

# Zusammenfassung SE

## Kapitel 1: Einführung; Software-Entwicklung und -Pflege als Problem

- Software ist überall im Leben
- Kosten für Software sehr hoch (halbe Billion pro Jahr auf der Welt). Software nimmt immer grösserer Bestandteil an Kosten für Informatiksysteme im Gegensatz zu Hardware.
- Prozess der Software-Entwicklung:
  - Das Problem verstehen (Anwendungsbereich verstehen, Anforderungen spezifizieren)
  - Lösung entwerfen
  - Lösung umsetzen
  - Lösung umsetzen und in Betrieb nehmen.
- Probleme bei SW-Entwicklung und Pflege: z.B. Aufwand steigt überproportional mit Produktgrösse, nicht linearer Arbeitsprozess, wird von Menschen gemacht (Fehler, Kommunikation steigt überproportional)

## Kapitel 2: Ziele und Qualität

- Formale Zielstrukturierung möglich (AND/OR-Baum)
- Zielsetzung ist notwendig, aber nicht hinreichend → Zielverfolgung.
- 2 Arten von Zielerreichung:

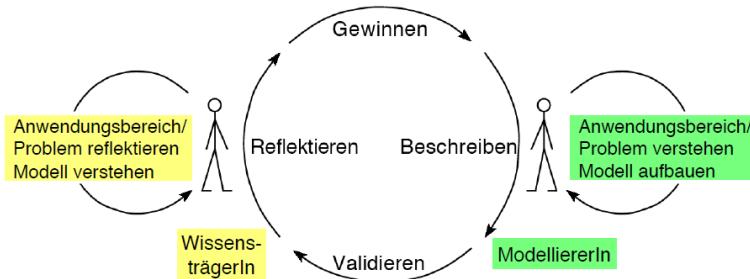


- Feststellung des Erreichen der Ziele via Prüfen (Review; harte Kriterien), Messen (weiche Kriterien, objektiv), Beurteilung (subjektiv, bei vage formulierten Zielen).
- Masse finden (direkt oder indirekt (mit dem Ziel korrelierende Indikatoren))
- Qualität ist Zielerfüllung (=der Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt.)
- Qualität kein absolutes Mass, muss definiert und geschaffen werden.
- SW-Qualitätsmodell mit den Kriterien: Effizienz, Änderbarkeit, Übertragbarkeit, Funktionalität, Zuverlässigkeit, Benutzbarkeit.

## Kapitel 3: Modelle

- Modell Charakteristika
  - Nicht werteneutral: durch modellierende Person und den Verwendungszweck beeinflusst.
  - Größtmögliche Ähnlichkeit zwischen Original und Modell kein Ziel: Bewusst Abstraktion.

- Validierung erforderlich: relevante Eigenschaften des Originals adäquat und vollständig abgebildet?
- ≠ Ausschnitt der Realität; vielmehr Konstruktion und Abbild der Realität.
- Modellierung ist das Fundament des SW-Engineering. Software =Modell?
- Modellbildungsprozess: Wissensträger und Modellierer. Mehrere reale Personen pro Rolle; oder auch eine Person hat beide Rollen. Iterativer Prozess.



- Deskriptive (bestehendes Original modellieren) vs. Präskriptive (zu gestaltendes Objekt modellieren) Modelle.
- Für Modelle/Modellierungssprachen siehe Info2.

## Kapitel 4: Spezifikation von Anwendungen

- DEFINITION Anforderungsspezifikation: Eine systematisch dargestellte und gegebenen Kriterien genügende Sammlung von Anforderungen, typischerweise an ein System oder eine Komponente.
- DEFINITION Requirements Engineering: Das systematische, disziplinierte Spezifizieren und Verwalten von Anforderungen mit dem Ziel:
  - Die Wünsche und Bedürfnisse der Interesseneigner zu verstehen.
  - Einen Konsens über die relevanten Anforderungen zu erzielen und diese Anforderungen systematisch zu dokumentieren und zu verwalten.
  - Das Risiko zu minimieren, dass das zu entwickelnde System die Wünsche und Bedürfnisse der Interesseneigner nicht erfüllt.
- Der Aufwand für das Requirements Engineering soll umgekehrt proportional zum Risiko sein, das man bereit ist, einzugehen.
- Mehrdeutiger Begriff Pflichtenheft.
- Kosten für Fehlerbehebung steigen mit der Verweildauer im System exponentiell an.
- Qualitätsmerkmale einer guten Anforderungsspezifikation (Undef. Anforderungen → undef. Qualität):
  - **Adäquat** – beschreibt das, was der Kunde will bzw. braucht
  - **Vollständig** – beschreibt alles, was der Kunde will bzw. braucht
  - **Widerspruchsfrei** – sonst ist die Spezifikation nicht realisierbar
  - **Verständlich** – für alle Beteiligten, Kunden wie Informatiker
  - **Eindeutig** – vermeidet Fehlinterpretationen
  - **Prüfbar** – feststellen können, ob das realisierte System die Anforderungen erfüllt
  - **Risikogerecht** – Umfang umgekehrt proportional zum Risiko, das man eingehen will



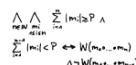
**Adäquat** – beschreibt das, was der Kunde will bzw. braucht



**Vollständig** – beschreibt alles, was der Kunde will bzw. braucht



**Widerspruchsfrei** – sonst ist die Spezifikation nicht realisierbar



**Verständlich** – für alle Beteiligten, Kunden wie Informatiker



**Eindeutig** – vermeidet Fehlinterpretationen

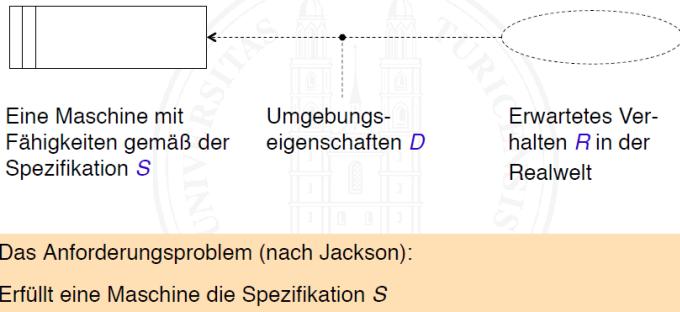


**Prüfbar** – feststellen können, ob das realisierte System die Anforderungen erfüllt

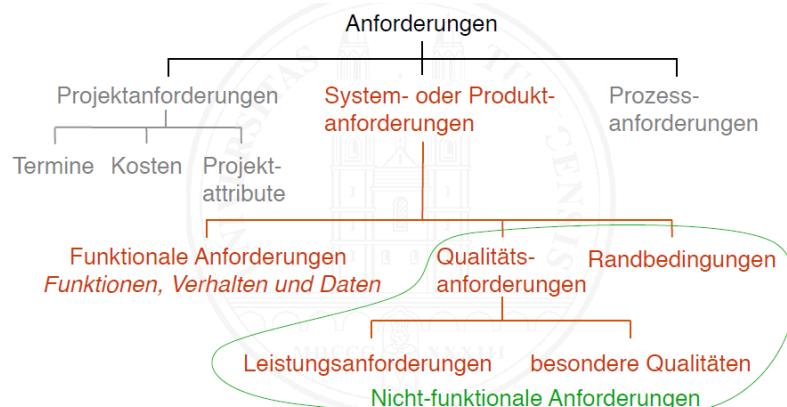


**Risikogerecht** – Umfang umgekehrt proportional zum Risiko, das man eingehen will

- Lösungen von Anforderungen können selbst wieder Anforderungen sein (also ein neues Problem).
- Anforderungsproblem nach Jackson:

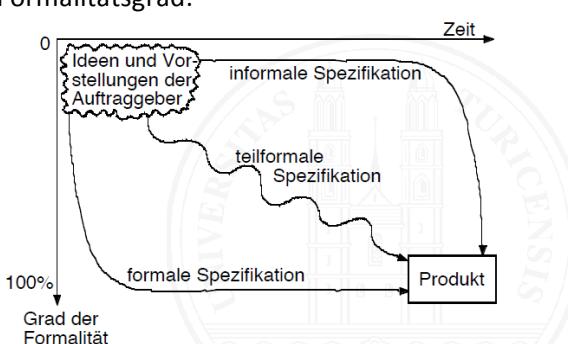


- System gebaut für den innerhalb der Kontextgrenze (Kontext=Derjenige Teil der Umgebung eines Systems, der für das Verständnis des Systems und seiner Anforderungen relevant ist).
  - System < Kontext < (Kontextgrenze) < Anwendungsbereich < (Systemgrenze) < Realität.
- Anforderungen können unterschieden werden in:
  - Repräsentation
    - Operational (Bsp: Der Kontostand wird angezeigt)
    - Quantitativ (Bsp: Antwortzeit <0,5 sec)
    - Qualitativ (Bsp: Das System muss hoch verfügbar sein)
    - Deklarativ (Bsp: Das System muss auf einer Linux-Plattform laufen)
  - Erfüllung
    - Hart (binäre Erfüllung) oder weich (graduelle Erfüllung)
  - Rolle
    - Vorschrift (an das System), Tatsache (Fakten in Systemumgebung) oder Annahme (über Verhalten von Akteuren in Systemumgebung)
  - Art

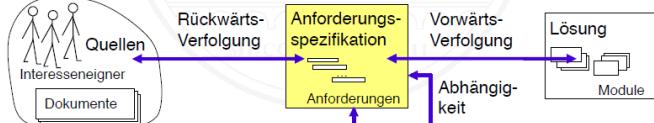


- Funktionale Anforderung (Spezifikation von Systemverhalten, Daten, Eingaben oder Reaktionen auf Eingaben)
- Qualitätsanforderungen:
  - Leistungsanforderung (z.B. Restriktionen bezgl. Datenmenge oder Reaktionszeit)
  - Besondere Qualität
- Randbedingung (Ist irgendeine andere Restriktion zu beachten?)
- Inhalt einer Anforderungsspezifikation:
  - Funktionaler Aspekt:
    - Daten: Struktur, Verwendung, Erzeugung, Speicherung, Übertragung, Veränderung.

