HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

Grundlagen des

Software-Quality-Management

Prof. Dr. Stephan Kleuker
Hochschule Osnabrück

Ich

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Prof. Dr. Stephan Kleuker, geboren 1967, verheiratet, 2 Kinder
- seit 1.9.09 an der FH, Professur für Software-Entwicklung
- vorher 4 Jahre FH Wiesbaden
- davor 3 Jahre an der privaten FH Nordakademie in Elmshorn
- davor 4 ½ Jahre tätig als Systemanalytiker und Systemberater in Wilhelmshaven

- s.kleuker@hs-osnabrueck.de, Raum SI 0109

Ablauf

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 11 Termine 2h Vorlesung + 2h Praktikum = 5 CP
- online: Strukturierung durch Lernnotizen zum Selbststudium
- Praktikum :
  - Anwesenheit = (Übungsblatt vorliegen + Lösungsversuche zum vorherigen Aufgabenblatt)
  - online: nach erstem gemeinsamen Termin mit Terminplan
  - Übungsblätter müssen sinnvoll bearbeitet sein
- Übungsblätter: Vertiefung, Erarbeitung verwandter Themen
- Prüfung: Hausarbeit (Kenntnisse aus der Vorlesung anwenden und/oder eigene Untersuchungen)

- Folienveranstaltungen sind schnell, nutzen Sie die Stopp-Taste
- von Studierenden wird hoher Anteil an Eigenarbeit erwartet

Verhaltenscodex

- Rechner sind zu Beginn der Veranstaltung aus
- Handys sind aus
- Wir sind pünktlich
- Es redet nur eine Person zur Zeit

- Sie haben die Folien zur Kommentierung in der Vorlesung vorliegen, ab Fr-Abend mit Aufgaben im Netz, Aufgabenzettel liegen in der Übung vor (Ihre Aufgabe)

  http://kleuker.iui.hs-osnabrueck.de/index.html

- Probleme sofort melden
- Wer aussteigt teilt mit warum

fachlicher Veranstaltungsrahmen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Schwerpunkt: Software- und Data Engineering

vorhandene Qualifikationen:

- gutes Abstraktionsniveau bei der SW-Entwicklung (Pattern, Frameworks, Metaprogrammierung)
- Verständnis von projektabhängigen Varianten von SW-Entwicklungsprozessen
- Grundwissen in automatisierbaren funktionalen Tests

Ziele:

- Verständnis, wie Prozessqualität, die Qualität eines Unternehmens beeinflusst
- Verständnis, wie Qualitätssicherung Qualität von SW-Produkten beeinflusst
- Verständnis der formalen Möglichkeiten und Grenzen der Qualitätssicherung

typische Zielrolle: Software-Entwickler*in mit Ziel SW-Entwicklungsleitung

Grundvoraussetzungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ordentliche Programmierkenntnisse (möglichst in C/C++ und Java; Java reicht wahrscheinlich aus)
- Erfahrungen in Vorgehensmodellen des SW-Engineering
- Erfahrung in der Entwicklung größerer SW-Projekte
- systematischer Einsatz von (J)Unit-Tests

Prüfungsform: Hausarbeit

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ergebnis: schriftliche Ausarbeitung zu einem möglichen Thema der Veranstaltung in Gruppen von 3 (2) Personen

mögliche Themengebiete:

- Einarbeitung in neue Werkzeuge; neuer Model Checker, Testwerkzeuge für neue Sprachen
- Analyse bekannter QS-Maßnahmen (z. B. aus Betrieb) mit Bewertung und Verbesserungsvorschlägen
- QS-gesicherte Beispielentwicklung mit Go
- Analyse von Möglichkeiten zur Entwicklung Go-Werkzeugen
- Zusammenfassende Darstellung eines Prozessstandards und Übertragung auf Software (-Entwicklung, -Wartung, -Nutzung)
- … (s. WebSeite)

Themengebiete (Planung)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 1 Einführung, Motivation, Grundlagen
- 2 Qualitätsmanagement und Prozessmodellierung
- 3 Qualitätssicherung
- 4 Formale Verifikation (Model Checking)

1. Einführung, Motivation, Grundlagen

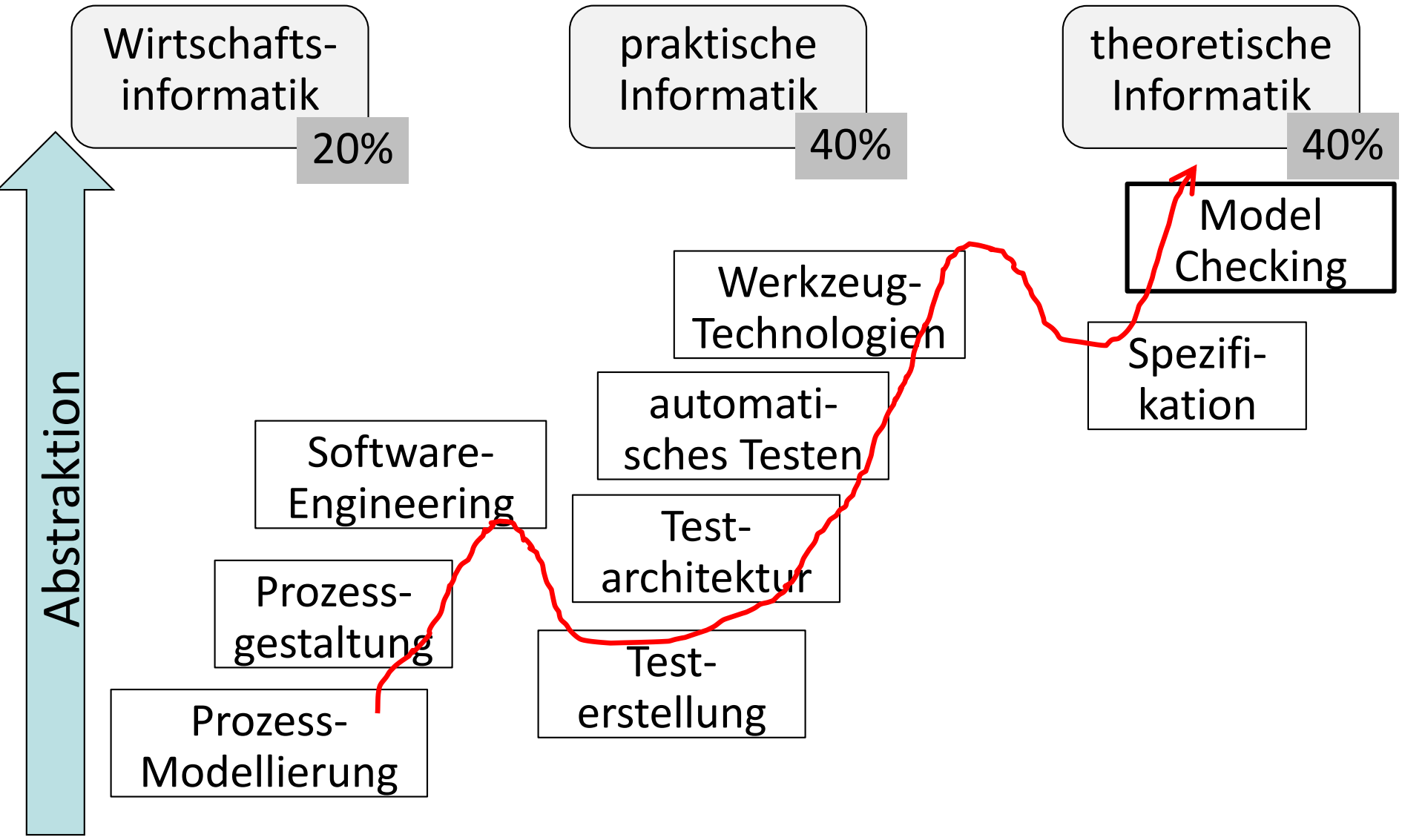
- Qualitätsbegriff
- Fehlerquellen
- Entscheidbarkeit

kleine korrekte Programme

- Spezifikation: Schreibe eine Methode max(), der drei int-Werte übergeben werden, die den größten Wert dieser Werte zurück gibt

```
public class Maxi {
  public static int max(int x, int y, int z){
    int max = 0;
    if (x > z) {
      max = x;
    }
    if (y > x) {
      max = y;
    }
    if (z > y) {
      max = z;
    }
    return max;
  }
}
```

Zuordnung zu Themenbereichen

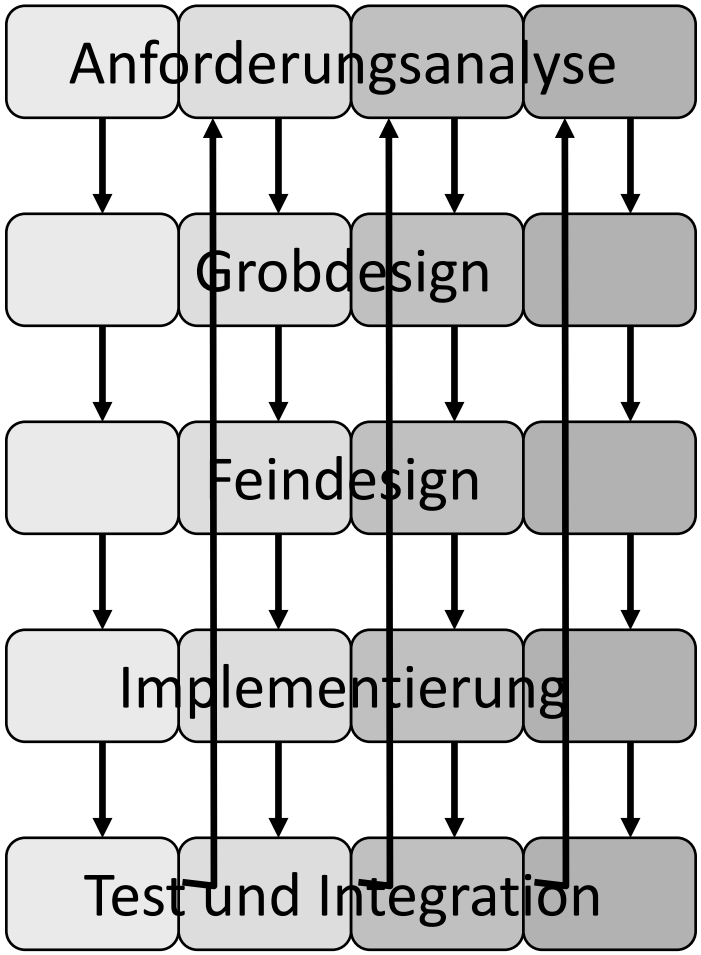


Fehlerquellen von Software

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Software-Fehler und ihre dramatischen Folgen
- Typische Entwicklungsphasen von Software
- Fehler in der Anforderungsanalyse
- Fehler im Design
- Fehler bei der Implementierung
- Fehler bei der Qualitätssicherung
- Fehler im Projektumfeld
- Fehler im Unternehmen

Skizze eines Vorgehensmodells



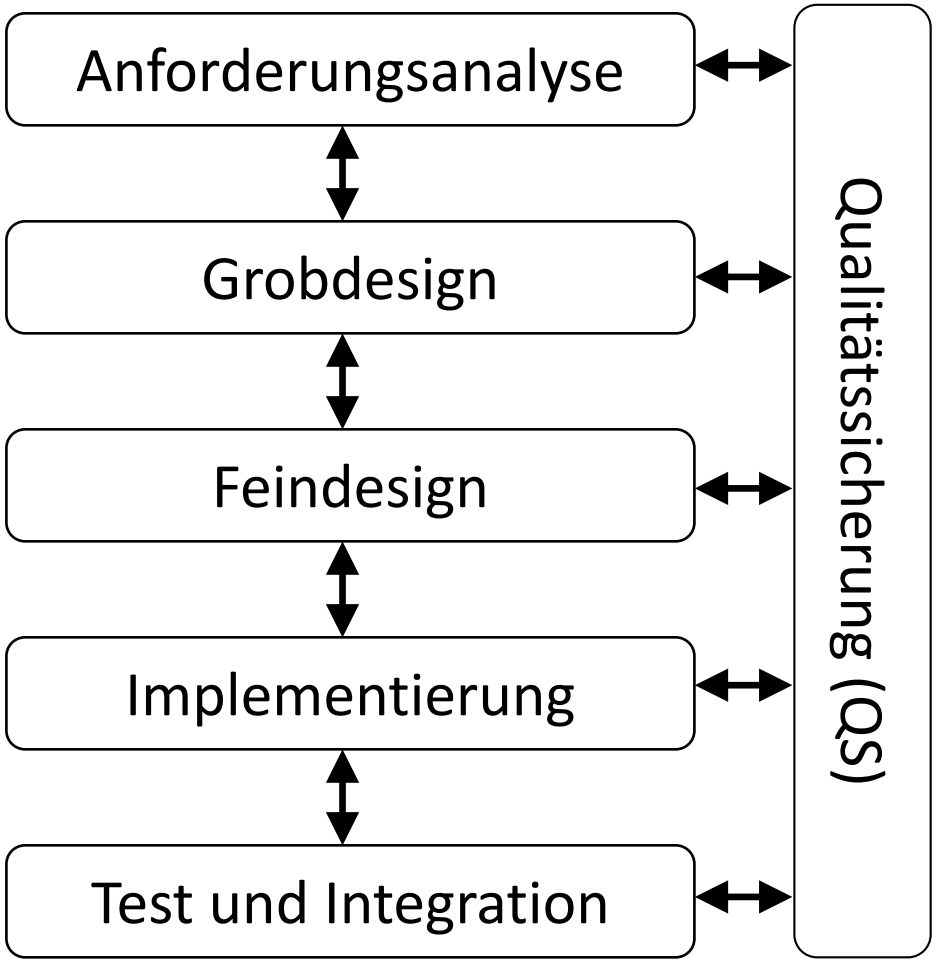
Bsp.: vier Inkremente

Merkmal:
Projekt in kleine Teilschritte zerlegt, n+1-ter Schritt kann Probleme des n-ten Schritts lösen

Vorteile:
- dynamische Reaktion auf Risiken
- Teilergebnisse mit Kunden diskutierbar

mögliche Nachteile:
- schwierige Projektplanung
- schwierige Vertragssituation
- Kunde erwartet zu schnell Endergebnis

Qualitätssicherung (1/2)



- QS ist eigenständiges Teilprojekt
- zu jedem Projekt gehört ein QS-Plan
- QS Plan: was wird wann von wem wie überprüft
- QS unabhängig vom Projekt organisiert
- Form der QS sehr stark projektabhängig
- häufig Forderung nach Normen (z.B. ISO 9000) und QS-Standards (z.B DO-178C)

Qualitätssicherung (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

konstruktive Qualitätssicherung:

- QS vor Projekt planen
- Auswahl von Methoden, Werkzeugen, Prozessen, Schulungen
- Normen, Guidelines und Styleguides

analytische Qualitätssicherung:

- manuelle Verfahren (z.B: Inspektion und Audit) zur Prüfung der Einhaltung konstruktiver Ansätze
- Reviews von Teilprodukten (z.B. Anforderungen, Design)
- Einsatz von Testverfahren (Äquivalenzklassen, Überdeckungen) in unterschiedlichen Testphasen (Entwicklertests, Integrationstests, Systemtests, Abnahmetests)

Typische Fehler der Anforderungsanalyse

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Kunde wird zu wenig in Planungen eingebunden
- Kunde versteht Analysedokumente der IT nicht; nickt sie trotzdem ab
- Rahmenbedingungen (HW, umgebende SW) nicht geklärt
- Stakeholder (in irgendeiner Form Projektbetroffene) nicht in Analyse involviert (z. B. Endnutzer vergessen)
- Analyse der vom Kunden genutzten SW nicht durchgeführt; z. B. Look-and-Feel an branchenüblicher Software orientieren

- Grundregel für IT-Projekte: Garbage in - Garbage out

Typische Fehler des Designs

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- keine Prüfung, ob genau die Anforderungen umgesetzt werden
- Design-Entscheidungen werden nicht dokumentiert (UML: Aktivitätsdiagramme, Klassendiagramme, Zustandsdiagramme, Sequenzdiagramme)
- mangelndes Design-Know-how macht resultierende Software langfristig unwartbar, nicht erweiterbar
- Randbedingungen z. B. der HW nicht berücksichtigt (Performance, Kommunikationsprotokolle, ...)

Typische Fehler der Implementierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Laufendes Programm – unerwünschte Funktionalität
  - Spezifikation war missverständlich
  - Überflüssige Goldrandlösung
- Mögliche Alternativen werden vergessen
- Programme zu komplex zum Testen (Schachteln von if, while, switch)
- Programm bei Wiederverwendung/Erweiterung unwartbar (versteckte Konstanten, versteckte Abhängigkeiten)
- Existierende Lösungen missachtet
- Kein Gedanke an Laufzeit

Typische Fehler der Qualitätssicherung

- alle Tests werden von den Entwicklern durchgeführt
- Fehlermöglichkeiten vergessen
- Testfälle können nicht nachvollzogen oder wiederholt werden; mangelnde Dokumentation der Testfälle
- Anforderungen nicht konsequent in Testfälle umgesetzt
- fehlende Werkzeuge zur Testautomatisierung; manuelle Tests zu zeitaufwändig
- Testumgebung passt nicht zur realen Zielumgebung des Kunden
- Informationen zum Performance- und Lastverhalten angeschlossener Systeme nicht berücksichtigt

Typische Fehler in der Projektorganisation

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Kunde wird zu selten über Projektstand informiert
- Kunde will vor Fertigstellung der Software „nicht belästigt“ werden
- Der Umgang mit Änderungswünschen des Kunden und des Auftragnehmers ist unterspezifiziert
- Auftragnehmer wird über projektrelevante Änderungen beim Kunden nicht informiert
- kein Risikomanagement (beim Auftragnehmer und beim Kunden)

Typische Fehler im Software-Unternehmen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- keine Strategie bei der Produktentwicklung bzw. Auswahl von Entwicklungsprojekten
- Orientierung an kurzfristigen Gewinnen
- zu wenig oder nur Selbstschulung
- keine Reaktionsmöglichkeit auf Kundenwünsche
- kein ordentlicher Vertrieb
- zu wenig Personal
- fachlich nicht ausgebildetes Personal (positiv: nur Fachinformatiker*innen und Informatik-Absolvent*inn*en in der SW-Entwicklung)
- zu niedrige Gehälter / Fluktuation

Schleichende Software-Fehler - Beispiel

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- kleines Unternehmen geht mit viel Know-how und neuer Individual-SW auf den Markt
- SW wird bei Kunden direkt vor Ort angepasst
- mit Kundenzahl wächst Zahl der Anpassungen und weiterer Kundenwünsche

- dadurch, dass zentrale Daten mehrfach in Modulen gehalten werden, Datenabhängigkeiten schwer analysierbar sind und Individual-SW nicht dokumentiert ist, wird SW unwartbar

- typisches Problem vieler SW-Systeme: am Anfang erfolgreich werden sie irgendwann nicht mehr weiterentwickelbar

Ursachen für Software-Fehler

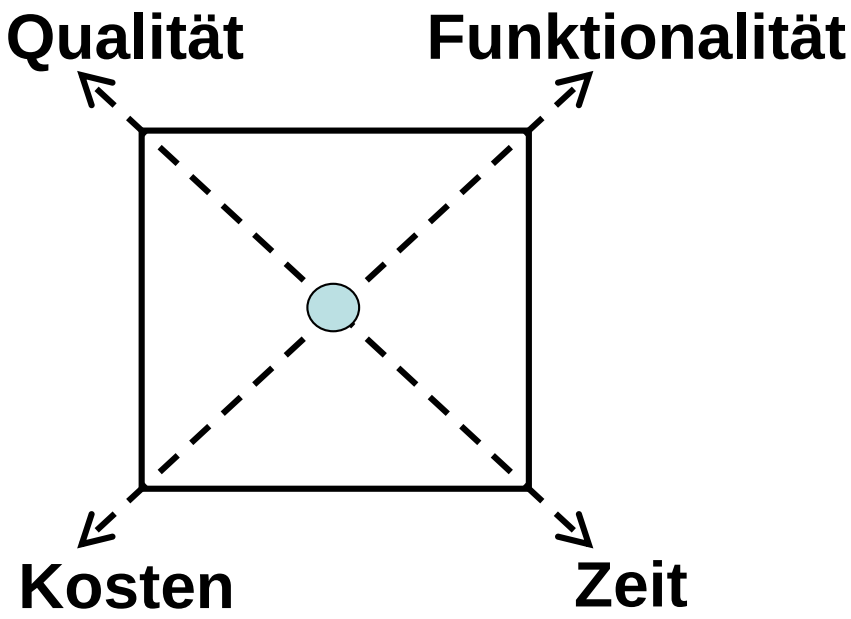
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- mangelndes Verständnis von Anforderungen
- Übersehen von Ablaufmöglichkeiten
- Programmierfehler

- zu wenig Zeit für Tests
- mangelnde Qualifikation der Mitarbeiter
- unpassende SW-Werkzeuge

- mangelndes Managementverständnis für IT-Entwicklung
- hoher Kostendruck, starker Preiskampf

Teufelsquadrat für IT-Projekte (Sneed)



- Konzentration auf ein Ziel benötigt bei gleichen Rahmenbedingungen Vernachlässigung mindestens eines anderen Ziels
- generelle Verbesserung nur durch Verbesserung der Rahmenbedingungen (z.B. der Prozesse)

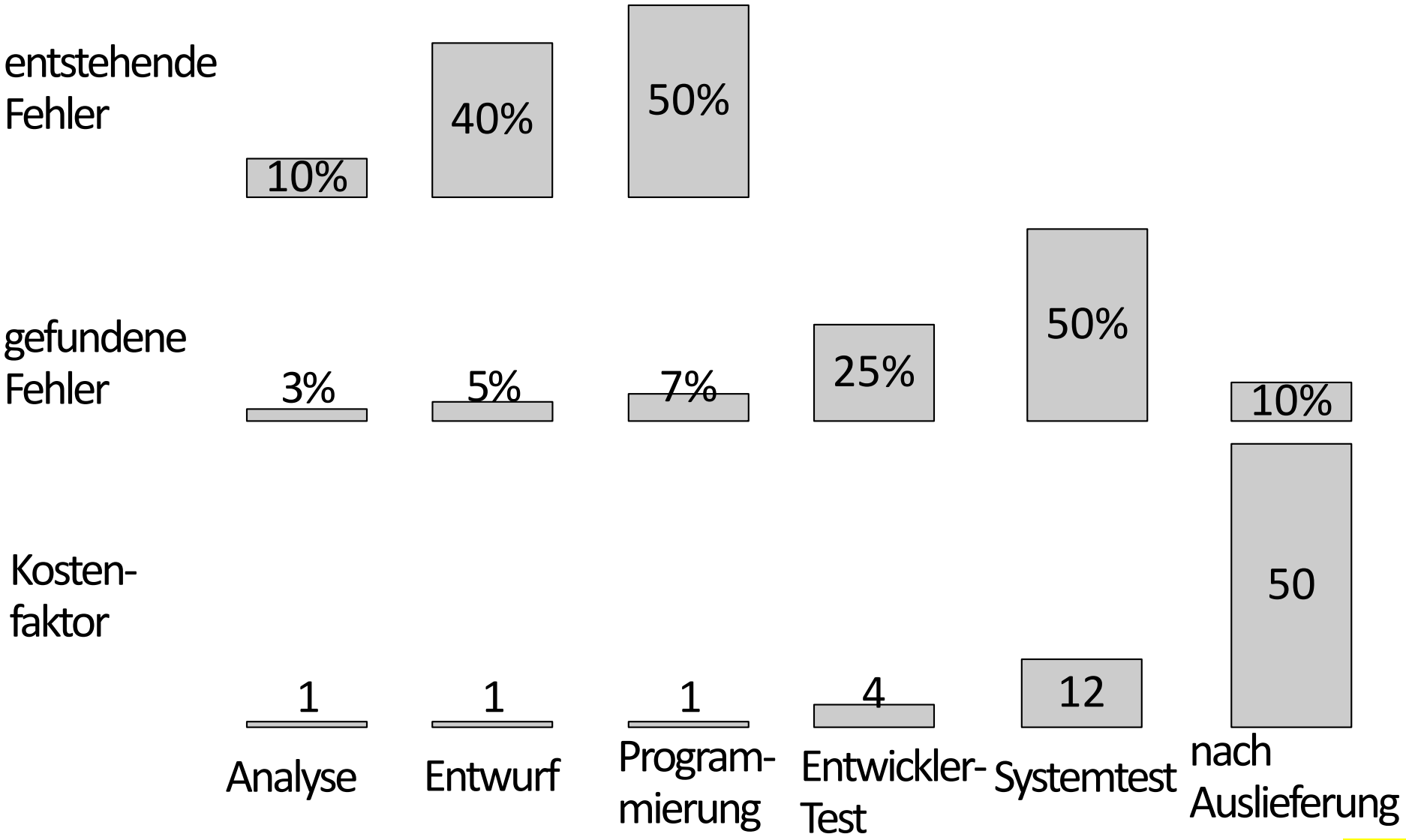
Qualitätssicherung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Qualitätssicherung (allgemein)
  Alle geplanten und systematischen Tätigkeiten, die notwendig sind, um ein angemessenes Vertrauen zu schaffen, dass ein Produkt oder eine Dienstleistung die gegebenen Qualitätsanforderungen erfüllt
- Qualitätssicherung (softwarespezifisch)
  Die Gesamtheit aller Maßnahmen und Hilfsmittel, die mit dem Ziel eingesetzt werden, die gestellten Anforderungen an den Entwicklungs- und Wartungsprozess sowie an das Softwareprodukt zu erreichen

aus DIN 55350-11: Begriffe zu Qualitätsmanagement und Statistik, Teil 11, 8/95

Wann werden Fehler gefunden?



nach K.-H. Möller, Ausgangsdaten für Qualitätsmetriken, 1996

Definitionen (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Qualität (DIN 55350-11) definiert als „Beschaffenheit einer Einheit bezüglich ihrer Eignung, festgelegte und abgeleitete Erfordernisse (Qualitätsanforderungen) zu erfüllen“
- Qualitätsanforderung: Gesamtheit der Einzelanforderungen an eine Einheit, die die Beschaffenheit dieser Einheit betreffen
- Qualitätsmerkmal: Die konkrete Beurteilung von Qualität geschieht durch so genannte Qualitätsmerkmale. Diese stellen Eigenschaften einer Funktionseinheit dar, anhand derer ihre Qualität beschrieben und beurteilt wird, die jedoch keine Aussage über den Grad der Ausprägung enthalten. Ein Qualitätsmerkmal kann über mehrere Stufen in Teilmerkmalen verfeinert werden.

Definitionen (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Qualitätsmaß: konkrete Ausprägung eines Qualitätsmerkmals geschieht durch so genannte Qualitätsmaße; dies sind Maße, die Rückschlüsse auf die Ausprägung bestimmter Qualitätsmerkmale zulassen (Beispiel: Durchschnittliche Antwortzeit für Laufzeiteffizienz)
- Fehlverhalten oder Ausfall (failure) zeigt sich dynamisch bei der Benutzung eines Produkts, beim dynamischen Test einer SW erkennt man keine Fehler, sondern Fehlverhalten bzw. Ausfälle.
- Fehler oder Defekt (fault, defect) ist bei SW die statisch im Programmcode vorhandene Ursache eines Fehlverhaltens oder Ausfalls

Einschub: Entscheidbarkeit

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Erinnerung, wichtigste Aussage aus dem Informatik-Studium:
- Es gibt unentscheidbare Probleme, d. h. Aufgaben für die kein Programm geschrieben werden kann
- theoretisches Beispiel: Halteproblem für Turing-Maschinen
- praktisches Beispiel: Schreibe ein Java Programm, dass ein Java-Programm mit einem String als Aufrufparameter und einen beliebigen String als Eingabe bekommt und berechnet, ob das Programm anhält
- Problem für einzelne konkrete Fälle lösbar (entscheidbar), aber generell nur aufzählbar
- Randnotiz: Das kann man doch mit KI lösen? NEIN, falls KI mit „hält oder hält nicht“ antwortet, zeigt Satz aber, dass die KI Fehler enthalten muss!! (KI-Antwort „ja“ „nein“ „weiß nicht“ ist ok)

Video

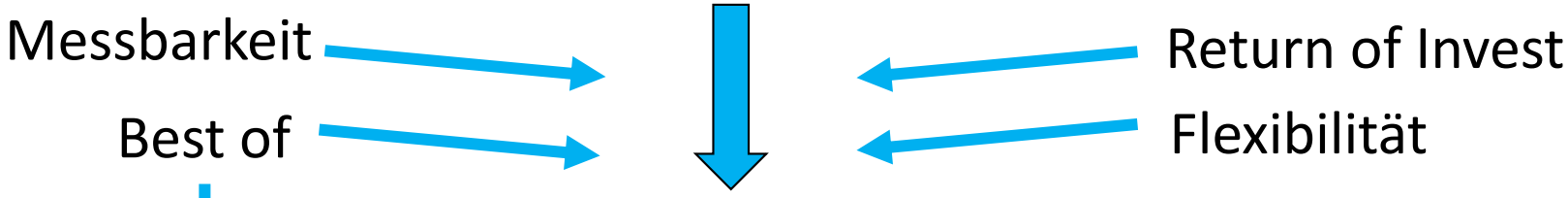
2. Qualitätsmanagement und Prozessmodellierung

- 2.1 Grundlagen von Prozessmodellierungen
- 2.2 Prozessstandards

Motivation / Ansatz

um Vorgehen zu analysieren muss es transparent sein

Prozessmodellierung (wer, was, wie , worum, mit wem, wann, …)



Ziel: gute Prozesse

Prozessstandards

Rahmenbedingungen für vernetzte Teilprozesse

Beispiel: Requirements Engineering

2.1 Grundlagen von Prozessmodellierungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Qualitätsmanagement
- Prozessdokumentation mit Aktivitätsdiagrammen
- ganzheitliche Modellierung
- Prozessoptimierung
- Ziele und Probleme von Prozessmodellierungen

Qualitätsmanagement

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- steuert die Qualität im gesamten Unternehmen
- typisches Ziel: Kontinuierliche Verbesserung aller Abläufe
- notwendig: Erfassung aller Abläufe durch Prozesse
- notwendig: Einheitliche Modellierung des Prozesse
- notwendig: Möglichkeiten zur Messung der Qualität (von Prozessen, Produkten, …)

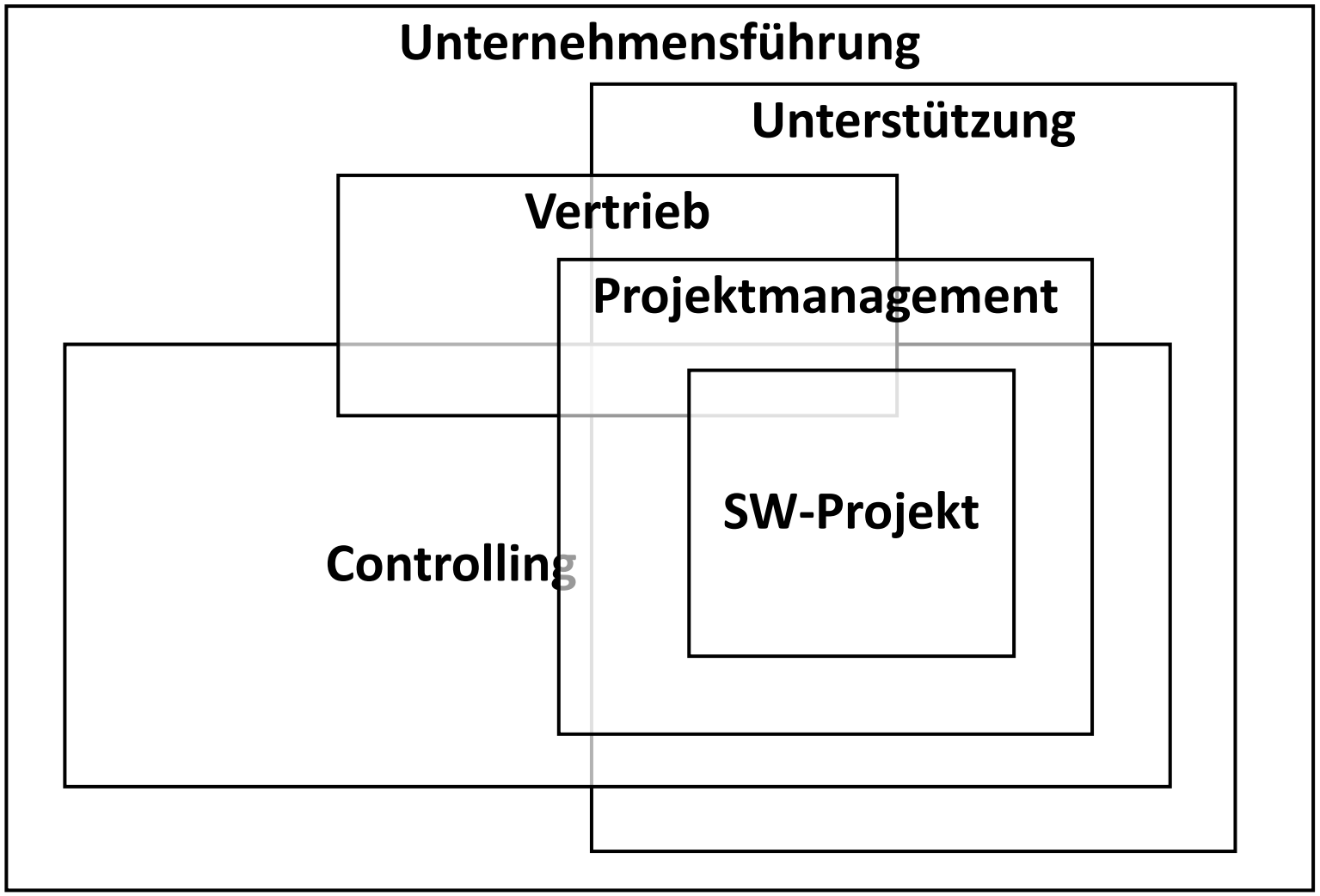
- ein Beispiel: Qualitätssicherungsprozess in der SW-Entwicklung

Ziele/Maße des QS-Prozesses der SW-Entwicklung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Ziel	Messmöglichkeit (Beispiele)
schnellere Entwicklung	• Output an Function/Feature Points • Anteil manueller Deployment-Schritte
weniger Fehler	• Incidents in Aufgabenverwaltung • Testüberdeckung
wart- und erweiterbarer Code	• Code-Metriken • Refaktorierungsaufwand bei ergänzten Modulen
höhere Produktqualität	• Kundenumfagen • Analyse Anzahl und Art der Anrufe bei Hotline
kontinuierliche Verbesserung	• Ableitung von evtl. kombinierten Kennzahlen aus Messmöglichkeiten

Umfeld von SW-Projekten – Prozesslandschaft



Prozesse in Unternehmen aus SW-Projektsicht

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

(Annahme SW ist wichtiges Kernprodukt)

- Unternehmensführung gibt Geschäftsfelder und Strategien vor
- Vertriebsleute müssen Kunden finden, überzeugen und Aufträge generieren
- Aufträge führen zu Verträgen, die geprüft werden müssen
- Das Personal für Aufträge muss ausgewählt werden und zur Verfügung stehen
- Der Projektablauf muss beobachtet werden, Abweichungen z. B. in Zeitplan müssen zu Steuerungsmaßnahmen führen
- Die SW muss realisiert werden
- …

Einstieg/Überblick Prozessmodellierung (+Optimierung)

- Vorbereitung
  - Grundsätzliche Ziele und Randbedingungen festlegen
  - Art der Modellierung mit Modellierungsregeln definieren
- Durchführung
  - beteiligte Rollen finden
  - Prozesse dokumentieren (mit zentralen Produkten)
  - Prozessoptimierungen überlegen
  - Prozesse in Pilotprojekten testen und verbessern
- langfristig
  - Prozesse kommunizieren und einführen
  - QM-Maßnahmen zur Bewertung festlegen
  - kontinuierlich Sinn /Erfolg / Optimierung prüfen

Rollenbegriff

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- Unterschiedliche Menschen arbeiten in verschiedenen Rollen zusammen
- Rolle: genaue Aufgabenbeschreibung, mit Verantwortlichkeiten (was soll gemacht werden) und Kompetenzen (welche Entscheidungen können getroffen werden, z. B. „Arbeit anweisen“)
- Mensch kann in einem Unternehmen/Projekt mehrere Rollen haben
- Eine Rolle kann von mehreren Menschen besetzt werden
- Beispielrollen: Vertriebsleiter, Vertriebsmitarbeiter, Projektleiter, Analytiker, Implementierer, Tester

Prozessbegriff

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

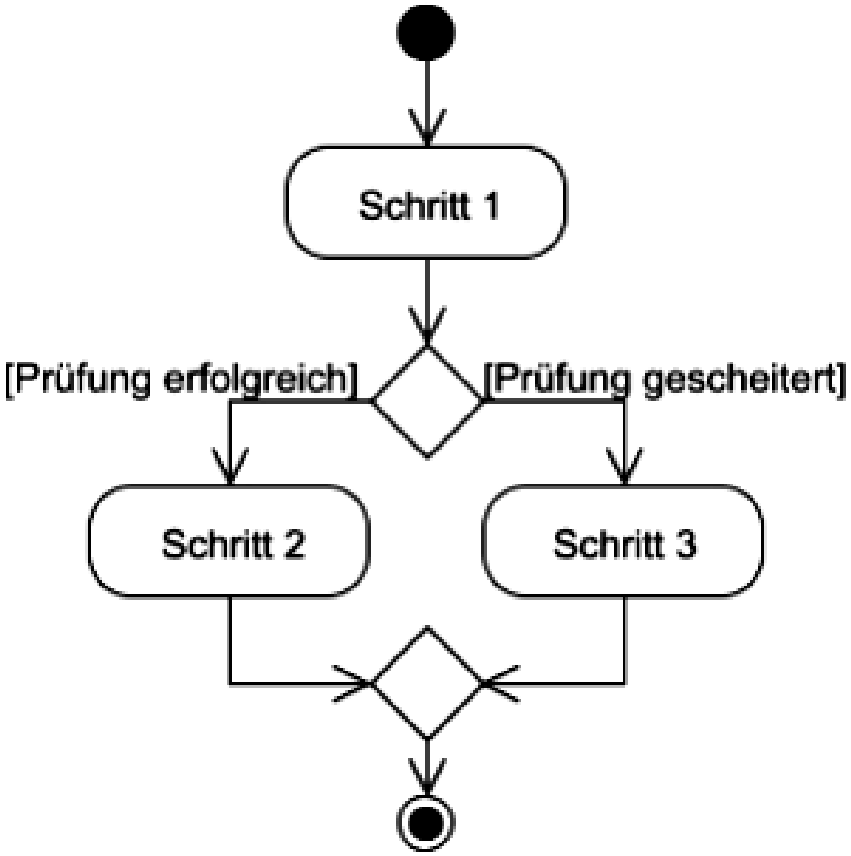
Prozessbeschreibungen regeln die Zusammenarbeit
verschiedene Menschen (genauer Rollen),

- Was soll in diesem Schritt getan werden?
- Wer ist verantwortlich für die Durchführung des Schritts?
- Wer arbeitet in welcher Rolle in diesem Schritt mit?
- Welche Voraussetzungen müssen erfüllt sein, damit der Schritt ausgeführt werden kann?
- Welche Teilschritte werden unter welchen Randbedingungen durchgeführt?
- Welche Ergebnisse kann der Schritt abhängig von welchen Bedingungen produzieren?
- Welche Hilfsmittel werden in dem Prozessschritt benötigt?
- Welche Randbedingungen müssen berücksichtigt werden?
- Wo wird der Schritt ausgeführt?

Prozesse sind zu dokumentieren und zu pflegen

Prozessmodellierung mit Aktivitätsdiagrammen

Zur Beschreibung werden folgende elementare Elemente genutzt:



genau ein Startpunkt

einzelner Prozessschritt (Aktion)

Kontrollknoten (Entscheidung)

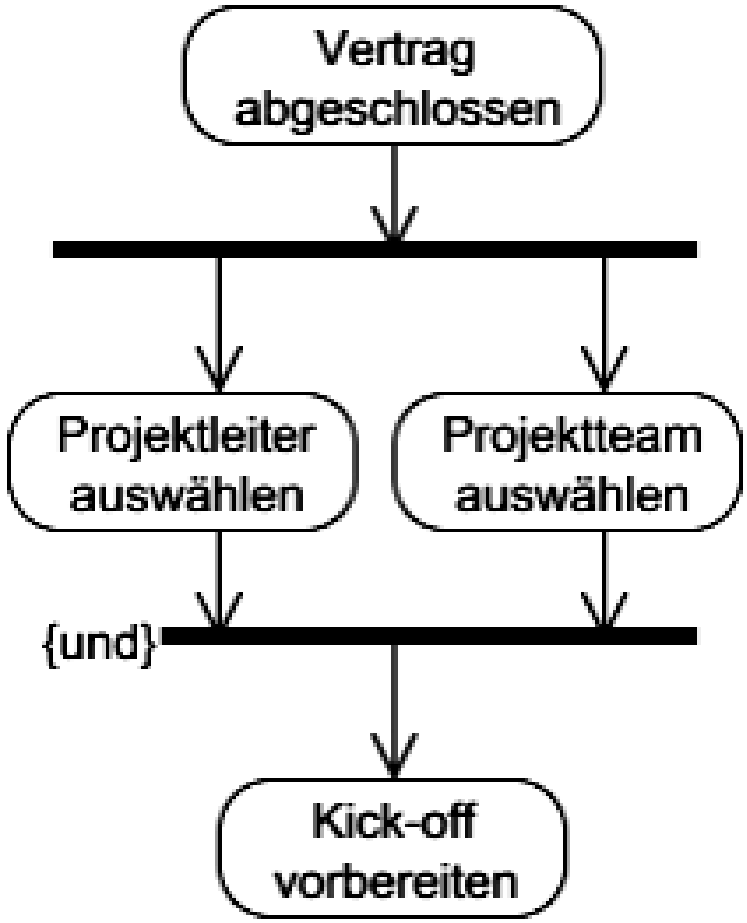
ausgehenden Kanten: Boolesche Bedingungen in eckigen Klammern

Kontrollknoten (Zusammenführung)

Endpunkt (Terminierung)

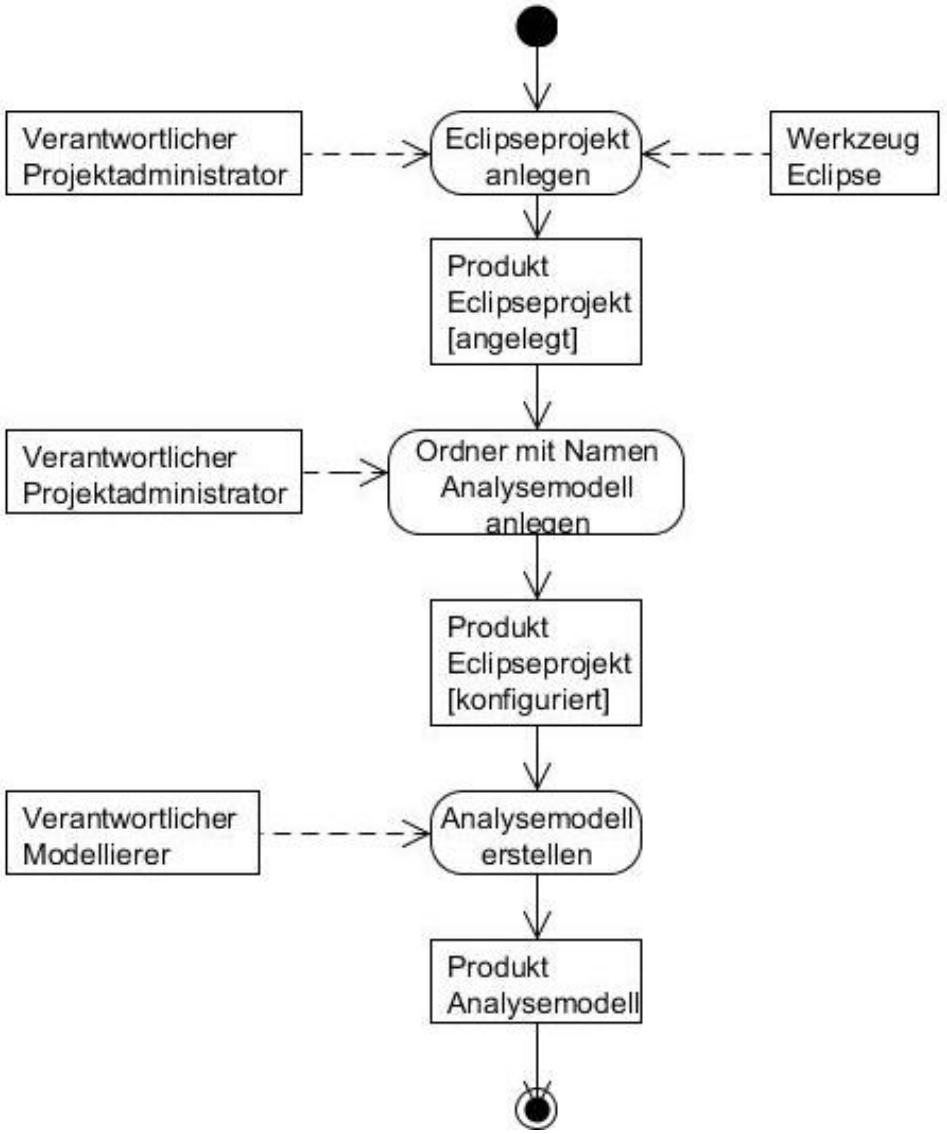
Parallelität in Prozessen

- Waagerechter oder senkrechter Strich steht für mögliche Prozessteilung (ein Pfeil rein, mehrere raus) oder Zusammenführung (mehrere Pfeile rein, ein Pfeil raus)
- Am zusammenführenden Strich steht Vereinigungsbedingung, z. B.
  - {und}: alle Aktionen abgeschlossen
  - {oder}: (mindestens) eine Aktion abgeschlossen
  - {2 von 3} …
- UML 1.1 hatte andere Restriktionen



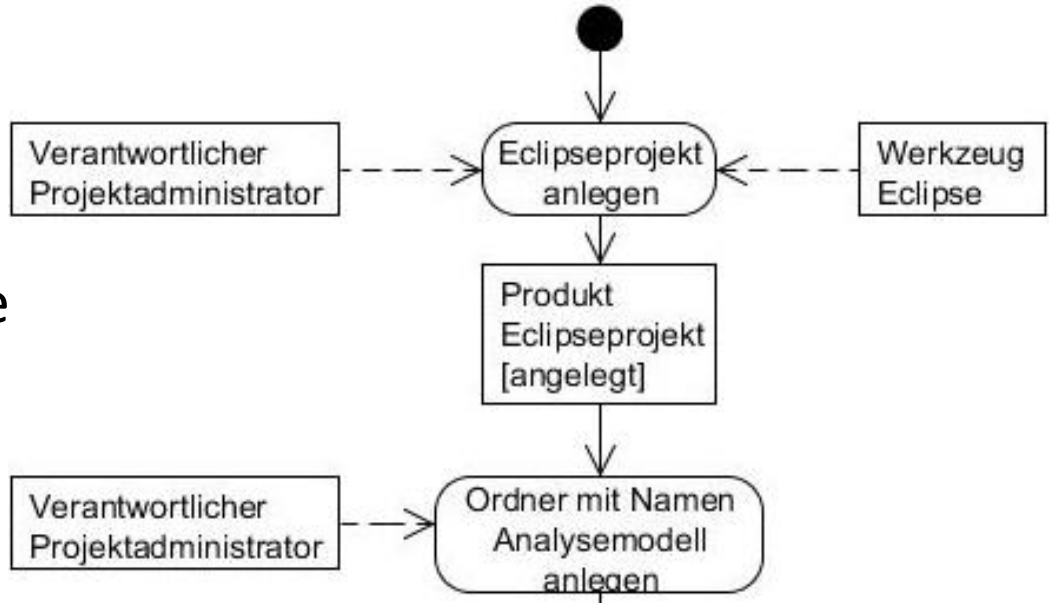
Beteiligte, Produkte, Werkzeuge (optional)

- Beteiligte, Produkte, Werkzeuge werden hier als einfache Datenobjekte modelliert, dabei steht zunächst die Objektart und dann die genaue Bezeichnung
- In eckigen Klammern kann der Zustand eines Objekts beschrieben werden
- neben „Verantwortlicher“ noch „Mitwirkender“ möglich
- auch Entscheidungen können Verantwortliche haben



Anmerkungen

- immer erst ohne "Kästen" modellieren
- häufig alternative Darstellungen für Rollen und Werkzeuge
- Variante: nur Ablauf, Rest in Textdokumentation
- Buch: alle Linien durchgezogen

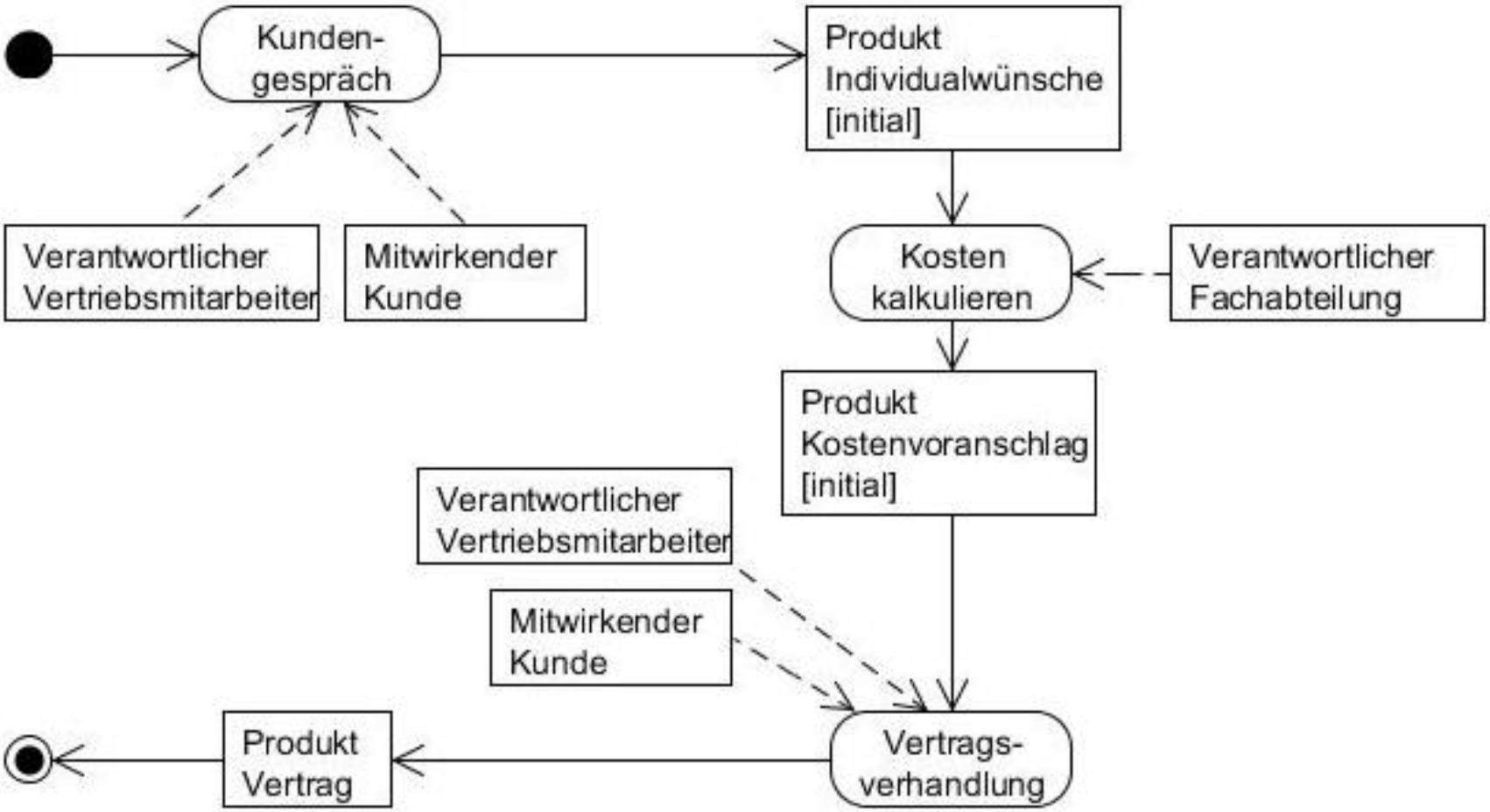


Beispiel: Vertrieb (1/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Zu modellieren ist der Vertriebsprozess eines Unternehmens, das SW verkauft, die individuell für den Kunden angepasst und erweitert werden kann
- Modelle werden wie SW inkrementell erstellt; zunächst der (bzw. ein) typische Ablauf, der dann ergänzt wird
- Typisches Szenario: Vertriebsmitarbeiter kontaktiert Kunden und arbeitet individuelle Wünsche heraus; Fachabteilung erstellt Kostenvoranschlag; Kunde unterschreibt Vertrag; Projekt geht in Prozess Projektdurchführung (nicht modelliert)
- Beteiligt: Vertriebsmitarbeiter, Kunde, Fachabteilung
- Produkt: Individualwünsche, Kostenvoranschlag, Vertrag
- Aktionen: Kundengespräch, Kosten kalkulieren, Vertragsverhandlung

Beispiel: Vertrieb (2/4)



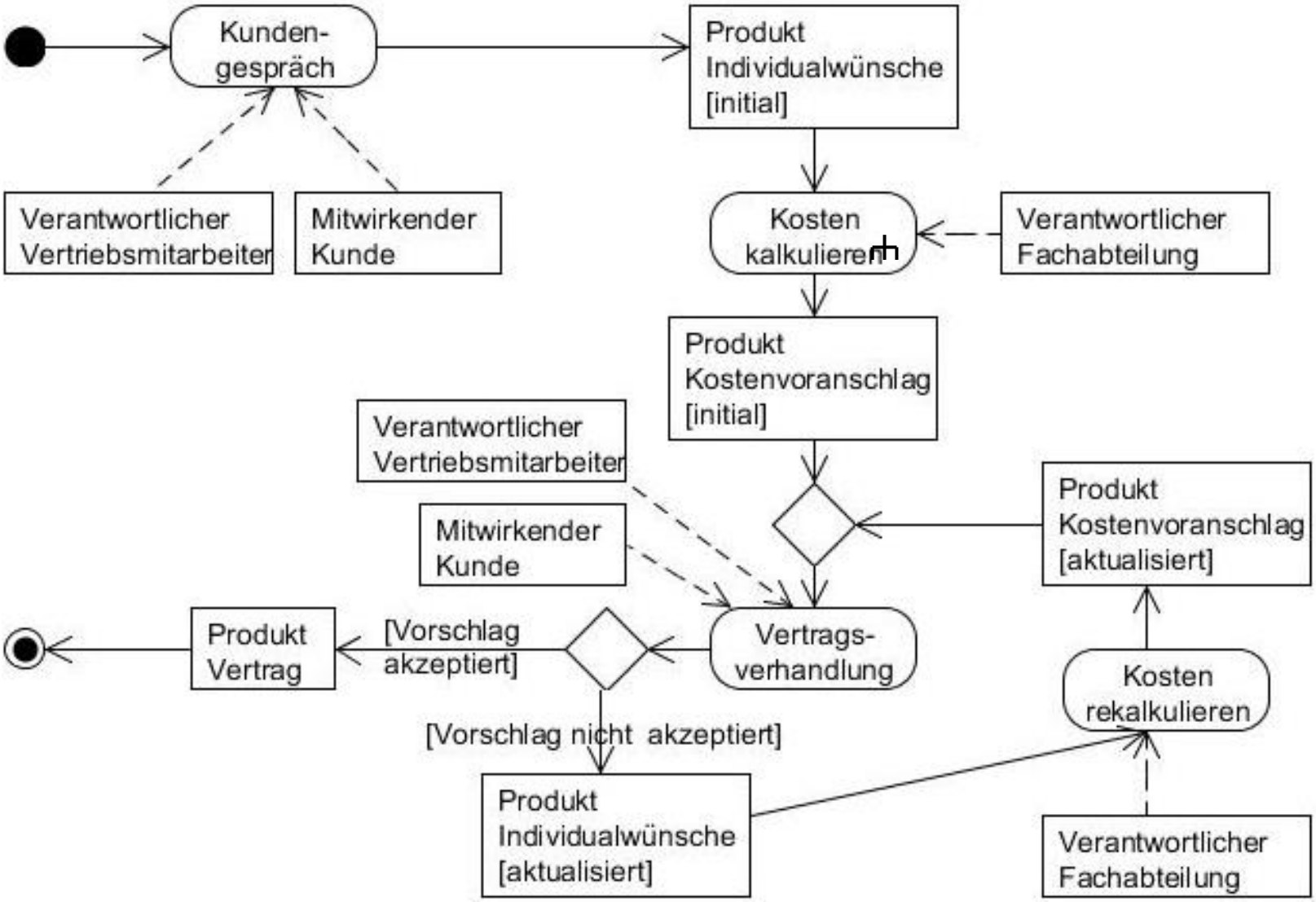
Beispiel: Vertrieb (3/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

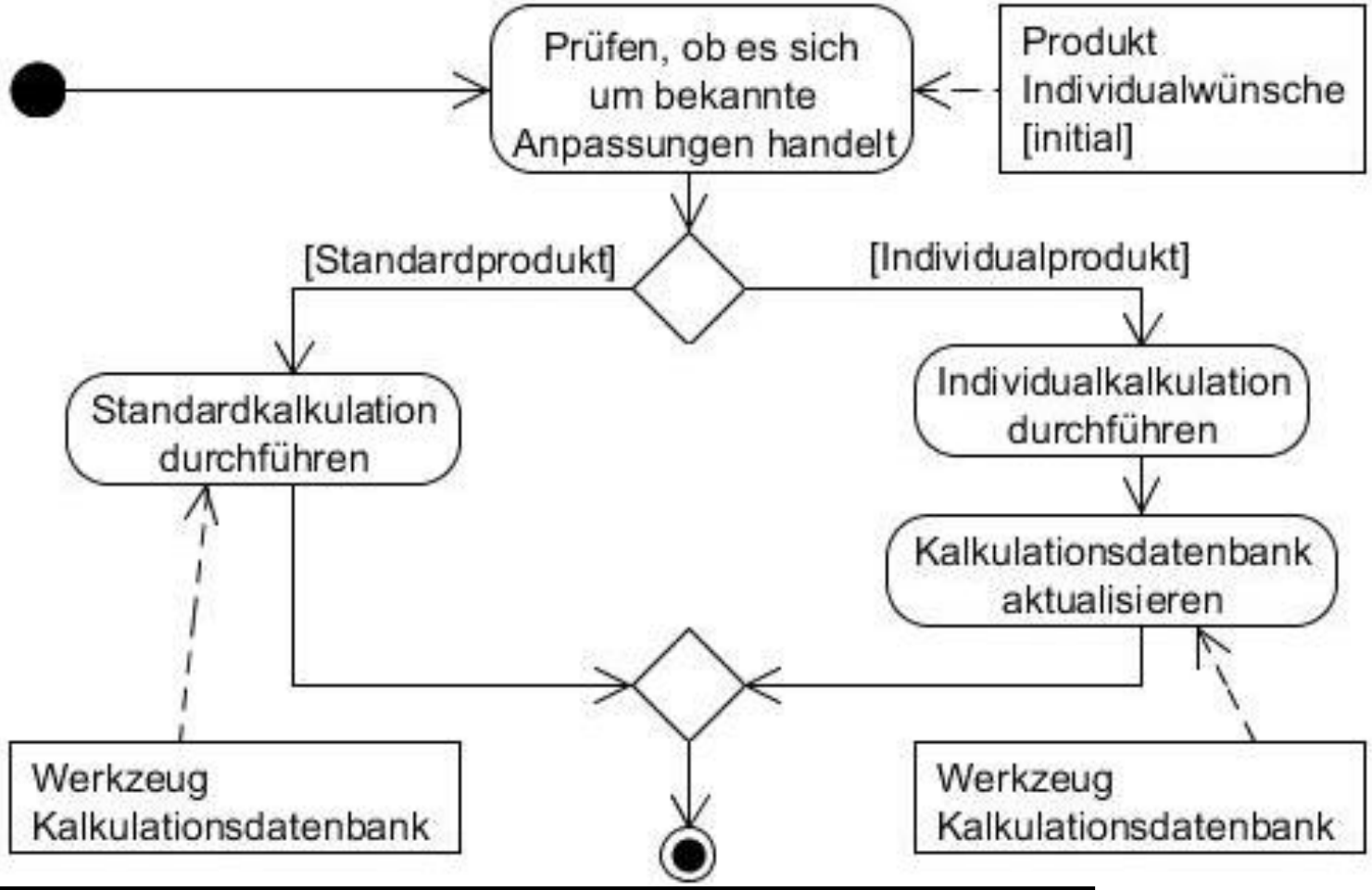
nächster Schritt: Einbau alternativer Abläufe

- Kunde ist am Angebot nicht interessiert
- In den Vertragsverhandlungen werden neue Rahmenbedingungen formuliert, so dass eine Nachkalkulation notwendig wird [nächste Folie]
- Bis zu einem Vertragsvolumen von 20 T€ entscheidet der Abteilungsleiter, darüber die Geschäftsleitung ob vorliegender Vertrag abgeschlossen werden soll oder Nachverhandlungen nötig sind
- Die Fachabteilung hat Nachfragen, die der Vertriebsmitarbeiter mit dem Kunden klären muss

Beispiel: Vertrieb (4/4)



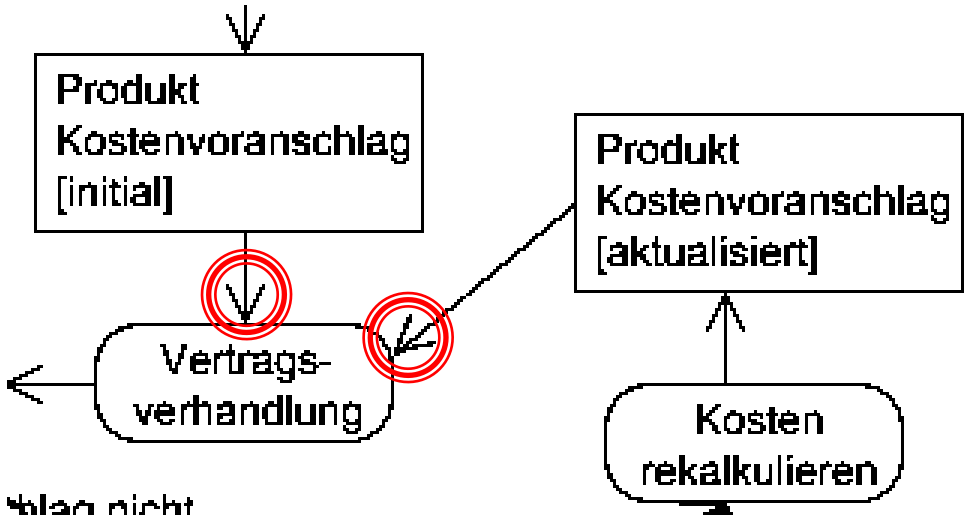
Prozessverfeinerung: Kosten kalkulieren



Anmerkung: Verantwortliche weggelassen, da immer „Projektbegleiter der Fachabteilung“

Modellierungsfalle

- Basierend auf Erfahrungen mit Flussdiagrammen könnte man zu folgender Modellierung kommen



- Dies würde nach UML-Semantik bedeuten, dass für die Aktion Vertragsverhandlung zwei Kostenvorschläge (initial und aktualisiert) vorliegen müssten
- Wenn verschiedenen Wege zu einer Aktion führen sollen, muss vor der Aktion ein Zusammenführungs-Kontrollknoten stehen

weitere Modellierungssprachen

Business Process and Model Notation (BPMN)

- OMG, Business Process Model and Notation (BPMN), Version 2.0.2, OMG Document Number: formal/13-12-09, über https://www.omg.org/spec/BPMN/, Januar 2014
- OMG, BPMN 2.0 by Example, Version 1.0, OMG Document Number: dtc/2010-06-02, http://www.omg.org/cgi-bin/doc?dtc/10-06-02.pdf , Juni 2010

Erweiterte ereignisgesteuerte Prozessketten (eEPK)

- A.-W. Scheer, ARIS — Vom Geschäftsprozess zum Anwendungssystem, Springer, Berlin Heidelberg, 2002
- in D entwickelt, Scheer GmbH, Werkzeug ARIS (Architektur integrierter Informationssysteme)

gemeinsam: deutlich mehr graphische Symbole, mächtiger, aber am Anfang schwerer lesbar; Kern wie bei Aktivitätsdiagrammen

Problem Lesbarkeit

- Diagramme können leicht komplex werden

Lösungsmöglichkeiten:

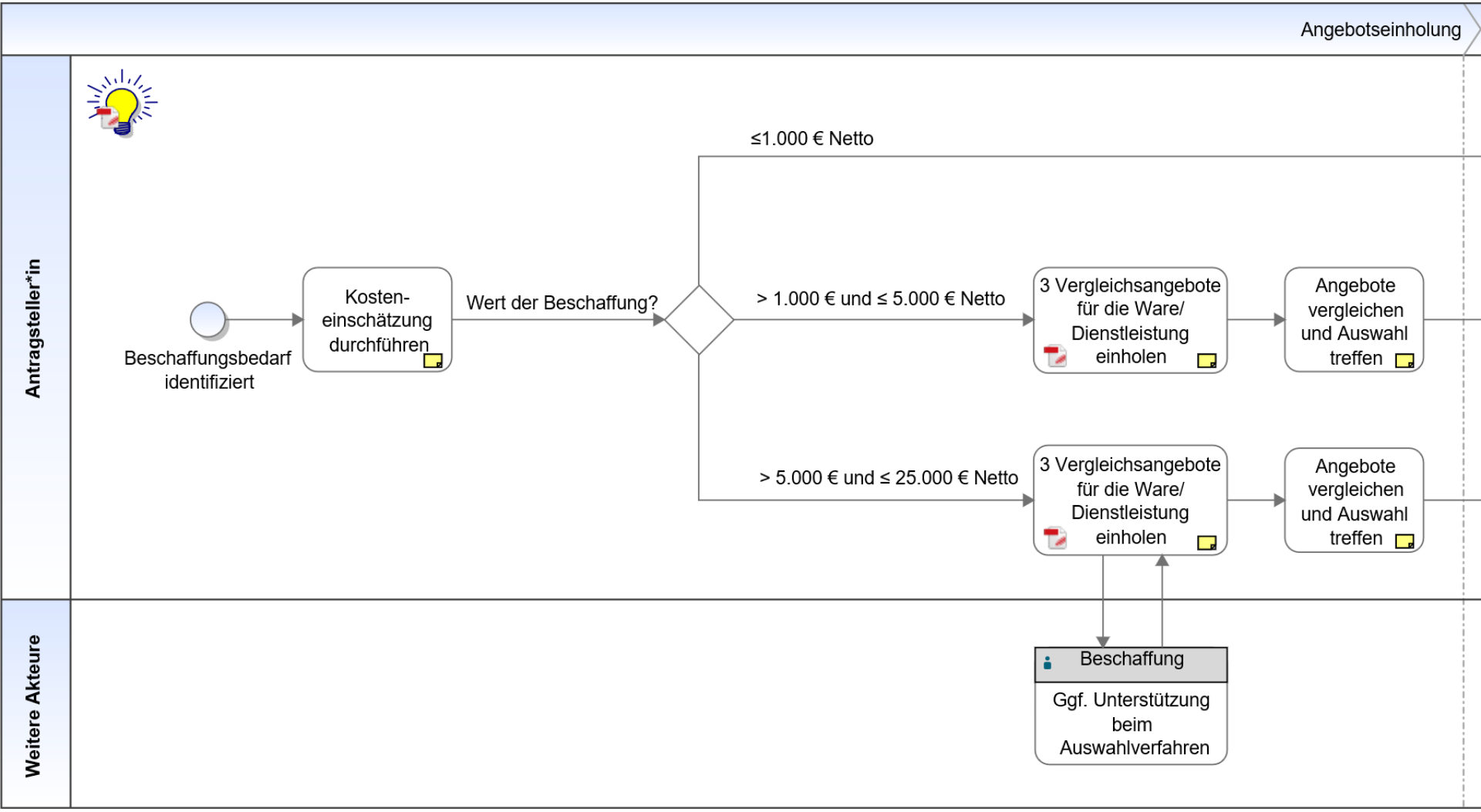
- Verteilung von Diagrammen auf mehrere Seiten mit Ankerpunkten
- Verzicht, alle Elemente in einem Diagramm darzustellen (z. B. Produkte weglassen; dies nur in der immer zu ergänzenden Dokumentation erwähnen)
- Diagramme hierarchisch gestalten; eine Aktion kann durch ein gesamtes Aktivitätsdiagramm verfeinert werden, z. B. ist „Kosten kalkulieren“ eigener Prozess; dies sollte im Modell sichtbar werden

Problem Abstraktionsgrad

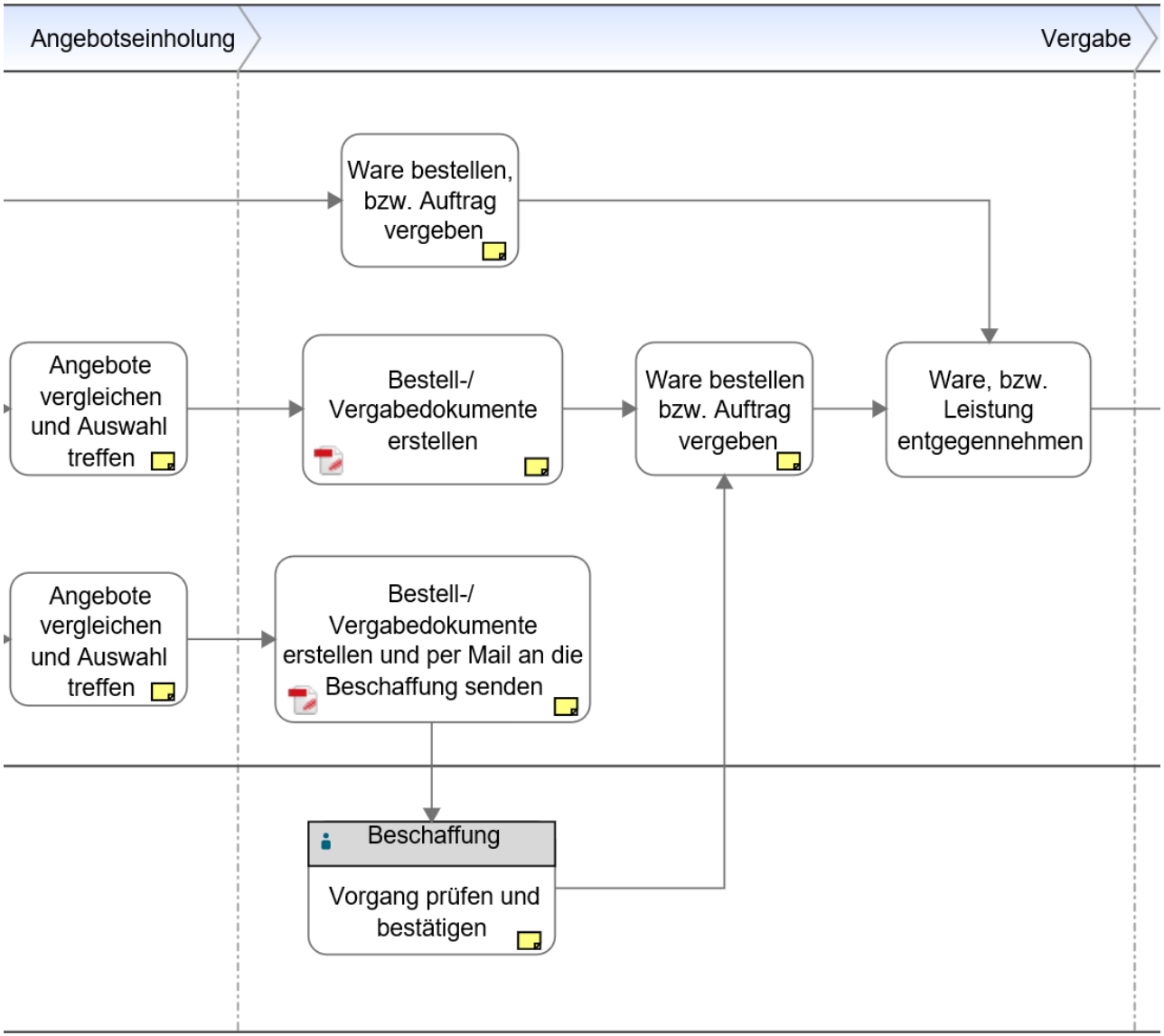
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Frage: Wann nur eine Aktion, wann mehrere Aktionen
- Indikator: Mehrere Aktionen zusammenfassen, wenn
  - nur ein Produkt entsteht, das ausschließlich in diesen Aktionen benötigt wird („lokale Variable“)
  - oder diese von nur einer Person bearbeitet werden
- Typischerweise Prozesshierarchie:
  - Unternehmensebene; d.h. ein Diagramm für jeden Prozess der Kern-, Management- und Supportprozesse
  - Prozessebene: Verfeinerung des Prozesses, so dass alle nur intern sichtbaren Rollen und Produkte sichtbar werden
  - Arbeitsprozess: Individuelle Beschreibung der Arbeitsschritte einer Rolle für eine/ mehrere Aktionen
- Probleme: Flexibilität und Akzeptanz

