

# ZUSAMMENFASSUNG

## Lernziele

- Aktuelle Themen aus der Software Entwicklung kennen, anwenden und kritisch einschätzen
- Werkzeuge und Techniken professioneller Software Entwicklung kennen und anwenden
- Pragmatische Prinzipien der Software Entwicklung kennen und anwenden

## Unterlagen / Bücher

- Keine, nur Vorlesungsfolien und Übungsaufgaben

## Lerninhalte

Der Lerninhalt dieses Kurses richtet sich stark an die aktuellen Themen der Software Engineering. Die Themen werden jedes Jahr auf deren Relevanz geprüft und bei Bedarf angepasst. Es wird daher stark empfohlen, die Prüfung im gleichen Semester zu belegen. Hier ist eine Liste der in der Vergangenheit behandelten Themen:

- Project Planning
- Project Automation
- Test Driven Development
- Pragmatic Software Engineering Practices
- Error Handling Design
- Concurrency Design
- Design by Contract
- Software Architecture
- Code Smells
- Design Patterns
- Refactoring
- Software Metrics
- Software Reviews
- Cost Estimation (Aufwandschätzung)
- Performance Profiling
- Agile Software Development
- Programming in the functional style
- Scripting Languages
- Software Failure Analysis

<b>PROJEKTPLANUNG .....</b>	<b>8</b>
DREI THEMENKREISE .....	8
AGIL, ITERATIV.....	8
<i>Ein Beispiel-Projekt mit 18 Iterationen .....</i>	8
<i>Checkliste End of Elaboration.....</i>	9
<i>Checkpoints / Meilensteine .....</i>	9
<i>Empfehlungen für Engineering-Projekte .....</i>	9
<i>Meilensteine und Reviews im Eng.-Projekt.....</i>	10
<i>Meilensteine und Versionen in Redmine .....</i>	10
<i>Vorschlag für Umsetzung der Meilensteine in Redmine-Zielversionen.....</i>	10
ANFORDERUNGEN: USE CASES ETC.....	10
<i>Use Case Diagramm .....</i>	10
<i>Use Cases eines fiktiven Beispieles.....</i>	10
<i>Nicht funktionale Anforderungen .....</i>	11
<i>Domain-Model.....</i>	11
ANFORDERUNGEN → ARBEITSPAKETE.....	11
DIAGRAMME UND DOKUMENTE .....	13
PROJEKTDOKUMENTATION .....	14
«END OF ELABORATION» = WENDEPUNKT .....	14
<b>PROJEKTAUTOMATATION .....</b>	<b>15</b>
WIE ENTWICKELN WIR SOFTWARE?.....	15
<i>Idee 1 – Build Skript.....</i>	15
<i>Was wir möchten (Wunschliste)?.....</i>	15
THE BEGINNING.....	16
<i>Der Beginn der Build Automatisierung – GNU Make (Imperativ).....</i>	16
<i>Apache Ant (Imperativ).....</i>	16
<i>Deklarative Builds mit Apache Maven.....</i>	16
<i>Post-Maven Tools .....</i>	17
THE FUTURE .....	17
<i>Zusammenfassung der Vorteile .....</i>	17
<i>Continuous Integration .....</i>	18
<b>SOFTWARE ENGINEERING PRACTICES .....</b>	<b>20</b>
REQUIREMENTS PRACTICES .....	20
1. <i>Dig for Requirements.....</i>	20
2. <i>Make quality a requirement.....</i>	20
3. <i>Deal with changes.....</i>	21
DESIGN PRACTICES.....	21
4. <i>Don't repeat yourself.....</i>	21
5. <i>Achieve orthogonality.....</i>	21
6. <i>Design to test.....</i>	22
IMPLEMENTATION PRACTICES.....	23
7. <i>Fix broken windows .....</i>	23
8. <i>Refactor early and often .....</i>	23
9. <i>Program deliberately.....</i>	23
VERIFICATION PRACTICES .....	23
10. <i>Test rigorously.....</i>	23
11. <i>Perform Reviews .....</i>	24

DEFENSIVE PROGRAMMIERUNG.....	24
<i>Schutz vor ungültigem Input.....</i>	25
<i>Prüfung von Routinen-Input.....</i>	25
<i>Abfangen ungültiger Fälle .....</i>	25
<i>Beispiel .....</i>	25
FEHLER-BARRIKADEN.....	26
FEHLERBEHANDLUNGS-TECHNIKEN .....	26
<i>Korrektheit versus Robustheit.....</i>	26
<i>Lokale vs. Globale Behandlung .....</i>	26
<i>Global Exception Handler .....</i>	26
<i>Risiko der Fehlerbehandlung.....</i>	27
<i>Exceptions vs. Assertions.....</i>	27
LOGGING.....	28
ANWENDUNG .....	28
<i>Wichtigkeit bei jedem Projekt.....</i>	28
<i>Error Handling Policy.....</i>	28
<i>Exceptions Policy.....</i>	29
<i>Assertions Policy.....</i>	29
<i>Logging Policy.....</i>	29
<b>DESIGN BY CONTRACT (DBC).....</b>	<b>29</b>
EINFÜHRUNG .....	29
<i>Beispiel Stack.....</i>	30
<i>Das Interface ist nicht die ganze Geschichte .....</i>	30
ZIELE.....	31
PRECONDITIONS, POSTCONDITIONS, CLASS INVARIANTS .....	31
<i>Preconditions.....</i>	31
<i>Postconditions.....</i>	31
<i>Invarianten .....</i>	31
<i>Contracts und Vererbung.....</i>	31
GUTE VERTRÄGE, ANWENDUNG IN AGILEM UMFELD .....	32
UNTERSTÜTZUNG VON DBC IN PROGRAMMIERSPRACHEN .....	32
<i>In Java.....</i>	33
<i>In .NET 4.....</i>	33
DISKUSSION.....	34
<i>Probleme.....</i>	34
<i>Concurrency.....</i>	34
<i>DbC und Unit Tests .....</i>	34
<i>Fazit .....</i>	34
<b>GROSSE ARBEITEN AUFTEILEN .....</b>	<b>34</b>
STORY SPLITTING – AUFTEILUNG VON ZU GROSSEN PROJEKTEN .....	34
<i>Aufteilung nach Kunden-Domäne .....</i>	35
<i>Aufteilung nach Geschäfts-Prozessen .....</i>	35
<i>Reihenfolge? Wichtigkeit.....</i>	35
<i>Projekt-Teillieferungen nach Rollen .....</i>	36
<i>Projekt-Aufteilung im Domain-Modell .....</i>	36
<i>Projekt-Teillieferungen geografisch .....</i>	36
STORY SPLITTING – AUFTEILUNG VON ZU GROSSEN ARBEITSPAKETEN.....	37

## Software Engineering 2

Von der Basis-Version zum Voll-Ausbau.....	37
Generell: Basis-Funktionalität ++ .....	38
Achtung: User Story nicht gleich User Case.....	39
STORY MAPPING.....	40
Epics – User Stories – Tasks.....	40
Die kleinste Einheit – das Arbeitspaket.....	40
Nutzen von Story Mapping .....	40
<b>TESTEN IN GRÖSSEREN PROJEKTEN.....</b>	<b>41</b>
UNIT TESTING – KLAR!.....	41
MICROTESTING .....	41
Facking & Mocking .....	42
Integrationstest .....	42
Schwachstellen von Microtesting allgemein .....	43
WANN IST GENUG GETESTET?.....	43
Testabdeckung.....	43
Ist das gut: «83 % Test-Abdeckung»? .....	45
KONSISTENZ-TESTER FÜR DIE DATENBANK .....	45
BUILD SERVER.....	46
Daily Build / Continuous Integration (CI) .....	46
Wahrscheinlichkeit für einen Build Break.....	47
SERVER SETUP .....	47
Server-Umgebungen .....	47
Automatisierte Migration .....	48
Test-Daten .....	48
DevOps .....	48
PROBLEME MIT TESTDATEN.....	48
<b>AUFWANDSCHÄTZUNGEN .....</b>	<b>49</b>
ERFOLGSQUOTEN VON SOFTWARE-PROJEKTEN .....	49
HORROR-GESCHICHTEN .....	49
International.....	49
Software-Projekte in CH-Verwaltungen .....	49
Scope Creep .....	49
AUF DIE GRÖSSE KOMMT ES AN .....	50
Grössenordnungen zusammengefasst.....	50
Wie wächst der Kommunikationsaufwand? .....	51
Nicht-lineare Faktoren nach McConnell.....	51
Weitere negative Grösseneffekte .....	51
Schlüsse.....	51
Faustregeln .....	52
WARUM VERHAUEN WIR UNS BEIM SCHÄTZEN?.....	52
AUFWAND – DIE DREI WICHTIGSTEN FAKTOREN.....	52
Einflussfaktoren nach ISBSG.org.....	52
TOP-DOWN SCHÄTZUNGEN .....	53
Beispiel Mehrfamilienhaus .....	53
ALGORITHMISCHE SCHÄTZUNGEN .....	53
FUNCTION POINTS .....	54
BOTTOM-UP SCHÄTZUNGEN .....	54
Gantt Charts .....	54
TOP-DOWN VS. BOTTOM-UP .....	55

## Software Engineering 2

DEFINITION - „ZEILEN CODE“ .....	55
<i>Zeilen pro Monat</i> .....	55
<i>Zeithorizont – ein Jahr</i> .....	55
BEISPIEL FÜR GROB-ABSCHÄTZUNG.....	56
<i>Eine grobe Rechnung</i> .....	56
<i>Mögliche Vorgehen</i> .....	56
<i>Warum nicht in 1 Jahr?</i> .....	57
METRIKEN ZUM FORTSCHRITT .....	57
<i>Warum Sigmoid?</i> .....	57
<b>SOFTWARE-METRIKEN.....</b>	<b>58</b>
ZIELE.....	58
EINFÜHRUNG – WAS SIND PRODUKTMETRIKEN? .....	58
<i>Software-Prüfung</i> .....	58
<i>Produkt- und Projektmetriken</i> .....	58
<i>Qualitätsmodell nach ISO 9126</i> .....	58
<i>Metrik</i> .....	58
<i>Metrik</i> .....	58
<i>Beispiel für Einstufungsniveau</i> .....	58
<i>Ziele von Metriken</i> .....	59
CODEMETRIKEN .....	59
<i>Traditionelle Sourcecode-Metriken</i> .....	59
<i>Die Zyklomatische Zahl (McCabe Metrik)</i> .....	59
METRIKEN FÜR OBJEKTOIENTIERTE SOFTWARE .....	60
<i>Sourcecode-Metriken für objektorientierte Software</i> .....	60
TOOLS FÜR SOURCECODE-METRIKEN.....	61
<i>Eclipse Metrics Plugin (continued)</i> .....	61
<i>State Of Flow Metrics</i> .....	62
<i>Structure 101</i> .....	62
<i>STAN – Structure Analysis for Java</i> .....	62
<i>Checkstyle</i> .....	62
<i>Findbugs</i> .....	62
<i>Sonar – umfassendes System zur Überwachung der Qualität mit Metriken</i> .....	62
DYNAMISCHE ANALYSE .....	63
<i>Code Coverage beim Testen</i> .....	63
NUTZEN VON CODEMETRIKEN .....	63
<i>Was bringt es? Qualität!</i> .....	63
<b>CODE REVIEWS .....</b>	<b>64</b>
<i>Wie funktioniert ein Review?</i> .....	64
<i>Konkretes Vorgehen</i> .....	64
<i>Was kostet es, einen Fehler zu fixen?</i> .....	64
<i>Eine wahre Geschichte</i> .....	65
<i>Was kostet es, einen Fehler zu fixen, in einer regulierten Umgebung?</i> .....	66
CODE ANALYSIS TOOLS .....	66
PRAKТИSCHE REGELN FÜR CODE REVIEWS.....	66
1. <i>Vorbreiten, einladen</i> .....	66
2. <i>Vorbedingungen checken</i> .....	67
3. <i>Code Review durchführen und protokollieren</i> .....	67
4. <i>Nacharbeiten</i> .....	68
5. <i>Eventuelle Nachkontrollen</i> .....	68

Software Engineering 2 .....	
GRAD DER FORMALITÄT DER REVIEWS .....	68
REQUIREMENTS REVIEWS .....	68
<i>Ziel der Requirements</i> .....	68
<i>Ziele des Requirements Review</i> .....	68
<i>Beispiel-Szenario für Requirements-Review</i> .....	69
<i>Proben über's Kreuz (Cross Checks)</i> .....	69
ARCHITEKTUR REVIEWS .....	69
<i>Beispiel-Fragen Architektur-Review</i> .....	70
KOSTEN/NUTZEN VON CODE-REVIEWS .....	70
<b>PERFORMANCE-MESSUNGEN</b> .....	<b>71</b>
PERFORMANCE-PROFILING (WHITE BOX) .....	71
LAST-MESSUNGEN (BLACK-BOX) .....	72
WIEDERHOLTE PERFORMANCE-MESSUNGEN .....	72
<b>PROBLEME MIT TESTDATEN</b> .....	<b>72</b>
MONITORING MIT DASHBOARDS .....	72
<b>USABILITY TESTING</b> .....	<b>73</b>
EINFÜHRUNG .....	73
TESTE DIE SOFTWARE, NICHT DIE ANWENDER! .....	73
DURCHFÜHRUNG .....	73
<i>Was kommt dabei heraus?</i> .....	73
<i>Wann und Wie?</i> .....	73
VARIANTEN .....	74
<b>SW ARCHITEKTUR – TEIL 3</b> .....	<b>74</b>
DEFINITION „WAS IST SW ARCHITEKTUR“ .....	74
VERSCHIEDENE ARCHITEKTUR-TYPEN .....	74
DIAGRAMME FÜR DIE ARCHITEKTUR-BESCHREIBUNG .....	76
INHALTE DES SOFTWARE ARCHITECTURE DOCUMENT (SAD) .....	77
<i>A – Dokumentation Umfeld</i> .....	77
<i>B – Dokumentation Technische Struktur</i> .....	77
ERFOLGSKRITERIEN FÜR SW-ARCHITEKTUR .....	77
<b>SCRUM ZUM ZWEITEN</b> .....	<b>78</b>
VORAUSSETZUNGEN FÜR SCRUM .....	78
<i>Was mir in Scrum fehlt</i> .....	78
PRODUCT OWNER VS. PROJEKTLTEITER .....	79
<i>Gesunder Backlog</i> .....	79
<i>Backlog Refinement</i> .....	79
<i>Gantt Charts</i> .....	79
IMPEDIMENTS .....	80
VALUE FOR THE CUSTOMER .....	80
MVP – MINIMUM VIABLE PRODUCT .....	80
<i>Was zuerst umsetzen, was später?</i> .....	80
MANAGEMENT WILL WASSERFALL? .....	81
<b>SOFTWARE PROJEKT-MANAGEMENT, TEIL 2</b> .....	<b>82</b>
TECHNISCHE SCHULD (TECHNICAL DEBT) .....	82
<i>Copy/Paste als technische Schuld</i> .....	82

<i>Technische Schuld als Zahl</i> .....	82
<i>Vier Arten von „technical debt“</i> .....	82
DER BLINDE PROJEKTEITER .....	83
<i>Ausgangslage (hypothetisch)</i> .....	83
<i>Andere Ausgangslage (real)</i> .....	83
DAS UNSICHTBARE SICHTBAR MACHEN .....	84
<i>Trends sind besser als absolute Werte</i> .....	84
SOFTWARE ENGINEERING – DREI PERSPEKTIVEN .....	85
<b>PROVING PROGRAMS CORRECT</b> .....	<b>86</b>
TESTING HAS ITS LIMITS.....	86
PROVING PROGRAM CORRECTNESS .....	86
THE SIMPLE IMPERATIVE LANGUAGE .....	86
HOARE TRIPLES.....	87
<i>Inference Rules of Hoare Logic</i> .....	87
<i>A simple Hoare logic proof</i> .....	87
<i>Weakest Precondition Style Proof Nr. 1</i> .....	88
<i>Automated Program Verification</i> .....	88
<i>Weakest Precondition Style Proof Nr. 2</i> .....	89
DAFNY .....	89

# Projektplanung

## Drei Themenkreise

Je mehr Leute an einem Projekt mitarbeiten, umso mehr müssen Sie organisieren und planen, insbesonders wenn nicht alle Leute an einem Ort sitzen.

Wie plant & organisiert man ein Software-Projekt?

Dieses Kapitel deckt folgende drei Themenkreise ab:

- Agil, RUP, Scrum, Wasserfall, iterativ?
- Von Use Cases zu Arbeitspaketen
- Diagramme und Dokumentationen in Software-Projekten

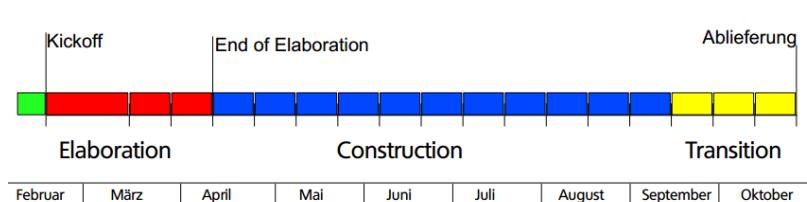
## Agil, iterativ

Ganz klar: mit dem Wasserfall-Modell (auch V-Modell) gewinnt man keinen Blumentopf, da sind sich praktisch alle einig. Heute gilt:

- Agil (Scrum; eXtreme Programming XP)
- Iterativ (RUP, Iterative Development, Spiral Development...)



## Ein Beispiel-Projekt mit 18 Iterationen



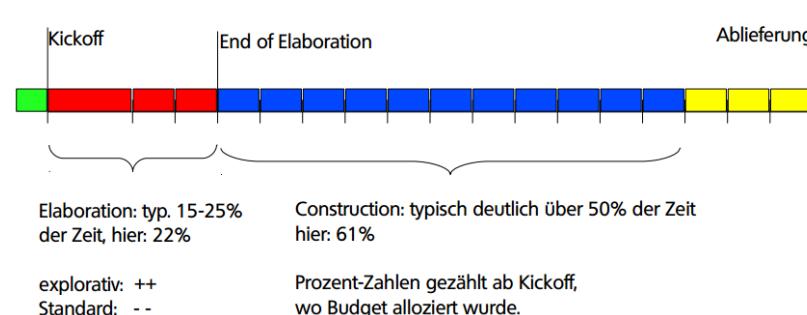
Total 8.5 Monate (37 Wochen) in 4 Phasen. Eine Inception von 10 Tagen, eine Elaboration von 8 Wochen in 3 Iterationen, eine Construction von 11 Iterationen à 2 W. und ein Transition von 3 Iterationen à 2 Wochen.

## Anzahl Personen



Personal wird vor allem in der intensiven Construction-Phase benötigt. Wichtig ist das der Chief Architect von Kickoff bis mindestens Mitte Projekt involviert ist.

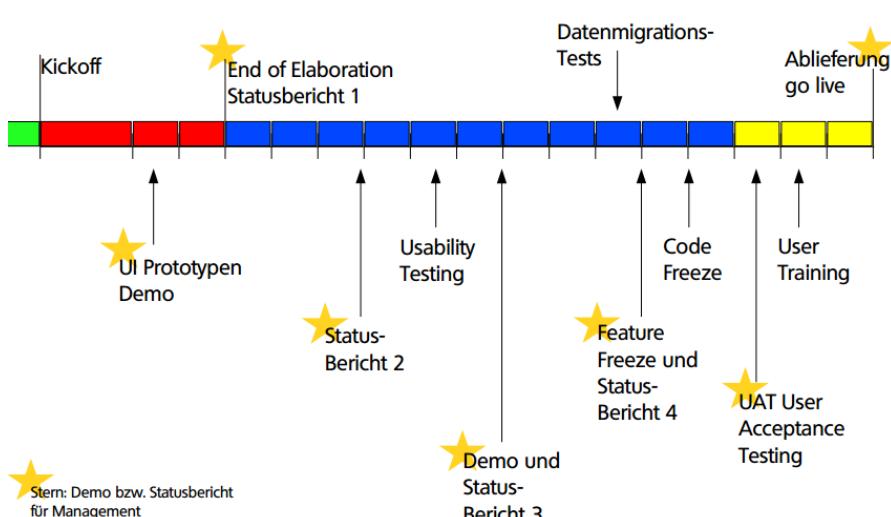
## Zeitaufteilung



Achtung: Die Prozent der Zeit ist nicht gleich die Prozent der Kosten.

- ✓ Anforderungen (Requirements): Haben wir den Kunden verstanden? Funktionsumfang (Scope) ist abgesteckt durch UCs, Domain Model, nicht-funktionale Anforderungen
- ✓ User Interface Design: Entwürfe gemacht, dem Kunden gezeigt; wenn möglich Clickable Prototypes plus Grafik-Entwürfe (Farben, Schriften)
- ✓ Software Architecture: Entwurf steht, Subsysteme und Interfaces definiert, Prototypen gemacht (Durchstich durch alle Schichten).
- ✓ Entwicklungs-Werkzeuge und Methoden: definiert und komplett aufgesetzt (IDE, version control system/server, build server, unit testing, static code analysis tools inkl. Konfig., DEV-TEST-PROD Server, ticketing & bug tracking, user story writing/proofing, etc.).
- ✓ Genaue Aufwandschätzung: Liste der Arbeitspakete

### Checkpoints / Meilensteine

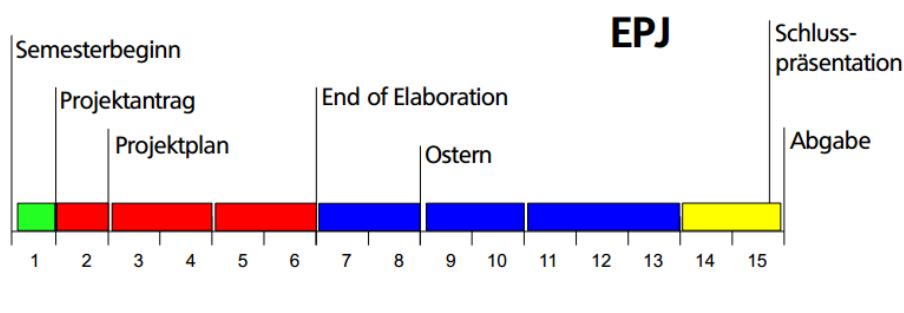


### Meilensteine dokumentieren

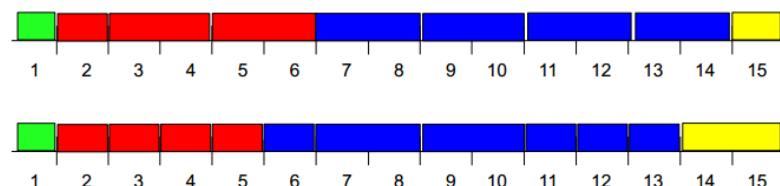
Die Iterationsplanung, d.h. die Beschreibung der Meilensteine "was läuft wann, bzw. was können wir wann zeigen", darf ruhig in einem Dokument erfolgen.

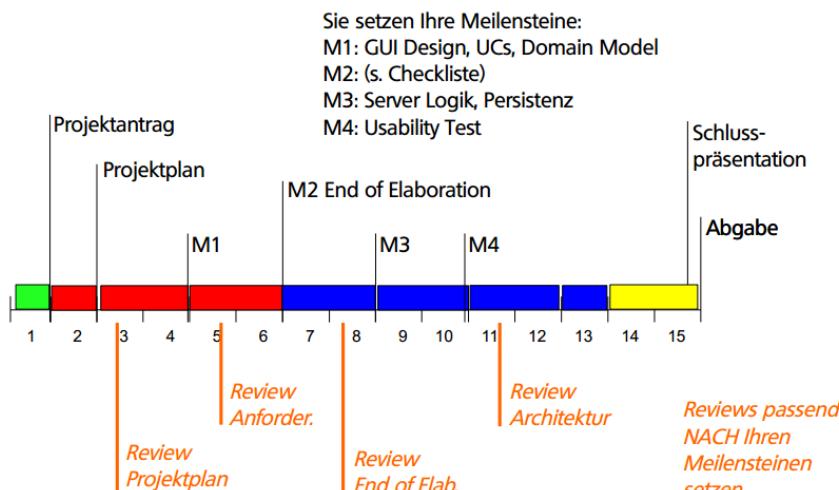
Sobald Sie die Meilensteine gesetzt haben, können Sie diese in Ihrem Arbeitspaket-Verwaltungssystem (Redmine) als Software-Versionen definieren.

### Empfehlungen für Engineering-Projekte



Es gibt verschiedene Varianten für das Engineering Projekt. Version 3 ist eher einem realen Projekt entsprechend mit etwas mehr Risiko. Grundsätzlich gilt die Version 2.





## Meilensteine und Versionen in Redmine

Vorschlag für Umsetzung der Meilensteine in Redmine-Zielversionen.  
Die Arbeitspakete werden dann einer dieser Versionen zugeordnet.

P.S.: Die Reviews finden i.d.R. nach Ihren entsprechenden Meilensteinen statt.

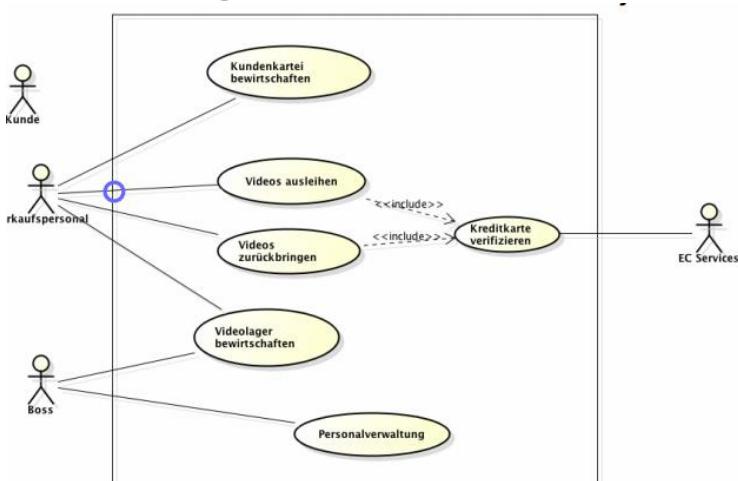
## Redmine-Versionen

- MS1: Projektplan
- MS2: Anforderungen und Analyse
- MS3: Ende Elaboration
- Alpha
- Beta (Code Freeze?)
- End of Construction
- Optional Features

## Anforderungen: Use Cases etc.

Use Cases sind Beschreibungen klar aus der Benutzersicht, also gut mit dem Kunden kommunizierbar. Use Case beschreiben die Funktionalität im Kontext (z.B. warum, was in welcher Reihenfolge und wie oft?) Es gibt verschiedene Detaillierungsgrade. Von Brief (3 Zeilen), über casual (eine halbe Seite) bis hin zu fully dressed (ein bis zwei Seiten mit vorgegebenen Abschnitten). Use Cases können auch gut zum Abstecken des Funktionsumfangs (Scope) verwendet werden.

## Use Case Diagramm



Ist gut für eine Übersicht, was im Scope ist (was nicht), wer darf was machen.

Ein Systemsequenzdiagramm würde den blauen Kreis abbilden.

## Use Cases eines filterten Beispieles

Insgesamt 15 Use Case. 4 wichtige, komplexe Use Cases. 5 mittel-komplexe Use Cases und 6 CRUD.

```

UC01: User wants to create new product
UC02: PM enriches product data
UC03: CRUD Users
UC04: Import of product data
UC05: Assign product to categories
UC06: Assign articles to product
UC07: Build category tree
UC08: CRUD categories
UC09: Establish product-product relation
UC10: Create print layouts for categories
UC11: CRUD products
UC12: CRUD orders
UC13: Move part of category tree
UC14: CRUD articles
UC15: CRUD return shipments

```

## Nicht funktionale Anforderungen

Use Cases beschreiben nur die Funktionalität. Nicht-funktionale Anforderungen ergänzen Use Cases und Domainmodell. Typische Beispiele für nicht-funktionale Anforderungen sind Mengen- und Qualitätsanforderungen:

- Performance ("Antwortzeiten für eine Produkt-Suche bei 100'000...")
- Mengengerüst ("50'000 Artikel; 200 gleichzeitige Besucher; ...")
- Sicherheit (Firewalls, Intrusion Detection, Logging, Plausibility checks...)
- Erweiterbarkeit ("Später automatischer Import von Lieferanten-Daten...")
- Benutzerfreundlichkeit ("Produkt-Manager Einführung in 2 Tagen...")

## Domain-Modell

Das Domain-Modell ist das Komplementär zu den Use Cases. Ein Domain-Modell beschreibt, was wir uns während der Laufzeit des Programmes merken, und was wir evtl. darüber hinaus speichern. Vereinfacht ausgedrückt: Use Cases = Dynamik, Domain Modell = Persistenz.

## Anforderungen → Arbeitspakete

Jetzt haben wir die Anforderungen (zumindest grob) definiert. Warum macht man jetzt Arbeitspakete? Was muss in Arbeitspaketen drinstehten? Wie gross sollen Arbeitspakete sein? Wie viele Arbeitspakete müssen entstehen?

### Inhalt von Arbeitspaketen

ID, Titel, Kurze Beschreibung (In der Form «Als AAA möchte ich BBB, weil CCC»), Akzeptanz-Kriterien, Schätzung des Aufwands, Priorität für Kunden, Geleistete Stunden (Zeiterfassung), Status und Arbeits-Kategorie.

### Grösse der Arbeitspakete

Maximal 50 – 70% von dem, was eine Person in einer Iteration schafft, damit die Chance gross ist, dass das Arbeitspaket innerhalb der Iteration fertig wird, denn Arbeitspakete dürfen nur «nicht angefangen», «angefangen» oder «fertig» sein. Und wenn ein Arbeitspaket nicht fertig ist, kommt es in die nächste Iteration. Im Falle des Engineering-Projektes mit Iterationen von 2 Woche und 17 Arbeitsstunden pro Person ist die empfohlene maximale Grösse 10 Arbeitsstunden.

### Anzahl Arbeitspakete im Projekt

In einem EPJ gibt es typischerweise ca. 80 – 160 Arbeitspakete. 4 Credits für das ganze Projekt ergibt 120 Arbeitsstunden pro Person. Bei einem Team von 4 Personen wären es dann 480 Arbeitsstunden pro Team.

### Arbeitspakete organisieren

Arbeitspakete müssen an einem Ort zentral gespeichert sein. Arbeitspakete müssen von allen eingesehen und editiert werden können. Zudem müssen Sie priorisierbar sein (sie werden häufig herumgeschoben). Die Pakete sollten sowohl Schätzungen als auch Ist-Zeiten (Zeiterfassungen) enthalten.

### Workflow Arbeitspakete

Definiere Rollen und Zustände/Übergänge für Arbeitspakete in einem Zustandsdiagramm-

Grundsätze zur Arbeitsaufteilung

Wenn man Arbeit auf verschiedene Teammitglieder verteilen will

(insbesonders wenn das Team noch geografisch verteilt ist), dann **muss man wissen, was der Kunde will/braucht**, bevor man die Arbeitsaufteilung machen kann. Wenn man Arbeit auf verschiedene Teammitglieder verteilen will (insbesonders wenn das Team noch geografisch verteilt ist), dann **muss die Architektur allen Beteiligten klar sein, bevor** man die Arbeitsaufteilung machen kann. Das heisst, dass man bis Ende Elaboration eng zusammen arbeitet (kleines Team an einem Ort) und dass die Arbeitsaufteilung erst nach Ende Elaboration klappt: erst danach kann man verteilt loslegen.

### Planung pro Iteration

Genug Arbeitspakete, damit alle im Team während der nächsten Iteration beschäftigt sind. Genau soviele Arbeitspakete wie die Schätzungen zulassen, dass sie auch innerhalb der Iteration fertig werden. Innerhalb des Teams werden die Arbeitspakete eigenverantwortlich zugeordnet, evtl. auch dynamisch verteilt.

### Regeln bei der Planung

Der Entwickler schätzt den Aufwand für die Arbeitspakete, während der Kunde die Arbeitspakete priorisiert. Und nicht umgekehrt. Die einzige Ausnahme sind die architekturelevanten Arbeitspakete. Diese können vom System-Architekten in bestimmte Iterationen gesetzt werden, weil sonst das System nicht schlau gebaut werden könnte. Dies muss aber den Kunden erklärt werden und der Kunde muss es absegnen.

### Fallstricke bei Arbeitspaketen

Nicht nur generische Arbeitspakete aufführen. Generische Arbeitspakete sind solche, welche in jedem Projekt vorkommen, z.B. 'Domainmodell machen', 'Use Cases schreiben'. Nicht-generische sind z.B. 'Funktionalität für Speichern des Warenkorbs definieren', oder 'Entwurf Level-Editor'. Auch Arbeitspakete für unproduktive Tätigkeiten erstellen, wie z.B. für Besprechungen, Einrichten des Servers, Schreiben des Testplans (werden manchmal separat als 'Chore' geführt, nebst 'Ticket' und 'Bug'). Formulieren Sie ihre Arbeitspakete (wie generell auch alle Anforderungen) 'abhakbar', d.h. tabellarisch und so, dass sie abgehakt werden können => klein genug und gute Akzeptanzkriterien. Und nicht vergessen, Arbeitspakete schreiben kostet auch Zeit.

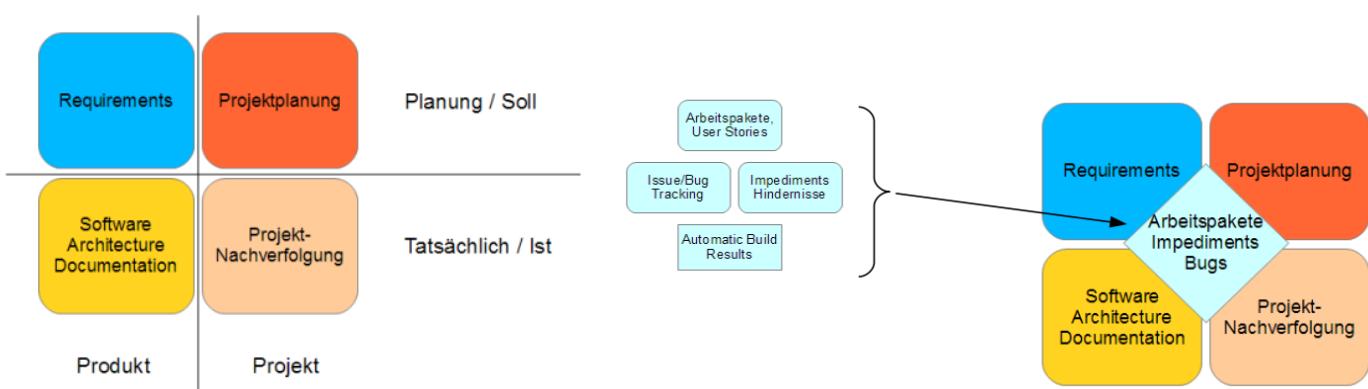
## Diagramme und Dokumente

Erste Spalte Inception, zweie Spalte Elaboration und dritte Spalte Construction.



Was gehört in welches Dokument?

Diese Frage ist zweitrangig. Hauptsache wir haben alles dokumentiert, was es wert ist, festgehalten zu werden. Wichtig ist m.E. nur die Zuordnung zu einem der vier Quadranten der Dokumentation. Tatsächlich sind es aber 5 Dokumenten-«Kübel».



## Projektdokumentation

Sie schauen, dass von all Ihrem Code und Ihren Dokumenten laufend Backup gemacht wird (git/SVN Backup), damit nichts verloren geht. Klar. Zusätzlich solle aber bei jedem Meilenstein die Doku eingefroren und gespeichert werden. Das heisst:

- Ein PDF von jedem relevanten Dokument mit Datum im Dateinamen
- Exporten von allen Web Tools, insbesondere Redmine (alle Arbeitspakete, Bugs und Zeitaufschreibungen als CSV Dateien mit Datum im Namen).
- Screenshots von allen wichtigen Funktionen (mit Funktion und Datum im Dateinamen)

### Es ist ihr Gedächtnis

Speichern Sie die Projektdokumentation wie oben beschrieben, denn Sie wissen nicht, was Sie in Zukunft aus den alten Projektdaten (zum Vergleich, zum Abschätzen) herausholen wollen.

Deshalb: speichern Sie alles mögliche über das Projekt in einem lange haltbaren und portablen Format (TXT, CSV, PDF), damit Sie später alles noch lesen, nachvollziehen und nachberechnen können.

