LEHRSTUHL 14
SOFTWARE ENGINEERING

Willkommen zur Vorlesung *Softwarekonstruktion* im Wintersemester 2011/2012

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

02. Software Engineering (SWE)

[inkl Beiträge von Prof. Volker Gruhn und Prof. Ian Summerville]

Themen der Vorlesung

Kapitel 1:	Intro mit Vorstellung der Professur und Gliederung der Vorlesung
Kapitel 2:	Allgemeine Prinzipien des SW-Engineering
Kapitel 3:	Spezifikation im Allgemeinen
Kapitel 4:	Algebraische Spezifikation
Kapitel 5:	Petrinetze
Kapitel 6:	Modellgetriebene SW-Entwicklung
Kapitel 7:	Object Constraint Language (OCL)
Kapitel 8:	Testen im Allgemeinen, Kontrollflussorientierte Testverfahren, Datenflussorientierte Testverfahren

LEHRSTUHL 14
SOFTWARE ENGINEERING

Agenda

Kap. 2 Software Engineering (SWE)

- 40 Jahre Software Engineering
- Software Engineering - wann braucht man‘s?
- Eigenschaften von Software
- Prinzipien des SWE

40 Jahre Software Engineering

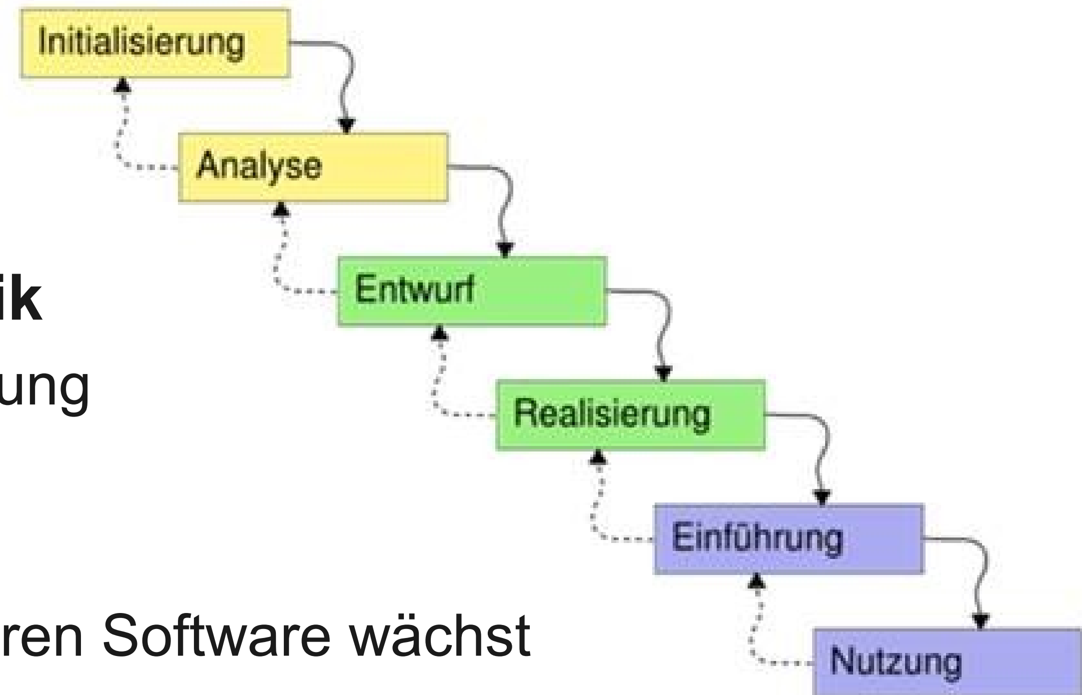
- 1968: **NATO-Konferenz in Garmisch-Partenkirchen**
  - Identifikation des Sachverhalts des Anwendungsstaus und Prägung des Begriffs der Software-Krise (nötige Software kann mit den zur Verfügung stehenden Ressourcen nicht entwickelt werden).
- Seitdem:
  - Vielzahl technologischer Lösungsansätze, die einzelne Aktivitäten der Software-Entwicklung unterstützen
  - lange Innovationszyklen
  - wenig Ansätze mit industriellen Auswirkungen
  - enorme Produktivitätsverbesserungen, aber keine Produktivitätssprünge
  - zusätzliche Software-Lösungen geraten in Reichweite,
    der Anwendungsstau bleibt bestehen

40 Jahre Software Engineering

- Software-Krise
  - **60er Jahre: Spezialrechner mit Spezialsoftware**
    - niedrige Rechenleistung
    - kleiner Hauptspeicher
    - hohe Kosten / punktuelle Rentabilität
    - batch-orienterte Software
    - Programme werden meist von den Anwendern erstellt
    - Software-Entwickler sind Autodidakten
    - Programmier*kunst*

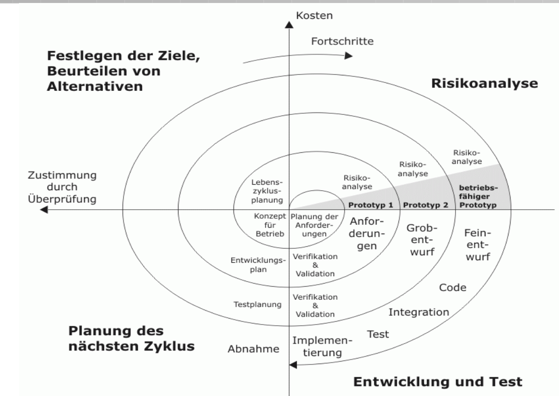
40 Jahre Software Engineering

- Software-Krise ff.
  - **70er Jahre: Mikroelektronik**
    - explodierende Rechenleistung
    - größerer Hauptspeicher
    - höhere Rentabilität
    - Menge der prinzipiell lösbaren Software wächst
    - Software kann nicht mehr von einzelnen erstellt werden

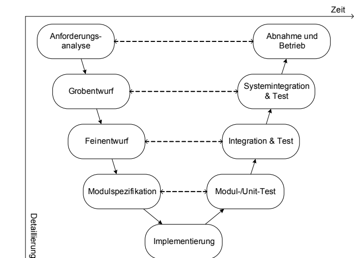


40 Jahre Software Engineering

- Software-Krise ff.



  - **80er Jahre: Software-Massenmarkt**
    - Trennung von Anwendern und Entwicklern
    - Software-Einsatz orientiert sich am wirtschaftlichen Nutzen
    - Software-Beschreibung bestimmt Vertragsverhältnisse
  - **90er Jahre: Komponenten-Markt und Migration**
    - Integration und Interoperabilität
    - Branchenmärkte



- Software-Krise ff.
  - **00er Jahre: Konzentration auf Anwendungswissen**
    - Modellgetriebene / Generative Entwicklung
    - Fokussierung auf Anwendungswissen
    - „Business Aligned“ IT
    - Vorsichtige Trends zu schlanker Software

40 Jahre Software Engineering

- Die Software-Krise als Katalysator für das Software Engineering
  - Nötig:
    - von der Kunst zur Ingenieurskunst
    - von der individuellen Bastelei zum Produktionsprozess
  - Aber:
    - Die Änderungen in den Rahmenbedingungen haben sich nur langsam und unter Schmerzen auf das Software Engineering niedergeschlagen.
    - Noch heute wird Software-Entwicklung teilweise als künstlerischer Prozess verstanden und als solcher akzeptiert.

40 Jahre Software Engineering

- Software Engineering bleibt Gegenstand des wissenschaftlichen Diskurses:

"Software development is and always will be somewhat experimental. The actual software construction isn't necessarily experimental, but its conception is. And this is where our focus ought to be."

Tom DeMarco: Software Engineering:
An Idea Whose Time Has Come and Gone.
In: IEEE Software, 26 (2009) Nr. 4

Wann braucht man's?

