Fachhochschule
Dortmund
University of Applied Sciences

Softwaretechnik D

Qualitätssicherung und Wartung

Prof. Dr. Dirk Wiesmann

Wintersemester 2024/2025

Warum spielt Qualität von Software ein Rolle?

- Software dringt in fast alle Bereiche des täglichen Lebens vor (Bestandteil von Gebrauchsgütern) *ubiquitäres Computerzeitalter*
- Die Systemkomplexität steigt an
- Software wird in sicherheitskritischen Bereichen eingesetzt (Luftfahrt, Auto, Medizintechnik, Kraftwerke)

- Die spezielle Sicht von Informatikern, die in Deutschland tätig sind
  - Softwareentwicklung ist personalintensiv
  - Die Arbeitskosten sind in Deutschland vergleichsweise hoch
  - Dadurch entsteht ein Wettbewerbsnachteil
  - Man muss sich von den billigeren Mitbewerbern absetzen
  - Möglichkeit: Eine hohe Qualität bieten

Erfordert qualifizierte Mitarbeiter *Software Made in Germany*

Was sind Merkmale einer qualitativ hochwertigen Software?

Hohe Rechengenauigkeit

Gute Testbarkeit

Ausfallsicherheit

Korrektheit

leichte Bedienbarkeit

Einfachheit

Gute Dokumentation

...

Geringer Ressourcenverbrauch

gute Portierbarkeit

Spezifikationsvollständigkeit

Wie entsteht Qualität?

„Von nichts kommt nichts“
oder
„Qualität entsteht nicht von alleine“

- Wir müssen etwas tun!
  - Was ist zu tun?
  - Wie ist es zu tun?
  - Wann ist es zu tun?
  - Wer hat es zu tun?

Mit welchen Mitteln können wir qualitativ hochwertige Software erstellen?

Dokumentation
Integrationstest
Durchsicht
Zweigüberdeckung
Review
Blackbox-Test
Standards
Typisierung
Unit-Test
Konstruktive Maßnahmen
Richtlinien
Äquivalenzklassenbildung
Programmdokumentation
Grenzwertanalyse
Organisatorische Maßnahmen
Schulungen
Whitebox-Test
...
Werkzeuge
Planung
Prozessqualität
Metriken
Lasttest

Ziel

- Vermittlung der erforderlichen Kenntnisse, um bei der Softwareentwicklung ein definiertes Qualitätsniveau zu erzielen

> Zielgerichtet, unter Einsatz der geeigneten Methoden, Prinzipien und Werkzeuge, das benötigte Qualitätsniveau erreichen

- Methodisches Vorgehen bei der Software-Wartung

Weg

- Operationalisierung des Qualitätsbegriffs über Qualitätsmodelle
- Differenzierung in organisatorische, analytische und konstruktive Maßnahmen

Themen

1) Qualität und Qualitätsmodelle
2) Typische Fehlerquellen
3) Bedeutung der Prozessqualität
4) Konstruktive Maßnahmen
5) Manuelle Prüfmethoden
6) Dynamische Prüfmethoden (Test)
7) Testautomatisierung
8) Testen objektorientierter Programme
9) Statische Prüfmethoden / Metriken
10) Werkzeuge / Build-Automatisierung
11) Wartung

Disclaimer

- Die vorliegende Präsentation reicht **nicht** aus, um den Stoff der Vorlesung SWT D zu erlernen!
- Neben dem Durcharbeiten der Präsentation empfehle ich Ihnen
  1. den aktiven Besuch der Vorlesung
  2. die Lösung der Übungsaufgaben
  3. die Bearbeitung der Praktikumsaufgaben
  4. das Durcharbeiten von zusätzlicher Literatur

- Bitte bringen Sie zu den Vorlesungen, Übungen und Praktika Stift und Papier mit!
- Bitte bringen Sie zu den Übungen und Praktika Ihre Vorlesungsnotizen und die Folien mit!

Voraussetzungen

- Qualitätssicherung und Wartung sind Bestandteil der Softwaretechnik
- Die Vorlesungen Softwaretechnik 1 und 2 werden als Basis vorausgesetzt
  - Analyse
  - Entwurf

- Zudem werden Erfahrungen in der Programmierung mit Java vorausgesetzt
  - Einführung in die Informatik
  - Programmierkurs 1, A

Klausur

- Für die Prüfung relevante Inhalte
  - Vorlesung
    - Folien
    - Tafel
  - Übungen
  - Praktikum

- Aufgabentypen
  - Abfrage von Definitionen
  - Verständnisfragen
  - Anwendung (z.B. Herleitung von Testfällen, Aufdeckung von Qualitätsmängeln, ...)

Literatur

[Bal98] H. Balzert. *Lehrbuch der Softwaretechnik, Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, Heidelberg, 1998

[Bal08] H. Balzert. *Lehrbuch der Softwaretechnik, Softwaremanagement*. Spektrum Akademischer Verlag, Heidelberg, 2008

[Hof13] D.W. Hoffmann. *Software-Qualität*. Springer Vieweg, Berlin, 2013

[Lig09] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum, Heidelberg, 2009

[LL10] J. Ludewig, H. Lichter. *Software Engineering*. dpunkt.verlag, Heidelberg, 2010

[SL12] A. Spillner, T. Linz. *Basiswissen Softwaretest*. dpunkt.verlag, Heidelberg, 2012

[Wag13] S. Wagner. *Software Product Quality Control*. Springer, Berlin, 2013

Software

- Was ist Software?

> Programme, Abläufe, gegebenenfalls Dokumentation und Daten, die mit dem Betrieb eines Rechnersystems zu tun haben

- Damit ist Software mehr als nur ein ausführbares Programm
- In einigen Aspekten unterscheidet sich Software von üblichen technischen Produkten

- Diese Besonderheiten von Software haben Einfluss auf die Entwicklung von Software

- Spezielle Eigenschaften von Software:
  1. Software ist ein immaterielles Produkt
  2. Software benötigt einen Träger (Papier, „Silizium“)
  3. Software ist leicht änderbar
  4. Schwache Kausalität zwischen Änderung und Wirkung
  5. Ein Programm realisiert keine stetige Funktion
  6. Software ist schwer zu vermessen
  7. Software wird nicht gefertigt, sondern nur entwickelt
  8. Software verschleißt nicht
  9. Software altert über die Umgebung

Was ist

Qualität

Qualität

> *Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf die Eignung zur Erfüllung gegebener Erfordernisse beziehen.*

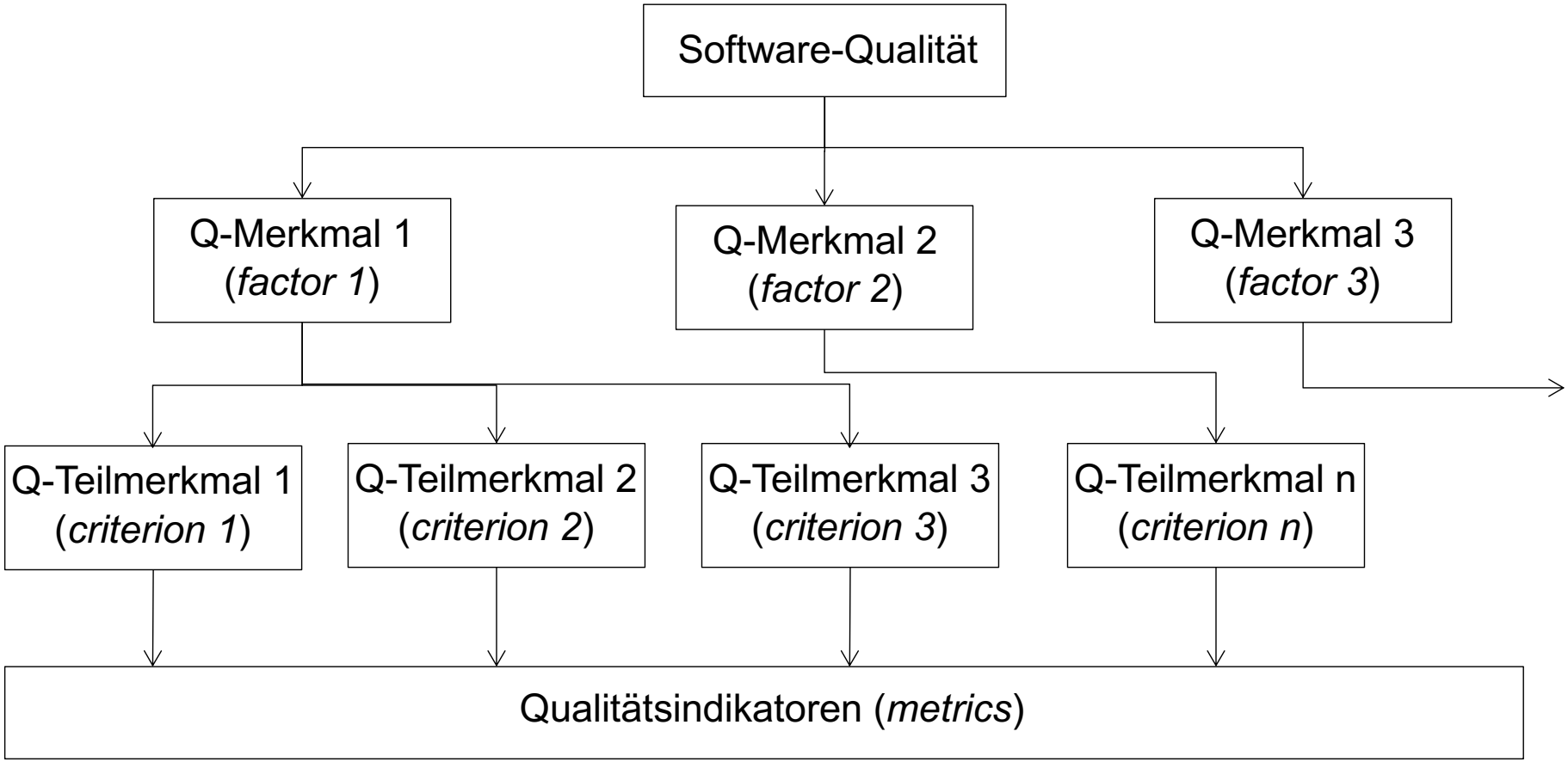
aus DIN 55350 – 11:1995-08

- Der Begriff Qualität ist
  - vielfältig
  - relativ
- In der Praxis
  - ist eine allgemeine Qualitätsdefinition nicht ausreichend
  - müssen die relevanten Qualitätsmerkmale identifiziert und benannt werden

Qualitätsmodell

- Man erstellt ein Qualitätsmodell, indem ausgehend vom allgemeinen Qualitätsbegriff Unterbegriffe (Qualitätsmerkmale, *factors*, *characteristics*) abgeleitet werden
- Diese Qualitätsmerkmale werden in einem weiteren Schritt in Teilmerkmale (atomare Qualitätsmerkmale, *criterions*, *subcharacteristics*) verfeinert
- Das Ziel ist die Messbarkeit/Überprüfbarkeit von atomaren Qualitätsmerkmalen durch Qualitätsindikatoren (Attributen)

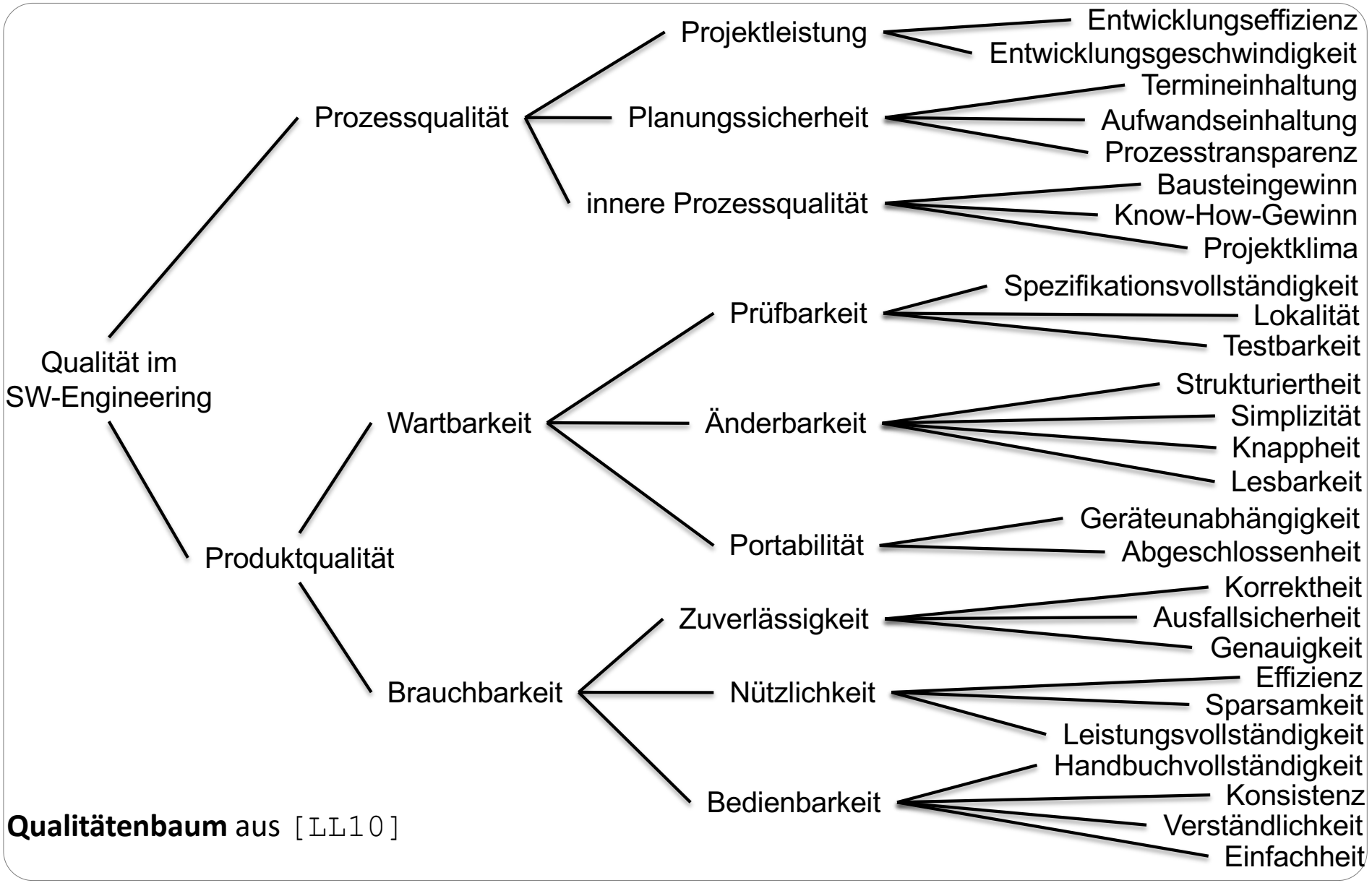
– Schematischer Aufbau eines FCM-Qualitätsmodells (*factor-criteria-metrics-models*)



- Im Software-Engineering gibt es zwei Sichtweisen auf die Qualität:
  - Produktqualität
  - Prozessqualität (siehe Bewertung von Prozessmodellen)

- Historisch sind verschiedene FCM-Qualitätsmodelle entstanden
  - Qualitätenbaum nach Boehm, Brown und Lipow (1976) aus [LL10]
  - ISO/IEC 9126
  - ISO/IEC 250*nn* (ab 2005 Nachfolge von 9126)

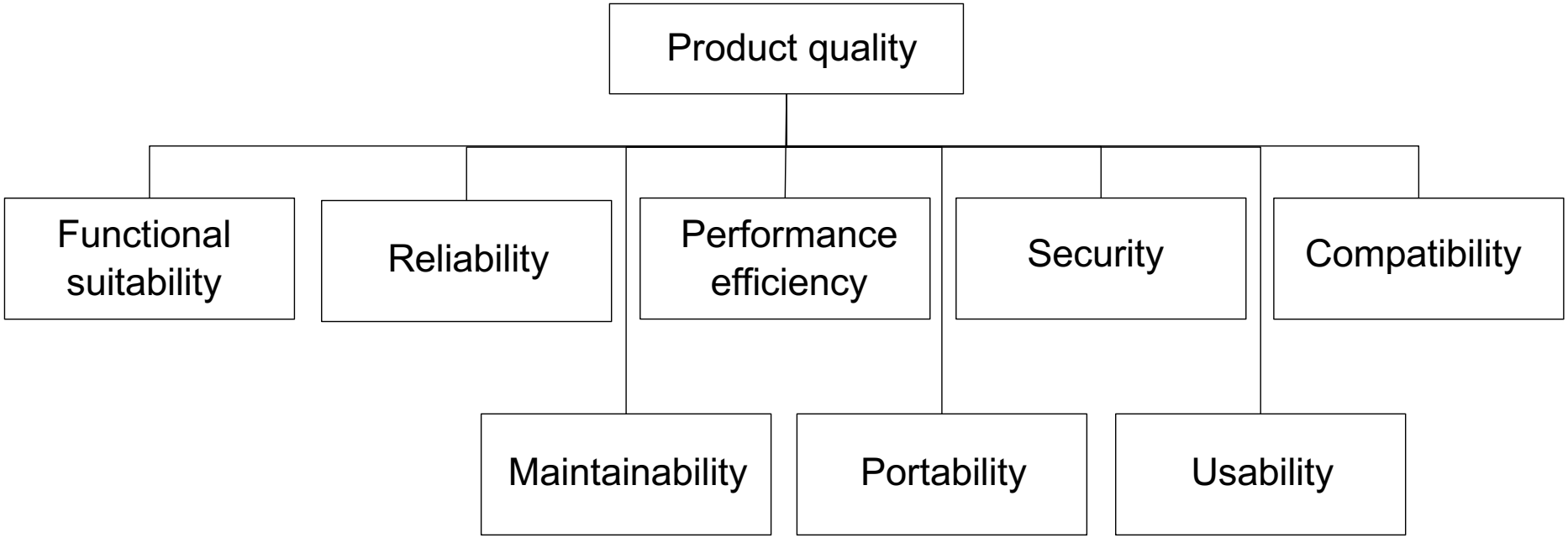
Qualitätsmodell



Qualitätenbaum aus [LL10]

- Die Norm ISO/ICE 25010 betrachtet nur die Produktqualität
- Hier werden zwei Qualitätsmodelle definiert
  - *Product quality*
  - *Quality in use*
- Definitionen aus ISO/IEC FCD 25010 aus 2010 *(final draft*)

– Das Qualitätsmodell *Product quality* gliedert sich in 8 Qualitätsmerkmale



- Functional suitability (funktionale Eignung)

  the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions

- Reliability (Ausfallsicherheit)

  the degree to which a system or component performs specified functions under specified conditions for a specified period of time

- Performance efficiency (Leistungseffizienz)

  the performance relative to the amount of resources used under stated conditions

- Security (Sicherheit)

  the degree of protection of information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them

- Compatibility (Kompatibilität)

  the degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment

- Maintainability (Wartbarkeit)

  the degree of effectiveness and effiviency with wich a product or system can be modified by the intended maintainers

- Portability (Portabilität)

  the degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another

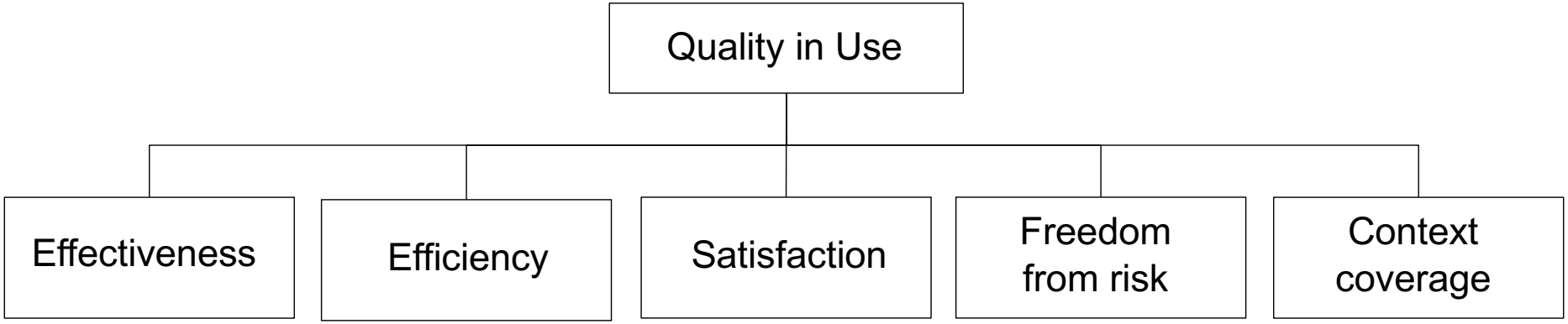
– Das Qualitätsmodell *Product quality* in der Übersicht

Qualitätsmerkmal	**Teilmerkmale**
Functional suitability	Functional appropriateness
	Functional correctness
	Functional completeness
Reliability	Maturity
	Availability
	Fault tolerance
	Recoverbility
Performance efficiency	Time behaviour
	Resource utilisation
	Capacity

Qualitätsmerkmal	Teilmerkmale
Security	Confidentiality
	Integrity
	Non-repudiation
	Accountability
	Authenticity
Compatibility	Co-existence
	Interoperability
Maintainability	Modularity
	Reusability
	Analysability
	Modifiability
	Testability

Qualitätsmerkmal	Teilmerkmale
Portability	Adaptability
	Installability
	Replaceability
Usability	Appropriatness recognizability
	Learnability
	Operability
	User error protection
	User interface aesthetics
	Accessibility

– *Quality in Use*:
  Zu welchem Grad kann ein spezifischer Nutzer spezifische Ziele unter Berücksichtigung von Effektivität, Effizienz, Zufriedenheit, Sicherheit und Brauchbarkeit erreichen (bezogen auf einen spezifischen Anwendungskontext)



- Effectiveness (Effektivität)

  the accuracy and completeness with which users achieve specified goals

- Efficiency (Effizienz)

  the resources expended in relation to the accuracy and completeness with which users achieve goals

- Satisfaction (Zufriedenstellung)

  the degree to which users are satisfied with the experience of using a product in a specified context of use

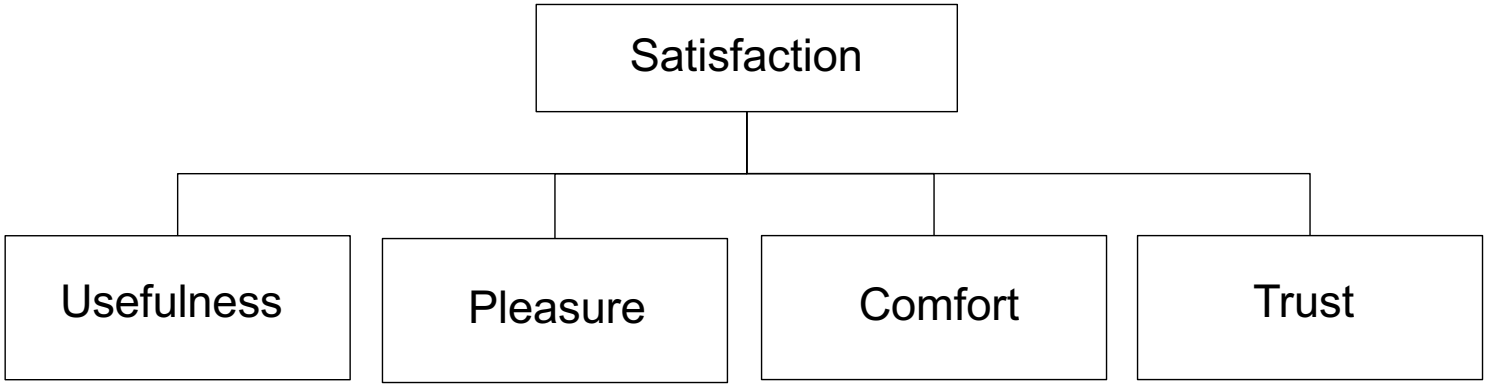
- Freedom from risk (Risikofreiheit)

  the degree to which a product or system mitigates the potential risk to economic status, human life, health, or the environment

- Context coverage (Nutzbarkeit, Brauchbarkeit)

  the degree to which a product or system can be used with effectiveness, efficiency, freedom from risk an satisfaction in both specified contexts of use and in contexts beyond those initialy explicitly identified

- Das Qualitätsmerkmal Zufriedenstellung (*characteristic satisfaction*) wird in vier Teilmerkmale unterteilt



- Usefulness (Nützlichkeit)

  the degree to which a user is satisfied with their perceived achievment of pragmatic goals, including the results of use and the consequences of use

- Pleasure (Freude, emotionale Zufriedenheit)

  the degree to which a user obtain pleasure from fulfilling their personal needs

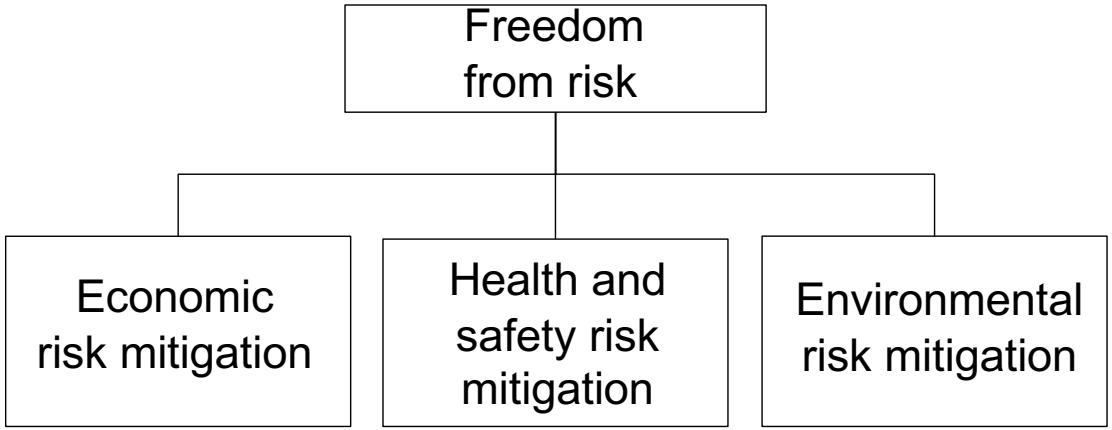
- Comfort (Komfort)

  the degree to which the user is satisfied with physical comfort

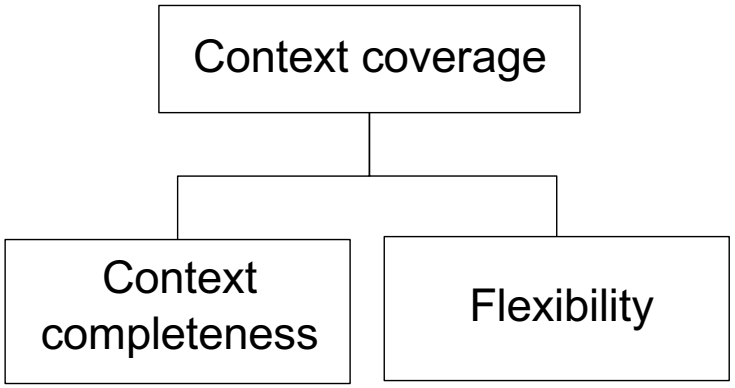
- Trust (Vertrauen)

  the degree to which a user or other stakeholder has confidence that a product or system will behave as intended

– Das Qualitätsmerkmal Risikofreiheit (*freedom from risk*) wird in drei Teilmerkmale unterteilt



– Das Qualitätsmerkmal Kontextüberdeckung (*context coverage*) wird in zwei Teilmerkmale unterteilt



- Qualitätsbewertung ist ein Soll-Ist-Vergleich
  - Soll-Wert: Vorgaben für ein Qualitätsmerkmal, abgeleitet aus den Erfordernissen
  - Ist-Wert: Tatsächliche Ausprägung des Qualitätsmerkmals

- Eine Abweichung zwischen Soll- und Ist-Wert bezeichnen wir als Fehler

Vereinfachte Definition

Eine Unterscheidung zwischen Fehlverhalten und Fehler ist z.B. möglich

- Zu jedem Teilqualitätsmerkmal wird ein geeigneter Qualitätsindikator (Metrik) benötigt *wird nicht zwingend durch ein Qualitätsmodell vorgegeben*
  - Vorgabe eines zu erreichenden Ziels (Soll-Wert)
  - Soll-Ist-Vergleich (Qualitätsmessung)

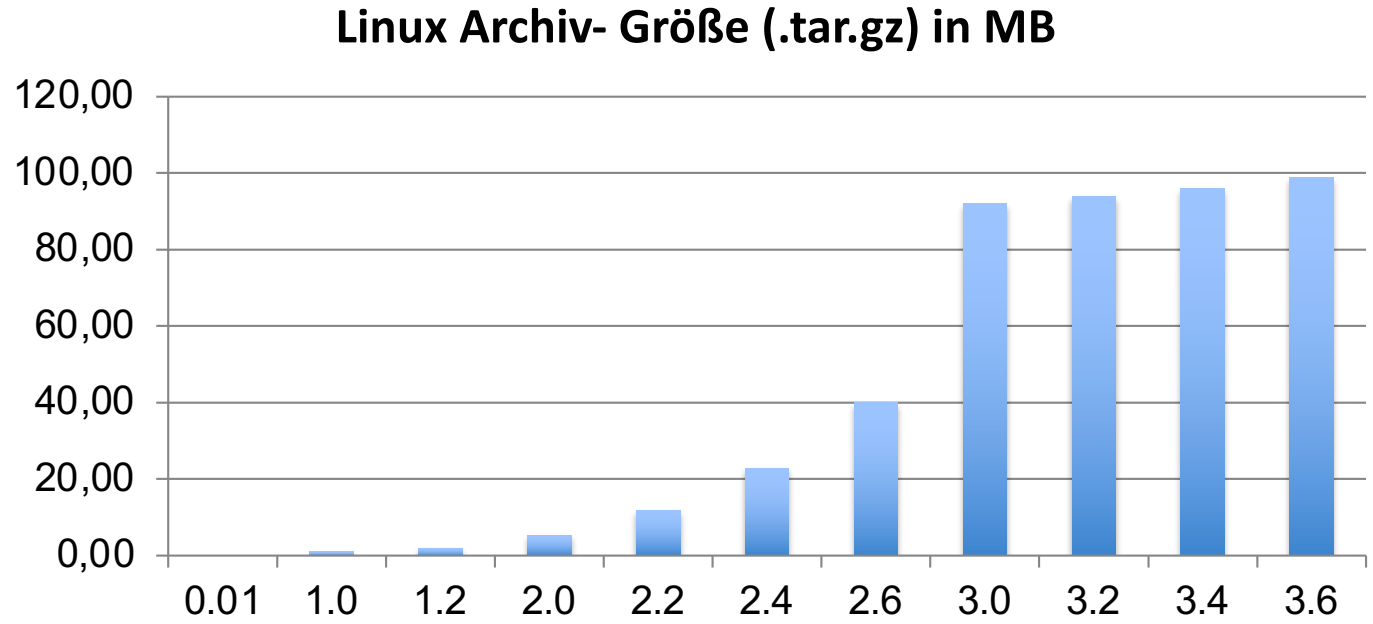
- Beispiele
  - Satisfaction (Zufriedenstellung)
    - Bewertung des Systems durch (Test-)Anwender bezüglich einer Ordinalskala (sehr zufrieden, zufrieden, unzufrieden, sehr unzufrieden)
    - Vorgabe eines Soll-Wertes durch eine Häufigkeitsverteilung
    - Soll-Ist-Vergleich z.B. über Median
  - Portability
    - 1 – (*Umgebungsabhängige Module* / *Module)*
  - Efficiency
    - 1 – (*Ist-Ressourcenverbrauch* / *Allokierte Ressourcen*)

Für ein Teilqualitätsmerkmal kann es unterschiedliche Metriken geben

Was sind typische

Fehlerquellen

- Gründe für die schlechte Qualität von Software
  a. Die speziellen Eigenschaften von Software führen schnell zu einer schwer zu beherrschenden Komplexität
  b. Software-Größe steigt stark an



  c. Starke Fluktuation von Mitarbeitern / Mangelnde Qualifikation
  d. Management versucht Erfahrungen aus Produktionsprozessen auf die Softwareentwicklung zu übertragen

Fehlerquellen nach [Hof13]

1. Lexikalische und syntaktische Fehlerquellen
   - Welche Ausgabe liefert die folgende Sequenz?

```
int a = 2;
int b = 1;
int c = a---b;
printf("%i\n", c);
```

   - Gibt es hier ein Problem?

```
y = x/*p /* p ist Pointer auf Divisor */;
```

   - Arbeitet diese Funktion korrekt?

```
float nrm(float x)
{
    while(abs(x)>0,1)
        x = x / 10;
    return x;
}
```

2. Numerische Fehlerquellen

- Welche Ausgabe wird geschrieben?

```
double x = 0.1*3.0;
double y = 3.0/10.0;

if (x == y) {
    System.out.println(x + " gleich " + y);
}
else {
    System.out.println(x + " ungleich " + y);
}
```

- Numerische Überläufe
  - Y2K-Bug
  - Zeitdarstellung im POSIX-Format (Problem am 19.01.2038 um 3:14:08 UTC)
  - ...

3. Semantische Fehlerquellen

- Wo liegt das Problem?

```
String input =
      JOptionPane.showInputDialog("Raumtemperatur (in
Celsius)");
float grad = Float.valueOf(input);
if (Util.isTooHot(grad)){
    JOptionPane.showMessageDialog(null, "Sie haben Hitzefrei!");
} else{
    JOptionPane.showMessageDialog(null, "Sie müssen arbeiten!");
}
```

```
public class Util {
    static boolean isTooHot(float temperature){
        if (temperature > 82.4)
            return true;
        return false;
    }
}
```

4. Parallelität als Fehlerquelle

- Wo liegt das Problem?

Eine Klasse Produzent produziert Waren, die in in einem Lager mit der Kapazität 20 abgelegt werden. Eine Klasse Konsument entnimmt aus dem Lager Waren. Produzent und Konsument werden als Threads gestartet

```
Lager lager = new Lager();

Thread tp = new Produzent(lager);
Thread tk = new Konsument(lager);

tp.start();
tk.start();
```

```
class Lager {
   private int bestand;
   private static final int kapazität = 20;

   public void einlagern() {
      if (bestand+5 <= kapazität) {
         bestand+=5;
         System.out.println("nach Einlagern :" + bestand);
      }
   }

   public void entnehmen() {
      if (bestand-3 >= 0) {
         bestand-=3;
         System.out.println("nach Entnehmen :" + bestand);
      }
   }
}
```

```
class ProduzentThread extends Thread {
   private Lager lager;

   ProduzentThread(Lager lager) {
      this.lager = lager;
   }

   public void run() {
      while(true){
         lager.einlagern();
      }
   }
}
```

Klasse KonsumentThread ist analog implementiert, nur wird hier die Methode lager.entnehmen() aufgerufen

Ein Programmstart führt zur folgenden Ausgabe:

```
...........
nach Entnehmen :  1
nach Einlagern :  6
nach Einlagern :  8
nach Einlagern : 13
nach Einlagern : 18
nach Entnehmen :  3
nach Entnehmen : 15
nach Entnehmen : 12
```

Wie kann das Problem behoben werden?

Race conditions

Deadlocks

5. Spezifikationsfehler
   - Bisher haben wir nur Implementierungsfehler betrachtet
   - Ein Fehler kann aber bereits schon in der Spezifikation enthalten sein
     - Unvollständige Vorgaben
     - Widersprüchliche Vorgaben
     - Ungeeignete Berechnungen
   - Solche Fehler können mit Verfahren, die gegen die Spezifikation testen, nicht entdeckt werden

6. Portabilitätsfehler
   - Es kann sein, dass eine Software auf einem System (Hardware + Betriebssystem + Laufzeitumgebung) fehlerfrei läuft, auf einem anderen System Fehler zum Vorschein kommen
   - Ursachen
     - CPU-Geschwindigkeit
     - Prozess-Scheduler
     - Speicherplatz / Speicherverwaltung
     - Zahlendarstellung

7. Hardwarefehler
   - Nicht immer muss die Software Schuld an einem Fehler sein
   - Ein Hardwarefehler kann zu fehlerhaften Berechnungen führen
     - Falsche Berechnung der Divisionseinheit von Pentium I – Prozessoren (Produktionsjahr 1993)

Qualitätssicherung
Qualitätsmanagement

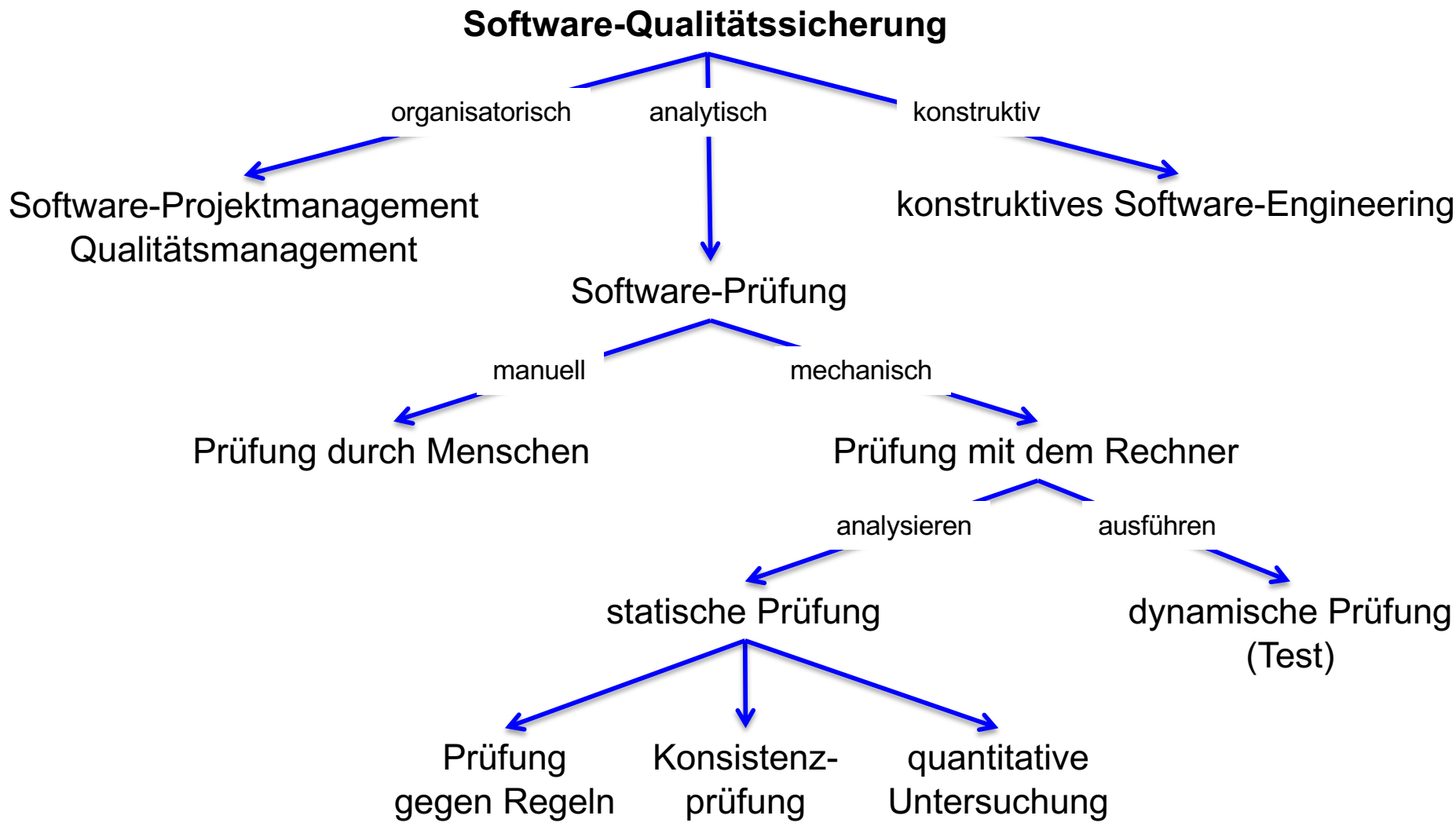
– Es werden drei Maßnahmen zur Software-Qualitätssicherung unterschieden:

1. Organisatorische Maßnahmen
   - Systematische Entwicklung und Qualitätssicherung
   - inkl. Zeitplanung für Prüfung und Korrekturen
   - inkl. Festlegung der Verantwortlichkeiten für Prüfungen
   - Vorgabe von Richtlinien, Standards und Checklisten

2. Konstruktive Maßnahmen
   - „Es kann keine Qualität in ein Produkt hineingeprüft werden“
   - Fehler und schlechte Qualität von Beginn an vermeiden

3. Analytische Maßnahmen
   - Software-Prüfung
   - Fehler und Mängel in den Arbeitsergebnissen erkennen

Organisatorische und konstruktive Maßnahmen sind hilfreicher als analytische Maßnahmen



Gliederung der Software-Qualitätssicherung nach [LL10]

- Die geforderten Qualitätsanforderungen müssen definiert, und die Einhaltung der Qualitätsanforderungen muss überprüft werden
- Es müssen geeignete Rahmenbedingungen geschaffen werden

- **Qualitätsmanagement:**

  Aufeinander abgestimmte Tätigkeiten zum Leiten und Lenken einer Organisation bezüglich Qualität, die üblicherweise das Festlegen der Qualitätspolitik und der Qualitätsziele, die Qualitätsplanung, die Qualitätslenkung, die Qualitätssicherung und die Qualitätsverbesserung umfassen.

Aktivitäten im Qualitätsmanagement

(nach Skript „System- und Softwaresicherung“ von Prof. Dr. Ecke-Schüth)

a) Qualitätsplanung
   - Festlegung von Qualitätsanforderungen an den Prozess und das Produkt in überprüfbarer Form

b) Qualitätslenkung und –sicherung
   - Umsetzung, Steuerung, Überwachung und Korrektur des Entwicklungsprozesses mit dem Ziel, die vorgegebenen Anforderungen zu erfüllen

c) Qualitätsprüfung
   - Durchführung der im Rahmen der Qualitätsplanung festgelegten Maßnahmen
     - Erfassung der Ist-Werte der Qualitätsindikatoren
     - Überwachung der Umsetzung der konstruktiven Maßnahmen
     - Tests, Durchsicht, Reviews, ...

d) Qualitätsverbesserung
   - Auswertung der Qualitätssicherungsergebnisse und der Prozessverbesserungen

– Dokumentation
   - Qualitätssicherungsplan (Prozess-orientiert)
     - Ergebnisse der Qualitätsplanung
   - Prüfplan (Produkt-orientiert)
     - Begleitende Maßnahmen

Qualitätsplanung ➔ Qualitätssicherungsplan

a) Festlegung der Aufgaben: Was ist zu tun?
   - Identifizierung der zu sichernden Produkte
   - Identifizierung der relevanten Qualitätsmerkmale
     - Relative Bedeutung
     - Quantifizierung in Form von Metriken

a) Festlegung der Vorgaben und Hilfsmittel: Wie ist es zu tun?
   - Auswahl der zur Datenerfassung und Qualitätsprüfung geeigneten Techniken und Methoden
   - Konstruktive Vorgaben (Richtlinien, Vorlagen, Werkzeuge, ...)
   - Analytische Vorgaben (Verfahren, Werkzeuge, ...)

c) Festlegung der Termine: (Bis) wann ist es zu tun?
   - Festlegung des zeitlichen Verlaufs der Datenerfassung/Dokumentation über den Entwicklungsprozess
   - Abstimmung des Prüfplans mit dem Projektplan

d) Festlegung der Verantwortlichkeiten: Wer hat es zu tun?
   - Festlegung der Verantwortlichkeiten für die Qualitätsprüfung und –lenkung
   - Definition und Besetzung von Rollen (Qualitätsmanager, Prüfer, Autor, Gutachter)

Qualitätslenkung

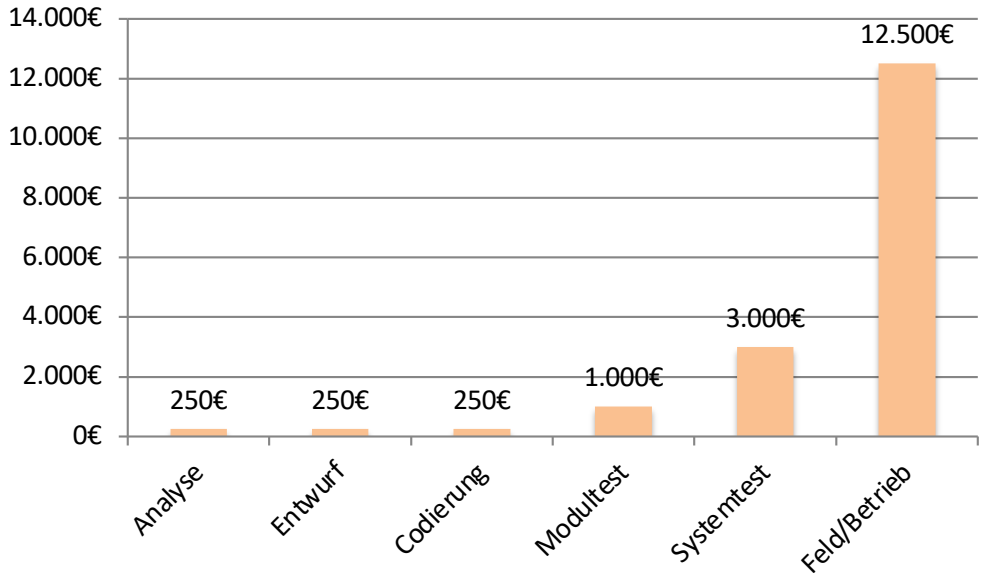
a) Reguläre Aktivitäten
   - Durch entwicklungsbegleitende Qualitätsprüfungen sind die Anforderungen sicherzustellen

b) Besondere Aktivitäten

c) Finale Aktivitäten
   - Wenn alle Qualitätsziele für ein (Teil-)Produkt erfüllt sind, kann eine formale Abnahme erfolgen (Produktzertifikat)

Wann sollte mit der Qualitätssicherung begonnen werden?