Beispielsweise: „Wieviele Stunden haben wir damals für die Vorbereitung und Durchführung der Usability Tests gebraucht? Wieviele Probanden waren dabei? Was für Szenarien haben wir benutzt? Was ist dabei herausgekommen?“

Projektdokumentation ist Ihr Erfahrungsschatz

### «End of Elaboration» = Wendepunkt

Der Zeitpunkt 'End of Elaboration' ist auch ein Wendepunkt für die Dokumentation und Kommunikation. Vor 'End of Elaboration' liegen die Haupt-Anstrengungen bei der Dokumentation darauf, zu zeigen, dass man den Kunden verstanden hat. Diese Dokumentation ist zum grössten Teil für die Kommunikation mit dem Kunden gedacht, sollte also auch in seiner Sprache (z.B. Deutsch) gehalten sein. Nach 'End of Elaboration' ist der Fokus auf dem Bauen der Lösung, d.h. die Dokumentation, die entsteht, ist hauptsächlich für die Entwickler. Da kann es sein – bei einem ausgelagerten Entwicklungsteam – dass die Dokumentation nach Ende Elaboration überwiegend auf Englisch gemacht werden muss.

### Diagramme sind Kommunikation

Diagramme sind oft besser als Worte, da Sie präziser/formaler sind. Sie müssen eine normierte Bedeutung (UML) haben. So spart man sich Erklärungen. Zudem sollten Sie kommunizierbar sein d.h. beschränke Grösse. (A3 Druck oder 3x FullHD Bildschirm).

### Je früher desto besser

So früh wie möglich, so formal wie möglich (Datenmodell, Zustandsdiagramme). Zudem so früh wie möglich so komplett wie möglich (Use Cases brief, Prototypes). Denn wenn man Fehler, Inkonsistenzen oder Auslassungen früh entdeckt, spart das viel Geld und Ärger.

# Projektautomatation

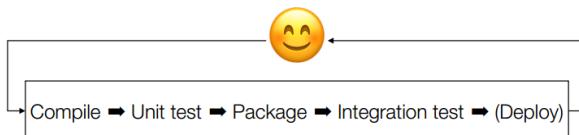
## Wie entwickeln wir Software?

Ein Entwickler führt immer wieder folgende Aktivitäten aus: Kompilieren, Unit Testing, Paketieren, Integrationstests und «Deployen/Veröffentlichen». Die Aktivitäten werden pro meist in sehr kleinen Abständen wiederholend ausgeführt. Dabei drehen wir als Entwickler fast durch, da wir so vieles machen müssen. Ein schlauer Mensch hat einmal gesagt: «Automatisiere alles, was du mehr als einmal brauchst».

### Idee 1 – Build Skript

Die erste Idee. Ein einfaches Skript, welches diese Schritte ausführt. Von den Kompilierung über das Testing bis hin zur Paketierung.

### Workflow



### Vorteile

Die ganze Sache ist nun automatisiert in einem nicht interaktiven Prozess. Das Skript kann mehrere Male ausgeführt werden, es ist also repetierbar. Zudem ist es unabhängig von der IDE. Nicht zuletzt können zeitintensive Tasks terminiert werden.

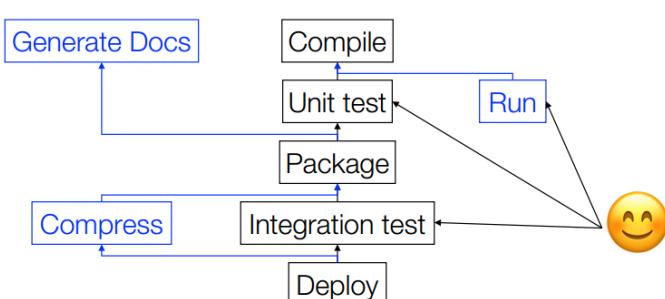
### Nachteile

Der Prozess ist zwar automatisiert, aber eben nicht interaktiv. Die Wartung und Erweiterung ist aufwändig und zudem sind die Skripts platformabhängig (Powershell, Bat, SH).

### Was wir möchten (Wunschliste)?

- Single Command Build (CRISP)
  - o Complete – Jeden Build von neu aufbauen
  - o Repeatable – Immer wieder anstossen, auch älteren Code wieder auschecken und bauen können
  - o Informative – Testresultate von Unit- und Integrationstests
  - o Schedulable – Zeitlich terminierbar
  - o Portable – An verschiedenen Orten ausführen können
- Flexibel
- Leistung – der Build sollte nicht allzu lange dauern
- Erweiterbarkeit

Die Lösung für die Wunschliste ist ein Build Tool, ein spezialisiertes System welches den ganzen Build Prozess verwaltet. Das Core Konzept schaut wie folgt aus.



Zu den weiteren Features gehört z.B. das Dependency Management oder die Optimierung des Build Prozesses (Parallel) sowie Anpassungen beim Testing oder beim Processing.

# The Beginning

Der Beginn der Build Automatisierung - GNU Make (Imperativ)

Der Beginn war Make für die Sprache C auf UNIX. Die erste Version wurde in 1976 von den Bell Labs entwickelt. Es lässt sich wie ein «flexibles» Build Skript vorstellen. Make hat das DAG Konzept mit Targets und Abhängigkeiten eingeführt.

Der Stil ist imperativ (Shell-Scripts). Die ganze Sache ist leider Plattform abhangig und hat kein automatischen Dependency Management. Gearbeitet wird mit Targets / Dependencies und Variables. Die Auflistung muss in der richtigen Reihenfolge erfolgen.

Der Build Author definiert explizit das DAG. Die Targets sind mit einer Skriptsprache implementiert und meistens basiert es auf einem externen Dependency Manager.

## Beispiel

Neben Make für C gibt es auch Jake (Javascript), nmake(.NET) oder Psake für Powershell. Um 2000 wurde dann Ant eingeführt, welches auf XML basiert. 2003 entstand Rake, in welchem die Targets mit Ruby definiert werden.

## Apache Ant (Imperativ)

```
<project name="Hello" default="compile">

    <target name="clean" description="remove compiled files">
        <delete dir="classes"/>
        <delete file="hello.jar"/>
    </target>

    <target name="compile" description="compile class files">
        <mkdir dir="classes"/>
        <javac srcdir=". " destdir="classes"/>
    </target>

    <target name="jar" depends="compile" description="...">
        <jar destfile="hello.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="HelloProgram"/>
            </manifest>
        </jar>
    </target>

</project>
```

Es ist ein XML-basiertes Skripting mit bereits integrierten Tasks wie mkdir oder jar oder condition. Der Fokus liegt auf der Portability. Eigene Tasks können in Java geschrieben werden.

## Vorteile

Es umfangreich und flexibel.

### **Nachteile**

Die Build Definitionen tendieren dazu sehr komplex zu werden. Zudem ist es schwierig die Build Logik wieder zu benutzen. Meist wird Copy Paste

angewendet.

# Deklarative Builds mit Apache Maven

Der erste Prototyp erschien im 2001, 2004 wurde Maven 1.0 lanciert. Damit soll aufgehört werden das Rad neu zu erfinden. Es ist entworfen für Konsistenz über mehrere Projekte. Zudem beinhaltet es ein automatisches Dependency Management.

## Beispiel

```
<project xmlns="..." xsi:schemaLocation="...">
  <modelVersion>4.0.0</modelVersion>

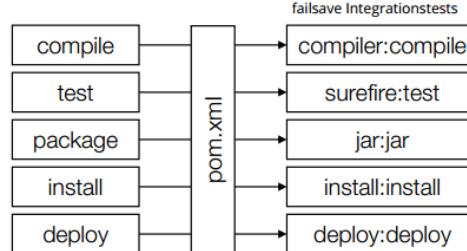
  <groupId>ch.hsr.app</groupId> Domain ID
  <artifactId>app</artifactId>
  <version>1.0-SNAPSHOT</version> Version-Nr.
  <packaging>jar</packaging> Ende
  <name>Example App</name> Name

  <dependencies> Abhängigkeiten
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

Es ist deklarativ und im XML gehalten. Die Konvention steht über der Konfiguration (Default Build, nur Abweichungen angeben). Die DAG's sind die vordefinierten Lifecycles. Zudem setzt Maven eine gewisse Projektstruktur voraus.

```
app
|- src/main/java      //production code
|- src/main/resources
|- src/test/java      //tests
|- src/test/resources
|- pom.xml
```



Im Gegensatz zu anderen Tools gibt der Build Author an, wie das Build Resultat sein sollte. Erweiterungen und Anpassungen findet über Plugins statt.

## Vorteile

Kleinere Build-Dateien, Wiederverwendbare Build Logik (Plugins), Automatisches Dependency Management

## Nachteile

Weniger generell und flexibel als «Imperative» Tools, macht mir Vorschriften wie meine Projektstruktur sein sollte.

## Post-Maven Tools

Deklarative Tools sind ein guter Ansatz, aber sind teilweise zur restriktiv. Die aktuellsten Tools probieren das Beste aus den beiden Welten zu verwenden. Zudem haben jene auch Fortschritte in Performance und User Experience gemacht.

2008 Apache Buildr

2008 SBT

2009 Gradle

All diese Tools sind mit dem Dependency Management von Maven kompatibel.

## The Future

In der Zukunft wird immer mehr Automatisierung gebraucht. Es lohnt sich also die Zeit zu investieren, ein Build Tool zu erlernen. Beginner sollten sich am besten mit Maven befassen. Die automatisierten Build sollten ab Tag 1 eingesetzt werden.

## Zusammenfassung der Vorteile

- Reduktion der repetitiven Tasks
- Unabhängigkeit von der DIE
- Reproduzierbare Resultate
- Zeit sparen
- Basis für Continuous Integration (nächstes Kapitel)

«Team-members integrate their work frequently. Usually, each person integrates at least daily, leading to multiple integrations per day.”

### Ziele

Wir möchten zur jeder Zeit ein lauffähiges Produkt haben. Feedback möchten wir sicher im Falle von Fehlern (Automatisierte Tests, Analysis Tools).

### 10 CI Praktiken

#### 1. Maintain a single source repository

Nutzen Sie ein Source Code Management System, so Weiss jeder wo der Code abgelegt wird. Gearbeitet werden soll nicht auf dem Master-Branch.

#### 2. Automatisierte Build

#### 3. Machen Sie Build selbst-testbar

Erstellen und pflegen Sie eine automatisierte Test Suite.

#### 4. Jedermann commitet täglich auf die Mainline

Sonst kann ich keine tägliche Builds machen. Zudem reduziert es den Merging Aufwand und neue Bugs können schnell gefunden werden.

#### 5. Jeder Commit auf die Mainline soll gebautet werden

Jeder Änderung auf der Mainline sollte als kompletter Build auf dem CI Server ausführbar sein.

#### 6. Der Build sollte schnell gehalten werden

Um schnelles Feedback zu bekommen. Meist geschieht dies in mehreren Schritten (1. Build, 2. Integrations Tests und 3. Performance Tests).

#### 7. Testen in einem Klon der Produktionsumgebung

Test und Produktion sollten so ähnlich wie möglich sein, damit das Test-Feedback so genau wie möglich ist. Hier könnte zum Beispiel Docker verwendet werden.

#### 8. Machen Sie es einfach den die letzten Produkte zu erhalten

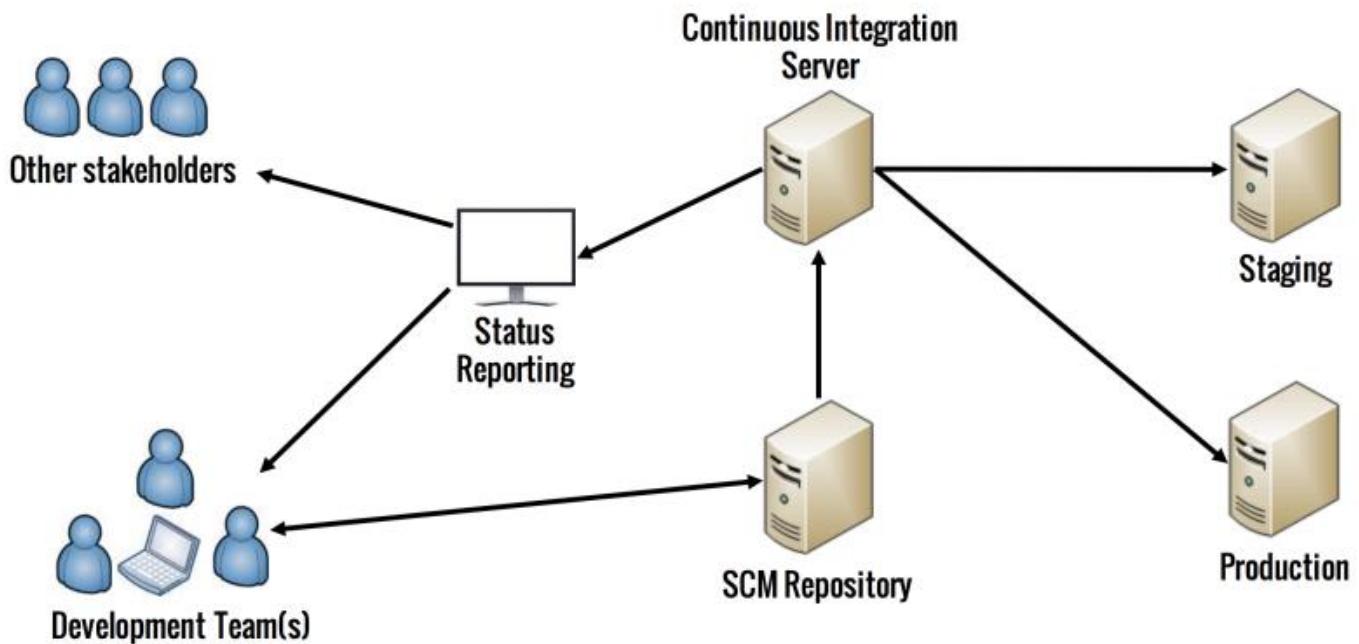
Jeder welcher in der Entwicklung des Projekt involviert ist sollte auf die aktuellste Version des Produktes Zugriff haben.

#### 9. Jedermann kann sehen was passiert

Status des Builds mit Rot und Grün.

#### 10. Automatisches Deployment

Automatisches Deployment vom Produkt nach den Tests auf eine Testumgebung.



### Beste Entwicklerpraktiken

- Der Code sollte häufig commited werden
  - o Nur kleine Änderungen machen
  - o Commiten nach jedem Task
- Commiten Sie keinen fehlerhaften Code
- Kaputte Builds sollten sofort repariert werden
- Schreiben Sie automatisierte Tests
- Lassen Sie Builds lokal laufen (bevor der Code commited wird)
- Setzen Sie ein CI ab Tag 0 auf

### CI Server

Ein CI Server lässt automatisierte Builds und Tests laufen. Über ein Web Interface oder ein Chat System publiziert dieser Resultate. Ein CI Server ist event-gesteuert. Entweder für Intervalle, manuell oder über Änderungen am Code (Commits). Es gibt noch weitere Features wie Quality Analysis oder IDE Integration. Dort sind keine Grenzen gesetzt.

### Beispiele

Open Source (Jenkins, Go, BuildBot, Strider), Commercial (Bamboo, TeamCity) und Cloud Based (Travis CI, Drone.io und GitLab CI)

# Software Engineering Practices

Sie sollen eine Link zwischen der Theorie und der Praxis herstellen. Die Erfahrung aus der Praxis für die Theorie und die Empfehlungen aus der Theorie für die Praxis. Das Ziel ist Good Engineering. Es ist aber kein absoluter Anspruch. Kritisches Denken und Abwägen bleibt daher essential.

## Requirements Practices

### 1. Dig for Requirements

Sie sollten in Zusammenarbeit mit dem Benutzer entstehen und ein Denken aus Benutzersicht sein. In der Erarbeitung sollte kritisches Hinterfragen und Nacharbeiten stattfinden. Die echten Anforderungen von Ad-Hoc Wünschen treffen und immer nach dem Grund fragen. Die Requirements sollten generell und abstrakt definiert werden. Details können schneller ändern und Details können somit konfigurierbar gehalten werden. Immer sollte der Ursprung mit Name und Grund verfolgt werden.

#### Beispiel

«Only an employee's supervisors and the personell department may view that employee's records». Es ist wahrscheinlich zu detailliert. Die Rechtegruppe kann eventuell schnell ändern, was aber schwierig ist wenn der Programmierer die Code-Privilegien hardcodet hat. Es sollte mit einer «group of people» genereller gehalten werden.

### Ermittlungstechniken

Dokumentenstudium, Befragung (Interview, Fragebogen, Workshop), Beobachtung (Fernbeobachtung, Apprenticing) oder Brainstorming. Jede dieser Techniken hat ihre Vor- und Nachteile. Wir möchten damit das Bewusste, das unterbewusste und das unbewusste Wissen herausfinden.

### 2. Make quality a requirement

Die Qualität sollten als NF-Anforderungen aufgenommen werden. Darunter zählt Performance, Scalability, Security und Robustness. Die Qualitätsanforderungen sollten möglichst testbar sein. Zum Beispiel max. Antwortzeiten unter definierten Umständen oder min. unterstützte Datenmengen. Die Anforderungen sollten auf echten Anforderungen basieren. Wie zum Beispiel konkrete Benutzer-Erwartungen oder externe Limiten. NF-Anforderungen sind schwer zu ermitteln, da es oft unbewusste Wünsche sind. Zu beachten ist aber, dass davon Architekturentscheide abhängen, welche später nicht geflickt werden können.

#### Beispiel

«Stock orders shall be placed instantaneously» und «The image processing should not exceed 2 seconds». Beispiele sind sehr ungenau formuliert. Folgende Formulierung ist besser.

Stock orders should be placed within 100 ms after their arrival at the frontend web-service. The processing of an image should not exceed 2 seconds for an image size up to 100MB and 30 seconds for up to 1GB.

### 3. Deal with changes

Dass die Requirements stabil sind ist ein Mythos. Ca. 2 Prozent ändern sich pro Monat.

#### Vorgehen mit Änderungen

##### Requirement-Änderungen anzipieren

Gründe und Details nachfragen, Genügend abstrakt definieren

##### Design for Change

Flexibleres Design, wo Änderungen passieren. Reversibilität vorstehen (z.B. andere DB).

##### Kurze Iterationen

User-Feedback mit funktionierendem Code, Unklare Bereiche früh adressieren

##### Change Assessment

Qualität der Requirements nach der Iteration prüfen. Falls ungenügend, Anforderungen überarbeiten.

#### Design Practices

##### 4. Don't repeat yourself

Repetition von Informationen sollte immer vermieden werden. Folgende Arten sind möglich:

- Code Duplikationen: Logiken, Daten, Bolier-Plate Code, etc.
- Dokumentation im Code: Redundante Beschreibungen
- Dokumentation separat zum Code: Wiederholung
- Wiederholungen wegen der Programmiersprache

DRY führt immer zu einer Gefahr der Inkonsistenzen bei Änderungen.

#### DRY Techniken

- Benannte Konstanten statt literale Konstantenwerte
- Gemeinsam genutzte Prozeduren/Funktionen statt Copy-Paste von Code-Snippets
- Code-Kommentare geben relevante Zusatzinformationen
- Externe Konfigurationsdaten
- Code-Generierung

##### 5. Achieve orthogonality

«Eliminate effects between unrelated things».

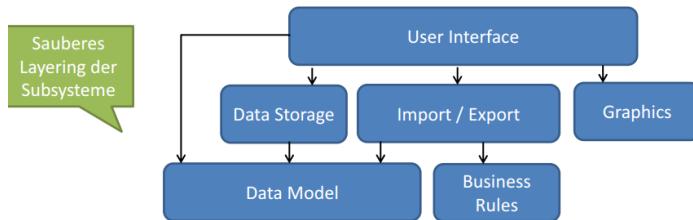
Keine Koppelung zwischen konzeptionell unabhängigen Aspekten. Nicht mehrere unabhängige Aufgabe als eine Routine. Nicht mehrere unabhängige Abstraktionen als ein Objekt. Ziel ist es eine hohe Kohäsion zu erreichen, also die Reduktion auf eine zusammengehörige Aufgabe/Abstraktion pro Komponente.

#### Vorteile der Orthogonalität

- Selektive Wiederverwendung eines Aspektes (Genereller als fixe Kombinationen)
- Ein Aspekt ist isoliert änderbar (Ohne Einfluss auf andere)
- Einfacher zu verstehen (weniger Abhängigkeiten, klare Funktion)
- Einfacher zu testen (weniger Fälle pro Komponente)

#### Koppelung in der Architektur

Eine hierarchische Zerlegung in Komponenten. Aussen relevante Eigenschaften in Schittstelle und die Details mittels Information Hiding verbergen. Zudem eine Reduzierung der Abhängigkeiten zwischen den Komponenten.



Mit möglichst wenig Abhängigkeiten (Statisch (Code Import) sowie dynamisch (Calls). Zudem das Acyclic Dependency Prinzip beachten. Daher direkte und indirekte Zyklen meiden (transitive Kopplung)).

### Das Gegenteil

In diesem Fall hängt alles von allem ab. «Spaghetti Code» bzw. «Big Ball of Mud». Eine Aufteilung ist zwecklos. Damit ist auch das Acycling Dependency Prinzip verletzt. Die Komponenten sind alle nicht einzeln wiederverwendbar. Die Komponenten sind nicht isoliert testbar und schwierig einzeln austauschbar.

## 6. Design to test

Die Testbarkeit sollte vor der Entwicklungszeit betrachtet werden. Sie hat einen Einfluss auf die Architektur. Folgende Betrachtungen der Testbarkeit können gemacht werden:

- Klare Interfaces und Kontrakte
- Überlegung der relevanten Testfälle
- Grad der Unit-Testbarkeit und Integration-Testbarkeit
- Evtl. Freiheitsgrade für «Dependency Injection»
  - Erzeugung und Initialisierung von benötigten Objekten ausserhalb des Benutzer Objekts
  - Isoliertes Testen mit anderen Komponenten («Fakes»)
  - Kann Design komplexer als nötig machen

### Test-Vokabular

**Fake** Vereinfachte Schnellere Implementierung (z.B. In-Memory DB)

**Mock** Auf Testfall zugeschnitten, prüft Reihenfolge und Inhalt der erwarteten Aufrufe

**Stub** Auf Testfall zugeschnittene Antworten

**Dummy** Objekte, die nur herumgereicht, aber nie inspiert werden

### Beispiel – Dependency Injection

<p>▪ Ausgangslage</p> <pre> public class TradingSystem {     private StockQuoteSystem stockInfo = new StockQuoteSystem();      public TradeResult Buy(string quote, int amount) {         Order o = stockInfo.placeOrder(STOCK_BUY, quote, amount);         ...     } } </pre> <p>Schwierig automatisch zu testen:  - Allen Fällen von StockQuoteSystem nicht steuerbar  - StockQuoteSystem ist eventuell langsam  - Läuft eventuell nur mit echtem Backend-Server</p>	<p>▪ Lösung</p> <p>Interface einführen</p> <pre> public class TradingSystem {     private StockInformation stockInfo;      public TradingSystem(StockInformation stockInfo) {         this.stockInfo = stockInfo;     }      public TradeResult Buy(string quote, int amount) {         Order o = stockInfo.placeOrder(STOCK_BUY, quote, amount);         ...     } } </pre> <p>Objekt ausserhalb erzeugen</p> <p>Helper für Testzwecke</p> <p>StockInformation</p> <p>StockQuoteSystem</p> <p>FakedStockInfo</p>
--	---

**7. Fix broken windows**

Die Probleme sollten behoben werden, wenn Sie entstehen. Kleine Probleme sofort beheben und grössere Probleme markieren, dass man sich darum kümmern wird. Dies gilt auch für Code- und Entwicklungsprozesse. Zu Grund liegt die Tatsache, dass man eine saubere Toilette auch eher sauber hinterlässt, als eine dreckige Toilette.

**8. Refactor early and often**

Refactoring ist der «Heilungsprozess». Ein konstanter Verbesserungsprozess während wachsendem Software-Projekt.

Dazu soll eine Liste von zu verbessernden Bereichen geführt werden und der betroffene Programmierer/Benutzer sollte informiert werden. Das Refactoring soll keine neuen Funktionen bieten. Die guten Tests sollte am bereits vor Beginn der Refactorings haben. Lieber mehrere kleinere Schritte statt einer Riesenänderung.

**9. Program deliberately**

Vermeide «Programming by Coincidence (or Luck)»

- Klares Ziel sehen und Design verfolgen
- Logisch rigoros analysieren, entwickeln und testen
- Nur auf spezifizierte Features von Libraries verlassen
- Eingesetzte Technologie beherrschen
- Annahmen dokumentieren und mit Assert & Tests prüfen
- Crash early: Alle ungültigen Zustände sollen Fehler erzeugen
- Exceptions richtig behandeln, nicht blind unterdrücken
- Debugging: Auf den Grund gehen, Fehler verstehen

**Verification Practices****10. Test rigorously**

Früh, häufig und automatisch testen

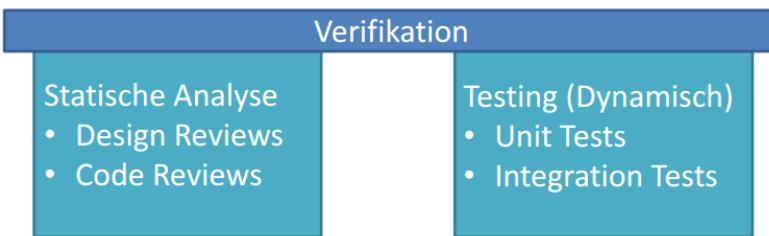
- Design to Test
- Test-Driven Development
- Hohe Code Coverage erzielen
- Automatische Unit und Integrationstests
- Änderungen testen
- Möglichst viele und realistische Testdaten

«Find Issues Once»

- Gefundene Fehler verstehen und einen Test dafür schreiben
- Gleicher Fehler in Zukunft automatisch finden

**Achtung beim Testen**

Bewusstes rigoroses Überlegen beim Programmieren nicht wegen den Tests vernachlässigen. Integrations Tests sind mindestens so wichtig wie Unit Tests. Auch wenn alle Teile eines Autos einzeln getestet sind, muss das Auto deswegen noch nicht richtig fahren. Gewisse Fehler sind schwierig mittels Tests zu finden. Auch eine 100 % Test Coverage bedeutet noch keine Korrektheit.



## 11. Perform Reviews

- Formal Inspections (Reviews)
  - Einzelne Reviews durch Experten
    - Ca. 60 Prozent von Defekten werden gefunden
  - Kombination Design & Code Reviews
    - Ca. 70 bis 85 Prozent weniger Fehler
- **Vorgehen**
  - In Sitzungen (mit Moderator und Vorbereitung)
  - Oder selbstständig (Zusammentragen der Findings)
  - Die Findings sollten unbedingt festgehalten werden (Severity, Action, Verantwortlicher)
- Das Ziel dabei ist, Defekte zu finden. Keine Diskussion der Architektur, alternativer Lösungen oder Leistungsbewertung des Autors.

## Error Handling Design

### Software Disasters

Es gab in der Geschichte einige schlimme Softwarepannen, welche auf Fehler in der Software zurückzuführen sind. Die Ariane 5 Rakete, welche aufgrund eines Type Cast Errors (Overflow) explodiert ist. Ein anderer Fall ist ein Therac, welches den Patienten verstrahlt. Ursache war ein Concurrency Fehler (Race Condition). Bei Toyota kam es zu verzögertem Bremsen. Die Ursache ist unbekannt, es war aber sicher ein Softwarefehler.

### Fehler-Prävention

- Design & Code Reviews
- Unit, Integration & System Testing
- Static Analysis
- Error Handling Design (Fokus in diesem Kapitel)
- Concurrency Design und Testing

### Überblick

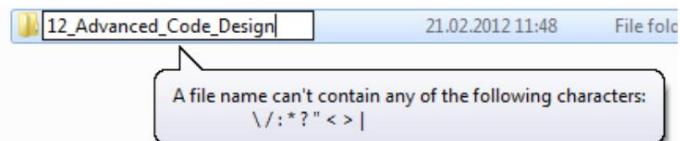
Ist ein wesentlicher Bestandteil der Software Architektur. Dazu gehört die Wahl der Error Handling Strategie sowie das Befolgen der Konsistenz. Zu den relevanten Fragen gehören: Welche Fehler müssen erkannt werden? Was für eine Systematik wird dabei eingesetzt? Wie werden Fehler behandelt? Wie unterscheidet sich Engineering und Produktion?

### Defensive Programmierung

Systematische Fehlerprüfung (Alle Werten von allen externen Quellen, alle Werten von Input-Parametern in Routine und nicht unterstützte Zustände in der Fallbehandlung. Dazu zählt auch eine systematische Fehlerbehandlung unter Abhängigkeit des kritischen Grades des Systems.

## Schutz vor ungültigem Input

Kein «Garbage in, garbage out». Das heisst ungültige Benutzer-Eingaben verhindern, Fehlererkennung und Meldung und keine Resultate bei ungültigem Input. Das heisst alle Werte von externen Quellen prüfen.



## Prüfung von Routinen-Input

Fehler von internen oder externen Quellen erkennen und dabei die Preconditions prüfen (Parameterwerte oder Zustände der Objekte).

```
void withdraw(int amount) {
    if (balance + MaxCredit < amount) {
        throw new BankException("Insufficient credit");
    }
    // ... perform actions
}
double root(double value, int degree) {
    if (degree <= 0) {
        throw new IllegalArgumentException("Negative degree");
    }
    // perform calculation
    assert result == power(degree, value);
}
```

SE2 - Error Handling

## Abfangen ungültiger Fälle

Ungültige Zustände systematisch abfangen (Bedingung für else, Default Case bei Switch) und damit durch spätere Erweiterungen erkennen.

```
if (l == TrafficLight.RED) { ...
} else {
    assert l == TrafficLight.YELLOW
    || l == TrafficLight.GREEN;
...
}
```

```
if (a && b) { ...
} else if (!a) { ...
} else {
    assert !b;
    ...
}
```

## Beispiel

### Ungültige Fälle

In diesem Beispiel gibt es keinen Default Case.

```
switch (motor.getCommand()) {
    case Command_Accelerate:
        accelerate();
        break;
    case Command_Turn:
        turn();
        break;
    // there are no other commands
}
```

```
switch (motor.getCommand()) {
    case Command_Accelerate:
        accelerate();
        break;
    case Command_Turn:
        turn();
        break;
    // there are no other commands
}
```

default:  
throw new AssertionError("Unsupported command");

Später wird dann doch noch  
Command\_Stop eingeführt

**Barrikaden im Programm gegen Fehler definieren**

Hinter den Barrikaden sind Daten gültig. Die Daten werden bei Grenzübertritt überprüft.

**Barrikade auf Klassenniveau**

Public Methoden überprüfen Daten (per Exception, externen Input). Die privaten Methoden gehen von gültigen Daten aus (Per Asserts, interner Input).

**Fehlerbehandlungs-Techiken****Konservative Behandlung**

- Error Handling Prozedur aufrufen
- Fehlermeldung anzeigen
- Shutdown

**Optimistische Behandlung**

- Neutrales Resultat
- Nächstmögliches plausibles Resultat
- Warnung in Stream loggen

**Korrektheit versus Robustheit****Korrektheit**

Niemals ungenaues Resultat liefern

**Robustheit**

Versuche Software am Laufen zu halten

Für was man sich entscheidet ist von System zu System verschieden. Bei sicherheitskritischen System setzt man oft auf Korrektheit, bei unkritischen Systemen auf Robustheit. Es ist aber je nach Funktion genauer zu betrachten.

**Lokale vs. Globale Behandlung****Lokal behandeln**

Nur für erwarteter Fall, der nicht höher relevant ist. Also nur wenn der Fall lokal abschliessend entscheidbar ist.

**An Aufrufer delegieren (Global)**

Wenn nicht lokal behandelbar oder entscheidbar. Zudem wenn der Fehler auf höherer Systemebene relevant ist.

Grundsätzlich sollte man keine ungültigen Zwischenzustände hinterlassen. Also finally-Block (auch ohne catch= und dort Locks (Ressourcen) freigeben und temporäre Daten löschen).

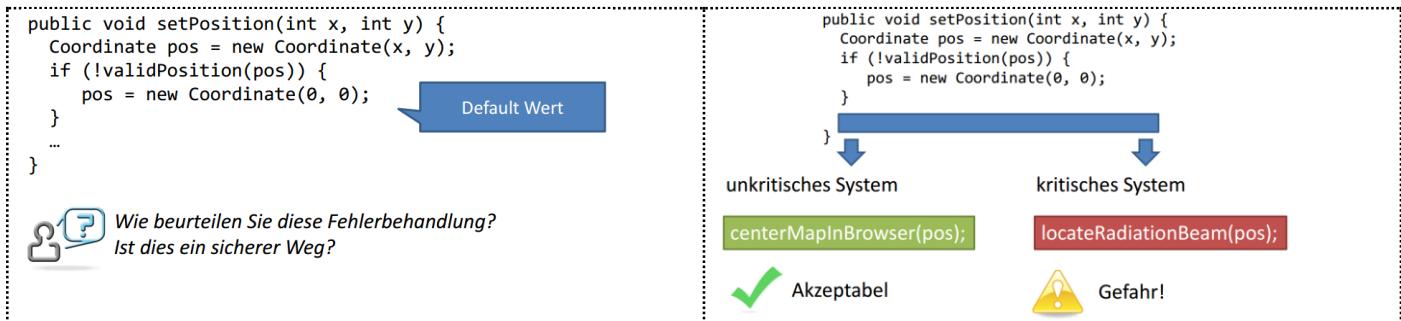
**Global Exception Handler**

Top-Level Routine zur Fehlerbehandlung

- Alle unbehandelten Fehler an den GEH delegieren
  - o Explizit mit try-catch für alle top-Most Thread-Routinen
  - o Pauschal mit Thread.setUncaughtExceptionHandler() (Pro Thread-Instantz)
  - o Static Thread.setDefaultUncaughtExceptionHandler(), Fallback für alle Threads
- Protokollierung und Benutzer-Meldung

- Eventuell Programm in einen konsistenten Zustand setzen
- Sonst Programm kontrolliert terminieren

## Beispiel Fehlerbehandlung



## Risiko der Fehlerbehandlung

Es kann auch in der Fehlerbehandlung Fehler geben. Daher auch testen, einfach halten und die Sicherheitsmassnahmen zuerst einleiten (vor dem Logging, Export oder der Fehlermeldung).

### Kritischer Grad der Fehler

#### Kritische Systeme

Globale Behandlung der Fehler, System in einen sicheren Zustand bringen (Shutdown von Maschinen, Starten von Sicherheitsmassnahmen, Korrupte Daten abschriften) und den Fehler danach protokollieren.

#### Unkritische Operationen

Lokale Behandlung der Fehler, Optimistische Behandlung und dabei eine Warnung mitteilen und protokollieren.

## Exceptions vs. Assertions

### Exceptions

- Für mögliche produktiv Fälle (externe Quelle)
- Für sicherheitsrelevante Fehler
- In Java: `Throwable` Error für nicht zu behandelnde Fehler

### Assertions

- Können eingeschaltet und abgeschaltet werden
- Für «Debug Mode» einschalten
- Für Programmierfehler, die nie auftreten sollen
- Für Postconditions
- Für Preconditions interner Quelle
- In den Assertions darf kein ausführbarer Code stehen

## Assert mit Seiteneffekt