- Funktionen: Ausgabe, Verarbeitung, Eingabe von Daten.
  - Verhalten: Sichtbares Dynamisches Systemverhalten, Zusammenspiel der Funktionen
  - Fehler: Normalfall und Fehlerfälle.
- Leistungsaspekt (wenn möglich messbare Angaben machen und Durchschnitts- und Extremwerte angeben):
  - Zeiten „....Reaktionszeit < 0,5 s...“
  - Mengen „....veraltet bis zu 10 000 Kunden...“
  - Raten „....verarbeitet maximal 100 Transaktionen/s...“
  - Ressourcen „....benötigt 2 GByte Hauptspeicher...“
  - Genauigkeit „....berechnet auf vier Nachkommastellen genau...“
- Qualitätsaspekt:
  - Geforderte besondere Qualitäten (z.B. Zuverlässigkeit, Änderbarkeit, usw)
- Randbedingungsaspekt:
  - Einzuhaltende/zu verwendende Schnittstellen, Plattformen, Nachbarsysteme,...
  - Normen und Gesetze
  - Datenschutz, Datensicherung
  - Kulturelles: zum Beispiel internationale Verständlichkeit von Symbolen
  - Explizite Vorgaben des Auftraggebers
  - usw...
- Anforderungsspezifikation kann konstruktiv oder deskriptiv sein. Zudem kann je nach Bedürfnis der Formalitäts- resp. der Detailierungsgrad erhöht werden. Es existieren eine IEEE-Norm und Firmeninterne Richtlinien zur Darstellung.
  - Deskriptiv: Funktion als Blackbox.
    - Natürliche Sprache: Die Funktion Kontostand liefert den Stand des Kontos für eine eingegebene Kontonummer.
    - Formale Notation auch möglich. Mit Pre- und Postbedingungen.
    - Nur Für Darstellungen der Anforderungen kleiner Probleme geeignet.
  - Konstruktiv: Funktion als Whitebox. Bsp: Datenflussdiagramm
    - Modell als Idealisierte Lösung (→ Gefahr von implementierungsorientierter Spezifikation)
    - Leicht verständlich, Teilung eines grösseren Problems in kleinere.
    - → Gut für Modellierung von Anforderungen im Grossen.
  - Formalitätsgrad:
    - **informal**, in der Regel deskriptiv mit natürlicher Sprache
    - **formal**, deskriptive und konstruktive Verfahren möglich
    - **teiformal** mit konstruktiven, anschaulichen Modellen
- Anforderungsermittlung: Dialog mit Stakeholder (typischerweise Auftraggeber oder Rollen), aufzeigen von Möglichkeiten, IST-Zustand erheben, Marktpotential klären etc.
- Aufgrund von Wichtigkeit der Stakeholder und der Höhe der Auswirkung, falls Anforderung verfehlt wird, den Aufwand rechtfertigen.



- Aufwand für Requirements Engineering sollt umgekehrt proportional zum Risiko sein, das man bereit ist, einzugehen.
  - Wert der Anforderung ist indirekt: Geringe Kosten für Nacharbeit, Risikobeherrschung, höhere Kundenzufriedenheit (Kunden zahlen nicht direkt für Anforderungsspezifikation)
  - Techniken für Validierung:
    - Review
      - Walkthrough: Autor führt durch das Review.
      - Inspektion: Gutachter prüfen eigenständig, danach Sitzung.
      - Autor-Kritiker-Zyklus: Kunde liest und kritisiert, bespricht danach mit Autor.
    - Prüf- und Analysemittel in Werkzeugen
      - Auffinden von Lücken und Widersprüchen.
    - Simulation / Animation
      - Untersuchung des dynamisches Systemverhaltens
      - Spezifikation durch Simulator ausgeführt oder durch Animator
    - Prototyp
      - Beurteilung der Adäquatheit (=praktische Brauchbarkeit) in der geplanten Einsatzumgebung
      - Sehr gutes aber auch sehr aufwändiges Mittel zur Validierung
    - Formale Verifikation / Model Checking
      - Formaler Beweis kritischer Eigenschaften
  - Spezifikation kann mit formaler (meist algebraisch), informaler (natürliche Sprache) oder teilformaler (z.B. Klassendiagramm) Sprache/Ausdrucksweise gemacht werden. Jede Art hat seine offensichtliche Vor- und Nachteile.
    - Unterschied Klassen- Objektdiagramm:
      - Objektmodelle **anstelle** von Klassenmodellen: große **Vorteile**, wenn
        - **verschiedene Objekte der gleichen Klasse** zu modellieren sind
        - ein Modell **hierarchisch** in Komponenten **zerlegt** werden soll
- 
- Weitere Option: Spezifikation mit Anwendungsfällen → Modellierung der Interaktion zwischen System und Umgebung.
    - DEFINITION. Anwendungsfall: Eine durch genau einen Akteur angestossene Folge von Systemereignissen, welche für den Akteur ein Ergebnis produziert und an welchem weitere Akteure teilnehmen können.
    - Kann wieder durch natürliche Sprache oder teilformal (z.B. durch Zustandsautomat) gemacht werden.
  - Spezifikation von Randbedingungen meist in natürlicher Sprache.
  - Anforderungen müssen priorisiert werden (kritisch, wichtig, nebensächlich). Nicht alle sind gleich wichtig. Um der wechselnden Umwelt gerecht zu werden, sollten sehr kurze Rückkoppelungszeiten eingerichtet werden (1-6 Wochen).
  - Explizites Requirements Engineering:
    - Konfigurationsmanagement für Anforderungen:
      - Anforderungen identifizierbar machen, geordneter Änderungsprozess erstellen → klare Zuständigkeiten und Verantwortlichkeiten.
    - Verfolgbarkeit
      - Rückwärts: Wo kommt welche Anforderung her?
      - Vorwärts: Welche Anforderung ist wo implementiert?
      - Weiter: wie hängen Anforderungen voneinander ab?



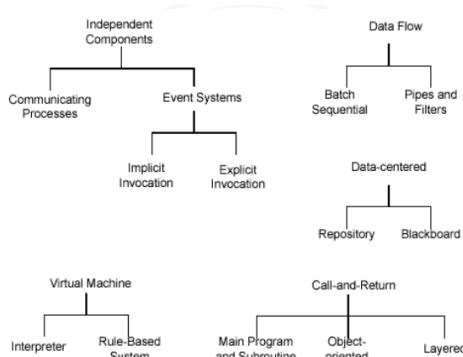
## Kapitel 5: Entwurf von Software

- Entwurfsprinzipien:
  - Struktur: Gliedern der Lösung in Komponenten und Interaktion
  - Abstraktion: Verstehen durch systematisches vergröbern/verfeinern.
    - Vier Arten von Abstraktion:
      - Komposition
      - Benutzung
      - Spezialisierung
      - Aspektbildung
    - Kapselnde Dekomposition:
      - Jeder kleine Teil sollte ohne /mit möglichst wenig Kenntnis über das Ganze verstanden werden können.
      - Das ganze sollte ohne/mit möglichst wenig Kenntnis über die einzelnen Teile verstanden werden können.
  - Modularität (Hauptaufgabe): Modul= in sich abgeschlossener Teil des Systems
    - Je höher die Kohäsion, desto besser die Modularisierung.
    - Je tiefer die Kopplung zwischen den Modulen, desto besser.
    - Kommunikation nur über Schnittstellen; Modulinneres von aussen nicht sichtbar.
  - Geheimprinzip:
    - Gegen aussen ist nur bekannt, *was* eine Komponente leistet, aber nicht *wie* sie ihre Leistung erbringt.
    - Fundamental zur Beherrschung komplexer Systeme
    - →Liefert gute Modularisierung.
  - Schnittstellen und Verträge:
    - Schnittstellen werden mit Verträgen mit Rechten und Pflichten beschrieben.
  - Nebenläufigkeit (= Die parallele oder zeitlich verzahnte Bearbeitung mehrerer Aufgaben):
  - Berücksichtigung der Ressourcen:
    - Abschätzung der technischen Machbarkeit
    - Erfüllbarkeit der gestellten Anforderungen (v.a. Leistung)
  - Aspektbildung:
    - Beschreibung von Querschnittsaufgaben des Systems. Typische Aspekte:
      - Datenhaltungskonzept: Datenbankschema usw.
      - Mensch-Maschine-Kommunikationskonzept: Gestaltung der Benutzerschnittstellen.
      - Fehlerbehandlungs-, Fehlertoleranz-, Sicherheitskonzepte
  - Nutzung von Vorhandenem:
    - Verwenden von Standardsoftware, abgeschlossene Teilsysteme, Komponenten, Architektur- und Entwurfsideen, usw...
  - Ästhetik:
    - Gewählter Architekturstil konsequent verwenden.
    - Einfachste und klarste Lösung wählen.

- Qualität:
  - Gute Entwürfe sind effektiv (löst das Problem), wirtschaftlich (gebrauchstauglich, kostengünstig, ...) und softwaretechnisch gut (leicht verständlich, robust, zuverlässig, ...)
  - → Kontinuierliche Prüfungsmassnahmen nötig.
- Aufgaben eines Architekturentwurfs: Aufgabe analysieren, Architektur modellieren und dokumentieren, Lösungskonzept prüfen.
- Ein Lösungskonzept dokumentiert das Ergebnis des Architekturentwurfs.
- Ein System hat zu jedem Zeitpunkt genau eine Architektur, welche sich über die Zeit aber ändern kann → Architektur ist temporär.
- Architektur kann präskriptiv oder deskriptiv sein.
- Architectural Degradation:
  - Architectural drift: Einführung von Design-Entscheidungen in die deskriptive Architektur, welche nicht durch die präskriptive impliziert werden oder in ihr enthalten sind, jedoch dieser nicht widersprechen.
  - Architectural erosion: Einführung von Design-Entscheidungen in die deskriptive Architektur eines Systems, welche der präskriptiven widersprechen.
- Architectural Recovery = Bestimmung der System-Architektur, ausgehend vom Implementationslevel. Dies ist nötig wenn Architectural Degradation möglich ist.
- In komplexen Systemen ist die Interaktion oft wichtiger und schwieriger als die Funktionen der einzelnen Komponenten.
- Architekturstil = Menge aller architektonischen Design-Entscheidungen.
- Architektur-Patterns sind Architekturstile, welche auf ähnliche Probleme angewandt werden können.
  - Bsp: Three-Tiered Pattern. Front erlaubt dem Benutzer mit dem System zu kommunizieren, Middle Tier enthält die Hauptfunktionalität, Back Tier erledigt den Datenzugriff und -speicherung.



- Architectural Model: Artefakt, welches einige oder alle Design-Entscheidungen eines Systems dokumentiert.
- Architectural Visualization: Eine Art, einige oder alle Design-Entscheidungen einem Stakeholder dazustellen.
- Architecture View: Teilmenge ähnlicher architektonischer Design-Entscheidungen.
- Architekturstile:



- Pipe-and-Filter-Style:
  - Filter sind unabhängige Komponenten, welche keinen Status mit anderen Filtern teilen und seine Nachbarn nicht kennt.
  - Pipes stellen den Output eines Filters als Input eines anderen zur Verfügung.
  - Mehr Vor- als Nachteile.
- Layered System-Style:
  - Komponenten sind Programme oder Unterprogramme
  - Connectors sind Methoden resp. Methodenaufrufe.
  - Jede Schicht stellt einen Dienst nur der nächst höheren Schicht zur Verfügung, und bezieht nur von der nächst unteren Schicht einen Dienst  
→ „Zwiebelmodell“
- Blackboard-Style:
  - Komponenten sind Blackboard client Programme
  - Connector ist das Blackboard, welches das geteilte Speichermedium darstellt.
  - → Alle Komponenten kommunizieren via den gespeicherten Daten auf dem Blackboard.
- Event-based Systems und Implicit Invocation-Style:
  - Komponenten sind Programme, welche Interesse an einem Event angemeldet haben.
  - Connector ist die Event broadcast and registration Infrastruktur.
  - Implicit Invocation, wenn Programm von Broadcast informiert wird, wenn ein Event stattgefunden hat. Explizit, wenn Programm nur dann Information bekommt, wenn es explizit nachfragt.
- Distributed P2P Systems:
  - Komponenten sind unabhängig entwickelte Objekte oder Programme, welche öffentliche Operationen oder Dienste zur Verfügung stellen.
  - Connectors sind entfernte Prozedur-Aufrufe über das Netzwerk.
- Design-Patterns sind Lösungsschablonen für ähnliche Probleme, die immer wieder auftauchen und immer damit nicht jedes Mal aufs Neue gelöst werden müssen.
- Elemente Eines Design-Patterns.
  - Name
  - Problem (Wann soll das Pattern verwendet werden, Liste von Bedingungen)
  - Lösung (Keine spez. Implementierung, abstrakte Beschreibung)
  - Anwendung und Trade-Offs (Einfluss auf System-Flexibilität, Erweiterbarkeit, Portabilität, usw...)
- Klassifikation von Design-Patterns:

		Purpose		
Scope	Class	Creational	Structural	Behavioral
		Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

- Zweck:
  - Creational: Objekt Erzeugung.
  - Structural: Komposition von Klassen oder Objekten.
  - Behavioral: Interaktion von Klassen und Objekten.
- Bereich:
  - Class: Relation zwischen Klassen und Subklassen durch Vererbung, statisch.
  - Object: Objekt Relationen, dynamischer.

