Software Engineering: Wartung und Qualitätssicherung

12. Januar 2020

1 Software-Entwicklung, Wartung und (Re)Engineering

Erstellte Software muss oft geändert werden, entweder aufgrund von geänderten Anforderungen oder neuen Anforderungen, welche eingebaut werden müssen.

1.1 Einleitung

1.1.1 Geschichte

- Softwarekrise 1968
- Nato **Working Conference on Software Engineering**
- Zuordnungen
  - Praktische Informatik
  - Theoretische Informatik
  - Projektplanung
  - Organisation
  - Psychologie
  - ...

1.1.2 "Software-Technik" Definition

Software-Engineering(Software-Technik) ist nach **Entwicklung**, **Pflege** und **Einsatz**.
Eingesetzt werden:

- Wissenschaftliche Methoden
- Wirtschaftliche Prinzipien
- Geplante Vorgehensmodellen
- Werkzeuge
- Quantifizierbare Ziele

1.1.3 50 Jahre nach Beginn der SSoftware-Krise"

- 19% aller Projekte sind gescheitert, früher 25%
- 52% aller Projekte sind dabei zu scheitern, früher 50%
- 29% aller betrachteten IT-Projekte sind erfolgreich, früher 25%

Hauptgründe fürs Scheitern der Projekte:
Unklare Anforderungen und Abhängigkeiten sowie Problemen beim Änderungsmanagement.

1.2 Software-Qualität

Ziel der Software-Technik ist die effiziente Entwicklung messbar qualitativ hochwertiger Software.

1.2.1 Qualitätsdefinition

Qualität ist der Grad, in dem ein System, eine Komponente oder ein Prozess die Kundenerwartungen und Kundenbedürfnisse erfüllt.

1.2.2 Softwarequalität

Softwarequalität ist die Gesamtheit der Funktionalitäten und Merkmale eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

1.2.3 Qualitätsmerkmale

- Funktionalität

1.2.4 Nichtfunktionale Merkmale

- Zuverlässigkeit (Reliability)
- Benutzbarkeit (Usability)
- Effizienz (Efficiency)
- Änderbarkeit (Maintainability)
- Übertragbarkeit (Portability)

1.2.5 Prinzipien der Qualitätssicherung

- **Qualitätszielbestimmung**: Auftraggeber und Auftragnehmer legen vor Beginn der Software-Entwicklung gemeinsames Qualitätsziel für Software-System mit nachprüfbaren Kriterienkatalog fest (als Bestandteil des abgeschlossenen Vertrags zur Software-Entwicklung)
- **Quantitative Qualitätssicherung**: Einsatz automatisch ermittelbaren Metriken zur Qualitätsbestimmung (objektivbare, ingenieursmäßige Vorgehensweise)
- **Konstruktive Qualitätssicherung**: Verwendung geeigneter Methoden, Sprachen und Werkzeuge (Vermeidung von Qualitätsproblemen)
- **Integrierte, frühzeitige, analytische Qualitätssicherung**: Systematische Prüfung aller erzeugter Dokumente (Aufdeckung von Qualitätsproblemen)
- **Unabhängige Qualitätssicherung**: Entwicklungsprodukte werden durch eigenständige Qualitätssicherungsabteilung überprüft und abgenommen (verhindert u.a. Verzicht auf Testen zugunsten Einhaltung des Entwicklungsplans)

1.2.6 Konstruktives Qualitätssicherung zur Fehlervermeidung:

- Technische Maßnahmen
  - Sprachen (UML, Java)
  - Werkzeuge (UML-CASE-TOOL)
- Organisatorische Maßnahmen
  - Richtlinien (Gliederungsschema für Pflichtenheft, Programmierrichtlinien)
  - Standards (für verwendete Sprachen, Dokumentformate, Management)
  - Checklisten

1.2.7 Analytisches Qualitätsmanagement für Fehleridentifikation

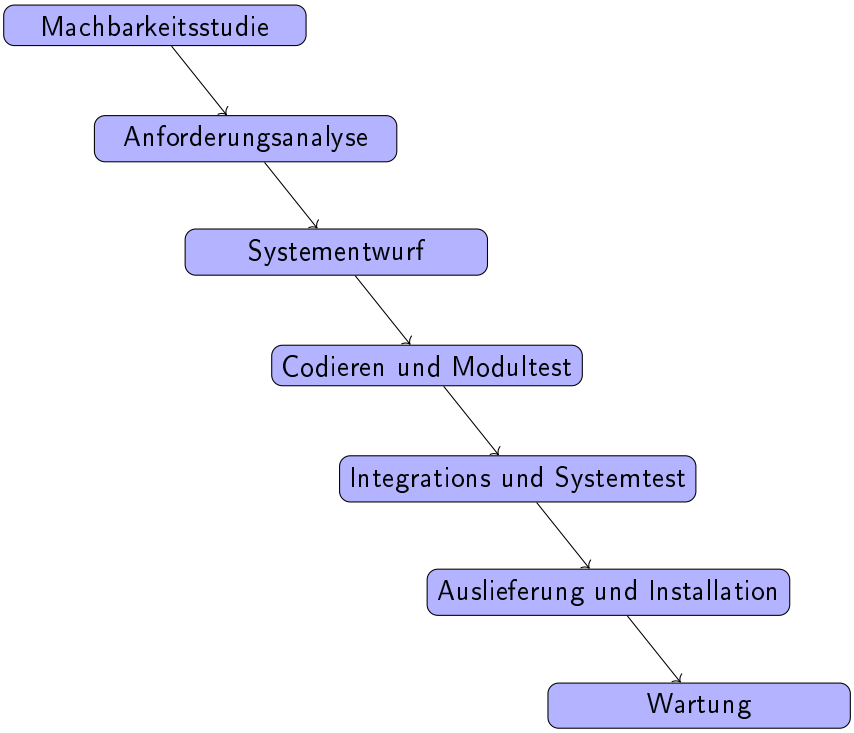
- **Analysierende Verfahren**: Der "Prüfling"(Programm, Modell, Dokumentation) wird von Menschen oder Werkzeugen auf Vorhandensein/Abwesenheit von Eigenschaften untersucht
  - **Review**: Prüfung durch Menschen
  - **Statische Analyse**: Werkzeuggestützte Ermittlung von Änomalien"
  - **Formale Verifikation**: Werkzeuggestützter Beweis von Eigenschaften
- **Testende Vefahren**: Der "Prüfling"wird mit konkreten oder abstrakten Eingabewerten auf einem Rechner ausgeführt
  - **Dynamischer Test**: "normaleÄusführung mit ganz konkreten Eingaben
  - **Symbolischer Test**: Ausführung mit symbolischen Eingaben

1.3 Iterative Softwareentwicklung

Voraussetzung für den sinnvollen Einsatz von Notationen und Werkzeugen zur Software-Entwicklung ist ein:

- **Vorgehensmodell**, das den Gesamtprozess der Software-Erstellung und pflege in einzelne Schritte aufteilt
- Zusätzlich müssen Verantwortlichkeiten der beteiligten Personen in Form von **Rollen** im Software-Entwicklungsprozess klar geregelt sein.

1.3.1 Übersicht der Phasen des Wasserfallmodells



1.3.2 Machbarkeitsstudie (feasability study)

Die Machbarkeitsstudie schätzt Kosten und Ertrag der geplanten Software-Entwicklung ab. Dazu grobe Analyse des Problems mit Lösungsvorschlägen.

- **Aufgaben**
  - Problem informell und abstrahiert beschreiben
  - Verschiedene Lösungsansätze erarbeiten
  - Kostenschätzung durchführen

  - Angebotserstellung

- **Ergebnisse**
  - Lastenheft
  - Projektkalkulation
  - Projektplan
  - Angebot an Auftraggeber

1.3.3 Anforderungsanalyse (requirements engineering)

In der Anforderungsanalyse wird exakt festgelegt, was die Software leisten soll, aber nicht wie diese Leistungsmerkmale erreicht werden.

- **Aufgaben**
  - genaue Festlegung der Systemeigenschaften wie Funktionalität, Leistung, Benutzungsschnittstelle, Portierbarkeit, ... im Pflichtenheft
  - Bestimmen von Testfällen
  - Festlegung erforderlicher Dokumentationsdokumente
- **Ergebnisse**
  - Pflichtenheft = Anforderungsanalysedokument
  - Akzeptanztestplan
  - Benutzungshandbuch (1-te Version)

1.3.4 Systementwurf (system design/programming-in-the-large)

Im Systementwurf wird exakt festgelegt, wie die Funktionen der Software zu realisieren sind. Es wird der Bauplan der Software, die Software-Architektur, entwickelt.

- **Aufgaben**
  - Programmieren-im-Großen = Entwicklung eines Bauplans
  - Grobentwurf, der System in Teilsysteme/Module zerlegt
  - Auswahl bereits existierender Software-Bibliotheken, Rahmenwerke, ...
  - Feinentwurf, der Modulschnittstellen und Algorithmen vorgibt
- **Ergebnisse**
  - Entwurfsdokument mit Software-Bauplan
  - detaillierte(re) Testpläne

1.3.5 Codieren und Modultest (programming-in-the-small)

Die eigentliche Implementierungs- und Testphase, in der einzelne Module (in einer bestimmten Reihenfolge) realisiert und validiert werden.

- **Aufgaben**
  - Programmieren-im-Kleinen = Implementierung einzelner Module
  - Einhaltung von Programmierrichtlinien
  - Code-Inspektionen kritischer Modulteile (Walkthroughs)
  - Test der erstellten Module
- **Ergebnisse**
  - Menge realisierter Module
  - Implementierungsberichte (Abweichungen vom Entwurf, Zeitplan, ...)
  - Technische Dokumentation einzelner Module
  - Testprotokolle

1.3.6 Integrations- und Systemtest

Die einzelnen Module werden schrittweise zum Gesamtsystem zusammengebaut. Diese Phase kann mit der vorigen Phase verschmolzen werden, falls der Test isolierter Module nicht praktikabel ist.

- **Aufgaben**
  - Systemintegration = Zusammenbau der Module
  - Gesamtsystemtest in Entwicklungsorganisation durch Kunden (alpha-Test)
  - Fertigstellung der Dokumentation
- **Ergebnisse**
  - Fertiges System
  - Benutzerhandbuch
  - Technische Dokumentation
  - Testprotokolle

1.3.7 Auslieferung und Installation

Die Auslieferung (Installation) und Inbetriebnahme der Software beim Kunden findet häufig in zwei Phasen statt.

- **Aufgaben**
  - Auslieferung an ausgewählte (Pilot-)Benutzer (Beta-Test)
  - Auslieferung an alle Benutzer
  - Schulung der Benutzer
- **Ergebnisse**
  - Fertiges System
  - Akzeptanztestdokument

1.3.8 Wartung (Maintenance)

Nach der ersten Auslieferung der Software an die Kunden beginnt das Elend der Software-Wartung, das ca. 60% der gesamten Software-Kosten ausmacht.

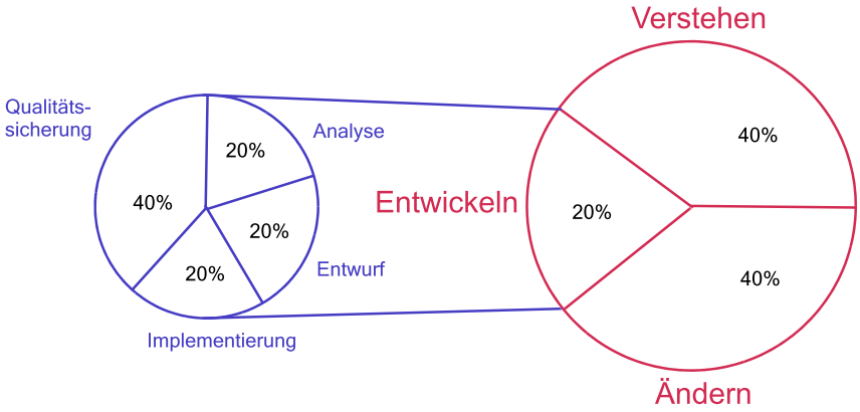
- **Aufgaben**
  - ca. 20% Fehler beheben (corrective maintenance)
  - ca. 20% Anpassungen durchführen (adaptive maintenance)
  - ca. 50% Verbesserungen vornehmen (perfective maintenance)
- **Ergebnisse**
  - Software-Problemberichte (bug reports)
  - Software-Änderungsvorschläge
  - Neue Software-Versionen

1.3.9 Probleme mit dem Wasserfallmodell

- zu Projektbeginn sind nur ungenaue Kosten- und Ressourcenschätzungen möglich
- ein Pflichtenheft kann nie den Umgang mit dem fertigen System ersetzen, das erste sehr spät entsteht (Risikomaximierung)
- es gibt Fälle, in denen zu Projektbeginn kein vollständiges Pflichtenheft erstellt werden kann (weil Anforderungen nicht klar)
- Anforderungen werden früh eingefroren, notwendiger Wandel (aufgrund organisatorischer, politischer, technischer, ... Änderungen) nicht eingeplant

- strikte Phaseneinteilung ist unrealistisch (Rückgriffe sind notwendig)
- Wartung mit ca. 60% des Gesamtaufwandes ist eine Phase

Abbildung 1: Andere Darstellung der Aufwandsverteilung



1.3.10 Typische Probleme in der Wartungsphase

- Einsatz wenig erfahrenen Personals (nicht Entwicklungspersonal)
- Fehlerbehebung führt neue Fehler ein
- Stetige Verschlechterung der Programmstruktur
- Zusammenhang zwischen Programm und Dokumentation geht verloren
- Zur Entwicklung eingesetzte Werkzeuge (CASE-Tools, Compiler, ...) sterben aus
- Benötigte Hardware steht nicht mehr zur Verfügung
- Resourcenkonflikte zwischen Fehlerbehebung und Anpassung/Erweiterung
- Völlig neue Ansprüche an Funktionalität und Benutzeroberfläche

1.4 Forward-, Reverse- und Reengineering

1.4.1 Software Evolution

- Wünsche
  - Wartung ändert Software kontrolliert ohne Design zu zerstören
  - Konsistenz aller Dokumente bleibt erhalten

- Wirklichkeit
  - Ursprüngliche Systemstruktur wird ignoriert
  - Dokumentation wird unvollständig oder unbrauchbar
  - Mitarbeiter verlassen Projekt

1.4.2 Forward Engineering

Beim Forward Engineering ist das fertige Softwaresystem das Ergebnis des Entwicklungsprozesses. Ausgehend von Anforderungsanalyse (Machbarkeitsstudie) wird ein neues Softwaresystem entwickelt.

1.4.3 Reverse Engineering

Beim Reverse Engineering ist das vorhandene Software-System der Ausgangspunkt der Analyse. Ausgehend von existierender Implementierung wird meist „nur“ das Design rekonstruiert und dokumentiert. Es wird (noch) nicht das betrachtete System modifiziert.

1.4.4 Reengineering

Reengineering befaßt sich mit der Sanierung eines vorhandenen Software-Systems bzw. seiner Neuimplementierung. Dabei werden die Ergebnisse des Reverse Engineerings als Ausgangspunkt genommen

Abbildung 2: Round Trip Engineering

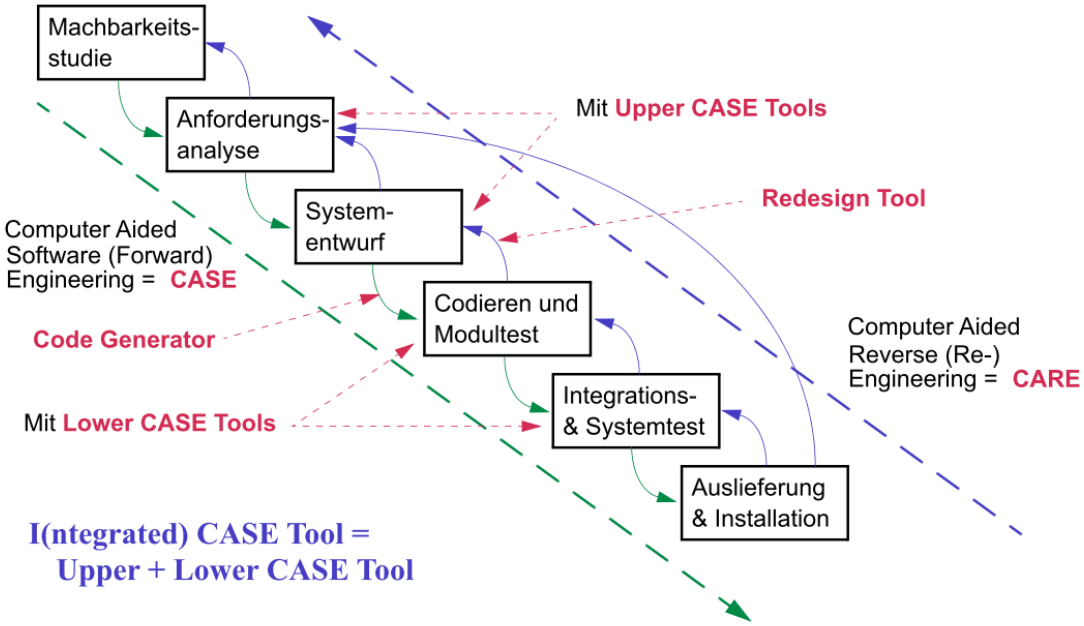
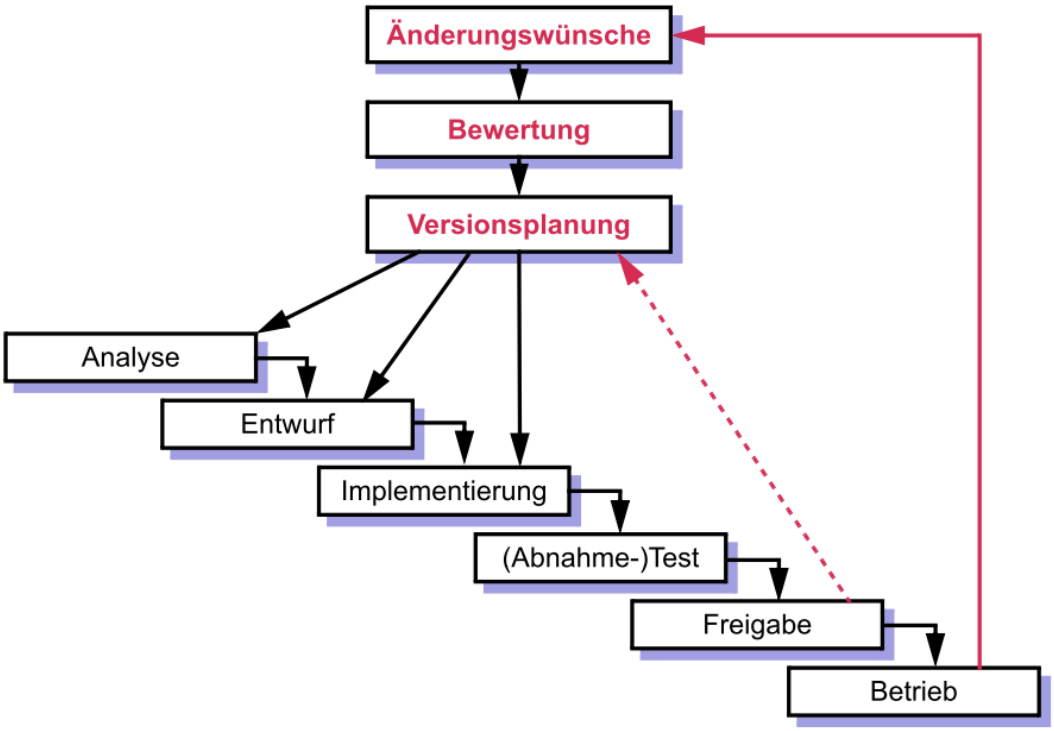