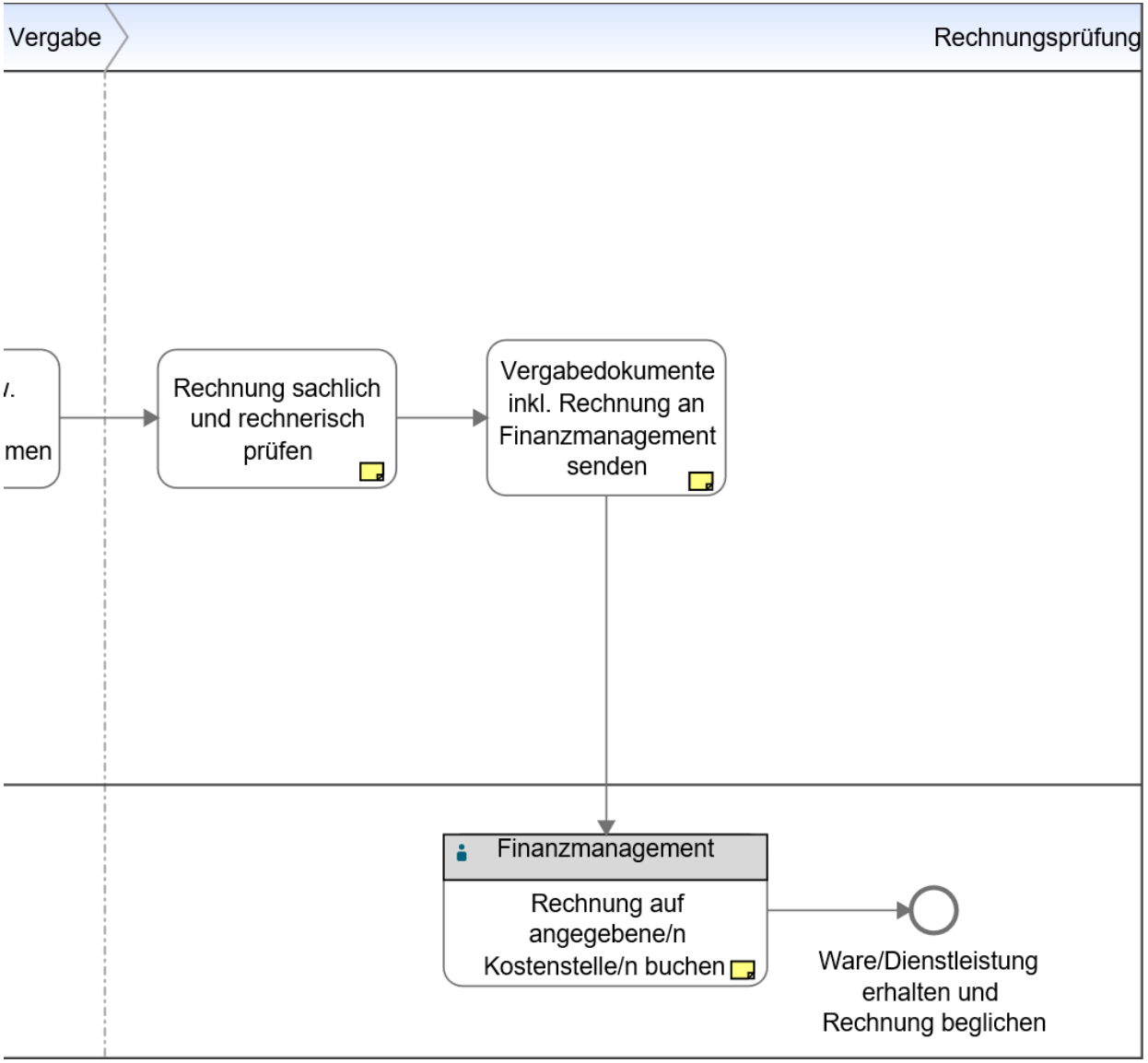
Beispiel: HS-Osnabrück Beschaffung bis 25 T€ (1/4)



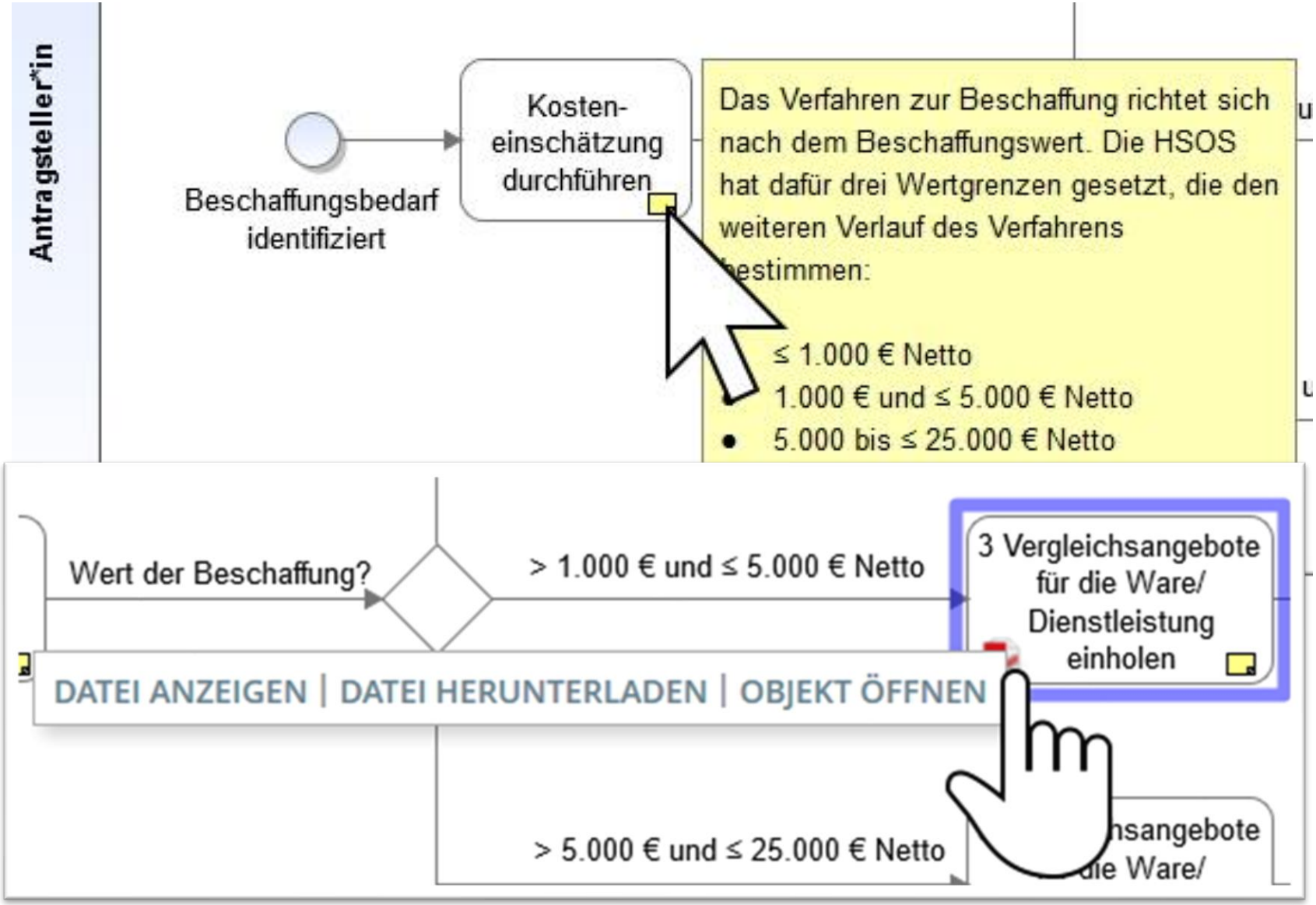
Beispiel: HS-Osnabrück Beschaffung bis 25 T€ (2/4)



Beispiel: HS-Osnabrück Beschaffung bis 25 T€ (3/4)



Beispiel: HS-Osnabrück Beschaffung bis 25 T€ (4/4)



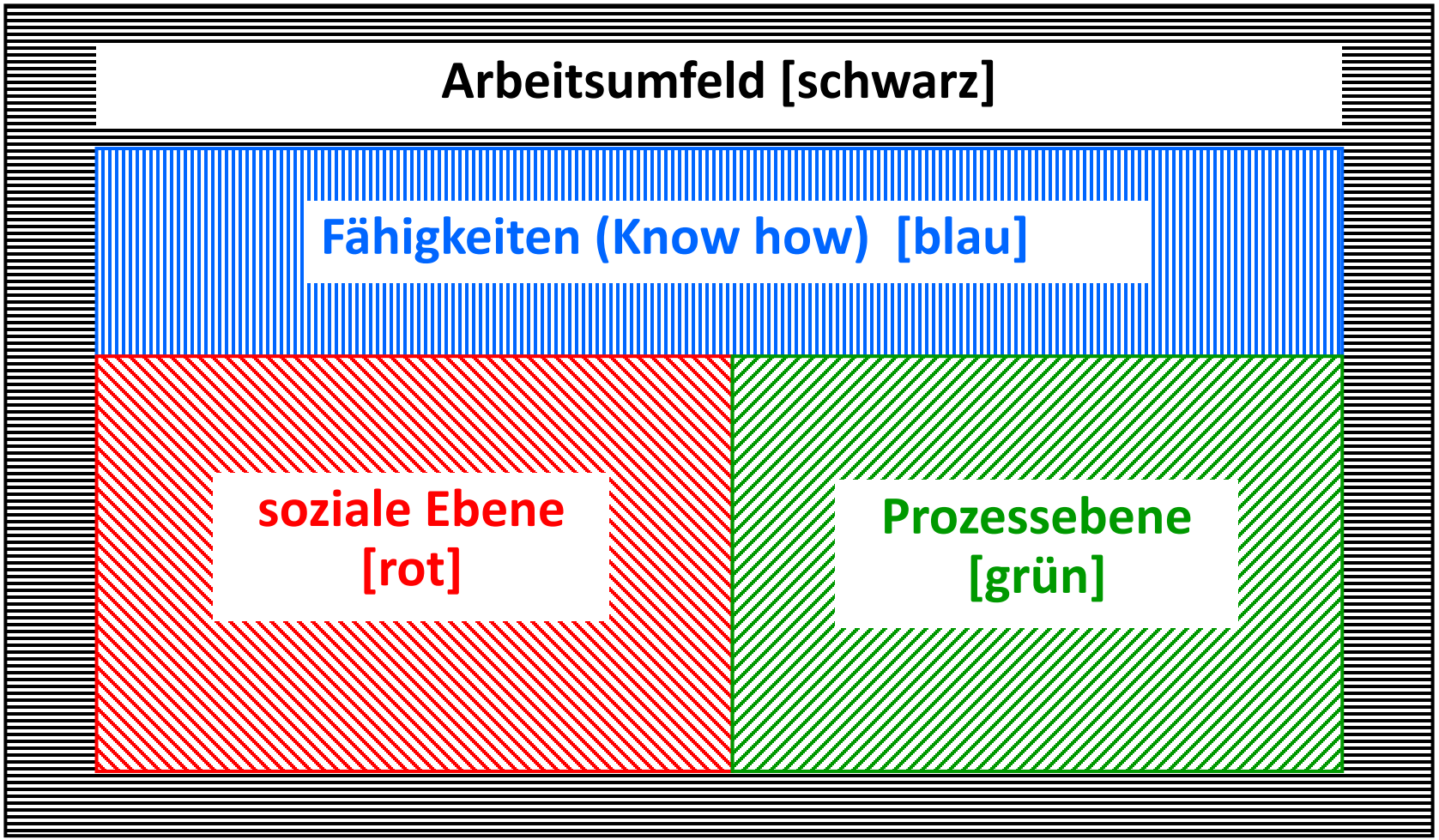
Prozessnutzung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Modellierung muss von allen Beteiligten verstanden, akzeptiert und genutzt werden
- Prozesse haben oft Interpretationsspielraum; für kritische Prozesse ein Prozessbegleiter sinnvoll
- Einhaltung von Prozessen muss geprüft werden
- Prozesse müssen zur Unternehmensstruktur passen; sind bei Umstrukturierungen anzupassen (z. B. Beschaffung ausschließlich über eine zentrale Abteilung)
- Prozesse müssen auf Sinnhaftigkeit und Aktualität geprüft werden (funktionieren Sie, entstehen woanders Defizite, sind Aktualisierungen [z. B. Geldbetragsgrenzen] notwendig
- QM der QM eigenständiger, wichtiger Prozess

Ganzheitliche Projekt- Unternehmenssicht

- Grenzen der Prozessorganisation, weitere Q-Einflussfaktoren



Ebenen des ganzheitlichen Projektmanagements

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Fähigkeiten: was können Projektmitglieder, welche Erfahrungen haben sie
- *Prozessebene: wie sind Arbeitsprozesse organisiert, sind Rollen klar abgegrenzt*
- Arbeitsumfeld: welche Werkzeuge gibt es, wie sind Arbeitsplätze organisiert
- soziale Ebene: wer kann wie mit wem

Grundregel: Probleme der einen Ebene können nicht auf der anderen Ebene gelöst werden

- wenn Leute nicht miteinander können, helfen Prozessregelungen wenig
- fähige Mitarbeiter werden durch schlechte Werkzeuge, große räumliche Trennung, unsaubere Toiletten ausgebremst
- neue Werkzeuge benötigen Schulungen und Einarbeitung

Prozesse spielen auf allen Ebenen eine Rolle

- Firmenumfeld [schwarz]
  - Beschaffung von Büromöbeln
  - Gestaltung des Arbeitsumfelds
- Fähigkeiten [blau]
  - Ausbildungsplanung
  - Wissensverwaltung
- soziale Ebene [rot]
  - Konfliktbewältigung
  - betriebliche Mitbestimmung
- Prozessebene [grün]
  - Prozessdokumentation
  - Prozessverbesserung

Prozesse unterstützen, können aber nur zur Lösung Probleme anderer Ebenen beitragen

Prozesse lösen nicht Probleme anderer Ebenen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Prozessebene:

- Zentral für die Prozessqualität (z.B. Schnittstellen) und kontinuierliche Anpassung
- kann aber nicht Probleme anderer Ebenen lösen, z.B.:
  - [schwarz] Firmenumfeld: zu wenig Speicherplatz => Regeln, wer, was auf welchem Server speichern darf
  - [blau] Fähigkeiten: kein Aufwandsschätz-Know-how => Vorgabe im Prozesshandbuch, Aufwand ist nach folgenden Aspekten ... abzuwägen
  - [rot] soziale Ebene: stark konkurrierende Projektleiter => Prozess, der die Zuordnung von Mitarbeitern zu Projekten nach gewissen Prioritäten regelt

Grundregel: Probleme müssen auf Ebene des Ursprungs gelöst werden, Prozessebene kann unterstützen

Beispiel einer Prozesseinführung

- Grundsätzlich ist es für jede Firma überlebenswichtig, ihre Prozesse zu kennen, zu optimieren und permanent den Gegebenheiten anzupassen
- Ansatz: Ziel-Vorgaben --- IST-Analyse --- SOLL-Konzept --- Umsetzung des SOLLs --- Prüfung (durch Kennzahlen)

zur Erinnerung:

- Frage: Muss jeder Prozess individuell von jeder Firma neu „entdeckt“ werden?
- Antwort: Nein, für verschiedenste Prozessbereiche gibt es bereits Prozessmodelle („best practice“), die genutzt werden können, dabei ist zu beachten:
  - Auswahl: Passt der Prozess zu meiner Firma
  - Umsetzbarkeit: Übertragung des meist sehr allgemeinen Modells auf Firmengegebenheiten

Worst-Case der Prozesseinführung (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- zentrale Forderung: Prozessverbesserungen müssen von möglichst allen Mitarbeitern getragen werden; sie müssen aktiv zur Optimierung beitragen
- häufiger alternativer Ansatz:
  - Firma erkennt, dass Gewinne einbrechen
  - typischer „Problemlösungsansatz“: wir arbeiten wohl zu ineffizient, wir brauchen eine Prozessverbesserung
  - Informationsfluss in der Firma: uns geht es schlecht, es sollen Mitarbeiter fliegen
  - Geschäftsführung kauft teure externe Berater („Armani-Gang“) zur Prozessoptimierung ein

Worst-Case der Prozesseinführung (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Situation: Armani-Gang weiß, dass erfolgreiche Prozesseinführung ohne Mitarbeiter nicht möglich ist, Wahrscheinlichkeit der Blockade durch Mitarbeiter aber groß
- Prozessmodell wird von Armani-Gang mit Fokus auf kurzfristige Kostenreduktion entwickelt und von der Geschäftsführung „verabschiedet“
- kurzfristige Ergebnisse:
  - Entlassung von x% der Mitarbeiter
  - gut dokumentiertes, auf aktuelle Geschäftssituation optimiertes Prozessmodell
  - flexible (meist fähige) Mitarbeiter verlassen Firma während der Prozessumgestaltung
  - restliche Mitarbeiter lassen Änderungen mit sich geschehen, reagieren mit „Dienst nach Vorschrift“
  - Schlechtes Jahresergebnis, wegen Beratungskosten

Worst-Case der Prozesseinführung (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- typische mittel- und langfristige Ergebnisse:
  - Firmenergebnisse verbessern sich im Folgegeschäftsjahr, da es weniger Ausgaben für Mitarbeiter gibt
  - Prozessmodell wird von vielen gelobt [war teuer und kommt von oben] und von wenigen gelebt
  - mittelfristig gerät die Firma in größere Schieflage, da z.B.
    - keine neuen Innovationen stattfinden, da kreative Leute die Firma verlassen haben
    - da sich Mitarbeiter nicht für die Firma verantwortlich fühlen, wenn sie das Prozessmodell erfüllen
    - das Prozessmodell nicht neuen Marktlagen angepasst werden kann/ angepasst wird
- aus Sicht des QM wichtig: Mitarbeiter, die einen vergleichbaren Ablauf miterlebt haben, sind kaum für Prozessverbesserungen zu motivieren, können Hemmschwellen für interne Verbesserungsansätze werden

2.2 Beispiele für Prozessstandards

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- Übersicht
- Beispielstandard CMMI
- Gemeinsamkeiten von Standards
- Total Quality Management (TQM)
- (bei Interesse Hausarbeiten möglich; weiteren Standard vorstellen und Nutzung in SW-Unternehmen oder SW-Entwicklung vorstellen)

QM-Standards : ISO 9000 unvollständige Übersicht

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- zentral auf alle Prozesse eines Unternehmens zielend
- ISO 9000 – Familie
- ISO 9000: generelle Prinzipien eines QM-Systems, Begriffsdefinitionen
- ISO 9001: was ist notwendig für ein Unternehmen um ein QM-System aufzusetzen (Zertifizierungsgrundlage, klassisches Plan – Do – Check – Act)
- ISO 9004: ergänzt 9004 wie QM-System umgesetzt werden kann

- ISO 19011: Anforderungen für das Auditieren von Management Systemen

Beispiel: Projektmanagement und Standards

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- IPMA/GPM: International Project Management Association Gesellschaft für Projektmanagement (Eu, D)
- DIN 69901: Deutsche Industrie-Norm, beschreibt Grundlagen, Prozesse, Prozessmodelle, Methoden, Daten, Datenmodelle und Begriffe im Projektmanagement, stark durch die GPM geprägt (D)
- PMI®: Project Management Institute (USA), gibt nationale Dependancen, Schwerpunkt auf Projektmanagement-prozessen (USA)
- ISO 21500:International Standard Organisation, "Guide to Project Management", am PMI orientiert (seit Ende 2012)
- PRINCE2: Projects in Controled Environments Version 2, gibt deutschen Verein dazu, standardisierten Projekten und der Sammlung Best Practices (UK, NL)

Prozess-Standard für QS (Ausschnitt)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

ISTQB, International Software Testing Qualifications Board, https://www.istqb.org/ , Zertifizierung als

- Foundation Level
- Foundation Level Automotive Software Tester
- Foundation Level Usability Testing
- Foundation Level Performance Testing
- Foundation Level Mobile Application Testing
- Foundation Level Acceptance Testing
- Foundation Level Agile Tester
- Foundation Level Model-Based Tester
- Advanced Level
- Advanced Level Security Tester
- Advanced Level Test Automation Engineer
- Expert Level

weitere Prozess-Standards

- IREB , International Requirements Engineering Board, https://www.ireb.org/de

  „Certified Professional for Requirements Engineering (CPRE) steht für ein dreistufiges Zertifizierungsmodell von IREB. Es richtet sich an Personen, die sich mit Business Analyse, Requirements Engineering oder Testing befassen und einen hohen Anspruch an die Qualität ihrer Arbeit haben.“

- ITIL, Information Technology Infrastructure Library : „best practices for IT Service Management“, Fokus auf Zusammenarbeit von
  - Organisation und Menschen
  - Information und Technologie
  - Partner und Lieferanten
  - Wertestrom und Prozesse

IEEE Standards für Software Entwicklung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- SQAP – Software Quality Assurance Plan IEEE 730
- SCMP – Software Configuration Management Plan IEEE 828
- STD – Software Test Documentation IEEE 829
- SRS – Software Requirements Specification IEEE 830
- SVVP – Software Validation & Verification Plan IEEE 1012
- SDD – Software Design Description IEEE 1016
- SPMP – Software Project Management Plan IEEE 1058
- Software Reviews and Audits IEEE 1028
- Software Life Cycle Processes IEEE 1074
- Systems and Software Engineering – Systems Life Cycle Processes IEEE 15288

siehe [Wal11] Ernest Wallmüller, Software Quality Engineering, 3. Auflage, Hanser, München, 2011

Weitere Standards im Gebiet Software Entwicklung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- International Software Architecture Qualification Board (iSAQB)
- V-Modell für IT-Entwicklungsprojekte in Deutschland (Behörden)
- ISO/IEC 25000 Software-Engineering – Qualitätskriterien und Bewertung von Softwareprodukten (SQuaRE) (250xx)
- alt: DOD-STD-2167A "Defense Systems Software Development“
- neuer: MIL-STD-498, Ziel: System Development Life Cycle
- aktuell: DO-178C, Software Considerations in Airborne Systems and Equipment Certification is the primary document; genutzt u. a. von FAA (Federal Aviation Administration) , EASA (European Union Aviation Safety Agency)
- Capability Maturity Model Integration (CMMI), USA

QM-Frameworks

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

statt sich mit konkreten Prozessen und Vorgaben von Prozessen zu beschäftigen, werden generelle Konzepte und Ideen vorgeben, z. B.

- Lean Management
  - Ziel verschlangte evtl. vereinfachte Prozesse, die schneller / effizienter zum Ziel führen
- Six Sigma
  - quantitative Methode (TODO: Maße entwickeln) mit Zielvorgaben und Fehlerabweichung
- Total Quality Management [s. spätere Folien]
  - Qualität durch Kunden(anforderungen) bestimmt
  - Management muss TQM leben und vorantreiben
  - Einbeziehung aller Mitarbeiter

Zertifizierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Unternehmen können zertifiziert werden
- typisch: Aussteller der QM-Norm zertifiziert Unternehmen, die dann QM-Norm zertifizieren dürfen
- oft wird für Zertifizierung externe Beratung benötigt, damit Dokumente entstehen, die Zertifizierer abnehmen können
- Zertifizierung läuft meist nach 3, 5, 7, … Jahren ab; Rezertifizierung deutlich nicht so aufwändig
- QM-Zertifikat kostet (viel) Geld; oft für Aufträge notwendig

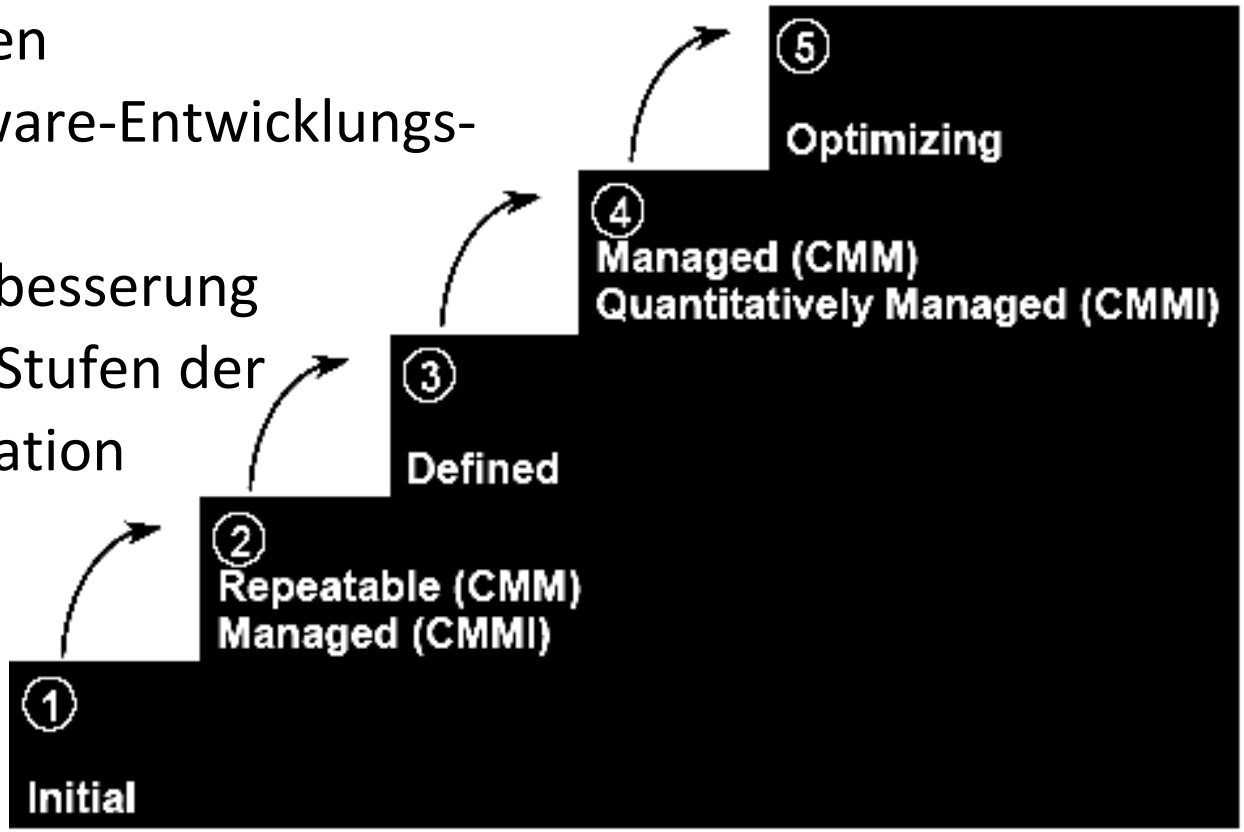
- nach anderen Normen können Mitarbeiter zertifiziert werden, müssen dann Anwendung des Wissens nachweisen, oft auch Rezertifizierung notwendig
- oft wichtig für Freelancer

Prozesse als Firmenqualitätsmaßstab: CMM[I]

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- 1993, ursprünglich Capability Maturity Model (CMM)
- Auftrag durch das DoD
- Hilfsmittel zur Beurteilung der Leistungsfähigkeit von Software-Lieferanten
- Reifegrad der Software-Entwicklungs-prozesse ermitteln
- gezielte Prozessverbesserung
- CMM(I) definiert 5 Stufen der Reife einer Organisation
- Version 1.3 (2010)



Stufe 1: Initial

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Merkmal: Keine stabile Umgebung, ad-hoc Durchführung
- Fokus: gute Mitarbeiter
- Unvorhersagbares Ergebnis: bei erneuter Durchführung des Projektes würde alles ganz anders laufen
- Brandbekämpfung statt Planung, Planabweichung in Krisensituationen
- Gelingen des Projektes nur durch äußersten Einsatz aller Beteiligter, hängt an Einzelpersonen
- Risikobehaftet, unplanbar und ineffizient

Stufe 2: Wiederholbar (repeatable/managed)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Merkmal: Gleiche Anforderungen ergeben gleiche Resultate
- Fokus: Projektmanagement
  - Richtlinien für Software-Projektmanagement existieren und werden umgesetzt
  - Überwachung von Zeit, Kosten, Produktqualität
- Basierend auf früheren Projekten
- Softwarestandards, Konfigurationsmanagement
- Stabiler Prozess

Stufe 3: Definiert (defined)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Merkmal: Wohldokumentierte Prozesse
- Fokus: Organisationsunterstützung
  - Stabile und wiederholbare SW-Entwicklungs- und SW-Projektmanagement-Prozesse
  - Abnahmekriterien, Standards, QS-Maßnahmen definiert und dokumentiert
  - Möglichkeit der Anpassung von Standards
- Rolle des Softwareprozess-Verantwortlichen
- Weiterbildungsmaßnahmen eingeplant
- Regelmäßige Expertenbegutachtung

Stufe 4: Beherrscht (managed/quality managed)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Merkmal: Quantitativ messbare Prozesse
- Fokus: Produkt- und Prozessqualität
  - Leistungsmessung von Produktivität und Qualität
  - Metriken für Software
- Messbare, vorhersagbare Prozesse in definierten Grenzen
- Messbare, vorhersagbare Produktqualität
- Steuerungsmöglichkeit bei Schwankungen

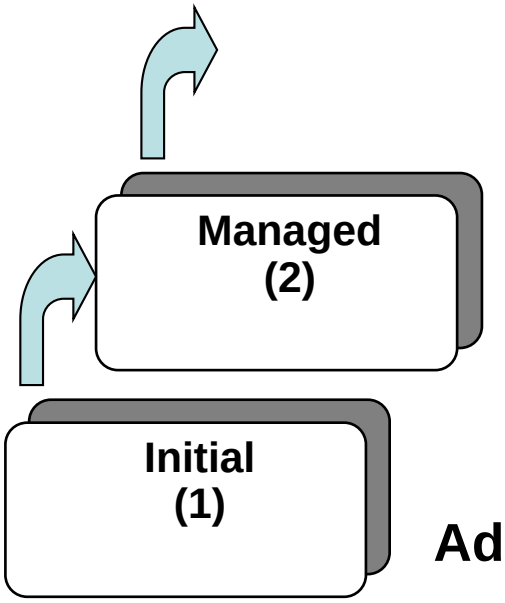
Stufe 5: Optimierend (optimizing)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Merkmal: Gesamte Organisation fokussiert auf kontinuierliche Prozessverbesserung
- Fokus: Ständige Evaluation und Einführung neuer Technologien und Verbesserungsmöglichkeiten
  - Möglichkeit zur Stärken/Schwächenanalyse
  - Proaktive Fehlervermeidung
  - Dauernde Implementierungsmöglichkeit für Verbesserungen der Softwareprozesse (direkte Einbringung von Verbesserungsvorschlägen)
- Verbesserungen auf der Meta-Ebene

Prozesse der Stufen des CMMI (1/2)

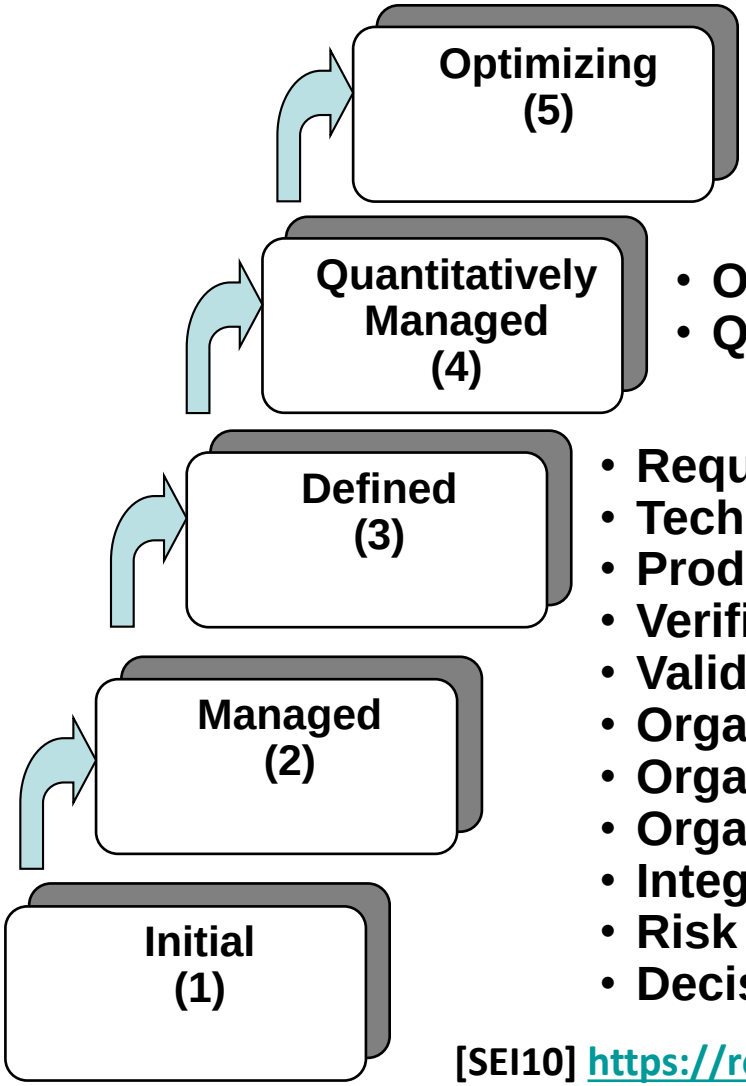
- Maturity Level (Reifegrade): Stufen in der Entwicklung einer Organisation auf dem Weg zu optimalen Softwareprozessen
- Reifegrad ist in Key Process Areas unterteilt, diese identifizieren eine Menge von zusammenhängenden Aktivitäten, welche bestimmte *Ziele* verfolgen.



- **Requirements Management (REQM)**
- **Project Planning (PP)**
- **Project Monitoring and Control (PMC)**
- **Supplier Agreement Management (SAM)**
- **Measurement and Analysis (M&A)**
- **Process and Product Quality Assurance (PPQA)**
- **Configuration Management (CM)**

Ad hoc, chaotic processes

Prozesse der Stufen des CMMI (2/2)



- Organizational Innovation and Deployment (OID)
- Causal Analysis and Resolution (CAR)

- Organizational Process Performance (OPP)
- Quantitative Project Management (QPM)

- Requirements Development (RD)
- Technical Solution (TS)
- Product Integration (PI)
- Verification (VER)
- Validation (VAL)
- Organizational Process Focus (OPF)
- Organizational Process Definition (OPD)
- Organization Training (OT)
- Integrated Project Management (IPM)
- Risk Management(RSKM)
- Decision Analysis and Resolution (DAR)

[SEI10] https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=9661

Reifegradbestimmung durch Fragenkatalog (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Beispielfragen zur Stufe 2

- Werden quantifizierte Vergleiche des Ist-Personaleinsatzes mit dem Soll-Personaleinsatz durchgeführt?
- Werden Statistiken über den Testfortschritt und die Lieferung der Softwarekomponenten aufgezeichnet?
- Werden Statistiken erstellt, die den Zusammenhang zwischen dem Umfang / Inhalt eines Software-Release und dem Aufwand aufzeigen?
- Werden Statistiken über die geplanten und tatsächlich aufgetretenen Fehler verglichen?
- Werden Statistiken über Softwareeinheiten in der Modul-/Programmtestphase erstellt?
- Wird die Speicherplatzbelegung gemessen und aufgezeichnet?
- Wird der Datendurchsatz gemessen und aufgezeichnet?
- Wird die Performance der Hardware gemessen und aufgezeichnet?
- Werden Softwareproblemberichte, die aus der Testphase resultieren, bis zu ihrem Abschluss verfolgt?

Reifegradbestimmung durch Fragenkatalog (2/2)

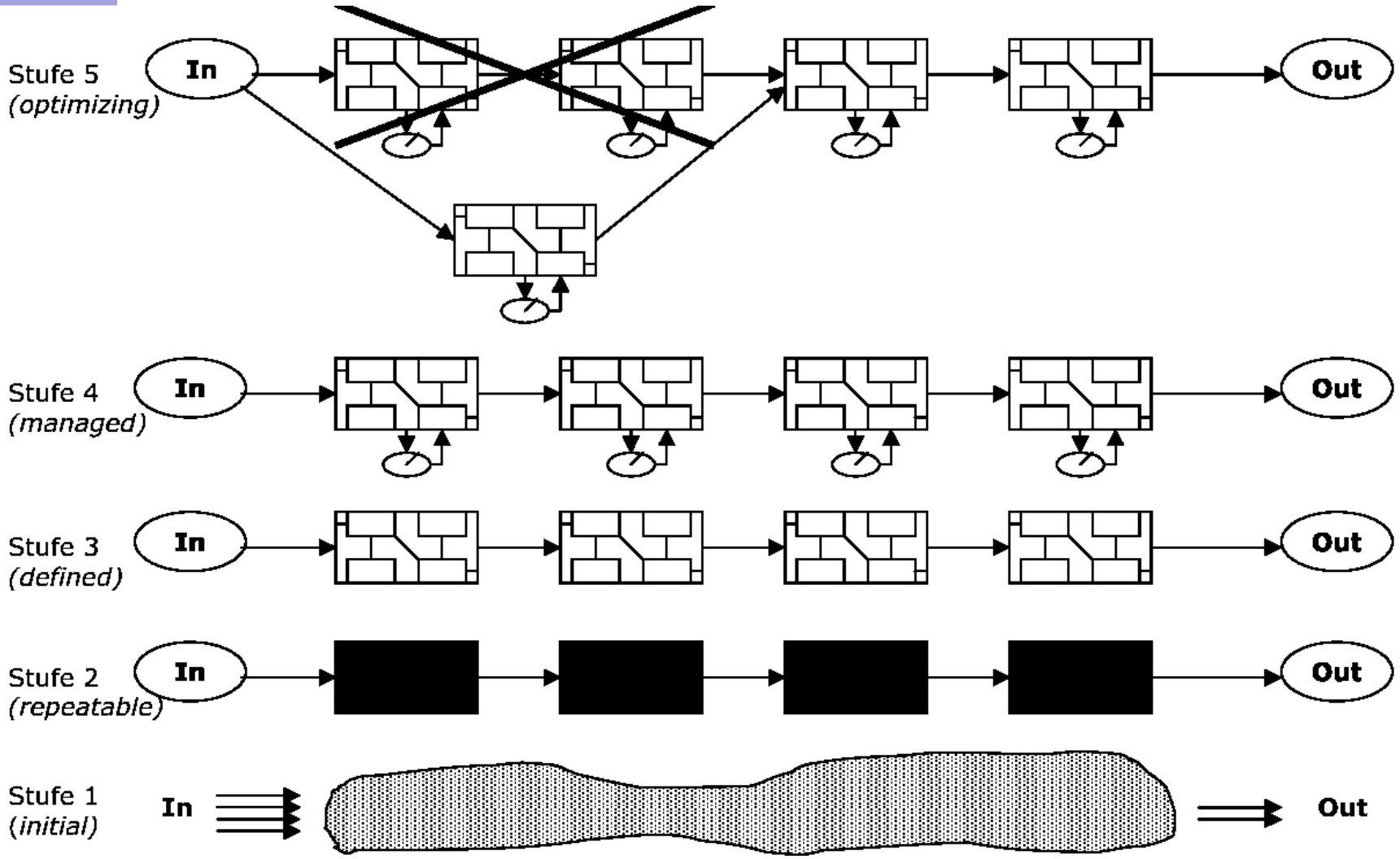
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- L2->L3 Werden Aufzeichnungen über die Größe jedes Softwarekonfigurationselements geführt?
- L2->L3 Werden Statistiken über Codefehler und Testfehler gesammelt?
- L3->L4 Werden Statistiken über Softwaredesignfehler gesammelt?
- L3->L4 Werden Maßnahmen, die aus Design-Reviews resultieren, auch nachgewiesen umgesetzt?
- L3->L4 Werden die Maßnahmen, die aus Code-Reviews resultieren, auch nachgewiesen umgesetzt?
- L4->L5 Wird die Anzahl der Designfehler geschätzt und mit der Anzahl der tatsächlich aufgetretenen Fehler verglichen?
- L4->L5 Wird die Abdeckung des Designs und Codes durch Reviews gemessen und aufgezeichnet?
- L4->L5 Wird die Testabdeckung (C0, C1) in jeder Testphase gemessen und aufgezeichnet?

Überblick: Verbesserungen mit dem CMM

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video



Beispiel: Project Planning (1/5) [SEI10]

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Purpose: The purpose of Project Planning (PP) is to establish and maintain plans that define project activities.

- Specific an Generic Goals
  - SG 1 Establish Estimates
  - SG 2 Develop a Project Plan
  - SG 3 Obtain Commitment to the Plan

- (kein Generic Goal)
- Specific Goals werden mit Specific Practises verfeinert
- Hinweis: viele Detailteste weggelassen

Beispiel: Project Planning (2/5) [SEI10]

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

SG 1 Establish Estimates

- Estimates of project planning parameters are established and maintained.

SP 1.1 Estimate the Scope of the Project

- Establish a top-level work breakdown structure (WBS) to estimate the scope of the project.

SP 1.2 Establish Estimates of Work Product and Task Attributes

- Establish and maintain estimates of work product and task attributes.

SP 1.3 Define Project Lifecycle Phases

- Define project lifecycle phases on which to scope the planning effort.

SP 1.4 Estimate Effort and Cost

- Estimate the project's effort and cost for work products and tasks based on estimation rationale.

Beispiel: Project Planning (3/5) [SEI10]

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

SP 1.1 Estimate the Scope of the Project

- The WBS evolves with the project. Initially a top-level WBS can serve to structure the initial estimating. The development of a WBS divides the overall project into an interconnected set of manageable components. The WBS is typically a product-oriented structure that provides a scheme for identifying and organizing the logical units of work to be managed, which are called “work packages.” The WBS provides a reference and organizational mechanism for assigning effort, schedule, and responsibility and is used as the underlying framework to plan, organize, and control the work done on the project. [PA163.IG101.SP101.N101]
- Typical Work Products
  1. Task descriptions [PA163.IG101.SP101.W101]
  2. Work package descriptions [PA163.IG101.SP101.W102]
  3. WBS [PA163.IG101.SP101.W103]

Beispiel: Project Planning (4/5) [SEI10]

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Subpractices

1. Develop a WBS based on the product architecture. [PA163.IG101.SP101.SubP101]

The WBS provides a scheme for organizing the project's work around the products that the work supports. The WBS should permit the identification of the following items: [PA163.IG101.SP101.SubP101.N101]

- Identified risks and their mitigation tasks
- Tasks for deliverables and supporting activities
- Tasks for skill and knowledge acquisition
- Tasks for development of needed support plans, such as configuration management, quality assurance, and verification plans
- Tasks for integration and management of non-developmental items

Beispiel: Project Planning (5/5)

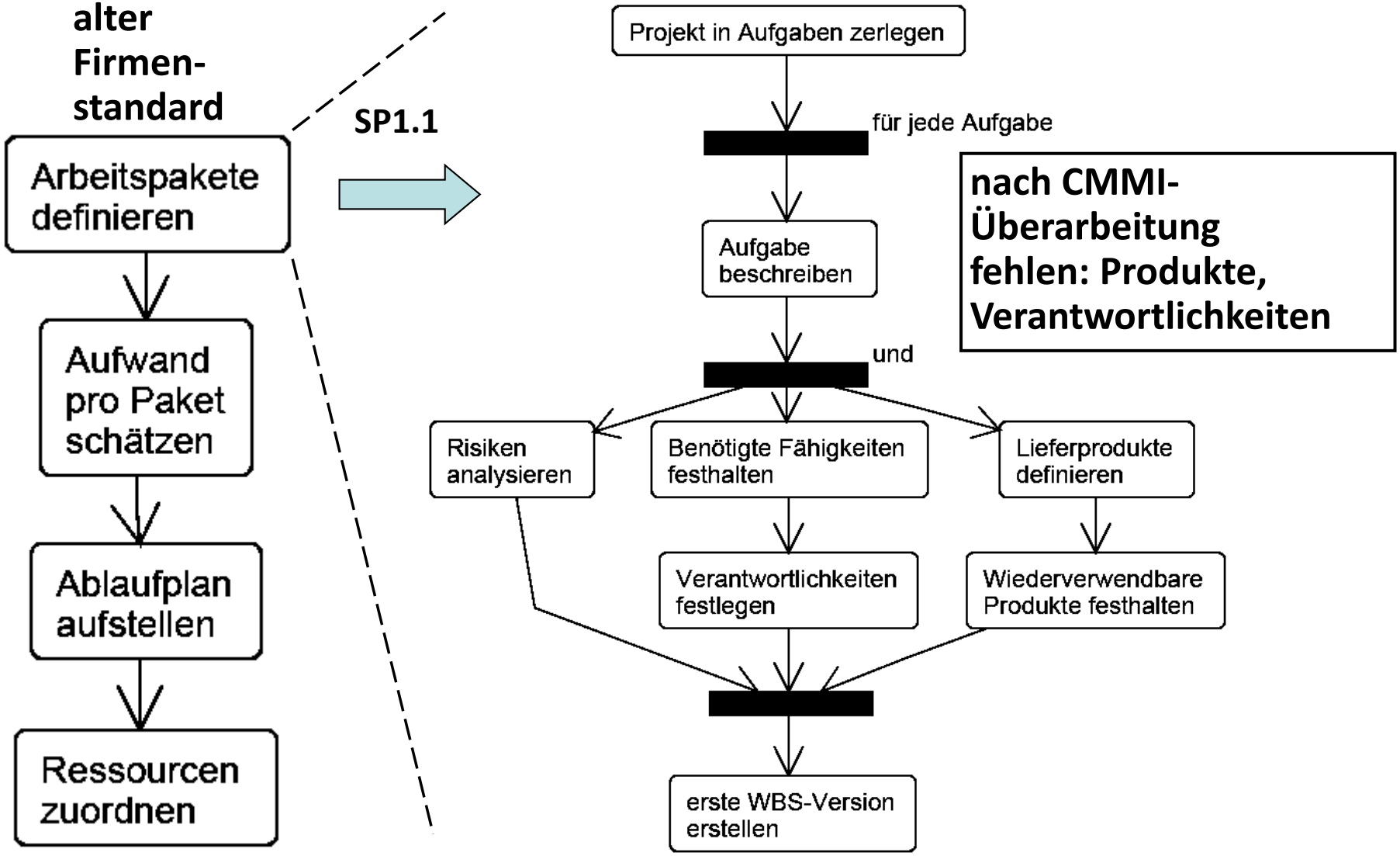
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

2. Identify the work packages in sufficient detail to specify estimates of project tasks, responsibilities, and schedule. [PA163.IG101.SP101.SubP102]

   The top-level WBS is intended to help in gauging the project work effort in terms of tasks and organizational roles and responsibilities. The amount of detail in the WBS at this more detailed level helps in developing realistic schedules, thereby minimizing the need for management reserve. [PA163.IG101.SP101.SubP102.N101]
3. Identify work products (or components of work products) that will be externally acquired. [PA163.IG101.SP101.SubP103]

   Refer to the Supplier Agreement Management process area for more information about acquiring work products from sources external to the project. [PA163.IG101.SP101.SubP103.R101]
4. Identify work products that will be reused. [PA163.IG101.SP101.SubP104]

Skizze einer CMMI-Anpassung



2. Beispiel: Umgang mit Anforderungen (1/10)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Process Area	***Category***	***Maturity Level***
Requirements Development (RD)	Engineering	3
Requirements Management (REQM)	Project Management	2

- nach CMMI zwei direkt betroffene Prozesse
- Achtung: querschnittliche Prozesse nicht beachtet („refer to“ in Spezifikation)
- diese Folien: nach [SEI10]

Beispiel: Umgang mit Anforderungen (2/10) – RD kurz

- Purpose: The purpose of Requirements Development (RD) is to elicit, analyze, and establish customer, product, and product component requirements.

Genereller Ansatz des CMMI (s. auch erstes Beispiel)

- Einleitung mit Nutzen und generellen Zielen
- Ableitung von “Specific Goals” (SG) als Liste
- Jedem SG werden ein oder mehrere “Specific Practice” (SP) zugeordnet
- SP geben generelle Prozessvorgaben, lassen aber Freiheiten für individuelle Umsetzung
- zu SP können “Example Work Products” und “Subpractices” gehören; deren Umsetzung (meist) prüfbar ist

Beispiel: Umgang mit Anforderungen (3/10) – RD SG SP

SG 1 Develop Customer Requirements

- SP 1.1 Elicit Needs
- SP 1.2 Transform Stakeholder Needs into Customer Requirements

SG 2 Develop Product Requirements

- SP 2.1 Establish Product and Product Component Requirements
- SP 2.2 Allocate Product Component Requirements
- SP 2.3 Identify Interface Requirements

SG 3 Analyze and Validate Requirements

- SP 3.1 Establish Operational Concepts and Scenarios
- SP 3.2 Establish a Definition of Required Functionality and Quality Attributes
- SP 3.3 Analyze Requirements
- SP 3.4 Analyze Requirements to Achieve Balance
- SP 3.5 Validate Requirements

Beispiel: Umgang mit Anforderungen (4/10) REQM

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Purpose: The purpose of Requirements Management (REQM) is to manage requirements of the project's products and product components and to ensure alignment between those requirements and the project's plans and work products.
- SG 1 Manage Requirements [nur ein SG]
  - SP 1.1 Understand Requirements
  - SP 1.2 Obtain Commitment to Requirements
  - SP 1.3 Manage Requirements Changes
  - SP 1.4 Maintain Bidirectional Traceability of Requirements
  - SP 1.5 Ensure Alignment Between Project Work and Requirements

Beispiel: Umgang mit Anforderungen (5/10) agil

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

REQM- Anmerkung zu „agil“

- In Agile environments, requirements are communicated and tracked through mechanisms such as product backlogs, story cards, and screen mock-ups. Commitments to requirements are either made collectively by the team or an empowered team leader. Work assignments are regularly (e.g., daily, weekly) adjusted based on progress made and as an improved understanding of the requirements and solution emerge. Traceability and consistency across requirements and work products is addressed through the mechanisms already mentioned as well as during start-of-iteration or end-of-iteration activities such as retrospectives and demo days. (See —Interpreting CMMI When Using Agile Approaches in Part I.)

Beispiel: Umgang mit Anforderungen (6/10) – SP 1.3

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

SP 1.3 Manage Requirements Changes

- Manage changes to requirements as they evolve during the project.
- Requirements change for a variety of reasons. As needs change and as work proceeds, changes may have to be made to existing requirements. It is essential to manage these additions and changes efficiently and effectively. To effectively analyze the impact of changes, it is necessary that the source of each requirement is known and the rationale for the change is documented. The project may want to track appropriate measures of requirements volatility to judge whether new or revised approach to change control is necessary.

Beispiel: Umgang mit Anforderungen (7/10) – SP 1.3

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Example Work Products

1. Requirements change requests
2. Requirements change impact reports
3. Requirements status
4. Requirements database

Beispiel: Umgang mit Anforderungen (8/10) – SP 1.3

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Subpractices

1. Document all requirements and requirements changes that are given to or generated by the project.
2. Maintain a requirements change history, including the rationale for changes. Maintaining the change history helps to track requirements volatility.
3. Evaluate the impact of requirement changes from the standpoint of relevant stakeholders. Requirements changes that affect the product architecture can affect many stakeholders.
4. Make requirements and change data available to the project.

Beispiel: Umgang mit Anforderungen (9/10) – SP 1.4

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

SP 1.4 Maintain Bidirectional Traceability of Requirements

- Maintain bidirectional traceability among requirements and work products.
- The intent of this specific practice is to maintain the bidirectional traceability of requirements. (See the definition of “bidirectional traceability” in the glossary.) When requirements are managed well, traceability can be established from a source requirement to its lower level requirements and from those lower level requirements back to their source requirements. Such bidirectional traceability helps to determine whether all source requirements have been completely addressed and whether all lower level requirements can be traced to a valid source.

Beispiel: Umgang mit Anforderungen (10/10) – SP 1.4

Example Work Products

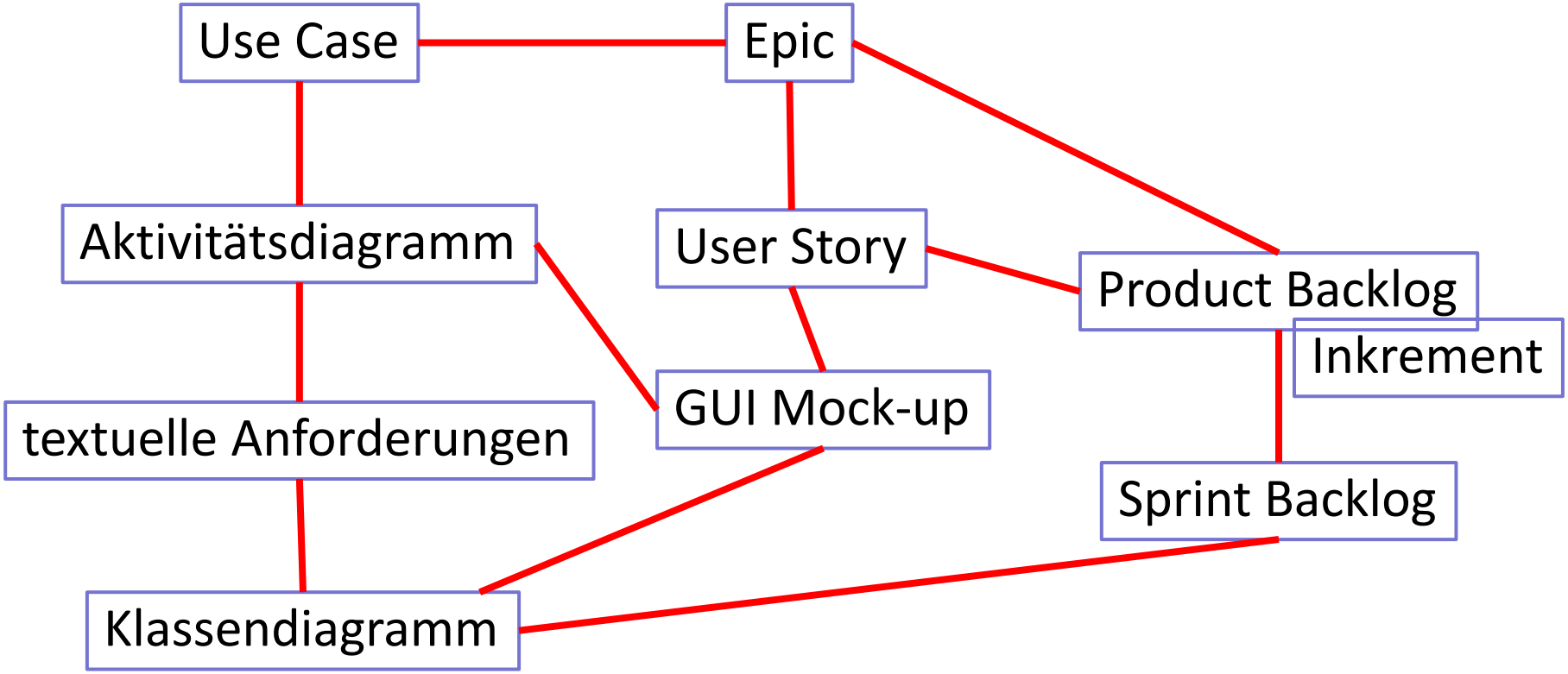
1. Requirements traceability matrix
2. Requirements tracking system

Subpractices

1. Maintain requirements traceability to ensure that the source of lower level (i.e., derived) requirements is documented.
2. Maintain requirements traceability from a requirement to its derived requirements and allocation to work products. Work products for which traceability may be maintained include the architecture, product components, development iterations (or increments), functions, interfaces, objects, people, processes, and other work products.
3. Generate a requirements traceability matrix.

Ansatz zur Umsetzung von REQM

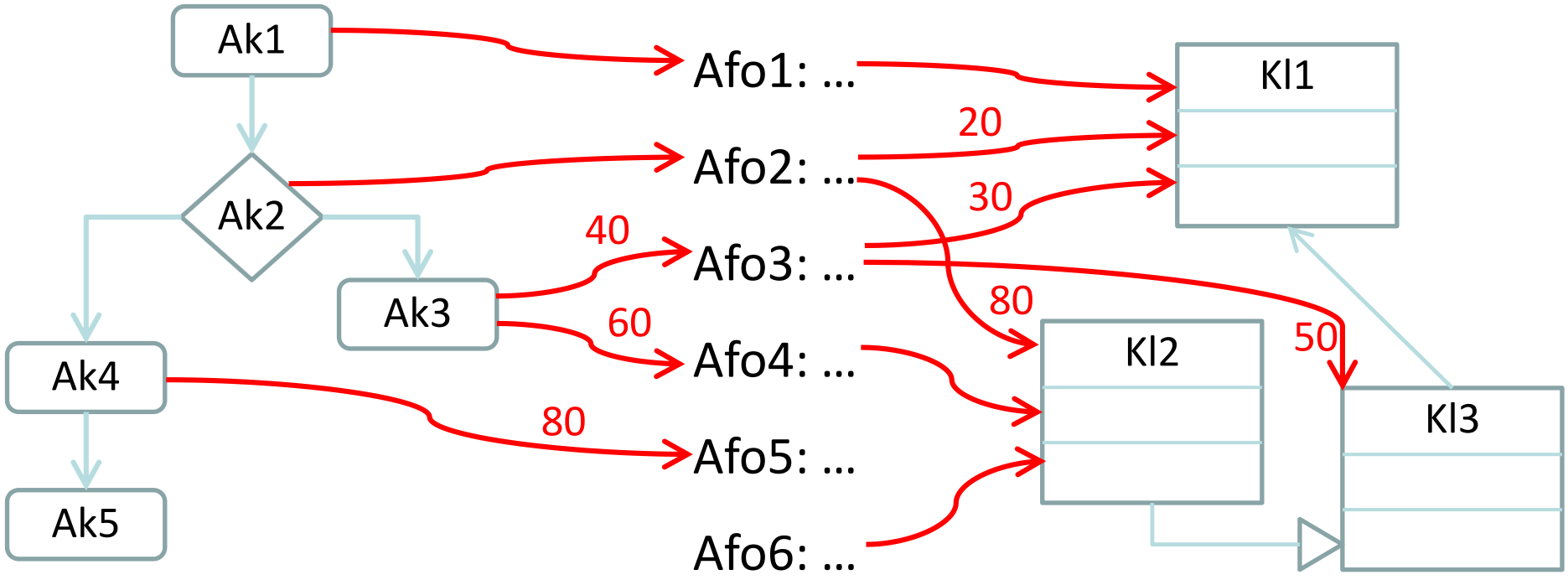
- Tracing (Verfolgbarkeit) zwischen folgenden Produkten denkbar



- generell Tracing zu Erstellern, Änderern, Versionen sinnvoll
- oft Tracing innerhalb eines Produkts sinnvoll (z. B. Abhängigkeiten)

Beispiel: kleiner Ausschnitt

- Pfeile zeigen Umsetzung in Prozentanteilen (ohne = 100)
- Information ist bidirektional, Pfeilspitze zeigt Entwicklungsrichtung



Tracing – Einstieg Datenmodellierung

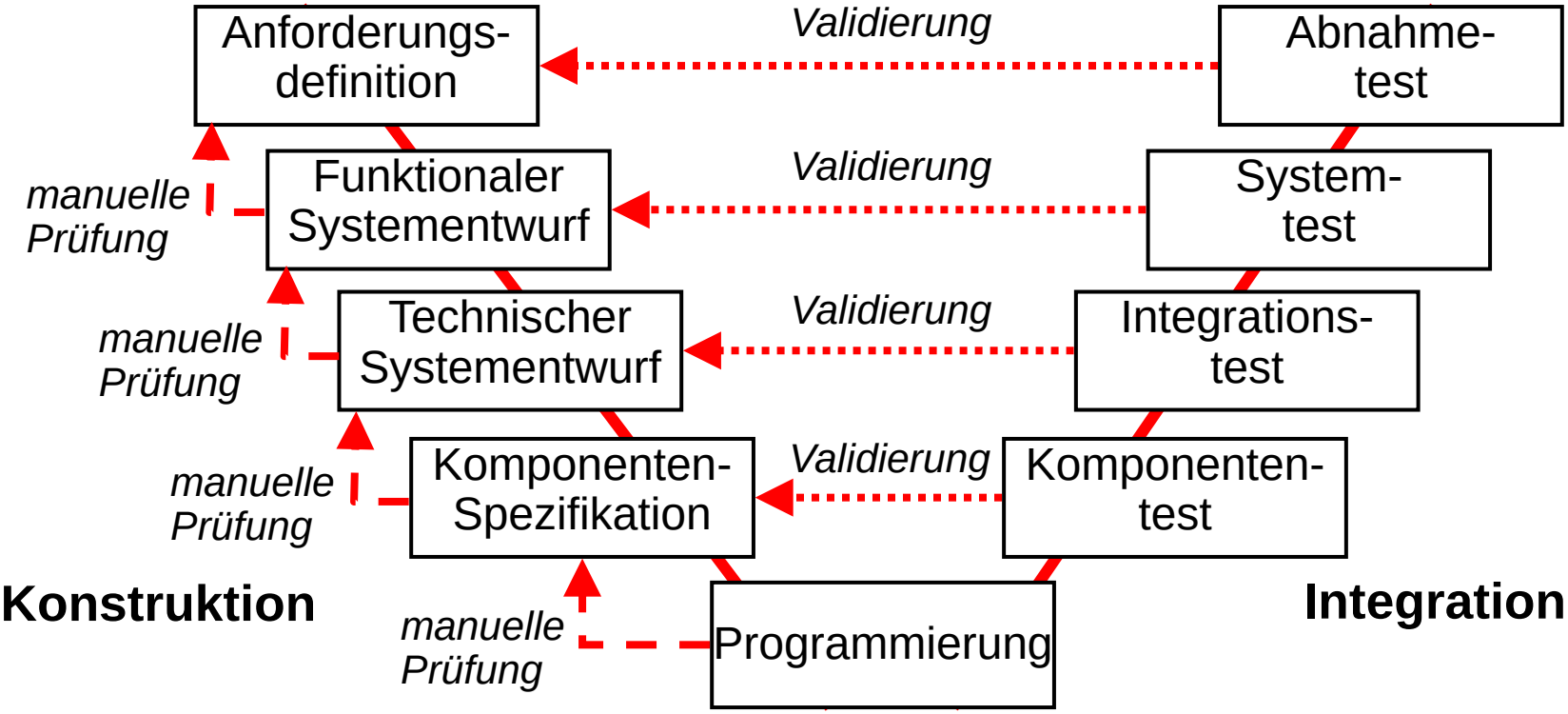
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- elementare Objekte: Artefakte
- Artefakt, muss haben: (42, „Ak1“)
  - eindeutige Id
  - Inhalt
- Artefakt, kann:
  - Zuordnung zum Entwicklungsschritt, Zuordnung zur Architekturschicht, Zuordnung zu Services, …

    -> als Gruppe von semantischen Informationen modellierbar

    ```
    ak1 := artefakt.New("Ak1", "Aktion")
    ```
  - detaillierter Inhalt
  - Ersteller
  - Historie
  - …

Typisches Tracing im V-Modell



- jede rote Verbindung sollte verfolgbar sein
- die Pflege jeder Verbindung (am Ende von Inkrementen oder Sprints) kostet Zeit (und damit Geld)
- jede gepflegte Verbindung erhöht die Qualität
- je höher die Kritikalität, desto genauer das Tracing
- je weniger änderbar (HW), desto genauer das Tracing

Werkzeuge des Requirements Engineering

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Hinweis: oft mit Entwicklungswerkzeugen direkt verknüpft
- Hinweis: oft über Schnittstellen erweiter- und anpassbar
- kommerziell (viele weitere Beispiele):
  - Rational DOORS NG (IBM)
  - Polarion (Siemens)
  - PTC Integrity
  - agosense.fidelia
  - Oracanos RMT
  - Austauschformat: Requirements Interchange Format (ReqIF)
- kleinere Projekte:
  - Excel-Sheets (au weia)
  - Wikis mit Link-Verknüpfungen
  - Annotationen oder spezielle Kommentare, im Programmcode nach festen Syntax-Regeln, über Werkzeuge ausgewertet

Gemeinsamkeiten von (fast allen) Prozessmodellen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

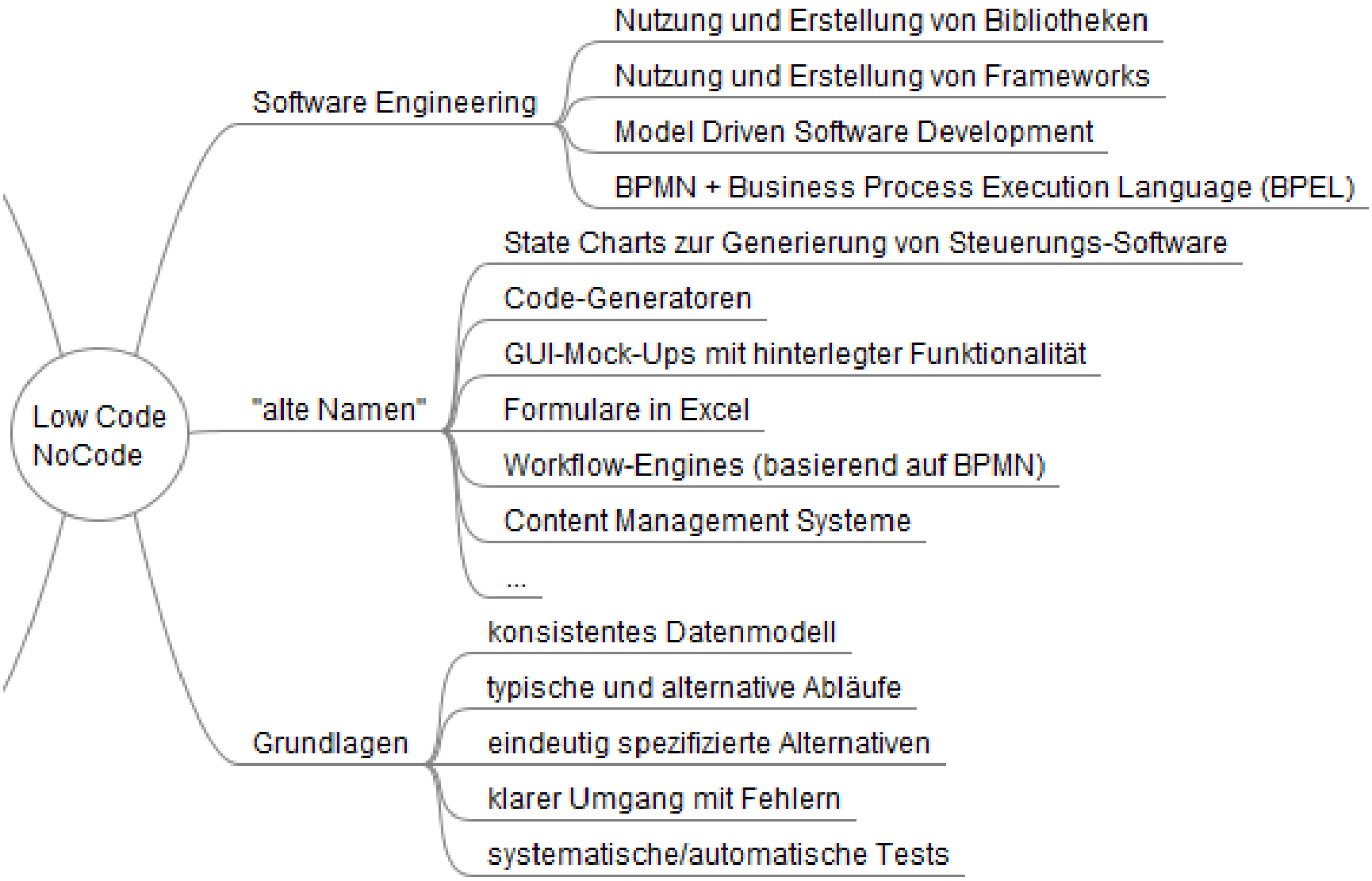
- strukturieren Themengebiet in fachliche Teilbereiche
- geben keine konkreten Prozesse mit Prozessschritten vor
- definieren notwendige Prozesse anhand von Zielen und Rahmenbedingungen
- liefern Fragen und Checklisten zur Prüfung der Zielerreichung
- Einführung und Nutzung sollte von Experten zumindest begleitet werden
- meist (teure) Standardzertifizierungen möglich

technische Umsetzung von Prozessen

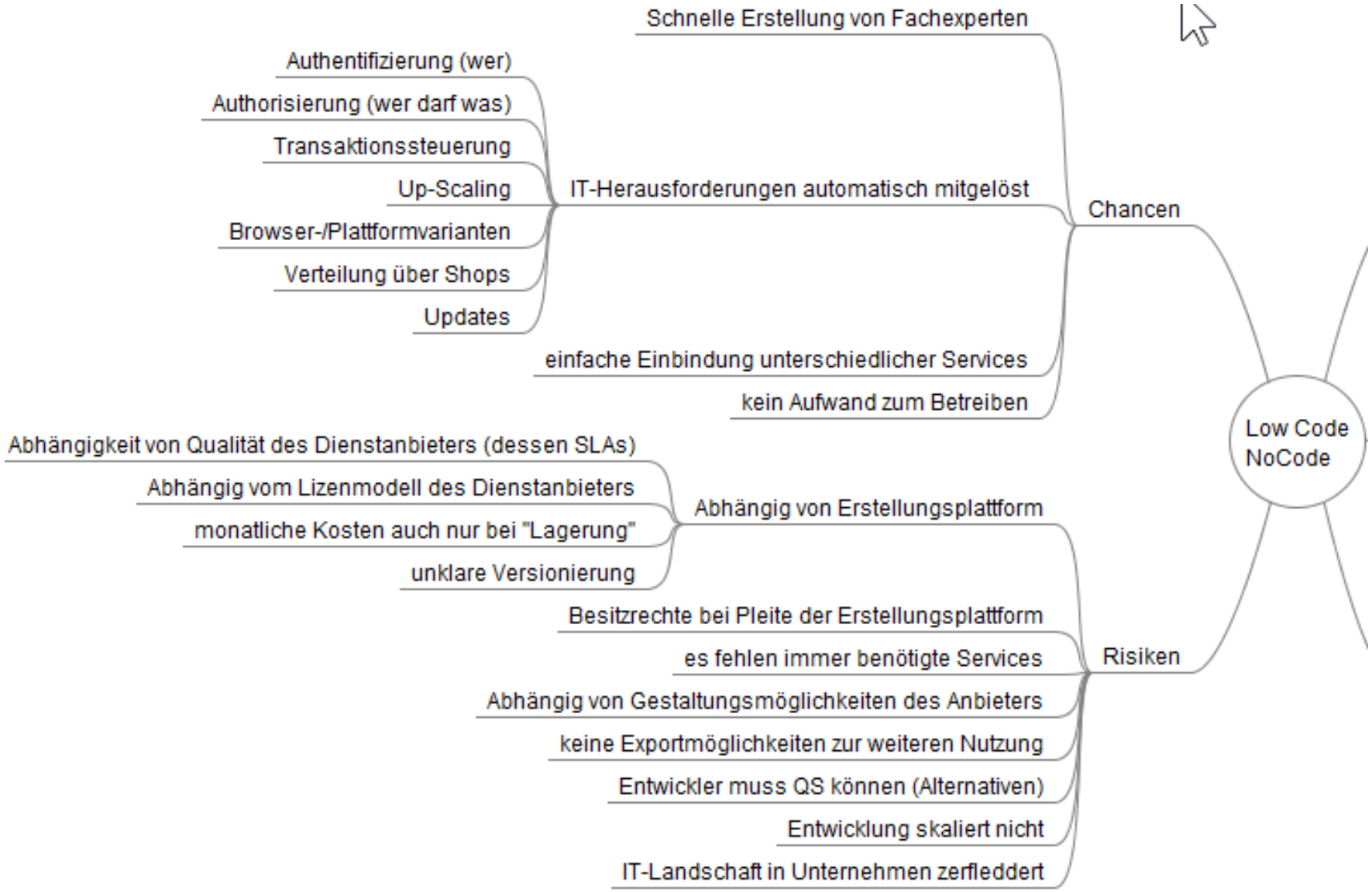
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Prozesse werden in Software abgebildet (egal ob Verwaltung, ERP oder Produktion)
- Variante: Prozesse modellieren, die dann automatisch in lauffähige Software umgewandelt wird
- benötigt: detaillierte Spezifikation, eindeutige Datenquellen
- eventuellst Kandidat von Low-Code oder NoCode-Ansätzen
- Ziel: Fachexperten modellieren Software und erhalten lauffähigen Code
- werden u. a. als Web-Plattformen angeboten

Analyse Low Code und NoCode (1/2)



Analyse Low Code und NoCode (2/2)

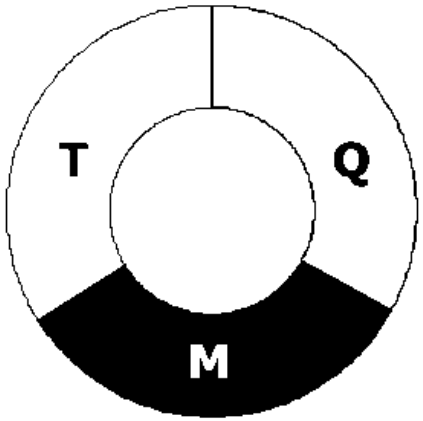


Überblick TQM (QM-Framework, nicht IT-spezifisch)

Total Quality Management

- Bezieht das gesamte Unternehmen in die QS ein
- Qualität wird vom Kunden bestimmt
- Das Management muss TQM vorantreiben
- Wurde zuerst in Japan ab 1950er entwickelt

- Bereichs- und funktionsübergreifend
- Kundenorientierung
- Einbeziehung aller Mitarbeiter



- Prozessqualität
- Produktqualität
- Kontinuierliche Qualitätsverbesserung

- Vorbildfunktion des Managements
- Qualität gleichberechtigt zu Kosten und Terminen

Sechs typische Prinzipien im TQM

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Prinzip:

- des Vorrangs der Qualität
- der Zuständigkeit aller Mitarbeiter
- der ständigen Verbesserung
- der Kundenorientierung
- des internen Kunden-Lieferanten-Verhältnisses
- der Prozessorientierung

Kommt aus Japan, in den letzten 60 Jahren entwickelt. Ideen der amerikanischen Autoren CROSBY, DEMING, FEIGENBAUM und JURAN wurden mit japanischen Auffassungen von ISHIKAWA und IMAI und insbesondere der japanischen Kultur, Gesellschaft und Geschichte verschmolzen.

