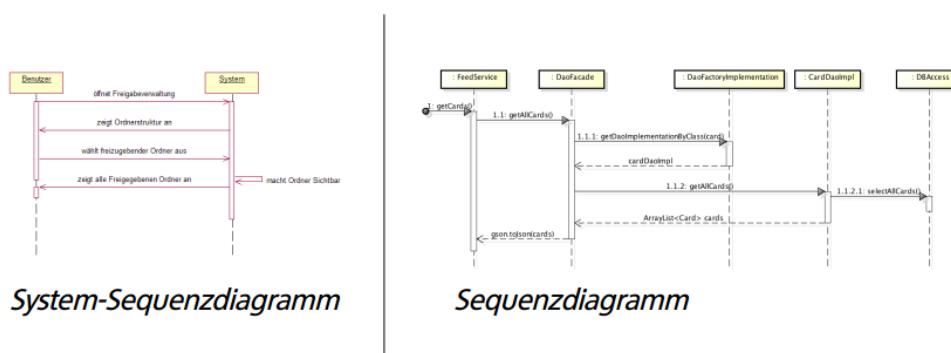
### Ein System-Sequenzdiagramm (SSD)

- Entsteht früh im Projekt, während des Erfassens von Anforderungen und während der Analyse
- Zeigt die Interaktion eines Users mit dem zu entwerfenden System
- Modelliert oft ein Use Case Szenario
- Hat nur zwei Mitspieler: Aktor und System als Black Box
- Kann mit Contracts versehen als Programmier-Spezifikation dienen

Mehr Details und Anwendungsbeispiele finden Sie im Buch von Larman. (Prüfungsstoff)

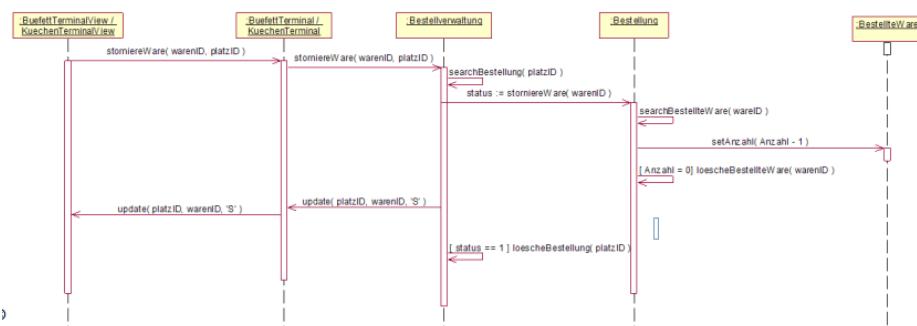
## Verwechslungsgefahr

Optisch sowie technisch gehören beide zur Klasse der UML Sequenzdiagramme. Es sind aber doch zwei sehr verschiedene Spezies.

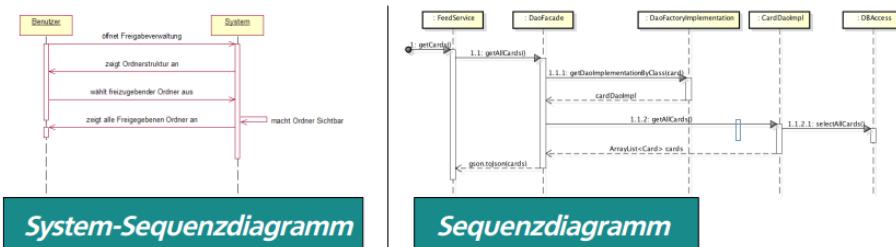


## Sequenzdiagramm im Vergleich

Es gibt immer mehr Handelspartner. Ein Sequenzdiagramm mit nur zwei Mitspielern bringt nichts. Dies kann man einfacher im Code lesen. Zudem ist das System meist als White Box. So wird das Zusammenspiel mehrerer Klassen gezeigt, was im Code schwieriger zu lesen wäre.



## Zusammenfassende Gegenüberstellung



System-Operationen

immer 2 Dialogpartner

System = Black Box

Teil von Requirements Analysis

Näher beim User/Kunden

Weniger bekannte Larman-Erfindung

Zusammenspiel von mehreren Klassen

sinnvoll mit mehr als 2 Dialogpartnern

System = White Box

Teil von OOD bzw. nachträgliche Doku

Nah beim Code/Entwickler

von/nach Code generierbar



Copyright © 2012, Kenneth S. Rubin and Innolution, LLC. All Rights Reserved.

## Definition of Scrum

Scrum: A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.

Scrum is:

- Lightweight
- Simple to understand
- Difficult to master

Scrum is a process framework that has been used to manage complex product development since the early 1990s. [ . . ] The Scrum framework consists of Scrum Teams and their associated roles, events, artifacts, and rules.

## Product Backlog

Stellen Sie sich eine sehr lange ToDoListe vor. Die Einträge sind kleine Arbeitspakete mit

- Aufgabenbeschreibung (Was der Kunde will?)
- Akzeptanzkriterien
- Aufwandschätzung
- Priorität

Die Arbeitspakete sind Programmieraufgaben (Features) oder Bugs.

## User Stories

Die Arbeitspakete im Backlog sind die User Stories mit Beschreibung, Akzeptanzkriterien, Aufwandschätzung und Priorität.

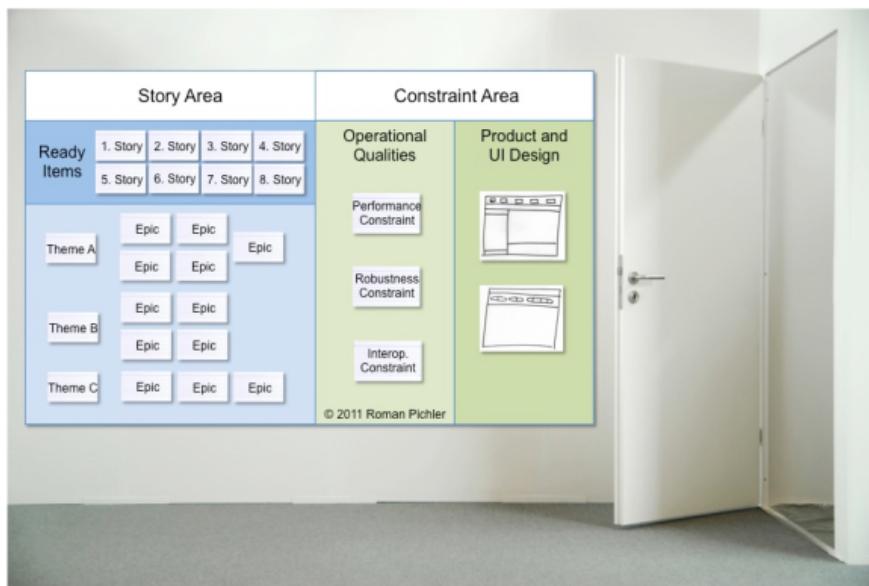
1 Elemente erstellen

2 Elemente in Rangfolge einordnen

3 Elemente schätzen

	A	B	C	D	E	F		
1	<b>Projekt:</b> Dev10Demo <b>Server:</b> processbuild02\DefaultCollection <b>Abfrage:</b> Produktrückstand <b>Listentyp:</b> Flach							
2	ID	Titel		Rang	Punkte	Zustand	Iteration	Pfad
3	58	Als Neukunde möchte ich Essen bestellen.		1	4	Gelöst	\Iteration 0	
4	68	Als Neukunde möchte ich, dass das Menü auf Lieferanten beschränkt wird, in deren Liefergebiet ich mich befinde.		2	4	Aktiv	\Iteration 0	
5	59	Als Neukunde möchte ich auf der Website auf einen Blick einen Eindruck vom DinnerNow-Angebot erhalten.		3	5	Aktiv	\Iteration 0	
6	60	Als Kunde, der bereits eine Bestellung aufgegeben hat, möchte ich, dass DinnerNow meine Vorlieben aufzeichnet.		4	9	Aktiv	\Iteration 0	

## Product Backlog Board



## User Story

Titel und Nummerierung. Entweder ist es im freien Format geschrieben oder formalisiert. Größen: kurzer Text (Story Card) mit wenigen Zeilen oder fast wie «fully dressed UC», je nachdem wieviel man wie genau schon weiß.

### Kurzes Beispiel

Oft formalisiert. Meist noch mit Randbedingungen/Erklärungen/Fehlersituationen und mit ausführlichen Akzeptanz-Kriterien.

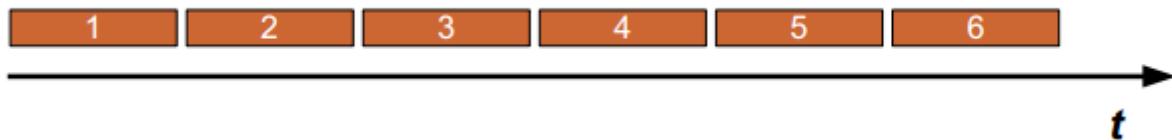
As a [ user/role ] I want to [ goal ] because [ reason/motivation ]

**US203 Anwendung starten:**

Als Autor möchte ich nach dem Start der Anwendung mein zuletzt bearbeitetes Dokument sehen, um Zeit zu sparen.

