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
* Programing in the functional style
* Scripting Languages
* Software Failure Analysis

Übersicht des Modules

[Projektplanung 4](#_Toc477087880)

[Drei Themenkreise 4](#_Toc477087881)

[Agil, iterativ 4](#_Toc477087882)

[Ein Beispiel-Projekt mit 18 Iterationen 4](#_Toc477087883)

[Checkliste End of Elaboration 5](#_Toc477087884)

[Checkpoints / Meilensteine 5](#_Toc477087885)

[Empfehlungen für Engineering-Projekte 5](#_Toc477087886)

[Meilensteine und Reviews im Eng.-Projekt 6](#_Toc477087887)

[Meilensteine und Versionen in Redmine 6](#_Toc477087888)

[Vorschlag für Umsetzung der Meilensteine in Redmine-Zielversionen. 6](#_Toc477087889)

[Anforderungen: Use Cases etc. 6](#_Toc477087890)

[Use Case Diagramm 6](#_Toc477087891)

[Use Cases eines filtiven Beispieles 6](#_Toc477087892)

[Nicht funktionale Anforderungen 7](#_Toc477087893)

[Domain-Model 7](#_Toc477087894)

[Anforderungen 🡪 Arbeitspakete 7](#_Toc477087895)

[Diagramme und Dokumente 9](#_Toc477087896)

[Projektdokumentation 10](#_Toc477087897)

[«End of Elaboration» = Wendepunkt 10](#_Toc477087898)

[Projektautomatation 11](#_Toc477087899)

[Wie entwickeln wir Software? 11](#_Toc477087900)

[Idee 1 – Build Skript 11](#_Toc477087901)

[Was wir möchten (Wunschliste)? 11](#_Toc477087902)

[The Beginning 12](#_Toc477087903)

[Der Beginn der Build Automatisierung – GNU Make (Imperativ) 12](#_Toc477087904)

[Apache Ant (Imperativ) 12](#_Toc477087905)

[Deklarative Builds mit Apache Maven 12](#_Toc477087906)

[Post-Maven Tools 13](#_Toc477087907)

[The Future 13](#_Toc477087908)

[Zusammenfassung der Vorteile 13](#_Toc477087909)

[Continuous Integration 14](#_Toc477087910)

[Software Engineering Parctices 16](#_Toc477087911)

[Requirements Practices 16](#_Toc477087912)

[1. Dig for Requirements 16](#_Toc477087913)

[2. Make quality a requirement 16](#_Toc477087914)

[3. Deal with changes 17](#_Toc477087915)

[Design Practices 17](#_Toc477087916)

[4. Don’t repeat yourself 17](#_Toc477087917)

[5. Achieve orthogonality 17](#_Toc477087918)

[6. Design to test 18](#_Toc477087919)

[Implementation Practices 19](#_Toc477087920)

[7. Fix broken windows 19](#_Toc477087921)

[8. Refactor early and often 19](#_Toc477087922)

[9. Program deliberately 19](#_Toc477087923)

[Verfication Practices 19](#_Toc477087924)

[10. Test rigorously 19](#_Toc477087925)

[11. Perform Reviews 20](#_Toc477087926)

[Vorlesung Woche 4 20](#_Toc477087927)

# 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?

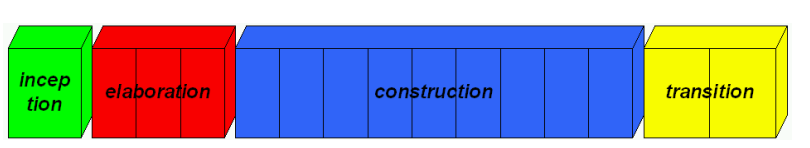
Dieses Kapitel deckt folgende drei Themenkreise ab:

* 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:

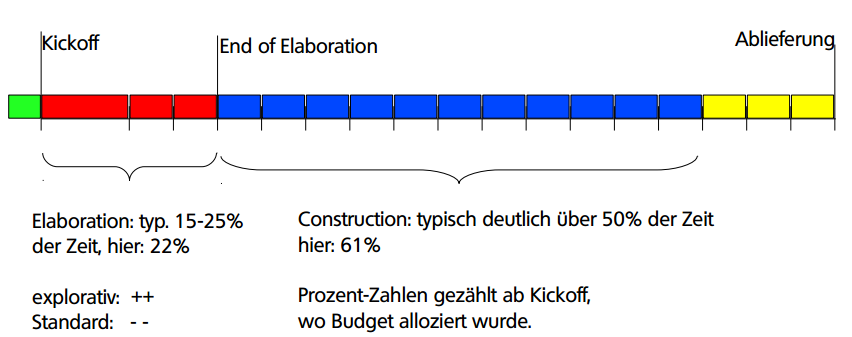
* Agil (Scrum; eXtreme Programming XP)
* Iterativ (RUP, Iterative Development, Spiral Development...)



### 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 Interationen, eine Construction von 11 Iterationen à 2 W. und ein Transistion von 3 Iterationen à 2 Wochen.

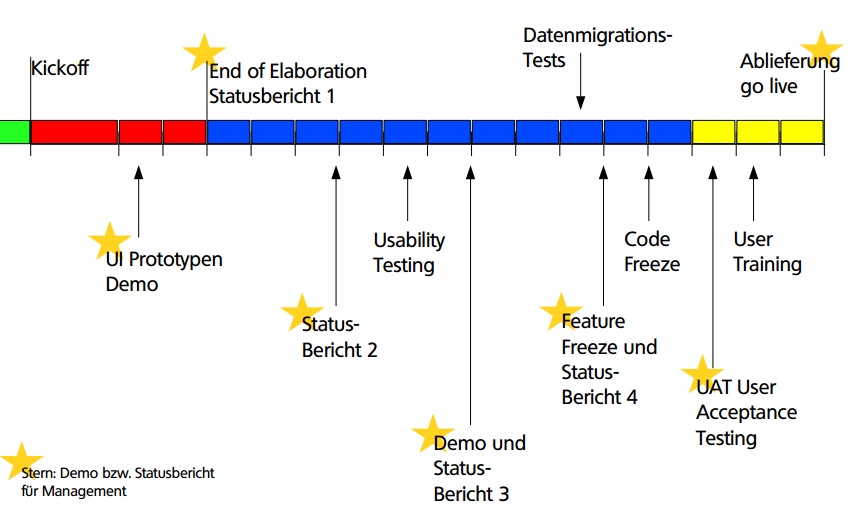
**Anzahl Personen**  
Personal wird vor allem in der intensiven Constuction-Phase benötigt. Wichtig ist das der Chief Architect von Kickoff bis mindestens Mitte Projekt involviert ist.

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

### Checkliste End of Elaboration

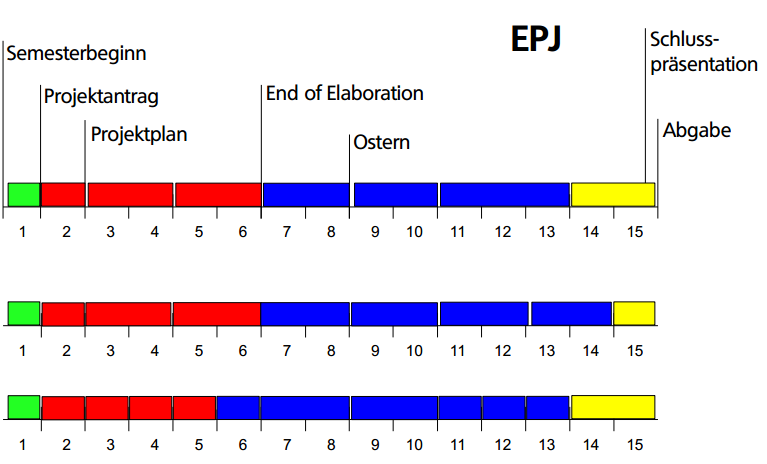
* Anforderungen (Requirements): Haben wir den Kunden verstanden? Funktionsumfang (Scope) ist abgesteckt durch UCs, Domain Model, nicht-funktionale Anforderungen
* User Interface Design: Entwürfe gemacht, dem Kunden gezeigt; wenn möglich Clickable Prototypes plus Grafik-Entwürfe (Farben, Schriften)
* 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.).
* Genauere 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 Reviews im Eng.-Projekt

### 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.

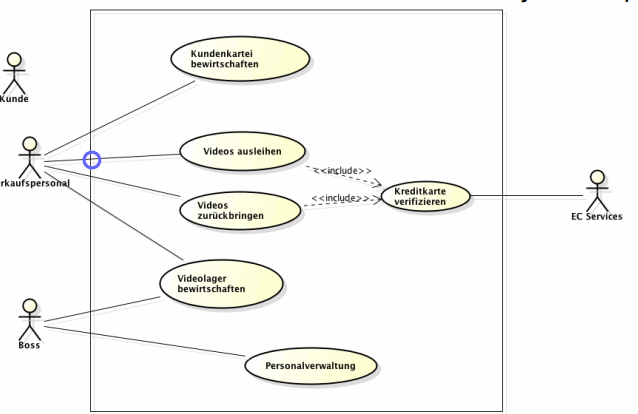
**Redmine-Versionen**

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

## Anforderungen: Use Cases etc.

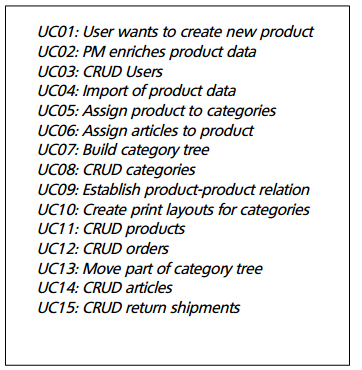
Use Cases sind Beschreibungen klar aus der Benutzersicht, also gut mit dem Kunden kommunizierbar. Use Case beschreiben die Funktionalität im Kontext (z.B. warum, was in welcher Reihenfolge und wie oft?) Es gibt verschiedene Detaillierungsgrade. Von Brief (3 Zeilen), über casual (eine halbe Seite) bis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

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

### 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-Model

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 drinstehen? 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 priorisier bar 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.