Vorrang der Qualität

- Alle Prozesse müssen Qualitätsprozesse sein
- An die Prozesse gestellte Anforderungen müssen 100%-tig erfüllt werden
- Jeder Mitarbeiter soll seine Arbeit sofort beim ersten Mal und jedes Mal erneut richtig tun
- Qualitätsverbesserung durch Verbesserung der Entwicklungsprozesse
- Vermeidung von Nacharbeit

Zuständigkeit aller Mitarbeiter

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

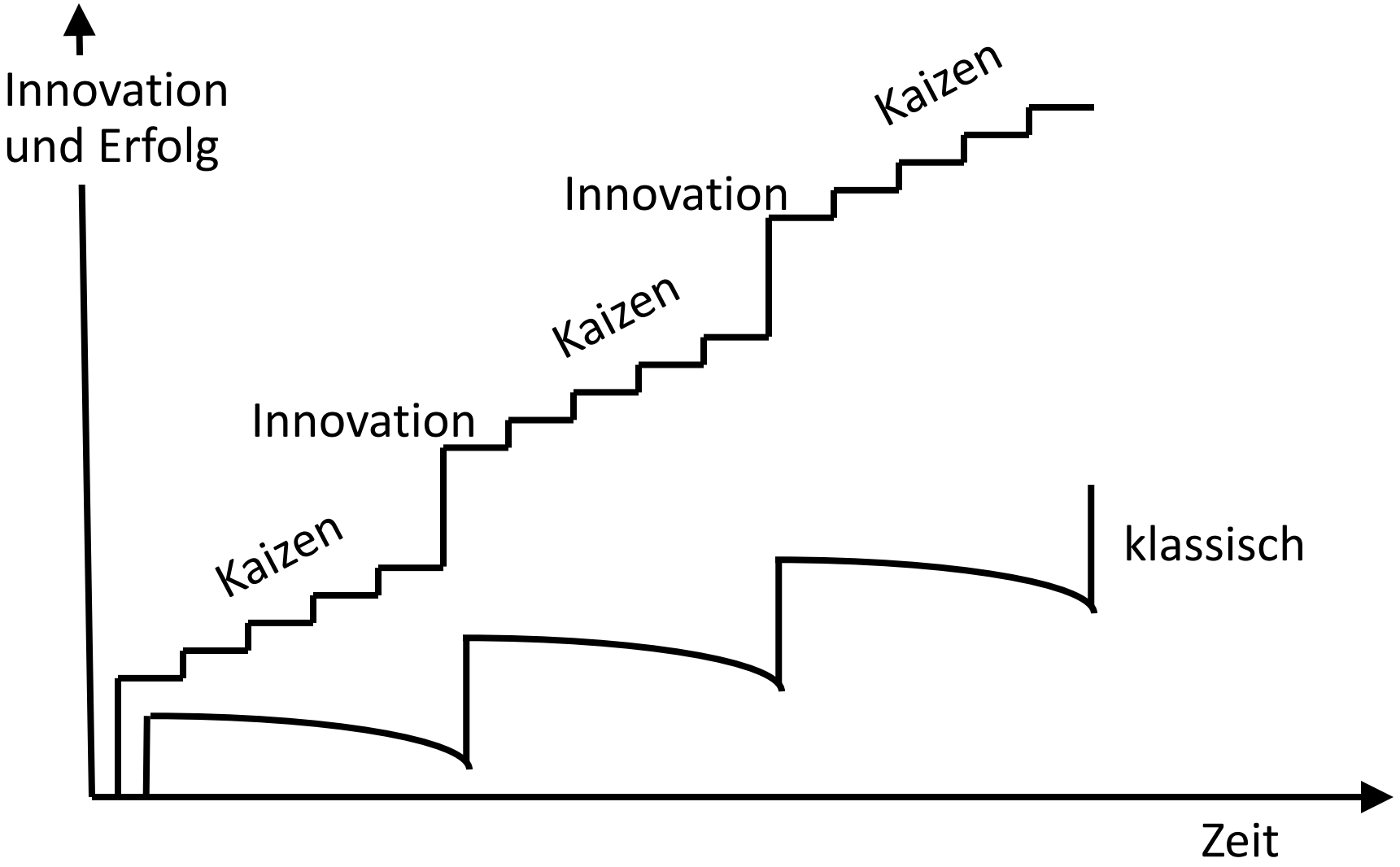
- Alle an der Erstellung und Vermarktung eines Produkts beteiligten Mitarbeiter müssen für dessen Qualität sorgen
- Jede Führungskraft muss es ihren Mitarbeitern ermöglichen, keine oder zumindest weniger Fehler zu machen
- Alle Prozesse eines Unternehmens müssen unter Qualitätsgesichtspunkten „gemanagt" werden
- Alle Mitarbeiter eines Teams sind für Fehler einzelner verantwortlich
- unabhängige QS-Abteilung ist überflüssig (!?)

Ständige Verbesserung (Kaizen)

Kaizen japanisch „ständige Verbesserung"

- lieber kleine aber kontinuierliche Schritte, statt Innovationsschübe
- setzt auf langfristige Perspektiven und Verhaltensänderungen
- soziale Strukturen eines Unternehmens sollen genutzt und nicht missachtet werden

Verbesserung in kleinen Schritten



Kundenorientierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Primäres Ziel: Erfüllung der Kundenanforderungen
- Enge Zusammenarbeit zwischen Entwicklung, Marketing und Kundendienst
- Kunde soll bei der Formulierungen seiner Bedürfnisse unterstützt werden (z. B. Individualsoftware)
- die Bedürfnisprofile der Hauptzielgruppen werden durch intensive Marktanalysen ermittelt (z.B. Zielrichtung/Ausprägung der Standardsoftware)

Kunden-Lieferanten-Verhältnis

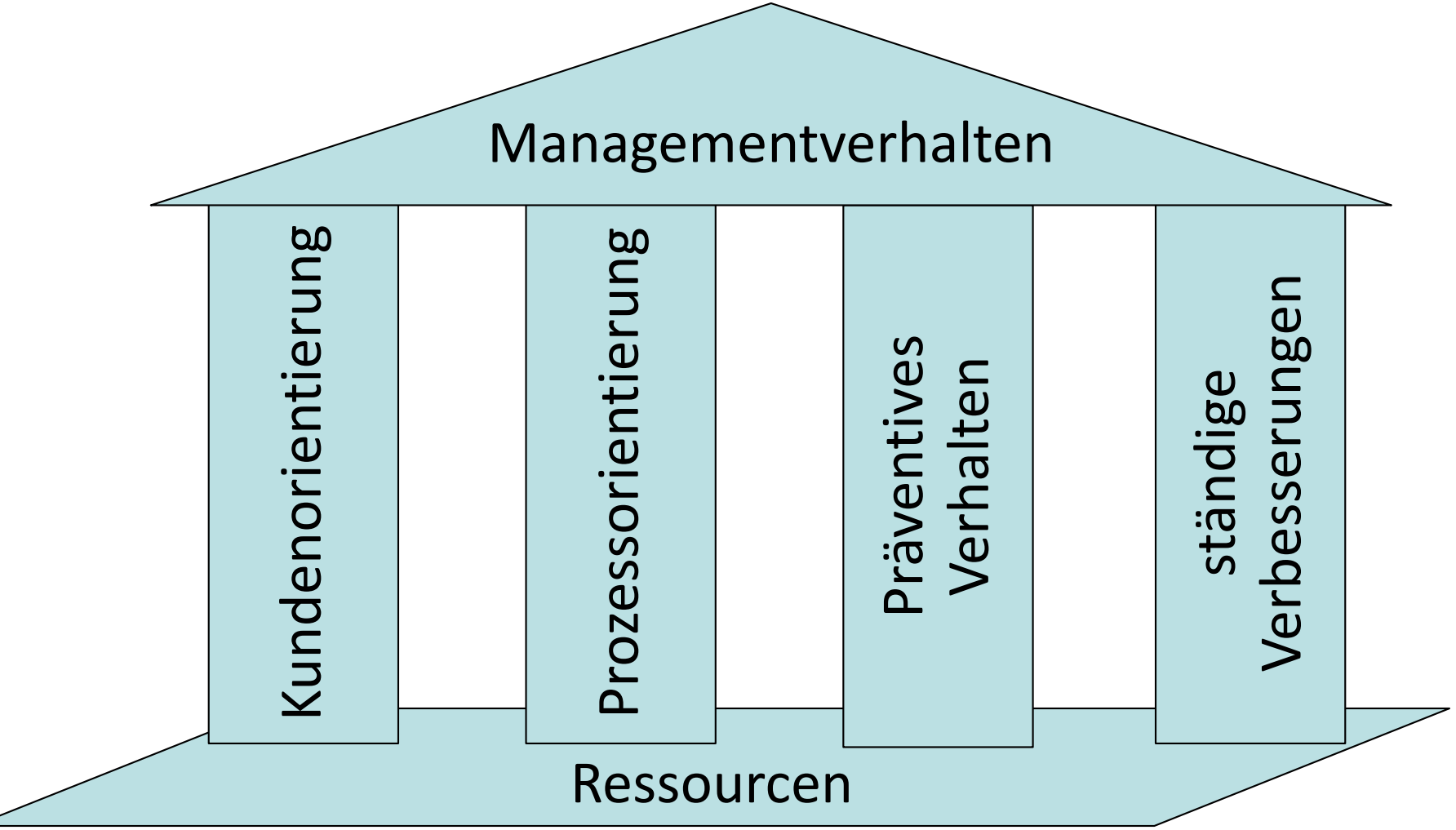
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Mitarbeiter-Integration in der internen Prozesskette:
  - Mitarbeiter gilt als Kunde vom Vorgänger-Prozess
  - Mitarbeiter gilt als Lieferant für den Folge-Prozess
- Die Übergabe der Leistung sollte formell abgenommen werden (ähnlich wie bei externen Leistungen)
- Alle Teams werden auf den Erfolg der jeweils nächsten Teams verpflichtet
- Jedes Team und jeder Mitarbeiter ist für die Qualität seines Teilproduktes selbst verantwortlich

Prozessorientierung

- Annahme: Fehler = Defizite des Entwicklungsprozesses
- Fehlerursachen werden ermittelt und behoben
- z. B. Software-Erstellung ist ein reproduzierbarer und verbesserungsfähiger Prozess (siehe CMMI)
- Es gilt: Fehlervermeidung geht vor Fehlerbehebung!

Säulen des TQM



Maßnahmen und Konzepte

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Wichtige Maßnahmen zur Realisierung von TQM
  - Klar formulierte Qualitätspolitik und nachvollziehbare Q-Ziele
  - Festlegung und Bekanntgabe der Kompetenzen
  - Konsequente Schulung aller Mitarbeiter in Sachen Qualität und Qualitätsmanagement

  Ziel: Einführung eines QM-Systems

- Typische Techniken des TQM
  - Qualitätszirkel
  - *Quality Function Deployment* (QFD)

Qualitätszirkel (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Dienen der Erreichung der Prinzipien
  - Vorrang der Qualität - alle Prozesse sind Qualitätsprozesse und jeder soll seine Arbeit 100%ig machen
  - Zuständigkeit aller Mitarbeiter
  - Ständige Verbesserung

- Vorgehen
  - Üblicherweise wöchentliche Besprechung mit dem Ziel, die im Arbeitsbereich auftretenden Q-Probleme zu lösen
  - Verbesserungen werden meist durch das Team selbst durchgeführt
  - Erfolgskontrolle (auch durch das Team selbst)

Qualitätszirkel (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Typische Arbeitsweise von Qualitätszirkeln:
  1. Problemidentifikation und Auswahl
     - Auswahl zu untersuchender Probleme
     - Einsatz von Kreativitätstechniken (z. B. Brainstorming) zur Problemidentifikation
     - Priorisierung der Probleme
  2. Problembearbeitung
     - Genehmigung durch Entscheidungsstelle
     - Abstimmung mit anderen Q-Zirkeln (wenn nötig)
     - Trennung: Haupt- / Nebenursachen
     - Ziele festlegen
     - Lösungen suchen
     - Alternativen bewerten und Lösungen auswählen

Qualitätszirkel (3/3)

3. Ergebnispräsentation
   - Lösung dem Entscheiderkreis präsentieren und Umsetzung vorbereiten

4. Einführung und Erfolgskontrolle
   - Lösung einführen
   - Dokumentation von Problem, Lösungsweg und Ergebnis
   - Erfolgskontrolle (möglichst quantitativ)
   - Generalisierung (Übertragen auf andere Organisationsteile)

Manifest für Agile Softwareentwicklung (2001)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen. Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt:

- Individuen und Interaktionen mehr als Prozesse und Werkzeuge
- Funktionierende Software mehr als umfassende Dokumentation
- Zusammenarbeit mit dem Kunden mehr als Vertragsverhandlung
- Reagieren auf Veränderung mehr als das Befolgen eines Plans

Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden, schätzen wir die Werte auf der linken Seite höher ein.

https://agilemanifesto.org/iso/de/manifesto.html
https://jaxenter.de/qualitatsmanagement-und-agilitat-was-beide-voneinander-lernen-konnen-4542

Fazit: Bedeutung des QM

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Die Wichtigkeit eines funktionierenden, sich selbst optimierenden QM-Systems steigt bei folgenden Aspekten:

- Projektergebnisse im Umfeld von „Gefahr für Leib und Leben“ und „hohe Finanzflüsse“
- Anzahl der Mitarbeiter*innen, deren Zusammenarbeit geregelt werden muss
- Komplexität einzelner Projekte
- Grad der Wiederverwendung und der Erweiterbarkeit
- hoher Status des Unternehmens

Da SW-QS Teil des QM ist, steigt deren Bedeutung parallel.

Video

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

3. Qualitätssicherung

- 3.1 Hintergründe von Java-Testwerkzeugen
- 3.2 JUnit 5 - kompakt
- 3.3 Äquivalenzklassen
- 3.4 Überdeckungen
- 3.5 Vorgehensmodelle und Testen
- 3.6 Konstruktive Qualitätssicherung
- 3.7 Metriken
- 3.8 Organisation des QS-Prozesses in IT-Projekten
- 3.9 Testautomatisierung
- 3.10 Testwerkzeuge basierend auf Byte-Code

Literatur

- Stephan Kleuker, Qualitätssicherung durch Softwaretests, Springer Vieweg, 2. Auflage, Wiesbaden, 2019
- Peter Liggesmeyer, Software-Qualität: Testen, Analysieren und Verifizieren von Software, Spektrum Akademischer Verlag, 2. Auflage, 2009
- Stephan Kleuker, Grundkurs Software-Engineering mit UML, Kapitel 11, Springer Vieweg, 4. Auflage, Wiesbaden, 2018

Annahme: Master-Studierende eines Informatik-Studiengangs haben Tests selbst geschrieben und in ihrer SW-Entwicklung genutzt

Motivation / Ansatz

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

auf Reflexion basierende Testwerkzeuge

Wdh.: JUnit

Testüberdeckungen: notwendig aber nicht hinreichend

Wdh.: Grundlagen der Testerstellung

Wdh.: QS als Teil des SW-Engineerings

fortgeschrittene Möglichkeiten zur Erstellung von QS-Werkzeugen

Ziel: umfassendes Verständnis der Möglichkeiten funktionaler Qualitätssicherung

3.1 Hintergründe von Java-Testwerkzeugen

- Reflection
- Annotationen
- Lambda-Ausdrücke
- Annotationen mit Reflection nutzen

Ansatz: Unit-Testwerkzeug selbst entwickeln, unter Nutzung eigener Annotationen, die mit Hilfe von Reflection ausgewertet werden

Nutzung von Class

- Klassenobjekt erlaubt
  - Abfrage von Variablen, Konstruktoren, Methoden (mit Sichtbarkeiten, Typen, Parametern, Exceptions,…)
  - Abfrage und Änderung von Objektwerten
  - Nutzung von Konstruktoren und Methoden
  - Abfrage von Oberklassen

- Insgesamt können damit zur Laufzeit beliebige Klassen analysiert und genutzt werden
- Zentrale Reflection-Klassen: Field, Constructor, Method, Modifier

Reflection

- auch Introspektion, Reflexion
- Reflexion erlaubt in Java Meta-Programmierung; Klassen selbst als Objekte nutzen
- Paket **`java.lang.reflect`**
- Zu jeder Klasse gibt es ein Klassenobjekt der Klasse **`Class`**

```
Class cl1 = String.class;
try { // oder
  Class cl2 = Class.forName("java.lang.String");
} catch (ClassNotFoundException e) {
  System.out.println("gibs nich")
}
```

Exemplarische Methoden der Klasse Class

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Annotation[]	getDeclaredAnnotations() Returns all annotations that are directly present on this element.
Constructor[]	getDeclaredConstructors() Returns an array of Constructor objects reflecting all the constructors declared by the class represented by this Class object.
Field[]	getDeclaredFields() Returns an array of Field objects reflecting all fields declared by the class or interface represented by this Class object.
Method[]	getDeclaredMethods() Returns an array of Method objects reflecting all the methods declared by the class or interface represented by this Class object.
Class[]	getInterfaces() Determines the interfaces implemented by the class or interface represented by this object.
int	getModifiers() Returns the Java language modifiers for this class or interface, encoded in an integer.
boolean	isArray() Determines if this Class object represents an array class.
T	newInstance() Creates a new instance of the class represented by this Class object.

http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html

Eigene Annotationen (nur beispielhaft)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
package annotation;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MeinTest {

}
```

welche Elemente können annotiert werden, hier Methode (Array)

wie lange soll Annotation leben (hier auch zur Laufzeit nutzbar)

Deklaration mit @interface

hier können Attribute mit ihren Typen definiert werden

Beispiel: eigene Testausführung mit Annotation (1/3)

- Ziel alle mit MeinTest annotierten Methoden sollen ausgeführt werden, auftretende Exceptions werden dokumentiert
- Randbedingungen:
  - Klasse, die Tests enthält, muss parameterlosen Konstruktor haben (sonst schwer Objekt automatisch zu erstellen, wenn Parameter geraten werden müssen)
  - Tests selber haben keine Parameter (so einfacher, sonst müssen Werte für Parameter angegeben werden)
  - Tests haben keine Rückgaben (könnten ignoriert werden)

Beispiel: eigene Testausführung mit Annotation (2/3)

```
package beispiel;

import annotation.MeinTest;

public class Beispiel { // zu analysierende Beispielklasse

  @MeinTest
  public void test1(){
  }

  @MeinTest
  protected void test2(){
    if (6*9 != 42){
      throw new ArithmeticException("stimmt die Frage nicht");
    } // statt Assertions werden Exceptions zur
      // Fehlererkennung genutzt
  }
```

Beispiel: eigene Testausführung mit Annotation (3/3)

```
    @MeinTest
    private void test3(){
    }

    @MeinTest
    void test4() throws Exception{
        throw new Exception("immerhin ausgefuehrt");
    }

    @MeinTest
    public static void test5(){
    }
}
```

Konzept der Testausführung

zentrale Ausführungsklasse Tester (oder TestRunner) macht folgende Schritte

- bekommt zu untersuchende Klasse übergeben
- erstellt ein Objekt davon (parameterloser Konstruktor gefordert)
- durchsuche alle Methoden der Klasse
  - wenn mit @MeinTest annotiert, Rückgabetyp void und parameterlos, dann führe Methode aus
  - überwache ob Methode Exception wird
  - dokumentiere Exceptions und gebe sie als Fehler aus

durch Auslesen des Class-Path und der Module-Paths könnten alle Klassen betrachtet werden

Annotationsnutzung (Prototyp) (1/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Tester {
  public Map<String,String> testAusfuehren(String klassenname) {
      // Ergebnis: Testname, eventueller Fehlertext oder „ok“
    Class clazz = null;
    if (klassenname == null) {
      throw new IllegalArgumentException("Existenz der Klasse"
          + " null versucht zu pruefen");
    }
    // pruefe ob Klasse bekannt (nutzt Class- (Module-Path))
    try {
      clazz = Class.forName(klassenname);
    } catch (ClassNotFoundException ex) {
      throw new IllegalArgumentException("Klasse "
          + klassenname + " nicht gefunden");
    }
```

Annotationsnutzung (Prototyp) (2/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// Suche parameterlosen Konstruktor
Constructor konstruktor = null;
try {
  konstruktor = clazz.getDeclaredConstructor(new Class[]{});
} catch (NoSuchMethodException ex) {
  throw new IllegalArgumentException("Klasse muss "
      + "parameterlosen Konstruktor haben");
} catch (SecurityException ex) {
  throw new IllegalArgumentException("kein ausfuehrbarer "
      + "parameterloser Konstruktor gefunden");
}

// berechne mit Test annotierter Methoden
// Hinweis: mit konfigurierbaren Security-Manager (und
//   Modulkonzept ab Java 9) ist Reflection verhinderbar
List<Method> auszufuehren = new ArrayList<>();
```

Array mit Typen der Parameter, hier keine Parameter

Annotationsnutzung (Prototyp) (3/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
for (Method methode : clazz.getDeclaredMethods()) {
  methode.setAccessible(true); // Exception moeglich
  for (Annotation anno : methode.getAnnotations()) {
   if (anno.annotationType() == MeinTest.class) {
    if (!methode.getReturnType().toString().equals("void")) {
        System.out.println(" Methode " + methode.getName()
            + " ist kein Test, da Return-Typ nicht void");
      } else {
        if (methode.getParameterCount() != 0) {
          System.out.println(" Methode " + methode.getName()
             + " ist kein Test, da Parameterliste nicht leer");
        } else {
          auszufuehren.add(methode);
        }
      }
    }
   }
  }
}
```

Annotationsnutzung (Prototyp) (4/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
Object objekt = null;
if (auszufuehren.isEmpty()) {
  System.out.println("keinen Test gefunden");
  return null;
}
konstruktor.setAccessible(true);
try {
  objekt = konstruktor.newInstance(new Object[]{});
} catch (Exception ex) {
  throw new IllegalArgumentException("unerwarteter Fehler: "
     + ex);
}

// fuehre Methoden aus und fange Exceptions
// es wird parameterlose Methode genutzt, zum Ausfuehren wird
// dem Methodenobjekt das Objekt und die Parameterliste
// uebergeben
```

Annotationsnutzung (Prototyp) (5/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
    Map<String,String> ergebnis = new HashMap<>();
    for (Method m : auszufuehren) {
      try {
        m.setAccessible(true);
        m.invoke(objekt, new Object[]{});
        ergebnis.put(m.getName(), "ok");
      } catch (InvocationTargetException e) {
        ergebnis.put(m.getName(), e.getCause().getClass()
            .getSimpleName() + " : "
                              + e.getCause().getMessage());
      } catch (Exception e) {
        ergebnis.put(m.getName(), e.getClass().getSimpleName()
            + " : " + e.getCause().getMessage());
      }
    }
    return ergebnis;
  }
}
```

Annotationsnutzung (Prototyp) (6/6) - Nutzung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Main {
  public static void main(String[] args) {
    Tester tester = new Tester();
    tester.testAusfuehren("beispiel.Beispiel")
          .forEach((k,v) -> System.out.println(k + ": " + v));
    }
}
```

```
test4: Exception : immerhin ausgefuehrt
test5: ok
test2: ArithmeticException : stimmt die Frage nicht
test3: ok
test1: ok
//TODO: durch Thread-Nutzung Gefahr der Endlosscheifen
//      vermeiden
```

Lambda-Ausdrücke seit Java 8

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ansatz: Funktionen als Parameter übergeben
- Vereinfachung für Interfaces, die genau eine Methode enthalten (auch SAM-Types für Single Abstract Method)
- selber explizit definierbar mit Annotation @FunctionalInterface

```
(Parameterliste) -> {Ausdruck bzw. Programmanweisungen}
```

- einige Notationsvarianten
- hier interessant, da in JUnit 5 genutzt

- Spezifikation: JSR 335: Lambda Expressions for the JavaTM Programming Language, https://jcp.org/en/jsr/detail?id=335

Möglichkeiten der Interface-Nutzung (1/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
package interfaces;

@FunctionalInterface
public interface BspInterface {
    public int mach(int x, int y);
}
```

```
public class Plus implements BspInterface{

     public int mach(int x, int y){
         System.out.println("plus");
         return x+y;
     }
}
```

Möglichkeiten der Interface-Nutzung (2/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// Beispielklasse fuer Interface-Nutzung
package nutzer;

import interfaces.BspInterface;

public class BspNutzer {
    public int nutzen (BspInterface b1, BspInterface b2
                       , BspInterface b3){
        return b3.mach(6, 9) - b2.mach(6, 9) - b1.mach(6, 9);
    }
}
```

Möglichkeiten der Interface-Nutzung (3/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// Beispielklasse fuer Interface-Nutzung
public static void main(String[] args) {
  BspInterface klassisch = new Plus();
  int erg = klassisch.mach(6, 9);
  System.out.println("Klassisch: " + erg);

  BspInterface direktesObjekt = new BspInterface() {
      @Override
      public int mach(int x, int y) {
        System.out.println("minus");
        return x - y;
      }
  };
  erg = direktesObjekt.mach(6, 9);
  System.out.println("direktes Objekt: " + erg);
```

```
plus
Klassisch: 15
```

```
minus
direktes Objekt: -3
```

Möglichkeiten der Interface-Nutzung (4/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
System.out.println("anonymes Objekt: "
  + (new BspInterface() {
      @Override
      public int mach(int x, int y) {
        System.out.println("mal");
          return x * y;
        }
      }).mach(6, 9));
```

```
mal
anonymes Objekt: 54
```

```
BspInterface direktesObjektMitLambda = (a,b) -> {
  System.out.println("oder");
  return a | b;
};
erg = direktesObjektMitLambda.mach(6, 9);
System.out.println("direktes Objekt mit Lambda: " + erg);
```

```
oder
direktes Objekt mit Lambda: 15
```

Möglichkeiten der Interface-Nutzung (5/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
BspInterface direktesObjektMitLambda2 =
                    (a,b) -> a & b;
erg = direktesObjektMitLambda2.mach(6, 9);
System.out.println("direktes Objekt mit Lambda: " + erg);
```

```
direktes Objekt mit Lambda: 0
```

```
BspNutzer nutzer = new BspNutzer();
System.out.println(nutzer.nutzen(
          klassisch
        , (a,b) -> {
            System.out.println("minus");
            return a - b;
          }
        , (a,b) -> a * b));
```

```
minus
plus
42
```

Möglichkeiten der Interface-Nutzung (6/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
    BspNutzer nutzer2 = new BspNutzer();
    System.out.println(nutzer2.nutzen(
                (int a, int b) -> {  // Typen angebbar
                    System.out.println("spielerei");
                    return 2 * b + 4 * a;
                }
                , (a,b) -> Math.addExact(a,b)
                , Math::addExact));
  }
}
```

```
spielerei
-42
```

```
// letzter Fall zeigt Abkuerzung bei identischen
Parametertypen
```

Beispiel: Programmanalyse (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ansatz: Übergabe beliebiger und beliebig vieler Programmstücke, die ausgeführt und deren Exceptions gefangen werden
- Ansatz: Interface für parameterlose Methoden

```
@FunctionalInterface
public interface Programm {
    public void ausfuehren();
}
```

- Erinnerung: Parameterlisten mit dynamischer Länge
- `public int mach (double d, int... x){  //normal weiter`
- Aufruf mit 0 bis beliebig vielen int-Werten möglich
- x ist Variable vom Typ int[]

Beispiel: Programmanalyse (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Analyse {
   public static List<String> analysieren(Programm... progs) {
     List<String> ergebnisse = new ArrayList<>();
     for (Programm p : progs) {
       try {
         p.ausfuehren();
         ergebnisse.add("ok");
       } catch (Throwable e) {
         ergebnisse.add(e.getClass().getSimpleName()
             + ": " + e.getMessage());
       }
     }
     return ergebnisse;
   }
}
```

Beispiel: Programmanalyse (3/3)

```
public static void main(String[] args) {
  System.out.println(Analyse.analysieren(
       () -> System.out.println("Hallo")
      ,() -> {
        System.out.println("durch 0");
        int x = 7 / 0;
       }
      ,() -> {
        System.out.println("Array");
        int[] x = {1, 2, 3};
        System.out.println(x[3]);
      }
      ,() -> {
        throw new IllegalArgumentException(
            "Kein Mensch ist illegal");
      }
  ));
}
```

```
Hallo
durch 0
Array
[ok,
ArithmeticException: /
by zero,
ArrayIndexOutOfBoundsEx
ception: 3,
IllegalArgumentExceptio
n: Kein Mensch ist
illegal]
```

3.2 JUnit 5 - kompakt

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- Hinweis: Um im Praktikum experimentieren zu können, erst „wie schreibe ich Tests“ und danach „welche Tests sind sinnvoll“
- Warnung für dieses und die weiteren Kapitel:

  Der Programmcode ist meist ein Beispiel für schlechte Formatierung; hier erlaubt, um möglichst viele Details auf einer Folie mit Ihnen zu diskutieren

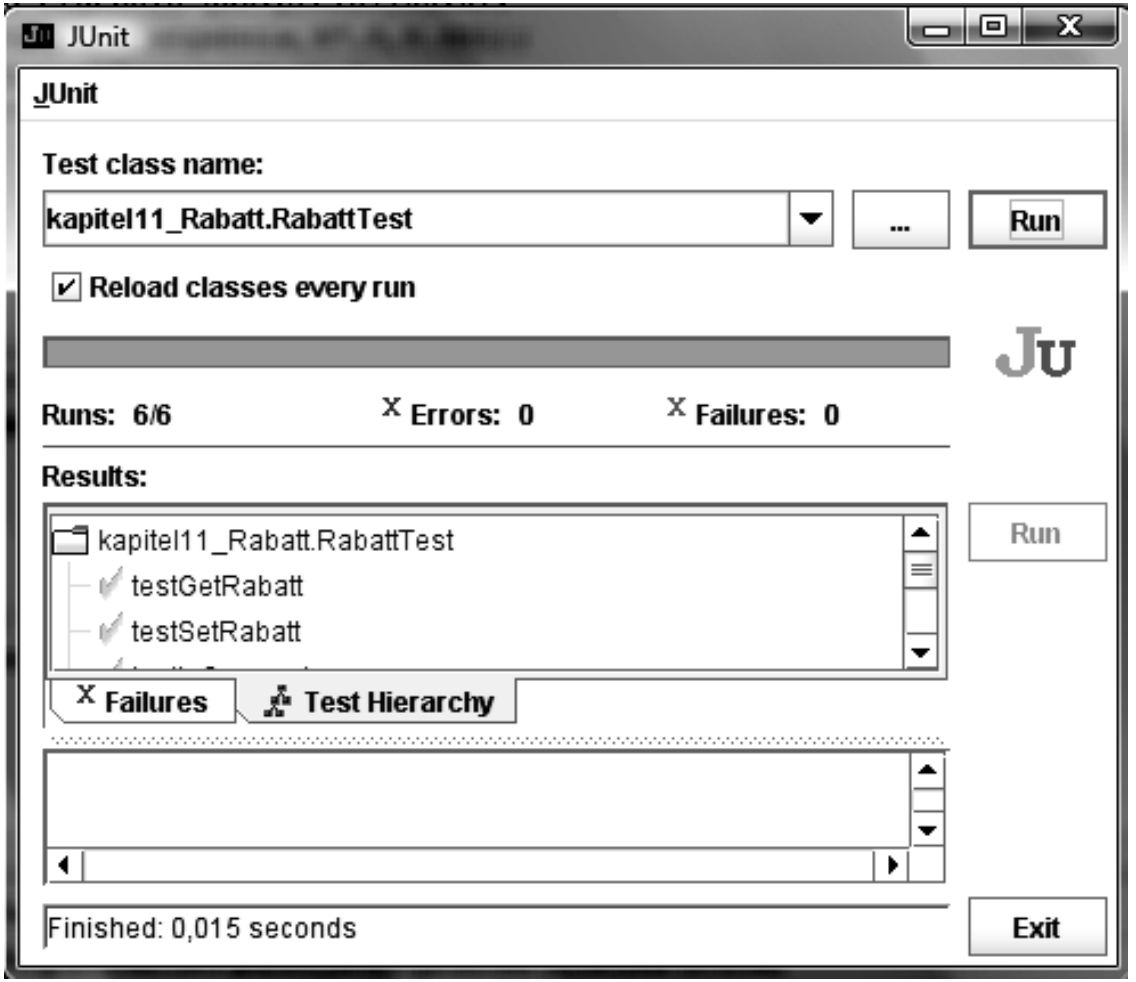
Testfall

- Vor dem Testen müssen Testfälle spezifiziert werden
- Vorbedingungen
  - Zu testende Software in klar definierte Ausgangslage bringen (z. B. Objekte mit zu testenden Methoden erzeugen)
  - Angeschlossene Systeme in definierten Zustand bringen
  - Weitere Rahmenbedingungen sichern (z. B. HW)
- Ausführung
  - Was muss wann gemacht werden (einfachster Fall: Methodenaufruf)
- Nachbedingungen
  - Welche Ergebnisse sollen vorliegen (einfachster Fall: Rückgabewerte)
  - Zustände anderer Objekte / angeschlossener Systeme

JUnit

- Framework, um den Unit-Test eines Java-Programms zu automatisieren
- einfacher Aufbau
- leicht erlernbar

- geht auf SUnit (Smalltalk) zurück
- mittlerweile für viele Sprachen verfügbar (NUnit, CPPUnit)



Konstruktiver Ansatz

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Testfälle werden in Java programmiert, keine spezielle Skriptsprache notwendig
- Idee ist inkrementeller Aufbau der Testfälle parallel zur Entwicklung
  - Pro Klasse wird mindestens eine Test-Klasse implementiert (oder in Klasse ergänzt)
- JUnit ist in Eclipse integriert
- JUnit als Library in NetBeans integriert
- sonst muss das Paket **`junit.jar`** zu CLASSPATH hinzugefügt werden
- seit JUnit 4.11 wird neben JUnit auch Hamcrest-Matcher-Bibliothek benötigt (vorher fest integriert)
- JUnit 5 bringt einige Änderungen (besser UND schlechter)

Spezifikationsausschnitt

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Gegeben sei eine Aufzählung mit den folgenden Werten

```
package verwaltung.mitarbeiter;
public enum Fachgebiet {
  ANALYSE, DESIGN, JAVA, C, TEST
}
```

- Zu entwickeln ist eine Klasse Mitarbeiter, wobei jedes Mitarbeiterobjekt
  - eine eindeutige Kennzeichnung (id) hat
  - einen änderbaren Vornamen haben kann
  - einen änderbaren Nachnamen mit mindestens zwei Zeichen hat
  - eine Informationssammlung mit maximal drei Fachgebieten hat, die ergänzt und gelöscht werden können

Klasse Mitarbeiter (1/4) - fast korrekt

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
package verwaltung.mitarbeiter;
import java.util.HashSet;
import java.util.Set;
public class Mitarbeiter {
  private int id;
  private static int idGenerator = 100;
  private String vorname;
  private String nachname;
  private Set<Fachgebiet> fachgebiete;

  public Mitarbeiter(String vorname, String nachname) {
    if (nachname == null || nachname.length() < 2)
      throw new IllegalArgumentException(
          "Nachname mit mindestens zwei Zeichen");
    this.vorname = vorname;
    this.nachname = nachname;
    this.id = idGenerator++;
    this.fachgebiete = new HashSet<>();
  }
```

Klasse Mitarbeiter (2/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getVorname() { return vorname; }

public void setVorname(String vorname) {
  this.vorname = vorname; }

public String getNachname() { return nachname; }

public void setNachname(String nachname) {
  this.nachname = nachname;}

public Set<Fachgebiet> getFachgebiete() {
  return fachgebiete;}

public void setFachgebiete(Set<Fachgebiet>
 fachgebiete) {
  this.fachgebiete = fachgebiete;
}
```

Klasse Mitarbeiter (3/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public void addFachgebiet(Fachgebiet f){
  this.fachgebiete.add(f);
  if(this.fachgebiete.size() > 3){
    this.fachgebiete.remove(f);
    throw new IllegalArgumentException(
                              "Maximal 3 Fachgebiete");
  }
}

public void removeFachgebiet(Fachgebiet f){
  this.fachgebiete.remove(f);
}

public boolean hatFachgebiet(Fachgebiet f){
  return this.fachgebiete.contains(f);
}
```

Klasse Mitarbeiter (4/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
@Override
public int hashCode() { return id; }

@Override
public boolean equals(Object obj) {
  if (obj == null || getClass() != obj.getClass())
    return false;
  Mitarbeiter other = (Mitarbeiter) obj;
  return (id == other.id);
}

@Override
public String toString(){
  StringBuilder erg = new StringBuilder(this.vorname
      + " " + this.nachname + " (" + this.id +")[ ");
  for(Fachgebiet f: this.fachgebiete)
    erg.append(f + " ");
  erg.append("]");
  return erg.toString();
}
```

Testen von Hand

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public static void main(String... s){
  Mitarbeiter m = new Mitarbeiter("Uwe","Mey");
  m.addFachgebiet(Fachgebiet.ANALYSE);
  m.addFachgebiet(Fachgebiet.C);
  m.addFachgebiet(Fachgebiet.JAVA);
  System.out.println(m);
  m.addFachgebiet(Fachgebiet.TEST);
}
```

```
Uwe Mey (100)[ C JAVA ANALYSE ]
Exception in thread "main"
java.lang.IllegalArgumentException: Maximal 3 Fachgebiete
  at
verwaltung.mitarbeiter.Mitarbeiter.addFachgebiet(Mitarbeiter
.java:58)  at
verwaltung.mitarbeiter.Mitarbeiter.main(Mitarbeiter.java:99)
```

Testen mit JUnit

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Tests werden mit Methoden durchgeführt, die mit Annotation @Test markiert sind
- Testmethoden haben typischerweise keinen Rückgabewert (nicht verboten)
- Tests stehen typischerweise in eigener Klasse; für Klasse X eine Testklasse XTest (können auch in zu testender Klasse stehen)
- Mit Klassenmethoden der Klasse Assertions werden gewünschte Eigenschaften geprüft

```
Assertions.assertTrue(m.getVorname().equals("Ute")
                                , "erwarteter Vorname Ute");
```

- generell nutzbar: **`Assertions.assertEquals(erwartet, gefunden)`**, Fehlermeldung optional letzter Parameter, Ausgabe im Fehlerfall
- JUnit steuert Testausführung, Verstöße bei Prüfungen werden protokolliert

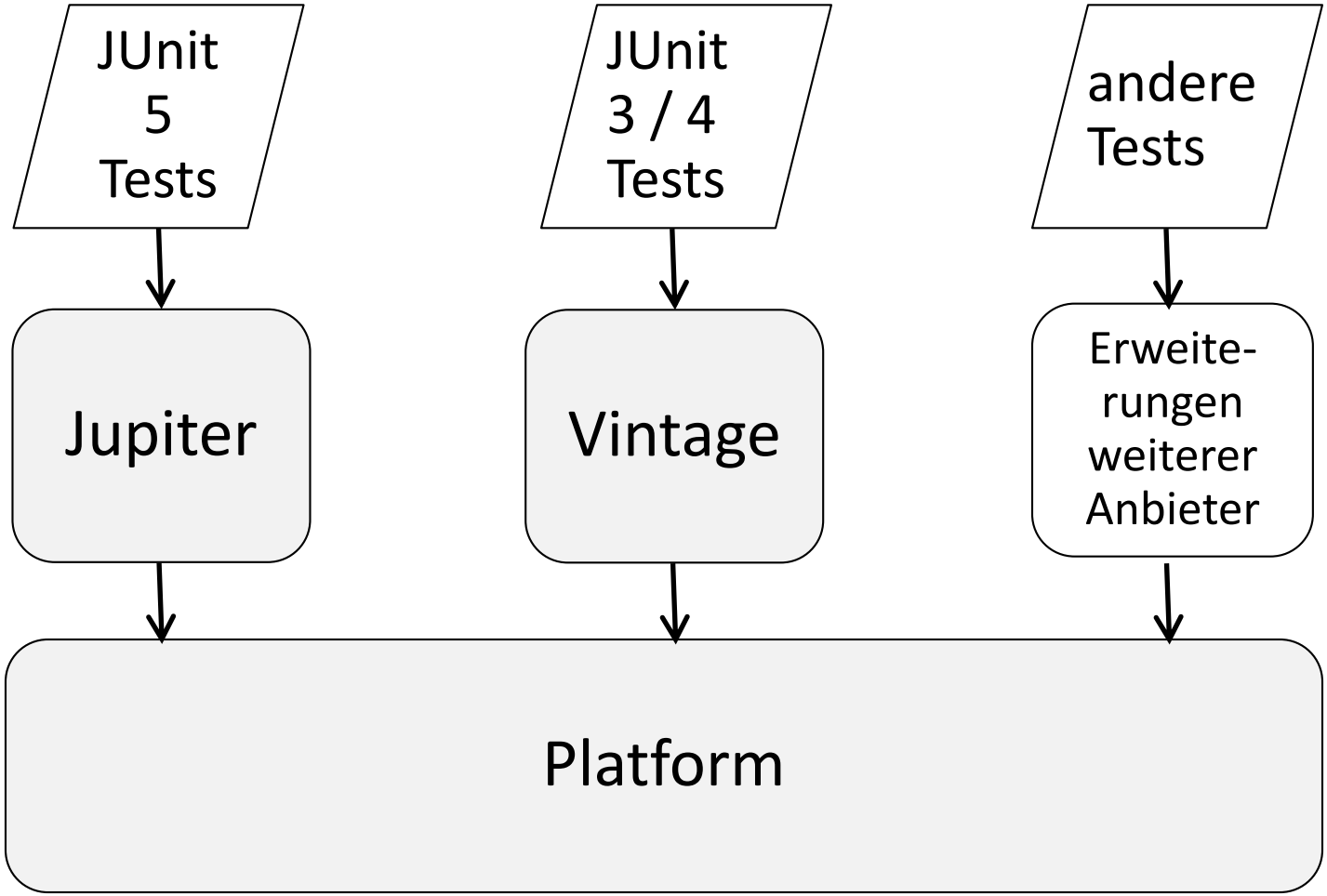
JUnit 5 - Ansatz

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Modularer Aufbau aus Teilprojekten für unterschiedliche Funktionalität
- JUnit Platform: Ausführung von Tests (TestEngines), Ausführung in Konsole, Plugins für Build-Werkzeuge (Maven, Gradle)
- JUnit Jupiter: Aufbau der Testlogik, Modell zur Erweiterung, Parametrisierung
- JUnit Vintage: Möglichkeit zur Ausführung von JUnit 3 und 4 Tests (wenn diese keinen eigenen TestRunner benötigen!)

- https://junit.org/junit5/docs/current/user-guide/
- B. Garcia, Mastering Software Testing with JUnit 5, Packt Publishing, Birmingham (UK), 2017

JUnit 5 (grau) – Architektur-Idee



JUnit 5 – Nutzungsbeispiele - Zusicherungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
@Test // Reihenfolge von TestNG (Pruefung, Fehlerkommentar)
public void testKonstruktor() {
  Mitarbeiter m = new Mitarbeiter("Ute", "Mai");
  Assertions.assertTrue(m.getVorname().equals("Ute")
      , " kein korrekter Vorname");
}

@Test // bessere Variante, alle Assertions werden ausgefuehrt
public void testKonstruktor2() {
  Mitarbeiter m = new Mitarbeiter("Ute", "Mai");
  Assertions.assertAll("Ueberschrift optional"
     , () -> Assertions.assertTrue(m.getVorname().equals("Ute")
                  , " kein korrekter Vorname")
     , () -> Assertions.assertTrue(m.getNachname().equals("Mai")
                  , " kein korrekter Nachname"));
}
```

JUnit 5 – Nutzungsbeispiele – Testfixture (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Mitarbeiter2Test { // mit Test-Fixture

    private Mitarbeiter m1;

    @BeforeEach // statt @Before (auch @AfterEach )
    public void setUp() throws Exception {
        m1 = new Mitarbeiter("Uwe", "Mey");
        m1.addFachgebiet(Fachgebiet.ANALYSE);
        m1.addFachgebiet(Fachgebiet.C);
        m1.addFachgebiet(Fachgebiet.JAVA);
    }

    @Test // Anzeige von alternativen Testnamen
    @DisplayName("Test-Fixture-Objekt soll Fachgebiet haben")
    public void testHatFaehigkeit1() {
        Assertions.assertTrue(m1.hatFachgebiet(Fachgebiet.C)
              , "vorhandene Faehigkeit fehlt");
    }
```

JUnit 5 – Nutzungsbeispiele – Testfixture (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class FixtureTest {

  @BeforeAll
  public static void setUpClass() {
     System.out.println("BeforeAll");
  }

  @AfterAll
  public static void tearDownClass() {
    System.out.println("AfterAll");
  }

  @BeforeEach public void setUp(){System.out.println("Before");}

  @AfterEach public void tearDown(){System.out.println("After");}

  @Test public void test1(){System.out.println("test1");}

  @Test public void test2(){ System.out.println("test2");}
}
```

```
BeforeAll
Before
test1
After
Before
test2
After
AfterAll
```

JUnit 5 – Nutzungsbeispiele – Exceptions (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
@Test // klassisch mit Assertions.fail() geht weiterhin
public void testKonstruktor() {
  Executable auszufuehren = () -> new Mitarbeiter(null, null);
  Assertions.assertThrows(IllegalArgumentException.class
                , auszufuehren);
}

@Test
public void testKonstruktor2() {
  Assertions.assertThrows(IllegalArgumentException.class
       , () -> {new Mitarbeiter(null, null);}
       , "erwartete Exception nicht geworfen");
}
```

JUnit 5 – Nutzungsbeispiele – Exceptions (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
@Test  // Analyse der erhaltenen Exception
public void testKonstruktor3() {
  Throwable ex = Assertions
                 .assertThrows(IllegalArgumentException.class
                       , () -> new Mitarbeiter(null, null) );
  Assertions.assertTrue(ex.getMessage().contains("Nachname")
                 , " 'Nachname fehlt in Exception-Text");
}

@Test
public void testAddFaehigkeit1() {
  Executable code = () -> m1.addFachgebiet(Fachgebiet.TEST);
  Assertions.assertThrows(IllegalArgumentException.class
      , code, "erwartete Exception nicht geworfen");
}

// @Test(expected = IllegalArgumentException.class) nicht mehr
```

JUnit 5 – Nutzungsbeispiele – Assumptions

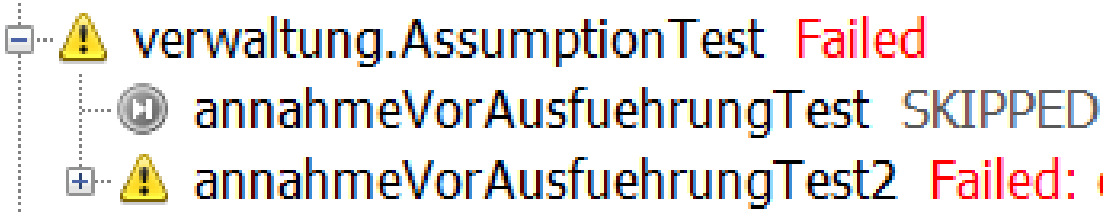
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// public class AssumptionTest {

    @Test //Test wird nur ausgefuehrt, wenn Annahme erfuellt
    public void annahmeVorAusfuehrungTest(){
        Assumptions.assumeTrue(42 == 43);
        System.out.println(" 42 == 43 ");
        Assertions.assertTrue( 1 == 2);
    }

    @Test
    public void annahmeVorAusfuehrungTest2(){
        Assumptions.assumeTrue(42 == 43 - 1);
        System.out.println(" 42 == 43 - 1 ");
        Assertions.assertTrue( 1 == 2);
    }
}
```

```
42 == 43 - 1
```



JUnit 5 – Nutzungsbeispiele – Parameter (1/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

```
// ab JUnit 5.1 eine Methode angebbar, die Stream<Arguments>
// liefern muss; jedes Element steht fuer einen Satz
// Testdaten

public static Stream<Arguments> daten() {
  Arguments[] testdaten = {
      Arguments.of(Fachgebiet.ANALYSE, Fachgebiet.C
                                     , Fachgebiet.C),
    , Arguments.of(Fachgebiet.ANALYSE, Fachgebiet.C
                                     , Fachgebiet.ANALYSE),
    , Arguments.of(Fachgebiet.C, Fachgebiet.C, Fachgebiet.C)
  };
  return Arrays.asList(testdaten).stream();
}
```

JUnit 5 – Nutzungsbeispiele – Parameter (2/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
@ParameterizedTest
@MethodSource({"daten"})
public void testHat( Fachgebiet f1, Fachgebiet f2
                   , Fachgebiet f3) {
  System.out.println("testHat");
  Mitarbeiter m1 = new Mitarbeiter("Oh", "Ha");
  m1.addFachgebiet(f1);
  m1.addFachgebiet(f2);
  Fachgebiet hat = f3;
  Assertions.assertTrue( m1.hatFachgebiet(hat) );
}
```

```
testHat
testHat
testHat
```

JUnit 5 – Nutzungsbeispiele – Parameter (3/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// eigene Umwandlungsklasse (hier mit merkwuerdiger Rueckgabe)
public class FachgebietConverter
        extends SimpleArgumentConverter {

  @Override
  protected Object convert(Object o, Class<?> type)
                          throws ArgumentConversionException {
    System.out.println("o: " + o
                     + " type: " + type.getSimpleName());
    // ueblich waere aus o passendes Objekt zu konstruieren
    return Fachgebiet.C;
  }

}
```

JUnit 5 – Nutzungsbeispiele – Parameter (4/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
 @ParameterizedTest
 @MethodSource("daten")
 public void testHatNicht(
     @ConvertWith(FachgebietConverter.class) Fachgebiet f1,
     @ConvertWith(FachgebietConverter.class) Fachgebiet f2,
     @ConvertWith(FachgebietConverter.class) Fachgebiet f3)
{
   System.out.println("testHatNicht: " + f1 + f2 + f3);
   Mitarbeiter m1 = new Mitarbeiter("Oh", "Ha");
   m1.addFachgebiet(f1);
   m1.addFachgebiet(f2);
   Fachgebiet hat = f3;
   Assertions.assertFalse(m1
       .hatFachgebiet(
           Fachgebiet.ANALYSE));
 }
```

```
o: ANALYSE type: Fachgebiet
o: C type: Fachgebiet
o: C type: Fachgebiet
testHatNicht: CCC
o: ANALYSE type: Fachgebiet
o: C type: Fachgebiet
o: ANALYSE type: Fachgebiet
testHatNicht: CCC
o: C type: Fachgebiet
o: C type: Fachgebiet
o: C type: Fachgebiet
testHatNicht: CCC
```

JUnit 5 – Nutzungsbeispiele – Parameter (5/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// konkrete Angaben von Werten, nutzt Konstruktor (wenn da)
// alternativ auch @ConvertWith nutzbar
public Mitarbeiter(String nach){
  this.nachname = nach;
  this.fachgebiete = new HashSet<Fachgebiet>();
  this.id = idGenerator++;
  System.out.println("fuer Tests");
}
```

```
@ParameterizedTest(name = "{0} and {1}")
@ValueSource(strings = {"Edna, Meier", "Kemal, Schmidt"})
public void testValSource(Mitarbeiter m) {
      System.out.println("m: " + m);
}
```

```
fuer Tests
m: null Edna, Meier (116)[ ]
fuer Tests
m: null Kemal, Schmidt (117)[ ]
```

JUnit 5 – Nutzungsbeispiele – Parameter (6/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// kommaseparierte interne Listen nutzbar
@ParameterizedTest(name = "{0} and {1}")
@CsvSource({"Edna, 'de, Meijer'", "Kemal, Schmidt"})
public void testCsvIntern(String vor, String nach) {
  System.out.println("csv intern: "
                     + new Mitarbeiter(vor, nach));
}
```

```
csv intern: Edna de, Meijer (116)[ ]
csv intern: Kemal Schmidt (117)[ ]
```

```
@ParameterizedTest
@CsvFileSource(resources ={"/bsp.csv"
   ,"/bsp.csv"}, numLinesToSkip = 1)
public void testWithCsvFileSource(
          String v, String n, int a) {
  System.out.printf("csv: %s %s %d\n"
                    , v, n, a);
}
```

```
bsp.csv:
Vorname, Nachname, alter
James T., Kirk, 87
 , Spock, 83
```

```
csv: James T. Kirk 87
csv: null Spock 83
csv: James T. Kirk 87
csv: null Spock 83
```

JUnit 5 – Nutzungsbeispiele – Parameter (7/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// Tests fuer alle Werte eines Enums
@ParameterizedTest
@EnumSource(Fachgebiet.class)
public void testMitEnumSource(Fachgebiet f) {
    System.out.println(" Fachgebiet: " + f);
}
```

```
Fachgebiet: ANALYSE
Fachgebiet: DESIGN
Fachgebiet: JAVA
Fachgebiet: C
Fachgebiet: TEST
```

```
@ParameterizedTest
@EnumSource(value = Fachgebiet.class, names = {"JAVA", "TEST"}
                                    , mode = Mode.INCLUDE)
public void testMitEnumSource2(Fachgebiet f) {
    System.out.println(" Fachgebiet2: " + f);
}
// INCLUDE ist default-Wert
```

```
Fachgebiet2: JAVA
Fachgebiet2: TEST
```

JUnit 5 – Nutzungsbeispiele - Timeout

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
@Test
public void testWarteMax2Minuten() {
  Assertions.assertTimeout(Duration.ofMinutes(2), () -> {});
}

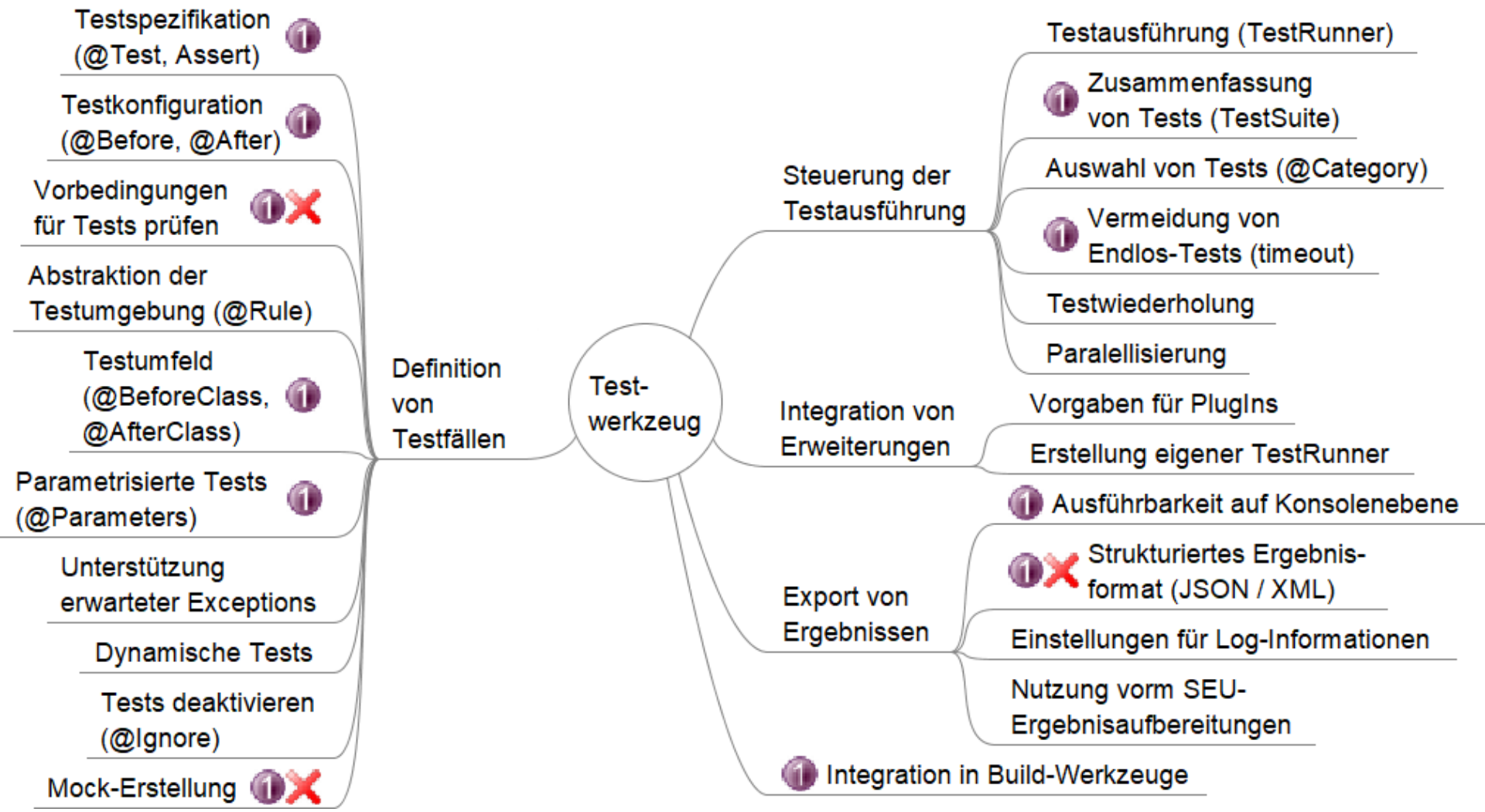
@Test
public void testMitTimeout() {
  Assertions.assertTimeout(Duration.ofMillis(10)
                          , () -> {Thread.sleep(100);}) ;
}

@Test
public void testTimeOutMitErgebnispruefung() {
  String erg = Assertions.assertTimeout(Duration.ofMinutes(1)
                          , () -> { return "moin";}) ;
  Assertions.assertEquals("moin", erg);
}
```

Auszug: Weitere JUnit 5 - Möglichkeiten

- `@EnabledOnOs({WINDOWS, MAC})`
- `@EnabledOnJre({JAVA_9, JAVA_10})`
- `@DisabledIf("Math.random() < 0.314159")`
- `@RepeatedTest(10) // Testwiederholung`

Anforderungen an ein funktionales Testwerkzeug



① elementar wichtig
X nicht in JUnit 4

3.3 Äquivalenzklassen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- was wann Testen

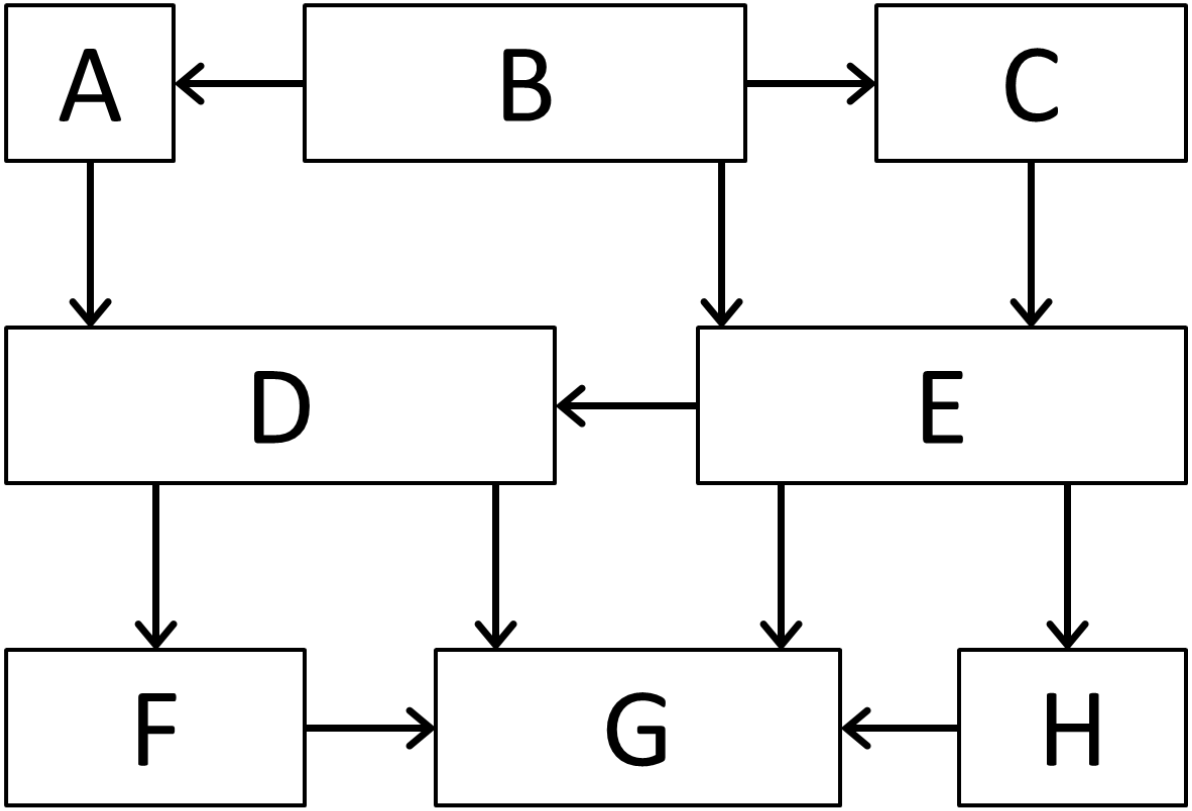
- wie Tests systematisch entwickeln
- Kompromiss: Vollständigkeit - Testauswahl

Was wann testen (1/3)

- Beispiel zeigt bereits, dass man sehr viele sinnvolle Tests schreiben kann
- Frage: Wieviele Tests sollen geschrieben werden?
  - jede Methode testen
  - doppelte Tests vermeiden
  - je kritischer eine SW, desto mehr Tests
  - Suche nach Kompromissen: Testkosten vs Kosten von Folgefehlern
  - ein Kompromiss: kein Test generierter Methoden (Konstruktoren, get, set, equals, hashCode)

Wann was testen (2/3)

- Abhängigkeiten beachten:



- sinnvoll: G F D A H E C B G F H D E A C B

Wann was testen (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- weitere Faktoren
  - Kritikalität einer Komponente
  - häufig benutzte Komponente
  - in welcher Reihenfolge fertig gestellt
  - Oberflächen, erst bei nur noch wenig erwarteten Änderungen

- Um unabhängiger von der Fertigstellung anderer zu werden:
  - Programmierung immer gegen Schnittstellen
  - Schnittstellen selbst minimal passend realisieren (Mocking)

Klassen von Testfällen

In der Literatur gibt es recht unterschiedlich detaillierte Klassifizierungen von Testfällen, eine erste grobe Einteilungsmöglichkeit ist:

- Datenbezogene Testfälle: Ausgehend von der Spezifikation des zu untersuchenden Objekts werden verschiedene Eingaben überlegt, deren gewünschtes Resultat aus der Spezifikation abzuleiten ist
- Ablaufbezogene Testfälle: Es wird die Struktur des zu untersuchenden Programms analysiert und versucht, möglichst alle Ablaufalternativen (if, while) durchzuspielen

Äquivalenzklassenbildung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Äquivalenzklassenbildung zerlegt Menge in disjunkte Teilmengen
- jeder Repräsentant einer Teilmenge hat das gleiche Verhalten bzgl. einer vorgegebenen Operation
- Beispiel: Restklassen (modulo x), werden zwei beliebige Repräsentanten aus Restklassen addiert, liegt das Ergebnis immer in der selben Restklasse

- Übertragungsidee auf Tests: Eingaben werden in Klassen unterteilt, die durch die Ausführung des zu testenden Systems zu „gleichartigen“ Ergebnissen führen

Beispiele für Äquivalenzklassen von Eingaben

- erlaubte Eingabe: 1 <= Wert <= 99 (Wert sei ganzzahlig)
  - eine gültige Äquivalenzklasse: 1 <= Wert <= 99
  - zwei ungültige Äquivalenzklassen: Wert < 1, Wert > 99
- erlaubte Eingabe in einer Textliste: für ein Auto können zwischen einem und sechs Besitzer eingetragen werden
  - eine gültige Äquivalenzklasse: ein bis sechs Besitzer
  - zwei ungültige Äquivalenzklassen: kein Besitzer, mehr als sechs Besitzer
- erlaubte Eingabe: Instrumente Klavier, Geige, Orgel, Pauke
  - vier gültige Äquivalenzklassen: Klavier, Geige, Orgel, Pauke
  - eine ungültige Äquivalenzklasse: alles andere, z.B. Zimbeln

Regeln zur Bildung von Äquivalenzklassen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- man muss mögliche Eingaben kennen (aus Spezifikation)
- für einfache Zahlenparameter meist einfach:
  - Intervall mit gültigen Werten
  - eventuell Intervall mit zu kleinen und Intervall mit zu großen Werten (wenn z. B. alle **`int`** erlaubt, gibt es nur eine Äquivalenzklasse, etwas schwieriger bei **`double`**)
- explizit eine Menge von Werten vorgegeben:
  - jeder Wert eine Äquivalenzklasse dar
  - andere Eingaben möglich: zusätzliche Äquivalenzklasse
- falls nach Analyse der Spezifikation Grund zur Annahme besteht, dass Elemente einer Äquivalenzklasse unterschiedlich behandelt werden, ist die Klasse aufzuspalten

Testfallerzeugung aus Äquivalenzklassen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Die Äquivalenzklassen sind eindeutig zu nummerieren. Für die Erzeugung von Testfällen aus den Äquivalenzklassen sind zwei Regeln zu beachten:
- gültige Äquivalenzklassen:
  - möglichst viele Klassen in einem Test kombinieren
- ungültige Äquivalenzklassen:
  - Auswahl eines Testdatums aus einer ungültigen Äquivalenzklasse
  - Kombination mit Werten, die ausschließlich aus gültigen Äquivalenzklassen entnommen sind.
  - Grund: für alle ungültigen Eingabewerte muss eine Fehlerbehandlung existieren

Grenzwertanalyse

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Viele Software-Fehler sind auf Schwierigkeiten in Grenzbereichen der Äquivalenzklassen zurück zu führen (z.B. Extremwert nicht berücksichtigt, Array um ein Feld zu klein)
- Untersuchung von Äquivalenzklassen um die Untersuchung der Grenzen ergänzt
- Beispiel: 1<=Wert<=99 (wobei Wert ganzzahlig ist)
  - Äquivalenzklasse Int-Wert<1: obere Grenze Wert=0 (untere Grenze spielt hier keine Rolle)
  - Äquivalenzklasse Int-Wert>99: untere Grenze Wert=100 (obere Grenze spielt keine Rolle)
  - Äquivalenzklasse 1<=Int-Wert<=99 : untere Grenze Wert=1 und obere Grenze Wert=99
- Grenzfallbetrachtung geht direkt in die Testfallerzeugung ein (es gibt Ansätze, bei denen zusätzlich ein Fall mit einem Wert aus der „Mitte“ der Äquivalenzklasse genommen wird)

Äquivalenzklassenbildung und Objektorientierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Äquivalenzklassenbildung ist ein zentrales Verfahren, um systematisch Tests aufzubauen
- für Methoden von Objekten spielt neben Ein- und Ausgaben der interne Zustand häufig eine wichtige Rolle (erst wenn Methode x ausführt wurde, dann kann Methode y sinnvoll ausgeführt werden)
- Konsequenterweise muss man sich also mit dem Ein-/Ausgabeverhalten pro möglichem Objektzustand beschäftigen (was noch extrem aufwändiger sein kann)
- das bisher vorgestellte Verfahren kann in der reinen Form nur für gedächtnislose Objekte genutzt werden

Fazit zur Äquivalenzklassenanalyse

- Äquivalenzklassen zentrales Hilfsmittel bei der Testerstellung; können auf Programmcodeebene, aber auch bei Abnahmen genutzt werden
- Grundregel: Zu jedem Stück Spezifikation sollte man Tests mit Äquivalenzklassen schreiben können
- typisch ist, dass nicht alle Kombinationen von elementaren Äquivalenzklassen betrachtet werden können (was in kritischen Fällen anzustreben sein sollte)
- Ansatz: Betrachte mehrere Variablen zusammen; trotzdem kombinatorische Explosion möglich

- hilfreich wäre ein Maß, wie weit man beim Testen bereits ist (→ Überdeckungsmaße)

3.4 Überdeckungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- Kontrollflussgraph
- Anweisungsüberdeckung
- Pfadüberdeckung
- einfache Bedingungsüberdeckung
- minimale Mehrfachüberdeckung
- Herausforderung Polymorphie
- Automatische Überdeckungsberechnung

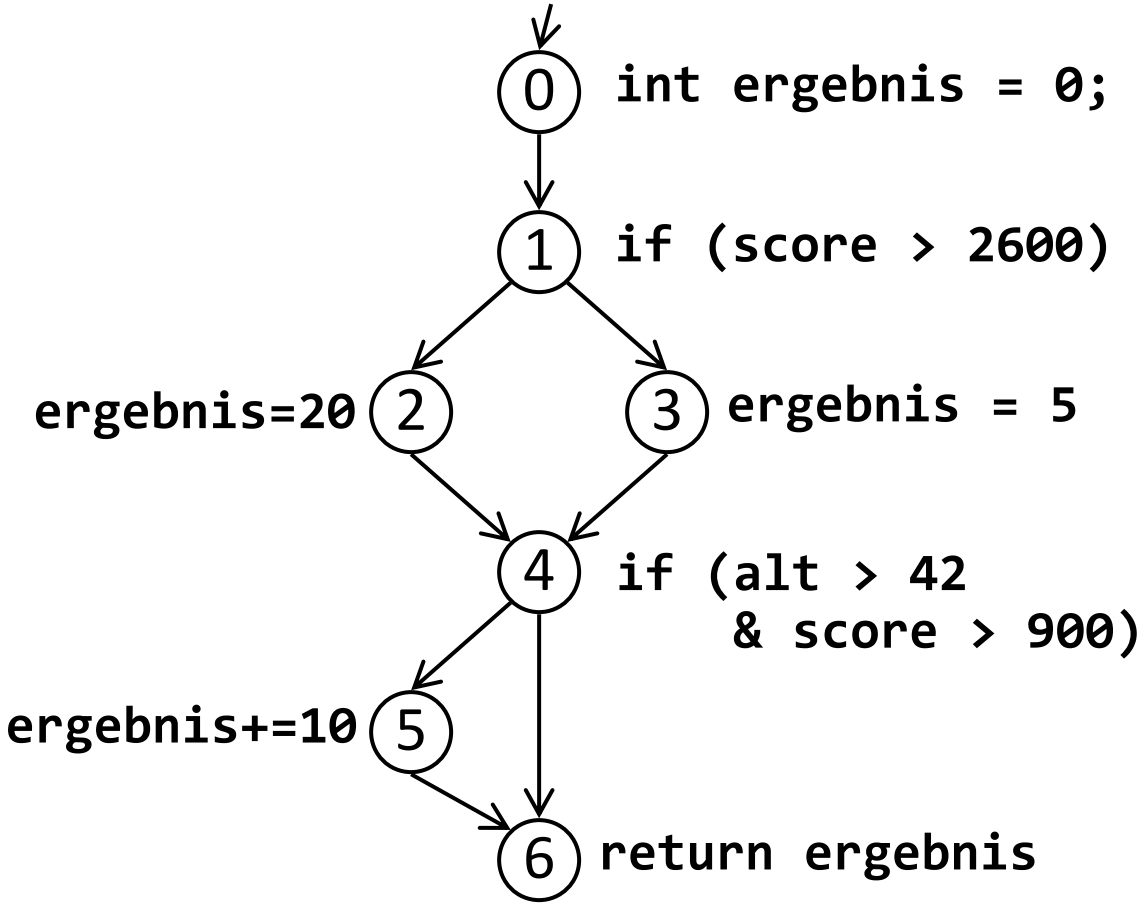
Spezifikation

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Zu entwickeln ist eine Methode mit der der Bonus eines Kunden berechnet wird. Hierbei wird der interne Score des Kunden und das Alter des Kunden berücksichtigt. Liegt der Score über 2600 wird der Ausgangswert des Bonusses mit 20, sonst mit 5 festgelegt. Liegt das Alter des Kunden über 42 und der Score über 900, wird der Bonus um 10 erhöht.