## Sprints

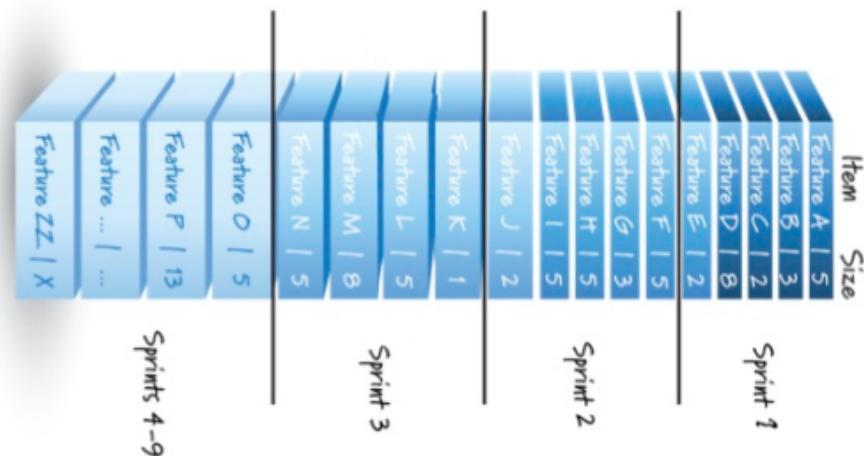
Sind Iterativ. Eine Aufteilung der Zeit in Sprints von zwei oder drei Wochen (die offizielle Empfehlung ist 1-4 Wochen). Sprints haben immer die gleiche Länge. Sprints sind timeboxed, daher was nicht fertig ist kommt im nächsten Sprint. Es gibt kein «100%» fertig. Nur «100%» fertig und abgenommen oder «nicht fertig».



## Ereignisse

### Sprint Planning

Soviel Arbeit, wie das Team in einem Sprint leisten kann, wird in den Sprint Backlog verschoben. In jedem Sprint werden nur Arbeiten gemacht, die im Sprint Backlog sind. Es gibt keine neu hinzugefügte Aufgaben im Sprint Backlog während eines Sprints (im Product Backlog schon).



### Wie weiss das Team, wieviel in einen Sprint passt?

Ganz alleine die Erfahrung. Mit der Zeit kennt das Team seinen «Speed», daher die Anzahl fertigstellter Story Points pro Sprint.

### Scrum Task Board

User Stories werden im Team selbstständig zugeteilt, niemand diktiert. Hier sind die User Stories noch in Tasks unterteilt. Mache machen das so, vor allem wenn die User Stories grösser sind.



## Sprint Review

Am Schluss jedes Sprints:

- Demo für Kunden bzw. Product Owner
- Alles muss vom Product Owner abgenommen werden
- Evtl. Diskussion, was alles wirklich abgeschlossen ist (Einigkeit zwischen Team und Product Owner)
- Speed ausrechnen (Story Points/Sprint)

## Sprint Retrospective

Regelmässig am Schluss eines Sprints, kann aber kurz ausfallen. Fokus auf den Prozess und nicht die Technologien:

- Was ist gut gelaufen?
- Was hat uns Probleme bereitet?
- Wie können wir uns in Zukunft verbessern?

Hier kann ein(e) Scrum Master die Stärken ausspielen.

## Vorausplanung für den nächsten Sprint

Am Schluss des vorangegangenen Sprints oder am ersten Tag des neuen Sprints.

- Backlog durchforsten, aufräumen und ergänzen
- Oberste User Stories: Aufwand schätzen
- Stories durch Product Owner priorisieren lassen
- Story-übergreifende Fragen klären, z.B. Architekturfragen oder Q-Anforderungen wie Performance. Falls nötig, Spike einplanen.

## Rollen

- Mitglied Entwicklungs-Team (keine Hierarchie)
- Product Owner (PO)
- Scrum Master
- Projektleiter (in Scrum nicht vorgesehen)
- Externe Tester (in Scrum nicht vorgesehen)
- Externe Mitarbeiterinnen, z.B. Grafiker (in Scrum nicht vorgesehen)

## Product Owner

- Kunden, Kunden-Proxy, Subject Matter Expert
- Klar auf der «Kunde wünscht» Seite und nicht auf der Seite des Entwicklungsteams
- Pro Scrum-Projekt ist ein Product Owner
- Ideal ist dieser im selben Büro wie das Entwicklungs-Team und ständig ansprechbar. (und jetzt sieht man den Widerspruch zum zweiten Punkt oben, siehe auch die Diskussion «Braucht es im Scrum einen Projektleiter?»)

## Scrum Master

Kennt Scrum sehr gut, hilft bei Schwierigkeiten vor allem mit dem Prozess. Ein Scrum Master pro ca. 5 Scrum Teams.

## Mitglieder des Entwicklungs-Teams

Verschiedene Kenntnisse und Fähigkeiten:

- Programmieren
- Testen
- User Stories schreiben
- User Experience Design, User Interface Design
- Architektur und Design machen
- Netzwerk, Server und Tool Chain pflegen

## Haupt-Tätigkeiten

Aufwand schätzen, Programmieren, Testen, UX Design und Architektur

### Das Schätz-Spiel

Schätzen im Entwicklungs-Team, evtl. auch mit Scrum Poker. Dabei stellt sich die Frage ob Story Points oder mit Stunden. Die Priorisierung erfolgt durch den Product Owner. Value fort he Customer vs.

Aufwand für Programmierung & Testen.

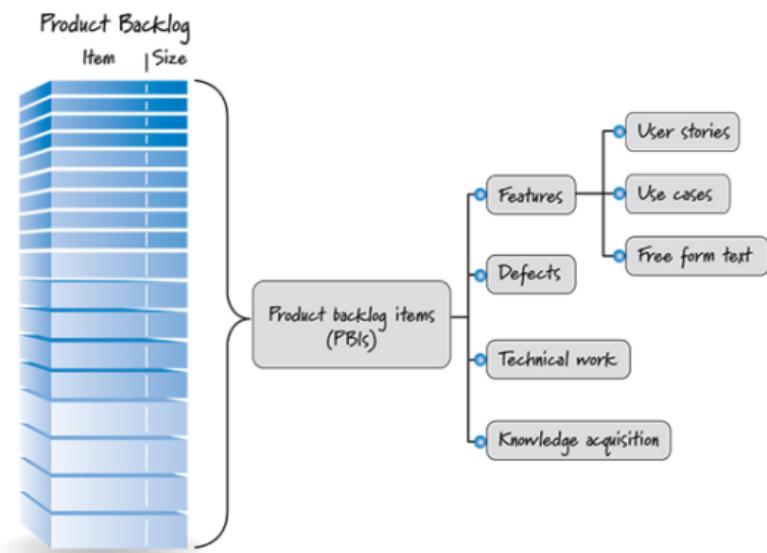
**Wichtig ist immer**      Entwickler schätzen und die Kunden priorisieren

### Weitere wichtige Tätigkeiten

- Dem Product Owner helfen User Stories zu schreiben
- Backlog Grooming (Product Owner? Entwicklungs-Team? Beide?)
- Software-Architektur überprüfen

### Und Sie müssen auch noch...

- Backlog füllen (features, bugs, chores/support)
- Bugs analysieren, priorisieren und lösen
- Stories genauer spezifizieren und in Taks aufbrechen
- Epics definieren bzw. Story Mapping machen
- Spikes definieren und durchziehen
- Impediments Backlog führen
- Impediments aus dem Weg räumen (oft mit Scrum Master)
- User Stories übersetzen (CH-Kunde/PO, Dev Team in Polen)
-



## Definition of Done

Eine Vereinbarung, was man meint, wenn man sagt ‚dieses Ding ist fertig‘. Eine ‚definition of done‘ sollte (zumindest für Code) allen im Team klar sein. Beispiel:

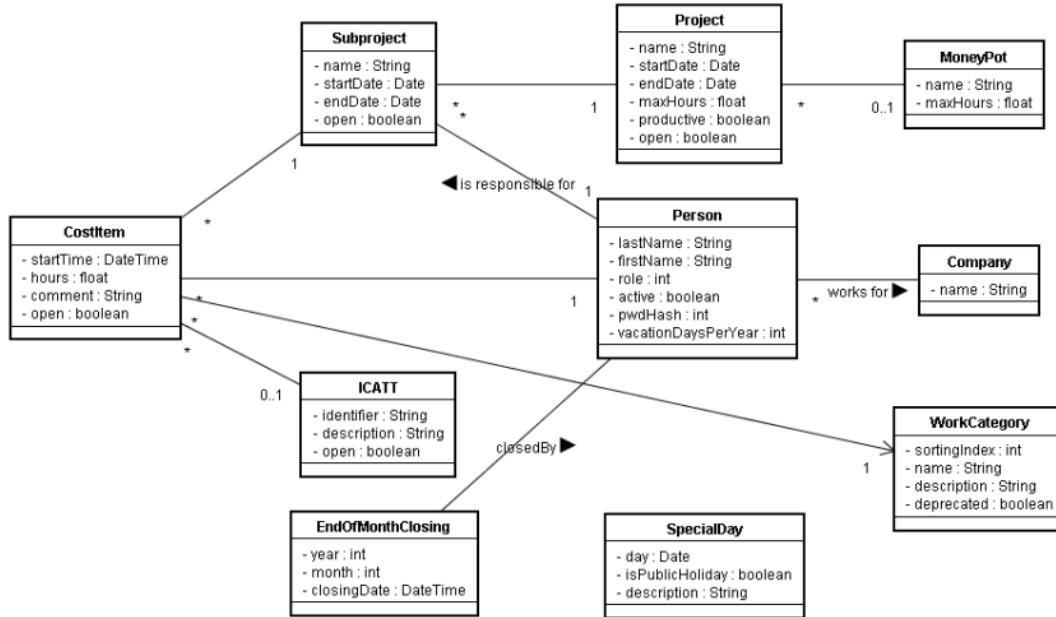
- Code übersetzt ohne Fehler und ohne Warnungen
- Unit Tests laufen fehlerfrei
- Test-Abdeckungsgrad wie definiert (in sep. Doku)
- Kein unaufgeräumter Code
- Alles unfertige ist markiert mit «TODO»
- Code ist wo nötig & richtig mit Kommentaren versehen
- Metrics, findbugs, ReShaper, strucutre101, STAN und andere eingesetzte Analysetools geben grünes Licht
- Vier-Augen-Prinzip: entweder pair Programming oder gerefviewet und Review dokumentiert.