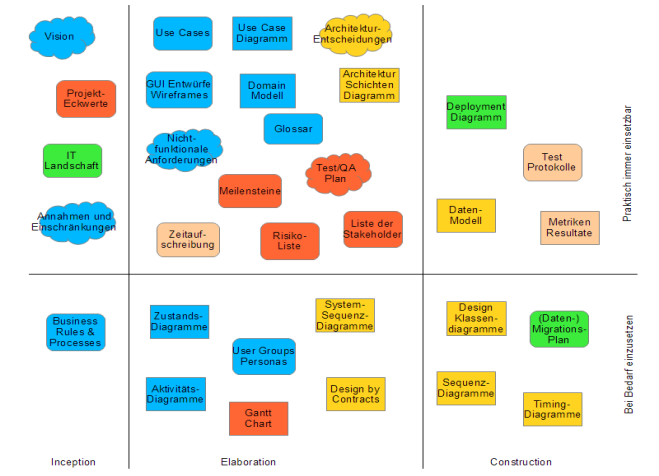
**Planung pro Iteration**  
Genug Arbeitspakete, damit alle im Team während der nächsten Iteration beschäftigt sind. Genau soviele Arbeitspakete wie die Schätzungen zulassen, dass sie auch innerhalb der Iteration fertig werden. Innerhalb des Teams werden die Arbeitspakete eigenverantwortlich zugeordnet,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

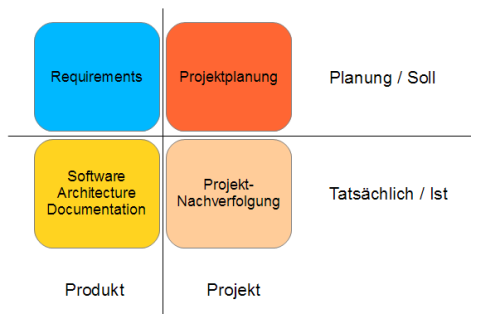
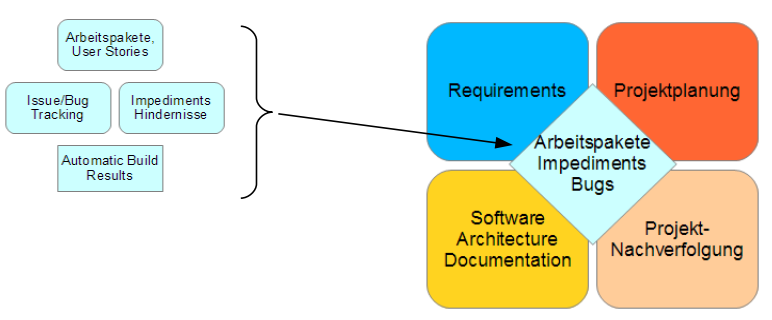
**Regeln bei der Planung**  
Der Entwickler schätzt den Aufwand für die Arbeitspakete, während der Kunde die Arbeitspakete priorisiert. Und nicht umgekehrt. Die einzige Ausnahme sind die architekturrelevanten Arbeitspakete. Diese können vom System-Architekten in bestimmte Iterationen gesetzt werden,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

**Fallstrickte 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 Constuction.



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, inbesondere 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.

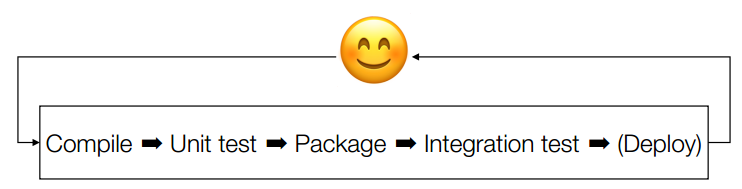
# Projektautomatation

## 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**  


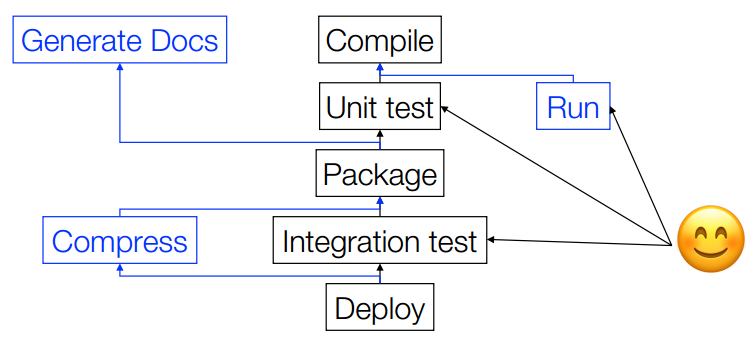
**Vorteile**  
Die ganze Sache ist nun automatisiert in einem nicht interaktiven Prozess. Das Skript kann mehrere Male ausgeführt werden, es ist also repetierbar. Zudem ist es unabhängig von der IDE. Nicht zuletzt können zeitintensive Tasks terminiert werden.

**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)
  + Complete – Jeden Build von neu aufbauen
  + Repeatable – Immer wieder anstossen, auch älteren Code wieder auschecken und builden können
  + Imformative – Testresultate von Unit- und Integrationstests
  + Schedulable – Zeitlich terminierbar
  + 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

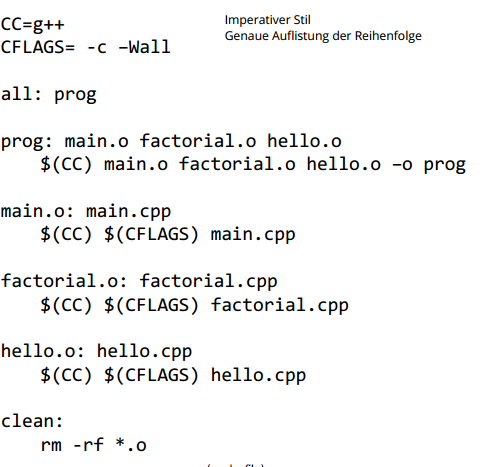
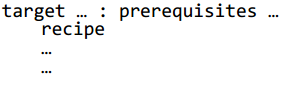
### Der Beginn der Build Automatisierung – GNU Make (Imperativ)

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-Scirpts). Die ganze Sache ist leider Plattform abhängig und hat kein automatischen Dependency Management. Gearbeitet wird mit Targets / Dependencies und Variables. Die Auflistung muss in der richtigen Reihenfolge erfolgen.

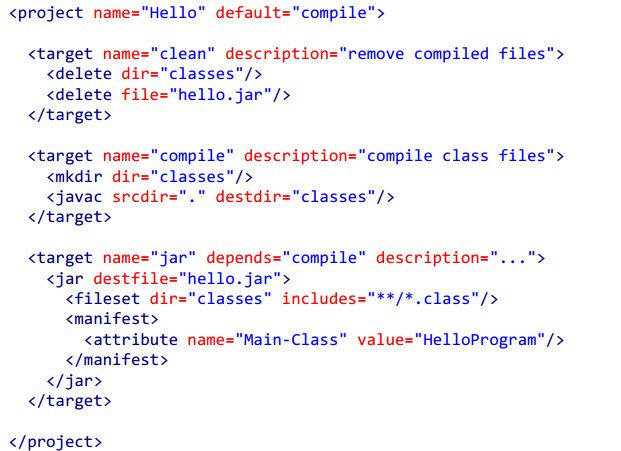
Der Build Author definiert explizit das DAG. Die Targets sind mit einer Skriptsprache implementiert und meistens basiert es auf einem externen Dependency Manager.

**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)

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

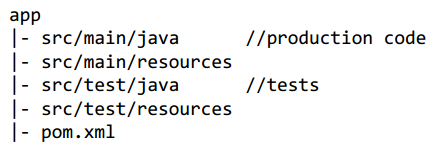
**Vorteile**  
Es umfangreich und flexibel.

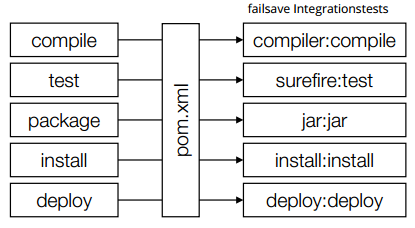
**Nachteile**  
Die Build Definitionen tendieren dazu sehr komplex zu werden. Zudem ist es schwierig die Build Logik wieder zu benutzen. Meist wird Copy Paste angewendet.