```
assert stopMachine();
```

Ausführbarer Code in Assert

Development (enable asserts)

Production (disable asserts)

Maschine stoppt (getestet)

Maschine stoppt nicht mehr



### Assertions- Ein oder ausschalten?

In der Produktion für Programmierfehler, die nie auftreten sollen. Diese evtl. als formale Kommentare verstehen. In der Regel sind Sie aber in der Produktion abgeschaltet.

### Bei Tests

Auch dort. Der Test muss Release Code identisch übernehmen. Wenn man Sie also in der Produktion ausschaltet, dann gilt das gleiche auf für die Tests.

### Primär

- Als Stütze während der Entwicklung
- Formale Kommentare

### Sekundär

- Bei Verdacht auf grobe logische Fehler
- Bei lokaler Fehlersuche

### Logging

Nur reinen diagnostischen Zwecken. Um Ursachen von Fehler von Fehler zu identifizieren oder System-Irregularitäten zu erkennen.

### Log Levels

Severe/Error, Warning, Info, Fine/Trace, ...

Dynamisch filterbar, formattierbar, behandelbar

### Frameworks

java.util.logging, Log4J

Verteilter Log-Server

- Beispiel: Log4J SocketAppender/SocketServer

.NET System.Diagnostics.Trace, Log4Net

### Anwendung

#### Wichtigkeit bei jedem Projekt

Alle Projekte haben unterschiedliche Anforderungen. Daher kann man die Policies nicht einfach eins zu eins übernehmen. Wichtig ist, dass man alle vier Policies klar definiert.

#### Error Handling Policy

Folgendes gehört in diese Policy:

- Welche Eingaben und Interaktionen sind erlaubt?
- Wie muss sich das System bei unerlaubten Eingaben oder Interaktionen verhalten?
  - o Rückmeldung an den Benutzer?
  - o Abbruch?
  - o Logging?

## Software Engineering 2

### Exceptions Policy

Folgendes gehört in diese Policy:

- Werden Exceptions benutzt?
- Wie?
  - o Für Behandlung von welchen Arten von Fehlern
  - o Local Handling /Global Handling?
  - o Checked /Unchecked?

### Assertions Policy

Folgendes gehört in diese Policy:

- Werden Assertions benutzt?
- Für was?
- Wann werden Sie eingeschaltet?

### Logging Policy

Folgendes gehört in diese Policy:

- Wird ein Log generiert?
- Was für Informationen müssen dort geschrieben werden?
- Wie detailliert?
- Welche Levels werden benutzt?

## Design by Contract (DbC)

### Einführung

Ein Vertrag (Contract) legt Rechte und Pflichten zweiter Parteien dar. Meistens zwischen Kunde (Client) und Lieferant (Supplier).

### Verträge auf Software angewandt (DbC)

Ein Software System ist eine Menge von Komponenten. Eine Komponente ist ein System oder Subsystem oder Klasse. Es hat damit eine Kunden/Lieferanten-Beziehung zwischen den Komponenten, welche durch «Verträge» geregelt wird.

### Contracts für Systemoperationen

Es ist Teil der Domainanalyse. Die betrachtete Komponente ist das System. Bei einer Precondition gibt man in diesem Fall an, was vor der Ausführung der Systemoperation gilt. Die Postconditions geben an, was nach der Ausführung der Systemoperation gilt.

### Contract für eine Klasse

Für jede Methode gibt man Pre- und Postconditions an:

#### *Preconditions*

Bedingungen, die vor dem Aufruf erfüllt sein müssen. Verantwortlich ist der Aufrufer.

#### *Postconditions*

Bedingungen, die nach dem Aufruf gültig sind. Verantwortlich ist die Implementation der Methode.

Postconditions, die für alle Methoden gelten lassen sich als Klasseninvariante formulieren. In diesem Fall ist die Implementation der Klasse verantwortlich.

<b>Pflichten</b>		<b>Rechte</b>	Beispiel in Eiffel für put
<i>put</i>	OBLIGATIONS	BENEFITS	put (x: ELEMENT; key: STRING) is -- Insert x so that it will be retrievable through key.
<i>Client</i>	(Satisfy precondition:) Only call <i>put</i> (x) on a non-full stack.	(From postcondition:) Get stack updated: not empty, x on top ( <i>item</i> yields x), <i>count</i> increased by 1).	<b>require</b> count < capacity not key.empty
<i>Supplier</i>	(Satisfy postcondition:) Update stack representation to have x on top ( <i>item</i> yields x), <i>count</i> increased by 1, not empty.	(From precondition:) Simpler processing thanks to the assumption that stack is not full.	<b>do</b> ... Some insertion algorithm ...

aus [Meyer97]

Beispiel in Eiffel für Klasseninvariante Stack	
class STACK	<b>Class invariant</b>
...	
<b>invariant</b>	
count_non_negative: 0 <= count	
count_bounded: count <= capacity	
consistent_with_array_size:	
capacity = representation lcapacity	
empty_if_no_elements: empty = (count = 0)	
item_at_top: (count > 0) implies	
(representationl item (count) = item)	

aus [Meyer97]

Beispiel Stack im icontract	
Preconditions, Postconditions und Klasseninvariante spezifizieren Stack. Die Spezifizierung aber auf das Interface und nicht die Implementation spezifizieren.	
/** * @inv !isEmpty() implies top() != null */	
public interface Stack {	
/**	
* @pre o != null	
* @post !isEmpty()	
* @post top() == o	
*/	
void push(Object o);	
/**	
* @pre !isEmpty()	
* @post @return == top()@pre	
*/	
Object pop();	
/**	
* @pre !isEmpty()	
*/	
Object top();	
boolean isEmpty();	

Achtung! Gilt nicht allgemein, ist Annahme hier!

## Das Interface ist nicht die ganze Geschichte

Auch die Reihenfolge beziehungsweise der Zustand ist entscheidend.

Method	Preconditions	Postconditions
open(filename, flags) signals UnableToOpen	None	If (for writing) if user has permission File is opened for writing  If (for reading) if file exists and user has permission File is opened for reading
read(buffer, count) signals EndOfFile, UnableToRead	File opened for reading	If not at end of file If count < bytes left in file Set file position to bytes after current else Set file position to end of file
write(buffer, count) signals UnableToWrite	File opened for writing	File position incremented by count
close()	File is open	File closed

## interface File

**open**(filename, flags) signals

UnableToOpen

**read**(buffer, count) signals

EndOfFile, UnableToRead

**write**(buffer, count) signals

UnableToWrite

**close**()

## Ziele

Es stellt eine grundlegende Entwurfsmethodik dar. Somit gehört es zum Grundwissen eines Informatik-Ingenieurs. Es ist nützlich, wenn es pragmatisch angewandt wird. Das heisst Fokus auf Spezifikation statt Implementation und ein klares Verständnis bezüglich den Verantwortlichkeiten.

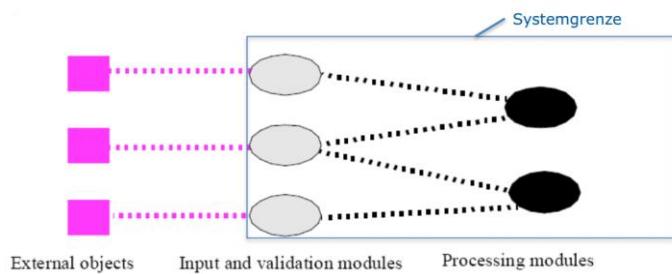
## Illustration

$Y = \text{SQRT}(X)$  → Wer ist jetzt verantwortlich zu prüfen, dass  $x \geq 0$  sein muss?

## Preconditions, Postconditions, Class Invariants

### Preconditions

Bedingungen, die vor dem Aufruf erfüllt sein müssen. Verantwortlich ist der Aufrufer. Die Preconditions in der aufgerufenen Methode nie überprüfen. Validierung von den Eingabeparametern nur wenn es explizit die Aufgabe der Methode ist. Die Validierung hat nichts mit der Überprüfung von Preconditions zu tun. Die Validierung sollte so nahe an den externen Quelle wie möglich durchgeführt werden, sodass der Kern des Systems davon frei gehalten wird.



### Postconditions

Bedingungen, die nach dem Aufruf gültig sind. Verantwortlich ist die Implementation der Methode. Sie wird eine Methode Exceptions, sodass Fehler unterscheidet werden können.

### Postconditions bei Query-Methoden?

Query Methoden sollten den Zustand nicht ändern (keine Nebeneffekte). Daher also auch keine Postconditions, die sich auf den Zustand des Objektes beziehen. Nur Postconditions für Rückgabewerte.

### Invarianten

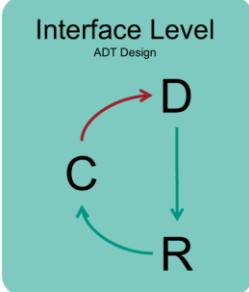
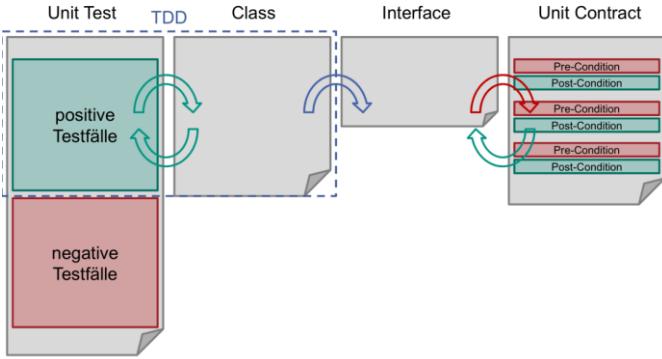
Gelten immer, daher nach Aufruf beliebiger Methode, aber natürlich nicht während Ausführung der Methode. Immer geltende (Post)conditions sollten unbedingt als Invarianten angegeben werden.

### Contracts und Vererbung

Die Contracts als Spezifikationsmittel und gehören daher also zum Interface. Sie gelten für die implementierende Klasse. Nach dem Liskov Substitution Prinzip gilt, Subtyp ist jederzeit an Stelle des Basistyps erlaubt. Der Subtyp muss daher mindestens Vertrag der Basisklasse erfüllen.

Der Subtyp muss mindestens den Vertrag der Basisklasse erfüllen. Daher gilt folgendes:

- Subtyp darf Preconditions lockern, aber nicht verschärfen (ODER Verknüpfung)
- Subtyp darf Postconditions verschärfen, aber nicht lockern (UND Verknüpfung)
- Subtyp darf Invariante verschärfen, aber nicht lockern (UND Verknüpfung)

<p><b>Wie formuliert man einen vollständigen Vertrag?</b> Die 6 Prinzipien der Vertragsentstehung (Clean Contract)</p> <div style="text-align: right;"></div> <ul style="list-style-type: none"> <li><b>Prinzip 6</b> Formuliere Invarianten um die unveränderlichen Eigenschaften des Objekts zu beschreiben. Wähle diejenigen Attribute, die den Anwender unterstützen, das konzeptionelle Modell der Klasse zu verstehen.</li> <li><b>Prinzip 5</b> Prüfe für jede Abfrage und jedes Kommando, ob eine Voraussetzung erforderlich ist.</li> <li><b>Prinzip 4</b> Erstelle für jedes Kommando eine Nachbedingung, die den Wert jeder elementaren Abfrage auflistet. Der vollständig sichtbare Effekt eines jeden Kommandos ist nun bekannt.</li> <li><b>Prinzip 3</b> Erstelle für jede abgeleitete Abfrage eine Nachbedingung, die beschreibt, welches Ergebnis zurückgegeben wird, ausgedrückt durch eine oder mehrere elementare Abfragen.</li> <li><b>Prinzip 2</b> Trenne elementare Abfragen von abgeleiteten Abfragen. Abgeleitete Abfragen können durch elementare Abfragen ausgedrückt werden.</li> <li><b>Prinzip 1</b> Trenne Abfragen durch <code>@Pure</code> von Kommandos. Abfragen geben ein Ergebnis zurück, ändern jedoch nicht die sichtbaren Eigenschaften des Objekts. Kommandos können das Objekt verändern, geben jedoch kein Ergebnis zurück.</li> </ul>	<p><b>Design by Contract</b> DbC-Zyklus: Declare – Refactor – Contract</p> <div style="text-align: right;"></div> <div style="text-align: center;">  <p>D : Declare R : Refactor C : Contract</p> </div> <ul style="list-style-type: none"> <li><b>Declare (Prinzip 1 &amp; Prinzip 2)</b> <ul style="list-style-type: none"> <li>Ergänze das Interface um die Deklaration einer neuen Methode           <ul style="list-style-type: none"> <li>Abfrage (<code>@Pure</code>)               <ul style="list-style-type: none"> <li>elementar</li> <li>abgeleitet</li> </ul> </li> <li>Kommandos</li> </ul> </li> </ul> </li> <li><b>Refactor (Prinzip 4 &amp; Prinzip 6)</b> <ul style="list-style-type: none"> <li>Bei Hinzufügen einer elementaren Abfrage           <ul style="list-style-type: none"> <li>Ergänze alle bestehenden Nachbedingungen um einen Ausdruck basierend auf dieser neuen Abfrage</li> <li>Ergänze ggf. die Klasseninvariante</li> </ul> </li> </ul> </li> <li><b>Assert (Prinzip 3 &amp; Prinzip 4 &amp; Prinzip 5)</b> <ul style="list-style-type: none"> <li>Für die neue Methode:           <ul style="list-style-type: none"> <li>Formuliere die Voraussetzung</li> <li>Formuliere die Nachbedingung</li> </ul> </li> </ul> </li> </ul>
<p><b>TDD with Contracts</b> Beschreibung des Vorgehens</p> <div style="text-align: right;"></div> <p>TDD-Zyklus (Arbeiten auf der Klassen-Ebene)</p> <ol style="list-style-type: none"> <li>1. Erstelle einen positiven Test für die Zielklasse.</li> <li>2. Der Test schlägt fehl.</li> <li>3. Überarbeitet die Zielklasse so, dass der neue Test grün wird.</li> <li>4. Führe ein Refactoring (Clean Code) auf der Zielklasse und der Testklasse aus.</li> <li>5. Wenn es weitere, redundanzfreie positive Tests gibt: Gehe zurück zu Schritt 1.</li> </ol> <p>Vorbereiten des DbC-Zyklus Kann ausgelassen werden</p> <ol style="list-style-type: none"> <li>6. Extrahiere das Interface aus der Zielklasse und passe die Zielklasse so an, dass sie dieses Interface implementiert.</li> <li>7. Erstelle eine Vertragsklasse für das Ziel-Interface.</li> </ol> <p>DbC-Zyklus (Arbeiten auf der Typen-Ebene)</p> <ol style="list-style-type: none"> <li>8. Erstelle einen negativen Test für die Zielklasse. <code>@Test(expected = AssertionError.class)</code></li> <li>9. Der Test schlägt fehl.</li> <li>10. Überarbeitet die Pre-Conditions der Vertragsklasse des Ziel-Interfaces so, dass der neue Test grün wird.</li> <li>11. Führe ein Refactoring (Clean Code) auf der Vertragsklasse und der Testklasse aus.</li> <li>12. Wenn es weitere, redundanzfreie negative Tests gibt: Gehe zurück zu Schritt 8.</li> </ol> <p>Abschließen des DbC-Zyklus</p> <ol style="list-style-type: none"> <li>13. Führe ein abschließendes Refactoring (Clean Contract) auf der Vertragsklasse aus, insbesondere:       <ul style="list-style-type: none"> <li>Clean Contract Prinzip 4: Erstelle als Verallgemeinerung der Aussagen der positiven Tests die Beschreibung der Veränderung des sichtbaren Zustands jeder Methode des Ziel-Interfaces in Form von Post-Conditions, in der über jede elementare Abfrage des Ziel-Interfaces via assert-Statement eine Aussage getroffen wird.</li> <li>Clean Contract Prinzip 6: Vermeide Redundanz durch Klassen-Invarianten.</li> </ul> </li> </ol>	<p><b>TDD with Contracts</b></p> <div style="text-align: right;"></div> 
<p><b>Vorteile von TDD with Contracts</b></p> <div style="text-align: right;"></div> <ul style="list-style-type: none"> <li><b>Design:</b> <ul style="list-style-type: none"> <li>Schnittstellen aus Client-Sicht entwickelt und vollständig spezifiziert</li> <li>Code ist mit einem feinmaschigen Sicherheitsnetz aus Unit Tests und Unit Contracts testbar</li> <li>Modularität, lose Kopplung</li> </ul> </li> <li><b>Effizienz:</b> <ul style="list-style-type: none"> <li>Fokussierung durch Abstraktion und explizite Annahmen, kein unnötiger Code</li> <li>Effizientes Debugging, da Fehler an der Quelle erkannt werden</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li><b>Dokumentation:</b> <ul style="list-style-type: none"> <li>Zur Laufzeit überprüfbare Spezifikation</li> <li>Automatische Dokumentation der Vor- und Nachbedingungen (und Klassen-Invarianten) in der JavaDoc-Dokumentation</li> </ul> </li> </ul> <p>Unit Tests und Unit Contracts als äußerst feinmaschiges Sicherheitsnetz Fehlerhafte Zustände werden bereits beim automatisierten Testen entdeckt</p>

## Unterstützung von DbC in Programmiersprachen

### Eingebaut

Ist es in Eiffel und wenig anderen weiteren Sprachen. Die meisten verbreiteten Programmiersprachen unterstützen Design by Contract nicht direkt.

### Zum Teil

Basiskonstrukt assert, mit welchem DBC-Konstrukte aufgebaut werden können.

### Verbreitet

- Spezielle Kommentare
  - Für Vor- und Nachbedingungen, Klasseninvarianten
  - Präprozessor instrumentiert Code
- Annotationen
- Libraries für DbC
- Kombination von Annotations und Library (Java: J4C)

Die Präprozessoren icontact und icontact2 sind nicht mehr aktuell. 2008 wurde ein Versuch einer Weiterentwicklung gestartet, welcher aber gescheitert ist.

### Contract4J

Verwendet Annotations und ist in AspectJ geschrieben. Letzte Version stammt aus dem 2011 und scheint daher eingeschlafen.

### JML (Java Modelling Language)

Verwendet Java Doclets (daher Kommentare mit @...). Es ist eine allgemeine formale Interface Spezifikationssprache. Verschiedene Tools sind vorhanden. z.B. OpenJML für Java 1.7. Es ist vor allem für akademische Untersuchungen geeignet.

### Oval (Object Validation Framework for Java)

Constraints mittels Annotations (z.B. @NotNull). Es verwendet AspectJ für DbC und hat damit einen breiteren Scope als DbC.

### Java Assert

Ermöglicht unter anderem Design by Contract im «Eigenbau»

Cofaja – Contracts for Java

Ein Open Source Projekt, welches von Google im Jahre 2011 gestartet wurde.

```
interface Time {
    ...
    @Requires({
        "h >= 0",
        "h <= 23"
    })
    @Ensures("result >= 0",
        "result <= 23")
    void setHour(int h);
    ...
    int getHour();
}
```

### In .NET 4

DbC wurde in .NET 4.0 eingeführt unter dem Namen Code Contracts (denn Design by Contract ist eine Hadnelsmarke von Eiffel Software). <http://research.microsoft.com/en-us/projects/contracts/>

### Beispiel

Precondition	Postcondition
<pre>using System.Diagnostics.Contracts;  private static void Main(string[] args) {     DoRequires(null); }  public static void DoRequires(string input) {     Contract.Requires(input != null);     Console.WriteLine(input); }</pre>	<pre>public static int DoEnsures() {     Contract.Ensures(Contract.Result&lt;int&gt;() &gt; 0);     return 0; }</pre>

## Diskussion

### Probleme

Es ist eine fehlende Sprachunterstützung vorhanden, da es sich in der Praxis nicht systematisch durchgesetzt hat. Es gibt aber immer wieder neue Anläufe, welche bis jetzt aber noch nicht erfolgreich waren.

### Concurrency

Die bisherigen Aussagen gelten, wenn entweder die Programme sequentiell ausgeführt werden oder es sich um «fully synchronized objects» handelt. Es existieren aber bereits Ansätze für «Design by Contract» in einer Multithreading-Umgebung.

### DbC und Unit Tests

Db Cist eine Designmethode. Es spezifiziert (dokumentiert) die Schnittstellen und macht klare Verantwortlichkeiten zwischen Caller und Callee. Contracts können automatisch geprüft werden. Während DbC Konstrukte deklarativ sind (Allgemeine Bedingungen werden überprüft), sind Unit Tests imperativ (Testen von spezifischen Werten).

⇒ Automatische Überprüfung von Contracts und Unit Tests ergänzen sich sehr gut.

### Fazit

Es ist ein sauberes methodisches Hilfsmittel Software-Komponenten zu spezifizieren. Zudem dient es als Dokumentationshilfsmittel. Des weiteren lässt es sich als ergänzendes Hilfsmittel für das Testen und das Debugging einsetzen.

## Grosse Arbeiten aufteilen

### Story Splitting – Aufteilung von zu grossen Projekten

#### Zu grosses Projekt – Was tun?

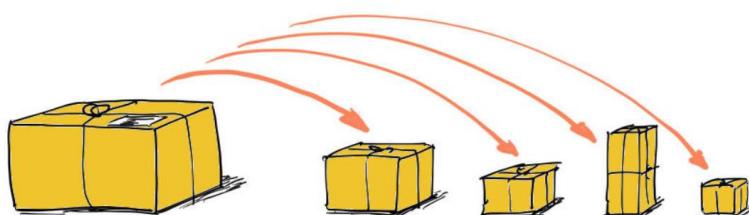
Schon fast alltägliche Schreckens-Geschichten:

“Das Führungsinformationssystem (FIS) Heer, vor zehn Jahren als Prestigeprojekt der Armee lanciert, werde definitiv nie so funktionieren wie vorgesehen. Deshalb [...] schätzungsweise 125 Millionen abgeschrieben werden.” (Tagesanzeiger 11.01.2017)

“Die neue Computerplattform des Bundesamts für Strassen verzögert sich weiter und wird nochmals um 6 Millionen Franken teurer. Entwicklungskosten: 8 Millionen Franken, Einführungsjahr: 2013. Das war der ursprüngliche Rahmen [...] Wie sich nun zeigt, reichen selbst die 32 Millionen nicht aus.” (Tagesanzeiger 17.3.2015)

Zur Erinnerung. Kein Projekt über eine Million und kein Projekt länger als 9 Monate.

#### Was tun, wenn ein Projekt zu gross ist?



Meist geht es darum, inkrementell abzuliefern, d.h. gestaffelt über die Zeit wachsende Anteile des Ganzen zu liefern. Daher Teil-Lieferungen (incremental delivery). Aber wie aufteilen?

## Aufteilung nach Kunden-Domäne

### Beispiel Versicherung

- Leben
- Nicht-Leben:
  - Rechtsschutz
  - Reise
  - Haftpflicht
  - Sachen:
    - o Gebäude
    - o Fahrzeuge
    - o Hausrat
    - o andere Sachen

Fangen Sie mit einem Teilbereich an, nehmen ihn wenn möglich in Betrieb - und packen dann den nächsten Teilbereich an.

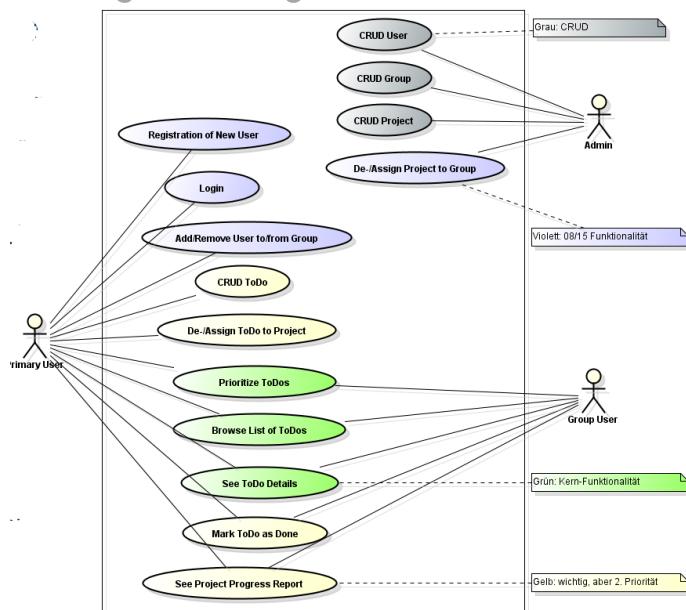
## Aufteilung nach Geschäfts-Prozessen

### Tätigkeiten/Prozesse im Beispiel Gebäudeversicherungen

- Haus schätzen
- Police erstellen
- Schaden durch Experten einschätzen
- Schadensreport ins System aufnehmen
- Kundenadresse ändern
- Etc.

Das ist ungefähr das, was Sie in einem Projekt mit Use Cases definieren. Und die können manchmal gut nacheinander ausgeliefert werden.

### Reihenfolge? Wichtigkeit



### Sortieren nach Wichtigkeit

- CRUD (grau) auf später verschieben
- 08/15 Funktionalität (violett) erst später implementieren
- Wichtige Funktionalität identifizieren «Ohne diese Funktion geht es nicht» (der Rest)
- Kern-Funktionalität identifizieren (grün) und zuerst machen

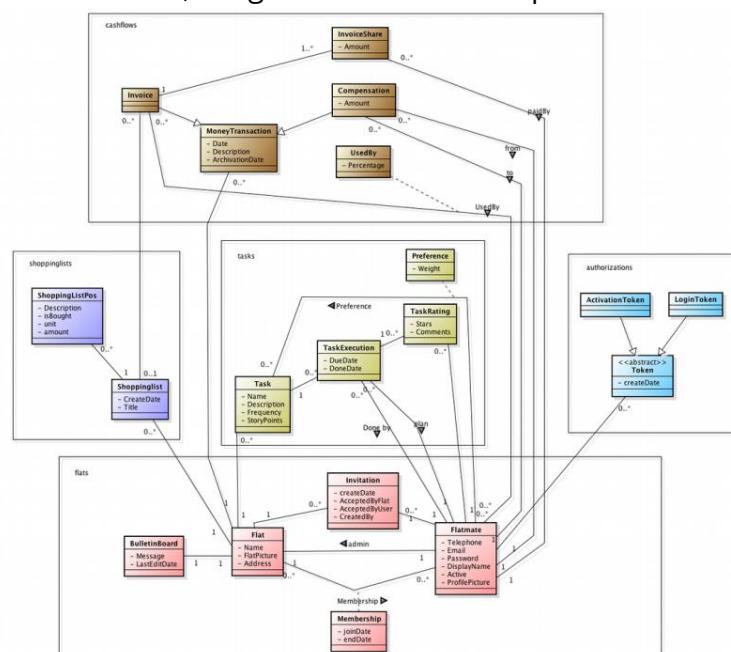
## Projekt-Teillieferungen nach Rollen

Beispiel e-Commerce und Online Shop:

- 👤 1. **Produktmanager (PM)** und **Admin** Dateneingabe/-pflege, Datenmigration (von altem Shop) und Datenbereinigung
- 👤 2. **Einzelkunde** ab hier kann verkauft werden
- 👤 3. **Marketing** für Verkaufs-Aktionen und Print (Flyers, Kataloge, Inserate)
- 👤 4. **Übersetzer** Texte de → fr/it
- 👤 5. **Firmenkunden** mit Sammel-Warenkörben und -Lieferungen
- 👤 6. **Aussendienstler** Tablets für Kundenbesuche
- 👤 7. **Drittfirmen-PM** für Datenimport und Datenpflege des eigenen Sortiments (z.B. BOSCH oder Alessi)

## Projekt-Aufteilung im Domain-Modell

Fünf Bereiche, die gut nacheinander implementiert werden können, von unten nach oben.



## Projekt-Teillieferungen geografisch

Beispiel Steuersoftware:

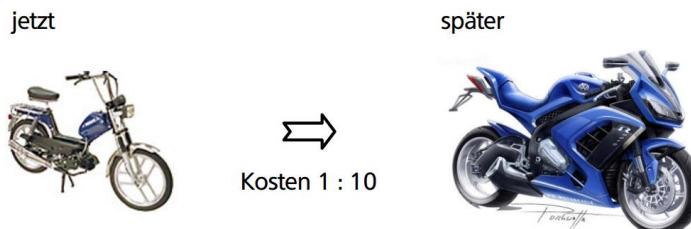
- Erst für Kanton Uri implementieren (relativ klein und einfach)
- Dann für Kanton Thurgau (grösser, wenig Sonderregelungen)
- Dann für Kanton Bern (viele Sonderregelungen wegen Bund)
- Dann für Kantone Zug und Schwyz (Sonderregelungen für internationale Gesellschaften)
- Danach den Rest der Schweiz

Funktioniert(e) auch so bei MIGROS-Kassen-SW, bei Bezahlsystem Twint, bei BMW-Vertragshändlern, etc.

## Story Splitting – Aufteilung von zu grossen Arbeitspaketen

### Von der Basis-Version zum Voll-Ausbau

Erst einmal alle Optionen, alles «nice to have» weglassen. Wir stellen etwas grundsätzlich Funktionierendes hin, aber mehr als die Basis-Funktionalität ist nicht da. Später wird inkrementell geliefert.



### Beispiel 1 – US033 Login

Login auf Website als Sicherheitsmassnahme:

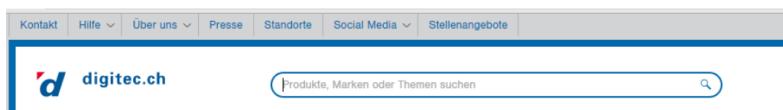
1. Gar nichts: Website ist nur firmenintern erreichbar.
2. Minimal: mit .htaccess, ist in einer halben Stunde gemacht  
<http://www.askapache.com/htaccess/htaccess.html>
3. Ausbaustufe: per Programmierung implementieren, mit abgestuften Berechtigungen: nicht alle sehen alle Seiten und dürfen alles machen. User, Rollen und Berechtigungen aus XML-Datei lesen.
4. Ausbaustufe: User, Rollen und Berechtigungen sind in DB und können von Admins via Web-Interface editiert werden.
5. Ausbaustufe: User, Rollen, Berechtigungen von Active Directory o.ä. holen (und dann war 3. überflüssige Arbeit...)

Jede Ausbaustufe kann als separates Arbeitspaket „verkauft“ werden.

### Beispiel 2 – US041 Produktsuche

Sprint Review: Die User Story "Produktsuche" ist in diesem Sprint nicht fertig geworden. Somit taucht diese Story im nächsten Sprint noch einmal auf. Aber bevor wir zum nächsten Thema übergehen schauen wir uns diese User Story genauer an. Könnten wir die US041 Produktsuche aufteilen?

- Was wäre hier eine Minimal-Version?
- Was wäre hier optional, erst später zu liefern?



## Abschluss – US041 Produktsuche aufgeteilt

Diese User Story wird in 8 User Stories (Tasks, Arbeitspakete) aufgeteilt: a) US041 umgeschrieben in "Basis-Version" und b) 7 zusätzliche Stories.

- **US041** Produktsuche Basis-Version (inkl. Teilwörter-Suche)
- **US084** Suche mit Kaskade in mehreren Feldern (Felder TBD)
- **US085** Suche mit Synonym-Liste
- **US086** Suche mit ähnlich geschriebenen Wörtern, inkl. Einzahl/Mehrzahl
- **US087** Suche mit Kategorie-Begriffen
- **US088** Erweiterte Suche (Kategorien, Publikations-Datum, ... TBD)
- **US089** Suche liefert nie leere Liste
- **US090** Vorschläge von Suchbegriffen während Eintippen

### Generell: Basis-Funktionalität ++

- Nicht nur 'sunny case' sondern alle Varianten und Ausnahmen implementiert
- Die SW reagiert tolerant und robust auf Fehleingaben
- Läuft auch auf Mobile und mit allen denkbaren Bildschirmgrößen, auch extrem hochauflösende (Retina)
- Mit Verschlüsselung, Security
- Hohe Abdeckung mit Unit Tests
- Volle Internationalisierung (UTF8, Texte übersetzt, Sortierreihenfolge...)
- Macht schöne Fehlermeldungen
- Kontext-sensitive Hilfe, auch in allen Sprachen
- Suche berücksichtigt auch Synonyme und Kategorien
- Berücksichtigung der lokalen Feiertage bei den Datumsberechnungen
- Bilder in verschiedenen Auflösungen

### Basis-Funktionalität vs. Voll-Ausbau funktioniert fast überall

Diese Methode „von primitiv zu vergoldet“ kann sowohl auf ein ganzes Projekt als auch auf Arbeitspakete angewendet werden.

User Story: „Einfach-Version“  „Superschönes Feature“

Projekt: „Es funktioniert“  „Wirklich alles fertig“

### Ideale Grösse von Arbeitspaketen (User Stories)

**Durchschnitt** Was 1 Person in ¼ eines Sprint schafft

**Maximum** Was 1 Person in 50 – 70 % eines Sprint schafft

### Fallbeispiel:

B2B Webshop; 100'000 Artikel bestellbar, 60 Mio Umsatz pro Jahr

Neuer Shop: 2 Jahre Entwicklungszeit, Kosten ca. 2.5 Mio CHF Team von 5 Entwicklern plus PL plus PO (ca. 2500 Personentage Entw.) 940 User Stories, 47 Epics

=> Durchschnitt: 2.5 Arbeitstage pro User Story bei Sprintlänge 2 Wochen genau des Sprints. ¼

Übrigens: somit kostete 1 User Story im Schnitt CHF 2500

## User Stories

Sind umsetzungsorientiert, weniger kundenorientiert. Sie müssen klein sein, damit sie in einen Sprint passen. Zudem müssen sie als Ganzes in kurzer Zeit programmierbar sein (deswegen auch «umsetzungsorientiert»). Sie bieten (leider) oft keinen Kontext.

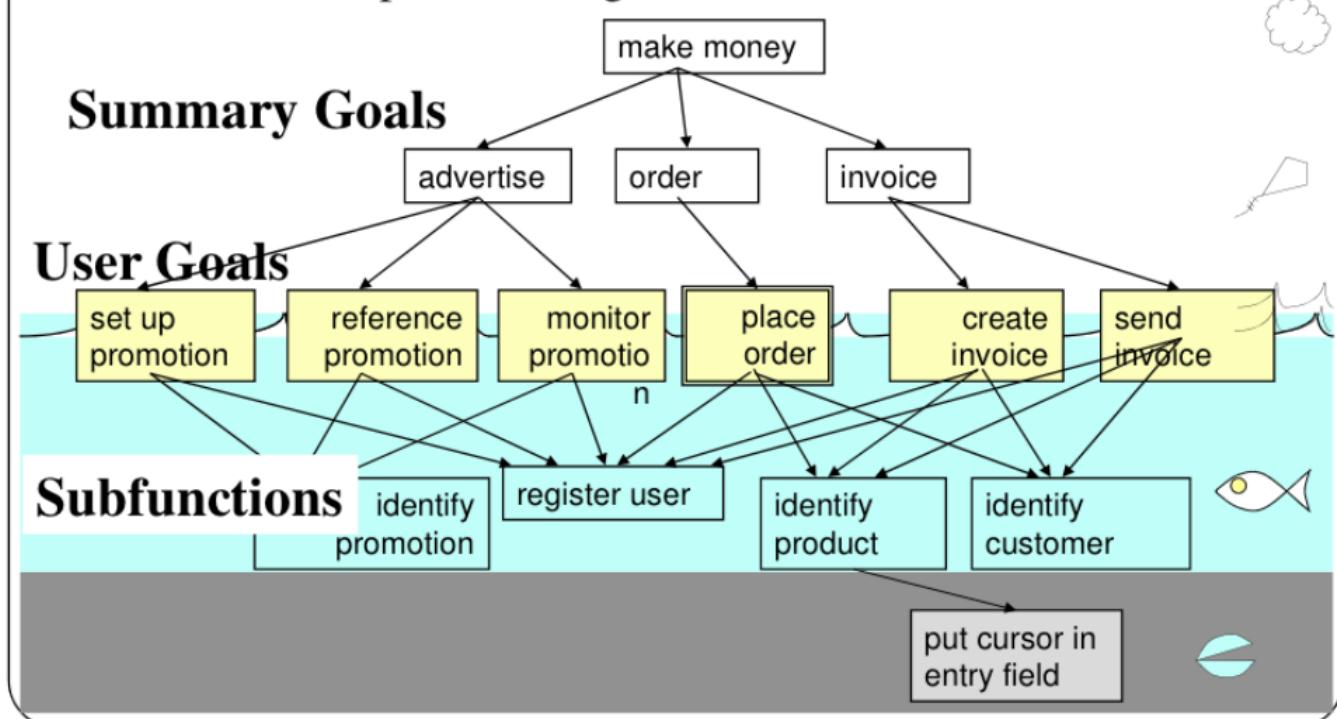
## Use Cases

Sind Kundenorientiert, oft komplett ohne Blick auf die Implementierung. Sie dürfen nicht zu klein sein (sind es aber oft, bei Ungeübten). Use Cases bilden oft Geschäftsprozesse ab (sind darum auch grösser/länger). Zudem zeigen Sie die Akteure und deren Motivation («Das will ich erreichen»). Des weiteren zeigen Sie einen Ablauf, der aus mehreren Operationen und Interaktionen besteht und die Funktionen im Kontext inklusive der Reihenfolge.

### Use Cases at User Goal Level = Sea Level

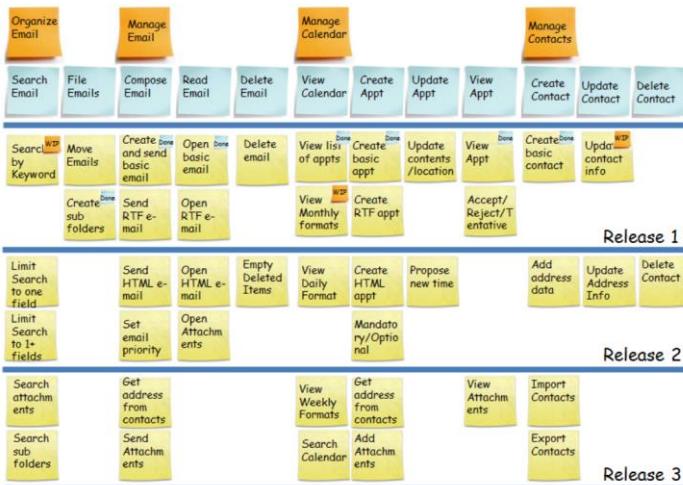
## Level: Summary --- user-goal --- subfunction ( or kite --- sea-level --- fish )

The *Altitude* metaphor: User goals are at sea level.



©Alistair Cockburn 2005

Slide 56



Umgekehrt zum Story Splitting gibt es auch das Story Mapping. Dabei werden kleine User Stories zu grösseren Themenbereichen (meist Epics) zusammengefasst.

### Sinn und Zweck

Besserer Überblick, besserer Fokus auf einzelne Themenbereiche in Bezug auf Priorisierung und Fortschritt. Es kann sein, dass der Kunde nur die Epics priorisieren will, nicht die einzelnen User Stories.

### Epics – User Stories – Tasks

Je nach Grösse des Projektes (Anzahl User Stories), lohnt es sich, verschieden tiefe Hierarchien aufzubauen.

Winziges Projekt Ohne Hierarchie	Kleines Projekt 2 Hierarchiestufen	Mittleres Projekt 3 Hierarchiestufen
50 User Stories	8 Epics ( $\approx$ Use Cases) 120 User Stories  oder: 8 User Stories ( $\approx$ Use Cases) 120 Tasks	12 Epics ( $\approx$ Use Cases) 200 User Stories 1000 Tasks

### Die kleinste Einheit – das Arbeitspaket

Je nach Aufbau der Hierarchie ist entweder eine **User Story** oder ein **Task** oder (bei ganz grossen Projekten mit 4-stufiger Hierarchie) ein **Sub-Task** das kleinste Element, das in einem Sprint Backlog abgearbeitet wird.

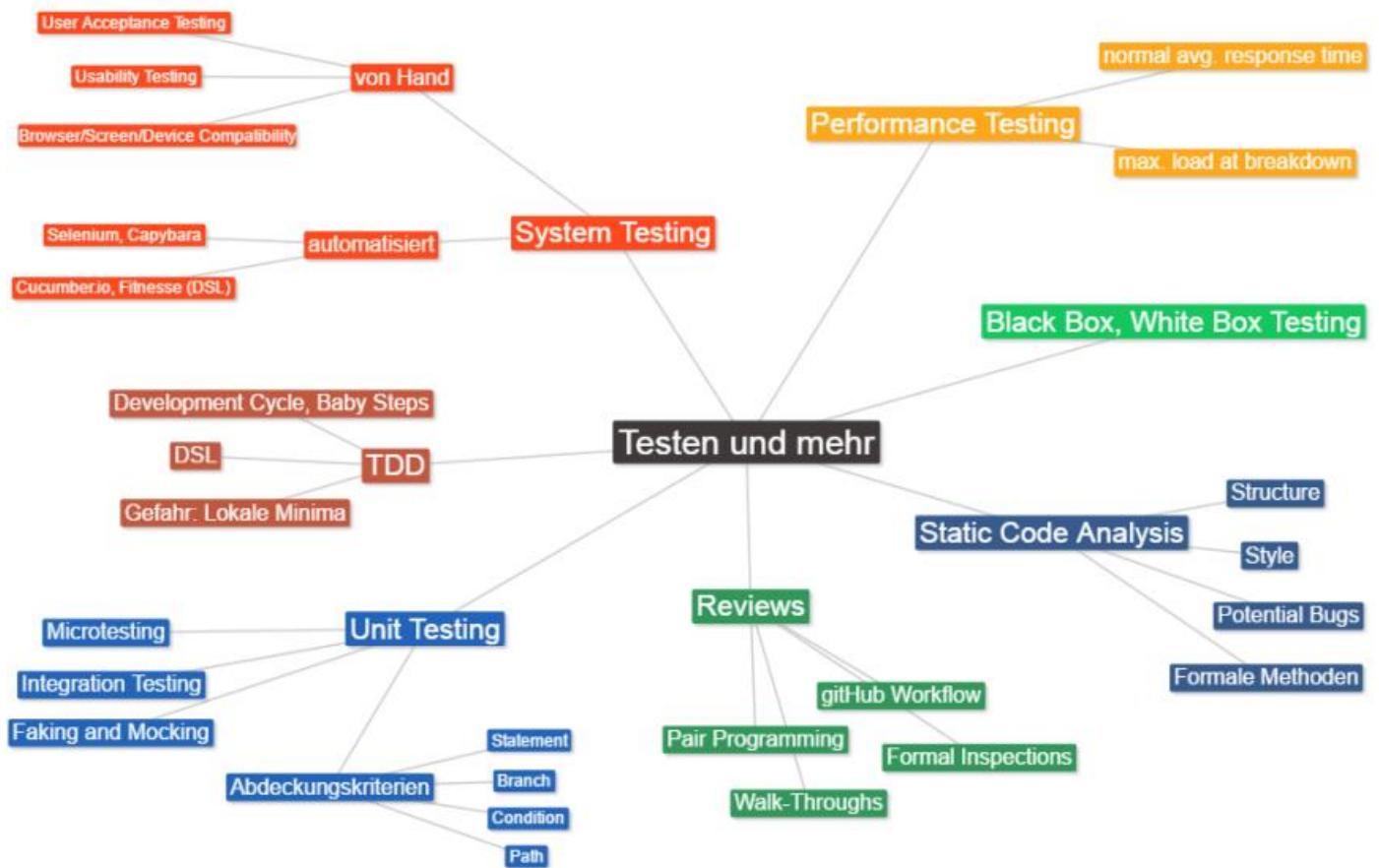
Dieses kleinste Element sollte in der durchschnittlichen Grösse etwa  $\frac{1}{4}$  einer Sprintlänge für eine 100% arbeitende Person sein. Und da sich sehr viele Menschen auf Sprintlänge 2 oder 3 Wochen geeinigt haben, ist diese durchschnittliche Grösse sehr oft ähnlich:

«Arbeitspakete im Sprint Backlog haben eine Grösse von durchschnittlich 2 – 4 Arbeitstagen.»

### Nutzen von Story Mapping

Eher ein Überblick «was ist fertig». Die Kosten-Aufschlüsselung wird hierarchisch dargestellt (bringt vielleicht nicht so viel). Zudem will der Kunde eventuell nur die höher liegenden Entitäten priorisieren, die tiefsten sind ihm zu detailliert und zu technisch. Vermutlich sind es in diesem Fall auch zuviele davon.

# Testen in grösseren Projekten



## Unit Testing – Klar!