- **Software Engineering hat was mit “viel” zu tun:**
  - Die zu erstellende Software hat
    VIELE Komponenten
  - Die zu erstellende Software wird von
    VIELEN Anwendern benutzt werden
  - Die zu erstellende Software wird von
    VIELEN Entwicklern erstellt
  - Die zu erstellende Software wird möglicherweise in
    VIELERLEI Hinsicht erweitert
  - Die zu erstellende Software soll auf
    VIELEN Plattformen laufen
  - In der Entwicklung der zu erstellenden Software kommen
    VIELE Rollen vor

LEHRSTUHL 14
SOFTWARE ENGINEERING

Frage

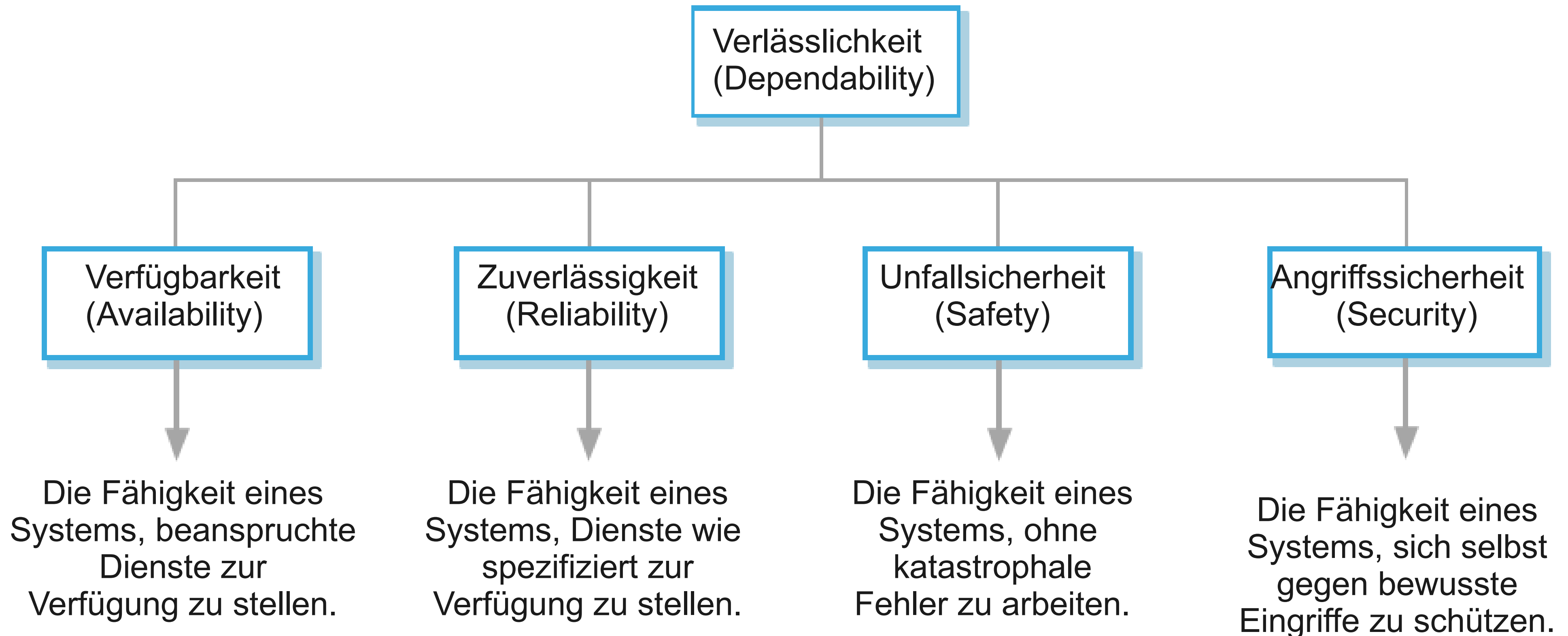
- **Software Engineering hat das Ziel, wichtige & schwierige Eigenschaften von Software sicherzustellen.**
- **Welche Eigenschaften sollte Software haben?**

Wann braucht man's?

- **Durch Software Engineering sicherzustellende Eigenschaften von Software (Übersicht):**
  - Verlässlichkeit / Korrektheit / Zuverlässigkeit / Sicherheit / Robustheit
  - Performanz
  - Benutzungsfreundlichkeit, Gebrauchstauglichkeit (Usability)
  - Wartbarkeit
  - Wiederverwendbarkeit
  - Portierbarkeit
  - Interoperabilität

Hauptsächliche Verlässlichkeitseigenschaften

LEHRSTUHL 14
SOFTWARE ENGINEERING



Verlässlichkeit

Verlässlichkeits-Eigenschaften (Dependability):

- Verfügbarkeit (Availability)
  - Die Wahrscheinlichkeit, dass das System in Betrieb genommen werden kann und in der Lage ist, nützliche Dienste für die Nutzer zu liefern.
- Zuverlässigkeit (Reliability)
  - Die Wahrscheinlichkeit, dass das System Dienste wie vom Benutzer erwartet liefert.
- Unfallsicherheit (Safety)
  - Ein Urteil darüber, wie wahrscheinlich es ist, dass das System Schaden an Menschen oder seiner Umgebung verursacht.
- Angriffssicherheit (Security)
  - Ein Urteil darüber, wie wahrscheinlich es ist, dass das System zufällige oder vorsätzliche Angriffe abwehren kann.

Verlässlichkeits-Eigenschaften

- Für viele IT-basierte Systeme ist Verlässlichkeit die wichtigste Systemeigenschaft.
- Die Verlässlichkeit eines Systems spiegelt das Maß an Vertrauen der Benutzer in das System wider. Es spiegelt das Ausmaß an Zuversicht wider, dass alles so funktioniert, wie der Benutzer es erwartet und nicht im Normalfall Fehler verursacht.
- Verlässlichkeit deckt die damit verbundenen Systemattribute der Zuverlässigkeit, Verfügbarkeit und Sicherheit ab. Diese sind alle voneinander abhängig.

Wichtigkeit der Verlässlichkeit

- Systemausfälle können weitreichende Wirkungen haben, durch die eine große Zahl von Menschen durch den Ausfall betroffen sein können.
- Systeme, die nicht verlässlich und unzuverlässig, oder unsicher (i.S.v. Safety und Security) sind, können von ihren Nutzern abgelehnt werden.
- Die Kosten eines Systemausfalls können sehr hoch sein, wenn das Versäumnis zu wirtschaftlichen Verlusten führt oder physikalische Schäden verursacht werden.
- Unzuverlässige Systeme können Informationsverlust mit großen Reparaturkosten verursachen.

Andere Verlässlichkeitseigenschaften

- Reparierbarkeit (Repairability)
  - Reflektiert das Ausmaß, in dem das System im Falle eines Ausfalls repariert werden kann.
- Wartbarkeit (Maintainability)
  - Reflektiert das Ausmaß, in dem das System an neue Anforderungen angepasst werden kann.
- Überlebensfähigkeit (Survivability)
  - Reflektiert das Ausmaß, in dem das System die Dienste während feindlicher Angriffe liefern kann.
- Fehlertoleranz (Error tolerance / fault tolerance)
  - Reflektiert das Ausmaß, inwieweit Eingabefehler durch den Benutzer oder Fehler durch die Hard- oder Software vermieden und toleriert werden können.

Erreichung der Verlässlichkeit

- Vermeidung von zufälligen Fehlern bei der Entwicklung des Systems.
- Prozesse für die Validierung und Verifikation des Entwurfs, die wirksam sind bei der Entdeckung von verbleibenden Fehlern im System.
- Design von Schutzmechanismen, die externe Angriffen abwehren.
- Ordnungsgemäße Konfiguration für die Betriebsumgebung.
- Recovery-Mechanismen zur Wiederherstellung des normalen Systembetriebs-Dienstes nach einem Ausfall.

Kosten der Verlässlichkeit

LEHRSTUHL 14
SOFTWARE ENGINEERING

- Die Kosten der Verlässlichkeit neigen exponentiell mit dem erforderlichen Grad an Verlässlichkeit anzusteigen.
- Zwei Gründe:
  - Die Verwendung von teureren Entwicklungs-Techniken und Hardware, die erforderlich sind, um die höheren Ebenen der Verlässlichkeit zu erreichen.
  - Die erhöhte Prüfung und Validierung von Systemen, die erforderlich ist, um Kunden und Behörden von der geforderten Verlässlichkeit zu überzeugen.

Eigenschaften von Software

- Korrektheit:
  - Übereinstimmung eines Programms mit seiner Spezifikation
    - ohne Spezifikation ist die Korrektheitsfrage nicht entscheidbar
    - bei informaler Spezifikation ist die Korrektheitsfrage nicht eindeutig entscheidbar
- Zuverlässigkeit:
  - Dauerhafte Einsetzbarkeit: Der Anwender kann sich erlauben, von der Software abzuhängen
    - Zuverlässigkeit ist ein relatives Kriterium, das vom Einsatzzweck der Software und vom Schadenspotential der Nichtverfügbarkeit der Software abhängt.
    - Zuverlässigkeit von Software wird bei neuer Software zuweilen nicht erwartet (Bananen-Software). Wesentlicher Unterschied zu fast allen anderen industriellen Produkten.

Frage

- Wie hängen Korrektheit und Zuverlässigkeit voneinander ab ?