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





Im Gegensatz zu anderen Tools gibt der Build Author an, wie das Build Resulat 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 restrektiv. 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 Resulate
* Zeit sparen
* Basis für Continuous Integration (nächstes Kaptiel)

### Continuous Integration

«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 gebuildet werden**Jeder Änderung auf der Mainline sollte als kompletter Build auf dem CI Server ausführbar sein.

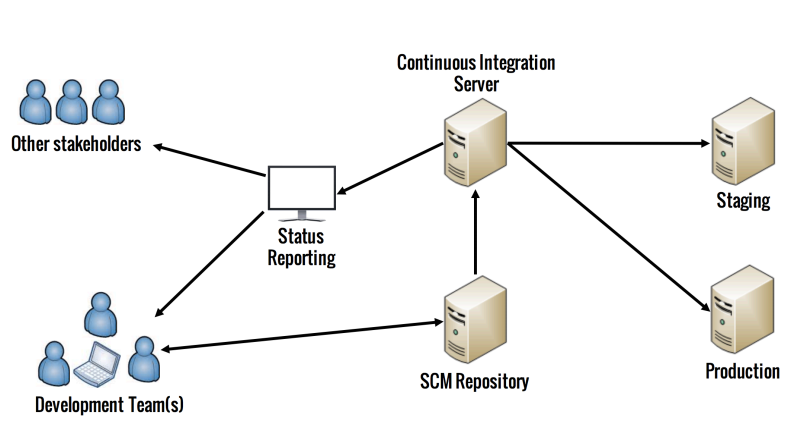
**6. Der Build sollte schnell gehalten werden**Um schnelles Feedback zu bekommen. Meist geschieht dies in mehreren Schritten (1. Build, 2. Integrations Tests und 3. Performance Tests).

**7. Testen in einem Klone 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
  + Nur kleine Änderungen machen
  + 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 Resulate. 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 Parctices

Sie sollen eine Link zwischen der Theorie und der Praxis herstelllen. 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 emplyee’s supervisors and the personell department may view that emplyee’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 anizipieren**  
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 Absraktionen 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.

#### Reduktion der Koppelung

Mit möglichst wenig Abhängigkeiten (Statisch (Code Import) sowie dynamisch (Calls). Zudem das Acylic 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 Kontrake
* 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 inspiziert werden

#### Beispiel – Dependency Injection

|  |  |
| --- | --- |
|  |  |

## Implementation Practices

### 7. Fix broken windows

Die Probleme sollten behoben warden, wenn Sie entstehent. Kleine Probleme sofort beheben und grössere Probleme markeiren, 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 Beriechen 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

## Verfication Practices

### 10. Test rigorously

Früh, häufig und automatisch testen

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

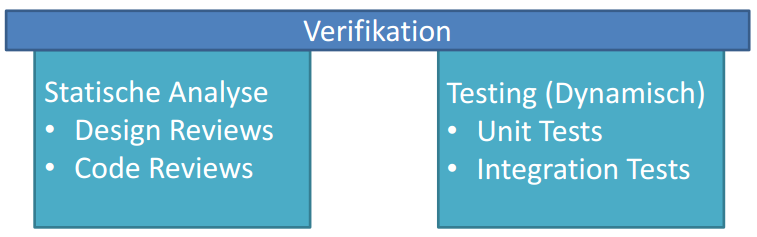
«Find Issues Once»

* 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.

#### Pfeiler der Verifikation



### 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 selbständig (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 Anaysis
* Error Handling Design (Fokus in diesem Kapitel)
* Concurrency Design und Testing

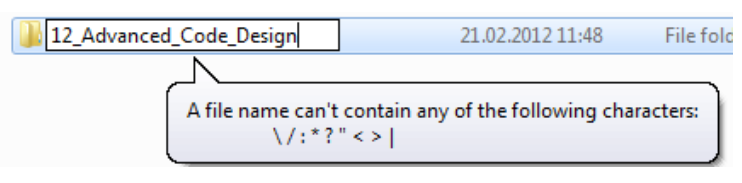
**Überblick**  
Ist ein wesentlicher Bestandteil der Software Architektur. Dazu gehört die Wahl der Error Handling Strategie sowie das Befolgen der Konsistent. Zu den relevanten Fragen gehören: Welche Fehler müssen erkannt werden? Was für eine Systematik wird dabei eingesetzt? Wie werden Fehler behandelt?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



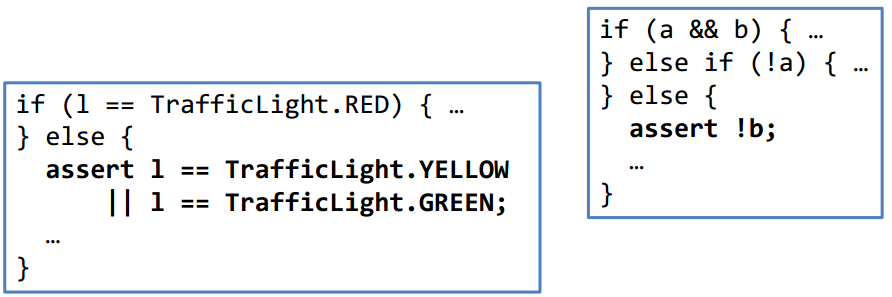
### Prüfung von Routinen-Input

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



### Abfangen ungültiger Fälle

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



### Beispiel

|  |  |
| --- | --- |
| **Ungültige Fälle**  In diesem Beispiel gibt es keinen Default Case. |  |

## Fehler-Barrikaden

**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-Techniken

**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
  + Explizit mit try-catch für alle top-Most Thread-Routinen
  + Pauschal mit Thread.setUncaughtExceptionHandler() (Pro Thread-Instantz)
  + 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 abschrimen) und den Fehler danach protokollieren.

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

### Exceptions vs. Assertions

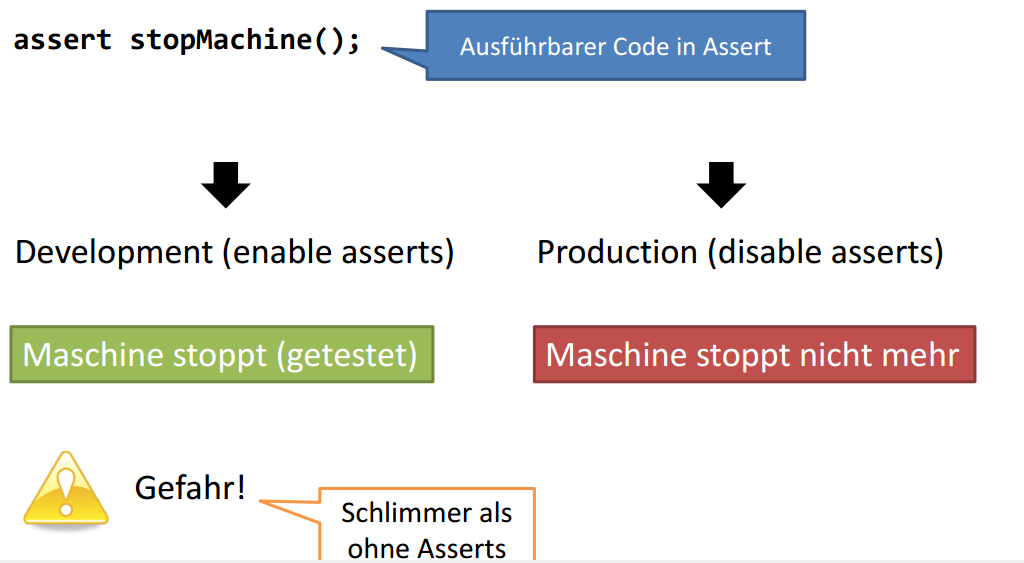
**Exceptions**

* 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



#### Assertions- Ein oder ausschalten?

In der Production 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**

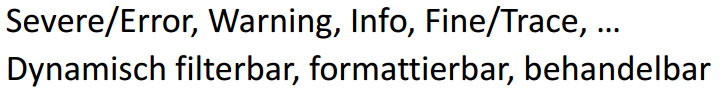
* Als Stütze während der Entwicklung
* Formale Kommentare

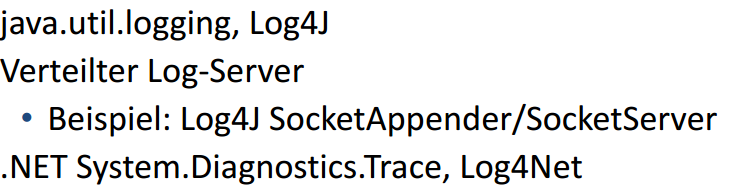
**Sekundär**

* 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**  


**Frameworks**  


## Anwendung

### Wichtung 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?
  + Rückmeldung an den Benutzer?
  + Abbruch?
  + Logging?

### Exceptions Policy

Folgendes gehört in diese Policy:

* Werden Exceptions benutzt?
* Wie?
  + Für Behandlung von welchen Arten von Fehlern
  + Local Handling /Global Handling?
  + 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.