- Nach Untersuchungen von Boehm führt eine späte Fehlerentdeckung zu einem exponentiellen Kostenanstieg
- Das Zahlenmaterial wurde von Boehm 1981 veröffentlicht. Ein überproportionaler Kostenanstieg wird aber auch in der heutigen Zeit als realistisch angesehen (siehe zu diesem Thema [Bal08])
- In [Bal08] sind Daten zu den Kosten pro Fehlerkorrektur angegeben, die von Möller im Jahr 1996 publiziert wurden:



Für uns sind die relativen Kostenangaben von Interesse

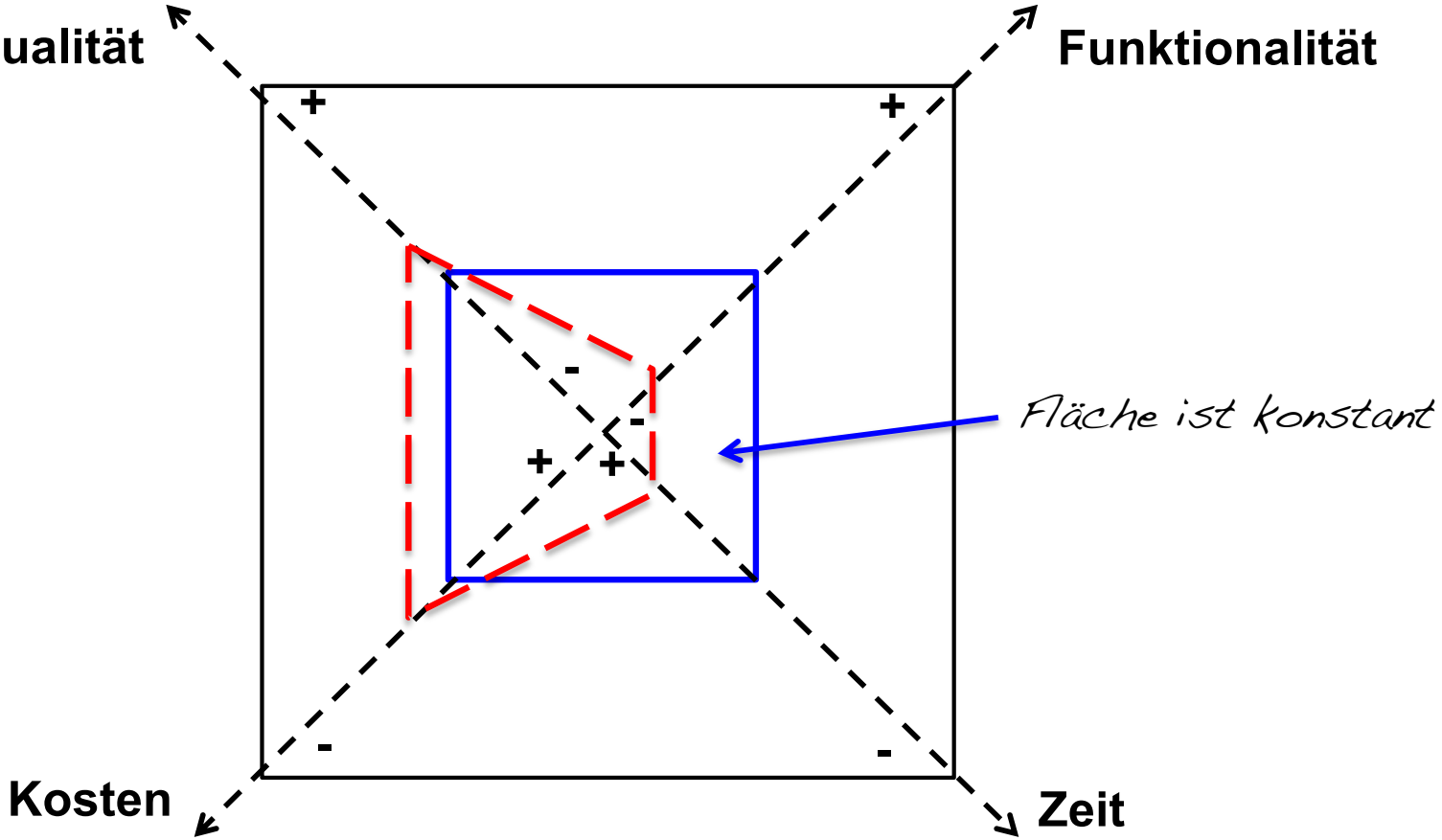
- Fehler, die früh in der Entwicklung entstehen und für einen längeren Zeitraum nicht entdeckt wurden, können zu Summationseffekten in späteren Phasen führen

  Je früher ein Fehler entdeckt wird, desto kostengünstiger kann er behoben werden

- Die beiden folgenden Ziele sollten in der angegebenen Reihenfolge verfolgt werden:
  1. keine Fehler machen (z.B. durch geeignete konstruktive Maßnahmen)
  2. Fehler, die dennoch gemacht wurden, möglichst früh entdecken und beseitigen

Qualität und Projekt

- Qualität gibt es nicht geschenkt
  - Qualität, Funktionalität, Kosten und Zeit sind über die Produktivität gekoppelt
  - Im Laufe eines Projekts ist die Produktivität konstant



Prozessqualität

Bezug zu SWT C

Aktivitäten der Softwareentwicklung

- Die vielfältigen Arbeiten, die bei der Entwicklung von Software anfallen, lassen sich unabhängig von der fachlichen Aufgabenstellung in Tätigkeitsbereiche einordnen:

- Die Tätigkeiten lassen sich noch weiter unterteilen

- Analyse [Analyse, Spezifikation der Anforderungen]
  - Verstehen der (fachlichen) Aufgabenstellung
  - Welche Aufgaben sollen von der Software übernommen werden?
  - Prüfen, Korrigieren und Dokumentieren der Anforderungen

- Entwurf
  - Berücksichtigung technischer Randbedingungen
  - Aufteilen der Gesamtfunktionalität auf Module/Komponenten
  - Festlegen von Schnittstellen
  - Strukturierung der Module/Komponenten
- Codierung/Implementierung [Codierung, Modultest]
  - Die Module werden gemäß der Spezifikation mit der gewählten Programmiersprache implementiert
  - Die Module werden auf Basis der Spezifikation getestet
- Integration [Integration, Test, Abnahme]
  - Das Gesamtsystem wird aus den einzelnen Komponenten/Modulen zusammengebaut
  - Test des Gesamtsystems
  - Abnahme durch den Kunden

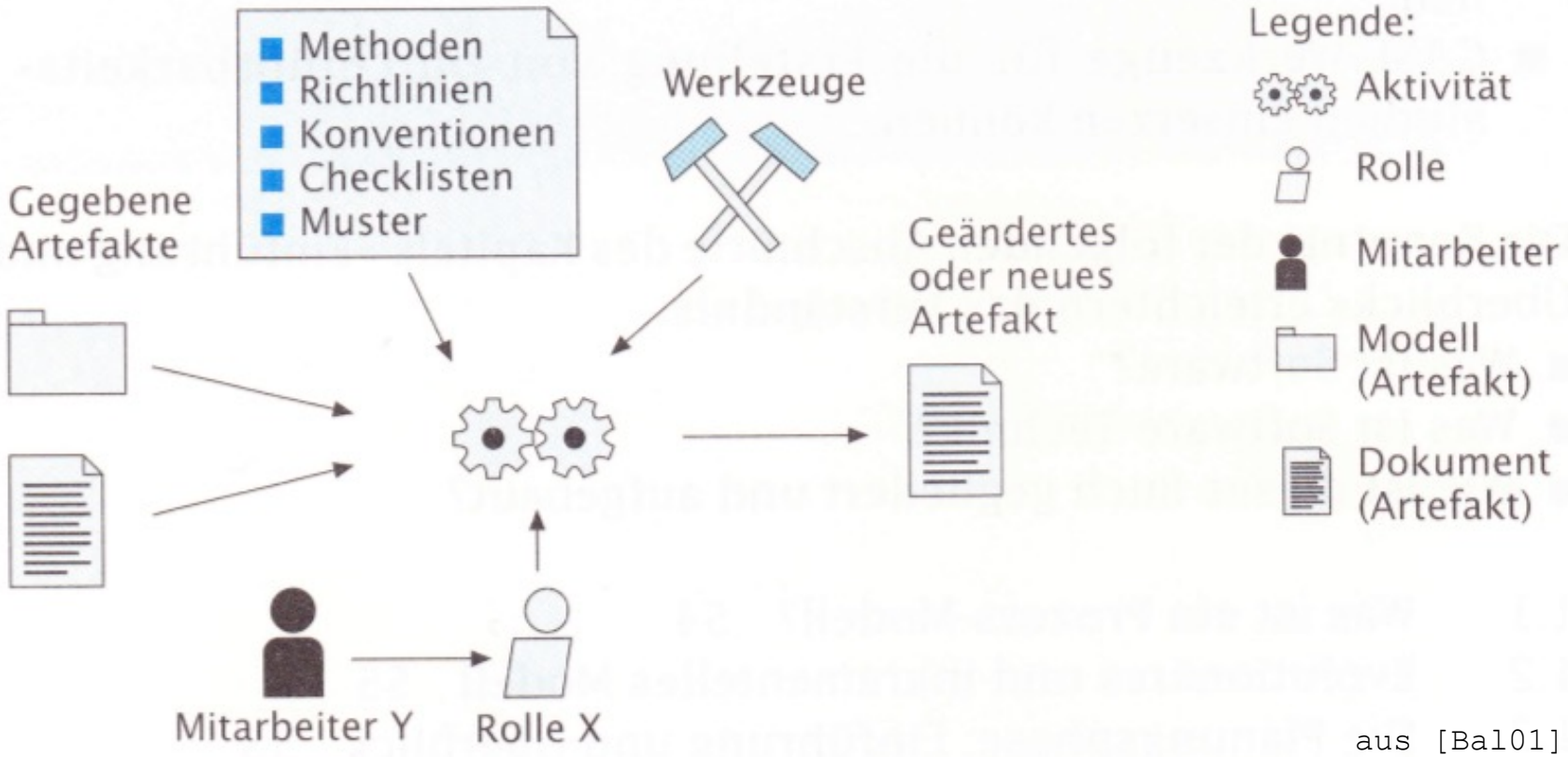
- Betrieb [Installation, Betrieb, Monitoring, Wartung, Pflege, Ersetzung]
  - Die Software wird in der Zielumgebung installiert und in Betrieb genommen
  - *Wartung*: Fehler, die sich während des Betriebs zeigen, werden beseitigt
  - *Pflege*: Es werden Änderungen/Erweiterungen der Funktionalität vorgenommen (z.B. auf Grund von gesetzlichen Anforderungen)
  - Bei Bedarf wird das System durch neue Software ersetzt. Der Übergang erfordert wieder eine präzise Planung und Umsetzung

- Eine Reihe von Tätigkeiten finden während der gesamten Entwicklungszeit statt:
  - Planung/Management
  - Dokumentation
  - Qualitätssicherung

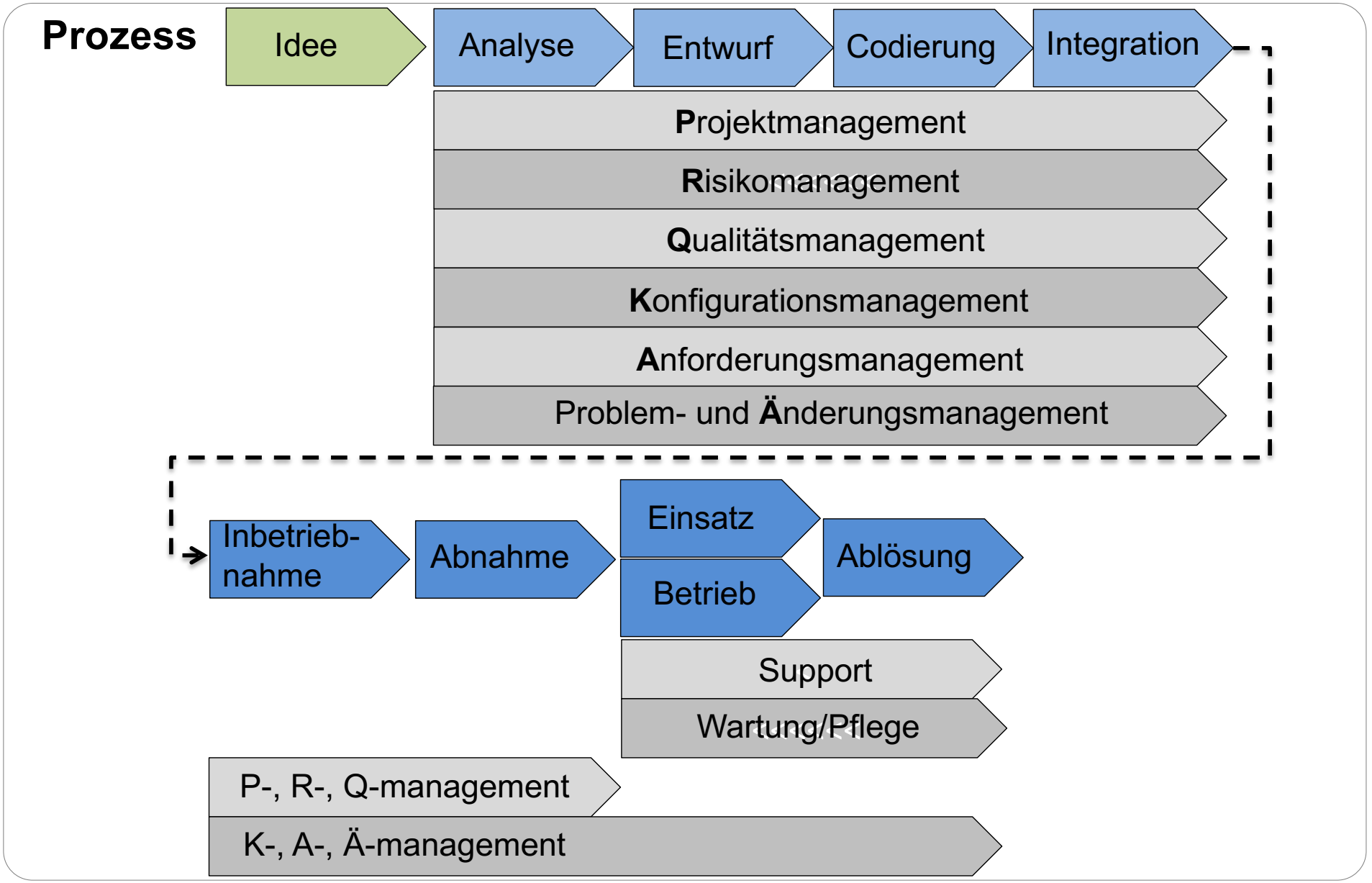
- Es ist festzulegen, in welcher Reihenfolge und wie häufig die einzelnen Tätigkeiten durchlaufen werden
- Hierzu existieren verschiedene **Vorgehensmodelle**

- Jede **Tätigkeit/Aktivität** wird von einem **Mitarbeiter** ausgeführt. Der Mitarbeiter nimmt dabei eine **Rolle** ein
- Jede Tätigkeit liefert **Ergebnisse**. Dabei kann es sich z.B. um Texte, Diagramme oder Quellcodes handeln. Die Ergebnisse werden allgemein auch als **Artefakte** bezeichnet

- **Prozessmodelle** legen neben der Vorgabe einer Reihenfolge der Tätigkeiten zusätzlich die folgenden Punkte fest:
  - personelle Organisation (Rollen)
  - Aufbau der Artefakte (z.B. Gliederung der Dokumentation)
  - Verantwortlichkeiten für Aktivitäten und Artefakte


Methoden
Richtlinien
Konventionen
Checklisten
Muster
Werkzeuge
Gegebene Artefakte
Geändertes oder neues Artefakt
Mitarbeiter Y
Rolle X
Legende:
Aktivität
Rolle
Mitarbeiter
Modell (Artefakt)
Dokument (Artefakt)

aus [Bal01]


Prozess
Idee
Analyse
Entwurf
Codierung
Integration
Projektmanagement
Risikomanagement
Qualitätsmanagement
Konfigurationsmanagement
Anforderungsmanagement
Problem- und Änderungsmanagement
Inbetrieb-nahme
Abnahme
Einsatz
Betrieb
Ablösung
Support
Wartung/Pflege
P-, R-, Q-management
K-, A-, Ä-management

Prozessqualität

- Ein guter Prozess führt nicht zwangsläufig zu guter Software, ein guter Prozess begünstigt aber die Erstellung guter Software
- Es gibt verschiedene Ansätze, um den Reifegrad eines Prozesses zu bestimmen
  - Ein hoher Reifegrad begünstigt eine hohe Prozessqualität

- Die folgenden Themen werden in der Vorlesung Softwaretechnik C behandelt
  - Softwareprojektmanagement
  - Vorgehens- und Prozessmodelle
  - Risikomanagement
  - Konfigurationsmanagement
  - Anforderungs- und Änderungsmanagement

Die Themen werden in SWT D pragmatisch aus Sicht der Software-qualitätssicherung aufgenommen

Konstruktive Maßnahmen

Konstruktive Maßnahmen

- Durch geeignete Maßnahmen soll die Entstehung von Fehlern von Anfang an vermieden werden
- Produkt- und Prozessqualität können durch konstruktive Maßnahmen beeinflusst werden
- Wir werden verschiedene konstruktive Maßnahmen betrachten

1. Qualifikation und Schulungen
   - Als Beispiel betrachten wird die Programmierung
   - Das eigentliche Programmieren (Erstellen des Quellcodes) wird häufig als reines Handwerk gesehen
   - Diese Sichtweise beruht auf dem Wunsch, dass das Ergebnis der Codierung ein normgerechtes Ingenieurprodukt sein soll, und kein künstlerisches Werk (siehe [LL10])
   - Dieser Manager-Traum ist aber nicht in Erfüllung gegangen

- Gute Programme werden (immer noch) nicht mit wenigen automatisierten Handgriffen am Fließband erstellt
- Die Codierung / Implementierung umfasst eine Reihe von Aktivitäten:
  - Konzeption von Datenstrukturen und Algorithmen
  - Strukturierung des Programms
  - Erstellung des Quellcodes in der gewählten Programmiersprache
  - Dokumentation und Kommentierung der Programme
  - Testplanung
  - Test des Programms

Gute Programme werden von hochqualifizierten und motivierten Mitarbeitern erstellt

2. Attraktive Arbeitsumgebung
   - Qualifizierte Mitarbeiter fordern in der Regel interessante Aufgaben und benötigen eine gewisse gestalterische Freiheit
     - „Aufstiegsmöglichkeiten“ sind im technischen Bereich noch häufig ein ungelöstes Problem
   - Die Arbeitszeiten müssen angemessen (keine systematischen Überstunden) und flexibel sein
   - Die Ausstattung des Arbeitsplatzes muss angemessen sein

3. Verwendung eines geeigneten Prozessmodells
   - Falls das Prozessmodell nicht passend gewählt wird, können sich negative Auswirkungen ergeben
     - Reduzierung der Produktivität
     - Schlechte Einbindung der erforderlichen Maßnahmen zur Qualitätssicherung
   - Die folgenden Punkte sind bei der Auswahl eines Prozessmodells zu berücksichtigen
     - Vollständigkeit der Anforderungen
     - Stabilität der Anforderungen
     - Technologisches Know-How
     - Risikobewertung
     - Teamgröße
     - Persönlichkeitsprofile der Teammitglieder / Gruppendynamik
     - Erfahrung des Teams mit dem Prozessmodell

4. Verwendung von Werkzeugen

- Die professionelle Softwareentwicklung erfordert den Einsatz von geeigneten Werkzeugen
- Dabei sind die Werkzeuge auch selbst Software
- Die Verwendung von **Softwareentwicklungswerkzeugen** wird auch unter dem Begriff **CASE** zusammengefasst:

> ***computer-aided software engineering (CASE) –*** *The use of computers to aid in the software engineering process. May include the application of software tools to software design, requirements tracing, code production, testing, document generation, and other software engineering activities*

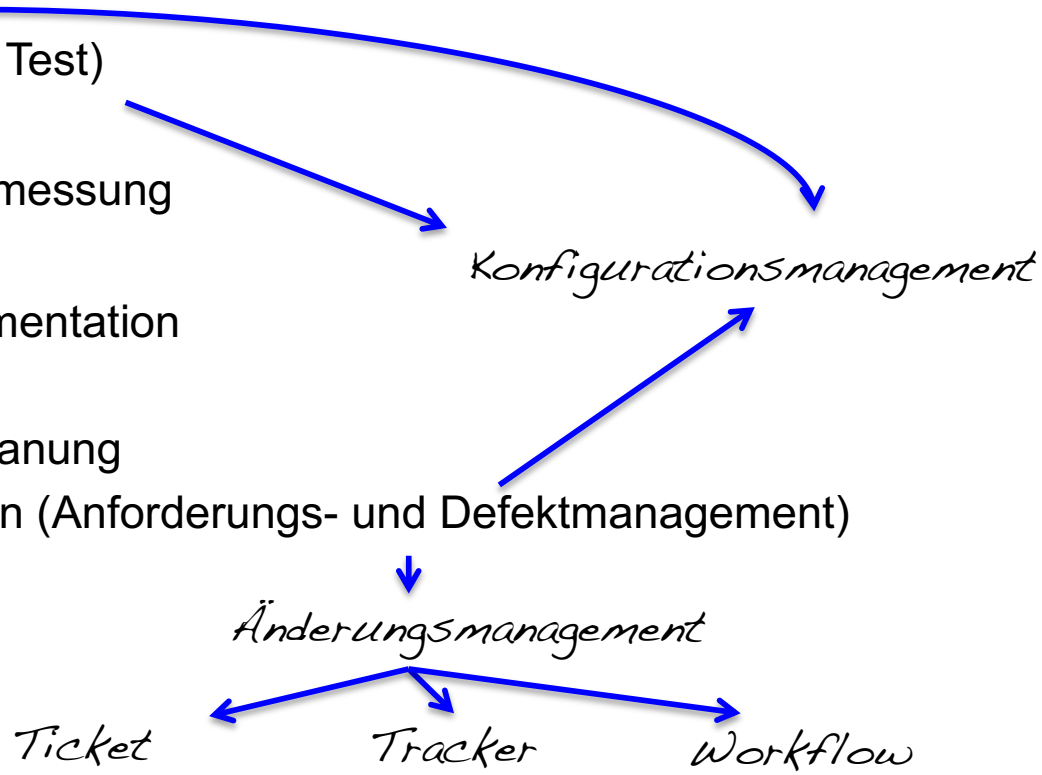
IEEE Std 610.12 (1990)

- Mit der Verwendung von Werkzeugen kann die Effizienz und die Effektivität der Softwareentwicklung gesteigert werden
- Damit ein Werkzeug mit Gewinn eingesetzt werden kann, müssen die folgenden Bedingungen erfüllt sein:
  - Das Werkzeug muss zur Aufgabe passen
  - Das Werkzeug muss sinnvoll eingesetzt werden
- Werkzeuge sind Teil einer Methode
- Werkzeuge unterstützen unterschiedliche Operationen

- Die Operationen lassen sich wie folgt klassifizieren:
  1. Editieren
  2. Transformieren
  3. Verwalten / Versionieren
  4. Suchen
  5. Nachvollziehen
  6. Messen
  7. Testen
  8. Verbinden
  9. Dokumentieren
  10. Verfolgen / Überwachen

- Werkzeug-Grundausstattung in einem professionellen Projekt
  - Methodenspezifischer Editor (Syntaxhighlighting, Autovervollständigung)
  - Compiler/Linker (inkl. Standardbibliotheken)
  - Versionsverwaltung
  - Automatisierung (Build, Test)
  - Überdeckungsprüfung
  - Statische Prüfung / Vermessung
  - Code Review
  - Kommunikation / Dokumentation
  - Aufwandserfassung
  - Projekt- / Ressourcenplanung
  - Verfolgen- / Überwachen (Anforderungs- und Defektmanagement)
  - ...

Integration über IDE



5. Verwendung einer geeigneten Programmiersprache
   - Die verwendete Programmiersprache hat Einfluss auf die Effizienz der Programmerstellung und die Qualität des Programms
   - Die Programmiersprache sollte zur Problemdomäne passen
     - Bessere Abstraktion
     - Klarere Struktur
     - Weniger Quellcode
     - Höhere Produktivität
   - Die Sprache sollte Eigenschaften besitzen, die sich positiv auf die Qualität der Programmierer und die Qualität der Programme auswirken (siehe[LL10]):
     - Strukturelemente zur Konstruktion modularer Programmeinheiten
     - Typsystem mit strenger Typprüfung
     - Trennung von Schnittstelle und Implementierung
     - Syntax, die zur Lesbarkeit des Codes beiträgt
     - Automatische Zeigerverwaltung
     - Ausnahmebehandlung
     - Gute Unterstützung durch Werkzeuge

6. Einhalten von Richtlinien

- Software wird in der Regel in Teams erstellt
- Dabei treten u.a. die folgenden Probleme auf
  - Zu viel Individualität erschwert die gemeinsame Arbeit an Artefakten (unterschiedliche Strukturen und Notationen)
  - Unter Zeitdruck wird vergessen wichtige Informationen zu dokumentieren
  - Hilfreiche Darstellungsformen sind nicht allen Teammitgliedern bekannt
- Durch die Vorgabe von Richtlinien will man die folgenden Punkte erreichen
  - Vereinheitlichung, um die Effizienz der Teamarbeit zu steigern
  - Fehlerreduktion, in dem problematische Darstellungs- und Notationsformen vermieden werden

Konkrete Richtlinien werden noch im weiteren Verlauf der Vorlesung betrachtet

- Richtlinien können für unterschiedliche Artefakte existieren
  - Gliederungsschema für ein Pflichtenheft
  - Verzeichnisstrukturen für Projektdateien
  - Aufbau eines Glossareintrags
  - ...
- Richtlinien können differenziert werden

a) Notationskonventionen
   - Schreibweise von Bezeichnern
   - Layout des Quelltextes
   - Aufbau von Kontrollstrukturen
   - ...

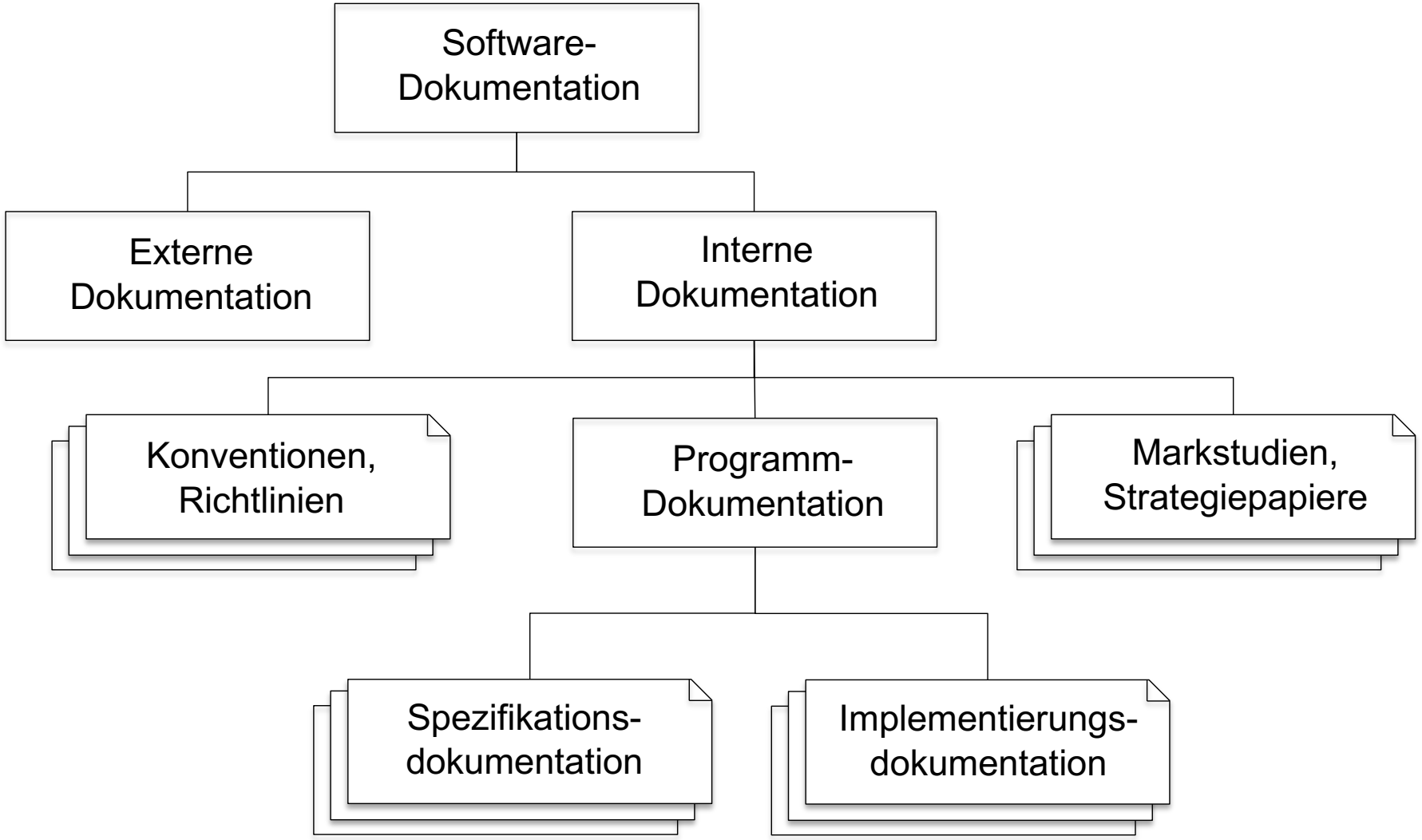
b) Sprachkonventionen
   - Regeln für die Formulierung von Anforderungen im Rahmen der Spezifikation
   - Umgang mit den semantischen Besonderheiten einer Programmiersprache
   - ...

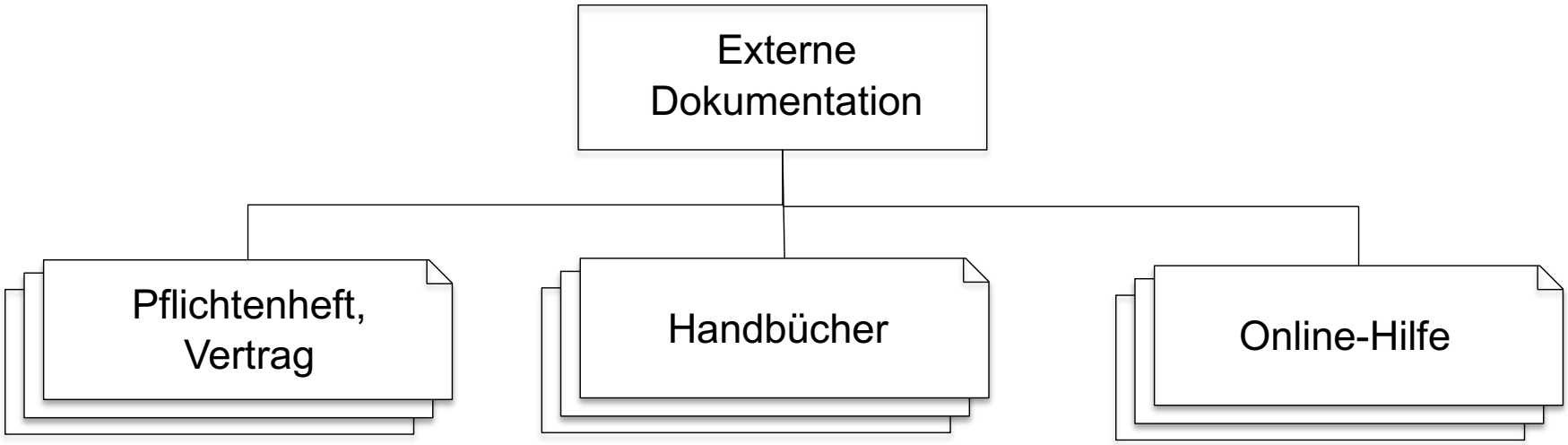
7. Erstellung einer angemessenen Dokumentation

- Wir erinnern uns: Software ist mehr als nur das ausführbare Programm
- Im Rahmen der Softwareentwicklung werden neben dem Quellcode noch zahlreiche weitere Artefakte erstellt
- Die Güte der erstellten Dokumente haben Einfluss auf unterschiedliche Qualitätsmerkmale
- Problem:
  - Manager unterschätzen der Wert einer angemessenen Dokumentation
  - Softwareentwickler (insbesondere Programmierer) fühlen sich durch die Dokumentation in ihrer „eigentlichen“ Arbeit gestört
- Lösung:
  - Festlegung auf eine sinnvolle Dokumentation
  - Unterstützung durch Richtlinien und Vorgaben
  - Sensibilisierung (Schulung) der Mitarbeiter (inkl. Management)

auch XP-Projekte kommen nicht ohne ein Handbuch und einer Online-Hilfe aus

- Dichotomie der Software-Dokumentation (nach [Hof13]):





- Externe Dokumentation
  - Mit der Erstellung kann/soll frühzeitig begonnen werden (Aufdeckung von widersprüchlichen und unvollständigen Anforderungen)
  - Hohe Anforderung an formale Kriterien (Rechtschreibung, Grammatik, Formatierung)
  - Hohe Anforderung an die Verständlichkeit

- Spezifikationsdokumentation

*Die **Spezifikation** dokumentiert die wesentlichen Anforderungen an eine Software und ihre Schnittstellen, und zwar präzise, vollständig und überprüfbar*

nach [LL10]

Darstellungsformen
- Informelle Spezifikation
- Semi-formale Spezifikation
- Formale Spezifikation

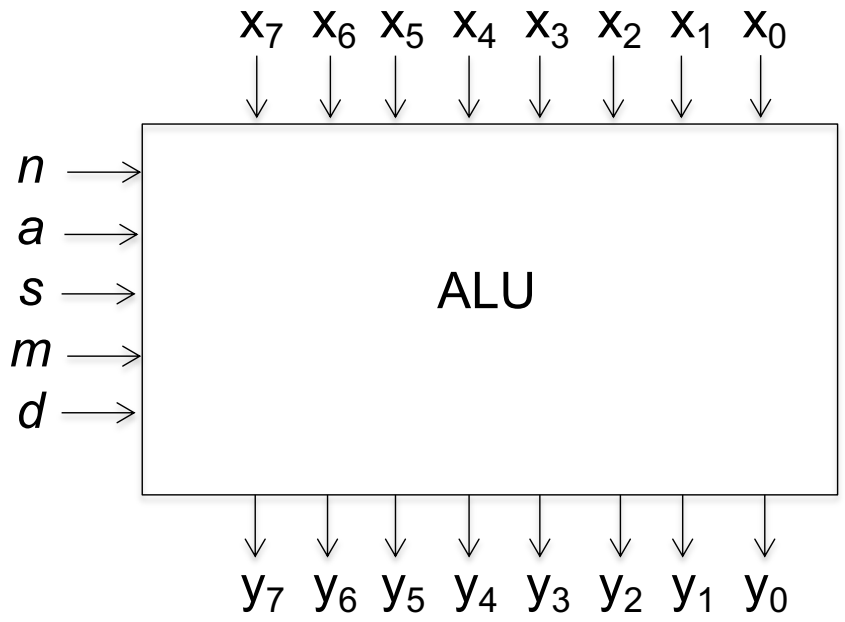
Eigenschaften
- zutreffend
- vollständig
- konsistent
- neutral
- nachvollziehbar
- überprüfbar (quantifizierbar)

Basis für:
- Entwurf und Implementierung
- Benutzerhandbuch
- Testvorbereitung
- Abnahme

Spezifikation beeinflusst die Qualität der nachfolgenden Aktivitäten

- Beispiel: Spezifikation einer ALU (serielle arithmetisch-logische Einheit)

nach [Hof13]



Ist die informelle Spezifikation konsistent und vollständig?

Informelle Spezifikation: Die ALU berechnet aus dem seriellen Eingabestrom (x_7,...,x_0) den Ausgabestrom (y_7,...,y_0). Die von der ALU ausgeführte Operation wird durch die Steuerleitungen *n*, *a*, *s*, *m*, *d* bestimmt. Für *n*=1 negiert die ALU den Eingabewert. Für *a*=1 berechnet sie die Summe, für *s*=1 die Differenz, für *m*=1 das Produkt und für *d*=1 den Quotienten der letzten beiden Eingabewerte

- Beispiel: Spezifikation einer ALU

Semi-formale Spezifikation

Eingabe:

`x[t]` : Eingabe zum Zeitpunkt *t* in 8-Bit-Zweierkomplementdarstellung

`n, a, s, m, d` : Steuerleitungen

Ausgabe:

`y[t]` : Ausgabe zum Zeitpunkt *t* in 8-Bit-Zweierkomplementdarstellung

Verhalten:

$$y[0] = \begin{cases} -x[0] & \text{für } n=1 \text{ und } a=s=m=d=0 \\ 0 & \text{sonst} \end{cases}$$

$$y[n+1] = \begin{cases} -x[n+1] & \text{für } n=1 \text{ und } a=s=m=d=0 \\ x[n] + x[n+1] & \text{für } a = 1 \text{ und } n=s=m=d=0 \\ x[n] - x[n+1] & \text{für } s = 1 \text{ und } n=a=m=d=0 \\ x[n] * x[n+1] & \text{für } m=1 \text{ und } n=a=s=d=0 \\ x[n] / x[n+1] & \text{für } d=1 \text{ und } n=a=s=m=0 \\ 0 & \text{sonst} \end{cases}$$

- Implementierungsdokumentation
  - Beschreibt, wie die Spezifikation umgesetzt wird
  - Man unterscheidet Code-Dokumentation und externe Dokumente
  - Externe Dokumente liefern in der Regel eine Schnittstellenbeschreibung (API) und geben einen Bezug zur Gesamtarchitektur
  - Der Wert einer guten Implementierungsdokumentation (gerade für die Wartung) wird häufig unterschätzt
  - Dritte können sich schneller einarbeiten, und die Wahrscheinlichkeit für fehlerhafte Änderungen wird minimiert
  - Auf Grund der hohen Mitarbeiterfluktuation im IT-Bereich ist die Implementierungsdokumentation von großer Wichtigkeit

- *Beispiel*: Sie sind neu in der Abteilung. Es tritt eine Fehlberechnung im produktiven Betrieb auf. Sie müssen schnellstmöglich den Fehler in der folgenden Funktion finden. Der Autor der Funktion hat die Firma verlassen

```
int ack(int n, int m)
{
    while(n != 0) {
        if (m == 0) {
            m = 1;
        } else {
            m = ack(m, n-1);
        }
        n--;
    }
    return m+1;
}
```

Kommentare würden die Fehlerbehebung deutlich beschleunigen

```
/*
            Funktion:  int ack (int n, int m)
               Autor:  Tom Hacker
                Date:  15/08/2013
   Revision History
         12/05/2010:  While-Iterator ergänzt
         20/12/2012:  Funktionsname geändert
*/
int ack(int n, int m)
{
    /* Solange die erste Variable nicht null ist ... */
    while(n != 0) {
        /* Teste zweite Variable auf null */
        if (m == 0) {
            m = 1;
        } else {
            /* Hier erfolgt der rekursive Aufruf */
            m = ack(m, n-1);
        }
        n--;
    }
    /* Liefert Ergebniswert zurück */
    return m+1;
}
```

Können Sie den Fehler nun schneller finden?

Streichen Sie alle Kommentare, die nicht hilfreich sind!

```
/* Autor       : Tom Hacker              Erstellungsdatum: 15/08/2013
   Beschreibung: Berechnet die Ackermann-Funktion
                 Die Ackermann Funktion ist berechenbar, aber
                 nicht primitiv berechenbar. Die Funktion ist
                 wie folgt definiert:
                 (1) ack(0,m) = m+1                    [m >= 0]
                 (2) ack(n,0) = ack(n-1,1)             [n > 0]
                 (3) ack(n,m) = ack(n-1, ack(n, m-1)) [m,n > 0]
*/
int ack(int n, int m)
{
    while(n != 0) {
        if (m == 0) {
         /* Fall(2) */
            m = 1;
        } else {
         /* Fall(3) */
            m = ack(m, n-1);
        }
        n--;
    }
    /* Fall(1) */
    return m+1;
}
```

Optimal!

Hier wird dokumentiert, was gemacht wird, und nicht, wie es gemacht wird

- Dokumentationsextraktion
  - Implementierungsdokumentation und externe Dokumentation laufen mit der Zeit auseinander
  - Informationen sollten daher nur an einer Stelle dokumentiert werden (in der Regel im Quellcode)
  - Es gibt Werkzeuge, die den Quelltext nach gekennzeichneten Kommentaren durchsuchen und daraus automatisch eine externe Dokumentation generieren (z.B. im HTML, oder PDF-Format)

JavaDoc

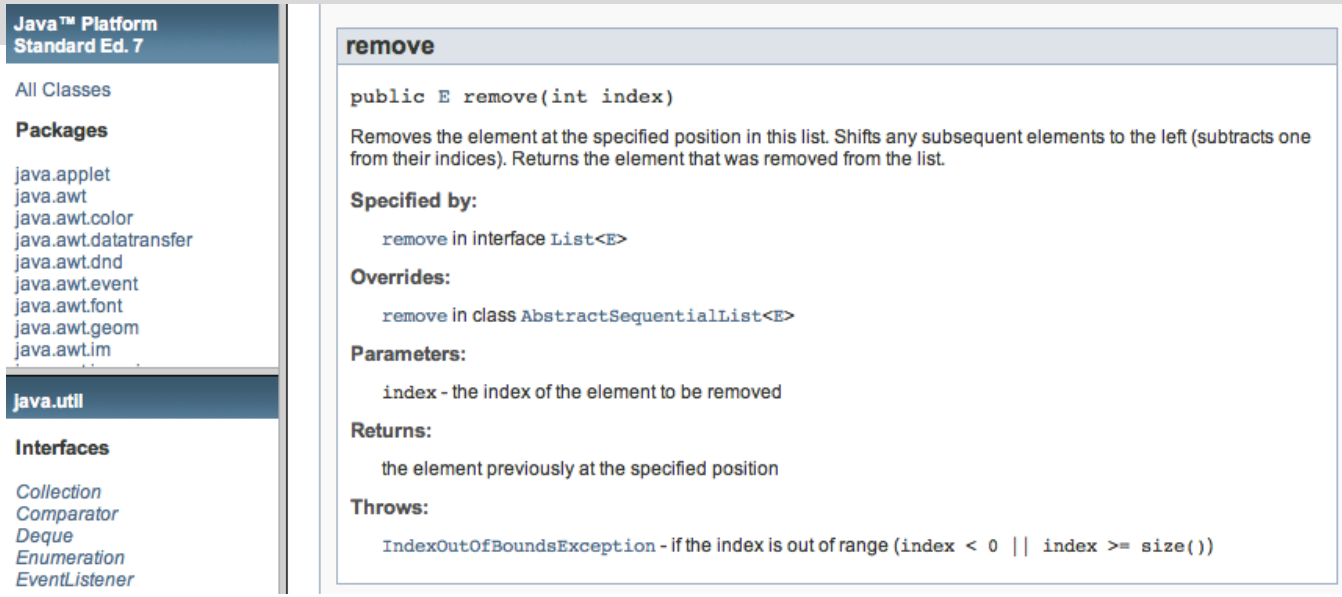
JavaDoc-Kommentare:
`/**  */`

Klassifikation der Kommentare:
Über Tags (mit @ als Prefix)

Tag	Bedeutung
@author	Name des Autors
@deprecated	Aktualitätsstatus
@exception	Ausgelöste Ausnahme
@param	Name und Funktion eines Parameters
@version	Version
@return	Rückgabewert
@since	Erstellungsdatum

```
/**
 * Removes the element at the specified position in this list.  Shifts any
 * subsequent elements to the left (subtracts one from their indices).
 * Returns the element that was removed from the list.
 *
 * @param index the index of the element to be removed
 * @return the element previously at the specified position
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public E remove(int index) {
   checkElementIndex(index);
   return unlink(node(index));
}
```

JavaDoc



8. Entwurf einer angemessenen Architektur
   - Komplexe Systeme sind schwer zu beherrschen und provozieren Fehler
   - Die Struktur eines Systems sollte übersichtlich und klar sein
   - Beim Entwurf sind insbesondere die folgenden Prinzipien zu befolgen
     - Modularisierung
     - Hierarchisierung
     - Starker Zusammenhalt und schwache Kopplung
     - Trennung von Zuständigkeiten
     - Information Hiding
   - Die Architektur hat auch einen direkten Einfluss auf die Qualitätsmerkmale Portabilität und Wartbarkeit

9. Automatisierung

- Immer wieder durchzuführende (mechanische) Routinetätigkeiten sind potentielle Fehlerquellen
  - Die Mitarbeiter werden nachlässig (Flüchtigkeitsfehler)
  - Die Zeit für die Routinetätigkeit kann nicht mehr in andere Maßnahmen zur Qualitätssicherung investiert werden
  - Neue Mitarbeiter müssen sich erst einarbeiten und machen zu Beginn viele Fehler
  - Wiederholbarkeit nicht gegeben
- Daher sind die folgenden Aktivitäten zu automatisieren
  1. Build-Prozess (inkl. Integration)
  2. Programmtest (Regressionstests)
  3. Versionierung
  4. Datensicherung der erstellten Artefakte
  5. Installation der Umgebungen (Entwicklung, Integration, Produktion)

- Wir werden nun das Thema Richtlinien vertiefen

- Um die allgemeine Lesbarkeit der Programme zu erhöhen, und um einen Standard zu etablieren, werden insbesondere **Richtlinien für die Codierung** vorgegeben (siehe `[LL10], [Hof13]`):
  - Wahl der Bezeichner
  - Layout (z.B. Einrückung)
  - Aufbau von Schnittstellen
  - Kommentierung
  - ...