Eigenschaften von Software

- Zusammenhang zwischen Korrektheit und Zuverlässigkeit

	zuverlässig	**nicht zuverlässig**
korrekt	alle Anforderungen spezifiziert und erfüllt (inkl. Zuverlässigkeits-anforderungen)	unspezifizierte Zuverlässigkeits-Anforderungen
nicht korrekt	Alle Zuverlässigkeits-anforderungen erfüllt, aber evt. nicht alle funktionalen Anforderungen	Fehler und Fehlverhalten

Zuverlässigkeit und Spezifikationen

- Zuverlässigkeit kann nur in Bezug auf eine System-Spezifikation formal definiert werden, d.h. ein Fehler ist eine Abweichung von einer Spezifikation.
- Allerdings sind viele Spezifikationen unvollständig oder unrichtig - d.h. ein System, das sich nach der Spezifikation richtet, kann aus der Sicht der Nutzer als "nicht bestanden" angesehen werden.
- Außerdem lesen Benutzer die Spezifikationen nicht und wissen damit nicht, wie sich das System verhalten soll.

Unfallsicherheit (Safety)

- Ist eine Eigenschaft eines Systems, welche die Fähigkeit des Systems darstellt (unter normalen oder unnormalen Umständen) ohne Gefahr, also ohne Verletzungen von Menschen und ohne Schäden der Systemumgebung zu arbeiten.
- Es ist wichtig, dabei die Sicherheit der Software zu betrachten, da die meisten Geräte, deren Ausfall kritisch ist, jetzt in Kontrolle von softwarebasierten Steuerungssystemen liegen.
- Sicherheitstechnische Anforderungen sind oft exklusive Anforderungen, d.h. sie schließen unerwünschte Situationen aus, anstatt die benötigten Systemdienste zu gewährleisten. Sie erzeugen funktionale Sicherheitsanforderungen.

Angriffsicherheit (Security)

- Die Angriffsicherheit eines Systems ist eine Systemeigenschaft, die die Fähigkeit des Systems widerspiegelt, sich gegen zufällige oder vorsätzliche Angriffe von außen schützen.
- Angriffsicherheit ist wichtig, da die meisten Systeme so vernetzt sind, dass der externe Zugriff auf das System über das Internet möglich ist.
- Angriffsicherheit ist eine wesentliche Voraussetzung für die Verfügbarkeit, Zuverlässigkeit und Unfallsicherheit (Safety).

=> Mehr hierzu: Vorlesung MGSE (SS 2012)

Eigenschaften von Software

- Robustheit:
  - Toleranz gegenüber nicht spezifizierter Bedienung / nicht spezifizierten Rahmenbedingungen

    - auch nicht robuste Software kann korrekt sein
    - nicht robuste Software kann leicht nutzlos werden
    - wesentliche Robustheitsanforderungen sollten deshalb spezifiziert werden

Eigenschaften von Software

- Performanz:
  - Erfüllung der Anforderungen an Antwortzeitverhalten, Ressourcenbedarf
  - prinzipielle Überprüfungsmethoden:
    - Messen
    - Berechnen
    - Simulieren
  - häufiger Umgang mit der Eigenschaft Performanz:
    - Erstellen einer initialen Version
    - Verbesserung zum Zweck des Erreichens der notwendigen Performanz
    - Problem: deutliche Verbesserungen erfordern zuweilen grundlegendes Redesign

Eigenschaften von Software

LEHRSTUHL 14
SOFTWARE ENGINEERING

- Benutzungsfreundlichkeit (Usability)
  - Software ist benutzungsfreundlich, wenn die Anwender sie für einfach benutzbar halten
    - Gestaltung der Dialoge
    - einfache Konfigurierbarkeit
    - einleuchtender Aufbau (so weit sichtbar)
- Alles andere als das Empfinden der Anwender zählt nicht!
- Häufiger Ansatz:
  Standardisierung der Benutzungsoberflächen

Eigenschaften von Software

- Wartbarkeit:
  - leichte Handhabbarkeit einer Software nach ihrer Auslieferung
- Arten der Wartung:
  - korrektive Wartung: Beseitigung von Fehlern
  - adaptive Wartung: nachträgliche Anpassung an neue Anforderungen, z.B. gesetzliche Änderungen, neue Organisation
  - perfektive Wartung: Verbesserung im Hinblick auf nicht-funktionale Anforderungen, z.B. Performanz, Ergonomie
- Wartbarkeit hängt stark von Strukturierung ab
- Wartbarkeit nimmt zumeist mit der Lebensdauer der Software ab
  - Lehmans Law of Increasing Software Entropy: die Struktur der Software nimmt mit zunehmender Lebensdauer ab.

Eigenschaften von Software

- Wiederverwendbarkeit:
  - Wahrscheinlichkeit, mit der Software in einem anderen Kontext wiederverwendet werden kann (oft bezogen auf Komponenten)
- Wiederverwendbarkeit entsteht nicht zufällig, sondern muss geplant sein (infrastrukturell unterstützt werden)
- Beispiele: parametrisierte Datenstrukturen, Klassenbibliotheken

Eigenschaften von Software

- Portierbarkeit
  - die Portierbarkeit einer Software ergibt sich aus dem Aufwand, der nötig ist, um eine Software auf einer anderen Plattform (Datenbank, Betriebssystem, Mobiles Gerät, etc.) lauffähig zu machen (im Verhältnis zu ihrem Entwicklungsaufwand)

- hoher Grad an Portierbarkeit durch Verkapselung von Plattformabhängigkeiten, vgl. Ansätze der modellgetriebenen Softwareentwicklung

Eigenschaften von Software

- Interoperabilität:
  - eine Software ist interoperabel, wenn sie sich mit geringem Aufwand mit anderer Software integrieren lässt
- PC-Werkzeuge sind mittlerweile großenteils interoperabel
- hochintegrierte Software-Systeme sind meist weniger interoperabel, weil sie „alles“ können
  - Beispiel: integrierte Entwicklungsumgebungen, Workflow-Management-Systeme und Datenmodellierung
- Interoperabilität wird immer mehr zu einem Muss, weil kaum noch Software in völlig neuen Gebieten eingesetzt wird
  - Beispiel: Interoperabilität mit Office-Werkzeugen, Großrechner-Software, Datenmodellierungswerkzeugen

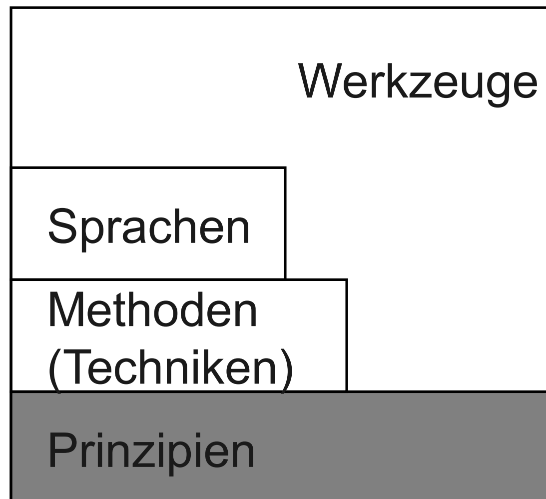
Software Engineering Prinzipien

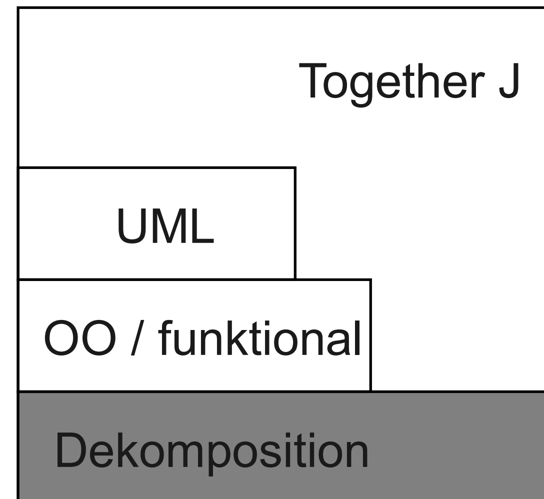
- Überblick (nach GJM91)
  - Striktheit und Formalität
  - Strukturierung
  - Modularität
  - Abstraktion
  - Änderbarkeit
  - Allgemeinheit
  - Inkrementalität

[GJM91]: C. Ghezzi, M. Jazayeri, D. Mandrioli, Fundamentals of Software Engineering, Prentice-Hall, 1991)