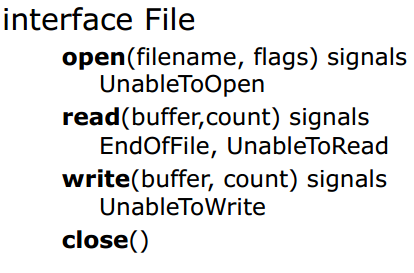
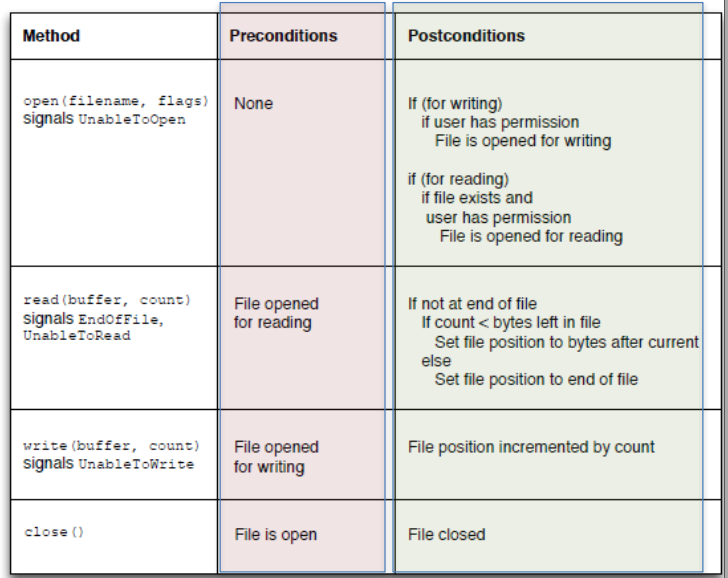
*Postconditions*  
Bedingungen, die nach dem Aufruf gültig sind. Verantwortlich ist die Implementation der Methode.

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

### Beispiel Stack

|  |  |
| --- | --- |
|  | Beispiel in Eiffel für put |
| Beispiel in Eiffel für Klasseninvariante Stack | **Beispiel Stack im icontract** Preconditions, Postconditions und Klasseninvariante spezifizieren Stack. Die Spezifizierung aber auf das Interface und nicht die Implementation spezifizieren. |

### Das Interface ist nicht die ganze Geschichte

Auch die Reihenfolge beziehungsweise der Zustand ist entscheidend.   


## Ziele

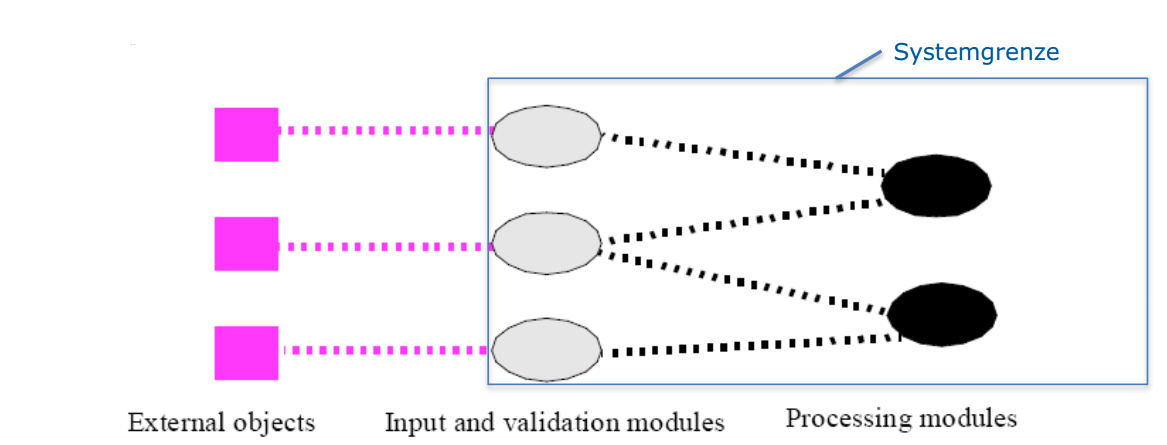
Es stellt eine grundlegende Entwurfsmethodik dar. Somit gehört es zum Grundwissen eines Informatik-Ingenieurs. Es ist nützlich, wenn es pragmatisch angewandt wird. Das heisst Fokus auf Spezifikation statt Implementation und ein klares Verständnis bezüglich den Verantwortlichkeiten.

**Illustration**  
Y = SQRT(X) 🡺 Wer ist jetzt verantwortlich zu prüfen, dass x>= 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.

**Postconditons 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 Postconditons 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)

## Gute Verträge, Anwendung in agilem Umfeld

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

## 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 Libary (Java: J4C)

### In Java

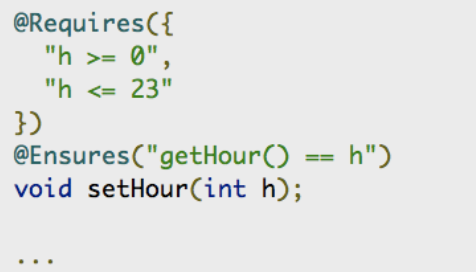
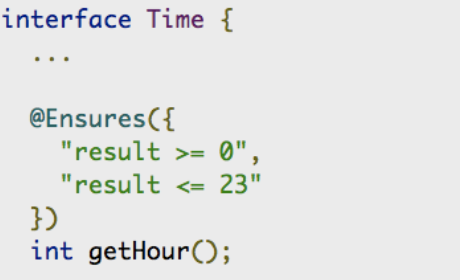
Die Präprozessoren icontract und icontract2 sind nicht mehr aktuell. 2008 wurde ein Versuch einer Weiterentwicklung gestartet, welcher aber gescheitert ist.

**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 vorallem 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»

Cofoja – Contracts for Java  
Ein Open Source Projekt, welches von Google im Jahre 2011 gestartet wurde.  


### 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** |

## 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 Werte).

* 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

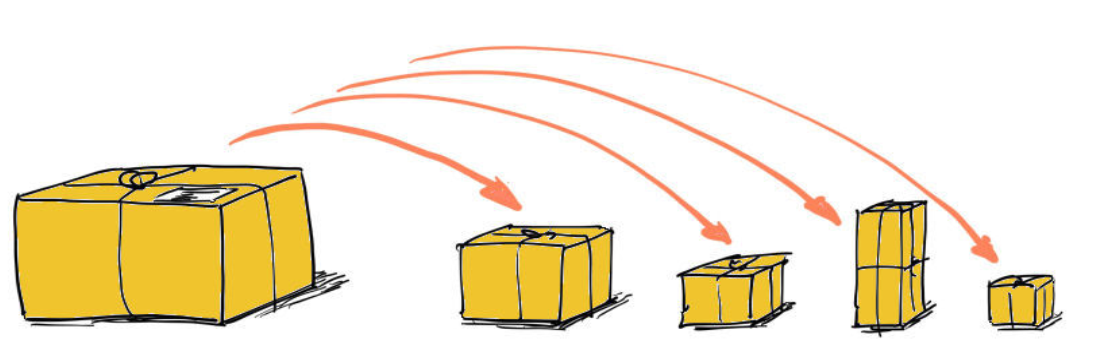
## Story Splitting – Aufteilung von zu grossen Projekten

**Zu grosses Projekt – Was tun?**  
Schon fast alltägliche Schreckens-Geschichten:

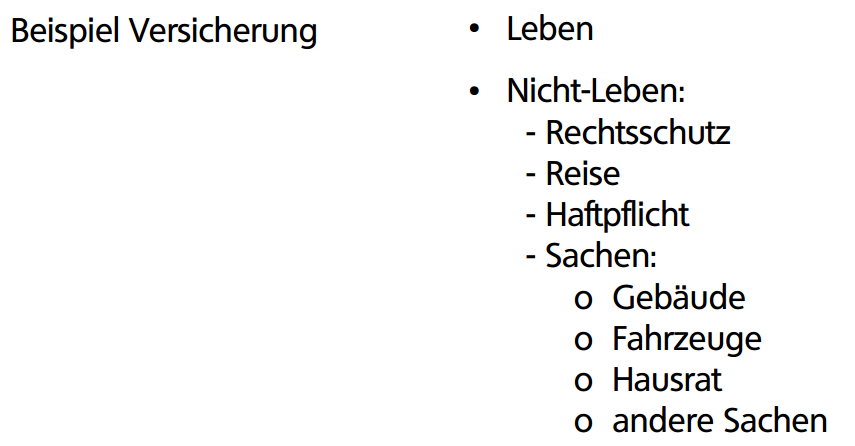
“Das Führungsinformationssystem (FIS) Heer, vor zehn Jahren als Prestigeprojekt der Armee lanciert, werde definitiv nie so funktionieren wie vorgesehen. Deshalb [ ... ] schätzungsweise 125 Millionen abgeschrieben werden.” (Tagesanzeiger 11.01.2017)