- Notationsstile für Bezeichner
  - Pascal Case
    - `BackgroundColor`
    - `LinkedList`
  - Camel Case
    - `indexOf`
    - `toString`
  - Upper Case
    - `MAXEINTRAEGE`
    - `MAX_EINTRAEGE`
  - Lower Case
    - `indexof`
    - `index_of`

- Ungarische Notation
  - Variablenname wird um ein Präfix ergänzt, um den Datentyp vorzuhalten

Typ	Beispiel
Character	`char cKennzeichen;`
Byte	`byte byAlter;`
Integer	`int nAnzahl;`
String	`String sname;`
Pointer	`int* pAnzahl`
...	`...`

Verletzt das DRY-Prinzip

sollte nur für untypisierte Sprachen verwendet werden

- Abhängig von der Programmiersprache sollte zur Vereinheitlichung ein *Coding Style* vorgegeben werden
- Ein *Coding Style* enthält Richtlinien für
  - Schreibweise von Bezeichnern
  - Anweisungen
  - Einrückungstiefe
  - maximale Zeilenlänge
  - Umbrüche / Leerzeichen
  - Deklarationsschemata
  - Dateiorganisation und -namen
  - Programmierpraktiken
- Man kann sich an einem vorhandenen *Coding Style* orientieren
  - Java Coding Style
  - .NET Coding Style
  - Linux Kernel Coding Style
  - ...

Werkzeugunterstützung

- Java Coding Style
  - Namenkonvention

Kategorie	Notation	Beispiel
Package	Lower case	`java.awt.event`
Class	Pascal case	`class LinkedList`
Interface	Pascal case	`interface WindowListener`
Methode	Camel case	`indexOf`
Variable	Camel case	`index`
Konstante	Upper case	`SPEED_OF_LIGHT`

  - Einrückung: 4 Leerzeichen
  - Zeilenlänge: 80 Zeichen
  - Umbrüche: nach Komma, vor Operator
  - ...

http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

- Linux Kernel Coding Style
  - Namenskonvention: Lower case
  - Einrückung: 8 Zeichen
  - Zeilenlänge: 80 Zeichen
  - Umbrüche: Umgebrochene Zeilen werden immer kürzer und werden nach rechts eingerückt
  - ...

```
int ggt(int a, int b)
{
        while (a != b) {
                if (a > b)
                        a=a-b;
                else
                        b=b-a;
        }
        return a;
}
```

- .NET Coding Style (Design guidelines)
  - Namenskonvention: *Camel case* für Parameter und Klassenelemente (protected), ansonsten *Pascal case*

    http://msdn.microsoft.com/en-us/library/ms229042.aspx

- Die Notationskonventionen haben sich auf das Layout und die Syntax des Quelltextes bezogen
- Es können aber auch Sprachkonventionen getroffen werden
- Spachkonventionen berücksichtigen semantische Besonderheiten einer Sprache
  - Verbot von Sprachkonstrukten
  - Eingeschränkte (spezielle) Anwendung von Sprachkonstrukten

- MISRA-C Sprachkonvention
  - Wird seit 1998 von der *Motor Industry Software Reliability Association* (MISRA) vorgeschlagen

    Guidelines for the Use of the C Language in Vehicle Based Software

  - Regelsatz soll klassische Programmierfehler in der Programmiersprache C vermeiden
  - Erhöhung der Qualität eingebetteter Software durch die Vermeidung von fehleranfälligen Programmstrukturen
  - Mit der Version MISRA-2004 wurde die automatisierte Prüfung der Regel verbessert und der Anwendungsbereich ausgedehnt

    Guidelines for the Use of the C Language in Critical Systems

  - Es existieren 141 Regeln, die in 21 Kategorien aufgeteilt sind

- Regelkategorie der MISRA-2004-Sprachkonvention

Regeln	Kategorie	Regeln	Kategorie
(1.1)-(1.5)	Übersetzungsumgebung	(12.1)-(12.3)	Ausdrücke
(2.1)-(2.4)	Spracherweiterungen	(13.1)-(13.7)	Kontrollstrukturen
(3.1)-(3.6)	Dokumentation	(14.1)-(14.10)	Kontrollfluss
(4.1)-(4.2)	Zeichensatz	(15.1)-(15.5)	Switch-Konstrukt
(5.1)-(5.7)	Bezeichner	(16.1)-(16.10)	Funktionen
(6.1)-(6.5)	Datentypen	(17.1)-(17.6)	Pointer und Arrays
(7.1)	Konstanten	(18.1)-(18.4)	Struct und Union
(8.1)-(8.12)	Deklarationen und Def.	(19.1)-(19.17)	Präprozessor
(9.1)-(9.3)	Initialisierung	(20.1)-(20.12)	Standardbibliothek
(10.1)-(10.6)	Typkonversion (Arithmetik)	(21.1)	Laufzeitfehler
(11.1)-(11.5)	Typkonversion (Pointer)		

aus [Hof13]

- Auszug aus dem MISRA-2004-Regelsatz

Regel	Beschreibung
(1.4)	„The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.“
(2.2)	„Source code shall only use `/* ...*/` style comments“
(3.2)	„The character set and the corresponding encoding shall be documented“
(4.2)	„Trigraphs shall not be used“
(5.1)	„Identifiers shall not rely on the significance of more than 31 characters“
(6.4)	„Bit fields shall only be defined to be of the type `unsigned int` or `signed int`“
(7.1)	„Octal constants (other than zero) and octal escape sequences shall not be used
(8.5)	„There shall be no definitions of objects or functions on a header file“
(9.1)	„All automatic variables shall have been assigned a value before being used“
(10.6)	„A U-suffix shall be applied to all constants of unsigned type“
(14.1)	„There shall be no unreachable code“

nach [Hof13]

– Für natürlichsprachige Spezifikationen kann z.B. die Befolgung von sprachlichen Regeln als Richtlinie vorgegeben werden

	Regel	**Erläuterung (Beispiele)**
R1	Formulieren Sie jede Anforderung im Aktiv	nicht „die Eingabe wird gelöscht“
R2	Drücken Sie Prozesse durch Vollverben aus	nicht „ist“ und „hat“
R3	Vermeiden Sie unvollständig spezifizierte Verben	Wer macht was?
R4	Vermeiden Sie unvollständig spezifizierte Bedingungen	„Wenn-dann-sonst“
R5	Überprüfen Sie Universalquantoren	Gibt es bei „nie“, „immer“, „alle“ wirklich keine Einschränkungen?
R6	Vermeiden Sie Nominalisierungen	Besser Satz mit „anmelden“ als mit „Anmeldung“
R7	Präzisieren Sie unbestimmte Substantive	Gibt es einen oder mehrere „Bediener“?
R8	Klären Sie die Zuständigkeiten	Wer erzwingt bzw. verhindert etwas?
R9	Vermeiden Sie implizite Annahmen	

nach [LL10] und Rupp et al.

Manuelle Prüfmethoden

Manuelle Prüfmethoden

- Diagnostische Verfahren, um das existierende Qualitätsniveau eines Produktes zu ermitteln
- Ziele der Prüfung sind
  - die Feststellung von Mängeln, Fehlern, Inkonsistenzen und Unvollständigkeiten
  - die Feststellung von Verstößen gegen Vorgaben, Richtlinien, Standards und Pläne
  - formale Abnahme des Prüfobjekts
- Für semantische Überprüfungen geeignet
- Beschreibungsform der Prüfobjekte
  - informal (z.B. Pflichtenheft)
  - semiformal (z.B. Pseudocode)
  - formal (z.B. Quellcode, OOA-Modell)
- Breiter Einsatzbereich (Analyse, Entwurf, Codierung, Integration)

- Folgende Voraussetzungen müssen für den Einsatz manueller Prüfmethoden erfüllt sein:
  1. Aufwand und Zeit müssen eingeplant sein
  2. Jeder Prüfer ist in der Prüfmethode geschult
  3. Prüfergebnisse dürfen nicht für die Beurteilung von Mitarbeitern verwendet werden (Vorgesetzte nehmen nicht an der Prüfung teil)
  4. Prüfmethode muss dokumentiert sein und Einhaltung der Methode muss geprüft werden

- Im Folgenden werden vier manuelle Prüfmethoden vorgestellt

1. Durchsicht
   - Prüfung wird vom Entwickler alleine durchgeführt
   - Hilfreich: Artefakt ausdrucken und Bildschirm verlassen, ruhige Umgebung
   - Selbstverständliche Maßnahme, bevor ein Arbeitsergebnis weitergegeben wird
   - Durchsicht ist effizienter als ein Test (siehe `[LL10]`)

2. Stellungnahme

- Der Autor gibt das Artefakt an Dritte weiter und bittet diese, das Artefakt zu beurteilen
- Auf Grundlage der Rückmeldungen kann er dann das Artefakt überarbeiten

- Die Stellungnahme weist eine Reihe von Nachteilen auf:
  - Aufwand ist nicht geplant
  - Autor ist gleichzeitig Moderator
  - Die Prüfergebnisse sind nicht dokumentiert, die Nacharbeit wird nicht kontrolliert

3. Review (nach [LL10]):

- Formalisierter Prozess zur Überprüfung von schriftlichen Dokumenten
- Rollen
  - *Moderator* : leitet das Review, ist kein Vorgesetzter
  - *Autor* : Urheber des Prüflings, nimmt an der Review-Sitzung teil, um auf Nachfrage Unklarheiten zu beseitigen; ist kein Verteidiger des Prüflings
  - *Gutachter* : Mitarbeiter, der den Prüfling beurteilen kann
  - *Notar* : führt in der Review-Sitzung das Protokoll (evtl. der Moderator)
  - *Review-Team* : alle Teilnehmer des Reviews ohne Autor

- Ablauf eines Reviews
  (1) Autor beantragt ein Review beim Manager (z.B. Projektleiter)
  (2) Manager plant das Review und bestellt einen Moderator
  (3) Moderator (und Manager) legen die Aspekte fest, nach denen das Prüfobjekt später begutachtet werden soll
  (4) Für jeden Aspekt wird mindestens ein kompetenter Prüfer ausgewählt
  (5) Manager/Moderator führen eine Eingangsprüfung durch
  (6) Moderator verteilt mit der Einladung zur Review-Sitzung das Prüfobjekt, den Prüfauftrag und die Referenzunterlagen an die Gutachter
  (7) Jeder Gutachter prüft in der Vorbereitung das Artefakt nach den zugeteilten Gesichtspunkten
  (8) In der Review-Sitzung tragen die Gutachter die entdeckten Mängel vor. Sie erheben, gewichten und protokollieren die Befunde (max. 2 Stunden)
  (9) In einer dritten Stunde können sich Gutachter und Autor ohne Regeln und Protokolle austauschen (Austausch von Lösungsideen)
  (10) Manager entscheidet über Art und Umfang der Nacharbeit
  (11) Die Nacharbeiten werden vom Autor durchgeführt
  (12) Bei umfangreichen Änderungen wird das überarbeitete Dokument einem weiteren Review unterzogen

- Review-Regeln
  - Review-Sitzung ist auf zwei Stunden beschränkt
  - Moderator bricht die Sitzung ab, wenn sie nicht erfolgreich durchgeführt werden kann
  - Das Resultat, nicht der Autor wird geprüft
  - Die Rollen werden nicht vermischt
  - Allgemeine Stilfragen außerhalb der Richtlinien werden nicht diskutiert
  - Entwicklung und Diskussion von Lösungen ist nicht Aufgabe des Reviews
  - Jeder Gutachter darf seine Befunde angemessen präsentieren
  - Die Ergebnisse der Gutachter werden sofort protokolliert

- Die Befunde werden gewichtet:
  - *Kritischer Fehler* (Prüfling unbrauchbar)
  - *Hauptfehler* (Nutzbarkeit merklich beeinträchtigt)
  - *Nebenfehler* (Nutzbarkeit wenig beeinträchtigt)
  - *Gut* (kein Mangel festgestellt)
- Das Review-Team gibt eine der folgenden Empfehlungen ab:
  - *Akzeptieren ohne Änderung*
  - *Akzeptieren mit Änderung*
  - *Nicht akzeptieren*
- Das Review-Protokoll wird von allen Teilnehmern unterschrieben

Hinweis:

- In der Literatur ist auch eine weitere Unterscheidung zwischen Inspektion und Review zu finden `[Bal08]`
- Das Review ist dort weniger formalisiert als die Inspektion

- Der Aufwand für ein Review wird unterschiedlich eingeschätzt
  - In [Bal08] wird als optimale Prüfgeschwindigkeit in der Vorbereitung eine Seite (+/- ½) pro Stunde angegeben
    (es wird darauf hingewiesen, dass die individuelle Prüfgeschwindigkeit im Verhältnis 1:10 variieren kann)
  - Von Frühauf, Ludewig und Sandmayr wurden im Jahr 1995 die folgenden Daten veröffentlicht (aus [Bal08]):

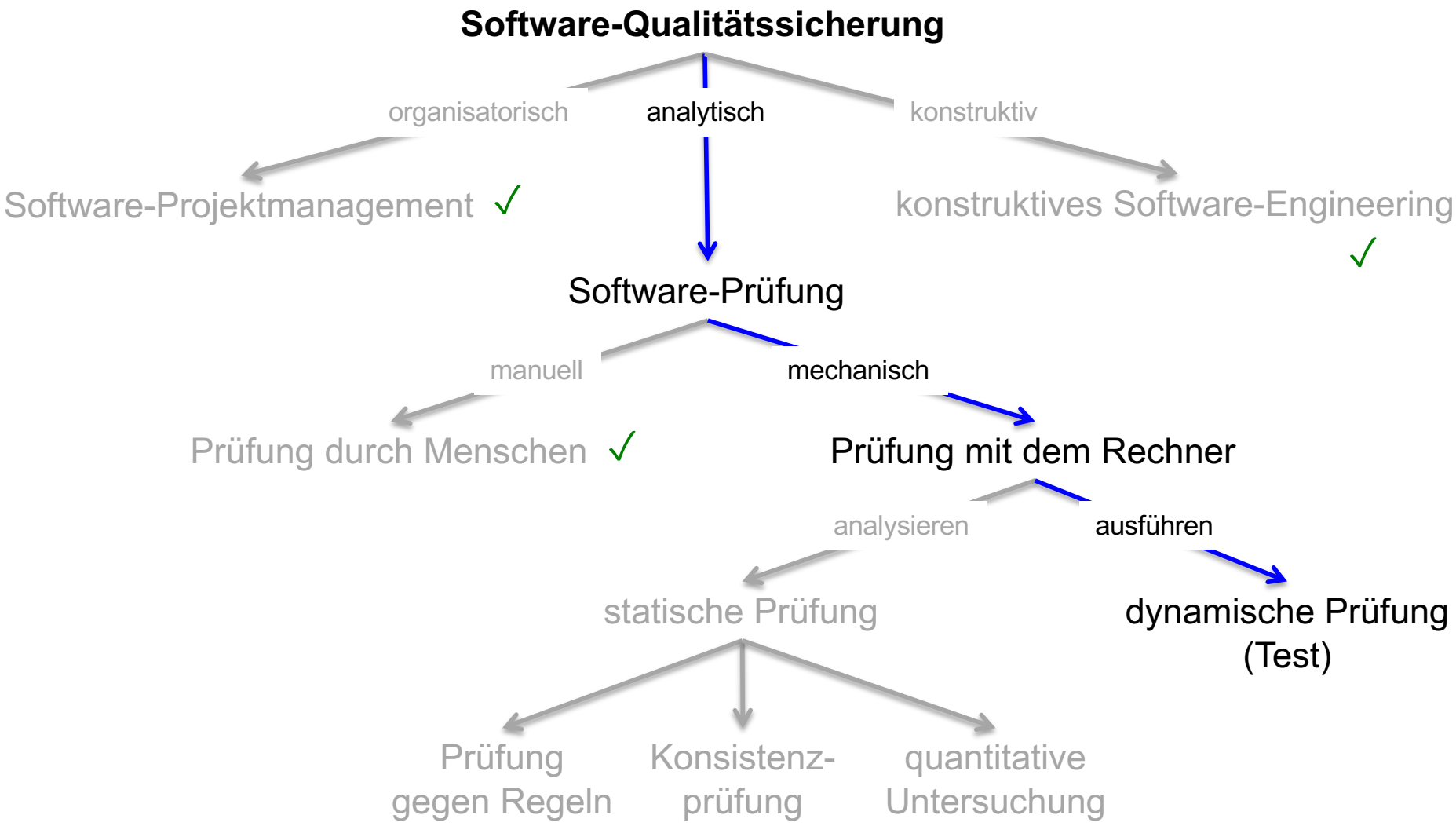
	Dokument	**Code**
Maximaler Umfang für Review	50 Seiten	20 Seiten
Zahl der Gutachter	5 (+ Moderator + Autor)	3 (+ Moderator + Autor)
Review-Vorbereitung relativ	10 Seiten/h	5 Seiten/h
Aufwand Review-Vorbereitung	25 Stunden	12 Stunden
Aufwand Review-Sitzung absolut	14 Stunden	10 Stunden
Summe Review-Aufwand	5 Personentage	3 Personentage
Erstellungsaufwand relativ	2 Seiten/Tag	1 Seite/Tag
Erstellungsaufwand absolut	25 Personentage	20 Personentage
Review zu Erstellungsaufwand	20%	15%

4. Structured Walkthrough

- Einfachere Variante des Review (Aufwand ist geringer, aber auch der Nutzen ist geringer)
- Der Autor lädt Gutachter (Kollegen) zu einer Walkthrough-Sitzung ein
  - Hier ist, falls möglich, ein etwas formalerer Ansatz ratsam
  - Autor meldet Artefakt zur Abnahme beim Projektleiter an
  - Der Projektleiter lädt zur Walkthrough-Sitzung ein
- Der Autor moderiert die Walkthrough-Sitzung und stellt dort seine Arbeitsergebnisse vor
- Die eingeladenen Gutachter stellen (spontane, vorbereitete) Fragen

Programmtest

Dynamische Prüfung



Gliederung der Software-Qualitätssicherung nach [LL10]

Programmtest

- Wir müssen die Fehlerfreiheit des fertigen Software-Systems nachweisen
- Aufgrund des erforderlichen Aufwands sind wir dazu in der Praxis nicht in der Lage
- Unser Ziel ist es daher, möglichst viele der vorhandenen Fehler durch Programmtests zu finden

> ***Testen*** *ist die Ausführung eines Programms auf einem Rechner mit dem Ziel, Fehler zu finden.*

nach [LL10]

- Das übersetzte Programm wird dazu ausgeführt und es werden konkrete Werte eingegeben
- Die Ausgaben (Ist-Werte) werden mit den Soll-Werten (aus der Spezifikation abgeleitet) verglichen
- Bei einer Abweichung zwischen Ist- und Soll-Wert liegt ein Fehler vor

- Ein spontanes Ausprobieren von Eingaben ist weder effizient noch sonderlich effektiv

- Wir konzentrieren uns daher auf **systematische Tests**:
  1. Randbedingungen (z.B. Systemumgebung) sind definiert
  2. Eingaben werden systematisch ausgewählt
  3. Soll-Werte werden vor dem Test festgelegt
  4. Der Testverlauf wird dokumentiert
  5. Test und Korrektur finden getrennt statt

- Ein systematischer Test ist reproduzierbar (vorausgesetzt, der Startzustand ist reproduzierbar). Damit ist der Test objektiv und wiederholbar

- Tests lassen sich nach der Prüfebene in verschiedene Teststufen einteilen
  - Unit-Test (Modultest, Komponententest)
  - Integrationstest
  - Systemtest
  - Abnahmetest

- Unit-Test (Modultest, Komponententest)
  - Es werden Programmteile getestet, die eine ausreichende Größe für einen eigenständigen Test haben
    - Funktionen/Methoden
    - Klassen
    - Komponenten / Verbund von Klassen
  - Wird in der Implementierung durchgeführt
  - Findet in der Entwicklungsumgebung statt

- Ein Testtreiber versorgt das Modul über Aufrufe der Schnittstelle mit Testdaten und nimmt die Antwort des Moduls entgegen
- Im Mittelpunkt steht der funktionale Test
- Optional
  - Test auf Robustheit
  - Test auf Effizienz

– Integration
  - Nachdem einzelne Module / Komponenten fertiggestellt und getestet wurden, müssen sie zu einem lauffähigen System zusammengebaut werden

> ***integration*** – *The process of combining software components, hardware components, or both into an overall system.*

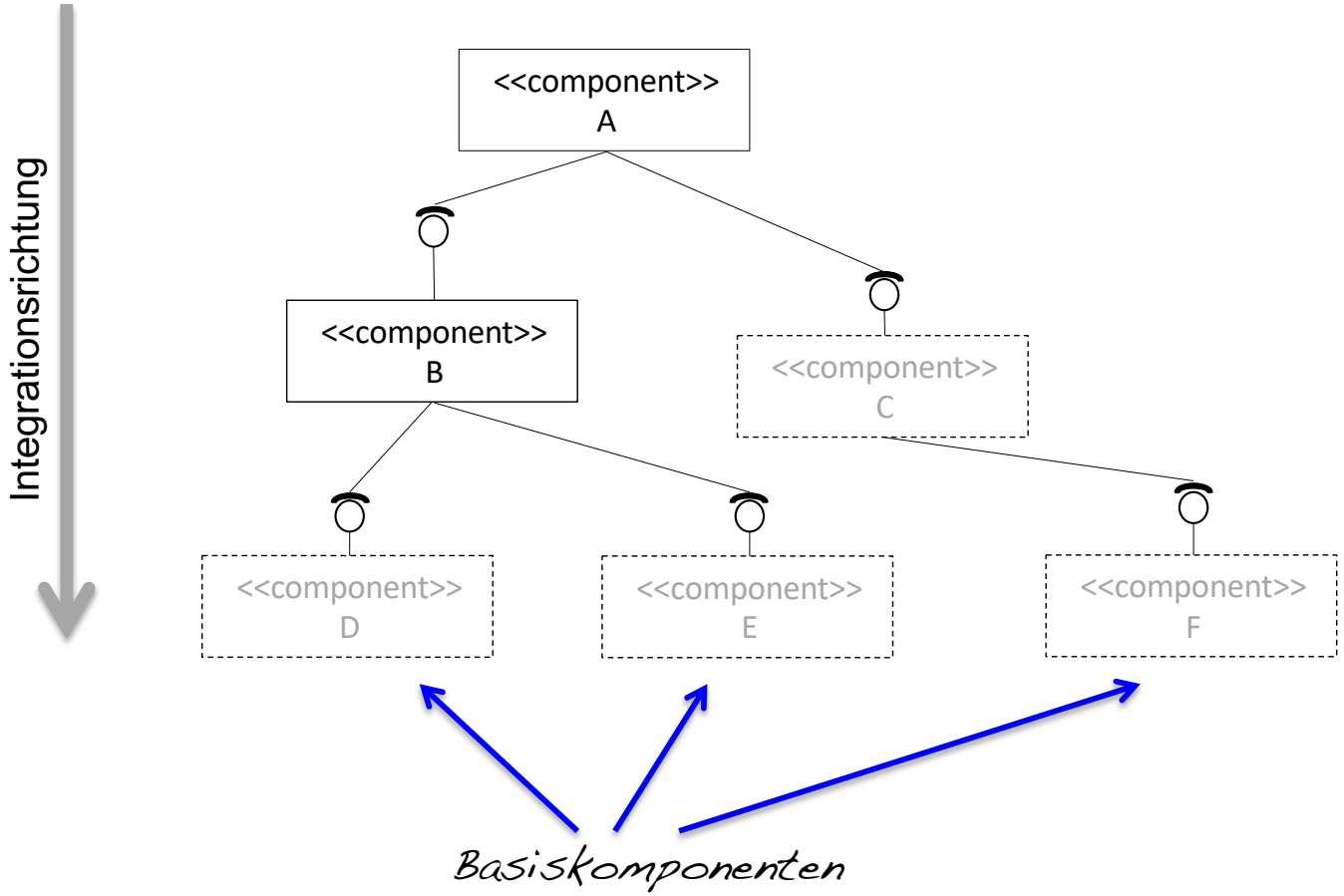
IEEE Std 610.12 (1990)

- Der Aufwand für die Integration darf nicht unterschätzt werden, da Fehler an den Schnittstellen auftreten können
- Der Grund für diese Fehler ist häufig eine unvollständige oder inkonsistente Spezifikation
- Bezüglich des Ablaufs unterscheidet man
  - die Integration in einem Schritt (Big-Bang-Integration)
  - die Inkrementelle Integration

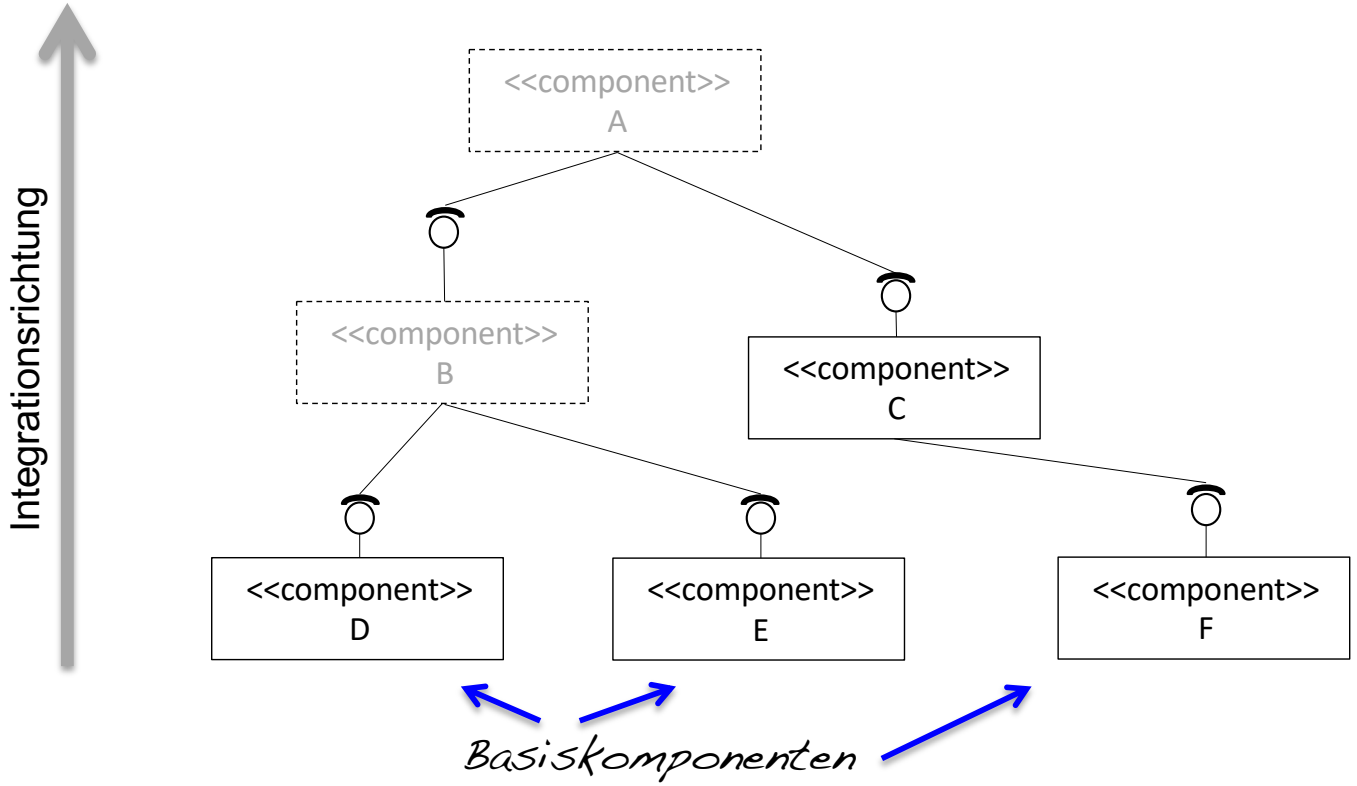
Warum kann in der Regel die Integration in einem Schritt in der Praxis nicht durchgeführt werden?

- Die inkrementelle Integration kann weiter differenziert werden
  - Top-down-Integration
  - Bottom-up-Integration
  - Outside-In-Integration
  - Kontinuierliche Integration

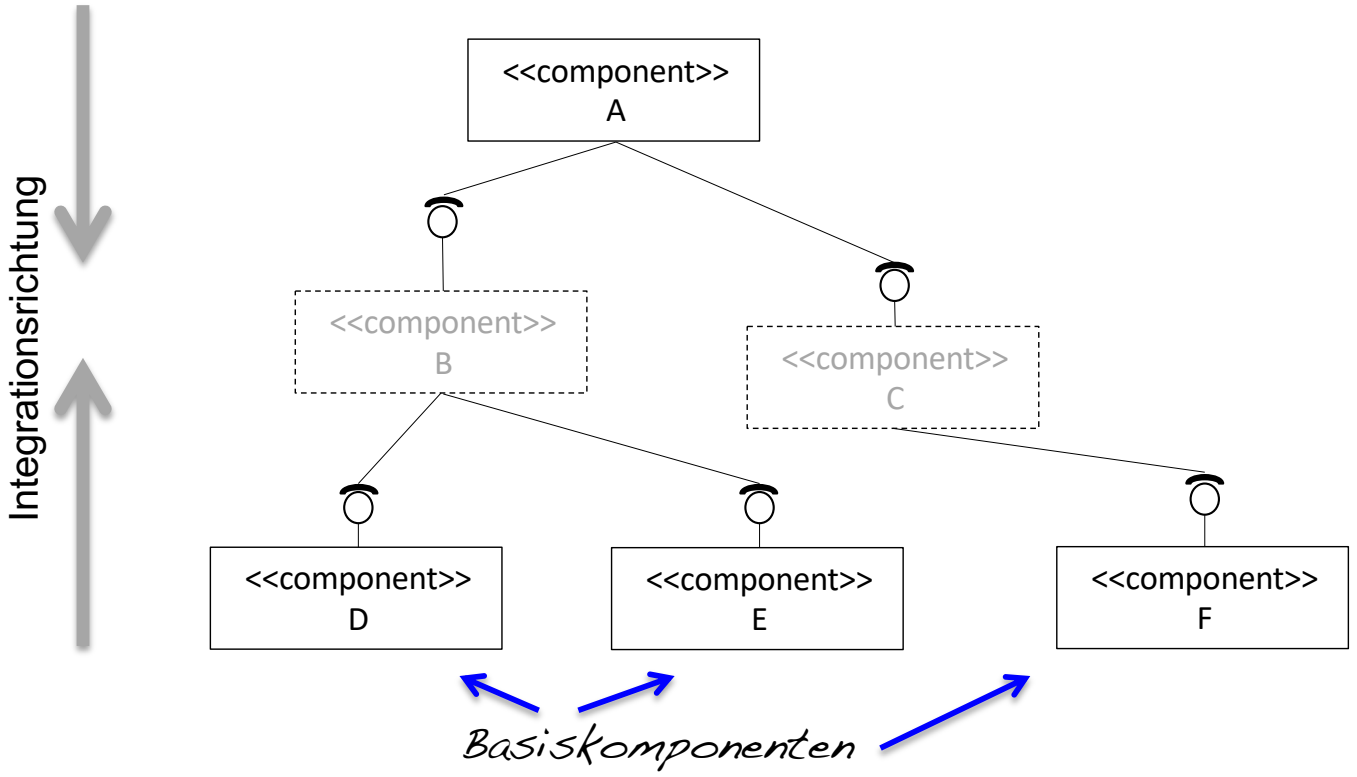
- Top-down-Integration
  - Die Integration beginnt auf der höchsten Hierarchieebene und arbeitet sich nach unten vor



- Bottom-up-Integration
  - Es wird mit den Komponenten begonnen, die keine Dienstleistungen anderer Komponenten benötigen (bis auf Betriebssystem-Dienste)
  - Danach werden die Komponenten integriert, die auf diese Basiskomponenten zugreifen



- Outside-in-Integration
  - Es werden die Komponenten der obersten und der untersten Schicht zuerst integriert
  - Danach arbeitet man sich von beiden Seiten zur Mitte vor

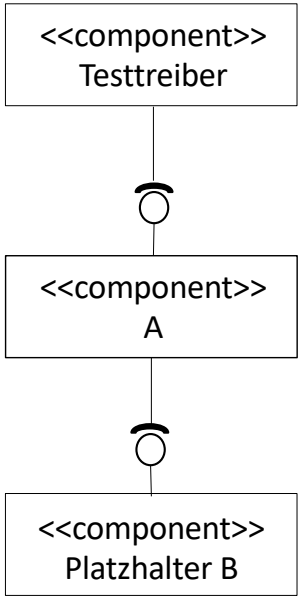


- Kontinuierliche Integration
  - Jede Komponente wird sofort integriert (siehe XP)
  - Es soll immer eine lauffähige (Teil-)Version in der Integrationsumgebung zur Verfügung stehen

- Auch die Integration muss geplant werden (Aufwand, Ablauf)
  - Termingetriebene Integration
  - Risikogetriebene Integration
  - Testgetriebene Integration

- Integrationstest
  - Es wird getestet, ob das Zusammenspiel der integrierten Module / Komponenten funktioniert
  - An den Schnittstellen können verschiedene Fehler auftreten
    - Inkompatible Schnittstellenformate
    - Protokollfehler
    - Semantische Fehler (unterschiedliche Einheiten)
    - Timing-Probleme
    - Kapazitäts- und Lastprobleme

- Falls eine Komponente *A* getestet werden soll, wird ggf. ein **Testtreiber** benötigt, der die Schnittstelle der Komponente *A* mit Testdaten versorgt
- Falls die Komponente *A* Dienste einer Komponente *B* nutzt, die noch nicht integriert ist, wird ein **Platzhalter** (*stub*) für *B* benötigt
- Der Platzhalter vertritt die fehlende Komponente und liefert entweder konstante Werte, oder simuliert das Verhalten der späteren Komponente in Ausschnitten



- Der Aufwand für die Erstellung von Platzhaltern und Testtreibern muss bei der Planung der Integration und der Integrationstests berücksichtigt werden

– Systemtest
  - Test des komplett integrierten Systems
  - Test aus Sicht des Anwenders
  - Validation, ob die Anforderungen vollständig und angemessen umgesetzt wurden
  - Testumgebung sollte der späteren Produktivumgebung möglichst ähnlich sein
  - Testtreiber und Platzhalter sind durch reale Komponenten ersetzt

- Abnahmetest
  - Erfolgt als letzter Test vor der Inbetriebnahme
  - Die Abnahmekriterien sind im Entwicklungsvertrag festgeschrieben (Individualsoftware)
  - Kunde ist in die Festlegung der Akzeptanztests involviert
  - Test aus Anwendersicht in der Abnahmeumgebung des Kunden

Grenzen von Programmtests

- Falls es für ein Programm *n* mögliche Eingaben gibt, und wir alle *n* Eingaben testen, dann haben wir einen vollständigen Test durchgeführt
- Treten bei einem vollständigen Test keine Fehler auf, dann ist die Korrektheit des Programms nachgewiesen

- Beispiel:
  - Wir wollen einen vollständigen Test für die Methode
    `public static long abs(long a)`
    der Klasse `java.lang.Math` durchführen
  - Da eine `long`-Variable eine Breite von 64-Bit hat, gibt es 2^{64} Testfälle
  - Falls ein Test nur eine Mikrosekunde dauern würde, müssten wir über 500000 Jahre auf den Nachweis der Korrektheit warten

nach [Hof13]

- Wir müssen also mit einer kleinen Teilmenge von Eingaben auskommen

> *Programm testing can be used to show the presence of bugs, but never to show their absence!*

E.W. Dijkstra (1970)

- Um mit den wenigen Eingaben trotzdem viele Fehler finden zu können, müssen wir die Eingaben geeignet wählen

- Die Festlegung der Eingabedaten reicht für einen systematischen Test nicht aus

- Ein **Testfall** beschreibt
  - die Ausgangssituation (Vorbedingungen, Randbedingungen)
  - Testdaten, die die vollständige Ausführung des Testobjekts bewirken
  - Sollwerte (das erwartete Ergebnis bzw. Verhalten)

- Die Sollwerte eines Testfalls werden aus der Spezifikation abgeleitet
- Oft können Testszenarien als Folge von Testfällen gebildet werden
- Ein guter Testfall sollte die folgenden Eigenschaften besitzen:
  - fehlersensitiv (er zeigt mit hoher Wahrscheinlichkeit einen Fehler an)
  - repräsentativ (er steht stellvertretend für viele andere Testfälle)
  - redundanzarm (er überdeckt keine anderen Testfälle)

Black-Box-Test (Funktionstest)

Die Testfälle werden auf Basis der in der Spezifikation geforderten Eigenschaften gewählt.
Die innere Beschaffenheit des Programms spielt keine Rolle.

- Der Black-Box-Test ist die wichtigste Form des dynamischen Tests
- Die Testfälle können/sollen bereits aufgestellt werden, sobald die entsprechende Spezifikation vorliegt
- Wir betrachten die folgenden Verfahren zur Bestimmung von Testfällen:
  ① Funktionale Äquivalenzklassenbildung
  ② Grenzwertanalyse
  ③ Zustandsbezogener Test
  ④ Test spezieller Werte
  ⑤ Zufallstest

1. Funktionale Äquivalenzklassenbildung
   - Angenommen, wir haben eine Menge $E=\{e_1,e_2,...,e_n\}$ von Eingaben, für die sich das Testobjekt gleich verhält
   - Dann reicht es aus, nur eine Eingabe aus dieser Menge zu testen (z.B. e_1)
   - Wenn diese Eingabe fehlerfrei ist, dann kann man davon ausgehen, dass auch die anderen Eingaben korrekt funktionieren
   - Eine solche Menge von Eingaben wird auch als Äquivalenzklasse bezeichnet
   - Jede Eingabe aus einer Äquivalenzklasse ist also repräsentativ für alle anderen Eingaben aus der Äquivalenzklasse
   - **Hinweis:** Der Begriff Äquivalenzklasse ist nicht im streng mathematischen Sinn zu sehen. In der Praxis können wir häufig nur vermuten, dass Eingaben äquivalent sind. Man spricht dann auch von schwacher Äquivalenz

Aufgabe: Welche Äquivalenzklassen lassen sich für die Methode `long abs(long a)` bilden?

- Neben gültigen Äquivalenzklassen gibt es aber auch ungültige Äquivalenzklassen (evtl. leer)
- Die ungültigen Äquivalenzklassen enthalten ungültige Eingabewerte
- Es ist natürlich auch zu prüfen, wie sich das Testobjekt bei ungültigen Eingaben verhält
  a) Angemessene Reaktion auf die ungültige Eingabe (Meldung, mit Möglichkeit der Korrektur)
  b) Ignorieren der ungültigen Eingabe
  c) Automatische Korrektur der ungültigen Eingabe
  d) Fehlberechnung
  e) Programmabbruch

} unerwünschtes Verhalten

- Beispiele für die Bildung von gültigen und ungültigen Äquivalenzklassen

Eingabe	gültige Äquivalenzklasse	ungültige Äquivalenzklasse
eine ganze Zahl x zwischen 1 und 31	ganze Zahl x mit $1 \leq x \leq 31$	1. ganze Zahl $x < 1$ 2. ganze Zahl $x > 31$
Zeichenketten, die mit einem großen Buchstaben beginnen	Zeichenketten, die mit einem großen Buchstaben beginnen	1. Zeichenketten, die mit einem kleinen Buchstaben beginnen 2. Zeichenketten, die mit einem Sonderzeichen beginnen 3. Zeichenketten, die mit einer Zahl beginnen
Rot, Grün, Blau	{Rot, Grün, Blau}	Eingabe nicht aus {Rot, Grün, Blau}

- Konstruktion von Testfällen
  - Für jeden Eingabeparameter werden Äquivalenzklassen gebildet
  - Für jeden einzelnen Parameter gibt es mindestens zwei Äquivalenzklassen (eine gültige und eine ungültige)
  - Die Repräsentanten aller gültiger Äquivalenzklassen werden zu Testfällen kombiniert. Alle Kombinationen sind zu bilden
  - Ein Repräsentant einer ungültigen Äquivalenzklasse wird nur mit gültigen Repräsentanten kombiniert. Pro ungültige Äquivalenzklasse (mindestens) 1 Testfall
  - Zu jedem Testfall wird der Sollwert bestimmt

- Aufgabe: An der Schnittstelle einer Komponente können die drei Parameter *a*, *b* und *c* übergeben werden. Für *a* wurden vier Äquivalenzklassen gebildet. Davon sind zwei Äquivalenzklassen ungültig. Für *b* existieren zwei gültige und eine ungültige Äquivalenzklasse. Für *c* existiert eine gültige und eine ungültige Äquivalenzklasse. Wie viele Testfälle sind nach der obigen Regel zu konstruieren?

- Abhängig von der Anzahl der Parameter und der (gültigen) Äquivalenzklassen kann die Anzahl der betrachteten Testfälle sehr groß werden
- Es gibt verschiedene Möglichkeiten, die Anzahl der Testfälle geeignet zu reduzieren
  - Äquivalenzklassen verschiedener Parameter, die sich überlagern, können verschmolzen werden
  - Typische Testfälle bevorzugen
  - Paarweise Kombination
  - Jeder Repräsentant einer Äquivalenzklasse kommt in mindestens einem Testfall vor

- Ein Testende-Kriterium kann über eine zu erreichende Äquivalenzklassen-Überdeckung vorgegeben werden

  Überdeckung = (*Anzahl getesteter ÄK* / *Gesamtzahl ÄK*) x 100

2. Grenzwertanalyse
   - Falls die Elemente einer Äquivalenzklasse eine natürliche Ordnung besitzen, dann sollten Elemente aus der Äquivalenzklasse gewählt werden, die an den Grenzen liegen
   - Die Erfahrung hat gezeigt, dass Eingaben, die die Grenzen von Äquivalenzklassen bilden, häufig Fehler aufdecken
   - Die Annäherung an die Grenzen kann vom gültigen und vom ungültigen Bereich der Äquivalenzklasse aus erfolgen
   - In einigen Situationen können Fehler nur gefunden werden, wenn die Grenzen mit drei Werten geprüft werden
     ① der exakte Grenzwert
     ② der zum Grenzwert benachbarte Wert innerhalb der Äquivalenzklasse
     ③ der zum Grenzwert benachbarte Wert außerhalb der Äquivalenzklasse

Übungsaufgabe

Die Methode

`public static boolean istErstesHalbjahr(int monat)`

bekommt einen Monat als ganze Zahl übergeben und entscheidet, ob es sich um einen Monat aus dem ersten Halbjahr handelt. Bilden Sie die Äquivalenzklassen und bestimmten Sie die Testfälle mit Hilfe der Grenzwertanalyse.

Setzen Sie an den markierten Stellen im Quellcode fehlerhafte Vergleichsoperatoren (<,>,<=,>=,!=,==) ein. Mit welchem Testfall wird der Fehler erkannt?