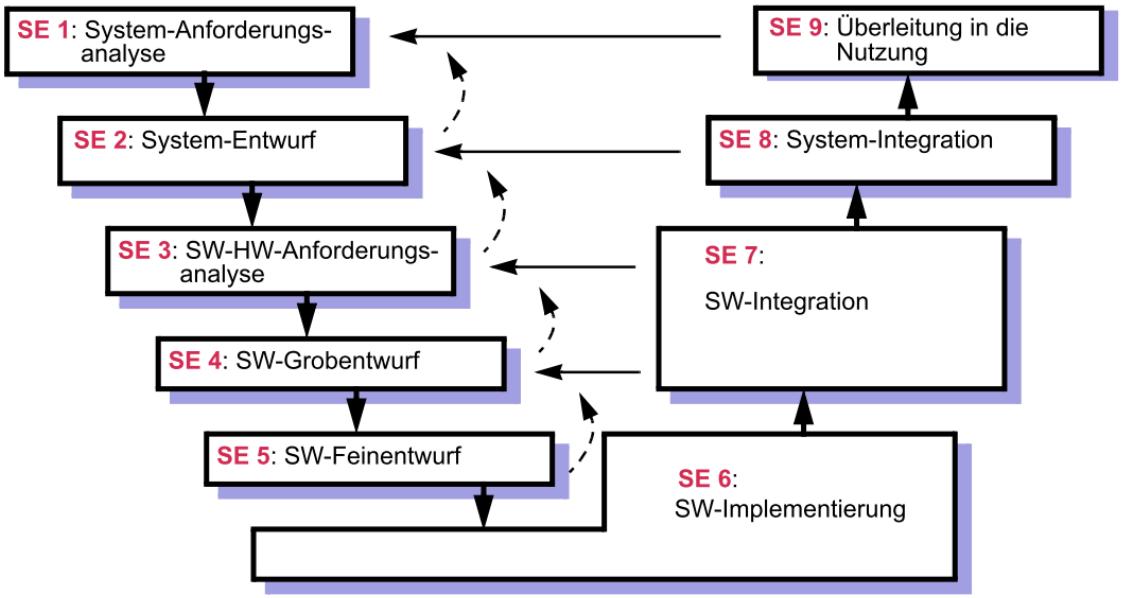
Abbildung 3: Einfaches Software-Lebenszyklus-Prozessmodell für die Wartung



1.4.5 Das V-Modell

- **Systemanforderungsanalyse**: Gesamtsystem einschließlich aller Nicht-DV-Komponenten wird beschrieben (fachliche Anforderungen und Risikoanalyse)
- **Systementwurf**: System wird in technische Komponenten (Subsysteme) zerlegt, also die Grobarchitektur des Systems definiert
- **Softwareanforderungsanalyse**: Technischen Anforderungen an die bereits identifizierten Komponenten werden definiert
- **Softwaregrobentwurf**: Softwarearchitektur wird bis auf Modulebene festgelegt
- **Softwarefeinentwurf**: Details einzelner Module werden festgelegt
- **Softwareimplementierung**: Wie beim Wasserfallmodell (inklusive Modultest)
- **Software-/Systemintegration:**: Schrittweise Integration und Test der verschiedenen Systemanteile
- **Überleitung in die Nutzung:**: Entspricht Auslieferung bei Wasserfallmodell

Abbildung 4: Das V-Modell



2 Konfigurationsmanagement

Beim Konfigurationsmanagement handelt es sich um die Entwicklung und Anwendung von Standards und Verfahren zur Verwaltung eines sich weiterentwickelnden Systemprodukts.

2.1 Einleitung

2.1.1 Fragestellungen

- Das System lief gestern noch; was hat sich seitdem geändert?
- Wer hat diese (fehlerhafte?) Änderung wann und warum durchgeführt?
- Wer ist von meinen Änderungen an dieser Datei betroffen?
- Auf welche Version des Systems bezieht sich die Fehlermeldung?
- Wie erzeuge ich Version x.y aus dem Jahre 1999 wieder?
- Welche Fehlermeldungen sind in dieser Version bereits bearbeitet?
- Welche Erweiterungswünsche liegen für das nächste Release vor?
- Die Platte ist hinüber; was für einen Status haben die Backups?

2.1.2 Definitionen

Definition von Software-KM nach IEEE-Standard 828-1988 SCM (Software Configuration Management) constitutes **good engineering practice** for all software projects, whether phased development, rapid prototyping, or ongoing maintenance. It enhances the reliability and quality of software by:

- Providing structure for **identifying and controlling** documentation, code, interfaces, and databases to support all life cycle phases
- Supporting a chosen **development/maintenance methodology** that fits the requirements, standards, policies, organization, and management philosophy
- Producing **management and product information** concerning the status of baselines, change control, tests, releases, audits etc.

Diese Definition ist jedoch nicht konkret und unabhängig vom Begriff "Software"

Definition nach DIN EN ISO 10007 KM (Konfigurationsmanagement) ist eine Managementdisziplin, die über die gesamte Entwicklungszeit eines Erzeugnisses angewandt wird, um Transparenz und Überwachung seiner funktionellen und physischen Merkmale sicherzustellen.
Der KM-Prozess umfasst die folgenden integrierten Tätigkeiten:

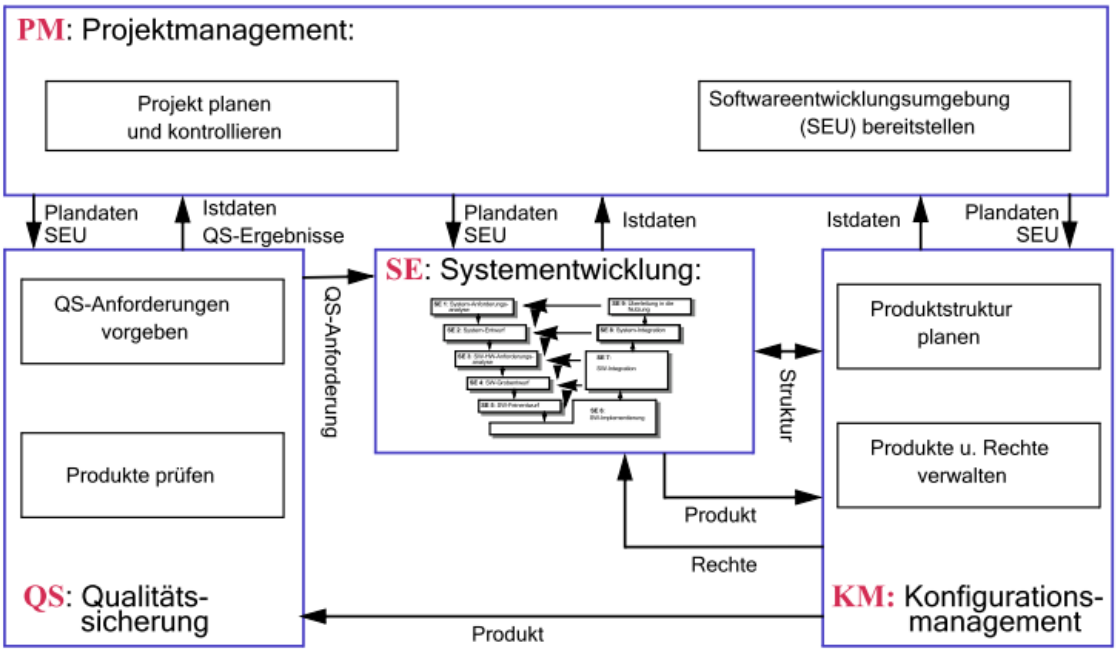
- **Konfigurationsidentifizierung**: Definition und Dokumentation der Bestandteile eines Erzeugnisses, Einrichten von Bezugskonfigurationen, ...
- **Konfigurationsüberwachung**: Dokumentation und Begründung von Änderungen, Genehmigung oder Ablehnung von Änderungen, Planung von Freigaben, ...
- **Konfigurationsbuchführung**: Rückverfolgung aller Änderungen bis zur letzten Bezugskonfiguration, ...
- **Konfigurationsauditierung**: Qualitätssicherungsmaßnahmen für Freigabe einer Konfiguration eines Erzeugnisses
- **KM-Planung**: Festlegung der Grundsätze und Verfahren zum KM in Form eines KM-Plans

Werkzeugorientierte Sicht auf KM-Aktivitäten

1. **KM-Planung**: Beschreibung der Standards, Verfahren und Werkzeuge, die für KM benutzt werden; wer darf/muss wann was machen
2. **Versionsmanagement**: Verwaltung der Entwicklungsgeschichte eines Produkts; also wer hat wann, wo, was und warum geändert
3. **Variantenmanagement**: Verwaltung parallel existierender Ausprägungen eines Produkts für verschiedene Anforderungen, Länder, Plattformen

4. **Releasemanagement**: Verwaltung und Planung von Auslieferungsständen; wann wird eine neue Produktversion mit welchen Features auf den Markt geworfen
5. **Buildmanagement**: Erzeugung des auszulieferenden Produkts; wann muss welche Datei mit welchem Werkzeug generiert, übersetzt, ... werden
6. **Änderungsmanagement**: Verwaltung von Änderungsanforderungen; also Bearbeitung von Fehlermeldungen und Änderungswünschen (Feature Requests) sowie Zuordnung zu Auslieferungsständen

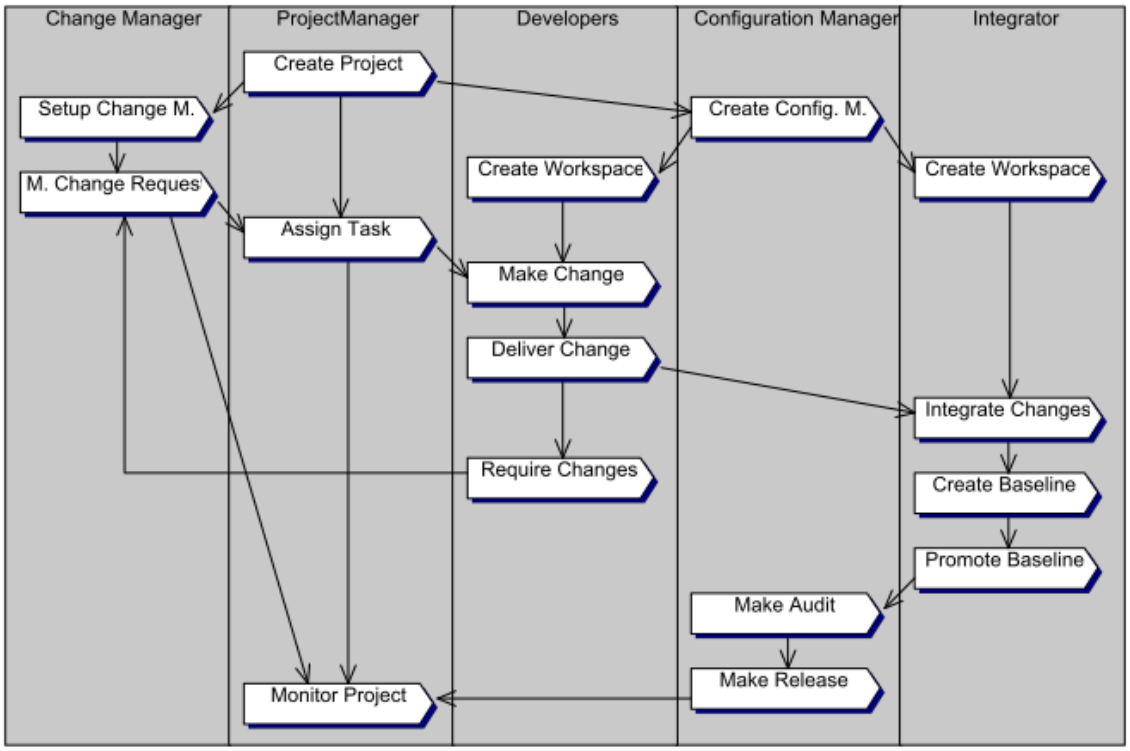
Abbildung 5: Integration des Konfigurationsmanagements im V-Modell



Workspaces für das Konfigurationsmanagement

- alle Dokumente (Objekte, Komponenten) zu einem bestimmten Projekt werden in einem gemeinsamen Repository (**public workspace**) aufgehoben
- im Repository werden nicht nur aktuelle Versionen, sondern auch alle **früheren Versionen** aller Dokumenten gehalten
- beteiligte Entwickler bearbeiten ihre eigenen Versionen dieser Dokumente in ihrem privaten Arbeitsbereich (private workspace, **developer workspace**)
- es gibt genau einen Integrationsarbeitsbereich (**integrations workspace**) für die Systemintegration

Abbildung 6: Grafische Übersicht über Aufgaben- und Rollenverteilung



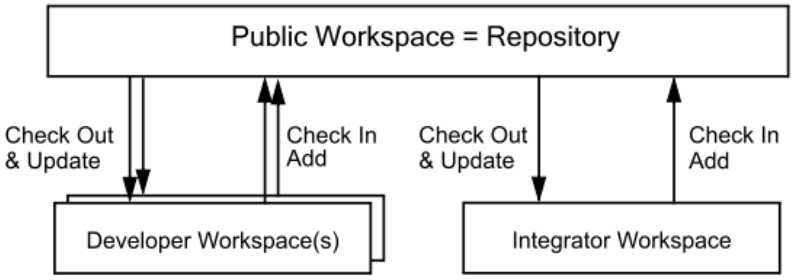
Aktivitäten bei der Arbeit mit Workspaces

- Personen holen sich Versionen neuer Dokumente, die von anderen Personen erstellt wurden(**checkout**), ih ihren privaten Arbeitsbereich
- Personen passen ihre Privatversionen ggf. von Zeit zu Zeit an neue Versionen im öffentlichen Repository an (**update**).
- Sie fügen (hoffentlich) nur konsistente Dokumente als neue Versionen in das allgemeine Repository ein (**checkin** = **commit**).
- Ab und an werden neue Dokumente dem Repository hinzugefügt (**add**).
- Jede Person kann alte/neue Versionen frei wählen.

Probleme

- Wie wird Konsistenz von Gruppen abhängiger Dokumente sichergestellt?
- Was passiert bei gleichzeitigen Änderungswünschen für ein Dokument?

Abbildung 7: Workspaces für das Konfigurationsmanagement



- Wie realisiert man die Repository-Operationen effizient?
- Wie unterstützt man "Offline"-Arbeit (ohne Zugriff auf Repository)?

Weitere Begriffe des Konfigurationsmanagements

- **Dokument** = Gegenstand, der der Konfigurationsverwaltung unterworfen wird (eine einzelne Datei oder ein ganzer Dateibaum oder ...)
- **(Versions-)Objekt** = Zustand einer Dokument zu einem bestimmten Zeitpunkt in einer bestimmten Ausprägung
- **Varianten** = parallel zueinander (gleichzeitig) existierende Ausprägungen eines Dokuments, die unterschiedliche Anforderungen erfüllen
- **Revisionen** = zeitlich aufeinander folgende Zustände eines Dokuments
- **Konfiguration** = komplexes Versionsobjekt, eine bestimmte Ausprägung eines Programmsystems (oft hierarchisch strukturierte Menge von Dokumenten)
- **Baseline** = eine Konfiguration, die zu einem Meilenstein (Ende einer Entwicklungsphase) gehört und evaluiert (getestet) wird
- **Release** = eine stabile Baseline, die ausgeliefert wird (intern an Entwickler oder extern an bestimmte Kunden oder ...)

2.2 Versionsmanagement

Bekannteste "open source"-Produkte (in zeitlicher Reihenfolge) sind:

- Source Code Control System **SCCS** von AT&T (Bell Labs):
  - effiziente Speicherung von Textdateiversionen als "Patches"
- Revision Control System **RCS** von Berkley/Purdue University

  - schnellerer Zugriff auf Textdateiversionen

- Concurrent Version (Control) System **CVS** (zunächst Skripte für RCS)
  - Verwaltung von Dateibäumen
  - parallele Bearbeitung von Textdateiversionen
- Subversion **SVN** - CVS-Nachfolger von CollabNet initiiert (http://www.collab.net)
  - Versionierung von Dateibäumen
- **Git**, Mercurial, ... als verteilte Versionsmanagementsysteme
  - jeder Entwickler hat eigene/lokale Versionsverwaltung

2.2.1 Source Code Control System SCCS von AT&T (Bell Labs)

Je Dokument (Quelltextdatei) gibt es eine eigene **History-Datei**, die alle Revisionen als eine Liste jeweils geänderter (Text-)Blöcke speichert:

- jeder Block ist ein **Delta**, das Änderungen zwischen Vorgängerrevision und aktueller Revision beschreibt
- jedes Delta hat **SCCS-Identifikationsnummer** der zugehörigen Revision: <ReleaseNo>.<LevelNo>.<BranchN

Revisionsbäume von SCSS

- Release 1.1 Neuentwicklung
- Release 1.1 Wartung
- Release 1.2 Weiterentwicklung
- Release 2 Weiterentwicklung

Erläuterungen zu "diff" und "patch"

- "**diff**"-Werkzeug bestimmt Unterschiede zwischen (Text-)Dateien = **Deltas**
- ein Delta(diff) zwischen zwei Textdateien besteht aus einer Folge von "Hunks", die jeweils Änderungen eines Zeilenbereichs beschreiben:
  - Änderungen von Zeilen: werden mit "**!**" markiert
  - Hinzufügen von Zeilen: werden mit "+" markiert
  - Löschen von Zeilen: werden mit "**-**" markiert
- reale Deltas enthalten unveränderte **Kontextzeilen** zur besseren Identifikation von Änderungsstellen

- ein **Vorwärtsdelta** zwischen zwei Dateien d1 und d2 kann als "**patch**" zur Erzeugung von Datei d2 auf Datei d1 angewendet werden
- inverses **Rückwärtsdelta** zwischen zwei Dateien d1 und d2 kann als "patch" zur Wiederherstellung von Datei d1 auf Datei d2 angewendet werden
- SCCS-Deltas sind in einer Datei gespeichert, deshalb weder Vorwärts- noch Rückwärts- sondern **Inline-Deltas**

Genauere Instruktionen zur Erzeugung von Deltas Jedes "diff"-Werkzeug hat seine eigenen Heuristiken, wie es möglichst kleine und/oder lesbare Deltas/Patches erzeugt, die die Unterschiede zweier Dateien darstellen. Ein möglicher (und in den Übungen verwendeter) Satz von Regeln zur Erzeugung von Deltas sieht wie folgt aus:

1. Die Anzahl der geänderten, gelöschten und neu erzeugten Zeilen aller Hunks eines **Deltas** zweier Dateien wird möglichst klein gehalten.
2. Jeder Hunk beginnt mit genau einer unveränderten **Kontextzeile** und enthält sonst nur geänderte, gelöschte oder neu eingefügte Zeilen (Ausnahme: Dateianfang).
3. Aufeinander folgende Hunks sind also durch jeweils **mindestens eine unveränderte Zeile** getrennt.
4. Optional: Anstelle von Löschen und Neuerzeugen einer Zeile i verwendet man die **Änderungsmarkierung "!"**

Durch diese Regeln nicht gelöstes Problem Wie erkenne ich, ob eine Änderung in Zeile i durch Einfügen einer neuen Zeile oder durch Ändern einer alten Zeile zustande gekommen ist?