Methode mit zugehörigem Kontrollflussgraph

```
public int bonus(
      int score,
      int alt){
  int ergebnis = 0;
  if (score > 2600) {
    ergebnis = 20;
  } else {
    ergebnis = 5;
  }
  if (alt > 42
      & score > 900) {
    ergebnis += 10;
  }
  return ergebnis;
}
```



Kontrollflussgraph (KFG)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

eines Programms P ist ein gerichteter Graph

$KFG(P) =_{def} G = (V,E, VStart, VZiel)$

- V: Menge der Knoten (Anweisungen des Programms)
- E ist Teilmenge von V×V: Menge der Kanten (Nachfolgerelation bezüglich der Ausführung des Programms)
- VStart , VZiel aus V : Ausgewählte Knoten für Start, Ende des Programms
  (VZiel kann auch eine Menge von Knoten sein)

Analyse vollständiger Pfade

- Ein *vollständiger Pfad* ist eine Folge von verbundenen Knoten (über Kanten) im KFG, die mit Vstart beginnt, und mit Vziel endet
- Die möglichen Ausführungsreihenfolgen des Programms sind eine Teilmenge der vollständigen Pfade
- Wunsch: Durchlauf „repräsentativer“ vollständiger Pfade beim Test
- Überdeckung aller vollständigen Pfade ist im allgemeinen nicht ausführbar
- Ansatz: Verschiedene Approximationsstufen (Anweisungsüberdeckungstest, ..., Mehrfach-Bedingungsüberdeckungstest) für die Menge der vollständigen Pfade bei Auswahl der Testdurchläufe wählen

Anweisungsüberdeckung (C0)

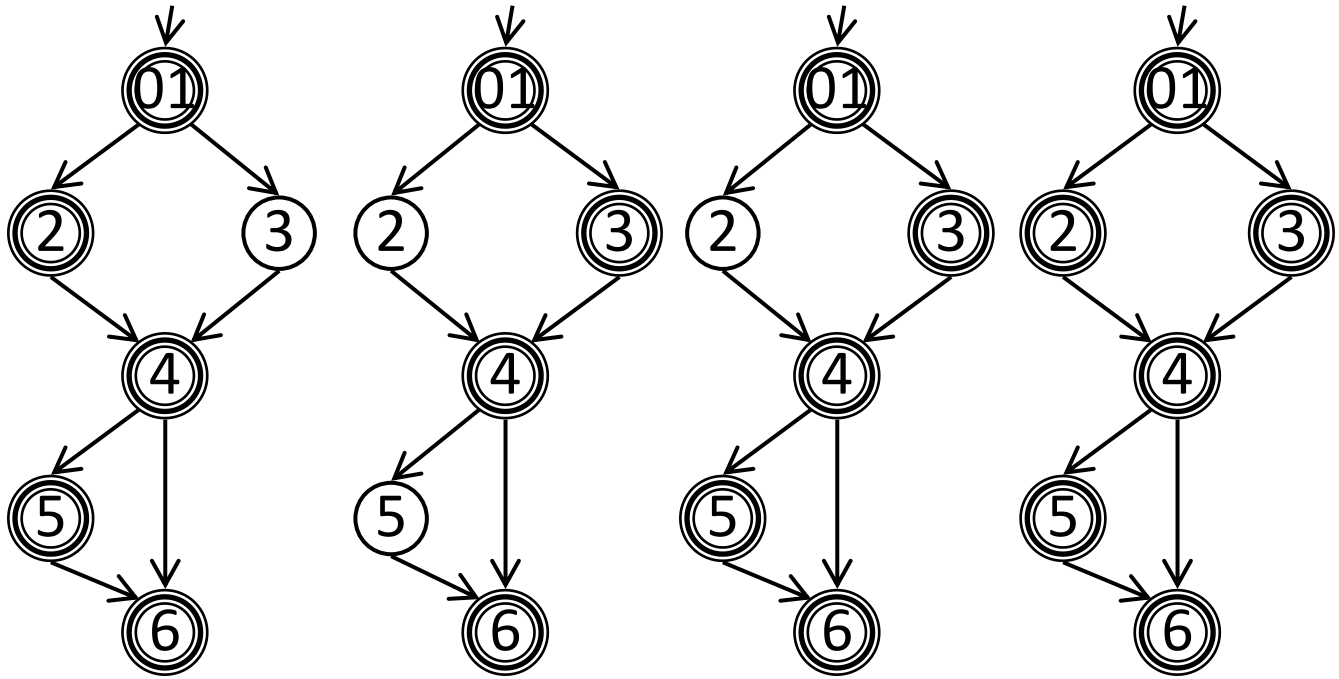
- Ziel: alle Anweisungen des Programms durch Wahl geeigneter Testdaten mindestens einmal ausführen, alle Knoten des KFG mindestens einmal besuchen.

- Testmaß $$C0 = \frac{Anzahl\ der\ ausgeführten\ Knoten}{|V|}$$

- weiterer Name: Knotenüberdeckung
- Ziel C0=1 (= 100%)
- typischerweise wird eine Menge von Testfällen benötigt, um C0=1 zu erreichen
- praktisch C0=1 oft schwierig (z. B. wg. Exceptions)
- Schwierigkeiten können auf schlechte Programmierung hindeuten

Beispiele für C0-Überdeckungen

Test	T1	T2	T3	T1+T3
`score`	2601	900	2600	
`alt`	43	43	88	
`ergebnis`	30	5	15	



C0	5/6	4/6	5/6	6/6

Zweigüberdeckung (C1)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ziel :alle Kanten des KFG überdecken, d.h. alle Pfade des Programms einmal durchlaufen

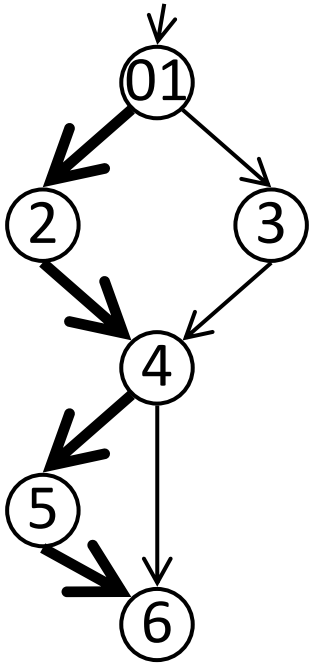
- Testmaß $$C1=\frac{Anzahl\ der\ durchlaufenen\ Kanten}{|E|}$$

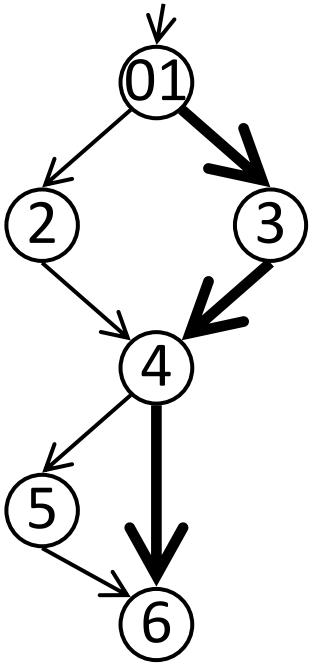
- Ziel C1=1 (= 100%)
- typischerweise wird eine Menge von Testfällen benötigt, um C1=1 zu erreichen
- C1=1 impliziert C0=1 (nicht andersherum)

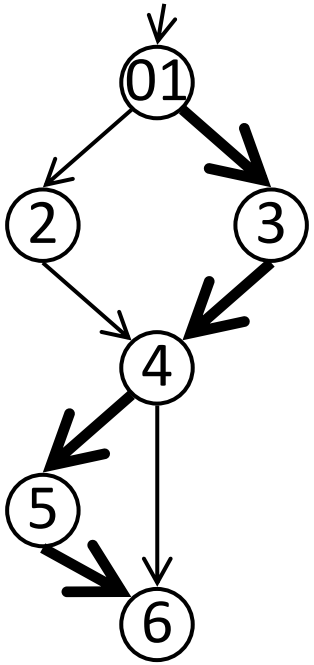
Beispiele für C1-Überdeckungen

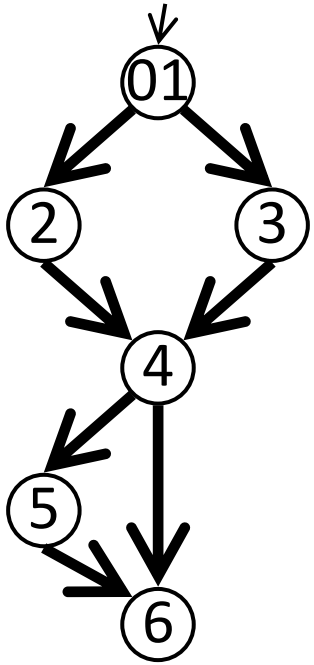
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Test	T1	T2	T3	T1+T3	T1+T2
`score`	2601	900	2600		
`alt`	43	43	88		
`ergebnis`	30	5	15		









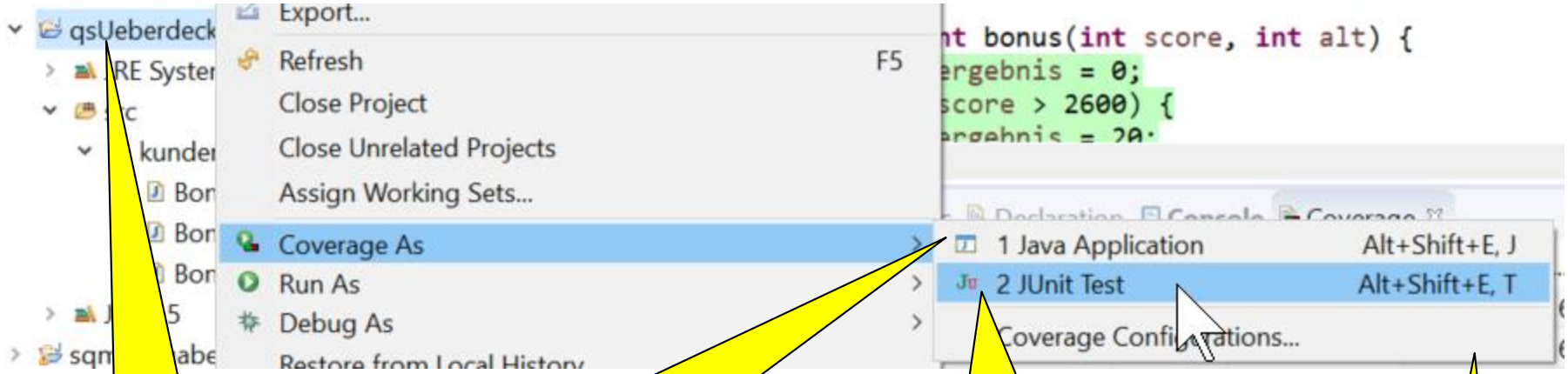
C0	4/7	3/7	4/7	6/7	7/7

Kommentare zu C0 und C1

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Vorteile der Anweisungsüberdeckung:
  - einfach
  - geringe Anzahl von Eingabedaten
  - nicht ausführbare Programmteile werden erkannt
- großer Nachteil der Anweisungsüberdeckung:
  - Logische Aspekte werden nicht überprüft
- deshalb: Zweigüberdeckungstest gilt als Minimalkriterium im Bereich des dynamischen Softwaretests,
  - schließt den Anweisungsüberdeckungstest ein,
  - fordert die Ausführung aller Zweige eines KFG,
  - jede Entscheidung mindestens einmal wahr und falsch
- Nachteile der Zweigüberdeckung:
  - Fehlende Zweige werden nicht automatisch entdeckt
  - Kombinationen von Zweigen sind unzureichend geprüft
  - Komplexe Bedingungen werden nicht analysiert

Beispiel: Coverage in Eclipse (1/2)



Rechtsklick auf Projekt

Wenn Überdeckung nur bei Ausführung (also ohne Tests) gemessen werden soll

Testüberdeckungsmessung

Detaileinstellungen

führt immer alle Testklassen aus, die auf "Test" enden

Beispiel: Coverage in Eclipse (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
3  public class Bonusrechnung {
4
5      public int bonus(int score, int alt) {
6          int ergebnis = 0;
7  All 2 branches covered.  > 2600) {
               ergebnis = 20·
```

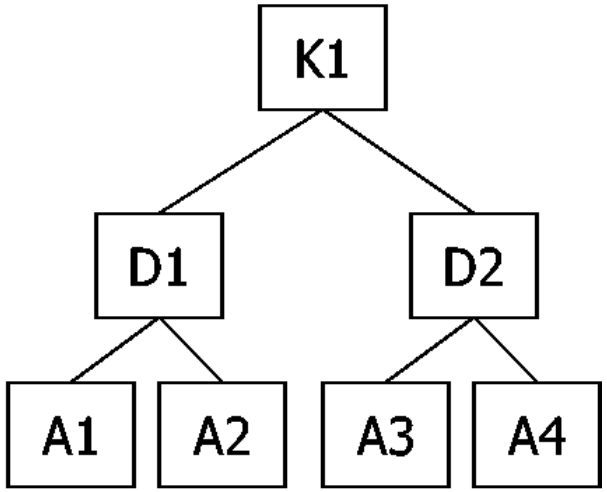
Problems | Javadoc | Declaration | Console | Coverage

Element	Covera...	Covered Ins...	Missed Instr...	Total Instruc...
qsUeberdeckungEinfuehrung	95,2 %	120	6	126
src	95,2 %	120	6	126
kunden.verwaltung	95,2 %	120	6	126
BonusrechnungTest.jav	93,6 %	73	5	78
Bonusrechnung2Test.ja	94,4 %	17	1	18
Bonusrechnung.java	100,0 %	30	0	30
Bonusrechnung	100,0 %	30	0	30
bonus(int, int)	100,0 %	27	0	27

Details zur Berechnung: http://eclemma.org/jacoco/trunk/doc/counters.html

Bedingungsüberdeckungstest

- Ziel: Teste gezielt Bedingungen in Schleifen und Auswahlkonstrukten
- Bedingungen sind Prädikate
  - A1,..., A4 atomar
    - z.B. (x==1) [auch !(x==1)]
  - zusammengesetzt
    - Konjunktion K
    - Disjunktion D
    - (((x==1) || (x==2))&&((y==3) || (y==4))) hat 7 Teilprädikate: x==1, x==2, y==3, y==4, (x==1) || (x==2), (y==3) || (y==4) , (((x==1) || (x==2))&&((y==3) || (y==4)))



- Hinweis: Unterschied zwischen || und && sowie | und & (

`if(true || 5/0==0)` läuft, `if(true | 5/0==0)` läuft nicht

Einfache Bedingungsüberdeckung (C2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ziel: alle atomaren Prädikate einmal TRUE, einmal FALSE
- Testmaß:

$$C2 = \frac{|\textit{wahre Atome}| + |\textit{falsche Atome}|}{2 * |\textit{alle Atome}|}$$

- |alle Atome| =Anzahl aller Atome,
- |wahre Atome|= Anzahl aller Atome, die nach true ausgewertet werden (analog | falsche Atome|)

Beispiele für C2-Überdeckungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

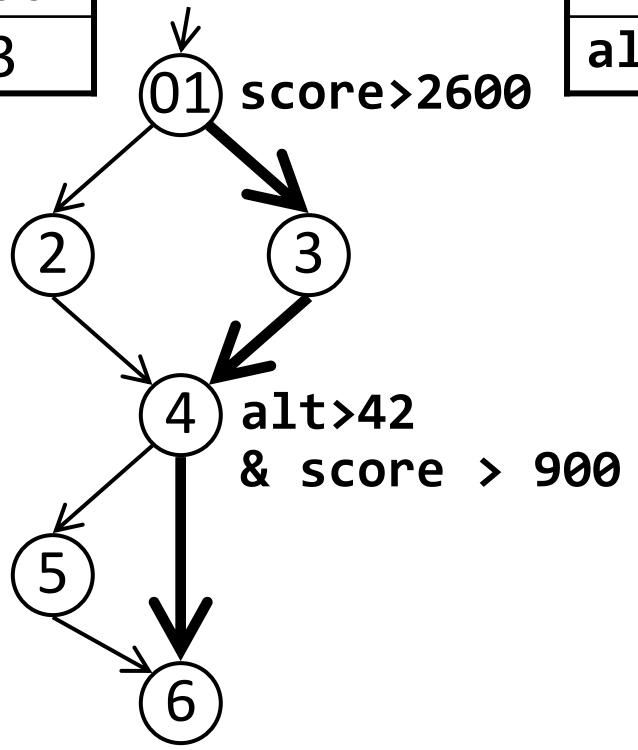
Test	T1	T2	T3	T1+T2	T4	T2+T4
`score`	2601	900	2600		2601	
`alt`	43	43	88		42	
`ergebnis`	30	5	15		20	
`score > 2600`	t	f	f	t f	t	f t
`alt > 42`	t	t	t	t	f	t f
`score > 900`	t	f	t	t f	t	f t
C2	3/6	3/6	3/6	5/6	3/6	6/6

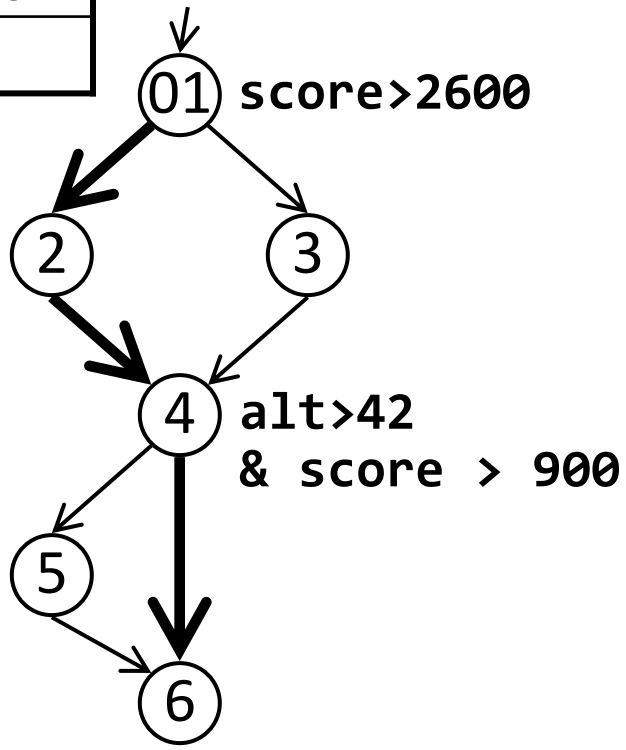
- T1+T2 ist 100% C1-Überdeckung, aber nicht C2-Überdeckung
- aus C1-Überdeckung folgt nicht C2-Überdeckung

Analyse von T2+T4 bzgl. C1

Test	T2
score	900
alt	43

Test	T4
score	2601
alt	42





- aus C2-Überdeckung muss nicht C1-Überdeckung folgen (!)

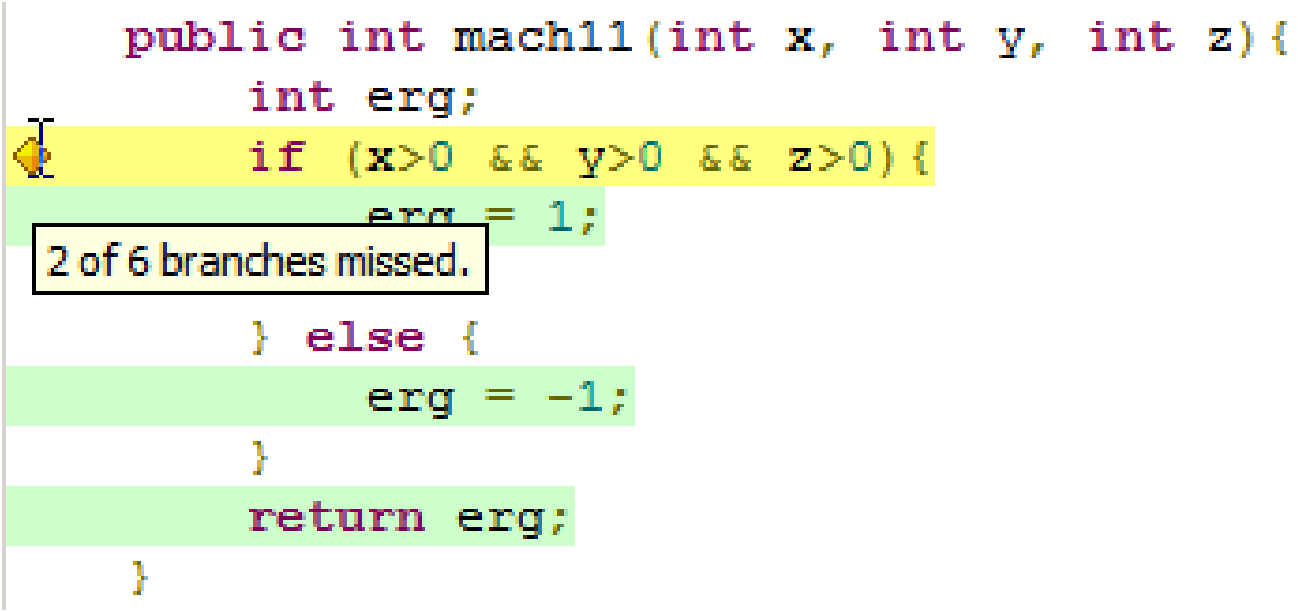
Kein Zusammenhang zwischen C1 und C2

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Die Nummerierung ist historisch gewachsen

- betrachte `if (a || b)`
  - a=true, b=false a=false, b=true
    - garantiert vollständige C2-Überdeckung
    - else-Zweig wird nicht durchlaufen, kein C1 oder C0
  - a=true, b=false a=false, b=false
    - if- und else-Zweig wird durchlaufen, damit C1 und C0
    - keine vollständige C2-Überdeckung, da b=true fehlt

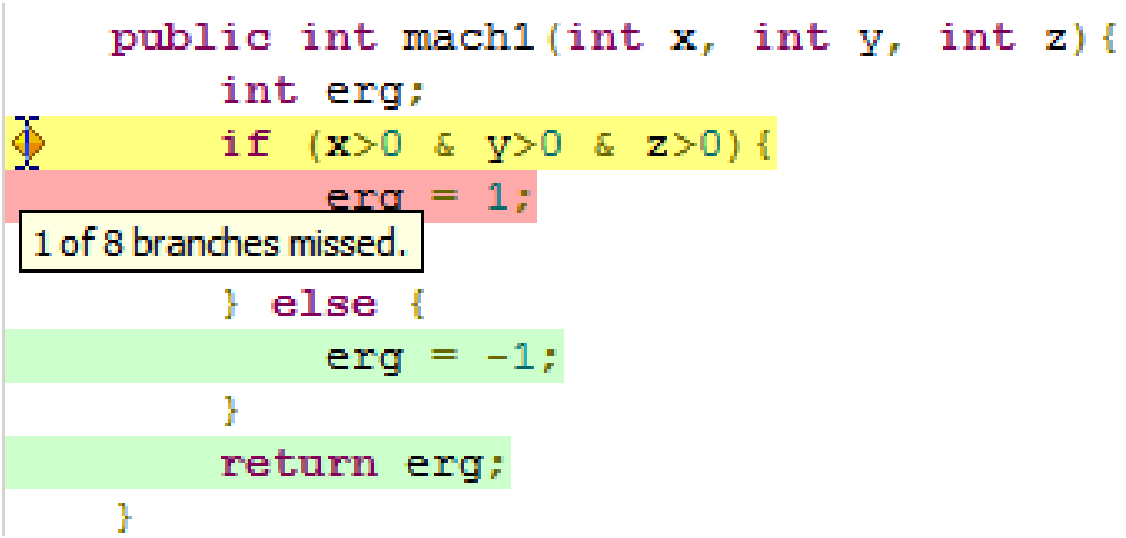
JaCoCo misst auch C2-Überdeckung




```
@Test
public void test11(){
  Assertions.assertEquals(1, b.mach11(1, 1, 1));
}
@Test
public void test13(){
  Assertions.assertEquals(-1, b.mach11(0, 0, 0));
}
```

- Achtung Kurzschlussauswertung bei zweitem Testfall

JaCoCo ohne Kurzschlussauswertung




```
@Test
public void test1(){
  Assertions.assertEquals(-1, b.mach1(0, 1, 0));
}
@Test
public void test3(){
  Assertions.assertEquals(-1, b.mach1(1, 0, 1));
}
```

- man beachte, jeder Ausdruck einmal false und true, aber Gesamtausdruck nicht (deshalb zwei Möglichkeiten mehr)

Minimale Mehrfachüberdeckung (C3)

- Ziel: alle Prädikate und Teil-Prädikate einmal TRUE, einmal FALSE
- Testmaß:

$$C3 = \frac{|\textit{wahre Teilprädikate}| + |\textit{falsche Teilprädikate}|}{2 * |\textit{alle Teilprädikate}|}$$

- Da immer auch gesamter Boolescher Ausdruck betrachtet, folgt aus 100% C3- immer 100% C1-Überdeckung

Beispiele für C3-Überdeckungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Test	T1	T2	T3	T4	T2+T4	T1+T2+T4
`score`	2601	900	2600	2601		
`alt`	43	43	88	42		
`ergebnis`	30	5	15	20		
`score > 2600`	t	f	f	t	f t	t f
`alt > 42`	t	t	t	f	t f	t f
`score > 900`	t	f	t	t	f t	t f
`alt > 42 && score >900`	t	f	t	f	f	t f

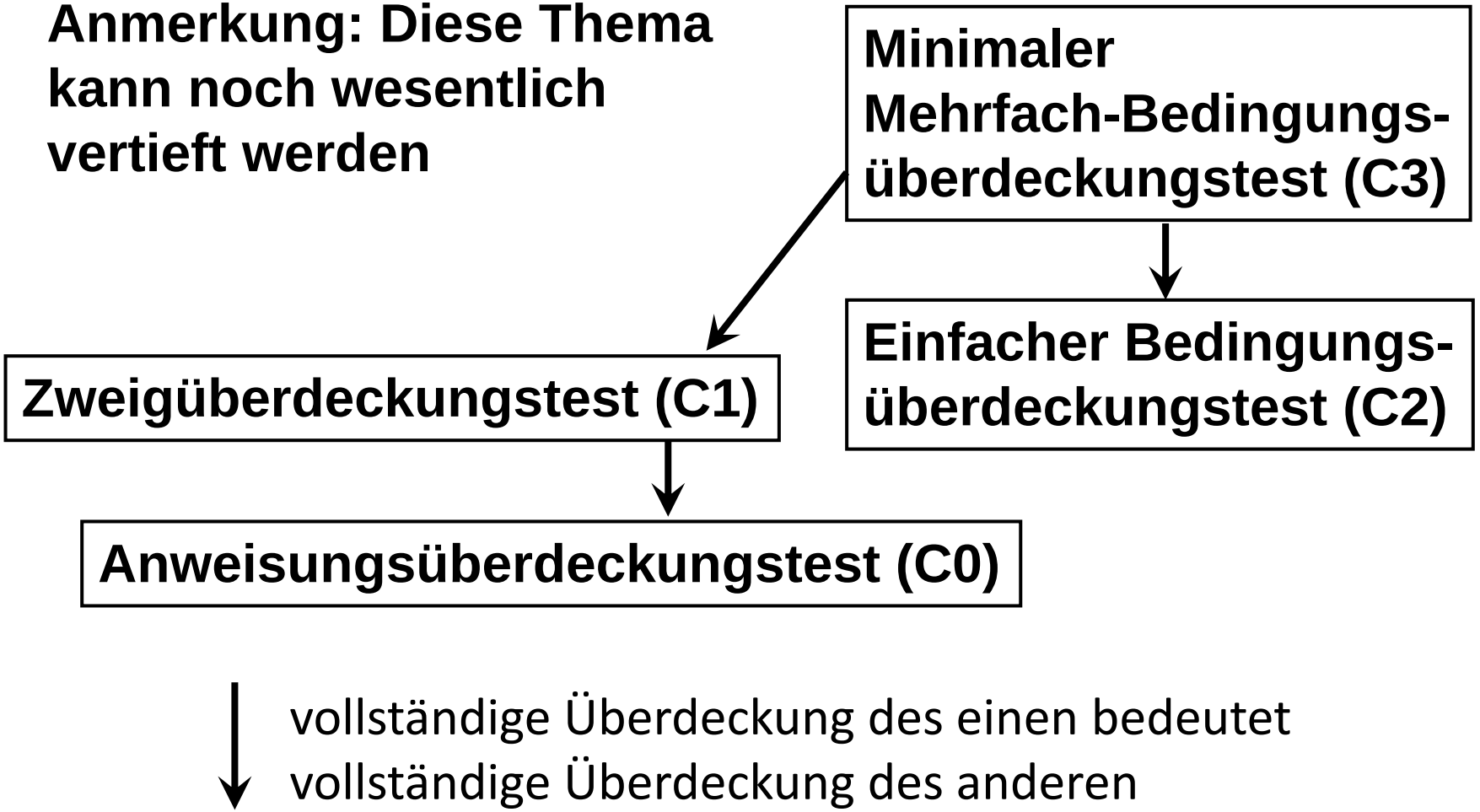
C3	4/8	4/8	4/8	4/8	7/8	8/8

- Es geht auch mit zwei Testfällen!

Zusammenhang zwischen den Testverfahren

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Anmerkung: Diese Thema kann noch wesentlich vertieft werden



↓ vollständige Überdeckung des einen bedeutet vollständige Überdeckung des anderen

(Auch) C3 findet nicht alle Probleme

- Annahme: Methode mach liefert nur positive Ergebnisse

```
public int mach(int y, int z){
        if(y==0 & z==0){
        //    t        t
        //    f        f
            return 1;
        } else {
            return y*z;
        }
    }
```

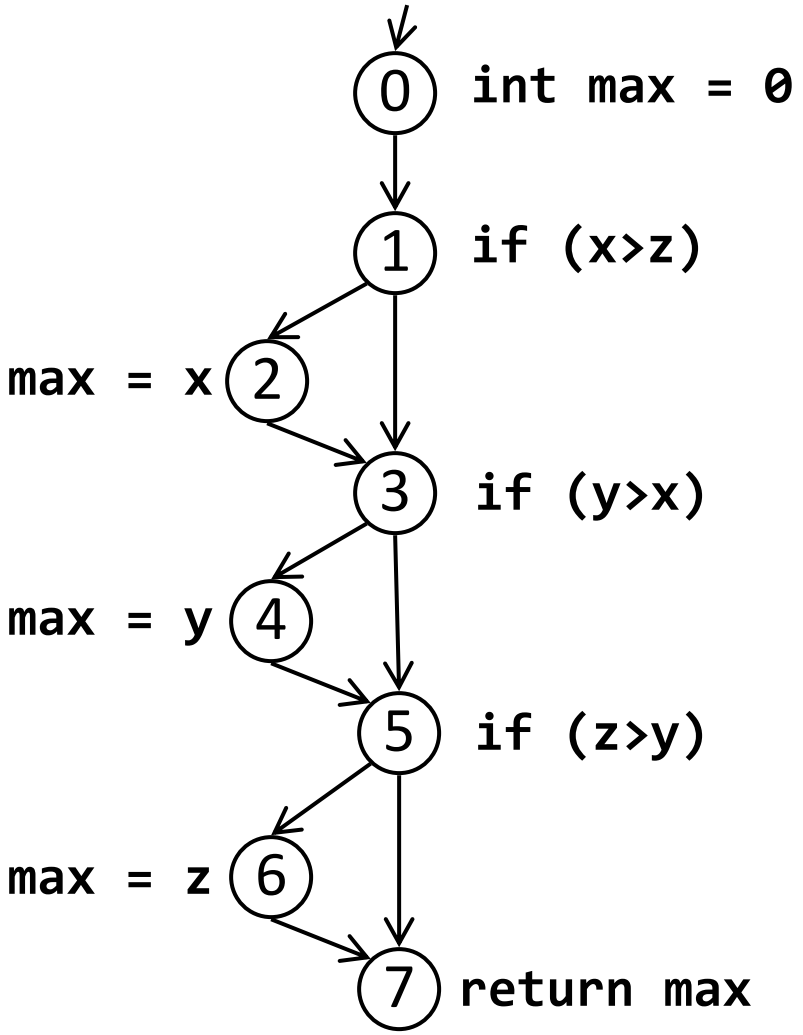
```
@Test
public void testTrueTrue(){
  Assertions.assertTrue(new Bsp().mach(0,0) > 0);
}

@Test
public void testFalseFalse(){
  Assertions.assertTrue(new Bsp().mach(1,1) > 0);
}
```

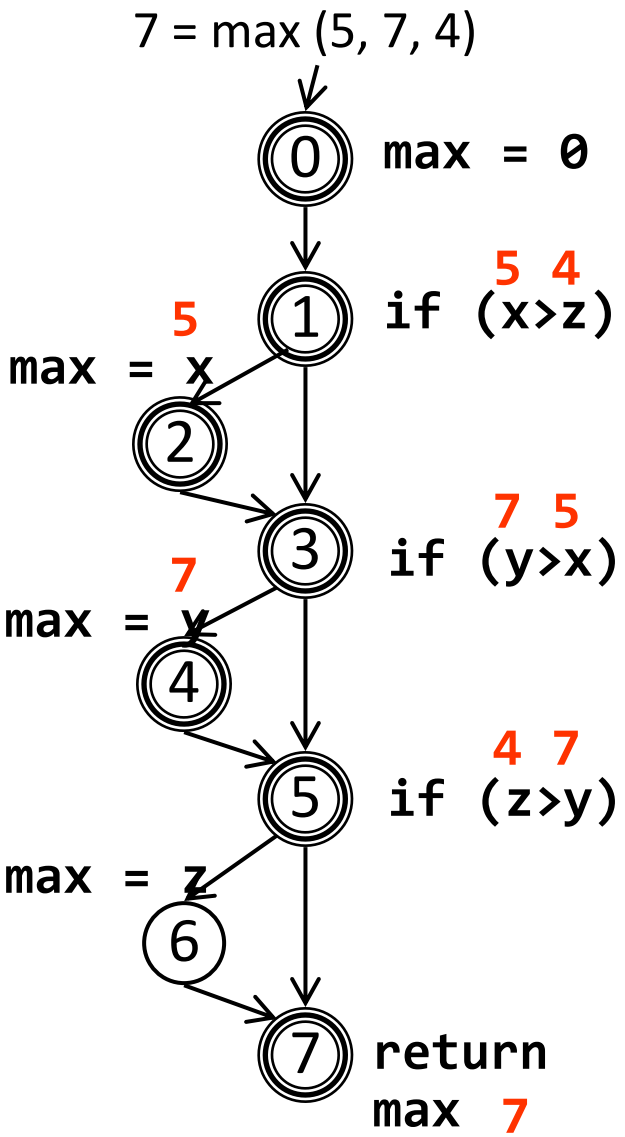
Abschlussbeispiel Ci-Überdeckungen

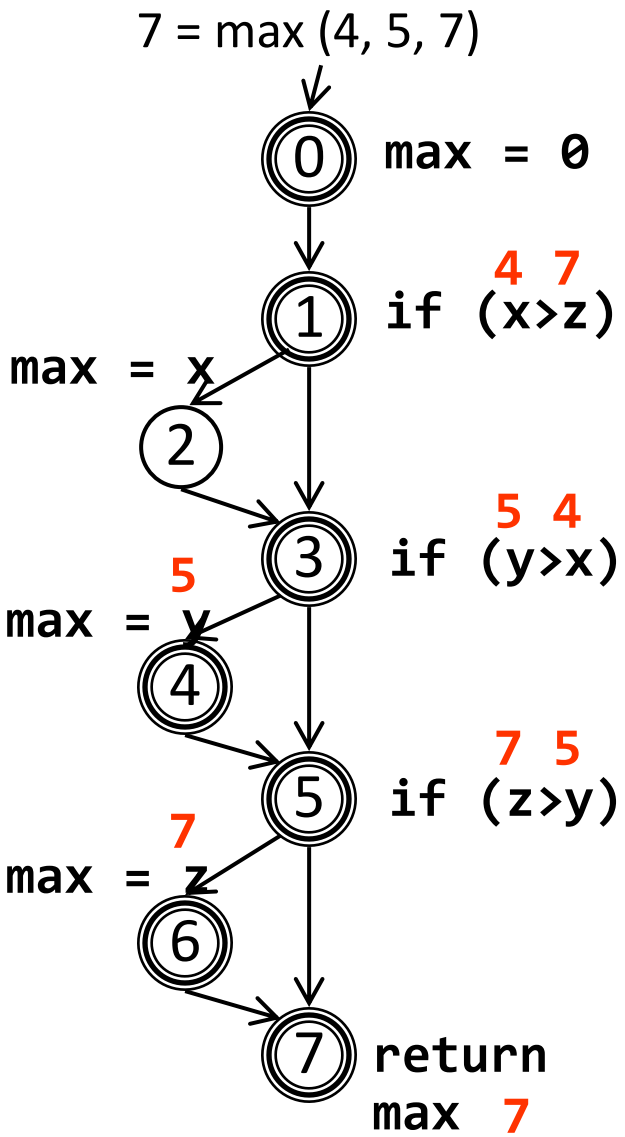
```
public int max(int x
      , int y, int z){
  int max = 0;
  if (x > z){
    max = x;
  }
  if (y > x){
    max = y;
  }
  if (z > y){
    max = z;
  }
  return max;
}
```

- Erinnerung: Suche Maximum von drei ganzen Zahlen

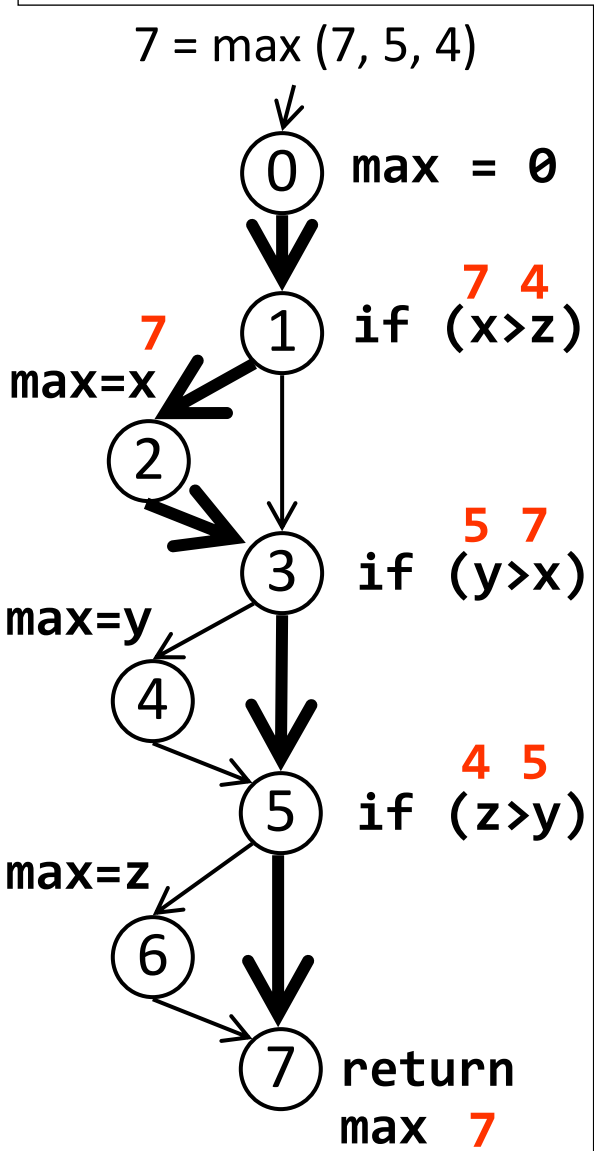


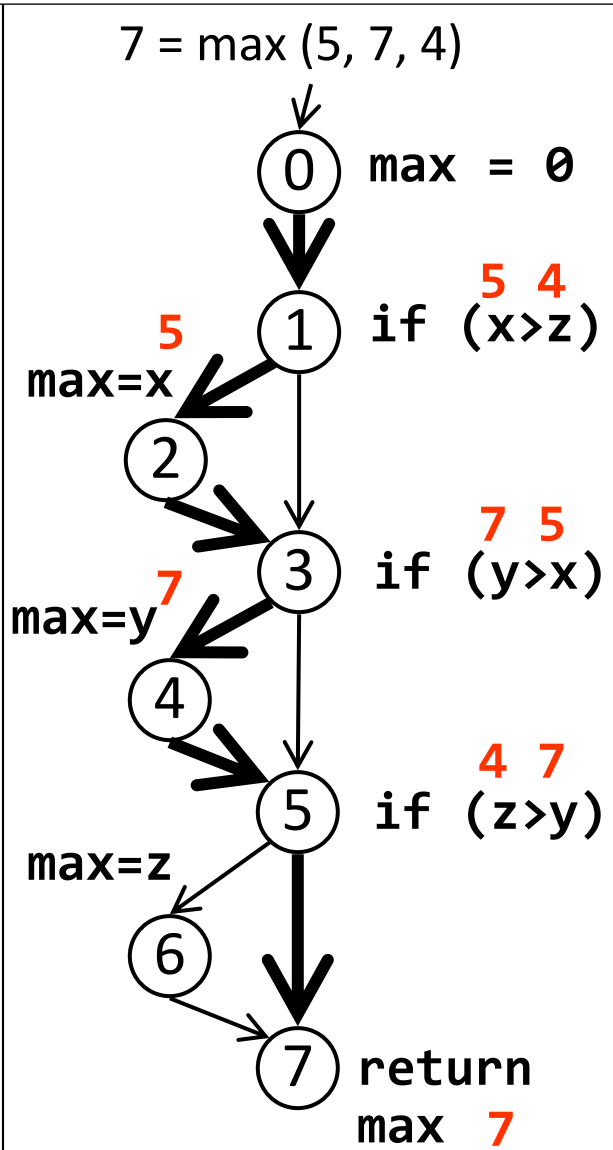
Anweisungsüberdeckung - jeder Knoten einmal

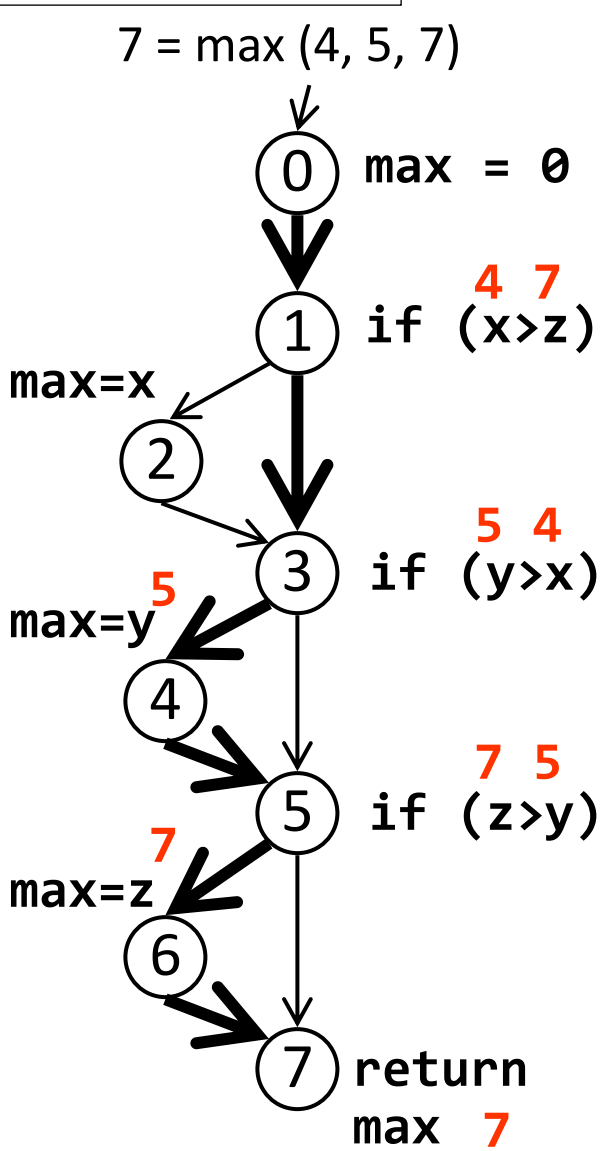




Zweigüberdeckung - jede Kante einmal







Fehler kann gefunden werden

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Kenntnisse über Äquivalenzklassen !

  x>y=z y=z>x y>x=z x=z>y z>y=x y=x>z

  z>y>x z>x>y y>z>x y>x>z x>z>y x>y>z x=y=z

- Kenntnisse über Datenflussanalyse finden Fehler!

- benötigt große Programmiererfahrung
- benötigt sehr strukturiertes Denken
- benötigt Forderung nach intensiven Tests (auch DO-178B)
- benötigt Werkzeuge zur Automatisierung
- benötigt Zeit und Geld
- benötigt Management mit Software-Verstand

Fazit: Äquivalenzklassen und Überdeckungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- letztes Beispiel zeigt deutlich, dass man trotz systematischer Analyse Fehler übersehen kann !!!
- sinnvolles Vorgehen:
  1. Testfälle aus typischen Verhalten ableiten
  2. Testfälle aus Ausnahmesituationen ableiten
  3. Äquivalenzklassen überlegen und kritische Fälle systematisch analysieren
  4. Überdeckung messen, bei niedrigen Prozentzahlen zunächst das „warum“ ergründen, dann ggfls. Testfälle für Überdeckungen konstruieren

Nie, nie nur als zentrales Testziel 90 % - Überdeckung angeben, da dann Orientierung an Nutzerprozessen verloren geht. Echte Prüfung erfolgt nur, wenn viele Asserts genutzt.

Überdeckungsproblem: Heimliche Verzweigungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

– jede Exception-Möglichkeit berücksichtigen

```
public class Sut {
  public int ganzzahligerTeiler(int x, int y){
    return x/y;
  }
}
```

– Test für 100%:

```
  @Test
  public void ueberdeckTest(){
    Sut s = new Sut();
    Assertions.assertEquals(0, s.ganzzahligerTeiler(42, 43));
  }
```

Heimliche Verzweigungen durch Vererbung (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Vererbung ist eine mächtige Möglichkeit Doppelimplementierungen zu vermeiden
- Vererbung erlaubt dynamische Polymorphie (zur Laufzeit wird erst bestimmt, welche Methode aufgerufen wird)
- Prüfung zur Laufzeit entspricht heimlichen if mit Überprüfung durch **`instanceof`**

- Generell macht Polymorphie Entwicklung einfacher, wartbarer und flexibler; macht aber Testen nicht einfacher
- Bei Testfallerstellung immer über durch Polymorphie entstehende Variationsmöglichkeiten nachdenken

Heimliche Verzweigungen durch Vererbung (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Oben {
  public int getA(){
    return 42;
  }
  public int getB(){
    return 43;
  }
}

public class Unten
            extends Oben{
  @Override
  public int getA(){
    return -42;
  }
  @Override
  public int getB(){
    return -43;
  }
}
```

Behauptung: Produkt aus getA() * getB() immer positiv: 100% durch

```
@Test
public void positivTest1(){
  Oben o1 = new Oben();
  Oben o2 = new Oben();
  Assertions.assertTrue(
      o1.getA() * o2.getB() > 0);
}

@Test
public void positivTest2(){
  Oben o1 = new Unten();
  Oben o2 = new Unten();
  Assertions.assertTrue(
      o1.getA() * o1.getB() > 0);
}
```

Vergessene Ablaufmöglichkeiten

```
public class Drei {
  private int c;

  public Drei (int c){
    this.c = c;
  }

  public int step(){
    this.c += 3;
    return this.c;
  }
}

public class DNutzer {
  private Drei ds = new Drei(0);

  public void setDs(Drei ds){
    this.ds = ds;
  }

  public int schritt(){
    return this.ds.step();
  }
}
```

Behauptung: der Differenz zwischen zwei schritt()-Aufrufen ist immer 3; 100% durch

```
@Test
public void testDS1(){
  DNutzer sut = new DNutzer();
  sut.setDs(new Drei(1));
  int start = sut.schritt();
  for(int i=0; i <10; i++){
    int neu = sut.schritt();
    Assertions.assertEquals(3
      , neu - start);
    start = neu;
  }
}

// was, wenn setDs()
 dazwischen?
```

Verteilte Systeme und Überdeckungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Prozesse A und B parallel ausgeführt
- man kann A evtl. mit Mock für B 100% überdecken
- man kann B evtl. mit Mock für A 100% überdecken
- Ergebnis sagt wenig aus

- man beachte, dass jeder Interaktionspunkt, also Zustandskombination aus (A,B), als erreichbarer Zustand geprüft werden muss
- (Anschaulich: Programmiere „A parallel B“ in nur einem Prozess mit allen Möglichkeiten; dieser wäre zu testen)

Info: Standard DO-178C (2012)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Zertifizierung Federal Aviation Administration (FAA), Software für Luftverkehrssysteme durch Standard DO-178C für requirement-based Tests and Code Coverage Analyse
- DO-178C-Levels orientieren sich an den Konsequenzen möglicher Softwarefehler: katastophal (Level A), gefährlich/schwerwiegend (Level B), erheblich (Level C), geringfügig (Level D) bzw. keine Auswirkungen (Level E)
- Je nach DO-178C-Level wird der 100%-ige Nachweis folgender Testabdeckungen (Code Coverages) verlangt:
- DO-178C Level A:
  - Modified Condition Decision Coverage (MC/DC)
  - Branch/Decision Coverage
  - Statement Coverage
- DO-178C Level B:
  - Branch/Decision Coverage
  - Statement Coverage

Auszug: Neuerungen DO-178C gegenüber DO-178B

- DO-331 "Model-Based Development and Verification Supplement to DO-178C and DO-278A" - addressing Model-Based Development (MBD) and verification and the ability to use modeling techniques to improve development and verification while avoiding pitfalls inherent in some modeling methods
- DO-332 "Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A" - addressing object-oriented software and the conditions under which it may be used
- DO-333 "Formal Methods Supplement to DO-178C and DO-278A" - addressing formal methods to complement (but not replace) testing

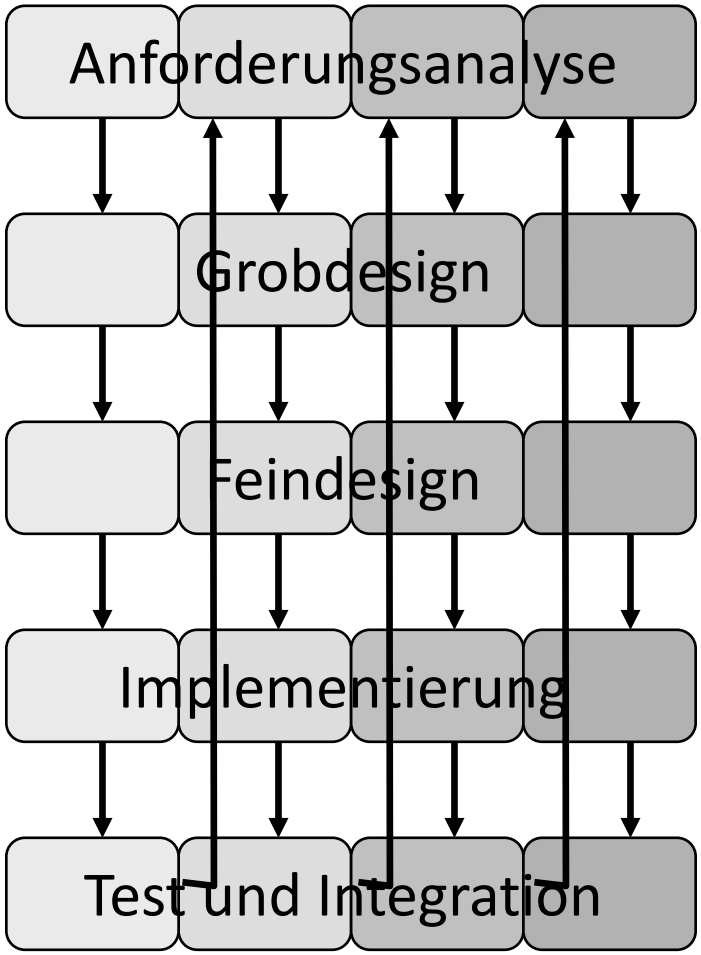
Quelle: https://en.wikipedia.org/wiki/DO-178C

3.5 Vorgehensmodelle und Testen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Vorgehensmodelle (Konzept)
- Testarten
- Testgetriebene Entwicklung
- Behaviour Driven Development (BDD) mit Cucumber

Erinnerung: Skizze eines Vorgehensmodells



Bsp.: vier Inkremente

Merkmal:
Projekt in kleine Teilschritte zerlegt, n+1-ter Schritt kann Probleme des n-ten Schritts lösen

Vorteile:
- dynamische Reaktion auf Risiken
- Teilergebnisse mit Kunden diskutierbar

mögliche Nachteile:
- schwierige Projektplanung
- schwierige Vertragssituation
- Kunde erwartet zu schnell Endergebnis

Vorgehensmodelle - Konzepte

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- generelle Thematiken überall identisch:

  Verstehen der Aufgabenstellung, Umsetzung, Überprüfung
- Unterschiede, wann, wie oft, welche Korrekturmöglichkeiten
- Unterschiede, wie Phasen/Schritte bearbeitet werden
- Unterschiede, welche Dokumentationsart genutzt wird

- Üblich: Testfallspezifikationen entstehen parallel zur Entwicklung
  - Beschreibung, was das System machen soll -> Testfälle die das Systemverhalten überprüfen
  - Beschreibung, was die Klasse machen soll -> Testfälle die das Klassenverhalten überprüfen
- offen: erst Tests entwickeln oder erst Code

Testgetriebene Entwicklung (TDD)

- Entwicklungsansatz zur Programmierung
- vor der eigentlichen Programmierung werden Tests geschrieben, die zu entwickelndes Teilprogramm prüfen
- Ergebnis: JUnit-Tests für anstehende Programmier-Teilaufgabe
- dann wird Funktionalität inkrementell entwickelt
- nach jedem Inkrement wird getestet; die Anzahl erfolgreicher Testfälle wächst von null bis zu 100%
- Vorteil: reiner Fokus darauf, was gemacht werden soll; nicht auf die Umsetzung
- Vorteil: spätere Testerstellung wäre sonst zu stark von Entwicklung beeinflusst
- Anmerkung: hilfreich ist Programmierung gegen Interfaces, da so Tests frühzeitig lauffähig

Behaviour Driven Development (BDD) - Konzept

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- ursprünglich aus TDD abgeleitete Entwicklungsmethode, initiiert von Dan North 2003
- generell Fokus auf Kommunikation zwischen allen Stakeholdern (relevanten Personen) eines SW-Projekts
- Ausschnitt auf Vorgehensweise
  - Fokus auf zentralen Aufgaben des Systems (Features)
  - Jedes Feature wird durch typische Verhaltensweisen (Scenario) beschrieben
  - Jedes Szenario wird mit einzelnen Schritten (Steps) beschrieben
  - Die Beschreibung erfolgt in (strukturierter) natürlicher Sprache; alle Stakeholder können lesen und schreiben
  - Erstellung von Szenarien dient u. a. dem Finden von offenen Problemen, die zuerst gelöst werden müssen
  - Fokus auf Features mit höchstem Stakeholder-Nutzen

BDD - technisch

- Features sind verwandt mit Use Cases, Szenarien mit Abläufen eines zugehörigen Aktivitätsdiagramms
- strukturierte natürliche Sprache wird von Werkzeugen unterstützt -> Beschreibungen werden als Abnahmetests formal umgesetzt

Werkzeuge:

- Cucumber (https://cucumber.io/)
- JBehave (http://jbehave.org/)

Literatur:

- S. Rose, M. Wynne, A. Hellesøy, The Cucumber For Java Book, The Pragmatic Programmers, LLC., Dallas, Raleigh (USA), 2015

BDD – Nutzungshinweis für folgende Folien

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Folgende Folien stellen Nutzungsmöglichkeiten von Cucumber in den Mittelpunkt; typischerweise sollte ein Werkzeug nicht im Mittelpunkt eines Vorgehensmodells stehen
- Cucumber erstellt Systemtests; da hier genutzte Beispiele für Verständlichkeit minimal gewählt, entsteht enge Verwandtschaft zu Unit-Tests

- aber: Cucumber unabhängig von BDD zur Testentwicklung nutzbar!

Beispiel-Feature

Feature: Mitarbeiter anlegen

As a einfacher Nutzer

In order um einen Mitarbeiter anzulegen

I want möchte ich nur Vor- und Nachnamen eingeben

Scenario: Korrektes Anlegen mit vollständigen Daten

Ein Mitarbeiter mit korrektem Vor- und Nachnamen wird angelegt

When Ich als Vorname "Edna" und als Nachname "Meier" eingebe

Then erhalte ich einen Mitarbeiter mit Mitarbeiternummer

And dem Vornamen "Edna"

And dem Nachnamen "Meier"

And mit 0 Fachgebieten

Beispiel-Step

- Jedem Step wird eine Methode zugeordnet
- Jedes Szenario wird einzeln abgearbeitet, d. h. jeder Step ausgeführt; scheitert er, dann das Szenario

```
@When("^Ich als Vorname \"(.*?)\" und als Nachname "
                                    + "\"(.*?)\" eingebe$")
public void ich_als_Vorname_und_als_Nachname_eingebe(
      String arg1 , String arg2) throws Throwable {
  this.mitarbeiter = new Mitarbeiter(arg1, arg2);
}
```

- Annotationsvariante:

```
@When("Ich als Vorname {string} und als Nachname "
                                    + "{string} eingebe")
```

3.6 Konstruktive Qualitätssicherung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- Idee
- Coding Guidelines
- Werkzeugeinstellungen
- weitere Maßnahmen

Ansatz

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- die analytische Qualitätssicherung greift erst, wenn ein Produkt erstellt wurde
- interessant ist der Versuch, Qualität bereits bei der Erstellung zu beachten
- typische konstruktive Qualitätsmaßnahmen sind
  - Vorgabe der SW-Entwicklungsumgebung mit projekteigenem Werkzeughandbuch, was wann wie zu nutzen und zu lassen ist
  - Stilvorgaben für Dokumente und Programme (sogenannte Coding-Guidelines)
- Die Frage ist, wie diese Maßnahmen überprüft werden

Coding Guidelines

- Detailliertes Beispiel: Taligent-Regeln für C++ (http://root.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/index.html)
- Sun hat auch Regeln für Java herausgegeben (nicht ganz so stark akzeptiert)
- z. B. Eclipse-Erweiterung Checkstyle
- Generell gibt es Regeln
  - zur Kommentierung,
  - zu Namen von Variablen und Objekten (z.B. Präfix-Regeln),
  - zum Aufbau eines Programms (am schwierigsten zu formulieren, da die Programmarchitektur betroffen ist und es nicht für alle Aspekte „die OO-Regeln“ gibt)

Beispiel-Coding-Regel

Line wrapping for `if` statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();           //MAKE THIS LINE EASY TO MISS
}

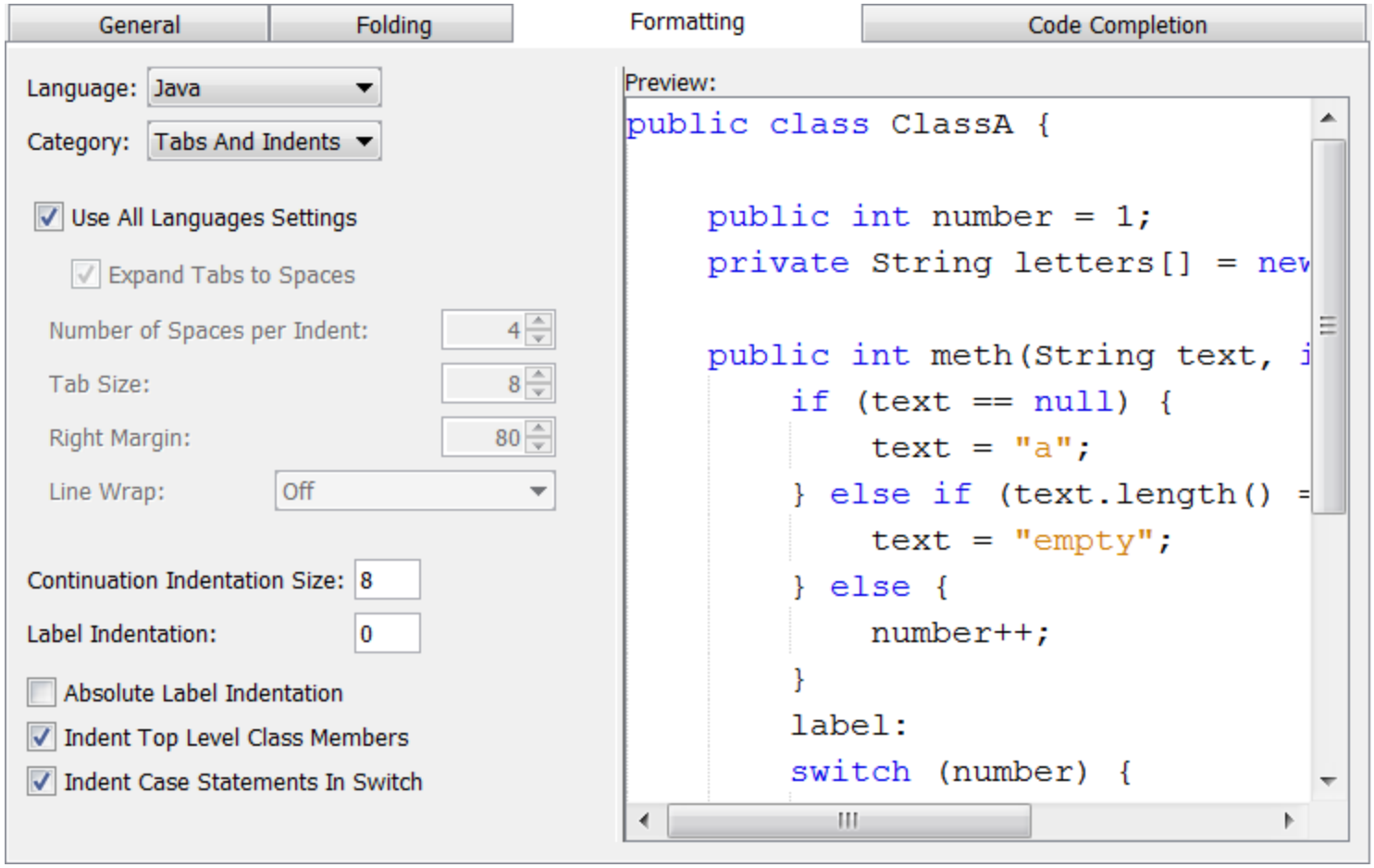
//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

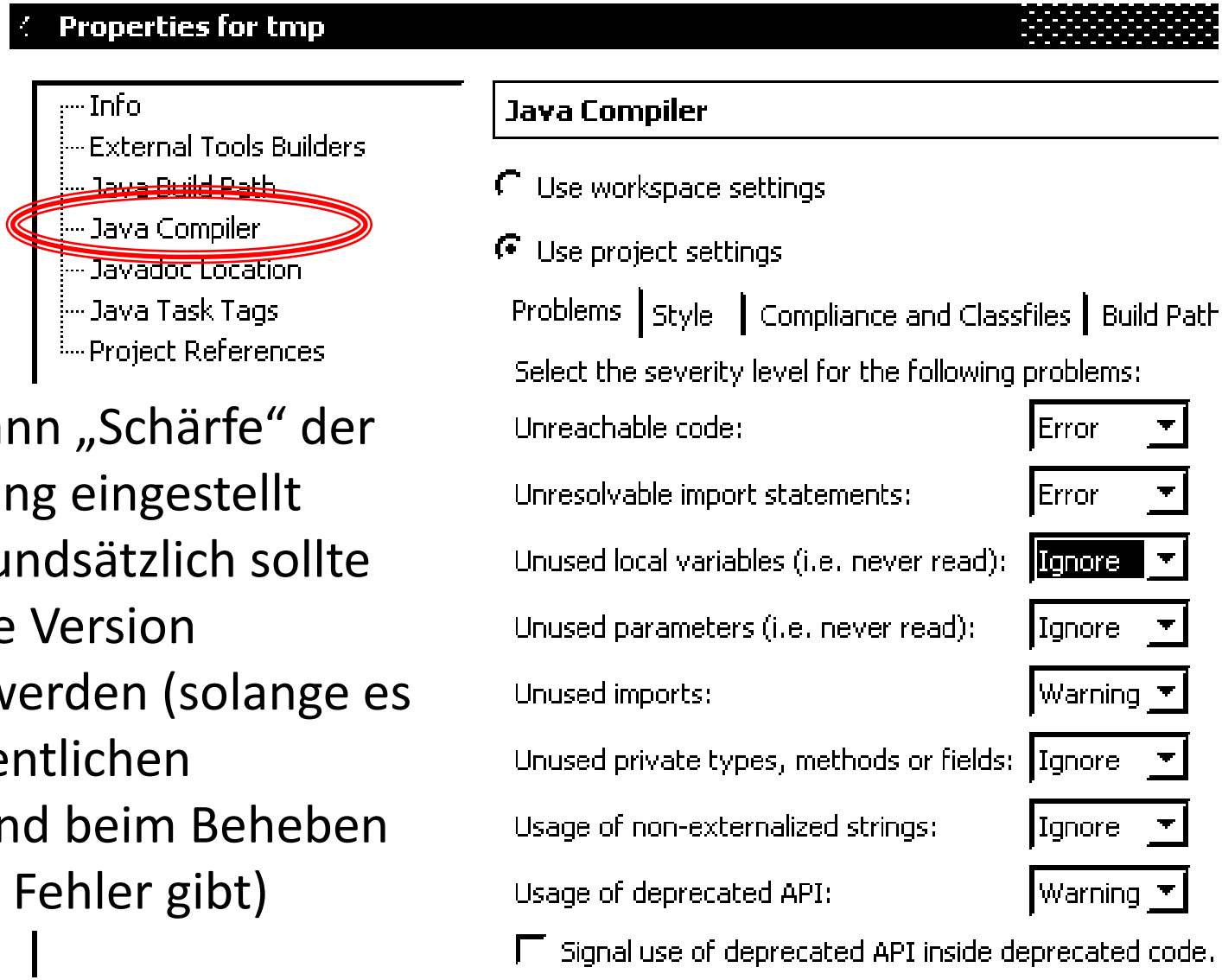
- Ausschnitt aus „Java Code Conventions“, Sun, 1997
- Inhalt soll sich nicht nur auf Formatierung beziehen

Einheitliche Werkzeugeinstellungen

- Vor Projekt einheitliche Formatierung festlegen
- Styleguide für verwendete Werkzeuge



Fehlerfindung bei der Syntaxanalyse



In Eclipse kann „Schärfe“ der Syntaxprüfung eingestellt werden. Grundsätzlich sollte die schärfste Version eingestellt werden (solange es keinen wesentlichen Mehraufwand beim Beheben potenzieller Fehler gibt)

Anti-Pattern verbieten

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Pattern dienen zur sinnvollen Strukturierung komplexer, aber gleichartiger Systeme
- Anti-Pattern sind wiederkehrende schlechte Lösungen, die man an Strukturen erkennen kann, z. B.
  - Spaghetti-Code, viele if, while und repeat-Schleifen gemischt, intensive Nutzung der Möglichkeiten mit break, früher: goto
  - Cut-and-Paste-Programmierung: „was oben funktionierte, funktioniert hier auch“
  - allmächtige Klassen, kennen jedes Objekt, sitzen als Spinne im Klassendiagramm, immer „gute“ Quelle für Erweiterungen
  - Rucksack-Programmierung: bei vergessenem Sonderfall in allen betroffenen Methoden

    if (Sonderfall){ Reaktion } else { altes Programm}
- Literatur (z. B.): W. J. Brown, R. C. Malveau, H. W. McCormick III, T. J. Mowbray, AntiPatterns, Wiley, 1998

weitere Maßnahmen

hierzu gehören einige Maßnahmen des proaktiven Risikomanagements

- Berücksichtigung von Standards
- richtiges Personal mit Erfahrungen und potenziellen Fähigkeiten finden (evtl. Coaching organisieren)
- frühzeitig Ausbildungen durchführen (niedriger Truckfaktor)
- frühzeitig passende Werkzeuge finden (Nutzungsregeln)
- Vorgehensmodell mit Reaktionsmöglichkeiten bei Problemen (generell: gelebtes flexibles Prozessmodell)
- Unabhängigkeit der Qualitätssicherung
- Erfahrungen im Anwendungsgebiet

3.7 Metriken

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Idee von Maßsystemen
- Beispiel McCabe-Zahl

Nutzung von Maßsystemen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- bisherigen Prüfverfahren sind aufwändig, besteht Wunsch, schneller zu Qualitätsaussagen zu kommen
- Ansatz: Nutzung von Maßsystemen, die Zahlenwerte generieren, deren Werte Indiz für Qualität der SW sind
- Werden Maße automatisch berechnet, kann man Qualitätsforderungen stellen, dass bestimmte Maßzahlen in bestimmten Bereichen liegen