# Software Architektur 2 – Modellierung im UML

## Vom Domainmodell zum Datenmodell

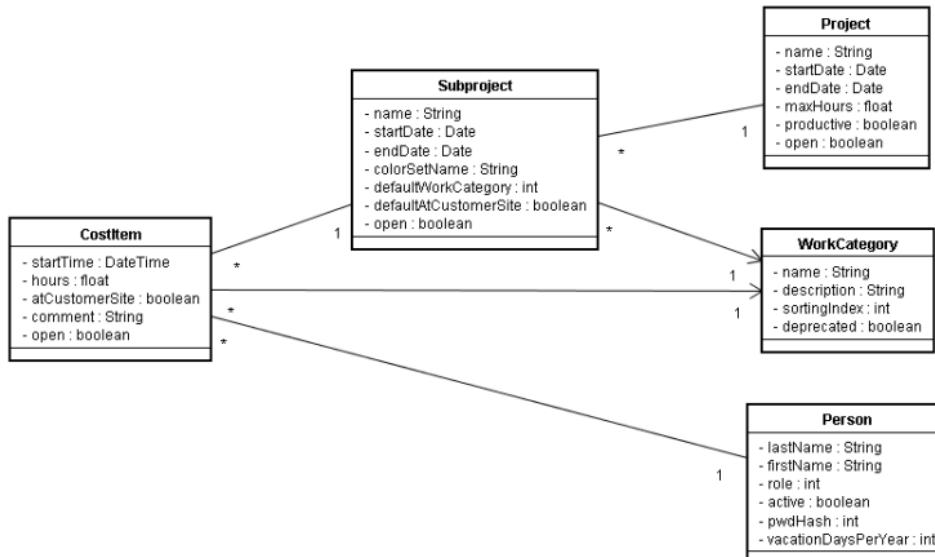
### Domainmodell

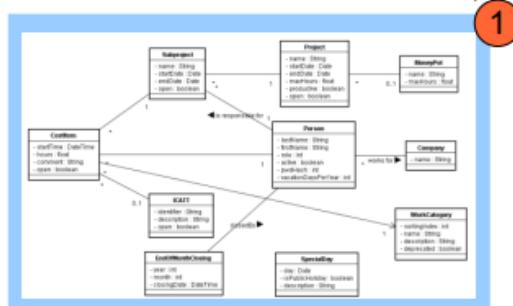
Entsteht sehr früh als Kommunikationsmittel und Glossar. Hauptsächlich aus Kunden-Sicht. Nicht optimiert (immer noch 1:1 Beziehungen, Vererbung,...).



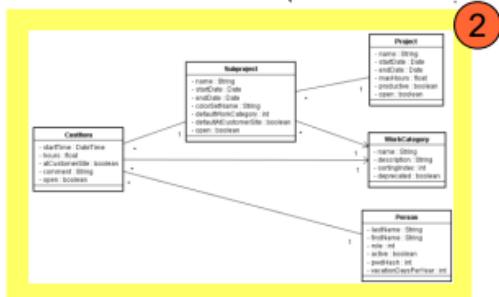
### Datenmodell

Ist idealerweise ein überarbeitetes, vereinfachtes Domainmodell. Reine Daten, ohne Operationen. Abbild der Datenhaltung (Persistenz), direkte Übersetzung in DB möglich.

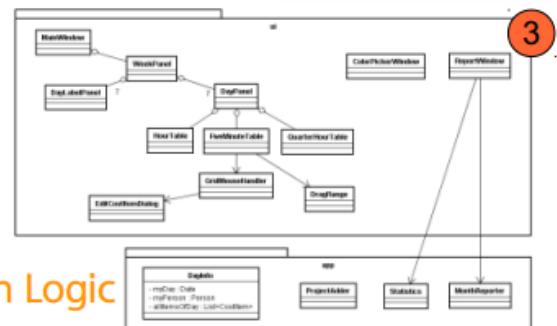




Domain-Modell



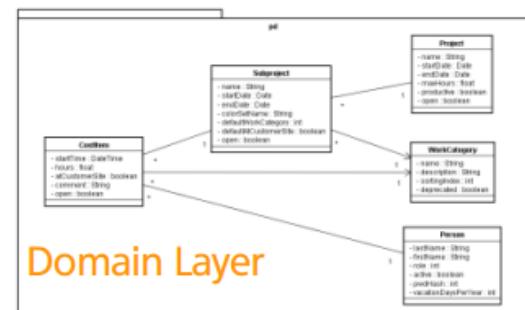
Datenmodell → 1:1 als eigene Schicht



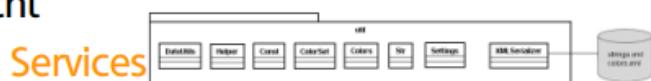
Application Logic



Data Access



Domain Layer

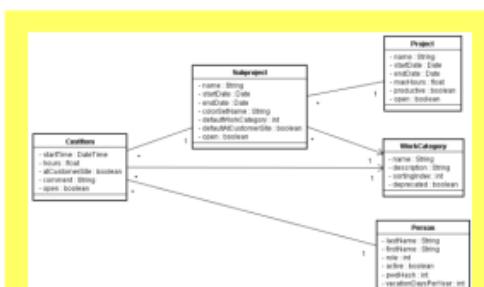


Services

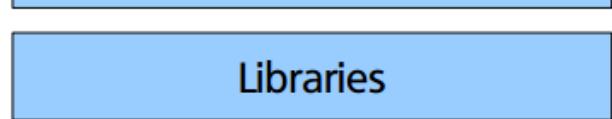
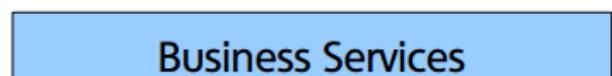
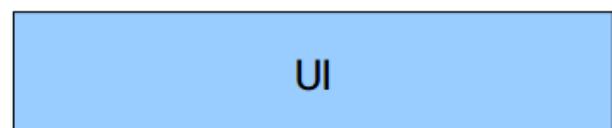
## Einfaches Architektur-Entwurfsmuster

Hier sitzen die UseCase Controller und die Facades

Kann sehr dünn sein, z.B. als O-R Mapper (JPA, Hibernate)



Hier sind die reinen Daten



## Was die Architektur beeinflusst

1. Deployment: Web? Mobile only?...
2. Performance
3. Security
4. Erweiterbarkeit
5. Anwendungsschwerpunkt: Mathematisch-analytisch, Grafik, Datenhaltung & -schieberei, Game, Helikoptersteuerung.
6. Usability (enge Benutzerführung).

Die Architektur ist vor allem durch die Q-Faktoren beeinflusst (Nicht nicht-funktionalen Anforderungen). Dazu zählen Performance, Security, Erweiterbarkeit und Usability.

## Kein Code-Design im UML

Warum machen wir an der HSR den direkten Sprung zu Patterns und der Architektur, warum gibt es kein «UML Design», wie z.B. bei Larman?

Viele bezeichnen das als sinnvollen Verfeinerungsschritt, als Zwischenschritt vor der Codierung.

Es gibt Tools, die können aus Diagrammen Code erzeugen (und umgekehrt):

- Klassendiagramme  $\Leftrightarrow$  Klassen (das kann jedes UML Tool)
- Klassendiagramme  $\Leftrightarrow$  Database Tables via DDL Statements
- Sequenzdiagramme  $\Leftrightarrow$  Methoden und Aufrufe (z.B. Enterprise Architect)
- Zustandsdiagramme  $\Leftrightarrow$  Abläufe (IBM Rhapsody)

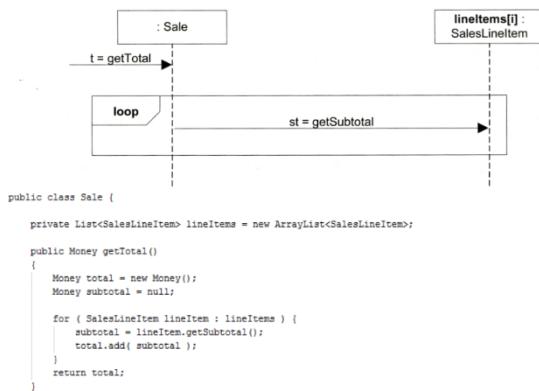
## Diagramme sind Kommunikationsmittel

UML  $\leftrightarrow$  Code 1:1 funktioniert nur in kleinen Beispielen. Bei allen praktischen (nicht winzigen) Projekten ist die Abbildung UML  $\leftrightarrow$  Code kaum hilfreich - weder die Generierung, noch Reverse Engineering, denn:

- Sobald ein Klassendiagramm zu gross wird, ist es nicht mehr kommunizierbar (siehe "max. A3" Regel), damit verliert es den Wert
- Es braucht trotz Klassen-, Zustands-, Sequenz-Diagrammen eine saubere Architektur und High-Level Dokumentation (mit der nötigen Abstraktion, nicht 1:1).
- Es gibt kein Diagramm-Tool ohne Code-Schnipsel.

## UML vs. Code (Zu nah am Code?)

Dies ist zu nahe am Code, nämlich 1:1.



Wenn Sie ein Diagramm sehen oder zeichnen, das sehr nahe am Code ist, dann sollten Sie sich fragen, ist das nötig? Ist dies hilfreich?