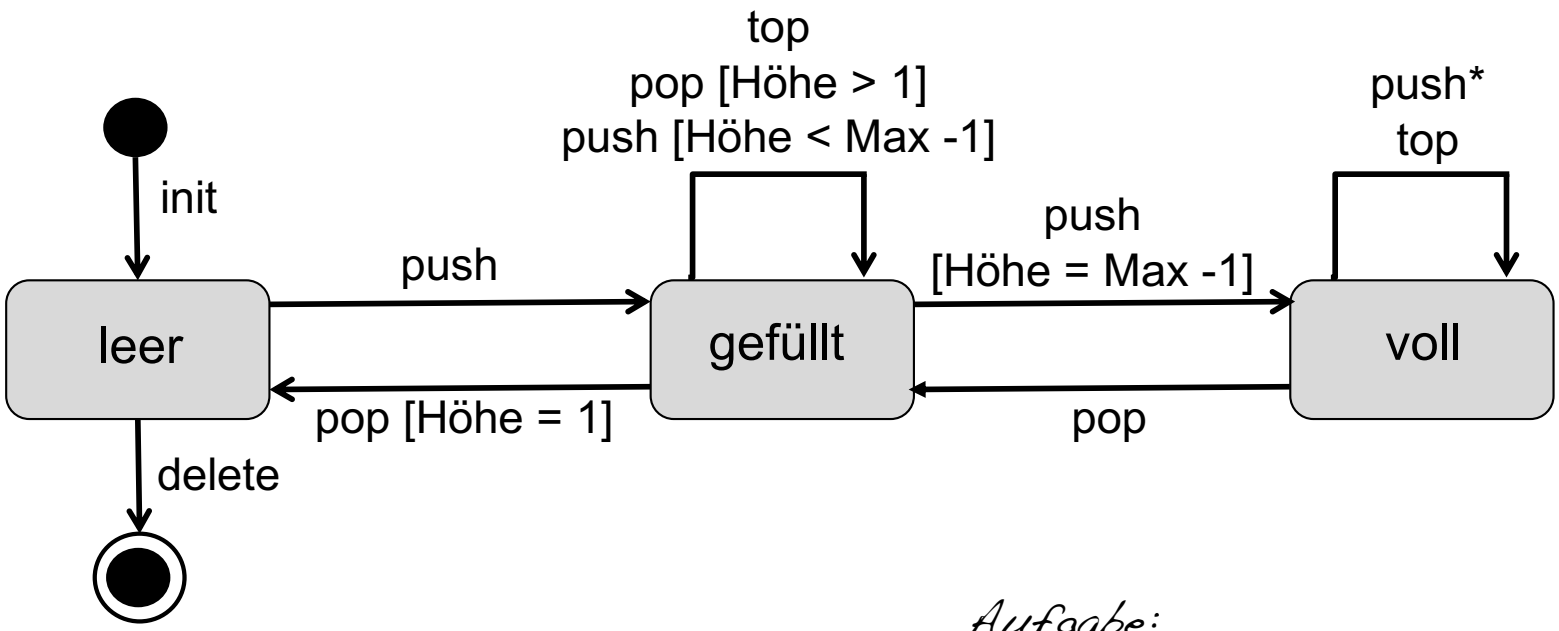
```
public static boolean istErstesHalbjahr(int monat){
   if ((monat □ 1) || (monat □ 12)) throw new IllegalArgumentException();
   if(monat □ 7) return true;
   return false;
}
```

3. Zustandsbezogener Test

- Es gibt Systeme, bei denen die Ausgabe nicht nur von der aktuellen Eingabe abhängt, sondern auch vom Ablauf der vorherigen Eingaben
- Ein solches System kann sich, abhängig von der Verarbeitungshistorie, in verschiedenen Zuständen befinden
- Die Reaktion auf eine Eingabe ist abhängig vom aktuellen Zustand
- Solche Systeme können mit Zustandsautomaten modelliert werden
- Ein Zustandsdiagramm enthält
  - Zustände
  - Anfangs- und Endzustand
  - Zustandsübergänge
  - Guards (Wächter) als Bedingung für einen Zustandsübergang

- Als Beispiel modellieren wir einen Stapel (*Stack*) durch einen Zustandsautomaten

nach [SL12]



- Minimalanforderung an einen Test
  a. Testfall erreicht alle Zustände
  b. Testfall ruft alle Funktionen auf

Aufgabe:
Erzeugen Sie für jede der beiden Minimalanforderungen einen passenden Testfall (mit Max=4)!

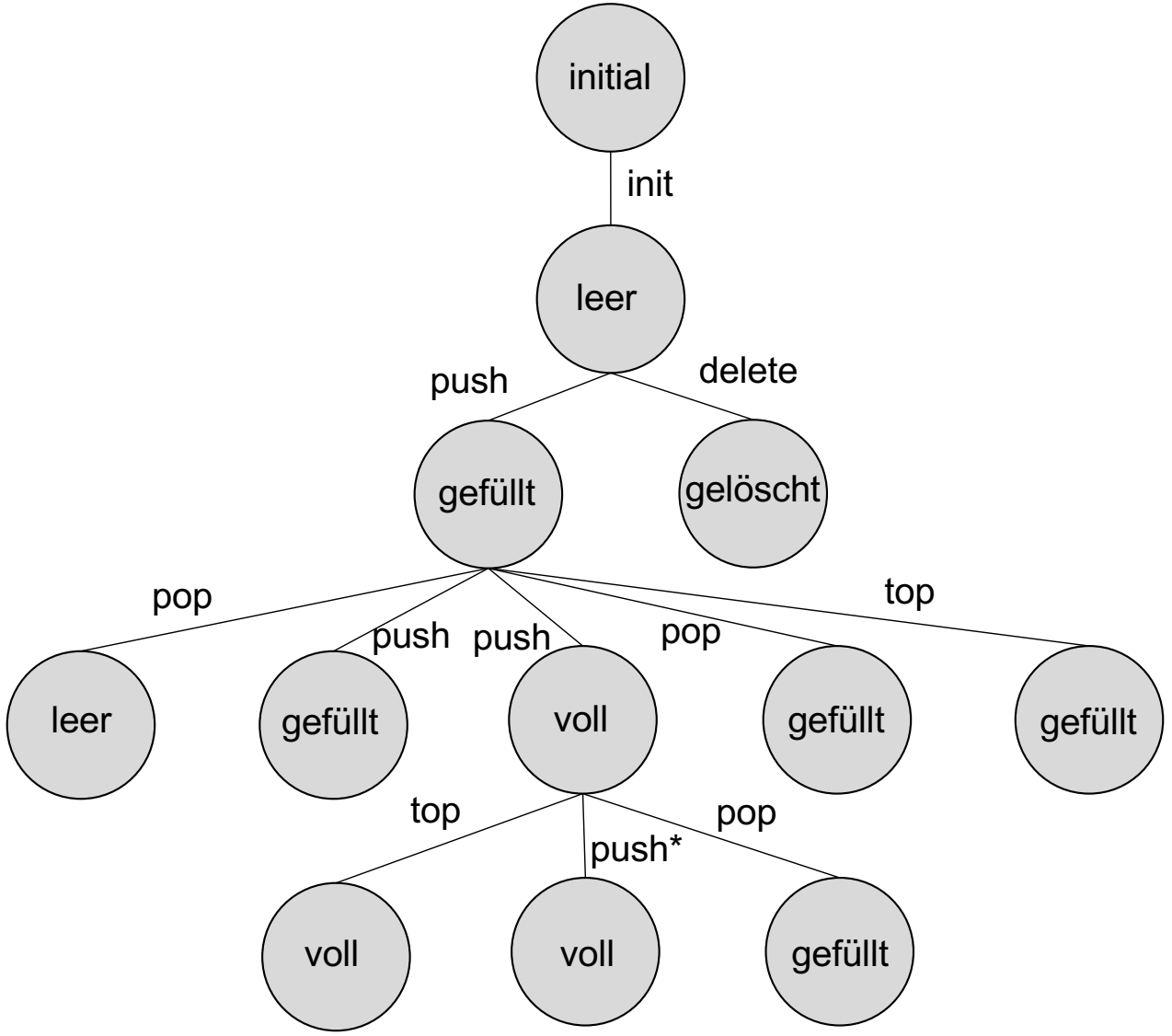
Wo liegen die Schwächen der beiden Testfälle?

- Testkriterium für einen zustandsbasierten Test

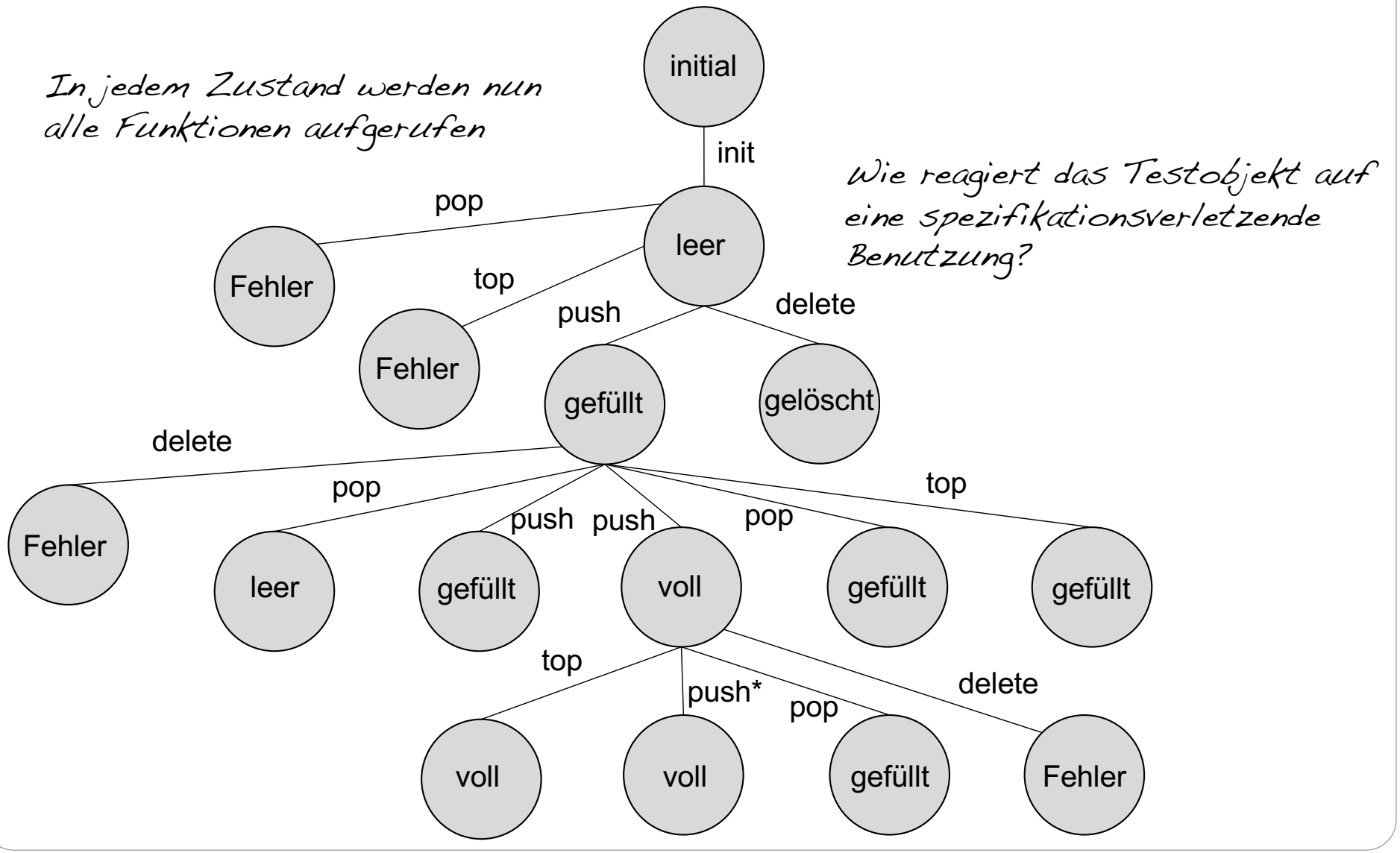
> *Für jeden Zustand sollen alle für diesen Zustand spezifizierten Funktionen mindestens einmal zur Ausführung kommen.*

- Zur Ermittlung der Testfälle wird aus einem zyklischen Zustandsdiagramm ein Entscheidungsbaum erstellt
- Der Entscheidungsbaum enthält alle Zustände und alle Zustandsübergänge
- Erzeugung des Entscheidungsbaums (Aufruf mit dem Startzustand)
  1. Für den aktuellen Zustand wir ein Knoten im Baum erzeugt
  2. Die Folgezustände werden als Söhne des aktuellen Knotens in den Baum eingehangen
  3. Die Folgezustände werden rekursiv traversiert, bis
     a. ein bereits besuchter Zustand erneut besucht wird
     b. ein Endzustand erreicht wird

- Übergangsbaum für den Stapel (Stack)



- Für einen Robustheitstest ist der Übergangsbaum zu erweitern



- Jeder Pfad im Übergangsbaum von der Wurzel bis zu einem Blatt repräsentiert einen Testfall
- Hinweis: Die Wächterbedingungen sind im Übergangsbaum nicht enthalten. Diese müssen aber bei der Bestimmung der Testfälle berücksichtigt werden. Z.B. dreimal push von „gefüllt“ nach „voll“ bei Max = 4.
- Definition Testfall
  - Anfangszustand
  - Eingabe
  - Soll: Ausgabe bzw. Verhalten
  - Soll: Endzustand
  - Pro Zustandsübergang: Ausgangszustand, Auslöser, Reaktion (Soll), Folgezustand (Soll)

Übungsaufgabe

Leiten Sie aus dem Übergangsbaum für den Stapel die enthaltenen Testfälle ab (ohne Robustheitstest).

- Der aktuelle Zustand eines Testobjekts ist häufig nur schwer zu ermitteln
- Überprüfung der Testfälle kann sehr aufwendig sein
- Überdeckungskriterien:
  - *x* Prozent der Zustände wurden mindestens einmal erreicht
  - *x* Prozent der Zustandsübergänge wurden mindestens einmal ausgeführt
  - *x* Prozent der spezifikationsverletzenden Zustandsübergänge wurden geprüft

4. Test spezieller Werte
   - Mit zunehmender Testerfahrung stellt man fest, dass bestimmte Eingaben, unabhängig von der konkreten Spezifikation, häufig Fehler aufdecken
   - Diese fehlersensitiven Eingaben sind zu dokumentieren und können dann in späteren Tests verwendet werden
   - Beispiele:
     - eine leere Eingabe
     - Eingabe mit Sonder- und Steuerzeichen

5. Zufallstest
   - Die Testfälle werden zufällig generiert
   - Die Datentypen und die Struktur der Eingaben müssen bei der zufälligen Generierung der Testdaten berücksichtigt werden
   - Eine entsprechende Werkzeugunterstützung ist erforderlich
   - Der Zufallstest ist ein ergänzendes Verfahren (z.B. können die Eingaben aus einer Äquivalenzklasse zufällig gewählt werden)
   - Der Vorteil von Zufallstests ist, dass auch nicht naheliegende Eingaben überprüft werden

Übungsaufgabe

Was spricht in der Praxis gegen die Durchführung von Zufallstests?

– Das *Property-based Testing* bietet ggf. eine praktikable Alternative zum Zufallstest

Property-based Testing

- Beim Unit-Test haben wir für Testfälle *x* geprüft, ob

$$f_{SUT}(x) = f_{Soll}(x)$$

  - Bei einer Abweichung liegt ein Fehler vor
  - Dabei wird $f_{Soll}(x)$ nicht berechnet, sondern ist in Form von Tabellen vorgegeben (in die Testfälle codiert). Dies erschwert Zufallstests

- Beim *property-based Testing* wird geprüft, ob ein Prädikat erfüllt ist

$$\forall x.P(x, f_{SUT}(x))$$

- Dabei ist f_{SUT} die zu testende Funktion und *P* ist eine ausführbare logische Relation zwischen der Eingabe *x* und der Ausgabe $f_{SUT}(x)$
  - Die Eingaben *x* können nun zufällig (in großer Anzahl) generiert werden
- Sobald das Prädikat nicht erfüllt ist, liegt ein Fehlschlag vor (dies muss nicht zwingend ein Fehler in f_{SUT} sein)

Aufgabe:

a) Es soll die statische Methode `String Util.reverse(String text)` getestet werden. Die Methode soll die Reihenfolge der Buchstaben im Text umkehren, z.B. `Util.reverse("Text").equals("txeT") == true` Welche Eigenschaft aus der Spezifikation von `reverse` hilft Ihnen dabei, das Prädikat zu formulieren? Hinweis: Es handelt sich um eine Invariante, die für alle Eingaben gilt.

b) Es soll die Methode `void Util.bubbleSort(Comparable[] feld)` getestet werden. Was wäre eine geeignete Möglichkeit, um das Prädikat zu formulieren?

c) Warum ist ein Fehlschlag (*failure*) nicht unbedingt ein Programmfehler (*error*)

- Ein bekanntes Werkzeug für das *property-based Testing* ist *QuickCheck* von Claessen und Hughes
  - Das Werkzeug generiert die Eingaben zufällig. Die Zeugen für einen Fehlschlag können daher unnötig komplex sein. Dies erschwert die Ursachensuche
  - Daher werden Zeugen von *QuickCheck* verkleinert (*shrinking*)

- QuickCheck ermöglicht die Betrachtung von extrem vielen Testfällen (im Vergleich zu klassischen Unit-Tests). Es gibt aber auch Einschränkungen
  - Es ist nicht ausgeschlossen, dass Eingaben mehrfach generiert werden
  - Es ist nicht ausgeschlossen, dass eine große Anzahl uninteressanter Eingaben generierte wird
  - Es existieren daher diverse Erweiterungen des Verfahrens (z.B. Berücksichtigung der bisherigen Code-Überdeckung bei der Generierung von Eingaben, kombinatorisches Vorgehen).

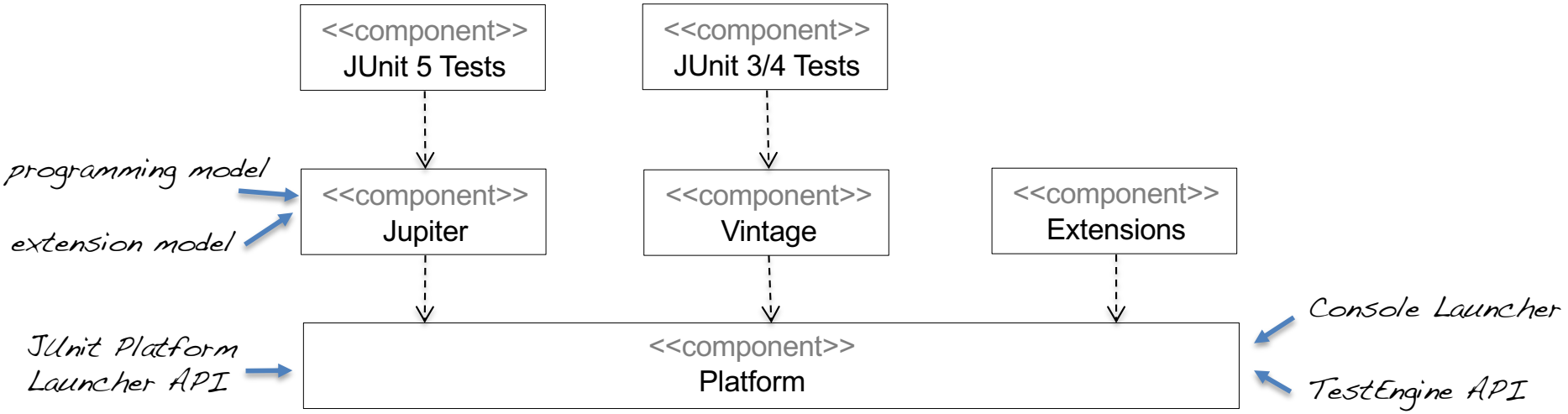
- Literaturauswahl:
  - Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Odersky, M., Wadler, P. (eds.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP ‘00), Montreal, Canada, September 18-21, 2000. pp. 268-279, ACM, 2000
  - Goldstein, H., Hughes, J., Lampropoulos, L., Pierce, B.: Do Judge a Test by its Cover. Combining Combinatorial and Property-Based Testing. In: Yosihda, N. (eds.) ESOP 2021, LNCS 12648, pp. 264-291, 2021

Testautomatisierung

- Der Vorteil von systematischen Tests ist die Wiederholbarkeit der Tests
- Wiederholung eines Tests auf Grund von
  - Programmänderungen (Fehlerkorrektur) -> Fehlernachtest
  - Programmerweiterungen -> Regressionstests
- Mit der Wiederholung von Tests können Modifikationen am Programm abgesichert werden
- Eine häufige Wiederholung erfordert eine Testautomation
- Die Testautomation kann auf Basis existierender Test-Frameworks und Werkzeugen erfolgen (Reduzierung des Aufwands)
- Als Beispiel betrachten wir das JUnit-Framework für Java-Programme

- JUnit bietet Unterstützung bei der Erstellung und der Durchführung von Programmtests
  - Entlastung von Routinetätigkeiten bei der Testfallprogrammierung
  - Automatisierung der Testdurchführung
  - Zählen und Berichten von Fehlern

- In der Praxis sind die Versionen JUnit 4 und JUnit 5 verbreitet. Wir betrachten JUnit 5 ← einen kleinen Ausschnitt
- JUnit 5 wurde zu einer Testplattform ausgebaut



- JUnit 5 - Testklasse
  - Die Testklasse muss weder eine spezielle Basisklasse noch einen speziellen Klassennamen besitzen (Empfehlung: der Klassenname erhält als Postfix den Zusatz `Test`)
  - Testmethoden werden durch die Annotation `@Test` ausgezeichnet

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class StringTest {

    @Test
    public void testStringtoUpperCase(){
        final String istWert = "Das ist ein Testfall.";
        final String sollWert = "DAS IST EIN TESTFALL.";

        assertEquals(sollWert, istWert.toUpperCase());
    }
}
```

Empfehlung: Nur ein Testfall pro Testmethode

- Falls eine Testklasse mehrere Testmethoden enthält, dann wird vor der Ausführung jeder einzelnen Testmethode eine neue Instanz der Testklasse erzeugt (Reduzierung von Seiteneffekten)

- Werden zur Erstellung einer konsistenten Testumgebung umfangreichere Initialisierungen benötigt, können diese in eine Methode ausgelagert werden, die mit `@BeforeEach` annotiert ist
  - die Methode wird dann vor jeder einzelnen Testmethode aufgerufen

- Die Freigabe von Ressourcen kann dann in einer mit `@AfterEach` annotierten Methode erfolgen.

- Wichtige JUnit 5-Annotationen
  - die annotierten Methoden sind `public`, `void` und ohne Parameter

`@Test`	Kennzeichnet eine Methode (`public`, `void` und ohne Parameter) als Testfall
`@BeforeEach`	Die gekennzeichnete Methode wird vor jeder Testmethode aufgerufen
`@AfterEach`	Die gekennzeichnete Methode wird nach jedem Testfall aufgerufen
`@BeforeAll`	Die gekennzeichnete statische Methode wird für eine Testklasse einmal ausgeführt, bevor die Testmethoden gestartet werden
`@AfterAll`	Die gekennzeichnete statische Methode wird für eine Testklasse einmal ausgeführt, nachdem die Testmethoden ausgeführt wurden
`@DisplayName("")`	Für Testklassen und Testmethoden können Namen für das Testprotokoll vergeben werden

  - Es können beliebig viele Methoden mit `@BeforeEach`, `@AfterEach`, `@BeforeAll` und `@AfterAll` gekennzeichnet werden
  - Die Reihenfolge der Ausführung ist nicht definiert

- Innerhalb einer Test-Methode findet ein Soll-Ist-Vergleich statt.
  - Für den Vergleich stehen eine Reihe von `assert`-Methoden zur Verfügung
  - Die `assert`-Methoden sorgen auch für eine Protokollierung des Testverlaufs

– Wichtige `assert`-Methoden

bei assertEquals und assertSame wird erst der Soll-Wert und dann der Ist-Wert angegeben

Methode	Beschreibung
`assertEquals(Object A, Object B, String message)`	Prüft `A.equals(B)`
`assertEquals(int a, int b, String message)`	Prüft `(a==b)`
`assertEquals(double a, double b, double v, String message)`	Prüft `abs(a-b)<=v`
`assertSame(Object A, Object B, String message)`	Prüft `(A==B)`
`assertTrue(boolean b, String message)`	Prüft `(b == true)`
`assertNotNull(Object A, String message)`	Prüft `(A != null)`

Fehlermeldung für JUnit-Protokoll, darf auch entfallen

– Eine beliebige Anzahl von Überprüfungen lässt sich mit `assertAll` und Lambda-Ausdrücken zusammenfassen

```
@DisplayName("Rationale Zahl“)
class RationalTest
    @DisplayName("multipliziert mit rationaler Zahlen")
    @Test
    public void testMultiply() {
       Rational l = new Rational(1,2);
       Rational r = new Rational(3,2);
       Rational e = l.multiply(r);
       assertAll("Fehler bei der Multiplikation",
                 ()-> assertEquals(3, e.getZaehler(),
                                   "Zähler falsch"),
                 ()-> assertEquals(4, e.getNenner(),
                                   "Nenner falsch"));
    }
```

- Erwartete Ausnahmen
  - Auf ungültige Eingaben muss das zu testende Programm geeignet reagieren
  - Eine Möglichkeit besteht im Auslösen (Werfen) einer Ausnahme (`Exception`)
  - Ein ungültiger Testfall würde in diesem Fall einen Fehler aufdecken, wenn keine Ausnahme ausgelöst wird
  - Diese Situation muss im Testfall geeignet abgebildet werden

- Wir testen die folgende Klasse

```
public class Calculator {
    public int add(int a, int b){
        return a+b;
    }

    public int divide(int n, int d){
        if(d == 0) throw
                    new ArithmeticException("Division durch 0");
        return n/d;
    }
}
```

die Methoden sind hier nur aus didaktischen Gründen nicht static (siehe @BeforeEach auf der nächsten Seite)

nur zur Verdeutlichung, nicht zwingend erforderlich

- Erwartete Ausnahme mit der Assertion `assertThrows`

```
public class CalculatorTest {
   private Calculator cal;

   @BeforeEach
   public void erzeugeCalculator(){
      cal = new Calculator();
   }

   @Test
   public void testDivision(){
      assertEquals(2, cal.divide(2,1));
   }

   @DisplayName("Test mit ungültigen Wert 0")
   @Test
   public void testDivisionDurchNull() {
      assertThrows(ArithmeticException.class,
                   ()->cal.divide(2,0),
                   "erwartete Exception nicht geworfen");

   }
}
```

Typ der erwarteten Ausnahme

Aufruf des SUT über Lambda-Ausdruck

Optionaler Text für das Testprotokoll

- Starten der Testfälle
  - über die Console

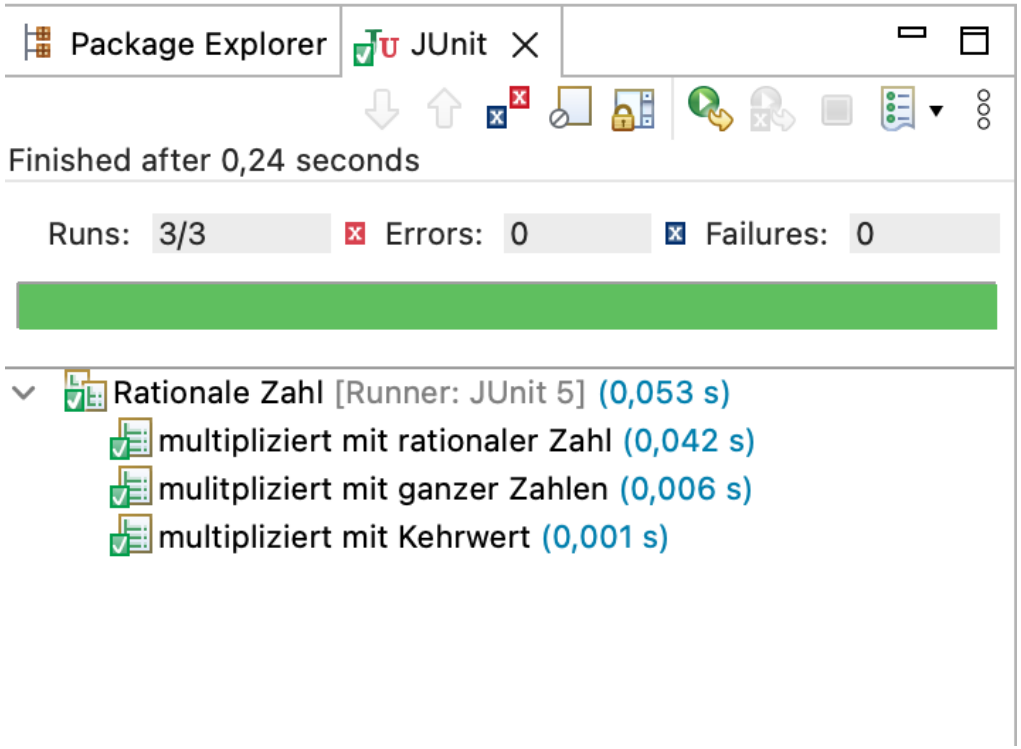
Die jar-Datei für den ConsoleLauncher kann über das Maven-Central-Repository bezogen werden

```
> java -jar junit-platform-console-standalone-1.9.1.jar
      --classpath /Users/dwiesmann/eclipse-workspace/SWTD/bin
      -c fh.swtd.rational.RationalTest
```

```
SWTD — -zsh — 80×31

dwiesmann@Dirks-MBP-16 SWTD % java -jar junit-platform-console-standalone-1.9.1.
jar --classpath /Users/dwiesmann/eclipse-workspace/SWTD/bin -c fh.swtd.rational.
RationalTest

Thanks for using JUnit! Support its development at https://junit.org/sponsoring

├─ JUnit Jupiter ✔
│  └─ Rationale Zahl ✔
│     ├─ multipliziert mit rationaler Zahl ✔
│     ├─ mulitpliziert mit ganzer Zahlen ✔
│     └─ multipliziert mit Kehrwert ✔
├─ JUnit Vintage ✔
└─ JUnit Platform Suite ✔

Test run finished after 180 ms
[         4 containers found      ]
[         0 containers skipped    ]
[         4 containers started    ]
[         0 containers aborted    ]
[         4 containers successful ]
[         0 containers failed     ]
[         3 tests found           ]
[         0 tests skipped         ]
[         3 tests started         ]
[         0 tests aborted         ]
[         3 tests successful      ]
[         0 tests failed          ]
```

über eine geeignete Benennung der Testmethoden, bzw. mit @DisplayName kann Einfluss auf die Lesbarkeit der Protokolle genommen werden

- Integration in IDE (z.B. Eclipse)



Errors: Unerwartete Ausnahmen
Failures: Abweichung beim Soll-/Ist-Vergleich (`assert`)

- Aufruf über Build-Tool mit Integration in das Berichtswesen (siehe später)

- Test-Suite
  - Die Testfälle (mit `@Test` annotierte Methoden) einer Testklasse werden von JUnit automatisch zu einer *Suite* zusammengefasst
  - Alle Testfälle einer *Suite* werden dann nacheinander abgearbeitet (auch wenn Tests fehlschlagen)
  - Es kann auch eine *Suite* von Testklassen gebildet werden, um alle Testfälle in den einzelnen Klassen ablaufen zu lassen

```
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectClasses({RationalTest.class, QueueTest.class})
public class AllTests {

}
```

- Parametrisierte Testmethoden
  - Eine Testmethode kann mehrfach mit unterschiedlichen Parametern aufgerufen werden
  - Eine Testmethode kann damit nacheinander verschiedene Testfälle realisieren
  - Die Testklasse stellt die Testfalldaten in einem (funktionalen) Stream zur Verfügung. Sie erhält dafür eine statische Methode als Datenquelle
  - Die Testfalldaten enthalten den Soll-Wert und die Eingabewerte
  - Ein spezieller Runner ruft dann für alle Testfalldaten aus dem Stream nacheinander die Test-Methode auf. Die Testmethode ist mit dem Namen der Datenquelle annotiert
  - Für jeden Testfall aus dem Stream wird eine neue Instanz der Testklasse erzeugt
  - Der Runner übergibt die Testfalldaten der Testmethode als Argumente
  - Ggf. ist ein spezieller `SimpleArgumentConverter` zu implementieren, um eine Typkonvertierung zwischen den in `Arguments` enthaltenen Typen und den Parametertypen der Testmethode vorzunehmen

```
public class CalculatorParameterizedTest {

   public static Stream<Arguments> daten(){
      Arguments[] testdaten = {Arguments.of(0, 2, 3),
                               Arguments.of(2, 4, 2),
                               Arguments.of(3, 7, 2),
                               Arguments.of(5, 10, 2)};
      return Arrays.asList(testdaten).stream();
   }

   @ParameterizedTest
   @MethodSource({"daten"})
   public void testDivisions(int soll, int zaehler, int nenner) {
      Calculator cal = new Calculator();
      assertEquals(soll, cal.divide(zaehler, nenner));
   }
}
```

Datenquelle

Daten für einen Testfall

Methodenname der Datenquelle

- Die Testfalldaten können auch aus einer csv-Datei eingelesen werden

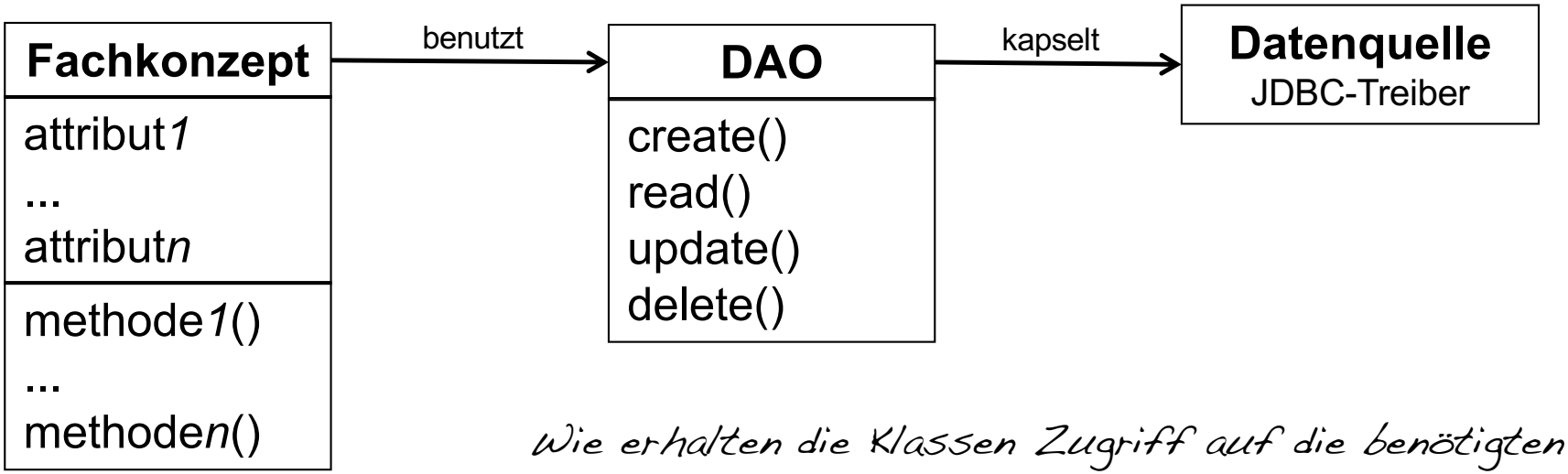
```
public class CalculatorParameterizedTest {

    @ParameterizedTest
    @CsvFileSource(resources= {"/div.csv"}, numLinesToSkip = 1)
    public void testDivisions(int soll, int zaehler, int nenner) {
        Calculator cal = new Calculator();
        assertEquals(soll, cal.divide(zaehler, nenner));
    }
}
```

Kopfzeile überlesen

```
Soll, Zaehler, Nenner
0, 2, 3
2, 4, 2
3, 7, 2
5,10, 2
```

div.csv

- Testbarkeit
  - Häufig ist es selbst bei einem Unit-Test nicht möglich, eine zu testende Klasse völlig isoliert zu betrachten
  - Beispiel: Es soll eine Fachkonzeptklasse getestet werden
    a. die verwendete DAO-Klasse ist noch nicht fertiggestellt
    b. das DAO soll nicht die produktive Datenbank verwenden, sondern die Datenbank in der Entwicklungsumgebung



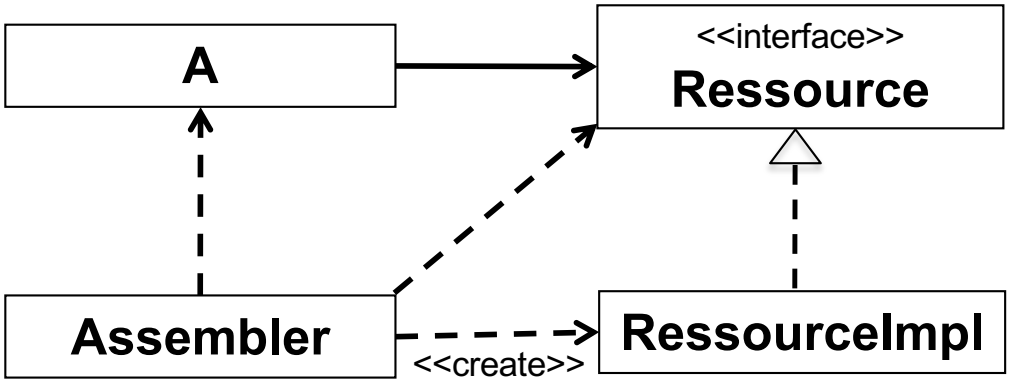
Wie erhalten die Klassen Zugriff auf die benötigten Ressourcen?

- Es gibt im Wesentlichen drei Möglichkeiten, wie ein Objekt *A* eine Referenz auf ein benötigtes Objekt *B* erhält
  1. A erzeugt selber die Instanz B
  2. Dependency Injection
  3. Verwendung eines Namensdienstes (Service Locator)

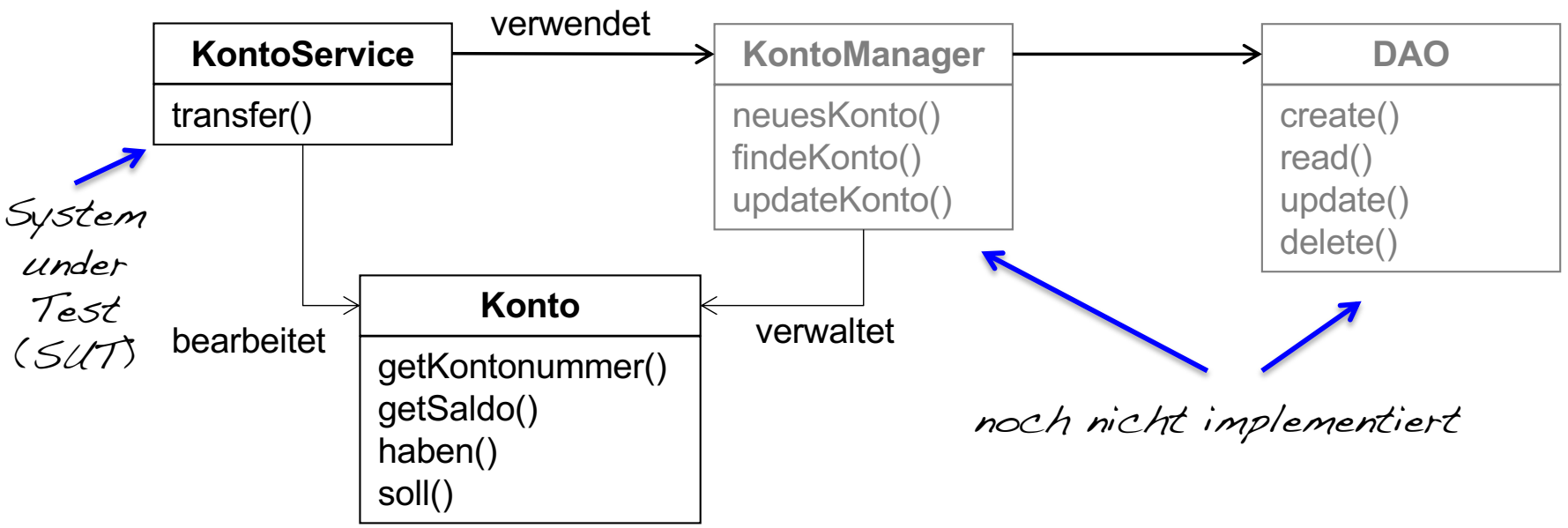
Einfluss auf die Testbarkeit?

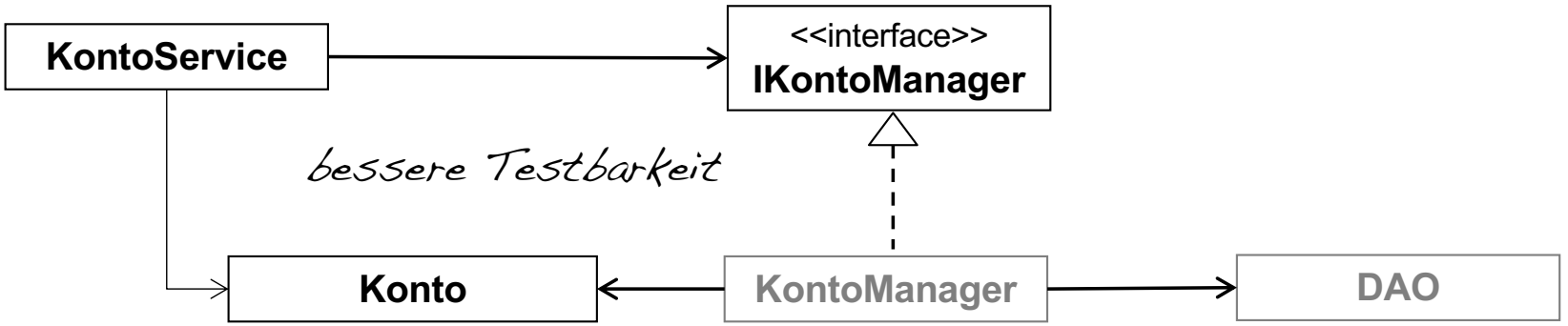
- Die Testbarkeit wird deutlich verbessert, wenn in einer zu testenden Klasse Assoziationen zu anderen Objekten nicht aktiv aufgebaut werden

- Dependency Injection (DI)
  - Eine höhere Flexibilität bezüglich der Umwelt wird erreicht, wenn Assoziationen zu anderen Objekten von außen injiziert werden
  - Ein Objekt kümmert sich also nicht mehr aktiv um eine Ressource, sondern bekommt die entsprechende Referenz von außen übergeben (von einem "Assembler")

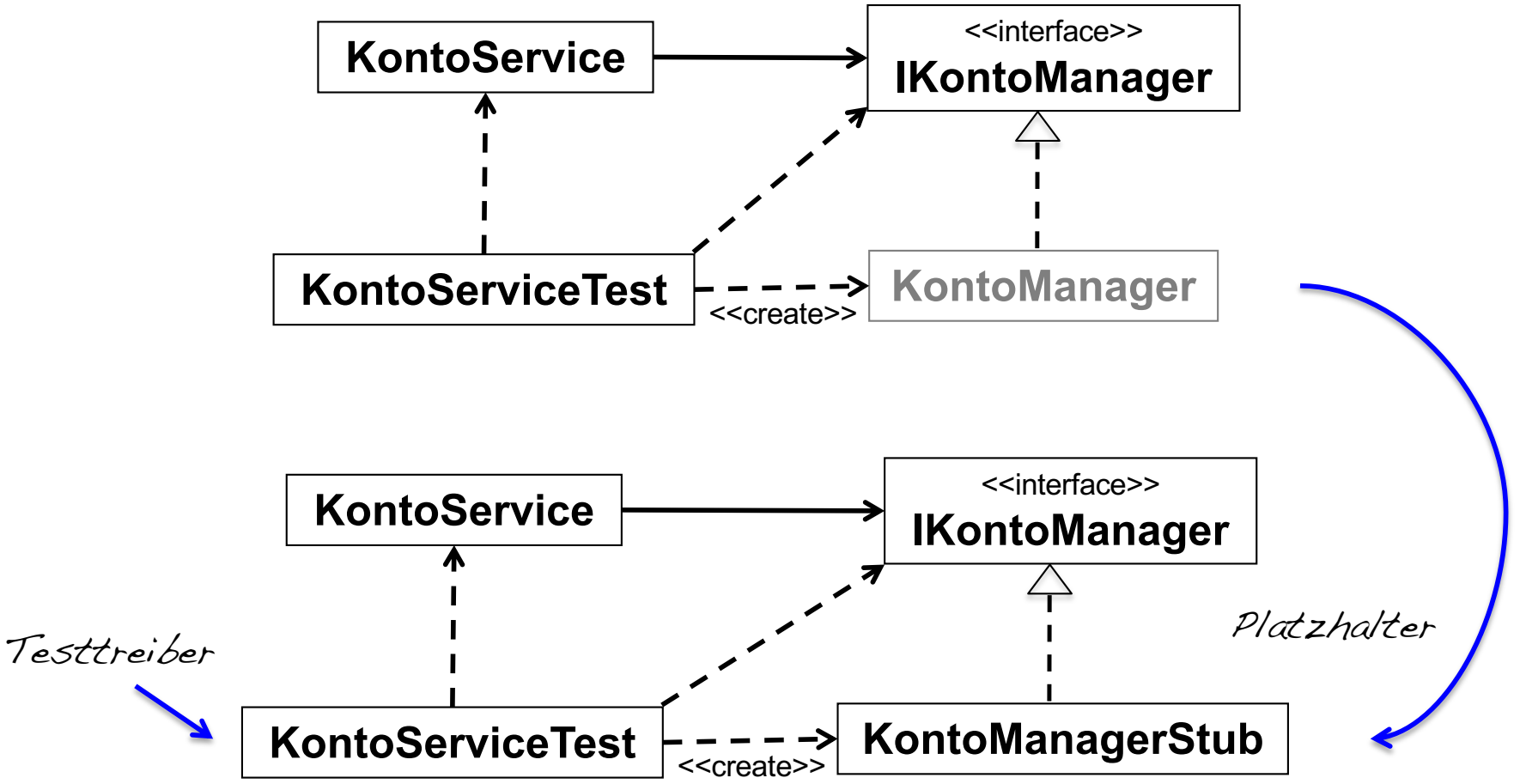


Fallbeispiel: Kontenverwaltung





– Einsatz von DI (am Beispiel eines JUnit-Tests)



- Die „Nebenklasse“ Konto (indirekter Test)