- Ähnliche Ansätze in der Projektverfolgung und Analyse der Firmengesamtlage (-> Balanced Scorecard)

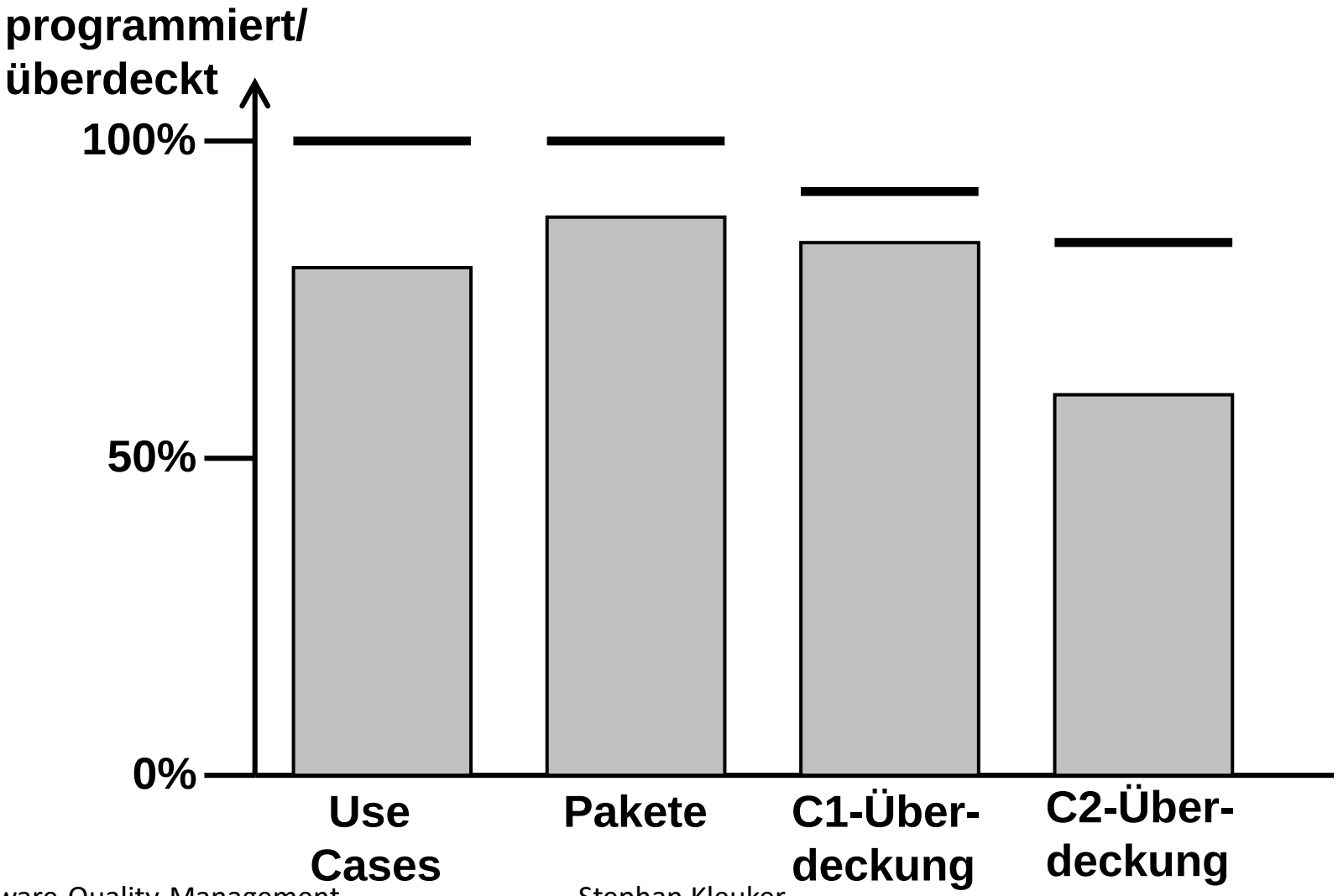
-> siehe Qualitätsmanagement

Chancen und Risiken von Maßsystemen

- Wichtig ist, dass man weiß, dass nur Indikatoren betrachtet werden, d.h. gewonnene Aussagen müssen nachgeprüft werden
- Kritisch wird es, wenn die Entwicklung an Maßen orientiert wird

- Beispiel: Maß für das Testniveau -> Überdeckungsmaße
- Fehler: mit wenig Tests eine hohe Abdeckung bekommen
- sinnvoll: typische Testfälle schreiben, dann Überdeckung messen; bei niedrigen Werten Hintergründe analysieren und ggfls. Testfälle ergänzen

Metriken zur Ermittlung des Projektstands



Zyklomatische Zahl nach McCabe

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

1. Man konstruiere die Kontrollflussgraphen
2. Man messe die strukturellen Komplexität

Die zyklomatische Zahl z(G) eines Kontrollflussgraphen G ist:

$z(G) = e - n + 2$ mit

e = Anzahl der Kanten des Kontrollflussgraphen

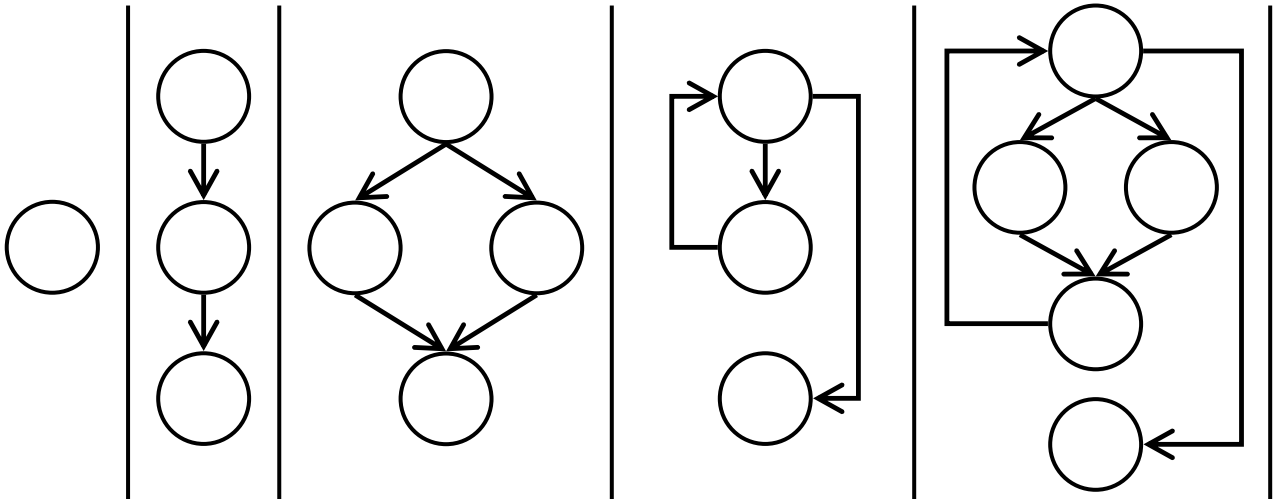
n = Anzahl der Knoten

Zyklomatische Komplexität gibt Obergrenze für die Testfallanzahl für den Zweigüberdeckungstest an

In der Literatur wird 10 oft als maximal vertretbarer Wert genommen (für OO-Programme geringer, z. B. 5)

für Java: #if + #do + #while + #switch-cases+ 1

Beispiele für die zyklomatische Zahl



Anzahl Kanten	**0**	**2**	**4**	**3**	**6**
Anzahl Knoten	**1**	**3**	**4**	**3**	**5**
McCabe Zahl	**1**	**1**	**2**	**2**	**3**

Erweiterte McCabe-Zahl

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Komplexität von Verzweigungen berücksichtigen
- gezählt werden alle Booleschen Bedingungen in if(<Bedingung>) und while(<Bedingung>): anzahlBedingung
- gezählt werden alle Vorkommen von atomaren Prädikaten: anzahlAtome

  z. B.: (a || x>3) && y<4 dann anzahlAtome=3
- erweitere McCabe-Zahl

  ez(G) = z(G) + anZahlAtome - anzahlBedingung

- wenn nur atomare Bedingungen, z. B. if(x>4), dann gilt ez(G) = z(G)

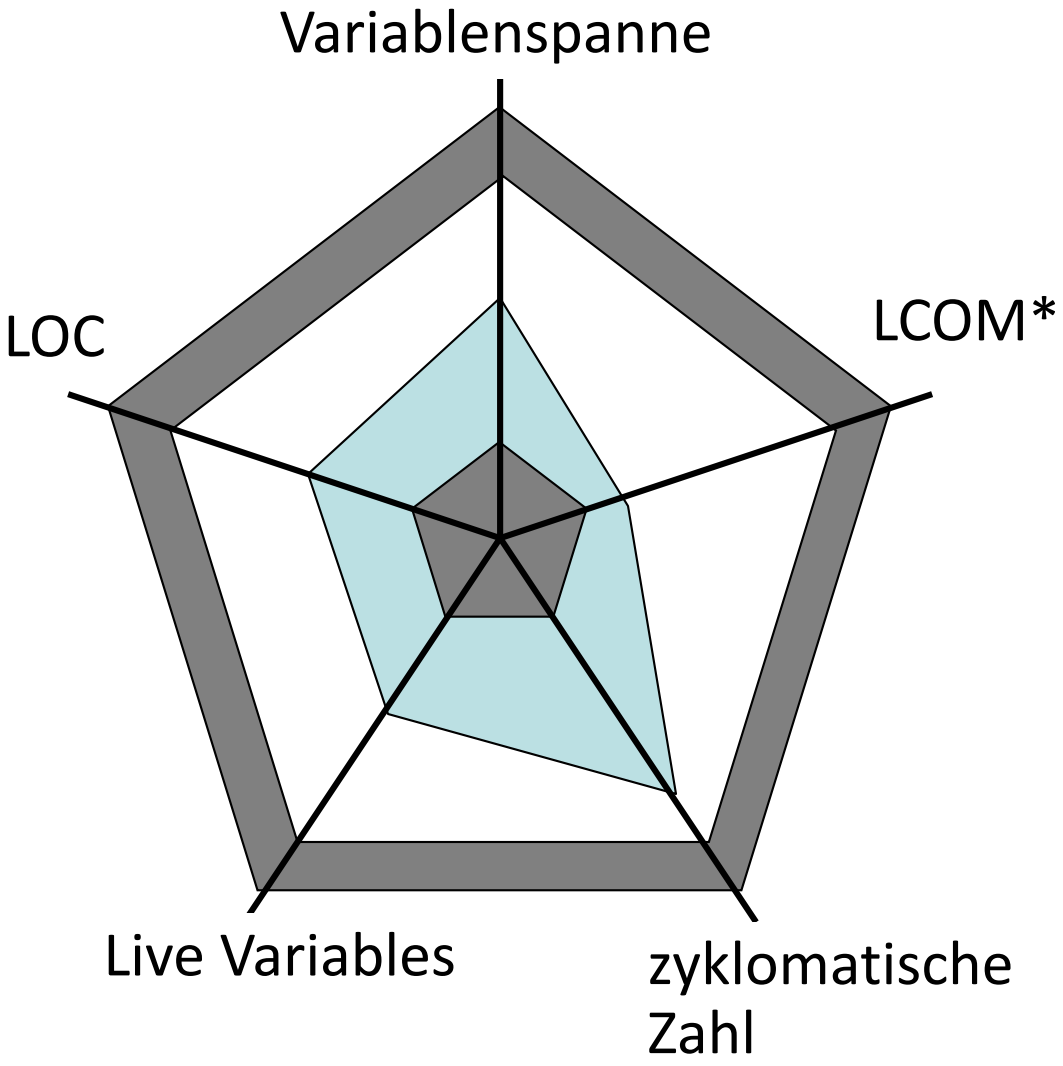
McCabe und Programmverständnis

- kurze Methoden mit selbsterklärenden Methodennamen fördern die Programmlesbarkeit wesentlich
- lokal zu optimieren: kurze Methoden (1), wenig Parameter (2), wenige Methoden (3)
- oftmals kann man Schleifen oder if-Blöcke in Methoden auslagern (Ansatz ist Teil der Refactoring-Idee)

- generelle Lösung zur Code-Verbesserung und bessere Metrik-Werte: Refactoring, d. h. Umbau von Programmen zur Erhöhung der Lesbarkeit, Wartbarkeit und Erweiterbarkeit

Beispiel eines Kivat-Diagramms

Maßzahlen können graphisch dargestellt werden, im Kivat-Diagramm steht jede Achse für eine Metrik, der weiße Bereich ist ok, die anderen Bereiche kritisch



3.8 Organisation des QS-Prozesses in IT-Projekten

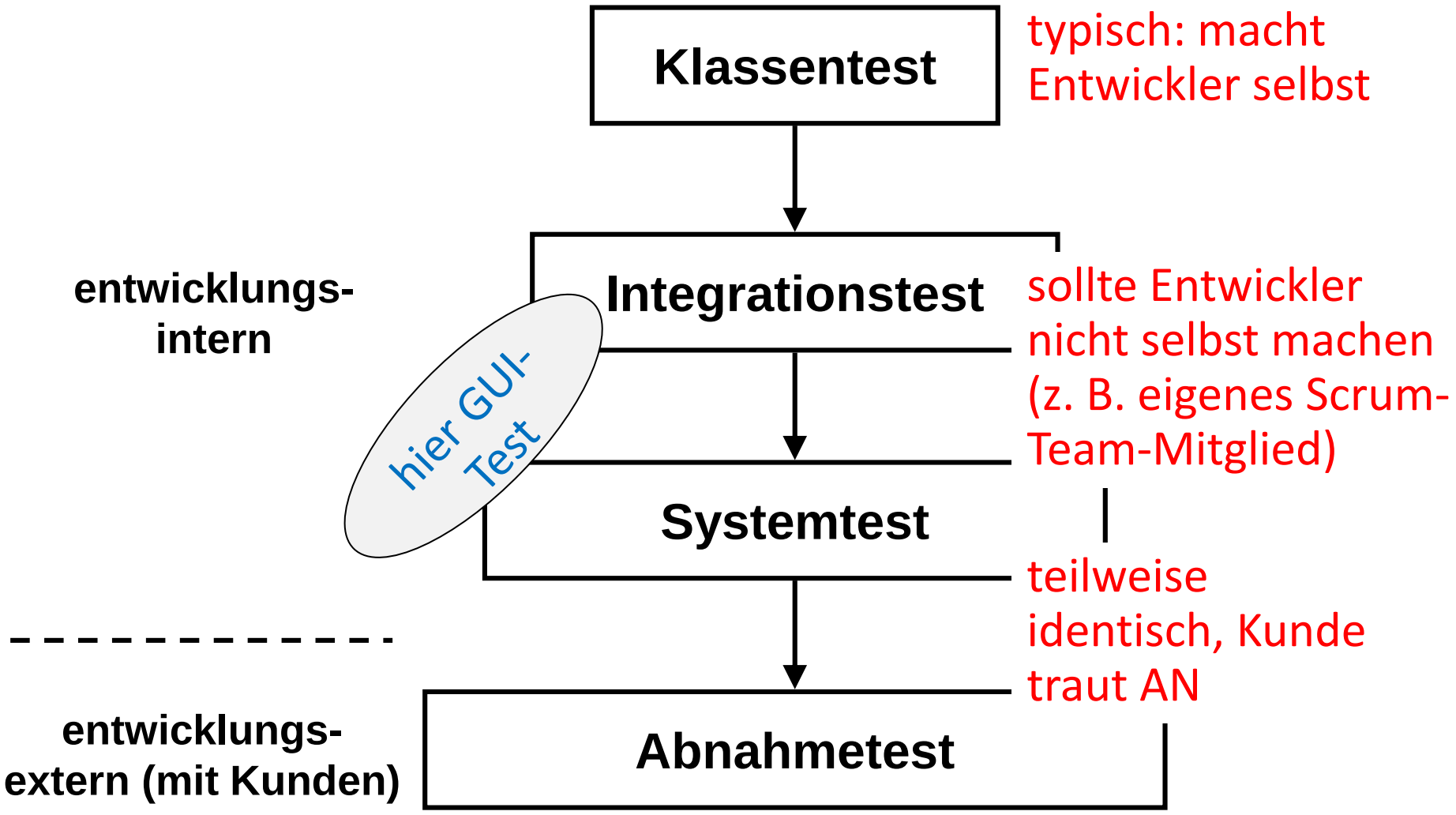
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Teststufen
- Regressionstest
- manuelle Prüfmethoden
- Testverfahren nach ANSI/IEEE-829
- Organisation der QS

siehe auch:

- H. M. Sneed, M. Winter, Testen objektorientierter Software, Hanser, München Wien
- A. Spillner, T. Roßner, T. Linz, Praxiswissen Softwaretest, ab 2. Auflage, dpunkt Verlag, Heidelberg

Erinnerung: Teststufen (grober Ablauf)



Klassentest (Modultest)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Varianten:
  - *Unit-Test*: einzelne Methoden und/oder Klassen
  - *Modultest*: logisch-zusammengehörige Klassen, z.B. ein Package in Java
- Testziel: Prüfung gegen *Feinspezifikation*
  - Architektur, Design, Programmierkonstrukte
- Testmethode: *White-Box-Test*
- *Alle Module* müssen getestet werden
  - eventuell mit unterschiedlicher Intensität

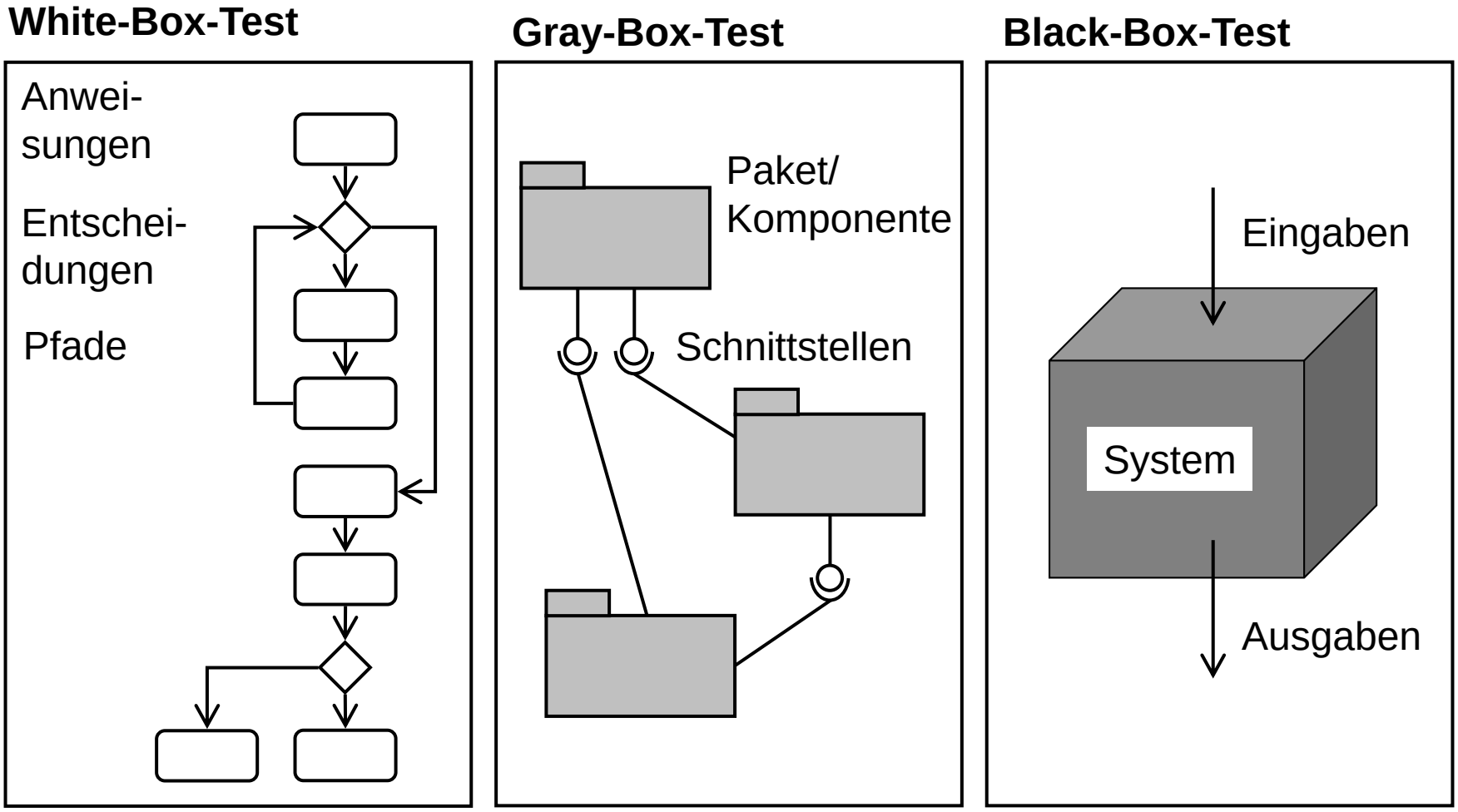
Integrationstest

- Module werden zu einem System integriert und getestet
- Testziele:
  - Werden Schnittstellen richtig benutzt?
  - Werden Klassen bzw. ihre Methoden richtig aufgerufen?
- Konzentration auf *(Export-) Schnittstellen*
  - Interne Schnittstellen können nicht mehr direkt beeinflusst werden
  - Geringere Testtiefe als beim Modultest
  - Grey-Box-Test (oder auch Black-Box)
- Techniken ähnlich wie bei Modultest
  - Pfadanalyse über die komplette Interaktion der Module oft nicht mehr sinnvoll
- Mit *minimaler Systemkonfiguration* beginnen, Integrationsstrategie spielt eine Rolle

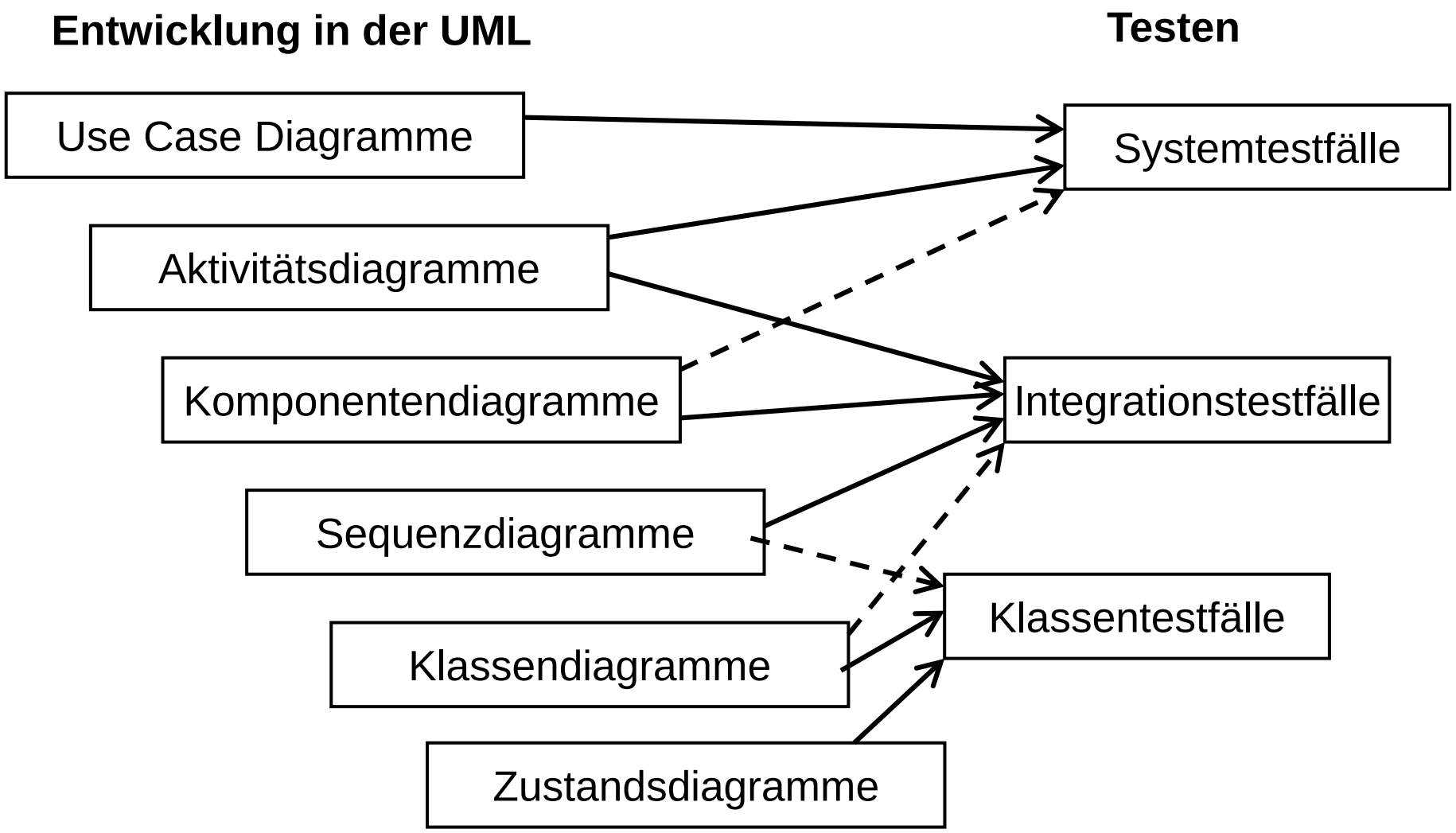
Systemtest

- Orientierung an den spezifizierten Systemaufgaben (z.B. Use Cases)
- Interaktion mit den (simulierten) Nachbarsystemen
- (endgültige) Validierung der *nicht-funktionalen* Anforderungen, z. B. Skalierbarkeit, Verfügbarkeit, Robustheit, ...
- möglichst interne Vorwegnahme des Abnahmetests

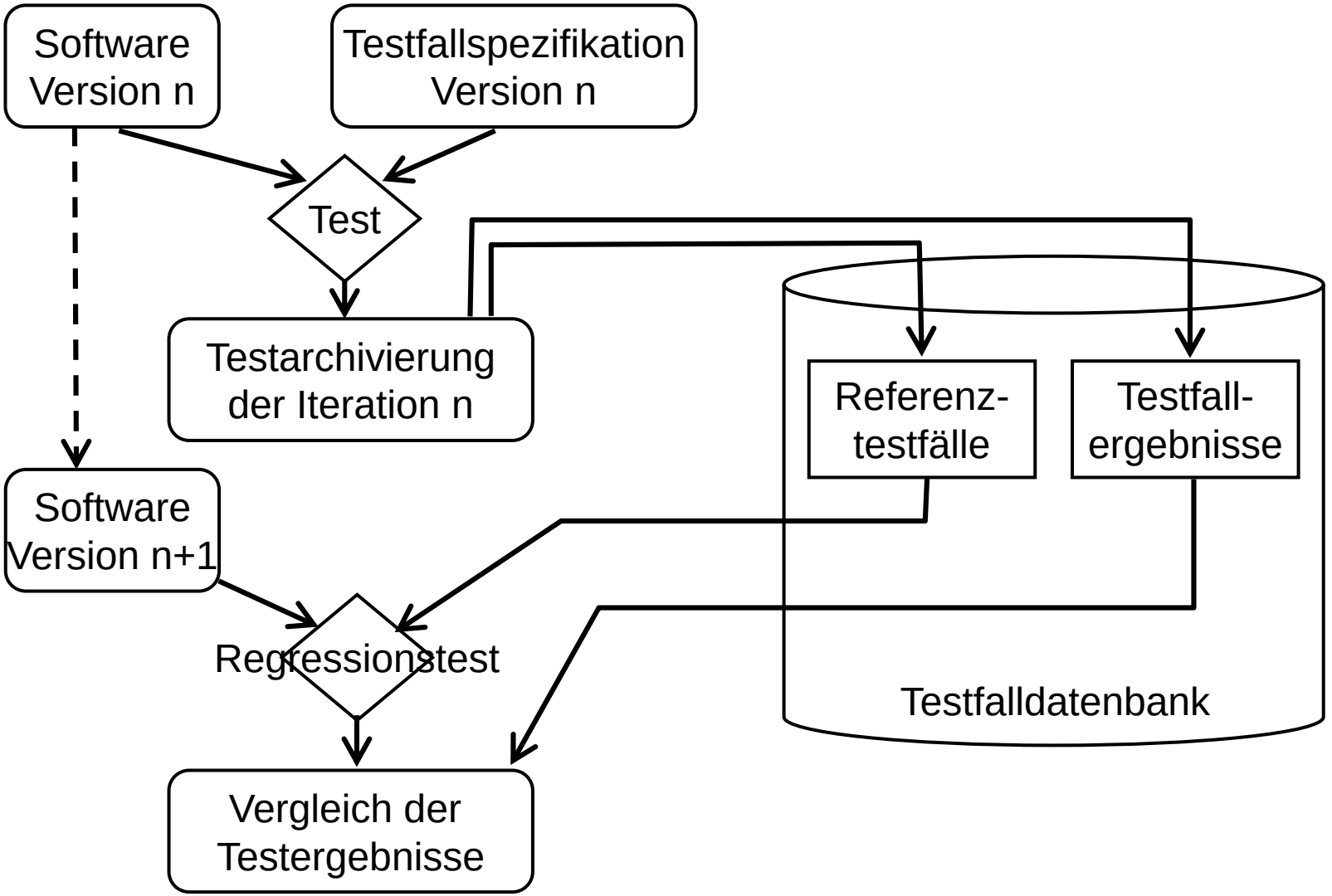
Testansätze zusammengefasst



Testfälle und die UML



Prinzip des Regressionstests



Wartung und Testen

- Der Test ist geteilt in Änderungstest (White-Box) und Regressionstest (Black-Box)

- Änderungstest vom Entwickler, er schreibt die Testfälle fort.
- Regressionstest von unabhängiger Testgruppe mit den alten plus neuen Testfällen durchgeführt
- Testgruppe ist für Pflege und Fortschreibung der Systemtestfälle verantwortlich

Lasttest

- Geforderte Performance
  - Durchsatz bzw. Transaktionsrate
  - Antwortzeiten
- Skalierbarkeit
  - Anzahl Endbenutzer
  - Datenvolumen
  - Geografische Verteilung
- Zugriffskonflikte konkurrierender Benutzer
- Entspricht dem Zeitraum nach der Inbetriebnahme
- Simulation von
  - Anzahl Endbenutzer,
  - Transaktionsrate , ...
  - Über einen signifikanten Zeitraum (mehrere Stunden)

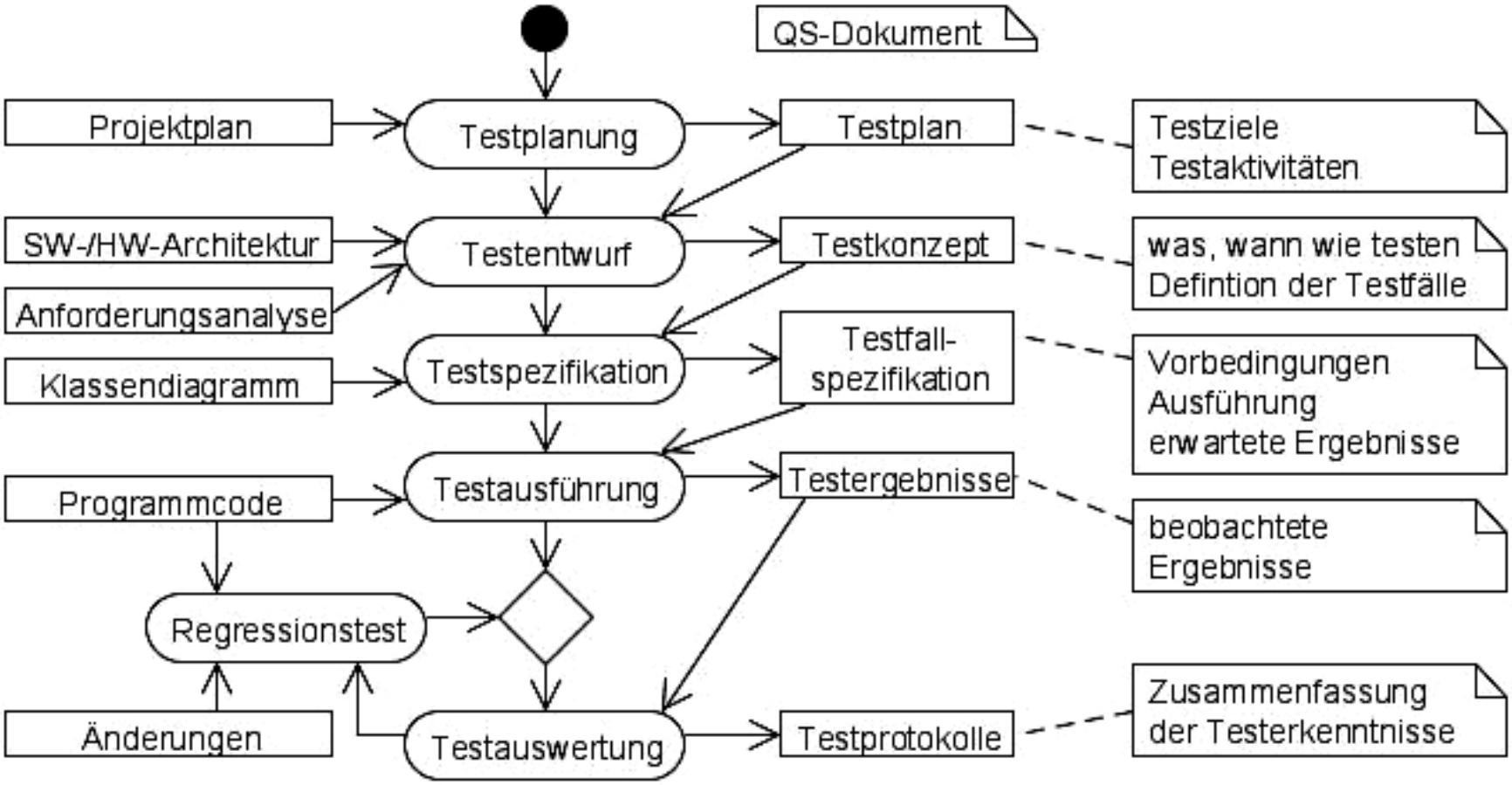
Manuelle Prüfmethoden berücksichtigen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Produkte und Teilprodukte werden manuell analysiert, geprüft und begutachtet
- Ziel ist es, Fehler, Defekte, Inkonsistenzen und Unvollständigkeiten zu finden
- Die Überprüfung erfolgt in einer Gruppensitzung durch ein kleines Team mit definierten Rollen
- Jedes Mitglied des Prüfteams muss in der Prüfmethode geschult sein
- notwendigen Aufwand und benötigte Zeit einplanen
- Vorgesetzte und Zuhörer sollen an den Prüfungen *nicht* teilnehmen
- Inspektionen, Reviews und Walkthroughs (in abnehmender Formalität), in der Literatur ist „Reviews“ teilweise der Oberbergriff

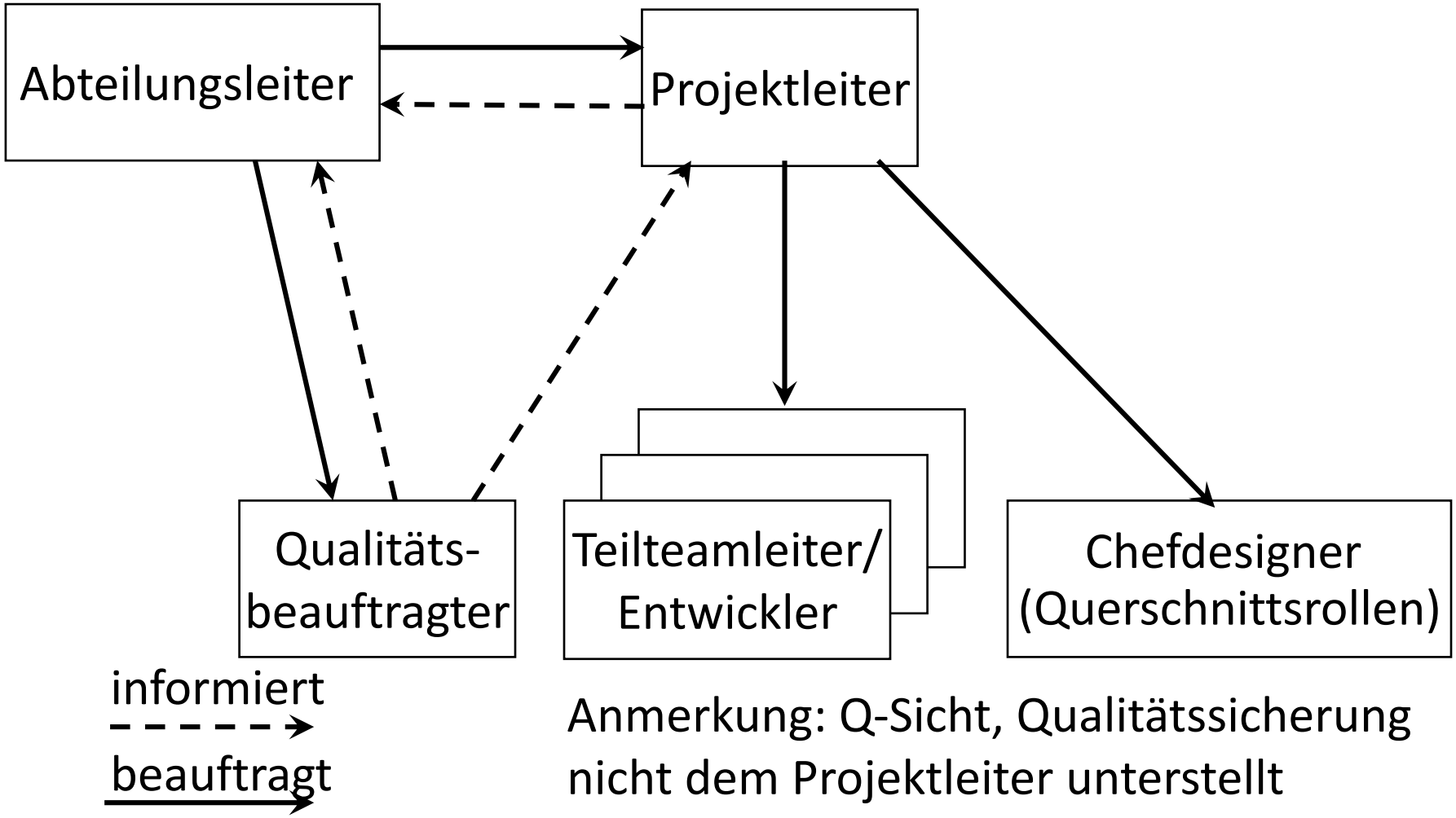
Iterativ inkrementelle Werkzeugauswahl

Dokumentation der Qualitätssicherung (Tests)



Organisation und Rollen (1/3)

Video



Organisation und Rollen (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Abteilungsleiter
  - hat Gesamtverantwortung für das Projekt
  - stellt Ressourcen zur Verfügung (Mitarbeiter, Budget, ...)
  - kontrolliert Budget
  - hält Kontakt zu den Entscheidungsträgern beim Kunden

- Projektleiter
  - hat Ergebnisverantwortung für Projekt einschl. Qualität
  - leitet die Teilteams und damit die Projektmitarbeiter
  - verantwortlich für inhaltlichen Kontakt zum Kunden
  - berichtet an den Abteilungsleiter

Organisation und Rollen (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Qualitätsbeauftragter
  - wird vom Abteilungsleiter ernannt
  - hat die Verantwortung für die QS und Durchführung von QS-Maßnahmen
  - berichtet an den Abteilungsleiter und den Projektleiter über den Stand der QS
  - QS-Maßnahmen sind sehr wichtige Aufgaben im Projekt
    - haben konkrete Ergebnisse
    - sind Grundlage für die Beurteilung des Projektfortschritts
  - Drückt den “roten Knopf“, um eine Auslieferung zu stoppen

- Frage: Was spricht für, was gegeben eigene Q-Abteilung?

3.9 Testautomatisierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

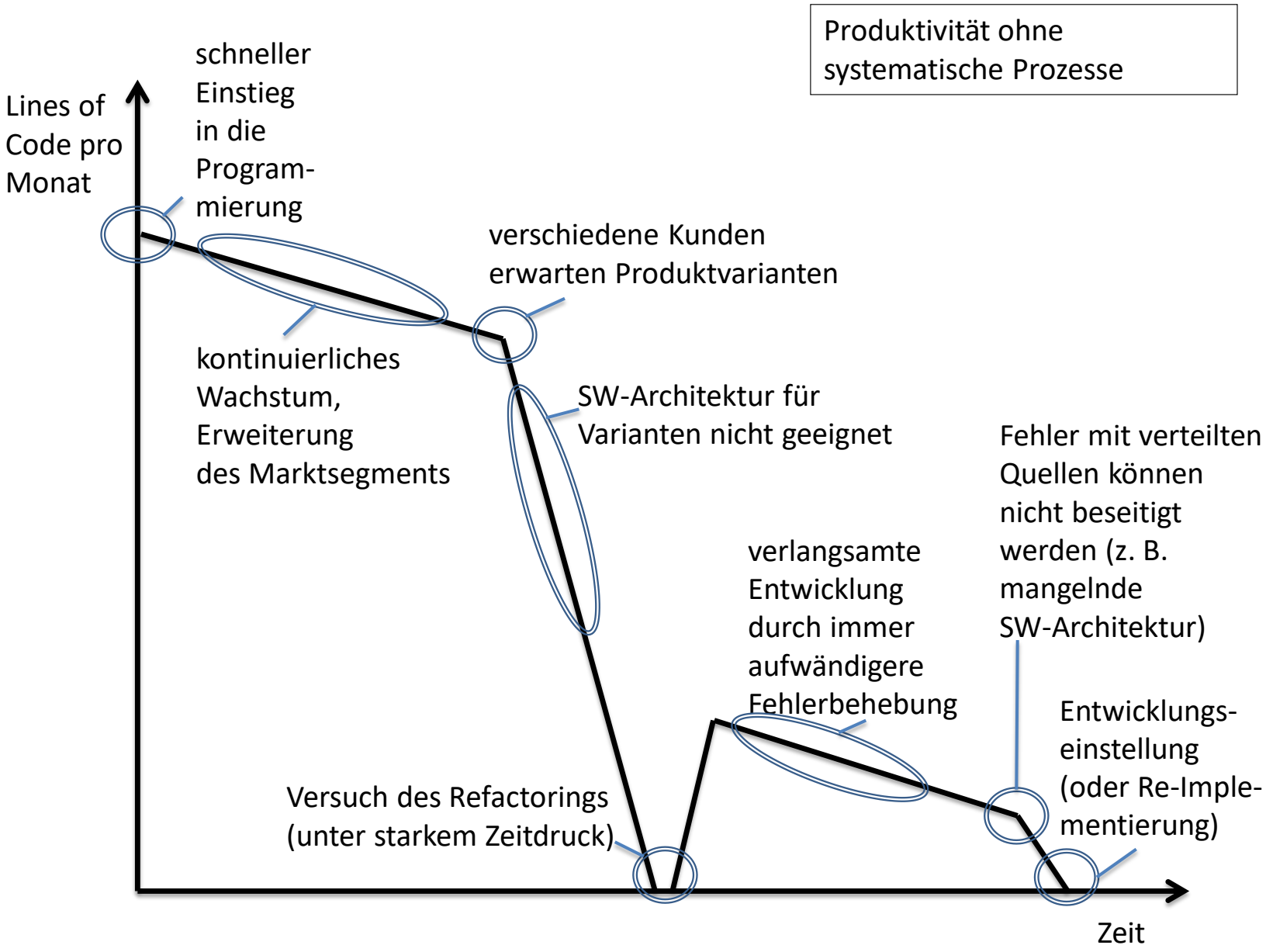
- Automatisierungsidee
- Beispiele für Werkzeuge
- Build-Server
- Idee von Build-Werkzeugen

Was meint hier Automatisierung

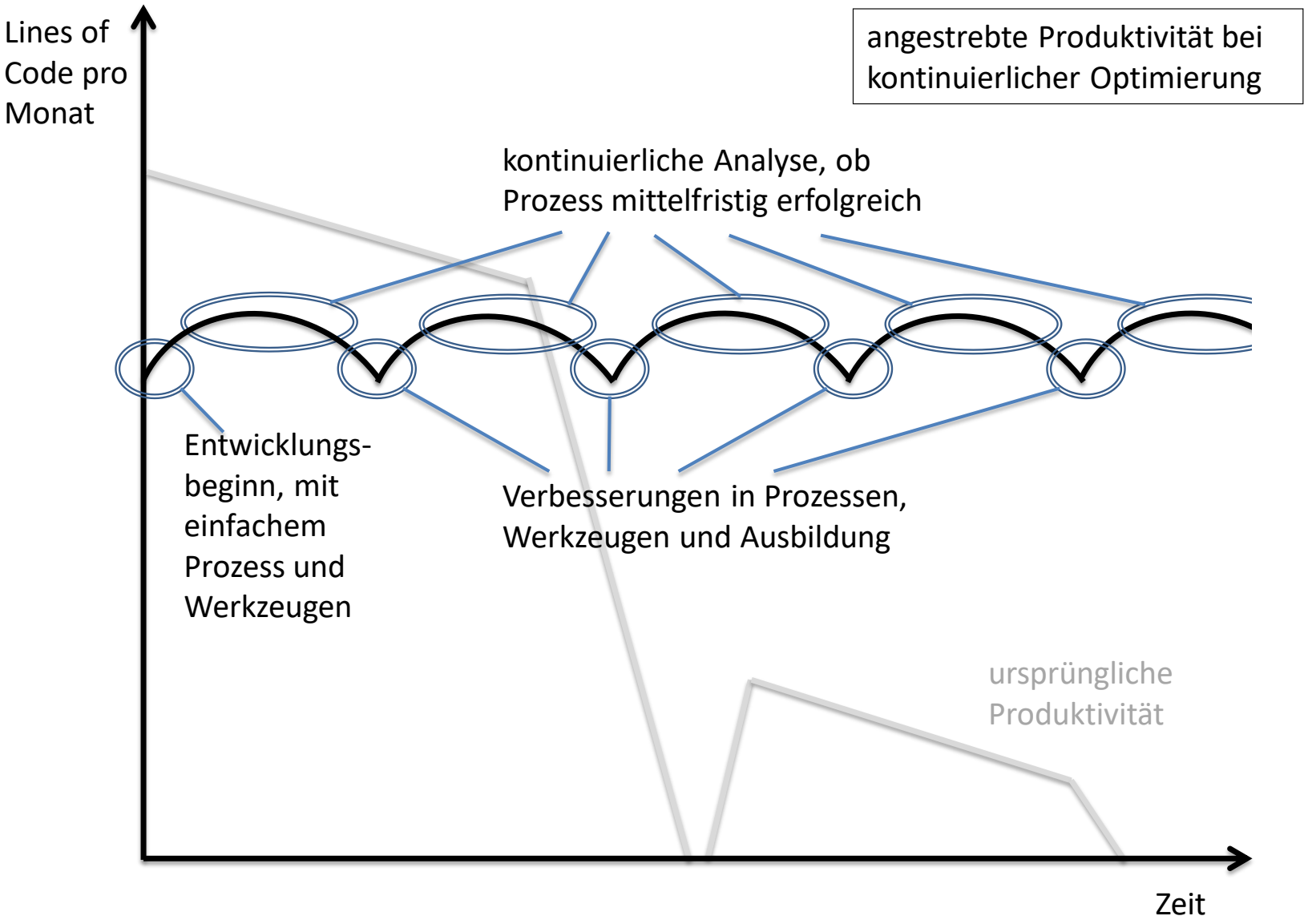
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- klassische Testansätze
  - Entwicklung einer Testspezifikation (Vorbedingung, Ausführung, erwartete Ergebnisse)
  - manuelle Testausführung
  - manuelle Erfassung und Auswertung der Testergebnisse
- erste Automatisierungsstufe
  - Werkzeuge wie JUnit, Marathon erlauben die automatische Testausführung und Protokollierung (teilweise Auswertung)
  - Werkzeuge müssen einzeln gestartet werden
- zweite Automatisierungsstufe
  - mehrere Werkzeuge laufen nacheinander / zusammen
  - Ergebnisse werden zentral protokolliert

Warum Automatisierung? Gefahr



Warum Automatisierung? Optimierter Prozess

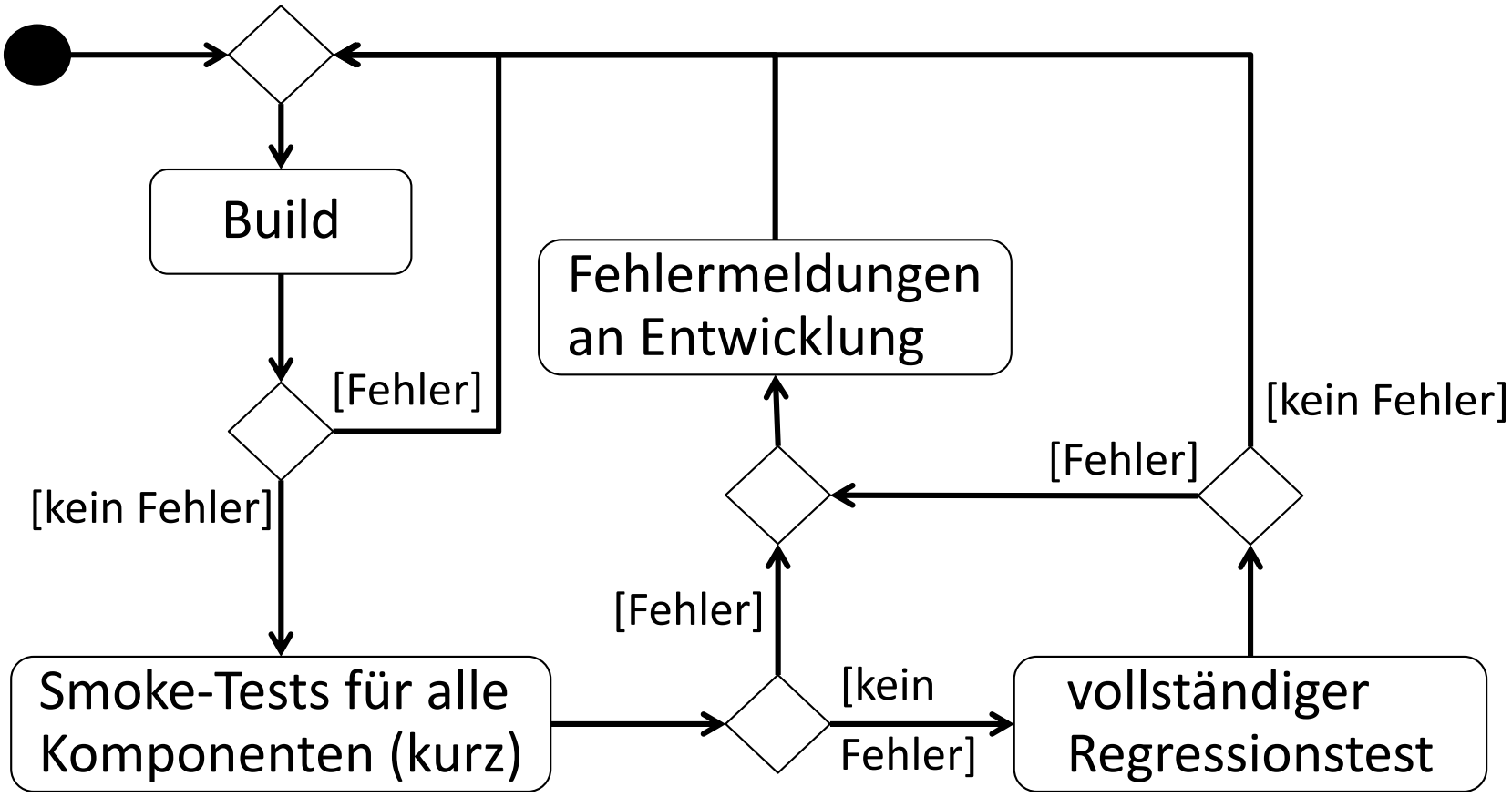


Beispiel: Ziele der QS - Optimierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Konzeption und Implementierung der Automatisierung:
  - Unit-Tests
  - GUI-Tests
  - Messung der Codeüberdeckung
  - Statische Codeanalyse
  - Softwaremetrik
- Integrierbarkeit in das bisherige Verfahren

Beispiel: Gesamtablauf



3.10 Testwerkzeuge basierend auf Byte-Code

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Byte-Code
- ClassLoader
- Konzept von Werkzeugen basierend auf Byte-Code
- Beispiel-Bibliotheken

Byte-Code

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Java (und C#) wandeln Programmcode in Byte-Code als Zwischensprache
- Byte-Code wird von betriebssystemabhängigen virtuellen Maschinen ausgeführt
- Vorteil: Byte-Code kann systemunabhängig sein
- Vorteil: Virtuelle Maschine kann für verschiedene Sprachen eingesetzt werden (Java, Groovy, Kotlin, JRuby, …), die so auch kombinierbar werden
- (kein) Nachteil: klassisch wird Byte-Code bei erster Nutzung übersetzt, so dass Systemstart verlangsamt ist (sonst keine Laufzeitunterschiede); mittlerweile Startvarianten möglich

- https://docs.oracle.com/javase/specs/index.html

eine „normale“ Java-Klasse

```
package sqmClassLoader;
public class Bsp {

  static {
    System.out.println("Das ist ja");
  }

  {
    System.out.println("toll");
  }

  public static void main(String[] args) {
    new Bsp();
    new Bsp();
  }
}
```

```
Das ist ja
toll
toll
```

Aufruf mit –verbose:class (Ausschnitt)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
[0.005s][info][class,load] opened: C:\kleukersSEU\java\lib\modules
[0.011s][info][class,load] java.lang.Object source: shared objects file
[0.011s][info][class,load] java.io.Serializable source: shared objects file
...
[0.035s][info][class,load] sun.security.util.Debug source: shared objects file
[0.035s][info][class,load] sqmClassLoader.Bsp source:
file:/F:/workspaces/eclipseWS/sqmClassLoader/bin/
[0.035s][info][class,load] java.lang.PublicMethods$Key source: shared objects file
[0.035s][info][class,load] java.lang.Void source: shared objects file
[0.035s][info][class,load] java.nio.charset.CoderResult source: shared objects file
Das ist ja
toll
toll
[0.035s][info][class,load] jdk.internal.misc.TerminatingThreadLocal$1 source:
shared objects file
[0.035s][info][class,load] java.lang.Shutdown source: shared objects file
[0.035s][info][class,load] java.lang.Shutdown$Lock source: shared objects file
```

ClassLoader

- verantwortlich zum Laden der Klassen
- ist Interface, Standardimplementierung SystemClassLoader
- Java-Basis-Klassen durch „bootstrap class loader“ geladen
- kann mehrere ClassLoader geben; jeder mit eigenem Verwaltungsraum (möglich, dass Klasse mit identischem Namen in mehreren Räumen vorliegt) -> nutzen u. a. Application Server, um mehrere Webanwendungen im selben Container zu deployen
- ClassLoader per Delegation verknüpft, Fragt bei Parent nach (Hierarchie)
- kann Klassen laden, die nicht auf dem Classpath liegen

ClassLoader – einfaches Beispiel

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public static void main(String[] args) {
  ClassLoader cl1 = MainClassLoader.class.getClassLoader();
  List<Integer> al = List.of(1, 2, 3, 4);
  // bootstrap class loader für java. …
  ClassLoader cl2 = al.getClass().getClassLoader();
  System.out.println((cl1 == cl2) + " " + cl1 + " ** "+ cl2);
  cl2 = ClassLoader.getSystemClassLoader();
  System.out.println((cl1 == cl2));
  //ClassLoader der Anwendungsklassen
  cl2 = Thread.currentThread().getContextClassLoader();
  System.out.println((cl1 == cl2));
  while (cl1 != null) {
    System.out.println (cl1.getClass().getName());
    cl1 = cl1.getParent();
  }
}
```

```
false jdk.internal.loader.ClassLoaders$AppClassLoader@73d16e93 ** null
true
true
jdk.internal.loader.ClassLoaders$AppClassLoader
jdk.internal.loader.ClassLoaders$PlatformClassLoader
```

Byte-Code-Bearbeitung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- Byte-Code kann gelesen, analysiert und verändert werden
- dadurch Klassen beim Laden veränderbar
- viele Werkzeuge, z. B. Mock-Frameworks und JPA als OR-Mapper basieren darauf
- mittlerweile auch Byte-Code während der Laufzeit modifizierbar

- her nicht behandelt: Java-Sicherheit; gibt viele Varianten schon Reflexion, aber auch Bye-Code-Bearbeitung zu verhindern, u. a.
  - Regelsystem für JVM-Ausführung, SecurityClassLoader, final
  - sealed Jars (code self contained, nur public-Zugriff)
  - obfuscating Code
- Byte-Code-Analysewerkzeug: https://github.com/Konloch/bytecode-viewer

häufiger genutzte Beispielklasse

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Konto {
    private int stand;

    public void einzahlen(int betrag) {
        this.stand += betrag;
    }

    public int getStand() {
        return this.stand;
    }

    public String toString() {
        return "Konto{" + "stand=" + this.stand + '}';
    }
}
```

Erinnerung Decorator (1/3)

- ergänze Interface

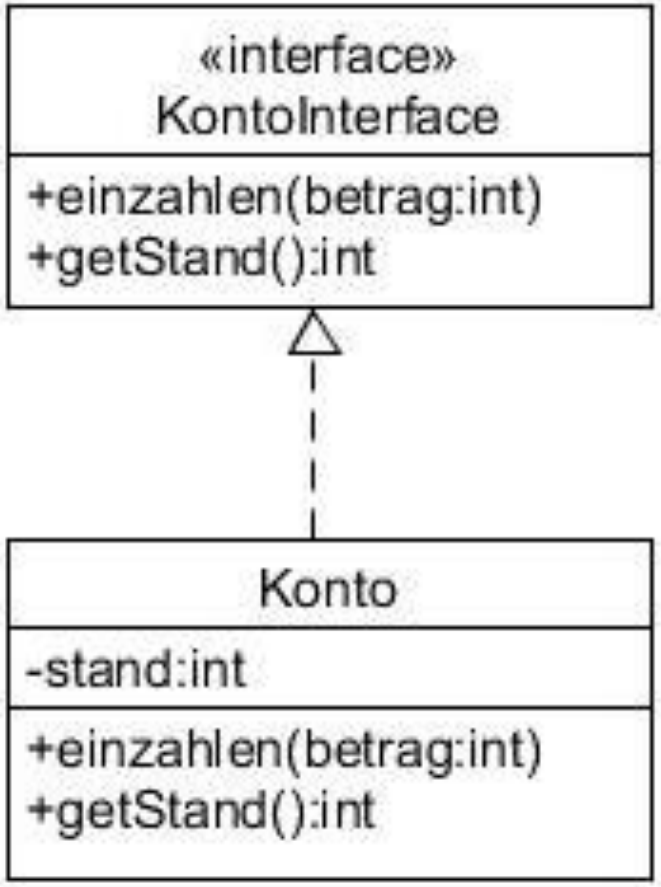
```
public interface KontoInterface {

    void einzahlen(int betrag);

    int getStand();

}


public class Konto
    implements KontoInterface { …
```

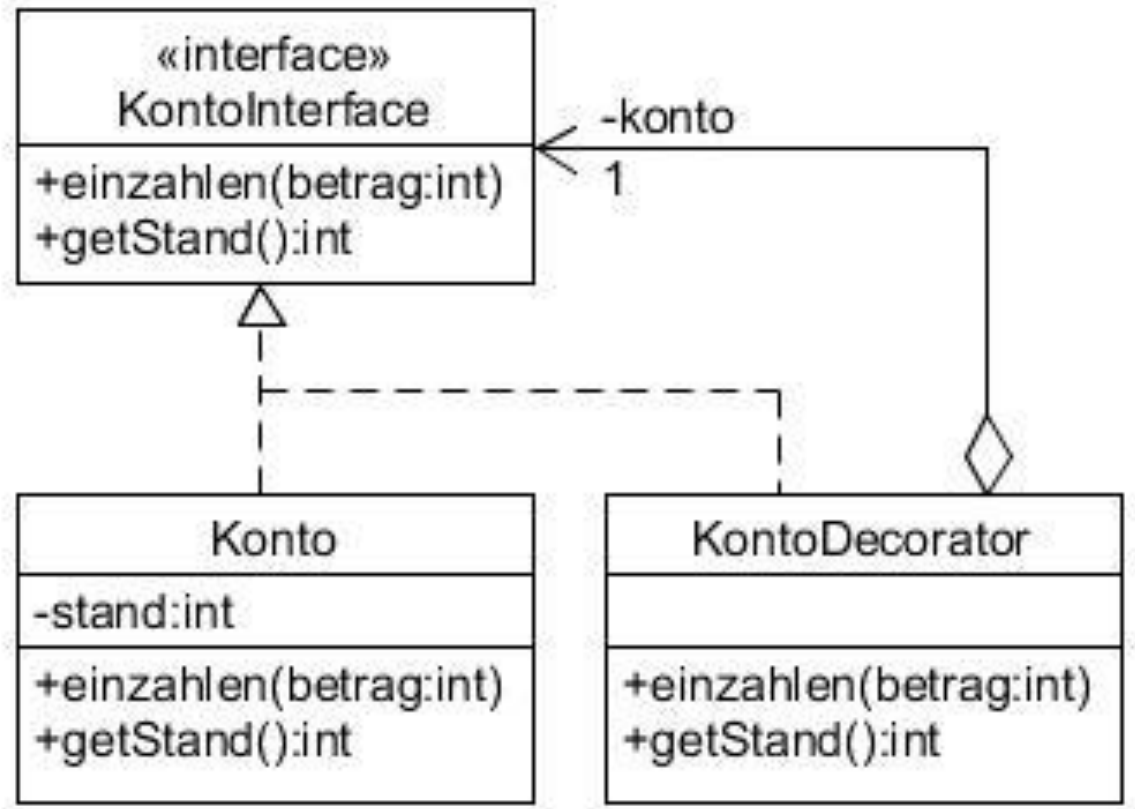


- Was ist aber, wenn Klasse Konto nicht verändert werden soll (hier Fallstudie: jedes Einzahlen soll 4 kosten, Kosten sollen gemerkt werden)

Erinnerung Decorator (2/3)

- ergänze neue Klasse (Decorator) die das Interface realisiert und ein Objekt der Klasse als Exemplarvariable hält

- Idee: delegiere Aufrufe an diese Exemplarvariable und ergänze drum herum neue Funktionalität
- flexibler: Exemplarvariable nutzt Interface-Typ



Erinnerung Decorator (3/3)

```
public class KontoDecorator implements KontoInterface {

    private KontoInterface konto;

    public KontoDecorator(KontoInterface konto){
        this.konto = konto;
    }

    @Override
    public void einzahlen(int betrag) {
        System.out.println("vor einzahlen");
        this.konto.einzahlen(betrag);
        System.out.println("nach einzahlen");
    }

    @Override
    public int getStand() {
        System.out.println("vor getStand");
        int ergebnis = this.konto.getStand();
        System.out.println("nach getStand");
        return ergebnis;
    }
}
```

cglib

- Code Generation Library, erlaubt die Veränderung von Byte-Code und Erstellung neuer Klassen auf Basis existierenden Byte Codes
- wurde/wird in Hibernate und Spring genutzt
- existierende (also kompilierte) Klassen werden bearbeitet
- u. a. Enhancer: Nutze existierende Klasse X, ermögliche es Aufrufe von Methoden von X zu verändern; resultierendes Objekt hat Typ X (anonyme von X abgeleitete Klasse)

- https://github.com/cglib/cglib
- https://dzone.com/articles/cglib-missing-manual

cglib: Einführendes Beispiel (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public static void main(String[] arg) {
  Konto k = new Konto();
  Enhancer enhancer = new Enhancer();
  enhancer.setSuperclass(Konto.class);
  enhancer.setCallback((MethodInterceptor)
      (obj, method, args, proxy) -> {
        System.out.println(// obj + // Endlosrekursion
            "\n method: " + method
                + "\n args: " + Arrays.asList(args)
                + "\n proxy: " + proxy + "\n");
        if (method.getDeclaringClass() != Object.class
            && method.getReturnType() == int.class) {
          return 42;
        } else  {
          return proxy.invokeSuper(obj, args);
        }
      });
  Konto k2 = (Konto) enhancer.create();
```

cglib: Einführendes Beispiel (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
System.out.println(k + " -- " + k2);
System.out.println(k.getStand() + " -- " + k2.getStand());
System.out.println(k.getClass() + " -- " + k2.getClass());
```

```
 method: public java.lang.String entity.Konto.toString()
 args: []
 proxy: net.sf.cglib.proxy.MethodProxy@6ad5c04e

Konto{stand=0} -- Konto{stand=0}

 method: public int entity.Konto.getStand()
 args: []
 proxy: net.sf.cglib.proxy.MethodProxy@2c8d66b2

0 -- 42
class entity.Konto -- class entity.Konto$$EnhancerByCGLIB$$b2cd91c6
```

typische Fehlermeldung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- generell ist cglib auf Java-Modulsystem umgestellt, trotzdem:
- Caused by: java.lang.reflect.InaccessibleObjectException: Unable to make protected final java.lang.Class java.lang.ClassLoader.defineClass(java.lang.String,byte[],int,int,java.security.ProtectionDomain) throws java.lang.ClassFormatError accessible: module java.base does not "opens java.lang" to module net.sf.cglib
- Lösung mit Hilfe des „Aufweichens“ des Modulsystems, mit VM-Argument:

```
--add-opens java.base/java.lang=net.sf.cglib
```

cglib – zweite Nutzungsmöglichkeit (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
Class<?>[] ca = new Class[0];
CallbackHelper callbackHelper
        = new CallbackHelper(Konto.class, ca) {
  @Override
  protected Object getCallback(Method method) {
    if (method.getDeclaringClass() != Object.class
        && method.getReturnType() == int.class) {
      return new FixedValue() {
        @Override
        public Object loadObject() throws Exception {
          return 41;
        }
      };
    } else {
      return NoOp.INSTANCE;
    }
  }
};
```

cglib – zweite Nutzungsmöglichkeit (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
enhancer.setCallbackFilter(callbackHelper);
enhancer.setCallbacks(callbackHelper.getCallbacks());
Konto k3 = (Konto) enhancer.create();
System.out.println("\n----------\n");
System.out.println("k3: " + k3);
System.out.println("getStand: " + k3.getStand());
System.out.println("getClass " + k3.getClass());
System.out.println("instanceof: " + (k3 instanceof Konto));
```

```
k3: Konto{stand=0}
getStand: 41
getClass class entity.Konto$$EnhancerByCGLIB$$7b5f140d
instanceof: true
```

Umsetzung Decorator

- Erinnerung Konto soll sich zusätzlich Gebühren merken
- cglib nicht direkt dafür vorgesehen Objektvariablen zu ergänzen
- da existierende Objekte nicht erweitert werden können, wird neuer Konstruktor für Konto (Klassenmethode) geschrieben
- in dieser Klasse werden Gebühren aller erzeugten Konten mitverwaltet (Map)

- Spielerei: Gebühren sollen bei getStand() zum Ergebnis dazugerechnet werden; zeigt Veränderungsmöglichkeiten von Ergebnisaufrufen

cglib – Decorator (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class KontoDecorator {
  private static Map<Konto, Integer> gebuehren = new HashMap<>();
  private static Enhancer enhancer;

  public static Konto decorate() {
    if (enhancer == null) {
      enhancer = new Enhancer();
      enhancer.setSuperclass(Konto.class);
      enhancer.setCallback(
          (MethodInterceptor) (obj, method, args, proxy) -> {
            if (obj instanceof Konto
                && method.getName().equals("einzahlen")) {
              gebuehren.put((Konto) obj, gebuehren.get(obj) + 4);
              args[0] = Integer.parseInt(args[0].toString()) - 4;
              return proxy.invokeSuper(obj, args);
            }
```

cglib – Decorator (2/3)

```
                if (obj instanceof Konto
                  && method.getName().equals("getStand")) {
                Object o = proxy.invokeSuper(obj, args);
                var stand = Integer.parseInt(o.toString());
                // Gebuehren verschleiern
                return stand + gebuehren.get(obj);
              }
              // evtl TODO toString() ersetzen
              return proxy.invokeSuper(obj, args);
            });
    }
    Konto k = (Konto) enhancer.create();
    gebuehren.put(k, 0);
    return k;
  }
}
```

cglib – Decorator (3/3)

```
public static void main(String[] args) {
  Konto k1 = KontoDecorator.decorate();
  Konto k2 = KontoDecorator.decorate();
  System.out.println("k1 == k2: " + (k1 == k2));
  k1.einzahlen(42);
  k1.einzahlen(4200);
  System.out.println("k1: " + k1
      +"\nstand: " + k1.getStand()
      +"\nk2: " + k2);
}
```

```
k1 == k2: false
k1: Konto{stand=4234}
stand: 4242
k2: Konto{stand=0}
```

cglib - Anmerkungen

- bietet im Wesentlichen die gezeigten Ansätze (noch Immutable Beans und „Kleinkram“)
- lange Zeit in Mock-Frameworks und OR-Mappern genutzt; reichte dann aber für neue Use Cases nicht mehr aus
- seit 2012 nicht intensiv weiterentwickelt
- trotzdem sinnvoll nutzbar (vorher eigene Use Cases durchgehen; gilt für alle vorgestellten Ansätze)
- ob wirklich nutzbar stellt sich oft erst mit Prototypen heraus (gilt für alle vorgestellten Ansätze)

Java Agents

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- cglib erzeugt neue Klassen
- um existierende Klassen zu bearbeiten, muss deren Byte-Code verändert werden
- dies ist mit Java Agenten (seit Java 1.5) möglich
- Java-Agent benötigt mindestens folgende Methode

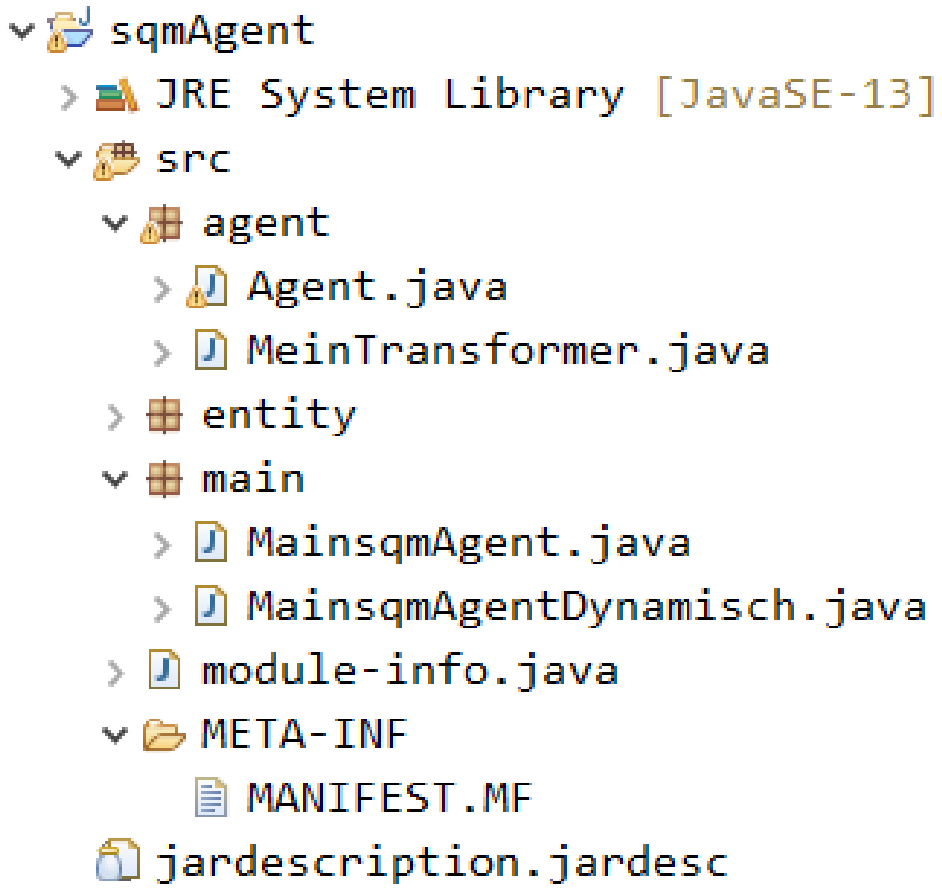
  `public static void premain(String arg, Instrumentation ins)`
- Agent muss in Jar-Datei gepackt werden
- Jar-Datei muss Manifest META-INF/MANIFEST.MF enthalten in der Agent-Klasse genannt wird
- Java-Agents können zum Programmstart übergeben werden