“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



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

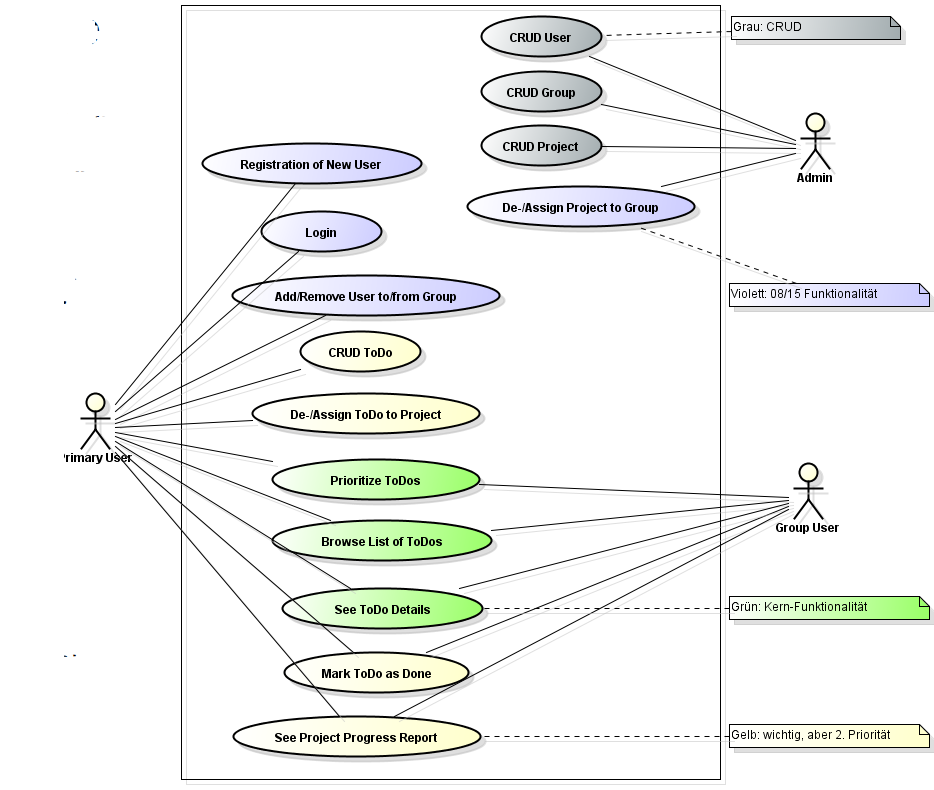
### Aufteilung nach Geschäfts-Prozessen

Tätigkeiten/Prozesse im Beispiel Gebäudeversicherungen

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

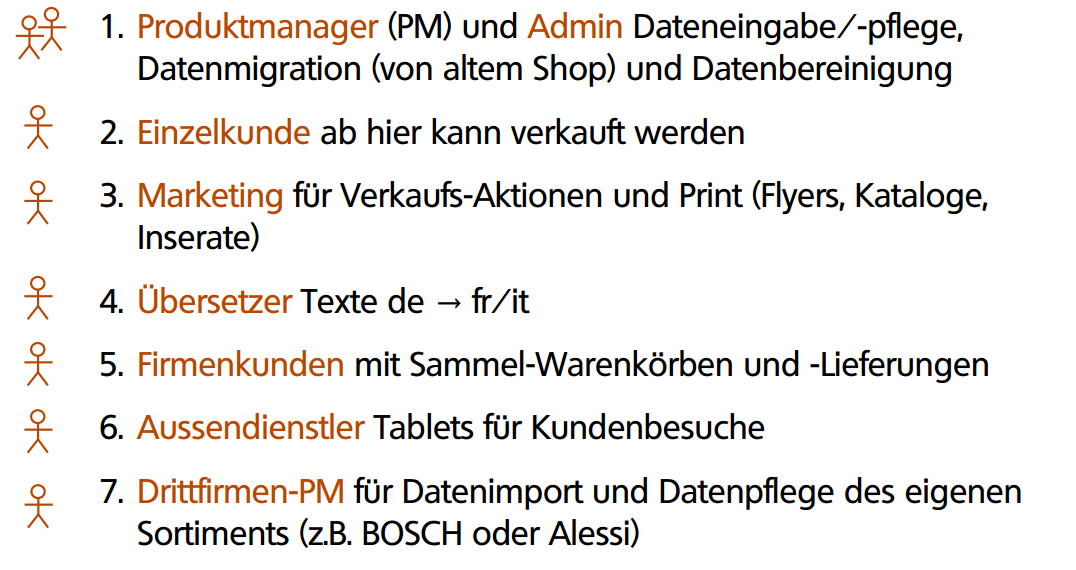
Das ist ungefähr das, was Sie in einem Projekt mit Use Cases definieren. Und die können manchmal gut nacheinander ausgeliefert werden.

### Reihenfolge? Wichtigkeit

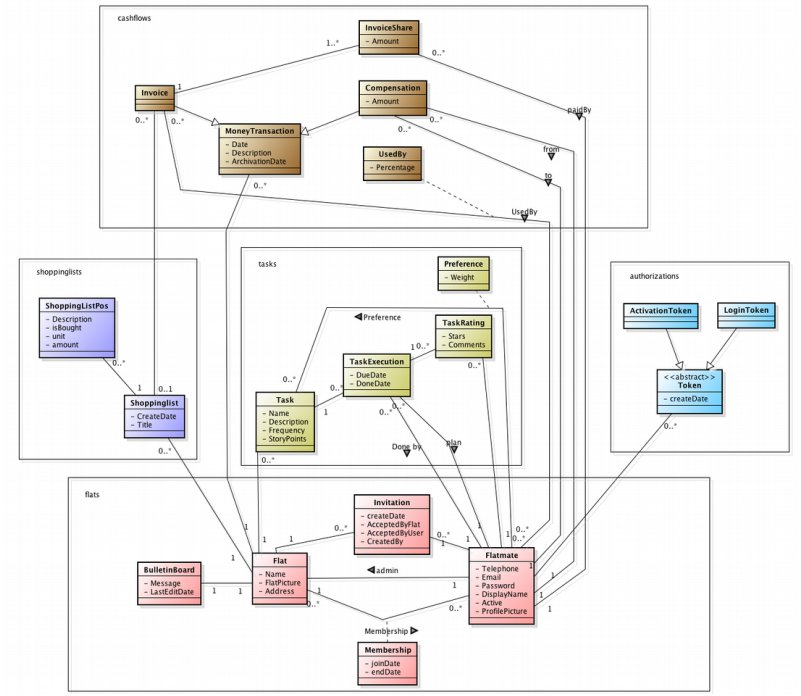
Sortieren nach Wichtigkeit

* 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:  


### Projekt-Aufteilung im Domain-Modell

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


### Projekt-Teillieferungen geografisch

Beispiel Steuersoftware:

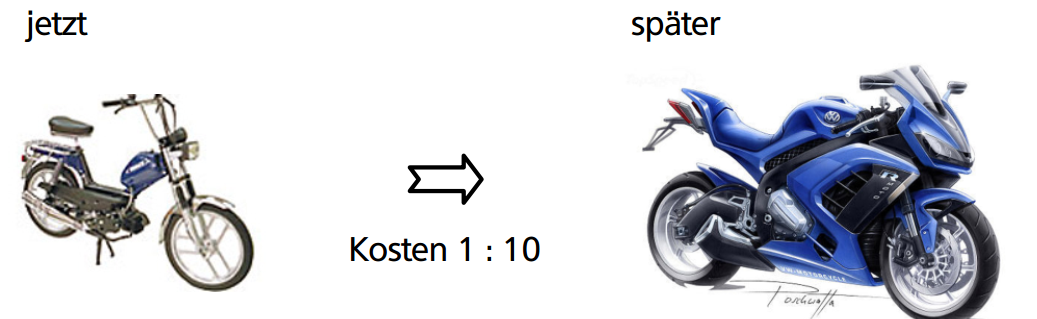
* Erst für Kanton Uri implementieren (relativ klein und einfach)
* Dann für Kanton Thurgau (grösser, wenig Sonderregelungen)
* Dann für Kanton Bern (viele Sonderregelungen wegen Bund)
* Dann für Kantone Zug und Schwyz (Sonderregelungen für internationale Geschellschaften)
* Danach den Rest der Schweiz

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.



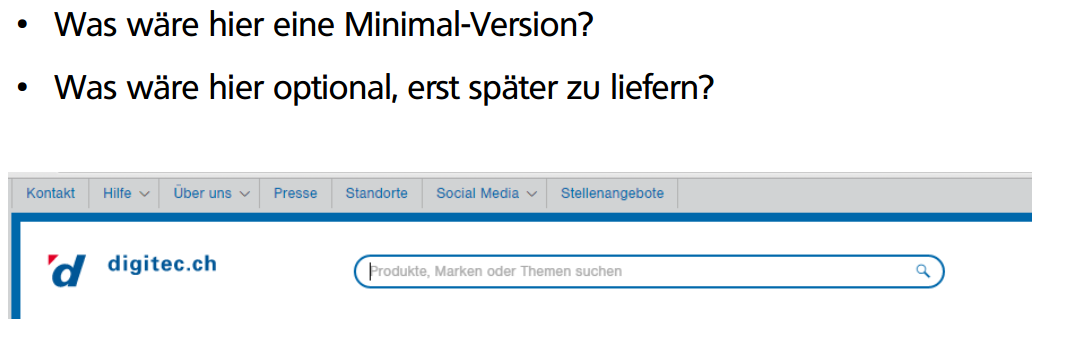
#### Beispiel 1 – US033 Login

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?  