	<b>Creational</b>	<b>Structural</b>	<b>Behavioral</b>
<b>Class</b>	Aufschieben von Teilen der Objekterzeugung zu Subklassen	Verwenden von Vererbung zur Klassen-Komposition	Verwenden von Vererbung um Algorithmen und Kontrollfluss zu beschreiben
<b>Object</b>	Aufschieben zu andern Objekten	Beschreiben Objekt-Komposition	Beschreiben, wie eine Gruppe von Objekten kooperiert, um einen Task auszuführen, den kein einzelnes Objekt allein erbringen kann.

- Design-Patterns vs. Frameworks
  - Framework = eine Menge kooperierender Klassen, die ein wiederverwendbares Design für eine spezifische Klasse von Software darstellen.
  - Designpattern sind abstrakter, haben keinerlei architektonische Elemente und sind weniger spezialisiert als Frameworks.
- Modellierungsarten:
  - Strukturorientiert:
    - Kriterien: Namensraum, Übersetzungseinheit.
    - Güte: zufällig
  - Funktionsorientiert:
    - Jedes Modul berechnet eine Funktion,
    - Güte: gut für rein zustandsfreie und rein funktionale Probleme und zur Submodularisierung von Klassen und abstrakten Datentypen. Sonst zu schwach.
  - Datenorientiert:
    - Fasst Datenstruktur und alle darauf möglichen Operationen zusammen.
    - Güte: gut
  - Objektorientiert:
    - Repräsentiert Objekt des Problembereichs oder benötigtes Informatik-Element
    - Güte: Wenn Klassen als ADT konzipiert sind gut, sonst mässig bis schlecht.
  - Komponentenorientiert:
    - Stark gekapselte Menge zusammengehöriger Elemente, die eine gemeinsame Aufgabe lösen und als Einheit von Dritten benutzt werden.
    - Güte: sehr gut.

- Schnittstellen müssen nach aussen sichtbar und dokumentiert sein. Ein Modul ist ausschliesslich über die Schnittstelle(n) zugänglich. Die Implementierung ist nach aussen unbekannt.
- Es existieren zwei verschiedene Arten von Modulen: Ein Dienstleistungsmodul erbringt die angebotenen Leistungen vollständig selbst, während ein Agentenmodul dazu Leistungen von Drittponenten benötigt (→ Angebots- und Bedarfsschnittstellen nötig)
- Schnittstellen können analog zu Klassen vererbt werden. Es existieren Steinbruchvererbung, Spezialisierung, Substituierbarkeit.
- Schnittstellen mit Verträgen: Vertrag = Menge von Zusicherungen.
- Vier Arten von Zusicherungen: Voraussetzungen, Ergebniszusicherung, Invarianten, Verpflichtungen.
- Modul nimmt Vertragstreue des Verwenders an! Voraussetzungen werden vom Modul nicht geprüft sondern als erfüllt angenommen.
- Am besten teilformale Verträge formulieren.
- Zusicherungen können dynamisch geprüft werden.
- Zusammenarbeit (definiert wesentlich den Entwurfsstil):
  - Leistungserbringung (Delegieren von Aufgaben)
  - Informationsaustausch (Bring-, Hol-, Abonnementsprinzip)
  - Informationsteilhabe (gemeinsame Speicher, Datenabstraktion, Informationsdepot)

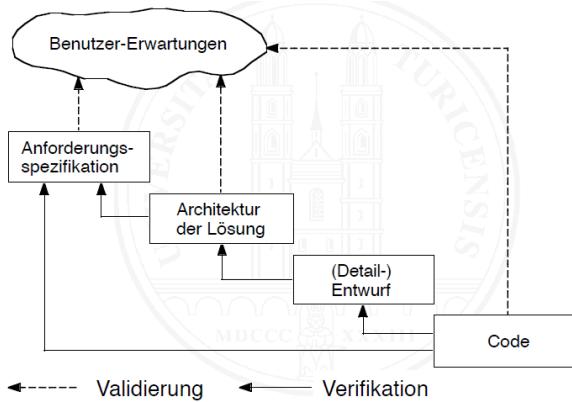
## **Kapitel 6: Systematisches Programmieren; Lesbare und änderbare Programme schreiben.**

- Problem: Programm hat mehr Leser als Schreiber und oft sind Schreiber ≠ Pfleger. Deshalb: Leichte Lesbarkeit ist wichtiger als leichte Schreibbarkeit. → Systematisches Programmieren.
- Namen für Variablen /Methoden usw. sollten so gewählt werden, dass etwas darunter vorzustellen ist. Viele verschiedene Standards und Konventionen für Namensgebung.
  - Wichtig: Codierrichtlinien der Organisation beachten und Stile bzw. Sprachen (engl, de) nicht mischen.
  - Kurznamen (i, x usw.) nur in Schleifen oder einfachen math. Formeln verwenden.
  - Faustregel: 8-20 Zeichen für Variablen, 15-30 Zeichen für Prozeduren/Methoden
  - Wahl der Namen der Variablen und Methoden entscheidend für das Verständnis des Codes.
- Globale vs. Lokale Variablen
  - Global: einfachste Form der Kommunikation zwischen Programmteilen. Nachteile: ungewünschte Nebeneffekte und stark verschlechterte Versteh- und Änderbarkeit.
  - Gültigkeits- und Sichtbarkeitsbereich der Variablen so klein wie möglich halten.
- Konstante (wie z.B. pi) als Variable deklarieren.
- Konsistenz und Verarbeitungssicherheit durch Verwendung von Typen und Typprüfung gewinnen.
- Statische und dynamische Struktur stimmen bei gut strukturierten Programmen überein.
  - Statements wie break sollten nur verwendet werden, wenn dies beibehalten werden kann.
- Geschlossene Ablaufkonstrukte: Element des Programmablaufs mit genau einem Eintritts- und einem Austrittspunkt. Sequenz, Alternative, Iteration. Alles kann daraus zusammengesetzt werden → Aneinanderreihung und Verschachtelung.

- Verschachtelung: Blockklammern und ELSE verwenden. Kein IF ohne ELSE. Große Fallunterscheidungen mit switch (o.ä.) nicht mit vielen if's.
- Schleifen:
  - Abweisende Schlaufen: Prüfung der Schleifenbedingung vor dem Durchlauf (while, for)
  - Annehmende Schlaufen: Prüfung der Bedingung erst nach der Ausführung des Schleifenkörpers (do-while). Falsch, wenn die Schleifenbedingung a priori nicht erfüllt ist.
  - Besser Abweisende Schleifen, weil sie sicherer sind.
  - Fälle kein Durchlauf, ein Durchlauf und max. Anzahl Durchläufe durchspielen; immer Blockklammern verwenden; Annehmende nur wenn minimal ein Durchgang.
  - Schleifeninvariante: Prädikat, das nach jeder Prüfung der Schleifenbedingung wahr ist.
  - Prüfen, ob Schleife terminiert.
  - Verifizieren der Korrektheit einer Schleife:
    - (i) Prädikat (=Voraussetzung für Schleife S) → Invariante
    - (ii) Invariante vor letztem Durchlauf ∧ Invariante nach letztem Durchlauf → Prädikat (=Erwartetes Ergebnis von S)
    - (iii) S terminiert.
- Rekursion vs. Iteration
  - Vorteile Rekursion: einfacher und kürzere Lösungen; Korrektheit einfacher zu zeigen als bei Schleifen (zu verifizieren: Rekursionsformel und Rekursionsverankerung)
  - Nachteile Rekursion: Laufzeit- Speicherprobleme (z.B. Fibonacci-Berechnung); Rekursion gedanklich schwieriger nachzuvollziehen als Schleife.
  - Beides kann ineinander transformiert werden.
- Unterprogramme (abgegrenztes, benanntes und aufrufbares Codestück) dienen der besseren Lesbarkeit und der Effizienz. → Gliederung in Unterprogramme= Entwurfsaufgabe. Formen:
  - Benannter Block: syntaktisch separiert, benannt und aufrufbar, kein separater Namensraum, keine lokalen Daten, keine Parametersetzung.
  - Prozedur: separater Namensraum, lokale Daten, Parametersetzung, Datenaustausch über globale Variablen möglich, statische Bindung
  - Methode: Wie Prozedur, aber dynamische Bindung. Polymorphie möglich.
  - Makro: eigentlich ≠ Unterprogramm, in maschinennahen Sprachen.
  - Unterprogramm kann Anweisung sein oder Wert zurückliefern (selten beides).
- Optimierung 3 Regeln:
  - Tu es nicht! // Code nicht auf Verdacht optimieren (Jackson)
  - Wenn du es dennoch tust oder tun musst, dann tu es später! // Zuerst messen; Flaschenhälse erkennen; Falls nötig, durch gezielte lokale Optimierung die Flaschenhälse beseitigen. (Jackson)
  - Tu es vorher! // Ersparen des Optimierens durch Wahl guter Datenstrukturen und effizienter Algorithmen.
- Dokumentation für bessere Änderbarkeit und Verstehbarkeit.
- Dokumentation muss: konsistent mit dem Code sein, kein Nachbeten des Codes sein, Coderichtlinien einhalten, nicht übermäßig sein.
- Gute Dok. beschreibt: Intention des Programms und für Daten, getroffene Annahmen, Bedeutung von Schnittstellen. Beinhaltet: Untertitel, Hinweise, Erläuterungen, Hinweise.

## Kapitel 7: Validierung von Verifikation

- Validierung: Der Prozess der Beurteilung eines Systems oder einer Komponente mit dem Ziel, festzustellen, ob die spezifischen Anforderungen erfüllt sind.
- Verifikation: Der Prozess der Beurteilung eines System oder einer Komponente mit dem Ziel, festzustellen, ob die Resultate einer gegebenen Entwicklungsphase den Vorgaben für diese Phase entsprechen. ODER: formaler Beweis der Korrektheit eines Programmes.