```

public void testFloatWithTwoDigits() {
    f = 34.5678F;
    expResult = " 34.57";
    result = Helper.floatWithTwoDigits(f);
    assertEquals(expResult, result);
}

public void testFloatWithTwoDigitsRounding() {
    float f = 2.009F;
    String expResult = " 2.01";
    String result = Helper.floatWithTwoDigits(f);
    assertEquals(expResult, result);
}

public void testFloatWithTwoDigitsLargeNegative() {
    f = -456.789F;
    ...
}
  
```

```

public static String floatWithTwoDigits( float f ) {
    return String.format( "%6.2f", f );
}
  
```

### Prinzip

Ist einfach zu machen, laufen automatisiert, sind schnell ablaufend und haben ein einfaches Resultat.

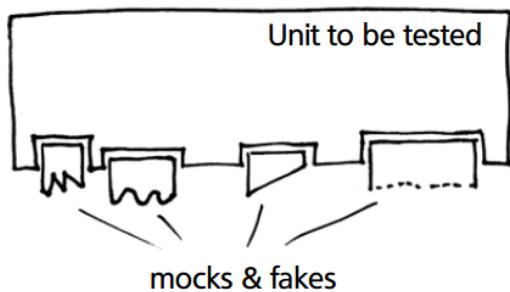
### Microtesting

Die Code Unit to be testet ist meistens eine Methoden (selten auch ganze Klasse). Ein Microtest ist isoliert, daher oft mit Faking & Mocking. Die Unit Tests sind automatisiert sodass «test – change/improve code – test» gemacht werden kann.

### Definition

Microtests sind Unit Tests, die eine Klasse (eine Methode einer Klasse) in Isolation testen. Generell ist es einfacher, die Units in den unteren Layern isoliert zu testen. Weiter oben wird Faking/Mocking eingesetzt, um das zu testende Teil wie gewünscht isolieren zu können.

Dieses Thema wurde bereits im Rahmen eines eLearning behandelt. Fakes (manchmal auch «Stubs» genannt) sind ein simpler Ersatz, meist fixe Daten. Mocking sind intelligente Rückgabewerte die schauen/verifizieren, wie das gemockte Teil aufgerufen wird.



### To Mock or not to Mock?

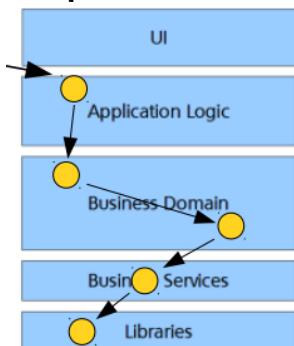
File-Synchronisation via FTP Server: Dateien hochladen, runterladen  
Mocken? den FTP-Server? vielleicht, aber eher nicht  
Laptop mit Linux hinstellen, FTP Server starten, lokal testen via Ethernet-Kabel. Test des kompletten Systems in kontrollierter Umgebung, Test-Szenarien einfach aufzusetzen (Copy Directory)  
Seltsame Effekte mit z.B. Dateinamen welche Leerzeichen enthalten tauchen dann tatsächlich auf (ein Mock würde das vermutlich übersehen). Ebenso beispielsweise Timeout-Probleme.

Externes System, beispielsweise Börsenwerte

Faken, Mocken? ja, klar, wir wollen ja bestimmte Szenarien testen, z.B. „steigende Pharma-Aktien“

## Integrationstest

### Beispiel



Sie Rufen in einem Unit Test getAllOpenIssues() auf. Dieser Aufruf wird dann eine ganze Reihe anderer Routinen aufrufen, bis hinunter zur Datenbank (In die Datenbank müssen – damit das funktioniert – zuerst die richtigen Testdaten eingeschoben werden, sonst ist der Test nicht reproduzierbar). Und wenn alles korrekt zurückgeliefert wird, dann ist dieser Unit Test (eben ein Integrationstest) grün.

### Higher-Level Unit Testing

Integrationstests testen das Zusammenspiel von Klassen - dort wo viele Missverständnisse entstehen können, trotz static type checking (Reihenfolge der Calls, Wertebereich von Parametern, Fehlerbehandlung, ...). Integrationstests auf oberster Ebene (direkt unter UI) testen das System als Black Box: früh definiert z.B. mit System-Sequenzdiagrammen, Contracts. Integrationstests direkt unter dem UI testen die Funktionalität des Systems, so wie es von aussen definiert wurde.

### Vorteile von Integrationstests

Integrationstests sind in der Praxis oft deutlich wertvoller als Mircotest Gründe dafür sind:

- Integrationstests entdecken mehr Fehler, weil sie realistischere Szenarios testen.
- Integrationstests sind langlebiger als Microtests, d.h. Integrationstests müssen nicht so oft umgeschrieben werden.
- Integrationstests können einen Teil der nichtautomatisierbaren End-To-End Tests (UI, Browser) ersetzen – und die verbleibenden konzentrieren sich dann rein auf die Darstellung.
- Integrationstests bieten die Möglichkeit des Einsatzes einer DSL (Domain Specific Language) und Einbindung der User beim Formulieren von automatischen Tests (z.B. cucumber.io). Das könnte sich bei Integrationstests lohnen.
- Integrationstests führen nicht zum ‚lokales Minimum‘ Effekt (s. unten)

## Nachteil von Integrationstests

Integrationstests können hohe Laufzeiten zur Folge haben, d.h. ein Build mit ausführlichen Tests dauert länger als ein paar Minuten, u.U. länger als eine Stunde. Echte Continuous Integration (build/test nach jedem Commit) ist dann nicht mehr machbar. Das ist aber der einzige wirkliche Nachteil.

### Beispiel für zeitintensive Integrationstests

DB-Szenario unten hineinschieben (ist jedesmal im Bereich von 20 Sekunden bis Minuten). Dann lässt man ein paar Unit Tests in Kombination laufen. Danach: nächstes Szenario.

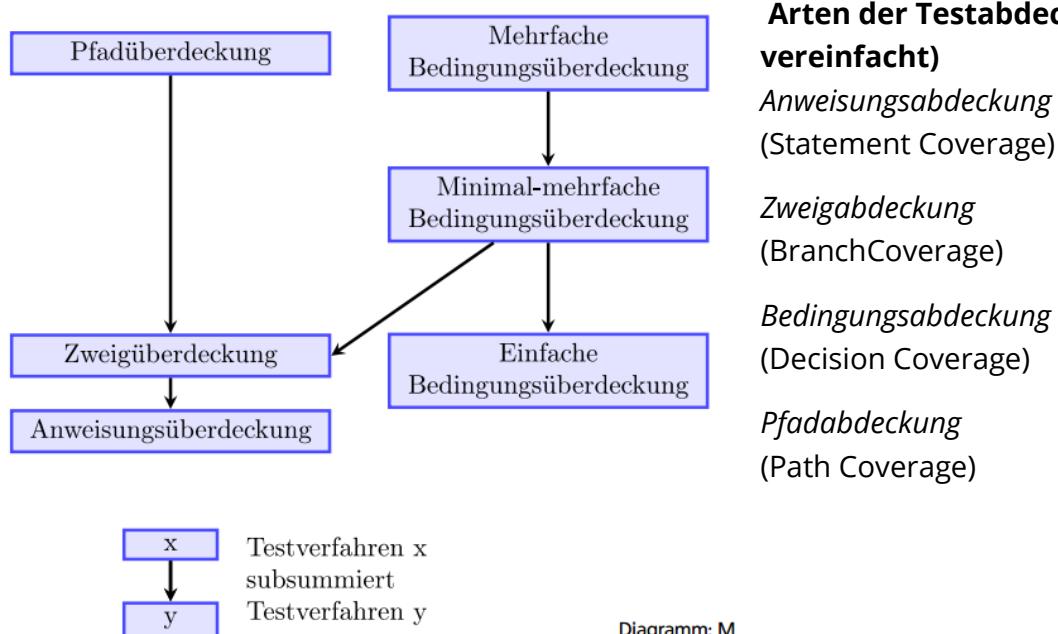
## Schwachstellen von Microtesting allgemein

- Testabdeckung sagt noch nicht viel aus – aber man fühlt sich sicher
- Unit Tests sind (leider oft) nichtssagend („ $2+2 = 4$ “)
- Testen in Isolation zeigt die Schwachstellen bei Integration nicht
- Unit Tests verleiten einen beim Refactoring oft dahingehend, dass man wie auf einem lokalen Minimum stecken bleibt – man kommt nicht auf die bessere Lösung, welche größeres Refactoring verlangen würde.
- Unit Tests überleben strukturell weitreichenderes (größeres) Refactoring meist nicht – verlorener Aufwand.
- Microtesting funktioniert gut bei Libraries (unterste Layers), darüber muss man dann Mocken: Fehlerquelle, da Simulation mit Annahmen

## Wann ist genug getestet?

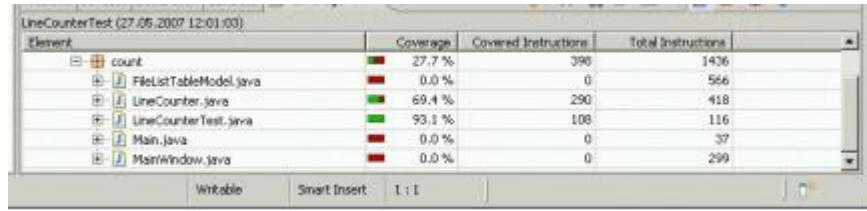
Haben wir genug Unit Tests? Oder sollten wir mehr testen? Darauf gibt es mehrere Antworten. «Wir haben seit Tagen keinen Fehler gefunden» oder «Wir haben 83 Prozent Testabdeckung erreicht». Grundsätzlich ist keine Antwort gut. Wir können auch 100% Testabdeckung erreichen, wenn wir alle speziellen Fälle nicht beachten.

## Testabdeckung



## **So funktioniert die Messung mit der Testabdeckung**

Der gesamte Code (Klassen + Unit Tests) wird mit Zählern auf jeder Anweisung (Zeile) instrumentiert.



Die Unit Tests werden auf dem zu testenden Code ausgeführt.

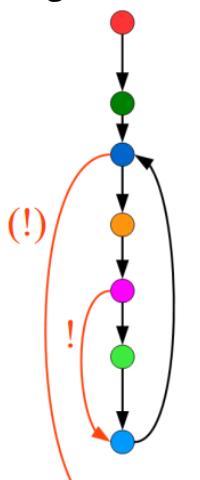
Die Zähler werden ausgewertet und der Code schön dargestellt: (Rot heisst: "Zähler ist Null"), plus Statistik.

## Anweisungsabdeckung

Anweisungsabdeckung heisst, dass jede Anweisung, welche durch die Unit Tests ausgeführt wird, zählt. Der Prozentsatz wird wie folgt berechnet. «Ausgeführte Anweisungen / Gesamtanzahl der Anweisungen».

Die Anweisungsabdeckung ist der **Standard** bei den Testabdeckungs-Verfahren. Alle mir bekannten Tools messen nur die Anweisungsabdeckung. Achtung: die Anweisungsabdeckung ist die minimalste und am wenigsten aussagekräftige aller Testabdeckungs-Verfahren (siehe folgende Folien). Dafür ist sie einfach zu ermitteln.

## Zweigabdeckung



```
void zaehleZeichen(int &VokalAnzahl, int &Gesamtanzahl) {  
    char Zchn;  
  
    cin >> Zchn;  
  
    while ((Zchn >= 'A') && (Zchn <= 'Z') &&  
           (Gesamtzahl < INT_MAX)) {  
  
        Gesamtzahl = Gesamtzahl + 1;  
  
        if ((Zchn == 'A') || (Zchn == 'E') || (Zchn == 'I') ||  
            (Zchn == 'O') || (Zchn == 'U')) {  
  
            VokalAnzahl = VokalAnzahl + 1;  
        }  
  
        cin >> Zchn;  
    }  
  
    return;  
}
```

Zweigabdeckung heisst, dass jeder Zweig, welcher durch die Unit Tests ausgeführt wird, zählt. Der Prozentsatz wird so berechnet «Abgedeckte Zweige / Gesamtanzahl der Zweige».

Hauptsächliche Abweichung zur Anweisungs-Abdeckung:

- IF ohne ELSE mit Testfall, wo das IF grad ganz ausgelassen wird
  - FOR und WHILE Testfälle bei denen man gar nicht in die Schleife geht

## Testdaten für minimale Abdeckung

## Anweisungsabdeckung

"A"

## Zweigabdeckung

"AB" (vermutlich auch "&")

### *Bedingungsabdeckung*

"\$", "G", "{" "A", "E", "I", "O", "U", "T"

**Bedingungsabdeckung:** "Das Verhältnis von ausgewerteten atomaren Werten (Term, Bedingung, ...) innerhalb von Ausdrücken zu allen vorhandenen atomaren Werten in einem Modul."

**Pfadabdeckung:** "Das Verhältnis der getesteten Pfade (Wege im Kontrollflussgraph), zu allen möglichen Pfeilen in einem Modul."

Das Kriterium der Bedingungsabdeckung kann sehr hilfreich sein, noch weitere Testfälle zu entdecken (s. auch Testdaten-Selektion mit Äquivalenzklassen).

Pfadabdeckung führt sehr schnell zu einer potentiell unendlichen Anzahl von Fällen (wegen Schleifen).

Ist das gut: «83 % Test-Abdeckung»?

100% Anweisungsabdeckung der gesamten Code-Basis kann fast nie erreicht werden. Gründe:

- UI Code kann i.d.R. nicht Unit getestet werden.
- Es gibt Code-Teile, die nur extrem schwer mit Unit Tests abzudecken sind, z.B. Exceptions, z.B. wie fange ich eine „Disk full“ Exception oder eine „Connection lost“ Exception? (wohlgemerkt: zuverlässig wiederholbar während eines Unit Tests).

100% Test-Abdeckung ist nur Anweisungsabdeckung und das ist für eine isolierte Methode manchmal schnell erreicht.

Implementierung:

```
public void static abs(int x) {
    return -x;
}
```

Unit Test:

```
public void testAbs() {
    assertTrue(MyMath.abs(-2) == 2);
}
```

100% Test Coverage!

## Trügerische Sicherheit mit Microtests

```
// Moves X-Ray to desired position
void positionXRay(Coordinate c) {
    if (c.x >= 0 && c.x <= 100 && c.y >= 0 && c.y <= 100) {
        moveXRayTube(c)
    }
}