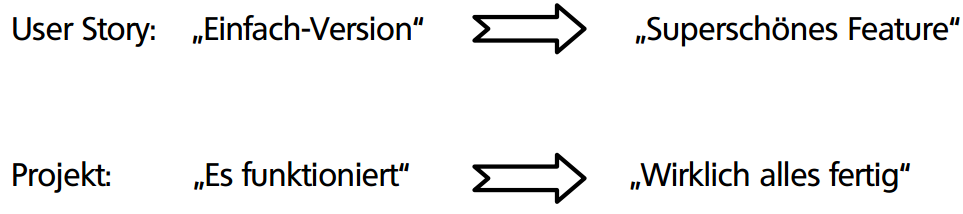
**Abschluss – US041 Produktsuche aufgeteilt**  
Diese User Story wird in 8 User Stories (Tasks, Arbeitspakete) aufgeteilt: a) US041 umgeschrieben in "Basis-Version" und b) 7 zusätzliche Stories.

* **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 Hife, 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.



**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

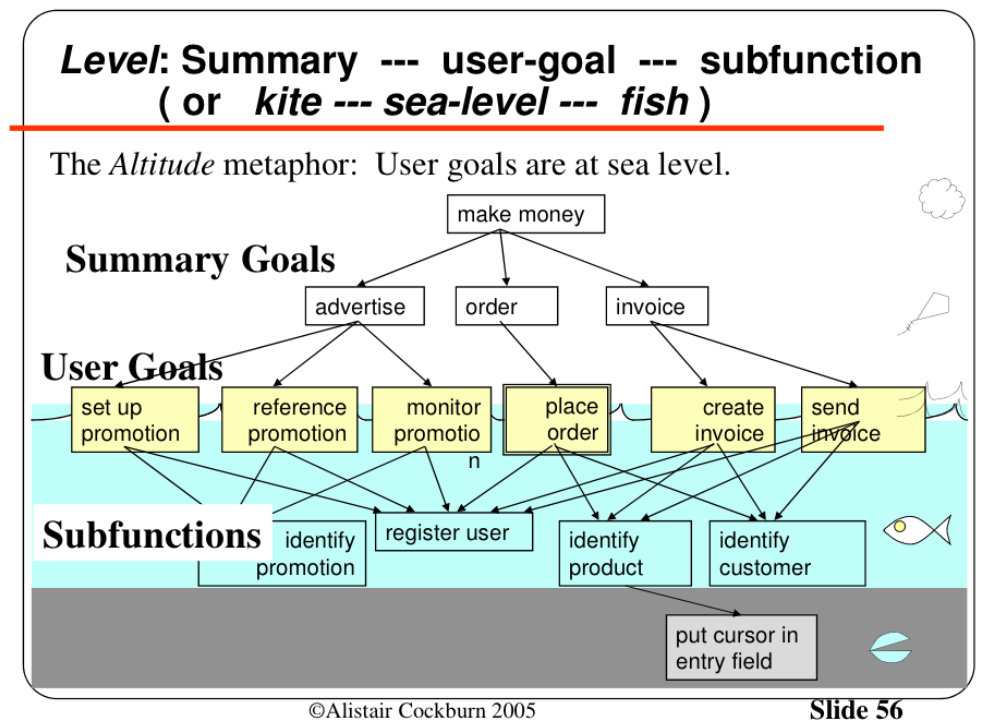
=> Durchschnitt: 2.5 Arbeitstage pro User Story bei Sprintlänge 2 Wochen genau des Sprints. ¼

Übrigens: somit kostete 1 User Story im Schnitt CHF 2500

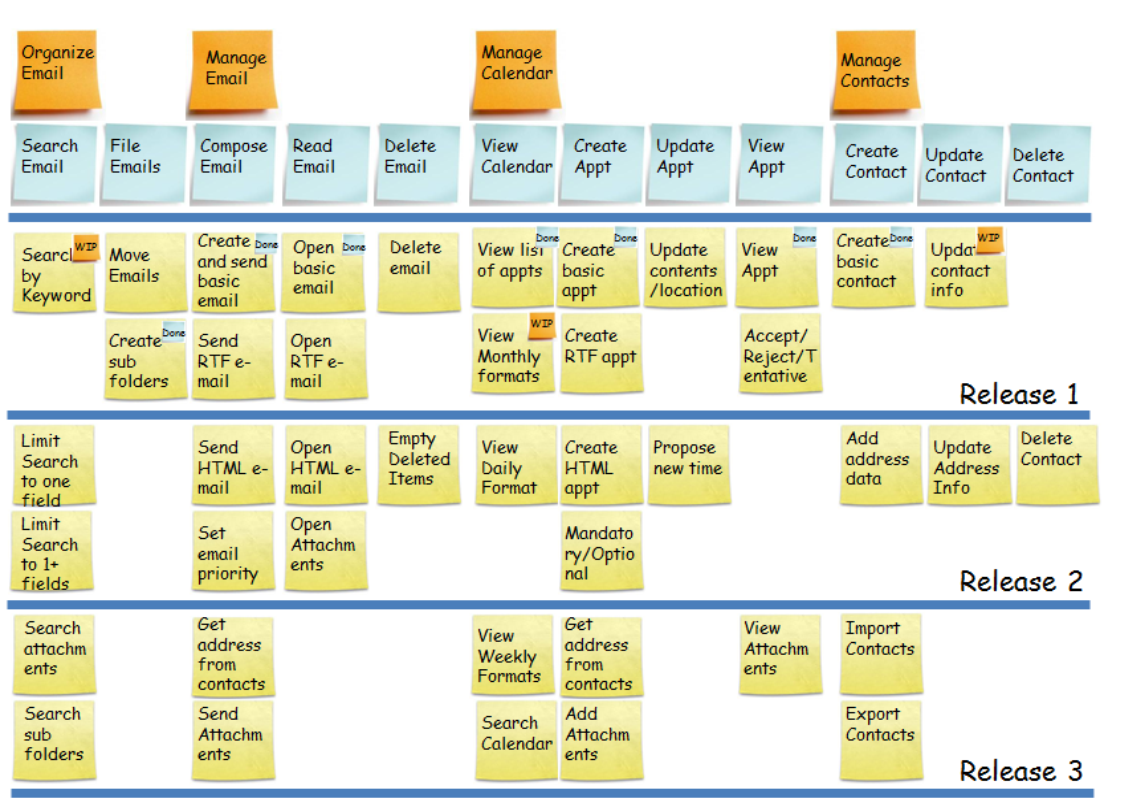
### Achtung: User Story nicht gleich User Case

**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 sind (sind es aber oft, bei Ungeübten). Use Cases bilden oft Geschäftsprozesse ab(sind darum auch grösser/länger). Zudem zeigen Sie die Aktoren und deren Motivation («Das will ich erreichen»). Des weiteren zeigen Sie einen Ablauf, der aus mehreren Operationen und Interkationen besteht und die Funktionen im Kontext inklusive der Reihenfolge.

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


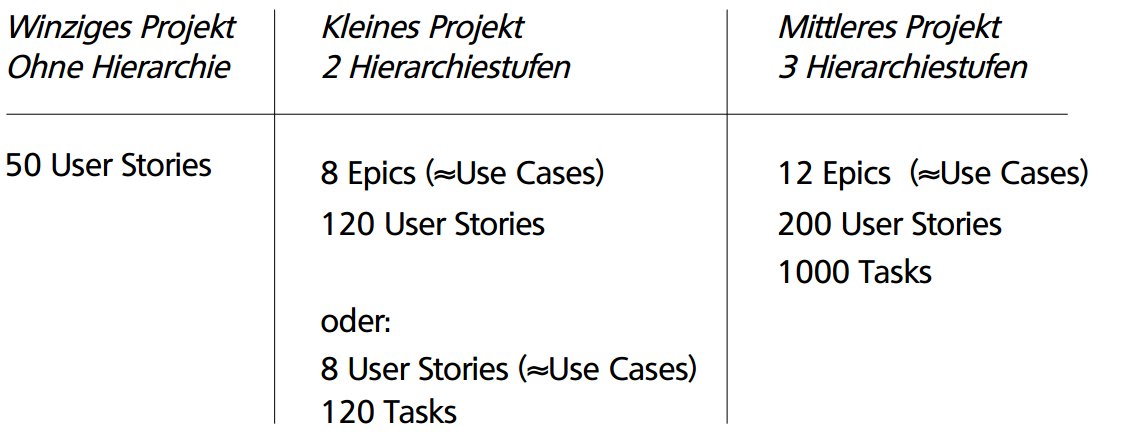
## Story Mapping



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.  