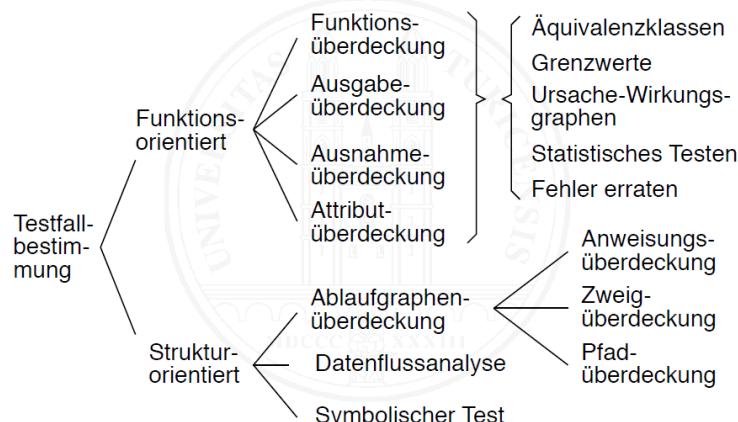
- Fehler, Fehlerursachen und Ausfälle müssen auseinander gehalten werden → saubere Terminologie.
  - Eine Person begeht einen **Irrtum** (mistake)
  - Als mögliche Folge davon enthält die Software einen **Defekt** (defect, fault)
  - Wird der Defekt durch Inspizieren der Software gefunden, so ergibt das einen **Befund** (finding)
  - Bei der Ausführung von Software mit einem Defekt kommt es zu einem **Fehler** (error): Die tatsächlichen Ergebnisse weichen von den erwarteten / den richtigen ab
  - Dies kann zum **Ausfall** (failure) eines software-basierten Systems führen
  - Wird ein Fehler festgestellt, so muss die Fehlerursache gefunden und behoben werden (**Fehlerbeseitigung, Debugging**)
- Prüfen von Software muss geregelt, dokumentiert, reproduzierbar etc. sein.

## Kapitel 8: Testen von Software

- Testen ist ein Prozess mit der Absicht, Fehler zu finden. Aber: Korrektheit des Programms kann allein durch testen nicht garantiert werden (ausser in trivialen Fällen).
- Meist kann aus Gründen der Zeit oder der nicht vorhandenen Kapazitäten nicht vollständig getestet werden.
- Testen findet nur Symptome, aber keine Fehlerursachen. Es können auch nicht alle Merkmale getestet werden (z.B. Wartbarkeit).
- Testsystematik:
  - Laufversuch: Der Entwickler testet. Sind die Ergebnisse falsch, sucht er die Defekte und behebt diese. Test fertig, wenn Ergebnisse vernünftig aussehen.
  - Wegwerf-Test: Jemand testet, ohne System. Bei erkannten Fehlern werden Defekte gesucht und behoben. Test fertig, wenn Tester findet Programm ist ok.

- Systematischer Test: Spezialisten Testen. Test geplant, Testfälle festgelegt, Fehlersuche und Fehlerbehebung verlaufen separat, etc..
- Testgegenstand und Testarten:
  - Komponententest, Modultest
  - Integrationstest
  - Systemtest
  - Abnahmetest (Nicht: Fehler finden. Sondern: Kunden zeigen, dass System funktioniert)
- Testablauf:
  - Planung (was, wann, wie, wo, wer, etc...)
  - Vorbereitung (Auswahl der Testfälle, Bereitstellen Testumgebung, Erstellen der Testvorschrift)
  - Durchführung (Testfälle nach Vorschrift testen, Ergebnisse notieren, Prüfling während des Tests nicht verändern)
  - Auswertung (Testbefunde zusammenstellen)
  - (Fehlerbehebung als nicht Bestandteil des Tests!)
- Auswahl von Testfällen zentrale Aufgabe des Testens. Müssen repräsentativ, fehlersensitiv, redundanzarm und ökonomisch sein.
  - Ziel: Mit möglichst wenigen Testfällen möglichst viele Fehler finden.

### Bestimmen von Testfällen



- Zwei Klassen von Testverfahren:
  - Funktionsorientierter Test (Blackbox-Test)
    - Testfallauswahl aufgrund der Spezifikation
    - Programmstruktur kann unbekannt sein.
  - Strukturorientierter Test (Whitebox- Test)
    - Testfallauswahl aufgrund von Programmstruktur
    - Spezifikation muss ebenfalls bekannt sein (wegen erwarteter Resultate).
- Auswahl Testfälle (Funktionsorientiert).
  - Äquivalenzklassen: Aus jeder „Eingabeklasse“ einen Repräsentanten auswählen.
  - Grenzwertanalyse: Grenzfälle der zulässigen Daten nehmen.

- Ursache-Wirkungs-Graphen: systematische Bestimmung von Eingabedaten, welche ein gewünschtes Ergebnis produzieren.
  - Beispiel: In folgendem Fragment soll die Ausnahme erzeugt werden

```
...  
if (bufferEmpty || refill < minLevel) {  
    for (i:=0; i<= nMax; i++) {  
        if (i >= 512 && sane) throw new OverflowException ("charBuffer")  
    ...
```

```

graph LR
    A[bufferEmpty  
oder  
refill < minLevel] --> B[nMax ≥ 512]
    B --> C[i ≥ 512]
    C --> D[sane]
    D --> E[throw exception]
    
```
- Statistisches Testen (random testing): zufällige Wahl von Eingabedaten. Dafür grosse Anzahl an Testfällen. Automatisches Testen mit Orakel (vergleicht SOLL und IST) notwendig.
- Fehler raten (error guessing): Intuitive Auswahl, ergänzt andere Verfahren, Qualität stark von Erfahrung des Tester abhängig.
- Strukturorientiert: systematisches Durchlaufen des Programms.
  - Bestimmung der Programmzweigen und -Pfaden:
    - If-und Schlaufen-Anweisungen haben 2 Zweige, switch soviele Zweige wie Fälle
    - Pfade: Alle Kombinationen aller Programmzweige bei maximalem Durchlauf der Schleifen.
  - Güte des Tests:
    - Abhängig von Überdeckung und erreichtem Überdeckungsgrad
    - Überdeckungsgrad = Prozentuales Verhältnis der Anzahl überdeckter Elemente (z.B. Zweige) zur Anzahl vorhandener Elemente (z.B. Zweige).
    - Anweisungsüberdeckung: schwaches Kriterium. Fehlende Anweisungen werden nicht entdeckt.
    - Zweigüberdeckung: wird in Praxis angestrebt. Dennoch: falsch formulierte Bedingungsterme werden nicht entdeckt.
    - Pfadüberdeckung ist in fast allen Programmen, die Schleifen mit Verzweigungen enthalten, nicht testbar.
- Testvorschrift wichtigstes Dokument. Weiter: Testprotokoll und Testzusammenfassung.  
Aufbau einer Testvorschrift – 1

## 1. Einleitung

### 1.1 Zweck

Art und Zweck des im Dokument beschriebenen Tests

### 1.2 Testumfang

Welche Konfigurations-Einheiten der entwickelten Lösung getestet werden

### 1.3 Referenzierte Unterlagen

Verzeichnis aller Unterlagen, auf die im Dokument Bezug genommen wird

## 2. Testumgebung

### 2.1 Überblick

Testgliederung, Testgüte, Annahmen und Hinweise

### 2.2 Testmittel

Test-Software und -Hardware, Betriebssystem, Testgeschirr, Werkzeuge

### **2.3 Testdaten, Testdatenbank**

Wo die für den Test benötigten Daten bereit liegen oder bereitzustellen sind

### **2.4 Personalbedarf**

wieviel Personen zur Testdurchführung benötigt werden

### **3. Annahmekriterien**

Kriterien für

- erfolgreichen Test-Abschluss
- Test-Abbruch
- Unterbrechung und Wiederaufnahme des Tests

### **4. Testfälle**

(siehe unten)

- Testfälle müssen nummeriert werden, zu Testabschnitten (mit Zweck, Vor- und Nacharbeit) zusammengefasst. Evtl. weiter Testsequenzen.
- Testgeschirr aus Testtreiber und Teststrümpfe zum Testen von unvollständiger Software.
  - Treiber ruft Prüfling auf und versorgt ihn mit Daten, Resultatentgegennahme und – Protokollierung.
  - Teststrumpf: Berechnet oder simuliert Ergebnisse einer vom Prüfling aufgerufenen Operation.
- Integration von Komponenten. Tests:
  - Aufwärtsintegration: Beginnt mit elementaren Komponenten / Braucht kein Strümpfe, nur Treiber.
  - Abwärtsintegration: Beginnt mit hohlem Gesamtsystem / Braucht keine Strümpfe, nur Treiber.
  - → Mischformen auch möglich.
- Besondere Testformen:
  - Testen von Leistungsanforderungen (z.B. Leistungstest, Lasttest, Stresstest, Ressourcenverbrauch)
  - Besondere Qualitäten
- Wann Test beenden?
  - Wenn mit der Testvorschrift keine Fehler mehr gefunden werden.
  - Wenn Prüfkosten pro Fehler eine im Voraus festgelegt Grenze überschreiten.
  - (bei random testing) Wenn nach gewisser Anzahl Testfälle keine Fehler mehr gefunden werden.
  - Wenn festgelegte Obergrenze für Fehlerdichte unterschritten wird.

## **Kapitel 9: Reviews**

- Review = Eine formell organisierte Zusammenkunft von Personen zur inhaltlichen oder formellen Überprüfung eines Produktteils nach vorgegebenen Prüfkriterien und –listen.
- Ergebnis eines Reviews: Liste mit Befunden.
- Reviewen oft billiger als testen, und sie finden die meisten Fehler. Nicht nur Symptome, sondern auch Fehlerursachen. Nicht jede Software ist testbar, aber jede reviewbar.
- Durch Reviewen wird man besser: Richtiges bestätigen, Arbeitsweise vereinheitlichen, Wissenstransfer von den Dummen zu den Schlauen.
- Reviews sind wirtschaftlich
- Inspektion
  - Prüfling wird von Gutachtern nach vorgegebenen Prüfkriterien inspiziert.
  - Gutachter bereiten sich individuell vor.
  - Arten: Klassische Code-Inspektion nach Fragen, Team Inspektion, N-Individuen Inspektion, Selbstinspektion.

- Team Inspektion (wichtigste): Jeder Gutachter macht individuelle Befundliste, Sitzung dient zum Zusammentragen und Bewerten der Befunde.
- N-Individuen-Inspektion: Gleich wie Team, einfach ohne Sitzung. Nachteile: keine Gewichtung der Befunde, keine Gemeinsame Liste. Vorteil: Billiger.
- Walkthrough: Autor geht Prüfling mit Gutachtern durch.
  - Nur beschränkte Vorbereitung möglich.
  - Erfordern keine klaren Vorgabedokumente. Aber: Weniger wirksam als Inspektion (geringe Fehlerentdeckungsquote). Autor grosser Einfluss durch Präsentation.
- Durchführung besteht aus Planung, Vorbereitung, Sitzung, Überarbeitung und Nachkontrolle.
- An Sitzung werden Befunde nur zusammengetragen – nicht gelöst!
- Aufwand für einen Review ist im Wesentlichen gleich dem Personalaufwand.
  - Gesamtaufwand = Zeit für Organisation + Gesamte Vorbereitungszeit der Sitzungsteilnehmer + Dauer der Sitzung \* Anz. Teilnehmer + Aufwand für Nachreview bei zu vielen Befunden.