...
read(c1);
positionXRay(c1);
img1 = takeXRay();
...
```

(Code-Beispiel von Luc Bläser)

Mit einem einzigen Testfall (z.B.  $c = (5,5)$ ) erreicht man für `positionXRay` 100% Anweisungsabdeckung. Aber so entdeckt man den Fehler nicht. Der unentdeckte Fehler hatte ernsthafte gesundheitliche Konsequenzen für Bestrahlungspatienten.

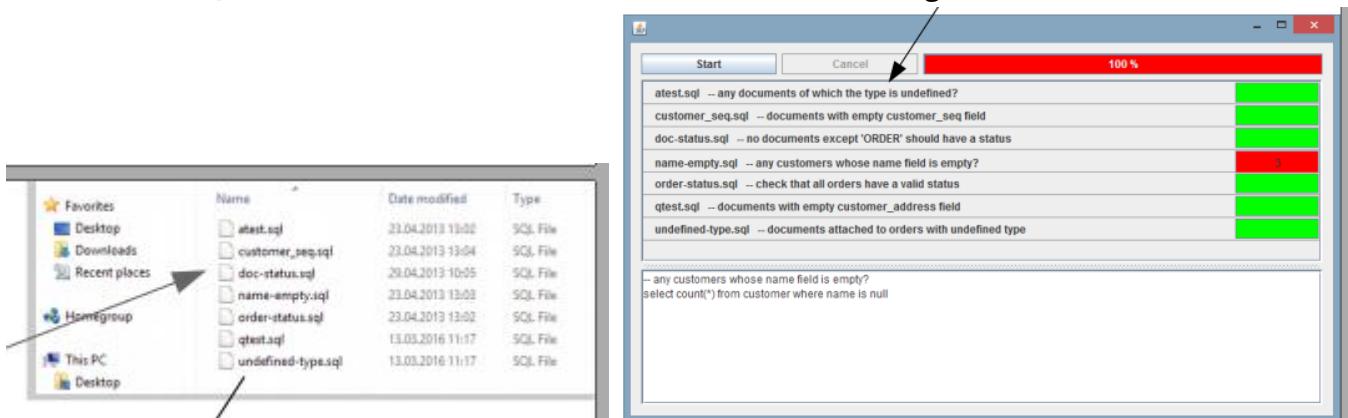
Eine hohe Testabdeckung ist eine notwendige aber nicht hinreichende Bedingung. Also streben Sie eine hohe Testabdeckung an, aber seien Sie mit der Zahl allein nicht zufrieden. Machen Sie lieber Integrationstest, nur nur Microtests. Zudem kommt es auf die Qualität der Testfälle an und nicht auf die Abdeckung.

## Konsistenz-Tester für die Datenbank

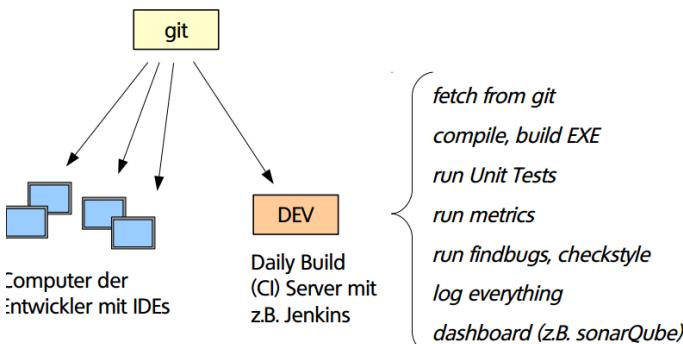
Auch die Datenbank sollte getestet werden und zwar auf ihre Konsistenzbedingungen.

<ul style="list-style-type: none"> <li>• Gibt es Kunden, bei denen das Namensfeld leer ist?</li> <li>• Gibt es Bestellpositionen ohne Bestellung?</li> <li>• Gibt es unerledigte Bestellungen, die älter als 60 Tage sind?</li> <li>• Gibt es Bestell-Dokumente, die einen Status 'cancelled' haben und dennoch Bestellpositionen aufweisen?</li> <li>• Gibt es Merklisten mit ungültigen/veralteten Artikelnummern?</li> <li>• Gibt es Rechnungen mit ungültigen Verweisen auf Bestellungen?</li> </ul>	<p><b>SQL Beispiele</b></p> <pre>-- Uebersetzungen ohne passenden Namensschlüssel SELECT * FROM TRANSLATION LEFT OUTER JOIN NAMES ON UPPER_STRING = NAME WHERE NAME IS NULL  -- gibt es AuslagerinSTRUCTIONS, die eine ungültige location.id haben? select inst.id, inst.statecode from instruction inst, location loc where inst.statecode in ('R', 'A', 'P') and inst.type = 'A' and inst.fromlocationid (+) = loc.id and loc.id is null;  -- steht dieser Artikel in locations, die nicht artikelrein sind? select c.id, c.code, a.code from carrier c, stockeditem st, article a, location loc where st.articleid = a.id and st.carrierid = c.id and c.locationid = loc.id and c.locationid in (select unique loc.id from stockeditem st, article a, carrier c, location loc where st.articleid = a.id and st.carrierid = c.id and c.locationid = loc.id and a.code = 'F00C3E2045') and a.code &lt;&gt; 'F00C3E2045';</pre>
--	---

Eine Lösung dafür ist ein Unit Test ähnliches Prinzip. Da werden SQL Files in einem Verzeichnis abgelegt. Als Kommentarzeile gibt man zu Beginn an, was das Skript testet. Jede SQL-Anweisung sollte dann im Gut-Fall die Zahl Null zurückgeben. Häufig wird mit left outer joins gearbeitet «Wo fehlt was?». Diese SQL Files werden dann nachher alle nacheinander ausgeführt.



## Build Server



## Daily Build / Continuous Integration (CI)

### Daily Build, Nightly Build

Einmal alle 24 Stunden (weil der Build und die Tests sehr lange dauern, z.B. 6 Stunden)

### “Daily Build»

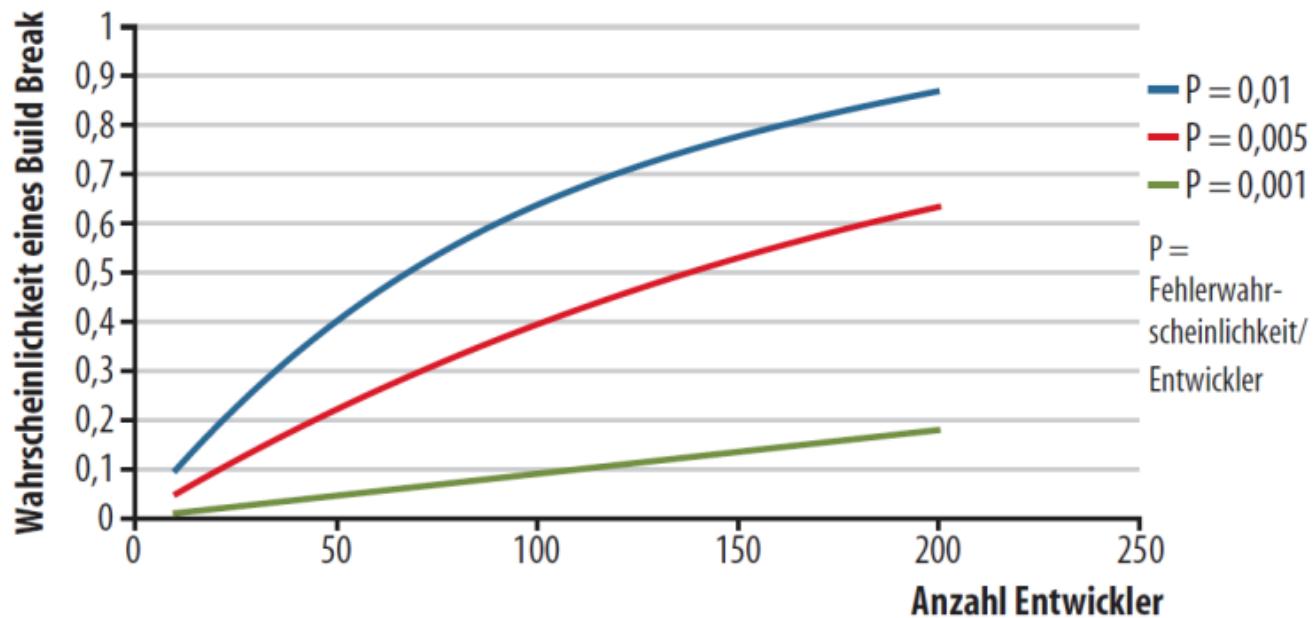
Alle drei Stunden (weil der Build jedesmal ca. eine Stunde dauert).

### Continous Integration

Jedesmal, wenn ein 'commit' auf den Haupt-Zweig gemacht wird (weil ein Build weniger als 10 Minuten dauert)

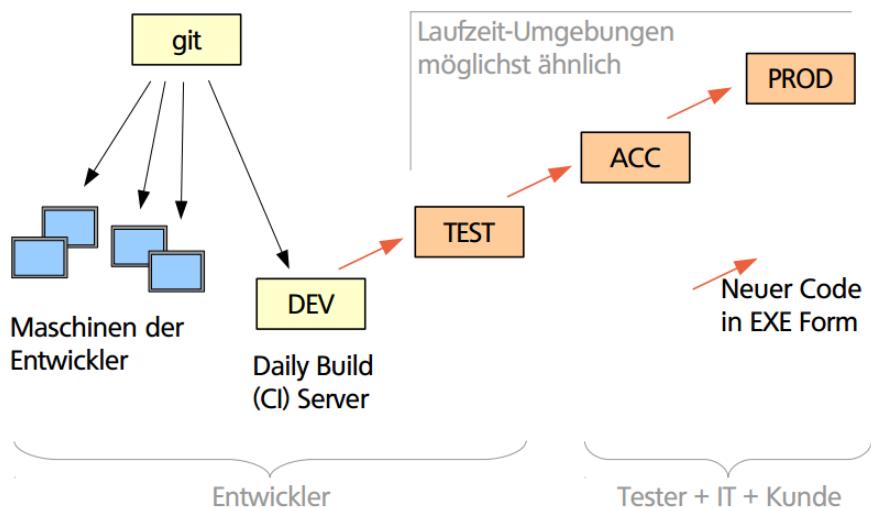
### CI Mixed

Tagsüber Continous Integration mit allen schnellen/billigen Tests, nachts (oder alle drei Stunden) die vollen Tests.



In einem Engineering Projekt wird dies wohl daher gar nicht eintreten. Als Problemlösung können Teilsysteme entkoppelt werden und separat gebaut werden.

## Server Setup



## Server-Umgebungen

### PROD

Produktiv-Umgebung, muss stabil sein, keine Experimente! Neue Releases nur alle paar Monate.

### ACC

Acceptance (manchmal auch STAGE 'Staging' genannt): manuelles Testen durch Kunde/PO und externe Tester; HW & Testdaten möglichst gleich wie PROD (ausser bei Deployment- Änderungen); hier auch Performance-Tests

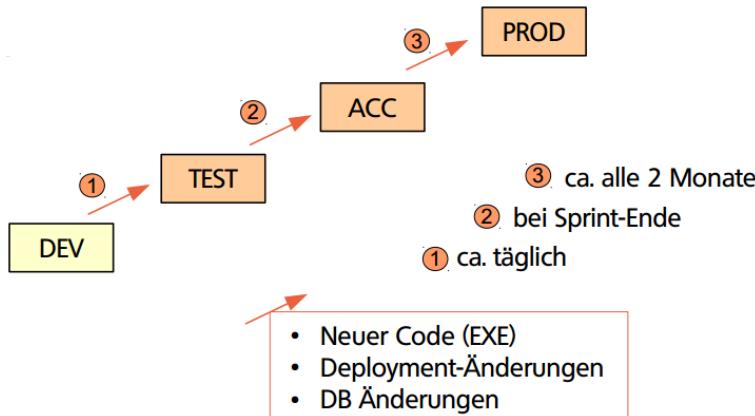
### TEST

Stabile Test-Umgebung für die Entwickler. Alle automatisierten Tests. Erste Performance-Tests

### DEV

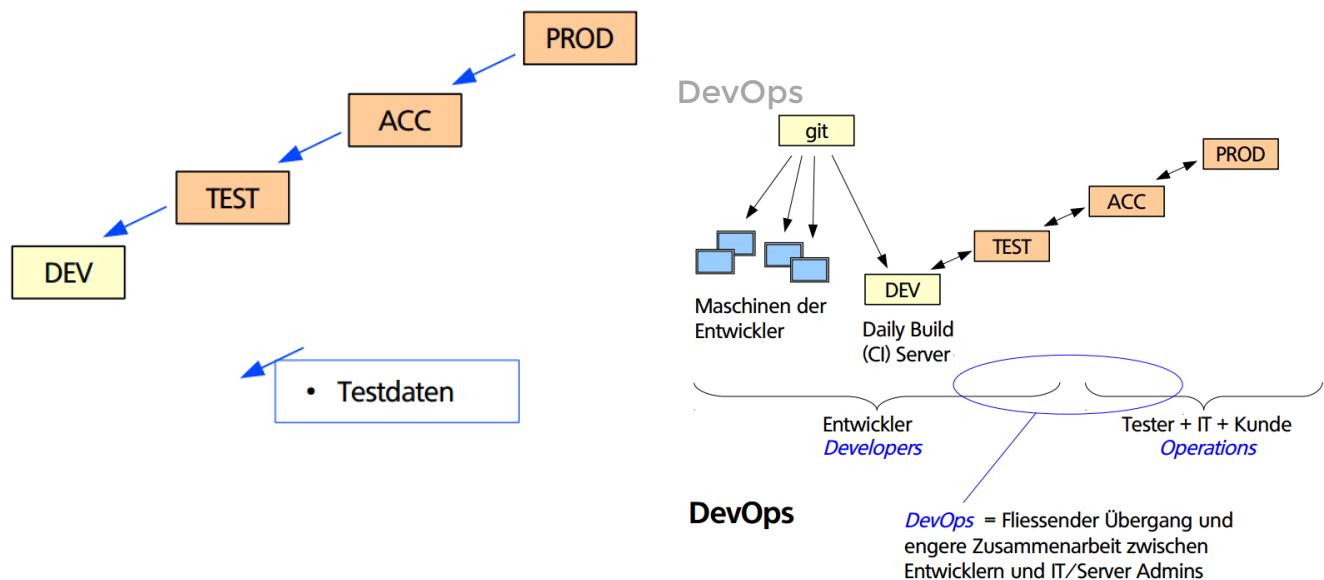
Entwicklersystem (relativ stabil für alle Entwickler), Daily Build / CI Machine

Migration des neuen Codes und des Server Setups mit Skripten. Ziel ist es die Migration auf die nächste Stufe per Knopfdruck durchzuführen.



### Test-Daten

Die Testdaten sollten auf allen Systemen möglichst ähnlich sein, in Menge, Art und Struktur. Die Testdaten kommen meist vom Produktiv-System und müssen oft anonymisiert werden. Daher sollte auch die Datenmigration automatisiert werden.



### Probleme mit Testdaten

Das Problem mit den Testdaten ist der Realitätsbezug. (Menge, echt aussehen & nicht ablenken, echte Probleme z.B. Umlaute und Sortierordnung; Duplikat).

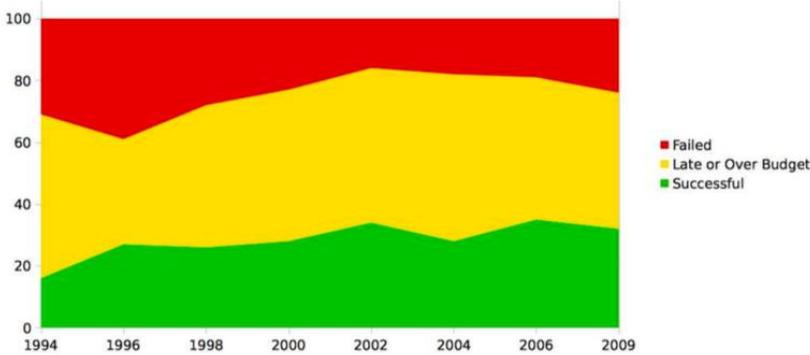
Zufällig generierte Daten sind mit

- Mockaroo.com
- Generatedata.com
- Und datasetgenerator.com möglich.

Diese Daten sind dann zwar auch Testdaten, sie kommen aber relativ nah an die Wahrheit heran.

# Aufwandschätzungen

## Erfolgsquoten von Software-Projekten



Grob gesagt: 30% erfolgreich, 25% scheitern komplett, 45% zu spät oder zu teuer oder beides.

## Horror-Geschichten

### International

- Seattle Mariners' new baseball stadium: estimated in 1995 to cost \$250 million. Completed in 1999 at a cost of \$517 million (Withers 1999)
- Boston's Big Dig Highway construction: estimated to \$2.6 billion, finished for \$15 billion, an overrun of 400% (Associated Press 2003).

### Software

- Irish Personnel, Payroll and Related Systems (PPARS) was cancelled after it overran its €8.8 million estimate by €140 million (The Irish Times 2005).
- The FBI's Virtual Case File project was shelved in March 2005 after costing \$170 million and delivering only 1/10 of its capabilities (Arnone 2005).
- Bank Vontobel and PriceWaterhouseCoopers, February 2001, online banking portal y-o-u, cancelled after cost of sfr. 180 million (estimated figure, reported "not over sfr. 250 mio").

## Software-Projekte in CH-Verwaltungen

Projekt 'Insieme' der Eidg. Steuerverwaltung: gestoppt Sept. 2012 nach ca. 110 Mio. Ausgaben.

«Millionen wegprogrammiert» titelte der «Tages-Anzeiger» über die Kosten des neuen Informatiksystems für das Stadtzürcher Sozial departement: 2006 ging man von Kosten von 11,6 Millionen Franken aus, 2007 waren es 12,08, dann 22 und 2011 schon 29,5 Millionen.

Hunderte nicht kompatible Informatiksysteme und ein ausuferndes Führungsinformationssystem FIS für 750 Millionen Franken: Die Informatikprobleme der Schweizer Armee sind legendär. Laut der Zeitung «Sonntag» werden in der ersten Dekade des 21. Jahrhunderts beim VBS «Hunderte von Millionen in Berater investiert», weil niemand den Überblick über die EDV-Architektur hat.

### Scope Creep

„Das Führungsinformationssystem (FIS) Heer, vor zehn Jahren als Prestigeprojekt der Armee lanciert, werde definitiv nie so funktionieren wie vorgesehen. Deshalb, so der Verteidigungsminister, müssten von den investierten 700 Millionen Franken schätzungsweise 125 Millionen abgeschrieben werden.“

„Die Sicherheitspolitiker staunten. Zum einen wegen der schieren Summe: Mit einem Schaden von 125 Millionen Franken übertrifft das Debakel bei FIS Heer den Abschreiber bei Insieme, dem bis dato grössten IT-Pannenprojekt des Bundes, um 10 Millionen Franken.“

## Software Engineering 2

„FIS Heer funktioniert nur, wenn ein militärischer Festnetzanschluss vorliegt, etwa bei Grossanlagen wie dem WEF. Sobald Aufklärungsfahrzeuge aber vom Kabel getrennt werden, sind sie blind. «Grund dafür ist die fehlende Bandbreite der vorhandenen militärischen Übermittlungsgeräte», teilte das VBS gestern trocken mit.“

„Insider führen das Debakel darauf zurück, dass die Anforderungen an FIS Heer während der Beschaffung laufend erhöht wurden. Sollten die mobilen Aussenposten zu Beginn nur sehr gezielt Informationen in die Zentrale schicken, habe die Projektleitung bald schon von stehenden Verbindungen und einem fortlaufenden Informationsfluss geträumt. Dabei habe das Bundesamt für Rüstung Armasuisse übersehen, dass die verfügbare Bandbreite dazu gar nicht ausreichte.“

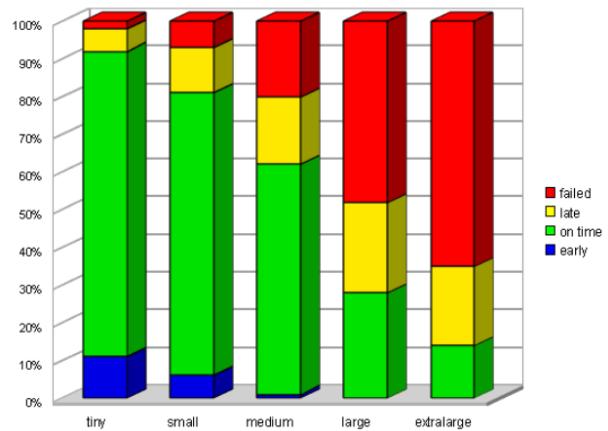
### Auf die Grösse kommt es an

Wann und ob ein Projekt erfolgreich abgeschlossen werden kann. Grundsätzlich ist damit gemeint, dass das Projekt im Budget geblieben ist und termingerecht abgeliefert wurde.

Table 3-1 Project Outcomes by Project Size

Size in Function Points (and Approximate Lines of Code)	Early	On Time	Late	Failed (Canceled)
10 FP (1,000 LOC)	11%	81%	6%	2%
100 FP (10,000 LOC)	6%	75%	12%	7%
1,000 FP (100,000 LOC)	1%	61%	18%	20%
10,000 FP (1,000,000 LOC)	<1%	28%	24%	48%
100,000 FP (10,000,000 LOC)	0%	14%	21%	65%

Source: *Estimating Software Costs* (Jones 1998).



Bei grösseren Projekt reichen nicht einfach nur mehr Leute und mehr Zeit, sondern man braucht aus grösseres Werkzeug und andereres Material & Methoden. (Analogie Hausbau aus SE1).

### Grössenordnungen zusammengefasst

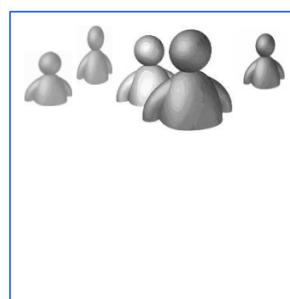
Projekt-Typ	Typische Teamgrösse & Laufzeit	Typische Grösse (lines of code)	Typische Anzahl Code Units	Grössen-Bereich (lines of code)	Engineering Methoden und Herausforderungen
Anfänger	Eine Person, lernt gerade programmieren	50	1	10-200	
	Winzig	1000	10	200-2000	Automatisierte Unit Tests
	Klein	8000	100	2000-20,000	Infrastruktur (SVN, backup, build server...), coding guidelines, patterns, reviews, metrics
Mittel	10 Entwickler, eine Teamleiterin, ein Jahr	70,000	1000	20,000-200,000	Geplante Architektur, spezialisierte UI Entwicklung, separates Test-Team, aufwendige Daten-Migration, int'l. Bug Tracking (aber immer noch: eine Person kann alles wissen)
	Gross	600,000	10,000	200,000-2 mio	Entwicklung an mehreren Standorten, Aufspaltung des Wissens, Sub-Systeme, die meisten Informationen schriftlich, erhöhter Strukturierungs-Aufwand (dev & org), komplexe IT Landschaft, branching, definierte interne Prozesse, juristische und politische Randbedingungen, QA
XXL	Banking; Microsoft (Win Vista: Entwicklungsteam von 2000, 70 mio Zeilen Code)	5 mio	100,000	> 2 mio	Stark verteilte Entwicklung, kompartimentalisiertes Wissen, nicht vernachlässigbare Build Zeiten, langfristige Planung

Dieser wächst nicht linear sondern mit  $n * (n-1) / 2 \rightarrow O(n^2)$ .

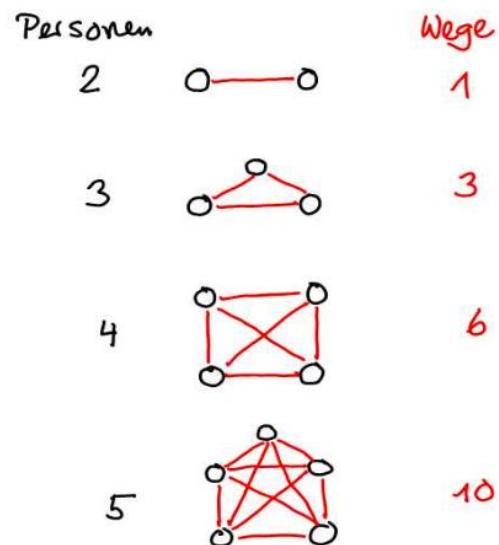
Was ist der Kommunikations-Aufwand bei zwei Personen?



Was ist der Kommunikations-Aufwand bei fünf Personen?



Der Aufwand wächst um den Faktor 10!  
(vgl. die Fläche des viereckigen Kastens)



## Nicht-lineare Faktoren nach McConnell

Kommunikation, Planung, Management, Anforderungsentwicklung, System Funktions Design, Interface Design und Spezifikation, Architektur, Integration, Systemtesting und Dokumentation der Production.

## Weitere negative Grösseneffekte

Es kann zudem zu Reibungsverlusten durch Unterschiede in der Vision, im technischen Verständnis oder bei den Sprachen kommen.

Des Weiteren kann die geographische Verteilung einen sehr grossen Effekt. Es fördert das Silo-Denken (eigener Garten) und behindert damit die Kommunikation.

Unbeabsichtigtes Duplizieren in grossen Organisationen kann auch passieren.

## Schlüsse

### „Small is beautiful“ -> „Keep it small“

Falls Sie die Grösse Ihres Projektes (den „scope“, die Menge der gewünschten Funktionalität) nicht im Griff haben, spielt es keine Rolle, ob Sie exzellente und top-motivierte Mitarbeiter haben, über die sorgfältigste Planung verfügen, und mit den besten Werkzeugen und Methoden arbeiten. Sie werden höchstwahrscheinlich keinen Erfolg haben.

**Grösse in Software hat viel mit Komplexität zu tun.** Und damit wachsen die Schwierigkeiten. Also auch: Komplexität tief halten.

Es gibt keine Wundermittel: seit Jahren ist die Erfolgsrate etwa gleich (siehe Standish CHAOS Reports). Weder AOP, TDD, MDA, MDD, etc. noch Werkzeug X oder Methode Y verbessern den Erfolg wesentlich (Ausnahme: OOx).

## Faustregeln

Machen Sie kein Software-Projekt, das länger als neun Monate dauert. Packen Sie kein Software-Projekt an, das teurer als eine Million wird. Kein Programmier-Team soll grösser als 10 Personen sein. Einziger Ausweg: Aufteilen und separat planen, ausführen und abliefern. Hilfslinien bei der Aufteilung/Staffelung:

- Erst Kernfunktionen, dann Zusätze, wie Zwiebelschalen (Bsp. Autovermietung: erst Mietgeschäft, dann erweiterte Kundenbetreuung mit Prämienshop, dann live GPS Tracking der Fahrzeuge)
- Geografisch (Bsp. Steuer-Software: zuerst für den Kt. Uri, dann Zug, dann Zürich, dann den Rest der Deutschschweiz ausrollen)
- Kategorien/Abteilungen/Sortiment (Bsp. Logistikkette: zuerst Tiefkühlprodukte, dann Obst und Gemüse, dann den Rest; Bsp. Versicherungen: Erst Lebens-, dann Sachversicherungen)

## Warum verhauen wir uns beim Schätzen?

Dafür gibt es viele Gründe: Politische Vorgaben dominieren (nicht realistische), Zuviel Optimismus, Fehlende Software, Mangelnde Qualität (Quick & dirty Starts), Komplexität und unerwartete Nebeneffekte sowie fehlende Erfahrung (weil jedes System ganz neue Teile/Aspekte hat).

Die vorher erwähnten Gründe sich sicher massgebend für Schätz-Fehler, aber der wichtigste Grund ist, weil wir zuwenig Übung haben.

Schätzen Ist-Zahlen ermitteln (Stundenaufschreibung) Soll-/Ist-Vergleich



... und das mehrmals hintereinander!

Machen Sie das drei oder viel Mal und Sie werden viel bessere Schätzungen abliefern. Schätzen ist Erfahrungssache.



## Aufwand – die drei wichtigsten Faktoren

Der Aufwand in einem SW-Projekt ist hauptsächlich bestimmt durch die **Grösse** (Funktionalität), **Art der Software** (und damit Komplexität) sowie der **Qualität** der **Mitarbeitenden**.

Zwischen der Grösse und dem Aufwand besteht ein nichtlinearer Zusammenhang. (eher exponentiell).

## Einflussfaktoren nach ISBSG.org

- Projektgrösse
- Grösse der Kundenbasis
- Stabilität der Anforderungen
- Kooperation mit dem Kunden (Kunde im Team)
- Team Qualität
- Komplexität des Systems
- Sicherheitsanforderungen

## Top-Down Schätzungen

Top-Down Schätzungen benutzen nur ganz wenige globale Parameter und ein paar Formeln und Annahmen. Üblicherweise wird dies zu Beginn des Projekts eingesetzt.

### Beispiel Mehrfamilienhaus

Was kostet es zwei Mehrfamilienhäuser zu bauen.

#### Infos vom Verband Schweizer Ingenieure und Architekten

Kosten pro Kubikmeter Innenraum für Mehrfamilienhäuser, mittlerer Ausbaustandard, auf flachem Land: 480 CHF. Für ein Einfamilienhaus: 580 CHF.

Annahmen:

- pro Stockwerk 2 Wohnungen à 100m<sup>2</sup> = 400 m<sup>2</sup> Grundfläche
- 4 Stockwerke (inkl. Keller) zu je 2.5m Höhe = 10m Gesamthöhe
- => 4000 m<sup>3</sup> Gebäudevolumen

Simple Rechnung:

4000 m<sup>3</sup> \* CHF 480 (Stand 2002) = 1'920'000 CHF (= 2 Mio)

mit der Teuerung seit 2002 sind damit etwa bei 2.5 Mio

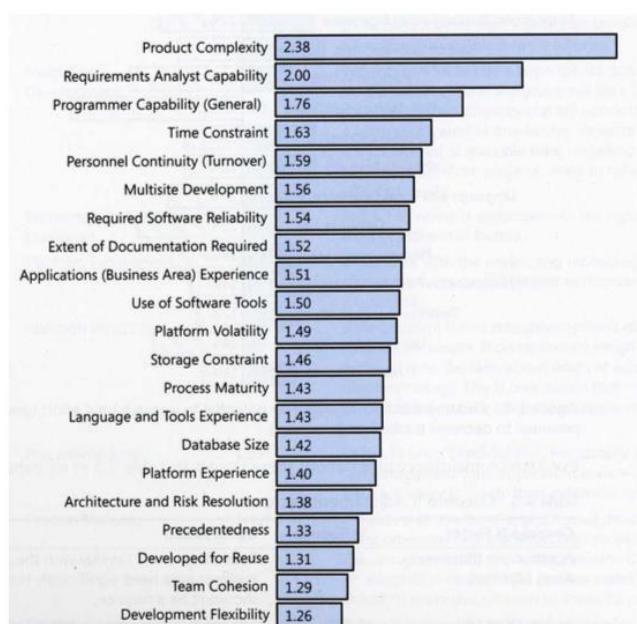
## Algorithmische Schätzungen

### Beispiel COCOMO

**Person-Months PM = 3.2 \* (KDSI) \*\* 1.05**

**Development time = 2.5 \* (PM) \*\* 0.38**

KDSI = K delivered source Instructions d.h. Zeilen Code. Dazu kommen dann die weiteren Einflussfaktoren.



## Function Points

Zählen Sie dafür

- External Inputs (Data Entry Screens)
- External Interface Files (Input, Outputs in Dateien)
- External Outputs (Reports)
- External Queries (Message oder externe Funktionen)
- Logische interne Tabellen

Gewichtet sieht dies dann wie folgt aus.

External inputs	4
External interface files	7
External outputs	5
External queries	4
Logical internal tables	10

Beispiel:

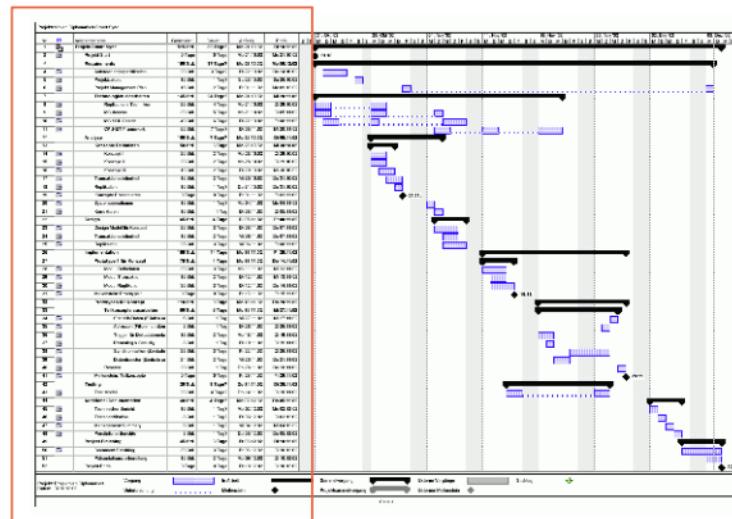
System mit 25 Eingabemasken, 5 interface files, 15 Reports, 10 external queries, 20 DB-Tabellen

$$25*4 + 5*7 + 15*5 + 10*4 + 20*10 = 450 \text{ Function Points}$$

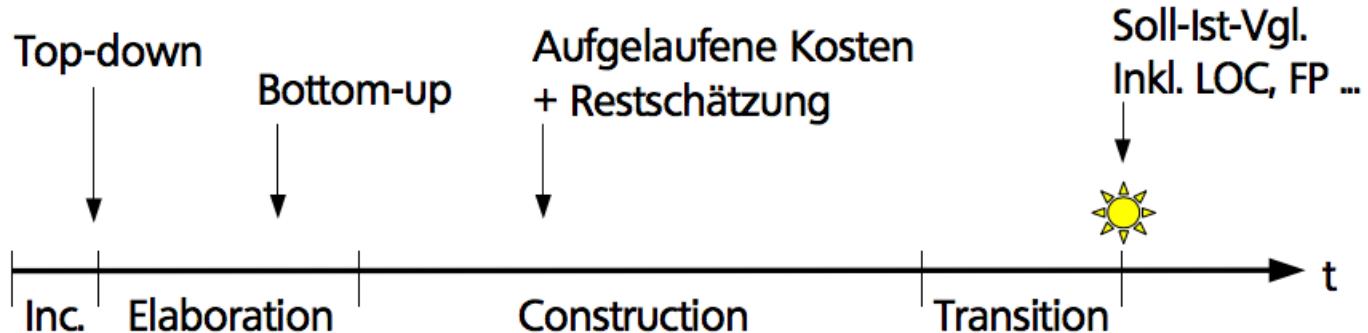
Nachschaufen in Formeln & Tabellen (für ein e-commerce System):  
total 82 Personen-Monate.

## Bottom-up Schätzungen

### Gantt Charts



Arbeitspakete mit den Aufwandsschätzungen auf einer Zeitlinie angeordnet. Inkl. Abhängigkeiten und 'wer-macht-was'. Interessant sind die Arbeitspakete: alle aufzählen, Aufwand schätzen, zusammenzählen (darum: bottom-up).



Zu Beginn Inception/Elaboration: top-down (bis Faktor 4 daneben). Ab zweite Hälfte Elaboration: bottom-up (optimal: +/- 20% ab Ende Elab.) Danach: aufgelaufene Kosten plus Schätzung des Rest-Aufwandes mittels bottom-up (kann evtl. noch sein, dass ganze Teile top-down geschätzt werden). Am Schluss: Daten archivieren, Parameter kalibrieren, auch für top-down.

### Definition - „Zeilen Code“

#### LOC (Lines of Code)

- Selbst geschriebene Zeilen Code, ohne Test-Code, ohne Prototypen (manchmal auch DLOC genannt: Delivered Lines of Code)
- Gezählt als „alle Zeilen minus Kommentarzeilen minus Leerzeilen“
- Dokumentiert und gut getesteter Code, hohe Qualität

Probleme bilden hier generierten Code (z.B. durch den GUI Build), XML, script und config Dateien.

#### Zeilen pro Monat

Pro Programmierer pro Monat:

- 80-150 LOC bei schwierigen Echtzeit-Projekten (Zahl v. IBM, Space Shuttle)
- 300-500 LOC im Schnitt mit einem guten, nicht zu grossen Team
- bis ca. 1000 LOC nur mit kleinen Spitzenteams
- über 1500 LOC ist unglaublich (Pfusch, viel Copy/Paste, generiert)

Wie immer: 'delivered lines of code', d.h. ohne Test-Code, hohe Qualität, 'minimalistic design', kein Copy/Paste, sorgfältig getestet und dokumentiert; Durchschnitt eines ganzen Entwicklungsteams, inkl. Requirements und technische Leitung.

Diese Zahlen basieren zur Hauptsache aus der Karriere von Daniel Keller.

#### Zeithorizont - ein Jahr

Programmieren ist immer ein Marathon, kein 100m Sprint

Deshalb: auch wenn Sie einmal in kurzer Zeit viel Code produziert haben, heisst das noch lange nicht, dass Sie dieses Tempo das ganze Jahr über halten können. Und das zählt.

100 - 1000 ausgelieferte Zeilen Code / Person & Monat

**Empfehlung:** wenn Sie keine anderen Erfahrungswerte haben, dann nehmen Sie 400 DLOC pro Monat und Entwickler an (wenn es ein grosses Projekt ist, dann kann das schon zu optimistisch sein).

## Beispiel für Grob-Abschätzung

Beispiel: 80 hoch gekoppelte Packages (s. rechts)

Das Management fragt Sie, was ein totaler Neubau kosten würde. Technologie egal, Programmier-Team(s) würden sich zusammenstellen lassen.

Eine grobe Rechnung

**Heute: 80 PL/SQL Packages mit insgesamt 88'000 LOC**

Annahmen:

- 88'000 LOC neu machen, z.T. weniger LOC weil besser programmiert, z.T. mehr LOC weil neue Funktionen gefragt sind.
- 400 LOC/Personenmonat (PM); 1 PM kostet 15'000
- Überschlagsrechnung:  $88'000 \text{ LOC} / 400 = 220 \text{ PM} = 3.3 \text{ Mio CHF}$
- Ausbuchstabiert: z.B. 20 Pers, 11 Monate = 1 Jahr mit grossem Team
- Versteckte Features gut bekannt, da langjährige MAs im Team

**Aber: Kosten > 1 Mio. und Gruppengrösse > 10 -- was tun?**

Mögliche Vorgehen

Beste 10 Programmierer nehmen

Project Split nach Kundendomäne: 4 Teile lassen sich identifizieren, also rund  $4 * 6$  Monate (mit 3 wärs auch gegangen, dann  $3 * 8$  Monate)

**Kommunikation für das Management:**

- Es kostet insgesamt ca. 3.5 Mio. CHF, total 2 Jahre
- Ein Team von ca. 10 Personen arbeitet daran (an einem Ort)
- Es sind 4 Teilprojekte, die im Rhythmus von 6 Monaten abgeliefert werden: 1. Kernfunktionen plus Immobilien-Versicherung, 2. Fahrzeugversicherungen + Hausrat, 3. Restliche Sachversicherungen, 4. Lebensversicherungen

In einem Jahr fertig werden heisst 20 Entwickler in 2 Teams zu haben. Mindestens.

### Argumente für das Management:

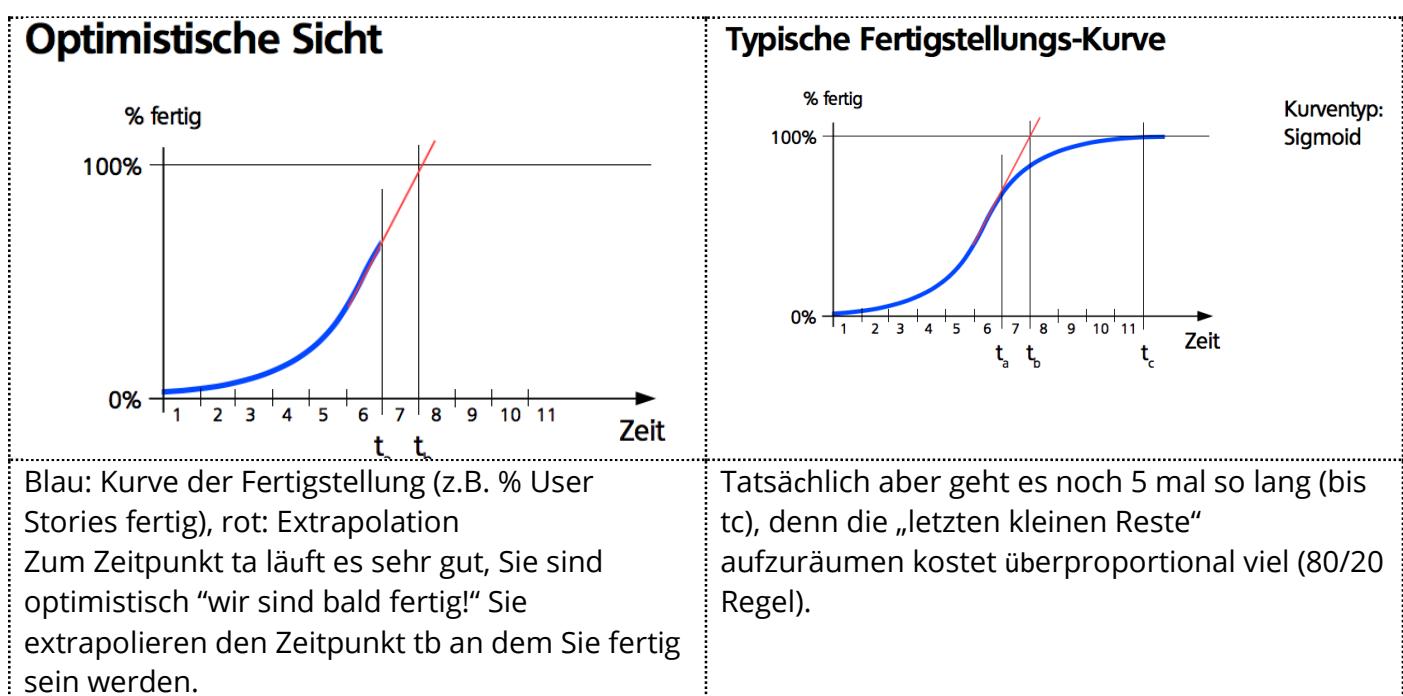
- Doppelt so viele Entwickler sind doppelt so schwer zu rekrutieren. Und dazu ein zweiter fähiger SW Architekt plus ein zweiter PL. Outsourcing kostet mindestens 40% mehr an Aufwand (geografische Aufteilung, sprachliche Hürden), d.h. mit zwei Teams ist es dann auch nicht getan.
- Zwei Teams zu koordinieren kostet +20% Aufwand, also dauert es doch länger als 1 Jahr, oder man braucht 25 Personen, und dann sind wir wieder bei drei Teams, mit noch mehr Koordinationsaufwand.

### Metriken zum Fortschritt

Manager fragen immer: Zu wieviel Prozent sind wir fertig? Wann werden wir ganz fertig?

... denn seit vier Monaten heisst es „wir sind zu 95% fertig“

Es findet sich nie ein linearer Zusammenhang zwischen was immer wir auch messen oder zählen und dem Projekt-Fortschritt – eine Tatsache, die die meisten Manager überfordert.



### Warum Sigmoid?

Gründe für den harzigen Abschluss:

- Schwieriges wird (unbewusst) hinausgeschoben, bzw. nicht genau genug abgeschätzt, Überraschung kommt später. Als einfach Eingeschätztes stellt sich als schwieriger heraus
- Umgesetzte Funktionen funktionieren zwar, verursachen aber an anderem Ort Probleme.
- Kunde sieht SW funktionieren, jetzt kommen Korrekturen und Wünsche. Eingesetzte SW (auch probeweise) verändert die Geschäftsprozesse und damit die Anforderungen.
- Refactoring verbessert den Code ohne dass mehr Funktionalität sichtbar ist: Kunde sieht den Fortschritt nicht.
- Verbesserungen von z.B. Robustheit oder Security kosten viel Arbeit, führen aber nicht zu mehr sichtbarer Funktionalität.

# Software-Metriken

## Ziele

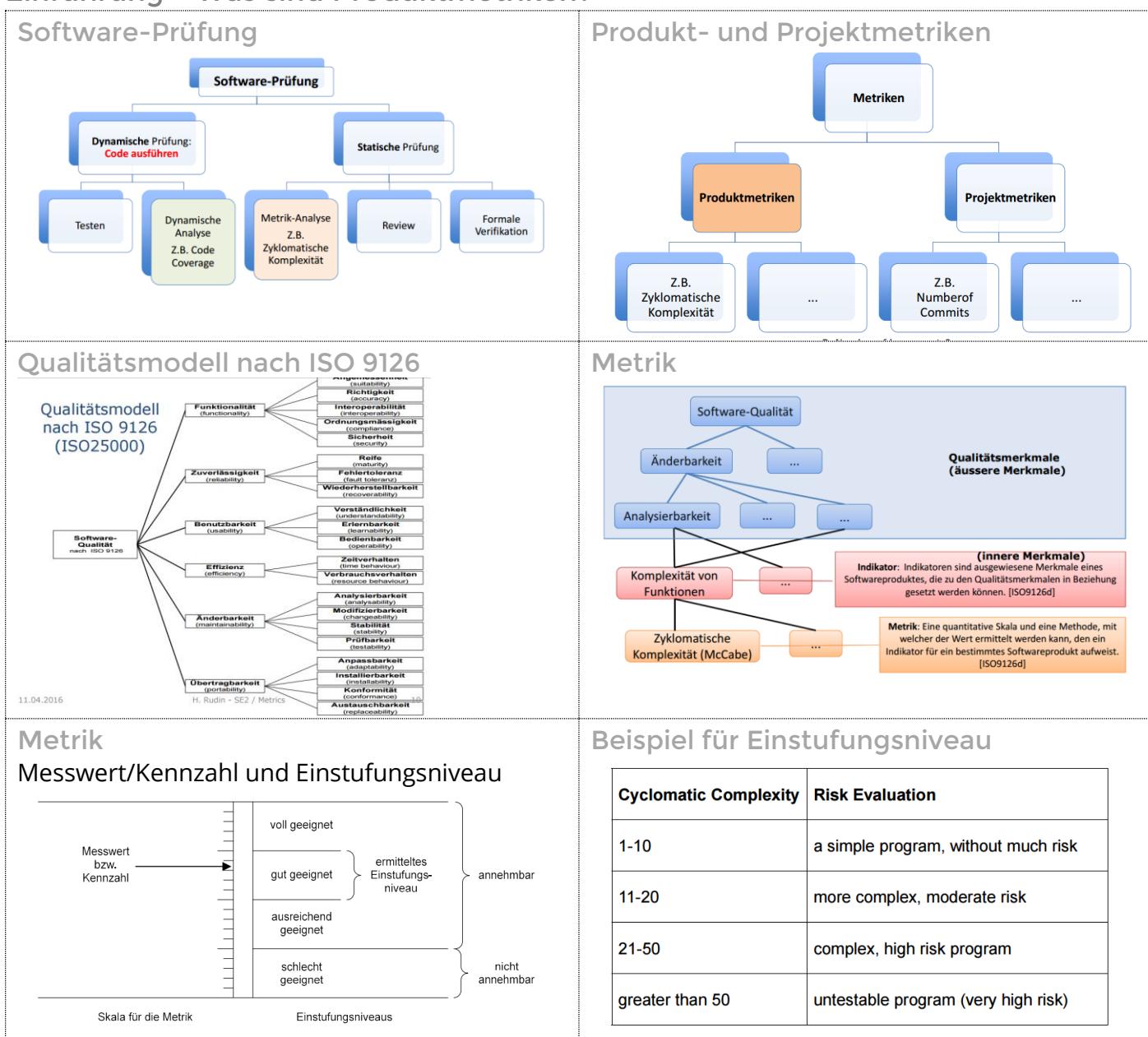
### Erster Tag in einem fremden Projekt

Sie haben vom Management den Auftrag bekommen, ein grösseres Projekt zu analysieren. Man fragt Sie "Wir kommen kaum mehr vom Fleck, wir haben zuviele Merge-Konflikte, es tauchen immer wieder komische Fehler auf, die System-Integration dauert zu lange... Können Sie uns sagen, was hier los ist?" Was können Sie tun, wenn Sie frisch an ein Projekt herankommen und wissen wollen, wie es um dieses Projekt steht? Dieselbe Frage stellt sich in jedem Projekt immer wieder – vielleicht einfach mit weniger Dringlichkeit.

### Warum behandeln wird das Thema?

Metriken sind wichtiges Mittel der Qualitätsicherung. Aber Achtung auch „Wer misst, misst Mist“.

### Einführung – Was sind Produktmetriken?



Softwarequalität bzw. Technical Debt besser einschätzen. Verbesserungspotential in Code, Design und Architektur entdecken. Zudem gibt es noch viele weitere Ziele:

- Projektgrösse abschätzen
- Kosten aktueller und zukünftiger Projekte besser vorherzusehen
- Komplexität besser einschätzen
- Personaleinsatz vorher bestimmen zu können
- Produktivität neuer Technologien evaluieren
- Zukünftige Wartungstätigkeiten prognostizieren

## Codemetriken

### Traditionelle Sourcecode-Metriken

#### Umfangsmetriken

Z.B. Anzahl Codezeilen, Anzahl Unterprogramme, Halstead- Metriken (zählen Anzahl unterschiedliche Operatoren und Operanden, sowie Gesamtzahl von Operatoren und Operanden)

#### Logische Strukturmetriken

Z.B. Zyklomatische Zahl von McCabe

#### Datenstrukturmetriken

Z.B. Anzahl Variablen, ihre Gültigkeit und Dauer

#### Stilmetriken

Z.B. Namenskonventionen

#### Bindungsmetriken (Kohäsion und Kopplung)

Z.B. Anzahl Packages ausserhalb, von denen Klassen im Package abhängen  $\square$  Efferent Coupling

#### Die Zyklomatische Zahl (McCabe Metrik)

Misst Komplexität der logischen Struktur. Basis Kontrollflussgraph G eines Programms. Die

**Zyklomatische Zahl** ist die Anzahl linear unabhängiger Pfade durch ein Programm.

$$V(G) = e - n + 2p$$

- e = Anzahl Kanten (edges)
- n = Anzahl Knoten (nodes)
- p = Anzahl Komponenten (components)

Für einzelne Funktion/Methode:  $V(G) = e - n + 2 \rightarrow V(G) = \text{Anzahl Verzweigungen (while, if, case)} + 1$ .

Beispiel	Bewertung
<p>Kontrollflussgraph Beispiel zaehleZeichen() [Balzert98]</p> <pre> void zaehleZeichen(int &amp;VokalAnzahl, int &amp;Gesamtanzahl) {     char Zchn;     cin &gt;&gt; Zchn;      while ((Zchn &gt;= 'A') &amp;&amp; (Zchn &lt;= 'Z') &amp;&amp; (Gesamtzahl &lt; INT_MAX)) {         Gesamtzahl = Gesamtzahl + 1;          if ((Zchn == 'A')    (Zchn == 'E')    (Zchn == 'I')    (Zchn == 'O')    (Zchn == 'U')) {             VokalAnzahl = VokalAnzahl + 1;         }         cin &gt;&gt; Zchn;     }     return; } </pre> <p><math>V(G) = e - n + 2</math>  <math>= 9 - 8 + 2 = 3</math></p> <p>oder einfacher:    Anzahl if + while + 1</p>	<ul style="list-style-type: none"> <li>• + einfach zu berechnen</li> <li>• + ergibt minimale Anzahl Testfälle für 100% Zweigüberdeckung</li> <li>• - vereinfacht zu stark z.B.       <ul style="list-style-type: none"> <li>◦ berücksichtigt nicht Verschachtelung von Bedingungen</li> <li>◦ berücksichtigt nicht zusammengesetzte Bedingungen</li> <li>◦ einfache switch-Anweisungen führen zu hohem Wert</li> </ul> </li> </ul> <p>Trotzdem Original häufig verwendet: Faustregel: <math>V(G)</math> einer Funktion/Methode <math>&gt; 10 \rightarrow</math> genauer anschauen.</p>

## Metriken für Objektorientierte Software

Verwendet im Eclipse Metrics Plugin (continued).

### Sourcecode-Metriken für objektorientierte Software

#### Umfangsmetriken

Z.B. Anzahl Klassen, Methoden und Attribute, Breite und Höhe der Vererbungshierarchie

#### Logische Strukturmetriken

Z.B. Zyklomatische Zahl von McCabe für Methoden

#### Metriken für Kohäsion und Kopplung

Z.B. „Lack of Cohesion of Methods (LCOM)“

#### Number of Classes (NOC)

Anzahl Klassen im ausgewählten Bereich

#### Number of Children (NSC)

Anzahl direkter direkter Unterklassen Unterklassen einer Klasse; Klasse, die Interface implementiert, zählt als Unterklasse dieses Interfaces

#### Number of Interfaces (NOI)

Anzahl Interfaces im ausgewählten Bereich

#### Depth of Inheritance Tree (DIT)

Distanz der Klasse zur Klasse Object in Vererbungshierarchie

#### Number of Overridden Methods (NORM)

Totale Anzahl der überschriebenen Methoden in ausgewähltem Bereich

#### Number of Methods (NOM)

Anzahl Methoden im ausgewählten Bereich

#### Number of Fields (NOF)

Anzahl Attribute im ausgewählten Bereich

#### Total Lines of Code (TLOC)

Anzahl Codezeilen im ausgewählten Bereich; Zählt nur Zeilen, die nicht leer sind (Zählt Zeilen mir nur { oder }), Zählt keine Kommentarzeilen, oft auch KLOC genannt.  $TLOC = KLOC * 1000$ .

#### Method Lines of Code (MLOC)

Zählt Codezeilen innerhalb Methoden im ausgewählten Bereich. (Zählt Zeilen mir nur { oder }).

#### Specialization Index

Mittelwert von  $NORM * DIT / NOM$ , Metrik auf Klassenebene

#### McCabe Cyclomatic Complexity (CC)

Zählt Anzahl Wege durch Methode, Wird um 1 erhöht für jede Verzweigung des Kontrollflusses (Startet bei 1). Verzweigungen: if, for, while, do, case, catch und ?: sowie && und || in Bedingungen (nicht in Originaldefinition).

#### Weighted Methods per Class (WMC)

Summe McCabe Cyclomatic Complexity für alle Methoden in Klassen

**Lack of Cohesion of Methods (LCOM\*)**

- \* Variante nach [Henderson96]
  - $m$  = Anzahl Methoden
  - $m(A)$  = Anzahl Methoden, die ein Attribut zugreifen
  - Mittelwert ( $m(A)$ ) über alle Attribute
- LCOM\* = ( m - Mittelwert (m(A)) ) / ( m - 1 )**

## Beispiele:

- Jede Methode greift nur ein Attribut zu  
→ LCOM\* = 1 → schlechte Kohäsion
- Jede Methode greift auf jedes Attribut zu  
→ LCOM\* = 0 → maximale Kohäsion

Achtung: Semantik  
wird nicht  
berücksichtigt!

Hier ist unschön, dass Getter/Setter den LCOM verschlechtern.

**Afferent Coupling -  $C_a$** 

Anzahl Klassen ausserhalb eines Packages, die von Klassen innerhalb des Packages abhängen

**Efferent Coupling -  $C_e$** 

Anzahl Klassen innerhalb eines Packages, die von Klassen ausserhalb des Package abhängen

**Instability eines Packages - I**

$I = C_e / (C_a + C_e) - a \rightarrow$  toward a central part -  $e \rightarrow$  away from a central part.

Aus [Martin03] S. 263:

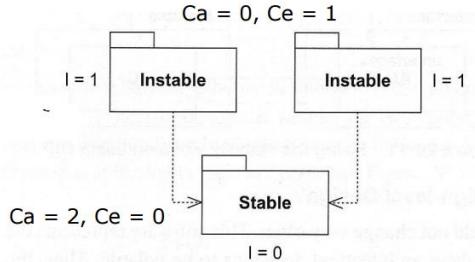


Figure 20-8 Ideal package configuration

**Abstractness - A**

$A$  = Anzahl abstrakter Klassen / Anzahl Klassen eines Packages

**Normalized Distance from Main Sequence -  $D_n$** 

$D_n$  sollte möglichst gering sein bei gutem Package Design.  $D_n = | A + I - 1 |$

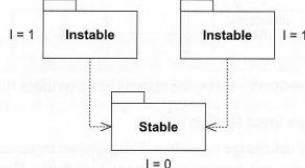


Figure 20-8 Ideal package configuration

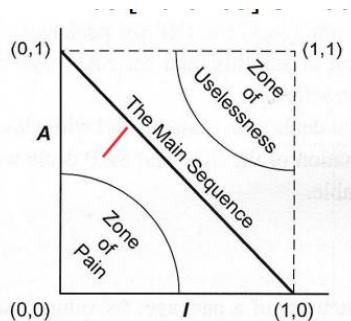


Figure 20-13 Zones of Exclusion

**Tools für Sourcecode-Metriken****Eclipse Metrics Plugin (continued)**

Viele objektorientierte Metriken, Safe Ranges konfigurierbar, Bei Projekteinstellungen aktivierbar, Darstellung in Metrics Views als Baumstruktur, Dependency Graph View.

Berechnet folgende Metriken: McCabe's Cyclomatic Complexity, Efferent Couplings, Feature Envy, Lack of Cohesion in Methods, Lines Of Code in Method, Number Of Fields, Number Of Levels, Number Of Parameters und Number Of Statements, Weighted Methods Per Class.

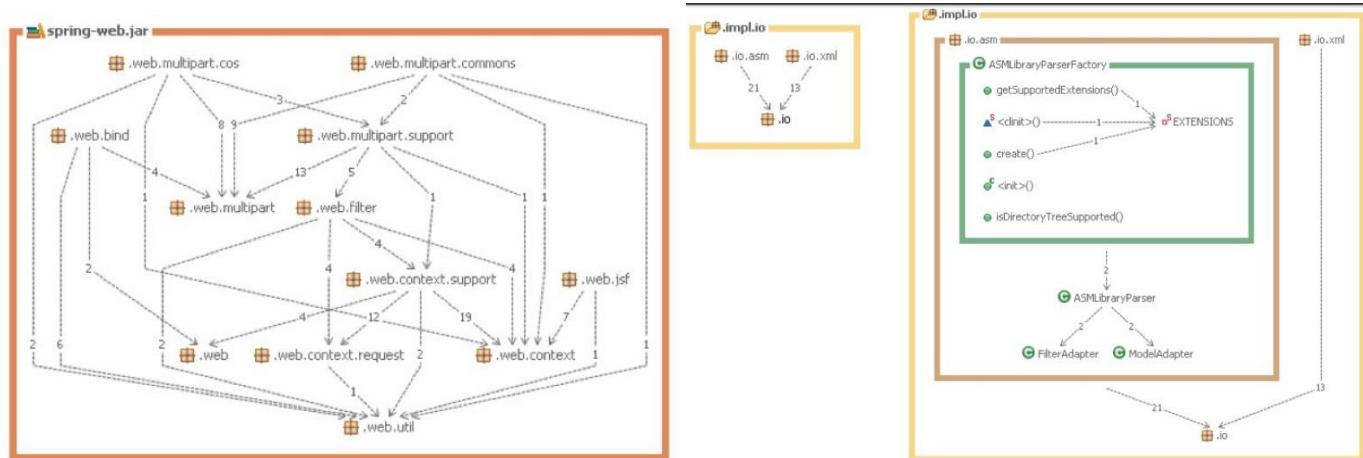
Verletzungen der Schwellwerte als Warnings in Problem-View . Bei Quellcode wird Icon eingeblendet, Mouse-hover bringt Tool-tip mit Details

### Structure 101

kommerzielles Tool zur Struktur- und Komplexitätsanalyse (für Java und .NET). Structure 101 bündelt Metriken und berechnet eine sog. "Excess Complexity" XS. Structure101 berechnet Abhängigkeiten im Code auf verschiedenen Strukturelementen

### STAN – Structure Analysis for Java

Arbeitet auf Bytecode, Filter Patterns (z.B. um Tests auszuschliessen), Detaillierungsgrad wählbar (Von Packages bis hin zu Instanzvariablen). Eclipse Plugin und Standalone Version.



### Checkstyle

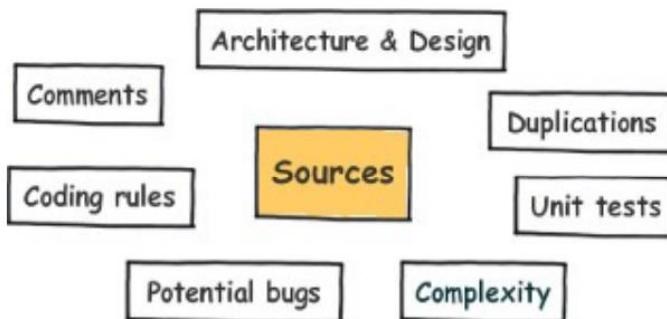
Prüft auf Quellcodeebene verschiedenste Dinge (Programmierkonventionen, Metriken, Duplikaterkennung). Achtung: kann viele Fehlermeldungen erzeugen, aber konfigurierbar für eigenen "Geschmack"

### Findbugs

Code-Analyse-Tool der University of Maryland, Statische Analyse von Java Byte Code, Eclipse Plug-in für interaktive Nutzung, Standalone Nutzung in Build Prozess

### Sonar – umfassendes System zur Überwachung der Qualität mit Metriken ..

- Code Qualität
- Architektur
- Unit Testabdeckung
- Duplicated Code
- Mögliche Bugs
- Komplexität
- Kommentare



**Prinzip:** Java Code wird instrumentiert und zur Laufzeit wird aufgezeichnet, welche Statements ausgeführt werden. Nutzen: Bei automatisierten (aber auch manuellen Tests) kann man feststellen, welche Code Teile ausgeführt und welche ausgelassen wurden.

**Nutzen von Codemetriken****Was bringt es? Qualität!**

- Metriken in Build Prozess und IDE eingebaut helfen beim Entwickeln → Hinweise auf nötiges Refactoring
- Strukturanalyse hilft die Abhängigkeiten zu reduzieren, Zyklen zu vermeiden und Architekturvorgaben (Layering) einzuhalten
- Tools haben sich in Studienarbeiten auch für Studenten bewährt ☐ in SE 2 Projekt ausprobieren, bei Studienarbeit von Anfang an einsetzen!
- Aber, Metriken mit Verstand verwenden, keine absoluten Massstäbe, sondern Faustregeln

# Code Reviews

## Wie funktioniert ein Review?

### Konkretes Vorgehen

1. Stück Code (oder Dokumentation) auswählen
2. Personen auswählen (peers!) und Rollen verteilen
3. Termin finden und dann Personen einladen
4. Review durchführen
  - a. Code walk-through, geführt vom Autor
  - b. Kommentare der Reviewer im Protokoll notieren
  - c. Abschliessend mit Bewertung „Gut“, „OK mit Nacharbeiten“ und „Nicht OK“.
5. Nacharbeiten begleiten und abschliessen. Protokoll ablegen.

### Beispiel

Code-Beispiel		Review Protokoll				
		Datum: 2. Mai 2017, 10h				
ID	Beschreibung	Schweregrad	Datum & Kürzel wenn behoben	Req. Ref.	Bem.	
01	Any file vs. sound file (file types/ext?) in dir	leicht				
02	Leere catch Blöcke	schwer				
03	dir.listFiles == null anders behandeln (?) als list.length == 0	mittel				
04	Bei rand.nextInt +1/-1 besser in Methode verschieben	mittel				