```
public class Konto {

    private String kontoNummer;
    private double saldo;

    public Konto(String kontoID, double saldo){
        this.kontoNummer = kontoID;
        this.saldo = saldo;
    }
    public void soll(double umsatz){
        saldo -= umsatz;
    }
    public void haben(double umsatz){
        saldo += umsatz;
    }
    public String getKontoNummer() {
        return kontoNummer;
    }
    public double getSaldo() {
        return saldo;
    }
}
```

- Die Schnittstelle `IKontoManager`

```
public interface IKontoManager {
    Konto findeKonto(String kontoNummer);
    void updateKonto(Konto konto);
    void neuesKonto(Konto konto);
}
```

- Implementierung der Klasse `KontoService` ist zu testen

```
public class KontoService {

   private IKontoManager kontoManager;

   public void setKontoManager(IKontoManager kontoManager){
      this.kontoManager = kontoManager;
   }

   public void transfer(String quelle, String ziel, double umsatz){
      Konto quelleKonto = kontoManager.findeKonto(quelle);
      Konto zielKonto = kontoManager.findeKonto(ziel);

      quelleKonto.soll(umsatz);
      zielKonto.haben(umsatz);

      kontoManager.updateKonto(quelleKonto);
      kontoManager.updateKonto(zielKonto);
    }
}
```

Programmierung gegen Interface

Instanz von KontoManager wird nicht selber erzeugt

- Platzhalter (Stub) für die Klasse `KontoManager`

```
public class KontoManagerStub implements IKontoManager{

   private Konto konto1;
   private Konto konto2;

   public void neuesKonto(Konto konto){
      if (konto1==null) konto1 = konto;
      konto2 = konto;
   }

   public void updateKonto(Konto konto){
   }

   public Konto findeKonto(String kontoNummer){
      if (konto1.getKontoNummer().equals(kontoNummer)) return konto1;
      if (konto2.getKontoNummer().equals(kontoNummer)) return konto2;
      throw new IllegalArgumentException();
   }
}
```

Rudimentäre Implementierung:

Es können nur zwei Konten „verwaltet" werden

- Testtreiber für die Klasse `KontoService`

```
public class KontoServiceTest {

   @Test
   public void testTransfer() {

      Konto quelle = new Konto("1", 1000.0);
      Konto ziel = new Konto("2", 100.0);
      IKontoManager stub = new KontoManagerStub();
      stub.neuesKonto(quelle);
      stub.neuesKonto(ziel);

      KontoService service = new KontoService();
      service.setKontoManager(stub);

      service.transfer("1", "2", 500.0);

      assertEquals(500.0, quelle.getSaldo(),0.0);
      assertEquals(600.0, ziel.getSaldo(), 0.0);
   }
}
```

Erzeugung und Konfiguration des Platzhalters

DI des Platzhalters

Aufruf der zu testenden Methode

Zustand überprüfen

Stubs vs Mocks

- Testen mit *Stubs*
  - Mit *Stubs* werden die zu testenden Einheiten von der Umgebung entkoppelt
  - *Stub* simuliert für den Testfall erforderliche Fachlogik
  - Zustandstest (Test der Zustandsänderung)

- Testen mit Mocks
  - Mocks sorgen ebenfalls für eine Entkopplung
  - Mocks kontrollieren die Anzahl der Methodenaufrufe
  - Mocks kontrollieren die Abfolge der Methodenaufrufe
  - Mocks enthalten keine/begrenzte Simulation der Fachlogik
  - Verhaltenstest

- Testen mit Mocks
  - Da Mocks keine Fachlogik enthalten, können sie aus einer Schnittstellendefinition automatisch generiert werden
  - Dies erfordert die Unterstützung durch entsprechende Werkzeuge /Frameworks (z.B. *EasyMock*)
  - Damit kann der Aufwand bei der Testimplementierung reduziert werden

- *EasyMock*-Framework
  - `easymock.jar` wird in den `classpath` der Anwendung aufgenommen
  - Aus einem Interface wird im Testtreiber mit der Methode `createMock` zur Testlaufzeit ein Mock-Objekt generiert
  - Das Mock-Objekt befindet sich dann automatisch in einer Aufzeichnungsphase

- Es werden alle, während des eigentlichen Tests erwarteten, Methodenaufrufe durchgespielt und vom Mock-Objekt aufgezeichnet
- Danach wird das Mock-Objekt in einen *Replay*-Modus versetzt
- Der eigentliche Testfall wird gestartet
- In einem *Verify*-Modus vergleicht das Mock-Objekt dann die aufgezeichneten Methodenaufrufe mit den tatsächlich im Test erfolgten Methodenaufrufen
- Abweichungen führen zum Scheitern des JUnit-Tests

```
public class KontoServiceEasyMockTest {
   private IKontoManager mock;
   @BeforeEach
   public void setUp(){
      mock = mock("mock", IKontoManager.class);
   }
   @Test
   public void test() {
      Konto quelle = new Konto("1", 1000.0);
      Konto ziel = new Konto("2", 100.0);
      expect(mock.findeKonto("1")).andReturn(quelle);
      expect(mock.findeKonto("2")).andReturn(ziel);
      mock.updateKonto(quelle);
      mock.updateKonto(ziel);
      replay(mock);
      KontoService service = new KontoService();
      service.setKontoManager(mock);
      service.transfer("1", "2", 500.0);
   }
   @AfterEach
   public void tearDown(){
      verify(mock);
   }
}
```

generiert Mock-Objekt aus Interface

Aufzeichnung des erwarteten Verhaltens

Aufruf der zu testenden Methode

Prüfen, ob erwartete Aufrufe nicht erfolgt sind

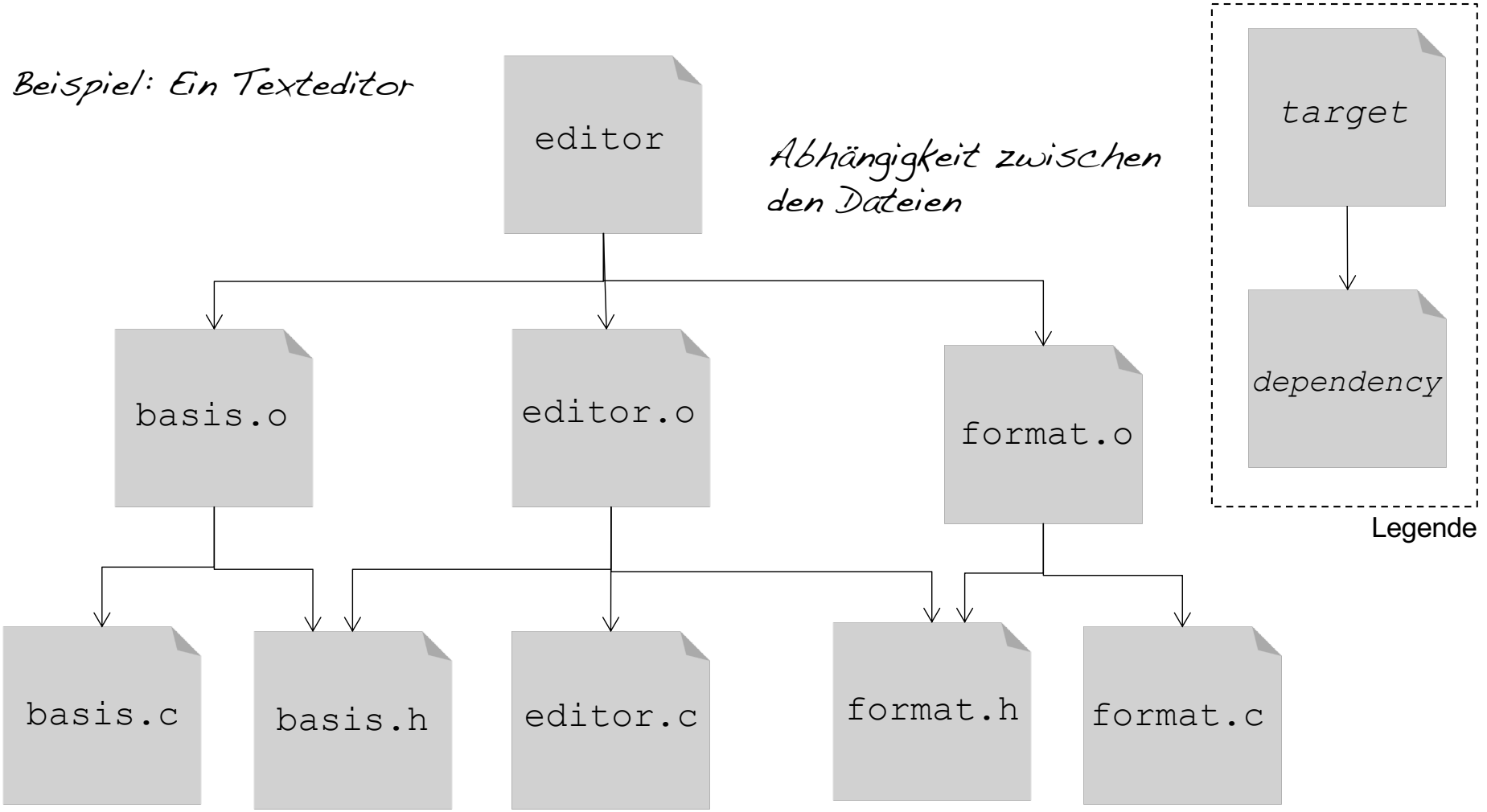
- Der Verhaltenstest mit einem Mock kann im obigen Fall noch um einen Zustandstest angereichert werden

```
//...
service.transfer("1", "2", 500.0);

assertEquals(500.0, quelle.getSaldo(),0.0);
assertEquals(600.0, ziel.getSaldo(), 0.0);
```

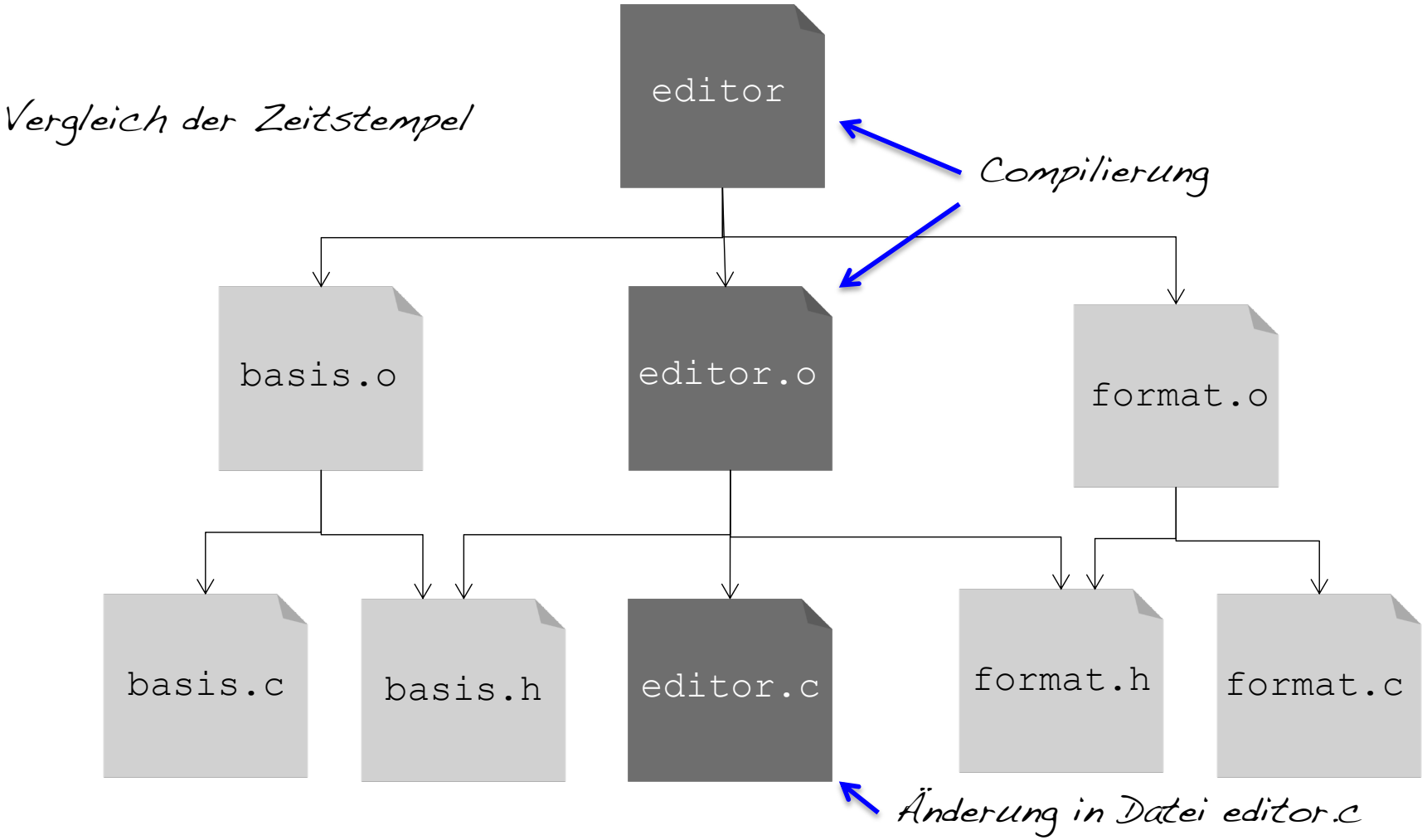
Build-Automatisierung

- Die (allgemeine) Automatisierung haben wir bereits als konstruktive Maßnahme der Qualitätssicherung identifiziert
- Wir betrachten hier als wichtigen Spezialfall die Automatisierung des Build-Prozesses
- Obwohl eine IDE in der Regel einen grundlegenden Build-Prozess bietet, sollte ein unabhängiger Build-Prozess aufgesetzt werden
- Dazu wird ein geeignetes Build-Tool benötigt:
  - make (C, C++, ...)
  - Ant (Java)
  - Maven (Java)
  - ...

– Neben der Automatisierung des Build-Prozesses bieten die Werkzeuge auch eine inkrementelle (bedarfsgerechte) Compilierung



– Bei einer nachträglichen Änderung in der Datei `editor.c` müssen die Dateien `basis.o` und `format.o` nicht neu übersetzt werden



- make
  - Bedingungsgesteuerte Ausführung von Shell-Skript-Kommandos
  - `make` steuert u.a. die folgenden Aktivitäten
    - Kopieren von Dateien
    - Inkrementelle Compilierung
    - Linken
  - Der Erstellungsprozess wird formal in einem *Makefile* beschrieben
  - Im *Makefile* sind alle Abhängigkeiten beschrieben und alle Aktivitäten definiert
  - Ein *Makefile* besteht aus Regeln der folgenden Form

```
Ziel : Voraussetzung ... ...
    Befehl
    ...
    ...
```

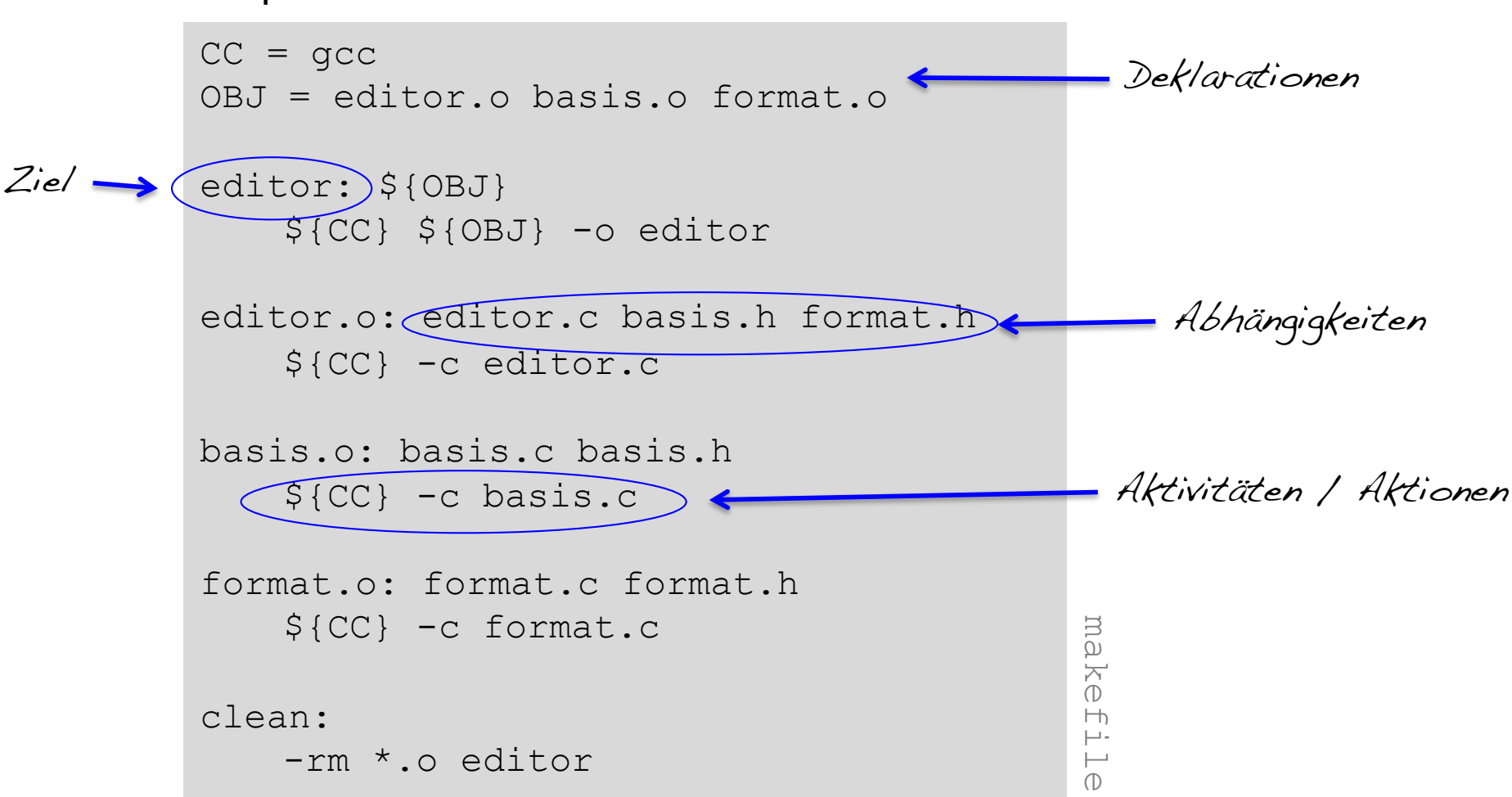
makefile

Jede Befehlszeile muss mit einem Tabulatorzeichen beginnen

- Beispiel: Editor

```
CC = gcc
OBJ = editor.o basis.o format.o

editor: ${OBJ}
    ${CC} ${OBJ} -o editor

editor.o: editor.c basis.h format.h
    ${CC} -c editor.c

basis.o: basis.c basis.h
    ${CC} -c basis.c

format.o: format.c format.h
    ${CC} -c format.c

clean:
    -rm *.o editor
```

makefile



- Mit `make` wird auf der Kommandozeilenebene die Verarbeitung der Datei `makefile` gestartet

- Ant
  - Da `make` die Ausführung von beliebigen Shell-Skript-Kommandos erlaubt, ist `make` plattformabhängig
  - Die Alternative *Ant* kommt im Java-Umfeld zum Einsatz und bietet eine angemessene Plattformunabhängigkeit
  - Die Konfigurationsdatei `build.xml` ähnelt dem `makefile`, wird aber im standardisierten XML-Format beschrieben
  - In der `build.xml` wird beschrieben, wie Ziele (*targets*), unter der Berücksichtigung von Abhängigkeiten (*dependencies*), durch die Ausführung von Kommandos erreicht werden können
  - Grundstruktur

```
<project name="..." default="..." basedir=".">
   <target name="..." depends="...">
      <command />
   </target>
</project>
```

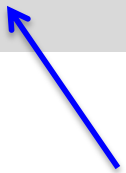
build.xml

- Attribute des Elements `project`:

Attribut	Bedeutung
`name`	Name des Projekts
`default`	Wird ausgeführt wenn Ant ohne Target aufgerufen wird
`basedir`	Basisverzeichnis für alle Targets

- Die Teilschritte eines Build-Prozesses werden als *Targets* bezeichnet
- Ein *Target* besitzt einen Namen und evtl. Abhängigkeiten zu anderen Targets

```
<target name="A" depends="B, C">
      <command />
</target>
```



Bevor Target A ausgeführt wird, werden zunächst die Targets B und C abgearbeitet

- Ein *Target* kann eine Folge von Kommandos (*Tasks*) enthalten
- *Ant* enthält über 100 *Tasks*
  - javac
  - mkdir
  - copy
  - junit
  - jar
  - cvs
  - ...

1. Einstellungen

- Definition von Variablen mit dem Target `property`
  - Einstellungen können zentral vorgenommen werden

```
<property name="key" value="wert" />
```

  - Die Zeichenfolge `${key}` wird dann in den `wert` aufgelöst

- Zugriffspfade auf externe Klassenbibliotheken setzen

```
<path id="project.classpath">
   <pathelement path="./${lib.dir}/log4j.jar" />
</path>
```

2. Vorbereiten der Verzeichnisse

- Verzeichnisse anlegen

```
<target name="prepare">
   <echo message="creating: ${build.dir}" />
   <mkdir dir="${build.dir}" />
   <echo message="creating: ${release.dir}" />
   <mkdir dir="${release.dir}" />
</target>
```

- Alte Dateien löschen

```
<target name="delete">
   <delete dir="${build.dir}" />
</target>
```

3. Übersetzen

- Compilieren von Klassen und Testklassen

```
<target name="compile" depends="prepare">
   <javac srcdir="${source.dir}" destdir="${build.dir}">
     <classpath>
        <path refid="project.classpath" />
     </classpath>
   </javac>

   <javac srcdir="${test.dir}" destdir="${build.dir}">
      <classpath>
         <path refid="project.classpath" />
         <path refid="test.classpath" />
      </classpath>
   </javac>
</target>
```

4. Testen

- JUnit 5-Launcher konfigurieren

```
<target name="test.junit.launcher" depends="compile">
   <junitlauncher haltOnFailure="true" printSummary="true">
      <classpath>
         <path refid="project.classpath" />
         <path refid="test.classpath"/>
      </classpath>
      <testclasses outputdir="${basedir}/${report.dir}/junit">
         <fileset dir="${build.dir}">
            <include name="**/*Test.class" />
         </fileset>
         <listener type="legacy-xml" sendSysOut="true“
                   sendSysErr="true" />
         <listener type="legacy-plain" sendSysOut ="true" />
      </testclasses>
   </junitlauncher>
</target>
```

- Test über Console-Launcher starten und Reporting

```
<target name="test.console.launcher" depends="compile">
   <java classname="org.junit.platform.console.ConsoleLauncher"
         fork="true" failonerror="true">
      <classpath>
         <path refid="project.classpath" />
         <path refid="test.classpath" />
      </classpath>
      <arg value="--scan-classpath" />
      <arg value="--reports-dir ${report.dir}/junit"/>
   </java>
   <junitreport todir="${report.dir}/junit">
      <fileset dir="${report.dir}/junit">
         <include name="TEST-*.xml" />
      </fileset>
      <report format="frames" todir="${report.dir}/junit"/>
   </junitreport>
</target>
```

- Beispiel für ein Testprotokoll

Home

Packages

de.swtd

de.swtd

Classes

BasisTest

Unit Test Results.

Designed for use with JUnit and Ant.

Class de.swtd.BasisTest

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
BasisTest	2	0	1	0	0.058	2013-12-14T14:45:17	localhost

Tests

Name	Status	Type	Time(s)
testGetter	Success		0.002
testAdd	Failure	expected:<2> but was:<-2> junit.framework.AssertionFailedError: expected:<2> but was:<-2> at de.swtd.BasisTest.testAdd(BasisTest.java:18) at org.eclipse.ant.internal.launching.remote.EclipseDefaultExecutor.executeTargets(EclipseDefaultExecutor.java:32) at org.eclipse.ant.internal.launching.remote.InternalAntRunner.run(InternalAntRunner.java:424) at org.eclipse.ant.internal.launching.remote.InternalAntRunner.main(InternalAntRunner.java:138)	0.010

Properties »

- Testaktivitäten

```
<target name="test" depends="test.junit.launcher, test.console.launcher" />
```

  - Abbrechen des Build-Prozesses bei Fehlern im Unit-Test

```
<fail if="tests.failed" >
</fail>
```

Task ist Ergänzung für Target „test"

4. Ausliefern

- Erzeugen einer `jar`-Datei

```
<target name="deploy" depends="test, compile">
   <jar destfile="${release.dir}/${appname}.jar">
      <fileset dir="${build.dir}" excludes="**/*Test.class" />
         <manifest>
            <attribute name="Main-Class" value="${mainclass}" />
         </manifest>
   </jar>
</target>
```

- Maven
  - Ant verfolgt einen imperativen Ansatz: Vorgabe der Arbeitsschritte
    - Ermöglicht sehr individuelle Prozesse
  - Maven verfolgt einen deklarativen Ansatz: Beschreibung der Ziele
    - Erzwingt eine Standardisierung

  - Merkmale
    - Ansatz: *Convention over Configuration*
    - Standardisierte Verzeichnisstruktur
    - Standardisierter Bearbeitungszyklus (*Life Cycle*)
    - Neue Projekte können über Projektvorlagen (*Archetypes*) angelegt werden
    - Management von Abhängigkeiten
    - Erweiterungen über *Plug-ins*

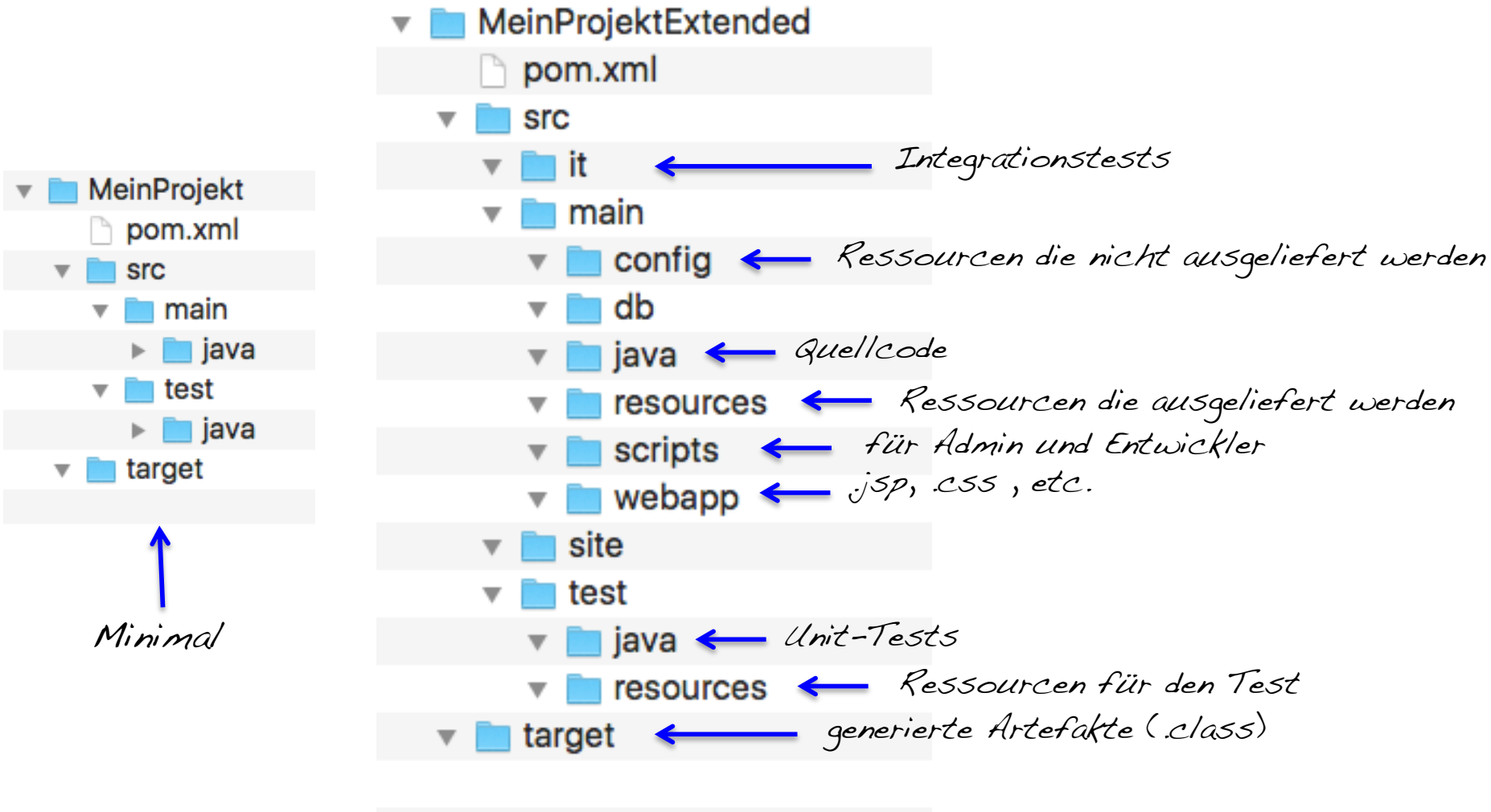
- Der Build-Prozess eines Projekts wird in der Datei `pom.xml` (pom : Project Object Model) konfiguriert

```
<project xmlns=http://maven.apache.org/POM/4.0.0
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>de.fhdortmund</groupId>
   <artifactId>Utility</artifactId>
   <packaging>jar</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>Utility</name>
   <url>http://www.fh-dortmund.de</url>
   <developers>
      <developer>
         <id>dwiesmann</id>
         <name>Dirk Wiesmann</name>
         <email>dirk.wiesmann@fh-dortmund.de</email>
         <properties><active>true</active></properties>
      </developer>
   </developers>
</project>
```

Ausgangsbasis

pom.xml

- Maven-Verzeichnisstruktur

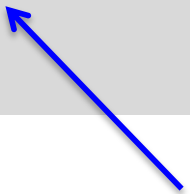


- Maven: Management von Abhängigkeiten
  - In einem Softwareprojekt bestehen in der Regel Abhängigkeiten zu weiteren (externen) Artefakten
  - Zum Beispiel zu Programmbibliotheken: `junit-jupiter-5.9.1.jar`, `log4j-api-2.19.0.jar`
  - Die Abhängigkeiten werden in Maven über die folgenden Attribute aufgelöst

Attribut	**Bedeutung**	**Beispiel**
`groupId`	Organisation, die für das Projekt/Produkt zuständig ist	`de.fh.dortmund`
`artifactId`	Identifizierung des generierten Artefakts	`myProg`
`version`	Versionsnummer	`1.0.0-SNAPSHOT`
`type`	Typ / Art der Paketierung	`WAR`

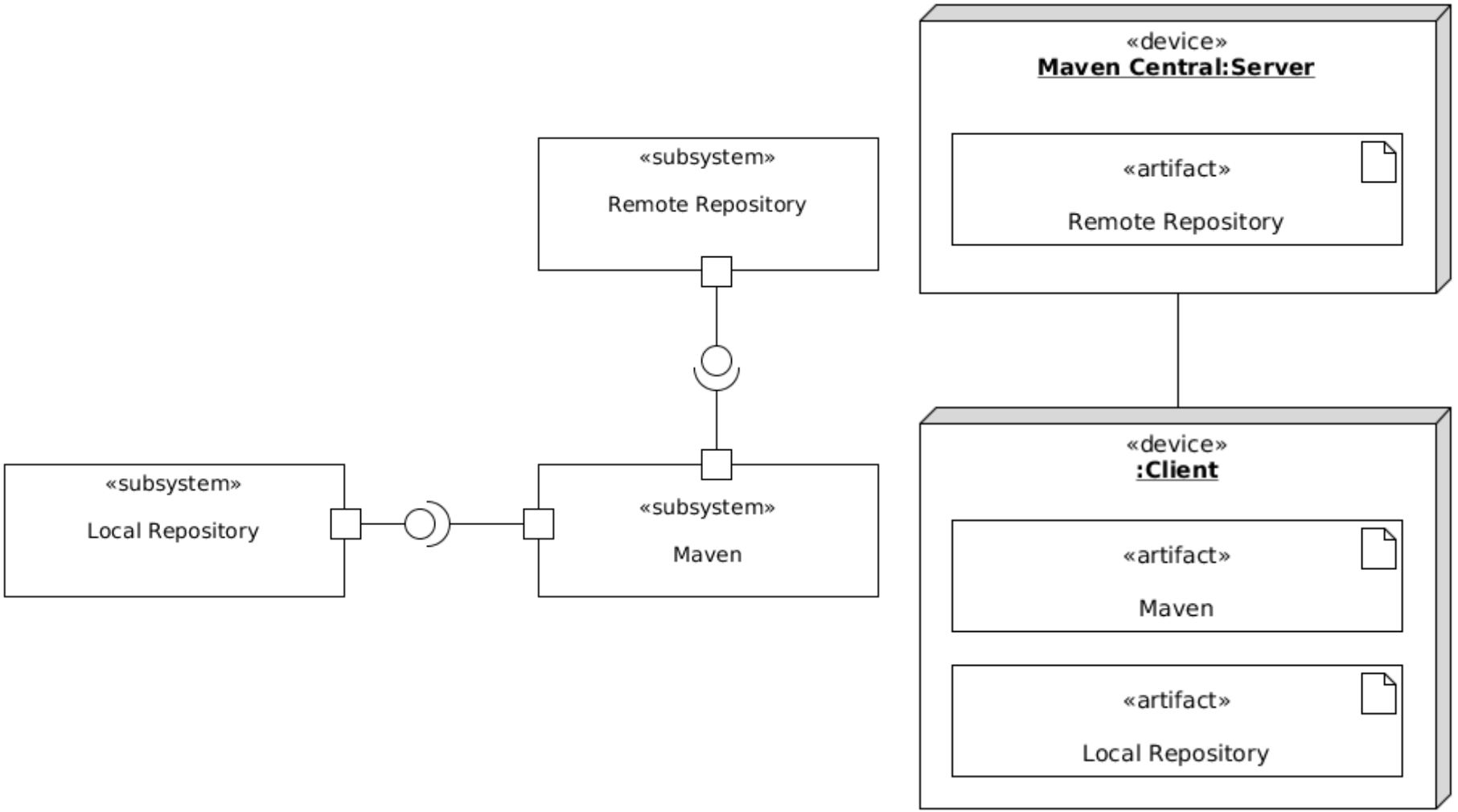
- Alle Abhängigkeiten werden unter dem Element `<dependencies>` aufgelistet
- Für jede Abhängigkeit wird ein `<dependency>`-Eintrag aufgenommen

```
<dependencies>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.9.1</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```



Zeitraum in der die Abhängigkeit zur Verfügung (im Klassenpfad) steht, z.B.
compile : immer (default)
test: während der Testaktivitäten

- Die erforderlichen Artefakte werden automatisch von einem zentralen Repository geladen und in einem lokalen Repository abgelegt



- Indirekte Abhängigkeiten werden automatisch aufgelöst

```
> mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Utility 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ Utility ---
[INFO] de.fhdortmund:Utility:jar:1.0-SNAPSHOT
[INFO] +- junit:junit:jar:4.11:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] \- log4j:log4j:jar:1.2.17:compile
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.808 s
[INFO] Finished at: 2015-11-19T13:32:03+01:00
[INFO] Final Memory: 12M/60M
[INFO] ------------------------------------------------------------------------
```

Anzeige der Abhängigkeiten in einem Projekt

- Der Einsatz eines eigenen Repository Managers ist möglich (z.B. Nexus)
  - Der eigene Repository Manager unterbindet den direkten Zugriff von Maven auf allgemeine Repositories im Internet (z.B. Maven Central)

- Maven: *Life Cycle* und Phasen
  - Standardmäßig werden drei verschiedene *Life Cycle* unterschieden
    a) `Default`: Übersetzen, Paketieren und Verteilen eines Projekts
    b) `Clean`: Löschen von temporären und generierten Artefakten
    c) `Site`: Erzeugung von Dokumentation
  - Jeder *Life Cycle* besteht aus mehreren Schritten, die als Phasen bezeichnet werden
  - Eine Phase kann gezielt ausgeführt werde
    - Alle Vorgängerphasen werden damit ebenfalls ausgeführt

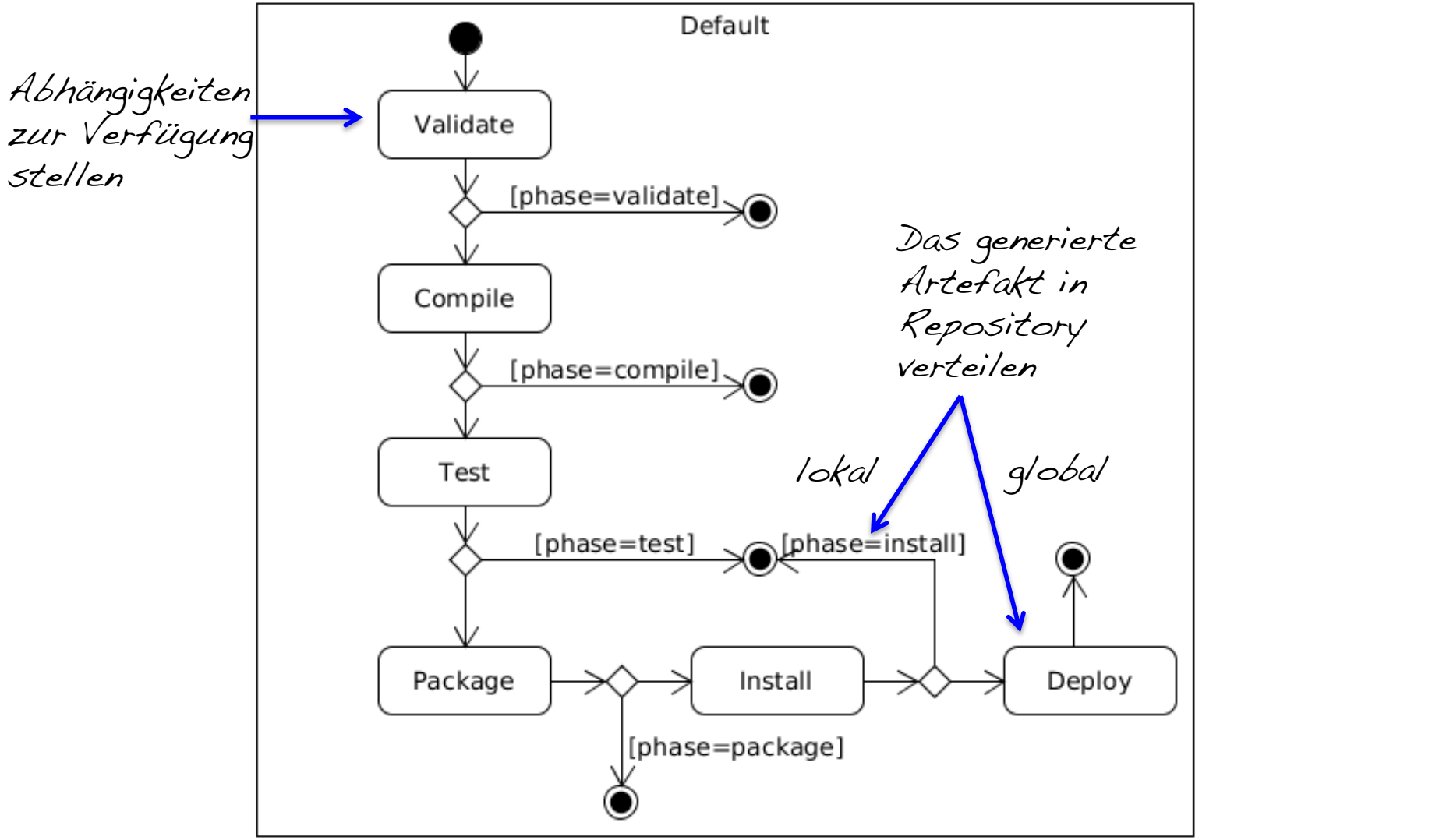
```
> mvn test
```

    - Eine Kombination von Phasen aus unterschiedlichen *Life Cycle* ist möglich

```
> mvn clean test
```

- Der `Default`-Life-Cycle

nur die zentralen Phasen sind dargestellt



- In jeder Phase werden spezielle Plug-Ins aufgerufen
  - In der Compile-Phase wird z.B. das Maven-Standard-Plug-In `Compiler` aufgerufen
- In den Plug-Ins werden verschiedene Ziele (*goals*) realisiert
  - Das Plug-In Compiler hat die beiden Ziele (*goals*) `compile` und `testCompile`
- Ein spezielles Ziel kann über die Notation *`Plug-In`*`:`*`Goal`* aufgerufen werden
- Beispiele:

```
> mvn compiler:compile
```

Die Quellcode-Dateien werden übersetzt (ohne Testcode)

```
> mvn compiler:testCompile
```

Nur der Testcode (Unit-Test) wird übersetzt

- Der Clean-Life-Cycle
  - In der zentralen Phase `clean` wird das Maven-Standard-Plugin `Clean` aufgerufen
  - Das `Clean`-Plug-In besitzt nur das Ziel `clean`
  - Dieses Ziel löscht alle temporären und automatisch generierten Dateien

```
> mvn clean
```

Phase

Standard Verbindung (lifecycle binding): Phase und Goal

```
> mvn clean:clean
```

Plug-In und Goal

- Der `Site`-Life-Cycle
  - In der zentralen Phase `site` wird eine Projektdokumentation erstellt.
  - Standardmäßig werden im Ordner `target/site` Web-Seiten angelegt, die Informationen aus der `pom.xml` aufbereiten (z.B. Abhängigkeiten und Plug-Ins)

einige Built-in Lifecycle Bindings

Phase	**Plug-In:Goal**
`clean`	`clean:clean`
`compile`	`compiler:compile`
`test`	`surefire:test`
`test-compile`	`compiler:testCompile`
`package`	`ejb:ejb, oder jar:jar, oder war:war`
`site`	`site:site`

- Maven: Projektvorlagen (*Archetypes*)
  - Die Verzeichnisstruktur und die `pom.xml` muss nicht manuell erstellt werden
  - Maven bietet eine Reihe von Projektvorlagen (*Archetypes*), aus denen die Grundstrukturen automatisch generiert werden können
  - Eine Katalog von Artefakten kann über das Plug-In `archetype` angezeigt werden

```
> mvn archetype:generate
```

    - Eine Vorlage kann dann interaktiv ausgewählt werden
    - Die Einrichtung des Projektes erfolgt über einen Dialog

  - Eine Vorlage kann auch direkt gewählt werden

```
> mvn archetype:generate
      -DarchetypeArtifactId=maven-archetype-quickstart
```

Basisstruktur

- Maven: Grundeinstellungen
  - Im Build-Bereich der `pom.xml` sind die folgenden Plugins einzutragen

```
<build>
   <plugins>
     …
   </plugins>
</build>
```

  - Aufgrund der schnell steigenden Anzahl an unterschiedlichen Java-Versionen ist es häufig notwendig, eine spezifische Version vorzugeben

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.8.1</version>
   <configuration>
      <source>14</source>
      <target>14</target>
   </configuration>
</plugin>
```

- Neben der `junit-jupiter`-Dependency wird von JUnit 5 eine aktuelle Version vom surefire-Plugin für die Ausführung der Unit-Tests benötigt

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <version>3.0.0-M5</version>
</plugin>
```

- Maven: Reports für die Qualitätssicherung
  - Unter dem Element `<reporting>` können Plug-Ins `<plugins>` für die Berichterstellung eingebunden werden

```
<reporting>
  <plugins>
  </plugins>
</reporting>
```

    - Erstellung einer HTML-Seite mit einer Verknüpfung zu den *Reports*

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-site-plugin</artifactId>
   <version>3.12.1</version>
</plugin>
```

    - Implementierungsdokumentation/Dokumentationsextraktion: Javadoc

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-javadoc-plugin</artifactId>
   <version>3.4.1</version>
</plugin>
```

- Unit-Test-Reports

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-report-plugin
   </artifactId>
   <version>3.0.0-M5</version>
</plugin>
```

pom.xml

surefire-report:report

-> erstellt ein Bericht der Testergebnisse als HTML-Seite

- Project-Info (Projektdaten)

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>
      maven-project-info-reports-plugin
   </artifactId>
   <version>2.9</version>
</plugin>
```

pom.xml

z.B. project-info-reports:summary
project-info-reports:dependencies

- Aus Kompatibilitätsgründen muss zurzeit das `project-info-reports-plugin` um den folgenden Eintrag ergänzt werden (innerhalb von `<plugin></plugin>`)

```
<dependencies>
   <dependency>
      <groupId>org.apache.maven.shared</groupId>
      <artifactId>maven-shared-jar</artifactId>
      <version>1.2</version>
      <exclusions>
         <exclusion>
            <groupId>com.google.code.findbugs</groupId>
            <artifactId>bcel-findbugs</artifactId>
         </exclusion>
      </exclusions>
   </dependency>
   <dependency>
      <groupId>org.apache.bcel</groupId>
      <artifactId>bcel</artifactId>
      <version>6.2</version>
   </dependency>
</dependencies>
```

Glass-Box-Test (White-Box-Test) / Strukturtest

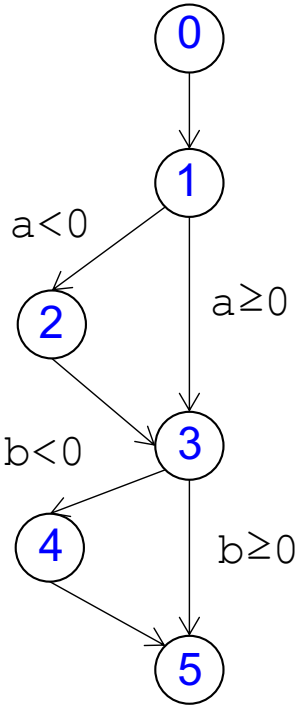
Der Black-Box-Test berücksichtigt nicht die innere Struktur des Programms
Über einen reinen Funktionstest (Black-Box-Test) können wir daher nicht sicherstellen, dass auch alle Teile des Programms getestet werden

Der Programmcode ist sichtbar. Es wird ein Strukturtest vorgenommen. Dabei wird die Überdeckung des Codes durch die Testfälle analysiert. Das Ziel ist es, eine vorgegebene Überdeckungsrate zu erreichen.

