



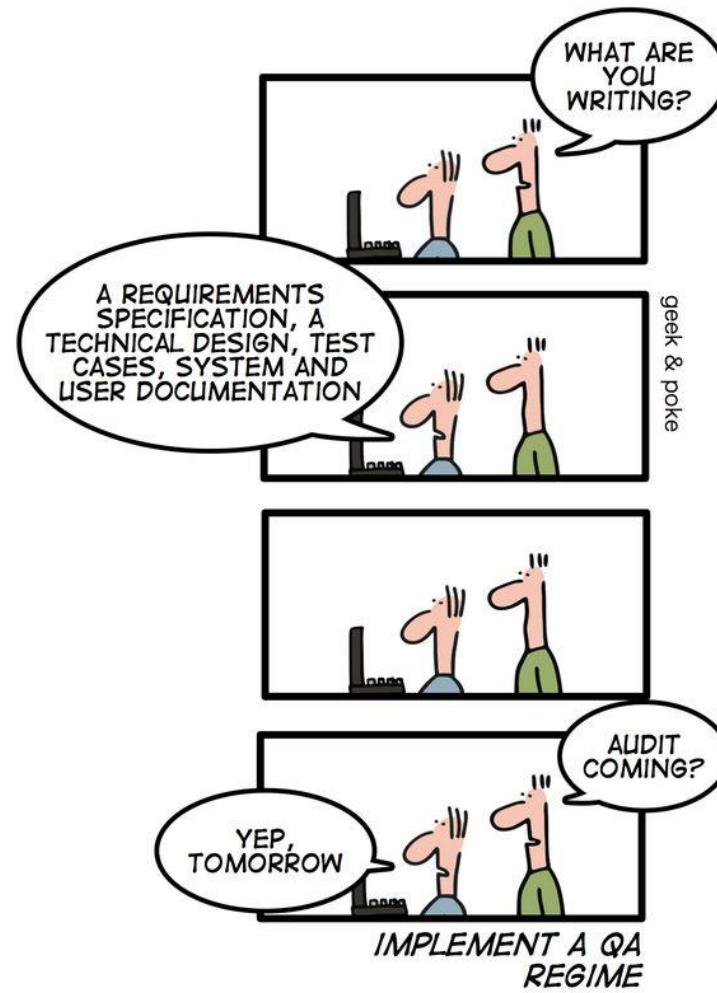
# Qualität

HAW Hamburg / Fachbereich Informatik

Tim Lüecke

([Tim.Lueecke@haw-hamburg.de](mailto:Tim.Lueecke@haw-hamburg.de))

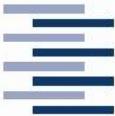
## HOW TO ENSURE QUALITY





*“Always code as if the guy/girl who ends up maintaining your code will be a violent psychopath who knows where you live.”*

(Martin Golding)



# Weshalb brauchen wir Qualität?

- Wir hören oftmals Aussagen wie
  - „*Wir produzieren mit Qualität*“
  - „*Bei uns ist alles qualitätsgesichert*“
- Was bedeutet aber Qualität?
  - muss jeder selbst entscheiden
  - Ansprüche ändern sich – persönlich und im Unternehmen
- Entscheidend ist ein gemeinsames Verständnis, was Qualität bedeutet und wie es erreicht werden kann, d.h. es gibt Normen / Best-practices



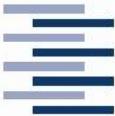


# Weshalb brauchen wir Qualität?

- Niemand kommt zur Arbeit und sagt „Heute mach ich aber mal einen richtig schlechten Job“
- Qualität ist das Salz in der Suppe
- Indizien für das Fehlen von Qualität
  - *Stress*: keine Zeit mehr, Überstunden
  - *Unzufriedenheit*: unrentable Projekte, unzufriedene Kunden, kein Spaß
  - *Unbrauchbarkeit*: Programm kann etwas nicht, Messer schneidet nicht
- Qualität ist ein tiefes menschliches Bedürfnis

***Qualität kann nicht verordnet werden,  
sie muss gelebt werden.***

***„Quality is not an act, it is a habit.“  
- Aristoteles***

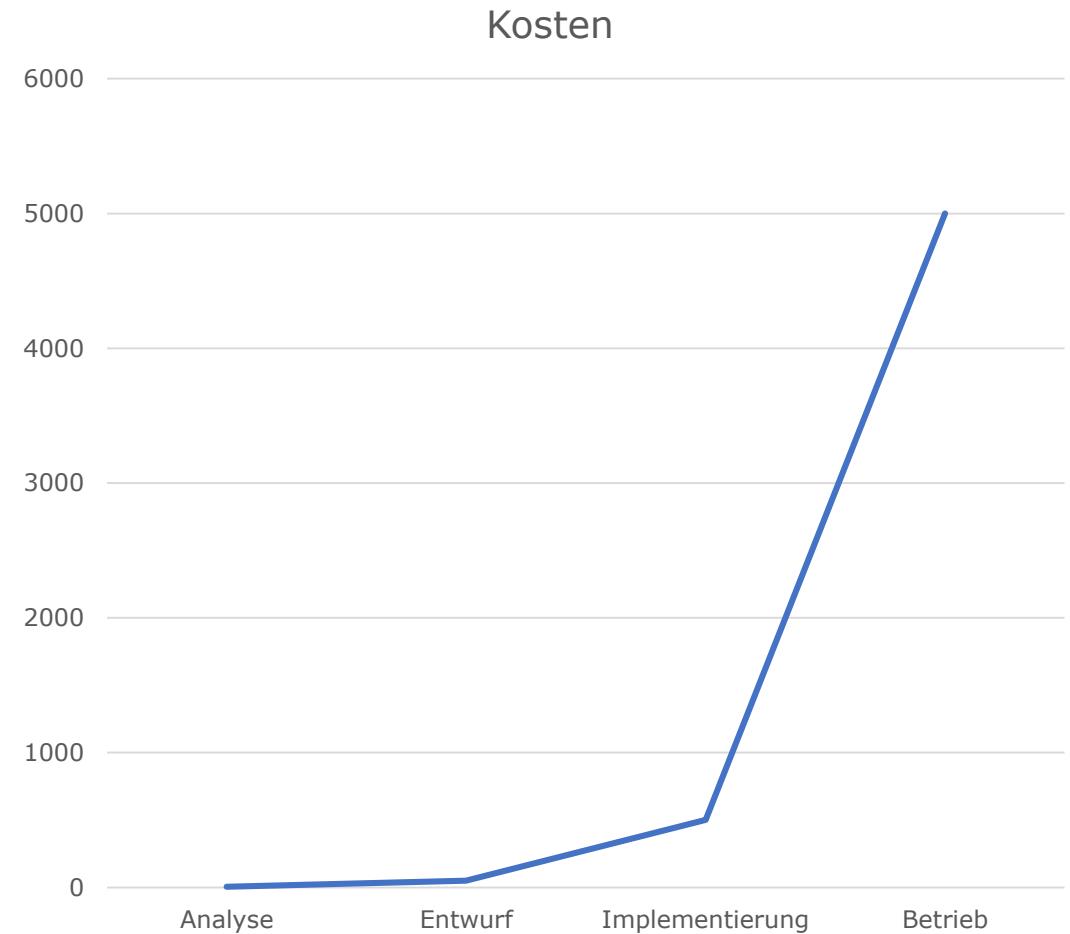


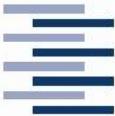
# Das Fehlen von Qualität kostet Geld

## Beispiel:

Adressverwaltung, das Feld „Adresszusatz“ fehlt

- Gefunden während Analyse  
Erweiterung Spezifikation, Feld einfügen, beschreiben, neue Version erstellen, 5 €
- Gefunden während Entwurf  
+ Entwurf von Komponenten, Klassen, Methoden, Persistenz, Prozessschritte anpassen, 50 €
- Gefunden während Implementierung  
+ bereits getestete Module ändern, neu testen, Testspezifikation erweitern, Designidee überarbeiten, Dokumentation, 500 €
- Gefunden während Betrieb  
Überraschung „Adresszusatz?“ Prozess neu, Daten neu importieren (5000 Datensätze waren aber schon manuell nachgearbeitet), System muss parallel u. U. trotzdem betrieben werden, Excel mit Kontaktdaten separat währenddessen pflegen, wann erfolgt die Migration? Unterschiedliche Zeitzonen? 5000 €

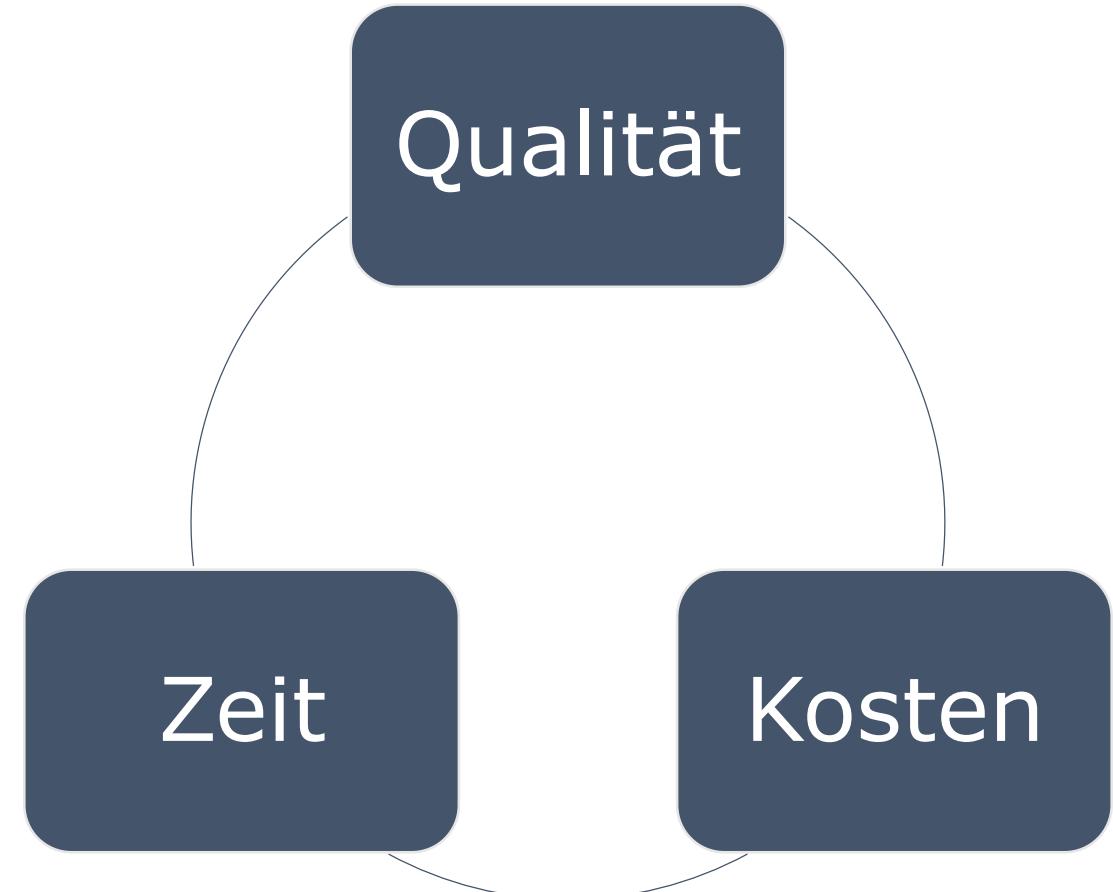




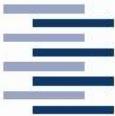
# Verschiedenen Kostenarten für fehlende Qualität

- **Direkte Kosten:** Diskussionen über Mängel, Überarbeitungen
- **Versteckte Kosten:** schwer zu erkennen: fehlende Nachvollziehbarkeit durch fehlende Dokumentation, unsinnigen Dateinamen. Diese Kosten werden fast nie dem Verursacher zugeordnet, machen aber einen Großteil der Gesamtkosten aus!

**Der Projektleiter muss dem Kunden glaubhaft machen, dass Qualität erzeugt wird, um nachträgliche direkte und versteckte Kosten zu vermeiden**



*Das magische Dreieck im Projektmanagement*



In kleinen Projekten ist schlechte Qualität problematisch;  
in großen Projekten ist sie ein Fiasko.

### **Beispiel Projekt BA-ALGII:**

Zu spät erkannte Fehler führten dort zu:

- verspäteten Zahlungen an bis zu drei Millionen Kunden
- Überzahlungen im mehrstelligen Millionenbereich
- Rechtstreits mit Betroffenen
- Chaos und Mehrarbeit in den Arbeitsagenturen
- internem Image-Verlust
- externem Image-Verlust der BA, des IT-Dienstleisters und führender Politiker
- enormen Mehrkosten bei der BA und dem IT-Dienstleister

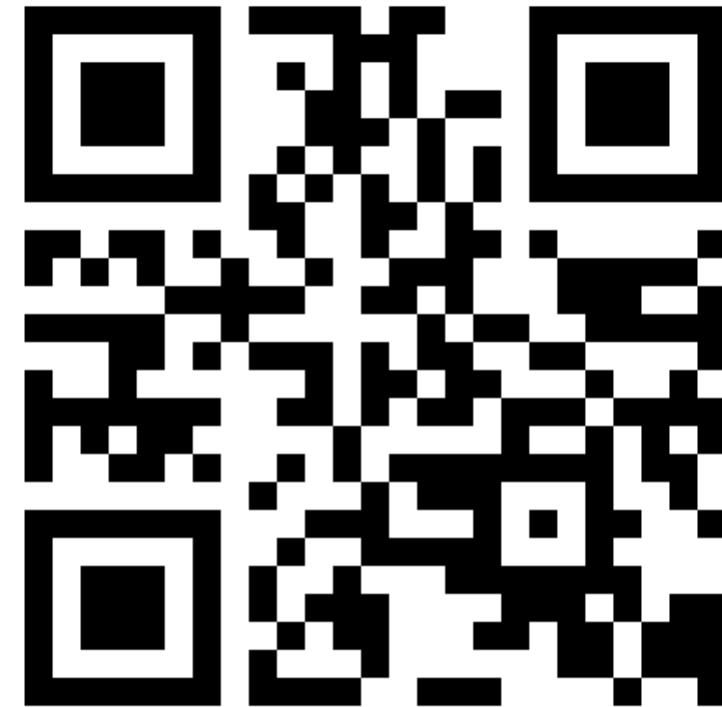


**Aufgrund der großen, absoluten Zahlen treten viele Qualitätsprobleme klarer zu Tage als in kleineren Projekten.**



# Was braucht man um Qualität sicherzustellen?

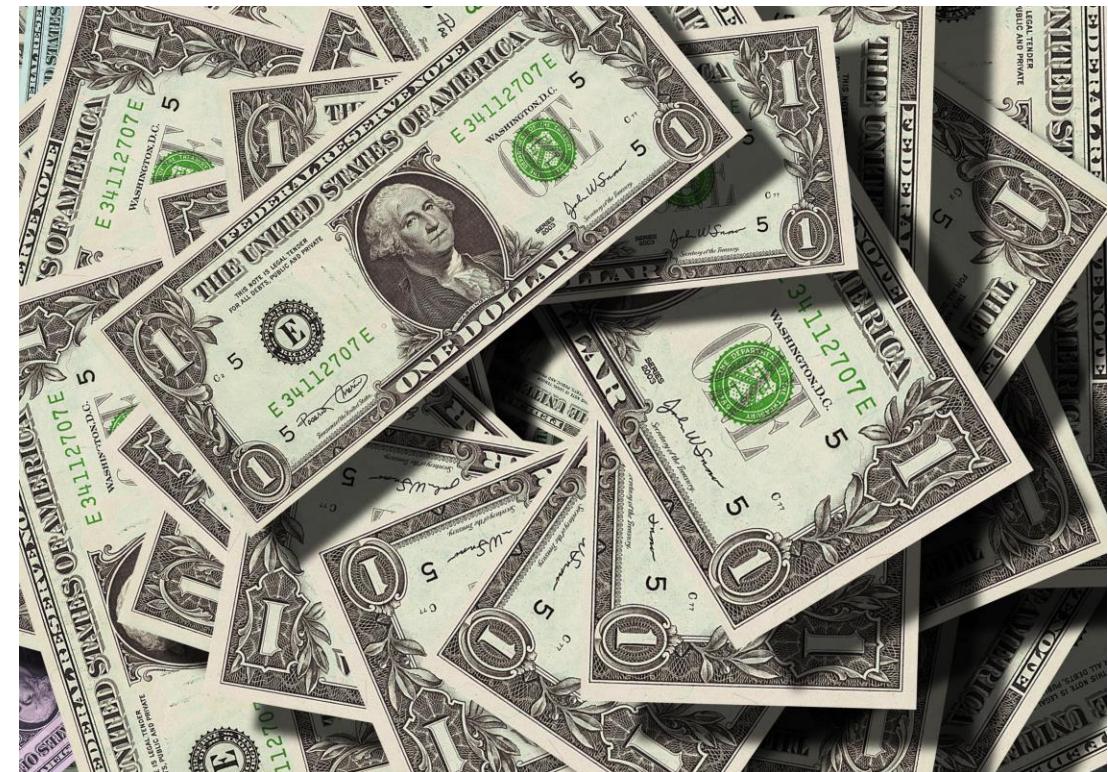
<http://pingo.upb.de/764286>





# Qualität zu Erreichen kostet Geld

- Angemessene Software-Architektur
- Angemessene Code-Kommentierung
- Angemessene Methodik  
(SCRUM, Kanban, V-Modell, ...)
- Tools für Source Code-Verwaltung,  
Konfigurationsmanagement, Fehlerverfolgung
- Systematisches Testen
- Continuous Integration
- Reviews auf Code und Dokumente  
(Spezifikation, Entwurf, Tests, ...)
- Dokumentation (Systemdokumentation,  
Anleitung für Installation, Schulungs-  
unterlagen, Word vs. Wiki, ...)
- Qualität braucht unternehmensweite Führung  
– so wie das Unternehmen einen  
Geschäftsführer braucht





# Qualität - Definition

## **Softwarequalität (DIN ISO 9126)**

*Softwarequalität ist die Gesamtheit der Merkmale eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte und vorausgesetzte Erfordernisse zu erfüllen.*

-> Konkretisierung nötig!



# Qualität - Roadmap

Wichtigste Fragen im Folgenden:

- Was ist Qualität und wie stellt sie sich dar? (**Qualitäts-Merkmale**)
- Wie kann man Qualität messen und vergleichen? (**Metriken**)
- Welche **Maßnahmen** dienen der Hebung der Qualität?





# Agenda

- Einführung

- Qualitäts-Merkmale

- Metriken
- Maßnahmen
- Zusammenfassung



# Qualität – Nicht-funktionale und funktionale Anforderungen

## Funktionale Anforderungen

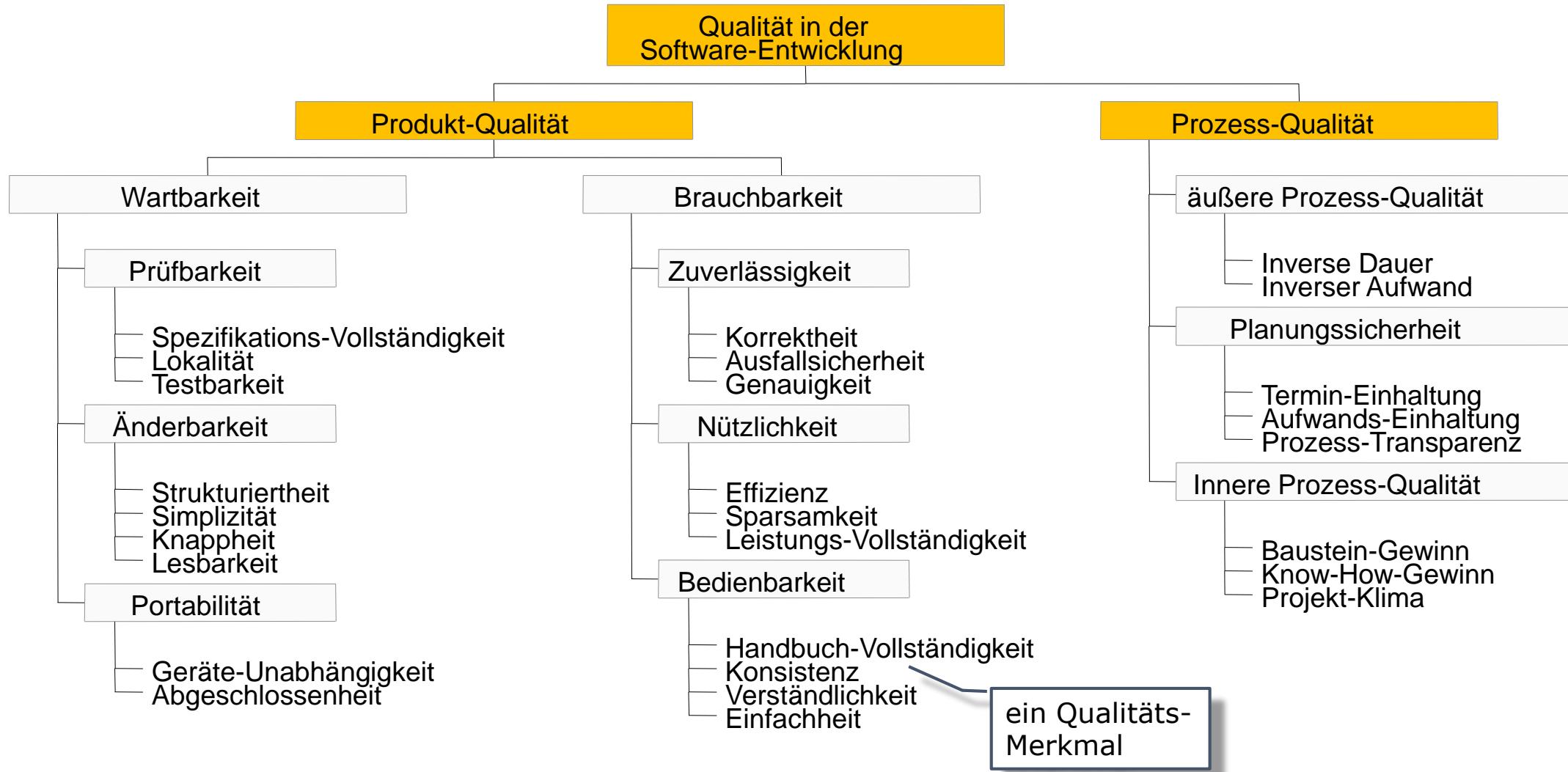
- Definieren die Funktionsweise des Systems
- Spezifizieren ganz genau welche Ausgabe bei welcher Eingabe erzeugt
- Kann auf beliebige Art und Weise sichergestellt werden (z.B. auch durch Spaghetti-Code oder eine einzige Klasse)

## Nicht-Funktionale Anforderungen

- Beschreiben wie das System betrieben werden können muss
- Definieren wie das System gewartet werden können muss
- Oftmals wesentlich schwieriger zu definieren

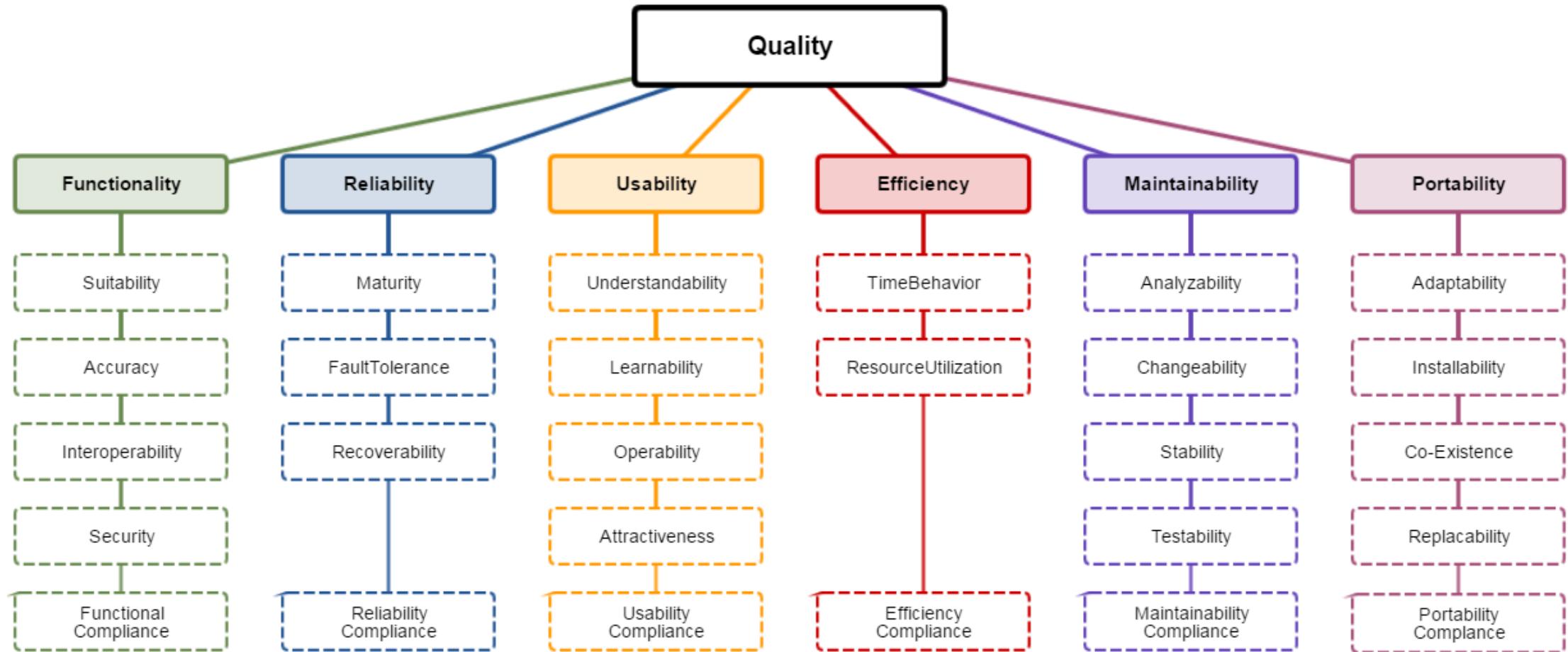


# Qualität – Merkmale nach Boehm



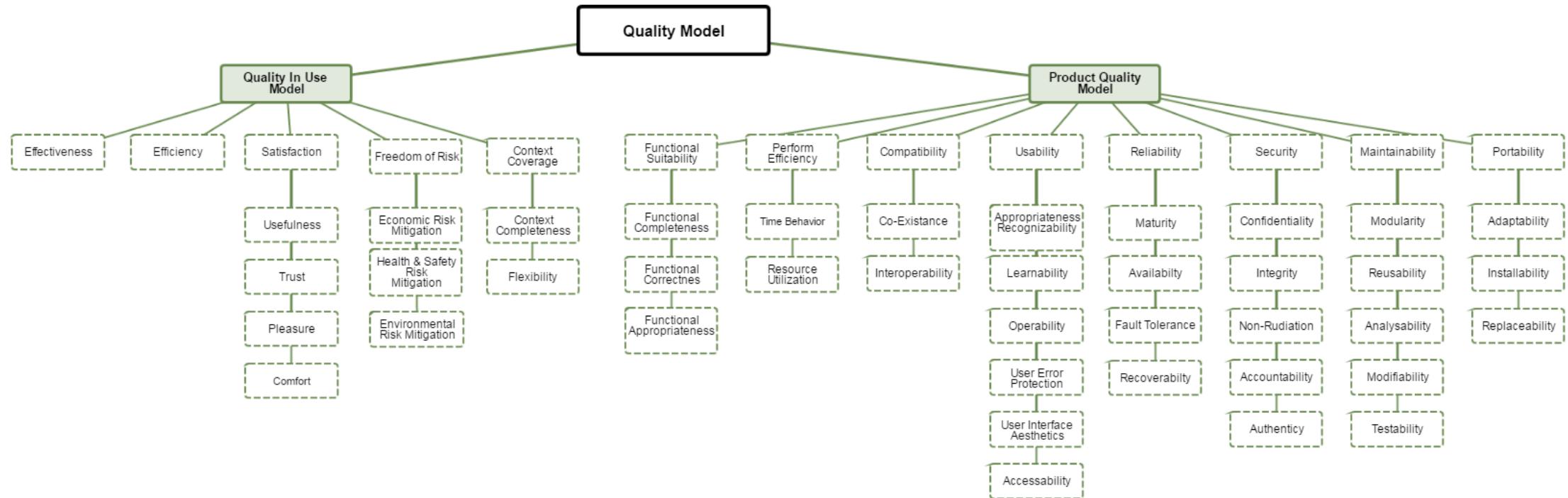


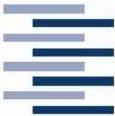
# Qualität – Merkmale nach ISO/IEC 9126





# Qualität – Merkmale nach ISO/IEC 25010





# Qualität – Liste von „ilities“

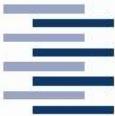
[accessibility](#)  
[accountability](#)  
[accuracy](#)  
[adaptability](#)  
[administrability](#)  
[affordability](#)  
[agility](#)  
[auditability](#)  
[autonomy](#)  
[availability](#)  
[compatibility](#)  
[composability](#)  
[configurability](#)  
[correctness](#)  
[credibility](#)  
[customizability](#)  
[debugability](#)