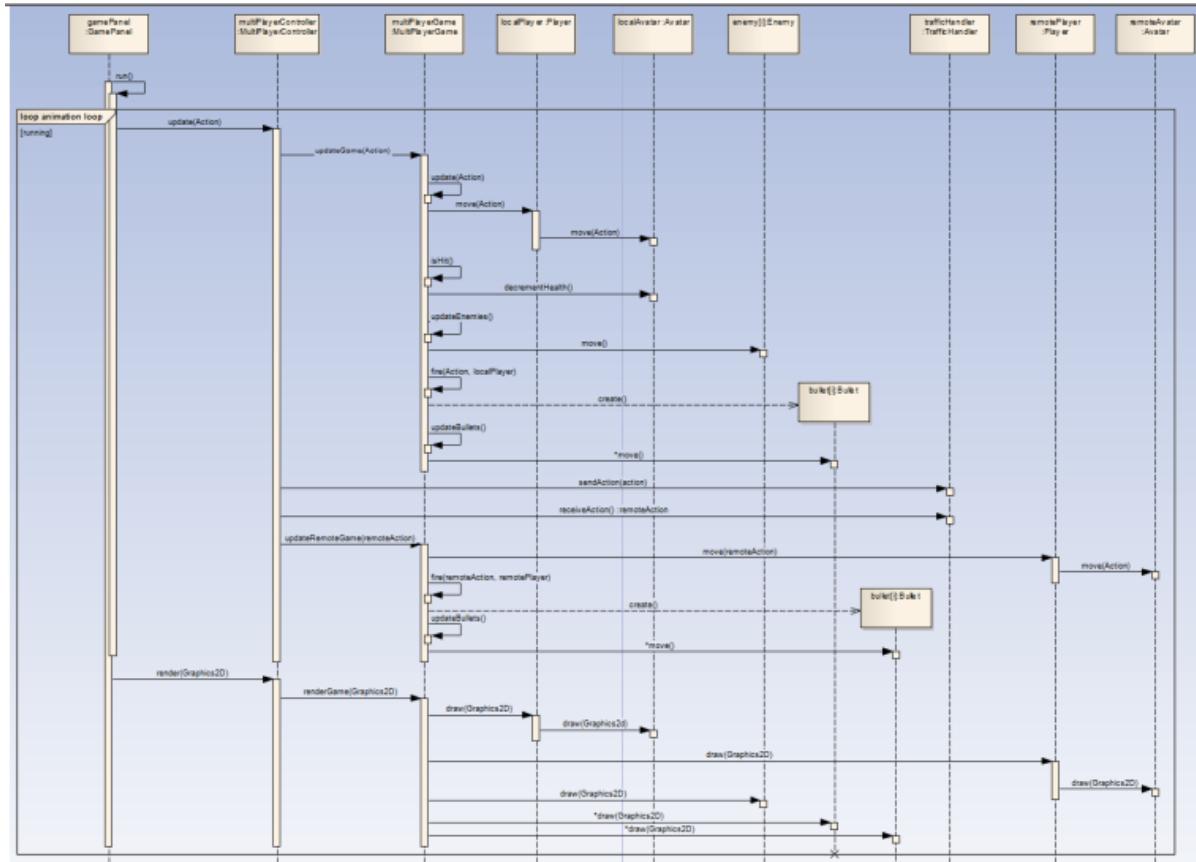
In den meisten Fällen wird die Antwort «Nein» sein.

Die häufigsten Ausnahmen zu dieser Regel – und damit potentiell nützlich – sind:

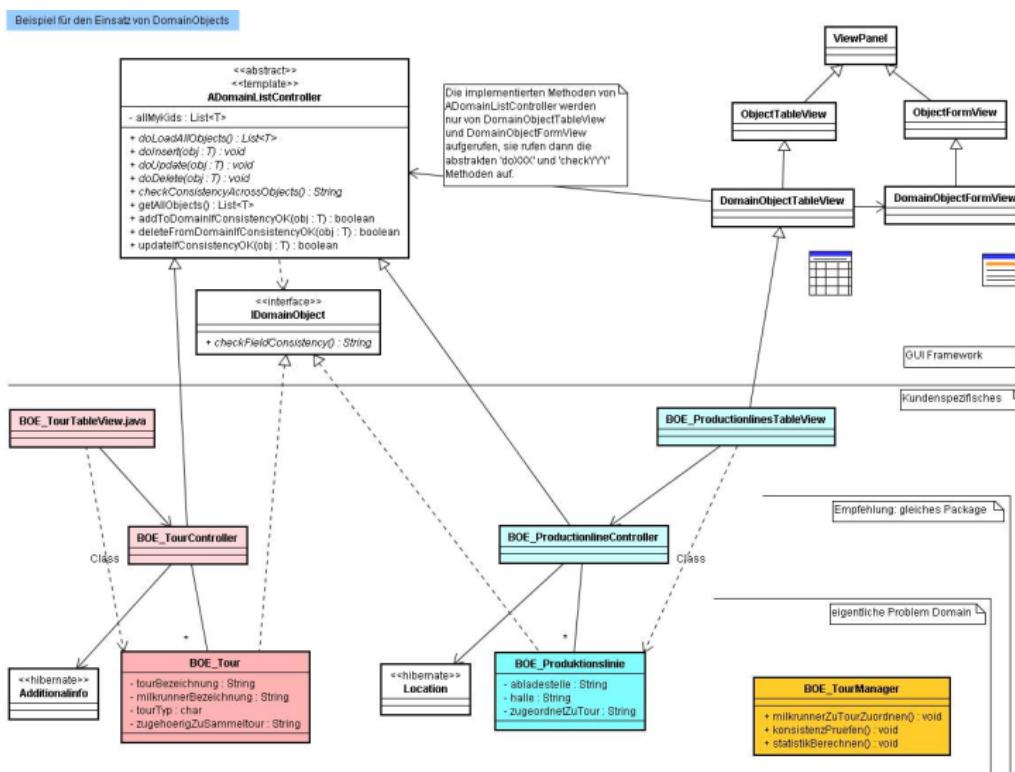
- Domain/Datenmodell (Übersetzung zu/von DDL): Zusammenhänge

- Sequenzdiagramme (1:1 Call-Sequenz): Zusammenspiel
- Zustandsdiagramm (State Machine): gute Basis für Workflows

Das Sequenzdiagramm zur Illustration eines komplexen Startup-Vorgangs kann sehr hilfreich sein – trotz/wegen der Details.



Ein Design-Klassendiagramm kann nützlich sein, ist aber eher die Ausnahme. Nützlich zum Beispiel, wenn es das komplexe Zusammenspiel mehrerer Klassen erklärt.



Die (objektorientierte) Analyse lebt zur Hauptsache von:

- Domain-Modellierung (Objekte, UML)
- Use Cases (nicht objektorientiert, wenig UML. UC Diagram, Activity Diagramm)
- Nicht-funktionalen Anforderungen (nicht objektorientiert; kein UML)
- Manchmal auch: Zustand-, Aktivitäts- und Sequenzdiagramme (UML)
- UI Design, Personas, Prototypen (nicht objektorientierte, kein UML)

## Objektorientiertes Design mit UML

- Package-/Schichtendiagramm ist immer gut (objektorientiert, UML)
- Deploymentdiagramme sind nützlich (nicht objektorientiert, UML)
- Sequenzdiagramme sind punktuell nützlich (objektorientiert, UML)
- Zustandsdiagramme sind punktuell nützlich (objektorientiert?, UML)
- Komponentendiagramme: evtl. nützlich (objektorientiert, UML)
- Design-Klassendiagramme sind von zweifelhaftem Wert, wenn Sie für OO-Design gebraucht werden.
- Design Patterns (objektorientiert, UML)

## Schwierigkeiten mit Diagrammen

- Was ist ein Diff (GIT) von zwei Diagrammen? Z.B. wie kommuniziert man eine Änderung?
- Wie macht man ein Merge von zwei Diagrammen? Ist nicht möglich → Konsequenz: an einem Diagramm kann immer nur einer arbeiten.
- Duplicate Code kann man finden. Duplicate Diagram Parts?
- Wie kann man ein Diagramm debuggen im Fehlerfall?
- Wartbarkeit? (Fehler finden, abändern, erweitern, Refactoring(!))

## Übersicht über alle Diagramme und Modelle

Name	Nah am Kunden	Nützliche Details	Einsatz-Möglichkeiten
Use Case Diagramm	ja	-	Übersicht, Scope/Funktionsumfang; zeigt auch (indirekt) Rollen und Berechtigungen
Domainmodell	ja	(ja)	Fast immer sinnvoll, zeigt Zusammenhänge; Basis für Glossar (ist oft > 80% des Inhalts)
Datenmodell	(ja)	ja	Fast immer sinnvoll, zeigt Zusammenarbeit; DDL kann generiert werden; meist sehr nahe am Domainmodell
Design-Klassenmodell	-	(ja)	Meist nicht sinnvoll, da 1:1 wie Code
Sequenz-Diagramm	-	ja	Manchmal sinnvoll, bei kooperierenden Teams und Klassen, z.B. komplexe Startup Sequenz; kann aus Code generiert werden
Zustands-Diagramm	ja	ja	Manchmal sinnvoll für zentrale Objekte (z.B. Product Life Cycle in Webshop) oder für Masch.-Steuerungen
Aktivitäts-Diagramm	ja	-	Manchmal sinnvolle Ergänzung für komplexe Use Cases
Package-Diagramm	-	-	reine Übersicht, zeigt Schichten, immer sinnvoll in Software Architecture Documentation (SAD)
Deployment-Diagramm	(ja)	(ja)	zeigt Verteilung auf Tiers, oft in mehreren Variationen; wichtig für Betrieb
Komponenten-Diagramm	-	(ja)	Kann – falls das System so entworfen wurde – die Übersicht zeigen und Schnittstellen definieren

# Feinheiten in der Datenmodellierung

## N:m – Beziehungen

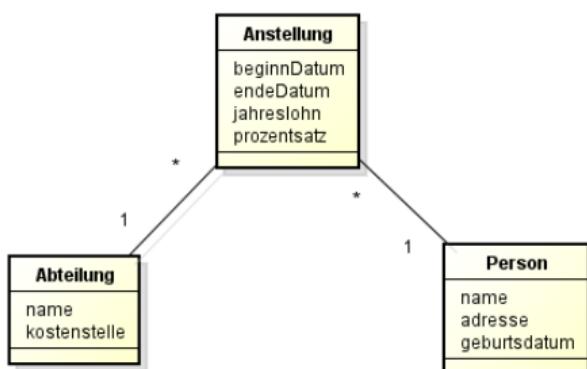
Eine n:m Beziehung in einem Domainmodell mit einer einfachen Assoziation zu modellieren ist grundsätzlich zulässig.



