Einführung in Software Engineering

Barbara Paech, Marcus Seiler

Institute of Computer Science
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
http://se.ifi.uni-heidelberg.de
paech@informatik.uni-heidelberg.de

Kernfragen der Softwareentwicklung

- **Qualität:** Wie stellen wir sicher, dass das Softwaresystem **tut, was es soll?**
- **Beteiligte:** Wie erreichen wir, dass das Softwareprodukt **für die NutzerInnen nützlich** ist?

- **Beteiligte:** Wie erreichen wir, dass das Softwaresystem **effizient** zu entwickeln und insbesondere **für neue EntwicklerInnen verständlich und langfristig weiterzuentwickeln** ist?

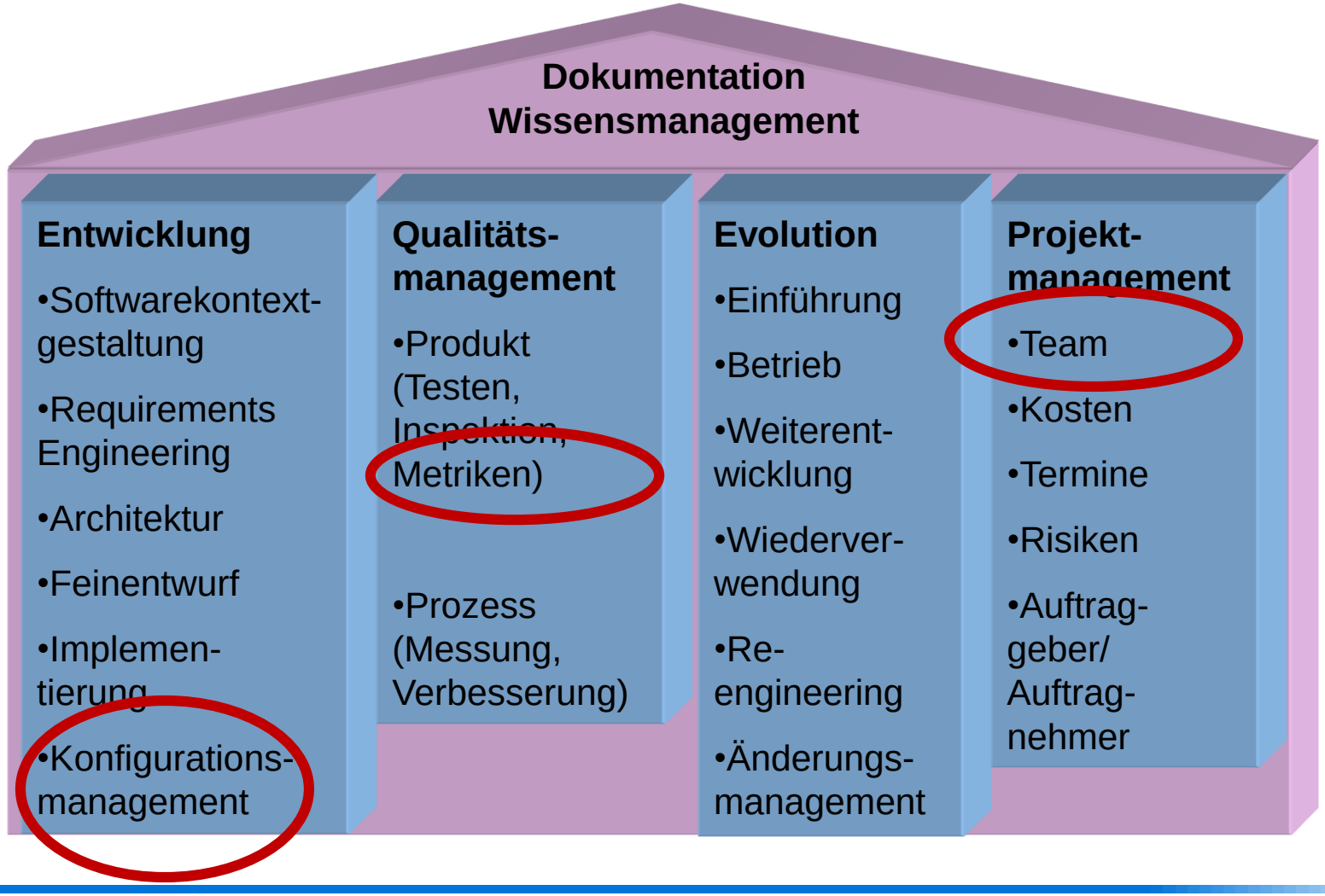
- **Kosten/Zeit:** Wie stellen wir sicher, dass das Softwaresystem mit den **vorgegebenen Ressourcen** (Geld, Technologie, Leute) **im Zeit- und Kostenrahmen fertiggestellt** wird?

Kapitel 2:
SWE im Kleinen

2. SWE im Kleinen (1. Teil)

- 2.1. Vorgehen
  - Prozess
  - Agiler Prozess
- 2.2. Einführung Qualitätssicherung
  - Motivation
  - Begriffe
- 2.3. Organisatorische Qualitätssicherung
  - Konfigurationsmanagement
  - Paarprogrammierung
- 2.4. Statische Prüfung
  - Einführung
  - Metriken

Aufgabenbereiche des Software Engineering



2.1. Vorgehen

Prozess

Agiler Prozess

Entdeckung der Software-Prozesse (1)

- Eine triviale Aufgabe (z.B. ein Geschenk einpacken) ist i.A. schnell und leicht lösbar; sie wird nicht weiter zergliedert und beschrieben.

- In der Frühzeit der Informatik wurde die Programmierung der Rechner als eine solche für Fachleute triviale Aufgabe betrachtet, Software = Programm.

- Software-Entwicklung:
  - Kopf → Papier (coding sheet) → Lochkarten
- Fehler dabei wurden als unvermeidlich und beherrschbar beurteilt.

Entdeckung der Software-Prozesse (2)

- **Ab den 60er Jahren: Systematik und Dokumentation** sollten dafür sorgen, dass mit akzeptablem Aufwand gute, fehlerarme Software entsteht.
- Typische Sichtweise: **Höhere Programmiersprachen** (Fortran, Algol, Cobol, Pl/I) und einige elementare **Programmierregeln** („structured programming“) lösen alle Probleme.
- Aber: Das funktioniert nur für einzelne EntwicklerInnen, nicht für Projekte mit mehreren beteiligten Personen.
- Neben dem **Programmieren im Kleinen** (programming in the small) gewann das **Programmieren im Großen** (programming in the large) an Bedeutung und Beachtung.
- Für die erforderlichen Schritte wurden **Vorgehens- oder Prozessmodelle** formuliert und eingeführt.
- In den Achtzigerjahren begann man schließlich, die **Qualität der Prozesse** systematisch zu **beurteilen** und zu **verbessern**.

Begriffsdefinition Prozess

- Prozess
  - Reihe von Aktivitäten, die einen gewissen Zusammenhang aufweisen und deren Ziel innerhalb einer Organisation grundsätzlich vereinbart wurde

Begriffsdefinition Projekt

DIN 69901 Projektmanagement – Projektmanagementsystem (2009)

- Projekt
  - Vorhaben, das im Wesentlichen durch **Einmaligkeit** der **Bedingungen** in ihrer Gesamtheit gekennzeichnet ist, wie z.B.
    - Zielvorgabe
    - Zeitliche, finanzielle oder andere Begrenzungen
    - Abgrenzungen gegenüber anderen Vorhaben
    - Projektspezifische Organisation, Prozess

PMBOK (http://www.pmi.org)

- Project
  - A temporary endeavor undertaken to create a unique product, service, or result

Prozess und Projekt

Projekt und **Prozess** stehen im gleichen Verhältnis wie

- Gegenstand und Modell
- Aufführung und Theaterstück
- Programmausführung und Programm
- Objekt und Klasse

- Das bedeutet: Ein Projekt ist einmalig, ein Prozess ist ein Schema.

Vorgehensmodell und Prozessmodell (1)

- Jedem Projekt liegt unvermeidlich ein Prozess zu Grunde, also eine **Vorstellung, wie ein Projekt ablaufen soll und kann**.
- Oft ist dieser Prozess weder bewusst gewählt noch niedergeschrieben oder irgendwie explizit formuliert. Das Projekt wird so durchgeführt, wie Projekte immer durchgeführt werden.
- Wenn **ausdrücklich vorgegeben** ist, wie Projekte ablaufen sollen, dann haben wir ein Vorgehens- oder Prozessmodell vor uns.

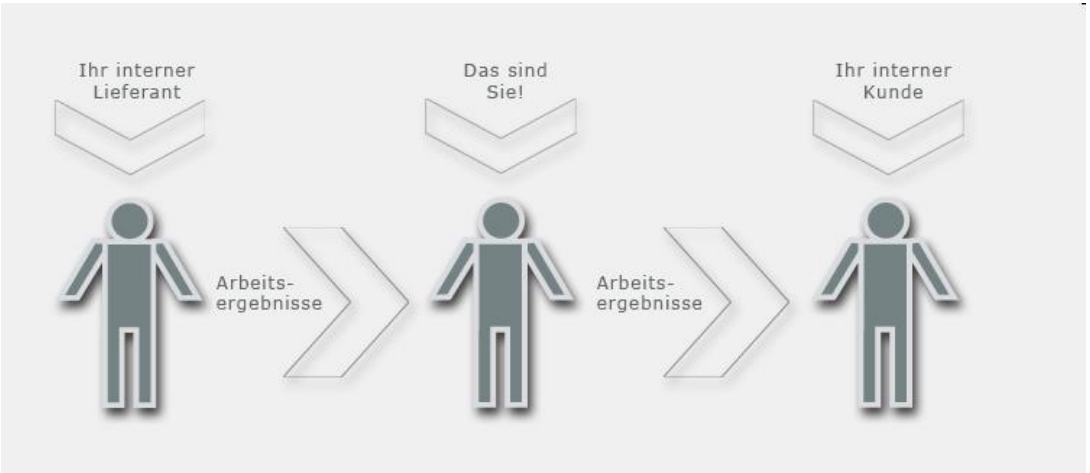
Vorgehensmodell und Prozessmodell (2)

- Vorgehensmodell:
  - Beschreibt die Maßnahmen für die Produkterstellung

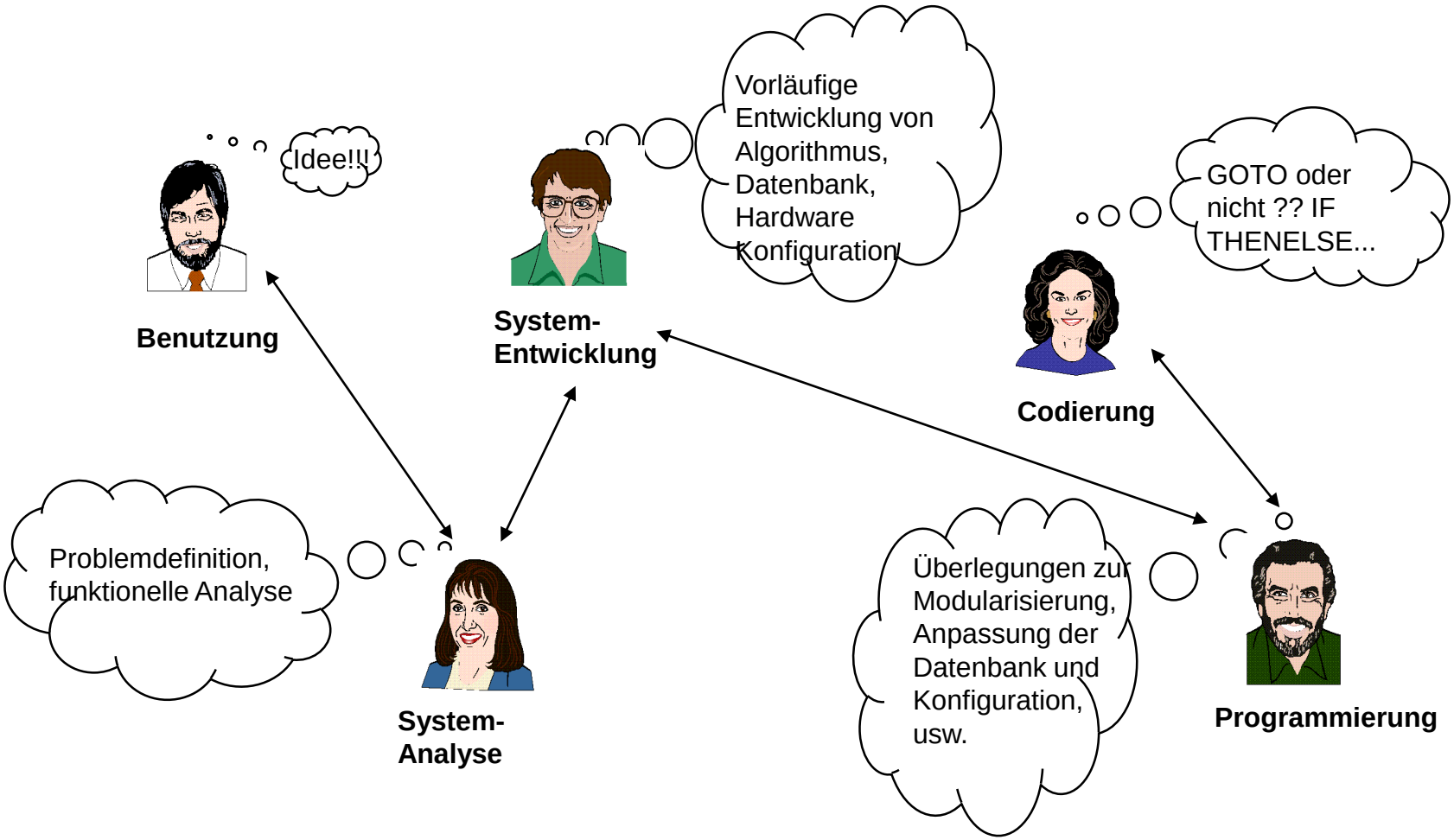
- **Differenzierung Vorgehensmodell / Prozessmodell**:
- „Prozess“ schließt mehr Aspekte ein als das „Vorgehen“.
  - Entsprechend gehören zum Prozessmodell neben einem Vorgehensmodell (das seinen Kern bildet) auch
    - die **Organisationsstrukturen**,
    - die Vorgaben für **Projektmanagement** und **Qualitätssicherung**,
    - die **Dokumentation** und die **Konfigurationsverwaltung**.