Create- und Apply-Patch in Eclipse Die "Create Patch"- und "Apply Patch"-Funktionen in Eclipse benutzen genau das gerade eingeführte "Unified Diff"-Format. Dabei werden bei der Erzeugung von Hunks wohl folgende Heuristiken/Regeln verwendet:

- ein Hunk scheint in der Regel mit drei unveränderten Kontextzeilen zu beginnen (inklusive Leerzeilen).
- zwei Blöcke geänderter Zeilen müssen durch mindestens sieben unveränderte Zeilen getrennt sein, damit dafür getrennte Hunks erzeugt werden

Bei der Anwendung von Patches werden folgende Heuristiken/Regeln verwendet:

- werden der Kontext oder die zu löschenden Zeilen eines Patches so nicht gefunden, dann endet die Patch-Anwendung mit einer Fehlermeldung
- befindet sich die zu patchende Stelle eines Textes nicht mehr an der angegebenen Stelle (Zeile), so wird trotzdem der Patch angewendet

- gibt es mehrere (identische) Stellen in einem Text, auf die ein Patch angewendet werden kann, so wird die Stelle verändert, die am nächsten zur alten Position ist

Eigenschaften von SCSS

- für beliebinge (Text-)Dateien verwendbar (und nur für solche
- Schreibsperren auf "ausgecheckten" Revisionen
- Revisionsbäume mit manuellem Konsistenthalten von Entwicklungszweigen
- Rekonstruktionszeit von Revisionen steigt linear mit der Anzahl der Revisionen (Durchlauf durch Blockliste)
- Revisionsidentifikation nur durch Nummer und Datum

Offene Probleme

- Kein Konfigurationsbegriff und kein Variantenbegriff
- Keine Unterstützung zur Verwaltung von Konsistenzbeziehungen zwischen verschiedenen Objekten

Probleme mit Schreibsperren SCCS realisiert ein sogenanntes "pessimistisches" Sperrkonzept. Gleichzeitige Bearbeitung einer Datei durch mehrere Personen wird verhindert:

- ein Checkout zum Schreiben (**single write access**)
- mehrere Checkouts zum Lesen (**multiple read access**)

In der Praxis kommt es aber öfter vor, dass mehrere Entwickler dieselbe Datei zeitgleich verändern müssen (oder Person mit Schreibrecht "commit" vergisst...)

Unbefriedigende Lösungen

- Entwickler mit Schreibrecht macht "commit" unfertiger Datei, Entwickler mit dringendstem Änderungswunsch macht "checkout" mit Schreibrecht
  - inkonsistente Zustände in Repository, nur einer darf "Arbeiten"
- weitere Entwickler mir Schreibwunsch "stehlen" Datei, machen also "checkout" mit Leserecht und modifizieren Datei trotzdem
  - Problem: Verschmelzen der verschiedenen Änderungen

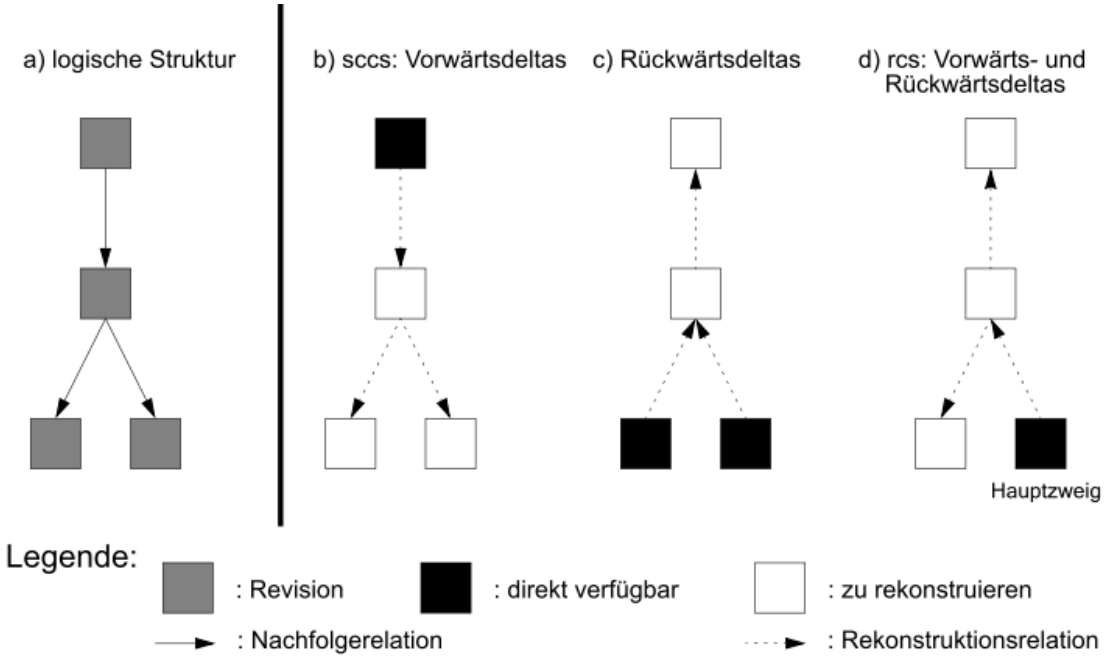
2.2.2 Revision Control System RCS von Berkley/Purdue University

Je Dokument (immer Textdatei) gibt es eine eigene History-Datei, die eine neueste Revision vollständig und andere Revisionen als **Deltas** speichert:

- optionale **Schreibsperren** (verhindern ggf. paralleles Ändern)
- **Revisionsbäume** mit besserem Zugriff auf Revisionen:
  - schneller Zugriff auf neueste Revision auf Hauptzweig
  - langsamer Zugriff auf ältere Revisionen auf Hauptzweig (mit **Rückwärtsdeltas**)
  - langsamer Zugriff auf Revisionen auf Nebenzweigen (mit **Vorwärtsdeltas**).
- Versionsidentifikation auch durch frei wählbare Bezeichner

Offene Probleme Kein Konfigurationsbegriff und kein Variantenbegriff

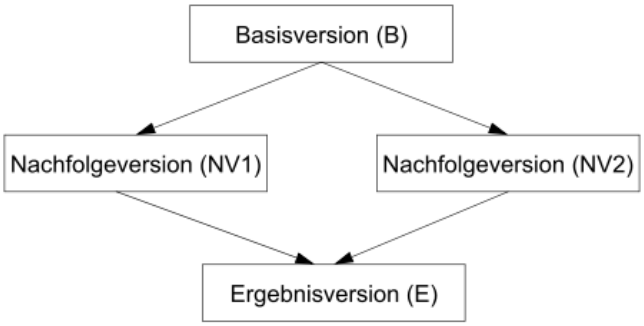
Abbildung 8: Deltaspeicherung von Revisionen als gerichtete Graphen



2.2.3 Concurrent Version (Management) System CVS

Zunächst Aufsatz auf RCS (später Reimplementierung), das Revisionsverwaltung für ganze Directorybäume durchführt und zusätzlich anbietet:

Abbildung 9: Dreiwegeverschmelzen von (Text-)Dateien:



- optimistisches Sperrkonzept mit (fast) **automatischem Verschmelzen** (merge) von parallel durchgeführten Änderungen in verschiedenen privaten Arbeitsbereichen
  - Dreiwegeverschmelzen mit manueller Konfliktbehebung
- **Revisionsidentifikation** und damit rudimentäres **Releasemanagement** auch durch frei wählbare Bezeichner
  - Auszeichnen zusammengehöriger Revisionen durch symbolische Namen (mit Hilfe sogenannter Tags)
- diverse **Hilfsprogramme**
  - wie z.B. "cvsann", das jeder Zeile einer Textdatei die Information voranstellt, wann sie von wem zum letzten Mal geändert wurde

Optimistisches Sperrkonzept mit "Merge"

- Entwickler A macht **checkout** einer Revision n
- Entwickler A verändert Revision n lokal zu n1
- Entwickler B macht **checkout** derselben Revision n
- Entwickler B verändert Revision n lokal zu n2
- Entwickler B macht **commit** seiner geänderten Revision n2
- Entwickler A versucht commit seiner geänderten Revision n1
  - wird mit Fehlermeldung abgebrochen
- Entwickler A macht **update** seiner geänderten Revision n1

  - automatisches merge von n1 und n2 mit Basis n führt zu n'

- Entwickler A löst Verschmelzungskonflikte manuell auf und erzeugt n"
- Entwickler A macht **commit** von Revision n" (inklusive Änderungen von B)

Synchronisierung mit Watch/Edit/Unedit: In manchen Fällen will man wegen größerer Umbauten eine Datei(-Revision) "ungestört" bearbeiten, also zumindest eine Meldung erhalten, wenn andere Entwickler dieselbe Datei bearbeiten (um sie zu warnen):

- **watch on** schaltet das Beobachten einer Datei ein; beim checkout wird die gewünschte Revision der Datei nur mit Leserecht lokal zur Verfügung gestellt
- **watch off** ist das Gegenstück zu watch on
- **watch add** nimmt Entwickler in Beobachtungsliste für Datei auf, für die er vorher ein checkout gemacht hat (Änderungen an Revisionen der Datei werden per Email allen Entwicklern auf Beobachtungsliste gemeldet)
- **watch remove** entfernt Entwickler von Beobachtungsliste für Datei
- **edit** verschafft Entwickler Schreibrecht auf lokal verfügbarer Revision einer Datei und meldet das anderen „interessierten" Entwicklern (enthält watch add)
- **unedit** nimmt Schreibrecht zurück und meldet das (enthält watch remove)

Identifikation gewünschter Revisionen - Zusammenfassung:

1. Verwendung von **Revisionsnummern** (Identifikatoren): jede Revision einer Datei erhält eine eigene eindeutige Nummer, über die sie angesprochen wird; Nummer werden nach einem bestimmten Schema erzeugt.
2. Verwendung von **Attributen** (Tags): Revisionen werden durch Attribute und deren Werte indirekt identifiziert; Beispiele sind:
   - Kunde (für den Revision erstellt wurde)
   - Entwicklungssprache (Java, C,..)
   - Entwicklungsstatus (in Bearbeitung, getestet, freigegeben,..)
3. Verwendung von **Zeitstempeln** (Spezialfall der attributbasierten Identifikation): für jede Revision ist der Zeitpunkt ihrer Fertigstellung (commit) bekannt; deshalb können Revisionen über den Zeitraum ihrer Erstellung (jüngste Revision, vor Mai 2003, etc.) identifiziert werden.

Offene Probleme mit Deltaspeicherung und Verschmelzen

- Berechnung von **Deltas** funktioniert hervorragend für Textdateien (mit Zeilenenden als "Synchronisationspunkte" für Vergleich)
- Berechnung minimaler Deltas von Binärdateien ist wesentlich schwieriger
- Textverarbeitungsprogramme wie Word oder CASE-Tools besitzen deshalb eigene Algorithmen zur Berechnung (und Anzeige) von Deltas
- Verschmelzung von Textdateien funktioniert meist gut (kann aber zu tückischen Inkonsistenzen führen)
- Verschmelzung von Binärdateien oder Grafiken/Diagrammen ist im allgemeinen nicht möglich
- Programme wie Word oder CASE-Tools besitzen deshalb eigene Verschmelzungsfunktionen

Verbleibende Mängel von CVS

- zugeschnitten auf Textdateien bei (Deltaberechnung und Verschmelzen)
- keine Versionierung von Verzeichnisstrukturen (Directories)
- nicht integriert mit "Build-" und "Changemanagement"
- keine gute Unterstützung für geographisch verteilte Software-Entwicklung Inhalt...

Aber:

- Als "Open Software" ohne Anschaffungskosten erhältlich
- Administrationsaufwand hält sich in Grenzen
- für Build- und Changemanagement gibt es ergänzende Punkte

2.2.4 CVS-Nachfolger Subversion (SVN

- immer ganze Dateibäume werden durch commit versioniert (inklusive Erzeugen, Löschen, Kopieren und Umbenennen von Dateien und Unterverzeichnissen)
- commit ganzer Verzeichnisse ist eine atomare Aktion (auch bei Server-Abstürzen, Netzwerkzusammenbrüchen, ...), die ganz oder gar nicht erfolgt
- Metadaten (Dateiattribute, Tags) werden als versionierte Objekte unterstützt
- Deltaberechnung funktioniert für beliebige Dateien (auch Binärdateien); Verschmelzungsoperationen sind aber weiterhin eher auf Textdateien zugeschnitten

- geographisch verteilte Softwareentwicklung wird besser unterstützt; Dateibäume und Dateien werden durch URLs identifiziert, Datenaustausch kann über http-Protokoll erfolgen
- Ungeöhnliche, aber einfache bzw. effiziente Behandlung von mehreren Entwicklungszweigen sowie von Systemrevisionen mit bestimmten Eigenschaften (Tags) als "lazy copies" (keine echten Kopien, sondern nur neue Verweise/Links)

Einsatz von merge in SVN svn merge -r <snapshopt1>:<snapshot2> <source> [<wcpath>]
Dieses Kommando berechnet

- alle Unterschiede zwischen <snapshot1> und <snapshot2> (als Vorwärts- bzw. Rückwärtsdelta) des Verzeichnisses <source> mit allen Unterverzeichnissen und Dateien
- und wendet die Änderungen auf die Dateien im aktuellen Workspace-Verzeichnis (ggf. angegeben durch <wcpath> = Working Copy Path) an
- geändert werden dabei einzelne Dateien und ganze (Unter-)Verzeichnisse

Variante des merge-Kommandos svn merge <src1>[@<snapshot1>]<src2>[@<snapshot2>][<wcpath>]
Dieses Kommando wendet das Delta zweier verschiedener Dateien oder Unterverzeichnisse vorgegebener Revisionen (durch Snapshot-Nummer) auf den aktuellen Workspace an. Wird keine Revision angegeben, so wird jeweils die neueste Revision (Head) verwendet.

Delta-Speicherung in svn:

- BDB-Lösung (basierend auf Berkely Database; älterer Ansatz);
  - alles wird in einer Datenbank gespeichert
  - Revisionen werden als Rückwärts-Deltas zu Nachfolgern gespeichert (ähnlich wie bei RCS)
- FSFS-Lösung (File System on top of File System; neuer Ansatz)
  - Versioniertes Dateisystem, das auf einem "normalen" Dateisystem aufsetzt
  - Revisionen werden als Vorwärts-Deltas zu direkten oder indirekten Vorgängern gespeichert
  - Die erste Revision ist eine leere Datei
  - sogenannter "Skip-List"-Ansatz reduziert Rekonstruktionszeiten durch geschickte Auswahl der Vorgänger

Vorwärts-Delta-Speicherung mit "Skip-List"- Ansatz in SVN Subversion kehrt für die effiziente Speicherung von Revisionen zum "Vorwärtsdelta"-Ansatz von SCCS zurück, benötigt aber bei einer Liste von n aufeinander folgenden Revisionen nur log n Rekonstruktionsschrite. Das funktioniert wie folgt mit $r_i = i - te$ Revision und $d_{i,j} = Delta$ von r_i nach r_j:

- Revision r_0 ist immer leere Datei.
- Revision r_1 wird durch Anwendung des Deltas $d_{0,1}$ auf r_0 erzeugt.
- Revision r_2 wird durch Anwendung des Deltas $d_{0,2} = d_{0,1} und d_{0,2}$ auf r_0 erzeugt.
- Revision r_3 wird durch Anwendung des Deltas $d_{2,3}$ auf r_2 erzeugt, das seinerseits wiederum durch ... erzeugt wird.

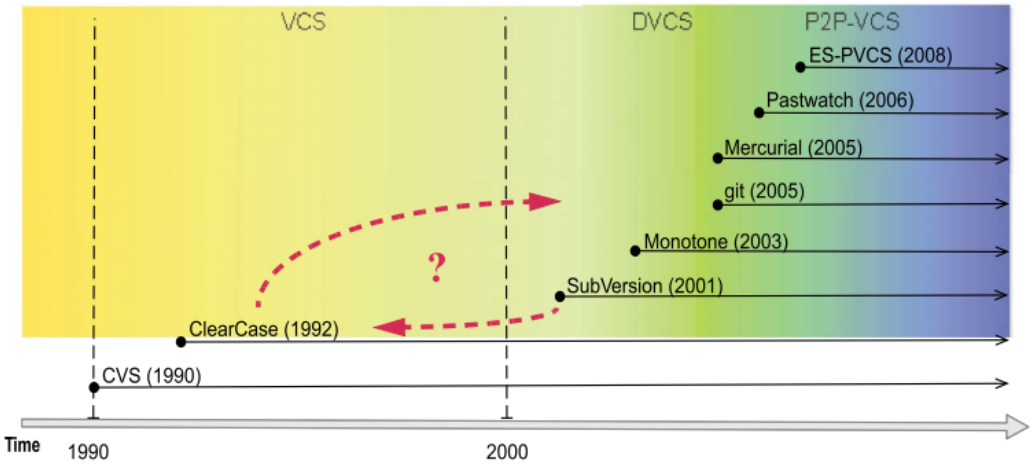
Achtung: Die Konkatenation $d_{i,j} und d_{j,k}$ zweier Deltas ist oft **deutlich kürzer** als die Summe der Länge der beiden konkatenierten Deltas (aufgrund sich aufhebender Editierschritte).

Für die Berechnung des direkten Vorgängers r' zu einer Revision r wird das letzte 1-Bit der Binärdarstellung von r auf "0" gesetzt; die sich daraus ergebende Zahl legt r' fest, auf das Delta $d_{r,r'}$ zur Konstruktion von r angwendet wird (rekursiver Prozess!).

2.2.5 Geschichte verteilter Versionierungs-Systeme

VCS = Version Control System
DVCS = Distributed Version Control System

Abbildung 10: Geschichte verteilter Versionierungs-Systeme



Anmerkung zu "Urahn" ClearCase: Das System ist ein Grenzfall zwischen VCS und DVCS. Es handelt sich zwar um kein reines CSS mehr, aber die Verteilung auf mehrere Server und deren Kooperation hat doch noch sehr zentralistischen Charakter im Gegensatz zu den "reinen" DVCS, die auf dem Rechner eines jeden Entwicklers eine eigene Instanz laufen haben.