  `-javaagent:<Pfad zum jar>\<Jardatei>=optionaleParameter`
- alternativ direkt im laufenden Programm ladbar, Programmstart mit **`-Djdk.attach.allowAttachSelf=true`**

Beispielprojekt (1/7)

- typisch ist eigenes Projekt mit Agent, der dann woanders genutzt wird
- geht auch als ein Projekt (zum Experimentieren)
- MANIFEST.MF

```
Manifest-Version: 1.0
Premain-Class: agent.Agent
Agent-Class: agent.Agent
Permissions: all-permissions
Can-Redefine-Classes: true
Can-Retransform-Classes: true
```



Beispielprojekt (2/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Agent {
  // nur aufgerufen, wenn Agent dynamisch geladen
  public static void agentmain(String agentArgument
            , Instrumentation instrumentation) {
    System.out.println("agentmain aufgerufen");
    premain(agentArgument, instrumentation);
  }

  // nur Aufgerufen, wenn Agent statisch geladen, d. h.
  // mit –javaagent: …   (eher ueblich, da so „normale“
  // Programme bearbeitbar)
  public static void premain(String agentArgument
            , Instrumentation instrumentation) {
    System.out.println("arg: " + agentArgument);
    instrumentation.addTransformer(new MeinTransformer());
  }
}
```

Beispielprojekt (3/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class MeinTransformer implements ClassFileTransformer {
  @Override
  public byte[] transform(ClassLoader loader
        , String className
        , Class<?> classBeingRedefined
        , ProtectionDomain protectionDomain
        , byte[] classfileBuffer)
                    throws IllegalClassFormatException {
    System.out.println("transform: " + className);
    // hier wuerden spannende Dinge passieren
    return classfileBuffer;
  }
}
// damit nicht direkt mit Byte-Code gearbeitet werden muss
// (geht auch), gibt es diverse Bibliotheken, u. a.
// ASM: https://asm.ow2.io/
// Javassist: https://www.javassist.org/
```

Beispielprojekt (4/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// Aufruf mit VM-Argument:
//-javaagent:F:\workspaces\exports\sqmAgent\sqmAgent.jar=optionen
public class MainsqmAgent {

  public static void main(String[] args) {
   Konto k = new Konto();
   k.einzahlen(42);
   System.out.println("k: " + k);
  }

}
```

Beispielprojekt (5/7) - Ausgabe

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
arg: optionen
transform: sun/launcher/LauncherHelper
transform: java/lang/WeakPairMap$Pair$Weak
transform: java/lang/WeakPairMap$WeakRefPeer
transform: java/lang/WeakPairMap$Pair$Weak$1
transform: java/lang/StringCoding
transform: java/lang/StringCoding$1
transform: java/lang/StringCoding$StringDecoder
transform: java/nio/charset/CharsetDecoder
transform: sun/nio/cs/SingleByte$Decoder
transform: sun/nio/cs/ArrayDecoder
transform: java/lang/StringCoding$Result
transform: main/MainsqmAgent
transform: entity/Konto
k: Konto{stand=42}
transform: java/util/IdentityHashMap$IdentityHashMapIterator
transform: java/util/IdentityHashMap$KeyIterator
transform: java/lang/Shutdown
transform: java/lang/Shutdown$Lock
```

Beispielprojekt (6/7) – neue Main-Klasse

```
public static void loadAgent() throws  IOException
    , AttachNotSupportedException, AgentLoadException
    , AgentInitializationException {
  long processId = ProcessHandle.current().pid();
  String agentJar = "..\\..\\exports\\sqmAgent\\sqmAgent.jar";
  VirtualMachine vm = VirtualMachine.attach(""+processId);
  try {
    vm.loadAgent(agentJar, "optionen");
  } finally {
    vm.detach();
  }
}

public static void main(String[] args) throws Exception {
  loadAgent(); // (fast) nur sinnvoll vor Klassennutzung
  Konto k = new Konto();
  k.einzahlen(42);
  System.out.println("k: " + k);
 }
```

Beispielprojekt (7/7) – andere Ausgabe

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
agentmain aufgerufen
arg: optionen
transform: java/lang/IndexOutOfBoundsException
transform: java/lang/WeakPairMap$Pair$Weak
transform: java/lang/WeakPairMap$WeakRefPeer
transform: java/lang/WeakPairMap$Pair$Weak$1
transform: entity/Konto
k: Konto{stand=42}
transform: java/util/IdentityHashMap$IdentityHashMapIterator
transform: java/util/IdentityHashMap$KeyIterator
transform: java/lang/Shutdown
transform: java/lang/Shutdown$Lock
```

Objekterzeugung intern genauer

- zu jedem Konstruktor K(.) gehört eine Initialisierungsmethode <init>(.) ; die spitzen Klammern gehören zum Namen
- Idee: jeder Konstruktor kann als Klassenmethode interpretiert werden

- gibt weiterhin pro Klasse eine Klasseninitialisierungsmethode <clinit>() möglich; die spitzen Klammern gehören zum Namen; nur eine parameterlose Methode
- Methode existiert, wenn static{}-Methode oder/und Klassenvariablen existieren

Byteman -Konzept

- ermöglicht das Beobachten und Verändern des Methodenverhaltens von außen
- realisiert als Agent
- System arbeitet regelbasiert mit Event – Condition –Action Paradigma
- Event: Methode wird aufgerufen, Methode wird verlassen
- Condition: Boolescher Java-Ausdruck (oft TRUE)
- Action: Ausgaben, Veränderung von Parametern, Veränderung von Ergebnissen; generell Aufruf von Klassenmethoden
- relativ viel Dokumentation; nicht sehr ins Detail gehend
- https://byteman.jboss.org/
- https://dl.acm.org/doi/10.1145/1960314.1960325 (frei in HS)

Byteman - Regelaufbau

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
# das danach ist Kommentar
RULE verfolge setX  # freier, aber eindeutiger Name
CLASS main.Beispiel # welche Klasse, auch INTERFACE nutzbar
METHOD setX      # welche Methode, Parametertypen in Klammern
AT ENTRY         # Event  AT EXIT, AT INVOKE (Methode)
                 # AT LINE 42, AFTER READ/WRITE (Variable)
BIND par = $1    # Hilfsvariablen, $i für i-ter Parameter
IF true          # Condition
DO traceln("in " + $0 +" x " + par)  #$0 oder $this für this
ENDRULE
```

- reine ASCII-Regeln, Byteman hat Programme zur Syntaxprüfung
- Bei Klassen- und Methodennamen kein * erlaubt
- BIND und DO: Befehlssequenzen mit ; getrennt
- traceln(.) ist Klassenmethode eines Hilfsobjekts
- DO, auch möglich Exception zu werfen
- hier nur grober Einblick (u. a. weitere Events, Thread-Handling)

Byteman - Spezialsymbole

Symbol	Bedeutung
$!	AT EXIT: Rückgabewert AT INVOKE: lokaler Rückgabewert
$#	Anzahl Parameter
$0, $this	aktuelles Objekt
$1, $2, …, $n	n-ter Parameter
$@	Object-Array [$0,$1,$2,…]
$CLASS	voll qualifizierter Klassenname
$METHOD	String, detaillierter Methodenname
$<name>	Parametername oder Name lokaler Variable

Byteman – einführendes (schräges) Beispiel (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Quadrat {
  private int i = 0;
  private boolean ende = false;

  public void zeigeQuadrate() {
    double x = 42.42;
    while(!this.ende) {
      this.ausgabe(i++);
    }
  }

  public void ausgabe(int i) {
    System.out.println(i + ": " + (i*i));
  }
}
```

Byteman – einführendes Beispiel (2/4) – regeln.btm

```
RULE Showcase
CLASS sqmBytemanEinfuehrung.Quadrat
METHOD zeigeQuadrate
AT EXIT
IF callerEquals("MainsqmEinfuehrung.main",true) #nur hier
DO System.out.println("" + $this + "\n# " + $# + "\n0 "
      + $0 + "\nclass " + $CLASS + "\nmethod " + $METHOD
      + "\ni " + $this.i + "\nx " + $x)
ENDRULE

RULE Stop die Schleife
CLASS sqmBytemanEinfuehrung.Quadrat
METHOD ausgabe
AT ENTRY
BIND par = $this.i
IF par > 3
DO System.out.println("Hallo");
   $this.ende = true
ENDRULE
```

Byteman – einführendes Beispiel (3/4) – regeln.btm


```
RULE Konstruktor
CLASS sqmBytemanEinfuehrung.Quadrat
METHOD <init>
AT EXIT
IF TRUE
DO System.out.println("" + $this + "\nclass " + $CLASS
        + "\nmethod " + $METHOD)
ENDRULE
```

- Falls eine Regel nicht anwendbar, oben z. B. Zugriff auf $1, wird sie ohne Fehlermeldung einfach nicht ausgeführt
- gibt Prüfprogramm bmcheck (erstmal aufwändig zu konfigurieren)
- https://downloads.jboss.org/byteman/4.0.13/byteman-programmers-guide.html

Byteman – einführendes Beispiel (4/4)

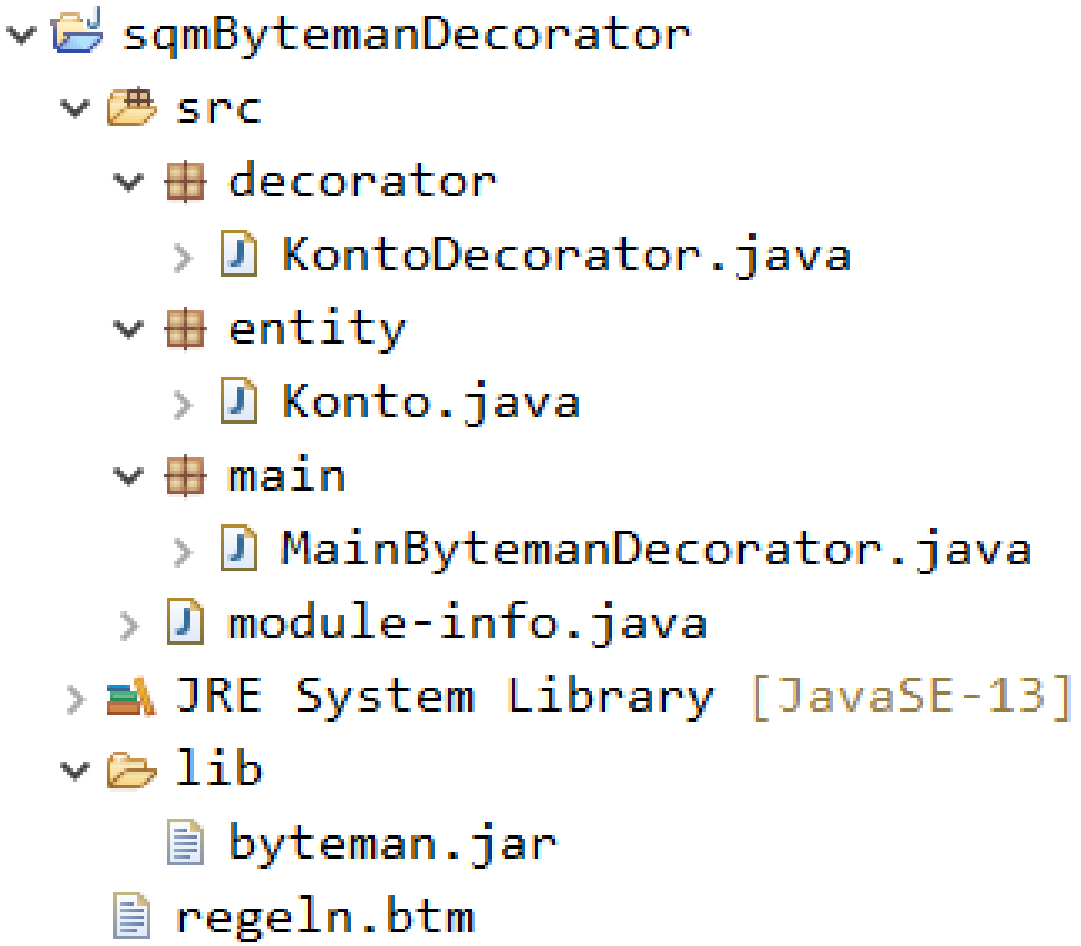
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class MainsqmEinfuehrung {
  //Aufruf mit -javaagent:lib\byteman.jar=script:regeln.btm
  public static void main(String[] args) {
    new Quadrat().zeigeQuadrate();
  }
}
```

```
sqmBytemanEinfuehrung.Quadrat@1810399e
class sqmBytemanEinfuehrung.Quadrat
method <init>() void
0: 0
1: 1
2: 4
Hallo
3: 9
sqmBytemanEinfuehrung.Quadrat@1810399e
# 0
0 sqmBytemanEinfuehrung.Quadrat@1810399e
class sqmBytemanEinfuehrung.Quadrat
method zeigeQuadrate() void
i 4
x 42.42
```

Byteman – Decorator (1/4)

- es können keine Variablen oder Methoden hinzugefügt werden
- aber Zugriffe können protokolliert und verändert werden
- Originalklasse (final) wird verändert und genutzt
- wieder Hilfsklasse zur Verwaltung der Gebühren
- wieder Key,Value-Pärchen (Konto, Gebühren)
- Byteman muss nicht im CLASSPATH sein



Byteman – Decorator (2/4) - Verwaltungsklasse

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class KontoDecorator {
  private static Map<Konto, Integer> gebuehren
                                      = new HashMap<>();

  public static void einzahlgebuehr(Konto k) {
    if (gebuehren.get(k) == null) {
      gebuehren.put(k, 4);
    } else {
      gebuehren.put(k, gebuehren.get(k) + 4);
    }
  }

  public static int gebuehren(Konto k) {
    if (gebuehren.get(k) == null) {
      return 0;
    }
    return gebuehren.get(k);
  }
}
```

Byteman – Decorator (3/4) – Regeln regeln.btm

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
RULE Einzahlsteuer
CLASS entity.Konto
METHOD einzahlen
AT ENTRY
BIND param = $1 - 4
IF true
DO decorator.KontoDecorator.einzahlgebuehr($0);
   $1=param
ENDRULE

RULE Beim Get Steuern Zurueckmogeln
CLASS entity.Konto
METHOD getStand
AT EXIT
BIND param = $! + decorator.KontoDecorator.gebuehren($0)
IF true
DO $!=param
ENDRULE
```

Byteman – Decorator (4/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// Aufruf mit -javaagent:lib\byteman.jar=script:regeln.btm
public class MainBytemanDecorator {

  public static void main(String[] args) {
    Konto k1 = new Konto();
    Konto k2 = new Konto();
    System.out.println("k1 == k2: " + (k1 == k2));
    k1.einzahlen(42);
    k1.einzahlen(4200);
    System.out.println("k1: " + k1
        +"\nstand: " + k1.getStand()
        +"\nk2: " + k2);
  }
}
```

```
k1 == k2: false
k1: Konto{stand=4234}
stand: 4242
k2: Konto{stand=0}
```

Byteman und Tests

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- generell kann Byteman in JUnit-Tests genutzt werden, da sie, wie jedes Java-Programm mit einem Agent aufgerufen werden können

```
@Test
 void test() {
   Konto sut = new Konto();
   sut.einzahlen(42);
   Assertions.assertEquals(42, sut.getStand());
   Assertions.assertTrue(sut.toString().contains("38"));
 }
```

- Byteman unterstützt aber JUnit-Tests darüber hinaus
- in Annotationen kann Regel-Datei ausgewählt werden, für alle Tests und Tests individuell
- Byteman-Regeln können als Annotationen angegeben werden
- JUnit 5-Erweiterung nicht kompatibel mit Modul-System (da Paketerweiterung)

Byteman-Tests (1/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
package main;
public class Beispiel { // zu testen
  public static int wert() {
    return 0;
  }

  public static int wert2() {
    return 0;
  }
}

RULE neues Endeergebnis # in regeln.btm
CLASS main.Beispiel
METHOD wert
AT EXIT
IF true
DO RETURN 42   # alternativ zu $!
ENDRULE
```

Byteman-Tests (2/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// immer benötigt
// VM: -Djdk.attach.allowAttachSelf=true -Xshare:off
// man kann Regelnutzung verfolgen
// VM: -Dorg.jboss.byteman.verbose
// kein –javaagent; erledigt Teststart selbst

@WithByteman
@BMUnitConfig(loadDirectory = ".") // hier ueberfluessig
@BMScript(value = "regeln.btm")
public class BeispielTest {

  @Test
  public void test1() {
    Assertions.assertEquals(42, Beispiel.wert());
    System.out.println("Hai");
  }
```

Byteman-Tests (3/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
@Test
@BMRule(
    name = "auch 41 moeglich",
    targetClass = "main.Beispiel",
    isInterface = false, // true, alle implement. Klassen
    targetMethod = "wert2",
    targetLocation = "EXIT",
    binding = "a=40; b=41",
    condition = "TRUE",
    action = "traceln(\"------\"); "
           + "RETURN b")
public void test2() {
  Assertions.assertEquals(41, Beispiel.wert2());
  System.out.println("Hai2");
}
// zur Veranschaulichung, Teile redundant
```

Byteman-Tests (4/5) – Fehler erzeugbar

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
@Test
@BMRule(
    name = "Fehler erzeugbar",
    targetClass = "main.Beispiel",
    targetMethod = "wert2",
    targetLocation = "EXIT",
    action = "throw new IllegalArgumentException()")
public void test3() {
  try {
    Beispiel.wert2();
    Assertions.fail();
  } catch (Exception e) {
    System.out.println("Hai3");
  }
}
```

```
org.jboss.byteman.contrib.bmunit.BMUnit5ConfigHandler installing main.BeispielTest
byteman jar is F:\workspaces\eclipseWS\sqmBytemanTest\lib\byteman.jar
Setting org.jboss.byteman.allow.config.update=true
Hai
------
Hai2
Hai3
```

Byteman-Tests (5/5) – Regeln kombinierbar

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
  @BMRules(rules = {
    @BMRule(
      name = "schnelles Ergebnis",
      targetClass = "main.Beispiel",
      targetMethod = "wert",
      targetLocation = "ENTRY",
      action = "RETURN 40")
  , @BMRule(
      name = "schnelles Ergebnis2",
      targetClass = "main.Beispiel",
      targetMethod = "wert2",
      targetLocation = "ENTRY",
      action = "RETURN 20")  })
  public void test4() {
    Assertions.assertEquals(800
                          , Beispiel.wert() * Beispiel.wert2());
  }
}
```

Byte Buddy

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- bisher Objekte nicht veränderbar
- Byte Buddy erlaubt es Klassen zu verändern, z. B. Objektvariablen und Methoden zu ergänzen
- zentraler Use Case vergleichbar zu Byteman, Ausführung von Methoden wird im Detail beobachtet und kann bearbeitet werden
- anders: Bearbeitungen können sich auf mehrere Klassen und Methoden beziehen; dazu stehen viele Filter zur Verfügung
- programmiert als Domain Specific Language (DSL) in Java, d. h. Fluent Programming über Method Chaining
- Detaillierte, aber nicht sehr systematische Dokumentation; hilft Testfälle des Werkzeugs selbst zu lesen
- https://bytebuddy.net/#/

Byte-Buddy – ein Ansatz

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Umsetzung als Agent
- mit ElementMatchers passende Methoden oder Klassen auswählen und an sogenannte Advice Klassen delegieren
- Advice-Klasse hat zwei annotierte Klassenmethoden, annotiert mit @Advice.OnEnter und @Advice.OnExit
- Advice-Methoden haben über annotierte Parameter Zugriff auf Objekte, Methoden und Variablen; obige Methoden können über Rückgabe-Typ auf gemeinsames Objekt zugreifen
- Klasse Konto darf weiterhin nicht final sein

- Erinnerung: verschiedene Bereiche pro ClassLoader

Byte Buddy – Decorator mit Advice (1/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Agent {

  public static void premain(String a, Instrumentation in) {
    Transformer tr = ((Builder<?> builder, TypeDescription t,
        ClassLoader classLoader, JavaModule javaModule) -> {
      Builder<?> tmp = builder
          .visit(Advice.to(KontoAdvice.class)
          .on(ElementMatchers.hasMethodName("einzahlen")));
      return tmp;
    });

    new AgentBuilder.Default()
        .with(RedefinitionStrategy.RETRANSFORMATION)
        .type(ElementMatchers.nameContains("Konto"))
        .transform(tr)
        .installOn(in);
  }
}
```

Byte Buddy – Decorator mit Advice (2/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class KontoDecorator {

  private static boolean initialized = false;
  private static Class<? extends Konto> dynamicUserType;

  public static Konto newKonto() throws Exception {
    if (!initialized) {
      dynamicUserType = new ByteBuddy()
          .subclass(Konto.class)
          .defineField("steuer", int.class, Visibility.PUBLIC)
          .make()
          .load(KontoDecorator.class.getClassLoader(),
              ClassLoadingStrategy.Default.CHILD_FIRST)
          .getLoaded();
      initialized = true;
    }
```

Byte Buddy – Decorator mit Advice (3/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
    try {
      Konto k = dynamicUserType.newInstance();
      return k;
    } catch (InstantiationException e) {
      return null;
    }
  }

  public static int steuer(Konto k) {
    try {
      Field feld = k.getClass().getField("steuer");
      return (Integer) feld.get(k);
    } catch (NoSuchFieldException | SecurityException
        | IllegalArgumentException | IllegalAccessException e) {
    }
    return Integer.MIN_VALUE;
  }
}
```

Byte Buddy – Decorator mit Advice (4/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class KontoAdvice {

  @Advice.OnMethodEnter(suppress = Throwable.class)
  static void enter(
         @Advice.Origin("#m") String method
      ,  @Advice.AllArguments Object[] args
      ,  @Advice.Argument(
           value = 0,
           readOnly = false, typing = Typing.DYNAMIC) Object o
      ,  @Advice.This Object konto)
      throws NoSuchFieldException, SecurityException,
      IllegalArgumentException, IllegalAccessException {
    if (method.equals("einzahlen")) {
      o = ((Integer) args[0]) - 4;
      Field feld = konto.getClass().getField("steuer");
      int alt = (Integer) feld.get(konto);
      feld.set(konto, alt + 4);
    }
  }
```

Byte Buddy – Decorator mit Advice (5/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
   @Advice.OnMethodExit(suppress = Throwable.class)
   static void exit(
       @Advice.Origin("#t  #m  #d  #s  #r") String method) {
     System.out.println("exit " + method);
   }
}

// Haette entry den Rueckgabetyp Blubb koennte in exit
// mit einem @Advice.Enter annotierten Parameter darauf
// zugegriffen werden

// Umsetzung erfolgt in einer Art Template, mit logischer
// Textersetzung; sollte dann Annotation keinen Sinn haben
// wird Advice ohne Fehlermeldung nicht genutzt !!!
```

Byte Buddy – Decorator mit Advice (6/6)

```
public static void main(String[] args) throws Exception {
   Konto k1 = KontoDecorator.newKonto();
   Konto k2 = KontoDecorator.newKonto();
   System.out.println("k1 == k2: " + (k1 == k2));
   k1.einzahlen(42);
   k1.einzahlen(4200);
   System.out.println("k1: " + k1
       +"\nstand: " + k1.getStand()
       +"\n Gebuehr: " + KontoDecorator.steuer(k1)
       +"\nk2: " + k2
       +"\nstand: " + k2.getStand());
 }
```

```
k1 == k2: false
exit entity.Konto  einzahlen  (I)V  (int)  void
exit entity.Konto  einzahlen  (I)V  (int)  void
k1: Konto{stand=4234}
stand: 4234
 Gebuehr: 8
k2: Konto{stand=0}
stand: 0
```

Video

4. Formale Verifikation (Model Checking)

4. Model Checking mit PROMELA und SPIN

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 4.1 Model Checking im Entwicklungskontext
- 4.2 Die Spezifikationssprache PROMELA
- 4.3 Simulation von PROMELA-Spezifikationen
- 4.4 Einfache Verifikationsmöglichkeiten
- 4.5 Verifikation von in LTL formulierten Anforderungen
- 4.6 Beispiele
- 4.7 Einsatzmöglichkeiten von Model Checkern

Literatur:

S. Kleuker, Formale Modelle der Softwareentwicklung, Vieweg+Teubner, Wiesbaden, 2009

G .J. Holzmann, The SPIN Model Checker, Addison Wesley, 2004

M. Ben-Ari, Principles of the Spin Model Checker, Springer, 2008

4.1 Model Checking im Entwicklungskontext

- Grundidee des Model Checkings
- Probleme der klassischen Software-Entwicklung
- Chancen und Probleme der Nutzung formaler Methoden

Begriff: Formale Modelle

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

formal kann bedeuten

- entweder: Formalismus und Bürokratie
- oder: Präzision und Exaktheit

Modell:

- Abbild der Realität, mit dem Aussagen über die Realität gewonnen werden können

passendes Modell:

- Modell, das zur Lösung einer Aufgabenstellung passt (Modell aus dem korrekte Aussagen bzgl. der Aufgabenstellung ableitbar sind)
- folgt: für verschiedene Aufgabenstellungen zur gleichen realen Situation können verschiedene Modelle sinnvoll sein

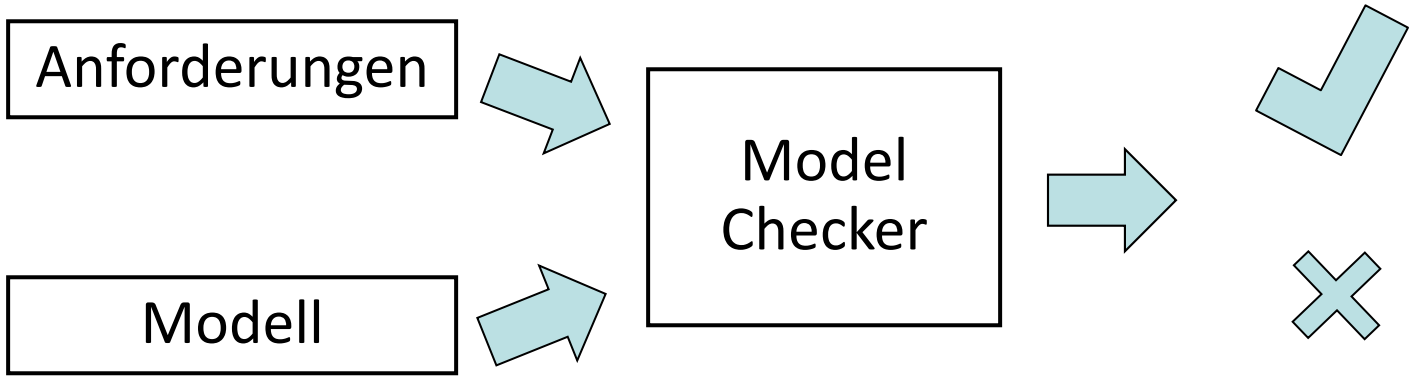
Motivation von Model Checking

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ansatz: Gegeben ist ein Modell (oder Spezifikation) und eine Anforderung, dann überprüft der Model-Checking-Algorithmus, ob das Modell die Anforderung erfüllt oder nicht. Falls das Modell die Anforderung nicht erfüllt, sollte der Algorithmus ein Gegenbeispiel liefern.

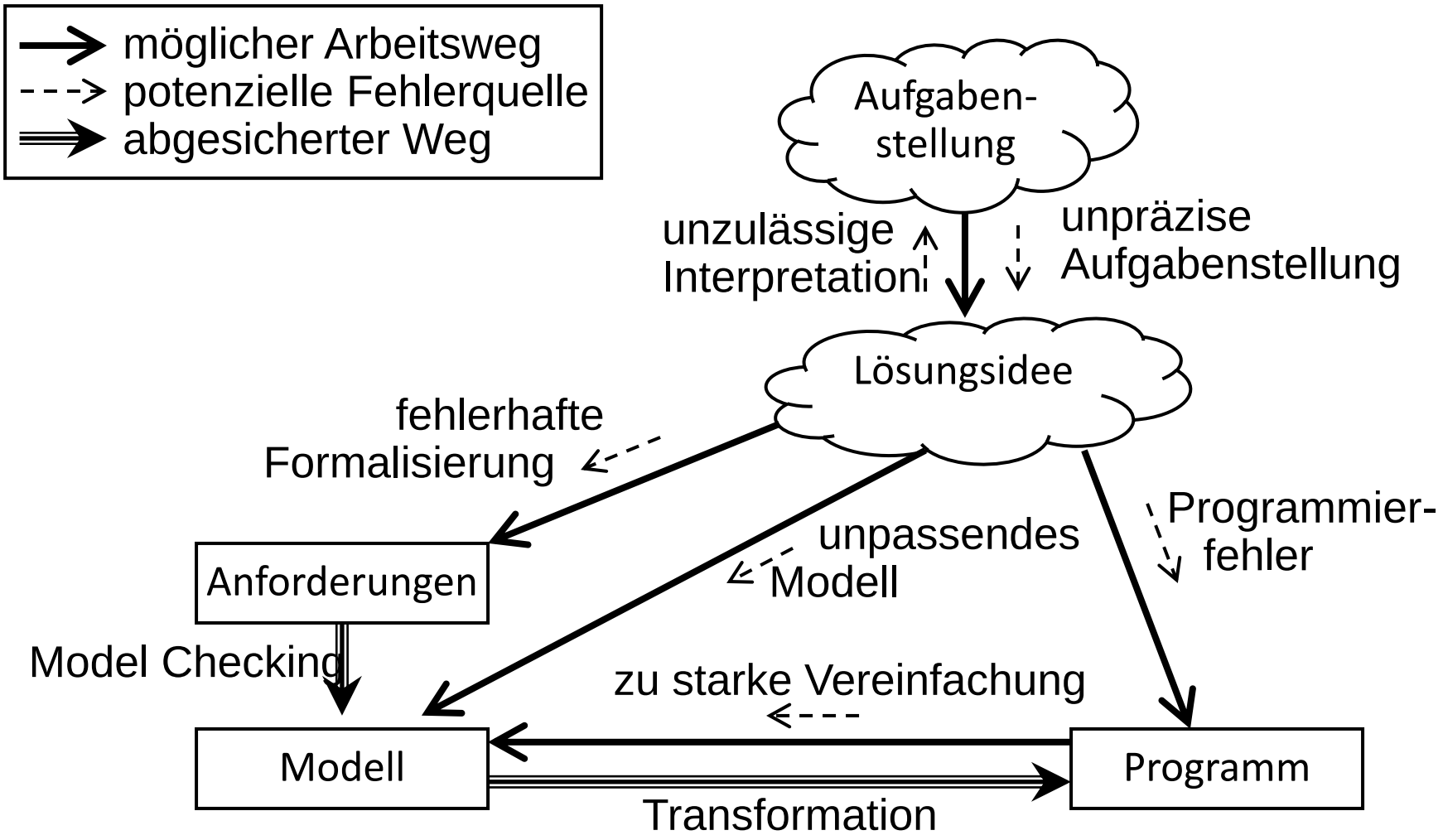
- SPIN entwickelt von Gerard Holzmann, zunächst als Simulator, dann Verifikationswerkzeug
- 2001 renommierten ACM Software System Award (z. B.: 1983 UNIX, 1987 Smalltalk, 1991 TCP/IP, 1995 World-Wide Web, 2002 Java)
- www.spinroot.com (frei verfügbar, seit 1991)
- Ansatz: Berechne alle erreichbaren Zustände und analysiere sie

Model Checking



- Anforderungen
  - implizit: terminiert immer, kein Deadlock, …
  - explizit: Zusicherungen, formale Logik, …
- Modell: Modellierungssprache mit formaler Semantik, bietet Sprachkonstrukte an (Nichtdeterminismus, Zeit, Wahrscheinlichkeit, …)
- Model Checker: Prüft, ob Modell Anforderungen erfüllt, liefert (wenn sinnvoll) Gegenbeispiel

Mögliche Korrektheitsprobleme



Varianten

- hier nur ein Ansatz: explizite Zustandsraum-Analyse
- viele weitere Varianten und Werkzeuge, z. B.
  - E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, (Eds.), Handbook of Model Checking, Springer, Charm (Schweiz), 2018 (1210 Seiten)
  - bei weitem nicht vollständig: https://en.wikipedia.org/wiki/List_of_model_checking_tools
  - weitere Werkzeug-Beispiele:

    https://formal.iti.kit.edu/tools/?lang=en

    http://seahorn.github.io/

    https://github.com/tlaplus/tlaplus
- (formale) Verifikation umfasst weitere Ansätze

Anwendungen als Teil der Entwicklung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- hoch-kritische Systeme: z B. Flut-Tor nahe Rotterdam
- Raumfahrt: Cassini, Mars-Rover
- Deadlocks in Treibern (Microsoft)
- Analyse von SPS-Programmen
- Speicherchips: Registertransferebene, Transistorschaltebene
- BMW: Komponente zur aktiven Lenkunterstützung (genauer Abschalten im Fehlerfall)
- Nissan: Modellebene vor Prototypenerstellung
- Analyse von Toyota SW auf irrtümliche Beschleunigungsereignisse
- Medizinische Übertragungsprotokolle
- „To Build Trust In Artificial Intelligence, IBM Wants Developers To Prove Their Algorithms Are Fair “
- …

4.2 Die Spezifikationssprache PROMELA

- Nichtdeterministische Schleifen und Alternativen
- Datentypen
- nebenläufige Prozesse
- atomare Bereiche
- synchrone und asynchrone Kommunikation
- Varianten der Nachrichtenabarbeitung

Ziele von Spezifikationssprachen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Erstellung von Modellen
  - möglichst einfach beschreibbar
  - fachlich möglichst kompakt

- Spezifikationssprachen bieten vielfältige Sprachkonstrukte an; generell nur die nutzen, die in der Praxis (realen Welten) vorhanden sind; also zur Realisierung zur Verfügung stehen

Erste Sprachkonstrukte von PROMELA

```
#define N 20
int erg;
proctype Ungerade(){
  int x = 1;
  do
  :: x <= N ->
     erg = erg + x;
     x = x + 2
  :: x > N -> break
  od
}
proctype Gerade(){
  int x = 0;
  do
  :: x <= N ->
     erg = erg + x;
     x = x + 2
  :: x>N -> break
  od
}
```

```
init{
  erg = 0;
  run Ungerade();
  run Gerade()
}
```

PROcess MEta LAnguage

- Prozesse spezifizieren
- Spezifikationen können Nichtdeterminismus enthalten
- Syntax lehnt sich leicht an C an
- globale und lokale Variablen

printf

- Spezifikationen können printf-Befehl aus C enthalten
- **printf("Text mit Platzhaltern", Var1, … VarN);**

```
#define PROZESSE 4
byte x=0;

active[PROZESSE] proctype P(){ // 4 Prozesse starten direkt
  do
  :: x<8 -> x=x+1;
            printf("Prozess %d: x=%d\n",_pid, x);
   :: else -> break
  od
}
```

Sichtbar sind Ausgaben nur Simulationsausgabe (genauer später):

```
24: proc  1 (P) line   5 "pan_in" (state 4) [x = (x+1)]
Prozess 1: x=6
 25:        proc  1 (P) line   7 "pan_in" (state 3)      [printf('Prozess %d: x=%d\\n',_pid,x)]
```

Nichtdeterminismus

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
if
:: x<10 -> x=x+2
:: x<15 -> x=x+1
:: else -> skip
fi;
```

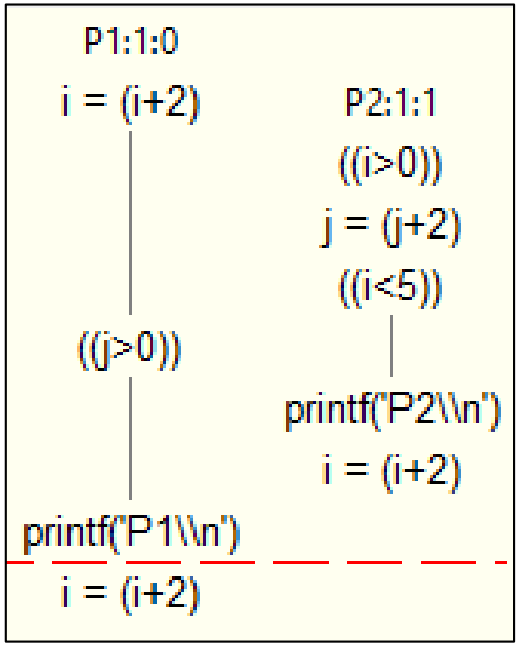
- Es werden alle ausführbaren Alternativen (stehen nach `::`) bestimmt, dann nicht-deterministisch eine ausgewählt (Guarded-Command-Language; Dijkstra)
- Boolesche Bedingungen sind ausführbar, genau dann, wenn sie wahr sind
- `x>4; x<3; x==5;` kann Spezifikationsteil sein (wait until …)
- else nur, wenn alle anderen Alternativen nicht möglich
- bei x =22 und ohne else bleibt Spezifikation bei if stehen
- Statt `->` könnte auch `;` stehen

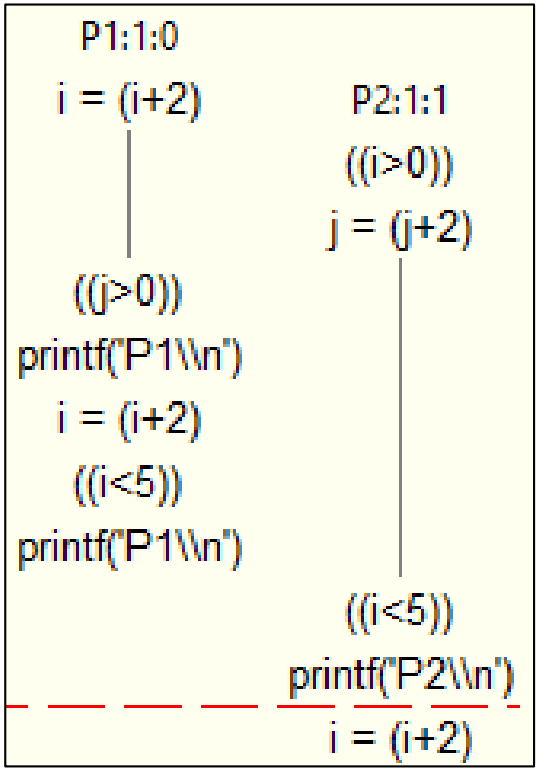
Ausführbarkeit Boolescher Bedingungen

```
byte i = 0;
byte j = 0;

active proctype P1(){
  i = i + 2;
  j > 0;
  printf("P1\n");
  i = i + 2;
  i < 5;
  printf("P1\n");
}

active proctype P2(){
  i > 0;
  j = j + 2;
  i < 5;
  printf("P2\n");
  i = i + 2;
}
```





Elementare Datentypen in PROMELA

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Datentyp	Wertebereich	Anmerkung
bit	0..1	
bool	0..1	auch Werte true (==1) und false (==0) möglich
byte	0..255	
chan	1..255	Kommunikationskanäle (später)
mtype	1..255	Nachrichtenwerte (gleich)
pid	0...255	für Prozessidentifikatoren (später)
short	$-2^{15}..2^{15}-1$	
int	$-2^{31}..2^{31}-1$	
unsigned	$0..2^{<Wert>}-1$	Anzahl Bits <Wert> wird angegeben, z.B. **`unsigned x:5`**; Maximalwert 31

Prioritäten in PROMELA

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Priorität	Operator	Kommentar
1	`() [] .`	Klammern, Feldklammern, Teilkomponente
2	`! ~`	Negation, Bit-Komplement
3	`* / %`	Multiplikation, Division, Modulo
4	`+ -`	Addition, Subtraktion
5	`<< >>`	bitweiser Links- und Recht-Shift
6	`< <= > >=`	kleiner, kleiner-gleich, größer, größer-gleich
7	`== !=`	Gleichheit, Ungleichheit
8	`&`	bitweise Und
9	`|`	bitweise Oder
10	`&&`	logisches Und
11	`||`	logisches Oder
12	`=`	Zuweisung

Optimierte Summenberechnung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
#define N 20
int erg;
proctype Summiere(int start){ // mehrere Parameter
                              // mit ; trennen
  int x=start;
  do
  :: x <= N -> erg = erg + x;
                x = x + 2
  :: x > N -> break
  od
}
init{
  erg = 0;
  run Summiere(1);
  run Summiere(0)
}
```

do, if, active, goto

```
#define PROZESSE 2
byte x=0;
byte y;

active [PROZESSE] proctype ifdoSpiel(){
  do
  :: x<10 -> x=x+1
  :: (x> 0 && x<10) -> x=x-1
  :: x< 9 -> x=x+3
  :: else -> x=x+4; break
  od;
  if
  :: x<10 -> y=0
  :: x>=10 && x<15 -> y=1
  :: else -> y=2
  fi;
  nochmal:
  if
  :: x<10 && y>0 -> goto nochmal
  :: else -> skip
  fi
}
```

goto sollte selten verwandt werden, häufiger benutzt, wenn aus anderer Sprache nach PROMELA übersetzt wird

active: Prozess startet sofort

active [3]: drei dieser Prozesse starten sofort

Vereinfachter Determinismus ab SPIN 6

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

```
init{
  int a[5] = {1,2,3,4};
  int i;
  for (i: 5..7){
    printf("%d\n",i);
  }
  for (i in a) {
    printf("a[%d] ist %d\n",i, a[i]);
  }
}
```

```
Simulationsausgabe:
5
6
7
a[0] ist 1
a[1] ist 2
a[2] ist 3
a[3] ist 4
a[4] ist 4
```

for-Schleife wird intern durch do ersetzt

Array mit letztem Wert fortgesetzt

Vereinfachter Nichtdeterminismus ab SPIN 6

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
init{
  int i;
  int j;
  for (i: 1..4){
    select (j: 1..100);
    printf("%d: %d\n",i,j);
  }
}
```

```
Simulationsausgabe:
1: 3
2: 1
3: 2
4: 2
```

Anmerkung: select wird automatisch ersetzt durch

```
j=1;
do
:: j < 100 -> j = j+1;
:: true -> break;
od;
```

Für Simulation kritisch (Ergebnisse nicht gleichverteilt); für Verifikation ok, da alle Werte erreicht werden

Datentypen in PROMELA

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
#define ANZAHL 3
/* Ein einziger Aufzählungstyp mtype, Werte von rechts nach
   links ausgehend von 1 (wasser==5) nummeriert */
mtype = {sekt,wodka,osaft,bier};
mtype = {whisky,wasser};

/* Records, wie in C, aber keine Zeiger */
typedef Person{
  short id;
  mtype trinkt;
};

/* Felder (Arrays) */
Person nase[ANZAHL];

/* Felder auch in Records erlaubt
typedef Kommuniziert{
  Person p[2];
};
```

Spezifikation verteilter Systeme

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
byte x=0;

active proctype P1(){
  x=x+1;
  x=x+1;
}

active proctype P2(){
  x=2;
}
```

- SPIN nutzt so genannte Interleaving-Semantik, Prozesse können abwechselnd voranschreiten
- Mögliche Endergebnisse x=2, x=3, x=4

Atomare Bereiche

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
byte x=0;

active proctype P1(){
  atomic{
    x=x+1;
    x=x+1;
  }
}

active proctype P2(){
  x=2;
}
```

- atomare Bereiche werden wie ein Schritt angesehen, sie sind nicht unterbrechbar (Ausnahme, wenn wartend)
- mögliche Ergebnisse: x=2, x=4

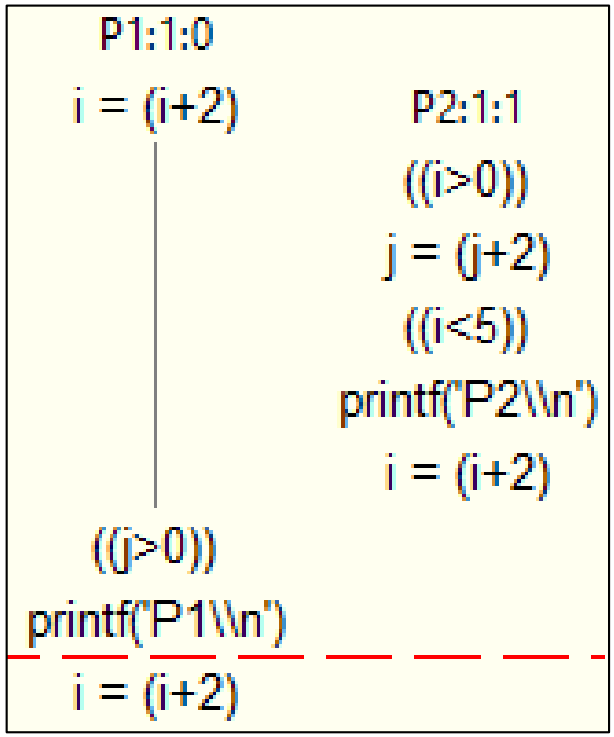
Atomare Bereiche genauer

```
byte i = 0;
byte j = 0;

active proctype P1(){
  atomic{
    i = i + 2;
    j > 0;
    printf("P1\n");
    i = i + 2;
    i < 5;
    printf("P1\n");
  }
}

active proctype P2(){
  atomic{
    i > 0;
    j = j + 2;
    i < 5;
    printf("P2\n");
    i = i + 2;
  }
}
```

- blockiert eine Anweisung im atomaren Block, wird dieser Block verlassen, wird er später wieder betreten, ist der Rest atomar
- einzig mögliche Ausführung:



d_step

- Atomare Bereiche dürfen Nichtdeterminismus haben
- d_step: Spezifikateur weiß, dieser Block deterministisch
- Wenn nichtdeterministisch, dann immer gleicher Weg
- Wenn nicht zu Ende ausführbar, dann Fehler
- Hinweis: Fehlermeldung nur bei Verifikation
- Beispiel vorheriger Folie liefert:

```
pan:1: block in d_step seq (at depth 0)
```

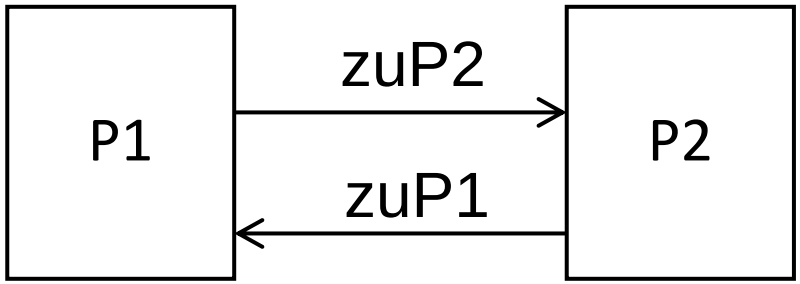
Interleaving genauer

- Analysiert man aus Sicht des Compilers, was bei x=x+1 passiert, so handelt es sich um mehrere Schritte:
  - Lade von der zu x gehörenden Speicheradresse den Wert von x in einen Speicher der CPU
  - Führe auf der CPU die Erhöhung um eins durch
  - Schreibe das Ergebnis aus der CPU wieder in die zu x gehörende Speicheradresse
- zwei parallele Teilprogramme, die mit x=1 starten und x=x+1 durchführen, können zu x=2 oder x=3 kommen

- Wichtig ist, welche Aktionen ununterbrechbar sind (in Semantik festgelegt), diese Aktionen heißen *atomar*

Kommunizierende Prozesse

Video



- Prozesse können über Kanäle kommunizieren (z. B. physikalische Leitungen)
- anders als im Bild können in PROMELA mehrere Prozesse einen Kanal nutzen
- anders als im Bild sind Kanäle nicht gerichtet, jeder kann lesen und schreiben
- Kanäle können gepuffert sein
- gewähltes Modell sollte zur Realität passen (deshalb Pfeile im Bild)

Beispiel: Kommunikationsprotokoll (1/3)

informelle Spezifikation:

- Der Prozess P1 versucht, über zuP2 die Werte von 1 bis 10 an P2 zu senden. Eine Nachricht besteht dabei aus den Teilinformationen Nachrichtenname (hier send) und dem übermittelten Wert. P2 empfängt den Wert und schickt zur Bestätigung den gleichen Wert mit dem Nachrichtennamen ack für Acknowledge, also Bestätigung, an P1 über zuP1 zurück. Nachdem P1 den bestätigten Wert erhalten hat, wird der nächste Wert an P2 übermittelt.
- Weiterhin soll modelliert werden, dass die Verbindung von P1 nach P2 nicht fehlerfrei funktioniert. Wenn P2 einen Wert empfängt und P2 an der Qualität zweifelt, wird der vorher gesandte Wert an P1 gesendet. P1 wiederholt daraufhin den zuletzt übertragenen Wert.

Beispiel: Kommunikationsprotokoll (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
mtype = {send,ack};
byte N=10;
chan zuP2 = [0] of {mtype,byte}; /*Rendez-vous oder*/
chan zuP1 = [0] of {mtype,byte};  /*synchrone Komm. */
active proctype P1(){
  byte wert = 1;
  byte antwort;
  do
  :: wert <= N ->
     zuP2!send,wert;  /* zuP2!send(wert); */
     zuP1?ack,antwort;
     if
     :: antwort == wert -> wert = wert + 1
     :: else -> skip
     fi
  :: else -> break
  od
}
```

Beispiel: Kommunikationsprotokoll (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
active proctype P2(){
  byte neu = 0;
  byte ein = 0;
  do
  :: zuP2?send,ein ->
        if
        :: true ->
           neu = ein;
           if
           :: neu == N -> break
           :: else -> skip
           fi
        :: true -> skip
        fi;
        zuP1!ack,neu;
  od;
}
```

Synchrone Kommunikation

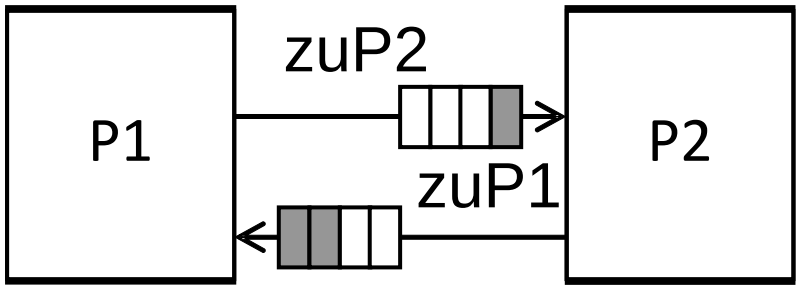
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
chan <Kanalname> = [<Puffergröße>] of {
                        <Nachrichtenteiltyp1>, ...
                        <NachrichtenteiltypN> };
```

- Puffergröße=0: Rendez-Vous oder synchrone Kommunikation
- Sender muss warten, wenn Empfänger nicht bereit
- Empfänger muss warten, wenn Sender nicht bereit

- Sender und Empfänger „wissen“, dass Kommunikation stattgefunden hat
- Sender und Empfänger kennen Reihenfolge der Nachrichten

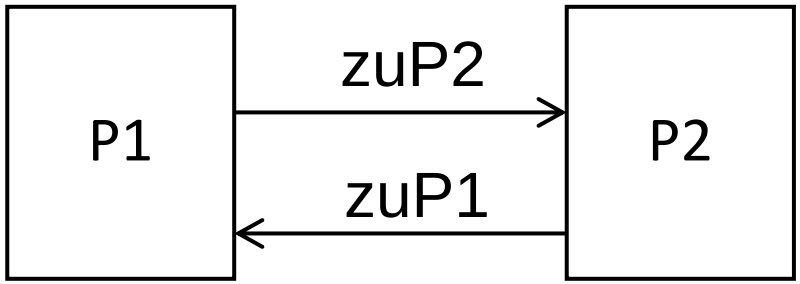
Asynchrone Kommunikation



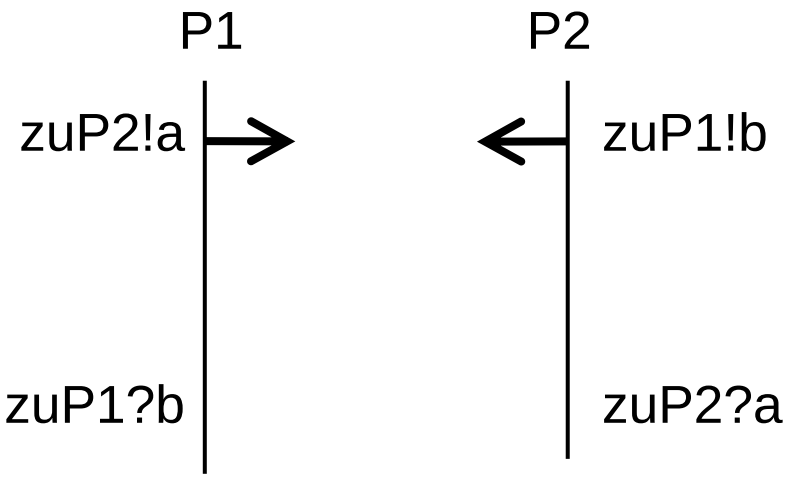
chan <Kanalname> = [<Puffergröße>] of
{ <Nachrichtenteiltyp1>, ... , <NachrichtenteiltypN> };

- Puffergröße>0: gepufferte oder asynchrone Kommunikation
- Sender sendet unabhängig vom Empfänger in dessen Puffer
- Empfänger entnimmt älteste Nachricht aus dem Puffer
- Falls keine Nachricht im Puffer, muss Empfänger warten
- Falls Puffer voll, können für SPIN zwei Varianten eingestellt werden:
  - Sender muss warten, bis Platz frei
  - Nachricht geht verloren

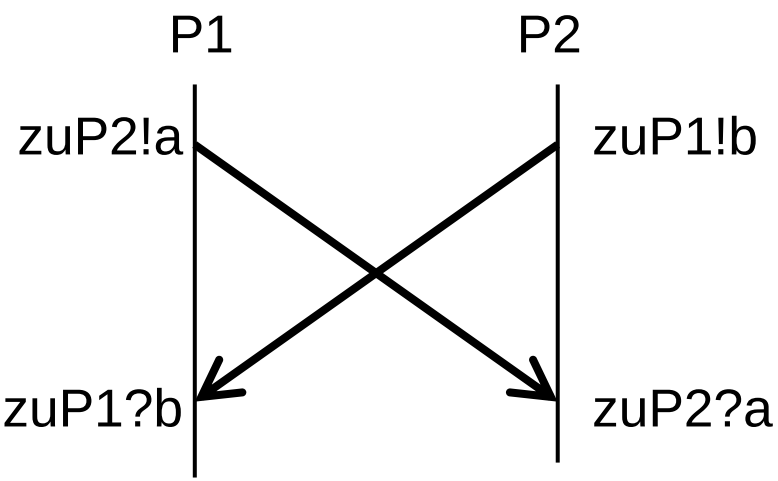
Typische Kommunikationsprobleme



synchron: Deadlock



asynchron: Race

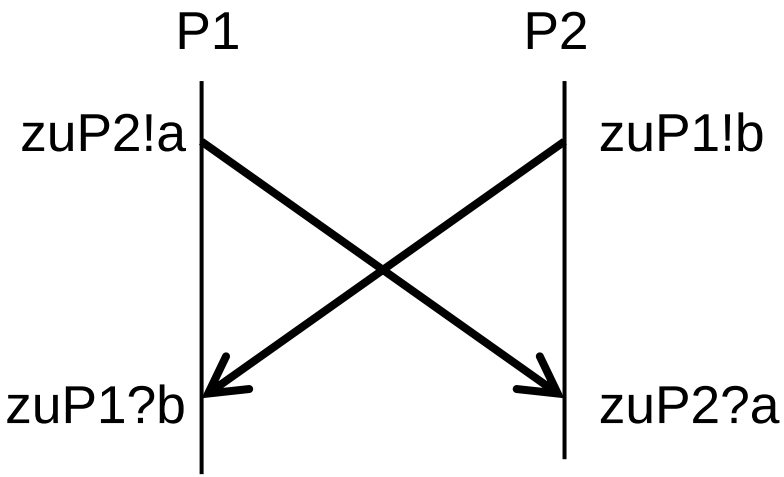


P1 meint erst a dann b

P2 meint erst b dann a

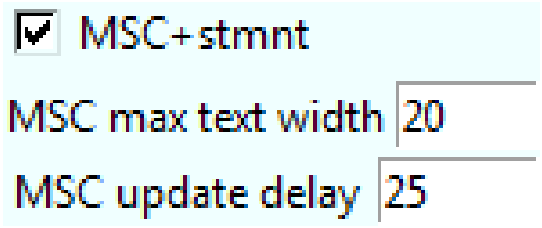
Message Sequence Charts (MSC)

- dienen zur Visualisierung der
  Prozesskommunikationen
- Jeder Prozess als senkrechter
  Balken
- Zeit verläuft von oben
  nach unten

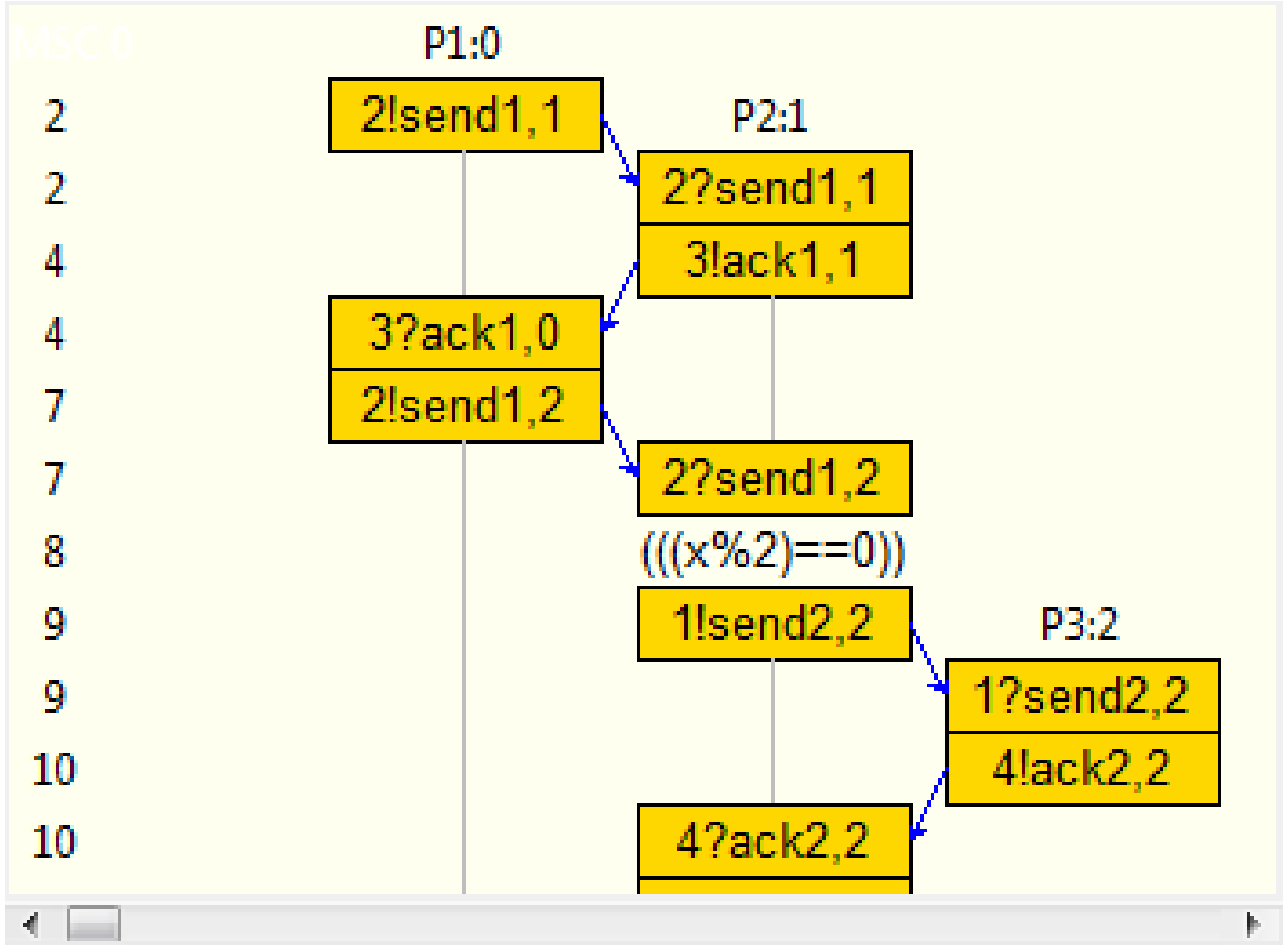


- Auf Lebenslinie des Prozesses sieht man, wann Prozess Kommunikation begonnen hat (senden) und wann sie empfangen wurde
- hilfreich um typische Abläufe zu visualisieren,
  um Testfälle zu definieren
- MSC in als Sequenzdiagramme mit erweiterten Möglichkeiten in UML 2.0

MSC in ispin (synchrones Beispiel)



Nachrichten-austausch zwischen P1 und P2 aus Beispiel



Kanalverhalten

- für die Simulation (später auch Verifikation) wird für alle Kanäle festgelegt, wie sie sich bei vollen Puffern verhalten
  - blockierend: Sender muss warten, bis ein Pufferplatz frei ist
  - verlierend: Sender kann immer senden, Nachricht geht eventuell verloren

Kanalanalysemöglichkeiten

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Funktion	Bedeutung / Rückgabewert
`len(k)`	Nachrichtenanzahl im Kanal k
`empty(k)`	ist Nachrichtenkanal k leer?
`nempty(k)`	ist Nachrichtenkanal k nicht leer?
`full(k)`	ist Nachrichtenkanal k voll?
`nfull(k)`	ist Nachrichtenkanal k nicht voll?

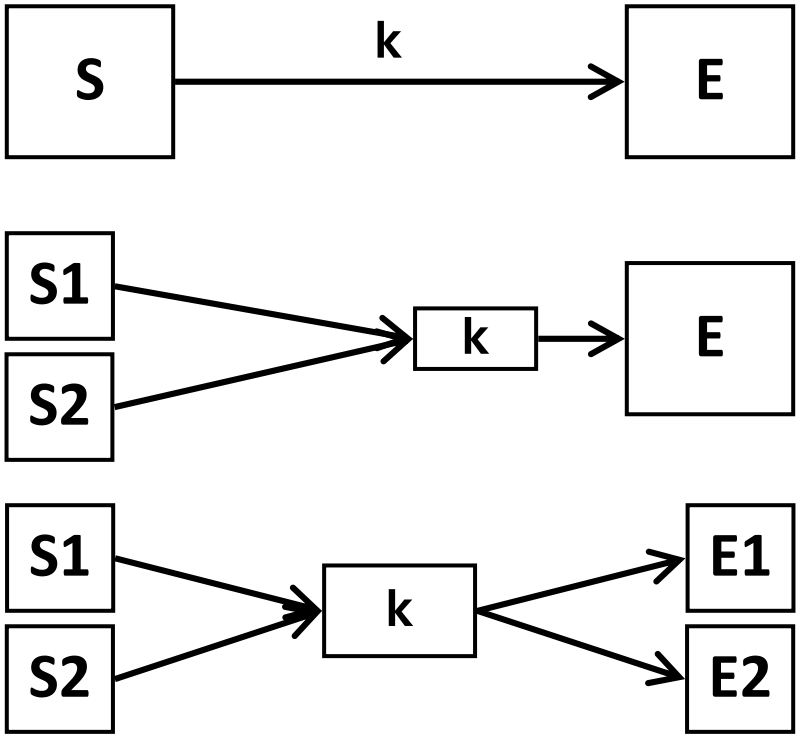
- zur Nutzung muss der Prozess den Kanal kennen (er muss sichtbar sein)
- hier: nur globale Kanäle (sonst auch als Variablen nutzbar)

Analyse der Kanäle in PROMELA

Spezifikation: `chan k = [2] of {mtype,byte}`

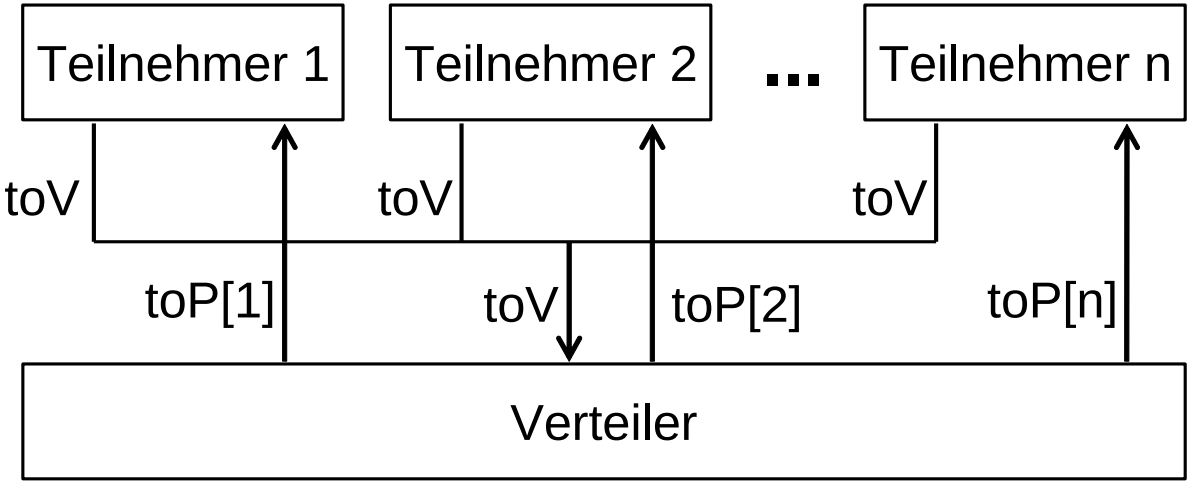
mögliche Visualisierungen

- gerichteter Kanal, nur ein Sender, nur ein Empfänger
- Kanal als Empfangsadresse, mehrere Sender, nur ein Empfänger
- Kanal als allgemeines Austauschmedium, mehrere Sender, mehrere Empfänger



- PROMELA erlaubt alle Interpretationen; es ist die jeweils zum Modell passende zu wählen

Beispiel: asynchrone Kommunikation (1/4)



informelle Spezifikation:

Beliebig viele Teilnehmer sind an Verteiler angeschlossen, alle Teilnehmer nutzen gleichen Kanal, um Informationen an Verteiler zu senden. Jede Information, die die Teilnehmer dem Verteiler schicken, wird an alle anderen Teilnehmer verteilt.

Im Beispiel schicken alle Teilnehmer die Zahlen 1 bis 10, jeder Teilnehmer summiert alle empfangenen Zahlen auf. Durch diese Festlegung ist es recht einfach, eine Terminierungsbedingung anzugeben; bei n Prozessen erhält man (n-1)-mal die Summe der Werte von 1 bis 10.