Beschreibung eines Prozesses

- Akteure (WER)
- Aktivitäten (WAS)
- Ergebnisse (WAS)
- Richtlinien (WIE)
- Rahmenbedingungen (WIE)

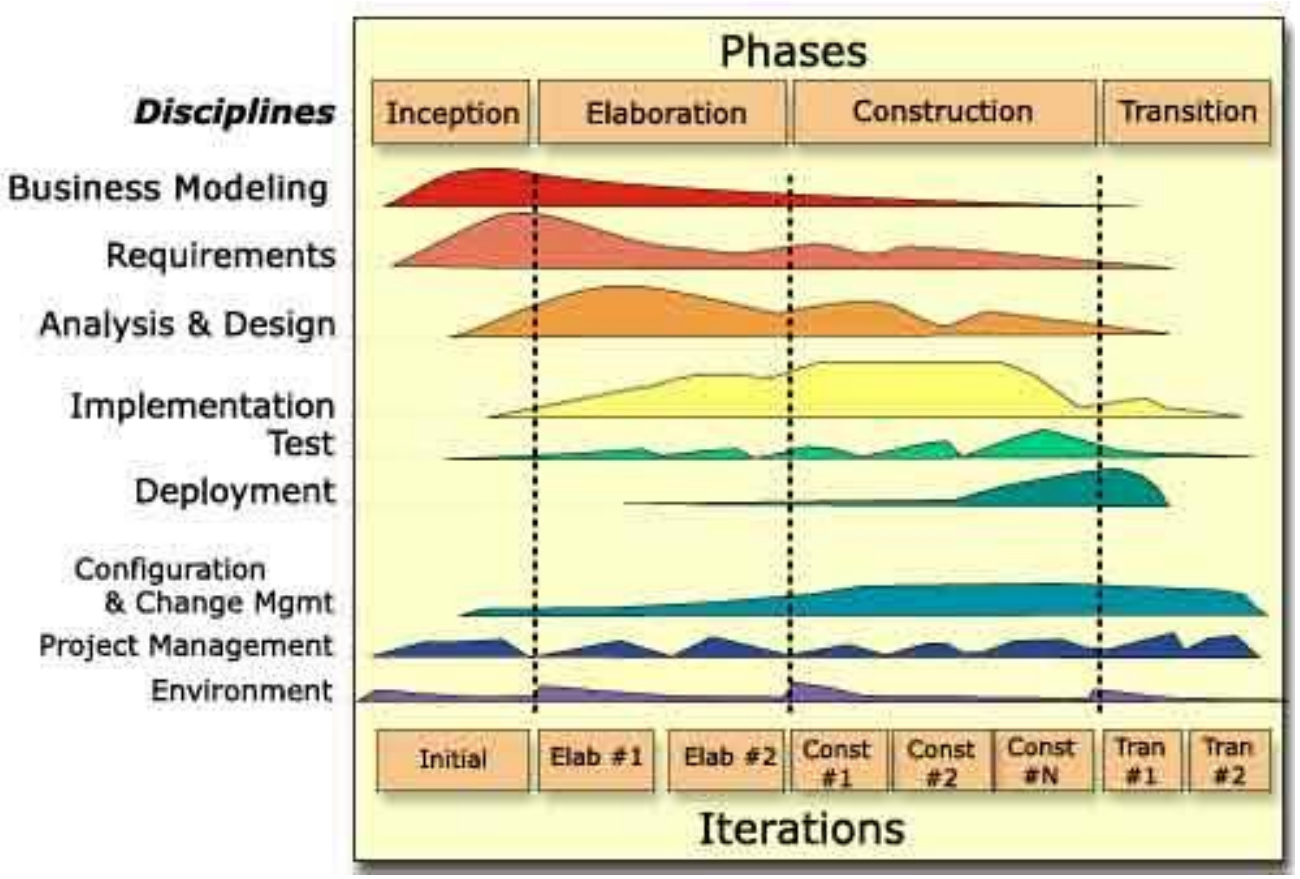


SWE-Vorgehen früher: Wasserfall



[Yourdon/Constantine 1979]

SWE-Vorgehen heute: Rational Unified Process



http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf /

Warum sind Prozessmodelle wichtig?

- Ohne Prozess ist kein gemeinsames Verständnis der Aufgaben möglich.
- Ohne Prozess ist keine Vorhersage (Planung) über das Projektergebnis möglich.
- Prozesse sind eine Form der Erfahrungssammlung. Aufbauend auf Prozessen ist Lernen in einer Organisation möglich.
- Prozesse geben Sicherheit (nichts Wesentliches zu übersehen).

2.1. Vorgehen

Prozess

Agiler Prozess

Beispiel Agiler Prozess: XP

- 1999:
  Buch von Kent Beck
- XP ist der bekannteste der „agilen Prozesse“

- Agile Prozess werden vor allem in kleinen Projekten angewendet

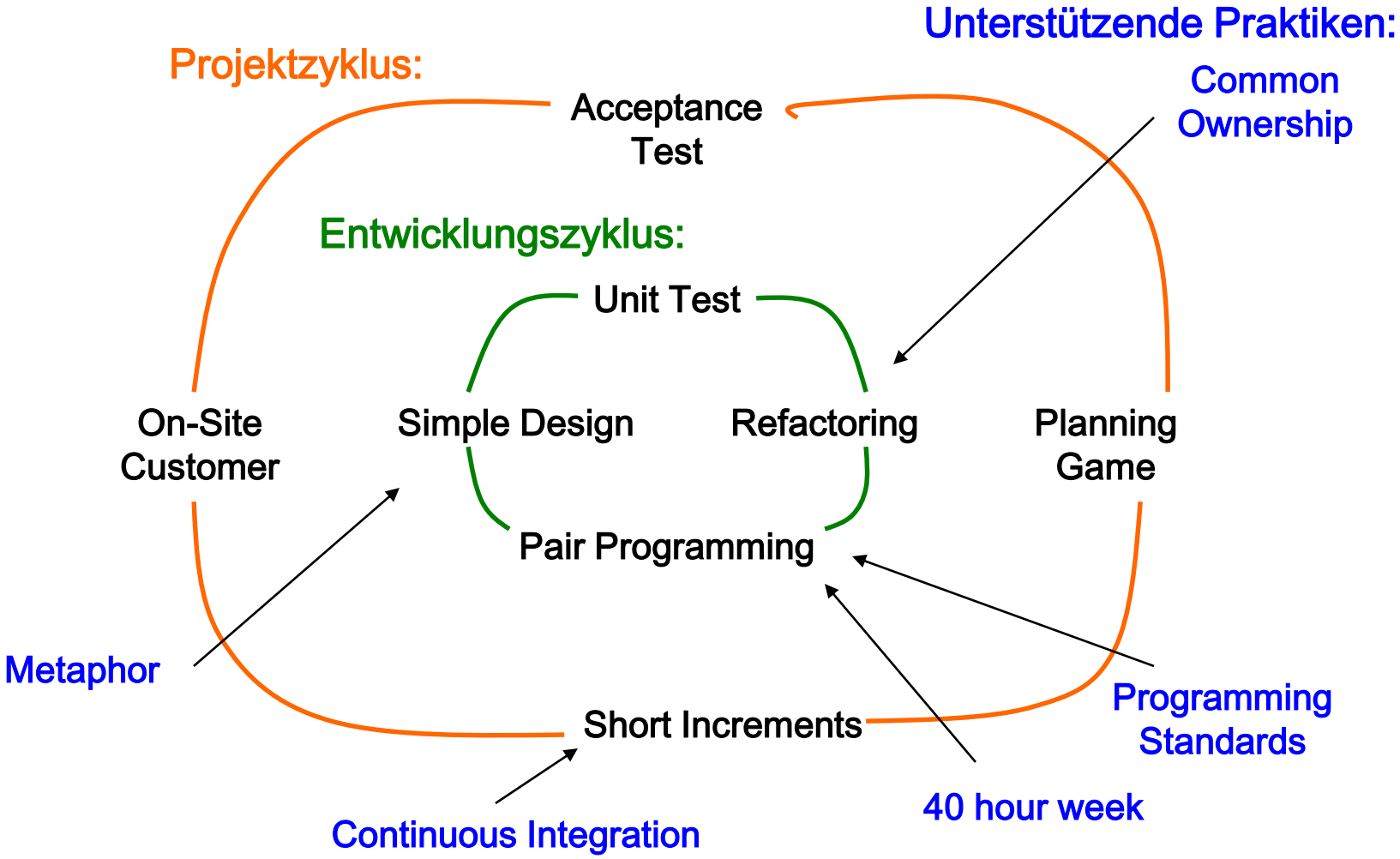
XP - 5 Grundprinzipien

1. Schnelle Rückmeldung
   - Der Lernprozess in einem Softwareprojekt hängt von der Zeit zwischen den einzelnen Aktionen ab.
2. Gradliniges Denken
   - Oft reicht eine einfache Lösung aus
3. Inkrementelle Änderungen
   - Ändere nicht alles auf einmal, sondern immer in kleinen Schritten
4. Änderungen erwarten und aufnehmen
   - Die Angst vor Änderungen ist kontraproduktiv – die Kosten des Festhaltens an einer schlechten Lösung sind höher
5. Qualitätsarbeit
   - Die Qualität bringt Flexibilität auf Änderungen schnell zu reagieren.

Zuordnung zu Kernfragen

	Hohe Qualität	Zufriedene NutzerInnen	Wartbarkeit (Entwick-lerInnen)	Kosten/ Zeit
Schnelle Rückmeldung		X		(x)
Gradliniges Denken			X	(x)
Inkrementelle Änderung			X	(x)
Änderung erwarten			X	(x)
Qualitätsarbeit	X			(x)

Übersicht: XP Praktiken (1)



Übersicht: XP Praktiken (2)

- **On-site Customer**: Ein kompetenter Vertreter der Kundenseite während der gesamten Entwicklungszeit bei den EntwicklerInnen
- **Short Increments**: häufige Iterationen mit lauffähigen Programmen als Ergebnis, welche der/die KundIn begutachtet
- **Planning Game:** Absprache zwischen EntwicklerInnen und KundIn für das nächste Inkrement (Abwägung Kosten/Nutzen)
  - **User Stories**: Anforderungen in User Stories erfasst, beschreiben GUIs, Funktionalitäten und Testszenarien.
- **Simple Design**: keine unnötigen Features implementiert
- **Refactoring**: Der Code wird eher früh restrukturiert (falls erforderlich).
- **Pair Programming**: jeweils zwei EntwicklerInnen entwickeln gemeinsam.
- **Unit Test**: Vor der Entwicklung einer Komponente werden automatisierbare Testfälle (Unit Tests) programmiert.
- **Continuous Integration**: Von verschiedenen Teammitgliedern produzierter Code wird sehr häufig zusammengeführt.
- **Collective Ownership**: Der entwickelte Code gehört dem gesamten Team, jeder ist für jeden Code verantwortlich. Die Teams rotieren zyklisch.
- **Coding Standards**: Konventionen zum Aufbau des Codes, um Lesbarkeit zu erleichtern.

Weiteres siehe www.extremeprogramming.org

Zuordnung zu Kernfragen

	Hohe Qualität	Zufriedene NutzerInnen	Wartbarkeit (EntwicklerInnen)	Kosten/ Zeit
On-site Customer		X		
Short increments				
Planning game				
Simple Design				
Refactoring				
Pair-programming				
Unit test				
Continuous integration				
Collective ownership				
Coding standards				

2.2. Einführung Qualitätssicherung

- Motivation
- Begriffe

Einführung Qualität

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

Anmerkung 2: *Ein Produkt ist z. B. jede Art von Waren, Rohstoffen, aber auch der Inhalt von Konzepten und Entwürfen. Eine Tätigkeit ist z. B. jede Art von Dienstleistung, aber auch ein maschineller Arbeitsablauf wie ein Verfahren oder ein Prozess.*

Anmerkung 4: *Die Qualität wird durch die Planungs- und Ausführungsqualitäten in allen Phasen des Qualitätskreises bestimmt.*

DIN 55350-11:1995-08 (Auszüge)

- Achtung: Erfordernisse (Anforderungen) **ändern** sich über die Zeit!
- Achtung: **Qualität** muss immer mit **Kosten und Zeit** in Bezug gesetzt werden

Qualität für große SW ist schwierig!