2.2.6 "Echt" verteilte Versionierungs-Systeme:

Jeder Benutzer hat sein eigenes vollständiges Repository (statt globale Repositories für Teilgruppen von Entwicklern wie bei ClearCase).

- **Push:** Revisionen zu anderen Repositories aktiv propagieren
- **Pull:** Revisionen von anderen Repositories aktiv holen

Prinzipien "echt" verteilter Versionierungs-Systeme wie "git":

- Jeder Entwickler hat eigenes lokales Versionierungs-Repository mit Snapshots wie bei SVN (Subversion) zusätzlich zu üblichen Arbeitskopien von Dateien
- Check-Out, Commit, ... arbeiten erst mal nur auf eigenen lokalem Repository
- Austausch zwischen Repositories erfolgt mit Push und Pull (zwischen bekannten Personen/Rechnern) via Email oder ssh oder ...
- Bei Push/Pull werden parallele Revisionen angelegt, die dann mit "merge" zusammengeführt werden (müssen)

Bewertung:

- + Entwickler können auch "offline" mit eigenem Repository arbeiten
- + Änderungen können zu selbst gewähtem Zeitpunkt integriert werden (merge)
- **-** Nach Platten-Crash ist lokales Repository weg
- **-** Hoher Aufwand für Verteilung von Änderungen an andere Entwickler (merge)

2.2.7 Verteilte "Peer-to-Peer"-Versionierungs-Systeme:

- **Virtuell** existiert ein **gemeinsames globales Repository** für alle Entwickler
- Es gibt dennoch keinen ausgewählten zentralen Server
- Tatsächlich besitzt jeder Peer ein eigenes Repository (mit allen Dateien oder ausgewählter Menge von Revisionen)
- Das P2P-Versionierungs-System ist "immun" gegen Ausscheiden einzelner Peers (Revisionen werden auf mehreren Peers gehalten)
- Bei **Netzpartitionierungen** (Spezialfall: einzelner Entwickler arbeitet "offline" wird je Partition ein eigener Branch angelegt
- Schließen sich Partitionen wieder zusammen, dann werden Branches wieder verschmolzen (im allgemeinen mit Entwicklerhilfe)

Fazit: P2P-Versionierungssysteme sollen die Vorteile von VCS und DVCS vereinen!

2.3 Variantenmanagement (Software-Produktlinien

Das Variantenmanagement befasst sich mit der Verwaltung nebeneinander existierender Versionen eines Dokuments, die jeweils eine zeitliche Entwicklungsgeschichte besitzen

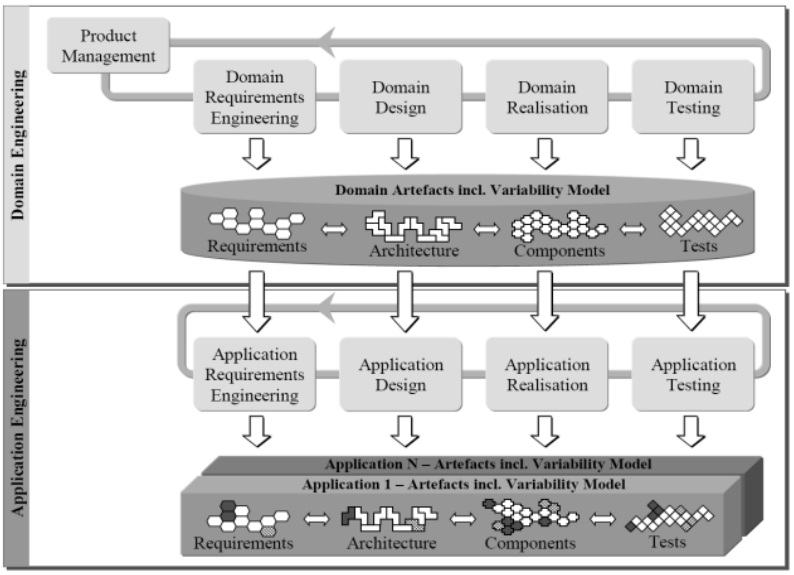
Software-Produktlinien-Entwicklung = "Variantenmanagement im Großen" Eine **Software-Produkt-Linie** (SPL) ist eine Menge "verwandter" Softwareysteme

- für eine bestimmte **Anwedungsdomäne**
- die auf Basis einer gemeinsamen **Plattform** (Rahmenwerk)

entwickelt werden. Die Plattform enthält alle SW-Artefakte, die für alle Instanzen der Produktlinie gleich sind. Damit ist die Software-Produktlinien-Entwicklung (SPLE = Software Product Line Engineering) eine logische Verallgemeinerung des Variantenmanagements eines Softwaresystems.
Man unterscheidet beim SPLE zwischen

- der **Domänenentwicklung** (Domain Engineering) der gemeinsamen Plattform (development **for** reuse)
- der **Anwendungsentwicklung** (Application Engineering) von SPL-Instanzen (development **with** reuse)

Abbildung 11: Domänen- und Anwendungsentwicklung



Variabilitätsmodell (Feature-Modell) einer SPL: So genannte Variabilitäts- oder Fature-Modelle beschreiben alle möglichen Eigenschaften einer Software-Produktlinie und zulässige Kombinationen (Konfigurationen, Varianten, Instanzen) dieser Produktlinie.

Eine Vielzahl von Notationen und Werkzeugen unterstützen die Erstellung solcher Modelle durch

- Festlegung aller auswählbaren Eigenschaften/Merkmale = **Features** der Instanzen der Produktlinie (vor allem die optionalen/alternativen Eigenschaften)
- **hierarchische Zerlegung** von Merkmalen in Untermerkmale
- Festlegung von **Auswahl-Optionen** der Art eins-aus-n, m-aus-n,..
- Definition von **Abhängigkeiten** und Ausschlusskriterien einzelner Merkmale in ganz verschiedenen Teilhierarchien

Werkzeugunterstützung für SPL-Entwicklung:

- (grafische) Erstellung von Variabilitätsmodellen
- Unterstützung bei der Auswahl erlaubter Merkmalkombinationen
- Erzeugung von Produktinstanzen (Implementierungen) für bestimmte Merkmalkombinationen
- Unterstützung beim systematischen Test von SPLs

Grafische Notation von FeatureIDE:

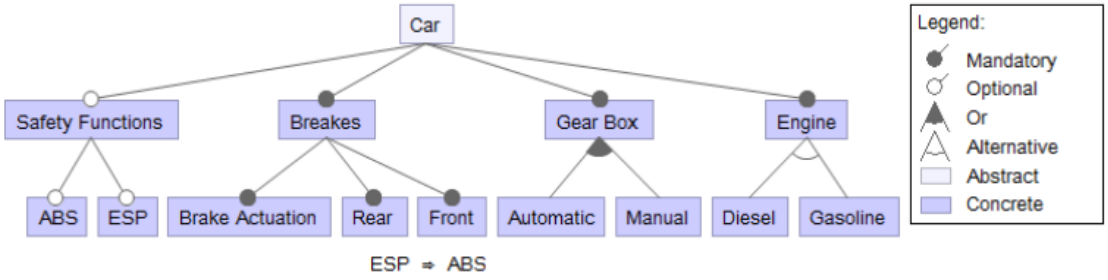
- **Mandatory:** ist immer ausgewählt (falls Elternfeature gewählt)
- **Optional:** kann beliebig an- oder abgewählt werden (falls Elternfeature gewählt)
- **Or:** mindestens ein Feature ist auszuwählen (falls Elternfeature gewählt)
- **Alternative:** genau eins muss ausgewählt werden
- **x -> y:** wenn X gewählt wird, dann auch Y

2.3.1 Variantenmanagement für Quelltextdateien:

Hierfür gibt es kaum eigene Unterstützung. Im wesentlichen bleiben folgende Möglichkeiten zur Verwaltung verschiedener Varianten eines Dokuments:

- Mit Hilfe der **Versionsverwaltung** wird je Variante ein eigener Entwicklungszweig gepflegt (mit entsprechendem Tag). Änderungen von einem Zweig werden mit Hilfe der Dreiwegeverschmelzung in andere Zweige propagiert.

Abbildung 12: Grafische Notation von FeatureIDE



- Die verschiedenen Varianten werden alle in einer Datei gespeichert; durch **Bedingungsmakros** (#ifdef Unix ...) werden die für eine Variante benötigten Quelltextteile ein- und ausgeblendet (siehe pure::variants)
- Die verschiedenen Varianten einer Klassenimplementierung werden in Unterklassen ausgelagert (Einsatz von **Vererbung**)
- Benötigte Varianten werden (aus einer domänenspezifischen Beschreibung) **generiert** (mit Quelltexttransformationen, aspektorientierter Programmierung, ...)
- **Achtung**: Auswahl/Konfiguration/Transformation benötigter Quelltextdateien durch das Build-Management

2.4 Releasemanagement

Ein Release ist eine an Kunden ausgelieferte Konfiguration eines (Software-)Systems bestehend aus ausführbaren Programmen, Bibliotheken, Dokumentation, Quelltexte, Installationsskripte, ...

Das Releasemangement dokumentiert ausgelieferte Konfigurationen und stellt deren Rekonstruierbarkeit sicher.

2.4.1 Aufgaben des Releasemanagement

- Festlegung der (zusätzlichen) Funktionalität eines neuen Releases
- Festlegung des Zeitpunktes der Freigabe eines neuen Releases
- Erstellung und Verbreitung eines Releases (siehe auch Buildmanagement)
- Dokumentation des Releases:
  - welche Revisionen welcher Dateien sind Bestandteil des Releases
  - welche Compilerversion wurde verwendet
  - Betriebssystemversionen auf Entwicklungs- und Zielplattform

2.4.2 Planungsprozess für neue Releases:

1. Vorbedingungen für neues Release werden überprüft:
   - viel Zeit vergangen (neues Release aus Publicity-Gründen)
   - viele Fehler behoben (neues Release mit allen "Patches")
   - viele neue Funktionen hinzugefügt
2. Weiterentwicklung wird eingefroren (freeze):
   - **Feature Freeze** (Soft Freeze): nur noch Fehlerkorrekturen und kleine Verbesserungen erlaubt (nur vermutlich nicht destabilisierende Änderungen)
   - **Code Freeze** (Hard Freeze): nur noch absolut notwendige Änderungen, selbst "gefährliche" Fehlerkorrekturen werden verboten
3. Letzte Fehlerkorrekturen und umfangreiche Qualitätssicherungsmaßnahmen (Tests) werden durchgeführt
4. Release wird freigegeben und weiterverbreitet:
   - Weiterentwicklung (neuer Releases) wird wieder aufgenommen
   - freigegebenes Release muss parallel dazu gepflegt werden

Standardlösung für parallele Pflege und Weiterentwicklung:

- für Weiterentwicklung des nächsten Releases und Wartung des gerade freigegebenen Releases werden unterschiedliche Revisionszweige verwendet
- alle auf dem Wartungszweig liegenden Revisionen erhalten den Namen des Releases (Versionsnummer) als Tag

Strategien für die Vergabe von Versions- und Revisionsnummern:

- Software-Releases erhalten üblicherweise zweistellige **Versionsnummern**
- die erste Stelle wird erhöht, wenn sich die Funktionalität signifikant ändert
- die zweite Stelle wird für kleinere Verbesserungen erhöht
- oft wird erwartet, dass x.2, x.4, ... stabiler als x.1, x.3, ... sind
- die (in cvs vierstelligen) internen **Revisionsnummern** eines KM-Systems müssen mit den extern sichtbaren Versionsnummern nichts zu tun haben
- in den vorigen Beispielen galt aber:
  Versionsnummer = ersten beiden Stellen der Revisionsnummer
- oft werden jedoch die ersten beiden Stellen der Revisionsnummern (in cvs) nicht verwendet und bleiben auf "1.1" gesetzt

2.5 Buildmanagement

Das Buildmanagement (Software Manufacturing) automatisiert den Erzeugungsprozess von Programmen (Software Releases).

2.5.1 Werkzeuge für das Build-Management

- Das Werkzeug **"make"** (im folgenden im Detail präsentiert):
  - deklarativer, regelbasierter Ansatz
  - es werden Abhängigkeiten zwischen Entwicklungsartefakten definiert (mit zusätzlichen Kommandos zur Erzeugung abgeleiteter Artefakte)
  - unabhängig v. Entwicklungsprozess, Programmiersprachen, Werkzeugen
- Das Werkzeug **"Ant"** (make-Nachfolger im Java-Umfeld):
  - gemischt deklarativer bzw. imperativer Ansatz
  - es werden Abhängigkeiten zwischen Aufgaben (Tasks) definiert
  - Tasks können mit Kontrollstrukturen "ausprogrammiert" werden
  - verwendet XML-Syntax für Konfigurationsdateien
  - in Java und vor allem für Java-Projekte entwickelt
  - aber: unabhängig von Entwicklungsprozess, Projektstruktur, ...
- Das Werkzeug **"Maven"**:
  - deklarativer Ansatz mit relativ einfachen Konfigurationsdateien
  - macht viele Annahmen über Entwicklungsprozess, Projektstruktur, ...
  - keine frei definierbaren Regeln für bestimmte Sprachen, Werkzeuge sondern stattdessen Plugins
  - Fokus liegt auf der Unterstützung von Java-Projekten
  - holt sich Plugins, Libraries, ... aus zentralen Repositories
  - oft kombiniert mit "Nexus" für Einrichten lokaler (Cache-)Repositories
- Das Werkzeug **"Jenkins"** (früher: "Hudson"):
  - ergänzt Build-Werkzeuge wie Ant und Maven
  - unterstützt die kontinuierliche Integration/Auslieferung von Produkten
  - automatische Integration von Build-, Test-, Reporting-Werkzeugen

2.5.2 Das Programm "make" zur Automatisierung von Build-Vorgängen:

- **make** wurde in den 70er Jahren "nebenbei" von Stuart F. Feldman für die Erzeugung von Programmen unter Unix programmiert
- Varianten von make gibt es heute auf allen Betriebssystemplattformen (oft integriert in oder spezialisiert auf Compiler einer Programmiersprache)
- make werden durch ein sogenanntes **makefile** Erzeugungsabhängigkeiten zwischen Dateien mitgeteilt
- make benutzt **Zeitmarken** um festzustellen, ob eine Datei noch aktuell ist (Zeitmarke jünger als die Zeitmarken der Dateien, von denen sie abhängt)

Erläuterungen der seltsamen Sonderzeichen-Makros:

- **$@** bezeichnet immer das Ziel einer Regel
- **$*** bezeichnet immer das Ziel einer Regel ohne Suffix
- **$^** bezeichnet alle Objekte einer Regel rechts vom ":"
- **$?** bezeichnet alle Objekte einer Regel rechts vom ":", die neuer als das Ziel sind
- **$<** bezeichnet das erste Objekte rechts vom ":"

Muster-Suche und Substitutionen in make: Oft benötigt man auf der rechten Seite einer Regel oder in der Kommandozeile alle Dateinamen (im aktuellen Verzeichnis), die einen bestimmten **Namensmuster** entsprechen (also etwa mit einem bestimmten Suffix enden).
$(wildcard Muster)
liefert bzw. **sucht** alle Dateien im aktuellen Verzeichnis, die dem angegebenen Muster entsprechen. Das Muster kann beispielsweise folgende Form haben:
$(wildcard*.c)
liefert alle Dateien zurück, die das Suffix ".c" besitzen (der "*" matcht alles).

Will man in einer Liste von Dateinamen (durch Leerzeichen getrennte Strings) einen Teilstring durch einen anderen Teilstring **ersetzen** (substituieren) verwendet man:
$(patsubst Suchmuster, Ersatzstring, Liste von Wörtern)
Folgender Ausdruck ersetzt alle ".c"-Suffixe durch ".o"-Suffixe:
$(patsubst%.c, %.0, Liste von Wörtern)
Das Zeichen "%" bezeichnet den gleichbleibenden Teil bei der Ersetzung.

Muster-Regeln in make: Manchmal will man Regeln schreiben, die alle Dateien mit einem bestimmten Präfix oder Suffix aus einer anderen Datei mit unterschiedlichem =Präfix oder Suffix aber ansonsten gleichem Namen errechnen. So wird z.B. jede c-Datei in eine o-Datei ansonsten gleichen Namens übersetzt. Die Muster-Regel ist für die Definition solcher Abhängigkeiten geeignet:
prefix1%suffix1 : prefix1%.suffix2 ...

Bewertung des Buildmanagements mit make:

- nur ein Bruchteil der Funktionen von make wurde vorgestellt
- ursprüngliches make führt Erzeugungsprozesse sequentiell aus
  - GNU make kann Arbeit auf mehrere Rechner verteilen und parallel durchführbare Erzeugungsschritte gleichzeitig starten
- Steuerung durch Zeitmarken ist sehr "grob":
  - **zu häufiges neu generieren**: irrelevante Änderungen (wie Ändern von Kommentaren für Übersetzung) werden nicht erkannt
  - **zu seltenes neu generieren**: Änderung von Übersetzerversionen, Übersetzungsoptionen etc. werden nicht erkannt
- kaum Verzahnung mit Versionsverwaltung:
  - make wird in aller Regel auf eigenem Repository durchgeführt
  - erzeugte/abgeleitete Objekte werden also immer privat gehalten
- makefiles selbst müssen manuell aktuell gehalten werden
  - programmiersprachenspezifisches makedepend erzeugt makefiles

Erzeugung von Makefiles:

1. in jedem Quelltextverzeichnis gibt es ein "normales" Makefile, das den Übersetzungsprozess mit make in diesem Verzeichnis steuert
2. "normale" Makefiles werden von makedepend durch Analyse von Quelltextdateien erzeugt (in C werden includes von .h-Dateien gesucht)
3. ein "Super"-Makefile sorgt dafür, dass
   - makedepend nach relevanten Quelltextänderungen in den entsprechenden Verzeichnissen aufgerufen wird
   - in allen "normalen" Makefiles dieselben Makrodefinitionen verwendet werden (soweit gewünscht)
   - geänderte Makefiles oder Makrodefinitionen dazu führen, dass in den betroffenen Verzeichnissen alle abgeleiteten Objekte neu erzeugt werden

Generierung von Makefiles mit GNU Autotools Die GNU **Autotools** sind mehrere Werkzeuge, die aufbauend auf make das Build-Management, also die Erstellung und Installation von Softwarekonfigurationen für verschiedenste Zielplattformen, erleichtern.