## Kapitel 10: Messen von Software

- Messen: Ein interessierendes Merkmal eines Gegenstandes quantitativ erfassen.
- Mass und Repräsentation:

Ein Maß (measure) für das Merkmal M einer Gegenstandsmenge D ist eine Abbildung  $\mu: D \rightarrow S$ , welche jedem  $d \in D$  einen Messwert  $\mu(d)$  auf einer Skala (scale) S so zuordnet, dass M und S strukturähnlich oder homomorph sind.

Strukturähnlich heißt: Zu jeder Relation R auf M gibt es eine Relation  $R^*$  auf S mit

- (1)  $R(M(d_1), M(d_2)) \Rightarrow R^*(\mu(d_1), \mu(d_2))$
- (2)  $R^*(\mu(d_1), \mu(d_2)) \Rightarrow R(M(d_1), M(d_2))$  und die Aussage  $R(M(d_1), M(d_2))$  ist sinnvoll interpretierbar.

(1) und (2) heißen auch Repräsentations- oder Homomorphiebedingung

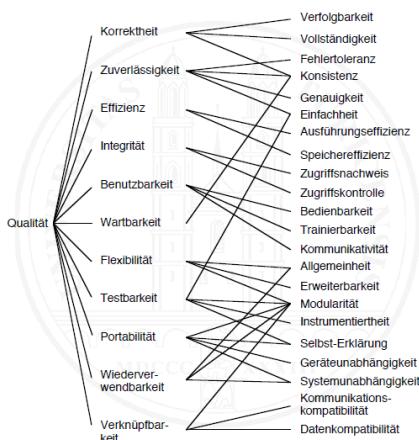
- Homomorphie für Beziehungen und Operationen.
- Mass für Software = Metrik
- Skalentypen:

Typ	erlaubt	Eigenschaften
Nominalskala	= ≠	Reine Kategorisierung von Werten Nur nicht-parametrische Statistik
Ordinalskala	= ≠ < >	Skalenwerte geordnet und vergleichbar Medianwert, sonst nur nicht-parametrische Statistik
Intervallskala	= ≠ < > Distanz	Werte geordnet, Distanzen bestimmbar Mittelwert, Standardabweichung
Verhältnisskala (auch Rational-skala genannt)	= ≠ < > Distanz, (+, -) Vielfaches, %	Werte geordnet und in der Regel additiv* Skala hat absoluten Nullpunkt Übliche parametrische Statistik
Absolutskala	= ≠ < > Distanz, (+, -) Vielfaches, %	Skalenwerte sind absolute Größen Sonst wie Verhältnisskala

\* Verhältnisskalen sind meistens additiv, müssen es aber nicht zwingend sein

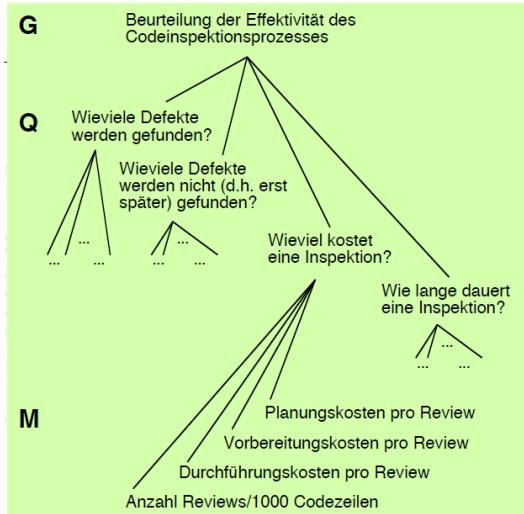
- Direkte Masse wie Kosten oder Durchlaufzeit und Indirekte Masse existieren. Indirekte Masse sind Masse, welche mit dem zu messenden Merkmal korreliert sind und existieren, weil die Messung des eigentlichen Massen zu teuer oder nicht möglich ist (z.B. Benutzerfreundlichkeit, Portabilität).
- Güte eines Massen:

- Validität: Misst das Mass tatsächlich das zu messende Merkmal?
  - Aussagekraft: Sind die Messwerte sinnvoll interpretierbar?
  - Schärfe: Werden wahrnehmbar verschiedene Merkmale auf verschiedene Masse abgebildet?
  - Auswertbarkeit: Welche Auswertungen sind möglich?
  - Verfügbarkeit: Kann dann gemessen werden, wenn es interessant ist? Wie viel kostet die Messung?
  - Stabilität / Reproduzierbarkeit: Wie empfindlich ist das Mass auf Störungen?
- Wichtige Messwerte von Software: Grösse, Komplexität, Zuverlässigkeit, Funktionalität.
- Komplexitätsmasse:
  - Zyklomatische Komplexität (McCabe): Messung des Flussgraphen G eines Programms mit der Formel:  $v(G) = e - n + 2 * p$ ; [e-Anzahl Kanten; n-Anzahl Knoten; p-Anzahl Endpunkte]
    - In einfacheren Programmen berechnet sich  $v(G)$  so:
      - Alle Alternativen und Schleifen zählen (if, while, for, etc)
      - Für jede Auswahl-Anweisung (switch, CASE, etc) die Zahl der Fälle -1 addieren.
      - +1.
    - Verhältnisskala, aber nicht additiv. Komplexität von 2 aneinandergereihten Programmstücken =  $v(G1) + v(G2) - 1$ .
  - Software Science. Krasse Berechnungen, ist aber veraltet und sollte nicht mehr verwendet werden.
- Zuverlässigkeitsmasse:
  - MTTF (= Mean Time to Failure). Das am häufigsten verwendete Zuverlässigkeitsmass.
    - Messung der Zeit zwischen 2 Fehlern.
  - Fehlerdichte gemessen in Anzahl Fehler/1000 NCSS
- Qualitätsmodelle ermöglichen Messen von Qualität mit standardisiertem Modell.
- Qualitätsmodell besteht aus:
  - Menge „allgemeingültiger“ Qualitätsziel e (Faktoren)
  - Satz charakteristische Merkmale zu jedem Faktor
  - Messbare Kenngrößen zu jedem Merkmal.
- Typische Modelle:
  - McCall (1980) [Verfügbarkeit kommt nicht vor. Aber oftmals wichtiger Faktor]



- ISO/IEC siehe Kap.2

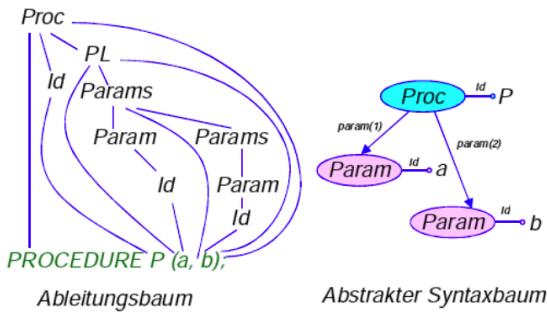
- Ansätze für Definition von Massen
  - Definitorischer Ansatz: Ausgehen von vordefinierten Massen (wie z.B. Halstead) mit der Hypothese, dass diese Masse eine gesuchte Größe messen.
    - → Praktisch, aber Gefahr von sinnlosen Messungen
  - Bequemlichkeitsansatz: Das messen, was einfach zu messen ist.
    - → Einfach und billig, aber Gefahr von sinnlosen Messungen.
  - Zielorientierter Ansatz: Ausgehen von einem zu erreichenden Ziel. Suche nach Massen, welche dieses Ziel quantitativ charakterisieren.
    - → Fokussierte Messung, zielorientierte Interpretation der Messwerte.
- Bekanntester zielorientierter Ansatz: GQM (Goal-Question-Metric)
  - Goal: Festlegen eines Qualitätsziels
  - Question: Fragen, mit welchen Zielerreichung festgestellt wird.
  - Metrics: Masse, mit welchen Fragen quantitativ beantwortet werden können.
  - Vorteile: Wir nur das gemessen, was gebraucht wird. Interpretation der Messergebnisse ist festgelegt.
  - Beispiel:



## Kapitel 11: Statische Analyse

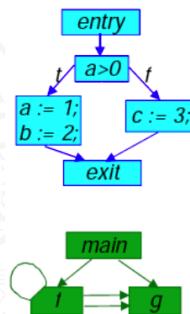
- Statische Analyse ist die Untersuchung des statischen Aufbaus eines Prüflings auf die Erfüllung vorgegebener Kriterien.
- Ziele:
  - Bewertung von Qualität wie Pflegbarkeit aufgrund struktureller Eigenschaften wie Komplexität.
  - Erkennen von Fehlern und Anomalien in neuer Software.
  - Erkennen und Analyse der Struktur zum Reengineering in Legacy Systems.
  - Prüfen, ob ein Programm formale Vorgaben wie Coderichtlinien einhält.
- Vorgehen: Programm wird durch ein Werkzeug geparsst und in einer analysefreundlichen internen Struktur dargestellt. Dabei werden erste Kenngrößen ermittelt und Syntaxfehler erkannt. Danach: verschiedene Analysen wie statische Aufrufhierarchie, Strukturkomplexität, Datenflussanalyse. Befunde sammeln, verdichten, bewerten, essen.

- Weiter können Algorithmen (mit Laufzeiteffizient, Speichereffizient, Gültigkeitsbereich) und Spezifikationen, Entwürfe, Prüfvorschriften (Syntaxanalyse, Flussanalysen, etc.) analysiert werden.
- Darstellung für die Programmanalyse: Abstrakte Syntaxbäume, Kontrollflussanalyse, Datenflussanalyse.
- Abstrakte Syntaxbäume: Darstellung der syntaktischen Dekomposition des Eingabeprogrammes. Sind vereinfachte Ableitungsbäume. Eine Syntaxbeschreibung legt syntaktische Struktur fest (z. B. BNF).