- Um die Überdeckung bestimmen zu können, muss das Programm mit zusätzlichen Anweisungen für die Messung instrumentiert werden
- Mit Hilfe der Instrumentierung kann z.B. gezählt werden, wie häufig eine Programmzeile ausgeführt wird
- Es werden verschiedene Überdeckungskriterien unterschieden:
  1. Anweisungsüberdeckung
  2. Zweigüberdeckung
  3. Bedingungsüberdeckung
  4. Datenflussbasierte-Überdeckung

- Kontrollflussgraph
  - Ein Glass-Box-Test orientiert sich am Kontrollflussgraphen einer Code-Sequenz
  - Anweisungen werden im Kontrollflussgraphen als Knoten dargestellt
  - Der Kontrollfluss zwischen den Anweisungen wird über die Kanten repräsentiert

```
0:  public static int manhatten(int a, int b){
1:     if (a < 0)
2:        a = -a;
3:     if (b < 0)
4:        b = -b;
5:     return a+b;
    }
```



Übungsaufgabe

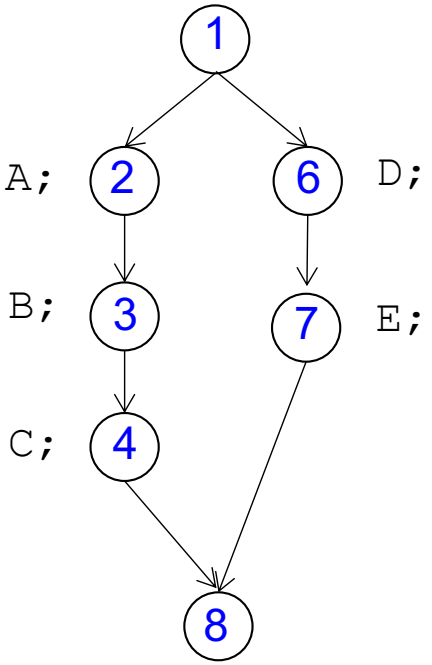
Erstellen Sie die Kontrollflussgraphen für die folgenden elementaren Verzweigungs- und Schleifenkonstrukte

- `if (B) X;`
- `if (B) X; else Y;`
- `while (B) X;`
- `do X; while (B);`

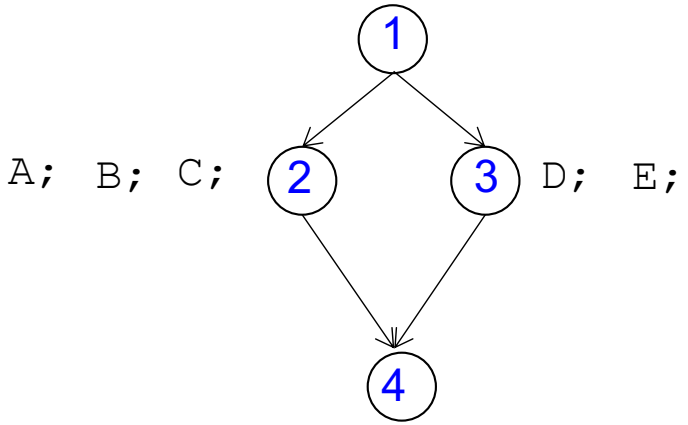
– Klassifikation von Kontrollflussgraphen

```
1:   if (B){
2:      A;
3:      B;
4:      C; }
5:   else  {
6:      D;
7:      E; }
```

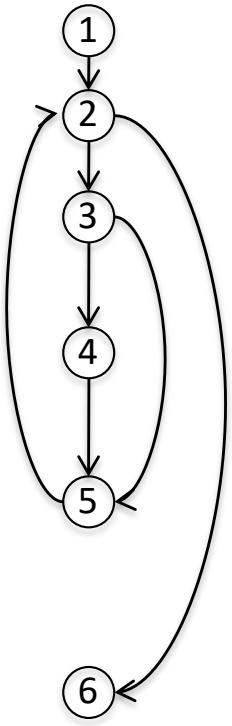
Expandiert



Kolabiert



– Als Beispiel betrachten wir den folgenden Programmausschnitt:



Kontrollflussgraph

```
// eingabe ist vom Typ String
int c=0, v=0;

while((c < eingabe.length()) && (eingabe.charAt(c) >= 'A')
                             && (eingabe.charAt(c) <= 'Z')){
  if((eingabe.charAt(c) == 'A') || (eingabe.charAt(c) == 'E') ||
     (eingabe.charAt(c) == 'I') || (eingabe.charAt(c) == 'O') ||
     (eingabe.charAt(c) == 'U')){
    v++;
  }

  c++;
}


System.out.println("Der Text enthält " + c + " Grossbuchstaben. "
                   + "Davon sind " + v + " Vokale.");
```

1. Anweisungsüberdeckung (*statement coverage*) / C_0-Test
   - Jede Anweisung im Programm muss im Test mindestens einmal ausgeführt werden (100% Anweisungsüberdeckung)
   - Eine Anweisungsüberdeckung > 80 % sollte angestrebt werden

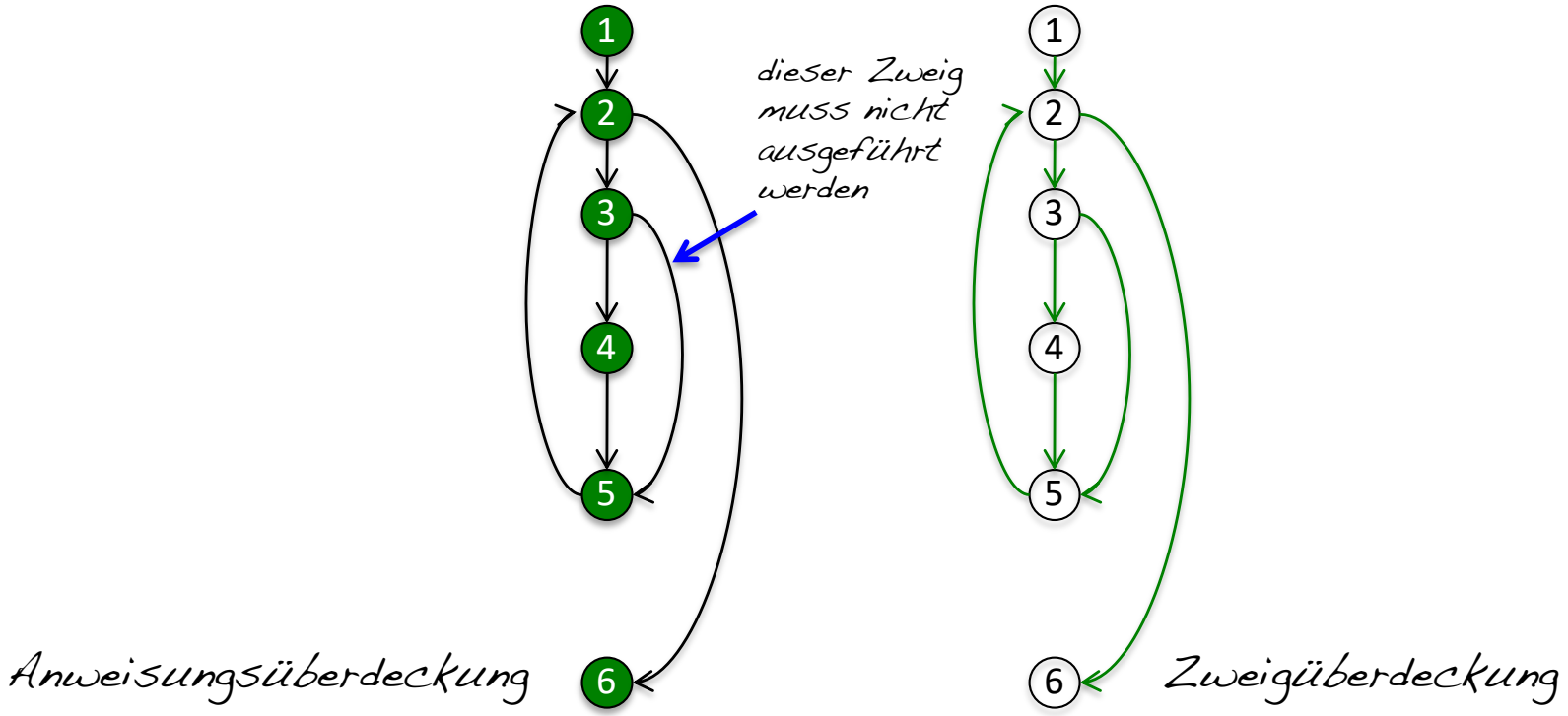
Übungsaufgabe

Kreuzen Sie die Anweisungen an, die für die jeweilige Testeingabe ausgeführt werden (siehe Kontrollflussgraph auf der vorherigen Seite)!

Eingabe	**1**	**2**	**3**	**4**	**5**	**6**
hallo						
FGH						
FOM						

2. Zweigüberdeckung (Entscheidungsüberdeckung) / C_1-Test

- Jeder Zweig im Programm wird im Test mindestens einmal ausgeführt (100% Zweigüberdeckung)
- Die Zweigüberdeckung ist strenger, als die Anweisungsüberdeckung, wenn das Programm leere Zweige enthält (z.B. `if-then` ohne `else`-Zweig)



- Automatisierung mit Maven
  - Codeüberdeckung: Cobertura

```
<plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>cobertura-maven-plugin</artifactId>
   <version>2.7</version>
</plugin>
```

zurzeit Probleme mit neueren Java-Versionen

  - Codeüberdeckung: JaCoCo

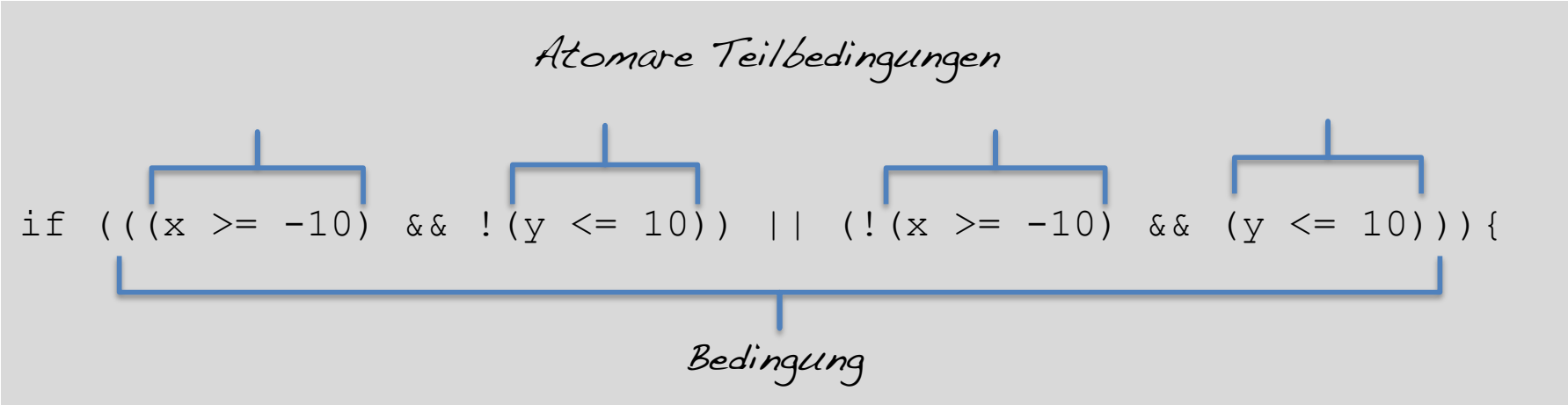
```
<build>
   <plugins>
      <plugin>
         <groupId>org.jacoco</groupId>
         <artifactId>jacoco-maven-plugin</artifactId>
         <version>0.8.6</version>
         <!-- weitere Konfiguration -->
      </plugin>
   </plugins>
</build>
```

siehe Praktikum

goal z.B. jacoco:report

3. Bedingungsüberdeckung

- Die Zweigüberdeckung stellt sicher, dass eine logische Bedingung während des Tests mindestens einmal `true` und einmal `false` ist
- Wenn die Bedingung zusammengesetzt ist, wird damit aber nicht sichergestellt, dass die Teilbedingungen korrekt formuliert sind
- Eine Bedingung ist eine Verknüpfung von Teilbedingungen mit den logischen Operatoren AND, OR und NOT
- Eine (atomare) Teilbedingung enthält keine logischen Operatoren, sondern höchstens Relationssymbole (<,>,==)



- Die Bedingungsüberdeckung bestimmt, wie genau die Bedingungen im Test überprüft werden
- Es werden dabei drei Arten unterschieden:

① **Einfache Bedingungsüberdeckung:** alle atomaren Teilbedingungen müssen während des Tests einmal `true` und einmal `false` ergeben

Testfälle (x=0, y=0), (x=-15, y=15)

② **Mehrfach-Bedingungsüberdeckung:** alle Variationen der Belegung der atomaren Teilbedingungen werden getestet. Achtung: 2^n Variationsmöglichkeiten bei *n* atomaren Bedingungen

Testfälle (x=0, y=15), (x=0, y=0), (x=-15, y=15), (x=-15, y=0)

③ **Minimale-Mehrfach-Bedingungsüberdeckung:** jede Teilbedingung muss mindestens einmal `true` und einmal `false` sein

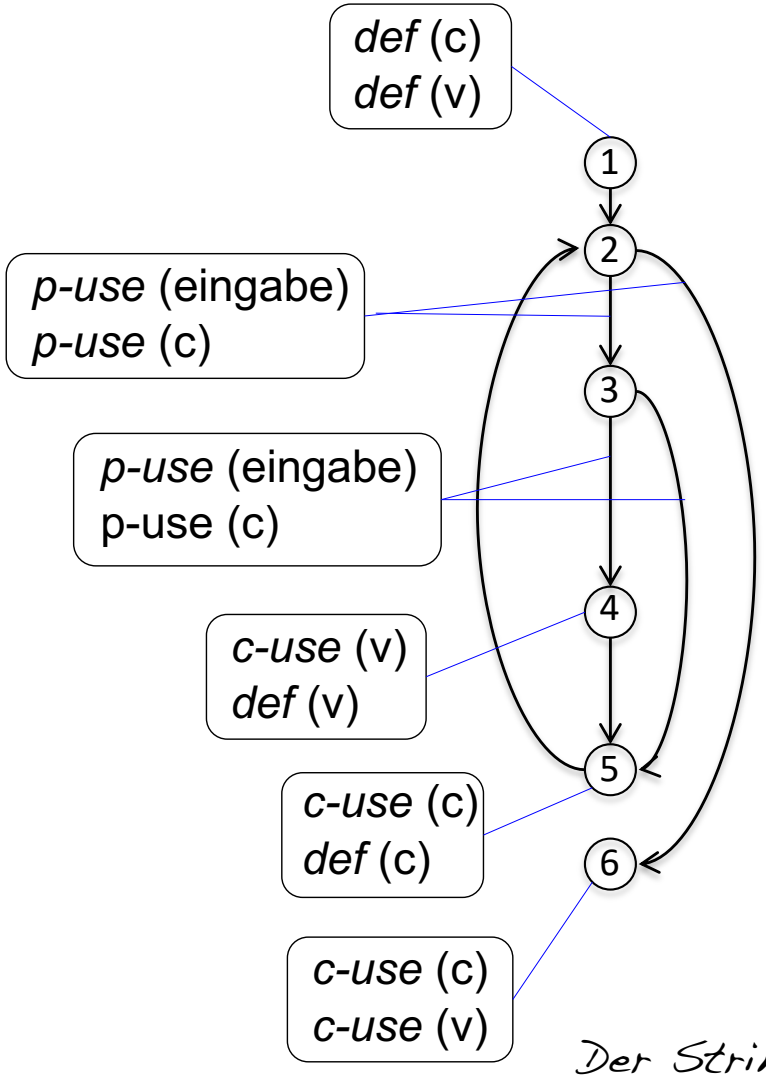
Testfälle (x=-15, y=15), (x=-15, y=0), (x=0, y=15)

- Eine einfache Bedingungsüberdeckung bietet weniger, als eine Zweig- und Anweisungsüberdeckung
- Die Mehrfach-Bedingungsüberdeckung enthält zwar die Zweigüberdeckung, sie ist aber sehr aufwendig. Zudem sind nicht alle Variationen realisierbar (z.B. können in dem Beispielprogramm niemals die Teilbedingungen `(eingabe.charAt(c) == 'A')` und `(eingabe.charAt(c) == 'E')` gleichzeitig den Wert `true` annehmen
- Die Minimale-Mehrfach-Bedingungsüberdeckung ist ein praktikabler Kompromiss

4. Datenflussbasierte Überdeckung (*Defs-Uses*-Überdeckung)

- Die Verwendung von Variablen wird analysiert
- Fragestellung: Führt der Wert einer Variablen, bei der Nutzung einer Variablen an weiteren Stellen im Programm zu einem Fehler?
- Es werden verschiedene Kategorien unterschieden
  - Die Definition (*def*) einer Variablen / Wertzuweisung
  - Die Verwendung einer Variablen in einem Ausdruck zur Berechnung eines Wertes (berechnende Benutzung, *computational-use*, *c-use*)
  - Die Verwendung einer Variablen in Bedingungen bzw. Prädikaten zur Berechnung von Wahrheitswerten (prädikative Benutzung, *predicate-use*, *p-use*)

- Kontrollflußgraph in der Datenflußdarstellung:




```
// eingabe ist vom Typ String
int c=0, v=0;

while((c < eingabe.length())
      && (eingabe.charAt(c) >= 'A')
      && (eingabe.charAt(c) <= 'Z')){
  if((eingabe.charAt(c) == 'A')
      || (eingabe.charAt(c) == 'E')
      || (eingabe.charAt(c) == 'I')
      || (eingabe.charAt(c) == 'O')
      || (eingabe.charAt(c) == 'U')){
    v++;
  }

  c++;
}
System.out.println("Der Text enthält "
    + c + " Grossbuchstaben. "
    + "Davon sind " + v + " Vokale.");
```

Der String eingabe wurde ausgeblendet

- Wir identifizieren nun bestimmte Kanten- und Kontenmengen zur späteren Definition von Überdeckungskriterien
- Sei $V=\{1,...,n\}$ die Menge der Knoten im Kontrollflussgraphen
- Dann ist $p=(s_0,...,s_k)$, mit $s_j \in V$ ein Pfad, wenn s_i mit s_{i+1} durch eine Kante verbunden ist (für $0 \leq i < k$)

- Definitionsfreier Pfad: Wird die Variable x im Knoten s_0 definiert, dann ist der Pfad $(s_0,...,s_k)$ genau dann definitionsfrei, wenn die Variable in den Knoten $s_1,...,s_k$ nicht erneut definiert wird

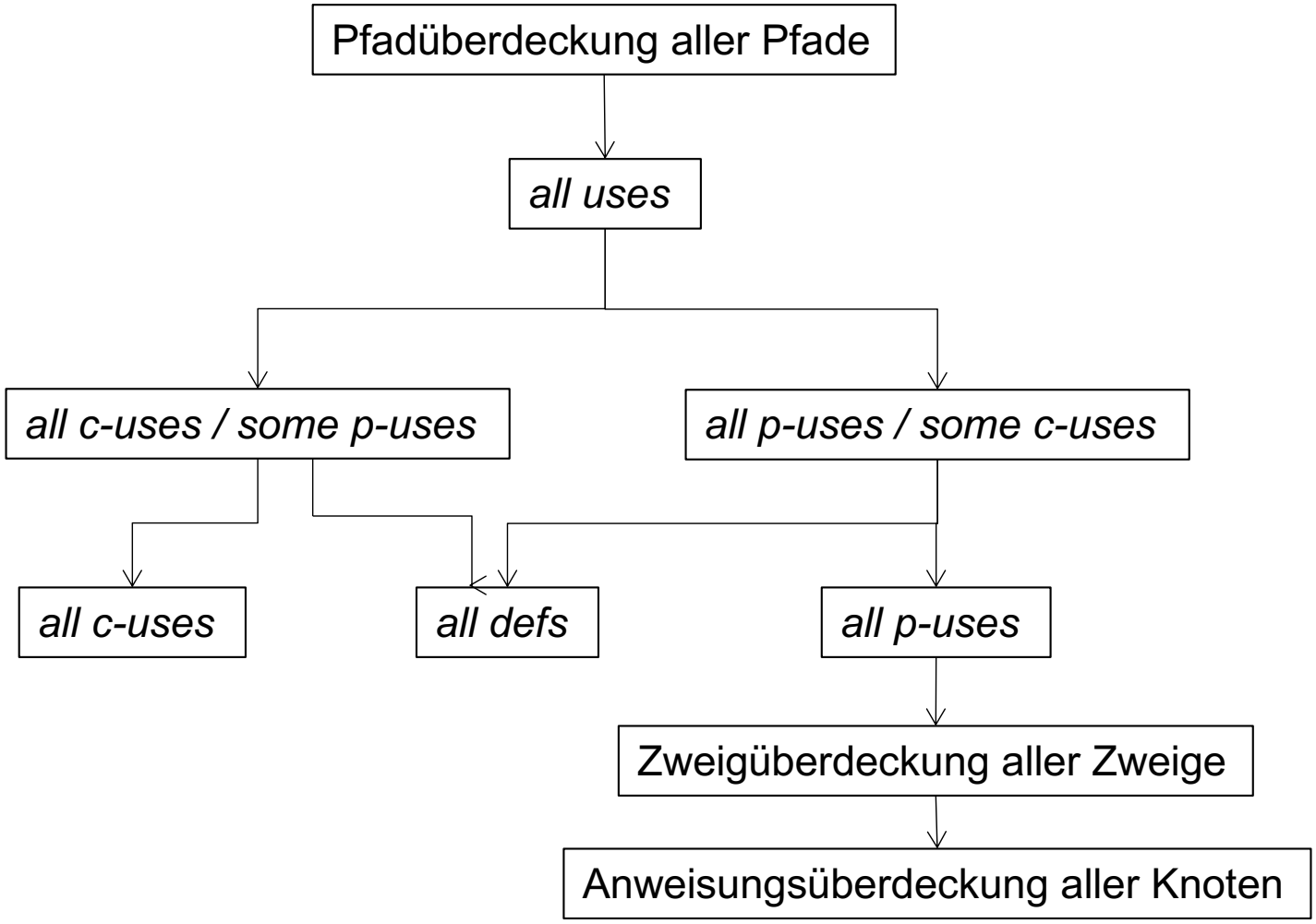
Schwaches Kriterium

- ***All Definitions*** (*All-defs*)
  - Die Testfälle durchlaufen für jede Definition einer Variablen einen definitionsfreien Pfad zu mindestens einem p-use oder c-use

- ***All c-uses***
  - Die Testfälle durchlaufen für jede Definition einer Variablen einen Pfad zu allen definitionsfrei erreichbaren c-uses

- ***All p-uses***
  - Die Testfälle durchlaufen für jede Definition einer Variablen einen Pfad zu allen definitionsfrei erreichbaren p-uses
- ***All uses***
  - Die Testfälle durchlaufen für jede Definition einer Variablen einen Pfad zu allen definitionsfrei erreichbaren p- und c-uses
- ***All c, some p***
  - Die Testfälle erfüllen das *All-c-uses*-Kriterium. Existiert zu einer Definition keine berechnende Nutzung, so wird zusätzlich mindestens ein definitionsfreier Pfad zu einer prädikativen Nutzung hinzugenommen
- ***All p, some c***
  - Analog zu *All-c-some-p*-Kriterium mit vertauschten Rollen

- Überblick über die Glas-Box-Testverfahren

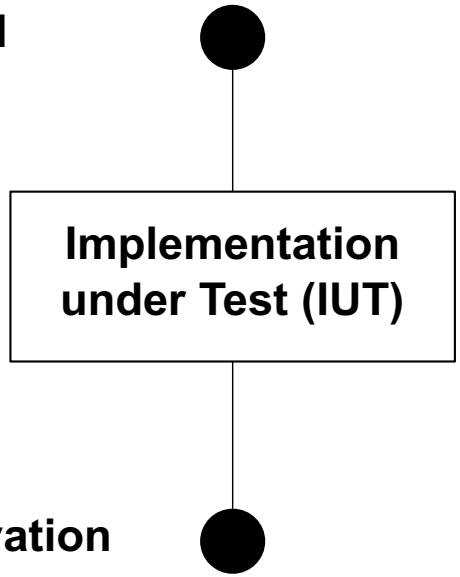


Test objektorientierter Programme

Dynamische Prüfung

Test objektorientierter Programme

– Die bisher betrachteten funktionalen Testverfahren waren auf die prozedurale Programmierung ausgerichtet



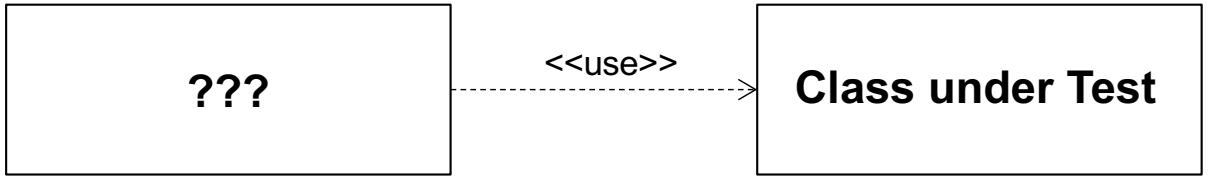

```
int q = quadrat(2);
```

```
int quadrat(int a){
   return a*a;
}
```

```
assertEquals(4, q);
```

- Bei objektorientierten Programmen sind zusätzliche Eigenschaften zu beachten
  - Das Ergebnis einer Methode kann zusätzlich zu den Parametern auch vom Zustand des Objekts (den Attributen) abhängen (Objekt ist immer implizit ein Parameter)
  - Parameter und Rückgabewerte können Objekte sein
  - Vererbung (überschreiben von Methoden) spielt eine Rolle
  - Assoziationen (Aggregation, Komposition)
  - Polymorphismus

- Wiederverwendbarkeit von Klassen erschwert das Testen
  - Einsatzzweck von Klassen nicht immer klar definiert
  - Allgemeinheit führt zu vielen möglichen Testfällen



  - Die Reihenfolge der Methodenaufrufe kann über den Zustand das Ergebnis beeinflussen
  - Aufrufreihenfolge bei Verwendung in einem Framework unbekannt

- Es sind die verschiedenen Arten von Klassen zu unterscheiden
  - normale Klassen
  - abstrakte Klassen und Schnittstellen
  - parametrisierte (generische) Klassen
  - Unterklassen

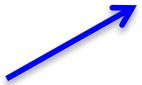
- Elemente der Objektorientierung erschweren das Testen
  - Vererbung von Attributen und Methoden
    - Redundanz wird eliminiert zu Lasten von zusätzlichen Abhängigkeiten
  - Polymorphismus und dynamische Bindung
    - Test jeder möglichen Bindung nötig

– Testen normaler Klassen

Addierer
- summe : int
+ setSumme(n : int) + getSumme() : int + add(n : int)

Diskutieren Sie das Testen von getter- und setter-Methoden

- Test zustandsverändernder Methoden
  1. Testfälle herleiten
  2. Operationen testen, die den Objektzustand nicht ändern
  3. Operationen testen, die den Objektzustand ändern
     Zustandsräume sind lokal an Objekt gebunden. Initialisierung und Auswertung der Tests erfolgt deshalb am Objekt
  4. Jede Folge abhängiger Operationen in der gleichen Klasse testen

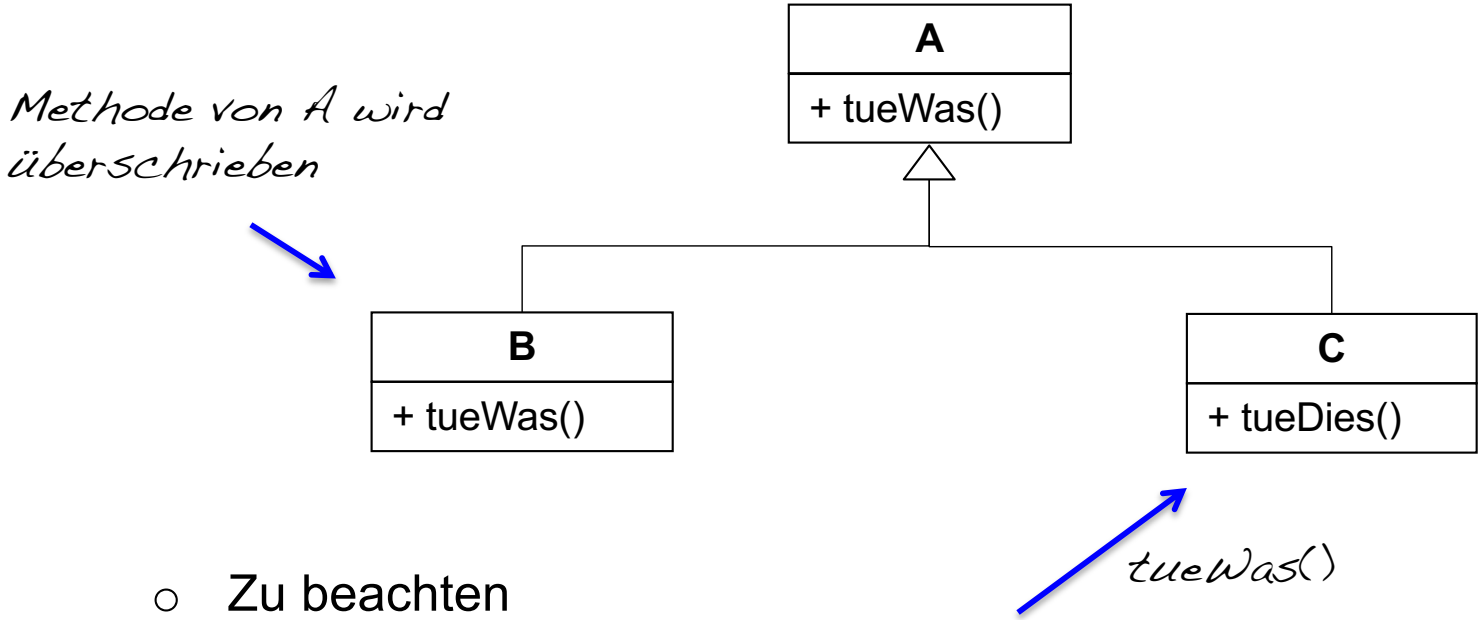


die Verwendung einer Operation sollte unter allen praktisch relevanten Bedingungen getestet werden

alle Äquivalenzklassen von Objektzuständen sind zu betrachten

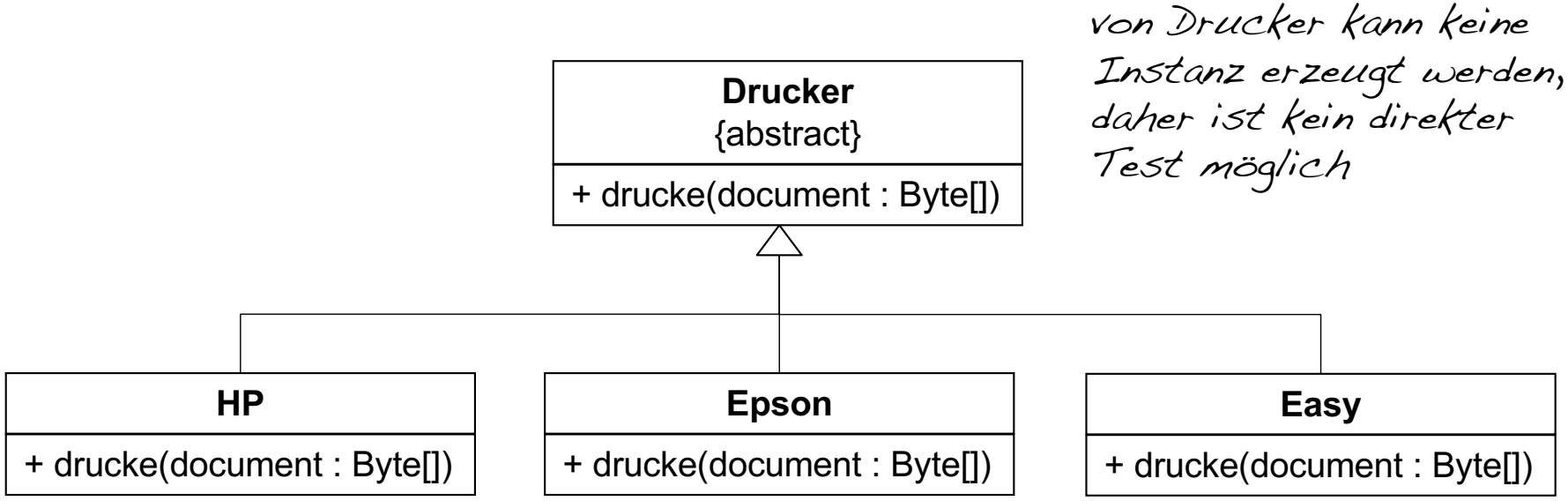
- Testen des Objektlebenszyklus
  - Falls Operationen abhängig vom Objektzustand sind, dann kann die Reihenfolge der Aufrufe bedeutend sein
  - Es sind dann ein zustandsbezogener Test mit den entsprechenden Überdeckungen durchzuführen
  - Verschiedene Laufzeitfehler lassen sich durch eine Datenflussanalyse aufdecken
    - Array-Grenzen
    - Division durch Null
    - Typecast
    - I/O
  - Eine Datenflussanalyse muss über Objektgrenzen hinweg erfolgen
  - Polymorphismus
    - Aufrufsignatur (dynamische Bindung)
    - Aufrufreihenfolge
    - Wertebereiche

- Testen von Unterklassen
  1. Systematischer Test aller Methoden der Oberklasse
  2. Systematischer Test aller Unterklassen



  - Zu beachten
    - Alle Testfälle für geerbte und nicht redefinierte Operationen der Oberklasse müssen erneut ausgewertet werden (Unterklasse definiert neuen Kontext)
    - Für redefinierte Operationen sind vollständig neue Testfälle zu erstellen

- Testen abstrakter Klassen
  - Möglichst einfache Unterklasse testen, die die abstrakte Klasse implementiert



- Testen nicht sichtbarer Bestandteilte
  - Test privater Methoden
  - Überprüfen privater Attribute (Kontrolle des Zustandes)

  - Alternativen
    - Es wird nur über die (öffentliche) Schnittstelle einer Klasse getestet. Ein Test der privaten Methoden und eine Kontrolle des Zustandes findet nur indirekt statt.
    - Lockerung der Sichtbarkeit auf Paketebene (*package scope*). Erfordert entsprechende Richtlinie für die Einstellung von Sichtbarkeiten. Testen im gleichen Paket ist möglich
    - Extra Testzugang (z.B. innere Testklassen)
    - Zugriff über Reflection

- Verwendung von Reflection für das Testen von privaten Attributen und Methoden

```
public class Calculator {
   private int methodCalls = 0;

   private boolean istNull(int i){
      methodCalls++;
      return (i==0);
   }

   public int divide(int n, int d){
      methodCalls++;
      if(istNull(d)) throw new ArithmeticException("Division durch 0");
      return n/d;
   }
}
```

Zustand soll geprüft werden

private Methode soll getestet werden

```
public class CalculatorTest {
   private Calculator cal;

   @Before
   public void erzeugeCalculator(){
      cal = new Calculator();
   }

   @Test
   public void testState(){
      final Class<?> clazz = cal.getClass();
      try {
         final Field privateAttr = clazz.getDeclaredField("methodCalls");
         privateAttr.setAccessible(true);
         assertEquals(0, ((Integer)privateAttr.get(cal)).intValue());
      } catch (Exception e) {
            e.printStackTrace();
      }
   }

   // Testmethode für private Methode auf der nächsten Seite
}
```

Achtung: Attributname muss bei Änderung angepasst werden

Differenziertes Exception-Handling erforderlich: NoSuchFieldException, SecurityException, IllegalArgumentExceptio, IllegalAccessException