- Fokus liegt auf Softwareprojekten, in denen vor allem C, C++ oder Fortran (77) als Programmiersprachen eingesetzt werden
- die Werkzeuge eignen sich nicht (kaum) für Java-Projekte oder SW-Entwicklung im Windows-Umfeld
- **Automake** unterstützt Generierung „guter“ makefiles aus deutlich kompakteren Konfigurationsdateien (die von allen populären make-Varianten ausführbar sind)
- **Autoconf** unterstützt Konfigurationsprozess von Software durch Generierung von config.h-Skeletten, Konfigurationsskripten, ...
- schließlich gibt es noch **Libtool**, das Umgang mit Bibliotheken erleichtert

2.5.3 Zusammenfassung und Ratschläge:

- Bereits in kleinsten Projekten ist die Automatisierung von Erzeugungsprozessen (Buildmanagement) ein Muss; in einfachen Fällen die notwendige Unterstützung.
- Im Linux/Unix-Umfeld ist "(GNU) make" das Standardwerkzeug für die Automatisierung von Erzeugungsprozessen.
- In jedem Projekt sollte es genau einen Verantwortlichen für die Pflege von Konfigurationsdateien (Makefiles) und Build-Prozesse geben.
- Für Java-Programmentwicklung gibt es mit Ant ein speziell zugeschnittenes moderneres "Build-Tool"
- Noch "moderner" sind Maven und Jenkins für "Continuous Integration", also die automatisierte, permanente Erzeugung von Software-Releases.
- Nicht generierte (Anteile von) Konfigurationsdateien müssen selbst unbedingt der Versionsverwaltung unterworfen werden.

2.6 Änderungsmanagement

Ein festgelegter Änderungsmanagementprozess sorgt dafür, dass Wünsche für Änderungen an einem Softwaresystem protokolliert, priorisiert und kosteneffektiv realisiert werden.

2.6.1 Änderungsmanagement frei nach [Li02]

Ausfüllen eines Änderungsantragsformulars;
Analyse des Änderungsantrags;
if Änderung notwendig (und noch nicht beantragt **then**

Bewertung wie Änderung zu implementieren ist;
Einschätzung der Änderungskosten;
Einreichen der Änderung bei Kommission;
if Änderung akzeptiert **then**
Durchführen der Änderung für Release...
else
Änderungsantrag ablehnen
else
Änderungsantrag ablehnen

2.6.2 Änderungsmanagement frei nach [Wh00]

1. ein Änderungswunsch (feature request) oder eine Fehlermeldung (bug report) wird eingereicht (Status **submitted**)
2. ein eingereichter Änderungswunsch wird evaluiert und dabei
   - entweder abgelehnt (Status **rejected**)
   - oder als Duplikat erkannt (Status **duplicate**)
   - oder mit Kategorie und Priorität versehen (Status **accepted**)
3. ein akzeptierter Änderungswunsch wird von dem für seine Kategorie Zuständigen
   - für ein bestimmtes Release zur Bearbeitung freigegeben (Status **assigned**)
   - oder vorerst aufgeschoben (Status **postponed**)
4. ein zugewiesener Änderungswunsch wird von dem zuständigen Bearbeiter in Angriff genommen (Status **opened**)
5. irgendwann ist die Bearbeitung eines Änderungswunsches beendet und die Änderung wird zur Prüfung freigegeben (Status **released**)
6. erfüllt die durchgeführte Änderung den Änderungswunsch, so wird schließlich der Änderungswunsch geschlossen (Status **closed**)

Funktionalität von Änderungsmanagement-Werkzeugen:

- Änderungswünsche können über Browserschnittstelle übermittelt werden
- Änderungswünsche werden in Datenbank verwaltet
- Betroffene werden vom Statuswechsel eines Änderungswunsches benachrichtigt
- Integration mit Projektmanagement und KM-Management
- Trendanalyse und Statistiken als Grafiken (Anzahl neuer Fehlermeldungen, ...)

Werkzeuge für das Änderungsmanagement:

- "Open Software"-Produkt **Sourceforge** (GForge), das cvs/svn/git/... mit Änderungsmanagementdiensten kombiniert
- Neuere Alternativen zu Sourceforge: GitHub, GitLab, BitBucket, ...
- "Open Software"-Produkt **Bugzilla** für reines Änderungsmanagement
- flexibles Projektmanagement-Werkzeug **Redmine** mit Aufgabenverwaltung, Versionsverwaltung (cvs/svn/git/...) etc.

2.7 Zusammenfassung

Mit einem für die jeweilige Projektgröße sinnvollen KM-Management steht und fällt die Qualität eines Softwareentwicklungsprozesses, insbesondere nach der Fertigstellung des ersten Softwarereleases.

Meine Ratschläge für das KM-Management:

- besteht ihr System aus mehr als einer Handvoll Dateien, so sollte ein **Buildmanagementsystem** wie make/Ant/Maven verwendet werden
- haben sie Entwicklungszeit von mehr als ein paar Tagen oder mehr als einem Entwickler, so ist ein **Versionsmanagementsystem** wie svn oder git ein Muss
- haben sie mehr als einen Anwender oder mehr als ein Release der Software, so ist ein **Changemanagementsystem** wie in Bugzilla/Redmine einzusetzen
- Werkzeuge wie Sourceforge, GitHub, GitLab, ... , die Versionsmanagement mit Bugtracking, Wiki-Dokumentation etc. kombinieren, immer einsetzen! Alles weitere hängt von Projektgröße und Kontext ab.

3 Statische Programmanalyse & Metriken

Statische Codeanalyse verlangt ein stures, monotones Anwenden von relativ ein- fachen Regeln auf oft umfangreichen Code. Diese Aufgabe erfordert keinerlei Kreativität aber eine sehr große Übersicht und Kontrolle. ... Statische Codeanalyse ist daher prädestiniert zur Automatisierung durch Werkzeuge. Ich empfehle Ihnen, diese Techniken entweder werkzeuggestützt oder gar nicht einzusetzen.

3.1 Einleitung

- Oft (fast immer) finden sich **80% aller Probleme** mit einem Softwaresystem in 20% des entwickelten Codes.

- Die statische Programmanalyse versucht meist werkzeuggestützt frühzeitig die **problematischen 20%** eines Softwaresystems zu finden.
- Statische Analyseverfahren identifizieren Programmteile von **fragwürdiger Qualität** und liefern damit Hinweise auf potentielle Fehlerstellen.
- Statische Analyseverfahren **versuchen** die Qualität von Software zu **messen**, können deshalb zur Festlegung von Qualitätsmaßstäben eingesetzt werden.
- Die statische Programmanalyse setzt **keine vollständig ausführbaren** Programme voraus.
- Die statische Programmanalyse kann also **frühzeitig** bei der Neuentwicklung und kontinuierlich bei der Wartung eines Softwaresystems eingesetzt werden.

Analytisches Qualitätsmanagement zur Fehleridentifikation:

- **analysierende Verfahren:**
  der "Prüfling" (Programm, Modell, Dokumentation) wird von Menschen oder Werkzeugen auf Vorhandensein/Abwesenheit von Eigenschaften untersucht
  - **Review** (Inspektion, Walkthrough): Prüfung durch (Gruppe v.) Menschen
  - **statische Analyse**: werkzeuggestützte Ermittlung von "Anomalien"
  - **(formale) Verifikation**: werkzeuggestützter Beweis von Eigenschaften
- **testende Verfahren:**
  der "Prüfling" wird mit konkreten oder abstrakten Eingabewerten auf einem Rechner ausgeführt
  - **dynamischer Test**: "normale" Ausführung mit ganz konkreten Eingaben
  - **[symbolischer Test**: Ausführung mit symbolischen Eingaben (die oft unendliche Mengen möglicher konkreter Eingaben repräsentieren)]

Arten der Programmanalyse

- **Visualisierung von Programmstrukturen**: unästhetisches Layout liefert Hinweise auf sanierungsbedürftige Teilsysteme
- **manuelle Reviews**: organisiertes Durchlesen u. Diskutieren von Entwicklungsdokumenten durch Menschen
- **Compilerprüfungen**: Syntaxprüfungen, Typprüfungen, ...
- **Programmverifikation und symbolische Ausführung**: Beweis der Korrektheit eines Programms mit Logikkalkül oder anderen mathematischen Mitteln
- **Stilanalysen**: Programmierkonventionen für Programmiersprachen

- **Kontroll- und Datenflussanalysen**: die Programmstruktur wird untersucht, um potentielle Zugriffe auf undefinierte Variablen, möglicherweise nie ausgeführten Code, etc. zu entdecken.
- **Metriken**: Programmeigenschaften werden gemessen und als Zahl repräsentiert - in der Hoffnung, dass kausaler Zusammenhang zwischen Softwarequalität (z.B. Fehlerzahl) und berechneter Maßzahl besteht.

3.2 Softwarearchitekturen und -visualisierung

Große Systeme sind immer in Subsysteme gegliedert, von denen jedes eine Anzahl von Diensten bereitstellt. Der fundamentale Prozess zur Definition dieser Subsysteme und zur Errichtung eines Rahmenwerkes für die Steuerung und Kommunikation dieser Subsysteme wird **Entwurf der Architektur** ... genannt.

Begriffe nach Sommerville:

- Ein **Softwaresystem** besteht aus Teilsystemen, die zusammengehörige Gruppen von Diensten anbieten und möglichst unabhängig voneinander realisiert sind.
- Ein **Teilsystem** kann wiederum aus Teilsystemen aufgebaut werden, die aus Moduln (Paketen) bestehen.
- Ein **Modul** (Paket) bietet über seine Schnittstelle Dienste an und benutzt (importiert) zu ihrer Realisierung Dienste anderer Module (Pakete).
- Ein Modul fasst "verwandte" Prozeduren, **Klassen**, ... zusammen.

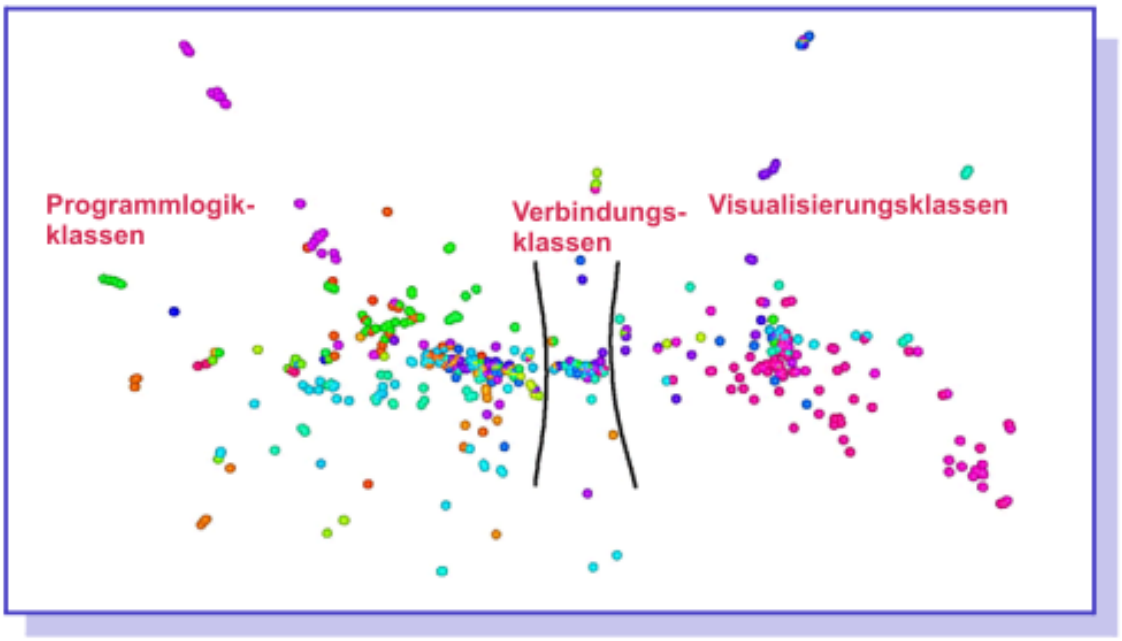
Softwarearchitekturen sind mehr als Teilsysteme und Module:

- In den 80er Jahren wurden Programmarchitekturen mit sogenannten "Module Interconnection Languages (**MIL**)" definiert, die nur Module und Import-Beziehungen kennen.
- Seit den 90er Jahren werden auch "Architecture Description Languages" (**ADLs**) eingesetzt, die Komponenten mit Eingängen und Ausgängen und Verbindungen dazwischen verwenden (siehe Vorlesung „Echtzeitsysteme“);
- Heute verwendet man einen noch umfassenderen Architekturbegriff; in "Software Engineering I" werden folgende **Architektursichten** im Zusammenhang mit der "Unified Modeling Language" (**UML**) eingeführt:
  - Teilsystem-Sicht (Paketdiagramme)
  - Struktur-Sicht (Klassendiagramme, Kollaborationsdiagramme)
  - Kontrollfluss-Sicht (Aktivitätsdiagramme, ...)
  - Datenfluss-Sicht (Aktivitätsdiagramme, ...)

Statische Visualisierung von Programmstruktur mit CrocoCosmos:

- grafische Darstellung eines Rahmenwerks für interaktive Anwendungen
- Klassen eines Teilsystems besitzen dieselbe Farbe, Beziehungen zwischen Klassen (aus Gründen der Übersichtlichkeit weggelasssen)

Abbildung 13: Grafische Darstellung eines Rahmenwerks für interaktive Anwendungen



Programmvisualisierung mit Doxygen und Graphviz:

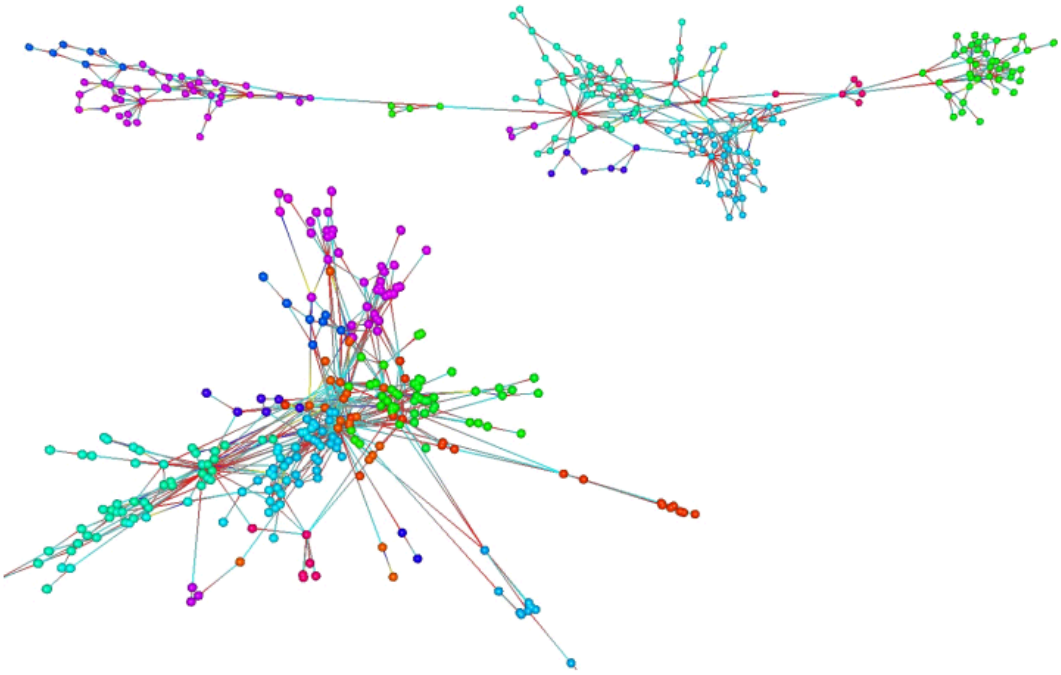
- Doxygen generiert Programmdokumentation, Architekturdiagramme, etc.
- Graphviz (dot) ist ein Hilfsprogramm für Layoutberechnung;

3.3 Strukturierte Gruppenprüfungen (Reviews)

Systematische Verfahren zur gemeinsamen „Durchsicht“ von Dokumenten (wie z.B. erstellte UML-Modelle, implementierten Klassen, ...):

- **Inspektion**: stark formalisiertes Verfahren bei dem Dokument nach genau festgelegter Vorgehensweise durch Gutachterteam untersucht wird
- **Technisches Review**: weniger formale manuelle Prüfmethode; weniger Aufwand als bei Inspektion, ähnlicher Nutzen

Abbildung 14: Ausschnitte der Programmlogik



- **Informelles Review (Walkthrough)**: unstrukturiertere Vorgehensweise; Autor des Dokuments liest vor, Gutachter stellen spontane Fragen
- **Pair Programming**: Programm wird von vornherein zu zweit erstellt

Empirische Ergebnisse zu Programminspektion:

- Prüfaufwand liegt bei ca. 15 bis 20% des Erstellungsaufwandes
- 60 bis 70% der Fehler in einem Dokument können gefunden werden
- Nettonutzen: 20% Ersparnis bei Entwicklung, 30% bei Wartung

Psychologische Probleme bei Reviews:

- **-**: Entwickler sind in aller Regel von der Korrektheit der erzeugten Komponenten überzeugt (ihre Komponenten werden höchstens falsch benutzt)
- **-**: Komponententest wird als lästige Pflicht aufgefasst, die
  - Folgearbeiten mit sich bringt (Fehlerbeseitigung)
  - Glauben in die eigene Unfehlbarkeit erschüttert

- -: Entwickler will eigene Fehler (unbewusst) nicht finden und kann sie auch oft nicht finden (da ggf. sein Testcode von denselben falschen Annahmen ausgeht)
- -: Fehlersuche durch getrennte Testabteilung ist noch ärgerlicher (die sind zu doof zum Entwickeln und weisen mir permanent meine Fehlbarkeit nach)
- +: Inspektion und seine Varianten sind u.a. ein Versuch, diese psychologischen Probleme in den Griff zu bekommen
- +: Rolle des Moderators ist von entscheidender Bedeutung für konstruktiven Verlauf von Inspektionen

Vorgehensweise bei der Inspektion

- **Inspektionsteam** besteht aus Moderator, Autor (passiv), Gutachter(n), Protokollführer und ggf. Vorleser (nicht dabei sind Vorgesetzte des Autors / Manager)
- **Gutachter** sind in aller Regel selbst (in anderen Projekten) Entwickler
- Inspektion **überprüft**, ob:
  - Dokument Spezifikation erfüllt (Implementierung konsistent zu Modell)
  - für Dokumenterstellung vorgeschriebene Standards eingehalten wurden
- Inspektion hat **nicht zum Ziel**:
  - zu untersuchen, wie entdeckte Fehler behoben werden können
  - Beurteilung der Fähigkeiten des Autors
  - lange Diskussion, ob ein entdeckter Fehler tatsächlich ein Fehler ist
- **Inspektionsergebnis:**
  - formalisiertes Inspektionsprotokoll mit Fehlerklassifizierung
  - Fehlerstatistiken zur Verbesserung des Entwicklungsprozesses

Ablauf einer Inspektion:

- **Planung** des gesamten Verfahrens durch Management und Moderator
- **Auslösung** der Inspektion durch Autor eines Dokumentes (z.B. durch Freigabe)
- **Eingangsprüfung** durch Moderator (bei vielen offensichtlichen Fehlern wird das Dokument sofort zurückgewiesen)
- **Einführungssitzung**, bei der Prüfling den Gutachtern vorgestellt wird

- **Individualuntersuchung** des Prüflings (Ausschnitt) durch Gutachter (zur **Vorbereitung** auf gemeinsame Sitzung) anhand ausgeteilter Referenzdokumente
- auf **Inspektionssitzung** werden Prüfergebnisse mitgeteilt und protokolliert sowie Prüfling gemeinsam untersucht
- **Nachbereitung** der Sitzung und **Freigabe** des Prüflings durch Moderator (oder Rückgabe zur Überarbeitung)

Technisches Review (abgeschwächte Form der Inspektion):

- Prozessverbesserung und Erstellung von Statistiken steht nicht im Vordergrund
- Moderator gibt Prüfling nicht frei, sondern nur Empfehlung an Manager
- kein formaler Inspektionsplan mit wohldefinierten Inspektionsregeln
- Ggf. auch Diskussion alternativer Realisierungsansätze

Informelles Review (Walkthrough):

- Autor des Prüflings liest ihn vor (ablauforientiert im Falle von Software)
- Gutachter versuchen beim Vorlesen ohne weitere Vorbereitung Fehler zu finden
- Autor entscheidet selbst über weitere Vorgehensweise
- Zielsetzungen:
  - Fehler/Probleme im Prüfling identifizieren
  - Ausbildung/Einarbeitung von Mitarbeitern

3.4 Kontroll- und Datenflussorientierte Analysen

Der Kontrollflussgraph ist ... eine häufig verwendete Methode zur Darstellung von Programmen. ... Die Verarbeitungssteuerung übernehmen die Kontrollstrukturen der Software unter Nutzung der Datenwerte. Eingabedaten werden gelesen, um Zwischenergebnisse zu bestimmen, die in den Speicher geschrieben werden, ... Die Daten „fließen“ quasi durch die Software; von Eingaben zu Ausgaben.
Ein **gerichteter Graph** G ist ein Tupel (N,E) mit N, eine Menge von **Knoten** (Nodes) und E einer Menge gerichteter **Kanten** (Edges).