[degradability](#)  
[determinability](#)  
[demonstrability](#)  
[dependability](#)  
[deployability](#)  
[discoverability](#)  
[distributability](#)  
[durability](#)  
[effectiveness](#)  
[efficiency](#)  
[evolvability](#)  
[extensibility](#)  
[failure](#)  
[transparency](#)  
[fault-tolerance](#)  
[fidelity](#)  
[flexibility](#)  
[inspectability](#)

[installability](#)  
[integrity](#)  
[interchangeability](#)  
[interoperability](#)  
[learnability](#)  
[maintainability](#)  
[manageability](#)  
[mobility](#)  
[modifiability](#)  
[modularity](#)  
[operability](#)  
[orthogonality](#)  
[portability](#)  
[precision](#)  
[predictability](#)  
[process](#)  
[capabilities](#)  
[producibility](#)

[provability](#)  
[recoverability](#)  
[relevance](#)  
[reliability](#)  
[repeatability](#)  
[reproducibility](#)  
[resilience](#)  
[responsiveness](#)  
[reusability](#)  
[robustness](#)  
[safety](#)  
[scalability](#)  
[seamlessness](#)  
[self-](#)  
[sustainability](#)  
[Serviceability](#)  
[securability](#)  
[simplicity](#)

[stability](#)  
[standards](#)  
[compliance](#)  
[survivability](#)  
[sustainability](#)  
[tailorability](#)  
[testability](#)  
[timeliness](#)  
[traceability](#)  
[timeliness](#)  
[traceability](#)  
[transparency](#)  
[ubiquity](#)  
[understandability](#)  
[upgradability](#)  
[usability](#)



# Merkmale unterscheiden sich von Projekt zu Projekt

- Jedes bekannte Qualitätsmerkmal zu überwachen wird schwierig
- Abhängig vom Projekt sind die Prioritäten unterschiedlich
  - Für eine Übergangslösung ist Wartbarkeit kein notwendiges Merkmal
  - Für eine Batch-Anwendung, die einmal im Monat läuft ist Effizienz nicht so wichtig...
  - ...
- Stattdessen macht die Erstellung eines eigenen, maßgeschneiderten Merkmal-Baums für das jeweilige Projekt Sinn  
(s. auch [Carnegie Mellon University SEI Quality Attribute Workshop](#))





# Übungsaufgabe

⌚ **Ziel:** Erstellen Sie Ihren eigenen Merkmal-Baum für Ihr SE2-Projekt

👤 SE2-Projektgruppen

⌚ 20 Minuten





# Agenda

- Einführung
- Qualitäts-Merkmale
- **Metriken**
- Maßnahmen
- Zusammenfassung



# Beispiel: „Messung“ der Zuverlässigkeit

- Erst mal sehr schwierig
- Untermerkmale (nach Boehm): **Korrektheit, Ausfallsicherheit**

## Korrektheit messen?

- {korrekt, falsch} entscheidbar?
- Beweisen: setzt Formale Spezifikation voraus, nur Korrektheit in Bezug auf eine („korrekte?“) Spezifikation, ...
- aber es gibt sinnvolle Teststrategien...
- Jedoch Testen: Stichprobe, vollständige Aufzählung aller Testfälle ist nicht möglich

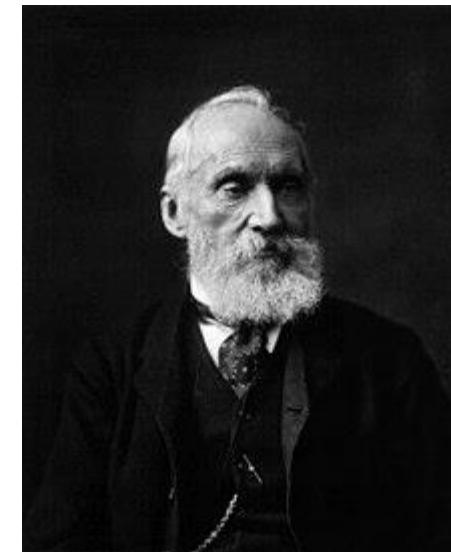
## Ausfallsicherheit messen?

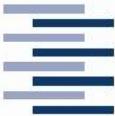
- Hardware: MTBF (Mean Time Between Failures)
- Software: ?
  - Keine statistischen Fehler, sondern systematische
  - „Ein Software-Fehler ist eine Katastrophe, die darauf wartet, zu passieren“
  - kein akzeptables Wahrscheinlichkeitsmodell



*„To measure is to know.“*

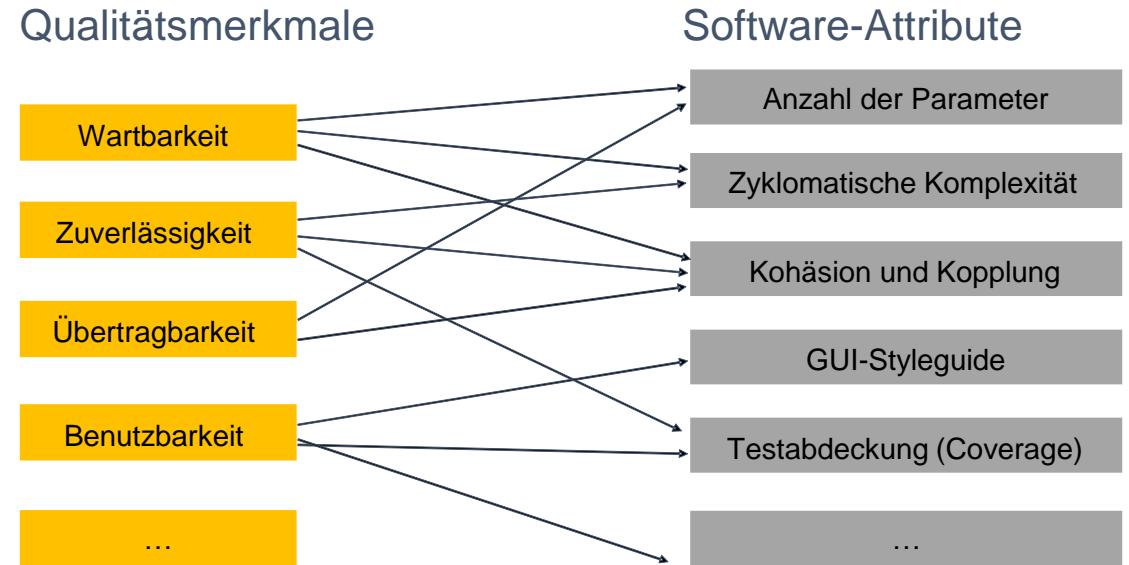
*Sir W. Thomson, 1824-1907*





# Qualität – Merkmale Messen

- Definition Softwaremaß – Metrik
  - Funktion, die Attribute von Elementen der Software-Entwicklung in eine (geordnete) Basismenge (z.B. reelle Zahlen) abbildet
  - In der Literatur > 1000 unterschiedliche Metriken für die verschiedensten Aspekte der Software-Entwicklung
- Beispiel:
  - Wie stellen sich diese Qualitätsmerkmale dar?
  - Wie können sie z. B. gemessen werden?





# Metriken – Nutzen und Schaden

- Wichtigste **Motive** und Ziele für den Einsatz
  - Bewerten der Qualität von Produkten und Prozessen
  - Prognosen (Kosten-, Termin- und Qualitätsprognosen)
  - Unterstützen von Entscheidungen
- Unterscheidung nach Anwendungsbereich
- **Gefahr:** Fehlinterpretation, die zu falschen Schlüssen und Aussagen führen

**Deshalb: Wenige, wichtige Metriken konsequent erheben und verwenden**



# Anforderungen an Metriken

- Metriken liefern keine Antworten im Sinne von „Ja“ oder „Nein“, sondern geben nur **Anhaltspunkte** und Informationen!
- Bei der Entwicklung und Auswahl von Metriken geht es darum, diese Entscheidungen **möglichst einfach** zu unterstützen

**Eine ideale Metrik liefert uns ein exaktes Bild eines Probanden (eines Systems, Projekts, Prozesses), reduziert auf den uns wichtigen Aspekt und in diesem Maße verdichtet.**



# Wichtige Eigenschaften von Metriken

Unterschiede des Probanden, die auf andere Weise erkennbar werden, sollen sich auch in der Metrik spiegeln

Merkmal	positives Beispiel	negatives Beispiel
differenziert	LOC	CMMI für Organisationen, die nicht Stufe 2 erreichen
vergleichbar	zyklomatische Komplexität	Textuelles Gutachten
reproduzierbar	Speicherbedarf	Prognose der Projektdauer
verfügbar	Zahl der Entwickler	Zahl der Fehler im Code
relevant	zu erwartende Entwicklungskosten	Zahl der Subklassen
rentabel	Zahl der entdeckten Fehler im Code	sehr detaillierte Arbeitszeiterfassung
plausibel	Schätzung nach Story Points	zyklomatische Komplexität eines Programms mit Zeigeroperationen

hohe Korrelation zwischen Bewertung und Beobachtung



# Metriken – Beispiele Entwurf/Codierung

## ▪ Größenmaße

- LOC (lines of code)
  - mit/ohne Kommentare? nicht leer?
  - ausgeliefert? Sprachabhängigkeit?
- Halstead-Metrik für Größe und Umfang

## ▪ Komplexitätsmaße

- Zyklomatische Komplexität (McCabe-Metrik)
- Halstead-Metrik für Komplexität
- Kopplung einer Klasse
  - Anzahl der „verbundenen“ Klassen (= Klassen, deren Operationen oder Instanzvariablen benutzt werden)
  - Anzahl der Operationen (anderer Klassen), die von Objekten dieser Klasse aufgerufen werden können

## ▪ Stil-Metriken

- Durchschnittliche Länge von Bezeichnern
- Anteil Kommentar-Zeilen oder Leerzeilen
- Anzahl von Sprüngen

## Game of Life in one line of APL

```
life←{ ⌈ John Conway's "Game of Life".  
      ↑1 ω V.∧3 4=+/,~1 0 1◦.⊖~1 0 1◦.① ⊂ ω  
      ⌋ Expression for next generation.  
 }
```

LOC is really a bad metric

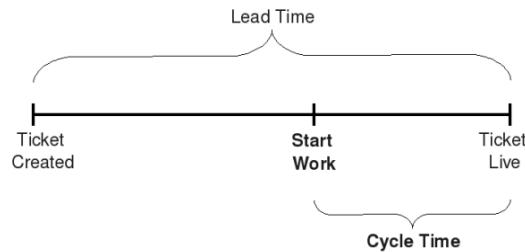
dyalog.com

Eberhard Wolff - @ewolff



# Metriken – Beispiele Controlling

- SCRUM: Story Points
- Kanban: Lead Time und Cycle Time



<https://stefanroock.wordpress.com/2010/03/02/kanban-definition-of-lead-time-and-cycle-time/>

- Function Points / Use-Case-Points
- Technical Debt (Technische Schuld)
- ...





# Arten von Metriken

objektive Metrik	
Verfahren	Messung, Zählung, evtl. auch Normierung
Vorteile	exakt, reproduzierbar, automatisierbar
Nachteile	nicht sicher relevant, meist unterlaufbar, keine Interpretation
Beispiele allgemein	Körpergröße, Luftdruck
Beispiele im Software Engineering	LOC („lines of code“), Zahl der gefundenen Fehler, Cycle Time
Typisch angewendet für	Sammlung einfacher Basismetriken



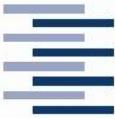
# Arten von Metriken

	<b>objektive Metrik</b>	<b>subjektive Metrik</b>
Verfahren	Messung, Zählung, evtl. auch Normierung	Beurteilung durch Gutachter
Vorteile	exakt, reproduzierbar, automatisierbar	nicht unterlauffbar, plausible Resultate, auch für komplexe Merkmale
Nachteile	nicht sicher relevant, meist unterlauffbar, keine Interpretation	Erhebung aufwändig, Qualität der Aussagen hängt stark vom Gutachter ab
Beispiele allgemein	Körpergröße, Luftdruck	Gesundheitszustand, Wetterlage
Beispiele im Software Engineering	LOC („lines of code“), Zahl der gefundenen Fehler, Cycle Time	Benutzerfreundlichkeit
Typisch angewendet für	Sammlung einfacher Basismetriken	Qualitätsbewertung, Fehlergewichtung



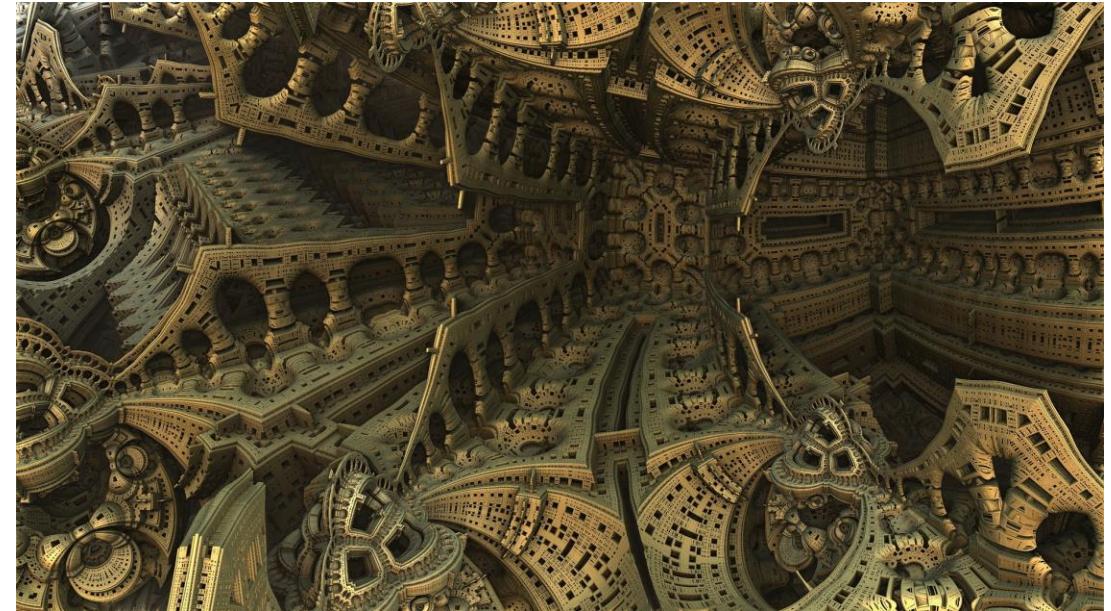
# Arten von Metriken

	<b>objektive Metrik</b>	<b>subjektive Metrik</b>	<b>Pseudometrik</b>
Verfahren	Messung, Zählung, evtl. auch Normierung	Beurteilung durch Gutachter	Berechnung (auf Basis von Messungen und/oder Beurteilungen)
Vorteile	exakt, reproduzierbar, automatisierbar	nicht unterlauffbar, plausible Resultate, auch für komplexe Merkmale	liefert relevante Aussagen über nicht sichtbares Merkmal
Nachteile	nicht sicher relevant, meist unterlauffbar, keine Interpretation	Erhebung aufwändig, Qualität der Aussagen hängt stark vom Gutachter ab	schwer nachvollziehbar, pseudoobjektiv
Beispiele allgemein	Körpergröße, Luftdruck	Gesundheitszustand, Wetterlage	BMI (Body Mass Index), Wettervorhersage
Beispiele im Software Engineering	LOC („lines of code“), Zahl der gefundenen Fehler, Cycle Time	Benutzerfreundlichkeit	Produktivität, Kosten nach CoCoMo
Typisch angewendet für	Sammlung einfacher Basismetriken	Qualitätsbewertung, Fehlergewichtung	Prognosen (Kostenschätzung), Bewertungen



# Metrik für die Komplexität eines Programms?

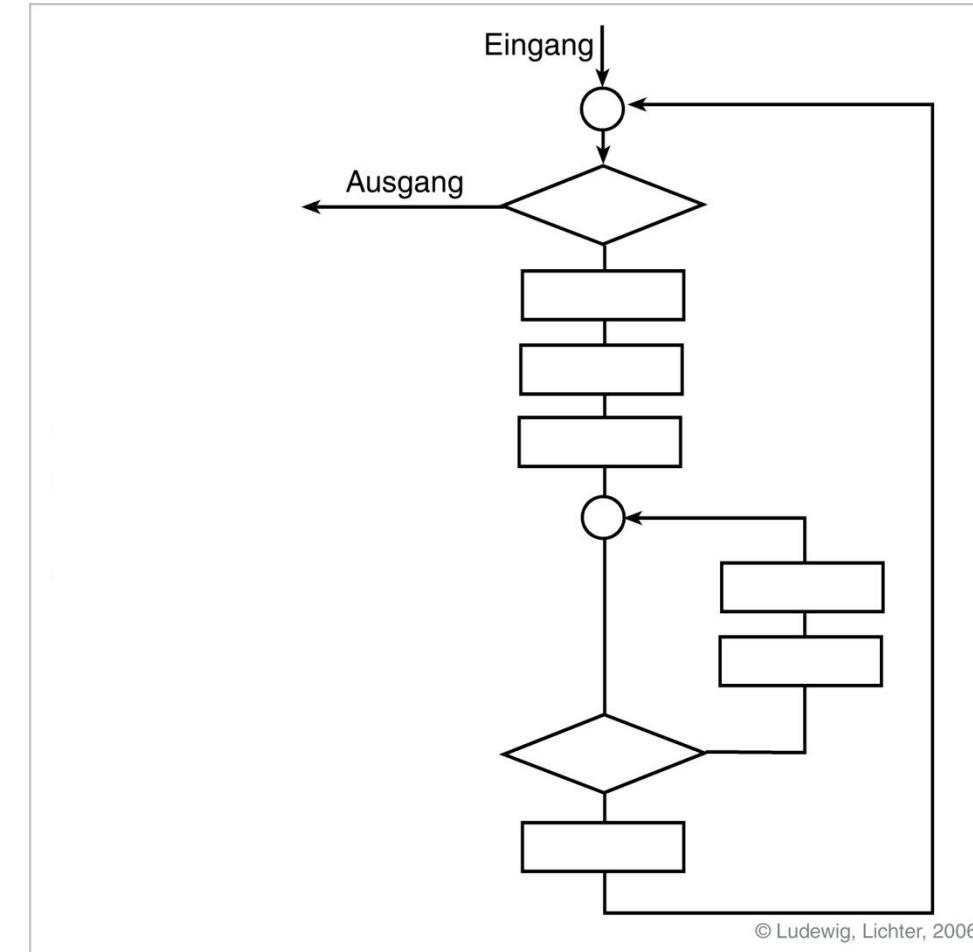
- **Komplexität** eines Programms  
(Funktion, Methode, Codestücks, ...) sollte im Sinne der Wartbarkeit so gering wie möglich sein
- Wie messen?

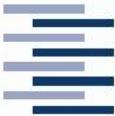




# Zyklomatische Komplexität – McCabe-Metrik

- Programmkomplexität wird gering bewertet, wenn man es leicht verstehen und prüfen kann
  - Länge des Programms hat Einfluss, ist aber keineswegs entscheidend!
  - z.B.: einfache Sequenz von Anweisungen vs. komplexe Kontrollstrukturen
- **Grundidee:** Komplexität eines Programms hängt von Zahl der Verzweigungen im Programmablauf ab
  - Maß für die Anzahl linear unabhängiger Ausführungspfade
  - Wird aus dem Kontrollflussgraphen eines Moduls abgeleitet





# Zyklomatische Komplexität – McCabe-Metrik

$$v(G) = e - n + p$$

G = Kontrollflussgraph

e = Anzahl der Kanten

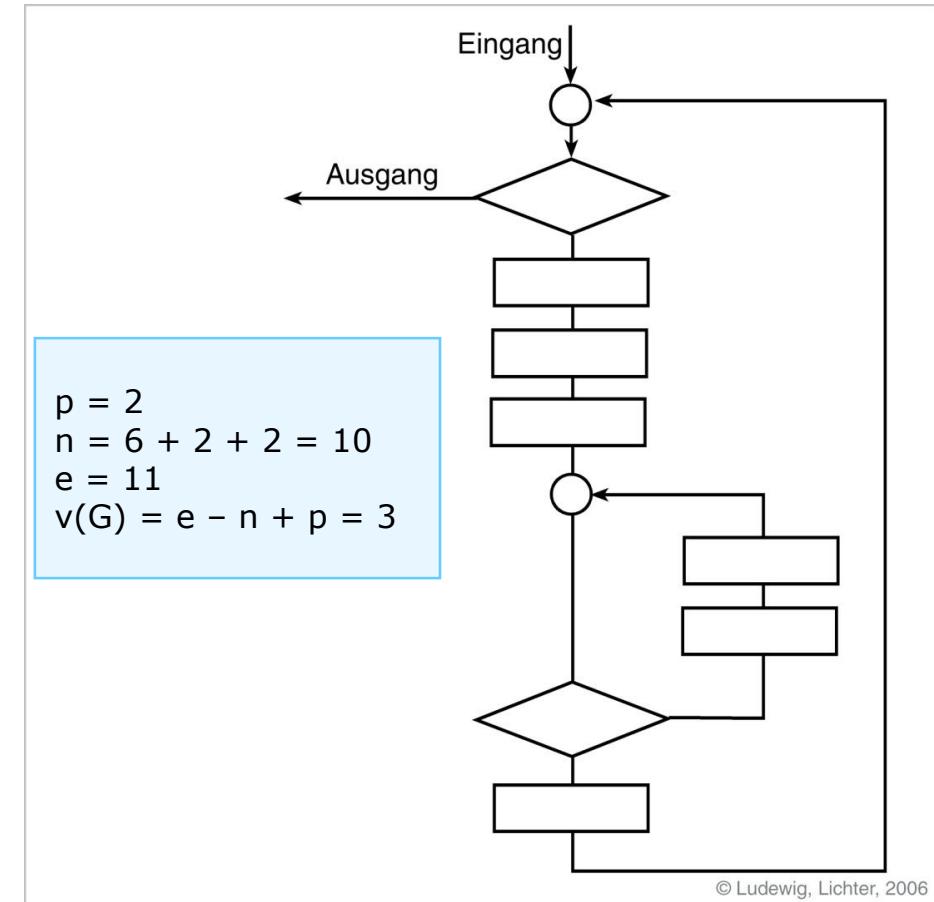
n = Anzahl der Knoten

p = Anzahl der Außenverbindungen

## Beispiel 1:

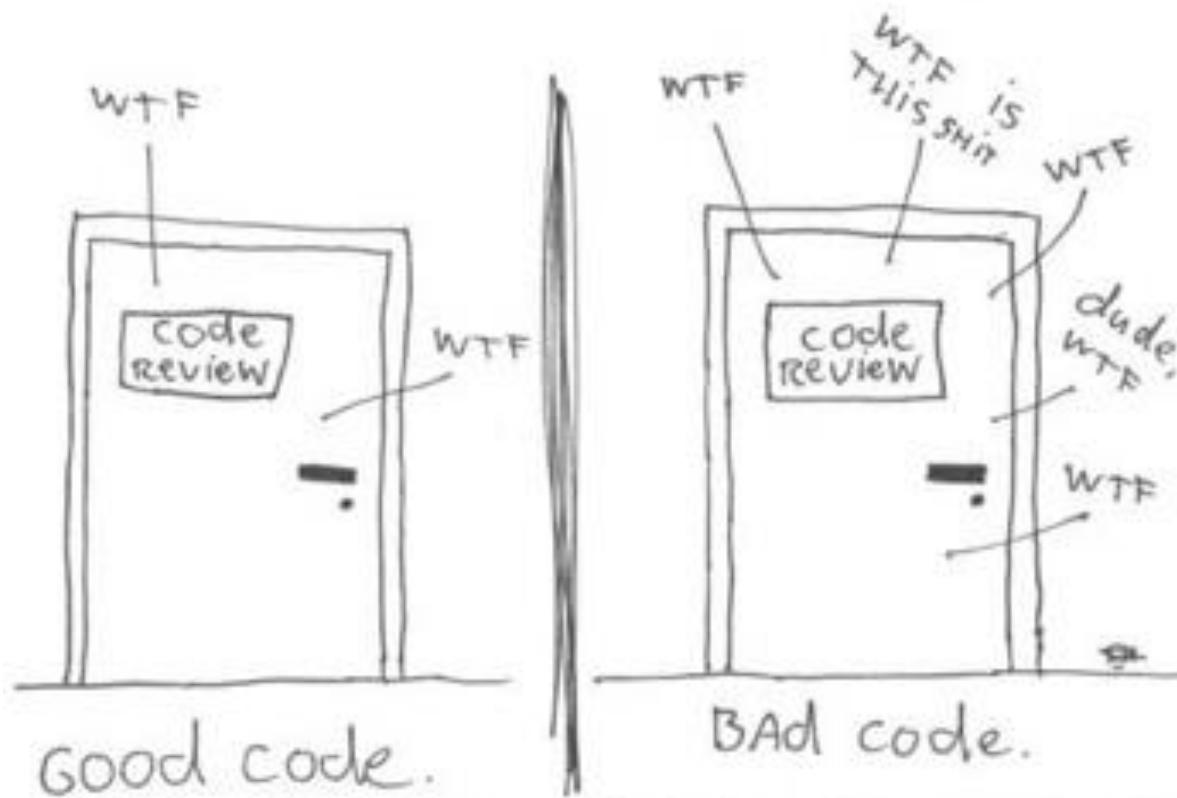
- Programm mit einem Ein- und Ausgang:  
 $p=2$   
Komplexität ist dann minimal:  
 $v(G)=0-1+2=1$
- Lineare Verkettung (Sequenz) hat damit die Komplexität „1“
- jede Verzweigung oder Schleife erhöht die Komplexität um 1

## Beispiel 2





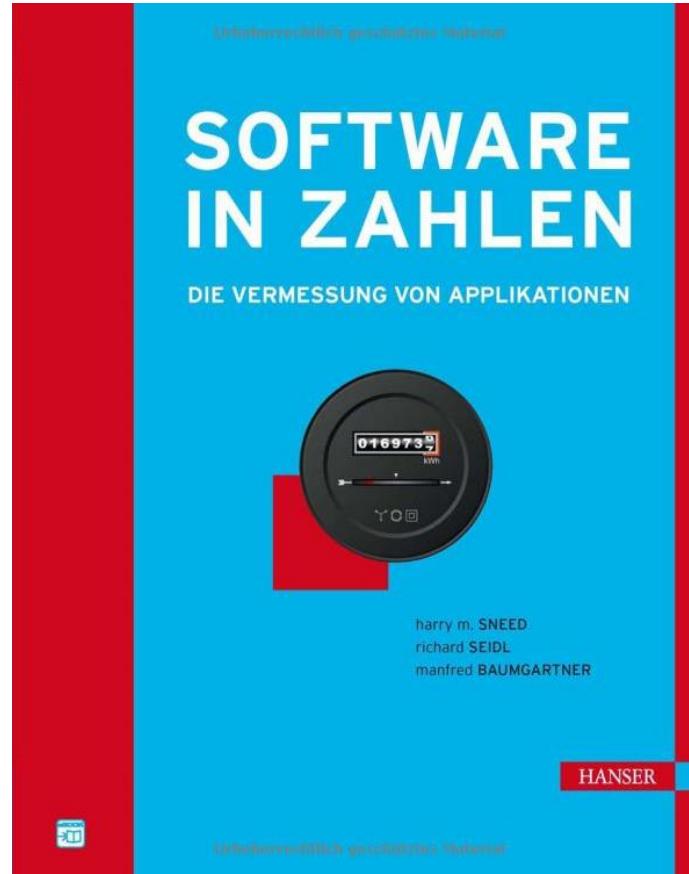
## The ONLY VALID MEASUREMENT OF Code QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>



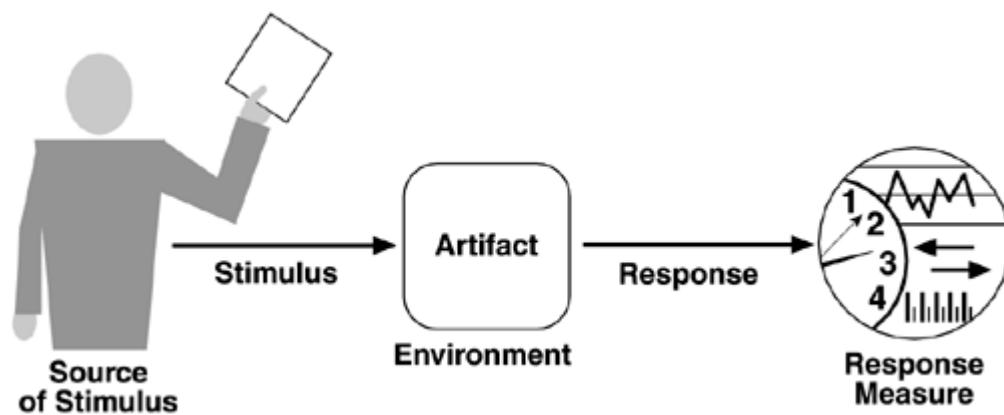
# Literatur





# Metriken anhand von Quality Attribute Scenarios

- **Methodik** zur Bestimmung nicht-funktionaler Anforderungen
- Beliebt in **agilen** Projekten
- Ziel ist die Identifikation von sogenannten **Quality Attribute Scenarios**
- Basiert auf der Erstellung eines **Quality Attribute Trees** zur Plausibilisierung hinsichtlich Vollständigkeit



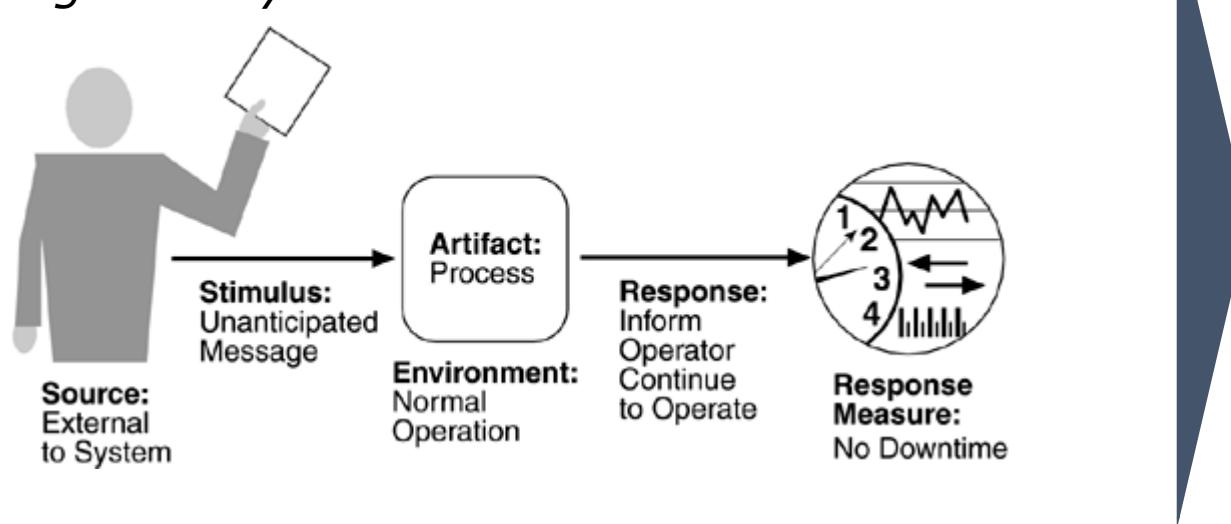
<https://www.cs.unb.ca/~wdu/cs6075w10/sa2.htm>

- **Source of stimulus**  
This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
- **Stimulus**  
The stimulus is a condition that needs to be considered when it arrives at a system.
- **Environment**  
The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.
- **Artifact**  
Some artifact is stimulated. This may be the whole system or some pieces of it.
- **Response**  
The response is the activity undertaken after the arrival of the stimulus.
- **Response measure**  
When the response occurs, it should be measurable in some fashion so that the requirement can be tested.