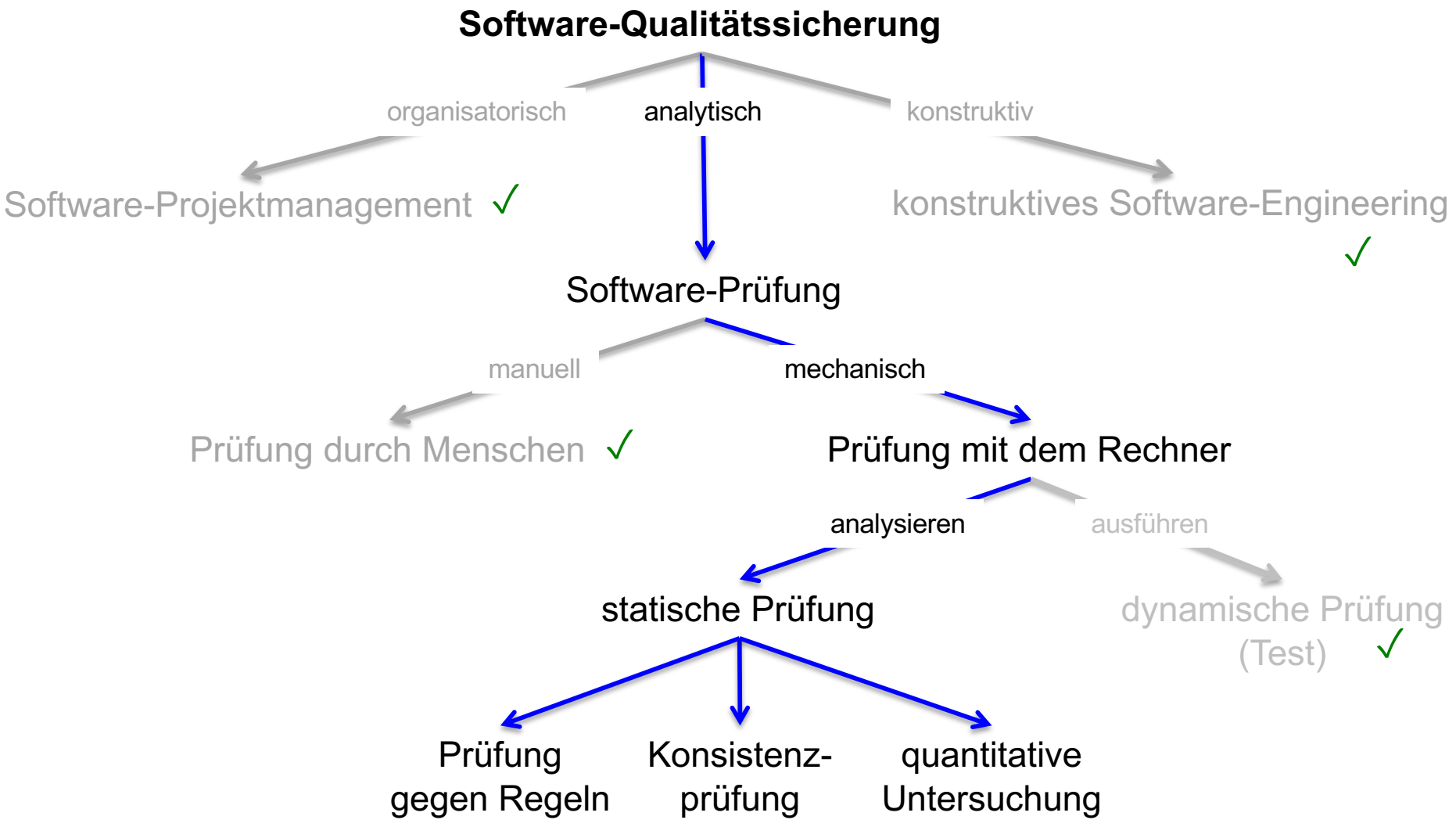
```
@Test
public void testIstNull(){
   final Class<?> clazz = cal.getClass();
   try {
      Class<?>[] param = {int.class};
      Method privat = clazz.getDeclaredMethod("istNull", param);
      privat.setAccessible(true);
      assertTrue(((Boolean)privat.invoke(cal, 0)).booleanValue());
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Achtung: Methodenname und Parametertypen müssen bei Änderung angepasst werden

Aufruf der privaten Methode

Statische Prüfung

Metriken



Gliederung der Software-Qualitätssicherung nach [LL10]

Statische Prüfung

- Durch eine statische Prüfung sollen vorhandene Fehler oder fehlerträchtige Stellen in einem Dokument gefunden werden
- Im Gegensatz zum Review wird die statische Prüfung durch Werkzeuge vorgenommen
- Die Prüfung durch Werkzeuge setzt voraus, dass das Dokument eine formale Struktur besitzt
- Beispiele:
  - Anforderungsdokument (Prosa): Rechtschreib- und Grammatikprüfung
  - UML-Diagramme: Prüfung auf Einhaltung der UML-Syntax
  - Programmcode : Syntaxanalyse, Konformitäts-/Konventionsanalyse, Prüfung auf Kontrollfluss- und Datenflussanomalien

Wir konzentrieren uns später auf die Analyse des Programmcodes

Die Bezeichnung „statische Prüfung" weist darauf hin, dass der Programmcode für die Prüfung nicht ausgeführt werden muss

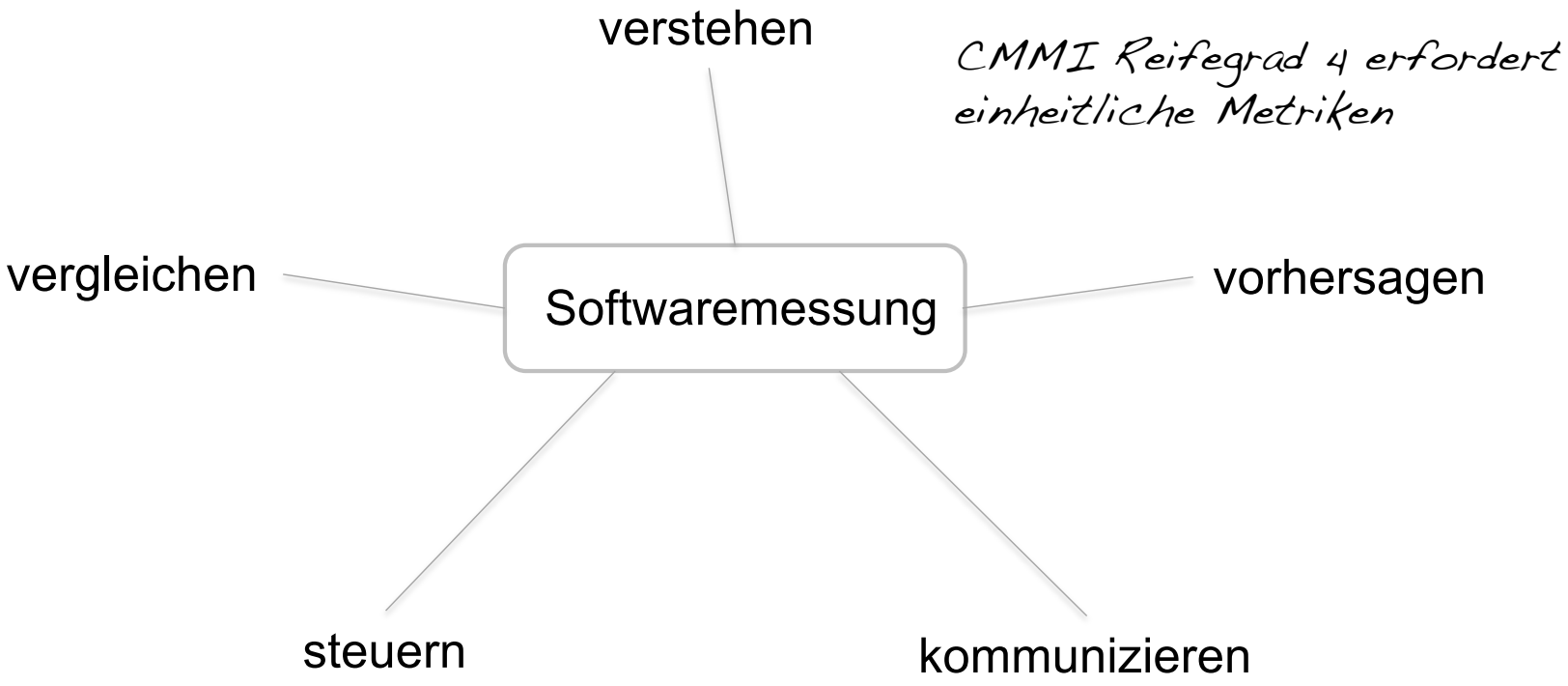
- Zeitpunkt einer statischen Prüfung (von Programmcode)
  - vor einer manuellen Prüfung (z.B. Review)
  - vor einem Komponententest
  - vor einem Integrationstest
- Die statischen Analysewerkzeuge werden von den Entwicklern/Testern eingesetzt
- Die statischen Analysewerkzeuge liefern häufig auch Messwerte, die Hinweise auf die Quantität und die Qualität geben können
- Wir betrachten daher zunächst Softwaremetriken

Metriken

- Vermessung und Prüfung des Quellcodes mit dem Ziel, die Qualität und Quantität zu bestimmen
- Für Software stehen leider keine aussagekräftigen physikalischen Maßeinheiten, wie z.B. Länge, Gewicht, Spannung, Widerstand, zur Verfügung
- Es stehen aber eine Vielzahl von Software-Metriken zur Verfügung
- Trotz ihres heuristischen Ansatzes können sinnvoll gewählte Software-Metriken hilfreiche Aussagen liefern

> *Quantitative Bewertung von Software-Systemen mit Hilfe von Kenngrößen*

- Neben dem Produkt kann auch der Prozess vermessen werden (z.B. Aufwand, Kosten, Dauer, ...)
- Was ist Sinn und Zweck der Softwaremessung? [SSB10]



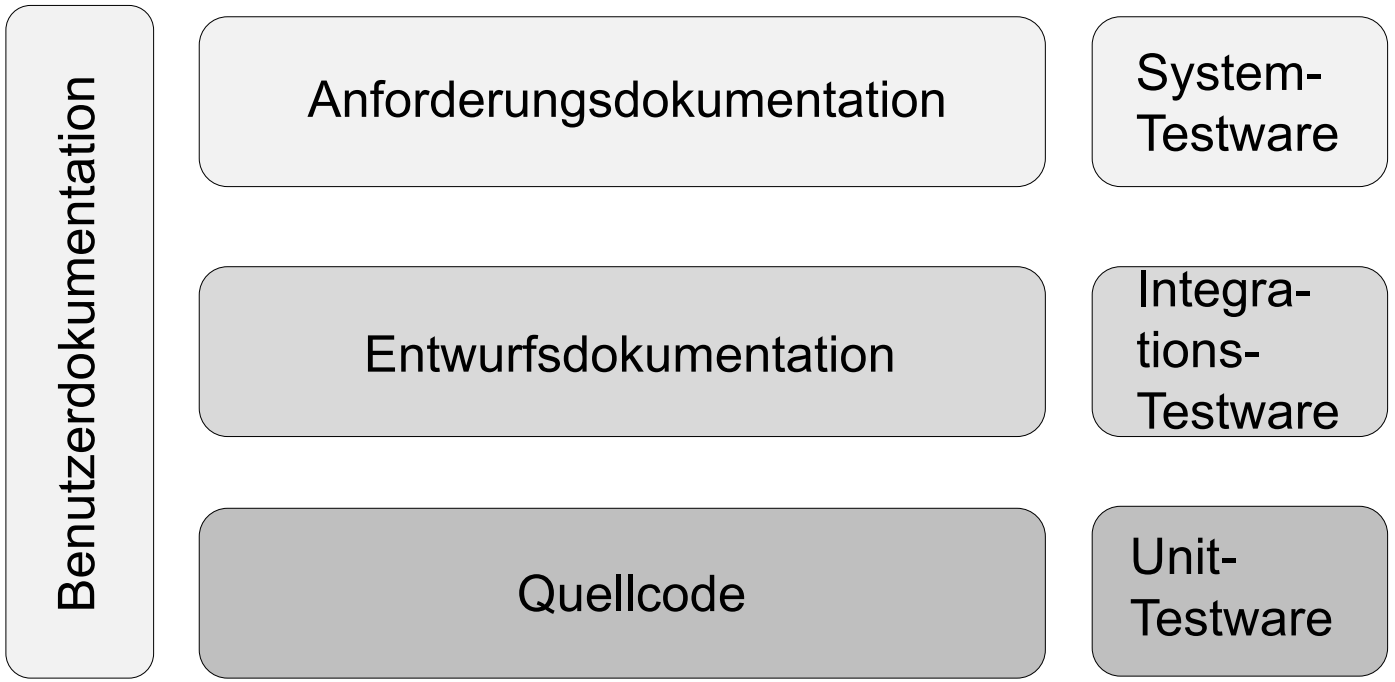
„You cannot control what you cannot measure" [DeMarco]

- Unterteilung von Software-Metriken
  - Quantitätsmetriken
  - Komplexitätsmetriken
  - Qualitätsmetriken
- Wir wissen bereits: Software ist mehr als nur das Programm
  - Anforderungsdokumentation/Spezifikation
  - Entwurfsdokumentation
  - Code
  - Testware
  - Benutzerhandbücher
- Daraus ergeben sich die unterschiedlichen Objekte der Softwaremessung
- Man kann unterschiedliche Sichten auf die Objekte einnehmen

- Sicht der Typen
  - Natürlichsprachliche Texte
  - Diagramme
  - Tabellen
  - Code
- Sicht des Zwecks
  - Anforderungsartefakt
  - Entwurfsartefakt
  - Codeartefakt
  - Testartefakt
  - Beschreibungsartefakt
- Sicht des Anwenders
  - System/Benutzerinteraktion / Systemausgabe
  - Benutzerdokumentation

Quantitätsmetriken

– Quellen



- Entwurfsgrößen
  - Strukturierte Entwurfsgrößen
    - Module
    - Funktionen
    - Datenobjekte
    - Schnittstellen
  - Objektmodellgrößen
    - Klassen / Klasseninteraktionen
    - Methoden
    - Attribute
  - Datenmodellgrößen
    - Datenentitäten
    - Datenattribute
    - Datenschlüssel
    - Datenbeziehungen
    - Datensichten

- Codegrößen:
  - Codedateien
  - Codezeilen
  - Anweisungen
  - Prozeduren / Funktionen / Methoden
  - Module / Klassen
  - Entscheidungen
  - Logikzweige
  - Aufrufe
  - Definierte Datenelemente
  - Benutzte Datenelemente
  - Benutzte Operanden
  - Datenobjekte
  - Datenzugriffe
  - Interaktionselemente
  - Kommentare

Codekomplexität

- Die LOC sagen wenig über die Komplexität der Software aus, weil die Beziehungen (Abfragen, Aufrufe, Sprünge, ...) zwischen den Anweisungen und Daten nicht berücksichtigt werden
- Einer der ersten Vorschläge für eine Metrik zur Messung der Codekomplexität stammt von Halstead

- Halstead-Metriken
  - Orientieren sich an dem Komplexitätsbegriff aus der Informationstheorie nach Shannon
    - Eine komplexe Nachricht enthält viele unterschiedliche Zeichen relativ zur Länge der Nachricht
  - Ein Quellcode ist nach Halstead komplex, wenn in den Befehlen viele unterschiedliche Operatoren und Operanden relativ zur Anzahl der Befehle vorkommen

- Halstead teilt die Programmelemente in Operatoren und Operanden ein
  - Operatoren
    - Schlüsselwörter
    - Operatoren
    - Präprozessoranweisungen
  - Operanden
    - Variablen
    - Konstanten
    - Bezeichner

- Für die Berechnung der Metriken werden die folgenden Basisparameter benötigt
  - t : Anzahl der unterschiedlichen Operatoren
  - d : Anzahl der unterschiedlichen Operanden
  - n_t : Gesamtzahl aller Operatoren im Programm
  - n_d : Gesamtzahl aller Operanden im Programm

können über Syntaxanalyse ermittelt werden

– Beispiel

```
int ggt(int a, int b){
   while(a != b){
      if (a > b)
         a -= b;
      else
         b -= a;
   }
   return a;
}
```

$t = 11$

$d = 2$

$n_t = 20$

$n_d = 11$

Operatoren

`int`	3 x	`(), {}`	5 x
`,`	1 x	`while`	1 x
`!=`	1 x	`if else`	1 x
`>`	1 x	`-=`	2 x
`return`	1 x	`;`	3 x
`ggt`	1 x		

Operanden

`a`	6 x	`b`	5 x

- Aus den Basisgrößen werden weitere Größen berechnet
  - Größe des Vokabulars: $G = t + d$
  - Länge des Programms: $N = n_t + n_d$
  - Volumen des Programms: $V = N \times \log_2 G$

Annahme: Operatoren und Operanden werden mit der gleichen Anzahl von Bits codiert

- Die Komplexität berechnet Halstead in Bezug auf eine minimale Implementierung (G und N minimal)
- Da zu einem beliebigen Programm die Angabe einer minimalen Implementierung nicht trivial ist, arbeitet Halstead mit unteren Schranken *d.h. keine Implementierung kann ein kleineres Volumen haben*
  - Eine untere Schranke für die Operanden ist die Anzahl der Ein- und Ausgabeoperanden: d^*
  - Wenn eine (fiktive) Programmiersprache einen speziellen Operator ◆ hat, welcher das komplette Programm berechnet, dann würde das Programm nur aus einer Zeile der folgenden Form bestehen

$$\texttt{Operand}_1 = \blacklozenge(\texttt{Operand}_2, \ldots, \texttt{Operand}_{d^*})$$

- Die Anzahl der Operatoren würde für dieses Programm 2 (◆, =) betragen
- Damit gilt für die Größe des Minimalvokabulars: $G^* = d^* + 2$
- Entsprechend wir ein Minimalvolumen des Programms definiert: $V^* = G^* \times \log_2 G^*$

Klammerung wird hier nicht gezählt (abhängig von der Programmiersprache)

– Aufgabe: Berechnen Sie G, N, d^*, t^*, G^*, V und V^* für das Beispielprogramm `ggt`

– Der Level L misst, zu welchem Grad die (reale) Implementierung von der Minimalimplementierung abweicht:

$$L = V^* / V$$

im „optimalen" Fall ist $L = 1$

soll eine Aussage über die Transparenz des Programmes liefern

- Ein Programm ist umso einfacher aufgebaut, je näher sich *L* der Zahl 1 nähert
- Daraus ergibt sich die Schwierigkeit (*difficulty*) eines Programms

$$D = 1 / L$$

- Der Aufwand (*effort*), der zum Erstellen bzw. Verstehen des Programms benötigt wird, ergibt sich nach Halstead als das Produkt von Programmvolumen und Schwierigkeitsgrad

$$E = V \times D$$

- Der Aufwand, der für ein Programmmodul verwendet werden muss, steigt damit überproportional mit dem Code-Volumen an

$$E = V \times D = V / L = V / (V^* / V) = V^2 / V^*$$

- Aufgabe: Berechnen Sie *L*, *D* und *E* für das Beispielprogramm `ggt`

- Kritik an Halstead-Metriken
  - Einteilung in Operatoren und Operanden nicht immer klar (z.B. Lisp)
  - Die Metriken beziehen sich allein auf syntaktische Eigenschaften des Programms. Semantische Eigenschaften, wie z.B. der Kontrollfluss, werden nicht berücksichtigt

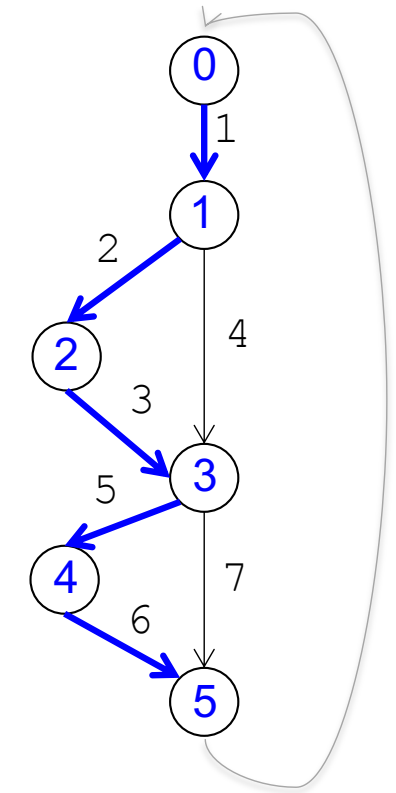
- McCabe-Metrik
  - Anstatt syntaktische Elemente zu betrachten (wie Halstead), konzentriert sich McCabe auf den Kontrollfluss eines Programms
  - Der Programmcode wird dazu in einen Graphen überführt (Kontrollflussgraph)
  - Mit Hilfe der Graphentheorie nach Euler werden dann Aussagen über den Kontrollflussgraphen getroffen
  - Sei $G=(N,E)$ ein Graph mit der Knotenmenge $N=\{1,...,|N|\}$ und der Kantenmenge $E=\{1,...,|E|\}$
  - Ein Pfad kann, wenn man von der Reihenfolge der besuchten Knoten abstrahiert, als Vektor $p=(p_1,...,p_{|E|}) \in \mathbb{N}^{|E|}$ dargestellt werden
  - p_i gibt dabei an, wie häufig der Pfad die i-te Kante des Graphen durchläuft
  - Man kann zeigen, dass sich jeder geschlossene Pfad (Start- und Endknoten sind gleich) als Linearkombination von Elementarpfaden darstellen kann
  - Die minimale Menge von Elementarpfaden wird als Basis bezeichnet

– Beispiel (aus [Hof13]):

```
0:  public static int manhatten(int a, int b){
1:     if (a < 0)
2:        a = -a;
3:     if (b < 0)
4:        b = -b;
5:     return a+b;
    }
```

$$\begin{pmatrix}1\\1\\1\\0\\1\\1\\0\end{pmatrix} = 1\cdot\begin{pmatrix}1\\1\\1\\0\\0\\0\\1\end{pmatrix} + 1\cdot\begin{pmatrix}1\\0\\0\\1\\1\\1\\0\end{pmatrix} - 1\cdot\begin{pmatrix}1\\0\\0\\1\\0\\0\\1\end{pmatrix}$$

Hilfskante wurde hier ausgeblendet



Hilfskante, um einen geschlossenen Pfad zu bekommen

- Die zyklomatische Zahl

hier: stark zusammenhängender Graph

$$V(G) = |E| - |N| + 1$$

Graphentheorie

gibt die Größe einer Basis für einen stark zusammenhängenden Graphen $G = (N, E)$ an

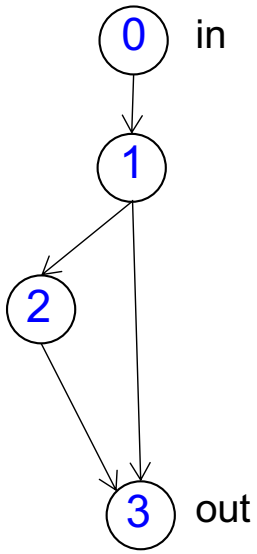
- Da ein Kontrollflussgraph durch eine Hilfskante erweitert werden muss, um einen stark zusammenhängenden Graphen zu erhalten, definiert McCabe die zyklomatische Komplexität eines Kontrollflussgraphen G durch

$$V(G) = |E| - |N| + 2$$

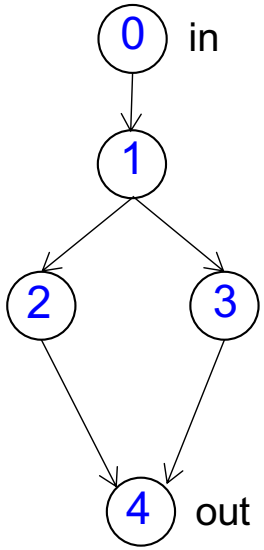
hier: Kontrollflussgraph

– Aufgabe: Berechnen Sie die zyklomatische Komplexität der beiden folgenden Abfragen

```
if (B) X;
```




```
if (B) X; else Y;
```



- Eigenschaften der zyklomatischen Komplexität
  - $V(G)$ ist unabhängig von der Programmlänge, da sequentiell durchlaufende Programmabschnitte beliebig verlängert werden können, ohne $V(G)$ zu verändern
  - $V(G)$ wird nur durch die Struktur des Kontrollflussgraphen beeinflusst, die Zusammenfassung von Befehlen spielt keine Rolle
  - Wird zum Kontrollflussgraphen eine einzelne Kante hinzugefügt, so erhöht sich $V(G)$ um 1
  - Wird aus dem Kontrollflussgraphen eine einzelne Kante entfernt, so verkleinert sich $V(G)$ um 1

- Empfehlung von McCabe: Die zyklomatische Komplexität eines Moduls sollte den Wert 10 nicht überschreiten

- Besteht ein Programm aus *n* unabhängigen Funktionen, dann ist der Kontrollflussgraph $G=(N, E)$ ein Wald aus Untergraphen $G_1,...,G_n$.
- Dann gilt eine verallgemeinerte Form der zyklomatischen Komplexität

$$v(G) = |E| - |N| + 2n = \sum_{i=1}^{n} V(G_i)$$

- Objektorientierte Metriken
  - Die einzelnen Methoden können mit den bekannten Metriken vermessen werden
  - Die folgenden Metriken bieten eine objektorientierte Sicht

Abkürzung	**Metrik**	**Typ**
OV	Object Variables	Umfangsmetrik
CV	Class Variables	Umfangsmetrik
NOA	Number of Attributes	Umfangsmetrik
WAC	Weighted Attribute per Class	Umfangsmetrik
WMC	Weighted Method per Class	Umfangsmetrik
DOI	Depth Of Inheritance	Vererbungsmetrik
NOD	Number of Descandants	Vererbungsmetrik
NORM	Number of Redefined Methods	Vererbungsmetrik

- Die Gewichtung bei WMC erfolgt häufig über die zyklomatische Komplexität
- Wenn A die Menge aller Attribute und M die Menge aller Methoden ist, ergibt sich damit die folgende Berechnung

$$WAC = \sum_{i=1}^{|A|} v(a_i)$$

Komplexität eines Attributs

$$WMC = \sum_{i=1}^{|M|} v(m_i)$$

Komplexität einer Methode

Konformitäts- / Konventionsanalyse

- Der Compiler als statisches Analysewerkzeug
- Der Compiler führt auf jeden Fall eine Syntaxanalyse durch
- Abhängig vom Compiler werden weitere Prüfungen durchgeführt
  - Typprüfung
  - Ermittlung von nicht initialisierten Variablen
  - Erstellung von Verwendungsnachweisen
  - Über- oder Unterschreitung von Feldgrenzen
  - Nicht erreichbarer Code
  - ...

Datenfluss-anomalieanalyse

Datenflussanomalieanalyse

Compilerunterstützung ist abhängig von der Programmiersprache

- Frage: Gibt es bei der Verarbeitung der Daten Auffälligkeiten, die auf einen Fehler hindeuten?
- Eine Anomalie ist eine (aus statischer Sicht) auffällige Anweisungs-sequenz
  - Eine Anomalie muss nicht zwingend zu einem Fehler führen
  - Eine Anomalie muss genauer untersucht werden, um einen Fehler auszuschließen
- Eine Datenflussanomalie äußert sich in unstimmigen Variablenzugriffen
- Aus Eingabedaten werden über eine Sequenz von Anweisungen und Variablenzugriffen Ausgabedaten berechnet
- Ein spezielle Bearbeitungssequenz von Daten entspricht einem Pfad im Kontrollflussgraphen des Programms

- Mit einer Variablen *x* können auf einem Pfad verschiedene Aktionen ausgeführt werden

  wenn der Kontext klar ist, dann kann die Angabe der Variablen auch entfallen

  - *x* wird definiert (*d(x)*)
    - Jede Wertzuweisung führt zu einer neuen Definition der Variablen
    - Das Anlegen einer neuen Variablen führt nicht zwingend zu einer Initialisierung des Speicherplatzes (z.B. in C)
    - Hinweis: Eine Deklaration ist nur die Bekanntgabe eines Namens und führt nicht zur Reservierung von Speicherplatz
  - *x* wird referenziert (*r(x)*)
    - Eine Referenzierung kann ein *c-use* oder ein *p-use* sein
  - *x* wird undefiniert (*u(x)*)
    a) Der Speicherplatz wird der Variablen entzogen (z.B. der Gültigkeitsbereich einer lokalen Variablen wird verlassen)
    b) Der reservierte Speicherplatz wird nicht initialisiert

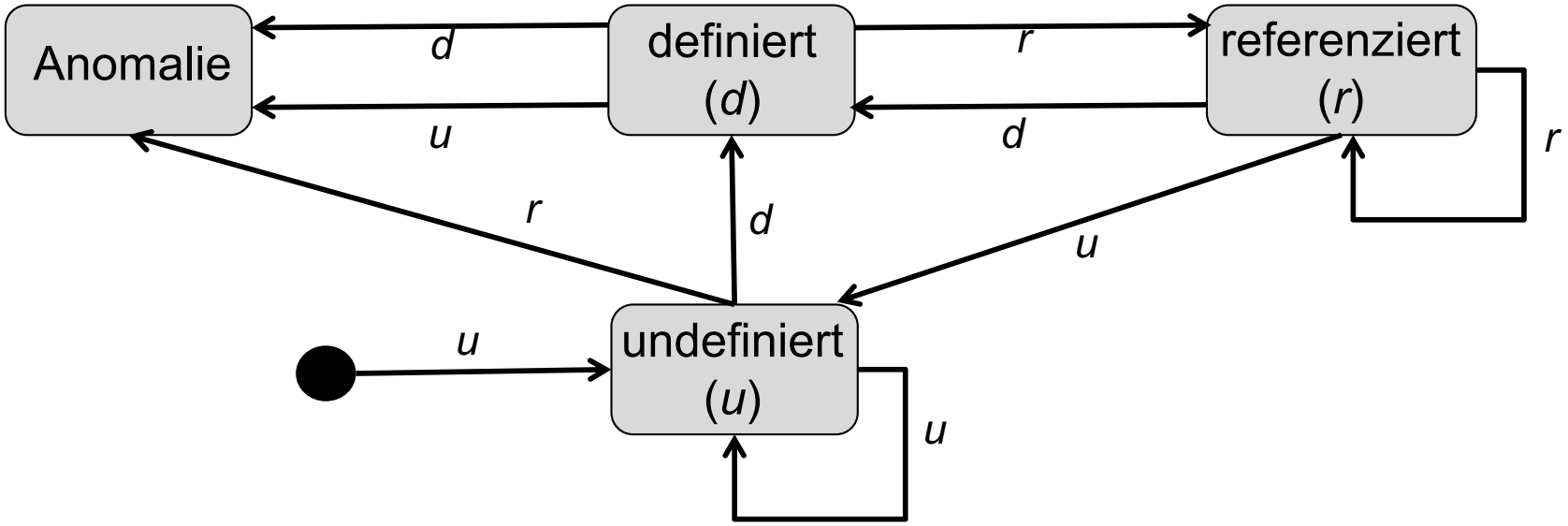
- Die folgenden Zugriffssequenzen auf eine Variable x stellen keine Anomalien dar

Zugriffssequenz	Beschreibung
dr	Variable bekommt neuen Wert und wird danach verwendet
rd	Variable wird verwendet und bekommt danach einen neuen Wert
rr	Variable wird zweimal hintereinander verwendet
ru	Variable wird verwendet und danach gelöscht
ud	Eine undefinierte Variable wird initialisiert
uu	Eine undefinierte Variable wird gelöscht

- Die folgenden Zugriffssequenzen stellen Anomalien dar

Zugriffssequenz	Beschreibung
ur	Variable wird ohne Initialisierung verwendet
du	Variable bekommt einen Wert und wird dann direkt gelöscht
dd	Variable wird zweimal hintereinander überschrieben

– Die Datenflussanomalieanalyse kann durch den folgenden Zustandsautomaten beschrieben werden



nach P. Liggesmeyer. *Software Qualität*. Spektrum Verlag, 2002

– Über den analogen endlichen Automaten kann eine reguläre Grammatik definiert werden
– Eine Datenflussanomalieanalyse ist damit effizient durch den Compiler realisierbar

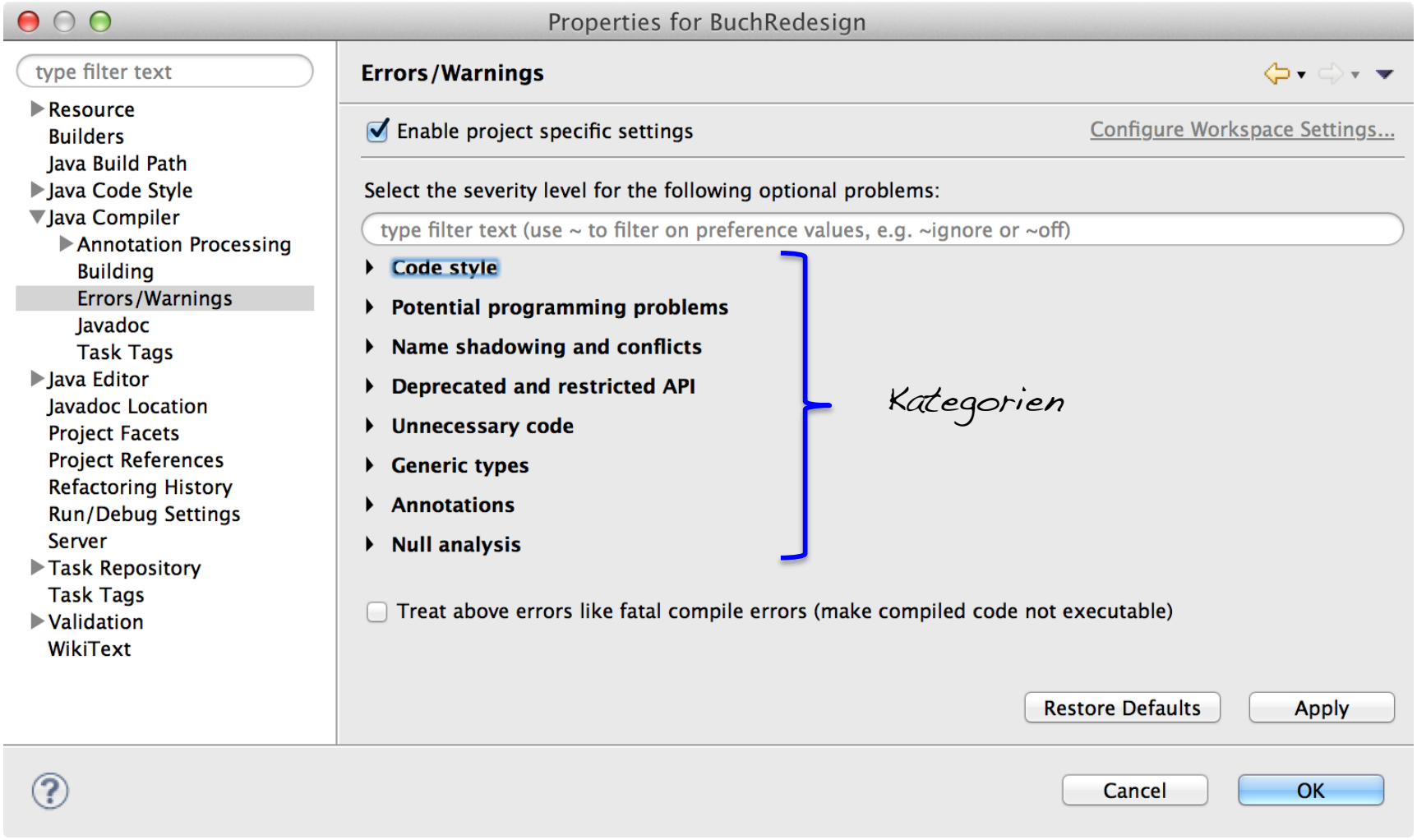
– Aufgabe: Führen Sie für die folgende Implementierung der Funktion `minmax` eine Datenflussanomalieanalyse durch! Die Funktion `minmax` erhält zwei Zahlen (*by reference*) übergeben und sortiert die Zahlen in eine aufsteigende Reihenfolge.

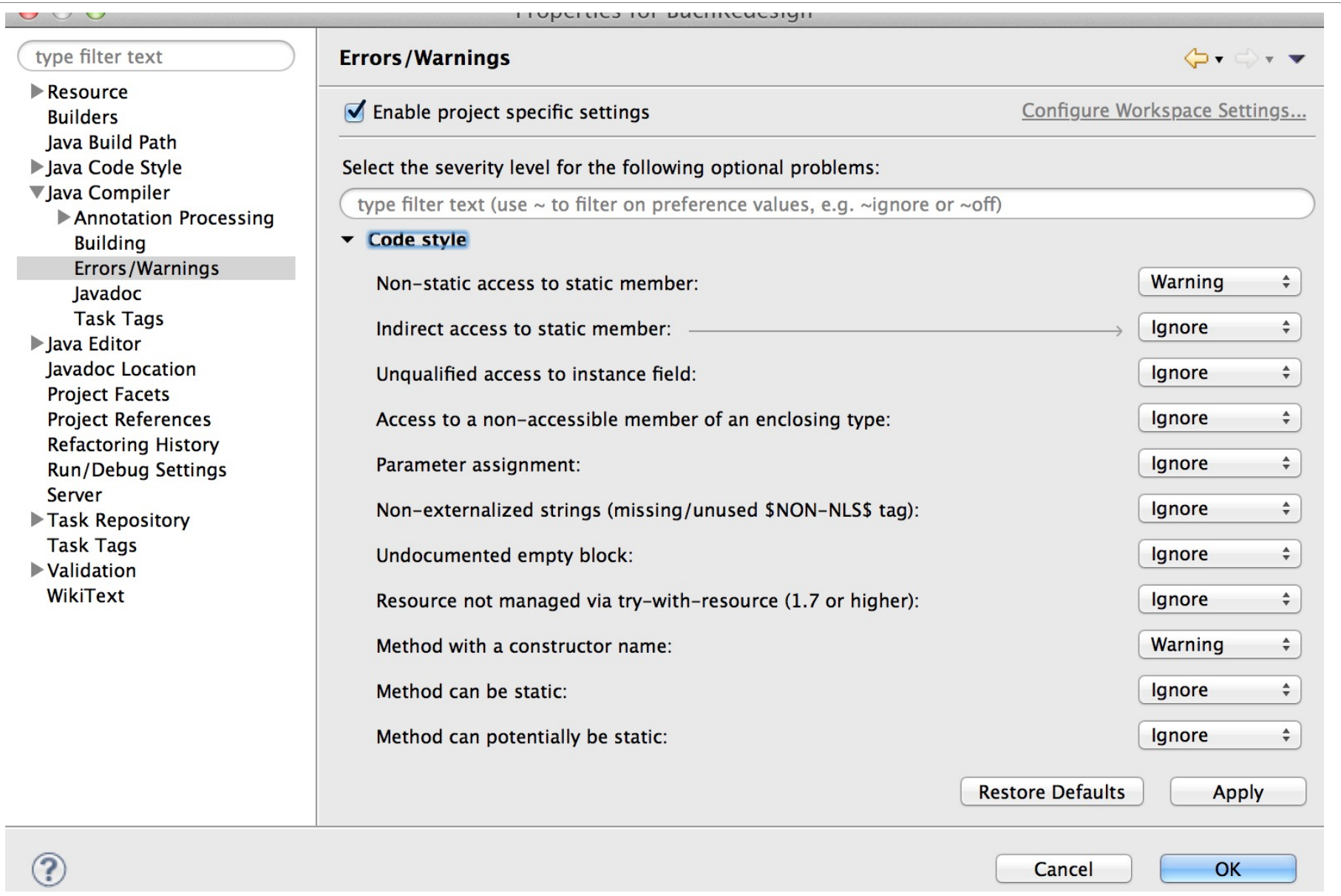
Aufgabe nach P. Liggesmeyer. *Software Qualität*. Spektrum Verlag, 2002

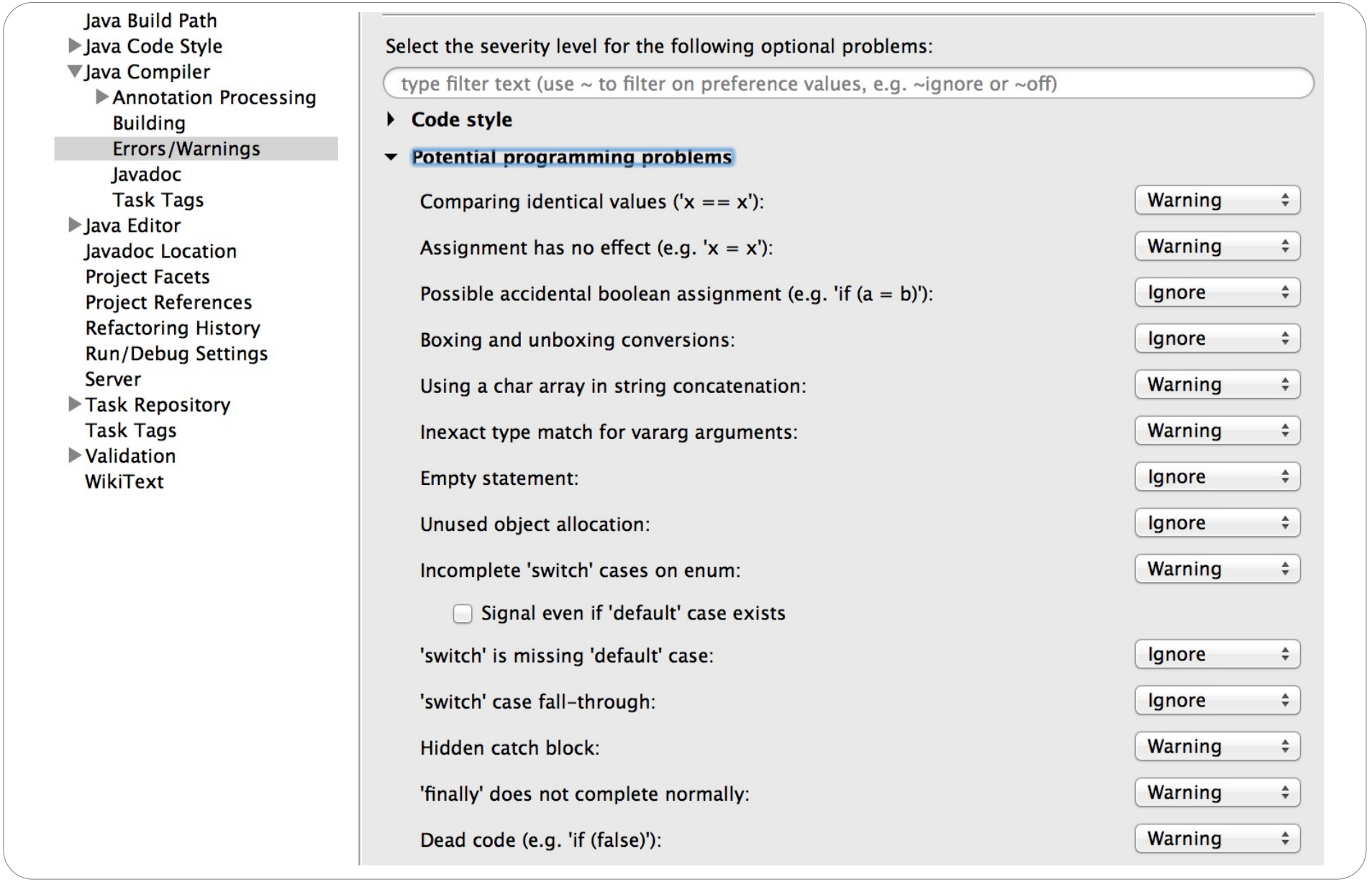
```
void minmax (int& min, int& max){
   int temp;