- Die Probleme, die durch die Größe entstehen, werden durch einen statistischen Effekt verschärft:
  - Während es bei einfachen Systemen aus wenigen Komponenten einfach ist, ihre Korrektheit zu überprüfen, wird das bei Systemen aus vielen Komponenten sehr viel schwieriger.

- Ein Software-System sei aus n Modulen zusammengesetzt.
  - Jedes Modul sei korrekt mit der Wahrscheinlichkeit p; n Module sind dann korrekt mit der Wahrscheinlichkeit p^n.
  - Für n = 100 können wir die folgenden Werte berechnen.

Wenn p	=	0,9	0,95	0,99	0,999
dann ist p^{100}	=	0,000 027	0,006	0,37	0,90

  - Für ein kleines Programm ist p = 0,9 ausreichend, bei einem System aus 100 Modulen muss dazu jedes Modul zu 99,9 % korrekt sein!

Berüchtigtes Beispiel: Ariane 501

- Folien dazu auszugsweise übernommen von Holger Schlingloff, Humboldt Universität Berlin

- Ariane5 als Nachfolgerin der Ariane4-Familie mit über 100 erfolgreichen Starts
  - Viel Code wiederverwendet

- 6-12t Nutzlast (gegenüber 2-5t A4)

- Jungfernflug am 4.6.1996

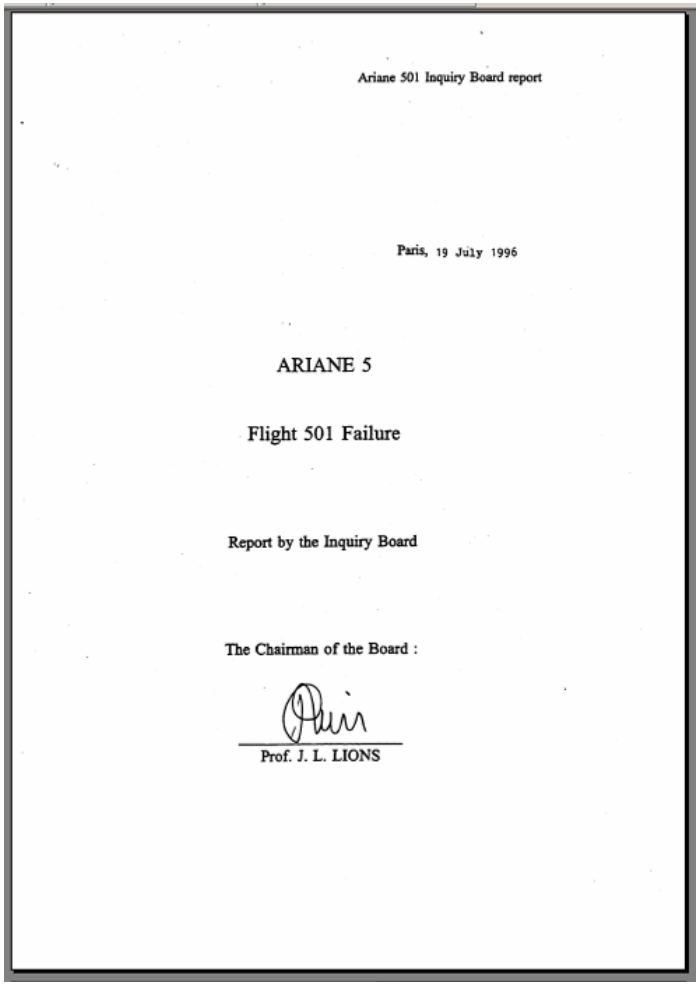
explosion.avi

Flugablauf

- **H_0-3 Sek.** Das Haupttriebwerk wird gezündet.
- **H_0 Sek.** Beide Booster werden gezündet, Start endgültig.
- **H_0+7,5 Sek.** Die Haltebolzen werden gelöst, A501 hebt ab.
- **H_0+37 Sek.** Die Rakete befindet sich in einer Höhe von 3,5km und hat eine Geschwindigkeit von 857 km/h, als plötzlich die Steuermotoren beide Boosterdüsen und das Haupttriebwerk bis zum Anschlag lenken.
- **H_0+39 Sek.** Die Lage der Rakete ist schräg zu ihrer Flugbahn. Durch die einwirkenden aerodynamischen
  Kräfte beginnt die Rakete auseinander zu brechen.
- **H_0+41 Sek.** Der automatische Selbstzerstörungsmechanismus wird ausgelöst und die Rakete planmäßig gesprengt.

„Ergebnis“

- Nach wenigen Sekunden sprengte sich die Rakete selbst
- Missionsverlust
  - Nutzlast zerstört, Kosten > 500 M€
  - Programm 3 Jahre aufgehalten
- Untersuchungskommission
  - Bericht vom 19.6.1996 (nach nur 14 Tagen !!!)
  - http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html



Ariane 501 Inquiry Board report

Paris, 19 July 1996

ARIANE 5

Flight 501 Failure

Report by the Inquiry Board

The Chairman of the Board :

Prof. J. L. LIONS

Fehlerursachen (1)

- Primäre Fehlerursache: Operandenfehler bei Konvertierung der Variablen horizontal_bias, Fehlen von Ausnahmebehandlungsroutinen (Programmierfehler).
- Nur für 4 von 7 Variablen waren die int- Umwandlungen geschützt, um die maximale Prozessorauslastung von 80% nicht zu überschreiten (Kostengründe).
- Für die 3 ungeschützten Variablen waren Wertebereiche angenommen worden, aber nicht dokumentiert (verteilte Verantwortlichkeit).
- Die Annahmen waren nicht anhand der geplanten Flugbahn verifizierbar, da diese nicht zur Anforderungsspezifikation gehörte (Management).

Fehlerursachen (2)

- Die Default-Einstellung bei einer Ausnahme war es, den Prozessor abzuschalten statt „so gut wie möglich“ weiterlaufen zu lassen. Die Fehlertoleranzmechanismen gingen ausschließlich von zufälligen (Hardware-), nicht von systematischen (Software-) Fehlern aus (Branchenkultur).
- Die Sensorkalibrierungsroutine wird nach dem Start nicht mehr benötigt und hätte ausgeschaltet werden können (nur bei Startabbruch der Ariane 4 sinnvoll, um Neukalibrierung zu vermeiden) (Wiederverwendung).
- Die Grundannahme, dass es nicht sinnvoll ist, funktionierende Software der Ariane 4 zu verändern, war falsch. Die Grundannahme, dass Software zuverlässig ist, falls keine Fehler offenbar sind, war falsch.
- Es gab keinen Review, bei dem die Entscheidung, den Systemtest auszulassen, kritisch hinterfragt worden wäre.

Fehler oft nicht schnell zu erkennen

September 2004: Drei Wochen Verspätung
Durch anhaltende Probleme mit der Computersoftware können die Arbeitsagenturen erst drei Wochen später als geplant die Antragsdaten für das Arbeitslosengeld II flächendeckend erfassen. Ursprünglich sollte das Programm schon ab 4. Oktober 2004 zur Verfügung stehen.
Mittlerweile haben sich die Bundesagentur für Arbeit (BA) und die Software-Entwickler der Telekom-Tochter T-Systems geeinigt: Das Programm soll stufenweise eingeführt werden. "Wir sind zurzeit in der Testphase, um Fehler zu ermitteln und zu beheben. Am 4. Oktober werden wir planmäßig starten und das System stufenweise hochfahren", sagte ein Sprecher von T-Systems im Gespräch mit **wdr.de** am Freitag (10.09.04). Das Computerprogramm werde Anfang Oktober in einigen Agenturen anlaufen, damit die BA Erfahrungen sammeln kann.

In einigen Agenturen wird das Programm getestet

September 2005: Durch einen Fehler in der Software A2LL kommt es derzeit zu falschen Krankenkassen-Meldungen, teilt die Bundesagentur für Arbeit (BA) mit. In mehreren hunderttausend Fällen seien Meldungen zur Krankenversicherung, also Anmeldungen, Abmeldungen, Veränderungsmitteilungen, von Arbeitslosengeld-II-Empfängern ohne Grund automatisch storniert worden.

http://www.heise.de/newsticker/meldung/62595

QS-Maßnahmen ARIANE 501 (1)

- **Planung / Monitoring**
  - **Risikomanagement:** Die *Risiken* erkennen, angemessene technische Maßnahmen planen, durchsetzen und überprüfen.
  - **Kostenmanagement:** Die Kosten einer vorbeugenden Maßnahme *in Relation* zu den Kosten eines Fehlers sehen

- **Wissensmanagement**
  - **Wiederverwendung:** Bestehende Software darf nicht unbesehen für eine neue Aufgabe *wiederverwendet* werden. Vorher muss geprüft werden, ob ihre Fähigkeiten den Anforderungen der neuen Aufgabe entsprechen.
  - **Spezifikation:** Die *Fähigkeiten* einer Software sowie alle *Annahmen*, die sie über ihre Umgebung macht, müssen sauber *spezifiziert* sein.
  - **Dokumentation:** Kooperieren zwei Software-Komponenten miteinander, so müssen eindeutige *Zusammenarbeitsregeln* definiert, dokumentiert und eingehalten werden: Wer liefert wem was unter welchen Bedingungen.

QS-Maßnahmen ARIANE 501 (2)

- **Produktprüfung /Richtlinien**
  - **Fehlerbehandlung:** Jede potentielle *Fehlersituation* in einer Software muss entweder *behandelt* werden oder die Gründe für die Nichtbehandlung müssen so *dokumentiert* werden, dass die Gültigkeit der dabei getroffenen Annahmen überprüfbar ist.
  - **Fehlertoleranz:** Mehrfache identische Auslegung von Systemen hilft nicht gegen Entwurfsfehler.
  - **Sicherer Zustand:** Bei Störungen in sicherheitskritischen Systemen ist *Abschalten* nur dann eine zulässige Maßnahme, wenn dadurch wieder ein sicherer Zustand erreicht wird.
  - **Systemtest:** Beim *Test* von Software, die aus mehreren Komponenten besteht, genügt es *nicht*, jede Komponente *nur isoliert* für sich zu testen. Umfangreiche Systemtests unter möglichst realistischen Bedingungen sind notwendig.
  - **Review:** Jedes Programm muss - neben einem sorgfältigen Test - durch kompetente Fachleute *inspiziert* werden, weil insbesondere die Erfüllbarkeit und Adäquatheit von Annahmen und Ergebnissen häufig nicht testbar ist.

- Achtung!
- Mit 

  gekennzeichnete Folien

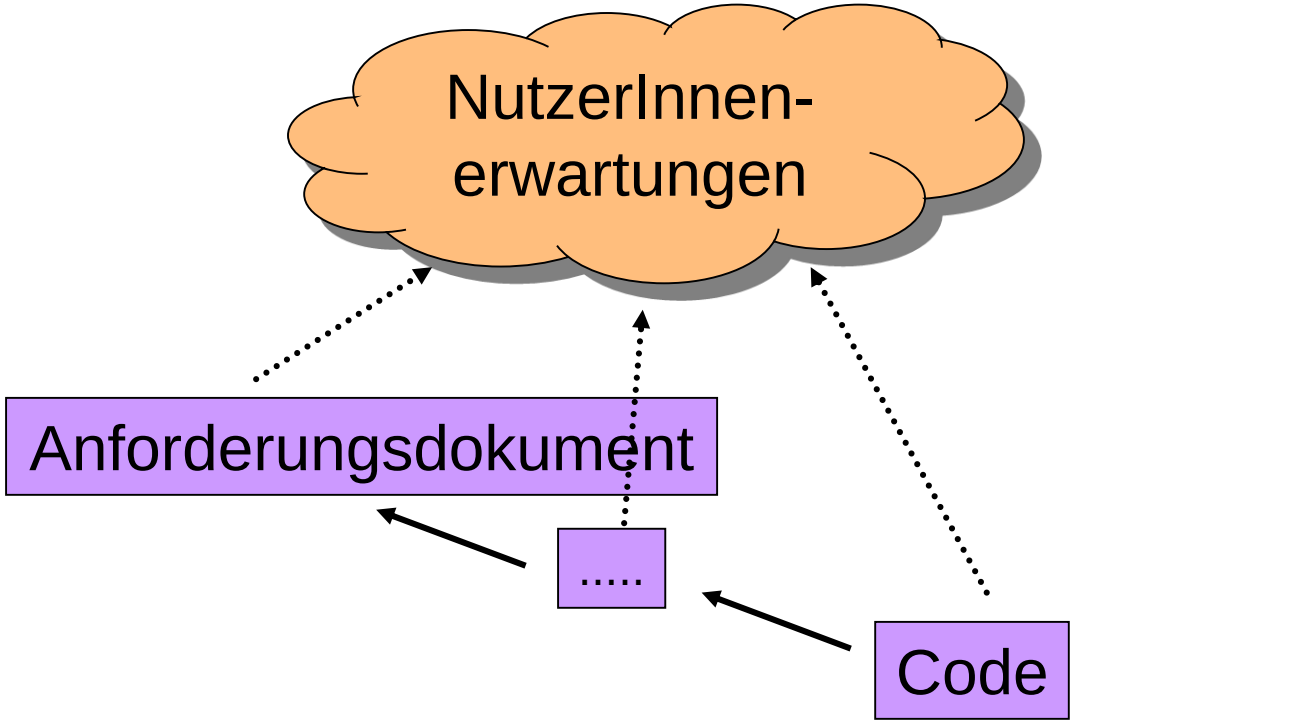
  dürfen AUF KEINEN FALL nach außen weitergegeben werden.