# Beispiel für ein Quality Attribute Scenario

*"Unexpected messages are handled gracefully."*



**Quality Attribute:** Downtime  
**Value:** 0 sec



# Übungsaufgabe

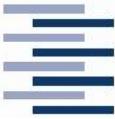
- ① **Ziel:** Bestimmen Sie basierend auf Ihrem Qualitätsmerkmalbaums aus der ersten Übung Quality Attribute Scenarios für Ihr Projekt
- ② SE2-Projektgruppen
- ③ 20 Minuten





# Agenda

- Einführung
- Qualitäts-Merkmale
- Metriken
- **Maßnahmen**
- Zusammenfassung



# Das übergreifende Qualitätsmanagement organisiert die Qualitätssicherung im konkreten Projekt

## Qualitätsmanagement

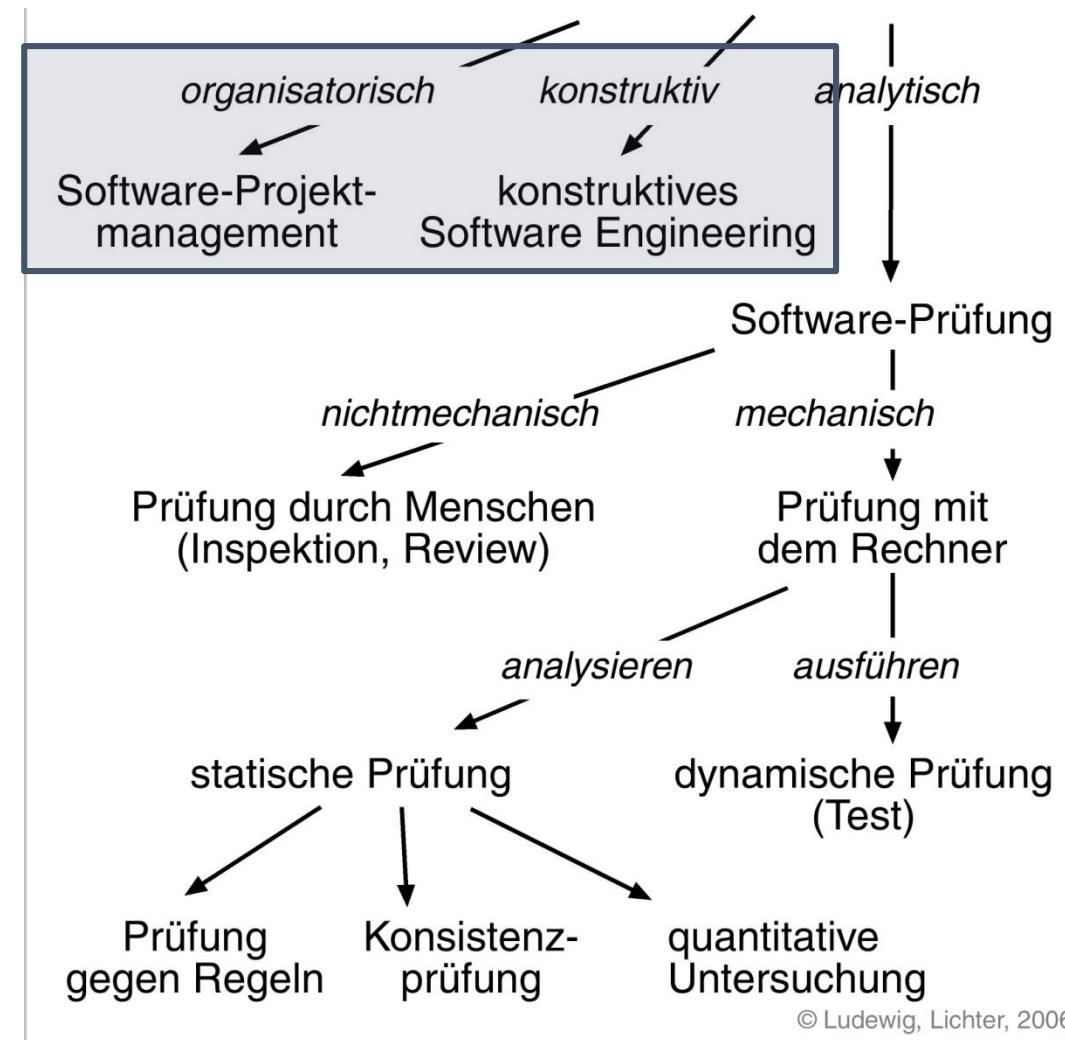
Alle Tätigkeiten der Gesamtführungs-aufgabe, welche die Qualitätspolitik, Ziele und Verantwortlichkeiten festlegen sowie diese durch Mittel wie Qualitätsplanung, Qualitätslenkung, Qualitätsprüfung und Qualitäts-verbesserung im Rahmen des Qualitätsmanagementsystems verwirklichen (ISO 8402).

## Qualitätssicherung

Alle **geplanten und systematischen** Tätigkeiten, die notwendig sind, um ein angemessenes Vertrauen zu schaffen, dass ein Produkt oder eine Dienstleistung die gegebenen Qualitäts-Anforderungen erfüllt. (DIN 55350-11)



# Arten von Software-Qualitätssicherung

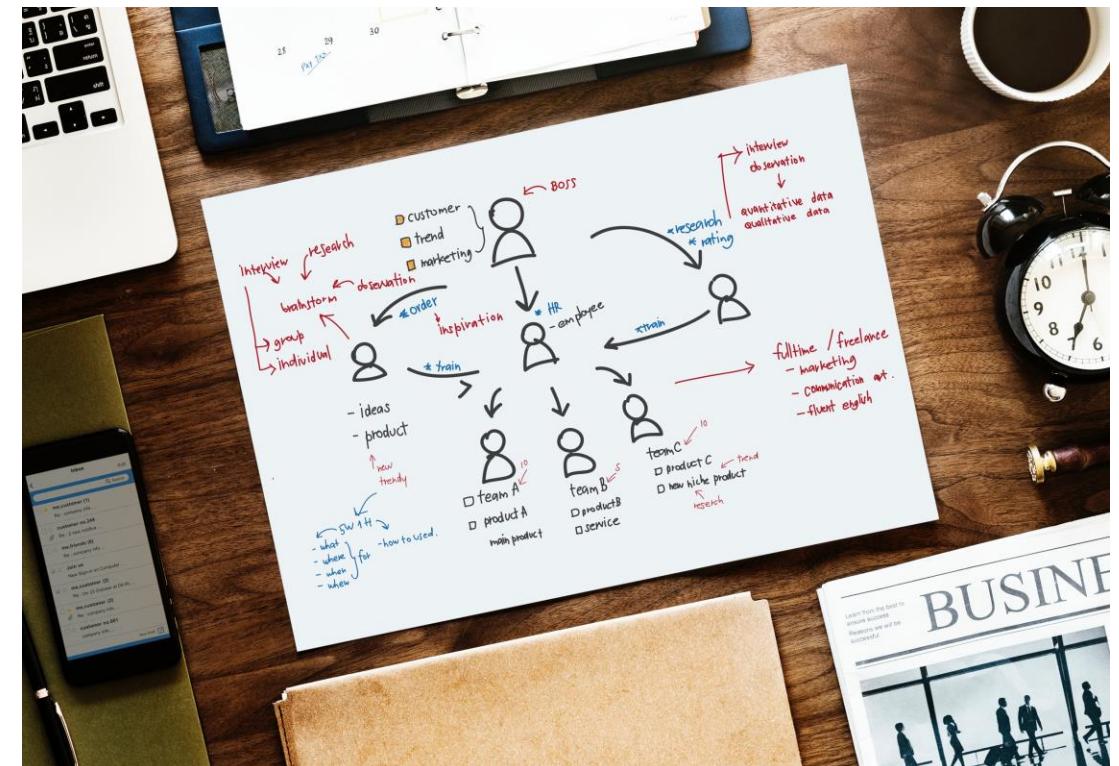


© Ludewig, Licher, 2006



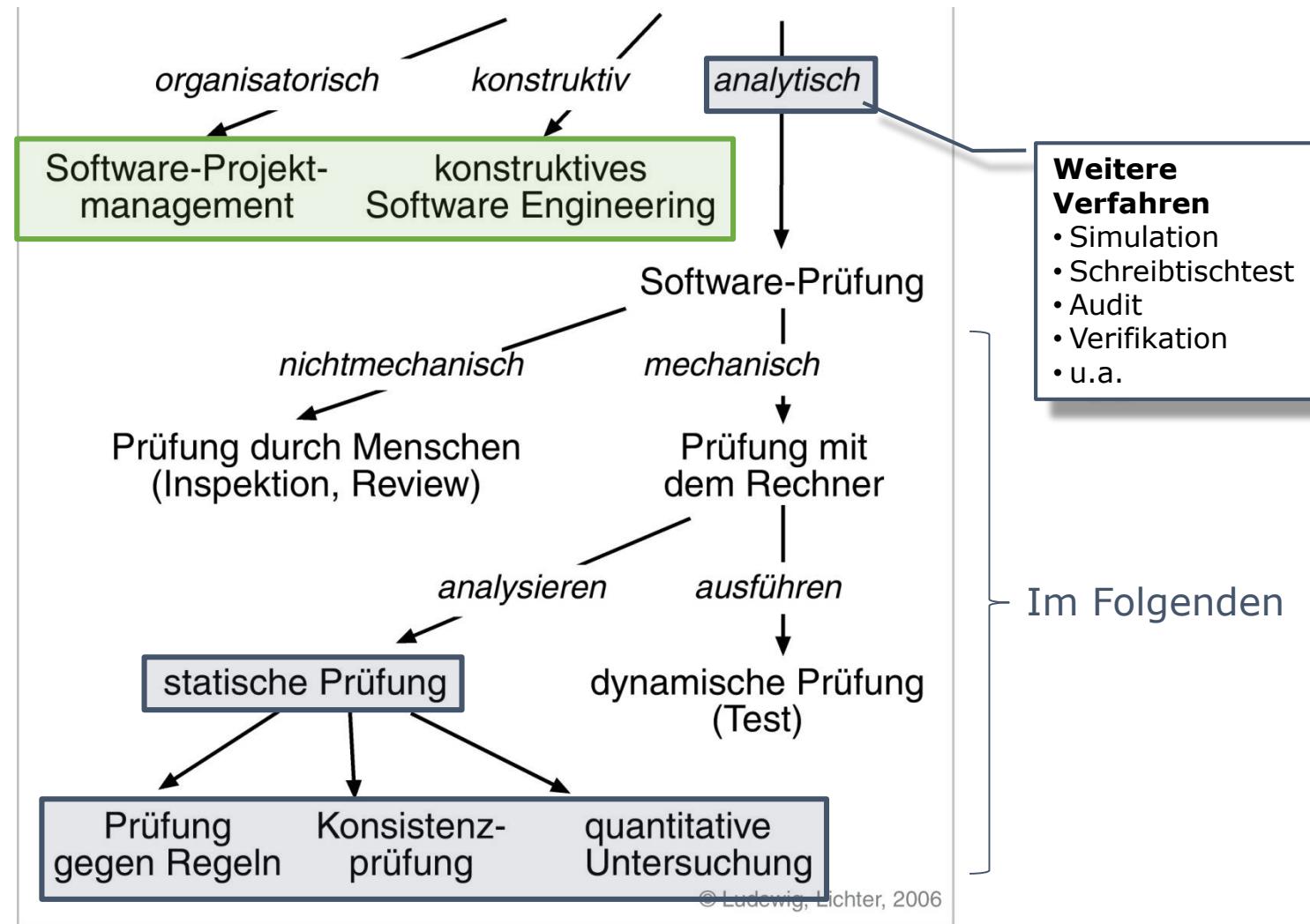
# Organisatorische und konstruktive Qualitätssicherung

- Verfahren zur Anforderungsanalyse / Dokumentation / Freigabe
- Dokumentenstandards (Wiki-Struktur, Lasten- und Pflichtenheft, Architekturbeschreibung, Testfallspezifikation, Protokolle, ...)
- Entwurfsmuster für die Softwarearchitektur
- Codierungsrichtlinien
- Continuous Integration / Continuous Deployment
- Vorgehensmodell (Kanban, Scrum, ...)
- Schulungen
- Prozess zum Behandeln von Änderungen („Change Requests“)
- Meilensteine und Freigabewesen der Teilprodukte
- Festlegung von Verantwortlichkeiten (bspw. Sicherstellung der organisatorischen Unabhängigkeit des Qualitätsmanagements)
- ...





# Arten von Software-Qualitätssicherung





# Was kann analysiert werden?

- Coding-Style
- „Dumme“ Fehler

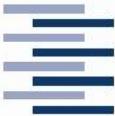
```
if (x < 0)
    new IllegalArgumentException("x must be nonnegative");
```

- Sprach-Antipatterns

```
// This is bad
String s = "";
for (int i = 0; i < field.length; ++i) {
    s = s + field[i];
}
```

- „Schlechte“ Fehlerbehandlung z.B. Stream nicht geschlossen
- Metriken, z.B. McCabe (Zyklomatische Komplexität)
- Technische Schulden
- ...

<http://findbugs.sourceforge.net/bugDescriptions.html>



# Technische Schulden

„Unter der technischen Schuld versteht man den zusätzlichen Aufwand, den man für Änderungen und Erweiterungen an schlecht geschriebener Software im Vergleich zu gut geschriebener Software einplanen muss.“

(Wikipedia)





# Statische Codeanalyse für Java - Checkstyle

Checkstyle is highly configurable and can be made to support almost any coding standard. An example configuration files are supplied supporting the Sun Code Conventions [\[1\]](#), Google Java Style [\[2\]](#).

<a href="#">AbstractClassName</a>	Ensures that the names of abstract classes conforming to some regular expression.
<a href="#">AnnotationUseStyle</a>	This check controls the style with the usage of annotations.
<a href="#">AnonInnerLength</a>	Checks for long anonymous inner classes.
<a href="#">ArrayTrailingComma</a>	Checks if array initialization contains optional trailing comma.
<a href="#">ArrayTypeStyle</a>	Checks the style of array type definitions.
<a href="#">AvoidInlineConditionals</a>	Detects inline conditionals.
<a href="#">AvoidNestedBlocks</a>	Finds nested blocks.
<a href="#">AvoidStarImport</a>	Check that finds import statements that use the * notation.
<a href="#">AvoidStaticImport</a>	Check that finds static imports.
<a href="#">BooleanExpressionComplexity</a>	Restricts nested boolean operators (&&,   , &,   and ^) to a specified depth (default = 3).
<a href="#">ClassDataAbstractionCoupling</a>	This metric measures the number of instantiations of other classes within the given class.
<a href="#">ClassFanOutComplexity</a>	The number of other classes a given class relies on.
<a href="#">ClassTypeParameterName</a>	Checks that class type parameter names conform to a format specified by the format property.
<a href="#">ConstantName</a>	Checks that constant names conform to a format specified by the format property.
<a href="#">CovariantEquals</a>	Checks that if a class defines a covariant method equals, then it defines method equals(java.lang.Object).
<a href="#">CyclomaticComplexity</a>	Checks cyclomatic complexity against a specified limit.

[Quelle: <http://checkstyle.sourceforge.net>]



# Statische Codeanalyse für Java - FindBugs

<a href="#">Nm: Class names shadow simple name or superclass</a>
<a href="#">Nm: Very confusing method names (but perhaps intentional)</a>
<a href="#">Nm: Method doesn't override method in superclass due to wrong package for parameter</a>
<a href="#">ODR: Method may fail to close database resource</a>
<a href="#">ODR: Method may fail to close database resource on exception</a>
<a href="#">OS: Method may fail to close stream</a>
<a href="#">OS: Method may fail to close stream on exception</a>
<a href="#">RC: Suspicious reference comparison</a>
<a href="#">RR: Method ignores results of InputStream.read()</a>
<a href="#">RR: Method ignores results of InputStream.skip()</a>
<a href="#">RV: Method ignores exceptional return value</a>
<a href="#">SI: Static initializer creates instance before all static final fields assigned</a>
<a href="#">SW: Certain swing methods needs to be invoked in Swing thread</a>
<a href="#">Se: Non-transient non-serializable instance field in serializable class</a>
<a href="#">Se: Non-serializable class has a serializable inner class</a>
<a href="#">Se: Non-serializable value stored into instance field of a serializable class</a>
<a href="#">Se: Comparator doesn't implement Comparable</a>
<a href="#">Se: Comparable inner class</a>
<a href="#">Se: serialVersionUID isn't final</a>
<a href="#">Se: serialVersionUID is final but not constant</a>

[Quelle: : <http://findbugs.sourceforge.net>]



# Statische Codeanalyse für C/C++ - FlexeLint

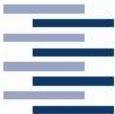
FlexeLint for C/C++ (Unix) Vers. 9.00b, Copyright Gimpel Software 1985-2008

--- Module: offbyone.c (C)

```
1  /* Off-By-One Example */
2  #include <stdio.h>
3  int main()
4  {
5      int i;
6      int a[] = {1,2,3};
7      int n = sizeof(a)/sizeof(int);
8      for(i=0;i<=n;i++)
9
9          printf("a[%d]=%d\n",i,a[i]);
offbyone.c 9  Warning 661: Possible access of out-of-bounds pointer (1 beyond end of data) by operator '['
10     return 0;
11 }
```

```
27         sum = sum + Value(Extract(*s++));
generaltest.cpp 27  Warning 666: Expression with side effects passed to repeated parameter 1 in macro 'Value'
28         return Abs( sum - 100 );
generaltest.cpp 28  Warning 665: Unparenthesized parameter 1 in macro 'Abs' is passed an expression
29     }
generaltest.cpp 29  Info 818: Pointer parameter 's' (line 23) could be declared as pointing to const
30
31 class String
```

[Quelle: <http://www.gimpel-online.com>]



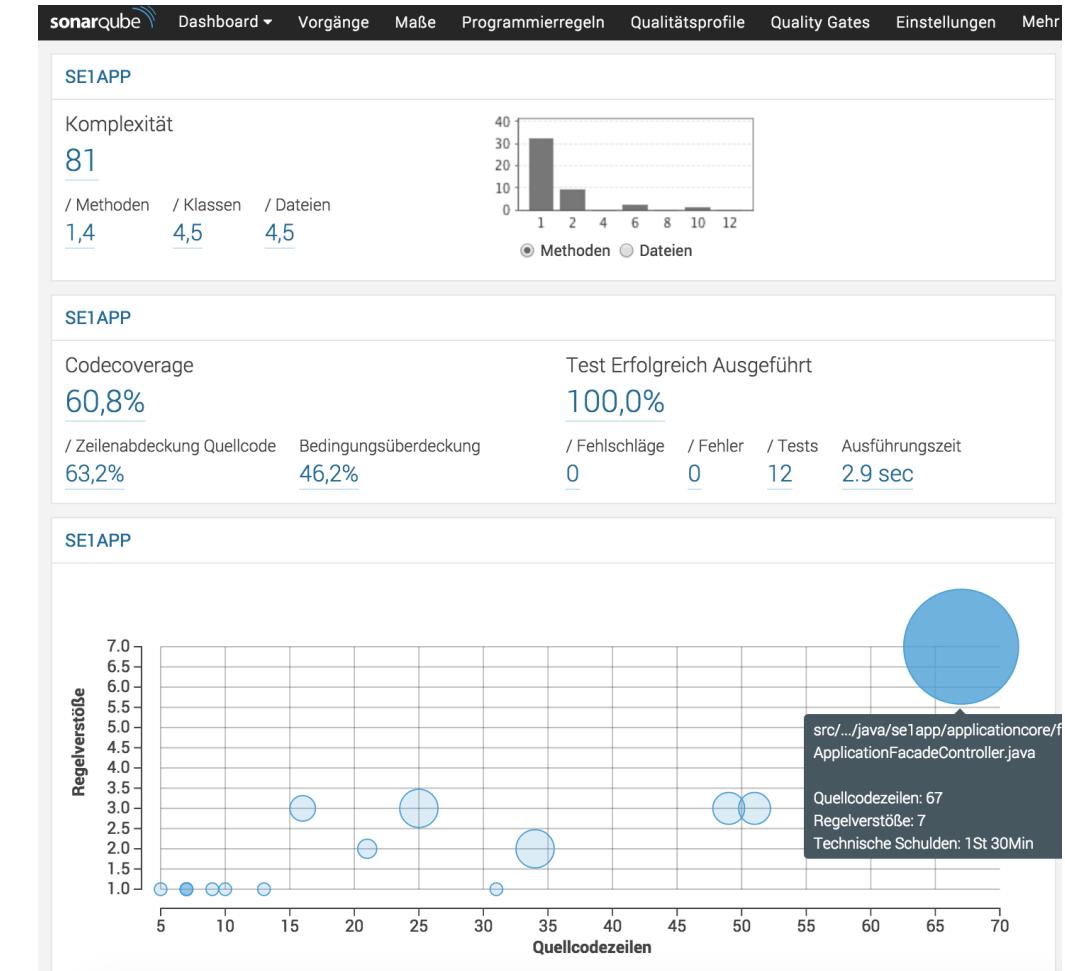
# Statische Codeanalyse – sonarqube®

- Mächtiges Werkzeug für Analysen
- Kann verschiedene Metriken berechnen und Analysen durchführen und die Ergebnisse auf einem konfigurierbaren Dashboard darstellen
- erweiterbar durch Plugins und für viele Sprachen geeignet

<http://www.sonarqube.org/>

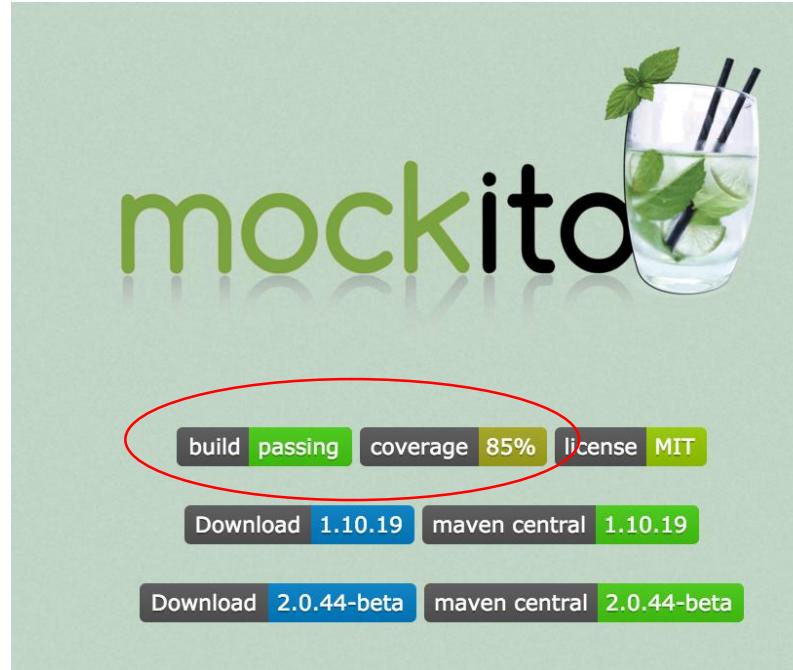
- sonarqube-Plugin für gradle verfügbar

```
Other tasks
-----
cleanIdeaWorkspace
dependencyManagement
jacocoTestReport
sonarqube - Analyzes root project 'se1' and its subprojects with SonarQube.
wrapper
```



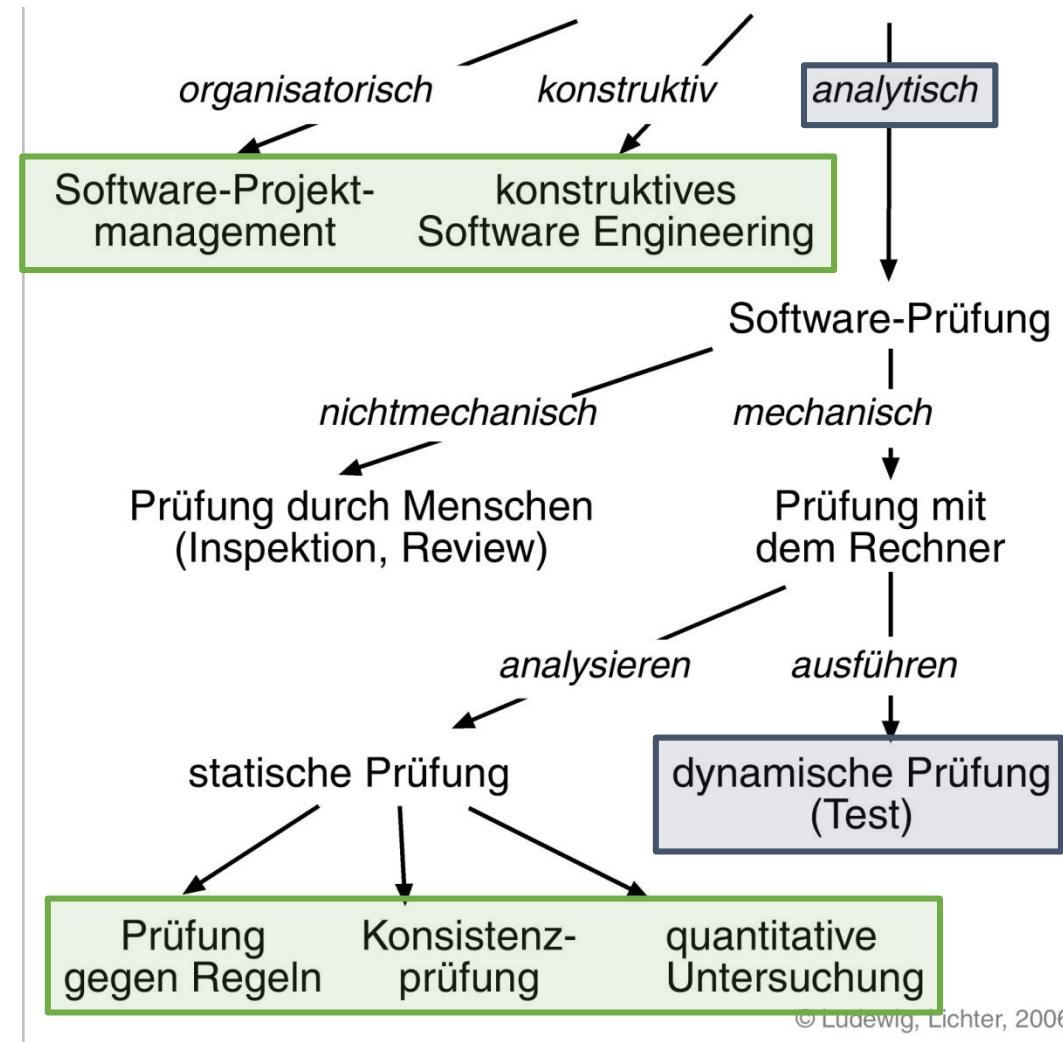


# Metriken in OpenSource-Projekten





# Arten von Software-Qualitätssicherung



© Ludewig, Licher, 2006



# Statischer vs. Dynamischer Test

Oft wird der Begriff „Test“ uneinheitlich verwendet:

- „Test“ = jede Art von Softwareprüfung
- „Test“ = ausschließlich dynamische Prüfung

Sprechweise **Certified Tester**:

- **Statischer Test:** Testobjekt wird **nicht** ausgeführt
  - Untersuchung von Dokumenten – Anforderungen, ...
  - Auch: Untersuchung von Code im Ausdruck
- **Dynamischer Test:** Testobjekt wird ausgeführt
  - Fehler-orientiert: Fehler finden
  - Konformitäts-orientiert: Testgegenstand verhält sich konform zu seiner Spezifikation (Konformitätstest oder Abnahmetest)
  - Auch von Hand!