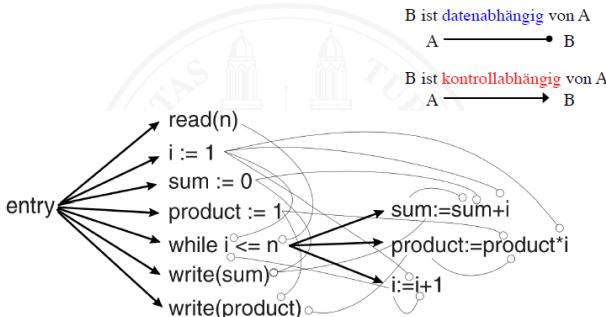


- Zwischendarstellungen sollen einerseits möglichst quellennah sein, aber gleichzeitig auch möglichst vereinheitlicht.
- Kontrollflussanalyse und Datenflussanalyse sind wichtigste Elemente der Darstellung dynamischer Semantiken.
- Kontrollflussinformation:

- **intra-procedural:** Flussgraph
  - Knoten: Grundblöcke
  - Kanten: (bedingter/unbedingter Kontrollfluss
  - ergibt sich aus syntaktischer Struktur und etwaigen Goto's, Exit's, Continue's, etc.
- **inter-procedural:** Aufrufgraph
  - Multigraph
  - Knoten: Prozeduren
  - Kanten: Aufruf
  - ergibt sich aus expliziten Aufrufen im Programmcode sowie Aufrufe über Funktionszeiger
- Knoten D heisst dominiert Knoten N, wenn D auf allen Pfaden von Startknoten zu N liegt. Direkter Dominant, falls D von allen anderen Dominaten von N dominiert wird.
- Postdominanz: D postdominiert Knoten N, wenn jeder Pfad von N zum Endknoten über D führt.
- Kontrollabhängigkeit: Bedingung B, von welcher die welche die Ausführung eines anderen Knotens X abhängt.
  - B darf nicht von X postdominiert sein.
  - B ist der letzte Knoten mit dieser Eigenschaft.
- Datenabhängigkeitsanalyse: Set = Setzen eines Werts; Use = Verwendung eines Werts.  
Daraus ergeben sich folgende Beziehungen zwischen zwei Anweisungen:
  - Set-Use (Datenabhängigkeit): A setzt den Wert, der von B verwendet wird.  
Wichtigste/ Bedeutenste Beziehung.
  - Use-Set (Anti-Dependency): A liest den Wert, der von B überschreibt.
  - Set-Set (Output-Dependency): der von A gesetzte Wert wird von B überschrieben.



- Program Dependency Graph (PDG): Gerichteter Multigraph für Daten- und Kontrollflussabhängigkeit.

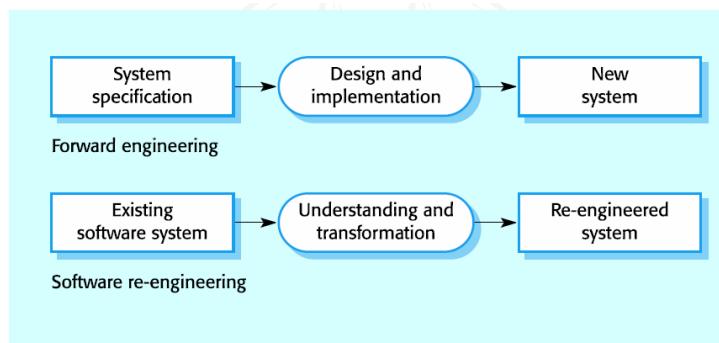


- System Dependency Graph (SDG) ist eine Vernetzung von mehreren PDG's mit interprozeduralen Kontroll- und Datenflusskanten.
- 

## Kapitel 12: Software Evolution und Reengineering

- Software-Anpassung ist unvermeidbar.
  - Neue Ansprüche, Umwelt ändert, Fehler müssen behoben werden, performance muss gesteigert werden etc.
- Verschiedene Lehman's Laws zu Software Evolution (siehe Wikipedia wenn nötig).
- Lehman's System-Typen:
  - S-System: Formal definiert, von einer Spezifikation ableitbar.
  - P-System: Können nicht spezifiziert werden. Real World bleibt unverändert.
  - E-System: In die reale Welt eingebettet und ändert sich simultan zur dieser.
- Software-Pflege mit Änderung bestehender oder Hinzufügen neuer Komponenten.
- Arten von Instandhaltung:
  - Fehler zu beheben.
  - Software an eine andere Umgebung anpassen.
  - Funktionalität modifizieren.
- Instandhaltungskosten 2 bis 100 Mal grösser als Entwicklungskosten → schlechte Instandhaltungsarbeiten machen Software noch komplexer, sodass zukünftige Arbeiten noch schwerer und teurer werden. Alte Software hat oft auch höhere Kosten.
- Kostenfaktoren für Instandhaltungsarbeiten:
  - Team Stabilität: Kosten sind tiefer, wenn das gleiche Personal für längere Zeit damit beschäftigt ist.
  - Vertragliche Verantwortung: Entwickler haben evtl. keine vertragliche Verantwortung um zukünftige durchzuführen, und designen deshalb auch nicht auf gute Pflegbarkeit hin.
  - Fachwissen Mitarbeiter: Oft fehlt das nötige Fachwissen bei unerfahrenen Arbeitern.
  - Programmalter und Programmstruktur: Je älter und oder je strukturell komplizierter/unklarer, desto teurer.
- Total maintenance effort =  $p$  (productive efforts) +  $K$  ( $\text{empirical const}^{(c-d)}$ ) ( $c$ = Komplexität des Systems aufgrund fehlender Struktur und Dokumentation;  $d$ = Vertrautheit der Teams mit der zu bearbeitenden Software)
- Schätzung auch via COCOMO II Formel möglich, aber ein wenig lange Formel.

- Change Identification – Change proposals → Software evolution process – New System → Change identification -- ...
- Dringende Änderungen müssen sofort implementiert werden, ohne die traditionellen Fehlerbehebungsschritte durchzuführen: schlimmer Systemfehler, Änderungen in der Umwelt haben unerwartete Auswirkungen auf das System (z.B. OS-Upgrade), business-change.
- Softwarewiederverjüngung: Redokumentation, Restrukturierung, Reverse Engineering, Reengineering (= Reverse Engineering + Delta + Forward Engineering)
- Reengineering bringt Kostenvorteile (oft ist es billiger als neue Software herzustellen) und Risikoreduktion (gibt sehr grosse Risiken bei der Herstellung neuer Software).



- Reengineering Kostenfaktoren:
  - Menge der Software
  - Tool-support
  - Höhe der Datenkonversion
  - Experten im Team (v.a. bei alten Systemen, welche auf alten Technologien basieren)
- Wichtigste Punkte dieses Kapitels:
  - Software Entwicklung und Evolution sollte ein einziger iterativer Prozess sein.
  - Lehman's Laws beschreiben einige Erkenntnisse der Systemevolution.
  - Es gibt drei verschiedene Arten von Instandhaltung: Fehlerbehebung, Software auf neue Umgebung anpassen und neue Ansprüche implementieren.
  - Meist grössere Pflege als Entwicklungskosten.
  - Motor des Evolutionsprozess sind Ansprüche/Anforderungen von System-Stakeholdern.
  - Der Business-value und die Qualität eines veralteten Systems (Legacy System) sind entscheidend für die zu brauchende Evolutionsstrategie.

## Kapitel 13: Prozesse und Prozessmodelle

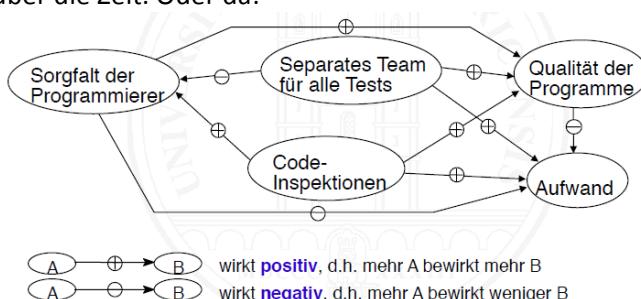
- Ad-hoc Prozesse für kleine Prozesse (SW-Herstellung), mit Einzelperson meistens → Experimentelle Entwicklung.
- Professionell anders → Mit Kostenplan, Qualitätsziele etc.
- SW-Entwicklung ist kein klassisches technisches Entwicklungsprojekt, da SW immateriell.
- Software-Komponente Lebenslauf: Initiierung, Entwicklung, Nutzung.
- Klassen von Software:
  - S-Typ: Vollständig durch Spezifikation beschreibbar.
  - P-Typ: Löst ein abgegrenztes Problem.
  - E-Typ: Realisiert eine in der Welt eingebettete Anwendung.
  - → S-Typ stabil. P- und E-Typ einer Evolution unterworfen.

- Meilenstein: Mittel zur Strukturierung eines Prozesses, Kontrolle des Fortschritts.
- Charakteristische Prozessmodelle:
  - Wasserfall-Modell:
    - Entwicklung als Sequenz aufeinanderfolgender Entwicklung- und Prüfschritte.
    - Phase wird verlassen, wenn geprüftes und akzeptiertes Ergebnis vorliegt.
    - Eher nicht mehr verwenden wegen nicht lokaler Iteration.
  - Ergebnisorientiertes Modell:
    - Entwicklung unterteilt in Sequenz aufeinanderfolgender Zeitabschnitte.
    - Keine Iteration.
    - Abschluss einer Phase mit einem Meilenstein (auch Untermeilensteine möglich).
    - Fördert planvolles Vorgehen, leicht verständlich, gut für frühzeitige Fehlererkennung, geeignet für Projektführung.
    - Ignoriert Evolution, lauffähige Systemteile entstehen erst spät (lange Papierdurststrecke, technische Risiken, ...), System auf einen Schlag in Betrieb genommen.
    - Für nicht so grosse Projekte mit guten Mitarbeitern ohne viel Risiko.
  - Wachstumsmodell:
    - Gliederung der zu liefernden Software in aufeinander aufbauender, betriebsfähigen Lieferungen.
    - Unterteilung in eine Folge von Iterationen.
    - Iterationen (Lieferungen) mit Meilensteinen abgegrenzt. Auch Untermeilensteine innerhalb der Iteration.
    - System immer betriebsfähig. Neue Komponenten werden integriert.
    - Geeignet für grosse Systeme, frühe Entstehung lauffähiger Teile (→ motiviert MA, besser für Nerven des Auftraggebers), Verkürzt Rückkoppelungszeit.
    - Struktur wird vielleicht zerstört durch schrittweisen Aufbau.
  - Spiralmodell:
    - Weiterentwicklung des Wasserfallmodells; aber für grosse, risikoreiche Projekte geeignet.
    - Vierteiliger Zyklus:
      - Planung
      - Zielsetzung/Zielkorrektur
      - Risiko-Untersuchung
      - Entwicklung und Prüfung
- Agile Softwareentwicklung: Verfahren für Entwicklung von Kleinsoftware auf grosse Software übertragen.
- Pflegeprozess:
  - Reaktive und proaktive Erfassung und Analyse der Kundenbedürfnisse.
  - Planung und Durchführung Pflegearbeiten.
- Prototypen = lauffähiges Stück Software, welches kritische Teile eines zu entwickelnden Systems vorab realisiert.
- Prototyping:
  - Explorativ
    - Klärung und Bestimmung von Anforderungen

- Demonstrationsprototyp: der prinzipiellen Machbarkeit und Nützlichkeit von Systemideen.
- Prototyp im engeren Sinne: Erprobbares und kritisierbares Modell einer geplanten Informatiklösung.
- Experimentell
  - Entwicklung von Labormustern
    - Untersuchung der Realisierbarkeit kritischer Systemteile
    - Bewertung von Entwurfsalternativen
- → Demonstrationsprototypen, Prototypen im engeren Sinne und Labormuster sind Wegwerf-Prototypen:
  - Dürfen nicht dokumentiert sein und schnelle, unsaubere Lösungen verwenden.
  - Das Wegwerfen muss sichergestellt werden.
- Evolutionär
  - Pilotensystem
  - Kern des zu entwickelnden Systems
    - Kein Wegwerf-Prototyp
  - Evolutionäres Prototyping entspricht Wachstumsmodell

## Kapitel 14: Software Projektführung

- Zu planen ist alles, was in einem normalen Projekt auch zu planen ist: Prozessmodell, Organisationsstruktur, Personal und Personaleinsatz, Termine und Kosten, Dokumente und Verfahren, ...
- Projektleiter ist Schlüsselfigur, das Team untereinander kann demokratisch oder hierarchisch aufgebaut sein.
- Ideal pro Person nur ein Projekt, sonst Umschaltverluste.
- Gibt Antwort auf sechs W-Fragen:
  - WARUM: Veranlassung und Projektziele
  - WAS: die zu liefernden Resultate (Produktziele)
  - WANN: die geplanten Termine
  - DURCH WEN: Personen und ihre Verantwortlichkeiten
  - WOMIT: die zur Verfügung stehenden Mittel (Geld, Geräte, Software...)
  - WIE: die Vorgehensweise und die Maßnahmen zur Sicherstellung des Projekterfolgs
- Planung mittels Meilensteine.
- Software Projekte sind nicht lineare Systeme. Das sieht man z.B. bei der Kostenverfolgung über die Zeit. Oder da:



- Nach dem Projekt ist es wichtig, dass Dokumentationen, Messungen, Projektgeschichte abgeschlossen, dokumentiert resp. archiviert werden.
  - → für weitere Projekte von Bedeutung.
- Bewertung Risiko: Risikofaktor= Eintretenswahrscheinlichkeit\*Schadenhöhe.