Näheres zu GTB unter http://www.german-testing-board.info/

2.2. Einführung Qualitätssicherung

- Motivation
- Begriffe

Validierung vs. Verifikation



Verifikation: haben wir es richtig gemacht?

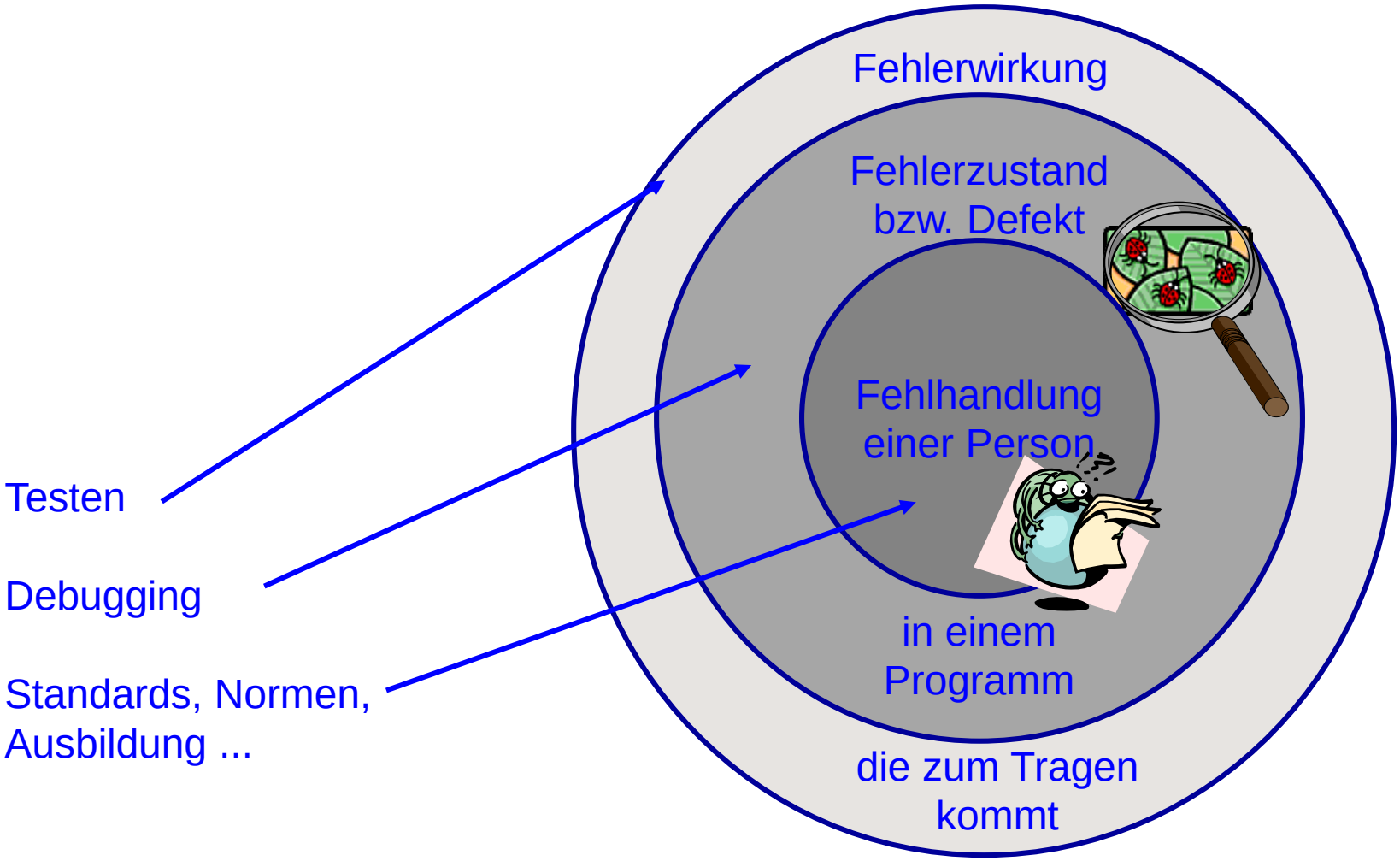
Validierung: haben wir das Richtige gemacht?

Was gilt als Fehler oder Mangel?

- Eine Situation kann nur dann als fehlerhaft eingestuft werden, wenn vorab festgelegt wurde, wie die erwartete, korrekte, also nicht fehlerhafte Situation aussehen soll.
- Ein Fehler
  ist somit die Nichterfüllung einer festgelegten Anforderung, eine Abweichung zwischen dem Ist-Verhalten (während der Ausführung der Tests oder des Betriebs festgestellt) und dem Soll-Verhalten (in der Spezifikation oder den Anforderungen festgelegt).
- Ein Mangel
  liegt vor, wenn eine gestellte Anforderung oder eine berechtigte Erwartung in Bezug auf einen beabsichtigten Gebrauch nicht angemessen erfüllt wird. Ein Mangel ist z.B. die Beeinträchtigung der Verwendbarkeit bei gleichzeitiger Erfüllung der Funktionalität oder die Nichterfüllung einer angemessenen Erwartung.

Angelehnt an: DIN EN ISO 9000:2005 Qualitätsmanagementsysteme – Grundlagen und Begriffe. Beuth Verlag, Berlin, 2005

Weitere Fehlerbegriffe



Fehlerzustand, Fehlerwirkung - Definition

- Fehlerzustand/Defekt (fault)
  1. Inkorrektes Teilprogramm z.B. mit inkorrekter Anweisung oder Datendefinition, die Ursache für eine Fehlerwirkung ist.
  2. Zustand eines (Software-)Produkts oder einer seiner Komponenten, der unter spezifischen Bedingungen (z.B. bei einer hohen Belastung) eine geforderte Funktion des Produkts beeinträchtigen kann bzw. zu einer Fehlerwirkung führt.

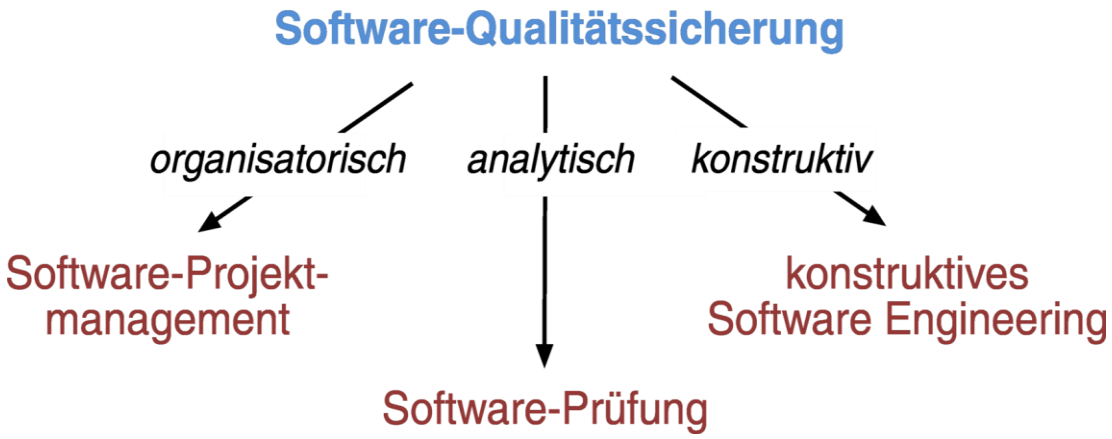
- Fehlerwirkung (failure)
  1. Wirkung eines Fehlerzustands, die bei der Ausführung eines Programms (Testobjekt) nach »außen« in Erscheinung tritt.
  2. Abweichung zwischen (spezifizierten) Soll-Wert und (beobachtetem) Ist-Wert (bzw. Soll- und Ist-Verhalten).
- Fehlerwirkungen können (selten) auch durch Höhenstrahlung, elektromagnetische Felder oder Hardwarefehler hervorgerufen werden.

Wie stellt man Qualität sicher?

- Qualitätsmanagement: etablierter Prozess zum Umgang mit Qualität
  - Wie kann man Qualität beschreiben (=> Kapitel zu Qualitätsanforderungen)
  - Wie kann man Qualität einer Software bewerten (=>Kapitel zu SW-Prüfung)
  - Wie kann man Qualität erreichen (=> gesamte Vorlesung)
  - Wie kann man Fehler/Mängel finden (=>Kapitel zu SW-Prüfung)
  - Wie kann man Fehler/Mängel beheben (=> Kapitel zu Umgang mit Fehlern)
  - Wie kann man Fehler/Mängel vermeiden (=> gesamte Vorlesung)

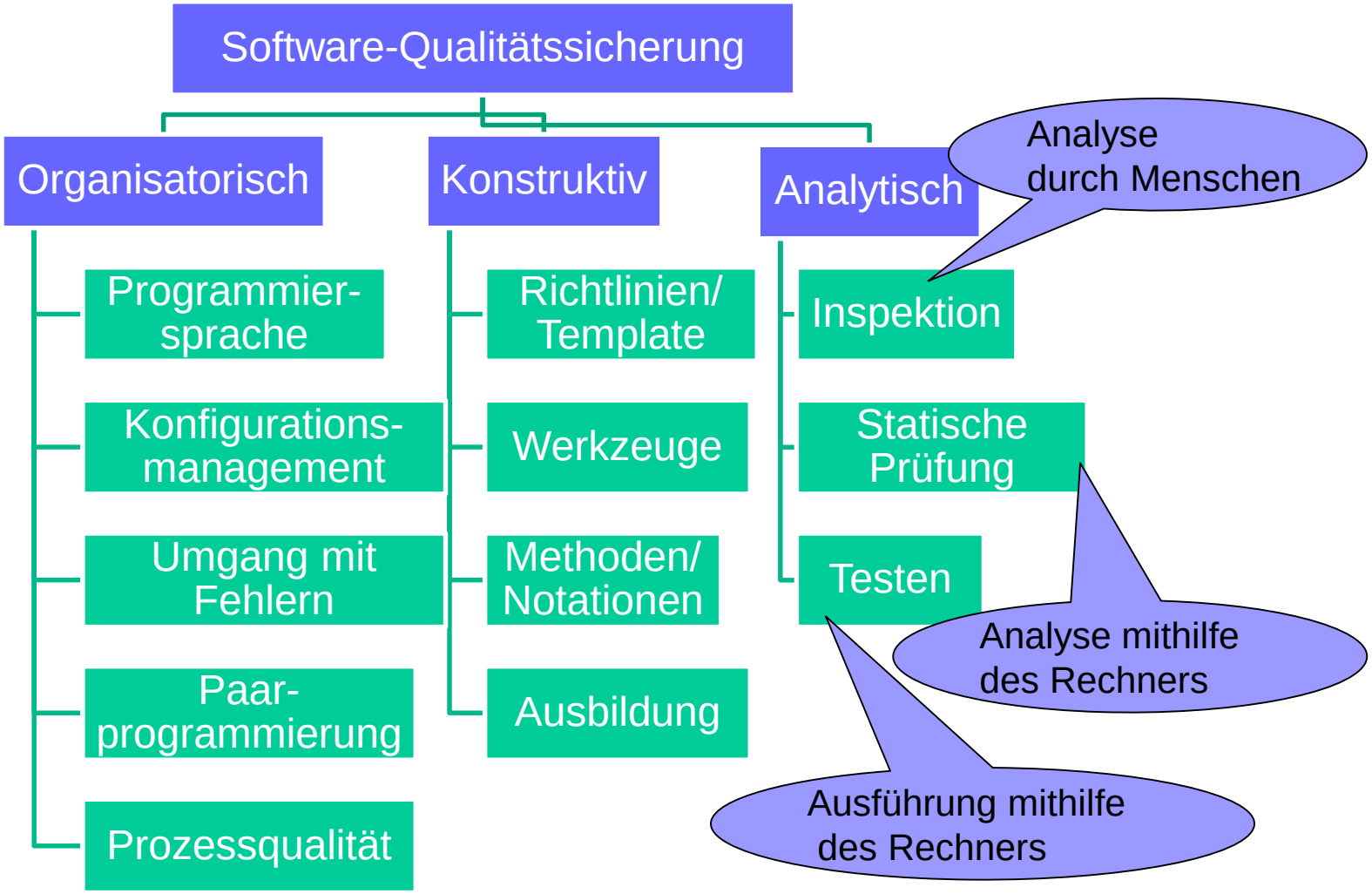
- Qualitätssicherung (QS): konkretes Vorgehen zur Sicherstellung der Qualität
  - Konstruktive QS
  - Analytische QS
  - Organisatorische QS

Schwerpunkte der SW-Qualitätssicherung



- Qualität lässt sich nicht in das Produkt „hinein prüfen“!
- Organisatorische und konstruktive Maßnahmen sind wesentlich wichtiger, sie werden durch analytische Maßnahmen ergänzt.