Prinzipien, Methoden, Sprachen, Werkzeuge

- Prinzip
  - Grundsatz, den man seinem Handeln zugrundelegt

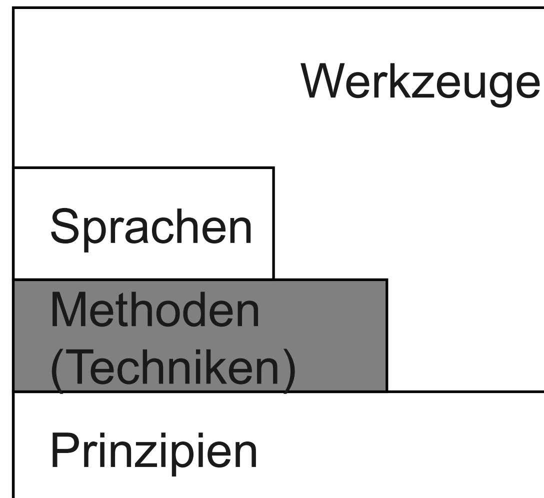


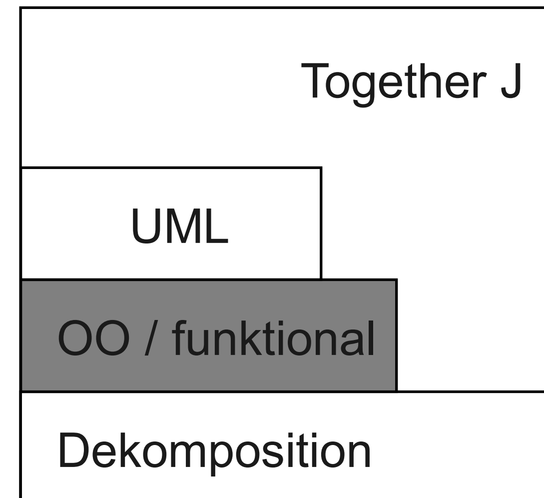


Prinzipien, Methoden, Sprachen, Werkzeuge

LEHRSTUHL 14
SOFTWARE ENGINEERING

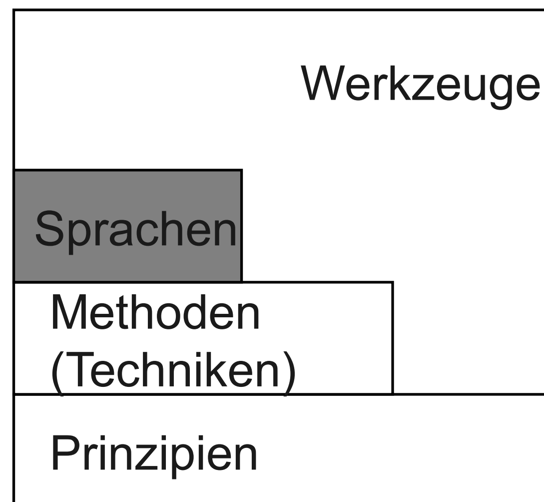
- Technik
  - Vorschrift zur Durchführung einer Tätigkeit (Was ist wie zu tun?)
- Methode
  - Planmäßig anwendbare, begründete Technik zur Erreichung vorgegebener Ziele (Was ist wie und unter welchen Rahmenbedingungen zu tun, so dass ein gutes Ergebnis erreicht wird?)

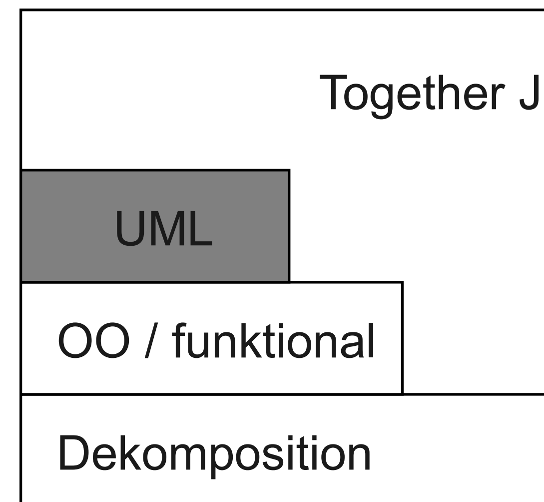




Prinzipien, Methoden, Sprachen, Werkzeuge

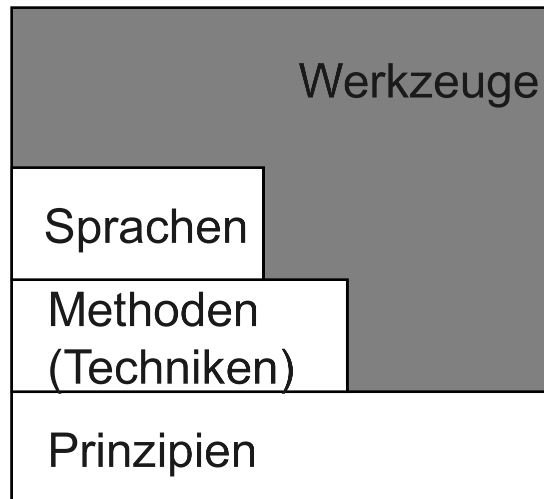
- Sprache
  - syntaktische Regeln zur Unterstützung einer Methode oder Technik. Eine Sprache besteht aus ihrer Syntax und ihrer Semantik:
    - formale Sprachen haben eine formale Syntax- und Semantikdefinition (Aussagenlogik, Prädikatenlogik)
    - semiformale Sprachen haben eine formale Syntax, aber keine klar definierte Semantik (UML)
    - informale Sprachen haben weder eine formale Syntax noch eine formale Semantik (Deutsch, Englisch)