Kontrollflussgraph - formale Definition: Ein **Kontrollflussgraph** eines Programms (Prozedur) ist ein Gerichteter Graph $G = (N, E, n_{start}, n_{final})$ mit

- N ist die Menge der **Anweisungen** (Knoten) eines Programms
- $E \subseteq NxN$ ist die Menge der **Zweige** (Kanten) zwischen den Anweisungen des Programms
- $n_{start} \epsilon N$ ist der **Startknoten** des Programms
- $n_{end} \epsilon N$ ist der **Endknoten** des Programms

Es gilt für den Kontrollflussgraphen:

- keine in den Startknoten einlaufenden Kanten
- keine aus dem Endknoten auslaufenden Kanten

Manchmal wird zusätzlich gefordert, dass Kontrollflussgraphen zusammenhängen sind:

- es gibt einen Pfad von n_{start} nach n
- es gibt einen Pfad von n nach $nfinal$

Pfade im Kontrollflussgraphen - formale Definition Ein **Pfad der Länge k** in einem Kontrollflussgraphen ist eine Knotenfolge.
Ein **zyklenfreier Pfad** enthält keinen Knoten zweimal.
Ein **Zyklus** ist ein Pfad mit $n_1 = n_k$.

Kontrollflussgraphsegmente - formale Definition Ein **Segment** oder **Block** eines Kontrollflussgraphen ist ein Knoten s, der einen Teilgraph ersetzt, der aus einem zyklenfreien Pfad besteht mit genau einer auslaufenden und einer einlaufenden Kante.

Datenflussattribute eines Kontrollflussgraphen Die Anweisungen eines Kontrollflussgraphen besitzen Datenflussattribute, die den Zugriffen auf Variable (Parameter,...) in den Anweisungen entsprechen:

- $n \epsilon N$ besitzt das Attribut **def(v)** oder kurz **d(v)**, falls n eine Zuweisung an v enthält (Wert von v definiert);
  gilt auch für Eingabeparameter bei n_{start}
- $n \epsilon N$ besitzt das Attribut **c-use(v)** oder kurz **c(v)**, falls n eine Berechnung mit Zugriff auf v enthält (c=compute);
  implizite Zuweisung an Ausgabeparameter am Ende ist auch c-use
- $n \epsilon N$ besitzt das Attribut **p-use(v)** oder kurz **p(v)**, falls n eine Fallentscheidung mit Zugriff auf v enthält (p=predicative)

- $n \epsilon N$ besitzt das Attribut **r(v)**, falls es das Attribut **c(v)** oder **p(v)** besitzt, also lesend auf v zugreift (r=reference);
  dient nur der Zusammenfassung von c(v) und p(v), wenn der Unterschied c/p irrelevant ist
- $n \epsilon N$ besitzt das Attribut **u(v)**, falls v in dieser Anweisung (noch) keinen definierten Wert (mehr) besitzen kann

Kontrollflussgraphsegmente mit Datenflussattributen

- Sei s ein Kontrollflussgraphsegment mit Pfad $n_{start} = n_1, ..., n_k = n_{final}$ und $attr(n_i)$ die Sequenz der Datenflussattribute der n_i. Dann ist $attr(s) := attr(n_1, ..., attr/n_k)$ die Konkatenierung der Datenflussattributsequenzen von n_1 bis n_k
- Ein Knoten n mit einer Sequenz von Datenflussattributen $attr(n) := attr_1, ..., attr_k$ ist immer eine abkürzende Schreibweise für einen Pfad von Knoten $n_1, ..., n_k$, sodass jeder Knoten n_i genau einen Datenflussattribut besitzt $attr(n_i) := attr_i$
- Allen folgenden Definitionen liegt immer ein "segmentfreier" Kontrollflussgraph zugrunde, in dem jeder Knoten genau ein Datenflussattribut besitzt

Inout-Parameterdiskussion und Aufruf von Prozeduren Die meisten Programmiersprachen erlauben **nicht** die Auszeichnung von Variablen, die reinen Ausgabeparameter-Charakter besitzen; in Pascal/Modula gibt es nur mit VAR gekennzeichnete **Inout-Parameter**, in Sprachen wie C oder C++ hat man nur die Möglichkeit, out-Parameter als Zeiger/Referenzen auf Variable/Objekte zu simulieren.

Für solche Parameter mit Ein- und Ausgabecharakter müssen wir wie folgt vorgehen:

- Deklaration der Prozedur countVowels(IN s:...; INOUT count: ...): für n_{start} wird d(count) sowie d(s) angenommen, da beide Variablen bei der Übergabe einen definierten Wert haben sollten.
  Für n_{final} wird c(count) angenommen, da am Ende der Prozedur der Wert von count durch versteckte Zuweisung an Aufrufstelle übergeben wird.
- Aufruf der Prozedur countVowels(aSentence, aCounter):
  es wird c(aSentence) und c(aCounter) in normaler Anweisung oder p(aSentence) und p(aCounter) in Prädikat gefolgt von d(aCounter) angenommen, da beim Aufruf versteckte Zuweisungen die Werte von aSentence und aCounter an die Parameter s und count zuweisen

Felder, globale Variablen und Strukturen

- Zugriff auf **globale Variablen** in einer Prozedur (Methode): werden bei Ein- und Austritt aus der Prozedur ignoriert und ansonsten wie lokale Variablen behandelt
- Zugriff auf **Felder (Arrays)**:
  - anArray[index] := ... wird als r(index) und d(anArray) gewertet

  - .. := .. anArray[index] ... wird als r(index) und r(anArray) gewertet

- Zugriff auf **Strukturen**: wenn notwendig, können die Bestandteile (Variablen) einer Struktur als einzelne Variablen behandelt werden

Datenflussgraph - formale Definition Ein **Datenflussgraph** $D = (V_d, E_d)$ zu einem Kontrollflussgraphen G eines Programms besteht aus

- einer Menge von Knoten V_d für alle Anweisungen V des Programms
- einer Menge von **Datenflusskanten** E_d:
  $(n_1, n_2) \epsilon E_d$ genau dann, wenn es einen Pfad p im Kontrollflussgraphen G von n_1 nach n_2 gibt, sodass für eine Variable/Parameter v gilt:
  1. d(v) für n_1: Anweisung n_1 definiert Wert für v
  2. r(v) für n_2: Anweisung n_2 benutzt Wert von v
  3. für alle Anweisungen n auf Pfad p (ohne n_1), gilt nicht d(v) für n:
     n ändert also nicht den bei n_1 festgelegten Wert von v

Die Kanten des Datenflussgraphen verbinden also Zuweisungen an Variablen oder Parameter mit den Anweisungen, in denen die zugewiesenen Werte benutzt werden.

Programm-Slices zur Fehlersuche und Änderungsfolgeschätzung Ein **Abhängigkeitsgraph** $A = (N_a, E_a)$ eines Programms ist ein gerichteter Graph, der

- alle Knoten und Kanten des Datenflussgraphen enthält (ohne Zusammenfassung von Teilgraphen zu Segmenten) sowie zusätzlich
- Kanten (des Kontrollflussgraphen) von allen Bedingungen zu direkt kontrollierten Anweisungen (das sind die Anweisungen, deren Ausführung von der Auswertung des betrachteten Bedingung abhängt).

Es gibt zwei Arten von Ausschnitten (Slices) eines Abhängigkeitsgraphen A:

- **Vorwärts-Slice** für Knoten $n \epsilon N_a$ mit Datenflussattribut d(v):
  alle Pfade in A, die von Knoten n ausgehen, der Variable v definiert (der Slice-Graph enthält alle Knoten und Kanten der Pfade)
  - Ein Vorwärts-Slice zu einer Anweisung n, die einer Variable v einen Wert zuweist, bestimmt alle die Stellen eines Programms, die von einer Änderung des zugewiesenen Wertes (Berechnungsvorschrift) betroffen sein könnten.
  - Ein Vorwärts-Slice dient der Abschätzung von Folgen einer Programmänderung.
- **Rückwärts-Slice** für Knoten $n \epsilon N_a$ mit Datenflussattribut r(v):
  alle Pfade in A, die in Knoten n einlaufen, der Variable v referenziert (der Slice-Graph enthält alle Knoten und Kanten der Pfade)

  - Ein **Rückwärts-Slice** zu einer Anweisung n, die eine Variable referenziert, bestimmt alle die Stellen eines Programms, die den Wert der Variable direkt oder indirekt bestimmt haben.
  - Ein Rückwärts-Slice ist bei der Fehlersuche hilfreich, um schnell irrelevante Programmteile ausblenden zu können.

Datenfluss- und Kontrollflussanomalien Eine **Anomalie** eines Programms ist eine verdächtige Stelle in dem Programm. Eine solche verdächtige Stelle ist keine garantiert fehlerhafte Stelle, aber eine potentiell fehlerhafte Stelle im Programm.

Datenflussanomalien (meist deutliche Hinweise auf Fehler) sind etwa:

- es gibt einen Pfad im Kontrollflussgraphen, auf dem eine Variable v referenziert wird bevor sie zum ersten Mal definiert wird (Zugriff auf undefinierte Variable)
- es gibt einen Pfad im Kontrollflussgraphen, auf dem eine Variable v zweimal definiert wird ohne zwischen den Definitionsstellen referenziert zu werden (nutzlose Zuweisung an Variable)

Kontrollflussanomalien sind bei modernen Programmiersprachen von geringerer Bedeutung. Im wesentlichen handelt es sich dabei bei Programmiersprachen ohne "goto"-Anweisungen um nicht erreichbaren Code (ansonsten beispielsweise Sprünge in Schleifen hinein).

Undefined-Reference-Datenflussanomalie: Eine **ur-Datenflussanomalie** bezüglich einer Variable v ist wie folgt definiert:

- es gibt einen segmentfreien Pfad $n_1, ...n_k$
- n_1 hat Attribut u(v), v besitzt also keinen definierten Wert bei n_1
- $n_2, ...n_{k-1}$ hat nicht Attribut d(v), v erhält also keinen definierten Wert
- n_k hat Attribut r(v), auf v wird also lesend zugegriffen

Defined-Defined-Datenflussanomalie: Eine **dd-Datenflussanomalie** bezüglich einer Variable v ist wie folgt definiert:

- es gibt einen segmentfreien Pfad $n_1, ..., n_k$
- n_1 hat Attribut d(v), v erhält also bei n_1 einen neuen Wert
- $n_2, ..., n_{k-1}$ hat nicht Attribut $[r|d|u](v)$, v wird also nicht bis n_k verwendet
- n_k hat Attribut d(v), alter Wert von v wird bei n_k unbenutzt überschrieben

Defined-Undefined-Datenflussanomalie: Eine **du-Datenflussanomalie** bezüglich einer Variable v ist wie folgt definiert:

- es gibt einen segmentfreien Pfad $n_1, ..., n_k$
- n_1 hat Attribut d(v), v erhält also einen definierten Wert bei n_1
- $n_2, ..., n_{k-1}$ hat nicht Attribut $[r|d|u](v)$, v wird also bis n_k nicht verwendet
- n_k hat Attribut u(v), v wird also auf undefiniert gesetzt

Probleme mit statischer Datenflussanalyse für Datenstrukturen

- Funktioniert nicht (gut) für komplexe Datenstrukturen wie Felder (Arrays), bei denen man jede einzelne Komponente wie eigene Variable behandeln müsste
- Noch größere Probleme hat man bei berzeigerten Datenstrukturen
- Unterscheidung von in-, out- und inout-Parametern (in Java, C++, ...)

Probleme mit statischer Datenflussanalyse bei Fallunterscheidungen Problem:

- reale Programme enthalten oft viele Anomalien, die nicht echte Programmierfehler sind (zu viele nutzlose Warnungen werden erzeugt)
- restriktivere Definitionen von sogenannten starken Anomalien übersezen andererseits u.U. zu viele echte Fehler

Lösung:

- zunächst neue Definitionen **"starker" Anomalien** verwenden
- dann bisherige Definitionen von **(schwachen) Anomalien** verwenden

Definitionen starker Datenflussanomalien:

- **starke ur-Anomalie:** zu Anweisungen n mit Attribut u(v) und über einen Pfad von n erreichbarem n' mit r(v) gibt es **keinen** segmentfreien Pfad in G $n = n_1, ..., n_k = n'$ in dem n' nur einmal auftritt und für den gilt:
  es existiert $i \epsilon 2, ..., k-1$ mit n_i besitzt Attribut d(v) oder u(v)
  Idee dieser Definition:
  - von n mit u(v) nach n' mit r(v) gibt es **mindestens** einen Ausführungspfad
  - ausgeschlossen werden **zyklische Pfade** durch n', um so die Analyse auf die erste Ausführung einer Anweisung in einer Schleife einzuschränken

  - auf **keinem** Pfad wird an Variable v ein Wert zugewiesen, bevor bei n' lesend auf v zugegriffen wird
  - des weiteren werden Situationen ausgeschlossen, bei denen die gerade betrachtete ur-Anomalie (teilweise) durch irgendeine andere Anomalie "**überlagert**" wird

- **starke du-Anomalie:** zu Anweisungen n mit Attribut d(v) und über einen Pfad von n erreichbarem n' mit u(v) gibt es **keinen** segmentfreien Pfad in G $n = n_1, ..., n_k = n'$ in dem n' nur einmal auftritt und für den gilt:
  es existiert $i \epsilon 2, ..., k-1$ mit: n_i besitzt Attribut d(v) oder u(v) oder r(v)
  Idee dieser Definition:
  - von n mit d(v) nach n' mit u(v) gibt es **mindestens** einen Ausführungspfad
  - ausgeschlossen werden wieder **bestimmte Zyklen**, um so die Analyse auf die erste Ausführung einer Anweisung in einer Schleife einzuschränken
  - auf **keinem** Pfad wird an Variable v bei n zugewiesener Wert verwendet, bevor er bei n' undefiniert wird
  - des weiteren werden Situationen ausgeschlossen, bei denen die gerade betrachtete du-Anomalie (teilweise) durch irgendeine andere Anomalie "**überlagert**" wird

- **starke dd-Anomalie:** zu Anweisungen n mit Attribut d(v) und über einen Pfad von n erreichbarem n' mit d(v) gibt es **keinen** segmentfreien Pfad in G $n = n_1, ..., n_k = n'$ in dem n' nach n_1 nur einmal auftritt und für den gilt
  es existiert $i \epsilon 2, ..., k-1$ mit: n_i besitzt Attribut d(v) oder u(v) oder r(v)
  Idee dieser Definition:
  - von n mit d(v) nach n' mit d(v) gibt es **mindestens** einen Ausführungspfad
  - ausgeschlossen werden wieder **bestimmte Zyklen**, um so die Analyse auf die erste Ausführung einer Anweisung in einer Schleife einzuschränken
  - auf **keinem** Pfad wird an Variable v bei n zugewiesener Wert verwendet, bevor bei n' erneut ein Wert zugewiesen wird
  - des weiteren werden Situationen ausgeschlossen, bei denen die gerade betrachtete dd-Anomalie (teilweise) durch irgendeine andere Anomalie "**überlagert**" wird

3.5 Softwaremetriken

Die Definition von Software-Maßen basiert auf dem Wunsch, einen quantitativen Zugang zum abstrakten Produkt Software zu gewinnen. Dabei ist zwischen der Vermessung von Eigenschaften einer Software und der quantitativen Kontrolle des zugrundeliegenden Entwicklungsprozesses zu unterscheiden

- **Produktmetriken** messen Eigenschaften der Software
  - Qualität der Software (z.B. Anzahl der gefundenen Fehler)

  - Einhaltung von Standards (z.B. als Anzahl Verletzung von Stilregeln)

- **Prozessmetriken** messen Eigenschaften des Entwicklungsprozesses:
  - Dauer oder Kosten der Entwicklung (z.B. als Mitarbeitermonate)
  - Zufriedenheit des Kunden (z.B. als Anzahl Änderungswünsche)

Gewünschte Eigenschaften von Maß/Metrik

- **Einfachheit**: berechnetes Maß lässt sich einfach interpretieren (z.B. Zeilenzahl einer Datei)
- **Eignung** (Validität): es besteht ein (einfacher) Zusammenhang zwischen der gemessenen Eigenschaft und der interessanten Eigenschaft (z.B. zwischen Programmlänge und Fehleranzahl)
- **Stabilität**: gemessene Werte sind stabil gegenüber Manipulationen untergeordneter Bedeutung (z.B. die Unterschiede zwischen zwei Projekten, wenn man aus erstem Projekt Rückschlüsse auf zweites Projekt ziehen will)
- **Rechtzeitigkeit**: das Maß kann zu einem Zeitpunkt berechnet werden, zu dem es noch zur Steuerung des Entwicklungsprozesses hilfreich ist (Gegenbeispiel: Programmlänge als Maß für Schätzung des Entwicklungsaufwandes)
- **Reproduzierbarkeit**: am besten automatisch berechenbar ohne subjektive Einflussnahme des Messenden (Gegenbeispiel: Beurteilung der Lesbarkeit eines Programms durch manuelle Durchsicht)

Maßskalen:

- **Nominalskala**: frei gewählte Menge von Bezeichnungen wie etwa Programm in C++, Java, Fortran, ... geschrieben
- **Ordinalskala**: geordnete Menge von Bezeichnern wie etwa Programm gut lesbar, einigermaßen lesbar, ... , absolut grauenhaft
- **Rationalskala**: Messwerte können zueinander in Relation gesetzt werden und prozentuale Aussagen mit Multiplikation und Division sind sinnvoll wie etwa Programm A besitzt doppelt/halb so viele Programmzeilen wie Programm B