Typische Techniken der Qualitätssicherung



Organisatorische Qualitätssicherung

- Organisatorische QS versucht die Qualität durch geeignete Planung und Organisation sicherzustellen, insbesondere
  - Geeignete Festlegung von Verantwortlichen

- Nicht ganz klar von konstruktiver QS zu trennen

Konstruktive Qualitätssicherung

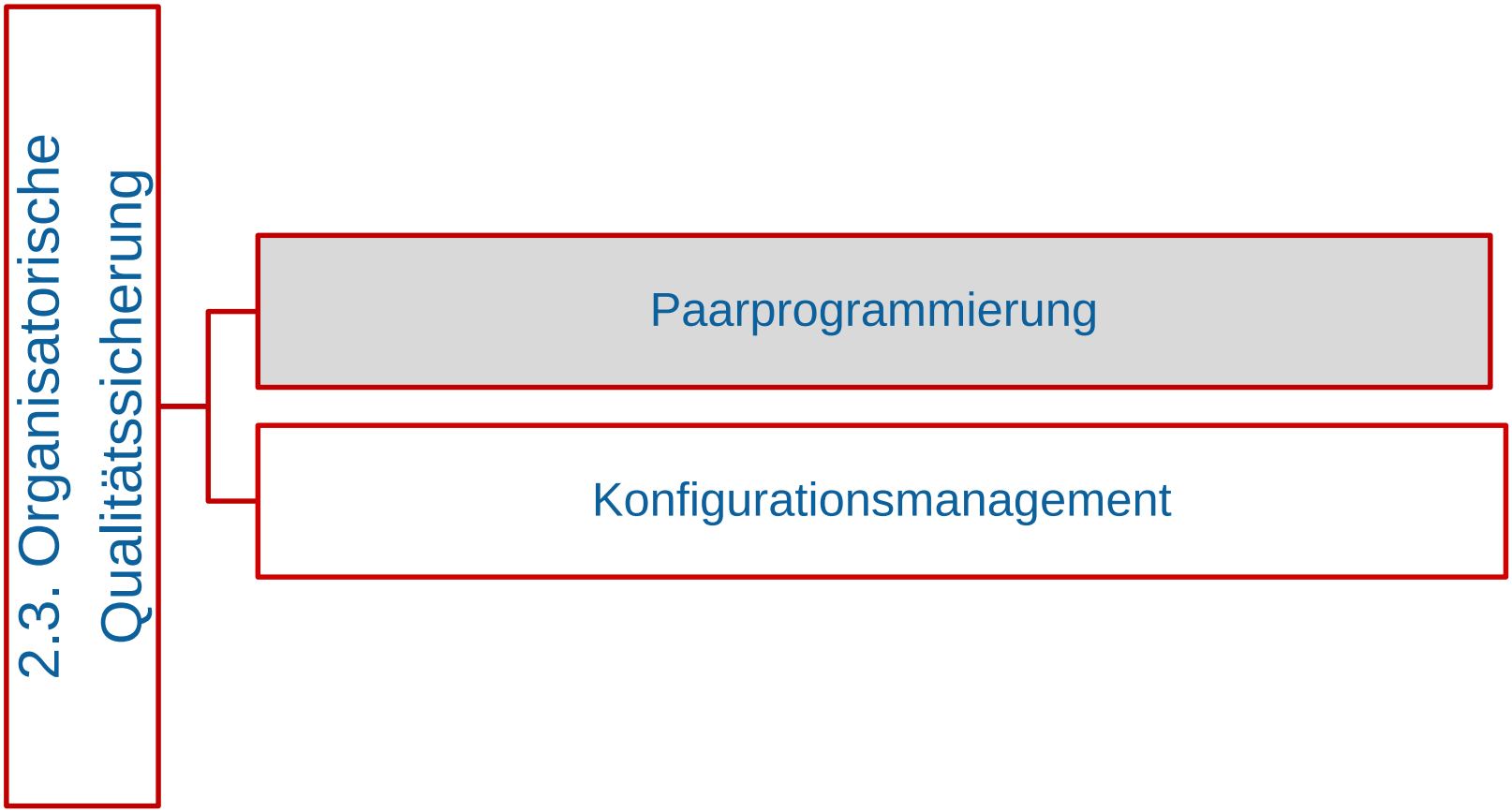
- Konstruktive QS versucht die Qualität durch systematische Entwicklung sicherzustellen
  - Richtlinien (insbesondere Programmierrichtlinien)
  - Vorlagen (Template)
  - Werkzeuge
  - Notationen
  - Methoden
  - Ausbildung

- Vorgaben entstehen meist aus Erfahrung mit analytischer QS

Analytische Qualitätssicherung

- Ziel der analytischen QS: Überprüfen, ob Produkt die Anforderungsspezifikation erfüllt

- Prüfmethoden:
  - **Beweis** (mathematischer Nachweis, nur bei einfachen Programmen händisch möglich, ansonsten komplexe Beweis-Werkzeuge)
  - **Test** (Ausprobieren für eine sorgfältig ausgewählte Menge von Eingaben)
  - **Inspektion / Review** (systematisches Durchlesen)
  - **Statische (Code-)Prüfung, insbes. Metriken** (automatische Bestimmung von Charakteristika)



Paarprogrammierung (Pair-Programming)

- Wichtige Praktik in XP
- Grundidee: kontinuierliche Inspektion (siehe analytische QS)
- 2 Leute implementieren zusammen
  - Eine Person ist „Driver“, also hat Tastatur und schreibt (und denkt dabei laut)
  - Eine Person ist „Observer“ und stellt Fragen
  - Die Leute wechseln oft ihre Rollen
  - Sie kommunizieren oft
- Für viele erstmal ungewohnt
- Ausführliche Diskussion unter http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF

Vorteile der Paarprogrammierung (1)

- Aussagen basieren auf empirischen Studien
- Kosten
  - Die zweite Person verursacht 15% Overhead (Gesamtentwicklungsaufwand), aber dafür 15% weniger Fehler
- Entwurfsqualität
  - Programme sind kürzer
- Zufriedenheit
  - Die meisten Leuten haben mehr Spaß zu zweit (nach einer Umgewöhnungszeit)
  - Schwierige Probleme lassen sich leichter lösen

Vorteile der Paarprogrammierung (2)

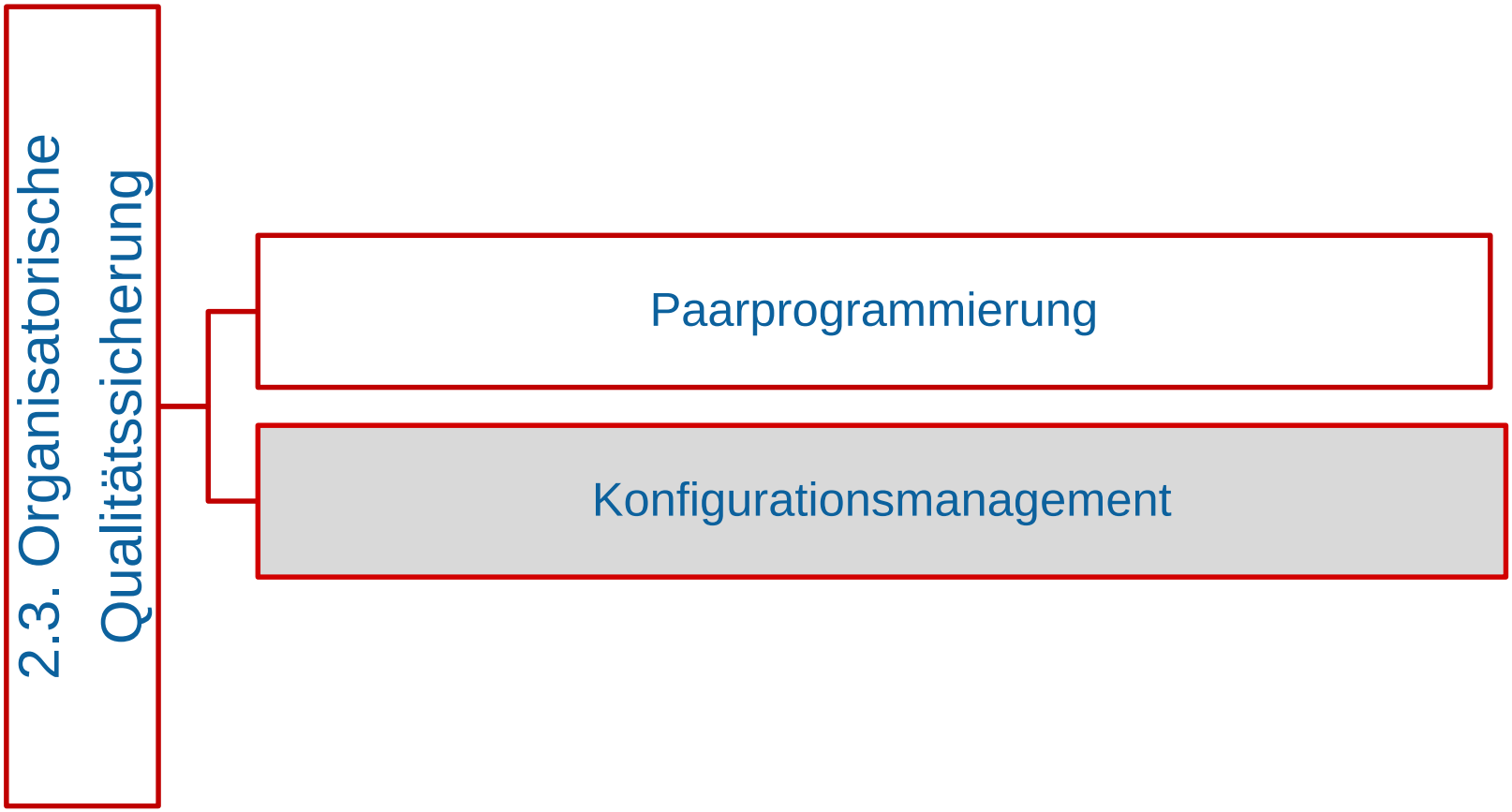
- Qualität des Projektteams
  - Die Personen lernen viel voneinander
  - Die Personen lernen gut zu kommunizieren
  - Mehrere Personen kennen den gleichen Code => weniger Risiken für das Team durch Ausfall / Weggang einer Person

Paarprogrammierung und Test First

- Paarprogrammierung wird oft mit „Test First“ kombiniert
  - Observer schreibt einen Testfall, der noch nicht erfüllt ist
  - Driver schreibt den Code dafür
  - Dann wird gewechselt

- Vorteile:
  - Testfälle sind vorhanden, sobald Code fertig ist. Code erfüllt diese Testfälle
  - Testfälle konkretisieren die Anforderungen bevor man implementiert

- Näheres z.B. unter http://www.extremeprogramming.org/rules/testfirst.html



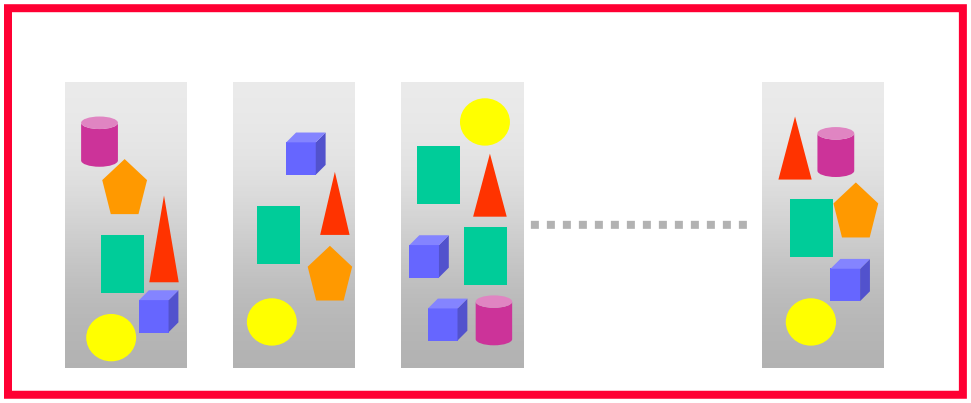
Konfigurationsmanagement

- Gemeinsame Erstellung von Code in verschiedenen Versionen und in Konfigurationen

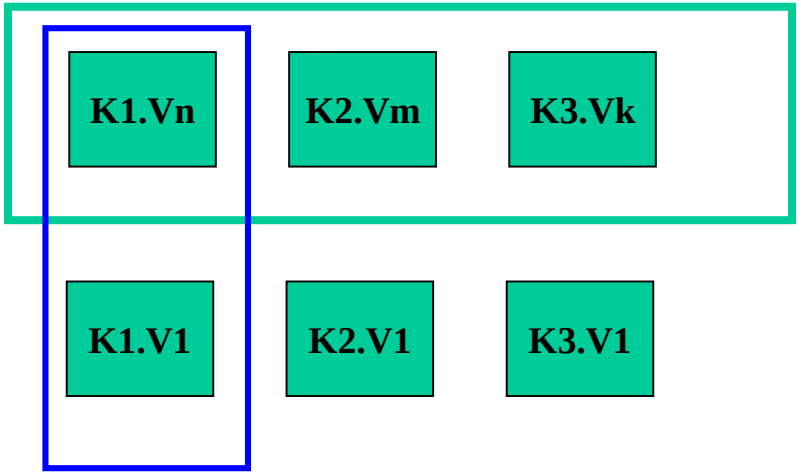
Siehe auch Kapitel 21 bei Ludewig/Lichter (Hausaufgabe 1.4)