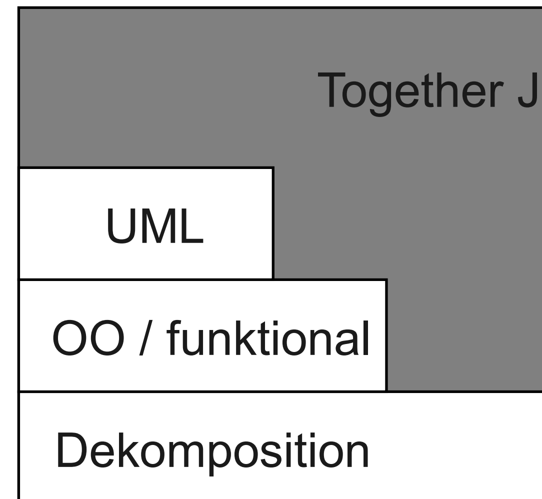




Prinzipien, Methoden, Sprachen, Werkzeuge

- Werkzeuge
  - Rechnerunterstützung für Techniken und Methoden

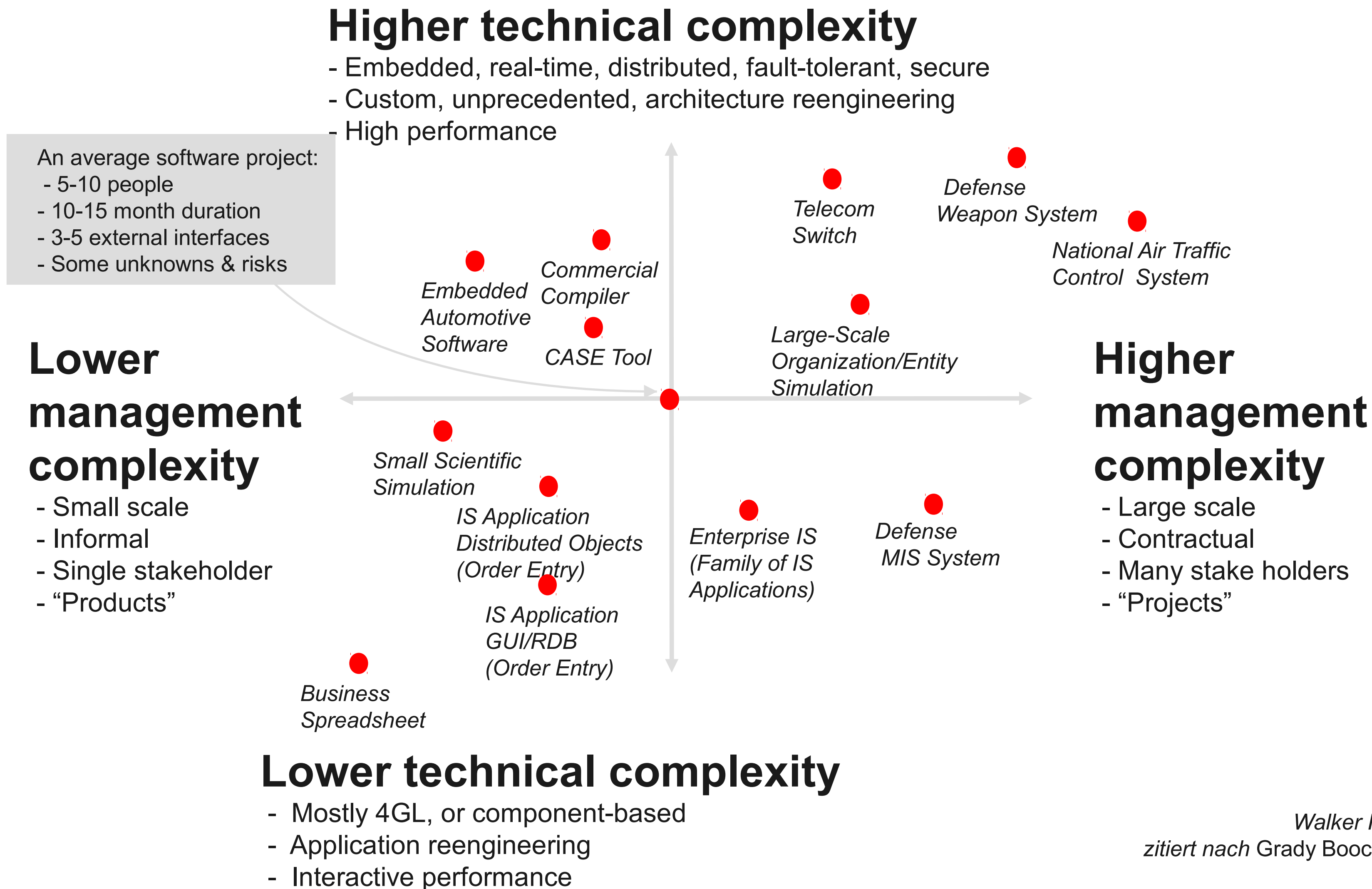




Prinzip: Strukturierung

- Prinzip: Strukturierung (separation of concerns / Unterteilung in Aspekte)
  - Ziel: Komplexität handhabbar machen...
  - Beispiel: Dekomposition ist eine Art der Strukturierung durch Unterteilung

Dimensionen der Softwarekomplexität



Walker Royce, Rational
zitiert nach Grady Booch, *Vortrag 2007*

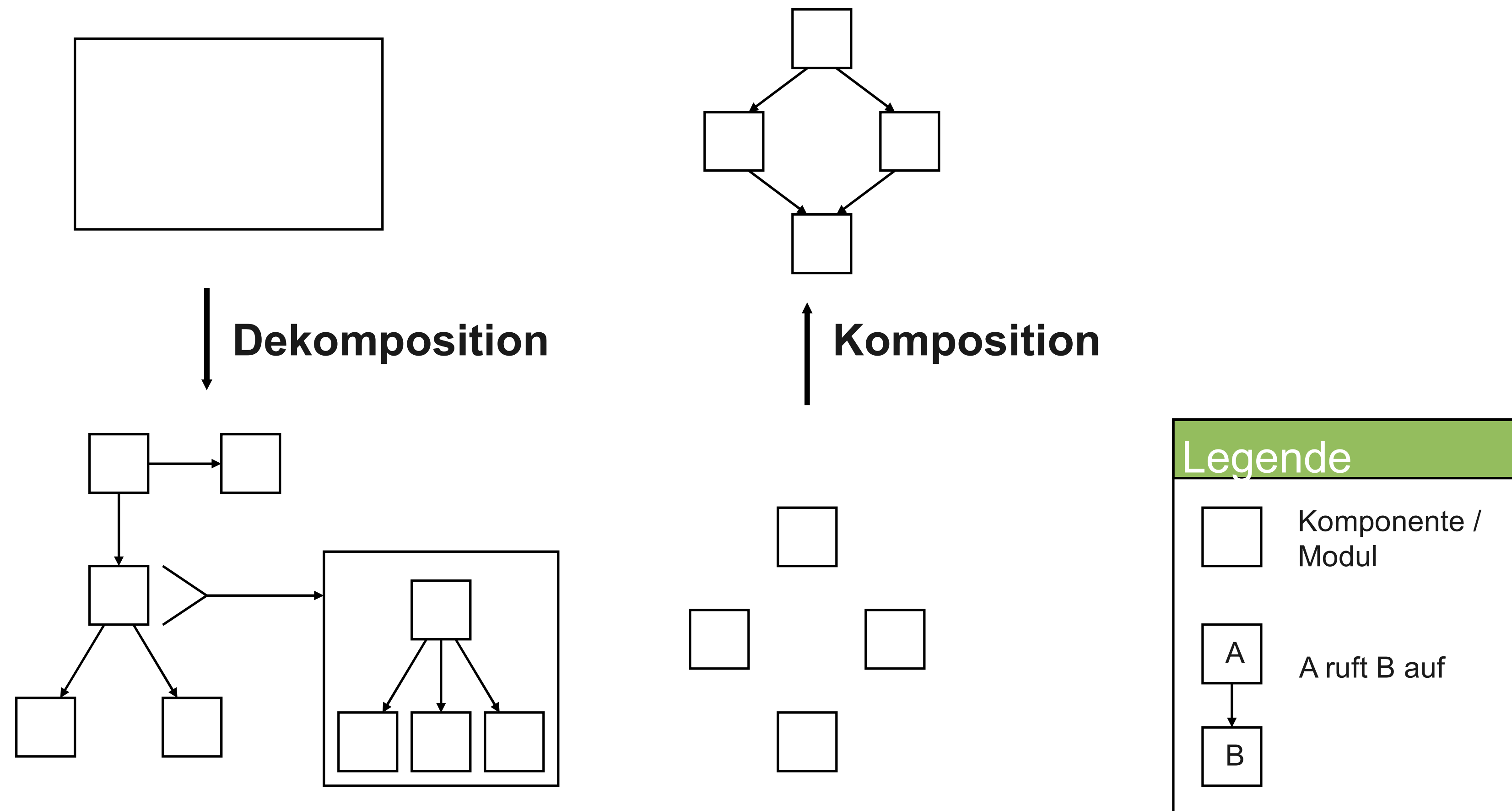
Prinzip: Strukturierung

- Prinzip: Strukturierung (separation of concerns / Unterteilung in Aspekte)
  - Beispiele von Aspekten
    - Funktionalität
    - Robustheit
    - Performanz und Ressourcenverbrauch
    - Organisation der Software-Entwicklung
    - Konfigurations-Management
    - Datensicherheit
  - Abhängigkeit zwischen Aspekten
    - Performanz und Datenintegrität
    - Funktionalität und Zielplattform

Prinzip: Strukturierung

- Arten der Unterteilung
  - zeitliche Unterteilung: Anforderungsanalyse / Entwurf / Programmierung / Test
  - qualitative Unterteilung: Effizienz / Robustheit / Korrektheit / Sicherheit
  - perspektivische Unterteilung: Verwendung von Datenstrukturen / Datenfluss / Kontrollfluss
  - Dekomposition (Unterteilung in Bestandteile): Komponente 1, Komponente 2 (-> diese Unterteilung ist so wesentlich, dass sie unter dem Prinzip Modularität gesondert betrachtet wird!)

Prinzip: Strukturierung



Prinzip: Strukturierung

- Bewertung des Prinzips Strukturierung:
  - +: Überschaubarkeit und Handhabbarkeit
  - -: globale Optimierungen geraten außer Sicht
  - Pragmatik: übergreifende Entscheidungen müssen vor der Unterteilung und der lokalen Bearbeitung der Aspekte getroffen werden

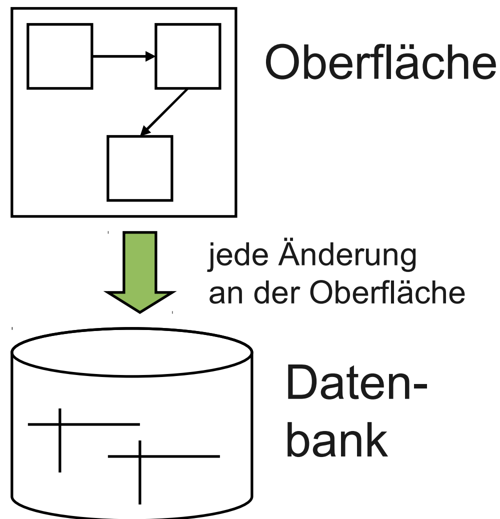
Prinzip: Strukturierung

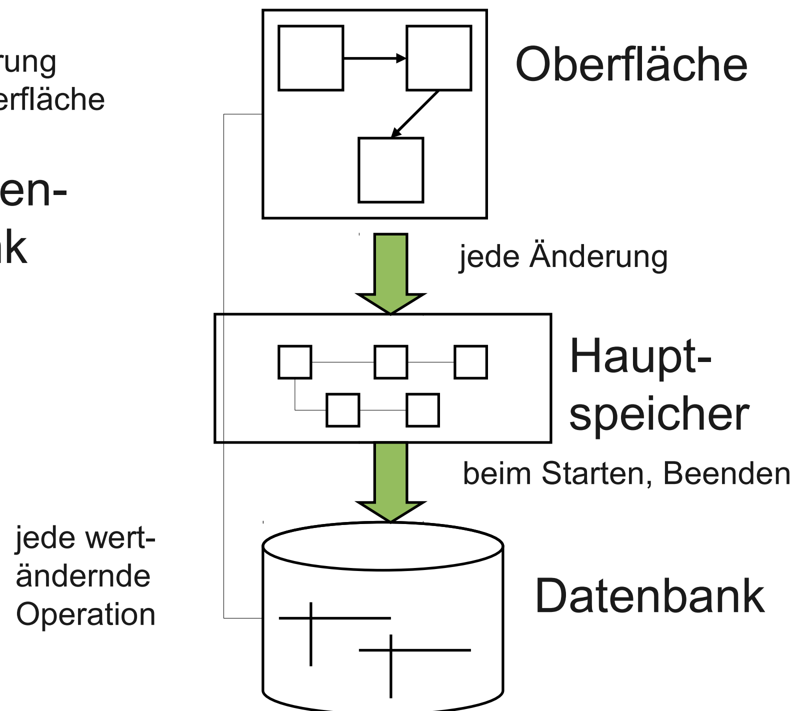
- Beispiel für notwendige, übergeordnete Entscheidungen:

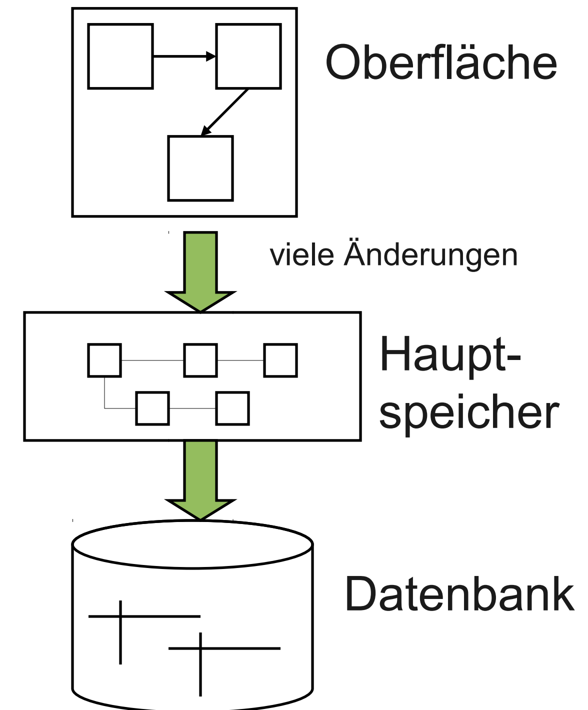
  Entwicklung eines graphischen Editors, der die editierten Objekte in einer Datenbank ablegt

  - Aspekt Datenintegrität: jede Änderung eines Objektes wird in der Datenbank gespeichert, jede Abfrage arbeitet auf der Datenbank
  - Aspekt Performanz: Zu Beginn der Bearbeitung eines Datenmodells wird eine Kopie des Datenbankinhalts in den Hauptspeicher geladen, Änderungen werden an der Hauptspeicherkopie vorgenommen, am Ende der Session werden Hauptspeicherdaten in die Datenbank übernommen
  - Nötig: übergeordnete Entscheidung, z.B. DB am Anfang in Hauptspeicher, werteändernde Operationen werden in Datenbank und Hauptspeicher vorgenommen
  - Dann: unabhängige Überprüfung der Integritäts- und Performanzanforderungen.

Prinzip: Strukturierung







Prinzipien des SWE

- Prinzip: Modularität
  - Unterteilung eines komplexen Systems in Komponenten / Module
  - Anwendungsbeispiele
    - Schreiben eines Berichts
    - Bau eines Autos
  - Modularität als Ergebnis der Dekomposition
  - Modularität als Grundlage der Komponierbarkeit
  - Modularität als Voraussetzung für Wiederverwendung
  - Modularität als Voraussetzung für lokale Änderungen
  - Bestandteile eines Moduls
    - Operationen
    - Objekttypen
    - Beziehungen zwischen Objekttypen

Prinzip: Modularität

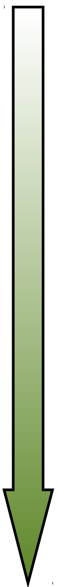
- Ziel
  - hohe Kohäsion: enger interner Zusammenhalt innerhalb eines Moduls
    => Wiederverwendung in kleinen Teilen
  - geringe Kopplung: wenig Wechselwirkungen mit anderen Modulen
    => lokale Änderbarkeit

Prinzip: Modularität

- Kohäsion und Kopplung

Kohäsionsgrade (innerhalb eines Moduls)	**Kopplungsgrade** (zwischen Modulen)
keine Bindung zwischen Bestandteilen	call by value (Datenfluss)
thematisch verwandte Funktionen	call by reference (Datenfluss)
zeitliche Bindung	Kontrollfluss
prozedurale Bindung	gemeinsame Datentypen / Objekttypen

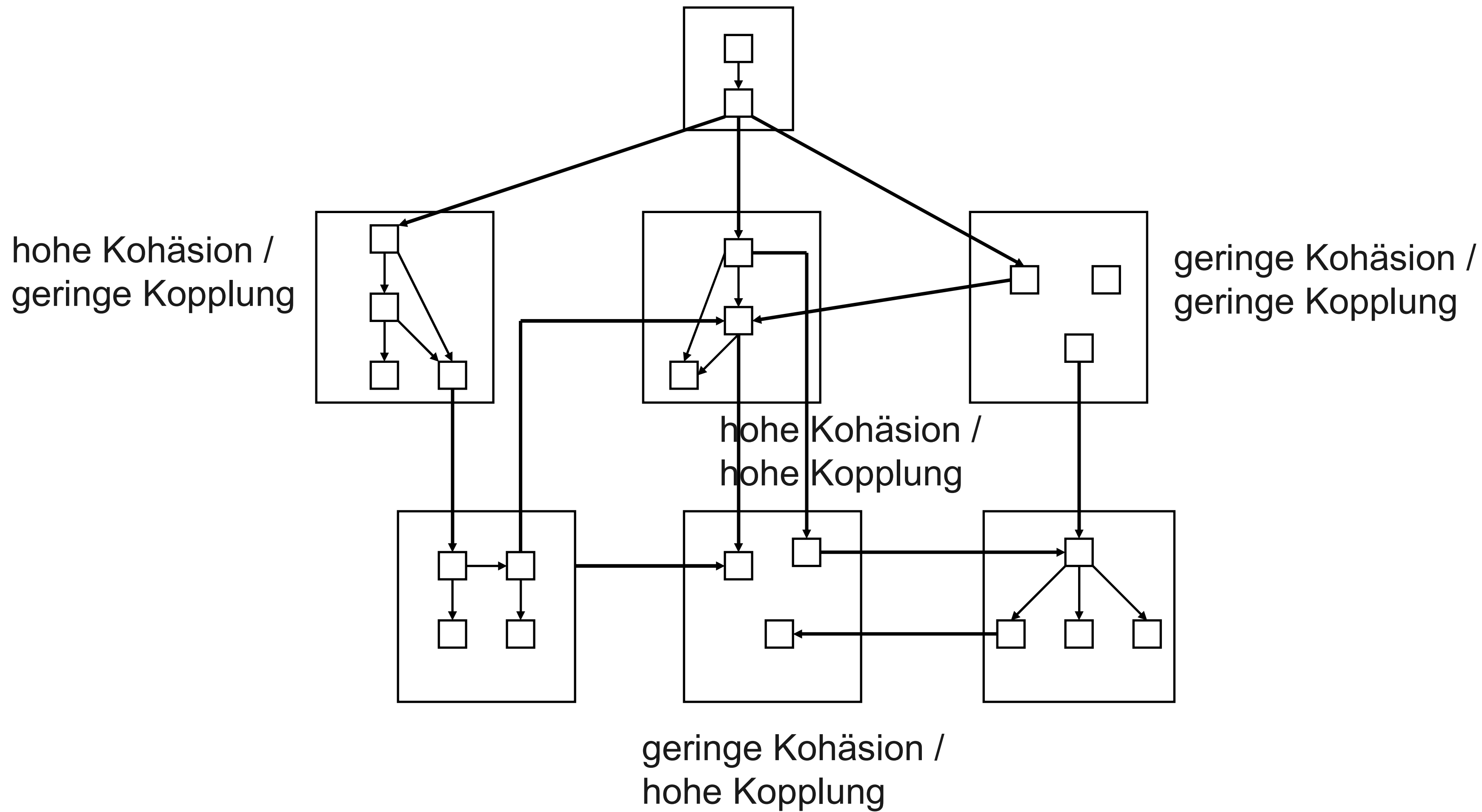
niedrig



hoch

Prinzip: Modularität

LEHRSTUHL 14
SOFTWARE ENGINEERING



LEHRSTUHL 14
SOFTWARE ENGINEERING

Prinzipien des SWE

- Prinzip: Abstraktion
  - Trennung von wichtigen und unwichtigen Merkmalen
  - „Wegabstraktion“ der unwichtigen und Betonung der wichtigen zum Zweck der Konzentration auf das Wesentliche
  - Gängige Abstraktionen
    - die Signatur einer Operation abstrahiert von der Realisierung der Operation
    - die Konstrukte einer Programmiersprache abstrahieren von Prozessorendetails
    - ein Datenflussdiagramm abstrahiert von den Aufrufstrukturen zwischen Komponenten