Quelle: Vorlesung Prof. Butth



# Motivation – Grundsätze des Testens

- Testen zeigt die Anwesenheit von Fehlern  
nicht ihre Abwesenheit!
- Vollständiges Testen ist nicht möglich  
außer bei trivialen Testobjekten
- Mit dem Testen frühzeitig beginnen  
spät erkannte Fehler sind teuer
- Häufung von Fehlern  
Fehler sind nicht gleichmäßig über die Software verteilt
- Zunehmende Testresistenz (Pesticide Paradox)  
Testfälle regelmäßig prüfen, erweitern, modifizieren
- Testen ist abhängig vom Umfeld  
Tests anpassen an Einsatzumgebung und Anwendungsrandbedingungen
- Trugschluss: kein Fehler bedeutet ein brauchbares System  
Spezieller Bereich: Usability Tests; Nutzeranforderungen mit Prototypen ausloten



# Testkomplexität

## Anzahl Parameter

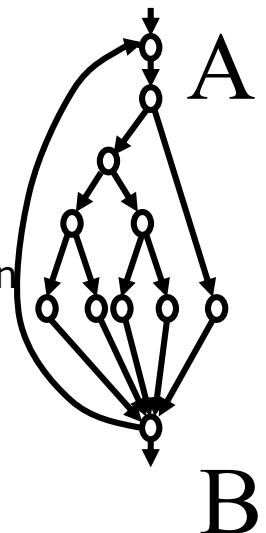
- Ein einfaches Programm, dass drei ganzzahlige Eingabewerte hat, soll getestet werden. Übrige Randbedingungen haben keinen Einfluss auf das Testobjekt
  - Jeder Eingabewert kann bei 16 Bit Integerzahlen 216 unterschiedliche Werte annehmen
  - Bei drei unabhängigen Eingabewerten ergeben sich  $216 * 216 * 216 = 248$  Kombinationen
  - Jede dieser Kombinationen ist zu testen
- Wie lange dauert es bei 100.000 Tests pro Sekunde?

**ca. 89 Jahre**

**Ziel:** Auswahl möglichst weniger Testfälle, die möglichst viel abdecken

## Orientiert an Struktur

- Ein einfaches Programm soll getestet werden, das aus vier Verzweigungen (if-Anweisungen) und einer umfassenden Schleife besteht.
- Annahmen:
  - Verzweigungen sind voneinander unabhängig
  - Schleife wird maximal 20 Mal durchlaufen
  - Anzahl unterschiedlicher Durchläufe =  $5^1 + 5^2 + \dots + 5^{18} + 5^{19} + 5^{20} = 100$  Billiarden Testfälle



Quelle: Vorlesung Prof. Butth



# Dynamischer Test

## Vorteile

- Testen ist ein natürliches Prüfverfahren
- Tests sind reproduzierbar (!= objektiv)
- Investierter Aufwand mehrfach nutzbar
- Zielumgebung wird mit geprüft
- Systemverhalten wird sichtbar gemacht

## Nachteile

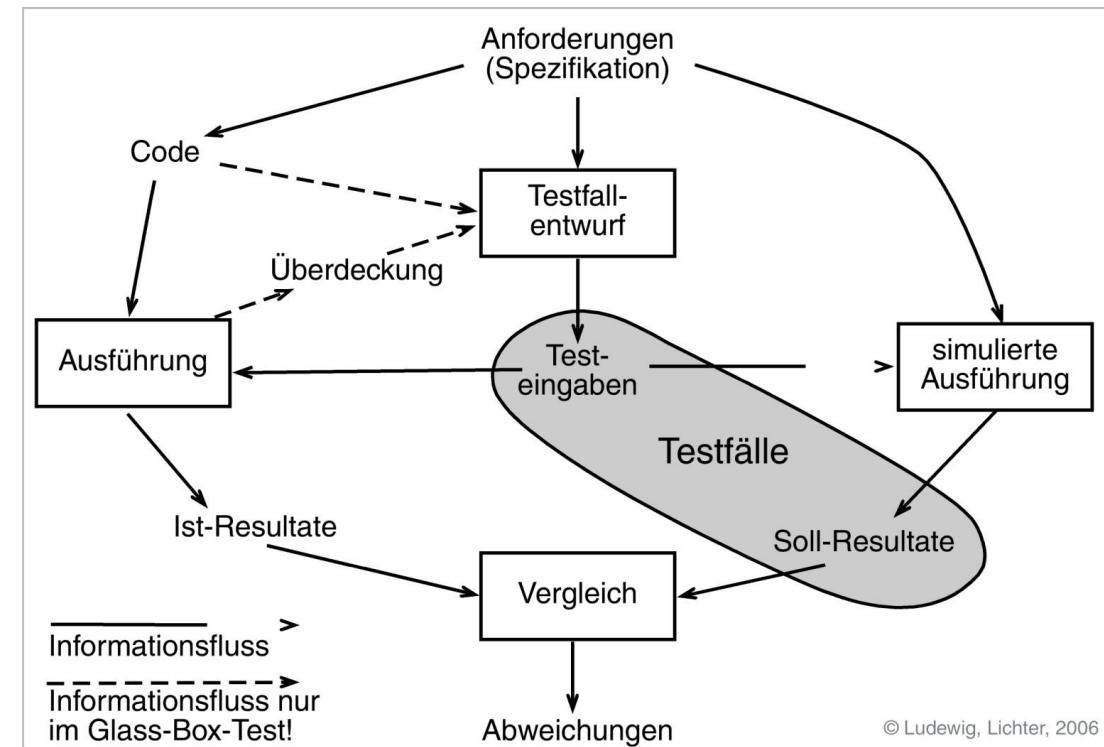
- Ergebnisse werden überschätzt (Korrekttheitsaussage unmöglich, Tests sind Stichproben)
- Nicht alle Programm-Eigenschaften sind testbar
- Nicht alle Anwendungssituationen sind nachbildbar
- Test zeigt die Fehlerursache nicht



# Dynamische Tests müssen systematisch sein

**Systematischer Test:** Ein Test, bei dem

- die **Randbedingungen** definiert oder präzise erfasst sind
  - Programm
  - Übersetzer
  - Betriebssystem
  - ...
- die **Eingaben** systematisch ausgewählt wurden
  - Tastatur
  - Dateien
  - Datenbank
  - Zustände der Geräte
- die **Ergebnisse** dokumentiert und nach Kriterien beurteilt werden, die vor dem Test festgelegt wurden





# Teststufen

## **Komponententest / Modultest (auch: Unit-Test)**

Testen einer Komponente bzw. eines „Moduls“ (z.B. einer Klasse)

## **Integrationstest**

Testen des Zusammenspiels mehrerer Komponenten.

## **Systemtest**

Testen des Gesamtsystems ohne reale Nachbarsysteme.  
Alternativ: Integrationstest

## **Verbundtest (keine offizielle Bezeich- nung – eigene Erfahrung)**

Testen der Software im Zusammenspiel mit den Nachbarsystemen.

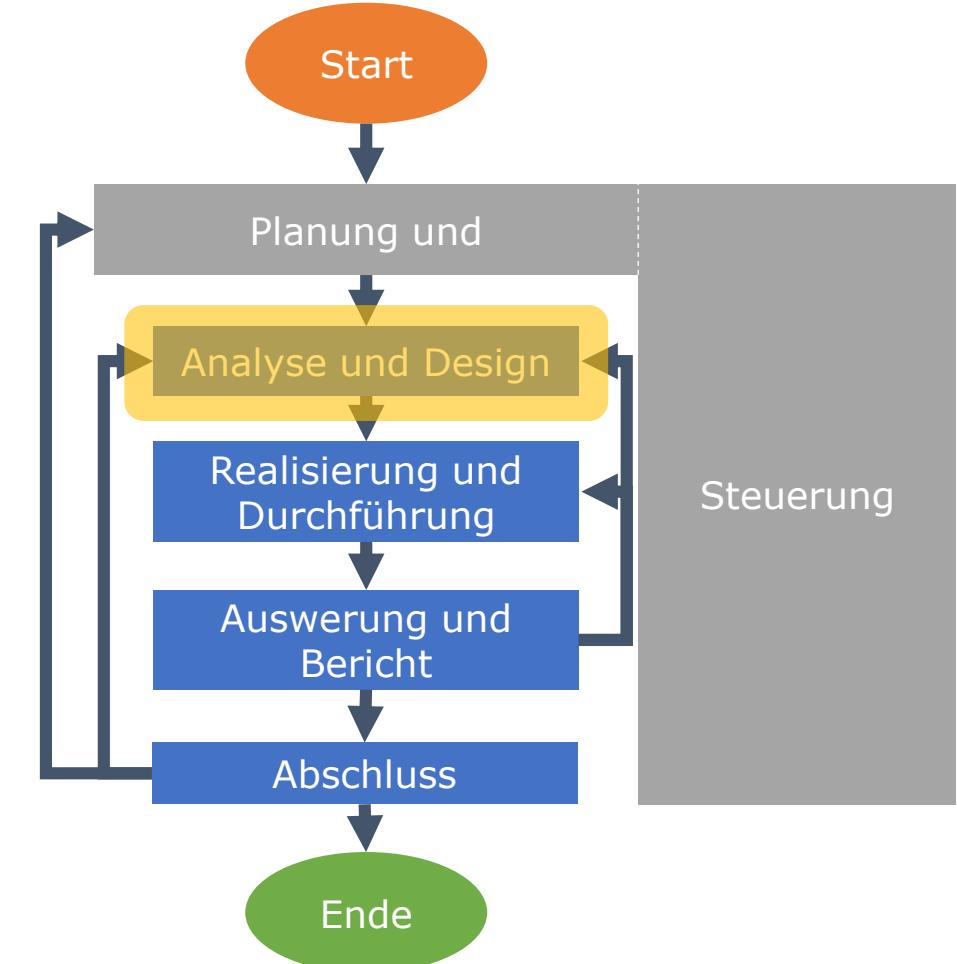
Alternativ: Integrationstest / Systemtest

## **Abnahmetest**

Testen in der realen Einsatzumgebung beim Kunden mit dem Ziel, das Projekt als „abgeschlossen“ kennzeichnen zu können.

# Allgemeiner Test-Prozess nach ISTQB

- **Planung und Steuerung:** Organisation und Planung der Ressourcen`  
(Thema einer späteren Vorlesung)
- **Analyse und Design:** Bestimmung und Spezifikation der Testfälle
- **Realisierung und Durchführung:**  
Ausführung der Testfälle inkl.  
Protokollierung
- **Testauswertung und Bericht:**  
Auswertung der Ergebnisse hinsichtlich  
Produktqualität und Fortschritt des Tests  
(Abdeckung)
- **Abschluss:** kontrollierte Beendigung  
des Test-prozesses nach





# Testfall-Bestimmung

## Black-Box-Test

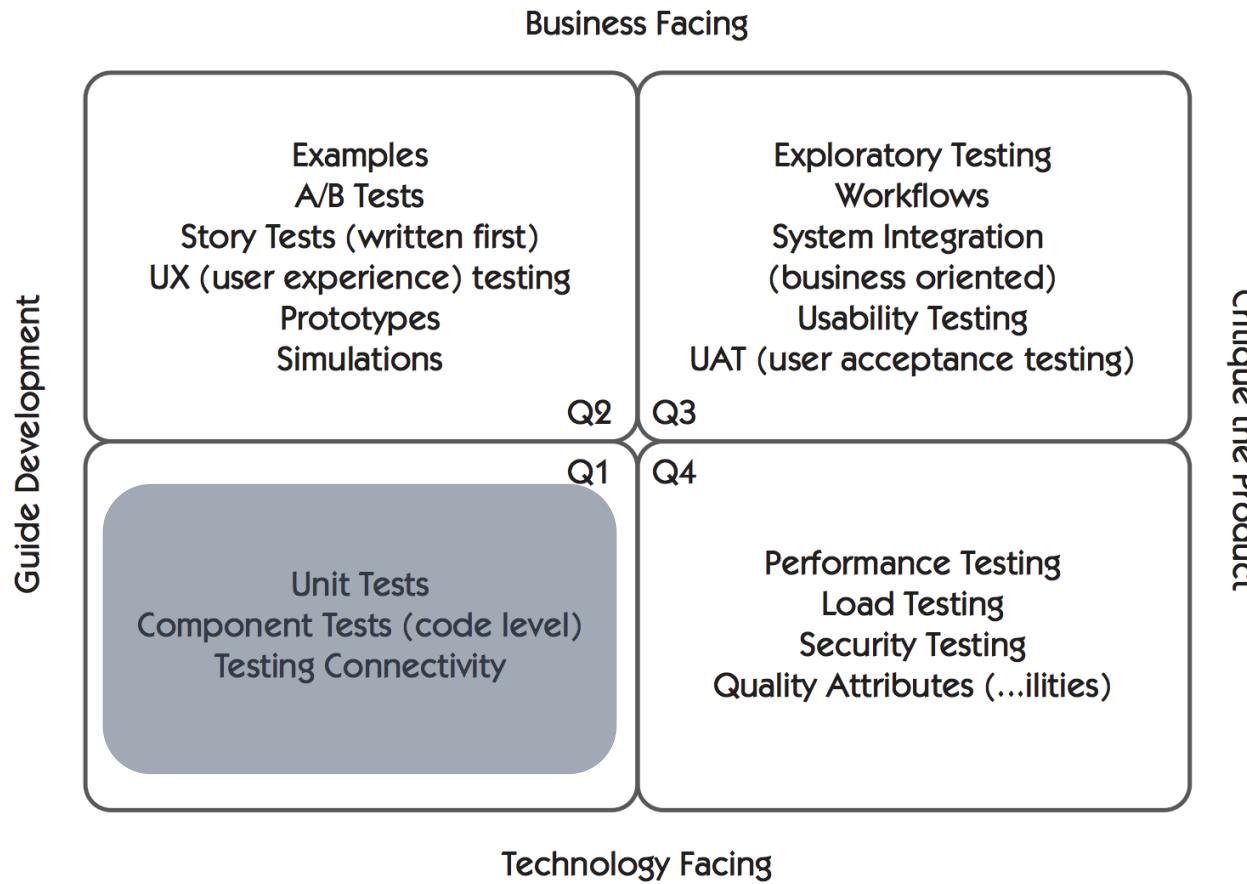
- Auswahl der Testfälle richtet sich nach Eingaben, Ausgaben und ihrer funktionellen Verknüpfung
  - Testdaten aus den Anforderungen
  - Kein Wissen über interne Struktur des Testobjektes
- **Ziel:** Herausfinden, ob vorgegebene Eingaben erwartete Resultate erzielen
- Kriterien für die Auswahl von Testfällen
  - *Funktionsüberdeckung*  
Jede Funktion wird in mindestens einem Testfall ausgeführt
  - *Eingabeüberdeckung*  
Jedes Eingabedatum wird in mindestens einem Testfall verwendet
  - *Ausgabeüberdeckung*  
Jedes Ausgabedatum wird in mindestens einem Testfall erzeugt

## Glass-Box-Test

- Betrachtung des Systems/des Moduls/der Komponente als „White Box“ mit Wissen über die Interna.
- Weitere Unterscheidungen:
  - *Anweisungsüberdeckung*: Alle Anweisungen eines Programms wurden ausgeführt
  - *Zweigüberdeckung*: Alle möglichen Verzweigungen wurden durchlaufen
  - *Termüberdeckung*: jeder logische Term, der eine Verzweigung steuert, ist mit beiden möglichen Werten (true/false) wirksam geworden



# Brian Maricks Testing Quadrant



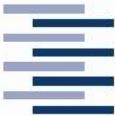
**Taxonomy von Testarten anhand von vier Dimensionen:**

**Guide Development:** Tests die integraler Bestandteil der Entwicklung sind

**Critique the Product:** Tests die sich auf ein finales Produkt fokussieren, um dort Fehler zu finden

**Business Facing:** für die Anwender

**Technology Facing:** mehr Team-intern



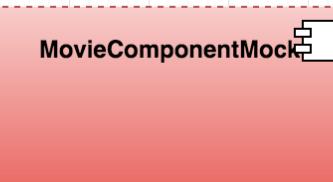
# Test Doubles

- Entwickler müssen (irgendwann) einzelne Systemteile isoliert testen
  - Systemteile können sein: Klassen, Komponenten, eine Auswahl von Komponenten, das eigene System ohne Nachbarsysteme, ...
  - abhängige Teile stehen noch nicht zur Verfügung
  - dies ermöglicht parallele Entwicklung in großen Teams
- Wie kann ein Entwickler trotzdem sicherstellen, dass „sein“ Teilsystem „funktioniert“, ohne auf „die anderen Teile“ warten zu müssen?
- Er kann Test Doubles nutzen

*Ich entwickle die „CustomerComponent“*



*Jemand anderes entwickelt die „MovieComponent“*



*Ich erstelle für mich ein Test Double der „MovieComponent“*



# Arten von Test Doubles

<b>Dummies</b>	Füllobjekt, werden aber niemals benutzt; z.B. zum Füllen von Parameterlisten.
<b>Fake objects</b>	Haben eine lauffähige Implementierung, diese ist jedoch „einfach“ gehalten und nicht für den Produktionseinsatz geeignet.  Beispiel: In-Memory-Datenbank (z.B. H2)
<b>Stubs</b>	Liefern vorgefertigte Ergebnisse für programmierte Testaufrufe. Zur Prüfung des Testergebnisses wird der Zustand im Anschluß geprüft.
<b>Spies</b>	Stubs die sich auch Informationen über Aufrufe „merken“ – bspw. die „gesendeten“ Nachrichten einer E-Mail-Komponente.
<b>Mocks</b>	Vorkonfigurierte Objekte mit Spezifikation der erwarteten Aufrufe und deren Ergebnisse. Im Gegensatz zu Stubs wird hier nur die Spezifikation des Aufrufs geprüft

Quelle: <http://martinfowler.com/articles/mocksArentStubs.html>



# Test Doubles - Mocks

Mockito ist ein Java-Frameworks zur einfachen Erstellung von Mocks  
<http://mockito.org/>

```
//You can mock concrete classes, not just interfaces
LinkedList mockedList = mock(LinkedList.class);

//stubbing
when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());

//following prints "first"
System.out.println(mockedList.get(0));

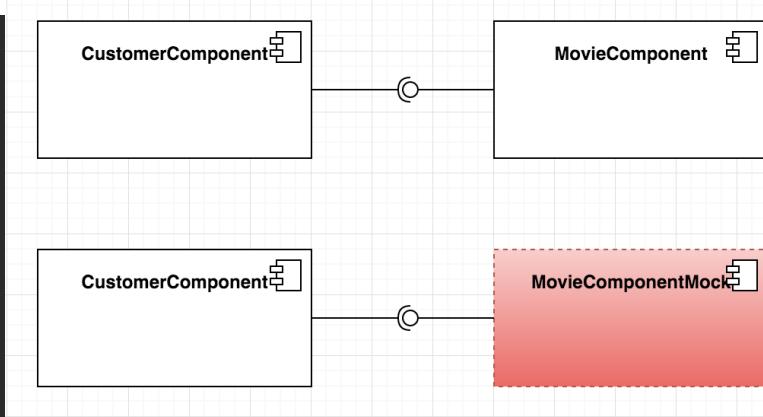
//following throws runtime exception
System.out.println(mockedList.get(1));

//following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));
```



# Test Doubles – Mockito in Unitests

```
@Before  
public void setup() {  
    // Testdaten für den Komponententest initialisieren  
    Customer customer = new Customer("Heinz");  
    customerRepository.save(customer);  
    customerHeinzId = customer.getId();  
  
    movie007 = new Movie("007");  
    movieRepository.save(movie007);  
  
    // Mock-Abhängigkeit zu MovieComponent übergeben  
    movieComponentInterface = mock(MovieComponentInterface.class);  
    customerComponentInterface = new CustomerComponent(customerRepository, movieComponentInterface);  
}
```



```
@Test  
public void testAddReservation() {  
    Customer customer = customerComponentInterface.getCustomer(customerHeinzId);  
    assertThat(customer).isNotNull();  
  
    // hier testen wir, ob Customercomponent die abhängige Komponente korrekt aufgerufen hat  
    try {  
        customerComponentInterface.addReservation(customerHeinzId, new Reservation(movie007));  
        verify(movieComponentInterface).increaseReservationStatistics(movie007.getTitle());  
    }  
}
```

Verhaltensprüfung!



# Test Doubles - Stubs

```
public interface MailService {  
    public void send (Message msg);  
}  
  
public class MailServiceStub implements MailService {  
    private List<Message> messages = new ArrayList<Message>();  
    public void send (Message msg) {  
        messages.add(msg);  
    }  
    public int numberSent() {  
        return messages.size();  
    }  
}
```

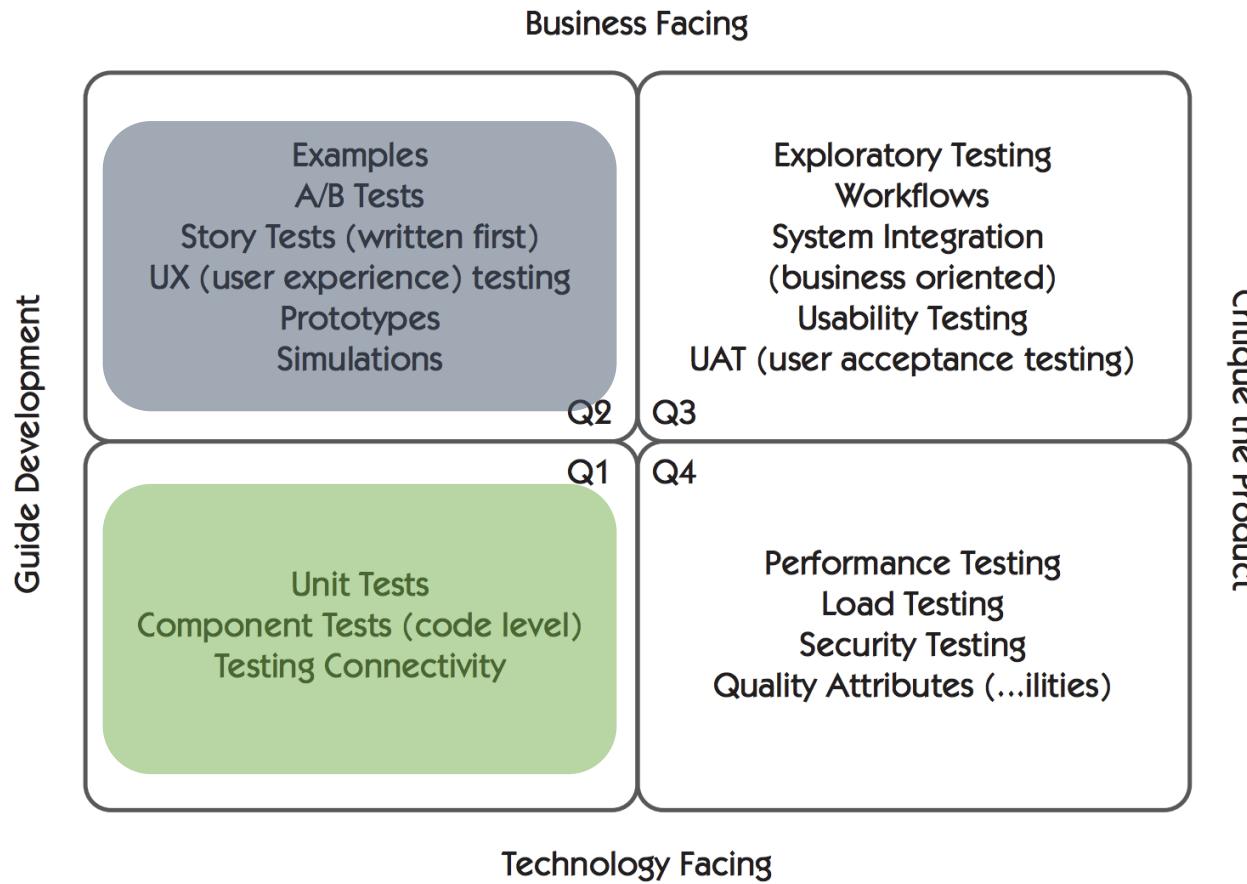
Zustandsprüfung

```
class OrderStateTester...  
    public void testOrderSendsMailIfUnfilled() {  
        Order order = new Order(TALISKER, 51);  
        MailServiceStub mailer = new MailServiceStub();  
        order.setMailer(mailer);  
        order.fill(warehouse);  
        assertEquals(1, mailer.numberSent());  
    }
```