## Kapitel 15: Software-Aufwandschätzung

- Aufwand für grosses Projekt darf nicht auf Basis des Aufwands kleiner Projekte hochgerechnet werden.
- Empirische Schätzverfahren:
  - Basiert auf Erfahrungen und Vergleich mit anderen Projekten
  - Schätzung durch einzelnen Experten (billig und einfach, aber krasse Fehler möglich) oder durch Gruppe (genauer (eliminiert Ausreisser), aber: Kosten)
- Algorithmische Schätzverfahren:
  - Berechnung von Kosten- und Durchlaufzeit-Funktionen.
  - Ohne Masszahlen über abgewickelte Projekte keine zuverlässigen Prognosen.
  - COCOMO
    - Geht von Schätzung der Produktgrösse aus. Grundgleichungen:
      - $MM = 2.4 * KDSI^{1.05}$
      - $TDEV = 2.5 * MM^{0.38}$
    - Für einfache Anwendungsssoftware in stabiler Umgebung.
    - Grundgleichungen für andere:
      - Programmsysteme:
        - $MM = 3.0 * KDSI^{1.12}$
        - $TDEV = 2.5 * MM^{0.35}$
      - Eingebettete Systeme:
        - $MM = 3.6 * KDSI^{1.2}$
        - $TDEV = 2.5 * MM^{0.32}$
    - Durch diese Kostenfaktoren kann der Aufwand unternehmensspezifisch kalibriert werden:

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
<b>Product Attributes</b>						
RELY Required software reliability	.75	.88	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.08	1.16	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
<b>Computer Attributes</b>						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility*		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	
<b>Personnel Attributes</b>						
ACAP Analyst capability	1.46	1.19	1.00	.86	.71	
AEXP Applications experience	1.29	1.13	1.00	.91	.82	
PCAP Programmer capability	1.42	1.17	1.00	.86	.70	
VEXP Virtual machine experience*	1.21	1.10	1.00	.90		
LEXP Programming language experience	1.14	1.07	1.00	.95		
<b>Project Attributes</b>						
MODP Use of modern programming practices	1.24	1.10	1.00	.91	.82	
TOOL Use of software tools	1.24	1.10	1.00	.91	.83	
SCED Required development schedule	1.23	1.08	1.00	1.04	1.10	

- $MM \text{ (korrigiert)} = \text{Produkt der Kostenfaktoren} * MM \text{ (oben ausgerechnet)}$ .
- COCOMO II
  - Weiterentwicklung von COCOMO
  - Neue universale Grundgleichungen:
    - $\text{Aufwand} = 2.45 * KSLOC^B * \text{Produkt der 17 Kostenfaktoren}$

- Durchlaufzeit =  $2.66 * \text{Aufwand}^{(0.33 + 0.2(B - 0.1))}$ 
  - Wachstumsfaktor B =  $1.01 + 0.01 * \text{Summe der 5 Skalierungsfaktoren}$
- Skalierungsfaktoren:

Faktor	Sehr gering	Gering	Nominal	Hoch	Sehr hoch	Extra hoch
Präzedenz	4,05	3,24	2,43	1,62	0,81	0
Flexibilität	6,07	4,86	3,64	2,43	1,21	0
Risiko-Umgang	4,22	3,38	2,53	1,69	0,84	0
Zusammenarbeit	4,94	3,95	2,97	1,98	0,99	0
Prozessreife	4,54	3,64	2,73	1,82	0,91	0

- **Präzedenz**: Vertrautheit des Entwicklungsteams mit dem Produkt
- **Flexibilität** bezüglich der Einhaltung von Anforderungen / Vorgaben
- **Risiko-Umgang**: Qualität des Risikomanagements
- Güte der **Zusammenarbeit** zwischen allen Projektbeteiligten
- **Reife** des Entwicklungsprozesses

- Kostenfaktoren:

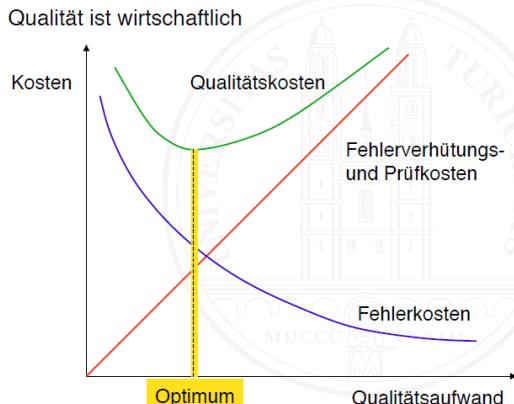
Factor	Very low	Low	Nominal	High	Very High	Extra high
Reliability required	0.75	0.88	1.00	1.15	1.39	
Database size	0.93	1.00	1.09	1.19		
Product complexity	0.75	0.88	1.00	1.15	1.30	1.66
Reuse required	0.91	1.00	1.14	1.29	1.49	
Documentation required	0.89	0.95	1.00	1.06	1.13	
Execution time constraint			1.00	1.11	1.31	1.67
Storage constraint			1.00	1.06	1.21	1.57
Platform volatility	0.87	1.00	1.15	1.30		
Analyst capability	1.50	1.22	1.00	0.83	0.67	
Programmer capability	1.37	1.16	1.00	0.87	0.74	
Personnel continuity (turnover)	1.24	1.10	1.00	0.92	0.84	
Application experience	1.22	1.10	1.00	0.89	0.81	
Platform experience	1.25	1.12	1.00	0.88	0.81	
Language and tool experience	1.22	1.10	1.00	0.91	0.84	
Use of software tools	1.24	1.12	1.00	0.86	0.72	
Team co-location and communications support	1.25	1.10	1.00	0.92	0.84	0.78
Required development schedule	1.29	1.10	1.00	1.00	1.00	

- Function-Point-Verfahren
  - Relatives Mass zur Bewertung der Funktionalität
  - Zählen von Eingaben, Ausgaben, Anfragen, Schnittstellen zu externen Datenbeständen, Interne Datenbestände.
  - Gewichtung der Werte anhand von Tabellen (siehe Skript S. 22 ff.)
  - Korrekturfaktor =  $0.65 + 0.01 * \text{Summe aller Einflussfaktoren}$ 
    - Dieser wird danach mit den Unadjusted Function Points multipliziert.
  - Mittlerer Aufwand pro Function Point muss bekannt sein.
  - Faktoren müssen unternehmens- und projektspezifisch sein.
  - Faustregeln von Jones zur Berechnung von Durchlaufzeit, Anzahl Mitarbeiter, Aufwand (= Durchlaufzeit \* Anzahl Mitarbeiter).
- Sonstige Methoden:
  - Egal, was es kostet.
  - Soviel schätzen, wie man denkt, dass der Kunde gerade noch bereit zu zahlen ist.
  - Parkinson'sches Gesetz: Der Aufwand passt sich der Kapazität an. → Schätzung und tatsächlicher Aufwand nicht voneinander unabhängig.
- Entwicklung : Pflege = 40:60 bis max. 50:50

- Faustregeln von Jones.
- Ca. pro 20 KDSI wird eine Person Vollzeit benötigt zur Pflege.

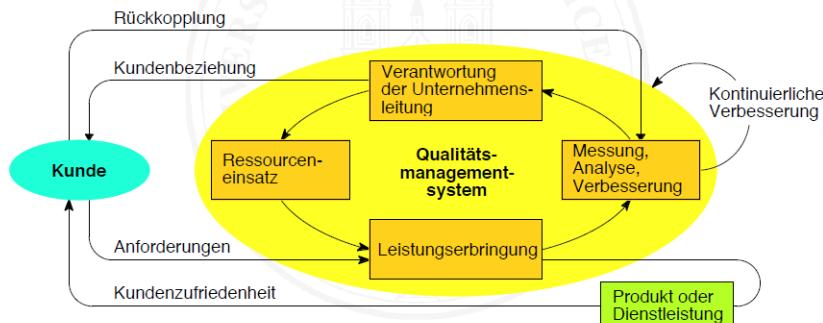
## Kapitel 16: Software-Qualitätsmanagement

- Explizites Qualitätsmanagement erst bei grossen, komplexen Entwicklungsvorhaben notwendig.



- Totales Qualitätsmanagement: Führungsmethode, welche Kundenzufriedenheit als oberstes Unternehmensziel postuliert. Qualität im Mittelpunkt und alle werden einbezogen.
- Qualitätsmanagementsystem orientiert sich an Kundenzufriedenheit und Selbstverantwortung aller Beteiligten:

- Prozessorientiert, systemischer Ansatz zur Realisierung

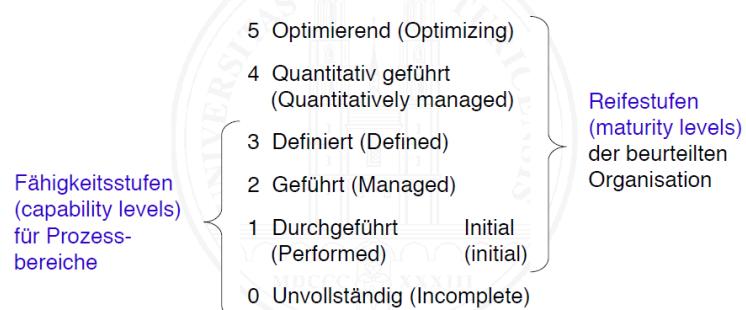


- Die Qualitätsaufgaben in die Unternehmensprozesse integrieren. Möglichst wenige Qualitätsaufgaben separat erledigen.
- Verfahren: Planung (was ist das Ziel?), Lenkung (konstruktiv [= eher präventiv, zu Lenkung], analytisch [= erkennend, nachtragend, zur Prüfung]), Verbesserung.
  - Verbesserung nimmt Einfluss auf die Planung und die Lenkung des Prozesses.
- Qualität muss dokumentiert werden, um
  - Kundenanforderungen zu erfüllen
  - Wiederholbarkeit von Verfahren sicherzustellen
  - Beurteilbarkeit und Wirksamkeit sicherzustellen usw...
- Qualitätsbezogene Dokumente:
  - Qualitätshandbuch (dokumentiert QM-System)
  - QM-Plan (QM für spezifisches Produkt)
  - Anforderungsspezifikation (zu erfüllende Anforderungen)
  - Verfahrens- und Arbeitsanweisungen.