Prinzipien des SWE: Abstraktion

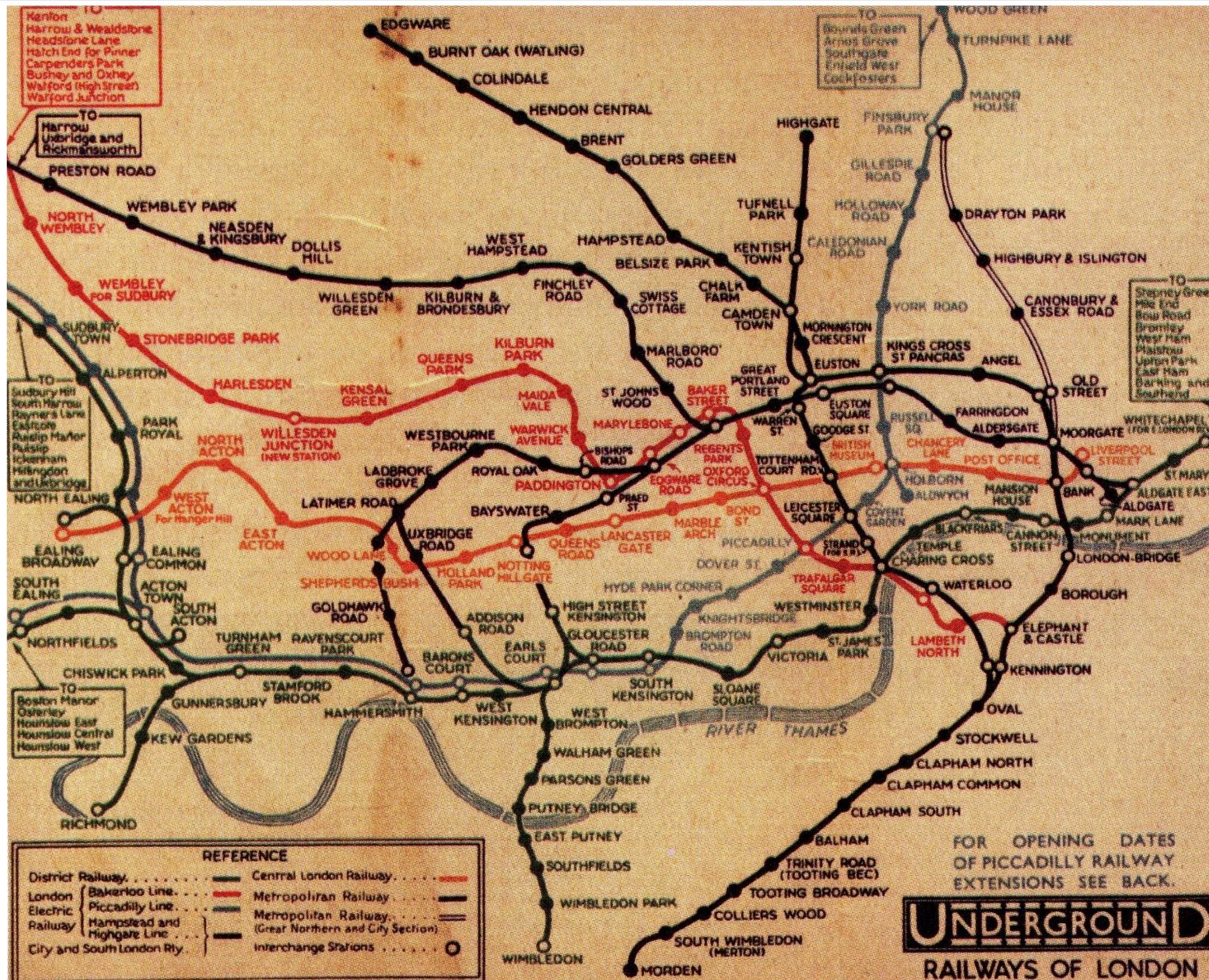
- Wichtigkeit und Unwichtigkeit ist relativ zum Zweck der Abstraktion

Abstraktion Straßenkarte	wichtig	unwichtig
für mit dem Auto Reisende	Straßen	Wassertemperatur
für mit dem Zug Reisende	Gleise, Hauptbahnhöfe	Straßen

Abstraktion Software	wichtig	unwichtig
für den Anwender	Oberflächen, Funktionalität	Interne Struktur
für den Vorstand	Rentabilität	Plattform
für den Entwickler	Schnittstellen zwischen Komponenten	Schulungsaufwand für neue Software

Prinzipien des SWE: Abstraktion

LEHRSTUHL 14
SOFTWARE ENGINEERING



Prinzipien des SWE: Abstraktion

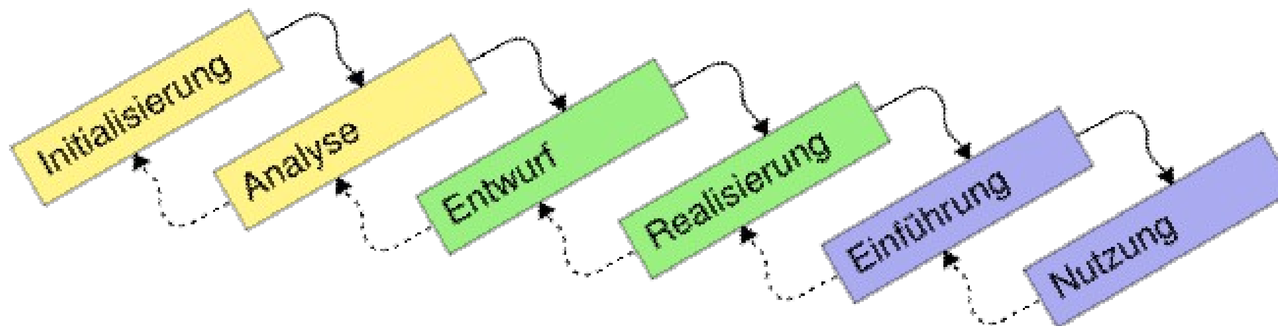
Prinzipien des SWE

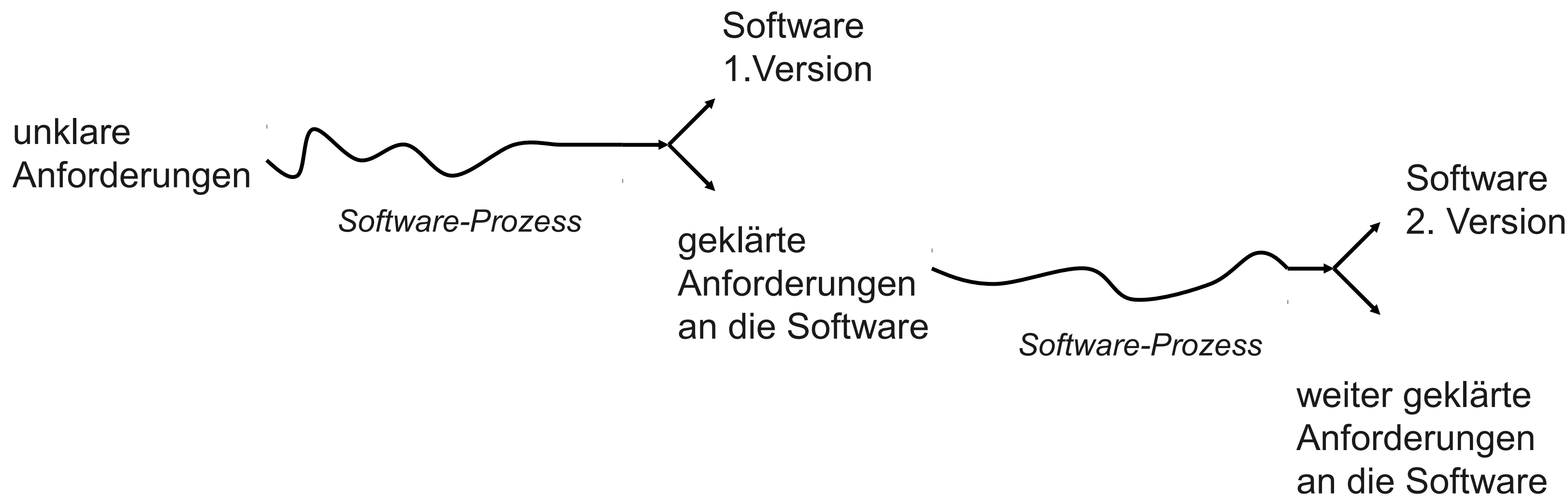
LEHRSTUHL 14
SOFTWARE ENGINEERING

- Prinzip: Annahme der Änderungsnotwendigkeit
  - Software ist Gegenstand von Änderungen. Ursachen für Änderungen:
    - Beseitigung von Fehlern (korrektive Wartung)
    - Verbesserung nicht-funktionaler Eigenschaften (perfektive Wartung)
    - Erweiterung der Funktionalität wegen sich ändernder Rahmenbedingungen (adaptive Wartung)
    - Erweiterung der Funktionalität wegen Erkenntnisgewinn während der Entwicklung