Quelle: <http://martinfowler.com/articles/mocksArentStubs.html>



# Brian Maricks Testing Quadrant



**Taxonomy von Testarten anhand von vier Dimensionen:**

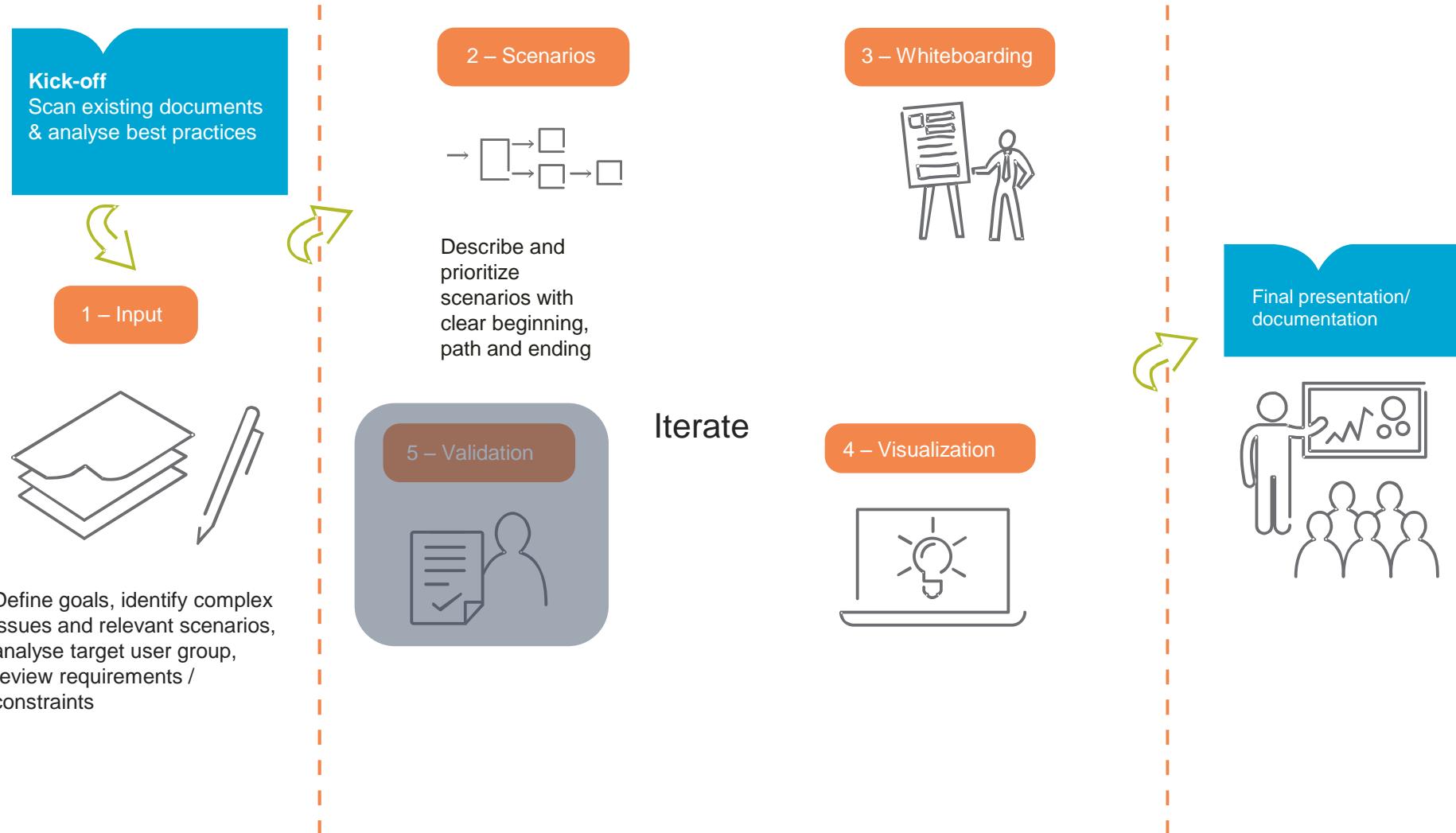
**Guide Development:** Tests die integraler Bestandteil der Entwicklung sind

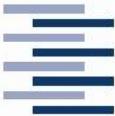
**Critique the Product:** Tests die sich auf ein finales Produkt fokussieren, um dort Fehler zu finden

**Business Facing:** für die Anwender

**Technology Facing:** mehr Team-intern

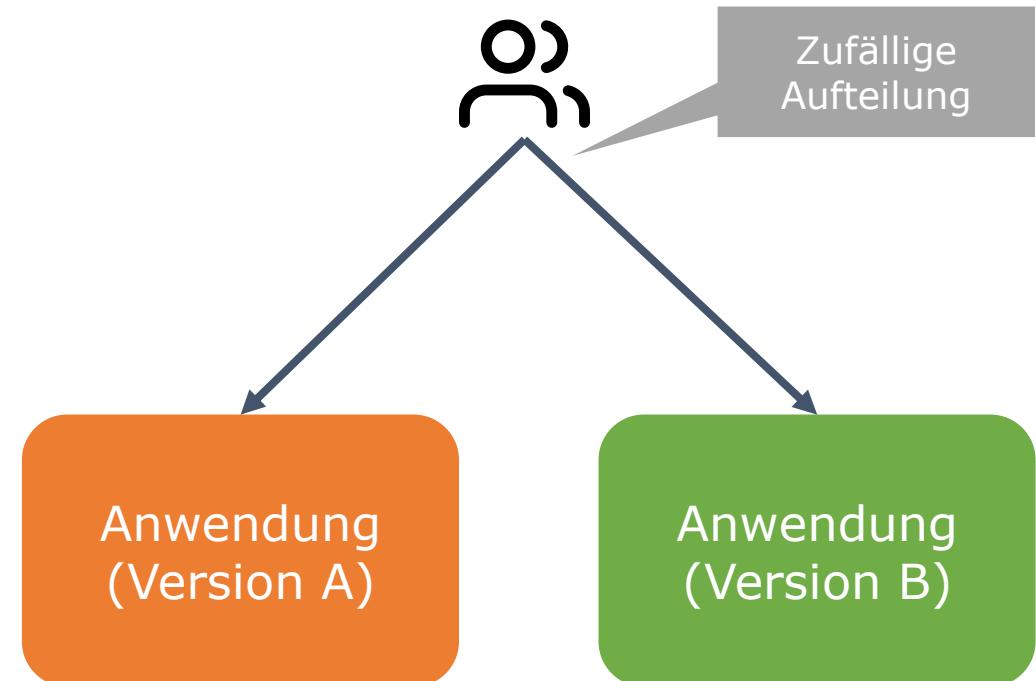
# UX Testing und Prototyping am Beispiel der Capgemini Rapid Design Visualization Methodik

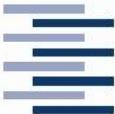




# A/B Testen

- Ermöglicht das Testen einer Funktionalität in unterschiedlichen Versionen typischerweise in Web-Anwendungen (aber nicht notwendigerweise!)
- **Version A:** typischer Weise die aktuelle Live-Version
- **Version B:** neue Version mit Veränderung des Features oder einem neuen Feature
- Beide Versionen werden parallel betrieben und Ergebnisse verglichen (z.B. Klick-Rate)
- Auswertung kann formalisiert werden, oder auf Feedback beruhen

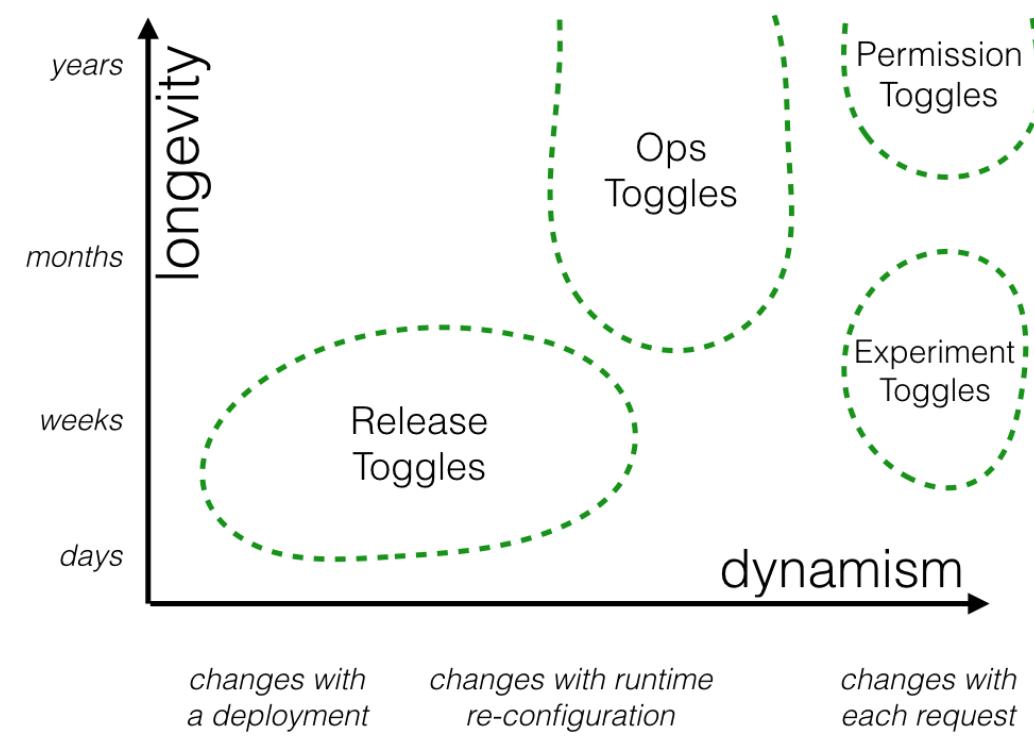




# Feature Toggles

- Weiterführung des A/B-Test-Konzepts:  
Implementierung von verschiedenen  
Versionen in einem Deployment
- Beispiel eines „Toggle Points“:

```
if (isRecommendationFeatureEnabled()) {  
    fetchRecommendations();  
    ...  
}
```
- Toggles können prinzipiell hinsichtlich  
zweier Dimensionen klassifiziert werden:
  - *Langlebigkeit*: wie lang soll der Toggle im  
Code existieren?
  - *Dynamik*: wann kann der Toggle verändert  
werden?



Quelle: <https://martinfowler.com/articles/feature-toggles.html>



# Umsetzung von Toggle Features über Togglz in Java

## Konfiguration:

```
public enum MyFeatures implements Feature {  
  
    @EnabledByDefault  
    @Label("First Feature")  
    FEATURE_ONE,  
  
    @Label("Second Feature")  
    FEATURE_TWO;  
  
    public boolean isActive() {  
        return FeatureContext.getFeatureManager().isActive(this);  
    }  
}
```

## Administration:

Feature	Status	Strategy	Actions
First Feature Owner: chkal	Red circle		gear icon
Second Feature ⓘ Owner: john Issue: TOGGLZ-134	Green circle	Gradual rollout Percentage: 10	gear icon
Third Feature ⓘ Owner: chkal Issue: TOGGLZ-68	Green circle	Users by name Users: tester	gear icon

Togglz 2.0.0.Final  
<http://www.togglz.org/>  
JBoss Web/7.0.13.Final

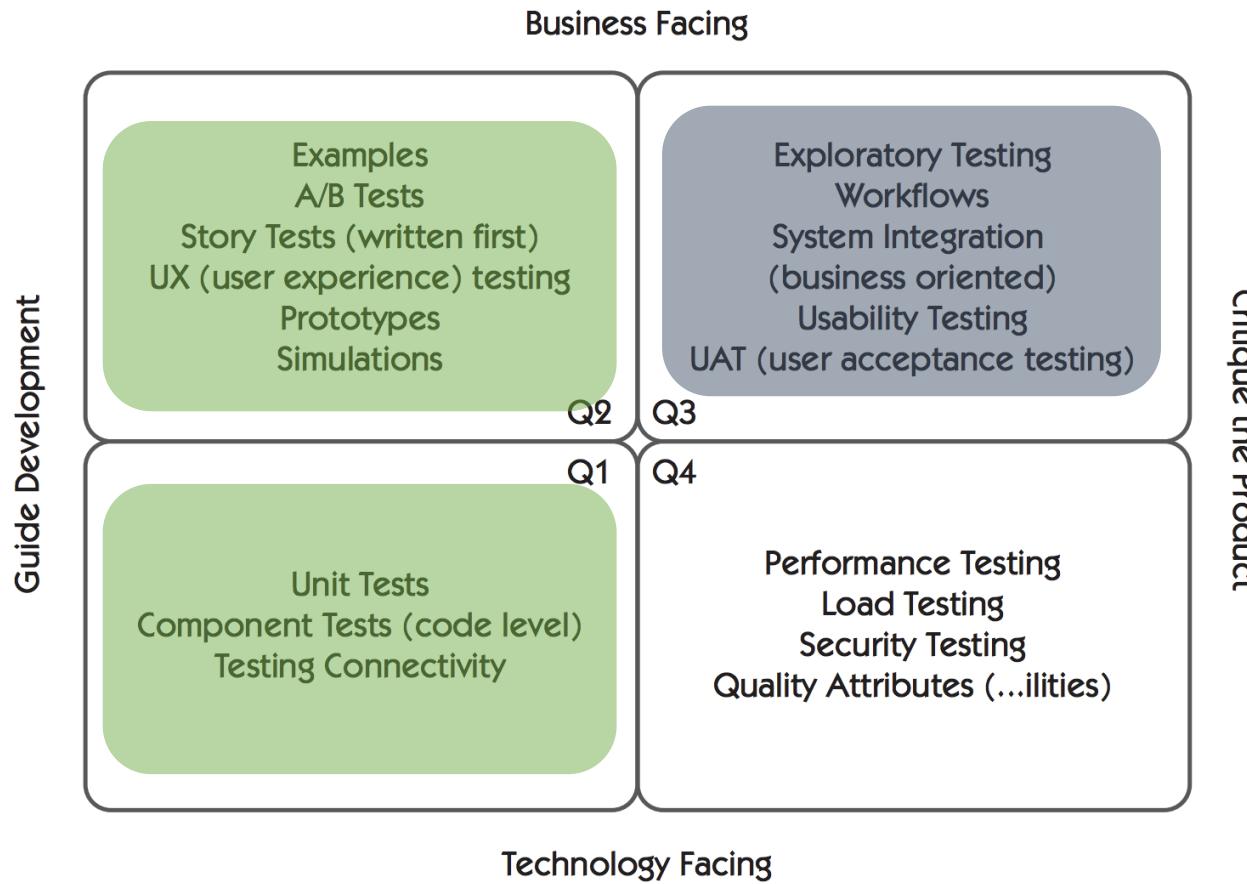
## Verwendung:

```
if( MyFeatures.FEATURE_ONE.isActive() ) {  
    // new stuff here  
}
```

Quelle: <https://www.togglz.org/quickstart.html>



# Brian Maricks Testing Quadrant



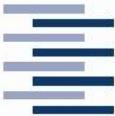
**Taxonomy von Testarten anhand von vier Dimensionen:**

**Guide Development:** Tests die integraler Bestandteil der Entwicklung sind

**Critique the Product:** Tests die sich auf ein finales Produkt fokussieren, um dort Fehler zu finden

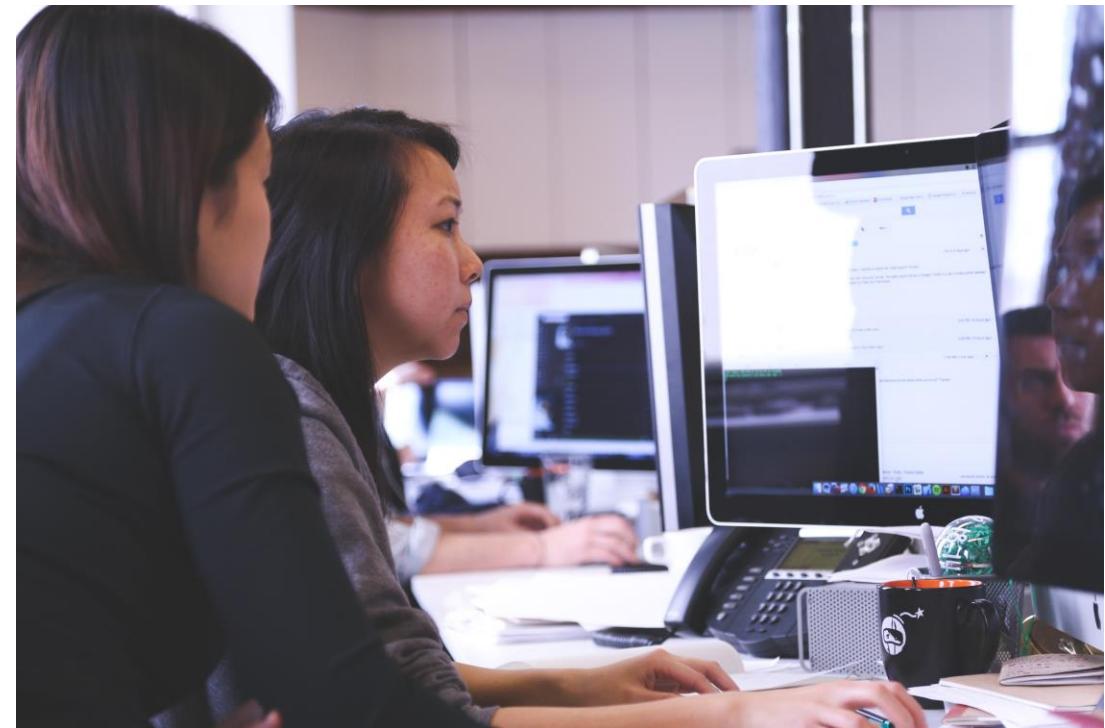
**Business Facing:** für die Anwender

**Technology Facing:** mehr Team-intern



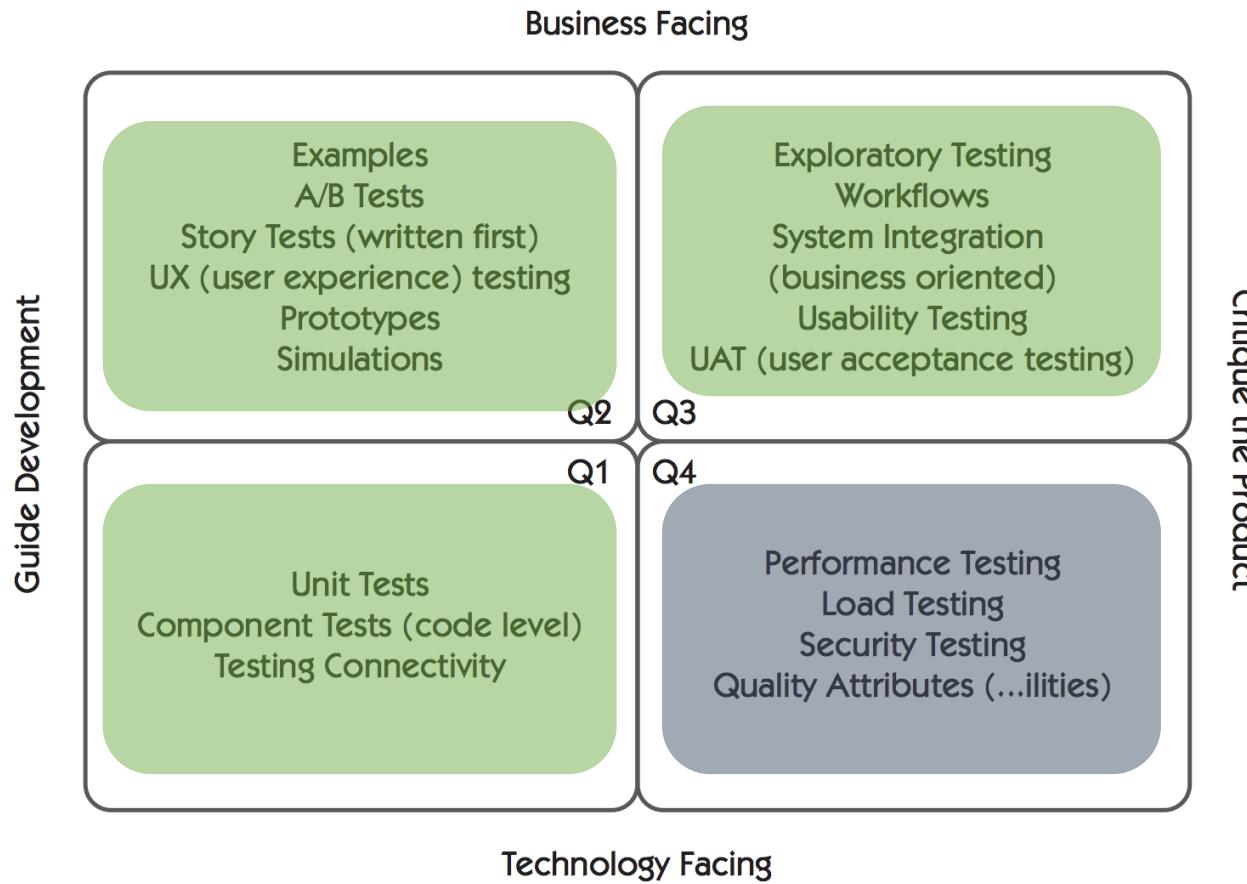
# Usability und User Acceptance Testing

- Prinzipielle Möglichkeiten:
  - Organisation eines systematischen Tests durch ausgewählte End-Anwender (typischerweise für Expertensysteme)
  - Zufällige Auswahl der Tester, um offensichtliche Probleme zu identifizieren (Hallway Testing)
  - Reviews und Audit durch Experten mit entsprechender Erfahrung
- Beim Testen der Usability wird das User-Verhalten observiert und ausgewertet (Remote oder Lokal)





# Brian Maricks Testing Quadrant



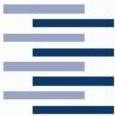
**Taxonomy von Testarten anhand von vier Dimensionen:**

**Guide Development:** Tests die integraler Bestandteil der Entwicklung sind

**Critique the Product:** Tests die sich auf ein finales Produkt fokussieren, um dort Fehler zu finden

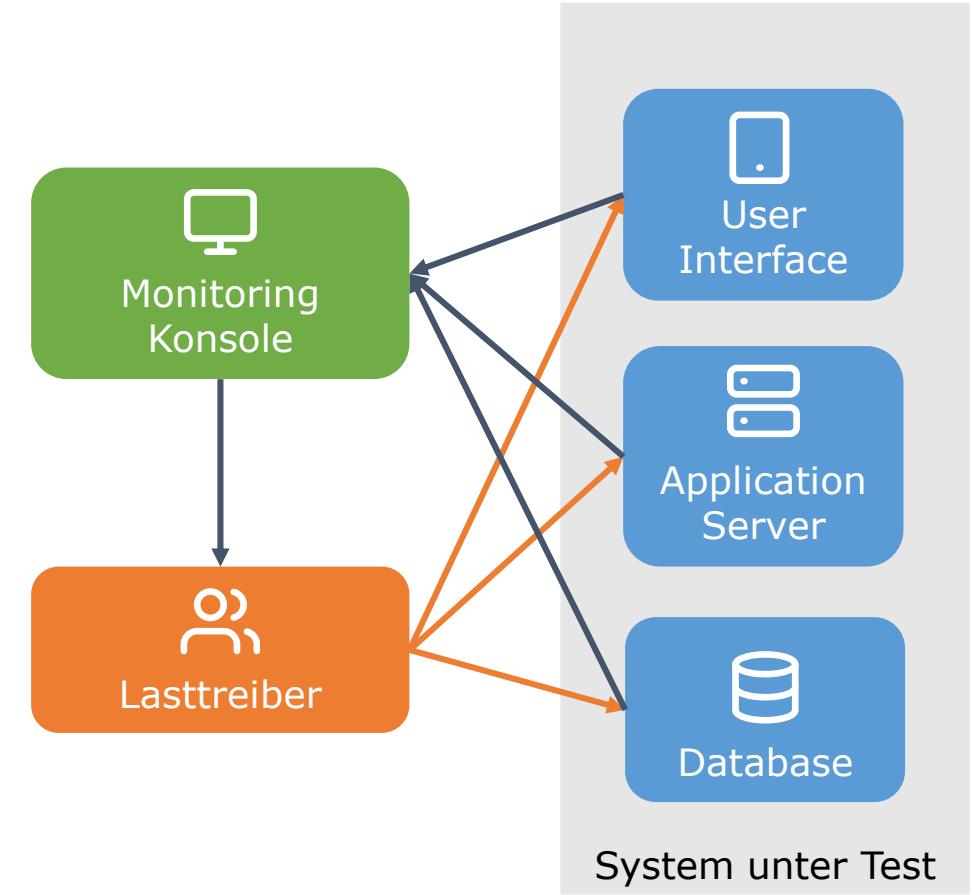
**Business Facing:** für die Anwender

**Technology Facing:** mehr Team-intern



# Überblick: Leistungstests

- Zur Durchführung von Leistungstests ist eine Umfangreiche Infrastruktur notwendig:
  - **Monitoring Konsole:** zur Durchführung und Überwachung des Tests
  - **Lasttreiber:** Simuliert die User
  - **Testsystem:** mit eigener kompletter Infrastruktur möglichst produktionsnah
- Leistungstests können auf unterschiedlichen Komponenten oder das Gesamt-System durchgeführt werden
- Wichtig ist die Möglichkeit der Überwachung!
- Arten von Leistungstests: Volumentest, Zeittest, Lasttest, Stresstest





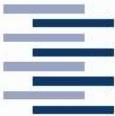
# Für den Leistungstest ist die Erzeugung der Last sowie Messung deren Auswirkung essentiell

## Wie erzeugt man Last?

- Geschäftsvorfälle festlegen
  - es können nur wenige getestet werden
  - müssen für viele Benutzer parallel durchführbar sein
- Testdaten festlegen und erzeugen
  - Benutzer und notwendige Basisdaten
  - Datenmenge ggf. künstlich erhöhen, d.h. kein Test mit „leerer“ Datenbank
- Skripte mit Tools aufzeichnen oder Batchdaten erzeugen
  - Variablen für Benutzer und Variationen einfügen
  - Test und Plausibilisierung der Skripte (für 1 Benutzer)

## Wie messe ich die Auswirkungen?

- Zunächst Wahl der **zu messenden Komponente**: wo ist es am sinnvollsten nach Flaschenhälzen zu prüfen?
- Was wird gemessen?
  - **Verwendung** (z.B. Auf/Abbau von HTTP Verbindungen, Auf/Abbau von Datenbankverbindungen, Anzahl SQL Statements pro Serverusecase, Anzahl der GC Major/Minor Collections, ...)
  - **Durchsatz** (z.B. Datenübertragung in KByte/s, Anzahl von gefundenen Rows pro DB-Abfrage, Anzahl von DB-Abfragen pro Sekunde, Laufzeit des GCs, ...)
  - **Latenzzeit** (z.B. Reaktion auf Benutzereingaben in msec, Netzwerk-Roundtrip Zeit in msec, Ausführungszeit einer DB-Abfrage, ...)