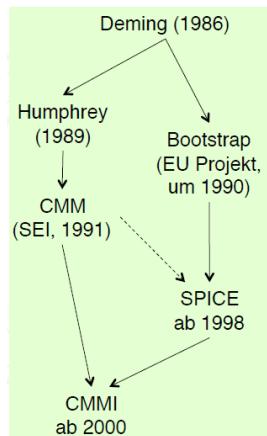
- Leitfäden (Empfehlungen und Vorschläge zur Vorgehensweise)
- Aufzeichnungen (vergangene Tätigkeiten aufgezeichnet)

## Kapitel 17: Bewertung und Verbesserung von Prozessen und Qualität

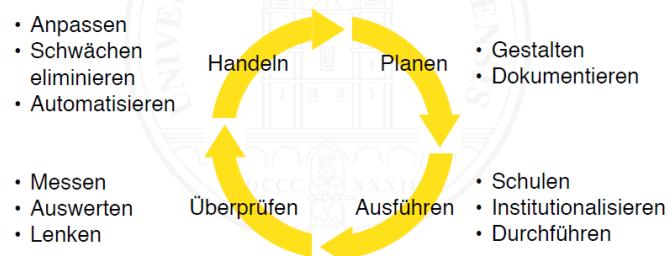
- Qualität muss gesichert werden. Mit Qualitätsdokumentation, Audits, Publikation von Messgrößen, Zertifizierungen. Qualitätssicherung ist auf Erzeugung von Vertrauen ausgerichtet.
  - Dokumentation: Wichtig zur Beschreibung von Prozessen. Durch Gewährung eines Einblicks in diese Dokumentation durch den Kunden kann Vertrauen geschaffen werden.
  - Audit: Existenz eines QM garantiert nicht dessen Wirksamkeit → regelmässige Überprüfung.
    - Interne Systemaudits: Unternehmensinterne, regelmässige Überprüfung des QM-System.
    - Interne Prozess-, Produkt- oder Projektaudits: Spontane Audits bei Anzeichen grösserer Probleme.
    - Lieferanten-/ Kundenaudits: Externe Audits (durch Dritte oder eigene Auditoren) bei Lieferanten (um Qualität der eingekauften Ware festzustellen) oder Kunde (um zu beurteilen, ob Partnerschaft eingegangen werden sollte)
    - Zertifizierungsaudit: durch Auditoren einer für Zertifizierung autorisierten Stelle.
    - Auditfragen:
      - Geschlossene Fragen: Antwort mit ja/nein.
      - Offene Fragen: Befragte müssen etwas schildern.
  - Publikation von Messgrößen: Ausweisen des Standes bzw. Fortschritt qualitätrelevanter Größen.
    - Als Ausweis gegenüber Kunden
    - Als Information, Bestätigung und Ansporn für Mitarbeiter.
  - Zertifizierung: Achtung, bedeutet nicht, dass UN Software hoher Güte herstellt.
    - QM kann zertifiziert werden, auch einzelne Produkte.
- Prozessbeurteilungsverfahren
  - CMMI (Capability Maturity Model Integrated) Standard nach SEI.
    - Ziele: Feststellen der Reife eines Software-Prozesses, Bereitstellung eines Handlungsrahmens zur schrittweisen Verbesserung des Reifegrades.
    - Darstellungen:  
Sechsstufige Ordinalskala:



- Verlaufsdarstellung: Jeder Prozessbereich wird einzeln beurteilt mit vier Fähigkeitsstufen → Fähigkeitsprofil. Prozessverbesserung mit Transition von IST- zu SOLL-Profil. Analog zu SPICE-Bewertungsmodell.
- Gestufte Darstellung: Bewertung mit Reifestufen. Prozessverbesserung durch schrittweisen Stufenanstieg. Entspricht etwa Bewertungsmodell des CMM.
- Insgesamt gibt es 22 Prozessbereiche. Für jeden Prozessbereich gibt es:
  - Spezifische Ziele, die verbindlich erreicht werden müssen.
  - Spezifische Praktiken zur Erreichung der Ziele.
- Alles kann sehr schön graphisch dargestellt werden, in Fähigkeitsdiagrammen etc. Es werden Generische Ziele und Praktiken (Massnahmen, welche zu diesen Zielen führen sollten) bestimmt, welche für alle Prozess gelten. Die Spezifischen werden dann für die einzelnen Bereiche definiert.
- Achtung: Gute Prozess sind notwendig, aber nicht hinreichend. UN mit gutem CMMI kann schlechte Software produzieren.
- SPICE (Software Process Impovement and Capability Determination) ISO 15504



- Meta-Prozess der Software-Prozessverbesserung:  
Zyklisches Vorgehen: Plan-Do-Check-Act (Deming 1986):



## Kapitel 18: Produktivitätsfaktoren

- Was beeinflusst die Produktivität?
  - Mensch (Ausbildung, Motivation, Förderung)
  - Technische Hilfsmittel (Werkzeuge)
  - Projektmanagement (Gut geplanter Personaleinsatz, Vermeidung von Kontextwechselverlusten, Vermeidung von Koordinations- und Kommunikationsverlusten)

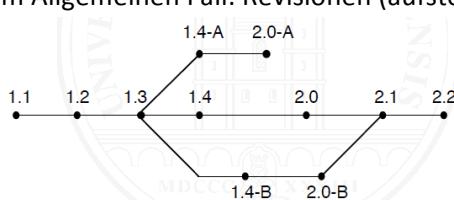
- Mehrfachverwendung

## Kapitel 19: Die Rolle der Menschen im Software-Engineering

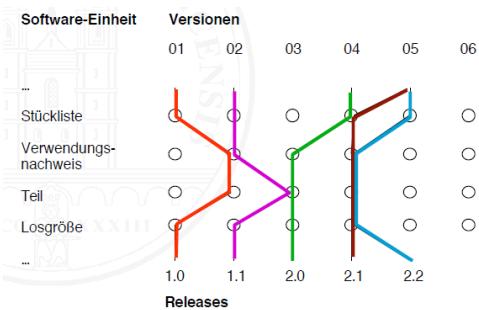
- Die Produktivität schwankt extrem zwischen einzelnen Entwickler und Entwicklergruppen.
- Personalbestände nur langsam auf- und abbaubar. Das Aufstocken des Personalbestandes in verspäteten Projekten führt zu noch mehr Verspätung.
- Nicht produktiver Aufwand wächst mit Gruppengröße überproportional.
- 3 Typen von Leuten: Gelernte, Angelernt, Ungelernte.
- Software Engineering funktioniert nur als dynamische Synthese einer technik- und einer menschenzentrierten Welt.
- Eine Software Engineering Kultur schaffen und Freude haben.

## Kapitel 20: Software Konfigurationsverwaltung

- Software Konfiguration: Eine Menge zusammenpassender Software-Einheiten
- Jede Software Einheit ist eindeutig identifizierbar durch eine Nummer, registriert und verwaltet.
- Versionierung:
  - Einfachste Art: Aufsteigende Versionsnummern
  - Im Allgemeinen Fall: Revisionen (aufsteigend) und Varianten (parallel)



- Release: Eine zur Benutzung freigegebene Konfiguration (= Konsistente Menge logisch zusammengehörender Software-Einheiten).
- Konfigurationen/Releases werden aus ausgewählten Software-Einheiten in bestimmten Versionen gebildet.

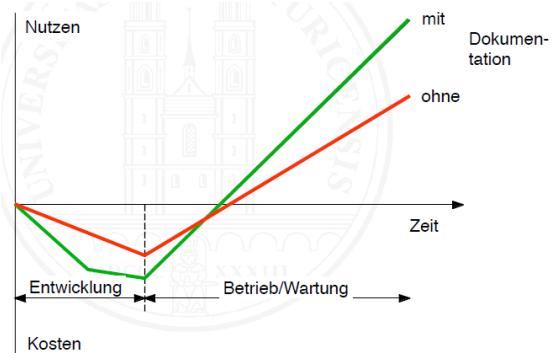


- Paralleles Arbeiten. Änderungen durch:
  - Mischen: Ermöglicht paralleles Arbeiten. Potentiell unsicher. Nur möglich wenn Artefakt mischbar ist, z.B. Code. Kommunikation erforderlich zwischen MA.
  - Sperren: Behindert paralleles Arbeiten. Sicher. Für jeden Typ von Artefakt anwendbar.
  - → Mischen ist beliebter im Moment.
- Problemmelbewesen: Problemmeldeprozess:
  - Eingegangene Problemmeldung (meist vom Kunden) registrieren.
  - Problem analysieren und priorisieren

- Entscheiden:
  - Jetzt bearbeiten: Problem beheben, gegebenenfalls neues Release herausgeben. Problemmeldung abschliessen und archivieren.
  - Später aufnehmen: Problem in Liste aufnehmen. Diese Liste in der Releaseplanung abarbeiten.
  - Nicht bearbeiten.

## Kapitel 21: Dokumentation

- Arten der Dokumentation:
  - Produktdokumentation: dokumentiert das Produkt und seine Benutzung.
  - Projektdokumentation: dokumentiert die Entstehung.
- Aufgaben der Dokumentation:
  - Wissenssicherung.
  - Kommunikation.
  - Sichtbarmachen des Projektfortschrittes.
- Nur so wenig wie nötig dokumentieren, aber dies sorgfältig und konsequent. → ökonomisch.



- Produktdokumentation beinhaltet:
  - Anforderungsspezifikation.
  - Lösungskonzept: Architektur der Lösung.
  - Detailentwurf und Code: Lösungsdetails wie Algorithmen und Datenstrukturen.
  - Testvorschriften.
  - Abnahmevereinbarungen: Formaler Abschluss einer Entwicklung.
  - Integrationsplan.
  - Installationsanleitung.
  - Benutzerhandbuch.
  - Glossar.
- Projektdokumente:
  - Projektplan: Stellt SOLL und IST gegenüber.
  - Qualitätsplan: Projektspezifische Vorgaben der Qualität.
  - Projektprotokoll.
- Dokumentation sollte während des Prozesses begleitend hergestellt werden und nicht hinterher.

## Kapitel 22: Werkzeuge

- Werkzeug: rechnergestütztes Hilfsmittel für die Entwicklung von Software.

- Bei der Einführung von Werkzeugen sinkt die Produktivität zunächst aufgrund von Schulung, Eingewöhnung, Verlagerung von Aufwendungen.
- Usw.