### 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.

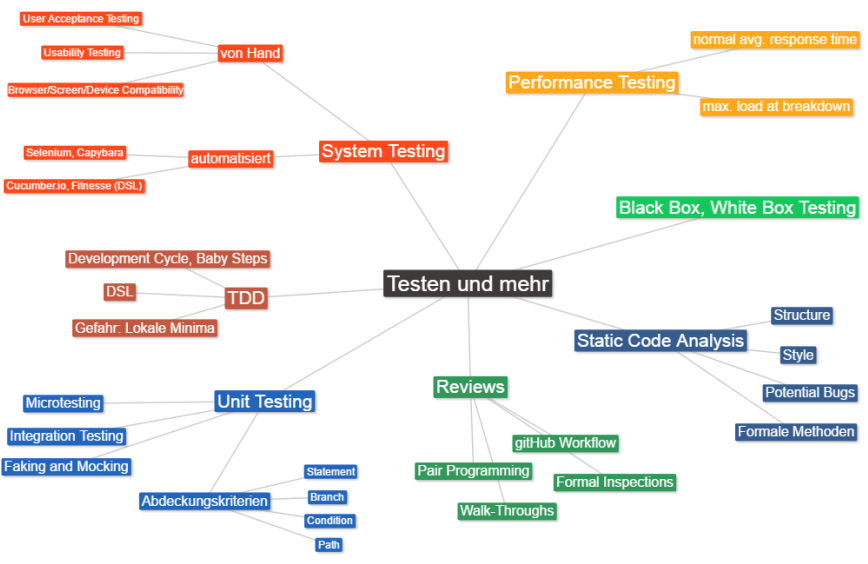
Dieses kleinste Element sollte in der durchschnittlichen Grösse etwa ¼ einer Sprintlänge für eine 100% arbeitende Person sein. Und da sich sehr viele Menschen auf Sprintlänge 2 oder 3 Wochen geeinigt haben, ist diese durchschnittliche Grösse sehr oft ähnlich:

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

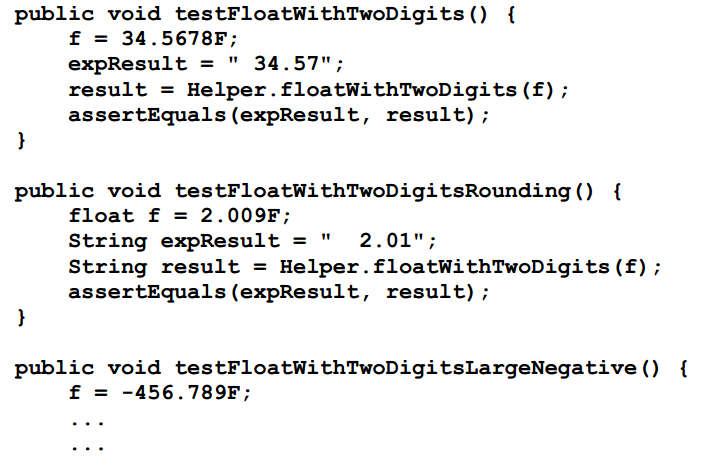
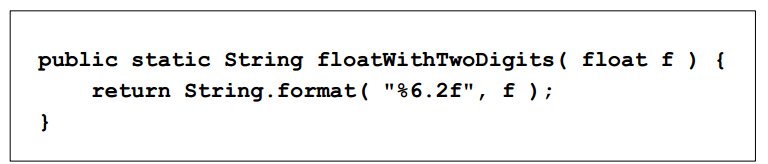
### Nutzen von Story Mapping

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!



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

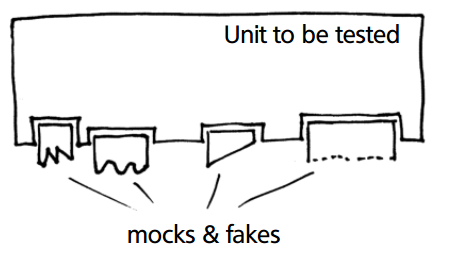
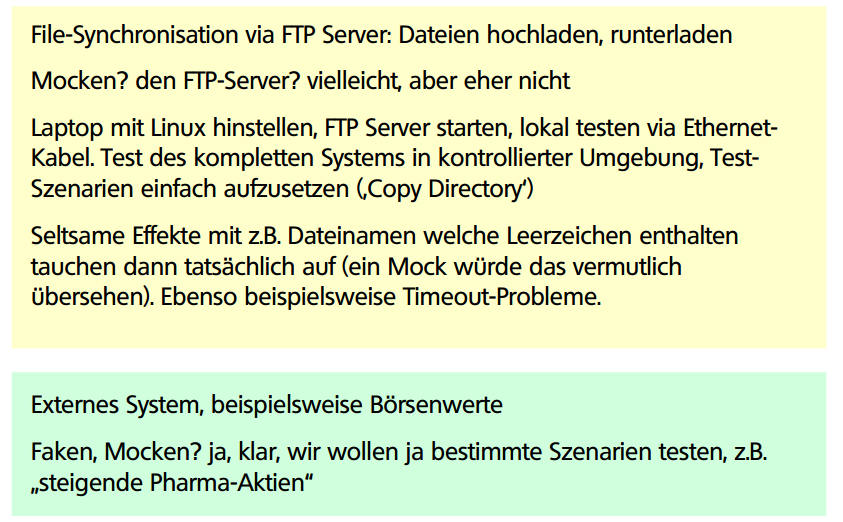
## Microtesting

Die Code Unit to be testet ist moistens 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.

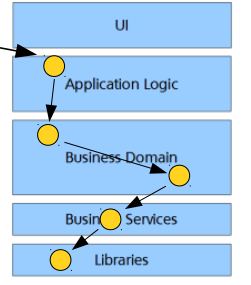
**Definition**  
Microtests sind Unit Tests, die eine Klasse (eine Methode einer Klasse) in Isolation testen. Generell ist es einfacher, die Units in den unteren Layern isoliert zu testen. Weiter oben wird Faking/Mocking eingesetzt, um das zu testende Teil wie gewünscht isolieren zu können.

### Facking & Mocking

Dieses Thema wurde bereits im Rahmen eines eLearing 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?**  


### 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, ...). ntegrationstests 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.

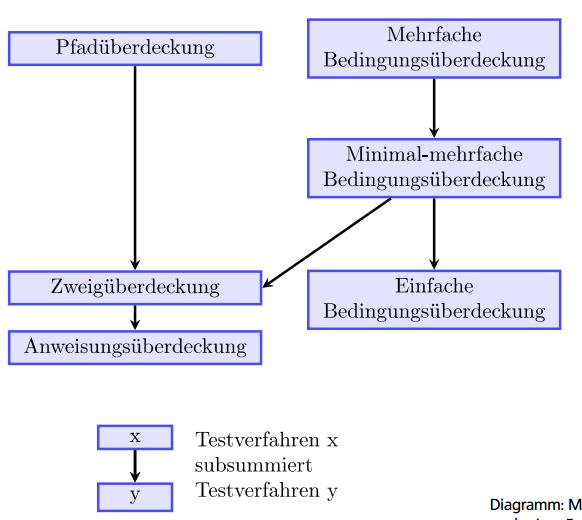
### Schwachstellen von Microtesting allgemein

* 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 gröberes Refactoring verlangen würde.
* Unit Tests überleben strukturell weitreichenderes (gröberes) 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.

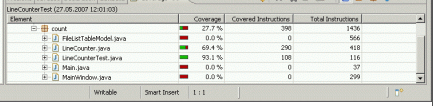
### Testabdeckung

 **Arten der Testabdeckung (leicht vereinfacht)**  
*Anweisungsabdeckung*  
(Statement Coverage)

*Zweigabdeckung*  
(BranchCoverage)

*Bedingungsabdeckung*  
(Decision Coverage)

*Pfadabdeckung*(Path Coverage)

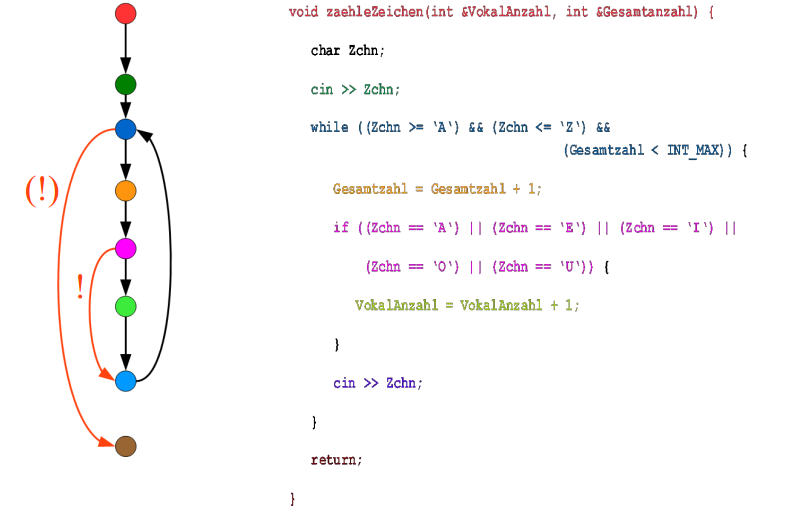
**So funktioniert die Messung mit der Testabdeckung**  
Der gesamte Code (Klassen + Unit Tests) wird mit Zählern auf jeder Anweisung (Zeile) instrumentiert.  


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**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"

#### Bedingungs- und Pfadabdeckung

**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 Pfaden in einem Modul."

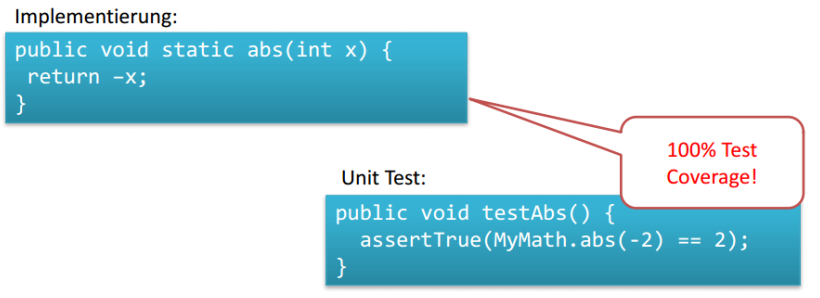
Das Kriterium der Bedingungsabdeckung kann sehr hilfreich sein, noch eitere Testfälle zu entdecken (s. auch Testdaten-Selektion mit Aequivalenzklassen).