Beispiel: asynchrone Kommunikation (2/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
#define PROZESSE 3
#define MAX 10
#define GESAMT ((MAX*(MAX+1))/2)
#define PUFFER 2
mtype={send, rec}
chan toV = [PUFFER] of {mtype, byte, byte};
chan toP[PROZESSE]= [PUFFER] of {mtype, byte, byte};
init {
  atomic {
    byte i = 0;
    do
    :: i < PROZESSE ->
       run Teilnehmer(i);
       i = i + 1
    :: else -> break
    od;
    run Verteiler()
  };
}
```

Beispiel: asynchrone Kommunikation (3/4)

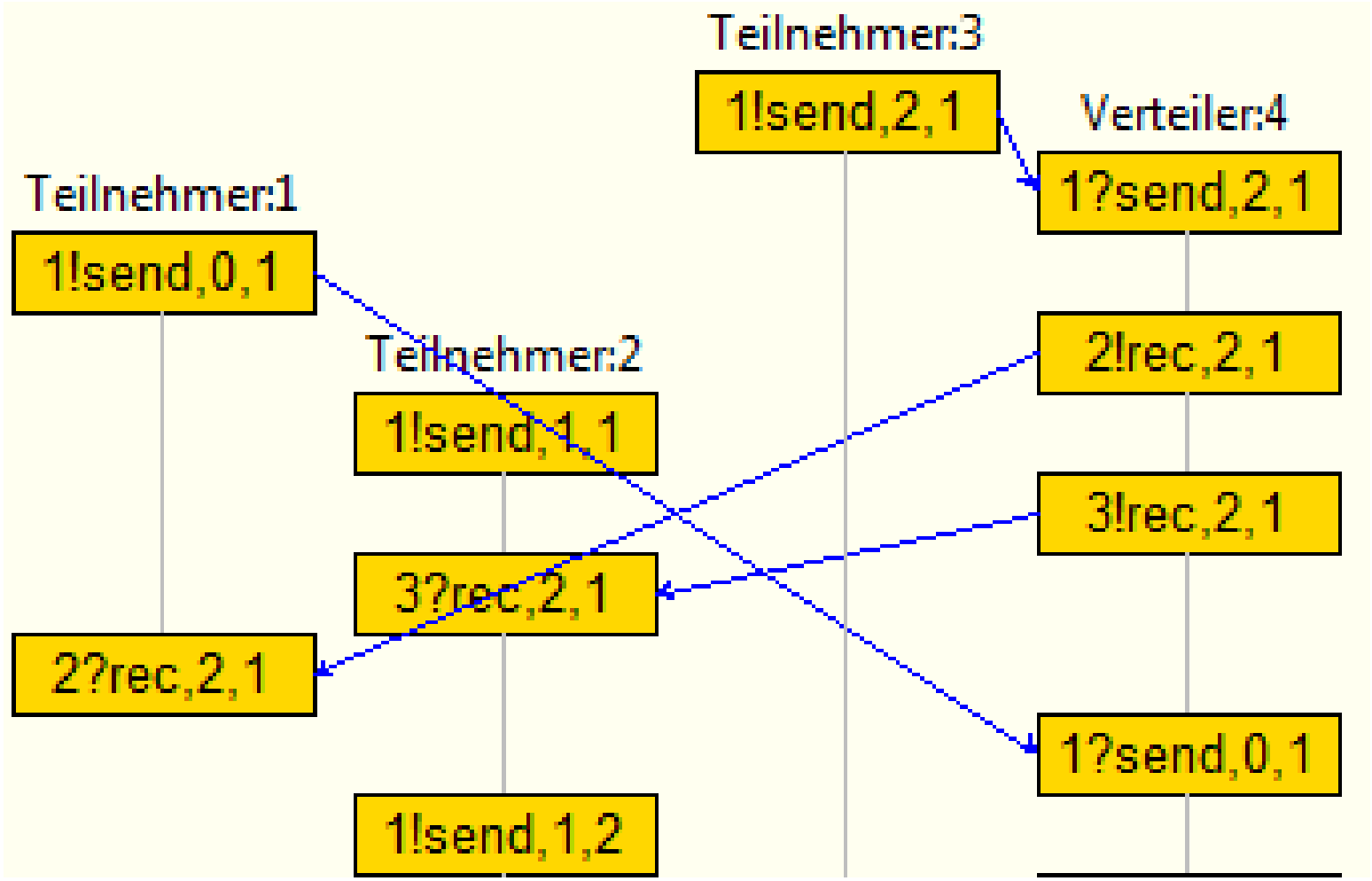
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
proctype Teilnehmer(byte id){
  byte i = 1;
  int summe = 0;
  byte wert;
  do
  :: atomic{
       i <= MAX && nfull(toV) ; /* 1 */
       toV!send, id, i;
       i = i + 1
     }
  :: summe < GESAMT * (PROZESSE - 1)
              && toP[id]?[rec,_,_] ->
     toP[id]?rec,_,wert;
     summe = summe + wert
  :: i == MAX + 1 && summe == GESAMT * (PROZESSE - 1) ->
     break
  od;
}
```

Beispiel: asynchrone Kommunikation (4/4)

```
proctype Verteiler(){
  byte name;
  byte wert;
  byte emp;
  do
  :: toV?send, name, wert ->
       emp = 0;
       do
       :: emp < PROZESSE && emp != name ->
          toP[emp]!rec, name, wert;
          emp = emp + 1
       :: emp == name -> emp = emp+1
       :: emp == PROZESSE -> break
       od;
  od;
}
```

Simulation: MSC asynchron



Empfangsarten

Befehl	Semantik	
`c?x,y`	empfangene Werte werden in x und y gespeichert	+
`c?42,y`	Empfang nur möglich, wenn erste Nachricht im Kanal c den Wert 42 hat	+
`c?eval(x),y`	Empfang nur möglich, wenn erster Wert dem Wert von x entspricht	+
`c?<x,y>`	Empfang in x und y, Nachricht wird nicht aus dem Puffer gelöscht	
`c??42,y`	es wird die älteste Nachricht aus dem Puffer empfangen, deren erster Wert 42 ist (wenn nicht vorhanden, dann warten)	
`c??<42,y>`	älteste passende Nachricht wird gelesen, bleibt aber im Puffer	

+ erlaubt auch bei synchroner Kommunikation

Prüfungen der Kommunikationsbereitschaft

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Befehl	Semantik
`c?[x,y]`	Prüfung, ob der Empfang möglich ist
`c?[42,y]`	ist c?42,y als nächstes möglich?
`c??[42,y]`	ist c??42,y möglich

erlaubt nur bei asynchroner Kommunikation

Asynchron: Bewachte Kommunikation

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- folgende Spezifikation funktioniert *nur asynchron* (leider synchron kein Syntaxproblem)
- _ für beliebiger Wert (wird weggeworfen)

```
chan a = [1] of {bit};
chan b = [1] of {bit};

active proctype Sender(){
  do
  :: a!1;
  :: b!0;
  od;
}

active proctype Empfaenger(){
  byte count = 0;
  do /* synchron syntaktisch ok, semantisch nicht */
  :: count%2 == 0 && a?[_] -> a?_; count=count+1;
  :: count%2 == 1 && b?[_] -> b?_; count=count+1;
  od;
}
```

Nie Kommunikation mit Bedingung verbinden

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- letzte Folie: Verknüpfung von Boolescher Bedingung und Kommunikationsbereitschaft ist ok
- *Prüfung der Kommunikationsbereitschaft nur bei asynchron sinnvoll / nutzbar*
- Verknüpfung von Boolescher Bedingung und Kommunikation nicht erlaubt [wäre semantisch Ausführbarkeit testen und gleichzeitig ausführen]

```
:: count%2==0 && a?_ -> count=count+1;
```

spin: line 18 "pan_in", Error: syntax error saw 'an identifier'

- Man beachte, folgendes hat andere Bedeutung

```
:: count%2==0 ->
     a?_;
     count=count+1;
```

Spezifikation eines Kommunikationskanals (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- Man kann Kommunikationskanäle auch „von Hand“ spezifizieren, dazu muss eine Queue als globale Variable spezifiziert werden
- Interleaving-Probleme sind zu beachten, Kommunikationsschritte sollten atomar sein

```
#define PUFFER 3
byte pos=0;  /* Realisierung einer Queue */
#define ISTVOLL (pos>=PUFFER)
mtype={send};
typedef Nachricht{
  mtype typ;
  byte inhalt;
};
Nachricht kanal[PUFFER];
```

Spezifikation eines Kommunikationskanals (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
active proctype Sender(){
  byte wert=1;
  Nachricht nach;
  do
  :: wert>=10 -> break;
  :: atomic{
        wert<=10 && !ISTVOLL;
        nach.typ=send; /* geht einfacher, zeigt
                          Prinzip */
        nach.inhalt=wert;
        kanal[pos].typ=nach.typ;
        kanal[pos].inhalt=nach.inhalt;
        pos=pos+1;
     }
     wert=wert+1;
  od;
}
```

Spezifikation eines Kommunikationskanals (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
active proctype Empfaenger(){
  byte summe=0;
  byte tmp;
  do
  :: atomic{
       pos>0;
       summe=summe+kanal[0].inhalt;
       tmp=0;
       do
       :: tmp<pos-1 ->
            kanal[tmp].typ=kanal[tmp+1].typ;
            kanal[tmp].inhalt=kanal[tmp+1].inhalt;
            tmp=tmp+1;
       :: else -> break;
       od;
       pos=pos-1;
     }
  od;
}
```

_pid

```
#define N 5
chan c[N] = [0] of {byte};
byte erg = 0;

active [N] proctype P(){
  c[_pid]!_pid;
}

active proctype Emp(){
  byte tmp = 0;
  byte wert;
  do
  :: tmp < N ->
       c[tmp]?wert;
       erg = erg + wert;
       tmp++;
  :: else -> break;
  od;
}
```

- Jeder Prozess erhält bei seiner Erzeugung eine lokale Variable _pid
- Die Erzeugung erfolgt in der Reihenfolge, in der die Spezifikationen in PROMELA stehen
- Typischer Einsatz: Erzeugung gleichartiger Prozesse mit _pid zur Unterscheidung
- Ergebnis: In erg steht die Summe von 1 bis N-1

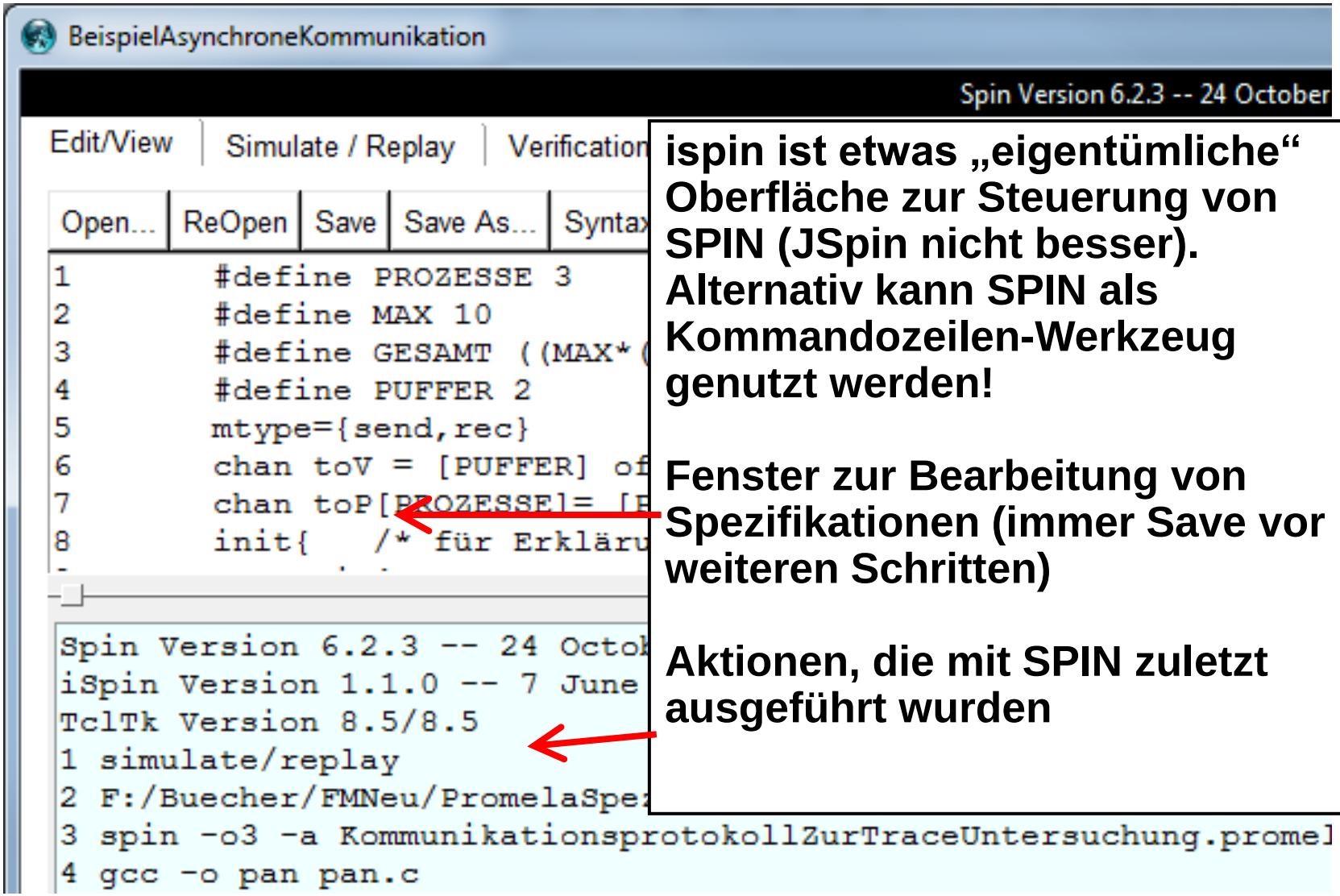
4.3 Simulation von PROMELA-Spezifikationen

- Einführung in ISPIN
- Syntaxprüfung
- Simulationseinstellung

Spezifikationsentwicklung mit PROMELA und ISPIN

- vorher auf Papier planen
- iterativ im Großen entwickeln
  - erst typischen Ablauf zum Laufen bringen
  - dann weitere Fälle ergänzen
- iterativ im Kleinen entwickeln
  - wenige syntaktisch korrekte (?) Zeilen eingeben und Syntax prüfen (Fehlermeldungen helfen nicht immer)
  - immer zu do ein od und zu if ein fi eingeben
- auf Sprachelemente der Vorlesung konzentrieren (Experimente getrennt durchführen)
- Simulieren; interaktiv und zufällig gemischt
- später: dann Verifikationsmöglichkeiten vorbereiten und verifizieren

Simulation mit ispin

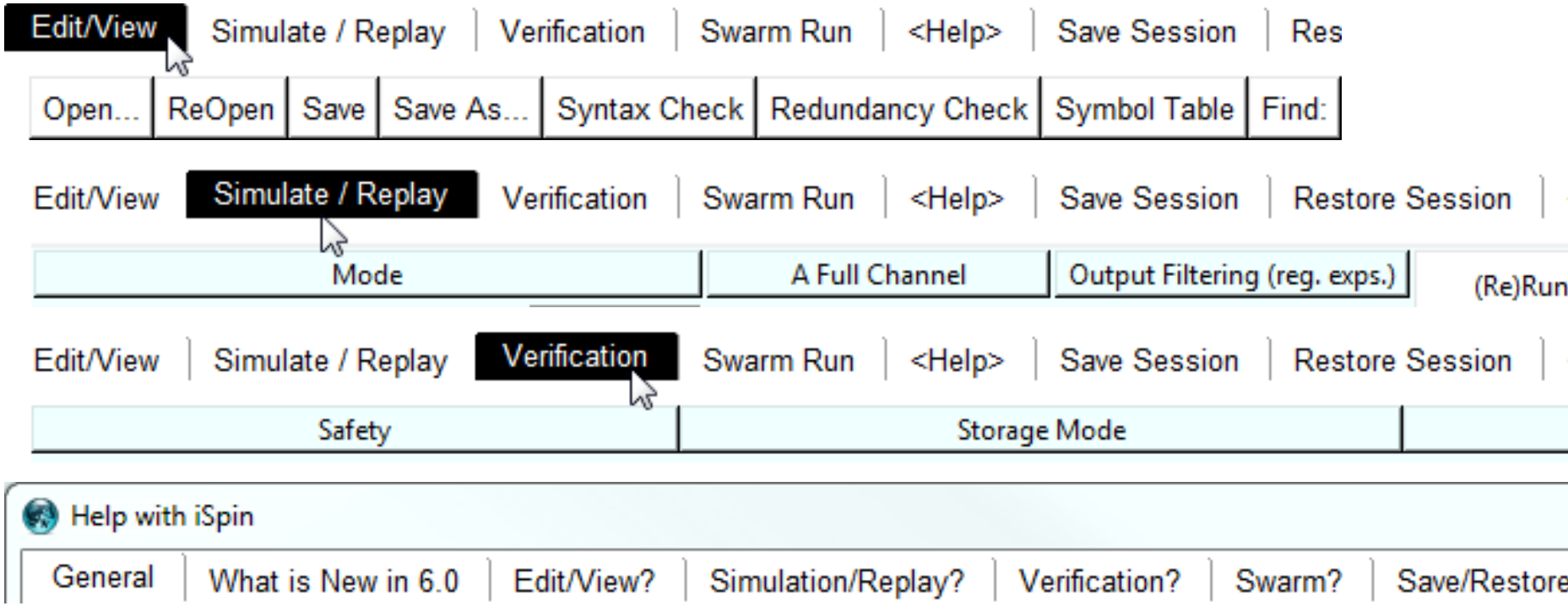


ispin ist etwas „eigentümliche“ Oberfläche zur Steuerung von SPIN (JSpin nicht besser). Alternativ kann SPIN als Kommandozeilen-Werkzeug genutzt werden!

Fenster zur Bearbeitung von Spezifikationen (immer Save vor weiteren Schritten)

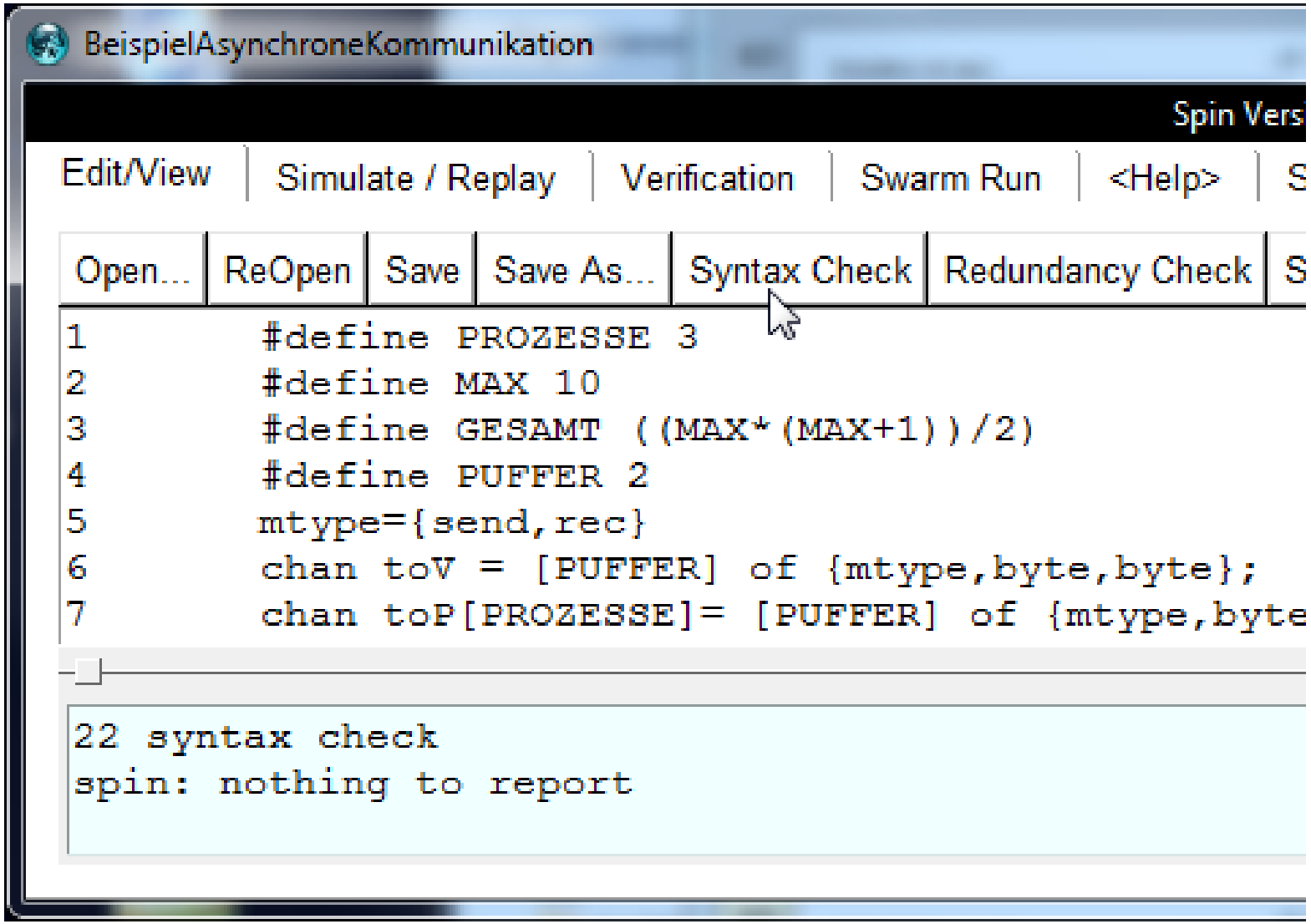
Aktionen, die mit SPIN zuletzt ausgeführt wurden

ispin-Menüs

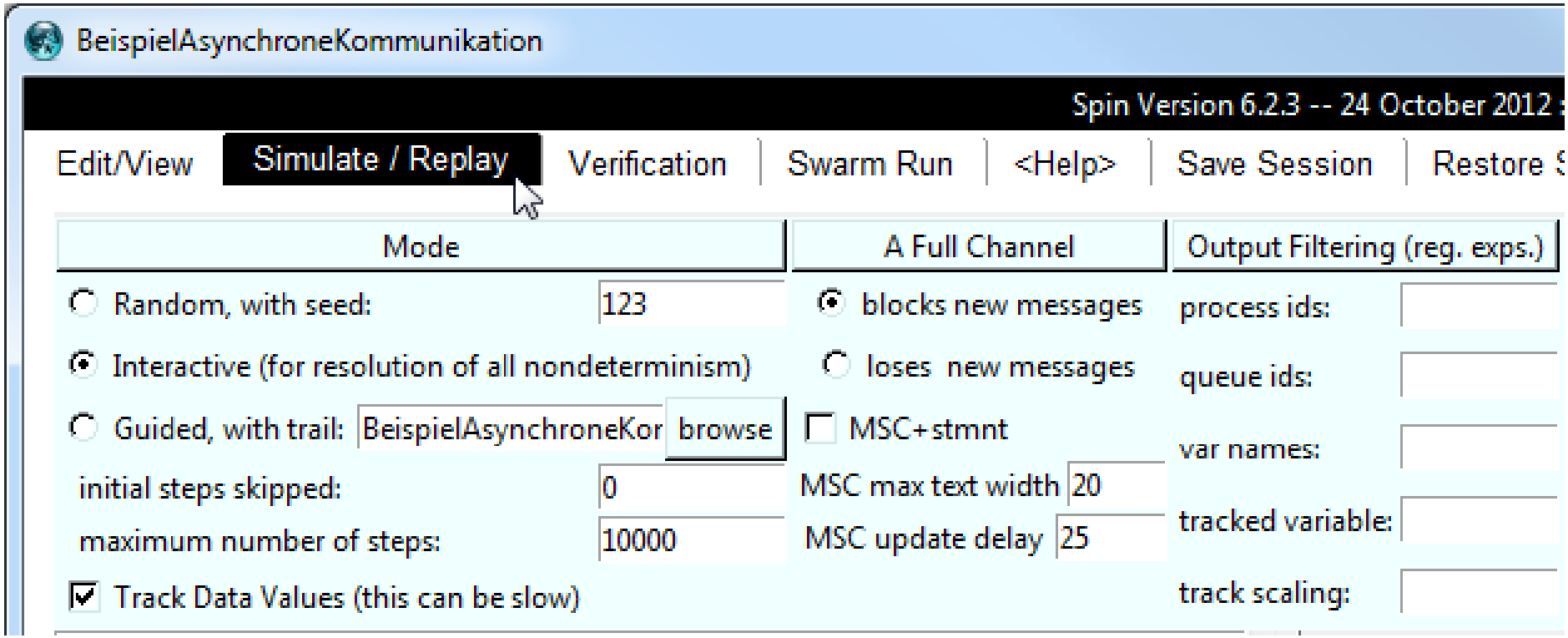


- Zeilennummern werden nach „Save“ angeordnet
- Syntax Check muss explizit aufgerufen werden

Syntax-Prüfung in ispin

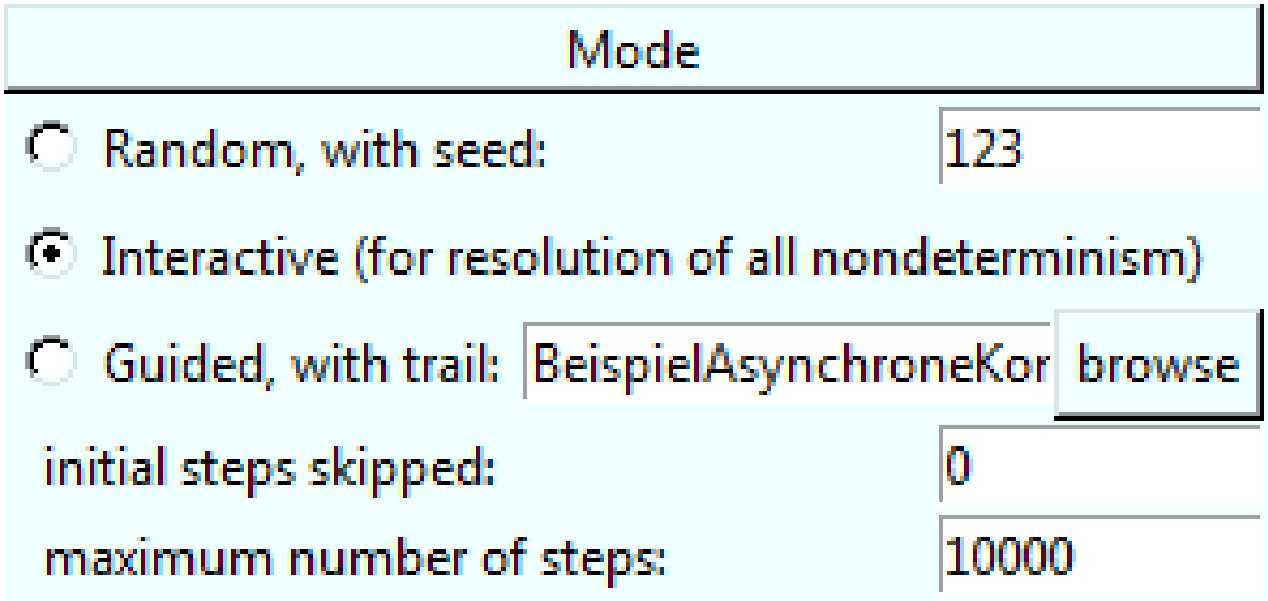


Simulationseinstellungen



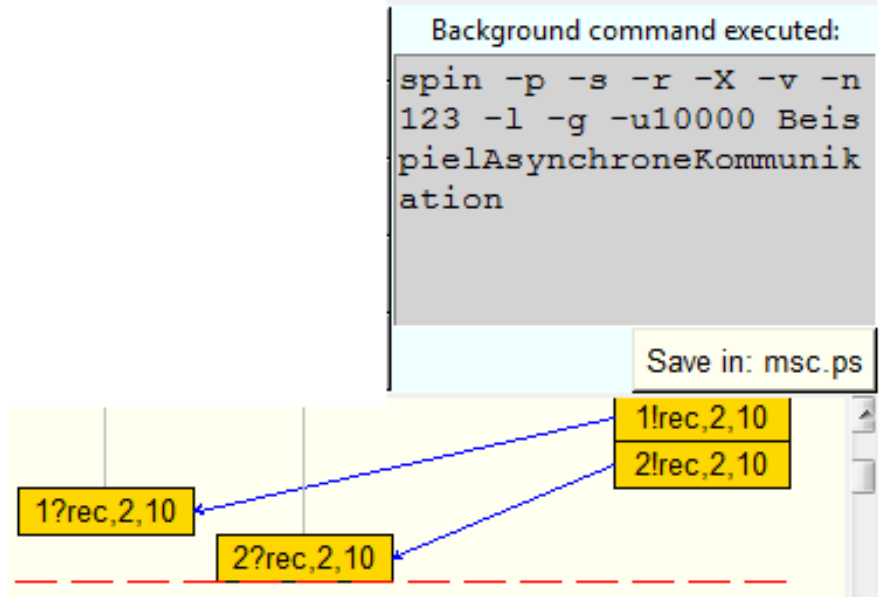
Simulationsarten

- Random: Wenn ausgewählt werden kann, welcher Schritt als nächstes möglich ist, wird diese Auswahl vom Simulator getroffen (seed steuert Zufallszahlen-generator)



  - Guided: nur sinnvoll, wenn in einer Datei abgespeichert ist, welche Schritte ausgeführt werden sollen, beim gescheiterten Model Checking gibt es für den Fehlerfall eine Ausführungssequenz
  - Interactive: Bei jeder Auswahlmöglichkeit, welcher Schritt als nächstes durchgeführt werden soll, wird der Nutzer befragt

Simulation



Variablenwerte ausgeführte Schritte Queues

```
[variable values, step  18]

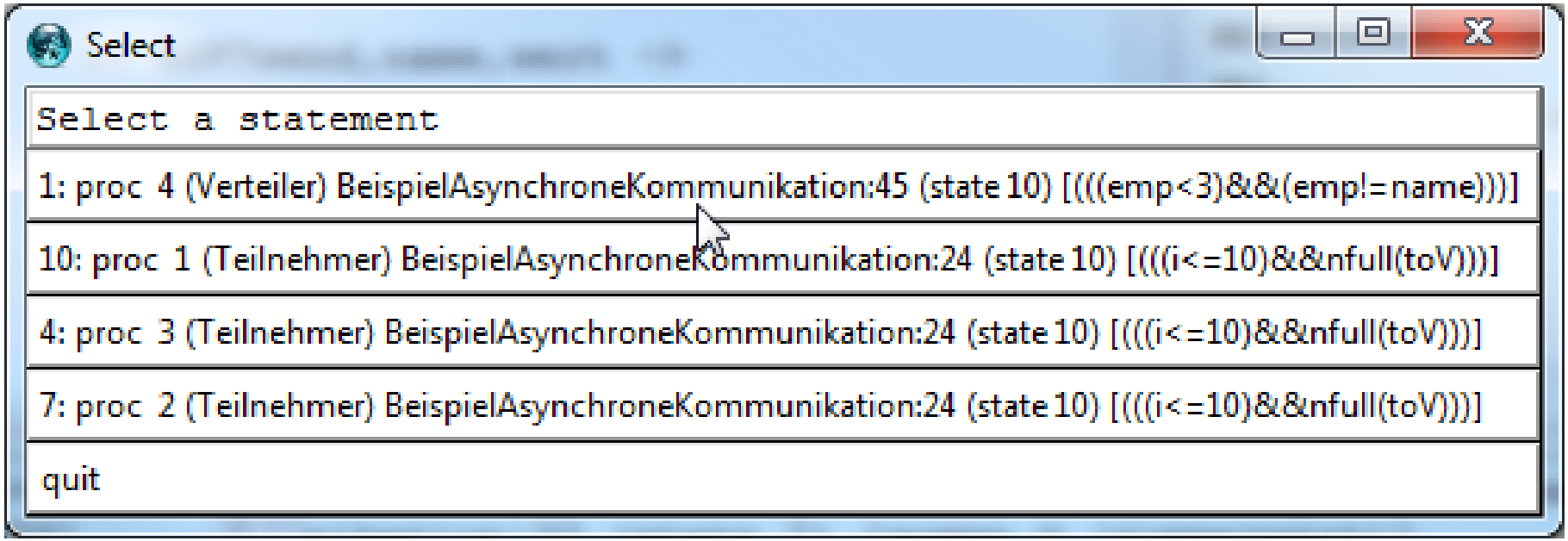
 :init:(0):i  =  3
 Teilnehmer(2):i  =  2
 Verteiler(4):emp  =  0
 Verteiler(4):name  =  1
 Verteiler(4):wert  =  1
```

```
Selected: 5
 14:    proc  2 (Tei
lnehmer) BeispielAsy
nchroneKommunikation
:24 (state 10) [(((i
<=10)&&nfull(toV)))]
 15:    proc  2 (Tei
```

```
[queues, step  17]

q 1  :: (toV):
```

Interaktive Simulation

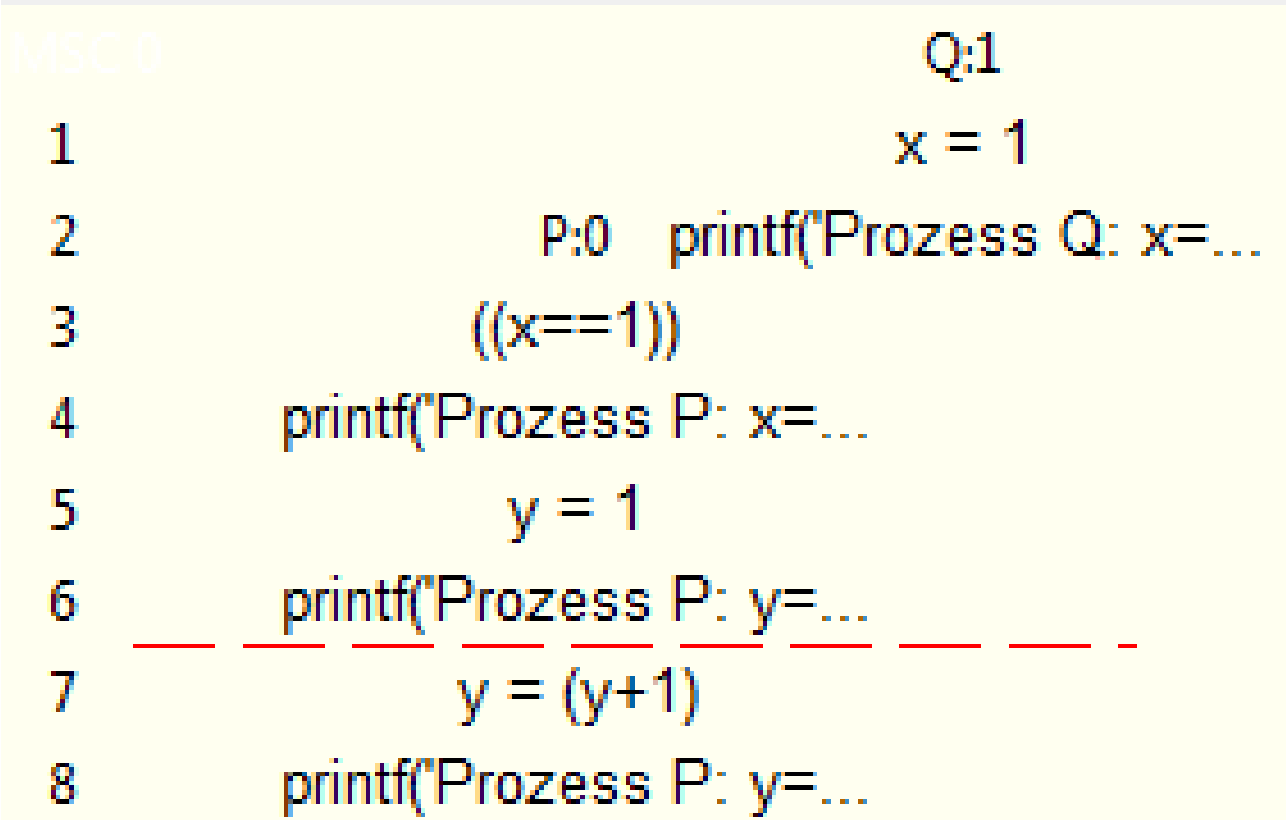


Analyse von atomic (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
int x = 0;
int y = 0;
active proctype P(){
  atomic{
    x == 1;
    printf("Prozess P: x==1\n");
    y = 1;
    printf("Prozess P: y=1\n");
    y = y + 1;
    printf("Prozess P: y=y+1\n")
  }
}
active proctype Q(){
  atomic{
    x = 1;
    printf("Prozess Q: x=1\n");
    y == 1;
    printf("Prozess Q: y==1\n");
    x = x + 1;
    printf("Prozess Q: x=x+1\n")
  }
}
```

Analyse von atomic (2/2)



- Spezifikation hat keinen alternativen Ablauf
- aber, atomic unterbrochen, wenn es nicht weiter geht

4.4 Einfache Verifikationsmöglichkeiten

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

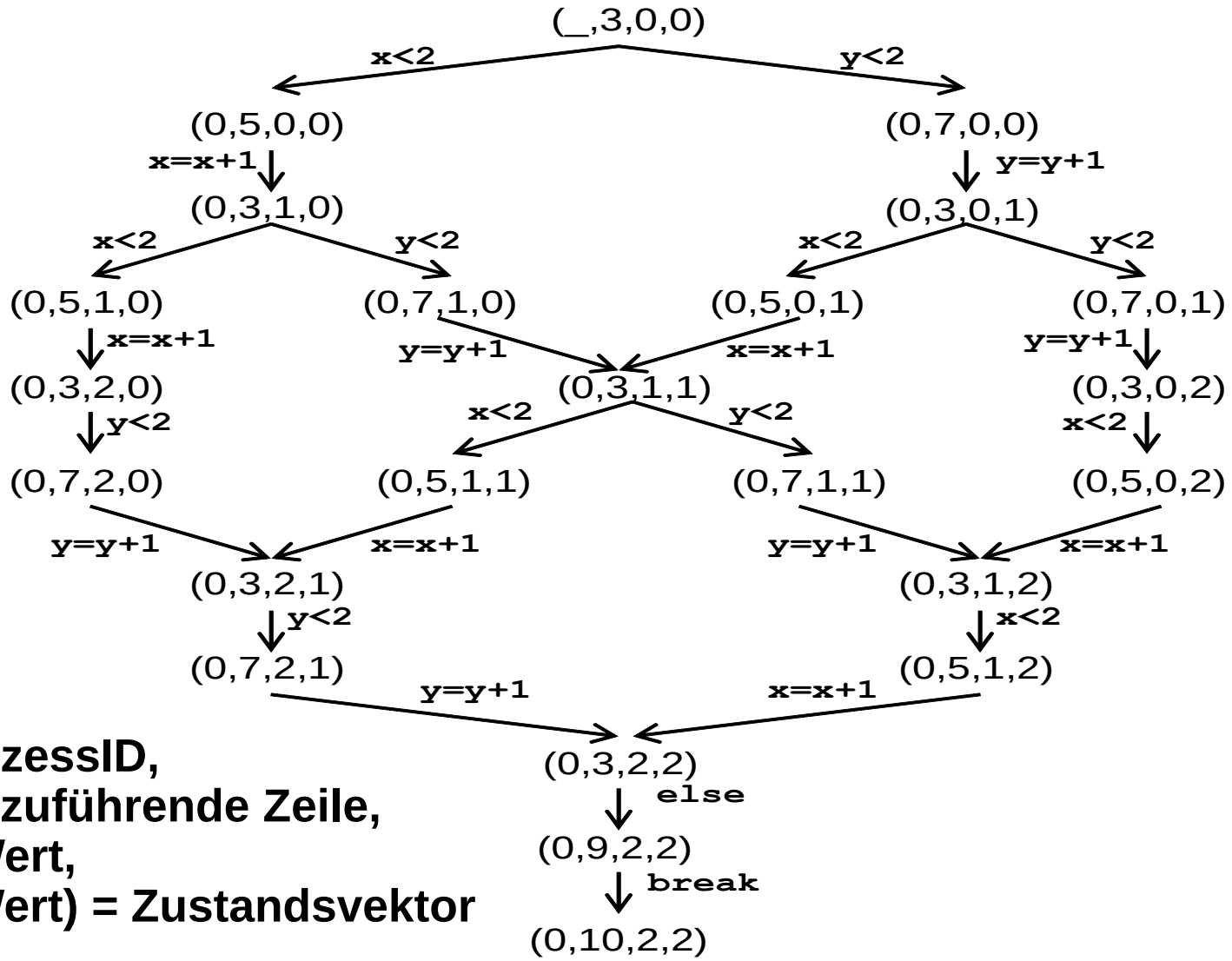
- 4.4.1 Grundideen des Model Checkings
- 4.4.2 Model Checking sehr großer Systeme
- 4.4.3 Lebendigkeit und Sicherheit
- 4.4.4 Zusicherungen
- 4.4.5 Prozessterminierung
- 4.4.6 Ausführung einfacher Verifikationen
- 4.4.7 Nachweis von Lebendigkeitseigenschaften

Verifikationsansatz

- SPIN berechnet alle möglichen Zustände, dabei werden Zustandseigenschaften bei Erstellung geprüft
- SPIN muss Zustandswiederholungen erkennen
- Beispielspezifikation:

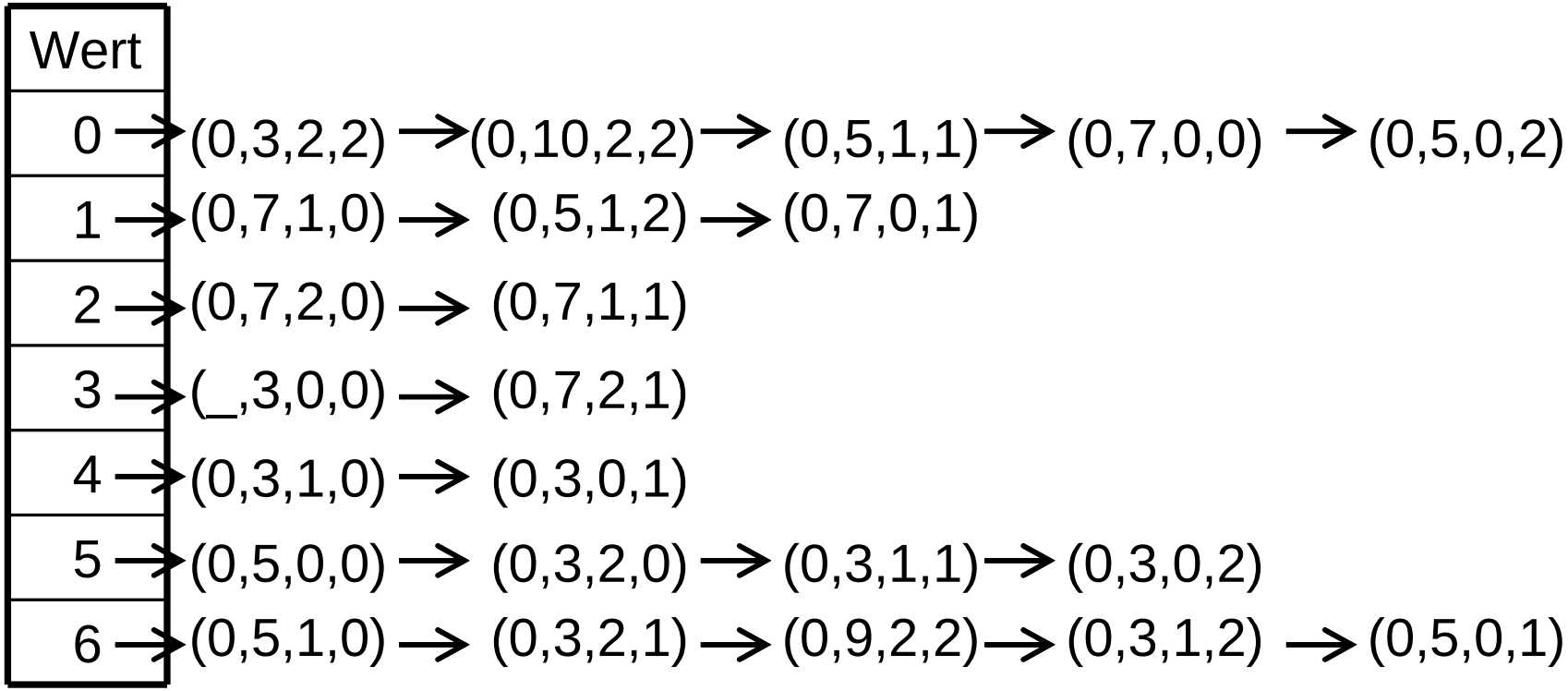
```
byte x = 0;                          /* 1 */
byte y = 0;                          /* 2 */
active proctype Mini(){
  do                                 /* 3 */
  :: x < 2 ->                        /* 4 */
     x = x + 1;                      /* 5 */
  :: y < 2 ->                        /* 6 */
     y = y + 1;                      /* 7 */
  :: else ->                         /* 8 */
     break;                          /* 9 */
  od;
}                                    /* 10 */
```

Zustandsraum



(ProzessID,
auszuführende Zeile,
x-Wert,
y-Wert) = Zustandsvektor

Verwaltung der Zustände mit Hash-Tabellen



- schnelle Hashfunktion zur Berechnung der Position in Hash-Tabelle, Hashfunktion: (Folgezeile+x+y) % 7
- wenn Tabelleneintrag belegt, prüfen ob gleich
- wenn nicht, gleich einfügen (anhängen), sonst gleicher Zustand gefunden

Zentrale Aufgabe: kleines Modell (1/2)

- es ist Aufgabe des Spezifizierers, das Modell möglichst klein und trotzdem realistisch zu halten
- möglichst klein:
  - wenig Prozesse
  - kleinst-mögliche Datentypen
  - keine überflüssigen Informationen in Kommunikationsprotokollen
  - möglichst kleine Puffer
  - wenig Nichtdeterminismus: **`atomic, d_step`**

realistisch:

- Antworten des Modells auf Fragen müssen sich auf Realität übertragen lassen
- z. B. kann man Puffer nicht einfach vergrößern oder verkleinern

Zentrale Aufgabe: kleines Modell (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

In Spezifikation:

- ```
  chan c1 =[2] of {byte,short};
  chan c2 =[2] of {short,bool};
  active proctype P(){
    xr c1;        // nur P liest  aus c1
    xs c2; ...  // nur P schreibt auf c2
  ```
  geht nur, wenn c1 und c2 keine Arrays sind
- `atomic{}`: Bei Blockade kann Kontrolle an anderen Prozess übergehen, wenn Kontrolle zurück, dann auch Rest atomar
- `d_step{}`: Spezifizierer garantiert, dass innerer Ablauf deterministisch
- alles deterministisch machen, was für Analyse unwichtig
- evtl. pro Aufgabenstellung neues Modell
  ```

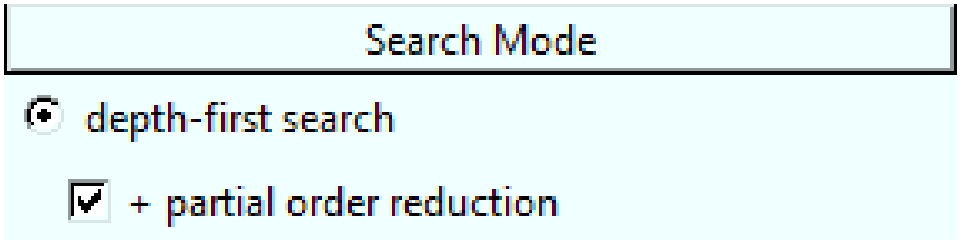
Optimierungen und Kompromisse

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

4.4.2

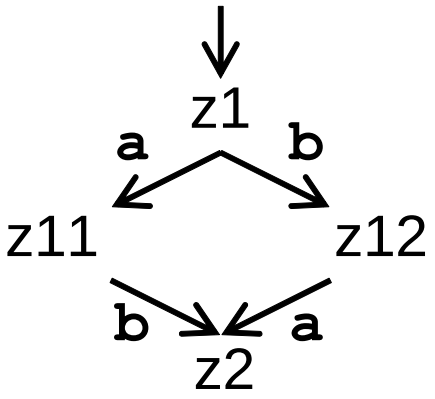
- Die Verifikation großer Modelle kann sehr lange dauern und mit unbefriedigenden Ergebnissen enden:
  - Speicherüberlauf
  - maximale Suchtiefe überschritten
- Lösungsansätze
  - weitere Modelloptimierungen, wenn möglich
  - Reduktion der Zustände mit bestimmten Regeln (partial order reduction)
  - Reduktion auf den Wunsch, möglichst viele Zustände zu untersuchen (gefundene Fehler sind echte Fehler, keine gefundenen Fehler keine Garantie)

Partial Order Reduction




```
proctype P1(){
   ...
   a;
   ...
}
```

```
proctype P2(){
   ...
   b;
   ...
}
```



- unabhängige Teilspezifikationen führen zu Rauten im Zustandsraum
- links und rechts zusammen nur interessant, wenn weitere Zustände abgehen oder nach Detailinformationen gesucht wird
- Ansatz: wenn uninteressant, lasse einen Zustand weg
- geht immer, wenn nur Endzustände interessant
- kritisch, wenn z. B. niemals $x>3 \wedge y<4$ gefordert

Beispiel: Partial Order Reduction

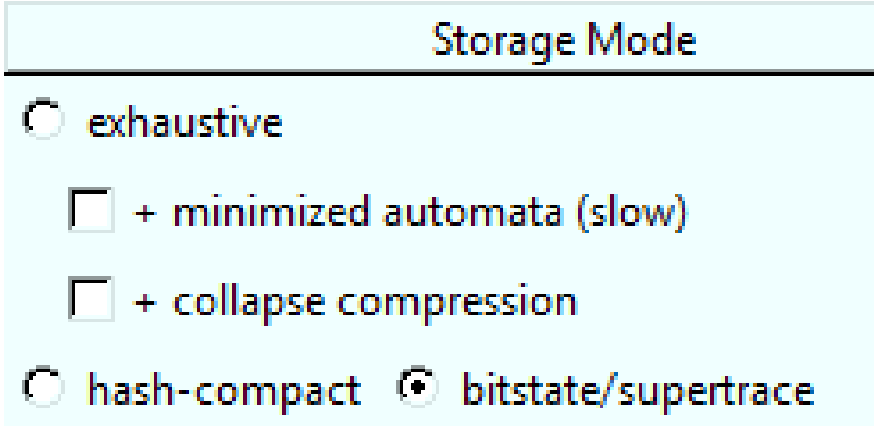
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
#define PROZESSE 2
#define GRENZE 10
byte x[PROZESSE];
active [PROZESSE] proctype P(){
  x[_pid] = 0;
  do
  :: x[_pid] < GRENZE -> x[_pid] = x[_pid] + 1;
  :: else -> break;
  od;
}
```

- Erinnerung: `_pid` steht für Systemvariable mit eindeutigem Prozessidentifikator
- ohne Partial Order Reduction: 553 stored + 506 matched
- mit Partial Order Reduction: 553 stored + 484 matched

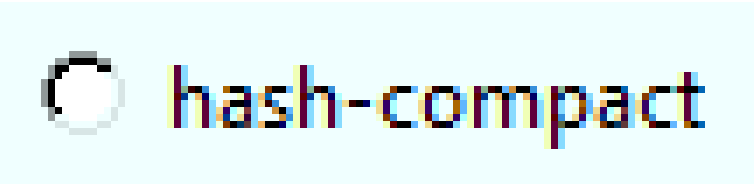
Bitstate-Hashing

- Standard-Hashfunktion
  speichert für jeden Zustand den Zustandsvektor (einige Bytes)
- Alternativ: Hashfunktion
  speichert nur ein Bit
- Annahme: Ist Bit gesetzt, wurde Zustand vorher besucht



- Annahme stimmt nicht immer, da Hash-Funktion unterschiedliche Zustände auf gleichen Wert abbilden kann
- Raum für Hashwerte sehr sehr groß gewählt, so die Gefahr reduzieren (mit Wahrscheinlichkeiten analysierbar)
- SPIN-Ansatz: Nutzung mehrerer unterschiedlicher Hashfunktionen (Standard:3) und mehrerer Bits (= Anzahl Hashfunktionen)

Variante



- Kleines Beispiel hat bereits Zustandsvektorgröße 12 Byte (typisch > 100 Byte)
- Hash-Variante nach Wolper: Nimm Hashfunktion, die sehr großen Zahlenbereich abdeckt, z.B. 0 – 2^{64}-1, Ergebnis benötigt 8 Byte
- Diese Zahl wird in echter Hashtabelle gespeichert
- Durch sehr großen Zahlenraum der ersten Hash-Funktion ist Wahrscheinlichkeit der unerwünschten Kollisionen sehr gering

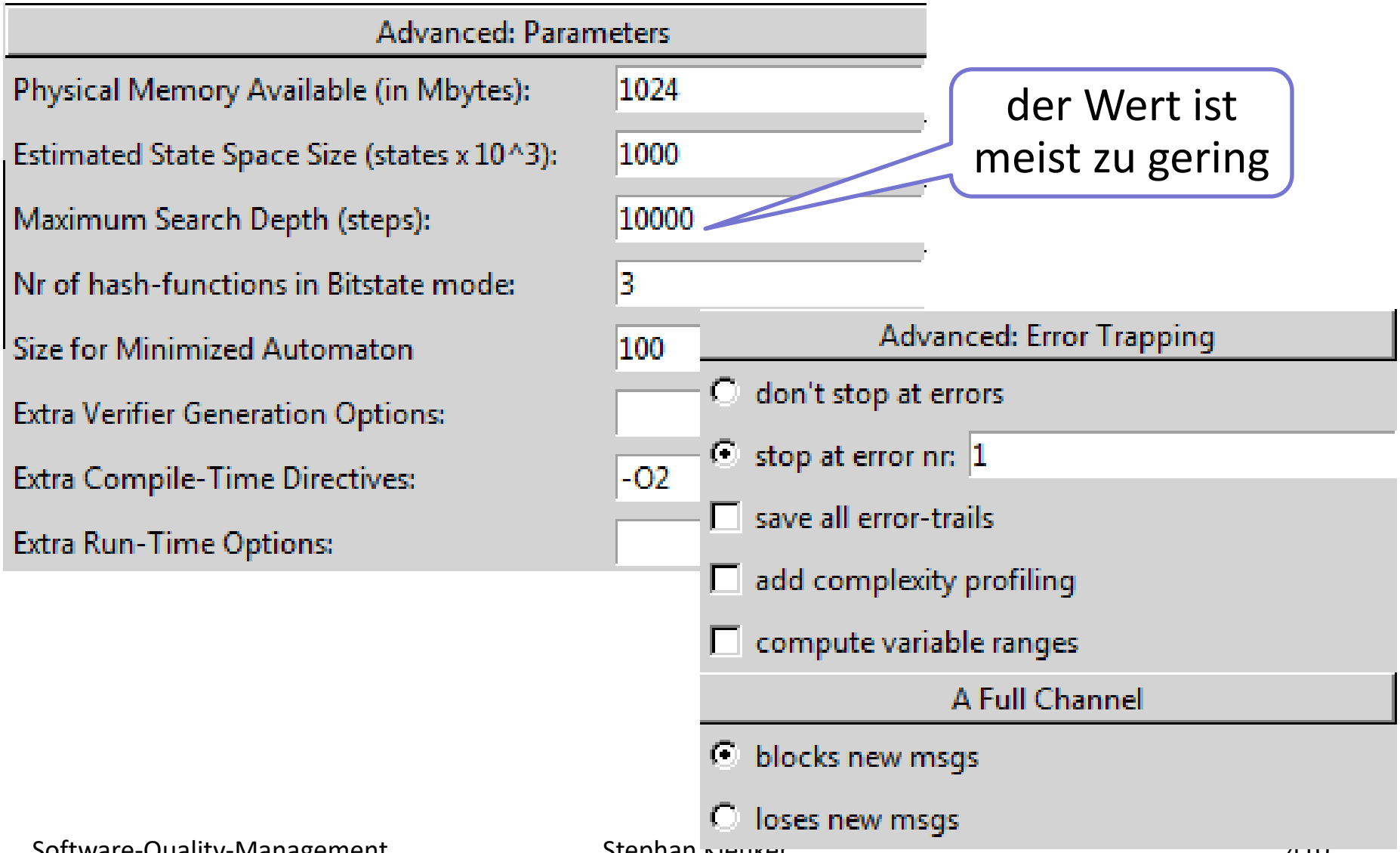
Speicheroptimierung dann Näherung (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

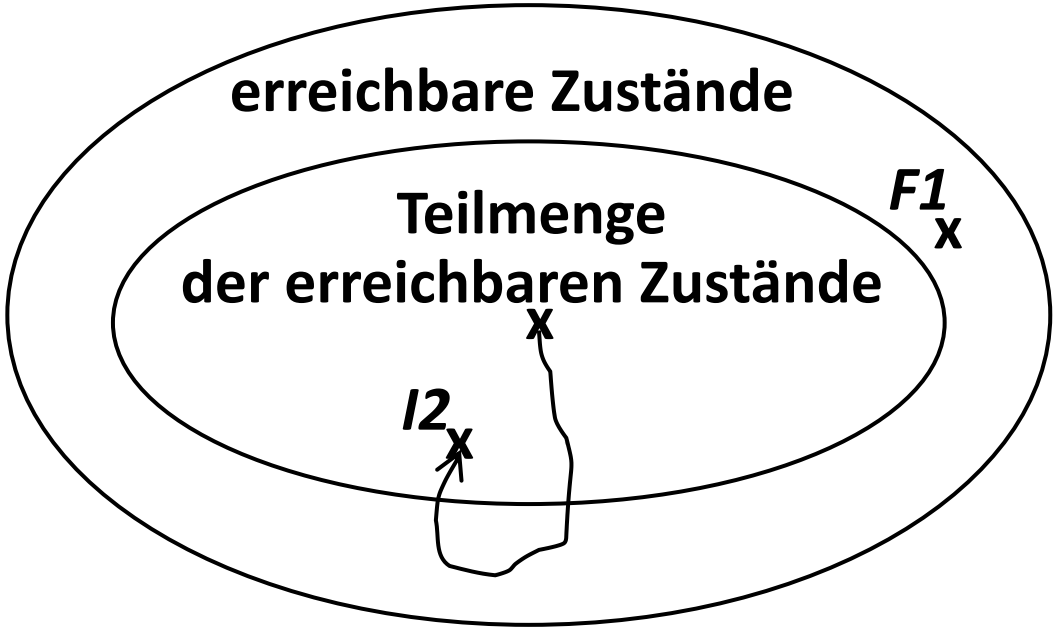
- Die fortgeschrittenen Parameter sind bei Problemen mit dem Speicher schrittweise zu optimieren (kann zeitaufwändig sein)
- zuerst: Speicheroptimierung durch Komprimierung (viel langsamer, aber vollständige Prüfung)
- wenn das nicht geht: eine der vorgestellten Annäherungen an die vollständige Durchsuchung nutzen

```
pan: out of memory
       3.73208e+08 bytes used
       102400 bytes more needed
       3.73293e+08 bytes limit
hint: to reduce memory, recompile with
  -DCOLLAPSE # good, fast compression, or
  -DMA=72   # better/slower compression, or
  -DHC # hash-compaction, approximation
  -DBITSTATE # supertrace, approximation
```

Speicheroptimierung dann Näherung (2/2)

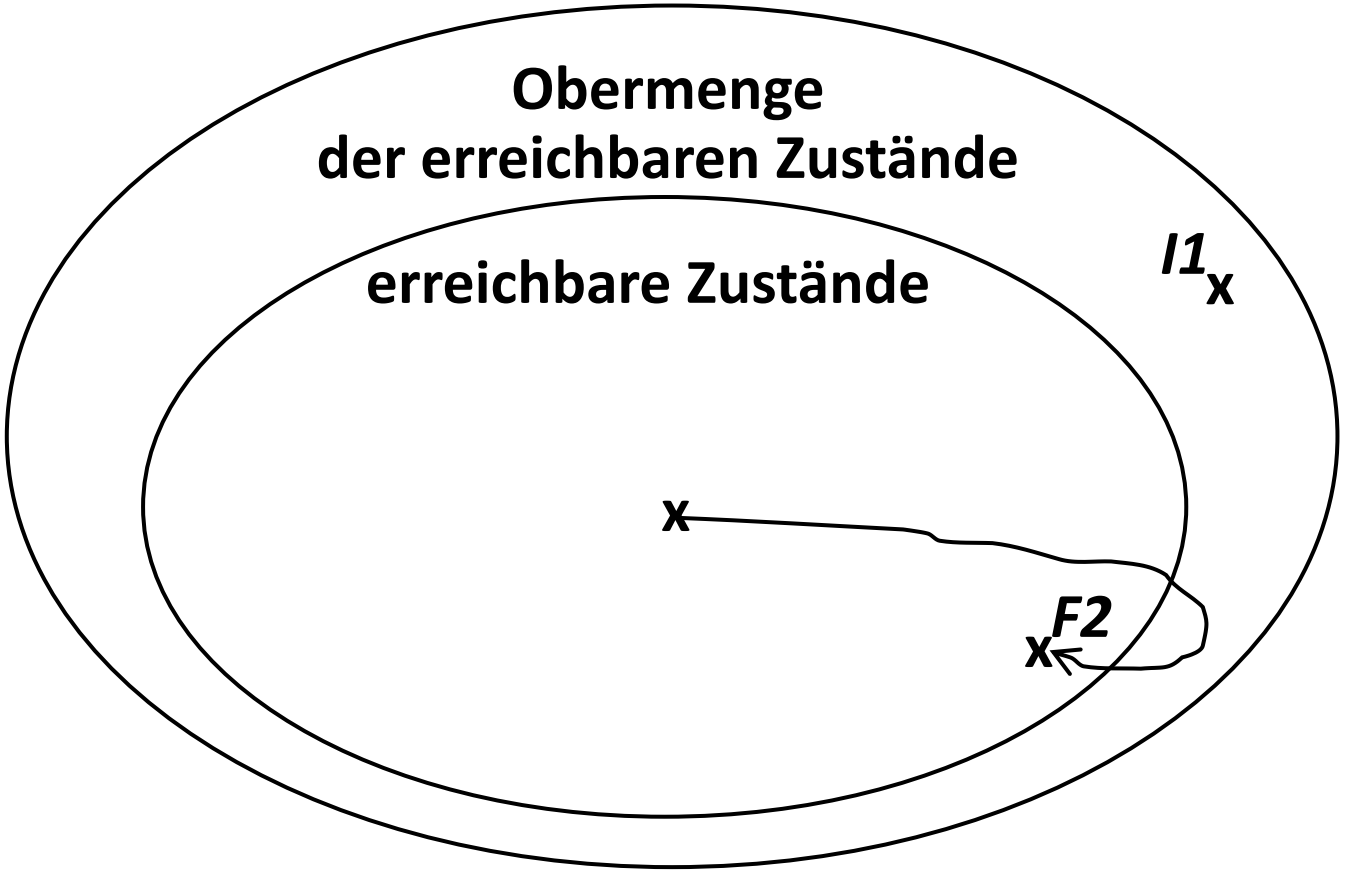


Überblick: Zustandsraumannäherung (1/3)



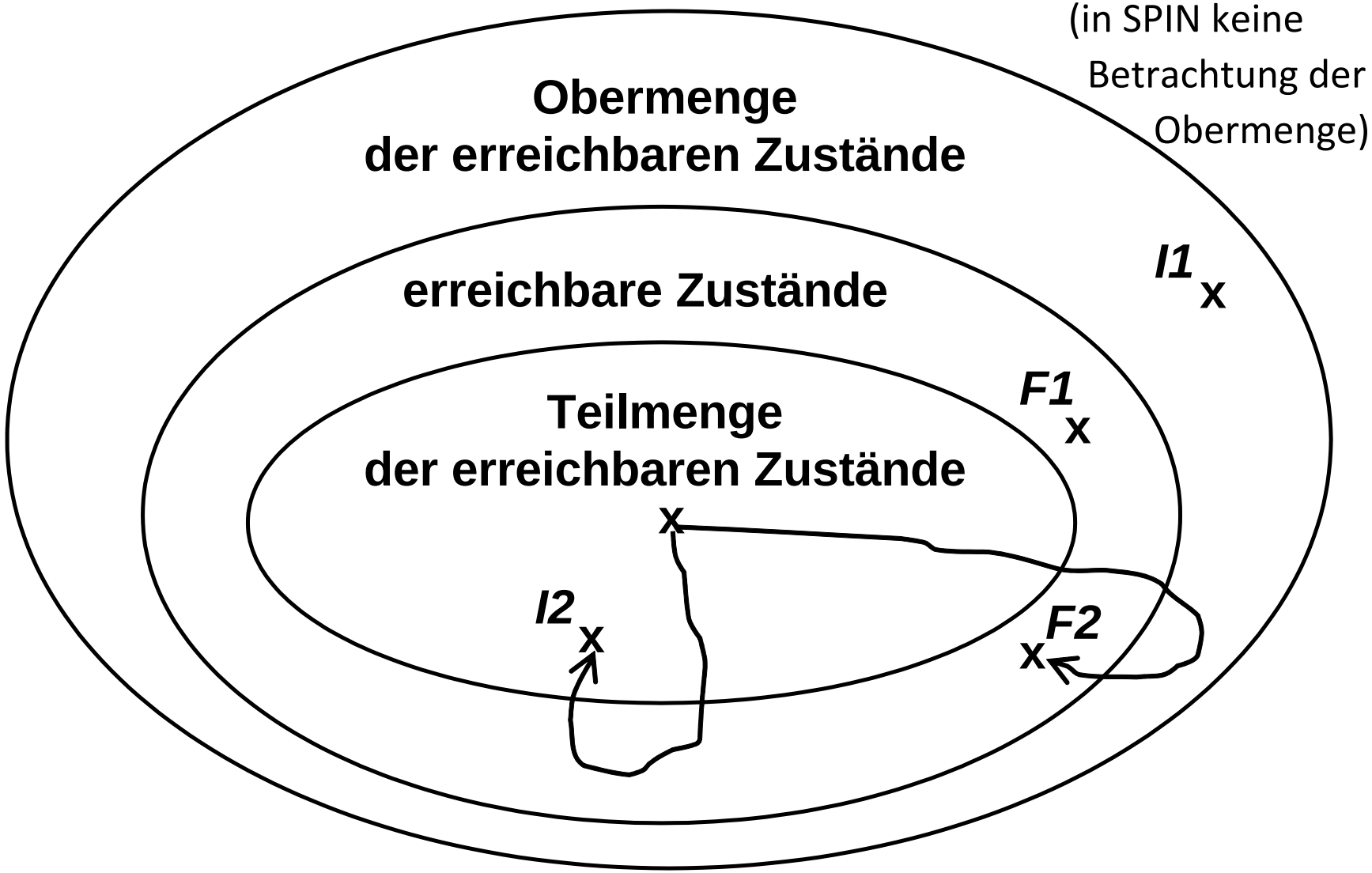
- Fehler F1 nicht gefunden, da nicht berechnet
- gewünschter Pfad zu I2 nicht gefunden, da Pfad nicht mehr nutzbar

Überblick: Zustandsraumannäherung (2/3)



- Pfad zu F2 wäre korrekt, da jetzt möglich
- Fehler I1 „gefunden“, da jetzt erreichbar

Überblick: Zustandsraumannäherung (3/3)



Arten von funktionalen Anforderungen

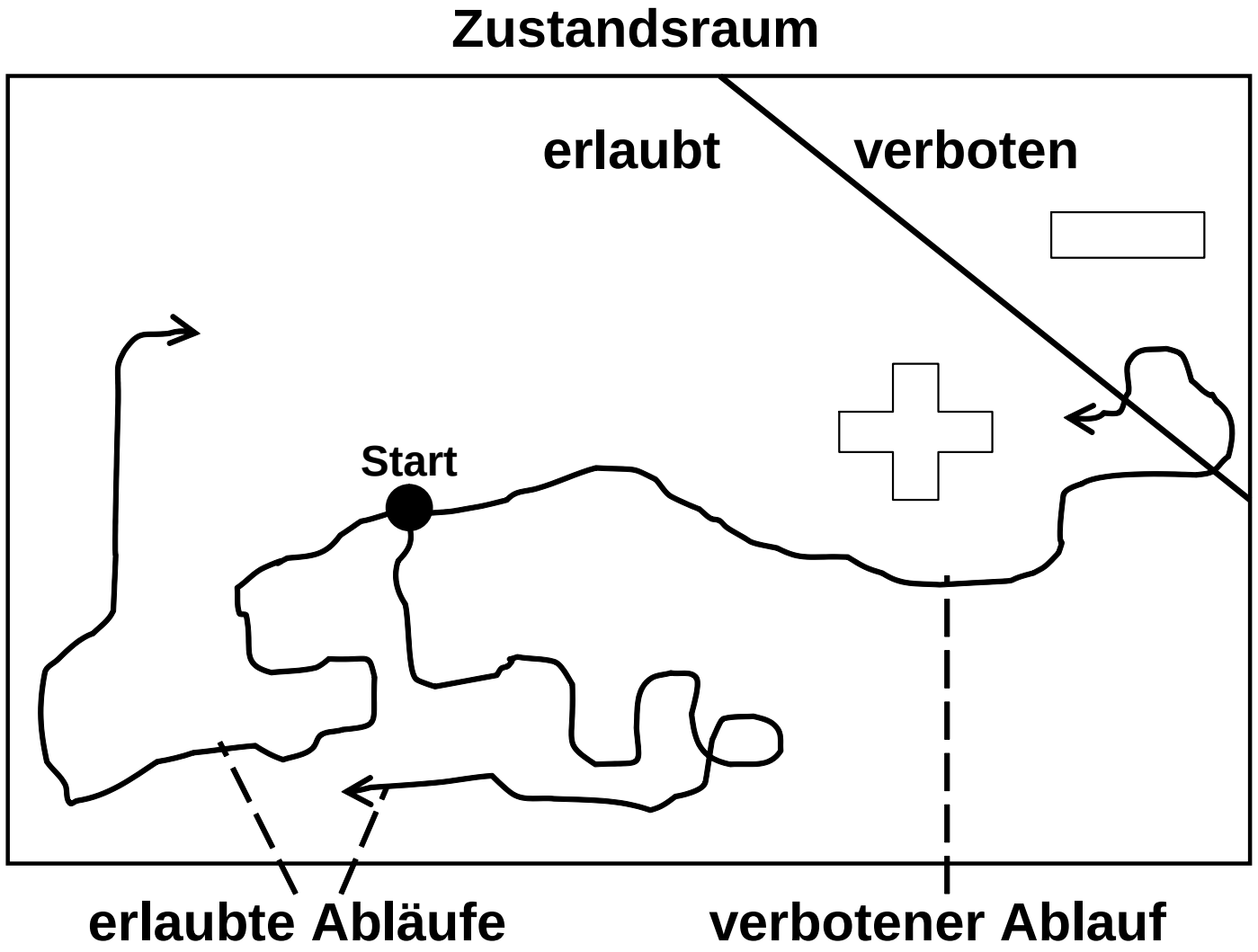
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

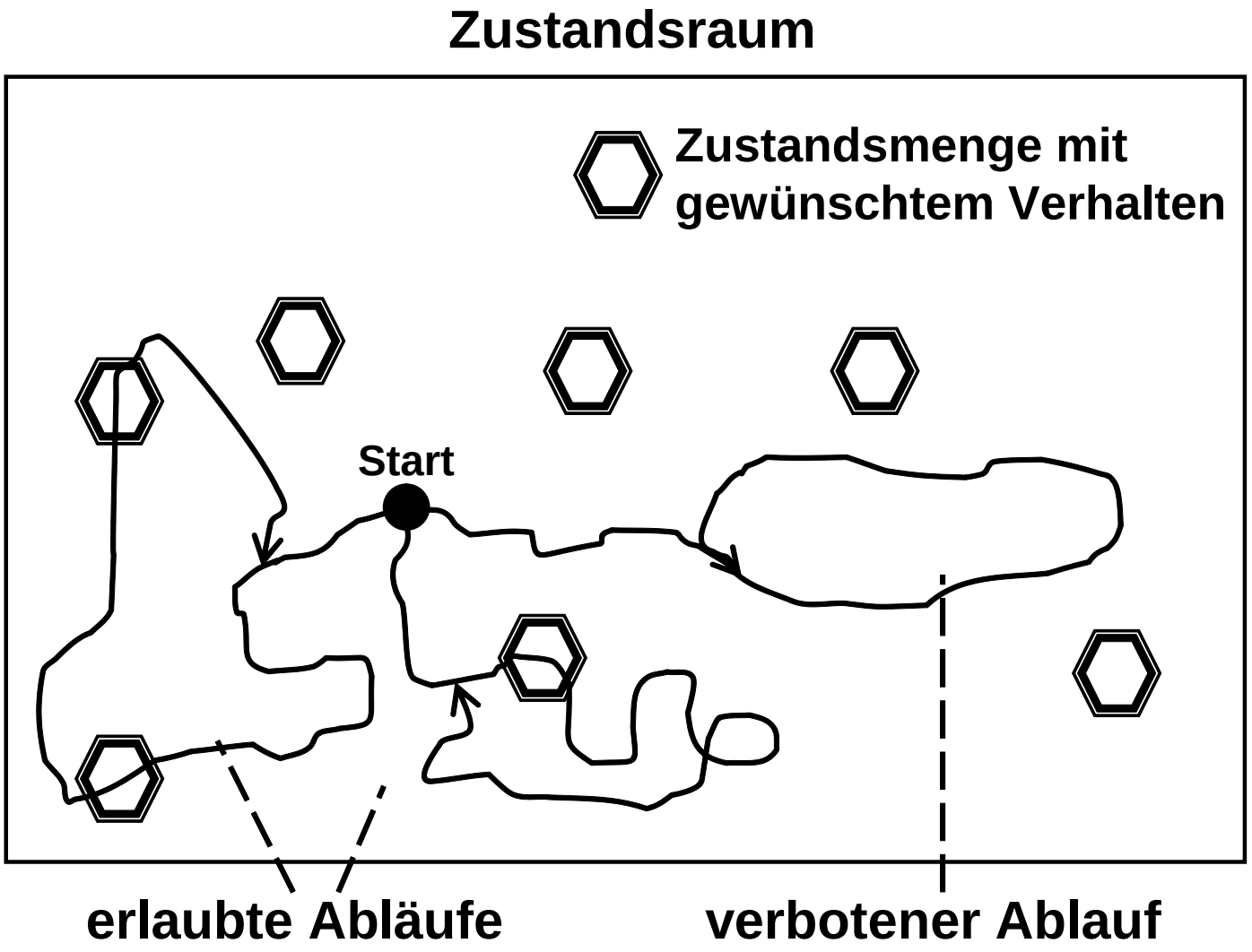
4.4.3

- Sicherheitseigenschaften (Safety): Dies sind Eigenschaften, die das System in bestimmten Zuständen erfüllen muss. Anschaulich wird so zugesichert, dass „nichts Schlechtes“ passiert. Zur Überprüfung muss für jeden relevanten Zustand überprüft werden, dass er die gewünschte Eigenschaft hat.
- Lebendigkeitseigenschaften (Liveness): Dies sind Forderungen an das System, dass unter bestimmten Bedingungen diese Eigenschaft erreicht werden muss. Anschaulich wird so zugesichert, dass „etwas Gutes“ passieren wird. Zur Überprüfung müssen für jeden relevanten Zustand alle möglichen Folgeabläufe analysiert werden.

Visualisierung von Sicherheit



Visualisierung von Lebendigkeit



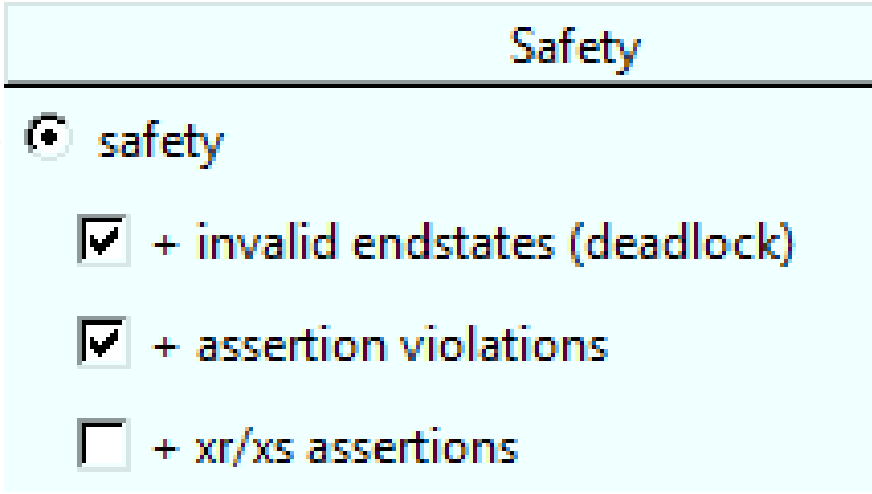
Zusicherungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

4.4.4

- Als Sicherheitseigenschaften können, wie aus Java und C bekannt, mit `assert` Zusicherungen eingebaut werden
- `assert(`<Boolescher Ausdruck>`)` ist immer ausführbar, meldet Fehler bei Auswertung nach false
- Möglichkeit, assert zur Prüfung von Invarianten zu nutzen:

```
byte y=0
bool b=false;
active proctype Invariante()
  assert( !b ||  y>42);
}
```



Maximumberechnung (1/2)

```
active proctype Max(){
  int x, y, z;
  if
  :: x = -1;
  :: x = 0;
  :: x = 1;
  fi;
  if
  :: y = -1;
  :: y = 0;
  :: y = 1;
  fi;
  if
  :: z = -1;
  :: z = 0;
  :: z = 1;
  fi;

  int max = 0;
  if
  :: x >= z -> max = x;
  :: else -> skip;
  fi;
  if
  :: y >= x -> max = y;
  :: else -> skip;
  fi;
  if
  :: z >= y -> max = z;
  :: else -> skip;
  fi;

  assert(max==x || max==y || max==z);
  assert(max>=x && max>=y && max>=z);
}
```

Maximumberechnung (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
[variable values,
step   7]

 Max(0):max  =  -1
 Max(0):x  =  0
 Max(0):y  =  -1
 Max(0):z  =  -1
```

```
using statement merging
  1:    proc  0 (Max:1) Max.pml:5 (state 2) [x = 0]
  2:    proc  0 (Max:1) Max.pml:9 (state 6) [y = -(1)]
  3:    proc  0 (Max:1) Max.pml:14 (state 11) [z = -(1)]
  3:    proc  0 (Max:1) Max.pml:20 (state 16) [max = 0]
  4:    proc  0 (Max:1) Max.pml:21 (state 17) [((x>=z))]
  4:    proc  0 (Max:1) Max.pml:21 (state 18) [max = x]
  5:    proc  0 (Max:1) Max.pml:26 (state 25) [else]
  6:    proc  0 (Max:1) Max.pml:26 (state 26) [(1)]
  7:    proc  0 (Max:1) Max.pml:29 (state 29) [((z>=y))]
  7:    proc  0 (Max:1) Max.pml:29 (state 30) [max = z]
  7:    proc  0 (Max:1) Max.pml:33 (state 35) [assert(((
(max==x)||(max==y))||(max==z)))]
spin: Max.pml:34, Error: assertion violated
spin: text of failed assertion: assert(((max>=x)&&(max>
=y))&&(max>=z)))
#processes: 1
  7:    proc  0 (Max:1) Max.pml:34 (state 36)
1 processes created
Exit-Status 0
```

Ausnutzung von Markierungen

- Zugriff auf Markierungen Prozess@Marke

```
/* Verifikation scheitert */
byte x=0;

active[3] proctype P(){
m1: do
    :: x<10 ->
       m2: x++
    ::  x>5 ->
       m3: break
    od
}

active proctype  Inv(){
  P[0]@m3;
  P[1]@m3;
  P[2]@m3;
  assert(x<=11)
}
```

auch P@m3 möglich (wahr wenn ein P-Prozess an Markierung m3 steht)

steht vorher
`active[4] proctype P(){`
dann muss in eckigen Klammern Prozessnummer stehen: P[4]@m3 für ersten P-Prozess

Anmerkung: Inv muss nicht termineren

Lesender Zugriff auf lokale Variablen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
bool dran = false

active proctype P(){
  byte x = 0;
  do
  :: x < 10 && dran ->
     x = x + 1;
     dran = !dran
  :: x >= 10 -> break
  od
}

active proctype Q(){
  byte x = 0;
  do
  :: x < 10 && ! dran ->
     x = x + 1;
     dran = !dran
  :: x >= 10 -> break
  od
}

active proctype Inv(){
  assert (Q:x - P:x >= 0 && Q:x - P:x <= 1)
}

//Anmerkung: wenn möglich, solche Zugriffe vermeiden
```

Erfolgreiche Terminierung

- SPIN geht davon aus, dass in einem Zustand, aus dem keine weiteren Schritte mehr möglich sind, ein Prozess nicht ordentlich terminiert ist
- so werden z.B. Deadlocks gefunden

- Idee klappt z. B. nicht bei Server-Prozessen, die haben Standard-Wartezustand, in dem sie auf Aufträge von Clienten warten
- In PROMELA können Zeilen, die sinnvolle Endzustände darstellen, mit einer „end“-Markierung versehen werden
- Markierungen innerhalb eines Prozesses müssen unterschiedlich heißen, „end“-Markierungen müssen nur mit ‚e‘ ‚n‘ ‚d‘ beginnen

```
        emp=0;
endOK:  do
        :: emp<prozesse && emp!=name ->
           toP[emp]!rec, name, wert;
```

Beispiel: Kommunikationsprotokoll (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

P1 schickt Informationen an P2, P2 kann diese bestätigen oder (zur Prüfung) an P3 schicken, dann bestätigt P3 an P2 und P2 an P1

```
mtype={send1, send2, ack1, ack2};
chan c12 = [0] of {mtype, byte};
chan c21 = [0] of {mtype, byte};
chan c23= [0] of {mtype, byte};
chan c32= [0] of {mtype, byte};
active proctype P1(){
  byte i = 1;
  do
  :: i <= 10 ->
     c12!send1, i;
     c21?ack1, _;
     i = i + 1
  od;
}
```

Beispiel: Kommunikationsprotokoll (2/2)

```
active proctype P2(){
  byte x;
  end : do
  :: c12?send1, x ->
     if
     :: x%2 == 1 -> c21!ack1, x
     :: x%2 == 0 ->
        c23!send2, x;
        c32?ack2, x;
        c21!ack1, x
     fi;
  od
}
active proctype P3(){
  byte x;
  end: do
  :: c23?send2, x ->
     c32!ack2, x
  od;
}
```

Verifikationseinstellungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

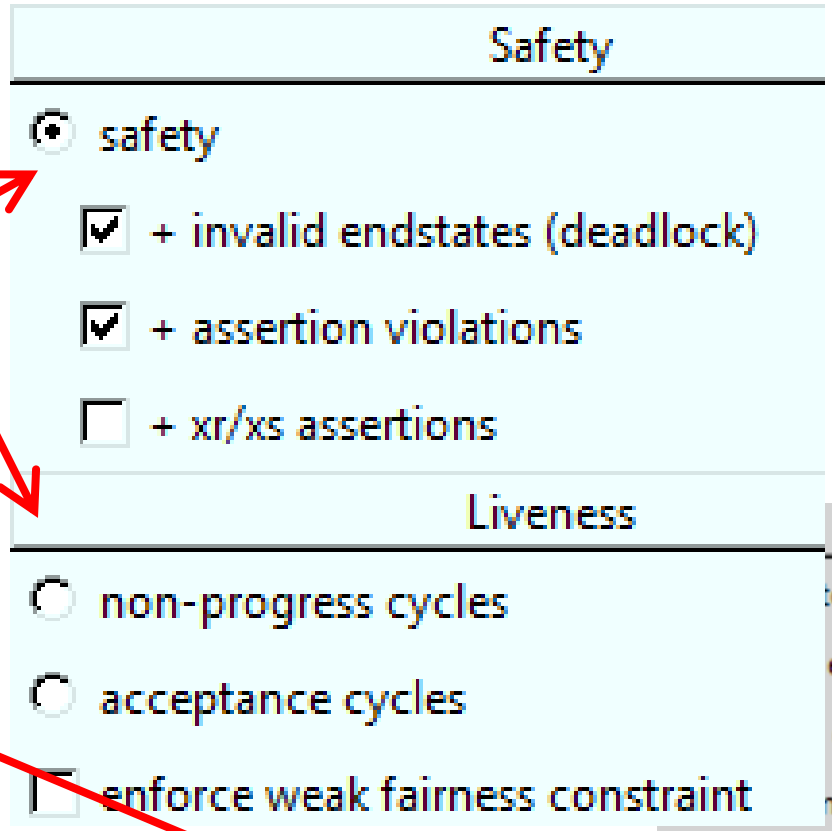
4.4.6

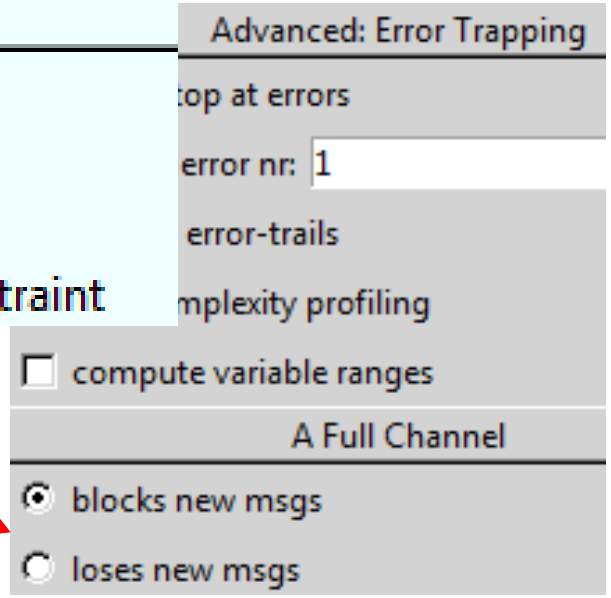
Sicherheits- und Lebendigkeits-anforderungen müssen getrennt geprüft werden

was bei Senden in volle Puffer?

weitere Einstellungen „Advanced“

(weitere Details später)





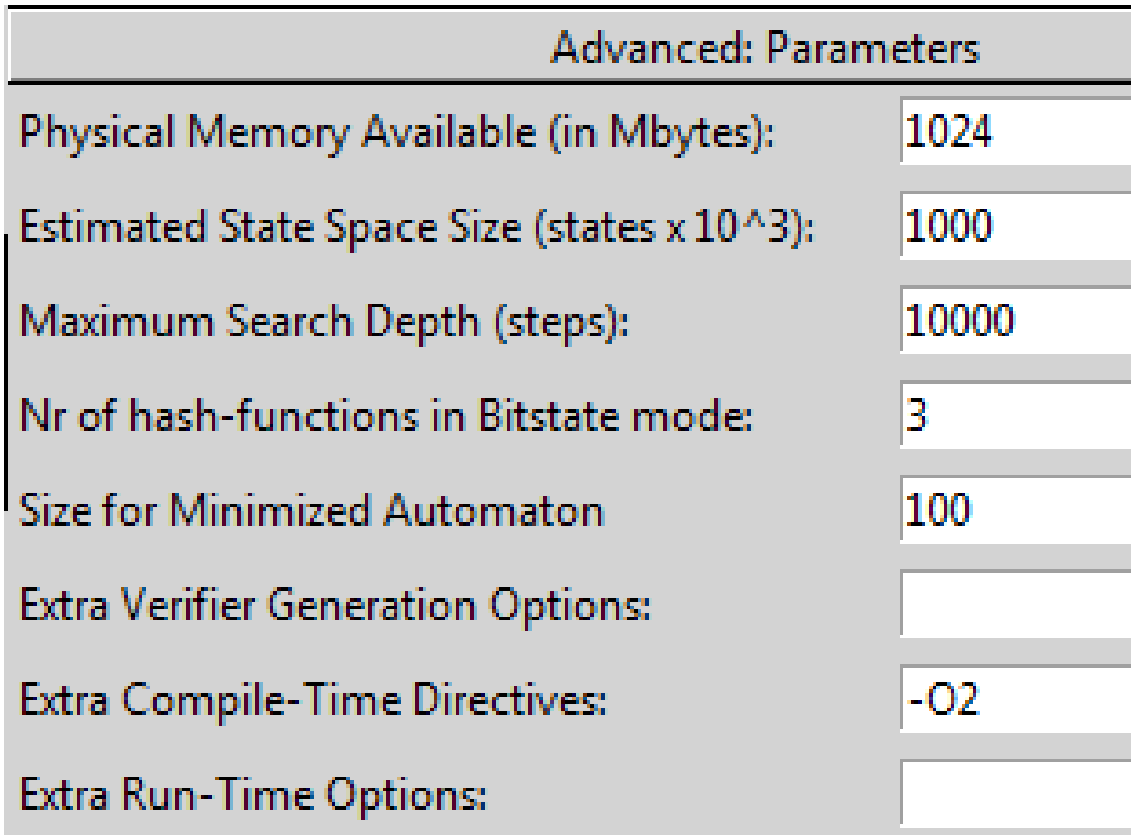
Detaileinstellungen der Verifikation

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Freier HW-Speicher

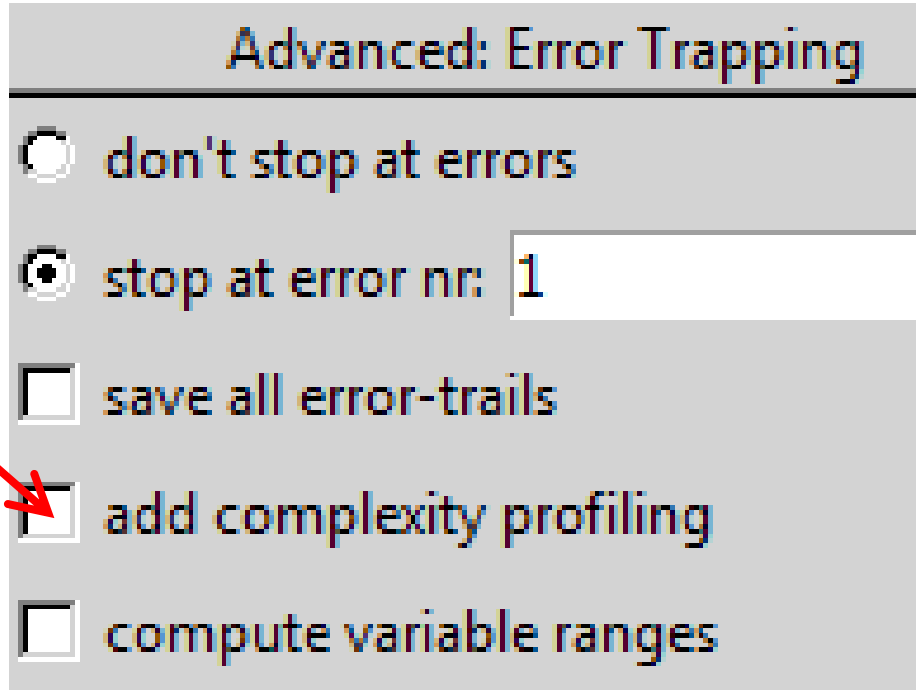
Zustandsschätzung (bei zweitem Lauf anpassen)

Suchtiefe, danach Abbruch

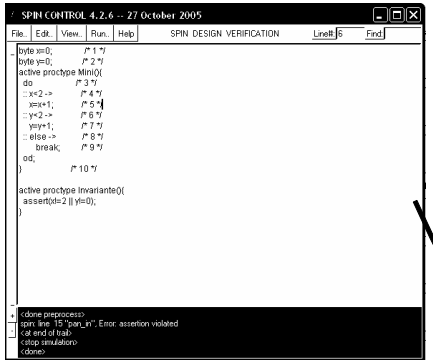


Einstellungen für Komplexitätsbetrachtung

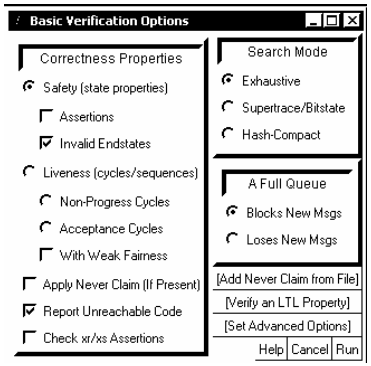
für jede Zeile der Spezifikation wird gezählt, wie häufig sie ausgeführt wurde



Verifikationsablauf



PROMELA-Spezifikation

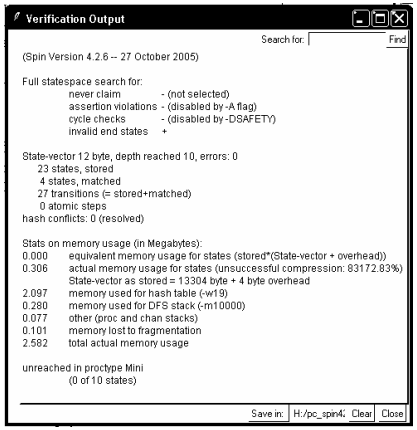


Verifikationseinstellung

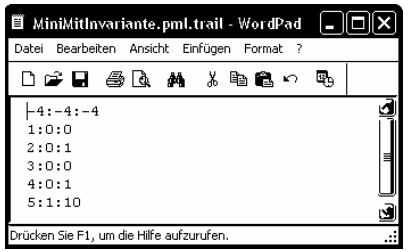
SPIN

C-Programm

kompilieren und ausführen



Ausgabe



Pfad zum Fehler

Beispiel: Ausgabe bei gescheitertem Beweis

```
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 9256
pan:1: assertion violated (x==2) (at depth 17)
pan: wrote add1.pml.trail

(Spin Version 6.2.3 -- 24 October 2012)
Warning: Search not completed
```

```
pan: elapsed time 0.073 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```

- Es wird die Art des Fehlers angegeben, hier Verletzung eines `assert`
- Es besteht sofort die Möglichkeit, einen Pfad zum Fehler im Simulator ablaufen zu lassen

Verifikationsergebnis (Standardeinstellungen) 1/2

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Verification Output

Search for: Find

```
(Spin Version 4.2.6 -- 27 October 2005)

Full statespace search for:
        never claim         - (not selected)
        assertion violations - (disabled by -A flag)
        cycle checks        - (disabled by -DSAFETY)
        invalid end states  +

State-vector 12 byte, depth reached 10, errors: 0
    23 states, stored
     4 states, matched
    27 transitions (= stored+matched)
     0 atomic steps
hash conflicts: 0 (resolved)
```

für Spezifikation `Mini`

Verifikationsergebnis (Standardeinstellungen) 2/2

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
(Spin Version 6.2.3 -- 24 October 2012)
        + Partial Order Reduction

Full statespace search for:
        never claim           - (not selected)
        assertion violations +
        cycle checks          - (disabled by -DSAFETY)
        invalid end states +

State-vector 12 byte, depth reached 10, errors: 0
       23 states, stored
        4 states, matched
       27 transitions (= stored+matched)
        0 atomic steps
hash conflicts:          0 (resolved)

Stats on memory usage (in Megabytes):
    0.001 equivalent memory usage for states (stored*(State-vector + overhead)
)
    0.292 actual memory usage for states
   64.000 memory used for hash table (-w24)
    0.343 memory used for DFS stack (-m10000)
   64.539 total actual memory usage


unreached in proctype Mini
        (0 of 10 states)

pan: elapsed time 0.022 seconds
```

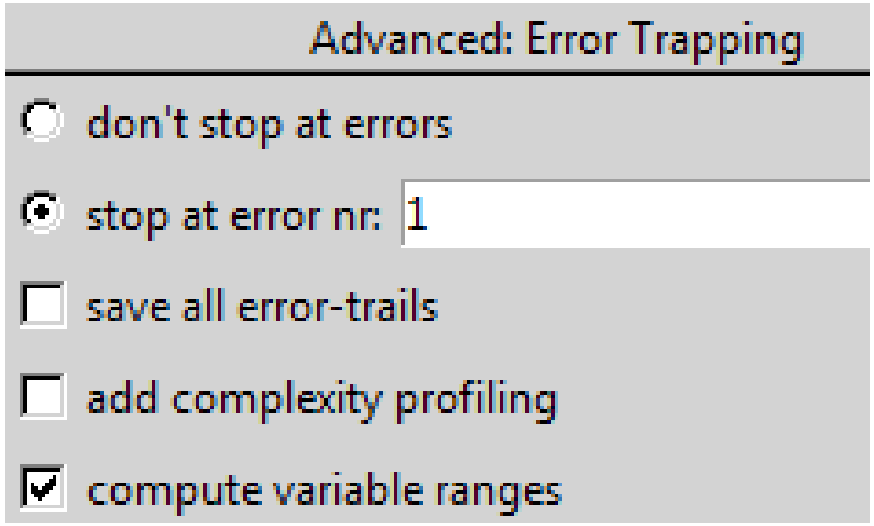
Längere Verifikationsläufe

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
Depth=  999999 States=   4e+006 Transitions= 6.07e+006 Memory=   220.598 t=     1.04 R=  4e+006
Depth=  999999 States=   5e+006 Transitions= 7.75e+006 Memory=   251.067 t=     1.38 R=  4e+006
Depth=  999999 States=   6e+006 Transitions= 9.43e+006 Memory=   281.633 t=     1.74 R=  3e+006
Depth=  999999 States=   7e+006 Transitions= 1.11e+007 Memory=   312.102 t=      2.1 R=  3e+006
Depth=  999999 States=   8e+006 Transitions= 1.28e+007 Memory=   342.668 t=     2.47 R=  3e+006
Depth=  999999 States=   9e+006 Transitions= 1.45e+007 Memory=   373.137 t=     2.89 R=  3e+006
Depth=  999999 States=   1e+007 Transitions= 1.61e+007 Memory=   403.703 t=     3.31 R=  3e+006
Depth=  999999 States= 1.1e+007 Transitions= 1.78e+007 Memory=   434.172 t=     3.75 R=  3e+006
Depth=  999999 States= 1.2e+007 Transitions= 1.95e+007 Memory=   464.739 t=     4.21 R=  3e+006
Depth=  999999 States= 1.3e+007 Transitions= 2.12e+007 Memory=   495.207 t=     4.64 R=  3e+006
Depth=  999999 States= 1.4e+007 Transitions= 2.28e+007 Memory=   525.774 t=      5.1 R=  3e+006
Depth=  999999 States= 1.5e+007 Transitions= 2.45e+007 Memory=   556.243 t=     5.59 R=  3e+006
Depth=  999999 States= 1.6e+007 Transitions= 2.62e+007 Memory=   586.809 t=     6.14 R=  3e+006
Depth=  999999 States= 1.7e+007 Transitions= 2.79e+007 Memory=   617.278 t=     6.63 R=  3e+006
Depth=  999999 States= 1.8e+007 Transitions= 2.95e+007 Memory=   647.844 t=     7.03 R=  3e+006
Depth=  999999 States= 1.9e+007 Transitions= 3.12e+007 Memory=   678.313 t=     7.43 R=  3e+006
Depth=  999999 States=   2e+007 Transitions= 3.29e+007 Memory=   708.879 t=     7.87 R=  3e+006
Depth=  999999 States= 2.1e+007 Transitions= 3.46e+007 Memory=   739.348 t=      8.3 R=  3e+006
Depth=  999999 States= 2.2e+007 Transitions= 3.62e+007 Memory=   769.914 t=     8.73 R=  3e+006
Depth=  999999 States= 2.3e+007 Transitions= 3.79e+007 Memory=   800.383 t=     9.21 R=  2e+006
```

Prüfung von Wertebereichen

- Spin erlaubt eine recht genaue Prüfung, welche Werte eine Variable annehmen kann.
- Folgende Verifikationseinstellungen werden benötigt [Set Advanced Options]:

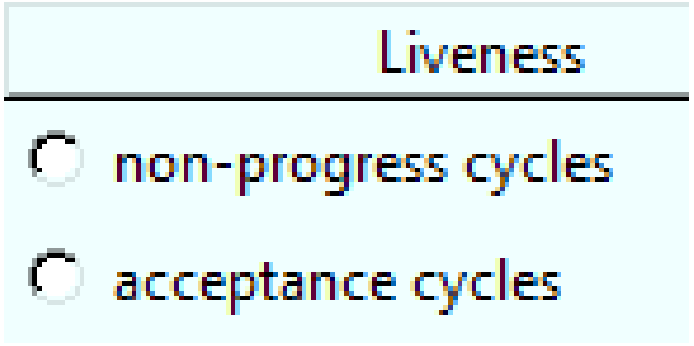


- Beispielausgabe:

```
Values assigned within interval [0..255]:
   Emp:wert       : 0-4,
   Emp:tmp : 0-5,
   erg     : 0-1, 3, 6, 10,
```

Lebendigkeitsanforderungen

4.4.7



- „etwas Gutes soll passieren“
- Wunsch: Das System soll immer sinnvoll voranschreiten
- genauer: Verhinderung von Livelocks, z. B. jeder fragt zyklisch: kann ich nächsten Schritt machen? Antwort: nein.
- Erfolgreicher Fortschritt wird in PROMELA mit „progress“-Markierungen definiert
- Forderung: Jeder unendliche Ablaufpfad durchläuft unendlich oft „progress“-Marken

Beispiel: Spezifikation mit „progress“-Marken

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
bool guard = true;
byte x = 1;
active proctype P1(){
  do
  :: x < 10 -> x = x + 1; guard = !guard
  :: x > 1 -> x = x - 1; guard = !guard
  :: guard = !guard
  od;
}
active proctype P2(){
  progress: do                                    /* 1 */
  :: x < 10 -> x = x + 1; guard = !guard
  :: x >1 -> x = x - 1; guard = !guard
  :: guard = !guard
  od;
}
active proctype P3(){
  do
  :: guard ->
        progress: guard = !guard                  /* 2 */
  od;
}
```

Verifikationsergebnis für Lebendigkeit

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
pan: non-progress cycle (at depth 50)
pan: wrote pan_in.trail
<<<<<START OF CYCLE>>>>>
 52:    proc  0 (P1) progressMarken.pml:5 (state 1) [((x<10))]
 54:    proc  0 (P1) progressMarken.pml:5 (state 2) [x = (x+1)]
 56:    proc  0 (P1) progressMarken.pml:5 (state 3) [guard = !(guard)]
 58:    proc  0 (P1) progressMarken.pml:6 (state 4) [((x>1))]
 60:    proc  0 (P1) progressMarken.pml:6 (state 5) [x = (x-1)]
 62:    proc  0 (P1) progressMarken.pml:6 (state 6) [guard = !(guard)]
spin: trail ends after 62 steps
#processes: 3
 62:    proc  2 (P3) progressMarken.pml:18 (state 3)
 62:    proc  1 (P2) progressMarken.pml:12 (state 3)
 62:    proc  0 (P1) progressMarken.pml:4 (state 8)
```

Kein Fortschritt, da P1 immer durch dritte Alternative läuft (realistisch?)

Anmerkung zu progress

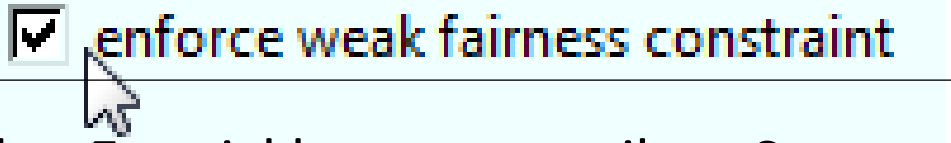
- nicht:

```
active proctype P(){
 do
 ::a!send ->
   if
   ::progress: b?ack,0;
   ::b?ack,1;
   fi;
 od;
}
```

```
error: (progressSyntaxproblem.pml:9)
label progress placed incorrectly
=====> instead of
         do (or if)
         :: ...
         :: Label: statement
         od (of fi)
=====> use
Label:   do (or if)
         :: ...
         :: statement
         od (or fi)
```

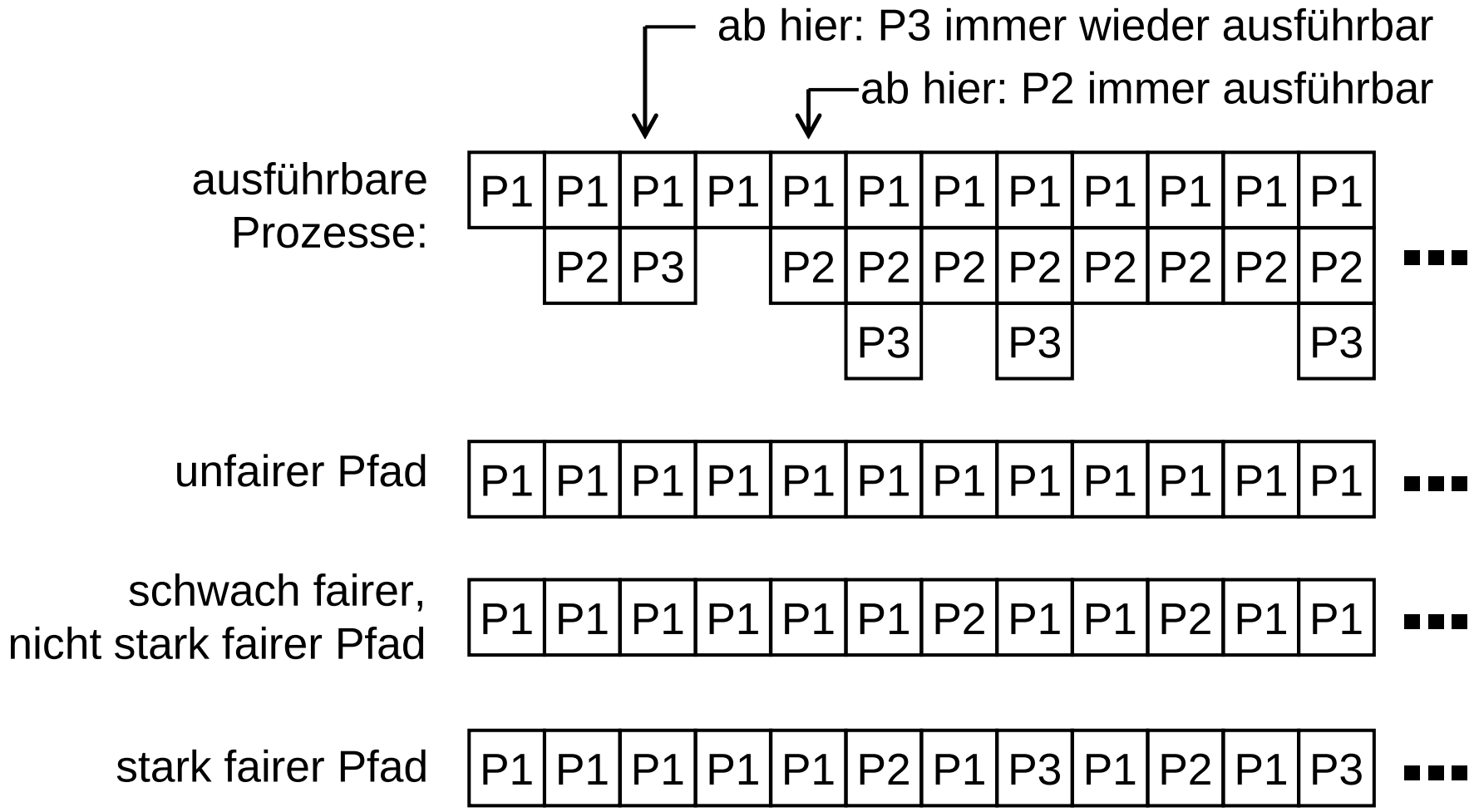
- Die Syntax erlaubt es nicht, progress-Markierungen direkt mit guards zu verbinden

Fairness



- Grundsätzlich sollen Entwickler von verteilten Systemen keine Annahmen machen, welcher Prozess wie schnell ist (Annahme muss sonst aufwändig nachgewiesen werden)
- Aber, dass ein Prozess immer und ein anderer bereiter Prozess nie rankommt ist unwahrscheinlich, deshalb können folgende Eigenschaften von Ausführungspfaden gefordert werden:
  - schwache Fairness: ein Prozess, der immer fortschreiten könnte, schreitet letztendlich auch fort
  - starke Fairness: ein Prozess, der immer wieder fortschreiten könnte, schreitet letztendlich auch fort
- in SPIN kann nur „schwache Fairness“ gefordert werden
- vorherige Verifikation klappt mit schwacher Fairness
- Die Implementierung von Fairness ist sehr aufwändig

Veranschaulichung von Fairness



4.5 Verifikation von in LTL formulierten Anforderungen

Video

- Syntax der Linearen Temporalen Logik (LTL)
- Semantik von LTL
- typische LTL-Anforderungen
- LTL in SPIN

Bedeutung von Anforderungen

- typischer Entwicklungsansatz mit inneren Iterationen:
  1. informelle Spezifikation
  2. Festlegung von Prozessen und wichtiger Variablen
  3. formale Anforderungen
  4. PROMELA-Spezifikation
  5. SPIN prüft Spezifikation gegen Anforderungen

- Zur Formalisierung von Anforderungen werden auch Logiken eingesetzt (Aussagenlogik und Prädikatenlogik erster Stufe sollten bekannt sein)

Syntax von LTL

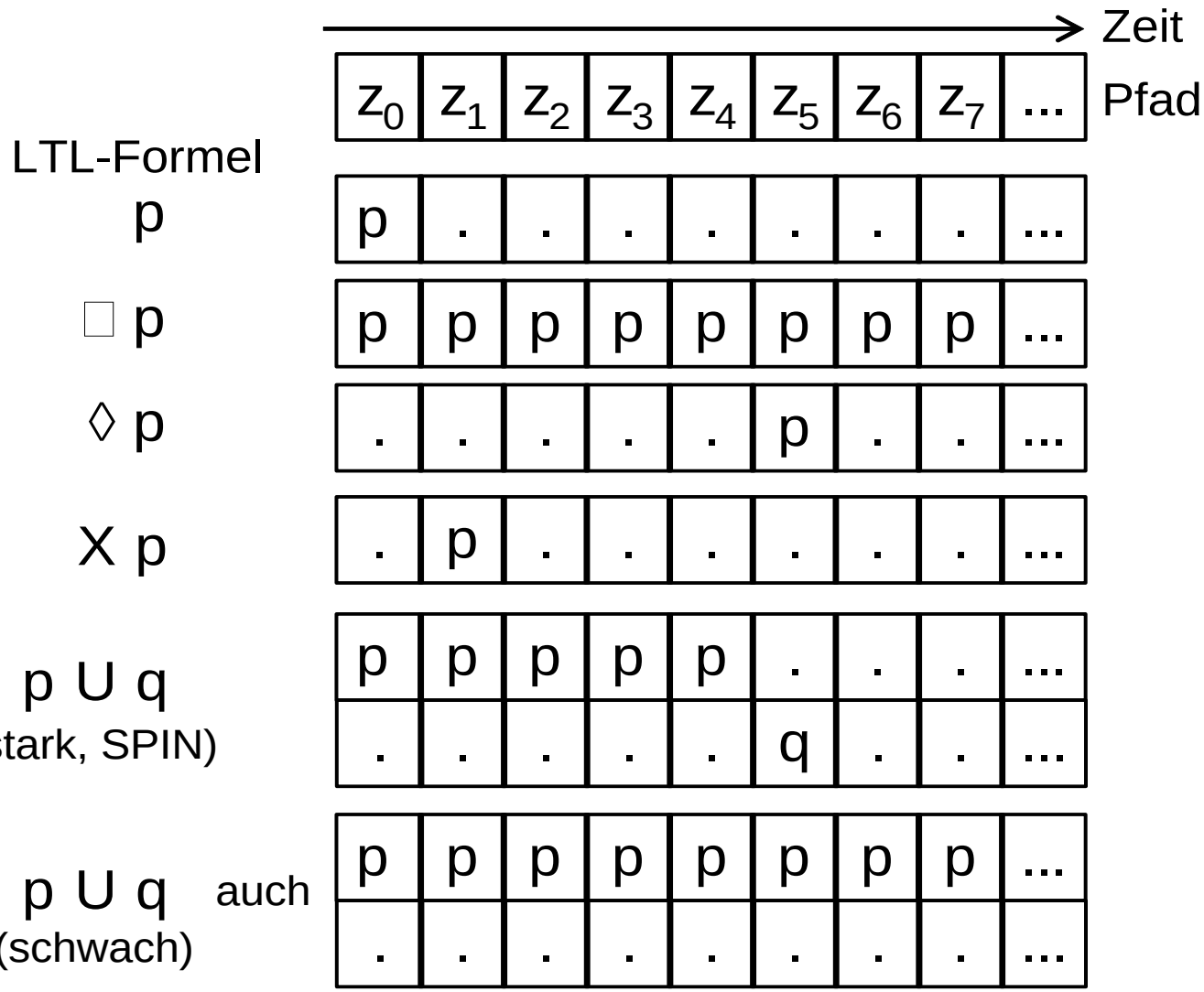
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition (Syntax der Linearen Temporalen Logik (LTL)): Zunächst sind alle einfachen Prädikate LTL-Formeln. Seien weiterhin p und q LTL-Formeln, dann sind auch

1. $\Box$ p , heißt „always p“, immer p
2. $\Diamond$ p , heißt „eventually p”, irgendwann p
3. X p , heißt „next p”, danach p
4. p U q , heißt „p until q“, p solange bis q

syntaktisch korrekte LTL-Formeln. Neue LTL-Formeln entstehen weiterhin durch die Verwendung der bekannten logischen Operatoren $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$ auch für LTL-Formeln. Klammern können genutzt werden, um die Auswertungsreihenfolge vorzugeben. Es gelten sonst die bekannten Ausführungsprioritäten.

Veranschaulichung von LTL-Formeln



Semantik von LTL (1/2)

Definition (Semantik von LTL-Formeln): Gegeben sei eine PROMELA-Spezifikation P und ein unendlicher Ausführungspfad z der Form $z_0\ z_1\ ...\ z_i\ ...$ von Zuständen einer Ausführung von P. Die Funktion Sem zur Berechnung der Semantik einfacher Prädikate ist aus der Programmverifikation (siehe später) entnommen.

[$Sem(a < b, z) \equiv z(a) < z(b) \equiv 3 < 42 \equiv$ true; z ordnet jeder Variable einen Wert zu, hier z(a)=3; z(b)=42]

1. Sei p eine einfaches Prädikat, dann erfüllt z die LTL-Formel p, wenn $Sem(p,z_0)$=wahr gilt, also der erste Zustand p erfüllt.
2. Sei □ p eine LTL-Formel, dann erfüllt z diese LTL-Formel, wenn alle Zustände p erfüllen, also für alle i $Sem(p,z_i)$=wahr gilt.
3. Sei ◊ p eine LTL-Formel, dann erfüllt z diese LTL-Formel, wenn ein Zustand p erfüllt, also es ein i mit $Sem(p,z_i)$=wahr gibt.

Semantik von LTL (2/2)

4. Sei X p eine LTL-Formel, dann erfüllt z diese LTL-Formel, wenn im folgenden Zustand p erfüllt ist, also Sem(p,z_1)=wahr gilt.
5. Sei p U q eine LTL Formel, dann erfüllt z die starke Variante des Until-Operators, wenn es einen Zustand z_j gibt, so dass für alle Zustände z_i mit i<j Sem(p,z_i)=wahr gilt und Sem(q,z_j)=wahr gilt.
6. Sei p U q eine LTL Formel, dann erfüllt z die schwache Variante des Until-Operators, wenn die starke Variante erfüllt ist oder wenn alle Zustände p erfüllen, also für alle i Sem(p,z_i)=wahr gilt.

- Anmerkung: SPIN unterstützt nur starke Variante des Until-Operators und keinen Next-Operator
- (Next-Operator nutzbar, mit speziellen Einstellungen, schwierig, da Spezifikation dann „stotterfrei“ sein muss)
- Spezifikation erfüllt LTL-Formel, wenn alle möglichen Pfade der Spezifikation die Formel erfüllen

Typische LTL-Formeln

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- $\Box\, p$ (Invariante)
- auf Versenden (Prädikat p) wird Antwort (q) erwartet
  $p \rightarrow \Diamond q$ (Antwort)
- immer wieder eine Antwort, also Invariante,
  $\Box\, (p \rightarrow \Diamond q)$ (Senden-Bestätigen, sauberer: $\Box\, (p \rightarrow X(\Diamond q))$
- kontinuierlicher Fortschritt p als Invariante
  $\Box\, (\Diamond\, p)$ (Fortschritt)
- ab einem bestimmten Zeitpunkt immer p
  $\Diamond\, (\Box\, p)$ (Stabilität)
- solange eine Eigenschaft p gilt, soll q nicht gelten
  $\Box\, (p \rightarrow \neg q)$ (Ausschluss)
- wechselseitiger Ausschluss von p und q
  $\Box\, (p \leftrightarrow \neg q)$ (wechselseitiger Ausschluss)

LTL-Rechenregeln

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

$$\neg(\Box\, p) \equiv \Diamond\, (\neg p)$$

$$\neg(\Diamond\, p) \equiv \Box\, (\neg p)$$

$$\Box\, (p \wedge q) \equiv (\Box\, p) \wedge (\Box\, q)$$

$$\Diamond\, (p \vee q) \equiv (\Diamond\, p) \vee (\Diamond\, q)$$

$$\Diamond\, p \equiv \text{true U } p \quad \text{(nur stark)}$$

$$\Diamond\Box\, (p \wedge q) \equiv (\Diamond\Box\, p) \wedge (\Diamond\Box\, q)$$

$$\Box\Diamond\, (p \vee q) \equiv (\Box\Diamond\, p) \vee (\Box\Diamond\, q)$$

$$p \text{ U } (q \vee r) \equiv (p \text{ U } q) \vee (p \text{ U } r)$$

$$(p \wedge q) \text{ U } r \equiv (p \text{ U } r) \wedge (q \text{ U } r)$$

LTL in ASCII

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

LTL-Formel	ASCII-Darstellung
p	p
□ p	[] p
◊ p	<> p
p U q	p U q
¬p	!p
p → q	p -> q
p ↔ q	p <-> q
p ∧ q	p && q
p ∨ q	p || q

Minimale Beispielnutzung (1/3)

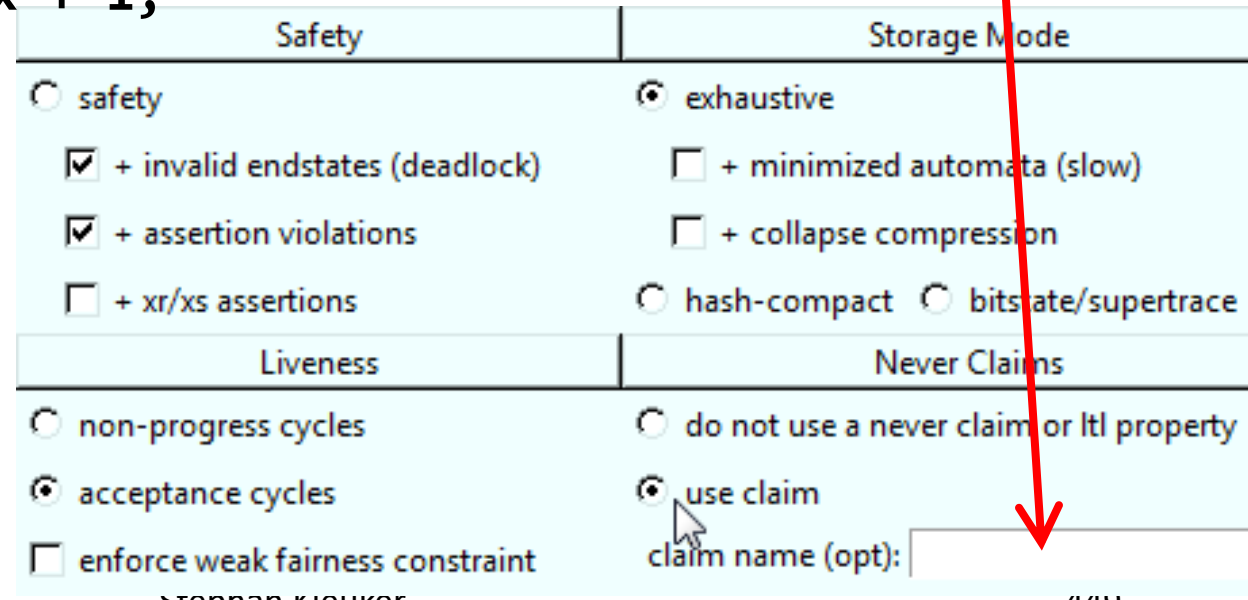
- #define für logische Ausdrücke notwendig

```
#define p (x<100)
byte x=3;

ltl formel1{[] p}

active proctype P(){
  do
  :: true -> x = x + 1;
  :: break
  od
}
```

mehrere Formeln angebbar; kann aber nur eine geprüft werden, Name steht hier



Minimale Beispielnutzung (2/3)

```
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DNOREDUCE -w -o pan pan.c
./pan -m10000  -a -n -c1
Pid: 14532
pan:1: end state in claim reached (at depth 391)
pan: wrote ltlErste.pml.trail

(Spin Version 6.2.3 -- 24 October 2012)
Warning: Search not completed

Full statespace search for:
        never claim          + (formel1)
        assertion violations + (if within scope of claim)
        acceptance   cycles  + (fairness disabled)
        invalid end states - (disabled by never claim)
```

Minimale Beispielnutzung (3/3)

```
[variable values,
step 390]

 0  =  P
 x  =  100
```

```
372:    proc  0 (P) ltlErste.pml:8 (state 2) [x = (x+1)]
374:    proc  0 (P) ltlErste.pml:8 (state 1) [(1)]
376:    proc  0 (P) ltlErste.pml:8 (state 2) [x = (x+1)]
378:    proc  0 (P) ltlErste.pml:8 (state 1) [(1)]
380:    proc  0 (P) ltlErste.pml:8 (state 2) [x = (x+1)]
382:    proc  0 (P) ltlErste.pml:8 (state 1) [(1)]
384:    proc  0 (P) ltlErste.pml:8 (state 2) [x = (x+1)]
386:    proc  0 (P) ltlErste.pml:8 (state 1) [(1)]
388:    proc  0 (P) ltlErste.pml:8 (state 2) [x = (x+1)]
390:    proc  0 (P) ltlErste.pml:8 (state 1) [(1)]
spin: trail ends after 391 steps
#processes: 1
391:    proc  0 (P) ltlErste.pml:8 (state 2)
1 processes created
Exit-Status 0
```

- Verifikation unterstützt direkt nur eine LTL-Formel
- bei mehreren Formeln muss eine ausgewählt werden
- C- bzw. C++-Kommentare zum Ein- und Auskommentieren gehen auch

4.6 Beispiele

Video

- deterministisches Programm (sortieren)
- Ampelsteuerung

Verifikation deterministischer Programme

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Sortierverfahren, informelle Spezifikation:

Laufe mit dem Zähler i von 0 bis zur Arraygröße-1

  Laufe mit dem Zähler j von i+1 bis zur Arraygröße

    falls das j-te Element kleiner als das i-te Element ist,
    vertausche diese

(Ist Min-Sort, nach i-tem Durchlauf steht i-t-kleinster Wert an Position i)

Sortierverfahren (1/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
#define N 5
#define MAX 3
byte array[N];
byte save[N];
bool initialized = false;
bool sorted = false;


init{
  run Initialize();
  run Sort();
  run Proof()
}
```

```
proctype Initialize(){  // in atomic
  byte count = 0;
  byte rnd = 0;
  do
  :: count < N ->
       rnd = 0;
       do
       :: rnd < MAX -> rnd = rnd + 1
       :: true ->
            array[count] = rnd;
            save[count] =array[count];
            break
       od;
       count = count + 1;
  :: else -> break
  od;
  initialized = true
}
```

Sortierverfahren (2/5)

```
proctype Sort(){
  byte i = 0;
  byte j;
  byte tmp;
  initialized;
  do
  :: i < N - 1 ->
       j = i + 1;
       do
       :: j < N - 1 ->
          /*< muss <= sein */
          if
          :: array[i] > array[j] ->
               tmp = array[i];
               array[i] = array[j];
               array[j] = tmp
          :: else ->
               skip
          fi;
          j = j + 1
          :: else ->
             break
       od;
       i = i + 1;
  :: else ->
       break
  od;
  sorted = true
}
```

Sortierverfahren (3/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
proctype Proof(){
  byte count = 0;
  byte count2 = 0;
  byte anzahl1 = 0;
  byte anzahl2 = 0;
  sorted;
  /* pruefe ob array sortiert ist */
  do
  :: count < N - 1 ->
       assert(array[count] <= array[count + 1]);
       count = count + 1
  :: else ->
       break
  od;
```

Sortierverfahren (4/5)

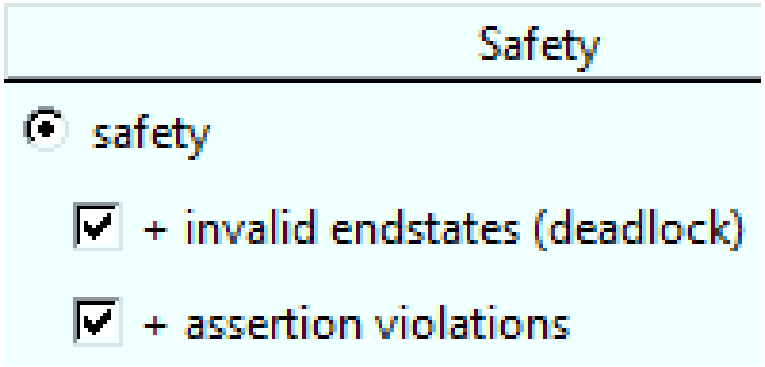
```
  count = 0; /* pruefe auf gleiche Elemente */
  do
  :: count<=MAX ->
       anzahl1 = 0;    anzahl2 = 0;     count2 = 0;
       do
       :: count2 < N ->
            if
            :: save[count2] == count -> anzahl1 = anzahl1 + 1
            :: else ->
                 skip
            fi;
            if
            :: array[count2] == count -> anzahl2 = anzahl2 + 1
            :: else -> skip
            fi;
            count2 = count2 + 1
       :: else -> break
       od;
       assert(anzahl1 == anzahl2);
       count = count + 1
  :: else -> break
  od
}
```

Sortierverfahren (5/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ausgabe der Simulation im Fehlerfall

```
array[0]  =  3
array[1]  =  3
array[2]  =  3
array[3]  =  3
array[4]  =  2
initialized  =  1
save[0]  =  3
save[1]  =  3
save[2]  =  3
save[3]  =  3
save[4]  =  2
sorted  =  1
```



Korrigierter Sortierer (j <= N-1)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

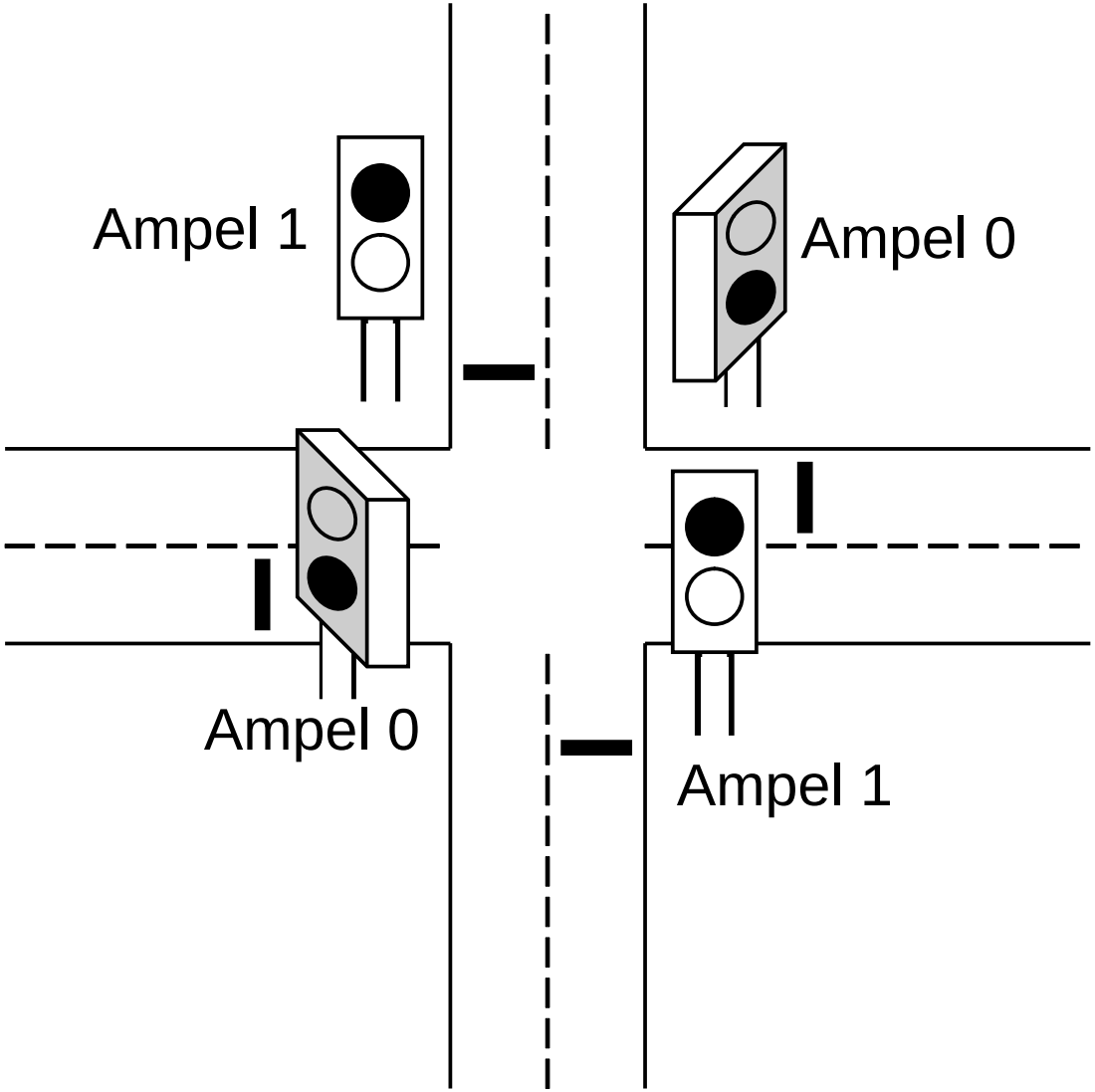
State-vector 48 byte, depth reached 240, errors: 0
   199108 states, stored
        0 states, matched
   199108 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         1 (resolved)

unreached in init
        (0 of 4 states)
unreached in proctype Initialize
        (0 of 19 states)
unreached in proctype Sort
        (0 of 26 states)
unreached in proctype Proof
        (0 of 41 states)
No errors found -- did you verify all claims?
```

Beispiel: Ampelsystem (1/6)

optimiertes Modell:

- nur Farben grün und rot
- Ampeln in einer Richtung gleich geschaltet
- nur eine Richtung grün



Beispiel: Ampelsystem (2/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- informelle Spezifikation:
  1. Die Ampeln zeigen entweder rot oder grün an.
  2. Die Ampeln schalten immer wieder zwischen rot und grün hin und her.
  3. Es ist immer maximal eine Ampel grün.
  4. Die Ampeln werden abwechselnd grün.
- typischer Entwicklungsweg:
  - informell Prädikate festlegen, dann mit Spezifikationsinformationen füllen
  - häufig muss Spezifikation angepasst werden (neue globale Hilfsvariablen)
  - ab und zu müssen Prädikate angepasst werden

Beispiel: Ampelsystem (3/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

informelle Prädikate:

- ampel0rot: gilt genau dann, wenn die Ampel 0 rot ist
- ampel0grün: gilt genau dann, wenn die Ampel 0 grün ist
- ampel1rot: gilt genau dann, wenn die Ampel 1 rot ist
- ampel1grün: gilt genau dann, wenn die Ampel 1 grün ist
- ampel0zuletzt: gilt genau dann, wenn Ampel 0 zuletzt grün war
- ampel1zuletzt: gilt genau dann, wenn Ampel 1 zuletzt grün war

Beispiel: Ampelsystem (4/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

1. □ (((ampel0rot ∨ ampel0grün) ∧ (ampel0rot ↔ ¬ampel0grün))
   ∧ ((ampel1rot ∨ ampel1grün) ∧ (ampel1rot ↔ ¬ampel1grün))

2. □ ((ampel0rot → (◊ ampel0grün))
   ∧ (ampel0grün → (◊ ampel0rot))
   ∧ (ampel1rot → (◊ ampel1grün))
   ∧ (ampel1grün → (◊ ampel1rot)))

3. □ ((ampel0grün → ¬ampel1grün)
   ∧ (ampel1grün → ¬ampel0grün))

4. □ (((ampel0zuletzt) U (ampel1zuletzt ∧ ¬ampel0zuletzt))
   ∧ ((ampel1zuletzt) U (ampel0zuletzt ∧ ¬ampel1zuletzt))
   ∧ (ampel0zuletzt ↔ ¬ampel1zuletzt))

Beispiel: Ampelsystem (5/6)

```
mtype={p,v, rot,gruen};
mtype zustand[2];
bool gestartet = false;
chan zentrale
       = [0] of {mtype,bit};
bit dran=0; /* wer nächstes */

proctype ampel(bit name){
  do
  :: zustand[name]==rot ->
     zentrale!p,name;
     zustand[name]=gruen;
     zustand[name]=rot;
     zentrale!v,name;
  od;
}
```

```
proctype Umschalter(){
  do
  ::
   zentrale?p,eval(dran);
     zentrale?v,_;
     dran=1-dran;
  od;
}

init{
  atomic{
    zustand[0]=rot;
    zustand[1]=rot;
    run ampel(0);
    run ampel(1);
    run Umschalter();
    gestartet=true;
  };
}
```

Beispiel: Ampelsystem (6/6)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
#define ampel0rot (zustand[0]==rot)
#define ampel0gruen (zustand[0]==gruen)
#define ampel1rot (zustand[1]==rot)
#define ampel1gruen (zustand[1]==gruen)
#define ampel0zuletzt (dran==1)
#define ampel1zuletzt (dran==0)
```

- Verifikation der ersten Anforderung scheitert, da vor Initialisierung alle Ampeln weder rot noch grün sind, typische Lösung
  - □(initialisiert → Anforderung)
  - **`#define initialisiert (gestartet==true)`**
  - ==true kann natürlich weggelassen werden

4.7 Einsatzmöglichkeiten von Model Checkern

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- Verknüpfung von Promela und C
- Planung der SPIN-Nutzung
- Verknüpfungsmöglichkeiten von SPIN-Ergebnissen mit der Praxis

Promela und C (1/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Erinnerung: Aus Spezifikation und Flags wird individuelles C-Programm als Model Checker generiert (pan.c, pan.h)
- Möglichkeiten:
  - C-Code während der Verifikation ausführen (nicht Simulation)
  - C-Code zur Prüfung von Bedingungen nutzen
  - Aus C-Code auf Promela-Variablen zugreifen
  - C-Code-Elemente zum Teil des Zustandsvektors machen

Promela und C (2/5)

```
c_decl {
  \#include <stdio.h>
  \#include <stdlib.h>
}

int spin_local;
c_code {double dbl;
        int gleich(int* w1, double* w2){
          printf("gleich: %d :: %f\n",*w1,*w2);
          return *w1 == (int)*w2;
   }
} ;
```

c_decl u. a. am Anfang zum Laden von Libraries (Backslash sinnvoll, damit nach Modellcodegenerierung genutzt); hier auch c_code nutzbar

„normale“ globale Promela-Variable

Programmanweisungen in c_code block, hier Variable und C-Funktion

Promela und C (3/5)

Prozesse können Promela und C-Code mischen

Promela-Anweisungen

```
init{
  printf("start\n");
  select(spin_local:40..44);
  c_code {dbl = 42.42;
          printf("init: %d\n",gleich(&(now.spin_local),&dbl));
  }
}
```

C-Code mit Zugriff auf Promela-Variable über now.

Promela und C (4/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ergebnis der Verifikation

```
verification result:
spin -a  C000.pml
gcc-4 -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o
   pan pan.c
./pan -m10000
Pid: 13704
gleich: 40 :: 42.420000
init: 0
gleich: 41 :: 42.420000
init: 0
gleich: 42 :: 42.420000
init: 1
gleich: 43 :: 42.420000
init: 0
gleich: 44 :: 42.420000
init: 0
```

Promela und C (5/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

weitere Möglichkeiten:

- c_decl kann Typdefinitionen enthalten, deren Variablen mit c_state in den Zustandsvektor aufgenommen werden
- mit c_track können in c_code deklarierte Variablen in den Zustandsvektor aufgenommen werden
- c_expr{.} enthält Booleschen C-Ausdruck (z. B. für do oder if)
- c_code[.]{.}, in optionalen eckigen Klammern kann Boolescher C-Ausdruck stehen, der assert entspricht

- G. Holzmann, R. Joshi, Model-Driven Software Verification, http://spinroot.com/gerard/pdf/spin04.pdf

Einsatzplanung für SPIN

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- SPIN eignet sich zur Verifikation sequenzieller und verteilter Systeme
- Basisansatz: Spezifiziere neuen Algorithmus in Promela und verifiziere mit SPIN
- Frühzeitig klären:
  - benötigte Wertebereiche
  - benötigte Parameter bei Kommunikation
  - benötigte Prozessanzahl
  - benötigte Puffergrößen
- Verifikation häufig über assert möglich, bei Verteilung oft LTL sinnvoll

Bedeutung von Model Checking

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Übertragung in die Praxis
- Komplexe Systeme nie vollständig verifizierbar
- Keine 100%-tige Sicherheit,
  - wenn Anforderungen vergessen
  - wenn Anforderungen falsch formalisiert
  - wenn Annahmen über die Umgebung nicht zutreffen
  - wenn Fehler aus der Entwicklung mit in die Verifikation übertragen werden
- Fazit: Model Checking ist ein weiteres wichtiges Hilfsmittel für Software-Qualität, liefert aber auch keine Garantien

Nutzungsmöglichkeiten für Model Checking

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Klassisch: Überprüfung eines kritischen Algorithmus

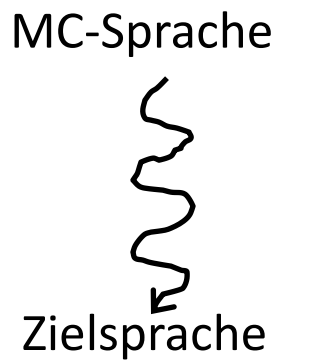
Auswahl des passenden MC-Werkzeugs

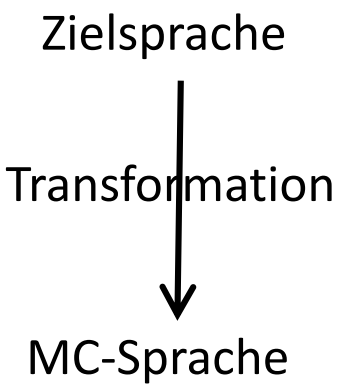
- welche Sprachkonstrukte werden nativ unterstützt, die man braucht
- lassen sich fehlende Sprachkonstrukte leicht nachbauen
- wie kann ich Erkenntnisse in die Praxis übertragen

  verhält sich Model Checker genau wie mein Programm in meiner Zielsprache -> semantische Äquivalenz

Model Checking-Ergebnisse in die Praxis (1/3)

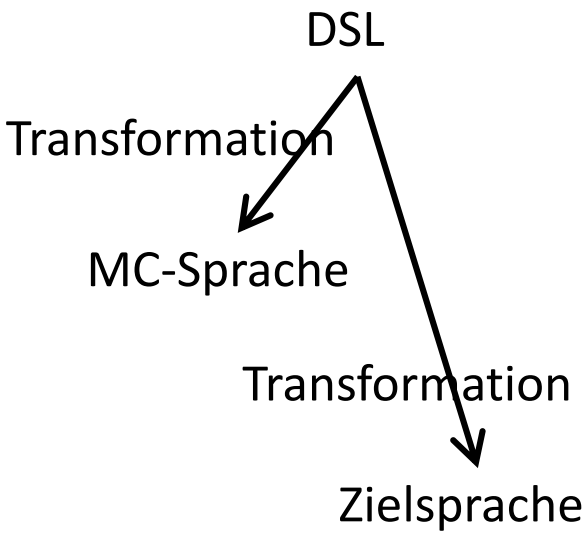
- Algorithmus wird in MC-Eingabesprache direkt geschrieben und verifiziert
  - Übertragung in andere Sprache wackelig
  - Variante: MC die direkt nutzbare Teilmengen von realen Sprachen (C oder Java) unterstützen
- Algorithmus in Zielsprache formuliert, dann mit Transformationsalgorithmus in MC-Sprache umgewandelt
  - Zielsprache oft einschränken
  - Transformation kaum formal als korrekt nachweisbar
  - trotzdem guter Ansatz





Model Checking-Ergebnisse in die Praxis (2/3)

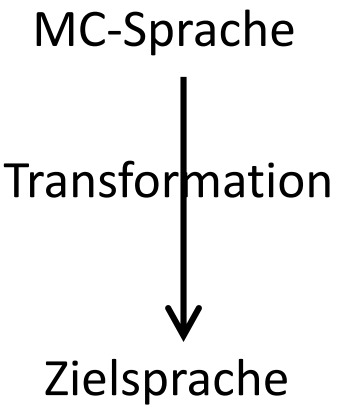
- neue Modellierungssprache entwickeln, die leicht in MC-Sprache und Zielsprache transformierbar ist
  - typisch für Model Driven Software Development (MDSD) mit Domain Specific Languages (DSL)
  - zwei semantisch kritische Transformationen
  - Beispiel: endliche Automaten leicht nach Promela, Timed Automata übersetzbar, ebenso nach C



Model Checking-Ergebnisse in die Praxis (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- MC-Sprache in Zielsprache transformieren
  - formalisiert ersten Ansatz
  - oft ineffiziente Ergebnisse, da alle Sprachkonstrukte nachgebildet werden müssen
  - Variante: Fokus auf Teile der MC-Sprache legen (verwandt zu DSL-Ansatz)



Kreativer Model Checking-Einsatz

- Lösungsraum formulieren, dann Model Checker nach Lösung suchen lassen
  - Optimieren, um minimale Lösung zu finden
- Optimierung von Algorithmen
  - man hat bekannte funktionierende Lösung, die nicht effizient ist; spezifiziert diese in MC-Sprache
  - man spezifiziert neue Lösung in MC-Sprache
  - man nutzt Model Checker um für möglichst viele Fälle zu prüfen, dass beide Algorithmen gleiches Ergebnis liefern
  - Beispiel auf folgenden Folien

Maximum von drei Werten (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
bool endeNeu = false;
bool endeOk = false;
int maxNeu;
int maxOk;

//bekanntes funktionierendes Verfahren
proctype Ok(int x; int y; int z){
  if
  :: x >= y && x >= z -> maxOk = x;
  :: y >= x && y >= z -> maxOk = y;
  :: z >= x && z >= y -> maxOk = z;
  fi;
  endeOk = true;
}
```

Maximum von drei Werten (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

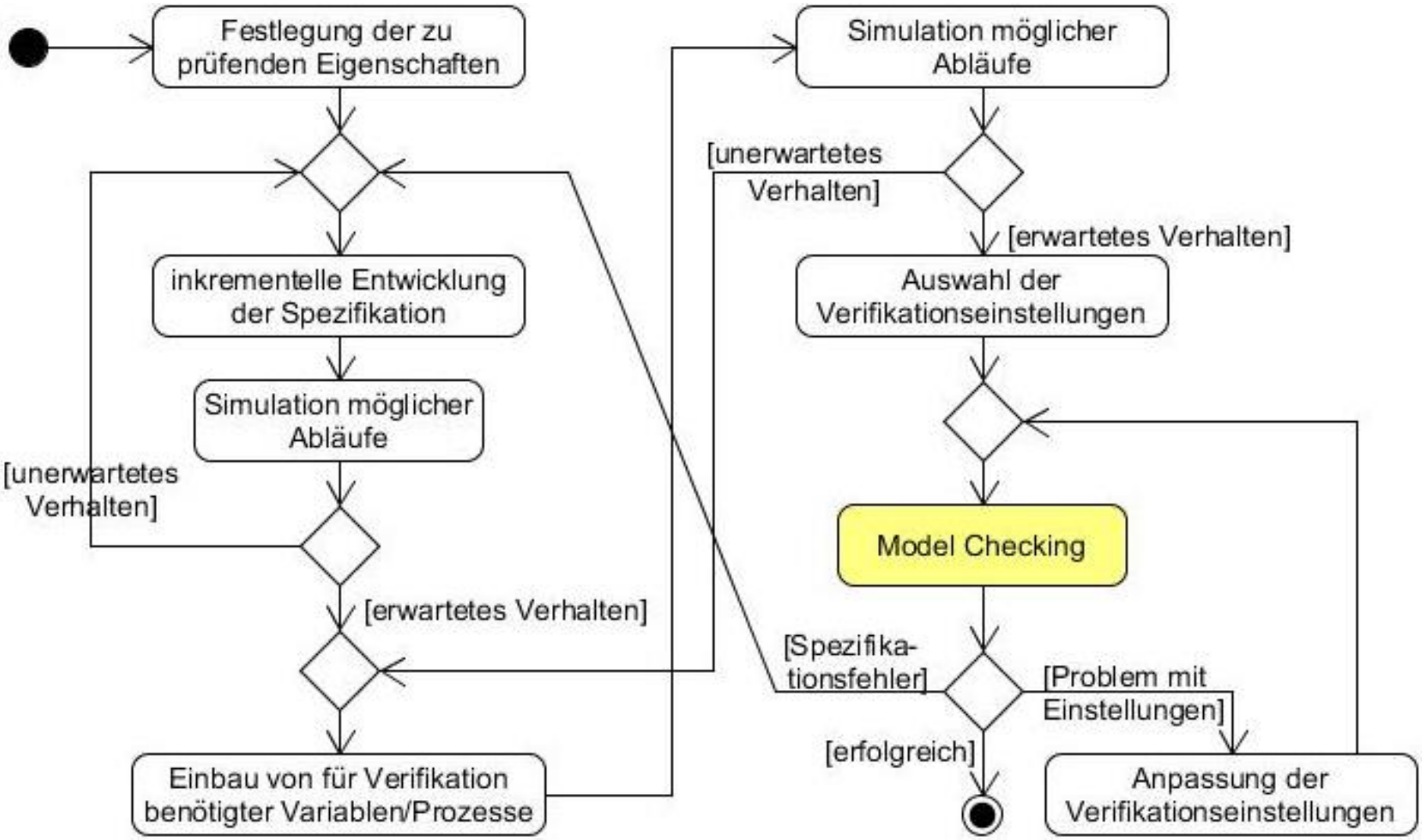
```
// zu ueberpruefendes Verfahren
proctype Neu(int x; int y; int z){
  if
  :: x > z -> maxNeu = x;
  :: else -> skip;
  fi;
  if
  :: y > x -> maxNeu = y;
  :: else -> skip;
  fi;
  if
  :: z > y -> maxNeu = z;
  :: else -> skip;
  fi;
  endeNeu = true;
}
```

Maximum von drei Werten (3/3)

```
init{
  int a, b, c;
  if // select(a: -1..1) funktioniert nicht 6.4.3 aber 6.3.2
  :: a =-1;
  :: a = 0;
  :: a = 1;
  fi;
  if
  :: b =-1;
  :: b = 0;
  :: b = 1;
  fi;
```

```
  if
  :: c =-1;
  :: c = 0;
  :: c = 1;
  fi;
  run Ok(a,b,c);
  endeOk;
  run Neu(a,b,c);
  endeNeu;
  assert(maxOk == maxNeu);
}
```

Standardprozess



Ausflug: Prozesse mit Go (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
package main
import "fmt"

func main() {  // eigentlich als letzte Funktion
   fmt.Printf("starten\n")
   puffer := 0
   kanal := make(chan int, puffer) // sync, Kanal mit int-Werten
   go Sende(kanal, 3, 7)           // Thread
   Empfange(kanal)                 // Funktionsaufruf
   fmt.Printf("ende\n")
}

func Sende(c chan int, von int, bis int) {
   fmt.Printf("senden\n")
   for i := von; i < bis; i++ {
       c <- i                      // senden
   }
   close(c)
   fmt.Printf("senden ende\n")
}
```

Ausflug: Prozesse mit Go (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
func Empfange3(c chan int) { // mit Nichtdeterminismus
   fmt.Printf("empfangen\n")
   ok := true
   lesen := 0
   for ok { // entspricht for ; ok ;
       select { // ueblich mit verschiedenen Kanaelen
       case lesen, ok = <-c:
              fmt.Printf(" gelesen 1: %v %v\n", lesen, ok)
       case lesen, ok = <-c:
              fmt.Printf(" gelesen 2: %v %v\n", lesen, ok)
       }
   }
   fmt.Printf("empfangen ende\n")
}
```

Ausflug: Prozesse mit Go (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
// Ausgabe
starten
empfangen
senden
 gelesen 2: 3 true
 gelesen 1: 4 true
 gelesen 1: 5 true
 gelesen 2: 6 true
senden ende
 gelesen 2: 0 false
empfangen ende
ende
```