Nachfolgend ein paar Folien dazu

Begriffsübersicht

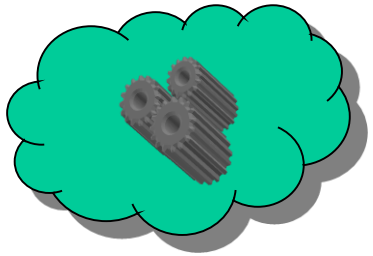


Variante, Produkt-Linie, Releases



Konfigura-tion

Versionen



Legende: Ki sind Komponenten, bunte Symbole sind Features (also Funktionalität des Systems)

Begriffe des Konfigurationsmanagements

- Entwicklung einer Komponente über die Zeit => Version
- Zusammensetzung aus verschiedenen Komponenten => Konfiguration
- Auslieferung eines Produktes (einer Konfiguration) in verschiedenen Ausbaustufen an einen Kunden => Release
- Auslieferung verschiedener Produkte (mit unterschiedlichen Features) an verschiedene Kunden => Varianten, Produktlinien

Version und Konfiguration

- Version ist die Ausprägung einer Komponente zu einem bestimmten Zeitpunkt
- Konfiguration ist eine konsistente Zusammenstellung von mehreren Softwarekomponenten

- Eine Version kann zu mehreren Konfigurationen gehören
- Zu einer Konfiguration kann von einer Komponente (höchstens) eine Version gehören

- Baseline: speziell ausgezeichnete Konfiguration, als Ausgangspunkt für Weiterentwicklung

Release

- Release: an Kunden ausgelieferte Konfiguration
- Ein Release enthält typischerweise
  - Konfigurationsdateien (Auswahl der Konfiguration)
  - Daten, die für einen erfolgreichen Betrieb nötig sind
  - Installationsprogramm
  - Systemhandbuch
  - Produktwerbung
- Neues Release notwendig, wenn
  - Ernsthafte Fehler behoben wurde (bei kleinen Fehlern Patches)
  - Wichtige neue Funktionalität implementiert wurde
    - Um Konkurrenzprodukt zu überholen
    - Um Marketingplanung einzuhalten
    - Um Änderungswünsche von Kunden aufzunehmen

Techniken des Konfigurationsmanagements

- Gemeinsame Erstellung von Code in verschiedenen Versionen
  - Versionskontrolle
    - Identifikation von Softwareeinheiten
  - Koordination der Teamarbeit
    - Konfliktlösung
    - Gemeinsame Referenzumgebung
  - Änderungskontrolle
    - Änderungshistorie

- und in Konfigurationen
  - Konfigurationskontrolle
  - Konstruktion ausführbarer Programme (Build Management)

[http://software-carpentry.org/]

Gemeinsame Erstellung von Code

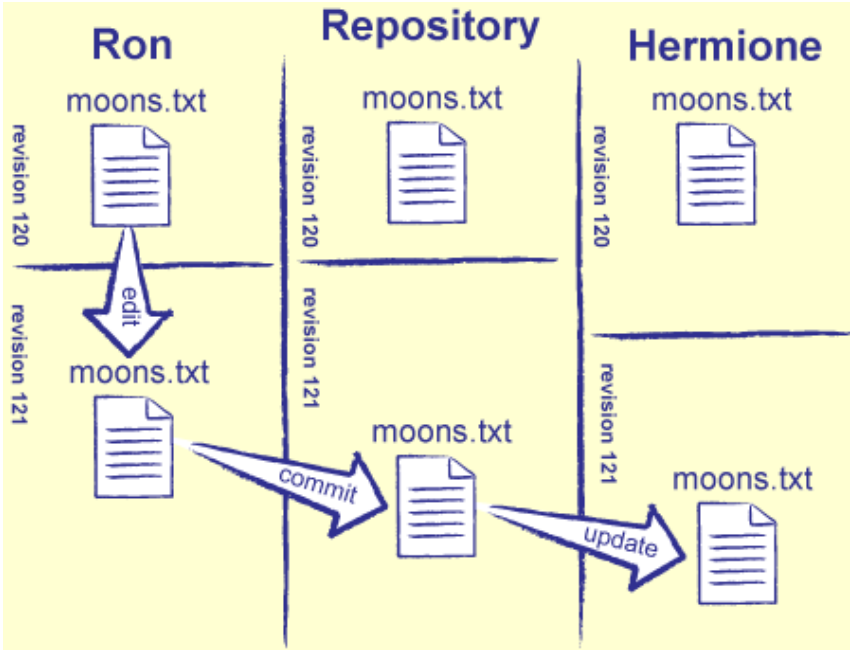
- Was passiert, wenn 2 Leute gleichzeitig am gleichen Code unterschiedliche Codeänderungen durchführen?

- Option 1: müssen sich abwechseln
  - Aber dann kann immer nur einer arbeiten
  - Wie kann man das erzwingen?

- Option 2: die Änderungen später zusammenführen
  - Kann viel Nacharbeit benötigen
  - Dabei kann etwas verloren gehen

Lösung: Versionsmanagement

- Die Lösung ist ein Versionsmanagement-System
- Grundiee: gemeinsame Ablage (repository)
- Alle haben eine eigene Arbeitskopie. Wenn sie diese mit anderen teilen wollen, laden sie das in das Repository (commit)
- Die anderen holen sich diese Kopie (update)

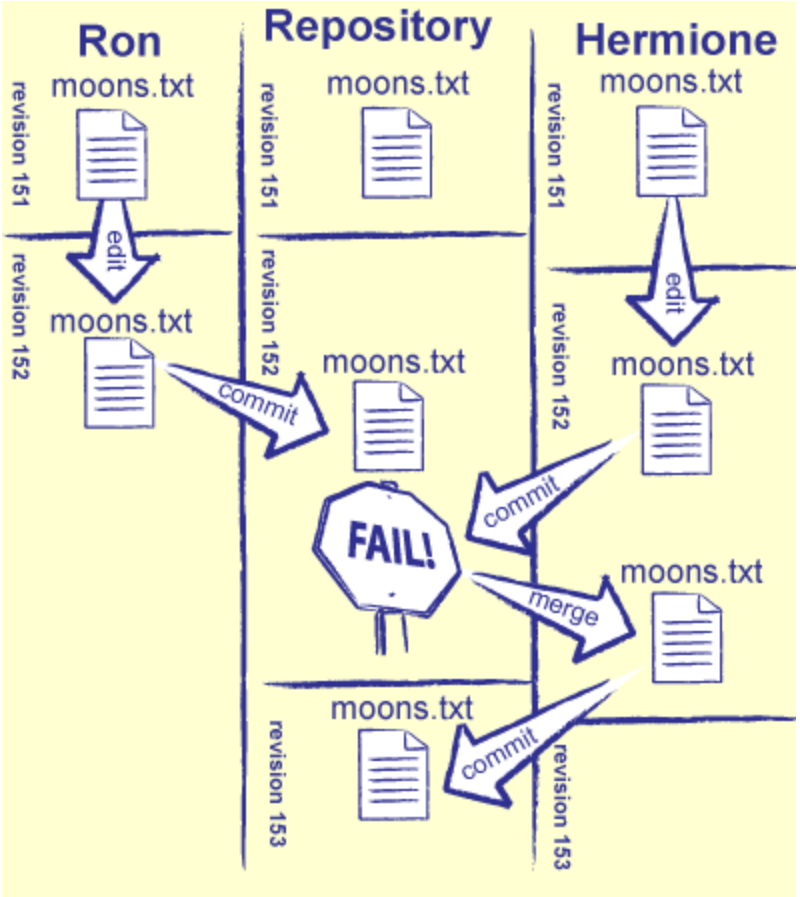


Gemeinsame Ablage (Repository)

- Registrierung aller Komponenten
  - Liste
  - Datenbank
- Eindeutige Bezeichnung
  - Linear: Doc1, Doc2, Doc3
  - Hierarchisch: LSB.BDS.ENT.Doc1 (Projekt LSB, Komponente BDS, Phase Entwurf)

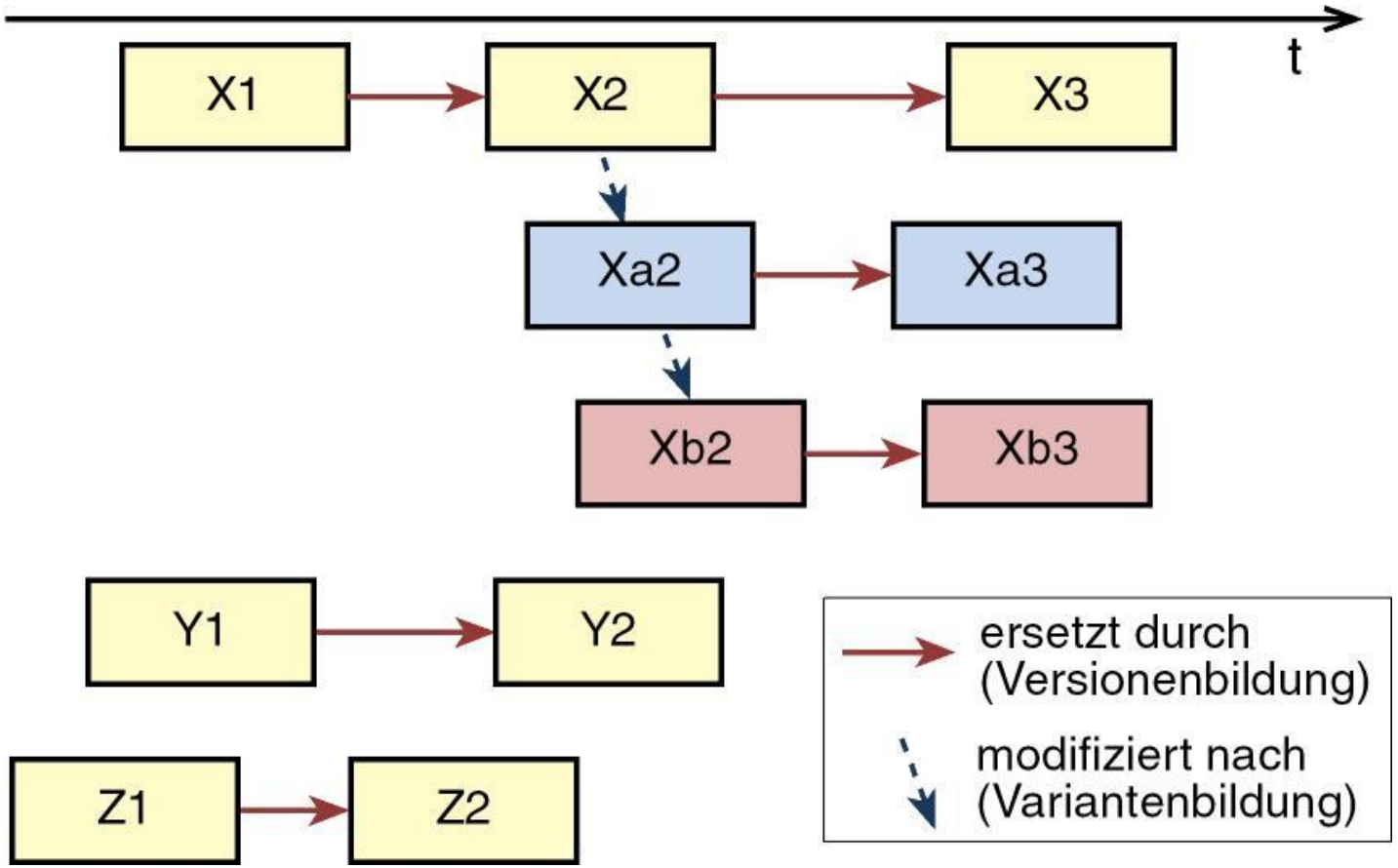
Konfliktlösung

- Option 1: nur einer schreibt
  - Pessimistische Nebenläufigkeit
  - Z.B. in Microsoft Visual SourceSafe
- Option 2: Zusammenfügen der Änderungen
  - Optimistische Nebenläufigkeit
  - "It's easier to get forgiveness than permission"
  - Machen die meisten modernen Systeme (z.B. Subversion, Git)