Berechnung der Regressionsgeraden: Gesucht wird: $Y = b_0 + b_1 X$
Gegeben sind paare von Messwerten: $(x_1, y_1), ..., (x_n, y_n)$ Berechnung der Mittelwerte.
Berechnung von Koeffizient b_1:

$$b_1 = \frac{\frac{1}{n-1} \cdot \sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} \quad (1)$$

Berechnung von Koeffizient b_0: $b_0 = \overline{y} - b_1\overline{x}$

Berechnung des Korrelationskoeffizienten r:

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x}) \cdot (y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2 \cdot \sum_{i=1}^{n}(y_i - \overline{y})^2}} \tag{2}$$

- man kann zeigen, dass der **Korrelationskoeffizient** $r\epsilon[-1... + 1]$ gilt
- die Grenzfälle $r = +1$ und $e = -1$ treten auf, wenn schon alle gemessenen Punkte (x_i, y_i) auf einer Gerade liegen
- die Regressionsgerade steigt für $r = +1$ und fällt für $r = -1$
- für $r = 0$ verläuft die Gerade parallel zur X-Achse, es besteht also kein (linearer) Zusammenhang zwischen X- und >-Werten
- r^2 heißt **Bestimmtheitsmaß** und lässt sich interpretieren als Anteil der durch die Regression erklärten Streuung der Y-Werte
- hat man z.B. r=0.7 erhalten, dann ist $r^2 = 0.49$ der Streuung der Y-Werte werden durch die lineare Abhängigkeit von X erklärt

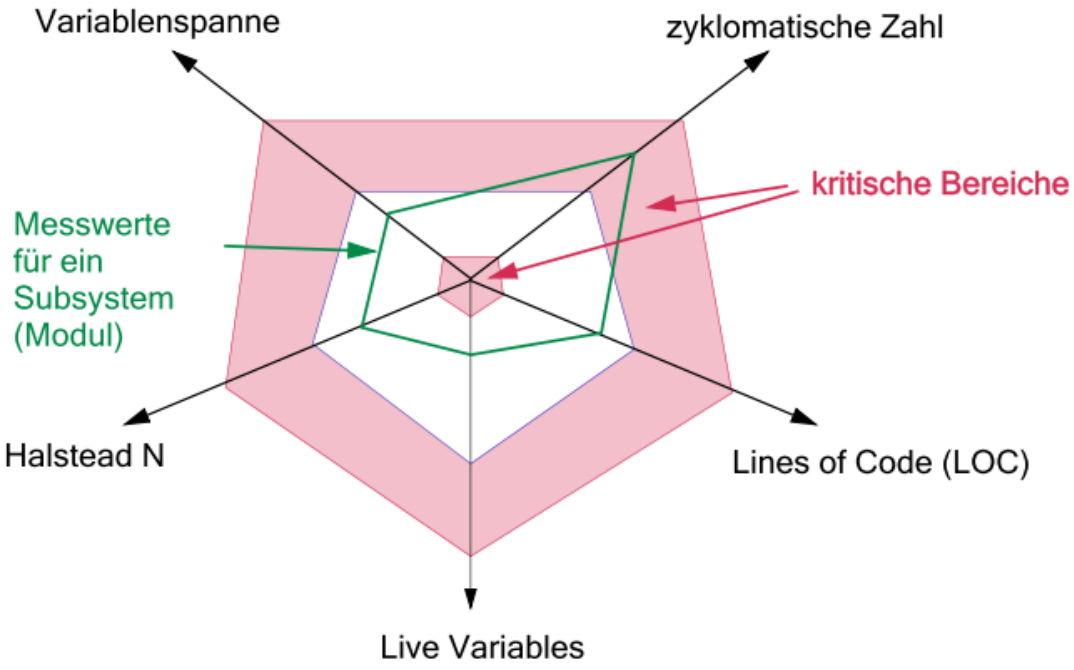
Auswertung von ordinalen/rationalen Metriken:

1. augrund von Erfahrungswerten sind sinnvolle untere und obere Grenzwerte für einen Messwert bekannt
   - alle Komponenten (Module, Klassen, Methoden,...) mit kritischen Werten werden genauer untersucht und ggf. saniert(neu geschrieben)
2. solche Grenzwerte für Messergebnisse sind nicht bekannt
   - alle Komponenten (Module, Klassen, Methoden,...) werden untersucht, deren Messwerte außerhalb des Bereichs liegen, in dem 95% der Messwerte liegen (oder 80% oder...)
3. funktionaler Zusammenhang zwischen Metrik und gewünschtem Qualitätsmerkmal genauer bekannt
   - zulässige Werte für Metrik werden aus Qualitätsanforderungen errechnet (ein Wunschtraum...)

Gleichzeitige Darstellung mehrerer Messwerte mit Kiviatdiagramm

Lines of Code = LOC LOC ist die naheliegendste Metrik. Festlegung des LOC:

- Anzahl aller Zeilen der Textdatei(en) des betrachteten Programmteils
- Anzahl Zeilen der Textdatei(en) ohne Kommentare und Leerzeilen
- Anzahl Trennzeichen zwischen Anweisungen, also ";" oder ...



- Anzahl Trennzeichen wie ";" plus Schlüsselwörter wie "IF",...
- Knoten im Kontrollflussgraphen
- Programmlänge nach Halstead

LOC(Programmteil) = Anzahl der Knoten im Kontrollflussgraphen Idee dieser Maßzahl:

- betrachtete Programmteile oder ganze Programme mit hoher LOC sind zu komplex (no separation of concerns) und deshalb fehlerträchtig
- Programmteile mit geringer LOC sind zu klein und führen zu unnötigen Schnittstellenproblemen

Probleme mit dieser Maßzahl:

- Kanten = Kontrollflusslogik spielen keine Rolle
- wie bewertet man geerbten Code einer Klasse

Zyklomatische Zahl(Komplexität) nach McCabe: ZZ(Programmteil) = |E| - |N| + 2k mit G als Kontrollflussgraph des Untersuchten Programmteils und

- |E|:= Anzahl Kanten von G

- |N|:= Anzahl Knoten von G
- k:= Anzahl Zusammenhangskomponenten von G (Anzahl der nicht miteinander verbundenen Teilgraphen von G)

Regel von McCabe: ZZ eines in sich abgeschlossenen Teilprogramms (Zusammenhangskomponente) sollte nicht höher als 10 sein, da sonst Programm zu komplex und zu schwer zu testen ist

Interpretation und Probleme mit der zyklomatischen Zahl (Komplexität):

- es wird die Anzahl der Verzweigungen (unabhängigen Pfade) in einem Programm gemessen
  - es wird davon ausgegangen, dass jede Zusammenhangskomponente (Teilprogramm) genau einen Eintritts- und einen Austrittsknoten hat
  - damit besitzt jede Zusammenhangskomponente mit n Knoten mindestens n-1 Kanten; diese immer vorhandenen Kanten werden nicht mitgezählt
  - die kleinste Komplexität einer Zusammenhangskomponente soll 1 sein, also wird von der Anzahl der Kanten n abgezogen und 2 addiert
- in GOTO-freien Programmen wird damit genau die Anzahl der bedingten Anweisungen und Schleifen (if/while-Statements) gemessen
- die Zahl ändert sich nicht beim Einfügen normaler Anweisungen
- deshalb ist die Regel von McCabe mit **ZZ(Komponente) $<$ 11** umstritten, da allenfalls eine Aussage über Testaufwand (Anzahl der zu testenden unabhängigen Programmpfade) getroffen wird

Halstead-Metriken - Eingangsgrößen: Die Halstead-Metriken messen verschiedene Eigenschaften einer Software-komponente. Als Eingabe dienen immer:

- η_1: Anzahl der unterschiedlichen Operatoren eines Programms (verwendete arithmetische Operatoren, Prozeduren, Methoden, ...)
- η_2: Anzahl der unterschiedlichen Operanden eines Programms (verwendete Variablen, Parameter, Konstanten, ...)
- N_1: Gesamtzahl der verwendeten Operatoren in einem Programm (jede Verwendungsstelle wird separat gezählt)
- N_2: Gesamtzahl der verwendeten Operanden in einem Programm (jede Verwendungsstelle wird separat gezählt)
- $\eta := \eta_1 + \eta_2$: Anzahl der verwendeten Deklarationen **(Programmvokabular)**
- $N := N_1 + N_2$:Anzahl der angewandten Auftreten von Deklarationen (wird auch „normale“ Programmlänge genannt)

In der Literatur vorgeschlagene Zählregeln für Operatoren in Java:

- Arithmetische und logische Standardoperatoren
- Sonderzweichen (Zuweisung, Konkatenation, Attributselektion)
- Reservierte Java-Schlüsselwörter
- Definitionen von Methoden und Funktionen

Halstead-Metriken - Definition:

1. **Berechnete Programmlänge** $L := \eta_1 log_2 \eta_1 + \eta_2 log_2 \eta_2$ (hängt also nur von Anzahl verwendeter Operatoren und Operanden ab; postuliert wird, dass man mit einer festen Anzahl von Operatoren und Operanden immer Programme einer bestimmten logischen Größe schreibt)
2. **Programmgröße** $V = N log_2 \eta$ (Programme Volume) (optimale Codierung des Programms als Bitvektor)
3. ...

Bewertung: Es gibt eine ganze Reihe weiterer Halstead-Metriken, deren Nutzen umstritten ist, und die versuchen zu bewerten:

- **Schwierigkeit** der Erstellung eines Programms
- **Adäquatheit** einer bestimmten Programmiersprache für Problemstellung
- **Aufwand** für Erstellung eines Programms

"Live Variable"-Definition: Die **"Live Variables"**-Metrik berechnet für eine Programmkomponente die durchschnittliche Anzahl lebendiger Variablen dieser Komponente je Knoten des zugehörigen Kontrollflussgraphen; eine Variable ist dabei von ihrer ersten Definitionsstelle (vom Startknoten aus) bis zur letzten Definitions- oder Referenzierungsstelle (vor dem Endknoten) **lebendig**.

Zusammenfassung: Live Variables einer Komponente ist durchschnittliche Anzahl lebendiger Variablen in einem Programm pro Zeile (Knoten im Kontrollflussgraphen). Eine Variable ist von ihrer ersten Definitionsstelle (vom Startknoten aus) bis zur letzten Definitions- oder Referenzierungsstelle (vor dem Endknoten) lebendig.

"Variablenspanne"-Definition: Die **"Variablenspannen"**-Metrik einer Programmkomponente berechnet die durchschnittliche Spanne zweier direkt aufeinander folgender definierender oder referenzierender Auftreten derselben Variable im zugehörigen Kontrollflussgraphen; die Spanne zweier Knoten in einem Kontrollflussgraphen entspricht der Länge des kürzesten Pfades (Anzahl Kanten dieses Pfades) zwischen diesen beiden Knoten.

Zusammenfassung: Variablenspanne einer Komponente ist die durchschnittliche Spanne zweier direkt aufeinander folgender definierender oder referenzierender Auftreten derselben Variable

Anmerkung zu Live Variables und Variablenspanne: Mit diesen beiden Metriken versucht man nicht die "Kontrollflusskomplexität" oder einfache Größe einer Softwarekomponente, sondern die Komplexität des Datenflusses zu bewerten (wieviele Variablen muss man wie lange beim Erstellen von Programmteilen oder beim Nachvollziehen des Programmablaufs "im Kopf behalten").

Überlegungen zu Metriken für objektorientierte Programme: Betrachtet wird oft Kopplung von Klassen = Benutzt-Beziehungen zwischen Klassen:

- **geringer fan-out** (wenige auslaufende Benutzt-Beziehungen) ist positiv, da sich dann eine Klasse auf wenige andere Klassen abstützt
- **hoher fan-in** (viele einlaufende Benutzt-Beziehungen) ist positiv, da dann eine Klasse von vielen Klassen (wieder-)verwendet wird
- beides kann nicht maximiert werden, da über alle Klassen hinweg gilt: **Summe fan-in = Summe fan-out**

Eine Klasse A benutzt eine Klasse B, wenn:

- in A ein Verweis auf Objekt der Klasse B verwendet wird
- in A eine Operation einen Parameter der Klasse B verwendet
- in A eine Operation der Klasse B aufgerufen wird

Gesucht werden Metriken, die neben der Kopplung von Klassen folgende Aspekte in Maßzahlen zusammenfassen:

- die Methoden einer Klasse sollten **enge Bindung (high cohesion)** besitzen, also einem ähnlichen Zweck dienen (wie misst man das?)
- die Klassen einer Vererbungshierarchie sollten ebenfalls **enge Bindung** besitzen
- die in einem Modul bzw. Paket zusammengefassten Klassen oder die in einer Klasse zusammengefassten Methoden sollten **enge Bindung** besitzen
- Klassen in verschiedenen Modulen bzw. Paketen sollte lose gekoppelt sein (wie misst man das?) **(loose coupling)**
- Klassen und Module bzw. Pakete sollten ein Implementierungsgeheimnis verbergen **(data abstraction, encapsulation)**
- ...

Bindungsmetriken - LOCOM (Low Cohesion Metric) Die Bindung der Methoden einer Klasse wird untersucht. Methoden sind eng miteinander gebunden, wenn sie auf viele gemeinsame Attribute oder Felder zugreifen.

LOCOM1

- P := Anzahl der Paare von Methoden ohne gemeinsame Attributzugriffe
- := Anzahl der Paare von Methoden mit gemeinsamen Attributzugriffen
- LOCOM1 := if P>Q then P-Q else 0
- Gewünscht wird Wert von LOCOM1 nahe 0

LOCOM2

- m := Anzahl Methoden m_i einer Klasse
  $m(a_i)$:= Anzahl Methoden die auf Attribut a_i zugreifen
- n := Anzahl Attribute a_i einer Klasse
- LOCOM1 := 1 - (m(a_1)+...+m(a_n))/(m*n)
- Gewünscht wird kleiner Wert von LOCOM2

Weitere Metriken

- **Afferent Coupling (Ca/AC)**: Die Anzahl der Klassen ausserhalb eines betrachteten Teilsystems (Kategorie), die von den Klassen innerhalb des Teilsystems abhängen
- **Efferent Coupling (Ce/EC):** Die Anzahl der Klassen innerhalb eines betrachteten Teilsystems (Kategorie), die von Klassen ausserhalb des betrachteten Teilsystems abhängen
- **Instabilität (I)**: I = Ce / (Ce+Ca)
  - I hat einen Wert zwischen 0 und 1, falls nicht Ce + Ca = 0 gilt mit 0 = max. stabil u. 1 = max. unstabil
  - der Wert 1 besagt, dass Ca = 0 ist; das betrachtete Teilsystem exportiert also nichts nach außen (keine Klassen und deren Methoden)
  - der Wert 0 besagt, dass Ce = 0 ist; das betrachtete Teilsystem importiert also nichts von außen (keine Klassen und deren Methoden)
  - Der "undefinierte" Fall Ca = 0 und Ce = 0 kann nur auf ein (sinnloses) isoliertes Teilsystem zutreffen, das weder importiert noch exportiert

- **Coupling als Change Dependency between Classes (CDBC)**. CDBC bewertet Aufwand, der mit Überarbeitung von CC wegen Änderung in SC verbunden sein könnte (Anzahl potentiell zu überarbeitender Methoden in CC).
  - n falls SC Oberklasse von CC ist (n = Anzahl Methoden in CC)
  - n falls CC ein Attribut des Typs SC hat
  - j falls SC in j Methoden von CC benutzt wird (als Typ lokaler Variable, Parameter oder Methodenaufruf von SC)
- **Encapsulation als Attribute Hiding Factor (AHF)**.
  - Sind alle Attribute als „private“ definiert, dann ist AHF = 1.
  - Summe der Unsichtbarkeiten aller Attribute in allen Klassen geteilt durch die Anzahl aller Attribute
  - Unsichtbarkeit eines Attributs := Prozentzahl der Klassen, für die das Attribut nicht sichtbar ist (abgesehen von eigener Klasse)
- **Tiefe von Vererbungshierarchien:** zu tiefe Hierarchien werden unübersichtlich; man weiss nicht mehr, was man erbt
- **Breite von Vererbungshierarchien:** zu breite Vererbungshierarchien deuten auf Fehlen von zusammenfassenden Klassen hin
- **Anzahl Redefinitionen in einer Klassenhierarchie:** je mehr desto gefährlicher
- **Anzahl Zugriffe auf geerbte Attribute:** sind ebenfalls gefährlich, da beim Ändern von Attributen oder Attributzugriffen in Oberklasse die Zugriffen in den Unterklassen oft vergessen werden

Komplexitätsmaße:

- **Response for Class (RFC):** die Anzahl der in der Klasse deklarierten Methoden + die Anzahl der geerbten Methoden + die Anzahl sichtbarer Methoden anderer Klassen (Alle Methoden, die aufgerufen werden können? Sehr schwammig definiert!!!)
- **Weighted Methods Per Class (WMPC1):**
  die Summe der zyklomatischen Zahlen ZZ aller Methoden der Klasse (ohne geerbte Methoden)
- **Number of Remote Methods (NORM):** die Anzahl der in einer Klasse gerufenen Methoden ”fremder” Klassen (also nicht die Klasse selbst oder eine ihrer Oberklassen)
- **Attribute Complexity (AC):** die gewichtete Summe der Attribute einer Klasse wird gebildet; Gewichte werden gemäß Typen/Klassen der Attribute vergeben.

3.6 Zusammenfassung

Die **Visualisierung von Software** ist sowohl beim "Forward Engineering" für den Entwurf neuer Programmarchitekturen als auch beim „Reverse Engineering“ für das Studium von "Legacy Software" mit unbekannter Programmstruktur sehr hilfreich.

Werkzeugunterstützte **statische Analyseverfahren** helfen frühzeitig bei der Identifikation kritischer Programmstellen. Es sollten folgende Analyseverfahren immer eingesetzt werden:

- **Stilanalyse** (Überprüfung vereinbarter Programmierkonventionen)
- **"dead code"-Analyse** (oft in Compiler eingebaut): nie verwendete Methoden, Variablen, Parameter, ... (wurde bisher nicht angesprochen)
- **Datenflussanalyse** (wenn Werkzeug verfügbar)

Weitere Analyseverfahren und vor allem **Metriken** sollten in großen Projekten zumindest versuchsweise eingesetzt werden.