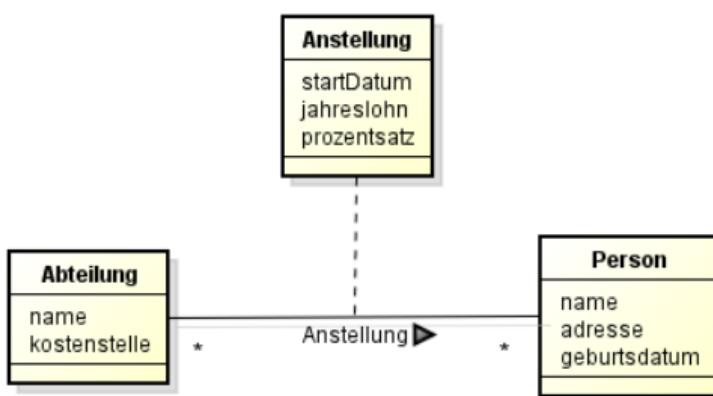
Einfach eine Assoziation zwischen zwei Klassen zu ziehen und deren beiden Enden mit einem Stern anzuschreiben ist aber sehr oft nicht richtig. Grund: in 97.865 % aller Fälle hat die Assoziation auch eigene Attribute und muss deshalb mit einer Zwischenklasse aufgelöst werden.

## Zwischenlösung A

Man nimmt eine ganz normale Klasse, die mit den zwei anderen Klassen mittels je einer n:1 - Beziehung verbunden ist. Das ist der Normalfall.



## Zwischenlösung B



Man nimmt eine sogenannte assoziative Klasse, die mit einer gestrichelten Linie direkt auf der n:m Beziehung sitzt.

Das ist ein Spezialfall mit einer besonderen Semantik: jedes Beziehungspaar darf nur genau einmal vorkommen (für Datenbanker: die Kombination der zwei foreign keys muss unique sein), d.h. wir können immer nur die aktuellste Anstellung speichern.

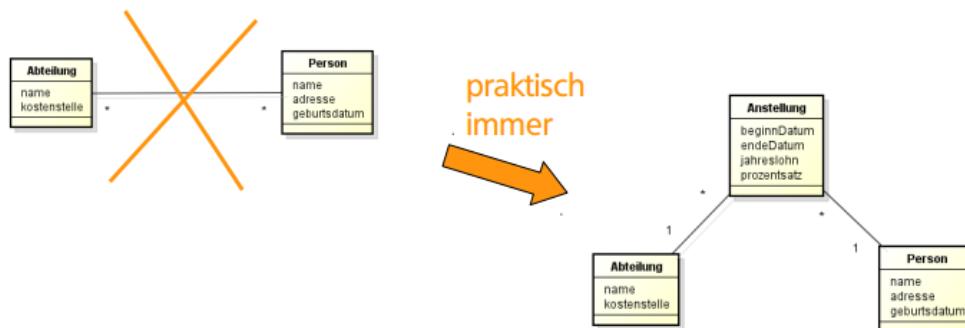
Zur Verdeutlichung im obigen Beispiel: wir haben ein Personalverwaltungssystem, in dem (unter anderem) die Personen mit ihren Anstellungsverhältnissen modelliert sind. Und zwar ist jede Person immer bei einer Abteilung angestellt, denn die Abteilungen verfügen über eine Kostenstelle.

Im Beispiel der Anstellungsverhältnisse darf es also bei der Modellierung b) (assoziative Klasse) nur genau einen Eintrag für jede Person geben, die für eine Abteilung gearbeitet hat. Damit ist also die Geschichtsschreibung nicht möglich ("in welchem Job hat Frau X im Januar 2008 bei Abteilung Y gearbeitet?"), noch ist es möglich, dass jemand seine Arbeitszeit auf zwei oder mehr Teilzeitjobs aufteilt.

Diese Einschränkung in der Semantik führt dazu, dass man die Variante b) sehr selten implementiert - nur in ganz speziellen Fällen kann es die richtige Lösung sein.

### Fazit für n:m Beziehungen

Eine n:m Beziehung in einem Domainmodell sollte eigentlich immer mit einer Zwischenklasse aufgelöst werden und die Zwischenklasse sollte bedeutungsvolle Attribute enthalten. Ausnahmen von dieser Regel sind möglich, aber sehr selten.



### 1:1 Beziehungen

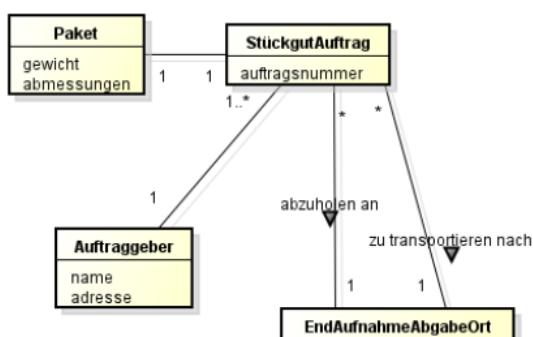
1:1 Beziehungen sind in einem Domainmodell OK, werden aber in einem darauffolgenden Datenmodell meist so aufgelöst, dass die zwei Klassen zusammengelegt werden.

1: 0..1 Beziehungen sind sowohl im Domainmodell als auch im Datenmodell OK.

**Domainmodell:** entsteht während Requirements & Analyse, versucht primär die Begriffswelt der Kunden zu modellieren. Nach Larman (und an HSR) keine Operationen, nur Daten.

**Datenmodell:** entsteht später als Entwurf für die zu speichernden Daten, i.d.R. als normalisiertes Modell für Datenbank-Tabellen.

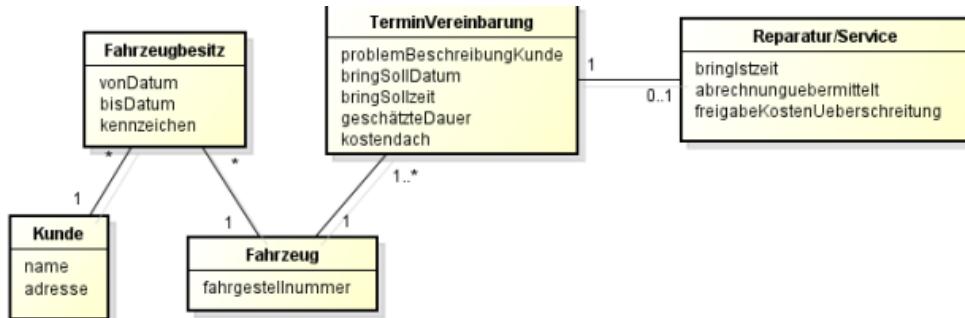
### Beispiel sinnvolle 1:1 Beziehung



1:1 zwischen Paket und Stückgut-Auftrag ist sinnvoll, weil die Entitäten aus zwei Domänen (Quellen/Systemen) stammen und zu zwei unterschiedlichen Zeiten und Zwecken erstellt wurden. Getrenntes Überleben ist möglich

1:0..1 Beziehungen sind viel häufiger als 1:1 Beziehungen.

- Pikettärztin – Piepsr
- Person – Führerschein
- Terminvereinbarung – Reparatur

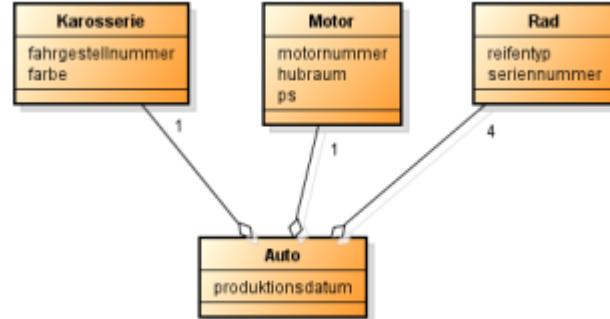


Beispiel falsche 1:1 Beziehungen



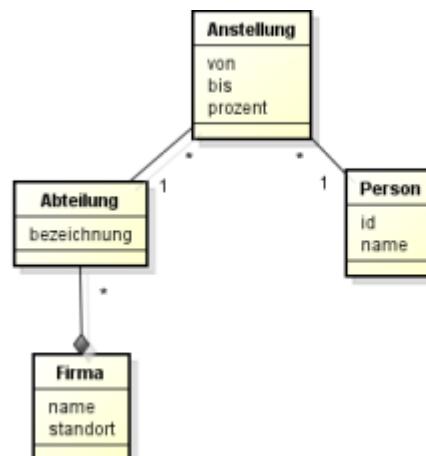
Teil-Ganzes Beziehungen

**Aggregation:** unabhängiges Überleben der Teile (weisser Diamant)



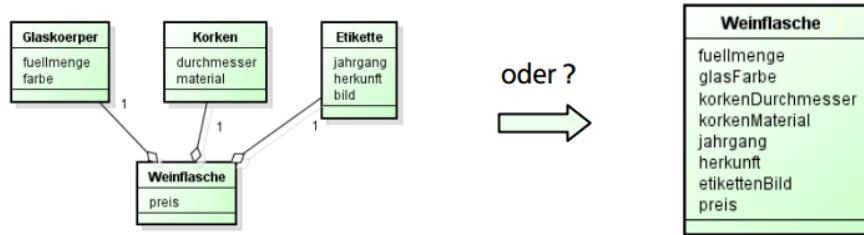
Diamant = i.d.R. Multiplizität 1

**Komposition:** beim Verschwinden des Ganzen existieren die Teile auch nicht mehr (schwarzer Diamant)



und die Anstellung?

Beispiel Weinfalsche für eventuell sinnvolle 1:1 Beziehungen.