Versionsfortschritt

Verzweigungen möglich!
Konfliktlösung muss nicht sofort erfolgen



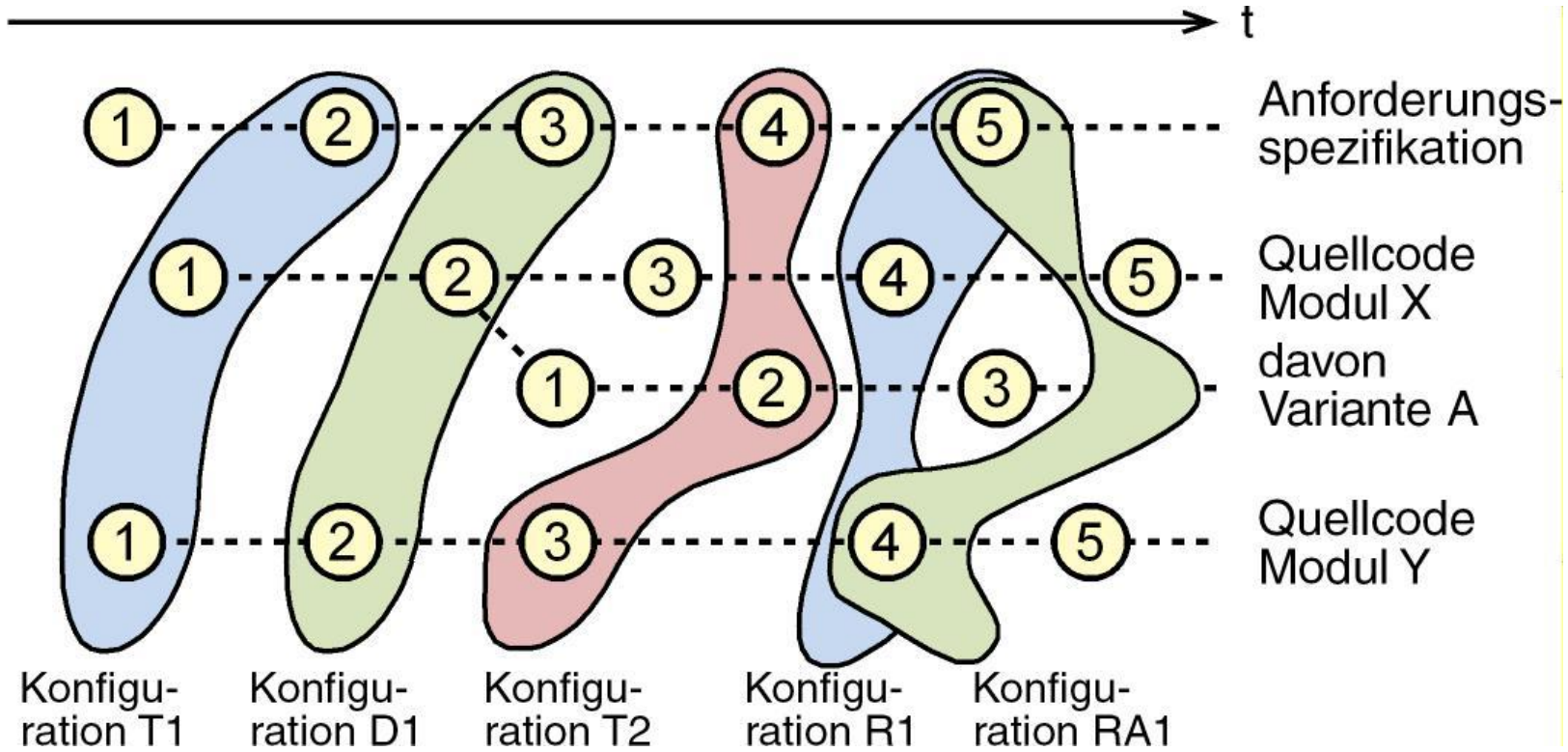
[http://software-carpentry.org/]

Änderungskontrolle

- Änderungen im Code rückgängig machen
  - Versionmanagementsystem protokolliert alle Änderungen (was, von wem)
  - Um Speicher zu sparen: Speicherung des Deltas zur wiederholten Herstellung
- Dies ermöglicht Zurücksetzen zu älterer Version („Roll back“)
- Auch für einzelne EntwicklerIn (ohne Team) hilfreich
- Alte Versionen auch wichtig, da alte Releases gewartet werden müssen

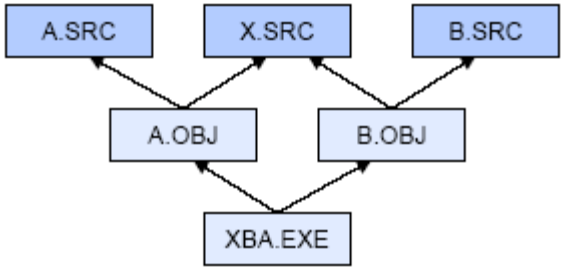
Konfigurationskontrolle

Auch hier Verzweigungen möglich!



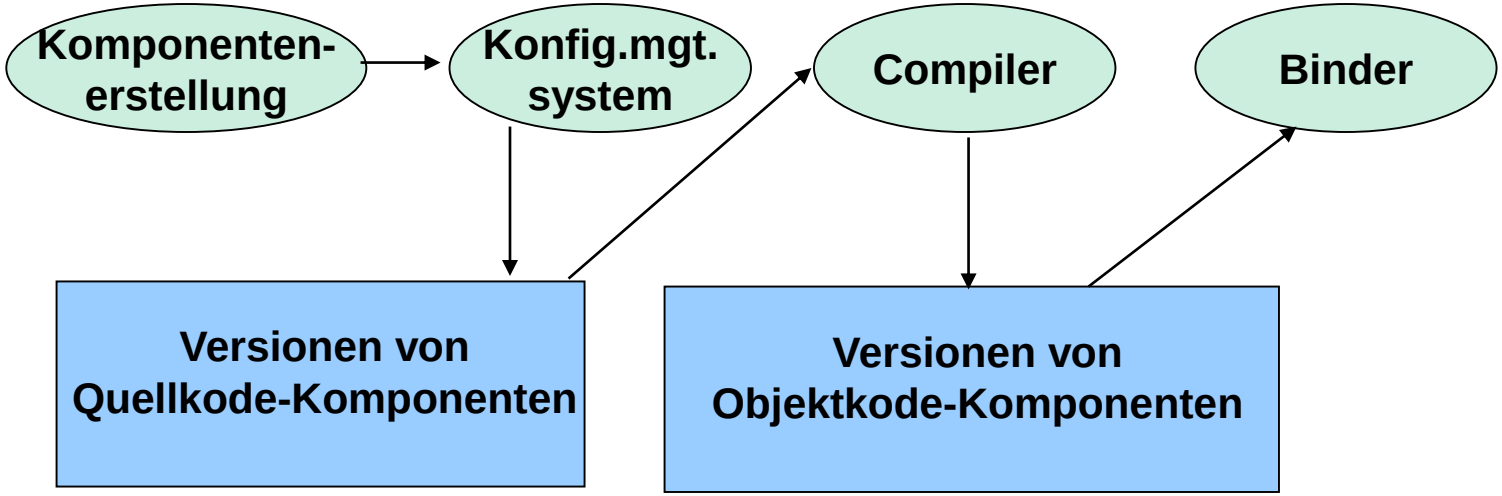
Build Management (1)

- Build Management = Automatisches Erstellen von ausführbarem Code und Dokumentation
- Originale (z.B. Anforderungsdokumente, Quellcode, Projektplan)
- Evtl. Ableitungen (z.B. Object-Files, Executable, Cross-Reference-List)
  - Müssen Abhängigkeiten berücksichtigen



Build Management (2)

- Verfahren zur Ableitung (Makefiles, Compiler, etc)

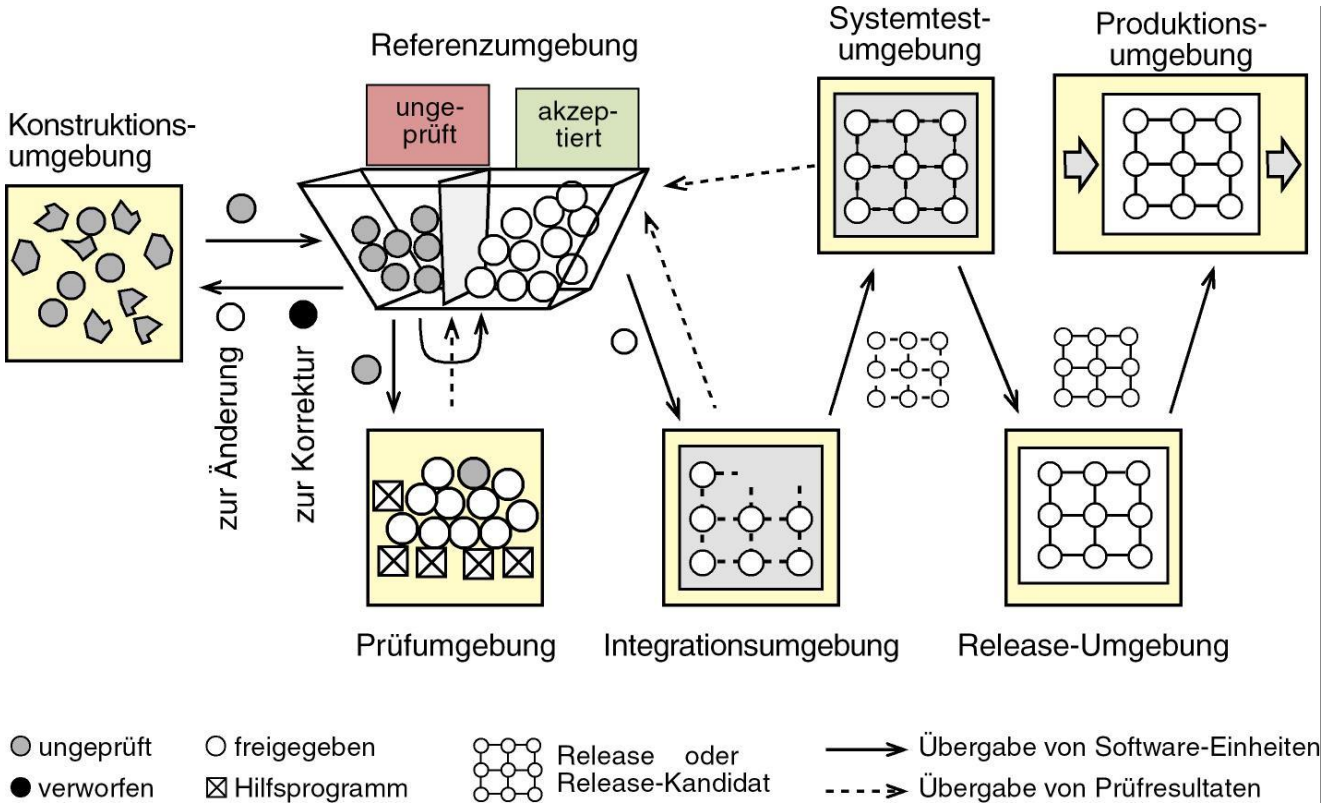


- Kann auch automatisierte Qualitätsanalysen (Metriken, Tests) und Berichte beinhalten
- Continuous build: Build wird bei jedem Code check-In durchgeführt
  - Stellt sicher, dass neuer Code „nichts kaputt macht“

Statusverwaltung

Referenzumgebung enthält auch Komponenten, die noch nicht akzeptiert sind

Prüfumgebung braucht auch Komponenten, die schon freigegeben sind (als Kontext der zu prüfenden Komponente)



Beitrag zu den Kernfragen

- Qualität
  - Qualität der Software durch Prüfumgebung sichergestellt
- Beteiligte (NutzerInnen)
  - Erstellung von Baselines und Releases in unterschiedlichen Varianten (angepasst an Kundenwünsche)
- Beteiligte (EntwicklerInnen)
  - Archivierung aller Dokumentenkomponenten
  - Modulare Weiterentwicklung durch Abspaltung von Versionen
  - Nachverfolgbarkeit, wer was gemacht hat, durch Änderungshistorie
- Kosten/Zeit
  - Weniger Kosten/Zeit für die Wartung, Behebung von Fehlern etc durch effiziente Werkzeugunterstützung

Vor- und Nachteile

- Vorteile
  - Vermindern bzw. Vermeiden von
    - Integration falscher Komponenten
    - Unkontrollierten Änderungen
    - Erneuter Entwicklung von Programmteilen
    - Verlust von Dokumenten und Komponenten
  - Archivierung aller Änderungen und Programme
- Nachteile (von Varianten)
  - Varianten erschweren die Wartung
    - Variantenspezifische Änderungen müssen einzeln durchgeführt werden
  - Kunden verwenden nicht die aktuellste Version
    - Pflege von unterschiedlichen Versionen