   if (min > max){
      max = temp;
      max = min;
      temp = min;
   }
}
```

Programmiersprache C++

– Beispiel: Statische Prüfungen des Java Compiler (Konfiguration über Eclipse)




type filter text
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Annotation Processing
Building
Errors/Warnings
Javadoc
Task Tags
Java Editor
Javadoc Location
Project Facets
Project References
Refactoring History
Run/Debug Settings
Server
Task Repository
Task Tags
Validation
WikiText
Errors/Warnings
Enable project specific settings
Configure Workspace Settings...
Select the severity level for the following optional problems:
type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)
Code style
Non-static access to static member: Warning
Indirect access to static member: Ignore
Unqualified access to instance field: Ignore
Access to a non-accessible member of an enclosing type: Ignore
Parameter assignment: Ignore
Non-externalized strings (missing/unused $NON-NLS$ tag): Ignore
Undocumented empty block: Ignore
Resource not managed via try-with-resource (1.7 or higher): Ignore
Method with a constructor name: Warning
Method can be static: Ignore
Method can potentially be static: Ignore
Restore Defaults
Apply
Cancel
OK

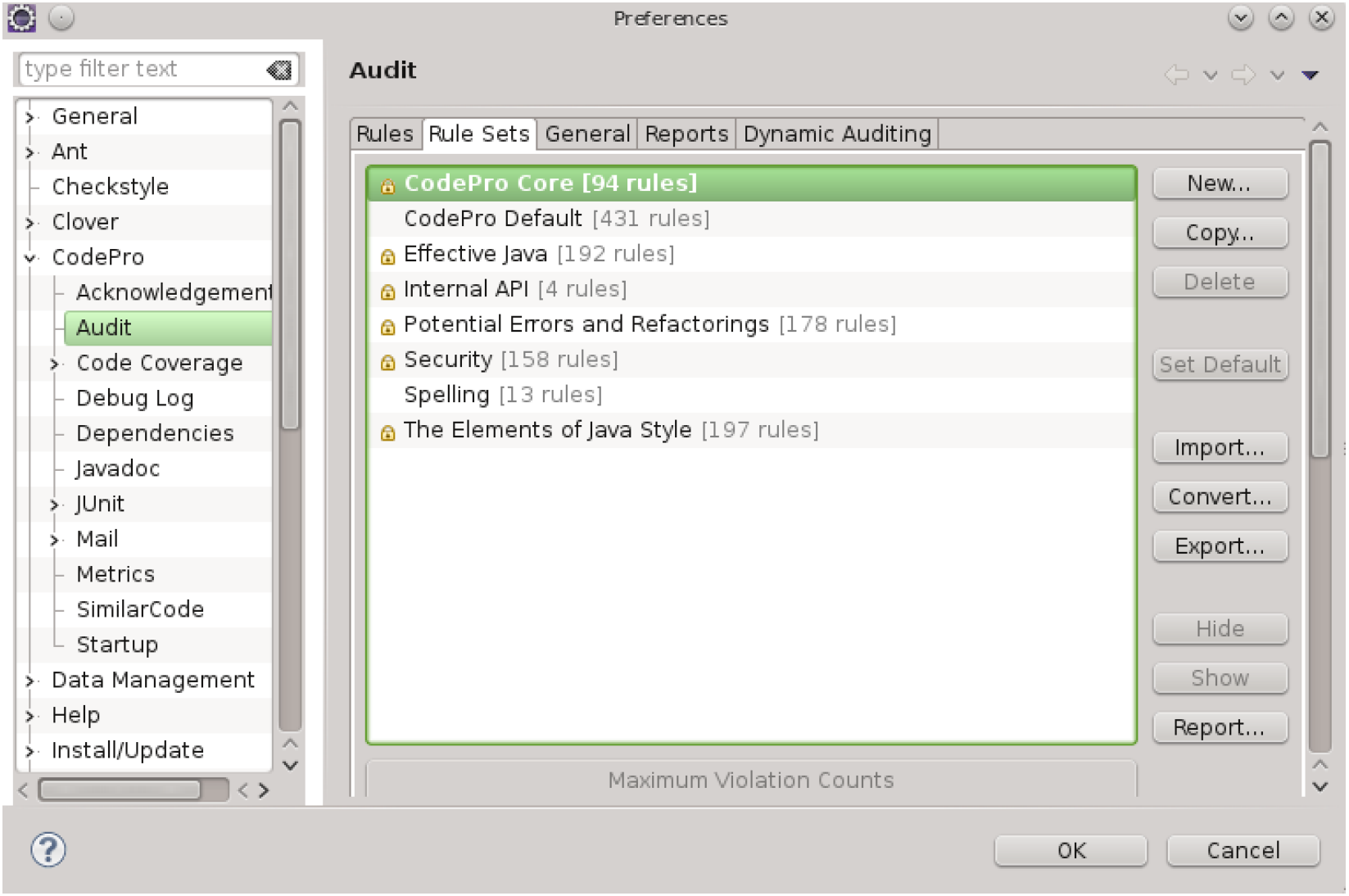

Java Build Path
Java Code Style
Java Compiler
Annotation Processing
Building
Errors/Warnings
Javadoc
Task Tags
Java Editor
Javadoc Location
Project Facets
Project References
Refactoring History
Run/Debug Settings
Server
Task Repository
Task Tags
Validation
WikiText
Select the severity level for the following optional problems:
type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)
Code style
Potential programming problems
Comparing identical values ('x == x'): Warning
Assignment has no effect (e.g. 'x = x'): Warning
Possible accidental boolean assignment (e.g. 'if (a = b)'): Ignore
Boxing and unboxing conversions: Ignore
Using a char array in string concatenation: Warning
Inexact type match for vararg arguments: Warning
Empty statement: Ignore
Unused object allocation: Ignore
Incomplete 'switch' cases on enum: Warning
Signal even if 'default' case exists
'switch' is missing 'default' case: Ignore
'switch' case fall-through: Ignore
Hidden catch block: Warning
'finally' does not complete normally: Warning
Dead code (e.g. 'if (false)'): Warning

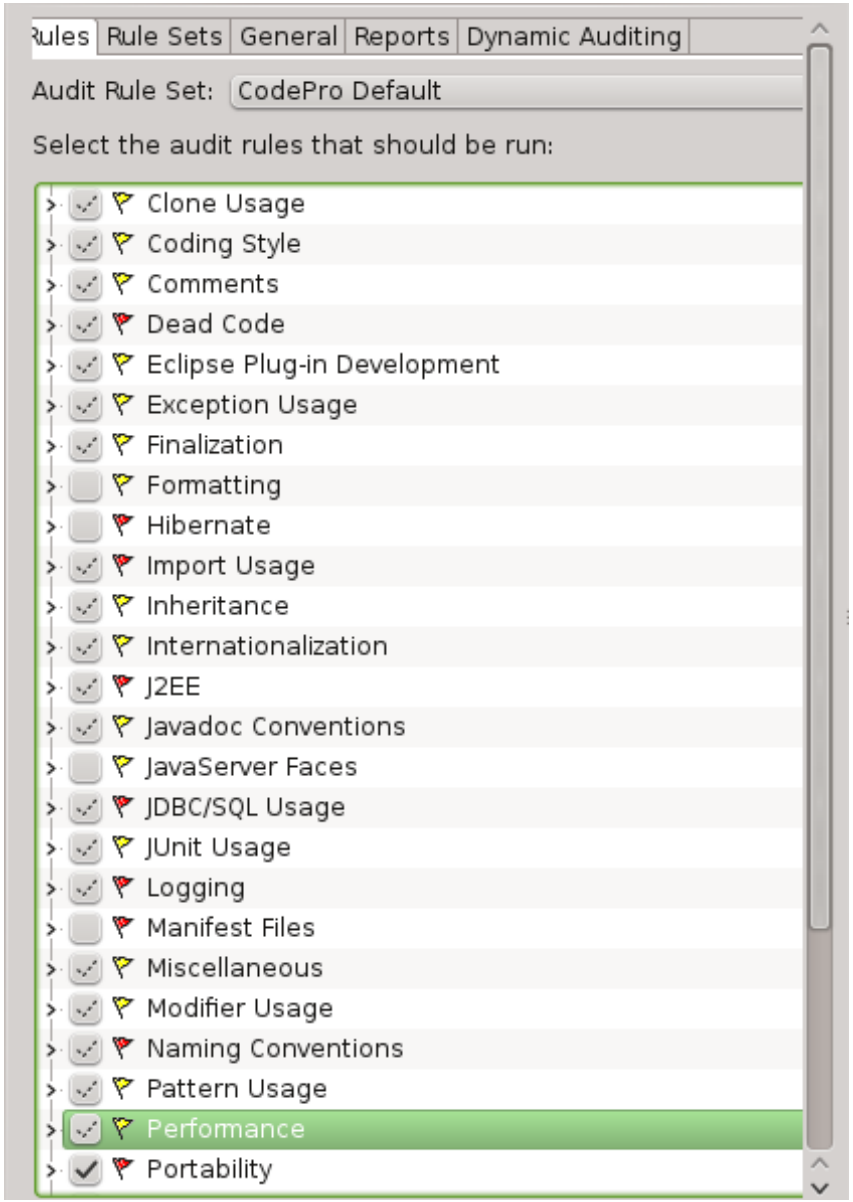
- Statische Analyse mit speziellen Werkzeugen (Sourcecode-Checker)
  - PMD
  - Checkstyle
  - CodePro Audit/Metrik
  - ...
- Die statischen Sourcecode-Checker führen Prüfungen nach (umfangreichen) Regelsätzen durch
  - Es können eigene Regelsätze konfiguriert werden
  - Regelverletzungen können als Warnungen oder Compilefehler klassifiziert werden
  - Es wird eine Auflistung der Regelverletzungen erstellt

Automatisierte Prüfung auf Einhaltung von Codierungsrichtlinien

- Folgende Prüfungen können z.B. vorgenommen werden:
  - Duplizierter Quellcode
  - nicht erreichbarer Code
  - Grad der Kommentierung
  - lange Klassen und Methoden
  - hohe Komplexität (zyklomatische Komplexität, Schwierigkeit)
  - unbenutzte Elemente
  - Literale in Abfragen/Berechnungen
  - fehlendes `final` für Methodenparameter und Attribute
  - Leere catch-Blöcke
  - Fehlendes Logging für gefangene Ausnahmen
  - ungenutzte lokale Variablen
  - Sichtbarkeit von Attributen
  - fehlender *load factor* bei Erzeugung einer `HashMap`
  - Klassennamen beginnen mit Kleinbuchstaben
  - ...

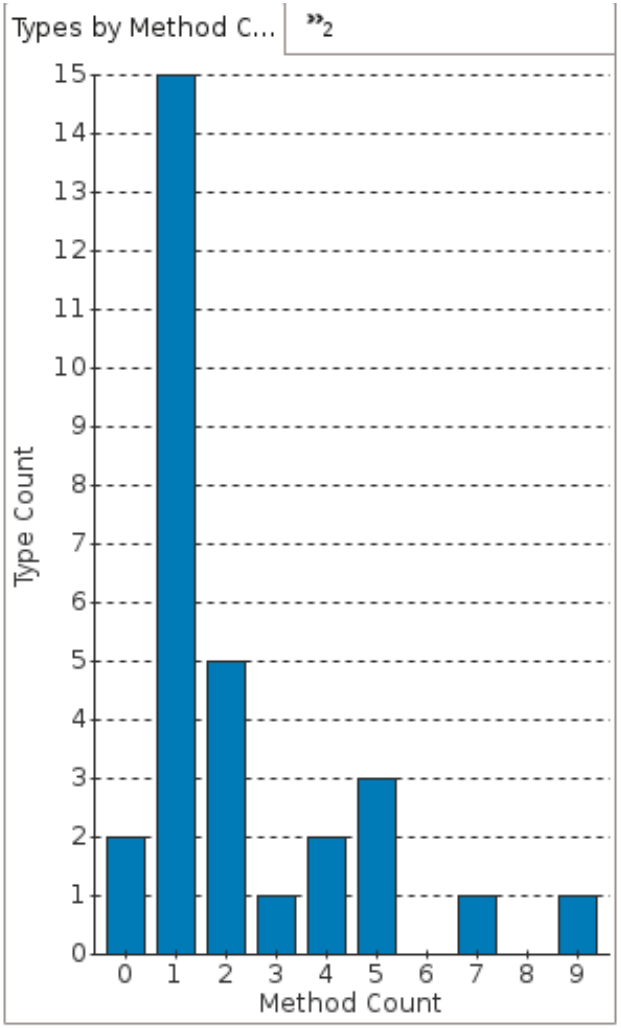
– Beispiel: CodePro Audit Regelsätze




Rules
Rule Sets
General
Reports
Dynamic Auditing
Audit Rule Set:
CodePro Default
Select the audit rules that should be run:
Clone Usage
Coding Style
Comments
Dead Code
Eclipse Plug-in Development
Exception Usage
Finalization
Formatting
Hibernate
Import Usage
Inheritance
Internationalization
J2EE
Javadoc Conventions
JavaServer Faces
JDBC/SQL Usage
JUnit Usage
Logging
Manifest Files
Miscellaneous
Modifier Usage
Naming Conventions
Pattern Usage
Performance
Portability

– Beispiel: Berechnung von Metriken mit CodePro

Metric	Value
Abstractness	6.6%
Average Block Depth	0.96
Average Cyclomatic Complexity	1.38
Average Lines Of Code Per Method	7.61
Average Number of Constructors Per Type	0.43
Average Number of Fields Per Type	0.90
Average Number of Methods Per Type	2.23
Average Number of Parameters	0.46
Comments Ratio	1.9%
Efferent Couplings	13
Lines of Code	711
Number of Characters	19,891
Number of Comments	14
Number of Constructors	13
Number of Fields	33
Number of Lines	892
Number of Methods	67
Number of Packages	11
Number of Semicolons	404
Number of Types	30
Weighted Methods	111



Statische Prüfung in der Praxis

- Es ist nicht sinnvoll, alle vorhandenen Prüfregeln anzuwenden, die ein Werkzeug bietet *Zu viele Warnungen, die nicht mit den Qualitätszielen des Projekts zu tun haben*
- Es müssen daher für das Unternehmen/das Projekt hilfreiche Prüfregeln identifiziert werden
- Für jede Prüfregel ist eine passende Warnstufe festzulegen (z.B. Warning oder Error)
  - Für jede Warnstufe ist eine passende Folge festzulegen (z.B. Build nicht möglich)
- Die statischen Prüfungen müssen in den Entwicklungsprozess eingebunden werden (Wer, Was, Wann, Wie, Womit)

  Qualitätssicherungsplan
- Die Werkzeuge sind entsprechend zu konfigurieren

- Einen Spezialfall stellt die Softwaremessung dar
- Der Wert einer einmaligen Messung ist beschränkt
- Das Ziel ist daher die Einrichtung einer andauernden (kontinuierlichen) Messung (nur so können die Messwerte die Grundlage für eine Steuerung oder Vorhersage sein)
- Es ist eine Messdatenbank aufzubauen
- Vermessung des Prozesses
  - Produktivität
  - Ressourceneinsatz
  - Termintreue
- Vermessung des Produkts
  - Quantität
  - Qualität
  - Komplexität

Erfordert eine kontinuierliche Erfassung der Aufwände

- Die Mitarbeiter stehen einer kontinuierlichen Messung in der Regel sehr skeptisch gegenüber
  - Sie befürchten eine Bewertung ihrer persönlichen Produktivität und die Bewertung der Qualität ihrer Arbeitsergebnisse
  - Sie zweifeln die Aussagekraft der/einiger Metriken an (evtl. auch zu Recht)

- Der Aufbau einer kontinuierlichen Messung ohne die Beteiligung der Mitarbeiter ist daher nicht ratsam

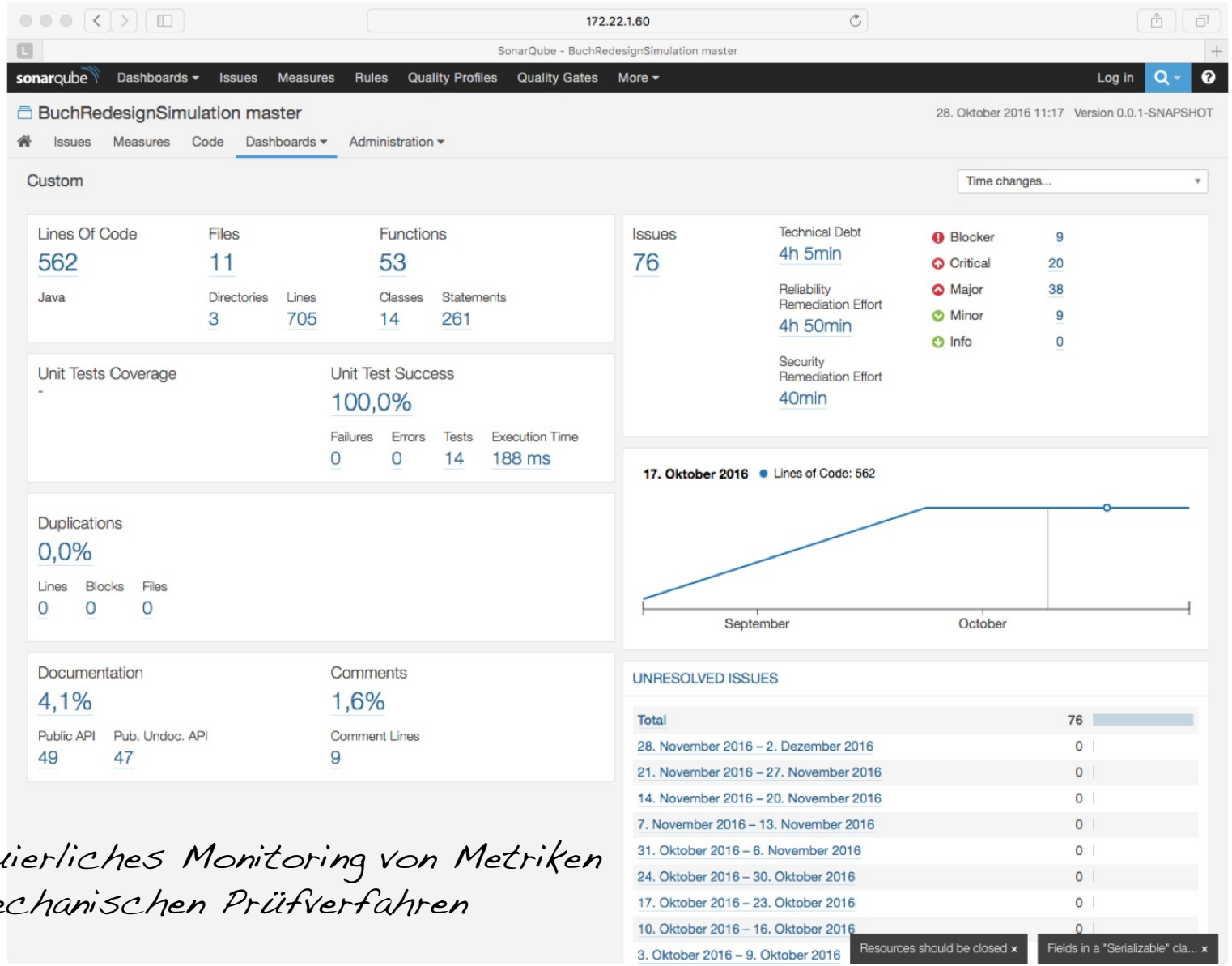
- Beteiligung der Mitarbeiter
  - Vorschlag von Metriken
  - Anpassung von Metriken
- Transparenz
  - Veröffentlichung der Messmethoden
  - Jeder Mitarbeiter hat Zugriff auf die Messdaten

- Automatisierung mit Maven
  - Konventionsanalyse: Checkstyle

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-checkstyle-plugin</artifactId>
   <version>3.1.0</version>
   <configuration>
      <configLocation>
         google_checks.xml
      </configLocation>
   </configuration>
</plugin>
```

  - Statische Codeanalyse (auf Bytecode-Ebene): SpotBugs (FindBugs)

```
<plugin>
   <groupId>com.github.spotbugs</groupId>
   <artifactId>spotbugs-maven-plugin</artifactId>
   <version>3.1.12.2</version>
</plugin>
```



Kontinuierliches Monitoring von Metriken und mechanischen Prüfverfahren

Kontinuierliche Integration (*Continuous Integration*)

- Kontinuierliche Integration (CI) ist eine Softwareentwicklungspraktik
  - Neu Artefakte / Komponenten werden häufig (täglich) integriert
  - Integration erfolgt durch einen vollautomatischen Build-Prozess

- Ziele von CI
  - Reduzierung von Risiken *primär*
    - Fehler bei der Integration können leichter/schneller lokalisiert werden
    - Eine lauffähige Produktversion kann jederzeit ausgeliefert werden
    - Nach einer Fehlerkorrektur kann schnell eine korrigierte Programmversion zur Verfügung gestellt werden
    - Probleme mit Nicht-funktionalen Anforderungen fallen frühzeitig auf

    Abhängig von den Maßnahmen im Qualitätssicherungsplan
  - Verbesserung der Produktqualität
    - Statische und dynamische Prüfungen werden regelmäßig (automatisiert) durchgeführt
    - Abweichungen von Soll-Werte werden frühzeitig erkannt

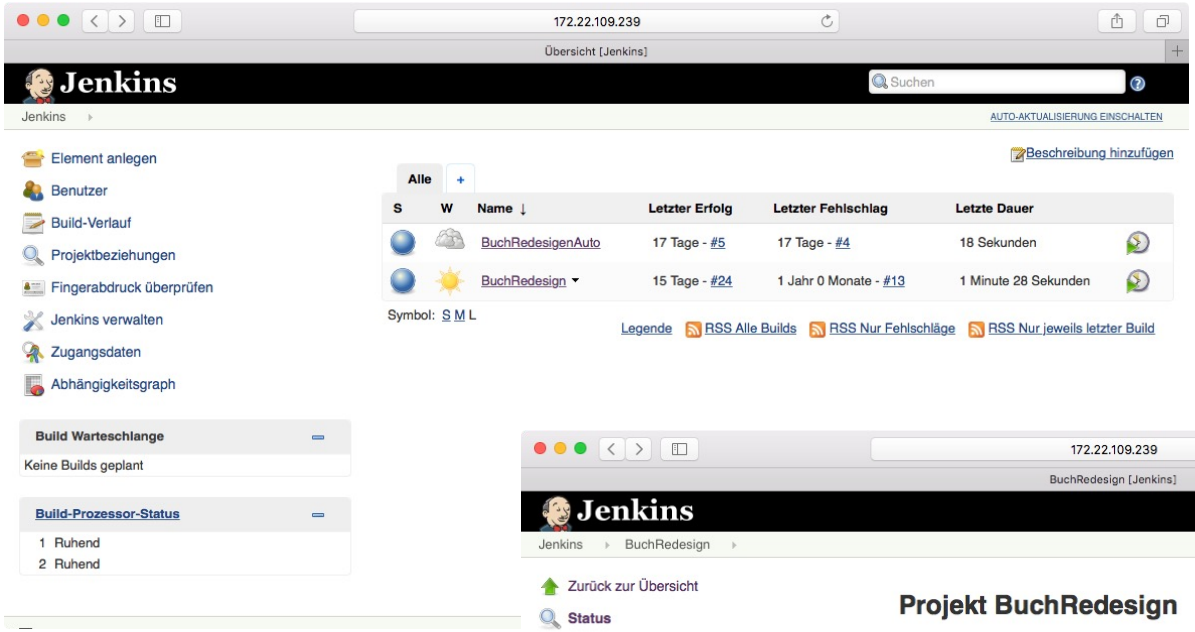
- Transparenz / Übersicht
  - Kennzahlen
  - Berichtswesen
  - Historie

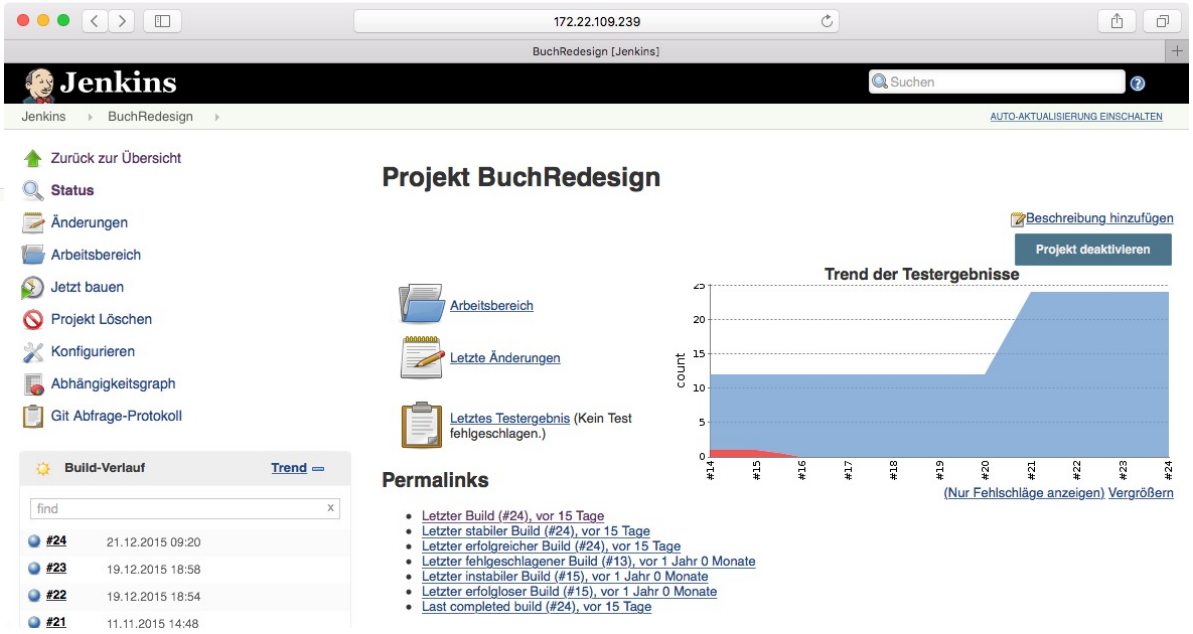
– CI-Server

unabhängig von der Entwicklungsumgebung

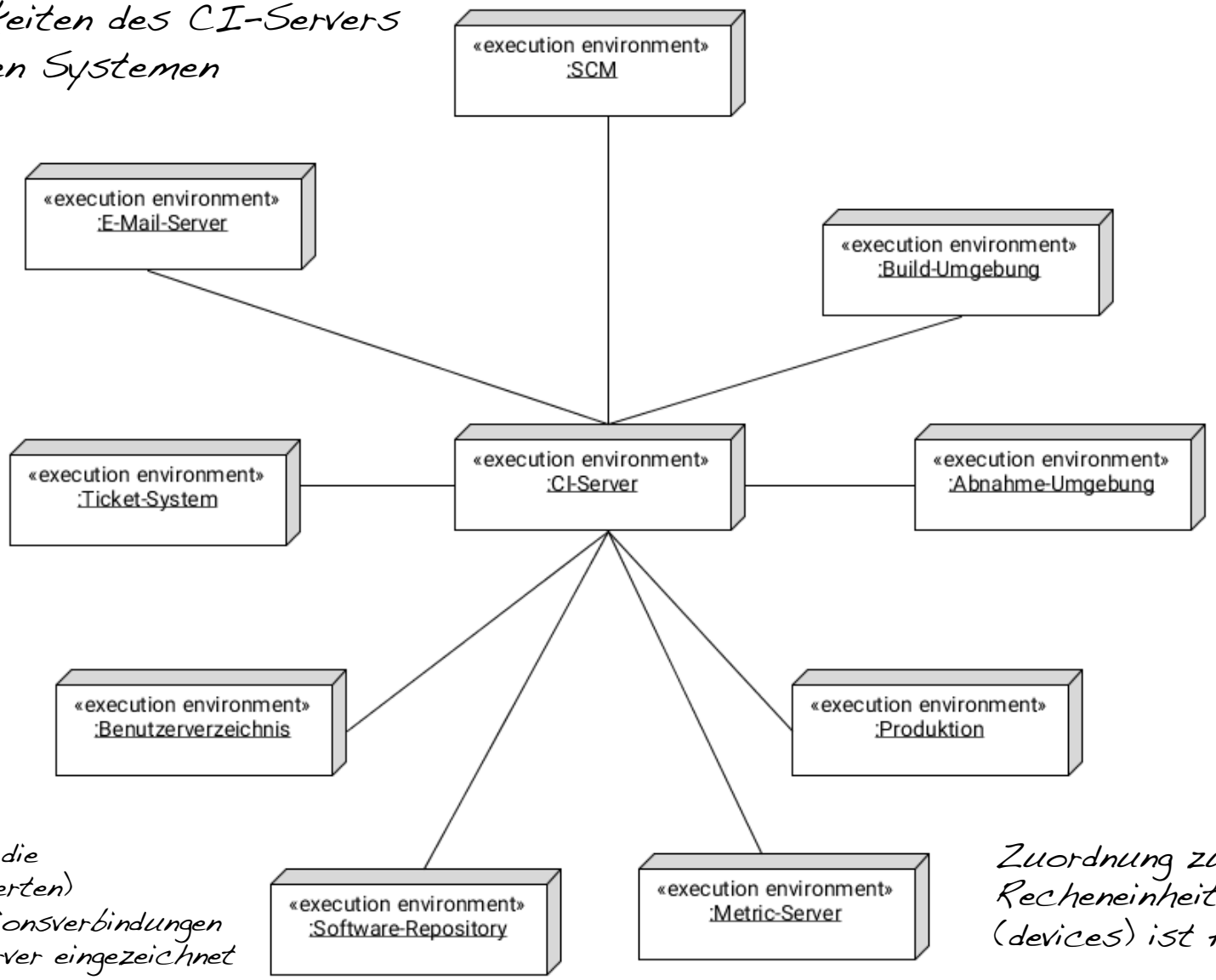
Laufzeitumgebung zur Durchführung des kompletten Integrationsprozesses

- Bereitstellung einer Oberfläche (häufig Web-Anwendung)
  - Konfiguration eines Build-Prozesses
  - Build-Status anzeigen
  - Historie von Builds anzeigen
- Auslösen eines Build-Prozesses
  a) Manuell
  b) Ereignisgesteuert (Änderungen im SCM)
  c) Zeitgesteuert (z.B. Nightly-Builds)


172.22.109.239
Übersicht [Jenkins]
Jenkins
Suchen
Jenkins
AUTO-AKTUALISIERUNG EINSCHALTEN
Element anlegen
Benutzer
Build-Verlauf
Projektbeziehungen
Fingerabdruck überprüfen
Jenkins verwalten
Zugangsdaten
Abhängigkeitsgraph
Beschreibung hinzufügen
Alle
S
W
Name ↓
Letzter Erfolg
Letzter Fehlschlag
Letzte Dauer
BuchRedesigenAuto
17 Tage - #5
17 Tage - #4
18 Sekunden
BuchRedesign
15 Tage - #24
1 Jahr 0 Monate - #13
1 Minute 28 Sekunden
Symbol: S M L
Legende
RSS Alle Builds
RSS Nur Fehlschläge
RSS Nur jeweils letzter Build
Build Warteschlange
Keine Builds geplant
Build-Prozessor-Status
1 Ruhend
2 Ruhend


172.22.109.239
BuchRedesign [Jenkins]
Jenkins
Suchen
Jenkins
BuchRedesign
AUTO-AKTUALISIERUNG EINSCHALTEN
Zurück zur Übersicht
Status
Änderungen
Arbeitsbereich
Jetzt bauen
Projekt Löschen
Konfigurieren
Abhängigkeitsgraph
Git Abfrage-Protokoll
Projekt BuchRedesign
Beschreibung hinzufügen
Projekt deaktivieren
Arbeitsbereich
Letzte Änderungen
Letztes Testergebnis (Kein Test fehlgeschlagen.)
Trend der Testergebnisse
count
#14
#15
#16
#17
#18
#19
#20
#21
#22
#23
#24
(Nur Fehlschläge anzeigen) Vergrößern
Permalinks
Letzter Build (#24), vor 15 Tage
Letzter stabiler Build (#24), vor 15 Tage
Letzter erfolgreicher Build (#24), vor 15 Tage
Letzter fehlgeschlagener Build (#13), vor 1 Jahr 0 Monate
Letzter instabiler Build (#15), vor 1 Jahr 0 Monate
Letzter erfolgloser Build (#15), vor 1 Jahr 0 Monate
Last completed build (#24), vor 15 Tage
Build-Verlauf
Trend
find
#24 21.12.2015 09:20
#23 19.12.2015 18:58
#22 19.12.2015 18:54
#21 11.11.2015 14:48

Abhängigkeiten des CI-Servers zu anderen Systemen



Es sind nur die (unspezifizierten) Kommunikationsverbindungen zum CI-Server eingezeichnet

Zuordnung zu Recheneinheiten (devices) ist flexibel

- Build-Prozess umfasst
  - Vorbereitende Maßnahmen
    - Löschen und Anlegen von Verzeichnissen
    - Auschecken von Artefakten aus dem SCM
    - Bereitstellen von Abhängigkeiten (z.B. Bibliotheken)
  - Übersetzen
    - Compilieren und Binden
  - Statische Prüfungen
  - Dynamische Prüfungen
    - Unit- und Integrationstests
    - Glass-Box-Test (Anweisungs- und Zweigüberdeckung)
  - Paketierung
  - Erzeugung von Berichten und Dokumentation
  - Bereitstellung / Verteilung

- CI-Server nutzt verschiedene Werkzeuge (häufig über Build-Tool)
  - Build-Tools (*Ant*, *Maven*)
  - Compiler / Linker (*javac*, *gcc*, *csc*)
  - Statische Analysewerkzeuge (*PMD*, *Checkstyle*, *FindBugs*)
  - Unit-Test-Runner (*JUnit*)
  - Testberichte (*Surefire*)
  - Glass-Box-Test (*Cobertura*)

- CI-Praktiken
  - Gemeinsame Codebasis im SCM
  - Automatisierte Builds
  - Automatisierte Durchführung von dynamischen und statischen Prüfungen
  - Automatisches Berichtswesen
  - Häufige Integration
  - Schnelle Build-Zyklen
  - Einfacher Zugriff auf Build-Ergebnisse und Berichte
  - Automatische Verteilung (und Bereitstellung der Umgebungen) -> Continuous Deliviery

Konfigurationsmanagement

- Softwaresysteme lassen sich durch folgende Eigenschaften charakterisieren
  - Bestehen aus zahlreichen Artefakten
  - Besitzen zahlreiche Abhängigkeiten (z.B. zu anderen Systemen)
  - Unterliegen laufend Änderungen (funktionale Erweiterungen, Fehlerbehebung, ...)
  - Der „Bau“ eines Releases kann ein komplexer Vorgang sein
  - Können sich in unterschiedlichen Versionen im Einsatz befinden
  - Werden in einem Team entwickelt

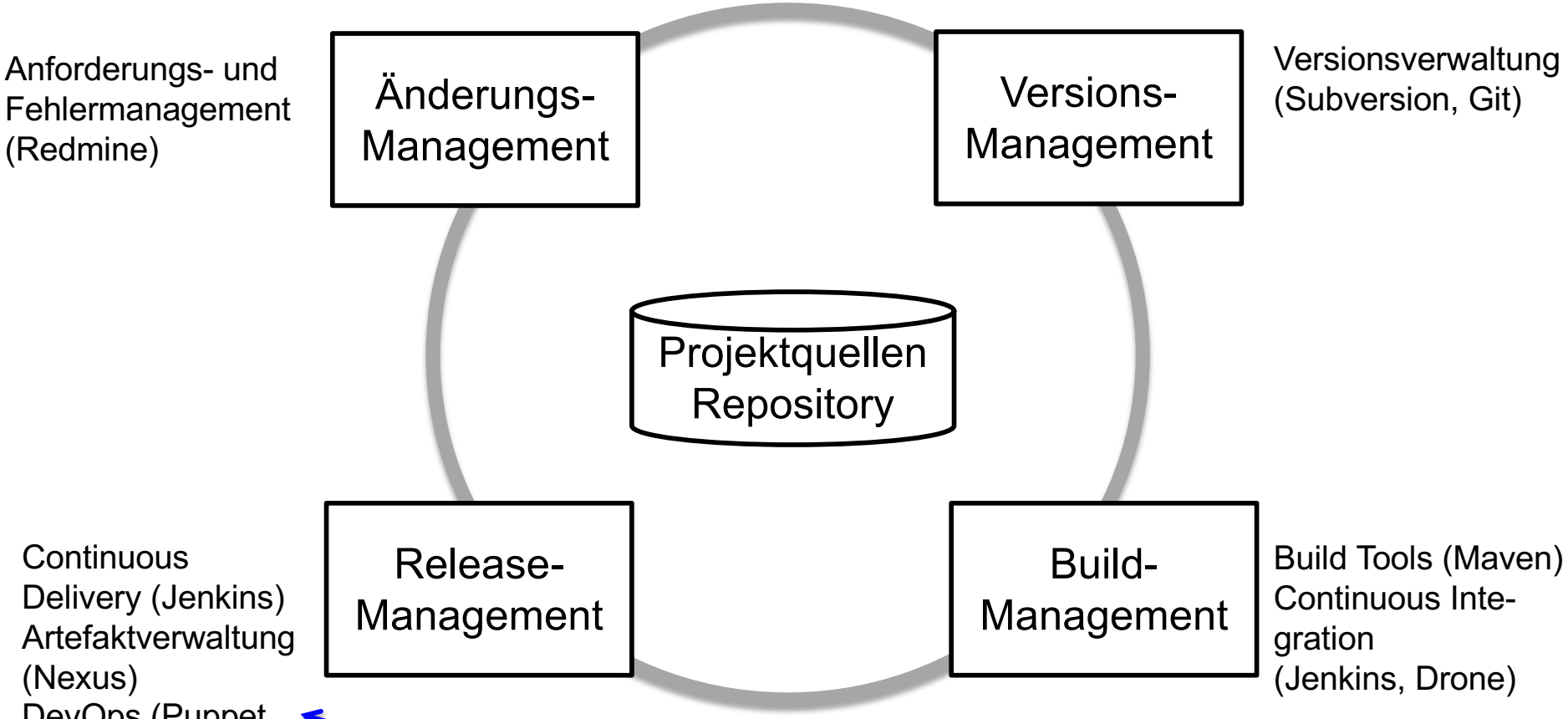
- Damit ergeben sich negative Auswirkungen auf die Qualität und Produktivität durch
  - Unkontrollierte Aktivitäten
  - Willkürliche Änderungen
  - Mangelnde Transparenz
  - Fehlende Nachvollziehbarkeit
  - unzureichende Automatisierung

- Konfigurationsmanagement
  - Ein sich änderndes Softwaresystem wird mit Hilfe von Richtlinien, Prozessen und Werkzeugen aktiv verwaltet
  - Das Konfigurationsmanagement ist eine konstruktive Maßnahme der Qualitätssicherung (Prozessqualität)

- Ein Konfigurationsmanagement stellt sicher, dass
  - alte Versionsstände wieder hergestellt werden können
  - jede Änderung nachvollzogen werden kann (Wer, Wann, Was, Warum)
  - nachvollzogen werden kann, welches Release wohin ausgeliefert wurde
  - der Zustand eines Änderungsauftrags stets eingesehen werden kann
  - die Teamarbeit koordiniert erfolgen kann
  - keine falsche Version ausgeliefert wird
  - ein hoher Automatisierungsgrad bei einem Auslieferungsprozess erreicht wird

– Das Konfigurationsmanagement umfasst vier miteinander verbundene Bereiche und erfordert geeignete Prozesse und den Einsatz unterschiedlicher Werkzeuge

hier: pragmatisch, mit Open-Source Werkzeugen



Wartung

Wartung

- Die Wartung bezieht sich auf den Zeitraum, in dem die Software beim Kunden in der produktiven Umgebung läuft

> *operation and maintenance phase – The period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements*

> *maintenance – The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment*

IEEE Std 610.12 (1990)

- Diese Definitionen bieten aber keine exakte Abgrenzung zu den Aktivitäten bei einer inkrementellen Software-Entwicklung

- Während bei einer inkrementellen Entwicklung die Tätigkeiten geplant sind, so zeichnet sich die Wartung durch einen hohen Anteil an nicht planbaren Aktivitäten aus
- Dies führt zu der folgenden Definition der Wartung [LL10]

> *Software-Wartung ist die Arbeit an einem bestehenden Software-System, die nicht von Beginn der Entwicklung an geplant war oder hätte geplant werden können, und die unmittelbaren Auswirkungen auf den Benutzer der Software hat*

Zur Abgrenzung vom Refactoring

- Die Bedeutung der Wartung ergibt sich durch ihren hohen Kostenanteil
- Fast alle Studien beziffern die Wartungskosten auf > 50 Prozent der Gesamtkosten einer Software (siehe [LL10])

- Es werden mehrere Arten der Wartung unterschieden (aus [LL10]):
  - Adaptive Wartung (Anpassung): Die Software wird so verändert, dass sie neue oder geänderte Anforderungen erfüllt; diese können funktional und nichtfunktional sein
  - Korrektive Wartung (Korrektur): Die Software wird so verändert, dass ein beobachteter Fehler nicht mehr auftritt

Es wurden noch weitere Arten der Wartung definiert. Diese zusätzlichen Definitionen haben sich aber als nicht hilfreich erwiesen

- Eingriffe in ein bestehendes Softwaresystem sind sehr fehleranfällig
- Die Wartung muss daher systematisch erfolgen
  1. Der Ausgangszustand wird präzise dokumentiert und archiviert
     ← eigentlich eine Grundvoraussetzung
  2. Der zu modifizierende Teil wird identifiziert
     ← wird durch eine geeignete Architektur erleichtert
  3. Testfälle für die durchzuführenden Änderungen werden systematisch entworfen
     ← die bereits bestehenden Testfälle bleiben für den Regressionstest bestehen
  4. Die zu ändernden Komponenten werden aus der Konfigurationsverwaltung entnommen, bearbeitet und lokal geprüft
     z.B. durch Modultest, Review oder Metriken →
  5. Die veränderten Komponenten werden integriert und getestet

Abweichungen beim Regressionstest müssen durch Spezifikation begründet sein ↗

  6. Der neue Zustand wird dokumentiert und archiviert
  7. Der benötigte Aufwand wird erfasst

- Im Rahmen der Konfigurationsverwaltung müssen Änderungen systematisch verwaltet werden (Change Management)
- Am Anfang einer jeden Änderung im Rahmen der Wartung steht eine Problemmeldung (Software Problem Report, SPR)
- Die Problemmeldung muss klassifiziert werden
  - Irrtum (z.B. falsche Verwendung der Software)
  - Fehlermeldung (bekannt, unbekannt)
  - Änderungswunsch
- Eine Problemmeldung kann dann eine Änderung der Software erfordern
- Eine Änderung erfordert einen entsprechenden Änderungsantrag (Change Request, CR)

- Die Verwaltung von Problemmeldungen und Änderungsanträgen erfordert den Einsatz von Werkzeugen

z.B. Bugzilla oder Trac

- Der Umgang mit Änderungs- und Erweiterungswünschen hängt von der Organisation ab
  - Externe Kunden: Abschluss von Wartungsverträgen oder neuer Vertrag pro Änderung (Budget muss vorhanden sein)
  - EDV-Projekt: Ein Gremium (*Change Control Board*, CCB) entscheidet

- Bearbeitung von Problemmeldungen
  1. Problemmeldung erfassen (inkl. Kennung und Datierung)
  2. Problemmeldung analysieren
     - Irrtum und Fehlbedienung ausschließen
     - Auswirkung eines Fehlers bzw. Nutzen einer Anpassung bestimmen
  3. Entscheidung des CCB vorbereiten
     - Finanzierung klären
  4. Absender über die Entscheidung des CCB informieren
  5. Änderungen durchführen und prüfen
  6. Problemmeldung schließen

- Um Wartungsaktivitäten effizient und effektiv durchführen zu können müssen die folgenden Fragen schnell beantwortet werden können
  a) Liegt aktuell ein Problem (Defekt, Ausfall, Störung) vor?
  b) Was ist die Ursache eines Problems?
  c) Wie wird die Anwendung genutzt? Gibt es Verbesserungspotenzial?

- Wenn das Programm geeignete Kontrollausgaben erzeugt, wird die Beantwortung der Fragen erleichtert (oder gar erst ermöglicht)

Logging

- Die Erzeugung von Kontrollausgaben (die für den Anwender nicht direkt ersichtlich sind) wird als *Logging* bezeichnet
  - Die Kontrollausgaben müssen persistent gespeichert werden, um eine spätere Analyse zu ermöglichen
    - Die Verarbeitungshistorie kann zudem wichtige Hinweise auf die Ursachen eines Problems liefern

- Das Schreiben/Speichern von *Logging*-Informationen benötigt Ressourcen
  - Rechenzeit
  - Speicherplatz
- Die *Logging*-Informationen werden daher nach ihrer Wichtigkeit klassifiziert
  - Jeder Meldung wird ein *Log-Level* zugeordnet
  - Nur Meldungen ab einem bestimmten *Log-Level* werden ausgegeben/gespeichert
- Während der Programmlaufzeit muss es möglich sein, zu bestimmen, welcher *Log-Level* mindestens für eine Meldung erforderlich ist
  - z.B. über Konfigurationsdateien, die in regelmäßigen Abständen eingelesen werden

– Beispiel: Die *Log-Level* der *Logging-Frameworks log4j* (`DEBUG` hat die geringste Wichtigkeit)

Log-Level	Bedeutung
`DEBUG`	Informationen zum Aufspüren von Fehlerursachen
`INFO`	Protokollierung des Programmflusses
`WARN`	Betriebssituation weicht vom erwarteten Verhalten ab, wird aber kontrolliert
`ERROR`	Programmablauf wird gestört (z.B. unvollständig behandelte `Exception`)
`FATAL`	Schwerwiegender Fehler. Programm wird beendet

- Eine *Logging*-Meldung sollte zumindest die folgenden Informationen enthalten
  1) Datum der Meldung
  2) Uhrzeit der Meldung
  3) Log-Level der Meldung
  4) Thread / Prozess
  5) Kategorie (z.B. Komponentenname, Klassenname)
  6) Zusätzliche Informationen (Freitext)

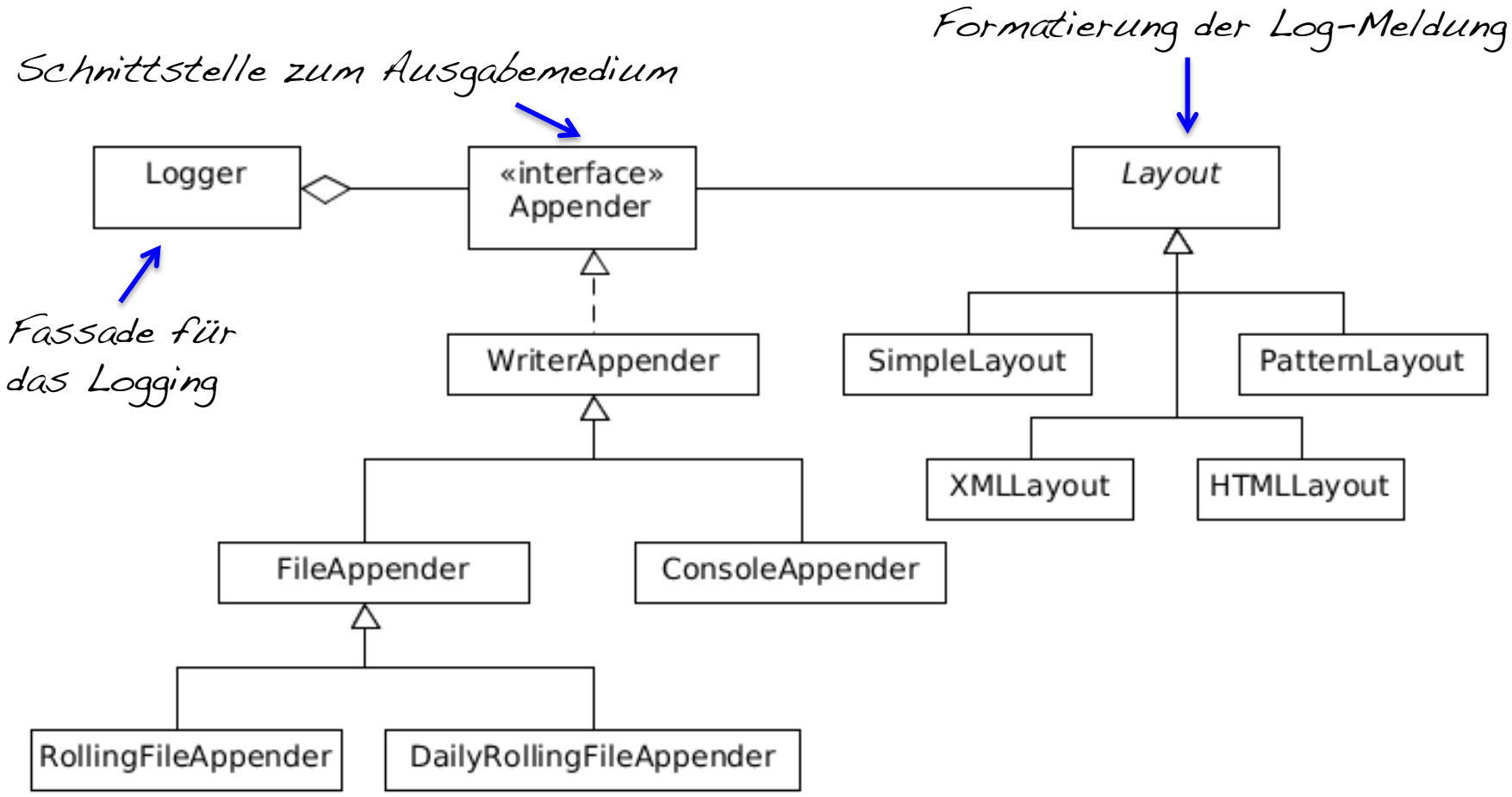
```
2015.11.25-15:08:54 WARN  [main] de.fhdortmund.swtd.Util:
istErstesHalbjahr wurde mit dem ungültigen Monat 0 aufgerufen.
```

Achtung: Rechtliche Aspekte sind zu beachten. Nicht alle Daten dürfen einer Auswertung zugänglich gemacht werden

- Als Beispiel betrachten wird das *Logging-Framework log4j*
  - Die zugehörige `jar`-Datei muss sich im Klassenpfad des Projekts befinden

```
<dependency>
   <groupId>log4j</groupId>
   <artifactId>log4j</artifactId>
   <version>1.2.17</version>
</dependency>
```

– Schnittstellen und Klassen aus dem Paket `org.apache.log4j`



a) Logging mit einer Grundkonfiguration

```
public class Util {
   private static Logger LOGGER = Logger.getLogger(Util.class);

   public static boolean istErstesHalbjahr(int monat) {
      LOGGER.info("istErstesHalbjahr(" + monat + ") aufgerufen.");
      if ((monat < 1) || (monat > 12)) {
         LOGGER.warn("istErstesHalbjahr mit ungültigem Monat");
         throw new IllegalArgumentException();
      }
      if (monat <= 6) return true;
      return false;
   }
   public static void main(String[] args) {
      BasicConfigurator.configure();
      if (istErstesHalbjahr(0)) { System.out.println("Ja"); }
    }
}
```

Kategorie (Logger-Name): hier Klassenname

Grundkonfiguration:
ConsoleAppender (Ausgabe über System.out)
+ einfaches PatternLayout

b) Programmatische Konfiguration eines *Loggers*

1) *Layout* erzeugen
2) *Appender* erzeugen und mit *Layout* versehen
3) *Appender* zum *Logger* hinzufügen
4) *Log-Level* einstellen

```
final Layout layout =
  new PatternLayout("%d{YYYY.MM.dd-HH:mm:ss} %-5p [%t] %c: %m%n");
try {
   final RollingFileAppender rfa =
               new RollingFileAppender(layout, "./app.log", true);
   rfa.setMaxFileSize("1MB");
   LOGGER.addAppender(rfa);
} catch (IOException e) {
   e.printStackTrace();
}
LOGGER.setLevel(Level.DEBUG);
```

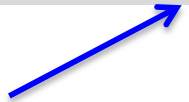
c) Verwendung einer Konfigurationsdatei (hier: `log4j.properties`)

```
log4j.rootLogger=INFO, MyAppender

log4j.appender.MyAppender=org.apache.log4j.RollingFileAppender
log4j.appender.MyAppender.File = app.log
log4j.appender.MyAppender.MaxFileSize = 1000KB
log4j.appender.MyAppender.layout =org.apache.log4j.PatternLayout
log4j.appender.MyAppender.layout.ConversionPattern=%d{YYYY.MM.dd-
HH:mm:ss} %-5p [%t] %c: %m%n
```

log4j.properties

- Einlesen der Konfiguration

```
PropertyConfigurator.configureAndWatch("log4j.properties", 5000);
```



Zeitintervall für das Prüfen auf Änderungen in der Konfiguration

- Einstellen von *Log-Level* auf Paket- und Klassenebene

```
log4j.logger.<paket>.<klasse>=<log-level>
```

- Abfrage der *Log-Level*

Wann ist eine solche Abfrage sinnvoll?

```
if (LOGGER.isInfoEnabled()){
   LOGGER.info("Aufruf der Service-Methode");
}
```

- Logging von Exceptions (*Stacktrace*)

```
try{
   writeBufferToFile(b);
} catch (IOException e){
   LOGGER.warn("Fehler bei der Ausgabe", e);
}
```