Prinzip: Annahme der Änderungsnotwendigkeit

LEHRSTUHL 14
SOFTWARE ENGINEERING





→ **Software-Prozess ist *kein* typischer Fertigungsprozess !**

Prinzip: Annahme der Änderungsnotwendigkeit

- Chancen der Änderbarkeit:
  - flexible Erfüllung zusätzlicher Kundenwünsche
  - Produktfamilien
- Risiken der Änderbarkeit:
  - Vgl.: „Grüne-Bananen“-Ansatz für SW-Entwicklung und Vertrieb
- Entwurf änderbarer Software (wesentlicher Unterschied zu Fertigungsprozessen)
- Häufige Änderungen als Ursachen des Konfigurations-Managements
- Änderungen als Ursache für Wartungsintensität und Wartung als (fast) eigenständiges Berufsbild

Prinzipien des SWE

- Prinzip: Allgemeinheit
  - Software wird durch Änderungen für verwandte Aufgaben und Zwecke eingesetzt. Beispiele:
    - SAP R3 Komponenten
    - Billing und Accounting Systeme
  - „Allgemeine“ Software lässt sich für verschiedene, verwandte Zwecke einsetzen. Rahmenbedingungen können als Parameter ergänzt werden.
    - SAP R3 Komponenten sind an Organisationsstrukturen anpassbar.
    - Billing und Accounting Systeme können durch konkrete Tarife parametriert werden.
    - Wohnungswirtschaftssysteme können im Hinblick auf ihren Abrechnungsmodus „eingestellt“ werden.
  - Analogie zur Entwicklung von Küchen und Autos
  - Allgemeinheit verursacht Aufwand

Prinzipien des SWE

- Prinzip: Inkrementalität
  - Inkrementalität einer Tätigkeit bedeutet, dass diese Tätigkeit in Schritten vorgenommen wird. Nach jedem Schritt wird rückgekoppelt. Beispiele:
    - Schon das erste Kapitel einer Anforderungsdefinition wird vorgestellt, sobald es vorliegt.
    - Es werden einzelne Dialoge entwickelt und abgestimmt, bevor alle Dialoge entwickelt werden.
    - Es werden Prototypen mit unvollständiger Funktionalität vorgestellt.
  - Inkrementalität basiert auf der Annahme, dass Änderungen aufgrund von Rückkopplung (z.B. mit Auftraggebern) nötig sind (und Aufwand verursachen, der minimiert werden soll).

Prinzipien des SWE

- Frage: Können Sie die bereits erwähnten Prinzipien und Ziele in Beziehung setzen ?

Strikheit und Formalität

Strukturierung

Modularität

Abstraktion

Annahme der Änderungsnotwendigkeit

Allgemeinheit

Inkrementalität

Korrektheit
Zuverlässigkeit
Robustheit

Performanz

Benutzungsfreundlichkeit

Wartbarkeit

Wiederverwendbarkeit

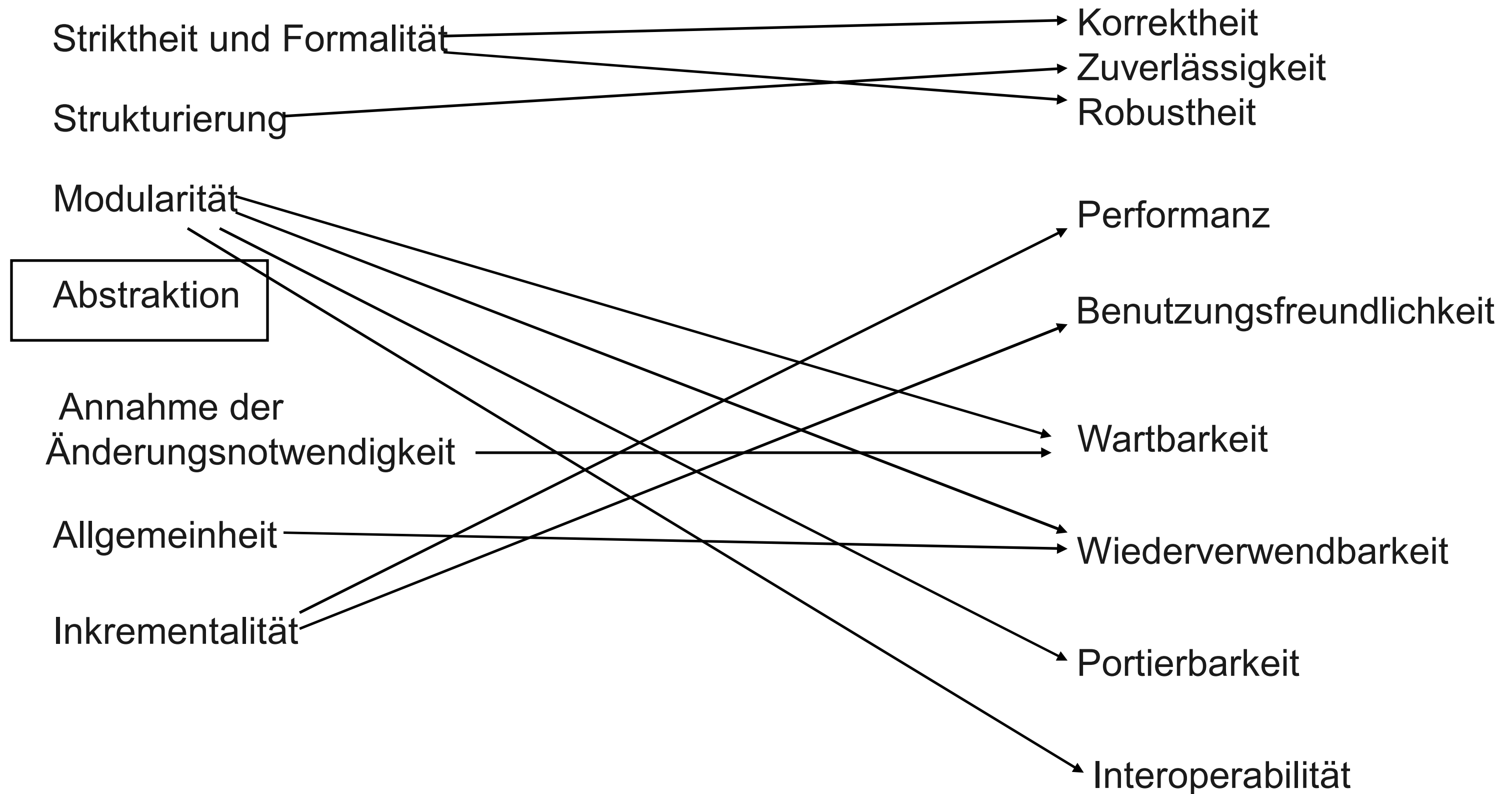
Portierbarkeit

Interoperabilität

Grundlegendes Prinzip, ohne Bezug zu ausgezeichneten Zielen

Prinzipien des SWE

- Prinzipien und Ziele:
  [NB: kein „exaktes“ Bild, weitere Bezüge könnten argumentiert werden]



Abstraktion: Grundlegendes Prinzip, ohne Bezug zu ausgezeichneten Zielen

Zusammenfassung und Ausblick

- Im Teil 2 haben wir besprochen:
  - 40 Jahre Software Engineering
  - Software Engineering - wann braucht man‘s?
  - Eigenschaften von Software
  - Prinzipien des SWE

Insbesondere haben wir gesehen, dass eine der Herausforderungen für das Software Engineering ist, mit der über die letzten Jahrzehnte zunehmenden Komplexität umgehen zu können. Wir haben weiter einige kritische Anforderungen an Software betrachtet, die besondere Herausforderungen für das Qualitätsmanagement mit sich bringen.

Beide Punkte bilden die Motivation, sich in dieser Vorlesung mit zwei wichtigen Themen des Qualitätsmanagements (Spezifikation und Testen) eingehender zu beschäftigen.

Literatur

GJM91: C. Ghezzi, M. Jazayeri, D. Mandrioli, Fundamentals of Software Engineering, Prentice-Hall, 1991)

T. de Marco, Controlling Software Projects, Yourdon Press, 1982

T. de Marco, Why does software cost so much, Dorset House Publishing, 1995