```

private String getRandomSoundfile(String directory) {
    File dir = new File(directory);
    Random rand = new Random();
    int min = 0, max = 0;
    FileFilter onlyFiles = new FileFilter() {
        @Override
        public boolean accept(File pathname) {
            return pathname.isFile();
        }
    };
    File[] list = dir.listFiles(onlyFiles);
    if (list == null || list.length == 0) {
        logger.debug("Did not find any soundfile in directory: " + directory);
        return "";
    } else {
        max = list.length - 1;
        int randomNum = rand.nextInt(max - min + 1) + min;
        return list[randomNum].getAbsolutePath();
    }
}

private void loadConfiguration() {
    BufferedInputStream stream;
    configuration = new Properties();
    try {
        stream = new BufferedInputStream(new FileInputStream("./config/feli.properties"));

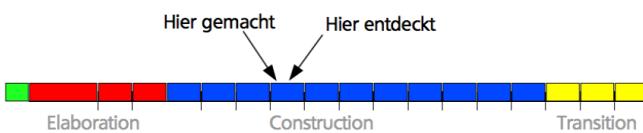
        configuration.load(stream);
        stream.close();
    } catch (FileNotFoundException e) {
    } catch (IOException e) {
    }
}
}

```

### Was kostet es, einen Fehler zu fixen?

Nehmen Sie an, jemand im Team hätte einen Fehler produziert entweder als Programmierfehler (z.B. „Off-by-one“ Fehler), als Fehlinterpretation eines Requirements oder gar als falsch aufgenommenes Requirement.

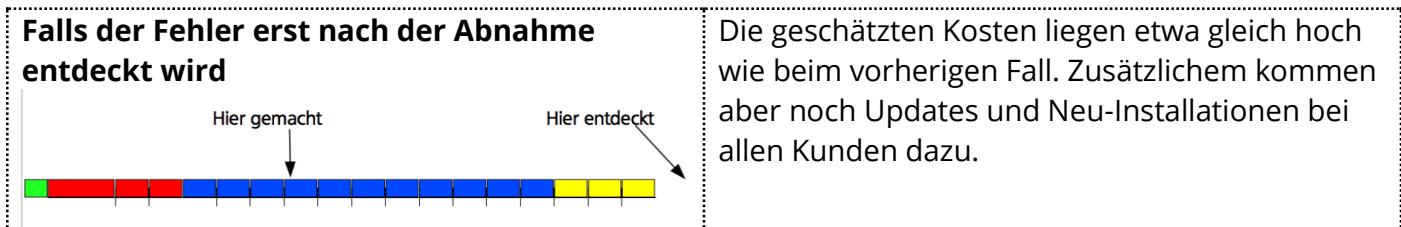
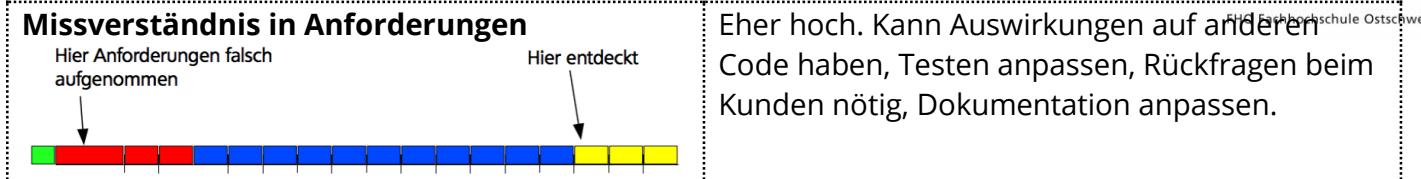
#### Falls der Fehler noch beim Programmieren oder Unit Testen entdeckt wird



Gering, da der Fehler schnell gefunden und schnell gefixt wird. Somit nix passiert.

#### Falls der Fehler erst beim Zusammenstellen eines Major Release entdeckt wird

Eher gering. Der Fehler lässt sich noch relativ einfach beheben. Es muss aber nachgeprüft werden, ob der Fehler nicht noch weitere Auswirkungen hat (Folgefehler).



### Eine wahre Geschichte

Zwei Reviews von neu geschriebenem Code in einer international operierenden Firma – die ersten Code Reviews seit Jahren.

### Aufwand

- Review 1 – Gruppe von 6, während 5 Stunden
- Review 2 – Gruppe von 5, während 3.5 Stunden
- Total 48 Personen-Stunden Aufwand im Review
- + 25 Personen-Stunden für Nacharbeiten

### Resultat

- Die Gruppen finden
  - o 31 schwere Programmierfehler
  - o 105 kleinere Fehler
  - o und 35 kosmetische Punkte
  - o dh. 136 Fehler in 73 Personenstunden gefunden und beseitigt →  $73 / 171 * \text{CHF } 120 =$  Kosten von 51 CHF pro Fehlerbeseitigung

### Kostenvergleich

48 Stunden in Reviews  
+ 25 Std. Fehlerbehebung haben  
320 Stunden (sehr vorsichtig geschätzt) an Fehler-Folgekosten erspart, ganz zu schweigen von möglichen Reputationsschäden.

Dazu der erweiterte Nutzen von Reviews: Know-how Transfer

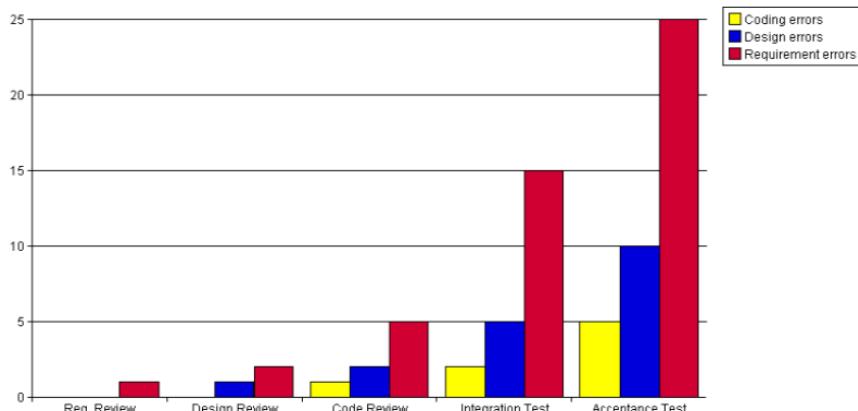
- Bessere Kenntnisse der Programmierer über die Software
- Besserer und einheitlicherer Programmierstil



## Fehler so früh wie möglich entdecken!

Es lohnt sich, gemachte Fehler so früh wie möglich zu entdecken. Je länger ein Fehler unentdeckt im System ist, desto teurer ist die Beseitigung. Es gibt gute Gründe zur Annahme, dass die Fehlerkosten in Abhängigkeit des Entdeckungszeitpunktes exponentiell ansteigen.

### Relative Kosten für die Fehlerbehebung



## Was kostet es, einen Fehler zu fixen, in einer regulierten Umgebung?

Falls sich dasselbe in einer regulierten Umgebung (Medizin/Pharma, Flugzeug-SW, Banken, Steuerung von Atomkraftwerken, kollaborative Roboter, etc.) abspielt, dann wird es noch viel teurer. Es kommen externe Qualitätssicherungs-Personen dazu, welche viel Zeit in ausführliche Tests stecken (auch Regressionstests), dazu die sorgfältige Dokumentation zuhanden z.B. des FDA, damit die Zertifizierung erreicht wird.

Überall, wo Sie einer externen Stelle beweisen müssen, dass keine (wesentlichen) Fehler in der Software stecken und dass die Software genau den Anforderungen entspricht: Bei jedem Fehler muss neu zertifiziert werden.

## Code Analysis Tools

Nützliche Helferlein, die viele Fehlerarten automatisch finden. Es gibt (sprachabhängig) verschiedene Arten von Analyse-Software zur Unterstützung von Code Reviews:

- Code Metriken
- Überprüfung des Programmier-Stils
- Prüfung auf Anomalien und mögliche Fehler
- Auswertung der Testabdeckung (ESLint, Metrics...)

Diese Tools sollen automatisch bei jedem Build laufen, sowohl auf jeder Entwickler-Machine als auch auf dem Build Server – auf jedem Fall vor jedem Review.

## Praktische Regeln für Code Reviews

### 1. Vorbreiten, einladen

- Stück Code (Klasse) auswählen. Ungefähr 500 LOC für 2h Review.
- Teilnehmende einladen, Link zu Code verschicken
- Links zu: Requirements Specification, System Architecture Document, Detailed Design für dieses Stück Code verschicken.
- Alle Teilnehmenden sollen sich vorbreiten: zumindest oberflächlich alles Relevante (Code und Doku) lesen.

## 2. Vorbedingungen checken

- Codingguidelines vorhanden
- Code ist sauber formatiert nach den Guidelines (Java checkstyle)
- Code übersetzt ohne Fehler, ohne Warnungen
- findbugs/ESLint/ReSharper/etc. ohne Warnungen & Fehler
- Relevante Dokumentation (Design/Architektur) ist up-to-date

Definition of Done spezifiziert genau, wann ein Code-Stück fertig ist. Wenn Unit Tests vorhanden, dann die Testabdeckung ermitteln und die Unit Tests auch lesen.

## 3. Code Review durchführen und protokollieren

**Üblicherweise:** Review-Leiter + Autor + Peer(s) + [Protokollführung]

- Code Zeile für Zeile durchgehen
- Bemerkungen der Teilnehmenden protokollieren (s. Tabelle). Das Nummer-Eins Kriterium für Code ist Lesbarkeit/Verständlichkeit.
- Sich konzentrieren auf Fehlerfindung. Lösungsvorschläge können zwar gemacht werden, sind aber nicht das wichtigste.
- Bleiben Sie sachlich, hacken Sie nicht auf einer Person rum.
- Nach zwei, allerspätestens drei Stunden aufhören
- Am Schluss Einigkeit über: „akzeptiert/akzeptiert mit Nacharbeiten/ zurückgewiesen“

### Darauf soll man sich besonders achten

- Verständlichkeit (Warnsignal, wenn nicht allen den Code verstehen)
- Namenswahl (Methoden, Variablen, Packages, das kann kein Tool)
- „Code Smells“ (die üblichen Verdächtigen)
- Übereinstimmung mit Architektur-Ideen und Diagrammen

Unabdingbar ist es, eine erfahrene Person in der Gruppe zu haben, die alle möglichen Code Smells kennt und im realen Code auch erkennt. Vielleicht haben Sie eine Checkliste mit Punkten, auf die man achten soll.

### Format des Protokolls

Teilnehmende und deren Rollen, Zeit, Ort, Identifikation des Code-Stücks und der relevanten Unterlagen (evtl. nur Verweis auf Einladung), Tabelle mit den offenen Punkten, Aufgabenverteilung, Verdikt (akzeptiert/akzeptiert mit Nacharbeiten/zurückgewiesen).

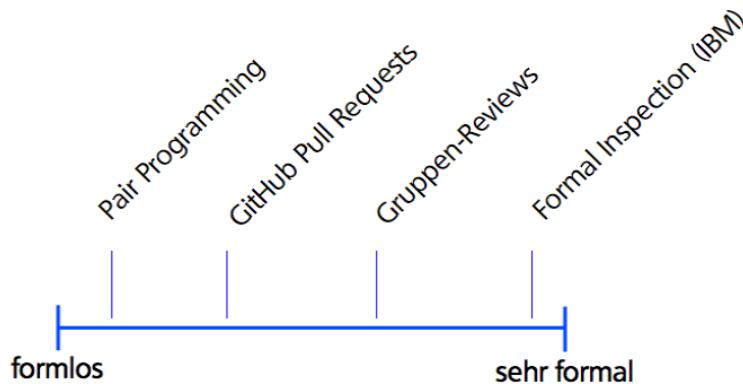
ID	Beschreibung	Schweregrad	Datum & Kürzel wenn behoben	Req. Ref.	Bem.

#### 4. Nacharbeiten

- Das Protokoll ist fertig geschrieben und abgelegt
- Für jeden gefundenen Programmier-Fehler:
  - o mindestens einen Unit Test schreiben, der den Fehler provoziert
  - o Fehler beheben (und/oder refactoring) und dokumentieren
  - o Unit Test muss jetzt OK sein
- Erneute Integrations-/Systemtests
- Bug tracking nachführen
- Nachkontrolle durch Review-Leiter: sind alle Fehler behoben?

#### 5. Eventuelle Nachkontrollen

##### Grad der Formalität der Reviews



Der Grad der Formalität korreliert nicht unbedingt mit dem Nutzen. Wie immer: es kommt darauf an...

##### GitHub Pull Request als eine Art Review

Ein kurzer Gegen-Check eines erfahrenen Programmierers mittels pull Request und Diff auf GitHub ist sehr wertvoll, ersetzt aber nicht immer einen Review in der Gruppe. Kleine Änderungen kann man so abhandeln, die Intensität und der Lerneffekt sind aber nicht so hoch. Wichtige, zentrale Dateien sollte man immer in der Gruppe reviewen.

##### Requirements Reviews

###### Ziel der Requirements

- „Ping Pong mit Kunde“ – Versucht, von den Kunden wolkigen Wünschen zu möglichst formaler Requirements Spec. zu kommen
- Wissens-Transfer vom Kunden zu den Entwicklern (und bis zu einem gewissen Grad auch zurück)
- Erster wichtiger Schritt für verlässlichere Kostenschätzungen

###### Ziele des Requirements Review

- Die Requirements sollten den Ansprüchen von QA und Testing genügen (complete, consistent, traceable, testable)
- Know-How Transfer „How to write requirements“

## Beispiel-Szenario für Requirements-Review

Nehmen Sie einen Use Case und das Datenmodell. Gehen Sie schrittweise durch den Use Case und schauen Sie, ob alle Daten, die eingegeben werden auch gespeichert werden können. Schauen Sie, ob alle Daten, die es zur Anzeige braucht, auch aus dem Speicher geholt werden können („Speicher“ = DB oder nur Objekt-Baum im RAM).

### Beispiel für weitere Fragen

- Stimmen die Aktivitäts-Diagramme mit den Use Cases überein und passen sie zu den Datenstrukturen?
- Sind die Abläufe in Aktivitäts-/Sequenzdiagrammen und die Zustände von Objekten in Zustandsdiagrammen korrekt reflektiert?
- Was muss man permanent speichern, was nur transient während z.B. einer Session?

### Proben über's Kreuz (Cross Checks)

**Use Cases mit Domain-/Datenmodell:** können alle notwendigen Daten gespeichert bzw. zur Anzeige abgerufen werden?

**Use Cases mit GUI-Entwürfen:** ist für jede Benutzerinteraktion ein Bildschirm-/Seitenentwurf vorhanden?

**Use Case Diagramm mit User Access Rights:** sollten überein stimmen

**GUI-Entwürfe mit Use Cases:** gibt es für jedes interaktive Element (Knopf, Feld zum Ausfüllen, Menupunkt..) eine beschreibende Passage in den Use Cases?

**GUI-Entwürfe mit Zustandsdiagrammen:** werden alle Entitäts-Zustände in allen Bildschirm-/Seitenentwürfen sichtbar und gleich abgebildet (z.B. Farben, Text, Icon)

**GUI-Entwürfe mit User Access Rights:** ist für jeden Bildschirm-/Seitenentwurf festgelegt, wer darauf kommen darf?

**Zustandsdiagramme mit Datenmodell:** Werden alle Zustände im Datenmodell abgebildet/gespeichert? (damit z.B. Zustände wiederhergestellt werden können)

**User Access Rights mit Datenmodell:** ist festgelegt, wer welche Daten sehen darf?

**Aktivitätsdiagramme mit den korrespondierenden Use Cases:** 100% Übereinstimmung.

**Domain-/Datenmodell mit Use Cases:** was ist mit den in den Use Cases NICHT gebrauchten Daten? (tauchen sie bei den nicht-funktionalen Anf. auf?)

**Nicht-funktionale Anforderungen und Datenmodell:** gibt es Datenfelder, die helfen, die nicht-funktionalen Anforderungen zu erfüllen, zB: e-Shop mit Verkaufsstatistiken und Profitabilität von Artikeln, Lagerlogistik mit Angaben zur Erreichbarkeit (Geschwindigkeit) der einzelnen Lagerfächer, Zeitstempel für Zugriffs-Statistiken, damit optimiert werden kann.

## Architektur Reviews

Bei Architektur-Reviews achten Sie besonders auf folgende Punkte:

- Nicht-funktionale Anforderungen bzw. Qualitäts-Merkmale, denn die wirken sich zu über 80% direkt auf die Architektur aus (z.B. Security und Performance, aber auch Wartbarkeit/Erweiterbarkeit).
- Kern-Charakteristiken des Systems müssen sichtbar sein (siehe Architektur-Vorlesung)
- Architektur-Dokumentation muss mit tatsächlicher Implementation übereinstimmen.
- Szenarien durchspielen, z.B. für Erweiterbarkeit

## Beispiel-Fragen Architektur-Review

- Sind bei allen Schnittstellen nicht nur die statischen Aspekte (file/live call, Parameter, Typen) sondern auch die dynamischen Aspekte (wann, wie oft, in welcher Reihenfolge, Rollen und Rechte) definiert?
- Angenommen, wir würden das geplante Intranet zu einer Art Social Network erweitern, wie würde die Architektur dann aussehen? (s. ATAM)
- Wie kann man die Performance skalieren, wenn die Besucherzahlen 5 x höher als geplant wären?
- Sind die Charakteristiken aller Prozesse definiert (z.B. single-/ multithreaded, stateful/stateless etc.)?
- Wie wird Fehler X behandelt? (X = z.B. Netzwerk-Störung, DB- Inkonsistenz, fehlende Daten). Wie sieht das Logging und das Exception Handling aus? Konsistent überall gleich? Fehlerfindung leicht?

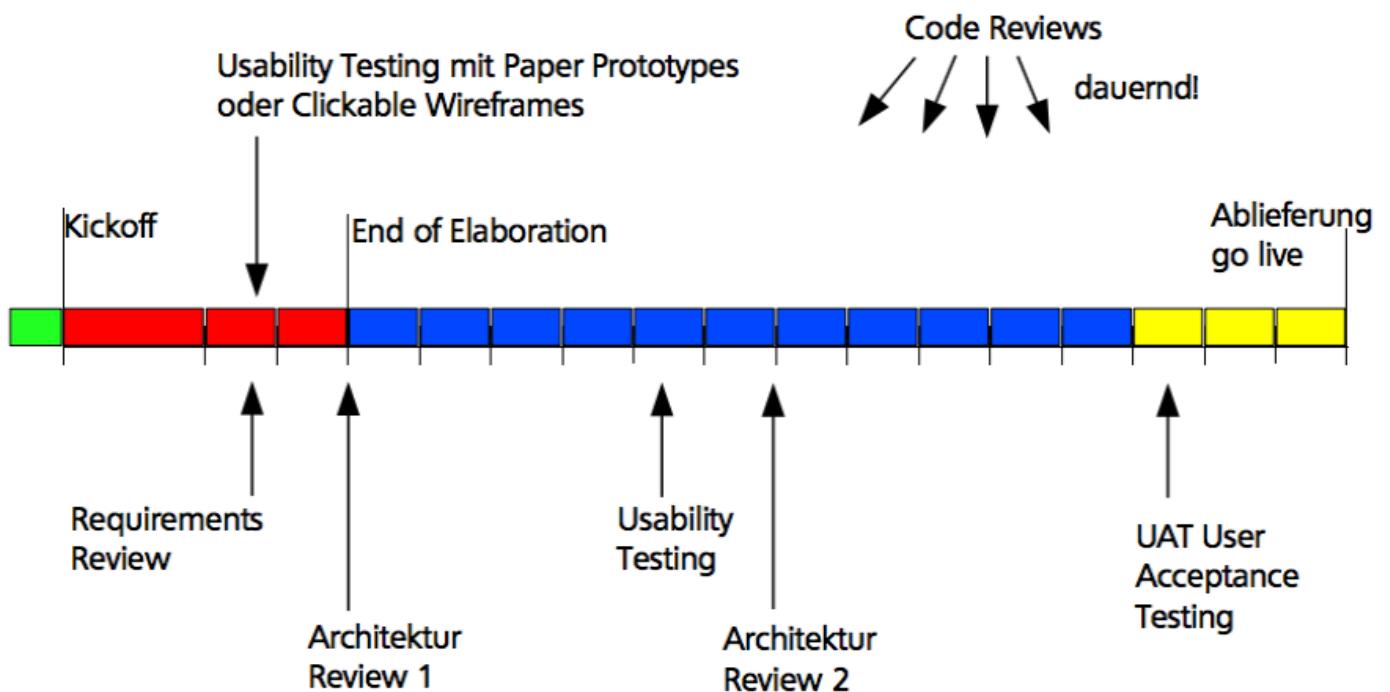
## Kosten/Nutzen von Code-Reviews

Je öfter man Review durchführt, desto weniger Fehler findet man. Nach wenigen Monaten dauern die Reviews nicht mehr lange.

**Kosten** Über den Daumen gepeilt etwa 5 % der Codier-Kosten, Dafür ca. 10x weniger Kosten, als wenn der Fehler erst später entdeckt wird.

## Sogar einen Dreifach-Nutzen

1. Fehler werden früher gefunden → deutliche Kosteneinsparung
2. Programmierer werden besser → Code-Qualität wird besser
3. Know-How (Besonders über System-Architektur) verbreitet sich



# Performance-Messungen

'Premature Optimizations' (voreilige Optimierungen) sind kontraproduktiv. Es gibt Programmierer, die wollen zu früh optimieren. Immer zuerst durch Messungen feststellen "Haben wir ein Problem?" Erst dann optimieren. Wenn wir also ein neues Feature in die Software einbauen gilt folgendes

1. Zuerst in einer Minimal-Version (sunny case) zum Laufen bringen, sodass es brauchbar bzw. demonstrierbar ist. Feedback zur Funktionalität einholen.
2. Danach alles „richtig machen“ d.h. robust, fehlerfrei, auch für alle Grenzfälle, internationalisiert, schön ausgerichtet, Balloon Help, ...
3. Erst jetzt sich um allfällige Performance-Probleme kümmern.

## Beispiel aus der Literatur

„Vergleichen Sie doch einmal die Laufzeiten zwischen sequenzieller und paralleler Abarbeitung auf Ihrem System. Beispielsweise ist auf meinem MacBook Pro mit einem Core i7 2,2 GHz die sequenzielle Variante der Filterung für 1 Million int Werte mit ca. 37 ms Laufzeit fast doppelt so schnell wie die parallele mit ungefähr 70 ms! Das Ganze ändert sich erst, wenn man die Datenmenge stark erhöht: Bei 1 Milliarde Einträgen messe ich eine Laufzeit von etwa 4,5 Sekunden für die sequenzielle und etwa 1 Sekunde für die parallele Filterung. Demnach kann die Parallelverarbeitung ihre Vorteile erst bei sehr großen Datenmengen ausspielen. Das sollten Sie immer im Hinterkopf behalten, bevor Sie unüberlegt durch einen Aufruf von parallel() die Performance steigern wollen.“

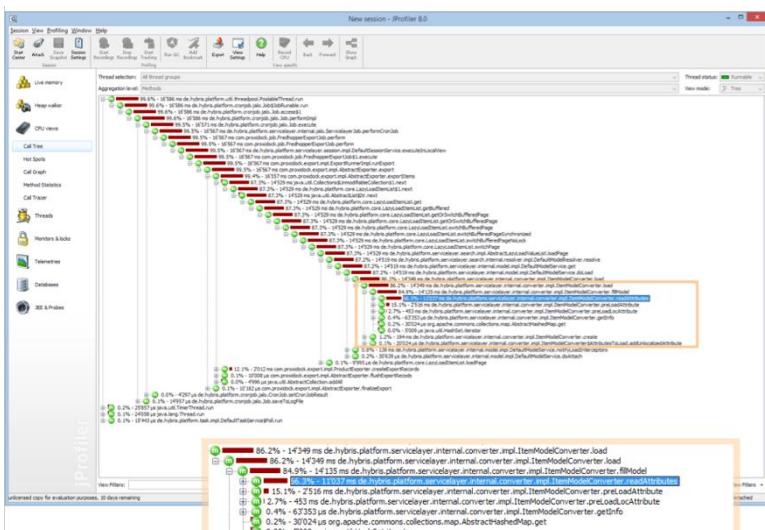
## Performance-Profiling (white box)

Antwort auf „Wo wird die Zeit verbraten?“

### Java Profiler in NetBeans

Die Analyse mit dem Profiler in NetBeans zeigte, dass eine einzige Routine für mehr als 70% des CPU-Verbrauchs verantwortlich war. Nach einem kleinen Umbau war die Applikation gefühlt dreimal so schnell.

### Wie funktioniert ein Profiler?



Methode hochgezählt.

- Am Schluss kann so die in jeder Methode verbrauchte Zeit angezeigt werden.

Damit ein Profiler funktioniert, muss der Quellcode zur Verfügung stehen (ja, es geht auch ohne, aber dann hat man nur statistische Daten, nichts Genaues, siehe JVM Monitoring unten)

- Jedem Methoden-Aufruf wird ein Timer-Feld angehängt.
- Es wird mitsamt dieser Instrumentierung nochmals compiliert. Code laufen lassen.
- Bei jedem Methoden-Aufruf wird das Timer-Feld mit der Ausführungszeit der

Profiler können oft so eingestellt werden, dass sie die Zeitnahme sogar auf jede einzelne Zeile machen.

## Last-Messungen (Black-Box)

**Antwort auf:** „Wann ghet der Server in die Knie?“

**Nützlich für:** Webserver, Datenbanken, Exposed Services (REST, SOAP)

**Tools:** gatling.io, JMeter

**Funktionsweise:** HTTP Requests absetzen, parameterisierbar, massiv parallel

## Wiederholte Performance-Messungen

**Antwort auf:** „Sind die typischen Antwortzeiten immer noch etwa gleich?“

**Nützlich für:** Webserver/shops, Exposed Services (REST, SOAP), Regressions-Tests

**Tools:** Selenium (evtl. JMeter/gatling, aber die können kein .js)

**Funktion:** „Headless Browser“ (kann JavaScript), programmierbar

Es gibt auch spezialisierte Firmen, die Cloud Services für Lasttests anbieten. (z.B. loadstorm.com).

## Probleme mit Testdaten

Nehmen Sie an, Sie wollen einen Webshop wiederholt auf Performance testen. Was nehmen Sie als Testdaten? Vermutlich simulieren Sie immer an einem Sonntagabend eine Menge (z.B. 300) gleichzeitige User, die einerseits via Inhalts-Kategorien navigieren, andererseits via Produkt-Suche die Seiten abrufen. Wenn eine Produkt-Detailseite gefunden ist, dann bestellt ein Prozentsatz (30%) der simulierten User etwas (je drei Artikel) im Shop -- via Spezial-Accounts, damit das System keine echten Bestellungen auslöst. Weil es möglichst echt wirken soll, haben wir ein zusätzliches Problem: Verteilung der Daten (NICHT random: populäre Artikel viel häufiger) und Caching im Shop (z.B. eben die populären Artikel) - und das Ganze in virtualisierten Umgebungen, die ja auch Memory Pages zwischenspeichern.

## Monitoring mit Dashboards

**Antwort auf:** „Wie geht es unserer Infrastruktur?“ „Wie geht es unserem Produktiv-System?“

**Tools:** Grafana/ Graphite



## Application Telemetry

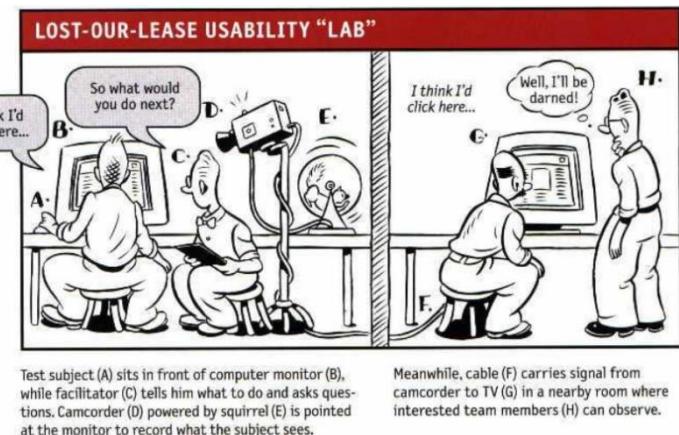
In einer Rakete stecken Tausende von Sensoren, die NICHT für die unmittelbare Flugsteuerung gebraucht werden. Zweck: viel Informationen zur Hand haben, falls etwas einmal schiefgeht, v.a. zur Lokalisierung der Fehler. Ähnliches kann/soll man auch in produktiver Software machen: Log output; Output für Dashboards.

# Usability Testing

## Einführung

Man nehme:

- Die neue, zu testende Software
- Mehrere nicht vorbelastete Nutzer (User)
- Ein Usability Lab
- Eine(n) Moderator(in)
- Mehrere Szenarien zum Testen (Use Cases)



## Teste die Software, nicht die Anwender!

- Kurze Einführung geben ("helfen Sie uns", "denken Sie laut mit", "Sie können nichts falsch machen", „Die Software wird getestet, nicht Sie“)
- Den Probanden Aufgaben (eines der Szenarien) stellen "Versuchen Sie, in diesem Shop ein Buch zu bestellen", oder "Finden Sie heraus, was beim Abo 'Basic S' ein Anruf vom Festnetz auf die Mobilnummer 076 555 4433 kostet" (offene vs. konkrete Fragestellungen).
- Den Probanden nicht helfen, nicht führen, nur beobachten
- Eventuell Fragen stellen ("Wo, denken Sie, würden Sie als nächstes Klicken?", "Wonach suchen Sie gerade?", "Ist etwas unklar?")
- Notizen machen
- Für's Mitmachen danken

## Durchführung

1 Computer-Arbeitsplatz

3 - 30 freiwillige Versuchspersonen, mit/ohne Vorkenntnisse

40 - 60 Minuten Zeit (pro Sitzung)

1 Moderator (Facilitator, Test-Assistenz)

Ein paar sinnvolle Benutzungs-Szenarien (Use Cases)

1 Videokamera und 1 Mikrofon, Übertragung in den Nebenraum

Stichwort-Protokoll durch Moderator/Assistent



## Was kommt dabei heraus?

Wertvolle Erkenntnisse, welches oft auch Überraschungen sind. Wo stolpern die User, wo bleiben Sie hängen, Zu häufige Maus/Tastatur-Wechsel, Unerkannte Features, Auf wieviele Arten kann man die Software „falsch“ bedienen, Unklare Begriffe/Beschriftungen... etc. Lassen Sie sich überraschen.

## Wann und Wie?

Etwa in der Mitte des Projektes, denn: Vorher steht vermutlich noch keine brauchbare Software zur Verfügung. Nachher kann es zu spät sein, um Erkenntnisse als Änderungen noch einfließen zu lassen („Dafür haben wir keine Zeit mehr“).

Mindestens 3 geeignete Testpersonen. Dabei „normale“ Anwender rekrutieren. Die Variation ist wichtig (z.B. un/erfahrene Computernutzer, alt/jung). Programmierer sind die denkbar ungeeignetsten „normalen User“.

## Varianten

- Mit hohem Aufwand (30 Testpersonen, teures Lab mir mehreren Plätzen)
- Test „light“ (3 Testpersonen mit Laptop und Kamera)
- Paper Prototyping (Vorteil: sehr früh machbar)
- Clickable Wireframes (früh möglich, erfordert aber Abstraktionsvermögen vom Anwender)
  - o Zeigen was Funktional möglich ist, Mockups zeigen das Design.
- A/B Testing (auch oft live gemacht)
  - o Man präsentiert (z.B. in einem Online Shop) den Besuchern zufällig eine von zwei Varianten und beobachtet das Verhalten (z.B. Abschlussquote und Absprungverhalten). Welche Variante – A oder B – wirkt besser?
- UX Reviews mit Experten: Einhaltung der Richtlinien, Konsistenz

## SW Architektur – Teil 3

Eine Freundin aus Finnland kommt das erste Mal zu Besuch nach Rapperswil. Sie möchten ihr die Stadt zeigen. Sie planen zwei Stunden Zeit dafür ein. Dazu machen Sie sich einen Plan. Einen solchen Plan kann man sich auch für die Architektur-Präsentation machen.

### Definition „Was ist SW Architektur“

Architektur ist die Summe der Design-Entscheide, die von grosser Tragweite sind, und die länger leben (Entscheide, die hoffentlich den ganzen Lebenszyklus des System unverändert bleiben). Beispiele:

- Interfaces zwischen Subsystemen
- Interfaces zwischen unabhängigen Prozessen
- Datenstrukturen „Daten leben ewig“; grösste Auswirkung auf Code, falls geändert werden muss
- Grosse Charakteristiken wie „Datenbank-orientierter Design“...
- Cross-cutting concerns: security, logging, ....

### “Architektur“ nach IEEE

IEEE STD 1472 definition: “An architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment and the principles guiding its design and evolution.

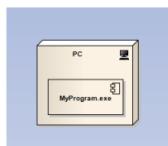
### Architektur-Ueberlegungen

System-übergreifende Überlegungen (Cross-cutting concerns): <ul style="list-style-type: none"> <li>• Fehlerbehandlung, Exception handling</li> <li>• Logging (spannend wenn über mehrere Maschinen hinweg)</li> <li>• Transaction handling</li> <li>• Internationalisierung</li> <li>• Backup, Löschen von Überflüssigem, z.B. Log-Entries in DB</li> <li>• Deployment</li> </ul>	Qualität: <ul style="list-style-type: none"> <li>• Performance-Überlegungen (Mengen, Zeit, Geschwindigkeit), eventuelle Real Time and Space constraints</li> <li>• Security/Safety-Überlegungen</li> <li>• andere Qualitäts-Überlegungen (Erweiterbarkeit, Plattform-Portabilität, Verfügbarkeit..)</li> </ul>
---	--

### Verschiedene Architektur-Typen

Einige Muster (3 x Deployment, 2 x innerer Aufbau), die Sie immer wieder sehen werden, mit typischen Anwendungen, dazu Vor- und Nachteile der einzelnen Entscheidungen.

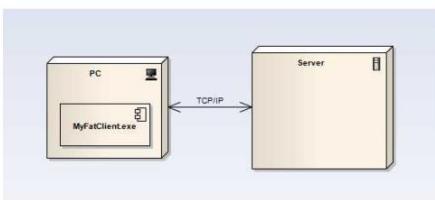
## 1) Single User Desktop Program



Beispiele: Adobe Photoshop; Microsoft Word 2007; Astah UML Tool, Print Layout, Musik-Aufnahme & Edit (Digital Audio Workstation)

- + UX, Performance, Sicherheit, Verfügbarkeit, geringe System-Komplexität
- Teamfähigkeit, Deployment/Cloud/Work Everywhere, Portierbarkeit

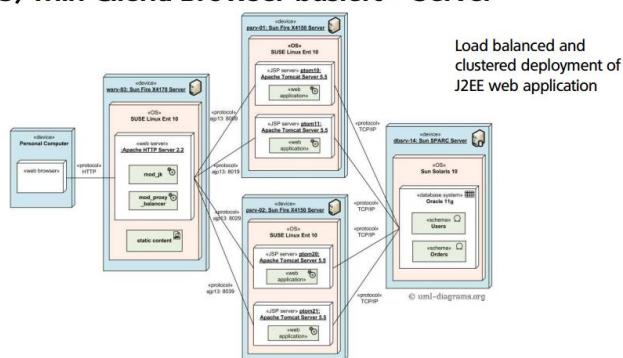
## 2) Fat Client + Server



Bsp.: Programmier-IDEs (Eclipse, IntelliJ etc), CAD, Adobe InDesign, UML: Sparx Enterprise Architect, Firmen-Buchhaltung, [MS Office]

- + User Experience (UX), Performance („Schwuppdizität“), Sicherheit
- Deployment, Portabilität, Verfügbarkeit

## 3) Thin Client: Browser-basiert + Server



- + Deployment/Cloud/Work Everywhere, Last-Skalierbarkeit, Portabilität
- Performance, UX, Browser-Kompatibilität, Sicherheit, Verfügbarkeit, Testbarkeit