# Bei der Last unterscheidet man üblicherweise unterschiedliche Level

## ▪ **L1 Niedriglast**

- Wenige Benutzer, keine Systemressource an der Grenze
- Antwortzeit praktisch konstant, d.h. unabhängig von Zahl der User

## ▪ **L2 Hochlast**

- Annäherung an die Grenzbelastung des Systems
- Antwortzeit steigt stark überproportional

## ▪ **L3 Grenze des Systems**

- Fehler (Timeouts) treten auf
- Antwortzeitmessung wird sinnlos (einige Antworten kommen nie mehr)

## ▪ **L4 Grenze der ersten Ressource**

- Jede zusätzliche Belastung kann nur noch Fehler produzieren

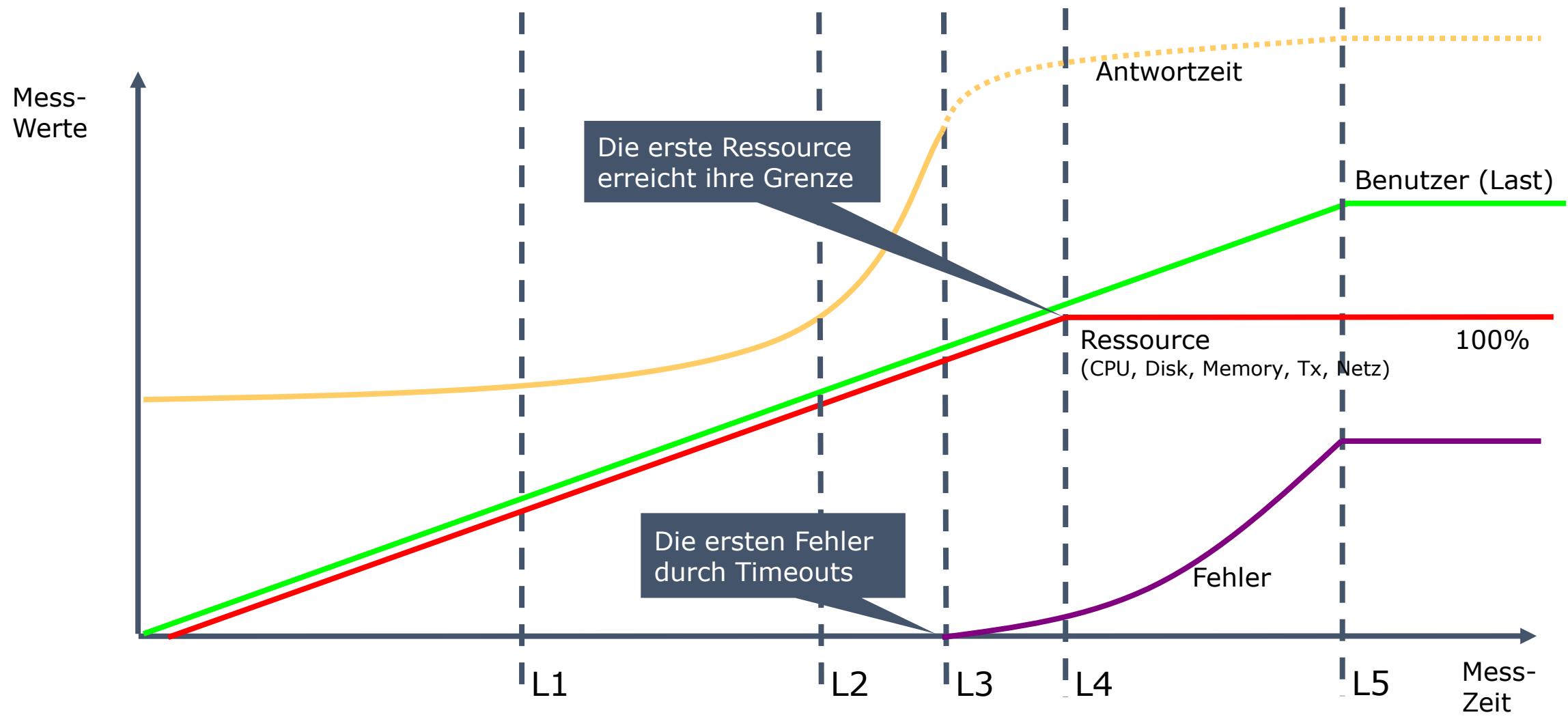
## ▪ **L5 Grenze der Lastgeneratoren**

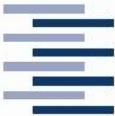
- Sollte idealerweise nicht der letzte Level sein





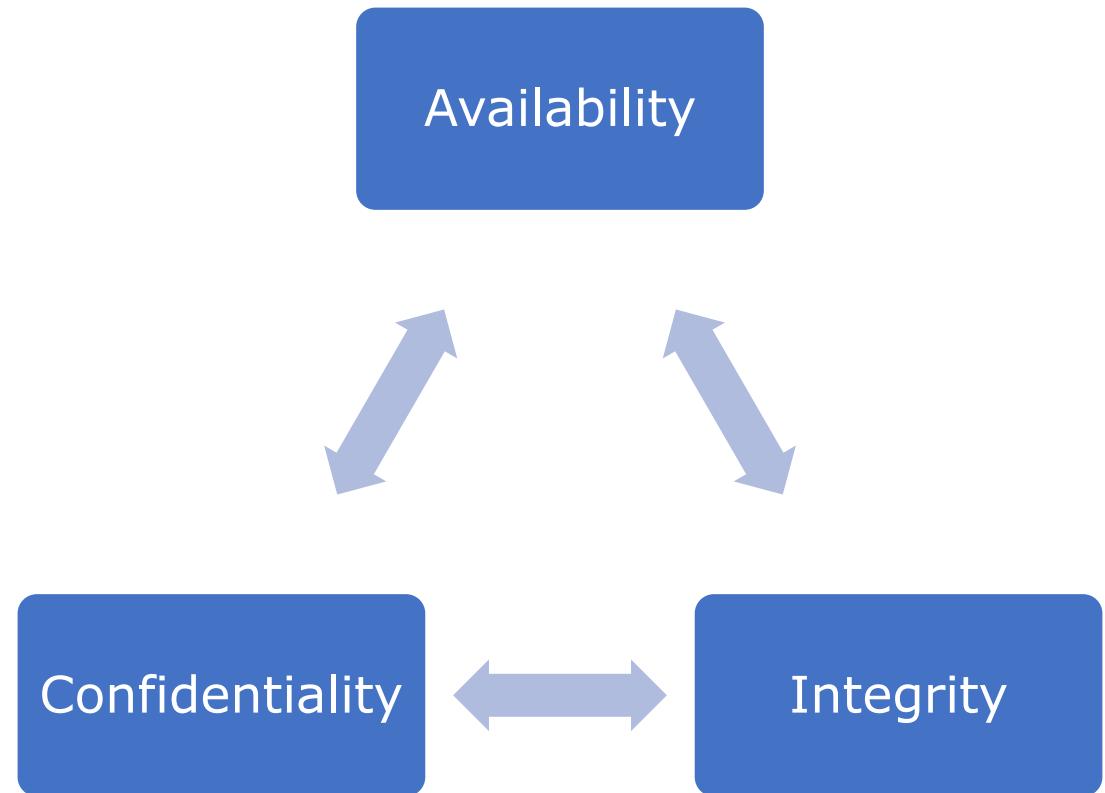
# Beispiel: Typisches Verhalten eines Systems bei kontinuierlicher Steigerung der Last

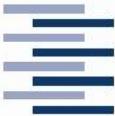




# Security Testing

- Prüfung der autorisierten Funktionalitäten typischerweise Teil des „normalen“ funktionalen Testens
- **Hier:** Aushebelung der Absicherung des Systems
- Mögliche Angriffspunkte: CIA Triade (Availability-Test überschneidet sich mit Robustheit-Tests)
- Mögliche Tests:
  - *Vulnerability Scan*: Automatische Prüfung auf bekannte Sicherheitslücken (OWASP)
  - *Penetration Test*: meist werkzeugunterstützte Ausführung von typischen Hacker-Angriffen
  - *Hacking Competitions*: Ausschreibung und Belohnung für erfolgreichen Hackversuch

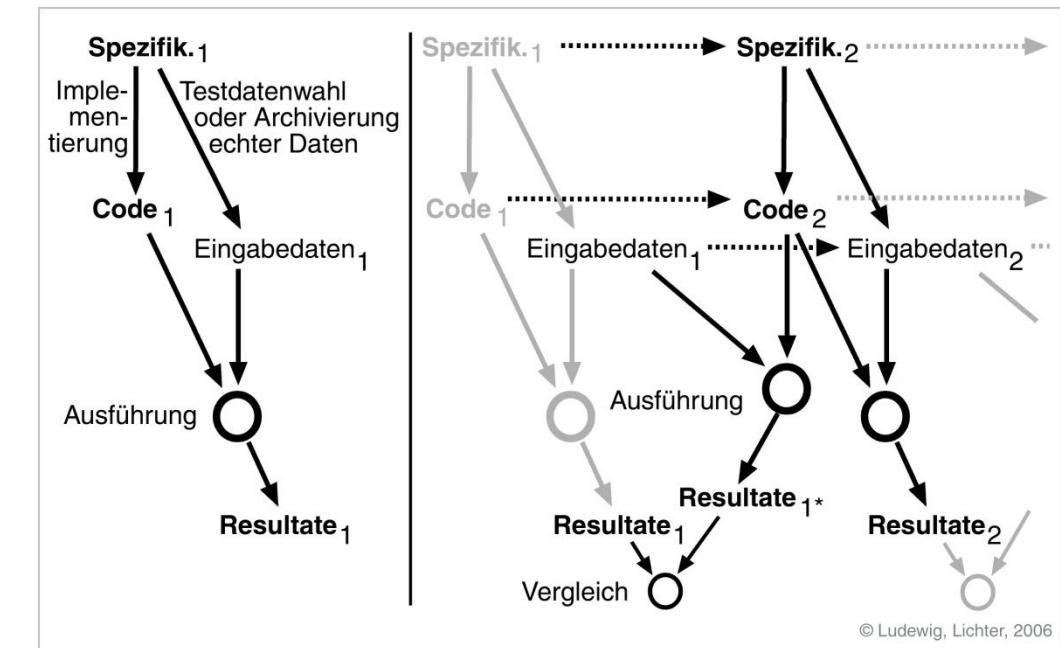




# Weiterer Begriff – Regressionstest

Selektive Wiederholung bestehender Tests um zu überprüfen, dass zwischenzeitliche Code-Modifikationen keine unerwünschten Effekte hatten und das System noch immer den Anforderungen genügt.

**Dauerhaft im Projektverlauf durch Continuous Integration!**





# Strukturierte Beschreibung von Testfällen

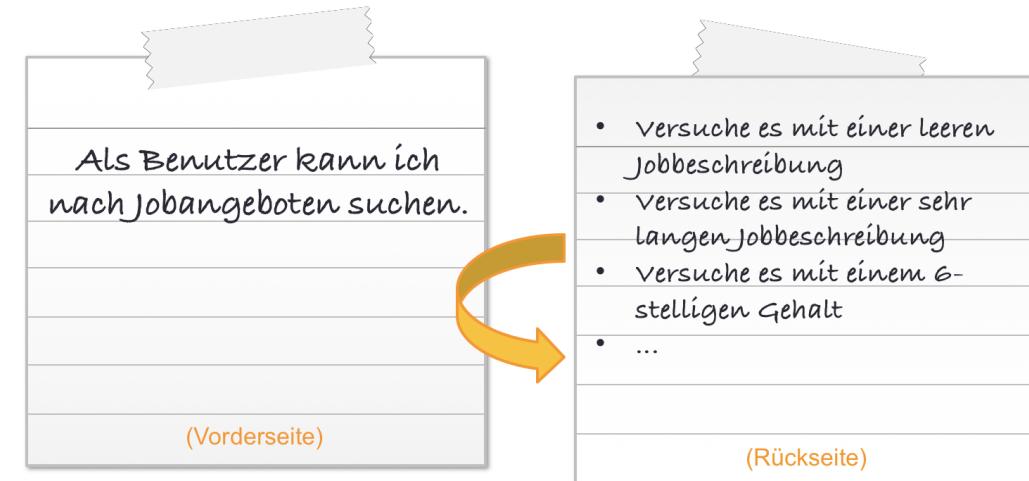
## Spezifikationsdokument (Schwergewichtig – eher Wasserfall)

<b>Name</b>	TF-31: Änderung der Telefonnummer eines Kunden
<b>Beschreibung</b>	Die Telefonnummer eines bestehenden Kunden soll geändert werden.
<b>Voraus-setzungen</b>	Die Stammdaten für den Kunden „testuser“ existieren. Die bisherige Telefonnummer lautet „0000-0000“.
<b>Eingabe</b>	Der Benutzer ändert im Dialog „Profildaten ändern“ die Telefonnummer auf den neuen Wert „1234-5678“.
<b>Ausgabe</b>	Das System meldet, dass die Änderung erfolgreich war.
<b>Verifizierung</b>	Nach der Änderung werden die Kundendaten erneut ausgelesen und die gelesene Telefonnummer mit dem erwarteten Wert verglichen.

## User Story Karten (Leichtgewichtig im agilen Umfeld)

### Akzeptanztests = Erwartungen

- **Erwartungen der Benutzer** werden durch Akzeptanztests auf den Rückseiten der Karten dokumentiert





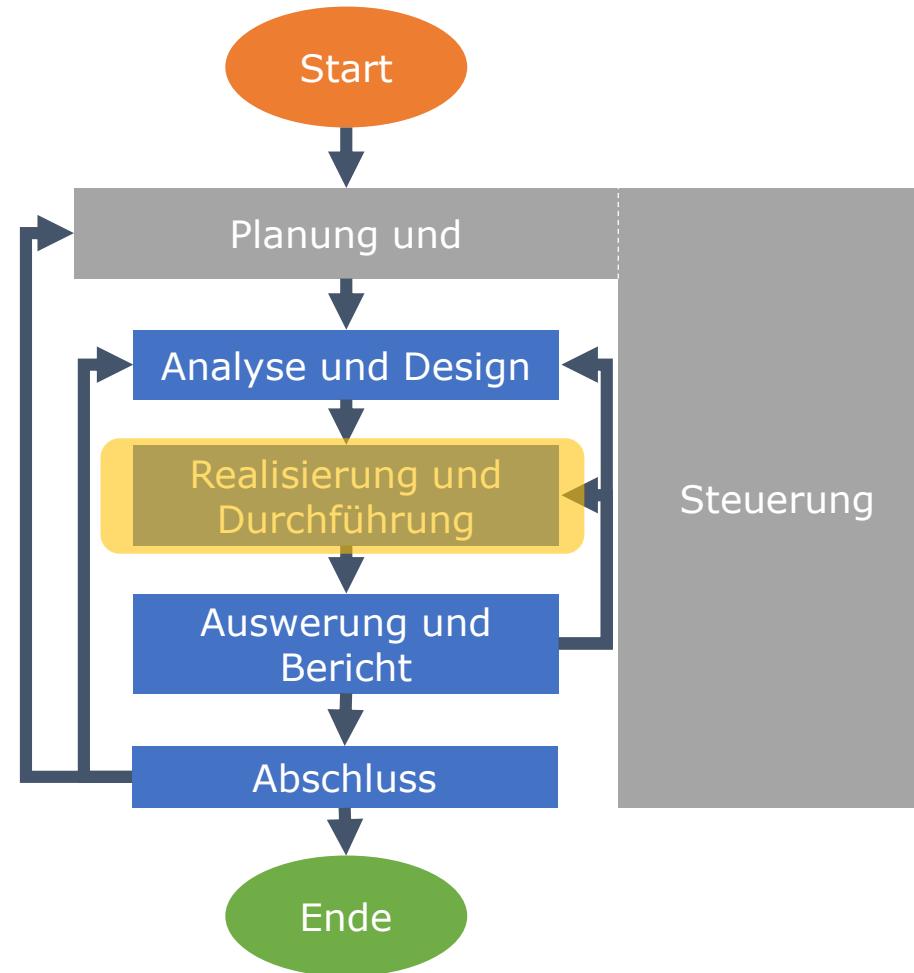
# Beschreibung von Testfällen

- Schwergewichtig vs. leichtgewichtig?
- Hängt von vielen Faktoren ab!
  - Kritikalität der Funktion
  - KnowHow des Teams
  - Kundenwunsch
  - ...
- → Guten Mittelweg finden!



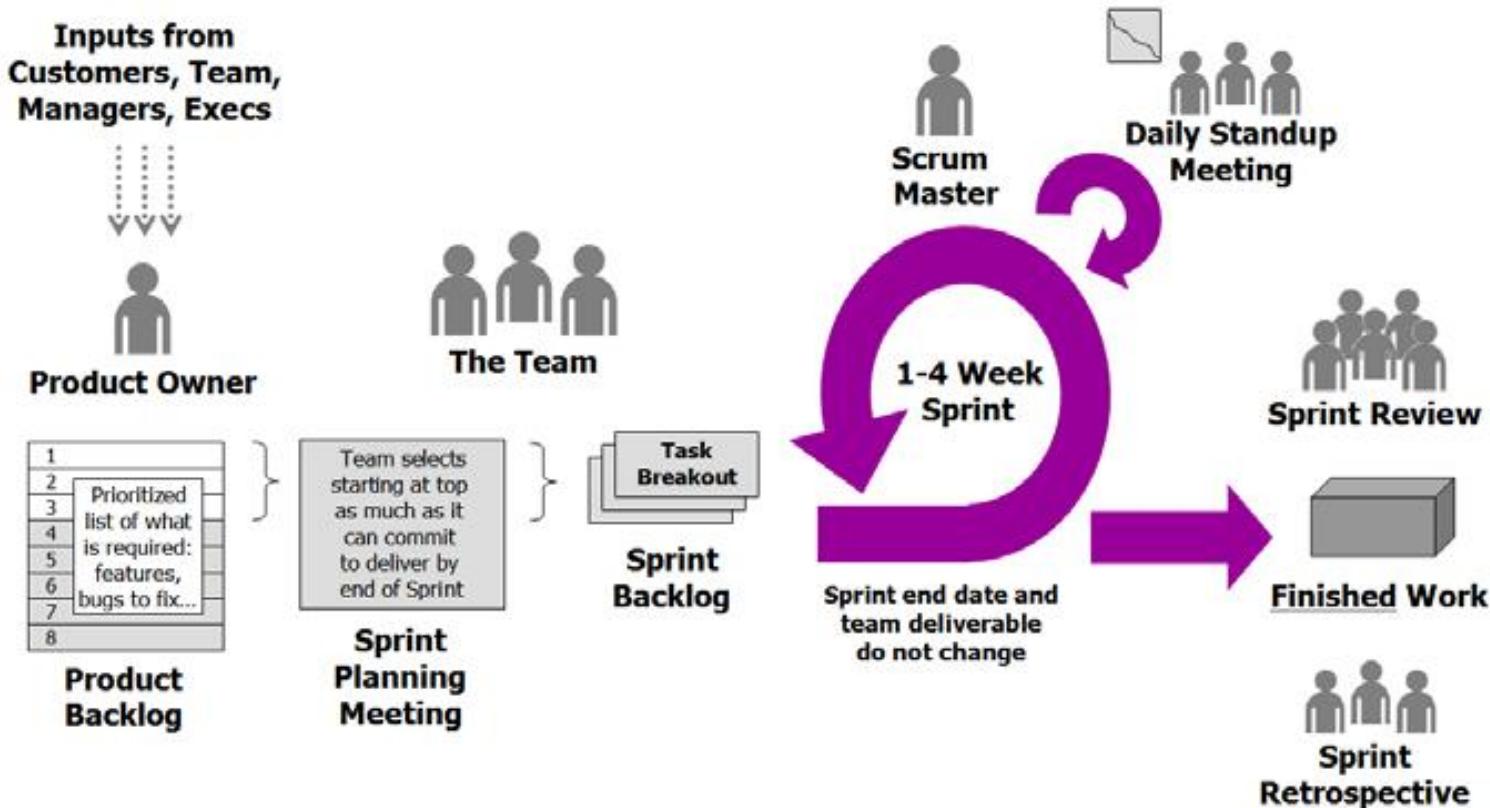


# Allgemeiner Test-Prozess nach ISTQB

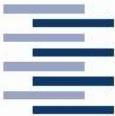




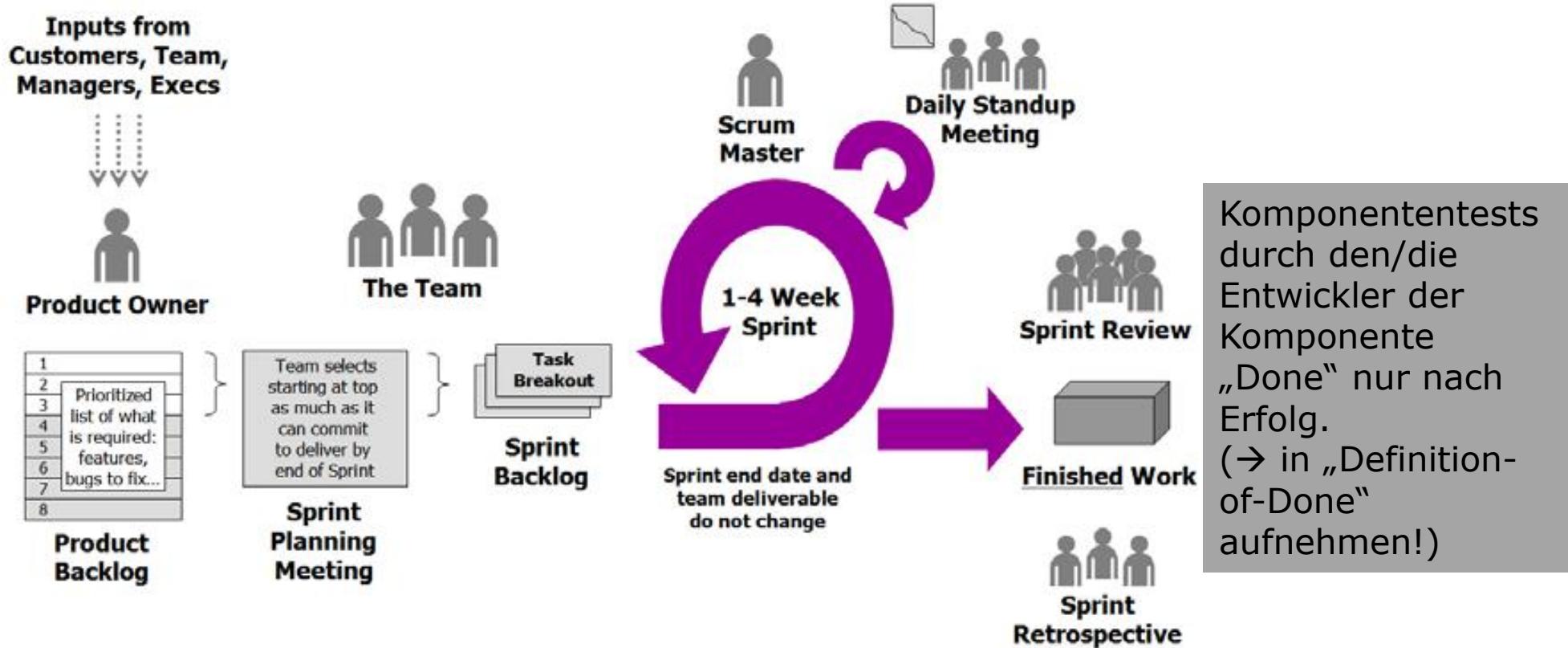
# Testdurchführung in SCRUM



*Wo in SCRUM Komponententests durchführen?  
Wo Integrations-/Systemtests?  
Und wer führt diese Tests durch?*



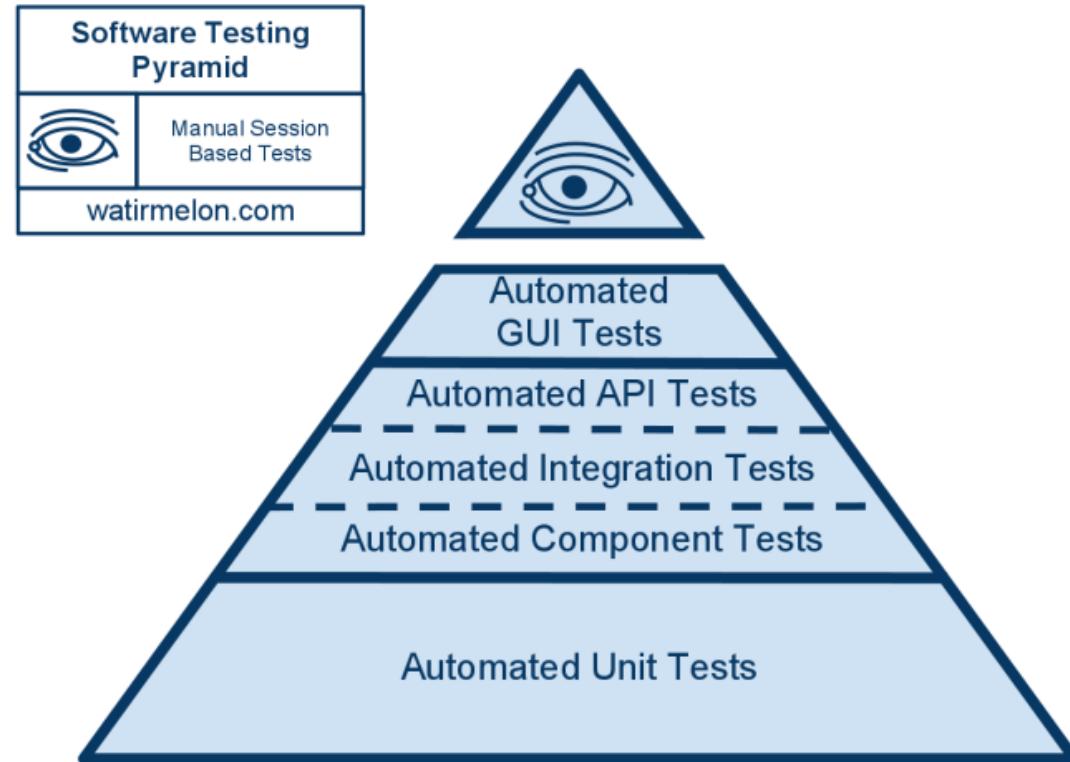
# Testdurchführung in SCRUM



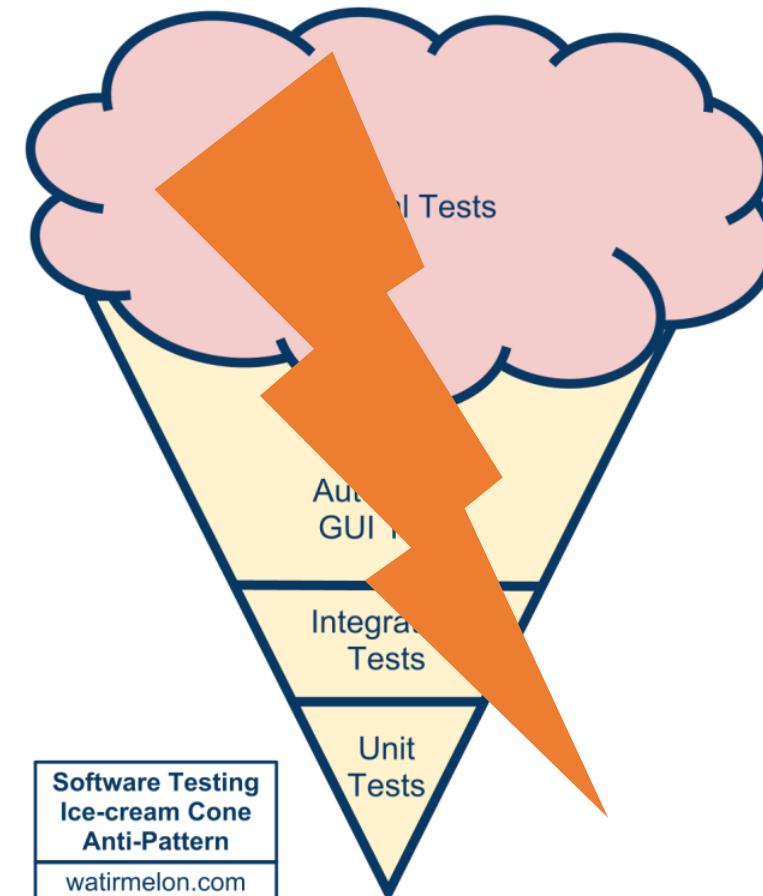
Integrationstests, Systemtests,  
Abnahmetests durch wen und wann?