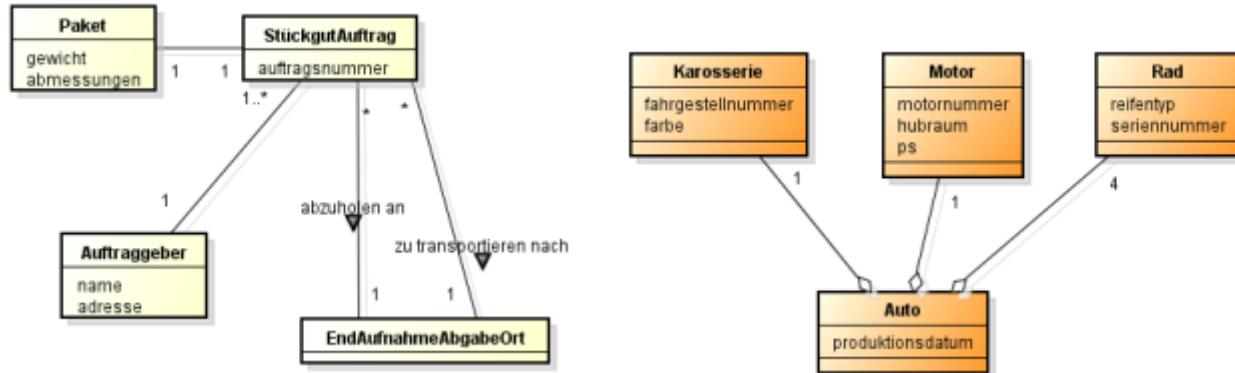


Es geht beides, es kommt ganz darauf an.

### Wann 1:1 Beziehungen?

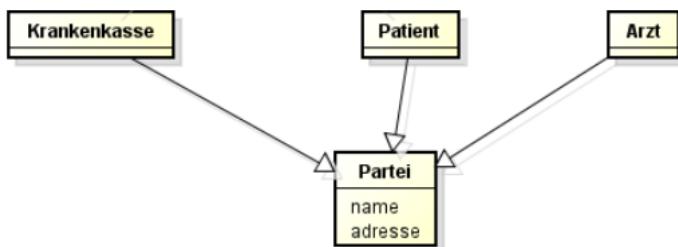
1:1 separat zu halten ist dann sinnvoll, wenn die Entitäten zu unterschiedlichen Zeiten und Zwecken erstellt wurden. Sinnvoll, wenn getrenntes Weiterleben möglich ist.

Sinnvoll, wenn z.B. das Zusammenbauen (Auto) überwacht bzw. protokolliert werden soll. Wenn mich nur das Endresultat interessiert, dann besser die Klassen zusammenlegen (Weinflasche?).



### Klassen ohne Attribute

Wenn Sie in einem Domain- oder Datenmodell eine Klasse zeichnen, welche keine Attribute hat, dann ist das ein starker Hinweis dafür, dass die Existenz dieser Klasse überdacht werden muss, insbesonders bei Vererbung oder bei 1:1 Beziehungen.

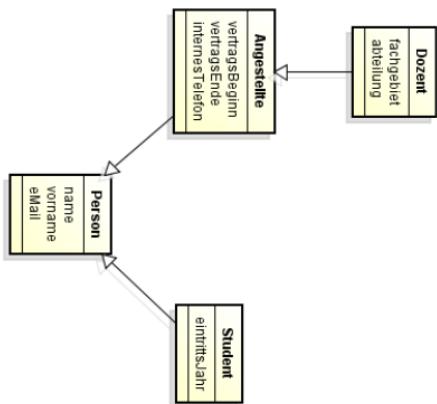


## Vererbung ohne Gemeinsamkeiten

Der Sinn der Verebung ist in der Hauptsache das zur-Vefügung-Stellen von Fähigkeiten/Methoden und Datenfeldern, welche den ableitenden Klassen zur Verfügung stehen. Wenn Sie eine Klassenhierarchie entwerfen, in der die Basisklasse keine gemeinsam ausführbare Methode und keine (oder sehr wenige) gemeinsamen Datenfelder hat, dann ist die Klassenhierarchie fragwürdig.

### Beispiel

Hier könnte eine gemeinsame Operation «schicke E-Mail» sein.



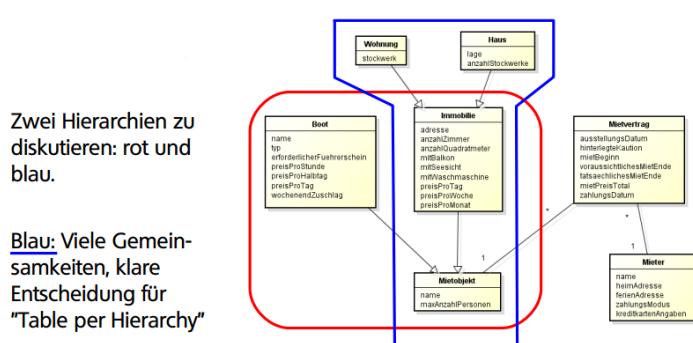
## Vererbung in relationaler DB abbilden

Es gibt zwei Möglichkeiten, eine Vererbungshierarchie in einer relationalen DB abzubilden:

- «Table per Class»: Hier wird pro modellierter Klasse eine Tabelle abgelegt. Beim Aufruf der Daten für eine abgeleitete Klasse muss ein Join gemacht werden. Dafür keine Platzverschwendungen.
- «Table per Hierarchy»: Hier macht man für die Vererbungs-Hierarchie nur eine Tabelle. Dabei werden nicht benutzte Felder leer gelassen.

Die Empfehlung in Kürzest-Form: Im Zweifelsfall «Table per Hierarchy», denn meistens haben gut entworfene Vererbungs-Hierarchien viele gemeinsame Felder, die Platzverschwendungen hält sich dann in Grenzen.

### Table per Class?



Zwei Hierarchien zu diskutieren: rot und blau.

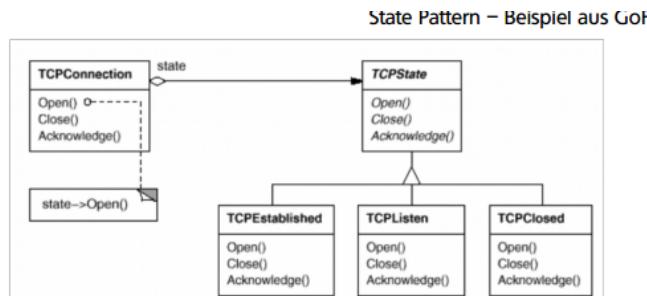
Blau: Viele Gemeinsamkeiten, klare Entscheidung für "Table per Hierarchy"

Rot: Boot und Immobilie haben so wenig gemeinsam, dass sie vielleicht besser nicht als Vererbungs-Hierarchie modelliert werden sollten.  
Andererseits spricht das Handling als Mietobjekt für eine Vererbungs-Hierarchie. Fazit: unentschieden.  
P.S. 'Boot' wurde erst nachträglich zur Wohnungsvermittlung hinzugefügt

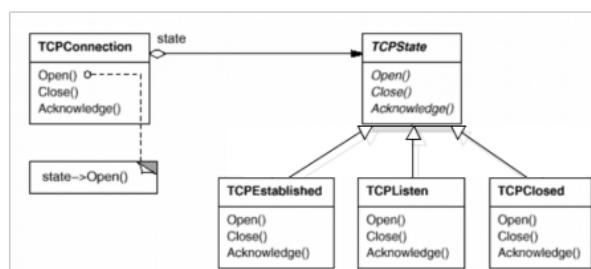
## Vererbung im Datenmodell?

Weil die Umsetzung von Vererbung in einer relationalen Datenbank so oder so nicht wahnsinnig schön ist, kommt man in Versuch, die Vererbung in einem Datenmodell nicht so häufig einzusetzen.

## Vererbungspfeile



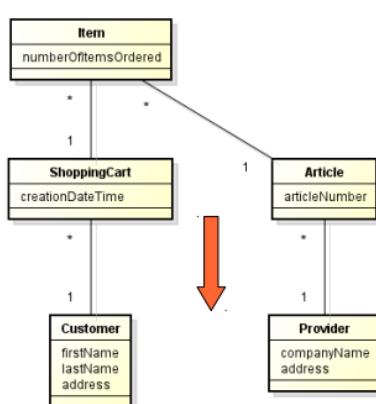
Beide Formen sind gleichwertig



## Gerichtete/ungerichtete Beziehungen

- In einem Domainmodell sind als Regel alle Beziehungen ungerichtet, man legt sich noch nicht fest.
- In einem Design-Klassendiagramm (Vorstufe von Code, was wir an der HSR aber fast nie einsetzen) kann man festlegen, wer wen sieht, indem man die Richtung der Beziehung mit einem Pfeil versieht.
- Man kann in einem Design-Klassendiagramm ein Ende einer Beziehung als 'nicht navigierbar' bezeichnen, indem man ein X darüber malt. Dann ist diese Richtung explizit nicht navigierbar

## Richtung von Beziehungen in Datenmodell



In einem Datenmodell sieht in einer 1:n Beziehung in der Regel die n-Klasse die 1-Klasse (foreign key bezieht sich auf primary key), aber nicht umgekehrt. In einer relationalen DB werden nur n:1 Beziehungen gespeichert. Wenn ich als 1-Klasse wissen will, welche n-Einträge sich auf mich beziehen, dann muss ich einen Query absetzen. Ein O-R Mapper kann auch beidseitig navigierbare Beziehungen machen, es wird aber aus Performance-Gründen nicht empfohlen.