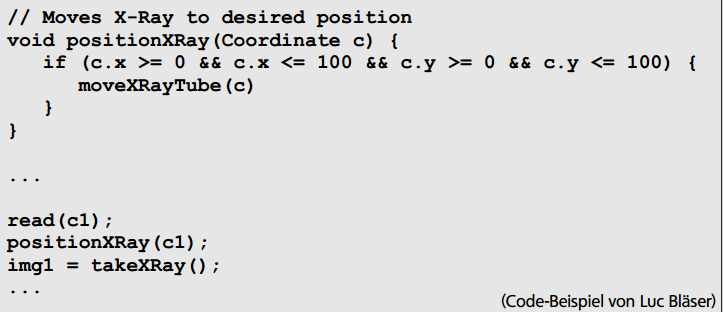
Pfadabdeckung führt sehr schnell zu einer potentiell unendlichen Anzahl von Fällen (wegen Schleifen).

### 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 fake 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.  


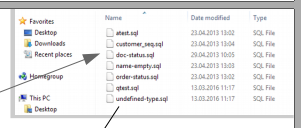
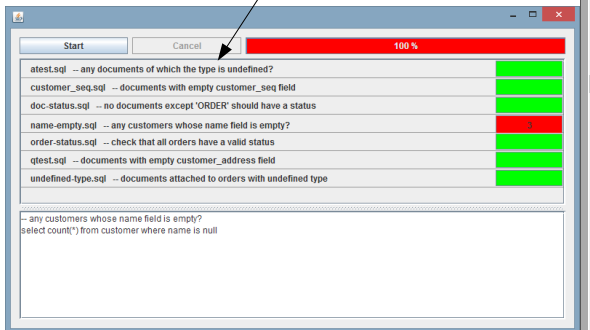
**Trügerische Sicherheit mit Microtests**  
Mit einem einzigen Testfall (z.B. c = (5,5)) erreicht man für positionXRay 100% Anweisungsabdeckung. Aber so entdeckt man den Fehler nicht. Der unentdeckte Fehler hatte ernsthafte gesundheitliche Konsequenzen für Bestrahlungspatienten.

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.

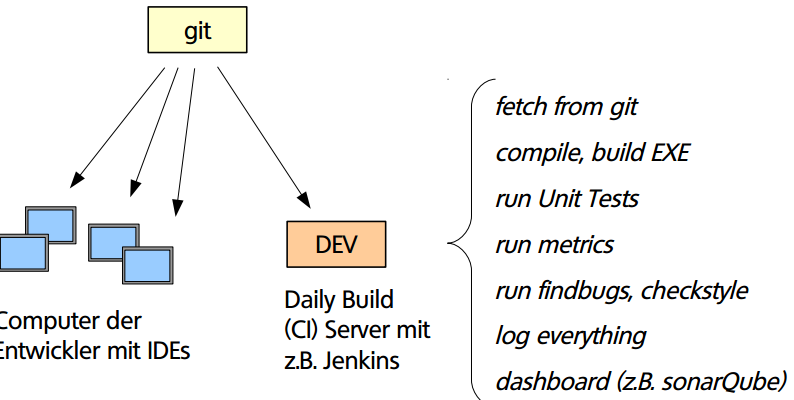
## Konsistenz-Tester für die Datenbank

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

|  |  |
| --- | --- |
|  | SQL Beispiele |

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)

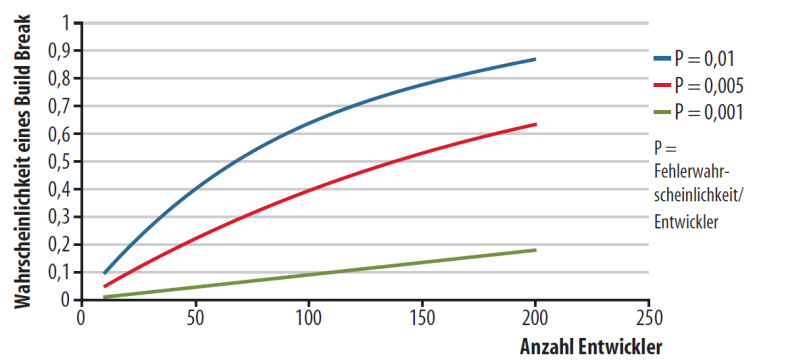
**Daily Build, Nightly Build**  
Einmal alle 24 Stunden (weil der Build und die Tests sehr lange dauern, z.B. 6 Stunden)

**“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 Continuous Integration mit allen schnellen/billigen Tests, nachts (oder alle drei Stunden) die vollen Tests.

### Wahrscheinlichkeit für einen Build Break



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

## Server Setup

### Server-Umgebungen

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

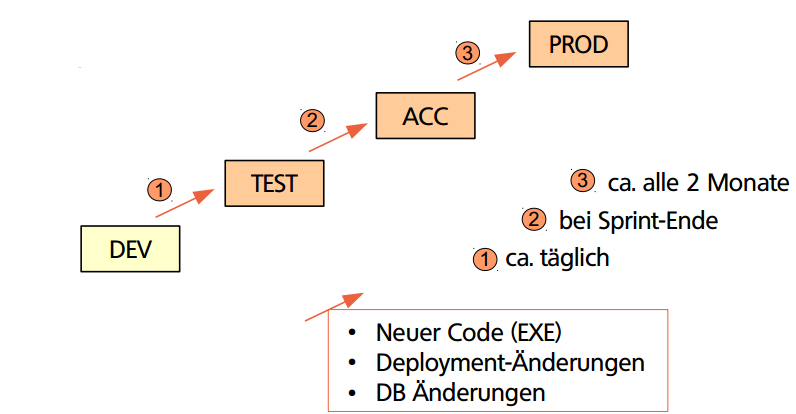
**ACC**  
Acceptance (manchmal auch STAGE ‚Staging‘ genannt): manuelles Testen durch Kunde/PO und externe Tester; HW & Testdaten möglichst gleich wie PROD (ausser bei Deployment- Änderungen); hier auch Performance-Tests

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

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

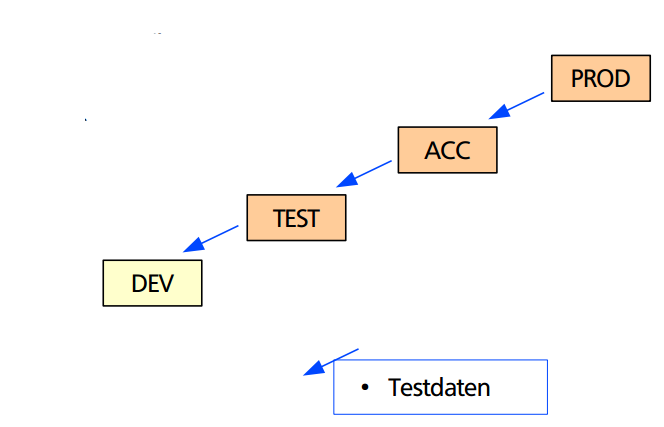
### Automatisierte Migration

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

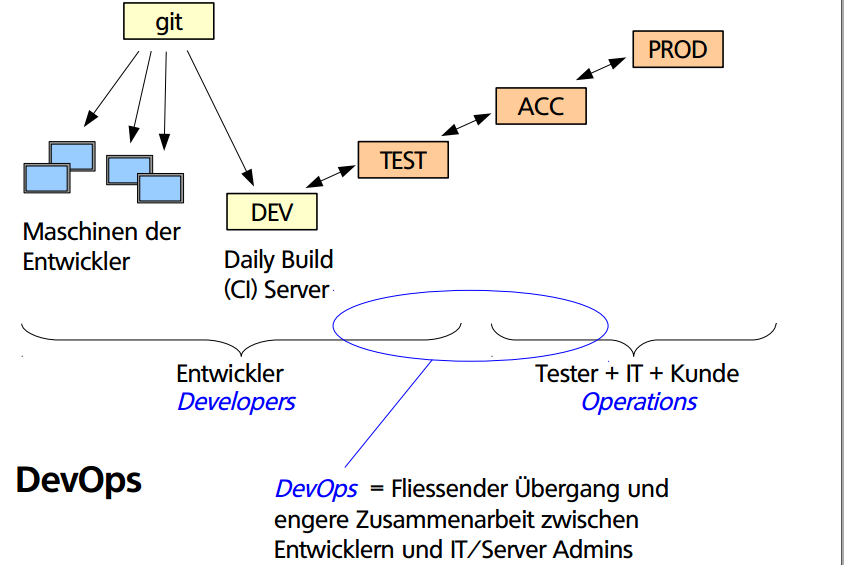


### Test-Daten

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.



### DevOps



## 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

* Mockarro.com
* Generatedata.com
* Und datasetgenerator.com möglich.

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

# Vorlesung Woche 8