# Test-Pyramide für automatisierte Test-Durchführung

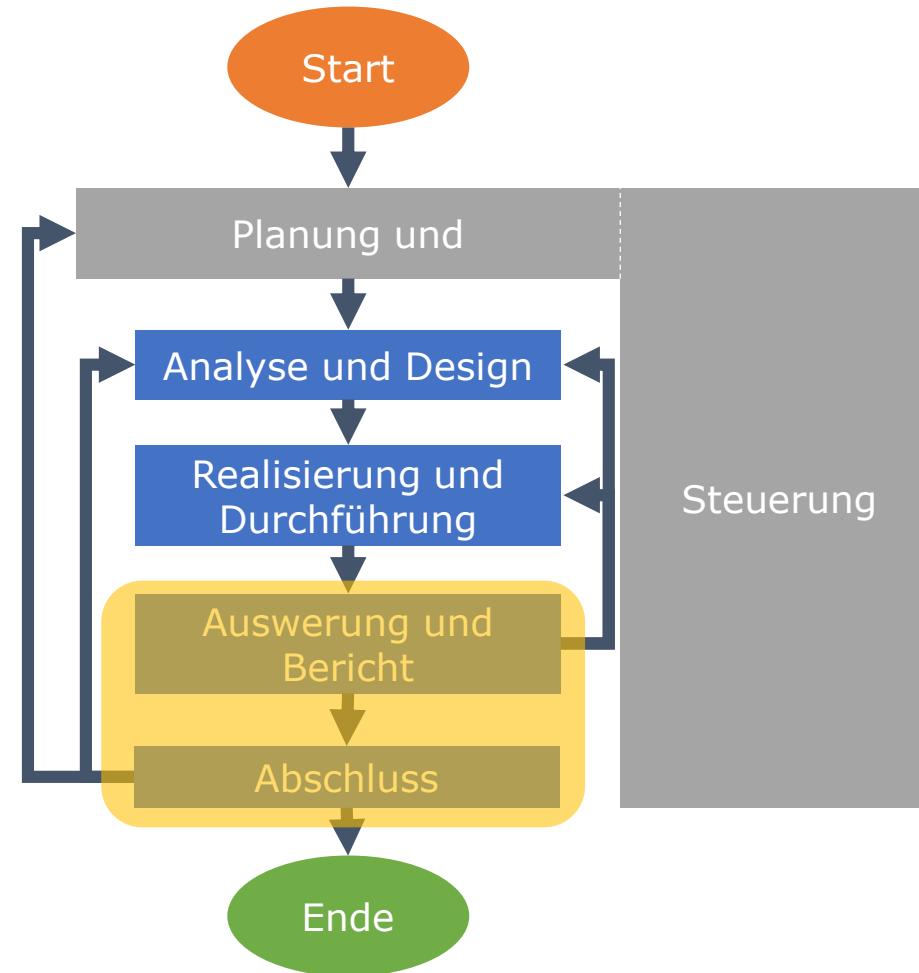


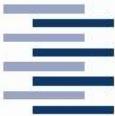
Quellen: <http://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/>  
<http://martinfowler.com/bliki/TestPyramid.html>





# Allgemeiner Test-Prozess nach ISTQB



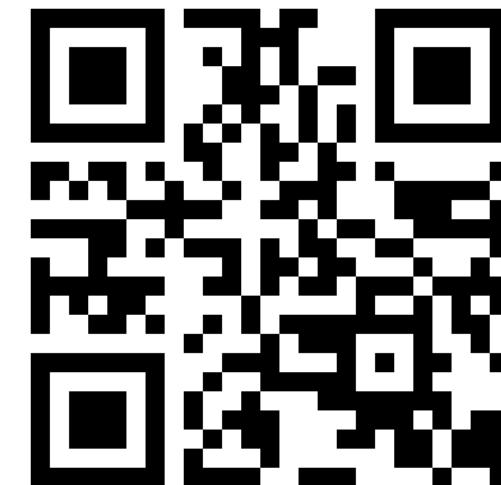


# Test-Ende – Wann ist man fertig?

- Frage sollte am Beginn des Projektes durch die Teststrategie geklärt sein
- Niemals: „*Die Zeit ist um!*“
- Stichwort Coverage (Testabdeckung)
  - *Code-Coverage*: Anteil der Anweisungen, die mindestens ein mal ausgeführt wurden.
  - *Requirements-Coverage*
  - etc.
  - Metriken können helfen (aber Achtung!)
- Um zu erkennen, dass die Überdeckung der Test-Suite ausreichend ist, sollte ein Testüberdeckungs-Tool verwendet werden (z.B. SonarQube)

Was ist ein sinnvolles Coverage-Ziel?

<http://pingo.upb.de/764286>





# Test-Ende – Code Coverage

The screenshot shows a Sonarqube interface for a Java application. On the left, a tree view displays project structure and coverage details:

- Reservation**: 55% methods, 53% lines covered.
- ReservationRepository**: 100% methods, 100% lines covered.
- facade**: 50% classes, 63% lines covered.
- moviecomponent**: 100% classes, 78% lines covered.
  - Movie**: 66% methods, 64% lines covered.
  - MovieComponent**: 100% methods, 92% lines covered.
  - MovieComponentInterface**: 100% methods, 100% lines covered.
  - MovieNotFoundException**: 100% methods, 100% lines covered.
  - MovieRepository**: 100% methods, 100% lines covered.
- util**: 100% classes, 71% lines covered.
  - Application**: 66% methods, 77% lines covered.
  - SecurityWebApplicationInitializer**: 0% methods, 0% lines covered.
  - WebSecurityConfig**: 0% methods, 0% lines covered.

On the right, the **increaseReservationStatistics** method is shown with its code and coverage status:

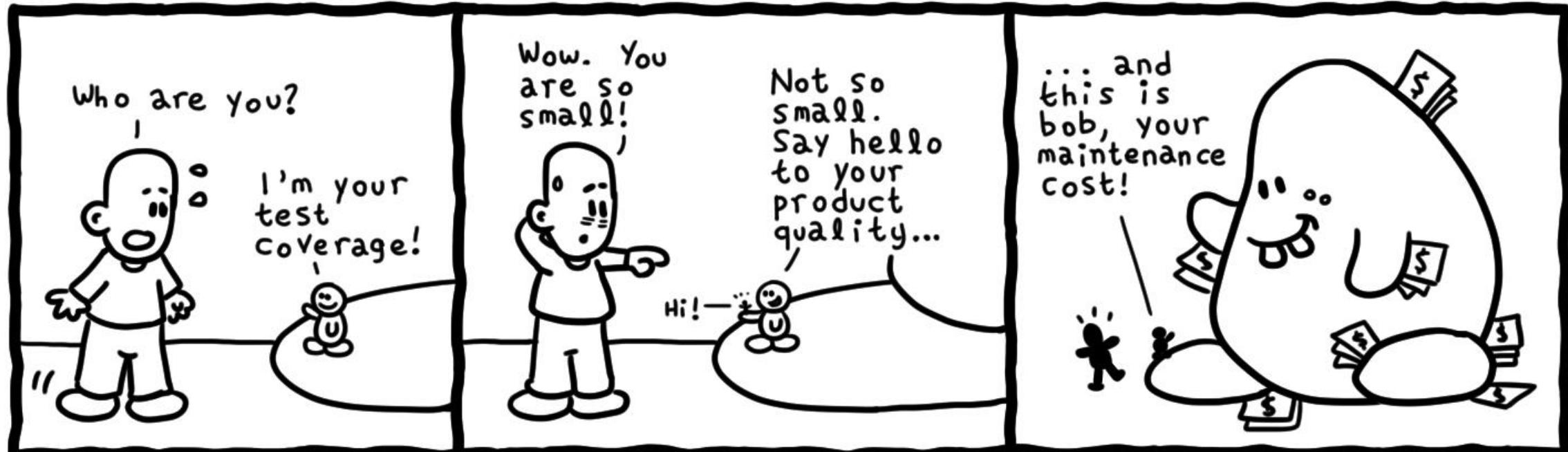
```
    }  
    return movie.getNumberOfReservations();  
  
    @Override  
    public void increaseReservationStatistics(String movieTitle) throws  
        Movie movie = movieRepository.findByTitle(movieTitle);  
        if (movie == null)  
        {  
            throw new MovieNotFoundException(movieTitle);  
        }  
  
        movie.increaseReservationStatistics();  
        movieRepository.save(movie);  
    }  
}
```

A vertical bar chart indicates coverage for each line of code. Below the code, a "Debt" section notes "started 4 days ago".

**Coverage** →

71.4%

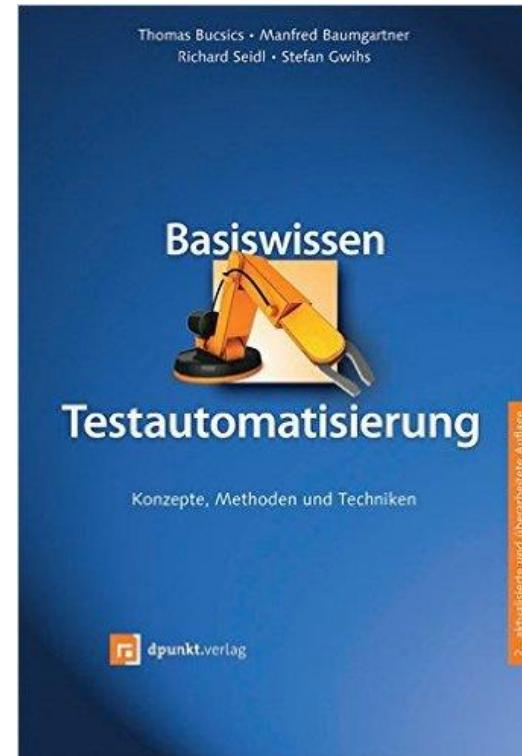
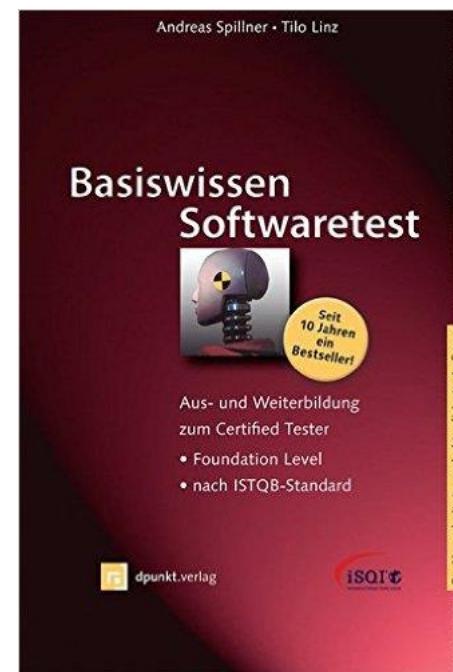
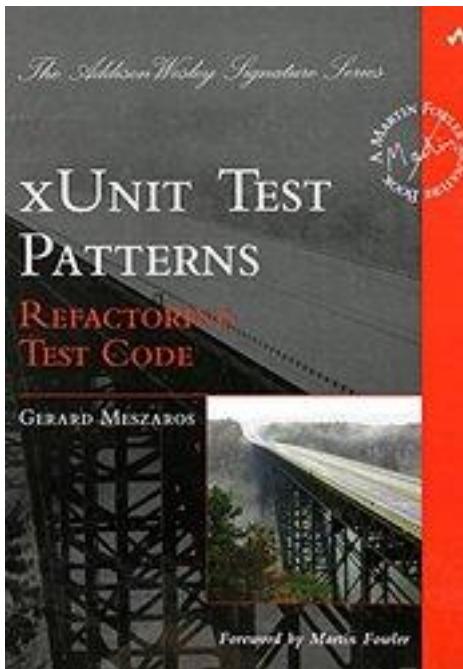
Coverage



Daniel Stori {turnoff.us}

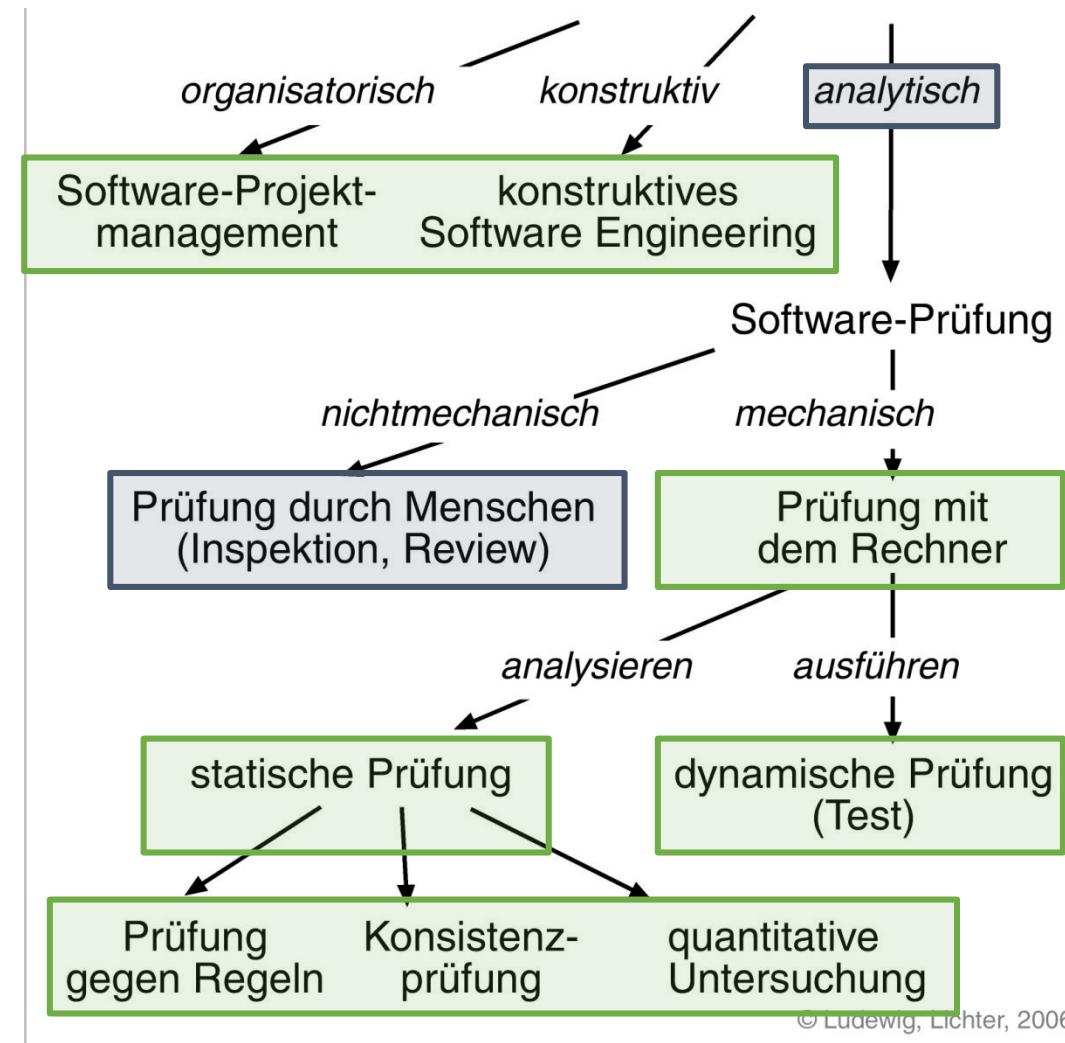


# Tests – Literatur





# Arten von Software-Qualitätssicherung





# Nichtmechanische Prüfungsarten

- **Review:** Prüfung eines Prüflings (Dokument, Zeichnung, Programm) durch mehrere Gutachter gegenüber Vorgaben und gültigen Richtlinien mit dem Ziel
  - Fehler und Schwächen aufzuzeigen
  - Positive Merkmale zu würdigen
- Eine **Inspektion** ist ein Review, bei dem der Prüfling von den Gutachtern systematisch Punkt für Punkt auf Stärken und Schwächen abgeklopft wird.
- Ein **Walkthrough** ist ein Review, bei dem der Autor oder die Autorin die Funktionsweise des Prüflings Schritt für Schritt beschreibt, während die Gutachter aufmerksam zuhören und überall einhaken, wo sie Mängel entdecken





# Review – Begriffe und Rollen

## Begriffe

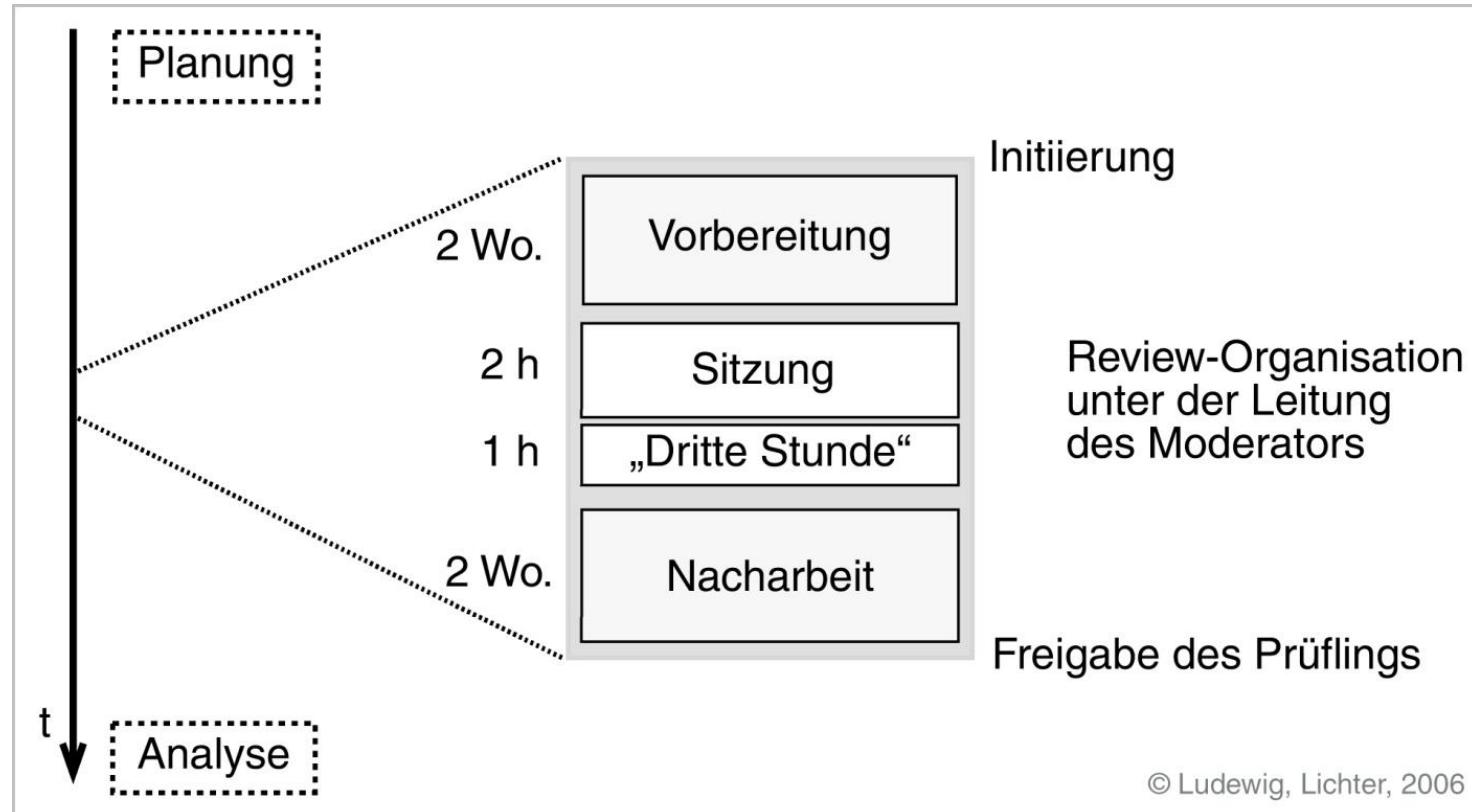
- **Prüfling**  
In sich abgeschlossener, für Menschen lesbarer Teil von Software (Dokument, Codemodul, Datenmodul)
- **Referenzunterlagen**  
Spezifikation, Richtlinien, Fragenkataloge

## Rollen

- **Manager/PL** (nimmt üblicherweise nicht teil)  
Vorgesetzter (verantwortlich für Auftrag zur Prüfung und Freigabe)
- **Autor** (hat nur passive Rolle für Rückfragen)  
Urheber des Prüflings (evtl. Repräsentant des Teams)
- **Moderator**  
Leiter der Review-Sitzung
- **Gutachter**  
Kollege, der den Prüfling kompetent beurteilen kann
- **Protokollant**  
Kollege, der protokolliert (kann von Autor oder Moderator übernommen werden)
- **Review-Team**  
Alle am Review beteiligten Personen



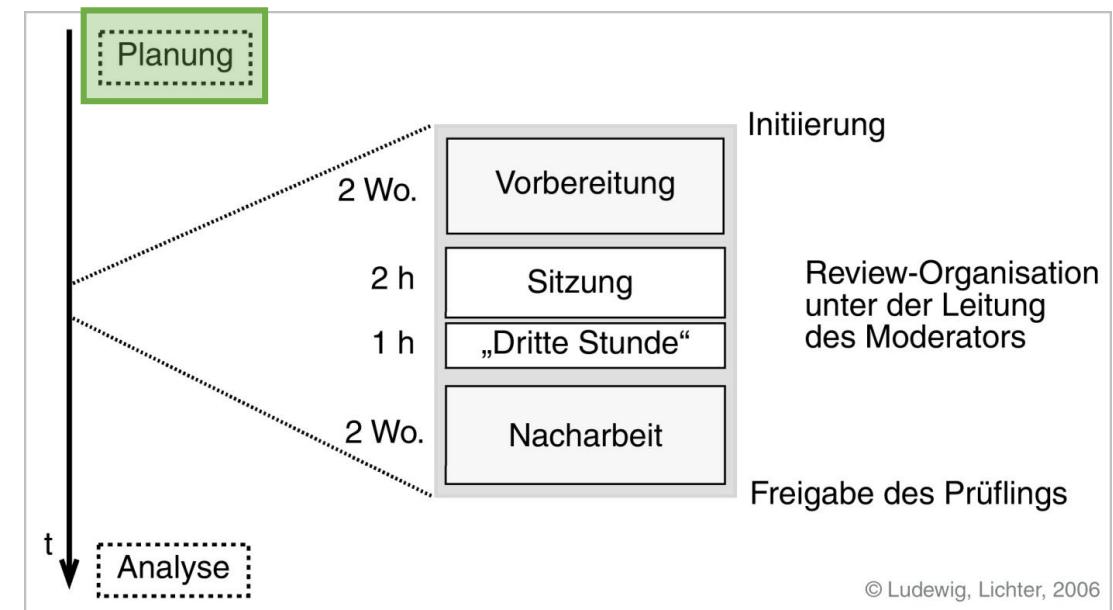
# Review - Ablauf





# Review – Ablauf / Planung

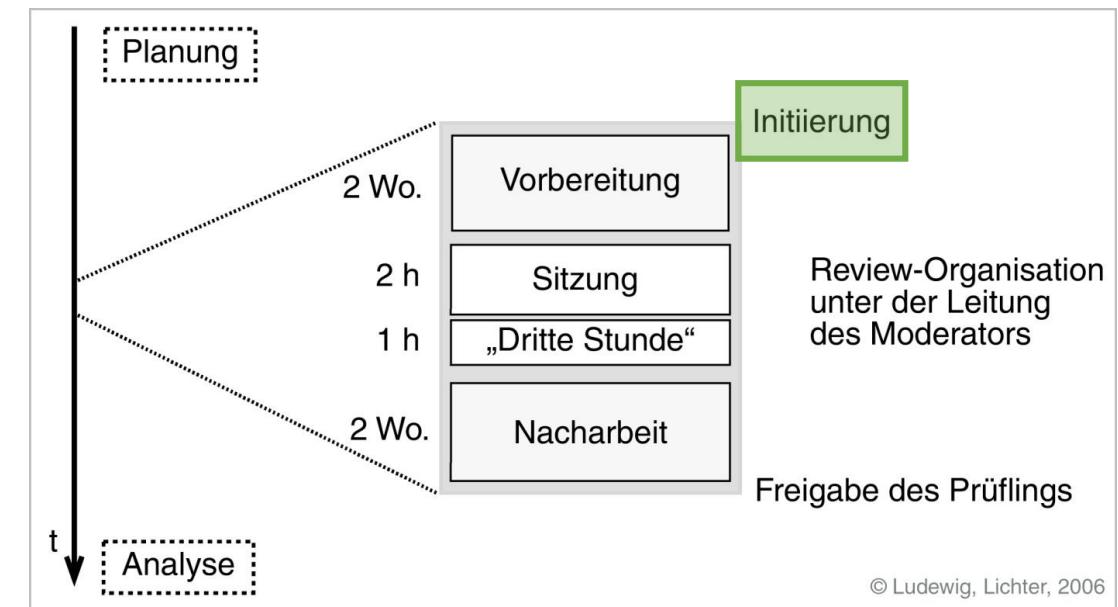
- Festlegung des Review-Teams
- Einplanung entsprechender Zeit für Reviews (bis 20% des Erstellungsaufwands) in den Arbeitspaketen des Projekts
- Festlegung der Review Termine





# Review – Ablauf / Initiierung

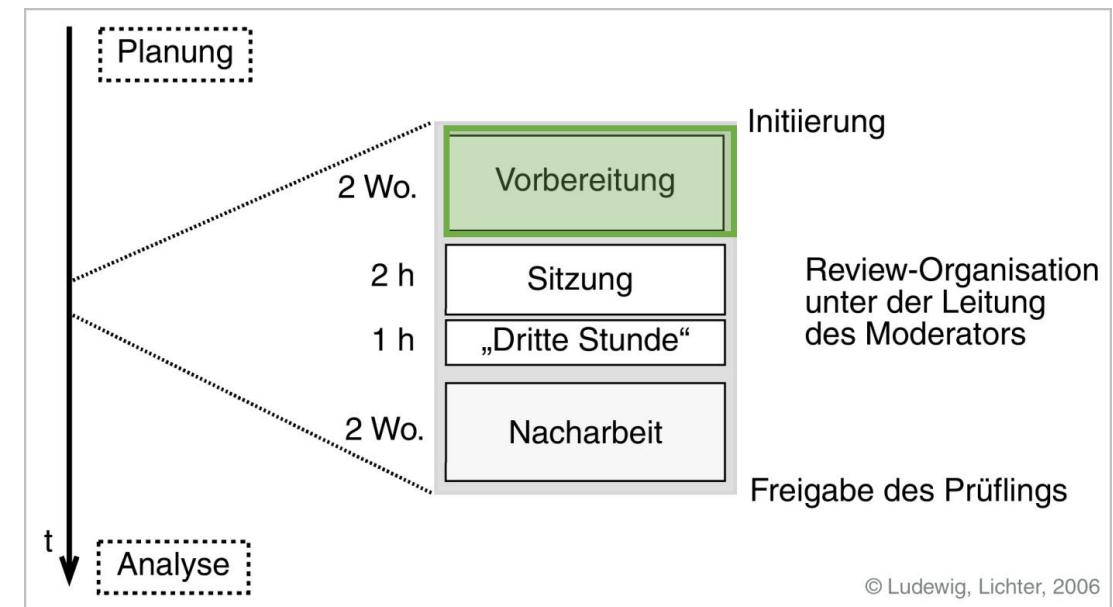
- Moderator versorgt Gutachter mit erforderlicher Information
  - Durch schriftliche Einladung
  - Im Rahmen einer Einführungssitzung
- Wichtig (unabhängige Beurteilung, Prüfung aus verschiedenen Blickwinkeln)
  - Jedem Gutachter kann mehr als ein Aspekt zugewiesen sein
  - Jeder Aspekt muss von mindestens zwei Gutachtern bearbeitet werden





# Review – Ablauf / Vorbereitung

- Vorbereitung
  - Gutachter „durchleuchten“ Prüfling nach vorgegebenen Aspekten (Fragenkataloge, Checklisten)
    - Erledigen Trivialfehler durch entsprechende Markierung des Prüflings
    - Erstellen Mängellisten
  - Moderator muss sicherstellen, dass Gutachter alle Rahmenbedingungen (Zeit, Unterlagen) vorfinden, um ihre Aufgabe zu erledigen (evtl. Abbruch des Reviews)
- Fragenkataloge
  - Ja/nein-Fragen (auf der Grundlage von Richtlinien und Erfahrung)
  - Präzise Fragestellung (erhöht Wahrscheinlichkeit gehaltvoller Antworten)
- Verschieden Typen von Fragenkatalogen für Dokumente/Code
  - Standard-Fragenkatalog (gemeinsame Eigenschaften aller Dokumente: Kennzeichnung, Layout, Aufbau, Textgestaltung; bzw. Kommentierung, Programmier-Richtlinien)
  - Standard-Fragenkataloge für verschiedene Dokumentarten und Programmiersprachen
  - Projekt- oder Dokumentspezifischer Fragenkatalog





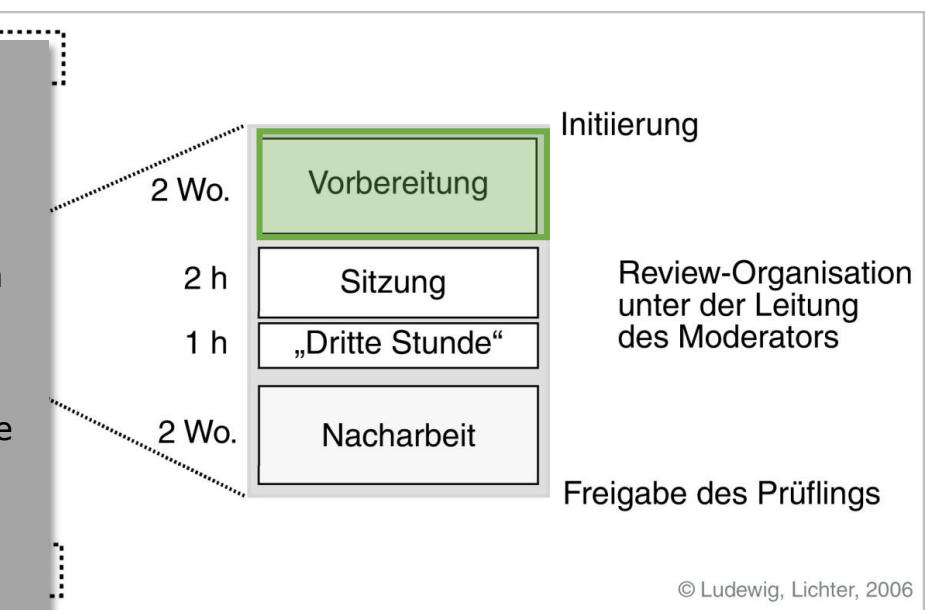
# Review – Ablauf / Vorbereitung

- Beispiele für Checklisten
  - Analyse
  - Spezifikation
  - Entwurf
  - Code
  - Test
  - ...