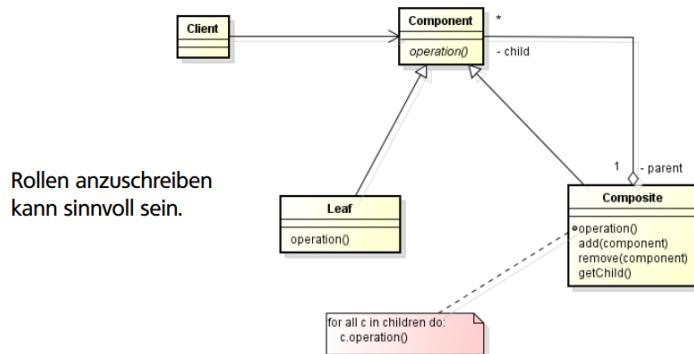
## Beziehungen und Rollen anschreiben?

UML bietet die Möglichkeit, eine Beziehung an drei Orten anzuschreiben:

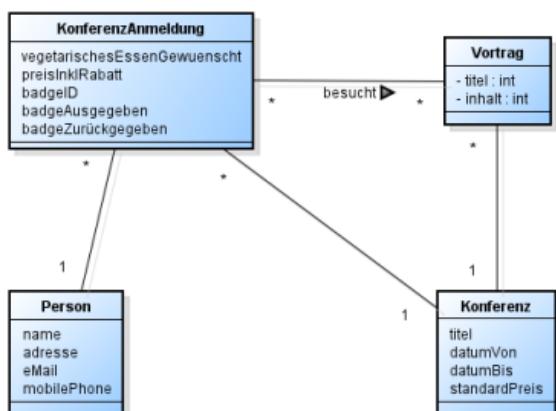
- In der Mitte kann man die Art der Beziehung anschreiben, zusammen mit einem Pfeil für die Leserichtung.
- An jedem Ende der Assoziation kann (je nahe bei einer Klasse) die Rolle, welche die Klasse in der Beziehung spielt, spezifiziert werden.

Oft sind aber die Beziehungen ganz einfach "hat ein" oder ein generelles "bezieht sich auf" (nicht "ist ein"! das wäre Vererbung) und dann lohnt sich ein Anschreiben nicht. Wenn die Namen der Klassen gut gewählt sind, sind sowohl Beziehungs-Typ als auch Rollen meist aus dem Kontext gut ersichtlich.

### Beziehung mit Rollen



### Wann Beziehungen anschreiben



Immer anschreiben, wenn es mehr als eine Beziehung zwischen zwei Klassen gibt. Zudem anschreiben, wenn der Sinn nicht sofort klar ist (es gibt auch noch Vortragende, hier nicht eingezeichnet.)

# Software Projekt-Management, Teil 1

## 1. Nichts vergessen dank Listen

Arbeitspakete für das Entwicklungs-Team (User Stories, Tasks) am besten in einem Ticketing/Backlog-Tool wie JIRA, MS TFS oder Redmine abspeichern. Hindernisse, die aus dem Weg geräumt werden müssen (Impediments), am besten als ToDo Liste für den Projektleiter führen. PL – Tätigkeiten wie «Ich muss noch den Ingenieur X fragen wegen...», «Peter hat seine Stunden noch nicht eingetragen», «Termin für zweiten Usability Test finden» am besten als ToDo-Liste.

## Prioritäten

Die einfachsten Priorisierungs-Stufen für Arbeitspakete bedeuten:

Prio **1** heisst: „Ich brauche dieses Feature unbedingt, ohne dies ist die Software nicht brauchbar. Und ich brauche dieses Feature gleich zu Beginn schon.“

Prio **2** heisst: „Ich brauche dieses Feature unbedingt, ohne dies ist die Software nicht brauchbar. Aber ich brauche dieses Feature nicht gleich am Anfang.“

Prio **3** heisst: „Es würde vielleicht auch ohne dieses Feature gehen; wäre aber schön, es zu kriegen.“

## 2. PL is the Keeper of the Scope

Wenn Sie als Projektleiter nur EINE Sache richtig machen wollen, dann diese: Halten Sie den Funktionsumfang des Projektes im Zaum.

Wenn Sie nämlich das nicht machen, dann können Sie ein noch so tolles Team haben, die sauberste Architektur entworfen haben, mit den besten Werkzeugen und Techniken arbeiten: Sie können dann das Projekt rauchen.

Wie beobachtet/misst man den Projektumfang (und zwar den Funktionsumfang, nicht die Kosten)?

### Funktionsumfang (Eckpunkte für die Beobachtung)

- Function Points (Anzahl Bildschirme/Masken, Anzahl Tabellen, Anzahl externe Schnittstellen)
- Anzahl UCs / story cards (mit einer durchschnittlichen Grösse)
- Anzahl Seiten Spezifikation
- Nur bedingt: Anzahl Zeilen Code.

### Scope Creep

Scope Creep = schleichende Erweiterung des gewünschten Funktionsumfangs. Gründe: nicht beachtete Features, Features die der Kunde als «selbstverständlich» bezeichnet, Gesetzesänderungen sowie Umgebungsänderungen (neue Tools/Releases).

2 % «scope creep» pro Monat sind normal, lassen sich nicht vermeiden. Das heisst aber auch, dass ein 20 Monate langes Projekt aller Wahrscheinlichkeit um 40 % umfangreicher wird als geplant.

**Faustregel:** Plane kein Projekt über mehr als neun Monate. Falls es doch grösser werden sollte: Plane in Teilen (z.B. 3x7 Monate) und verkaufe es auch so.

## Software ist Kommunikations-intensiv

Es ist von grossem Vorteil, wenn das Team an einem Ort ist. Schauen Sie, dass möglichst viele in ihrer Muttersprache kommunizieren können. Eine Teamwand (oder etwas Gleichwertiges im Intranet, z.B. Confluence) enthält wichtige Informationen für das Team. Viel und regelmässige (z.B. daily standup meeting) kommunizieren, falls nötig mit Skype, GoToMeeting, WebEx oder ähnliches. Und sonst «Management by walking around».

### 3. Gehen Sie täglich auf die Baustelle

Als Projektleiter sollten Sie regelmässig, am liebsten täglich, auf die Baustelle gehen und sich umschauen.

#### Auf der Software-Baustelle

##### Fragen

- Wer ist auf der Baustelle?
- Wer macht grad was?
- Wie weit sind wir?

Das sichtbare Endprodukt ist nicht die Baustelle, sondern:

- Programm-Code (plus config/localization files etc.)
- Versionskontroll-Werkzeug (git, svn), Wer macht commits ? Welche Unit wird wie oft angepasst?
- Bug Tracking Tool
- Tests, d.h. Testabläufe und -Protokolle
- Grafische Entwürfe
- Architektur-Dokumentationen (der Plan des Hauses, das gebaut wird)

### 4. Entfernung ist teuer

Ideal: Alle Arbeiten auf einem Stockwerk

- + 10 %: Alle arbeiten im selben Gebäude, aber nicht mehr auf einem Stock
- + 10 %: Alle arbeiten in derselben Ortschaft, aber nicht mehr im gleichen Gebäude
- + 10 %: Alle arbeiten im gleichen Land, ....
- + 10 %: Alle arbeiten in derselben Zeitzone
- + 10 %: nicht mehr gleiche Zeitzone, Zeitverschiebung mehr als 4. Stunden
- + 20 %: Nicht alle haben dieselbe Muttersprache (=Kommunikationssprache)

➔ Aufschlag von Aufwand

### 5. Übergaben sind teuer

Wenn man eine Arbeit jemand anders übergeben muss, dann kostet das einen erheblichen Aufwand und es ist mit Kommunikations-Fehlern zu rechnen. Der Aufwand und die Fehlerrate ist umso höher, je weiter auseinander man arbeitet. Der Aufwand und die Fehlerrate ist deutlich höher, wenn die Übergabepartner nicht in ihrer Muttersprache kommunizieren können.

#### Abhilfe

- Prinzip Pilot/Copilot für wichtige Rollen
- Der SW Architekt muss früh im Projekt involviert sein
- Örtliche Nähe begünstigt den Informationsaustausch
-

## 6. Die hohe Kunst: sichtbar machen

Software ist für Nicht-Programmierer unsichtbar. Es ist die Kunst des Projektleiters, das Unsichtbare sichtbar zu machen. Das heisst Fortschritt, Ideen & Konzepte (Architektur, Design, Schnittstellen).

Nutzen Sie Metaphern (Analogien) dem Ziel-Publikum angepasst. Zudem Diagramme. Dieses sind präsentierbar, genormt (UML), mit Abstraktion. Gut ist zu wissen, welche Diagramme sich wofür eignen.

### Nützliche Diagramme

- Context | very high-level Ansicht des Systems
- Domainmodell | Grösse, Aufteilung, Fortschritt
- Activity | für komplexe Use Cases
- State | für wichtige Entitäten als Basis für Workflow (z.B. Vertrag); HW-nahe Systeme (z.B. Ticketautomat im Parkhaus).
- Sequenz | für komplexe System-Interaktionen; für aufwendige Initialisierung
- Package/Schichten | SW Architektur Kern-Sicht
- Deployment

### Grafiken für den Fortschritt

- Burn-down chart
- Bug count
- Time to fix Bugs
- Build breaks
- Metriken (size, complexity,...)

## 7. Immer iterativ vorgehen

Die Agilen Methoden (XP, Scrum, RUP) haben sich bewährt weil:

- Kunde sieht immer wieder Resultate und kann Feedback geben
- Team hat Zwischenresultate, die ein Gefühl von "wir haben was geschafft!" vermitteln.
- Es ist einfacher, ein System schrittweise auf- und auszubauen (mit Refactoring) als eine "Big bang" Integration gegen Schluss zu machen.
- Fehler und Fehleinschätzungen („das will der Kunde so“) werden früher entdeckt
- Auch Requirements iterativ erfassen (mit OOA zwischendrin)
- Sprints: 2-3 Wochen Dauer, Kunden-Demos: alle 2-3 Monate

### Kurze Iterationen



Empfehlung: Iterations-Länge von zwei oder drei Wochen

Immer timeboxed, d.h. man schliesst die Iteration zum vorgesehenen Datum ab, egal wie weit man gekommen ist. Unfertige Arbeitspakete werden auf die nächste Iteration übertragen.