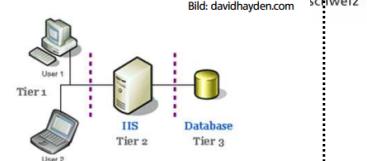
## Produzent → Konsument

Tellerwerfer, Druckerei - Falzmaschine, Gaströcknungsanlage, ERP (z.B. SAP) - CRM (z.B. SalesForce)

- Entweder ist der Konsument immer schneller als der Produzent (oder es ist egal, wenn der Konsument 'mal was fallen lässt', s. VoIP/UDP mit 'Stottern')
- Oder dann wird über Warteschlangen als Puffer kommuniziert. Das braucht Synchronisation bzw. gegenseitigen Ausschluss.



## Multi-Tier Architectures



Generell:

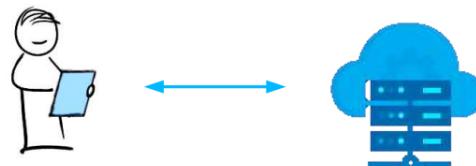
### Haupt-Vorteile

- Skalierbarkeit, Performance: Jeder Tier kann entsprechend seinen Fähigkeiten/Zuständigkeiten optimiert werden, z.B. kann die HW passend gewählt und aufgerüstet werden, aber auch SW-Optimierungen, bis hin zur Wahl der Betriebssysteme.
- Verteiltes & koordiniertes Arbeiten

### Nachteile

- Performance (asynchron, nicht-deterministische Antwortzeiten), Verfügbarkeit, Sicherheit

## 2a) Mobile = „Fat Client“ + Server



Mobile Clients: Android/iOS Apps sind die neuen Fat Clients (wenn native programmiert, und nicht HTML/Browser-basiert)

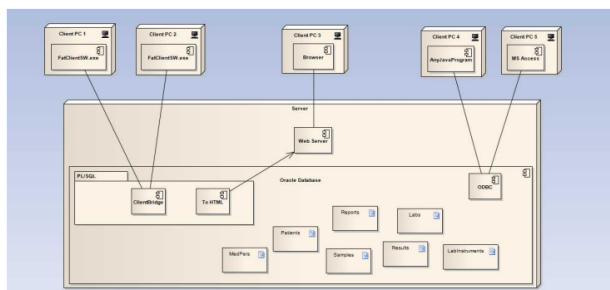
### Warum native:

- bessere UX
- mehr Sicherheit, da nicht-Standard-Verbindungen möglich

### Hauptnachteile von ‚native‘:

- Deployment
- teuer, weil nicht portabel (Android, iOS, Win)

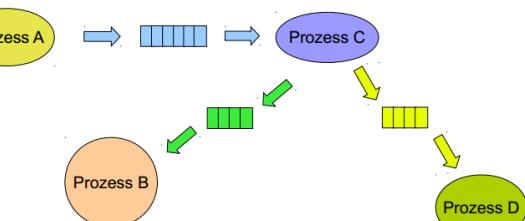
## 4) DB-zentrierte Architektur



+ Einfacher Aufbau, Erweiterbarkeit im Kleinen, Last-Skalierbarkeit

- Erweiterbarkeit (größer = unübersichtlicher, Gefahr von Wildwuchs gerade weil es so einfach ist), Hammer-Nagel-Syndrom, Wartbarkeit, dem DB-Lieferanten ausgeliefert (Vendor Lock-in)

## 5) Message-basierte Systeme

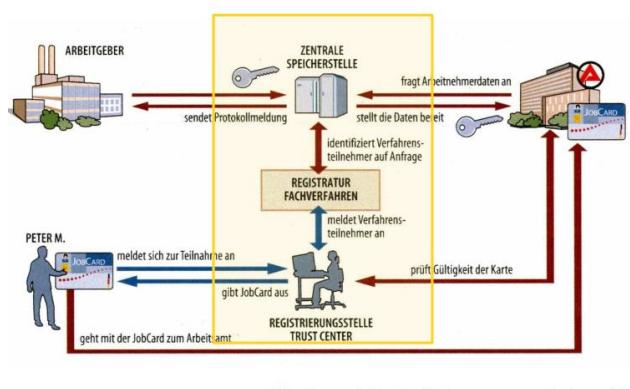


Verteilte Prozesse, die asynchron über Message-Queues kommunizieren.

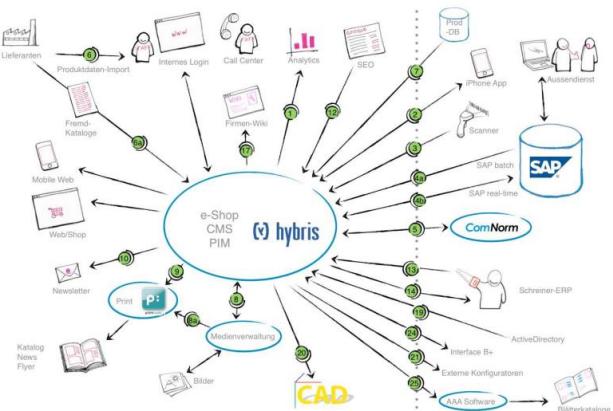
- + Skalierbarkeit, Performance, Separation of Concerns top umsetzbar
- Schwieriger Entwurf, schwierigere Fehlersuche (Logging hilft)

## Diagramme für die Architektur-Beschreibung

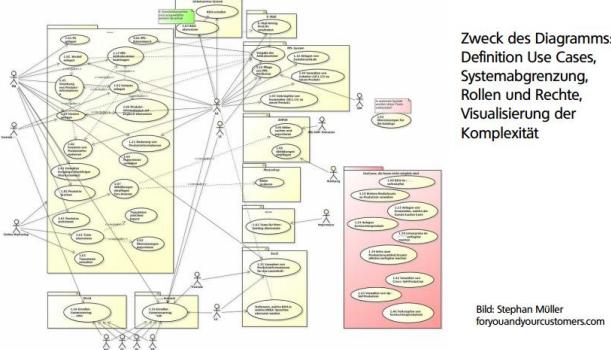
### Context Diagramm informell



### Schnittstellen zu Umsystemen



### Use Case Diagramm - eine Nummer grösser



Firma mit 35'000 Beschäftigten, Aufgabe: Ersatz des Produkt-Informationssystems

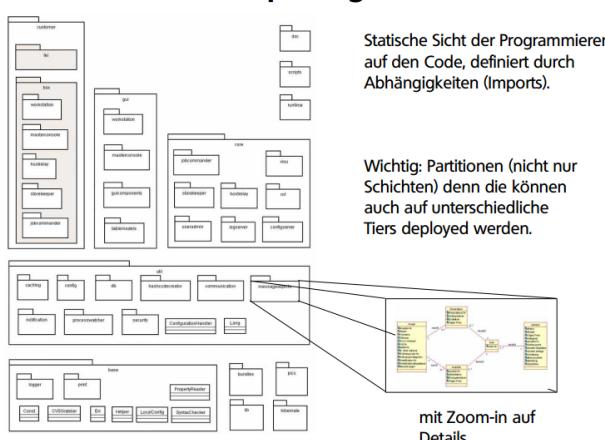
### Anforderungen und Umsysteme

So far: no architecture yet.

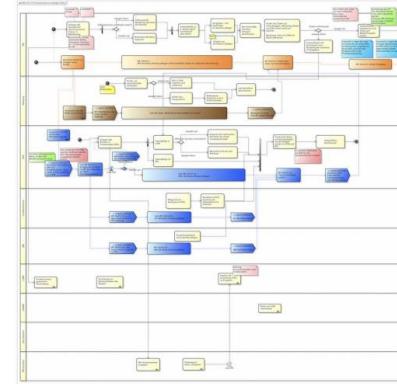
Bis jetzt haben wir noch nicht viel entworfen, eigentlich nur dokumentiert. Das ist nicht wirklich schwierig. Die Schwierigkeit bis jetzt war, zu entscheiden, was in welcher Tiefe dokumentiert werden soll.

Im Folgenden beschreiben wir den inneren Aufbau von Systemen. Nach einer Übersicht zoomen wir Schritt für Schritt in das System hinein.

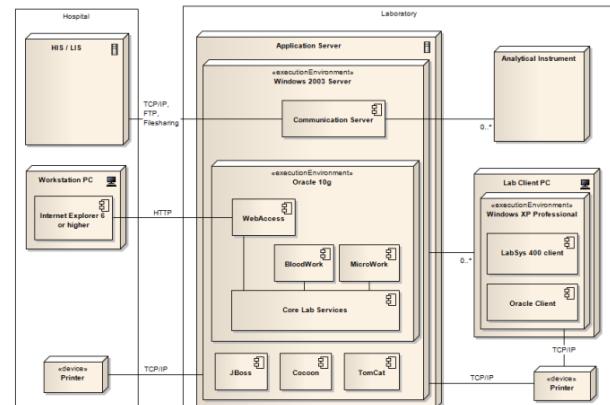
### Schichten-Architektur (packages)



### Prozessbeschreibung: Aktivitätsdiagramm



### Deployment-Diagramm: wo läuft was



### Analyse zu den Requirements:

- Beschreibung der umliegenden Systeme und Akteure, Schnittstellen
- Datenmodellierung (statisch, "was merken wir uns?")
- Prozessbeschreibungen (dynamisch, "was läuft ab?")
- Randbedingungen (IT Landschaft, politisch, juristisch)
- Nutzer, Rollen & Rechte
- Vorgaben zu Security, Performance, Usability, Portability...
- UX-Vorgaben (oft Browser; besonders wichtig bei iOS/Android)

## Inhalte des Software Architecture Document (SAD)

Sie beschreiben, WIE Sie das System gebaut haben und WARUM Sie es so gebaut haben (in den Requirements/Analyse steht, WAS der Kunde wollte).

Grob zwei Teile:

- A Umfeld und Randbedingungen
- B Technische Struktur

In beiden Teilen beschreiben Sie, wie es ist. Dabei ist durchaus Kreativität gefragt, aber: erfinden Sie nichts dazu. Und: Die Bestandteile der Beschreibungen sind von Projekt zu Projekt unterschiedlich

### A - Dokumentation Umfeld

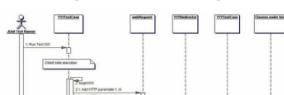
Context Diagramm (Übersichtsbild, die umliegenden Systeme und Aktoren), IT-Landschaft mit Schnittstellen zu anderen Systemen, Beschreibung der Datenhoheit in anderen Systemen (wo werden die Daten „geboren“...), Beschreibung der Aktoren Rollen und Rechte, Business Process Modelling, Randbedingungen (juristisch, historisch).

### B - Dokumentation Technische Struktur

- Deployment diagram (was läuft wo), oft verschiedene DeploymentSzenarien, (meist sehr high-level)
- Schichten-Architektur, UML top level package/class diagram (was auf eine A3-Seite passt), zeigt, wie die Klassen/Pakete strukturiert sind.
- Datenmodell: UML class diagram oder E-R Diagramm. Bei den Beschreibungen v.a. auch auf die Assoziationen achten.
- Zustandsdiagramme/Lebenszyklen der wichtigsten Entitäten
- Prozesse, Threads mit ihren Charakteristiken; Message Queues
- UX Skizzen, Personas & Szenarien, Screen shots, Erklärungen

#### B2) Zoom: Dynamik

- Prozesse, Threads mit ihren Charakteristiken; Prozess-Interaktion über Message Queues (z.B. JMS)
- UX Architektur (welche Screens folgen auf welche)
- Ablauf-Szenarien, z.B. Click - call - message - call - message, etc. (Sequenz-Diagramme)
- Data flows, Control flows (Aktivitäts-Diagramme)



#### B1) Zoom: Technische Details

- Dann 'zoom in' in einzelne Teile/Subsysteme; je Subsystem:
- UML Klassendiagramm als Übersicht: nicht zu detailliert, oder dann 1:1 aus Code generiert (aber: "Welchen Sinn hat eine Landkarte im Maßstab 1:1 ?" Zitat Umberto Eco)
  - UML Klassendiagramm als Detailbeschreibung v. etwas Komplizierterem
  - UML Zustandsdiagramme
  - UML Sequenzdiagramme
  - parallele Prozesse (UML?)

Immer mit Kommentaren, v.a. WARUM so und nicht anders.

## Erfolgskriterien für SW-Architektur

- Einfach, verständlich - und damit auch gut dokumentierbar und kommunizierbar
- Stabil, d.h. vorausschauend, langlebig
- Skalierbar, z.B. durch verschiedene Deployment-Varianten
- Gut aufgeteilt, und somit gut parallel entwickelbar
- Gut testbar, z.B. durch gut sichtbare Schnittstellen
- Erweiterbar (mit zusätzlicher Funktionalität)
- Adäquat (kein Over-Engineering) für: Performace, Security, Stability...

# Scrum zum Zweiten

## Was ich an Scrum toll finde.

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>- Iteratives Vorgehen: Sprints (2 oder 3 Wochen) mit inkrementeller Ablieferung und Feedback</li> <li>- User Stories mit Akzeptanz-Kriterien (gute Arbeitspakete)</li> <li>- Product Backlog, Sprint Backlog</li> <li>- Sprint planning</li> <li>- Sprint reviews, sprint retrospectives</li> <li>- Rollen/Verantwortung des Product Owners und des Scrum Masters</li> </ul> | <ul style="list-style-type: none"> <li>- Daily standup (sofern richtig gemacht)</li> <li>- Das Konzept der Team Speed („17 Story Points pro Woche“)</li> <li>- Eigenverantwortliche, polyvalente Teams</li> <li>- Entwickler schätzen - Kunde priorisiert</li> <li>- Definition of Done (mit Einschluss der Tester)</li> </ul> |
|---|--|

## Voraussetzungen für Scrum

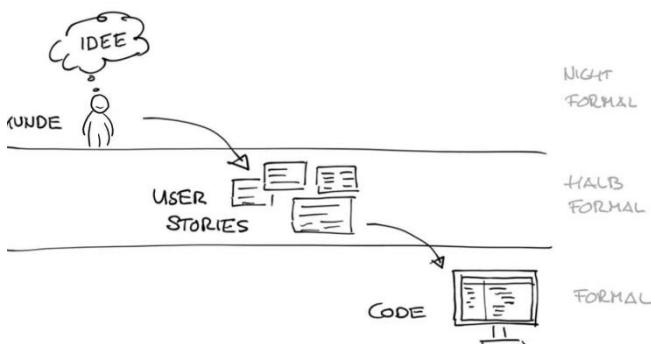
Zwingende Voraussetzungen für ideales Scrum: Teamgrösse max. 10, kurzes Projekt max. 9 Monate, Kunde weiss noch nicht so genau was er will (kein klar definierter Scope, eher experimentelles Projekt), Ganzes Team an einem Ort, Team deckt alle Fähigkeiten ab, Häufige und mündliche Kommunikation, Kunde im Team (ständig verfügbar), Kunde will und unterstützt agiles Vorgehen.

## Was mir in Scrum fehlt

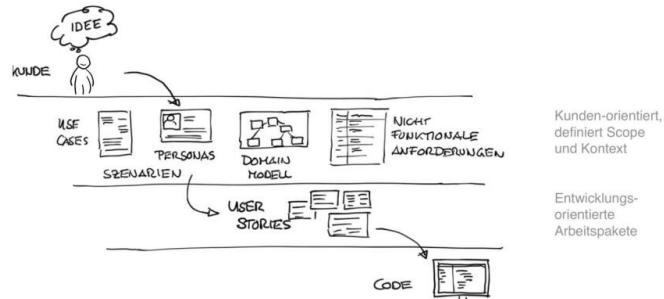
Wenn nicht strikt alle Voraussetzungen für ideales Scrum gegeben sind (oft: wenn wir ein verteiltes Team oder ein nicht-triviales Projekt haben), dann bieten sich folgende Ergänzungen an:

- Checkpoint End of Elaboration (wie im Unified Process)
- Zweite, höhere Abstraktionsstufe zu User Stories im Backlog
- Zusätzliche Rolle des Projektleiters, für all das, was der Product Owner und das Team nicht abdecken
- ⇒ Don't go overboard with Agile !!!!

Im klassischen Scrum gibt es zwischen den wolkigen Wünschen des Kunden und dem gnadenlos formalen Code nur eine Abstraktionsebene: den Backlog mit den User Stories.



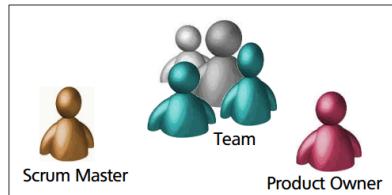
Zum allgemeinen Verständnis und zur Kommunikation mit dem Kunden braucht es eine zusätzliche höhere Abstraktionsebene als nur den Backlog (vgl. Epics, Story Mapping).



## Product Owner vs. Projektleiter

- Der Product Owner (PO) macht keine aktive Risiko-Beobachtung
- PO macht normalerweise keine Stakeholderanalyse
- PO hält den Scope nicht im Auge, evtl. wäre das sogar ein Interessenskonflikt
- PO kann Qualität nicht einschätzen, wird ihn so auch kaum interessieren (im Endresultat schon, aber wer kennt einen PO, der einen Code Review machen kann?)
- Wer macht die Planung der Datenmigration?
- Evaluation Datenqualität (das könnte der PO, aber nicht die Konsequenzen abschätzen)
- Wer macht die Koordination mit externer Agentur (Grafiker, externe Programmierer, UX Designer, Crowd Testers...)
- Wer trifft Umsetzungsentscheide bei Unstimmigkeiten?
- Wer verhandelt mit dem SAP Team?

- Wer verhandelt mit Kunden-IT wenn die nicht vorwärts machen?
- Kümmert sich der PO ums Testen? ... sorgfältiges Testen?
- Wer macht/kontrolliert Stundenaufschreibung und Rechnungen?
- Wer macht Zeit-Auswertung und Soll/Ist Vergleiche?

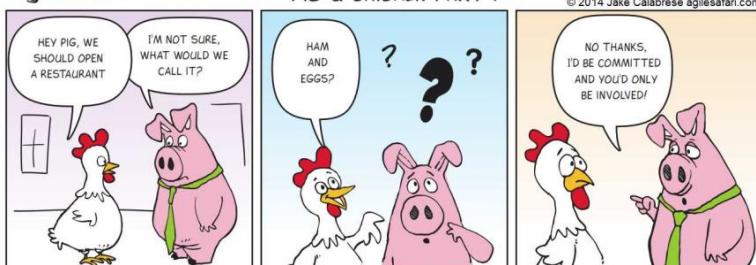


Wer macht's?  
Die Projektleitung



Einige Alternative zu einem Extra-PL: das Team dazu zu ermächtigen (mit Kursen und Überzeugungsarbeit), alle diese Tätigkeiten zu übernehmen.

### Agile Safari

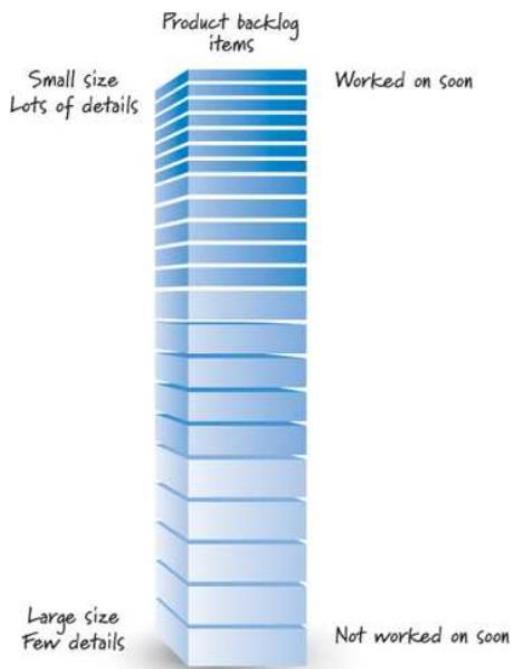


**Chicken:** only involved (d.h. nur beteiligt), z.B. Endkunde, Management, GrafikSpezialist und andere Stakeholder. Dürfen am Daily Standup teilnehmen, aber nichts sagen.

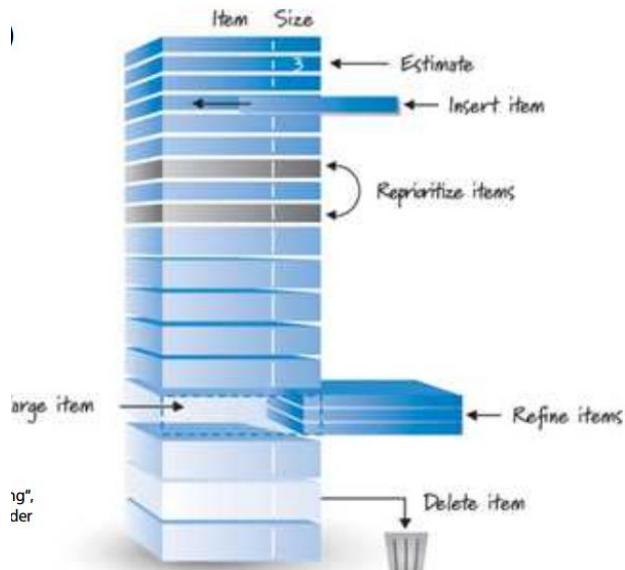
**Pigs:** committed (d.h. verbindlich eingebunden), das Programmier-Team,

Product Owner und Scrum Master (+Projektleiter, falls vorhanden). Müssen am Daily Standup teilnehmen. Dürfen/sollen am Daily Standup reden.

### Gesunder Backlog



### Backlog Refinement



Viel Arbeit für PO, Team, PL: Schätzen, Aufteilen, Löschen und Priorisieren.

### Gantt Charts

Traditionelle Projektplanung à la Microsoft Project: Tasks, Abhängigkeiten, Laufzeiten, critical path, schier endlose Vorausplanung. In Scrum sehen die Charts anders aus. MS Project Fans werden enttäuscht sein. Hier sind die Tickets genau auf die Releases/Iterationen (alles gleich grosse Balken) ausgerichtet. Das ist gut so bei Scrum.

## Impediments

In Scrum wird meist - analog zum Bug Tracking - ein Impediment Tracking geführt.

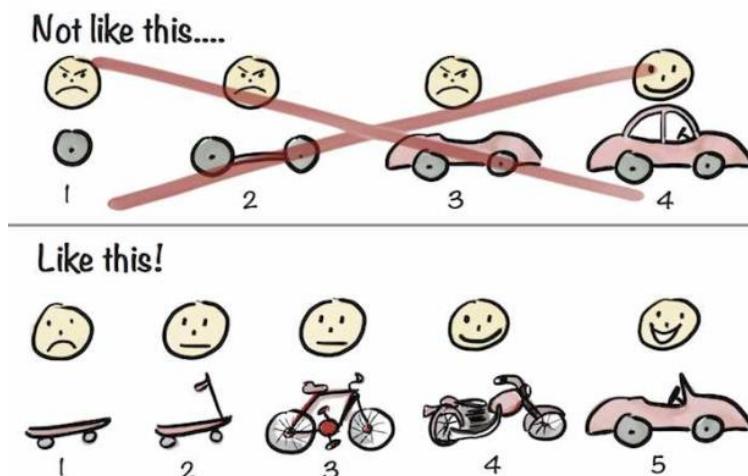
**Impediment** (dt. Hindernis): Ein Hemmnis, eine Blockade, die nicht innerhalb des Scrum Teams beseitigt werden kann. Gründe und Auslöser sind ausserhalb des Teams.

Beispiel: Das Scrum Team arbeitet an einem E-Commerce System. Dabei gibt es Schnittstellen zu SAP. Mit den Echtzeit-Abfragen ist SAP überfordert, die Antworten müssen um mindestens den Faktor 10 schneller werden. Das liegt aber voll in der Verantwortung des (im Moment überlasteten) SAP Teams.

Wer räumt so eine Schwierigkeit aus dem Weg?

Hier kann ein PL oder der Scrum Master eingreifen und dem SAP Team die Wichtigkeit des Anliegens erklären (und evtl. etwas Druck machen). außer die Kunden sind ebenfalls Entwickler – der Hauptgrund, warum Open Source für Entwickler-Tools so gut funktioniert (IDEs, Emacs, git, Linux, JUnit, findbugs, xamarin, redmine, ...) - und sonst nicht sooo gut funktioniert (z.B. Open Source ERP?).

## Value for the customer



Bei jedem Sprint soll der Gesamtwert für den Kunden steigen. Diese Einschätzung gelingt nur in sehr engem Kontakt mit dem Kunden. Weder Team noch PL sollten sich anmassen, den Wert für den Kunden zu kennen. \*).

Also: Immer etwas nutzbares, nützliches abliefern, auch schon in den ganz frühen Sprints

## MVP – Minimum Viable Product

**Kriterien:** Nützlich, einsetzbar, schnell entwickelt – minimal. **Hauptziel:** Feedback von den Nutzern, Erfahrungen sammeln (auch Entw.).

**Minimum Viable Product:** ich kanns in meinem Alltag, bei meiner täglichen Arbeit einsetzen - und werde damit Erfahrungen sammeln, den Entwicklern feedback geben, ... d.h. nutzbare Funktionen, schon ziemlich robust, ziemlich sicher

## Was zuerst umsetzen, was später?

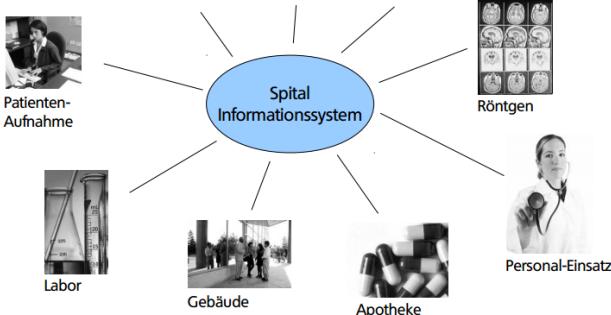
Negativ-Beispiel: Projekt Bücherbörse, da wollte man zuerst ,registration, login, user CRUD' umsetzen (Plan für Constr. 1)

Was haben Sie dann: eine Login App für die Sie sich registrieren können, und sonst sieht man nichts

Besser: Liste der angebotenen Bücher, Liste der Anbieter, kann ich auch ein Buch suchen? (später), einfache Bewertung der Anbieter (später), kann ich auch eine Anzeige aufgeben „ich bräuchte ,DB Systems‘ von Navathe“? (später).

⇒ Leitlinie «Value for the customer».

### Extrem-Beispiel: Minimum Viable Product?



Es wird das existierende Computer-System in einem Spital ersetzt:

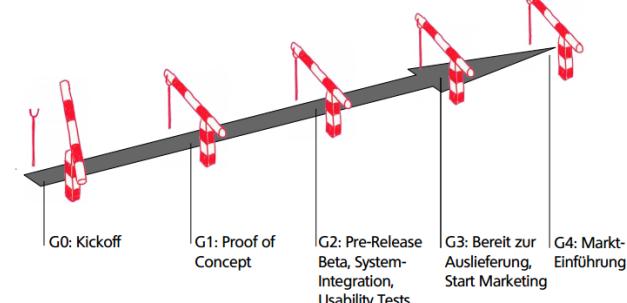
- Wie oft kann ein neuer Release ausgeliefert und getestet werden? *Vermutlich nur ein einziges Mal.*
- Parallel an zwei Systemen (alt/neu) arbeiten? *Niemals.*

### Zuerst umsetzen: Kern-Funktionalität

Grüne Wiese (da gab es bisher noch nichts; viel experimentieren, hoch agil) vs. me too (wir wollen auch ein Stück von diesem Kuchen; Anforderungen sind grossteils bekannt und fix; USP?)  
whatsapp, tinder, uber, snapchat  
Das funktioniert leider aber nur bei neuen Produkten.  
Sobald man eine bestehende Software ersetzen soll, tauchen Schwierigkeiten auf:  
MVP = bestehende SW plus neue Features ?! viel zu gross!  
Value for their Customer? Abgrenzung zum bestehenden System?  
Unbekannte, verborgene Features im bestehenden System?  
Wie oft kann ich die SW den Nutzern zum Testen in die Hand drücken?  
Eine der Grundlagen der agilen Entwicklung fällt somit weg !!  
Mind. 80% der SW Entwicklungen sind nicht auf der grünen Wiese.

## Management will Wasserfall?

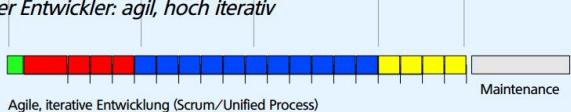
Sie handeln als Entwicklungsteam agil und iterativ, trotzdem will das Management/der Kunde einen fixen Endtermin und ein Kostendach. Wenn Termin und Kosten fix sind, dann muss die Funktionalität (Scope) flexibel sein. Dieses Argument sollte auch den Kunden überzeugen. Zusammen mit dem Vorgehen „erst Basisfunktionalität, danach die weiteren Features“ und der 80/20 Regel kann das gut klappen. Behalten Sie Ihre agile Vorgehensweise mit eigenen Phasen und Meilensteinen (z.B. End of Elaboration). Oft genügt es, wenn Sie sich in den Terminen anpassen, nicht in der Methodik.



Bei jedem Gate wird entschieden, ob das Geld für die nächste Phase gesprochen wird.  
Vor allem bei G1 ist es noch relativ günstig, aus dem Projekt auszusteigen, wenn es kaum Chancen hat.

Sicht des Managements: linear, Gates				
Kickoff	G1: Proof of Concept	G2: Pre-Release Beta, System-Integration, Usability Tests	G3: Bereit zur Auslieferung, Start Marketing	G4: Markt-Einführung

Sicht der Entwickler: agil, hoch iterativ				
				
Agile, iterative Entwicklung (Scrum/Unified Process)				
Maintenance				

# Software Projekt-Management, Teil 2

## Technische Schuld (technical debt)

Was passiert, wenn man auf Kredit einen neuen Fernseher kauft? Barpreis: CHF 1000.- Ratenpreis: 12 x 96.- = 1152.-. **Vorteil** des Kreditkaufs: schnelle Befriedigung des Bedürfnisses. **Nachteile**: der Preis ist höher, der Schmerz ist nur aufgeschoben. Normalerweise fehlt einem das Geld in der Zukunft genauso wie jetzt. Die verführerische Illusion der schnellen Bedürfnisbefriedigung. Klar gibt es manchmal gute Gründe, etwas auf Kredit zu kaufen.

### Copy/Paste als technische Schuld

Angenommen, Sie implementieren ein neues Feature, indem Sie existierenden Code kopieren, kurz anpassen, kurz testen, und dann belassen Sie es so.

Wir haben jetzt duplizierten Code.

Das ist wie ein Kredit: Sie kommen schnell zu einer funktionierenden Lösung, aber das eigentlich notwendige Refactoring wird nicht gemacht oder auf später verschoben.

Später ist so oder so der Preis (Refactoring) zu zahlen, in der Zwischenzeit zahlen wir aber bereits Zinsen: aufgeblähter Code, Irritationen („welches ist jetzt die richtige Lösung?“) und eventuell eingeschleppte/ausgelöste Fehler.

Quick and dirty: Long after quick is gone, dirty remains.

### Technische Schuld als Zahl

SonarQube weist eine Zahl aus, die nach ihren Berechnungen\*) die technische Schuld darstellt, in Arbeitstagen. Interessant, anschaulich, aber nicht vollständig, da nur auf Code basierend.

Woraus besteht eigentlich „technische Schuld“? und wie misst man das? ➔ siehe nächstes Kapitel.

### Vier Arten von „technical debt“

	Code Smells und Architektur-Smells (Duplicated Code, Large Class, hohe Kopplung, Conditional Complexity, etc.) d.h. nicht gemachte Refactorings
<b>JUnit ? Usability ?</b>	Fehlende Tests (Unit, Integration, System, Usability, ...)
	Q-Mängel: Performance (mangelnde Skalierbarkeit), Security (nicht gestopfte Löcher, alte Lib/OS-Versionen), Safety (schlechtes Fehler-Verhalten), etc.
<i>„Keine Zeit!“</i>	Prozesse werden nicht mehr beachtet (Keine Code Reviews mehr, Schätzung von Arbeitspaketen nicht mehr verifiziert, Requirements nicht mehr von PO abgenommen, etc.)

## Der blinde Projektleiter

### Ausgangslage (hypothetisch)

Stellen Sie sich vor, Sie seien verantwortlich für den Bau eines Geschäftshauses mit Wohnungen. Sie sind Projektleiter mit der Gesamt-Verantwortung für das Budget (Plan: 20 Mio.) und die Einhaltung des Zeitplans (Bauzeit geplant 1 Jahr).

#### Baustellen-Verbot

... und stellen Sie sich vor, Sie hätten ein Baustellen-Verbot.

Ja, richtig gehört: Sie dürfen nicht auf die Baustelle gehen.

Nie. (es gibt für Sie auch keine Fotos, keine Videos, kein 3D-Modell . . .)

Was nun?

### Andere Ausgangslage (real)

Ein Software-Projekt von nicht-trivialer Grösse. Ein Entwicklungsteam von acht Personen, alles Informatiker. Ein Projektleiter, gleichzeitig Product Owner. Ein Kunde. Entwicklungsmethodik Scrum. Der Projektleiter hat PL-Erfahrung, kann aber nicht programmieren.

#### Der blinde PL

Ein Projektleiter der nicht programmieren kann, kann nicht auf die Baustelle. Er ist blind. Er muss sich jeden Montag in den Sitzungen erzählen lassen, wie es auf der Baustelle aussieht.

#### Was ist die Software-Baustelle?

Wie sieht man z.B., wieviele Leute zur Zeit grad auf der Programmier Baustelle sind? und ob sie arbeiten - oder nur Red Bull trinken und Facebook checken? und ob wir im Plan sind?

#### Die Baustelle der Programmierer

Was man auf einer Software-Baustelle (nicht  ) sehen kann:

-  Programm-Code
-  Code-Versionier-System (git, SVN) "wer hat wann was am Code gemacht", damit sieht man z.B. wer die eifrigsten Committer sind und an welchen Teilen am häufigsten programmiert wird.
-  Konfigurations/XML/Skript-Dateien
-  Build Server Output (Build fails, Testabdeckung, metrics tools)
-  Backlog o.ä. (JIRA, Redmine)
-  Issue/Bug Tracking
-  Test-Vorlagen (Unit Tests, Integrations- & Systemtests), Test-Protokolle

#### Dashboards

Der nicht-programmierende PL kann viele der vorliegenden Informationen nicht interpretieren

Einige Informationen kann man mit Dashboards sichtbar machen. Das ist dann aber immer noch Blindflug, einfach ein Blindflug mit ein paar Instrumenten - besser als nichts

#### Erster Offizier an Deck

Ein PL, der nicht programmieren kann, braucht einen "Ersten Offizier", dem er voll vertrauen kann. Eigenschaften dieses ersten Offiziers:

- Muss in den benutzten Sprachen programmieren können.
- Muss die eingesetzten Tools beherrschen (Versionskontrollsystem, Integration Server, Libraries/Frameworks).
- Muss sowohl mit Programmierern (Fach-Terminologie) als auch mit Managern (bildhaft) gut kommunizieren können - und von beiden akzeptiert sein.
- Muss während der ganzen Projektdauer anwesend sein.

## Das Unsichtbare sichtbar machen

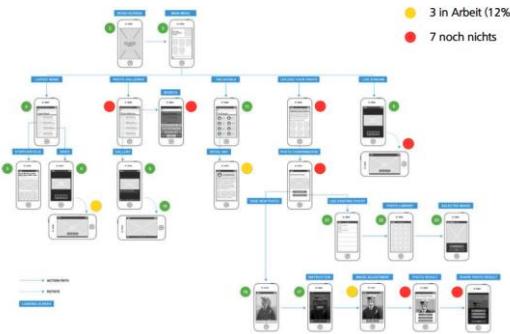
Sichtbar ist eigentlich nur das Endprodukt, bzw. die Prototypen davor. Was kann man sonst sichtbar machen? Drei Perspektiven:

- Für den Kunden und das Management: Burn-Down Chart, Story Map mit Farben, Trends (Bug Reports, build failures, Metriken, ...)
- Für die Entwickler (alles wie f. Kunde, plus:), Backlog, Metriken, Testabdeckung, SonarQube
- Für die IT Infrastruktur: Dashboards mit Ressourcen-Verbrauch (Disk, CPU...), Log-Auswertungen visualisiert (s. auch die Vorlesung zu Performance)

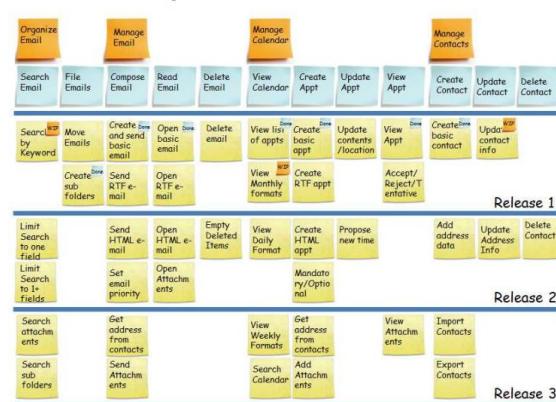
### Trends sind besser als absolute Werte

Schwierig zu interpretierende Zahl „Diese Woche haben wir 23 open bugs“. Ist das gut oder schlecht? Ist das normal? Eine Grafik mit Trend ist einfacher zu verstehen. Die Trends sich auch für Manager interpretierbar (Anzahl offene Bugs, Fertige User Stories, Anzahl Build Breaks pro Sprint, Sprint Speed).

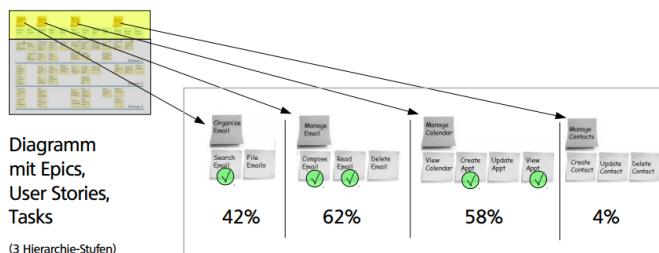
#### Darstellung des Fortschritts anhand von Mobile Screens



#### Hierarchie: Epics - Stories - Tasks



#### Fortschritt sichtbar machen, z.B. so:



Präsentation für das Management:

Grün abgehakt: fertige User Stories (drüber: Epics)

Die Prozentzahlen basieren auf der Vollständigkeit der darunterliegenden Arbeitspakete (Tasks, darunter, nicht gezeigt)

#### Fortschritt nach Ziel-Erfüllung bei den Rollen

Beispiel e-Commerce und Online Shop:

Zielerfüllung \*)

- Produktmanager und Admin Datenpflege
- Einzelkunde kann einkaufen
- Marketing Aktionen und Print
- Übersetzer Texte de → fr/it
- Firmenkunden Sammel-Konto
- Aussendienstler Tablets
- Drittfirmen-PM Datenimport

- (Null)
- (Basis)
- ✓ (Silber)
- (Gold)

\*) Zielerfüllung: Null, Basis, Silber, Gold

Programming

**Individuell besser Programmieren**

- Clean Code, (Unit) Tests, Metrics, Code Smells, Refactoring
- Design Patterns, Modellierung mit UML, OOA, Abstraktion



Near-Programming

**Als Team erfolgreich Software entwickeln**

- Zusammenarbeit und Kommunikation: CVS, Scrum, Continuous Integr., Testen, Reviews, Pair Programming, DoD, GitHub Pull Requests, UML, Patterns, Doku, Wiki
- Aufteilung: SW Architektur, Backlog, git Branching, Assignment of Responsibilities (single resp.), DRY

Non-Programming

**Software Projekt-Management**

- Scrum/PO, Unified Process, Continuous Integr., Testen, Requirements, Story Splitting & Mapping, DoD
- Nichts vergessen, Unsicheres sichtbar machen, Metriken, Kommunizieren, „Keeper of the Scope“

# Proving Programs Correct

## Testing has its limits

Even total code coverage (regardless of the metric used) does not guarantee correctness. "Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence."

Implementation:

```
public void static abs(int x) {
    return -x;
}
```

Test code:

```
public void testAbs() {
    assertTrue(MyMath.abs(-2) == 2);
}
```

100% path coverage

Better coverage of the problem domain (e.g. using equivalence partitioning & boundary value analysis) could catch this bug.

Q: But how do I know if this "better coverage" is "good enough"?

Implementation:

```
public class QuickSort<T extends Comparable<T>> {
    public void sortArray(T[] array) {
        quickSort(array, 0, array.length - 1);
    }

    private void quickSort(T[] array, int left, int right) {
        int i = left, j = right;
        T t = array[(left + right) / 2];
        do {
            while (array[i].compareTo(t) < 0) {
                i++;
            }
            while (array[j].compareTo(t) > 0) {
                j--;
            }
            if (i <= j) {
                T temp = array[i];
                array[i] = array[j];
                array[j] = temp;
                i++;
            }
        } while (i <= j);
        if (j > left) {
            quickSort(array, left, j);
        }
        if (i < right) {
            quickSort(array, i, right);
        }
    }
}
```

Test code:

```
@Test
public void testSimple() {
    Integer[] array = new Integer[] { 1, 3, 4, 2, 5, 8 };
    new QuickSort<Integer>().sortArray(array);
    Integer[] clone = Arrays.copyOf(array, array.length);
    Arrays.sort(clone);
    assertEquals(array, clone);
}
```

100% statement coverage

The implementation still has a major bug.

Ex: Find a test case that exposes this bug. Use the systematic techniques that you have learnt till now (equivalence classes, boundary value analysis).

Finding a good set of test cases is difficult work.

There is no guarantee that a set of test cases is "good enough".

Test Cases:

1. Empty: {} → passes
2. Singleton: {1} → passes
3. OddUnsorted: {1, 3, 2} → passes
4. EvenUnsorted: {1, 3, 2, 0} → passes
5. Repeating: {1, 2, 2, 1} → fails! (stack overflow)

This was relatively easy, but:

- I have little information on how to fix the bug.
- I still do not know if there are more bugs.

Q: Would you use this code to control an X-ray beam?

5

## Proving Program Correctness

Computer programs are formal texts. It should therefore be possible to prove their correctness in the same way that it is possible to prove  $x(x+1) = x^2 + x$ . My aim: To show you that this is indeed possible.

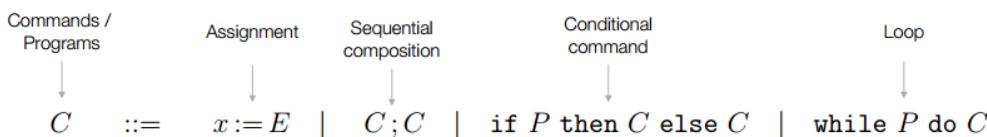
A Software Engineering dream that is reality only for a select few. Considered by many to be "hard core" Software Engineering (i.e. application of mathematical principles to systematically and rigorously design software). Opinions on this are divided.

Scope for this chapter: imperative single threaded programs

**Note:** The following slides also contain formal explanations on why the approach we use is correct. These explanations, although helpful in understanding why the approach works, will not be directly tested during the SE2 examination.

**Relevant** for the SE2 exam: The application of the techniques presented to prove the correctness of concrete programs (as stated in the goals).

## The simple Imperative Language



Some examples of programs:

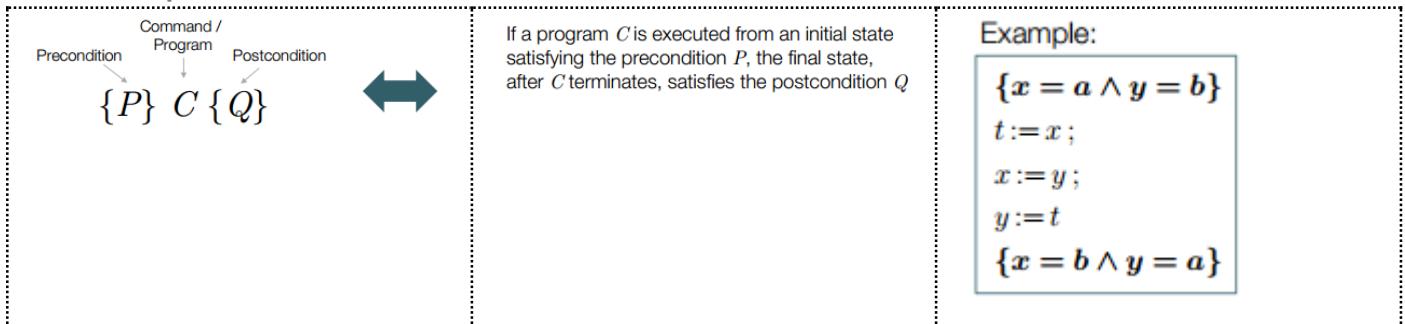
```
t := x;
x := y;
y := t
```

```
if x < 0 then
    y := -x
else
    y := x
```

```
i := 0;
while i ≠ x do
    i := i + 1
```

Note:

- The parentheses {}, or indentation will be used to resolve syntax ambiguities.
- Convention: Sequential composition will be left associative, i.e.  $C_1; C_2; C_3$  represents  $\{C_1; C_2\}; C_3$



Used to express the correctness properties of a program. The precondition, postcondition and the Hoare triple itself are predicates: they have a truth (i.e. Boolean) value.  $P$  and  $Q$  express properties of the program state by referring to the variables used in  $C$  as free variables

### Inference Rules of Hoare Logic

Syntactic substitution	Assignment
$\frac{}{\vdash \{[x := E]P\} x := E \{P\}}$	<i>assn</i>
$\frac{\vdash \{P\} C_1 \{Q\} \quad \vdash \{Q\} C_2 \{R\}}{\vdash \{P\} C_1 ; C_2 \{R\}}$	<i>seq</i>
$\frac{\vdash \{P \wedge B\} C_1 \{Q\} \quad \vdash \{P \wedge \neg B\} C_2 \{Q\}}{\vdash \{P\} \text{ if } B \text{ then } C_1 \text{ else } C_2 \{Q\}}$	<i>if</i>
$\frac{\vdash \{I \wedge B\} C \{I\}}{\vdash \{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\}}$	<i>while</i>
$\frac{\vdash P \Rightarrow P' \quad \vdash \{P'\} C \{Q'\} \quad \vdash Q' \Rightarrow Q}{\vdash \{P\} C \{Q\}}$	<i>conseq</i>

Aka “axiomatic semantics” of imperative programs, Defines the “meaning” of an imperative program in terms of its properties, Can be used to construct proofs of Hoare triples, Used later as the basis for generating weakest preconditions.

### A simple Hoare logic proof

$\frac{\vdash x = a \Rightarrow (x + 1) + 2 = a + 3 \quad \frac{\vdash x := x + 1 \quad \frac{\vdash x + 2 \quad \frac{\vdash x = a + 3}{\vdash x = a \wedge x := x + 1 \quad \{x + 2 = a + 3\}}}{\vdash x + 2 = a + 3 \Rightarrow x + 2 = a + 3}}{\vdash x = a \wedge x := x + 1 \quad \{x + 2 = a + 3\}}$	$\frac{\vdash x = a \wedge x := x + 1 \quad \frac{\vdash x := x + 2 \quad \frac{\vdash x = a + 3 \quad \frac{\vdash x = a \wedge x := x + 1; x := x + 2 \quad \{x = a + 3\}}{\vdash x = a \wedge x := x + 1; x := x + 2 \quad \{x = a + 3\}}}{\vdash x = a \wedge x := x + 2 \quad \{x = a + 3\}}}{\vdash x = a \wedge x := x + 2 \quad \{x = a + 3\}}$
---	---

The inference rules for Hoare logic are declarative: they specify what proofs are valid, but not specify any strategy on how to construct valid proofs. Proofs of program correctness in this (proof tree) style are possible, but very repetitive and not very intuitive. There has to be a simpler and more intuitive style to construct and write such proofs.

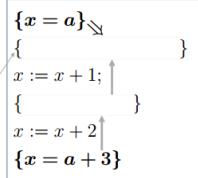
## Weakest Precondition Style Proof

Consider the following strategy to prove  $\{P\} C \{Q\}$ :

1. Start with the required postcondition  $Q$ .
2. Work your way backwards through the program and calculate, at each command  $C_p$ , the weakest precondition  $wp(C_p, Q)$  that must hold for the postcondition  $Q$ .
3. Once you have reached the beginning of the program  $C$ , you have calculated the weakest precondition of the entire program  $wp(C, Q)$ .
4. Prove that the specified precondition  $P$  implies the weakest precondition  $wp(C, Q)$  you have just calculated. Note that this proof is the proof of a formula without any commands.

$$\begin{aligned} wp(x := E, P) &\triangleq [x := E]P \\ wp(C_1 ; C_2, P) &\triangleq wp(C_1, wp(C_2, P)) \end{aligned}$$

Example 1:

Resulting Verification Condition:  
 $\vdash (x = a) \Rightarrow ((x + 1) + 2 = a + 3)$  ✓

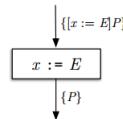
## Calculating Weakest Preconditions

The  $_:=$  command

Here is a diagrammatic explanation with flowcharts to aid your understanding:

$$\vdash \{[x := E]P\} x := E \{P\} \text{ assn}$$

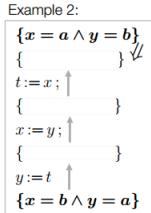
$$wp(x := E, P) \triangleq [x := E]P$$



## Weakest Precondition Style Proof

$$\begin{aligned} wp(x := E, P) &\triangleq [x := E]P \\ wp(C_1 ; C_2, P) &\triangleq wp(C_1, wp(C_2, P)) \end{aligned}$$

Lets try another example:

Resulting Verification Condition:  
 $\vdash (x = a \wedge y = b) \Rightarrow (y = b \wedge x = a)$  ✓

## Calculating Weakest Preconditions

The  $_:=$  and  $_;$  commandsWeakest preconditions for the  $_:=$  and  $_;$  commands can be calculated using the following definitions:

$$\begin{aligned} wp(x := E, P) &\triangleq [x := E]P \\ wp(C_1 ; C_2, P) &\triangleq wp(C_1, wp(C_2, P)) \end{aligned}$$

Their validity can be seen by plugging them into the relevant Hoare logic inference rules:

$$\frac{\vdash \{x := E\}P \quad x := E \{P\} \text{ assn}}{\vdash \{P\} C_1 \{Q\} \quad \vdash \{Q\} C_2 \{R\}} \text{ seq}$$

In general, it can be proven, that the weakest precondition is a valid precondition w.r.t. the rules of Hoare logic:

$$\vdash \{wp(C, P)\} C \{P\} \text{ wp_sound}$$

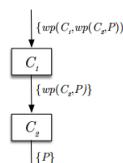
## Calculating Weakest Preconditions

The  $_;$  command

Here is a diagrammatic explanations with flowcharts to aid your understanding:

$$\frac{\vdash \{P\} C_1 \{Q\} \quad \vdash \{Q\} C_2 \{R\}}{\vdash \{P\} C_1 ; C_2 \{R\}} \text{ seq}$$

$$wp(C_1 ; C_2, P) \triangleq wp(C_1, wp(C_2, P))$$

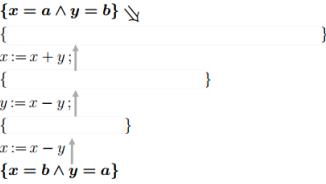


$$\begin{aligned} wp(x := E, P) &\triangleq [x := E]P \\ wp(C_1 ; C_2, P) &\triangleq wp(C_1, wp(C_2, P)) \end{aligned}$$

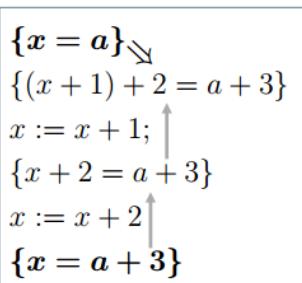
## Weakest Precondition Style Proof

Lets try something non-trivial:

Example 3:

Resulting Verification Condition:  
 $\vdash (x = a \wedge y = b) \Rightarrow ((x + y) - ((x + y) - y) = b \wedge ((x + y) - y) = a)$  ✓

## Automated Program Verification



Resulting Verification Conditions:

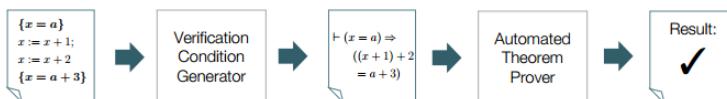
$$\vdash (x = a) \Rightarrow ((x + 1) + 2 = a + 3) \quad \checkmark$$

The generation of verification conditions is tedious, but can be done automatically. The generated verification conditions are logical formulae in the underlying logic that do not contain any commands. There exist automated theorem provers for many useful underlying logics (e.g. arithmetic, arrays, lists, etc.). A large part of program verification can therefore be automated.

General Workflow:



Example:



## Weakest Precondition Style Proof Nr. 2

We have just seen that this works for programs that only contain `_:=_` and `_;_`. But this technique also works for programs with if and while.

### Weakest Precondition Style Proof

Fortunately for us, the weakest precondition of any IMP program exists and can be calculated recursively using the following definitions:

$$\begin{aligned} wp(x := E, P) &\triangleq [x := E]P \\ wp(C_1 ; C_2, P) &\triangleq wp(C_1, wp(C_2, P)) \quad (\triangleq_{wp_{\text{assoc}}}) \\ wp(\text{if } B \text{ then } C_1 \text{ else } C_2, P) &\triangleq (B \Rightarrow wp(C_1, P)) \wedge \\ &\quad (\neg B \Rightarrow wp(C_2, P)) \quad (\triangleq_{wp_{\text{if}}}) \\ wp(\text{while } B \text{ do } C, P) &\triangleq I \\ &\quad \text{if } (I \wedge B) \Rightarrow wp(C, I) \quad (\triangleq_{wp_{\text{while}}}) \\ &\quad \text{and } (I \wedge \neg B) \Rightarrow P \end{aligned}$$

It can be proven, that the weakest precondition is a valid precondition w.r.t. the rules of Hoare logic:

$$H \vdash \{wp(C, P)\} C \{P\} \quad wp_{\text{sound}}$$

As we can see from the above definitions, the weakest preconditions for if and while are more complicated.

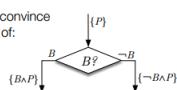
### Calculating Weakest Preconditions

#### The if command

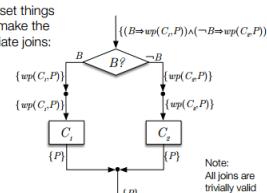
Here is a diagrammatic explanation with flowcharts to aid your understanding:

$$\begin{array}{c} H \vdash \{P \wedge B\} C_1 \{Q\} \quad H \vdash \{P \wedge \neg B\} C_2 \{Q\} \quad \text{if} \\ H \vdash \{P\} \text{ if } B \text{ then } C_1 \text{ else } C_2 \{Q\} \end{array}$$

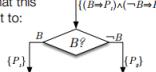
1. First, convince yourself of:



3. Then set things up and make the appropriate joins:



2 ... and that this is equivalent to:



### Calculating Weakest Preconditions

#### The while command

Weakest preconditions for the while command can be calculated using the following definition:

$$wp(\text{while } B \text{ do } C, P) \triangleq I \quad \text{if } (I \wedge B) \Rightarrow wp(C, I) \quad \text{side conditions} \quad \text{and } (I \wedge \neg B) \Rightarrow P$$

Note:

- The weakest preconditions for a loop is the loop invariant  $I$ .
- Two side conditions need to be separately proven in order for  $I$  to be used as the weakest precondition.
- There is no way of automatically computing  $I$ .
- $I$  must therefore be developed manually as a program annotation within the program specification.
- The choice of  $I$  depends on what needs to be proven (i.e. the pre- and post conditions).
- Finding a valid loop invariant that can be used to prove a program correct is often the most mentally challenging task in program verification.

$$H \vdash \{I \wedge B\} C \{I\} \quad H \vdash \{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\} \quad \text{while}$$

$$H \vdash \{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\} \quad \text{while}$$

### Weakest Precondition Style Proof

#### Lets try an example with while:

Example 'countUp':

```
{x ≥ 0}
{ }
i := 0;↑
{ }
while i ≠ x do {Inv : {i ≤ x}}
{ }
{i := i + 1;↑}
{ }
{i = x} ↴②
```

$$wp(\text{while } B \text{ do } C, P) \triangleq I \quad \text{if } (I \wedge B) \Rightarrow wp(C, I) \quad \text{and } (I \wedge \neg B) \Rightarrow P$$

Resulting Verification Conditions:

$$\begin{aligned} &\vdash (x \geq 0) \Rightarrow (0 \leq x) & ① & \checkmark \\ &\vdash (i \leq x \wedge i \neq x) \Rightarrow (i + 1 \leq x) & ① & \checkmark \\ &\vdash (i \leq x \wedge \neg(i \neq x)) \Rightarrow (i = x) & ② & \checkmark \end{aligned}$$

Note: We were lucky that our first attempt at finding an invariant was successful. Ordinarily, a number of attempts are needed to find the right invariant. Automated program verification makes this process faster.

### Calculating Weakest Preconditions

#### The if command

Weakest preconditions for the if command can be calculated using the following definition:

$$wp(\text{if } B \text{ then } C_1 \text{ else } C_2, P) \triangleq (B \Rightarrow wp(C_1, P)) \wedge (\neg B \Rightarrow wp(C_2, P))$$

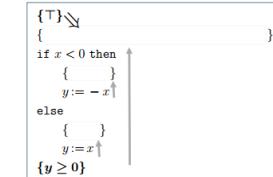
Its validity can be seen by plugging them into the following Hoare logic inference rule:

$$\frac{H \vdash \{P \wedge B\} C_1 \{Q\} \quad H \vdash \{P \wedge \neg B\} C_2 \{Q\}}{H \vdash \{P\} \text{ if } B \text{ then } C_1 \text{ else } C_2 \{Q\}} \text{ if}$$

### Weakest Precondition Style Proof

#### Lets try an example with if:

Example 'abs':



Resulting Verification Condition:

$$\vdash T \Rightarrow ((x < 0 \Rightarrow -x \geq 0) \wedge (\neg(x < 0) \Rightarrow x \geq 0)) \quad \checkmark$$

### Calculating Weakest Preconditions

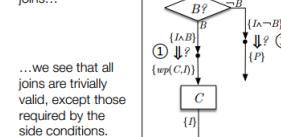
#### The while command

Here is a diagrammatic explanation with flowcharts to aid your understanding:

$$H \vdash \{I \wedge B\} C \{I\} \quad H \vdash \{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\} \quad \text{while}$$

$$wp(\text{while } B \text{ do } C, P) \triangleq I \quad \text{if } (I \wedge B) \Rightarrow wp(C, I) \quad \text{①} \quad \text{and } (I \wedge \neg B) \Rightarrow P \quad \text{②}$$

Setting things up and making the appropriate joins...



...we see that all joins are trivially valid, except those required by the side conditions.

The side conditions can be read as follows:

- ① The loop body maintains the invariant.  
② The invariant establishes the post condition.

The side-conditions are added to the verification conditions during automated program verification.

### Weakest Precondition Style Proof

Lets extend the last example to do something interesting:

Example 'multi':

```
{x ≥ 0} ↴①
{0 ≤ x}
x := 0;↑
{ }
while i ≠ x do {Inv : {i ≤ x}}
{ }
{i := i + 1;↑}
{ }
{i = x} ↴②
```

$$H \vdash \{I \wedge B\} C \{I\} \quad H \vdash \{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\} \quad \text{while}$$

$$wp(\text{while } B \text{ do } C, P) \triangleq I \quad \text{if } (I \wedge B) \Rightarrow wp(C, I) \quad \text{and } (I \wedge \neg B) \Rightarrow P$$

Resulting Verification Conditions:

$$\begin{aligned} &\vdash (x \geq 0) \Rightarrow (0 \leq x) & ① & \checkmark \\ &\vdash (i \leq x \wedge i \neq x) \Rightarrow (i + 1 \leq x) & ① & \checkmark \\ &\vdash (i \leq x \wedge \neg(i \neq x)) \Rightarrow (i = x) & ② & \checkmark \end{aligned}$$

## Dafny

A Language and Program Verifier for Functional Correctness, From Microsoft Research, Uses the same techniques seen till now, With extensions for total correctness, methods, classes, heap datatypes, ...  
Can be compiled into C#.

### Homepage

<https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/>

### Weitere Infos

Sie Dafny for VSCode Presentation.