## Beispiel: Mögliche Checkliste für Anforderungen

1. Kann man für jede Anforderung zweifelsfrei feststellen, ob sie im fertigen System erfüllt ist oder nicht?
2. Sind alle wesentlichen Begriffe in den Anforderungen klar definiert?
3. Wurden die künftigen Benutzer des Systems in die Anforderungsklärung einbezogen?
4. Versteckt sich hinter einer Anforderung vielleicht eine Lösung?
5. Wurden Konflikte zwischen Anforderungen herausgearbeitet und ausbalanciert?

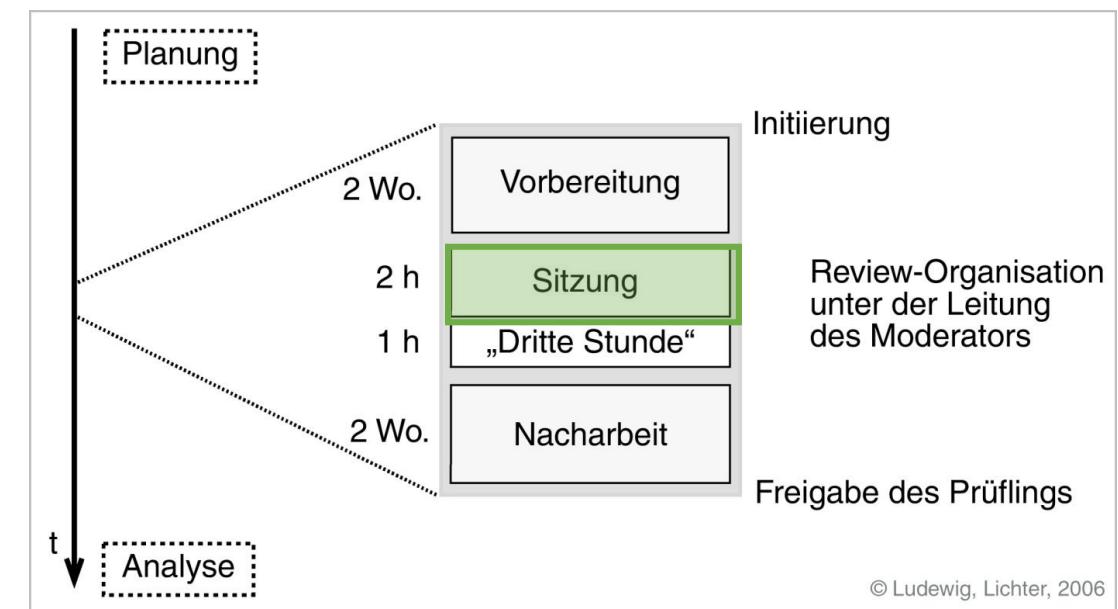
u.v.m.





# Review – Ablauf / Sitzung

- Geleitet durch Moderator (nach gewissen „Spielregeln“)
- Abschnittsweise (Seiten, Kapitel) Behandlung des Prüflings (jeder Gutachter kommt zu Wort)
- Gutachter berichten präzise und prägnant über jeweilige Befunde
- Protokollant protokolliert Befunde
- Autor hat ausschließlich passive Rolle
- Moderator versucht Konsens zu erreichen





# Mögliche Form eines Review-Protokolls

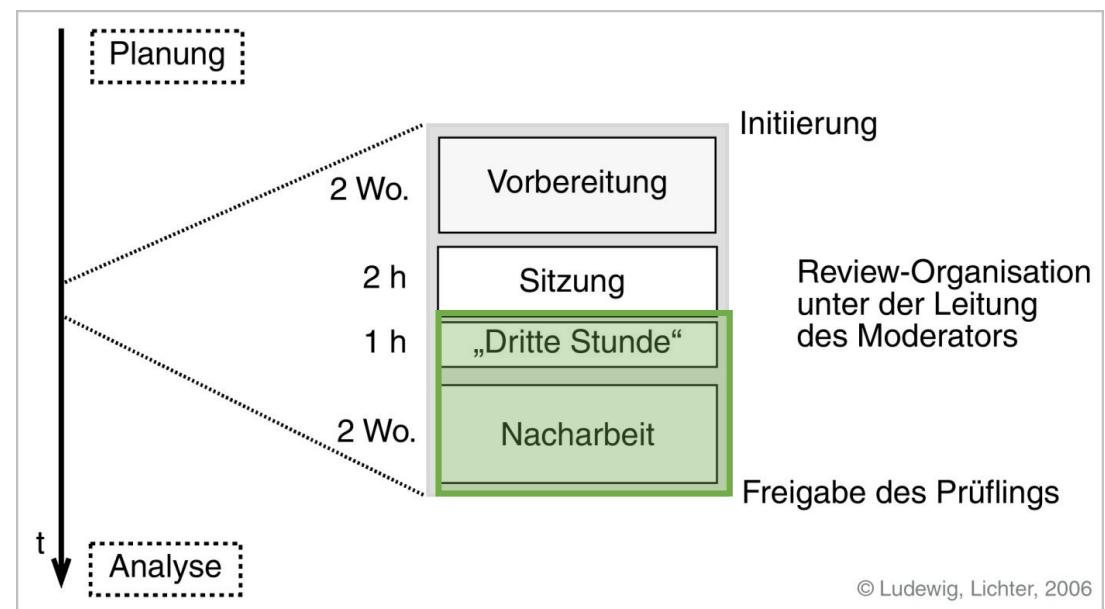
Review-Protokoll			
1. Review			
Review-Name/Id.: _____ Datum: _____ Beginn: _____ Ende: _____			
2. Prüfgegenstand/Prüfgegenstände			
Version	Seiten	Titel	
3. Referenzunterlagen			
Version	Seiten	Titel	
4. Zusammenfassung/Ergebnis			Identifizierte Mängel <input type="checkbox"/> kritisch (K) <input type="checkbox"/> bedeutend (B) <input type="checkbox"/> unbedeutend (U)
5. Empfehlung			
<input checked="" type="checkbox"/>	Keine nennenswerten Mängel erkannt.		
<input type="checkbox"/>	Geringe Mängel vorhanden - Behebung erforderlich.		
<input checked="" type="checkbox"/>	Schwerwiegende Mängel - Behebung wird in Folge-Review kontrolliert.		
6. Anhänge			
Dokument		Seiten	
Mängelliste für			
Mängelliste für			
Mängelliste für			
7. Review-Teilnehmer			
Name (Firma/Abteilung)	Rolle	Datum	Unterschrift
	Autor		
	Moderator		
	Inspektor		

Mängelliste			
Dokument: _____			
Review-Name/Id.: _____			
Lfd.-Nr.	Inspektor	Position	Beschreibung
1.			
2.			
3.			
Kritikalität K = kritisch B = bedeutend U = unbedeutend			

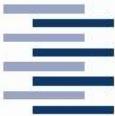


# Review – Ablauf / Dritte Stunde und Nacharbeit

- “Dritte Stunde”
  - ist optional
  - Informeller Rahmen (Kaffee, Essen)
  - Möglichkeit über Lösungen zu diskutieren
  - Manöverkritik
- Nacharbeit
  - Manager muss auf der Grundlage des Review-Berichts Entscheidung fällen (Review-Team erarbeitet nur Empfehlung)
  - Autor muss ggf. Prüfling überarbeiten (nur bezüglich negativer Befunde)
  - Prüfung des überarbeiteten Prüflings
  - Kleine Mängel: Kontrolle durch Moderator
  - Große Mängel: neues Review (durch dasselbe Team: Kontinuität)



© Ludewig, Licher, 2006



# Review – Ablauf / Analyse

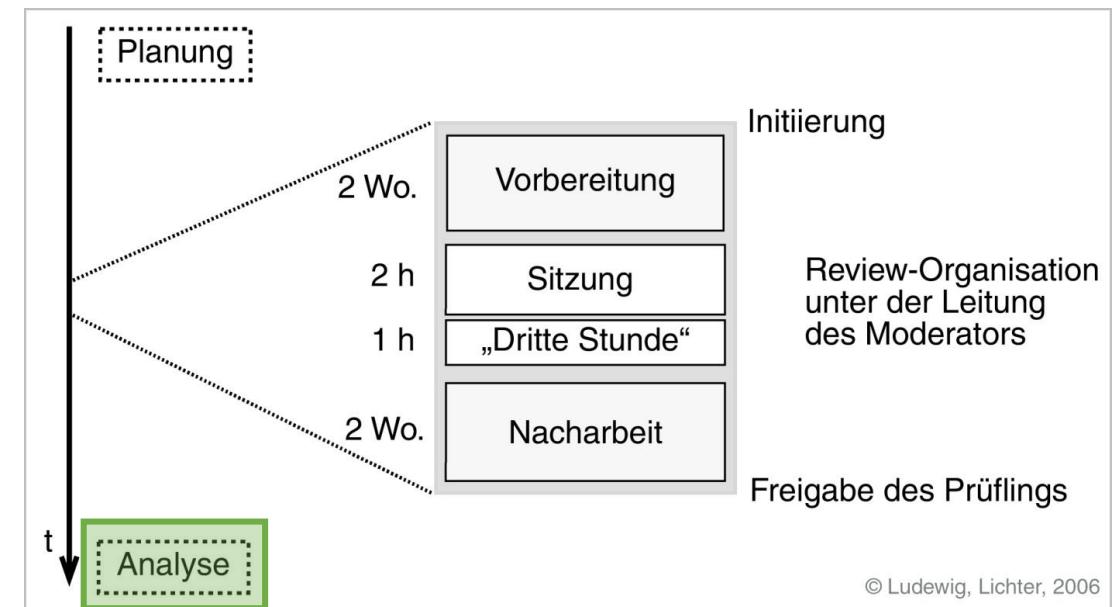
## ▪ **Kurzfristiger Nutzen**

Implizite Schulung der Mitglieder des Review-Teams

## ▪ **Langfristiger Nutzen**

durch systematische Auswertung der Resultate, z. B.

- Einfluss verschiedener Parameter (Organisationsstruktur, Werkzeugeinsatz, Programmiersprache) durch Quervergleich (mehrerer Projekte)
- Einfluss von früher erkannten Problemen auf Projektverlauf durch Längsvergleich (in einem Projekt)
- Analyse der Wirksamkeit von Änderungen im Entwicklungsablauf durch Längsvergleich (über mehrere Projekte)





# Review – Stärken und Schwächen

## Stärken

- Können früh im Entwicklungsprozess durchgeführt werden (auch in „frühen Phasen“)
- Alle Entwicklungsergebnisse prüfbar (nicht nur Code)
- Selbsteinschätzung der Entwickler wird relativiert
- Hohe Wirksamkeit

## Schwächen

- Gekaufte Software nur teilweise für Reviews zugänglich
- Können Bedrohung für den Einzelnen darstellen (vom Autor zum „Angeklagten“)
- Können Gräben im Team aufreißen (bei unterschiedlichen Wertmaßstäben)
- Sind nicht billig (ca. 15% des reinen Erstellungsaufwands) – aber lohnend



# Inspektionen und Walkthroughs

## Code-Inspektion

- Das intensive **Durcharbeiten** („Zeile für Zeile“) eines Einzelergebnisses durch einen Teamkollegen nennt man Inspektion.
- Es hat im Gegensatz zu einem Review **informellen Charakter** und ist keine moderierte Veranstaltung.
- Inspektionen setzt man **vorwiegend zur Begutachtung von Sourcecode** („Code-Inspektion“) und von Ergebnissen fertiger Software ein.

## Walkthroughs

- Bei Walkthroughs wird die Funktionalität des Prüfgegenstandes **anhand von Beispielen und Testfällen** (Szenarien) durchgespielt.
- Walkthroughs eignen sich unter anderem auch besonders zur **Ausbildung von Mitarbeitern** und fördern die Teamkommunikation.
  - Sie regen zu lebhafter Diskussion an und damit zu einer hohen Interaktion zwischen Vortragendem und den Teilnehmern.
- Diese Technik wird vor allem zur **Begutachtung von technischen Konzepten** (und alternativer Lösungswege) und Spezifikationen eingesetzt.



# Übungsaufgabe

- 🎯 **Ziel:** Führen Sie eine Inspektion auf den nachfolgenden Code durch.

- 👥 Zweier Teams

- ⌚ 10 Minuten





# Code-Beispiel für die Übung

```
package de.haw.demo;

import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class Account {

    public String accountNumber;

    public Account(String accountNumber){
        // Constructor
        this.accountNumber = accountNumber;
    }

    public String getAccountNumber(){
        return accountNumber; // return the account number
    }

    // Override the equals method
    public boolean equals(Account o) {
        return o.getAccountNumber() == getAccountNumber(); // check
account numbers are the same
    }
}
```

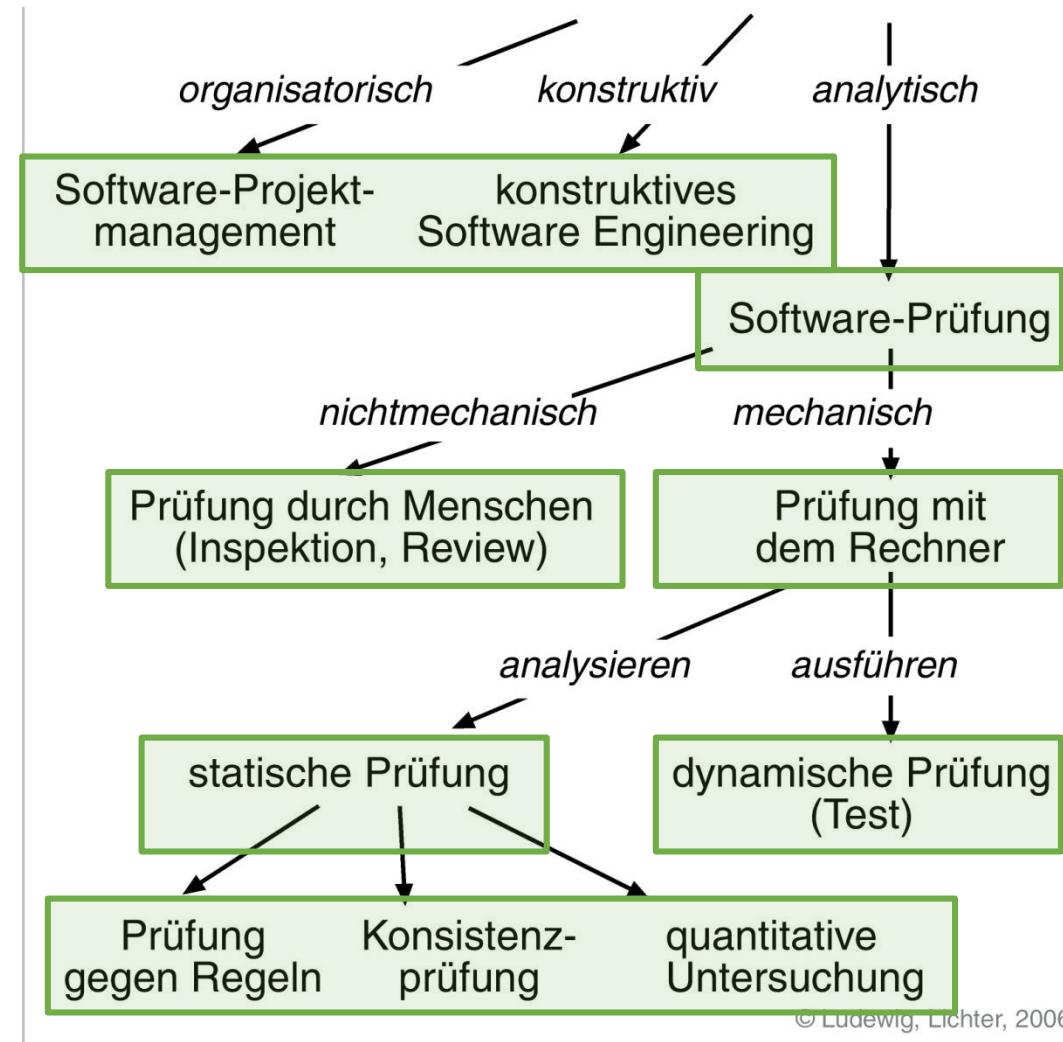
[continued on the left]

```
public ArrayList getTransactions() throws Exception{
    try{
        //Get the list of transactions
        List dbTransactionList = Db.getTransactions(
            accountNumber.trim());
        ArrayList transactionList = new ArrayList();
        int i;
        for(i=0; i<dbTransactionList.size(); i++){
            DbRow dbRow = (DbRow) dbTransactionList.get(i);
            Transaction trans = makeTransactionFromDbRow(dbRow);
            transactionList.add(trans);
        }
        return transactionList;
    } catch (SQLException ex){
        // There was a database error
        throw new Exception("Can't retrieve transactions");
    }
}

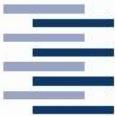
public Transaction makeTransactionFromDbRow(DbRow row){
    double currencyAmountInPounds = Double.parseDouble(
        row.getValueForField("amt"));
    String description = row.getValueForField("desc");
    // return the new Transaction object
    return new Transaction(description, currencyAmountInPounds);
}
```



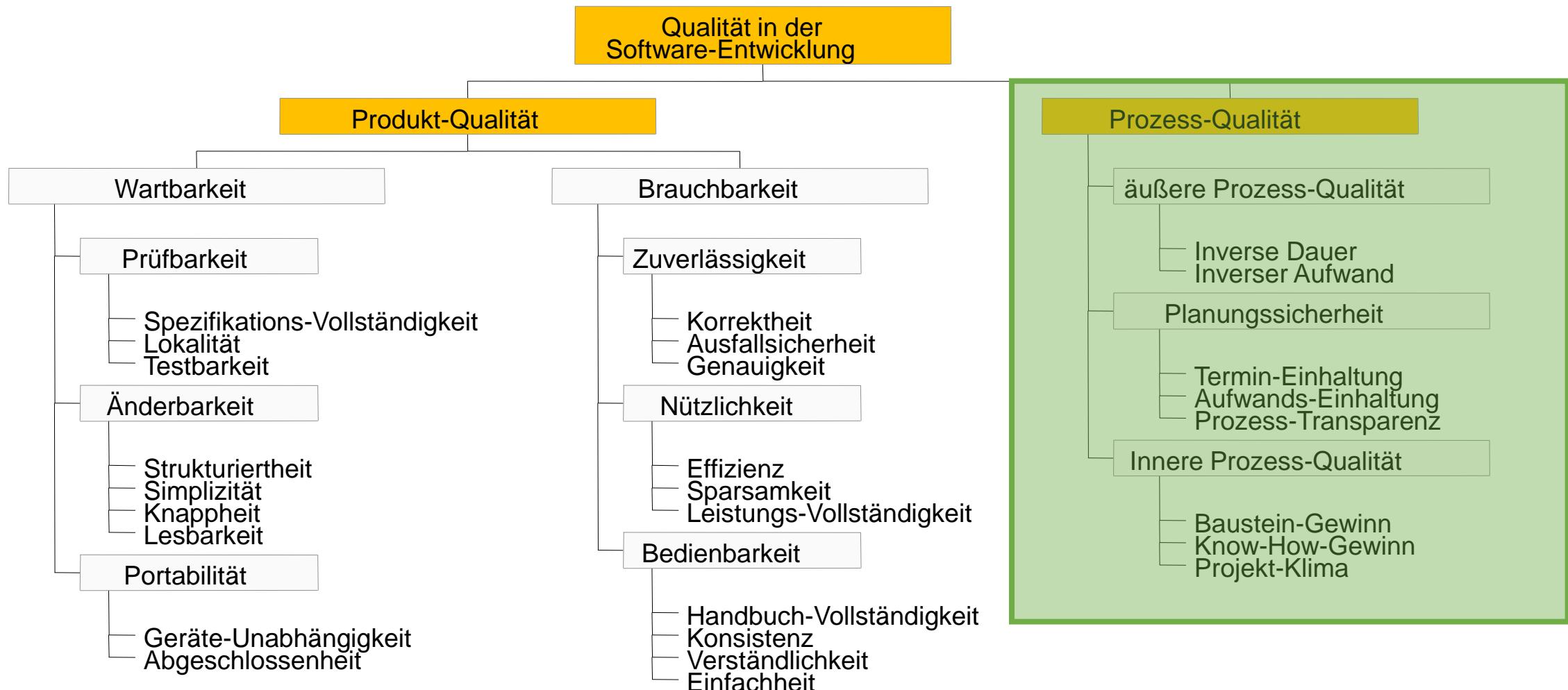
# Arten von Software-Qualitätssicherung – Fertig?



© Ludewig, Licher, 2006



# Qualität – Merkmale nach Boehm





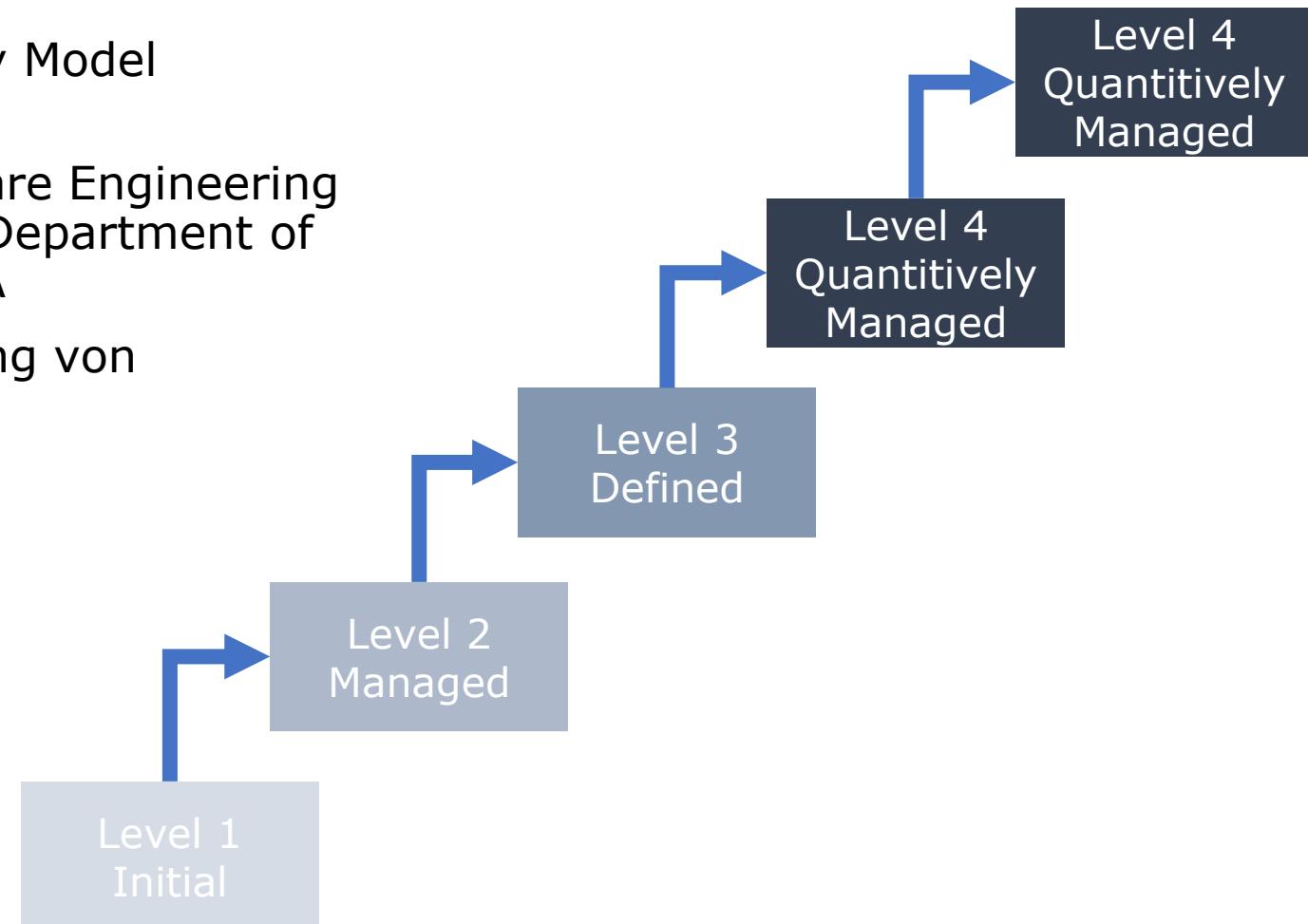
# Bewertung und Verbesserung von Prozessen

- Seit Beginn des 20. Jahrhunderts ist die Verbesserung der Prozesse in der Automobilindustrie gängige Praxis
- Großer Einfluss auf Produktivität und Qualität der Produkte
- Übertragen der Grundideen auf die Softwareentwicklung
- 80er: verschiedene Ansätze zur Bewertung und Verbesserung von **Software-Prozessen**
  - CMMI
  - SPICE
  - ISO-9000 (nicht Software-spezifisch)
- Perfekter Prozess ist keine Garantie für eine gute Entwicklung
- Ein **Entwicklungsprozess** sollte ein Projekt in allen Punkten, die einheitlich organisiert und durchgeführt werden sollten, festlegen
- **Sprachen, Methoden und Werkzeuge** müssen die ausgewählten Tätigkeiten gut unterstützen
- **Mitarbeiter** müssen entsprechend des Prozesses und der Methoden/Werkzeuge ausgebildet sein
- **Im Folgenden:**  
Prozessbewertungsmethode CMMI



# Prozessbewertung – CMMI

- **CMMI:** Capability Maturity Model Integrated (von 1997)
- Entwickelt am SEI (Software Engineering Institute) im Auftrag des Department of Defence (DoD) in den USA
- Verfahren zur Durchführung von Prozessbewertungen





# CMMI - Grundprinzipien

- **Sichtbarmachen der Schwächen** des gelebten Prozesses durch Checklisten
- Bewertung **neutral** gegenüber der eingesetzten Technik (z. B. OO)
- **Höhere Stufe** setzt alle niedrigen Stufen voraus
- **Hohe Prozessreife** bedeutet
  - Prozesse sind besser beschrieben, geplant und gesteuert/kontrolliert
  - Statistische Kontrolle (Kennzahlen) sind vorhanden
  - Termine, Kosten, Qualität kann besser prognostiziert werden
- CMMI definiert 22 **Prozessbereiche** (process areas) mit entsprechenden Zielen zur Strukturierung, z. B.
  - Requirements Management (REQM)
  - Project Planning (PP)
  - Organizational Training (OT)
  - Causal Analysis and Resolution (CAR)



# Prozessbewertung – CMMI

Stufe	Beschreibung
1 <b>(initial)</b>	Prozess <b>nicht bewusst</b> gestaltet, sondern <b>zufällig</b> entwickelt. Dadurch meist Planung unzulänglich, Überschreitung von Kosten und Terminen. Erfolg ist nicht planbar und hängt von einzelnen Mitarbeitern ab.
2 <b>(managed)</b>	Vorgaben für <b>wichtige Bereiche</b> (Anforderungen, QS, PM). Kein definierter Software-Prozess. Jedes Projekt kann einem eigenen Prozess folgen.
3 <b>(defined)</b>	Alle Projekte einer Organisation folgen einem einheitlichen Schema. <b>Definition eines Standardprozesses</b> . Freiheiten für Projekte.
4 <b>(quantitatively managed)</b>	Einführung <b>einheitlicher Metriken</b> zur Erkennung von Problemen.
5 <b>(optimizing)</b>	<b>Stetige</b> technische und organisatorische <b>Verbesserung</b> der Prozesse und Anpassung an neue Randbedingungen. Systematische Analyse von Fehlern und Problemen



# Agenda

- Einführung
- Qualitäts-Merkmale
- Metriken
- Maßnahmen
- **Zusammenfassung**



# Zusammenfassung

- Qualität muss (wie vieles anderes) für jedes Projekt spezifisch definiert werden
- Qualitätsmerkmale sind gut dokumentiert und können wiederverwendet werden
- Metriken können auf Qualitätsprobleme hindeuten, müssen aber immer subjektiv beurteilt werden
- Diverse Qualitätsmaßnahmen ergänzen sich gegenseitig, um Qualität abzusichern und können abhängig von den Anforderungen eingesetzt werden