Falls Schwierigkeiten: de-scope, d.h. Funktionalität weglassen oder auf später verschieben.

## Feedback vom Kunden

Wenn Sie mit dem Kunden kommunizieren, dann sollten Sie auch zuhören, nicht nur präsentieren.  
Regelmässig dem Kunden Teilresultate zeigen und Rückmeldungen einholen.

## 8. Inspect – Adapt

Projekte sind nicht wie abgeschossene Pfeile, sondern eher wie bei einer langen Reise: Ständig muss man auf veränderte Umstände reagieren. Man macht zwar eine Planung, legt die groben Etappen fest, einigt sich auf ein Budget und einen Zeitplan, aber dann:

- Ist ein Flieger verspätet (Nebel in Frankfurt)
- Ist das Hotel überbucht (oder mit Baustelle, oder mit Kakerlaken..)
- Wird ein Teilnehmer krankt
- Ändert der Wechselkurs

Also immer schön achtgeben und ggf. reagieren.

## Projektkontrolle

### Budgetkontrolle plus Fortschrittskontrolle

- Management by walking around; Anwesenheit schafft Verbindlichkeit
- Kein Abtauchen (seit drei Wochen programmiert Joachim...)

## Weiter:

- Daten als Feedback sammeln und publizieren, z.B. Fehlerrate, Burn down chart (= completed user stories).
- Keine reine Instrumenten-Navigation - klar auch die Instrumente (=KPIs, Key Performance Indicators) nutzen.
- Risiken identifizieren und evtl. präventiv reagieren.
- Hot spots identifizieren (SW-Teile die Probleme machen; Features die Probleme machen).
- Reviews machen: Requirements, (Architektur-)Dokumentation, Code.

## Projekt Geschichtsschreibung

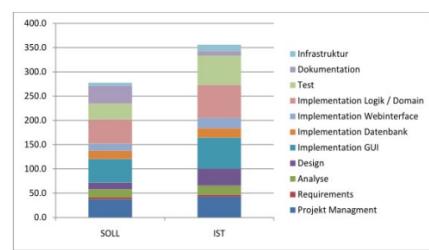
Regelmässig schreiben

Meilensteine (erreicht/verpasst), Anzahl Mitarbeitende, Ausgegebenes Geld, Schwierigkeiten  
Besondere Vorfälle

### Tipp: Ablegen in Ordnern, chronologisch:

- Kompletter Export Redmine als CSV zu jedem Review und am Schluss
- Screenshots von Meilensteinen und Schwierigkeiten
- Falls vorhanden: Sitzungsprotokolle (v.a. mit Kunden (Scope))

## Projekt-Review nach Abschluss



«Lessons learned» (Soll-Ist-Vergleiche)      Zeit, Geld, Funktionalität,  
Qualität, MA-Zufriedenheit, Kundenzufriedenheit, Fehlerquellen

Abbildung 2: Ist / Soll Vergleich auf die einzelnen Arbeitspakete

## 9. Daily Build und Branches

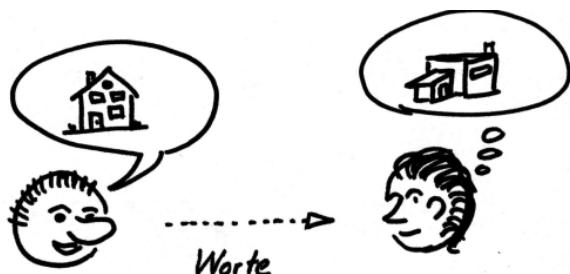
Wenn ein Projekt den ‚Daily Build‘ nicht mehr (fehlerfrei) hinkriegt, dann ist das ein Alarmzeichen.

“To Branch or not to Branch” Nicht zuviele (zentrale) Seitenzweige, denn Verzweigungen können den Punkt „wie weit sind wir?“ stark verwedeln.

Abtauchen ist teuer („Alex ist schon seit vier Wochen an der neuen SAP Schnittstelle dran und ich habe schon lange nichts mehr von ihm gehört“), egal ob auf eigenem Branch oder nicht.

Mindestens Daily Build + Regelmässige Releases: 2-3 Wochen für Team/Sprints; 2-3 Monate für Kunden

## 10. Versehen, was der Kunde will



Wir müssen den Kunden gut verstehen, denn wir bauen SW mit Nutzen für den Kunden, nicht für uns.

Optimal: Kunde im Team. Gut: “Kunden-Proxy” im Team.  
Schlecht: der Kunde hat keine Zeit

### Anforderungen, Requirements

Sie müssen den Kunden WIRKLICH verstehen – nur so gelingt es.

Wie zeigt man einem Kunden, dass man ihn verstanden hat? Feedback einholen!

- Verständlich geschriebene Anforderungen, d.h. im Minimum Use Cases, Nicht-funktionale Anforderungen, Domainmodell/Glossar.
- Prototypen
- Grafische Entwürfe

Wichtig: frühes ‚Scoping‘, d.h. grobes Abstecken des Umfangs.

Tipp: beschreiben Sie auch, was nicht gemacht werden wird.

### Frage die Endkunden, nicht einen Proxy

Wenn Sie ein Laborinformationssystem entwerfen, dann fragen Sie die Endnutzer: LaborantInnen.

Und nicht den CEO oder Ingenieur der Firma, die das System bei Ihnen bestellt.

aber:

„Das Hundefutter wird nicht vom Hund gekauft“

... dann wird für's Management halt etwas flashiges eingebaut ;-)

### Partner sein, Vertrauen schaffen

Es kommt ein Punkt, wo Sie dem Kunden sagen können: „Ich denke, wir haben jetzt wirklich verstanden was Sie wollen und benötigen.“ Und der Kunde sagt: „Ja ich denke, die Jungs haben wirklich verstanden, was ich will.“

Das ist ein Meilenstein im Projekt. Machen Sie eine Zeremonie (Meeting) draus. Meist ca. End of Elaboration, d.h. erst nach ca. 20% der Zeit. Ab jetzt kann man eigentlich zu bauen anfangen (Arbeiten verteilen). Holen Sie sich die Zustimmung (wenn möglich auch die Präsenz bei der entscheidenden Sitzung) von möglichst hoher Stelle

## 11. So früh wie möglich, so formal wie möglich

Je früher man formal wird, desto mehr Zeit spart man sich später. Dazu gehören UML und halb-formale Texte (UCs, Risikoliste, Tests).

### Beispiel

Sobald Sie anfangen, das Domainmodell zu zeichnen, tauchen Fragen auf: Sie merken, dass Sie etwas noch nicht wissen. Wenn diese Fragen so früh auftauchen (und beantwortet werden) ist das ein grosser Zeitgewinn. Stellen Sie sich vor, es wird programmiert, dann getestet, und erst jetzt merkt man, dass etwas nicht stimmt

### Bilder können ein Projekt retten

Sichtbar machen: Bilder für Konzepte, Grafiken für Fortschritt und wenn möglich Grafiken mit definierter Bedeutung. (z.B. UML).

### Grad der Schriftlichkeit

Eigentlich: Kommunikation im Entwicklerteam und mit dem Kunden am besten mündlich, zusammen mit Beispielen, Bilder, Diagrammen.

Schriftlich nur für Leute, die weit entfernt sind, oder in dem Falle, wo eine Informations-Quelle nur begrenzt mündlich zur Verfügung steht (Kunde, Wissens-Quelle) und viele Adressaten erreicht werden müssen (man kann nicht 25x hintereinander dasselbe erzählen, ist auch nicht kosteneffizient).  
Schriftlich auch dann, wenn über einen längeren Zeitraum etwas kommuniziert werden soll.

Der Grad der Schriftlichkeit hängt primär vom Entwicklungsprozess, von der Team-Grösse und von der geographischen Verteilung der Personen ab.

### Eingefrorene Spezifikation

*Walking on water and programming to specification are easy – if both are frozen.*

Warnung: Stabile Requirements von Beginn bis Ende sind eine Illusion, denn SW ist immer was Neues - Altbekanntes würde fixfertig gekauft.

Häufig weiss der Kunde nicht so genau, was er will. Erst wenn er das (fast) fertige Produkt sieht, kann er sich eine Meinung bilden.

Und: Die SW wird im Einsatz die Prozesse der Firma verändern und damit wiederum Änderungen an der Software auslösen.

## 12. Die vier Projekt-Variablen

Kosten/Aufwand, Zeit, Funktionalität sowie sowie die Qualität.

Einfach zu verstehen und zu messen sind die Kosten sowie die Zeit. Schwierig ist die Funktionalität und echt schwierig ist die Qualität.

### Was tun, wen überbestimmt?

- Immer zuerst die Funktionalität reduzieren, einfache Lösungsvorschlägen (s. auch 80/20 Regel).
- Wenn das nicht geht: Zeitplan verlängern.
- Nur in den seltensten Fällen funktioniert es, wenn man mehr Leute auf das Projekt ansetzt.  
Brooks' Law: „Adding manpower to a late software project makes it later“. F.P. Brooks, 1975 (!)
- Bei der Qualität Kompromisse einzugehen (sloppy coding) zahlt sich nie aus.

### Zusammenfassung

#### *Anforderungen*

10. Verstehen, was der Kunde will
11. So früh wie möglich so formal wie möglich
12. Die vier Projekt-Variablen
2. PL is the Keeper of the Scope

#### *Software ist Kommunikations-intensiv*

3. Gehen Sie täglich auf die Baustelle
4. Entfernung ist teuer
5. Übergaben sind teuer
6. Die hohe Kunst: sichtbar machen

#### *Praktische Tipps*

1. Nichts vergessen dank Listen
7. Immer iterativ vorgehen
8. Inspect - Adapt
9. Daily Build und Branches