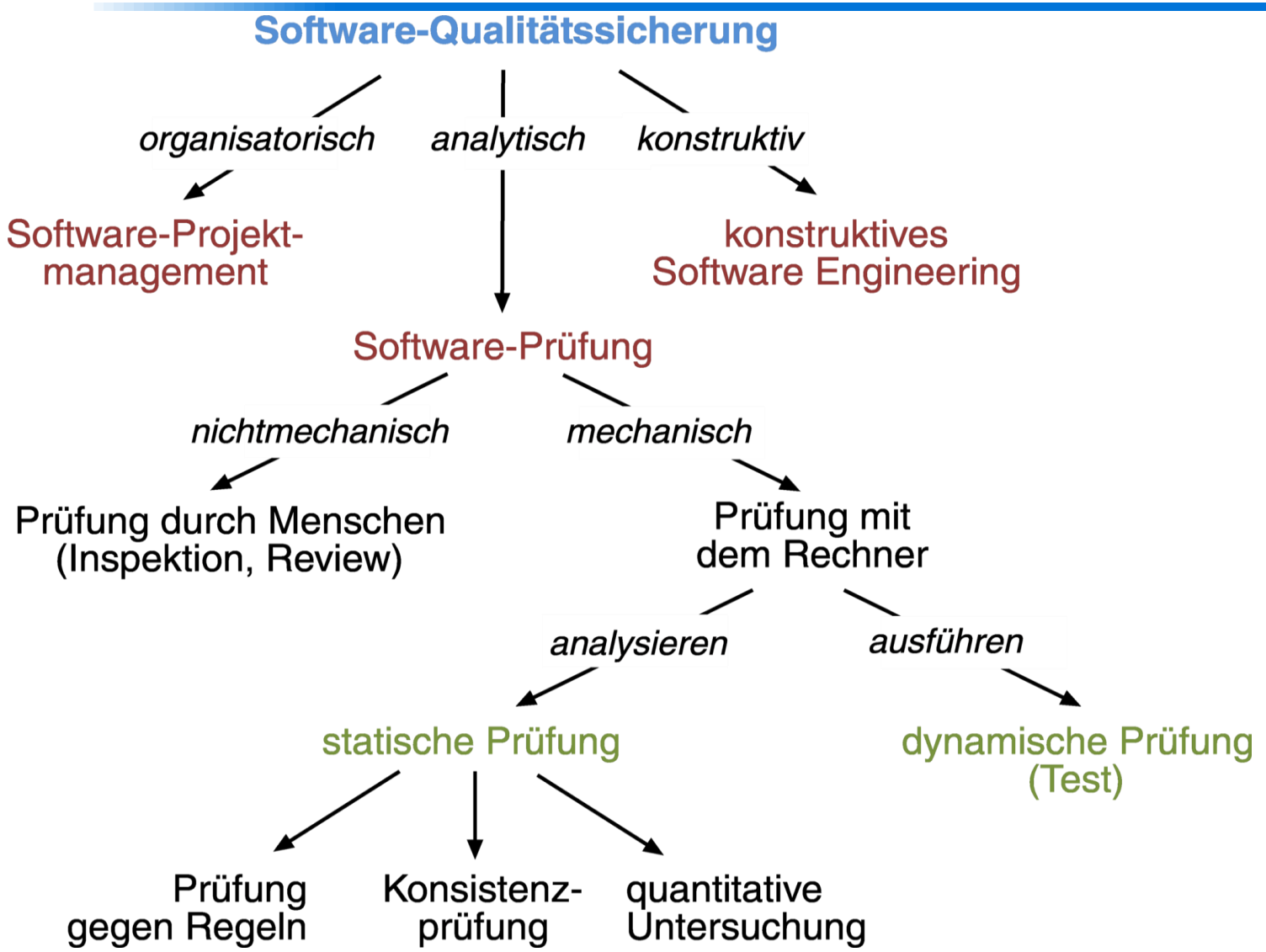
Zusammenfassung Konfigurationsmanagement

- Versionsmanagement unverzichtbar für produktive Zusammenarbeit von EntwicklerInnen
- Bei Auslieferung von Produkten Konfigurationsmanagement wichtig
- Mehr dazu in der Zentralübung

2.4. Statische Prüfung

- Einführung
- Metriken

Schwerpunkte der Qualitätssicherung



Statische Analyse von Programmcode

- Compiler führen eine statische Analyse des Programmtextes durch, indem sie die Einhaltung der Syntax der jeweiligen Programmiersprache prüfen

- Die meisten Compiler bieten darüber hinaus noch zusätzliche Informationen, die durch statische Analyse ermittelt werden
- Neben den Compilern gibt es Werkzeuge, sogenannte Analysatoren, die speziell zur gezielten Durchführung einzelner oder ganzer Gruppen von Analysen eingesetzt werden
- Fehler(zustände) bzw. fehlerträchtige Situationen, die mit der statischen Analyse von Programmcode nachgewiesen werden können, sind beispielsweise
  - Verletzung der Syntax
  - Abweichungen von Konventionen und Standards
  - Auffällige Metriken
  - Kontrollfluss- und Datenflussanomalien

Syntaxprüfung

- Verletzung der Syntax der Programmiersprache werden als Fehler oder Warnung gemeldet

- Viele Compiler führen darüber hinaus weitere Überprüfungen durch, z.B.
  - Verwendung der einzelnen Programmelemente (Variablen, Funktionen, ...)
  - Prüfung der typgerechten Verwendung der Daten und Variablen bei streng typisierten Programmiersprachen (C, C++, Java)
  - Ermittlung von nicht deklarierten Variablen
  - Nicht erreichbarer Code (»dead code«)
  - Fehler bei Überschreitung von Feldgrenzen
  - Prüfung der Konsistenz von Schnittstellen
  - Prüfung der Verwendung aller Marken als Sprunganweisung und Sprungziel

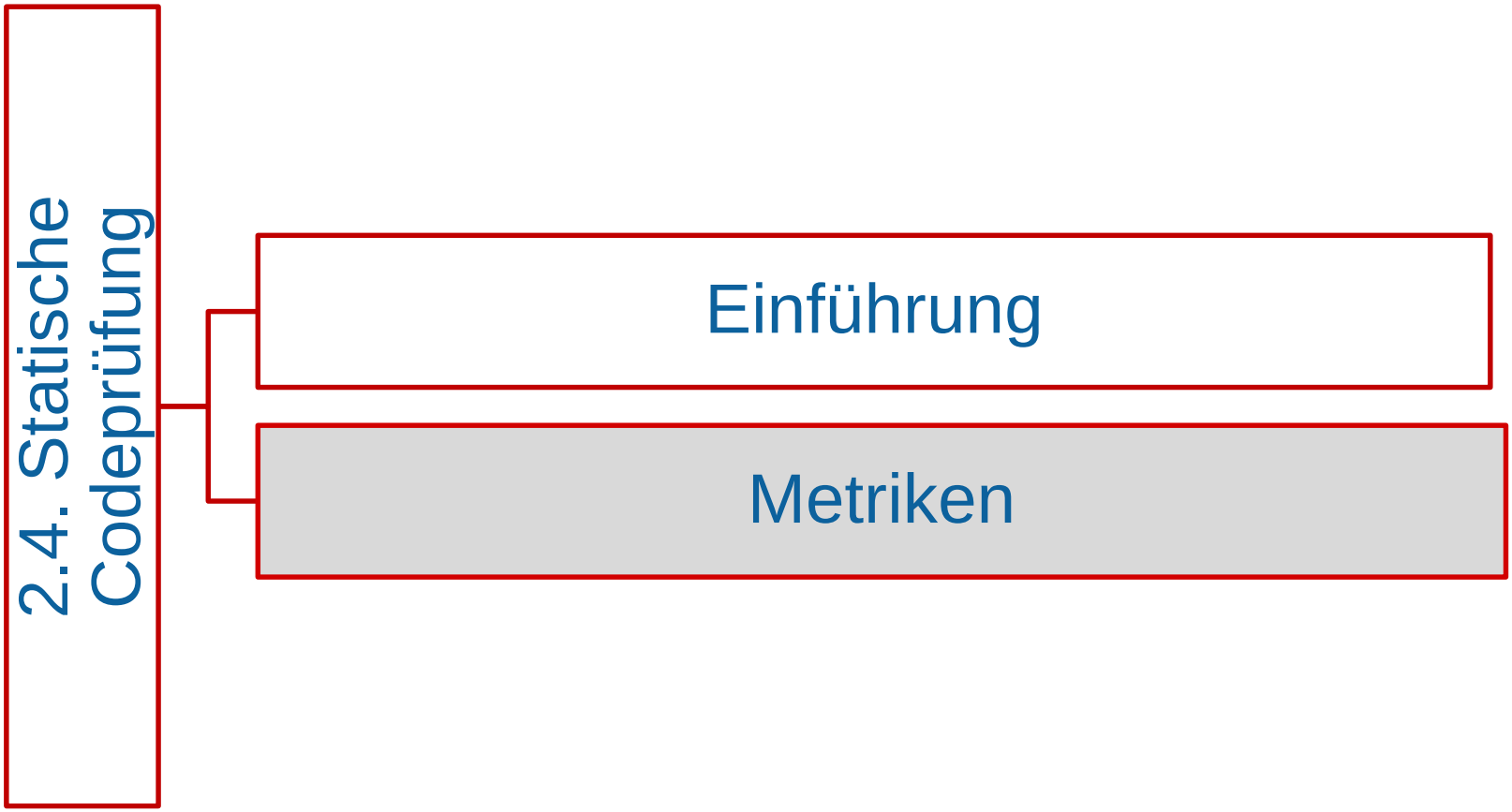
Einhaltung von Konventionen und Standards

- Einhaltung der Programmier- oder Modellierkonventionen durch entsprechende Analysatoren nachweisbar
  - Überprüfung benötigt wenig Zeit und so gut wie keine Personalressourcen
  - Darüber hinaus ist meist ein weiterer Vorteil gegeben: Ist den Software-EntwicklerInnenn bewusst, dass der Programmcode bzw. die Modelle von einem Werkzeug auf Einhaltung der Richtlinien überprüft wird, ist deren Bereitschaft zu ihrer Umsetzung bedeutend höher als ohne eine automatische Prüfung

- Tipp: Nur solche Richtlinien in einem Projekt zulassen, zu denen **Prüfwerkzeuge** vorhanden sind (andere Vorschriften erweisen sich in der Praxis ohnehin schnell als bürokratischer Ballast)
- Aber: Werkzeuge für statische Analyse können große Mengen von Warnungen und Hinweisen erzeugen, die gut verwaltet werden müssen, um eine effektive Nutzung des Werkzeugs zu erlauben
- Näheres siehe (Zentral-)Übung

Nutzen von statischen Analysen

- Frühe Erkennung von Fehlern vor der Testdurchführung
- Frühe Warnung vor verdächtigen Aspekten in Code oder Design, durch Berechnung von Metriken wie z.B. hohes Komplexitätsmaß
- Identifizierung von Fehlern, die durch Test nicht effektiv und effizient aufzudecken sind
- Aufdecken von Abhängigkeiten und Inkonsistenzen in Softwaremodellen, wie z.B. redundante oder die Architektur verletzende Links
- Verbesserte Lesbarkeit, Änderbarkeit und Wartbarkeit von Code und Design
- Vorbeugung von Fehlerzuständen, falls aus Erfahrung gelernt wird und sich dies in der Entwicklung niederschlägt



Metriken

- Metriken messen Eigenschaften (erwartete oder vorhandene) von Produkten oder Prozessen, z.B.
  - Kosten des Entwicklungsprozesses (Zeit, Personal,..)
  - Fehler
  - Volumen und Umfang (von Dokumenten und Code)
  - Qualität (z.B. Komplexität)

- Metriken unterstützen
  - Qualitätsbewertung (Verstehen, z.B. welche Klasse ist am größten)
  - Quantifikation von Erfahrungen: Steuern, insbes. Nutzung für
    - Prognosen (Z.B. Aufwandsabschätzung)
    - Entscheidungen (z.B. angemessener Aufwand)

Prozessmetriken

- Im Rahmen von Prozessverbesserung wird auch das Verhalten der Prozessbeteiligten (Historie der Artefakte) gemessen
- Beispiele
  - Wie oft wurde eine Komponente geändert?
  - Wann traten diese Änderungen auf (z.B. kurz vor Release)?
  - Wie viele Leute haben an einer Komponente gearbeitet?
  - Auch soziale Metriken
    - Wie oft kommunizierten welche EntwicklerInnen miteinander? Kommunikationsstruktur oft ähnlich zu Komponentenstruktur.

Dokumentenmetriken

- Typischerweise werden Eigenschaften von Dokumenten (insbesondere Code) gemessen
  - Qualität
  - Volumen und Umfang
- Geben insbesondere Hinweise auf Verständlichkeit und Wartbarkeit
- Benötigen Werkzeugunterstützung

Beispiele für Code Metriken

Code Länge (LOC)	LOC misst die Programmgröße: Hoher LOC einer Komponente deutet auf Komplexität und Fehleranfälligkeit
Fan in/Fan-out	Fan-in misst die Anzahl von Funktionen, die eine bestimmte Funktion X aufrufen. Fan-out misst die Zahl der Funktionen, die X aufruft. Hoher Fan-in: X eng gekoppelt mit vielen anderen Funktionen. Änderung hat große Auswirkung. Hoher Fan-out: X hat wahrscheinlich komplexe Kontrollogik (um die vielen aufgerufenen Funktionen zu synchronisieren).
Verzweigungs-tiefe	Misst die Tiefe der ineinander geschachtelten IF-Anweisungen. Hohe Tiefe deutet auf schlechte Verständlichkeit und Fehleranfälligkeit.

Objekt-Orientierte Metriken

- Komplexität einer Klasse
  - Operationsanzahl (gewichtet mit Komplexität)
  - Tiefe des Vererbungsbaumes
  - Anzahl der überschreibenden Operationen einer Superklasse
- Kohäsion einer Klasse (innerer Zusammenhang)
  - Anzahl der Operationspaare, die kein gemeinsames Attribut benutzen – Anzahl der Operationspaare, die ein gemeinsames Attribut benutzen
- Kopplung von Klassen (externe Abhängigkeiten)
  - Fan-in/Fan-out bezogen auf Operationen
  - Unterscheide Aufruf von anderen Klassen bzw. zu / von der eigenen Klasse