Vorgehensweise beim Einsatz von Maßen

1. Fragen zur Ausgangssituation
   - In welcher Phase (Aktivitätsbereich) des Softwareentwicklungsprozesses soll eine Verbesserung eingeführt werden (z.B. Design, Codierung, ...)?
   - Was soll damit erreicht werden bzw. welche Art von Fehler soll reduziert werden (z.B. Reduktion C++ Codierungsfehler)?
   - Welche Methode soll eingesetzt werden (z.B. OO-Metriken)?
   - Welche Technik/Werkzeug soll eingesetzt werden
2. **Bewertung des aktuellen Standes** des Entwicklungsprozesses:
   - Welche Kosten u. welcher Aufwand entstehen in welcher Phase?
   - Wie ist die Qualität der Ergebnisse jeder Phase?
   - In welcher Phase entsteht welcher Anteil an Fehlern und welcher Teil der Fehlerbeseitigungskosten?
3. Mittel zur Bestimmung des aktuellen Standes, **zu messende Aspekte:**
   - Kosten- und Zeitverfolgung beim Entwicklungsprozess
   - Definition von Qualitätsmaßen für Produkt pro Phase
   - Erhebung von Fehlerstatistiken
4. **Analyse der Ergebnisse** und Erarbeitung von Verbesserungsvorschlägen:

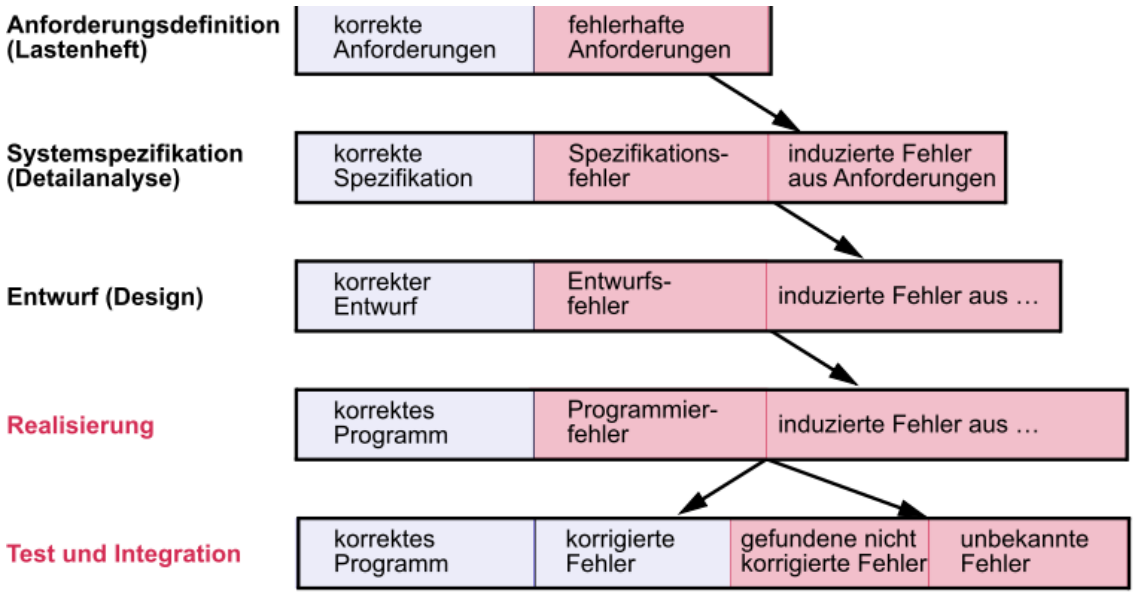
- Auswertung der Maße
- Definition von Zielen auf Basis der Messwerte
- Entscheidung für Verbesserung in bestimmten Phasen
- Auswahl geeigneter Methoden und Werkzeuge
- Einführung der Methoden und Werkzeuge in Entwicklungsprozess

5. Bewertung der durchgeführten Änderungen:
   - Kontinuierliche Weiterauswertung der Maße
   - erneute Analyse nach "Abklingen von Einschwingvorgängen"

4 Dynamische Programmanalysen und Testen

The older I get, the more aggressive I get about testing. I like Kent Beck's rule of thumb that a developer should write at least as much test code as production code. Testing should be a continuous process. No code should be written until you know how to test it. Once you have written it, write the tests for it. Until the test works, you cannot claim to have finished writing the code.

4.1 Einleitung



Fehlerzustand, Fehlerwirkung und Fehlhandlung (DIN 66271):

- **Fehlerzustand (fault) - direkt erkennbar durch statische Tests:**
  - inkorrektes Teilprogramm, inkorrekte Anweisung oder Datendefinition, die Ursache für Fehlerwirkung ist
  - Zustand eines Softwareprodukts oder einer seiner Komponenten, der unter spezifischen Bedingungen eine geforderte Funktion beeinträchtigen kann
- **Fehlerwirkung (failure) - direkt erkennbar durch dynamische Tests:**
  - Wirkung eines Fehlerzustandes, die bei der Ausführung des Testobjektes nach "außen" in Erscheinung tritt
  - Abweichung zwischen spezifiziertem Soll-Wert (Anforderungsdefinition) und beobachtetem Ist-Wert (bzw. Soll- und Ist-Verhalten)
- **Fehlhandlung (error):**
  - menschliche Handlung (des Entwicklers), die zu einem Fehlerzustand in der Software führt
  - **NICHT einbezogen:** menschliche Handlung eines Anwenders, die ein unerwünschtes Ergebnis zur Folge hat

Ursachenkette für Fehler (in Anlehnung an DIN 66271):

- jeder Fehler (**fault**) oder Mangel ist seit dem Zeitpunkt der Entwicklung in der Software vorhanden - Software nützt sich nicht ab
- er ist aufgrund des fehlerhaften Verhaltens (**error**) eines Entwicklers entstanden (und wegen mangelhafter Qualitätssicherungsmaßnahmen nicht entdeckt worden)
- ein Softwarefehler kommt nur bei der Ausführung der Software als Fehlerwirkung (**failure**) zum Tragen und führt dann zu einer ggf. sichtbaren Abweichung des tatsächlichen Programmverhaltens vom gewünschten Programmverhalten
- Fehler in einem Programm können durch andere Fehler **maskiert** werden und kommen somit ggf. nie zum Tragen (bis diese anderen Fehler behoben sind)

Validation und Verifikation (durch dynamische Tests):

- **Validation von Software:**
  - Prüfung, ob die Software das vom Anwender „wirklich“ gewünschte Verhalten zeigt (in einem bestimmten Anwendungsszenario)
  - Haben wir das richtige Softwaresystem realisiert?

- **Verifikation von Software:**
  - Prüfung, ob die Implementierung der Software die Anforderungen erfüllt, die vorab (vertraglich) festgelegt wurden
  - Haben wir das Softwaresystem richtig realisiert?

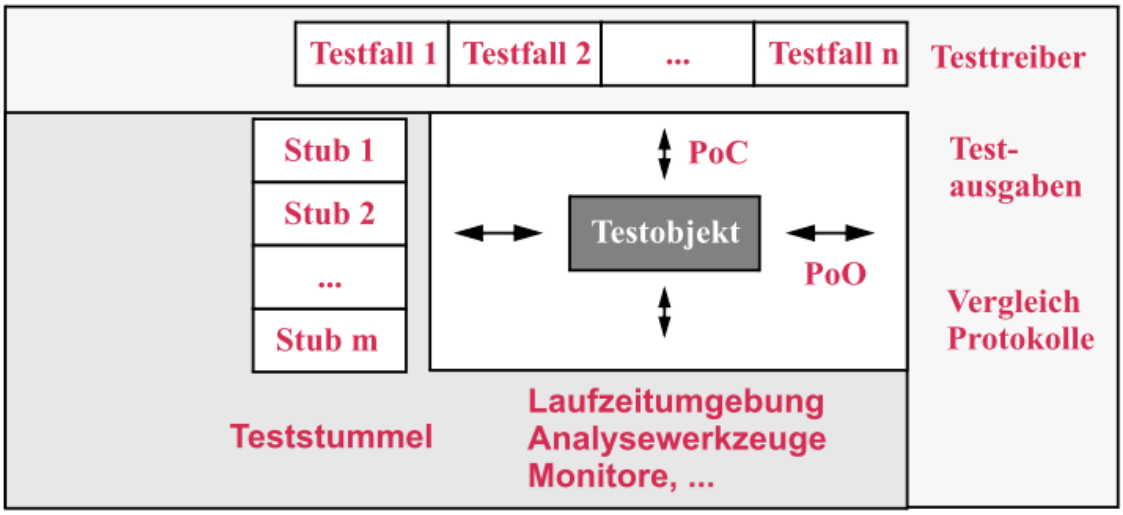
Achtung: Eine "richtig realisierte" = korrekte Software (erfüllt die spezifizierten Anforderungen) muss noch lange nicht das "wirklich" gewünschte Verhalten zeigen!

Typische Programmierfehler nach [BP84]:

- **Berechnungsfehler**: Komponente berechnet falsche Funktion
  - z.B. Konvertierungsfehler in Fortran bei Variablen, die mit I, J oder K anfangen und damit implizit als Integer deklariert sind
- **Schnittstellenfehler**: Inkonsistenz (bezüglich erwarteter Funktionsweise) zwischen Aufrufsstelle und Deklaration
  - Übergabe falscher Parameter, Vertauschen von Parametern
  - Verletzung der Randbedingungen, unter denen aufgerufene Komponente funktioniert
- **Kontrollflussfehler**: Ausführung eines falschen Programmpfades
  - Vertauschung von Anweisungen
  - falsche Kontrollbedingung (z.B. "kleiner" statt "kleiner gleich"), "off by one": Schleife wird einmal zuwenig oder zu oft durchlaufen
- **Datenflussfehler**: falscher Zugriff auf Variablen und Datenstrukturen
  - Variable wird nicht initialisiert (Initialisierungsfehler)
  - falsche Arrayindizierung
  - Zuweisung an falsche Variable
  - Zugriff auf Nil-Pointer oder bereits freigegebenes Objekt
  - Objekt wird nicht freigegeben
- **Zeitfehler**: gefordertes Zeitverhalten wird nicht eingehalten
  - Implementierung ist nicht effizient genug
  - wichtige Interrupts werden zu lange blockiert
- **Redefinitionsfehler:** geerbte Operation wird nicht semantikerhaltend redefiniert
  - ein "Nutzer" der Oberklasse geht von Eigenschaften der aufgerufenen Operation aus, die Redefinition in Unterklasse nicht (mehr) erfüllt

Was wird also getestet: Testverfahren für Softwarekomponenten (Operation, Klasse, Modul/Paket, System) können danach klassifiziert werden, was getestet wird:

- **Funktionalitätstest**: das Ein-/Ausgabeverhalten der Software; das steht beim Testen (zunächst) stark im Vordergrund
- **Benutzbarkeitstest**: es geht um die „gute“ Gestaltung der Benutzeroberfläche; schwieriges Thema, das hier nicht weiter vertieft wird
- **Performanztest**: Laufzeitverhalten und Speicherplatzverbrauch einer Komponente werden gemessen und dabei oft durchlaufene ineffiziente Programmteile oder Speicherlecks identifiziert
- **Lasttest**: die Komponente wird mit schrittweise zunehmender Systemlast **innerhalb des zugelassenen/spezifizierten Bereiches** (für Eingabedaten) getestet
- **Stresstest**: die Systemlast wird solange erhöht, bis sie **außerhalb des zugelassenen/spezifizierten Bereiches** (für Eingabedaten) liegt; damit wird das Verhalten des Systems unter Überlast beobachtet



Wie wird getestet - Aufbau eines Testrahmens

- **Point of Control (PoC):**
  - Schnittstelle, über die Testobjekt mit Testdaten versorgt wird
- **Point of Observation (PoO)**
  - Schnittstelle, über die Reaktionen/Ausgaben des Testobjekts beobachtet werden

Wie wird getestet - Arten von Testverfahren:

- **Funktionstest**(black box test): die interne Struktur der Komponente wird nicht betrachtet; getestet wird Ein-/Ausgabeverhalten gegen Spezifikation (informell oder formal);
- **Strukturtest** (white box test): interne Struktur der Komponente wird zur Testplanung und -Überwachung herangezogen:
  - Kontrollflussgraph
  - Datenflussgraph
  - Automaten
- **Diversifikationstest** Verhalten einer Komponentenversion wird mit Verhalten anderer Komponentenversionen verglichen

Diversifikationstestverfahren: Das Verhalten verschiedener Varianten eines Programms wird verglichen

- **Mutationstestverfahren**: ein Programm wird absichtlich durch Transformationen verändert und damit in aller Regel mit Fehlern versehen
  - Einbau von Fehlern lässt sich (mehr oder weniger) automatisieren
  - eingebaute Fehler entsprechen oft nicht tatsächlich gemachten Fehlern
  - eingebaute Fehler stören Suche nach echten Fehlern
- **N-Versionen-Programmierung**: verschiedene Versionen eines Programms werden völlig unabhängig voneinander entwickelt
  - sehr aufwändig, da man mehrere Entwicklerteams braucht (und gegebenfalls sogar Hardware mehrfach beschaffen muß)
  - Fehler der verschiedenen Teams nicht unabhängig voneinander (z.B. haben alle Versionen dieselben Anforderungsdefinitionsfehlern)

Mutationstestverfahren: Erzeugung von Mutationen durch:
Vertauschen von Anweisungsfolgen, Umdrehen von Kontrollflussbedingungen, Löschen von Zuweisungen, Ändern von Konstanten, ...

Zielsetzungen:

- **Identifikation fehlender Testfälle**: für jeden Mutanten sollte mindestens ein Testfall existieren, der Original und Mutant unterscheiden kann
- **Elimininationen nutzloser Testfälle**: Gruppe von Testfällen verhält sich bezüglich der Erkennung von Mutanten völlig gleich

- Schätzung der Restfehlermenge RF: RF $\tilde{}$ GF*(M/GM-1)
  Annahme: GF/F GM/M (Korrelation gilt allenfalls für bestimmte Fehler)
  mit:
  - GM = Anzahl gefundener eingebauter Fehler (Mutationen)
  - M = Gesamtzahl absichtlich eingebauter Fehler
  - GF = Anzahl gefundener echter Fehler
  - F = Gesamtzahl echter Fehler = GF + RF

N-Versionen-Programmierung (Back-to-Back-Testing): Zielsetzungen:

- eine Version kann als Orakel für die Korrektheit der Ausgaben einer anderen Version herangezogen werden (geht ab 2 Versionen)
- Robustheit der ausgelieferten Software kann erhöht werden durch gleichzeitige Berechnung eines benötigten Ergebnisses durch mehrere Versionen einer Software
- Liefern verschiedene Versionen unterschiedliche Ergebnisse, so wird Mehrheitsentscheidung verwendet (geht ab 3 Versionen)

Probleme:

- N Versionen enthalten mehr Fehler als eine Version: falsche Versionen können richtige überstimmen oder gemeinsame Ressourcen blockieren
- Fehler verschiedener Versionen sind nicht immer unabhängig voneinander: Fehler aus der Anforderungsdefinition oder typische Programmiersprachenfehler oder falsche Algorithmen können alle Versionen enthalten

Exkurs zur Berechnung von Ausfallwahrscheinlichkeiten: Für die Berechnung der Ausfallwahrscheinlichkeit eines (Software-)Systems wird der innere Aufbau des Systems wie folgt als ein gerichteter (Kontrollfluss-)Graph bzw. Abhängigkeitsgraph $S = (N, E, n_{start}, n_{final})$ dargestellt:

- die Knoten N sind die **Komponenten**, aus denen das System besteht
- die Kanten E beschreiben **Berechnungsabhängigkeiten** bzw. -pfade zwischen den Komponenten des Systems
- eine zusätzlich gegebene Funktion A: N ->[0..1] legt für jede Komponente n deren **Ausfallwahrscheinlichkeit** A(n) fest, die unabhängig von den Ausfallwahrscheinlichkeiten anderer Komponenten sein soll

Ein solches System gilt im einfachsten Fall als genau dann **ausgefallen**, sobald es keinen Pfad von n_{start} nach n_{final} gibt, af dem keine Komponente ausgefallen ist.

Einfache Rechenregeln für System-Ausfallwahrscheinlichkeit: Eine **"Parallelschaltung"** zweier unabhängiger Komponenten n_1 und n_2 fällt dann aus, wenn beide Komponenten ausfallen. Das Bild rechts zeigt die Ersetzung der Parallelschaltung der beiden Komponenten n_1 und n_2 durch eine Komponente n mit gleicher Ausfallwahrscheinlichkeit.

Eine **"Reihenschaltung"** zweier unabhängiger Komponenten n_1 und n_2 ist dann nicht ausgefallen, wenn beide Komponenten nicht ausgefallen sind. Das Bild rechts zeigt die Ersetzung der Reihenschaltung von n_1 und n_2 durch eine Komponente n mit gleicher Ausfallwahrscheinlichkeit.

Rekursive Rechenregel für System-Ausfallwahrscheinlichkeit Die Berechnung der Ausfallwahrscheinlichkeit eines Systems S lässt sich bei Fokus auf den Status einer bestimmten Komponente n in zwei Fälle zerlegen:

1. Berechnung der Ausfallwahrscheinlichkeit von S unter der Annahme, dass die **Komponente n ausgefallen** ist $= A(S)_{nistausgefallen}$; n ist ausgefallen ; dabei sei A(n) die Wahrscheinlichkeit, dass Komponente n ausgefallen ist.

2. Berechnung der Ausfallwahrscheinlichkeit von S unter der Annahme, dass die **Komponente n nicht ausgefallen** ist $= A(S)_{nistnichtausgefallen}$; dabei sei 1-A(n) die Wahrscheinlichkeit, dass Komponente n nicht ausgefallen ist.

Damit ergibt sich

$$A(S) = A(n) \cdot A(s)_{nistausgefallen} + (a - A(n)) \cdot A(S)_{nistnichtausgefallen} \tag{3}$$

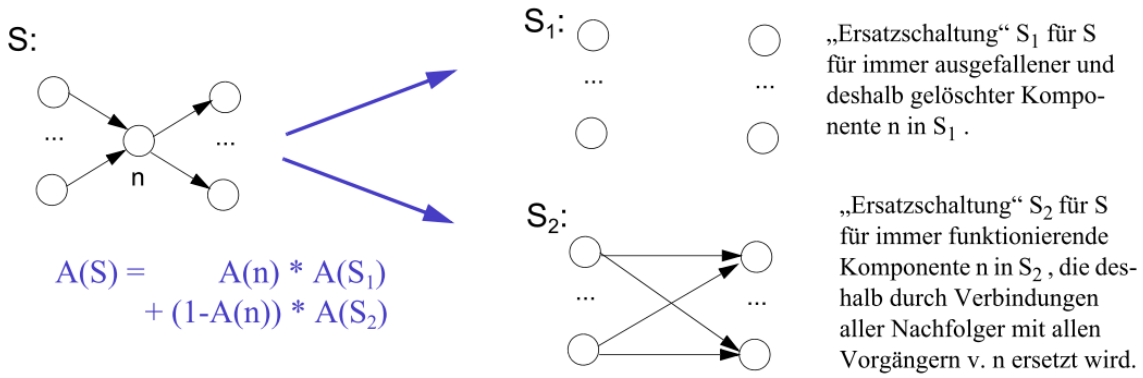
Als Komponente n für die Fallunterscheidung wählt man geschickterweise eine Komponente, die ansonsten die vollständige Dekomposition des betrachteten Graphen in einfach zu behandelnde Serien- und Parallelschaltungen verhindert. Die Berechnung erfolgt durch die Berechnung der Ausfallwahrscheinlichkeiten zweier "Ersatzschaltbilder":

1. $A(S)_{nistausgefallen} = A(S1)$
2. $A(S)_{nistnichtausgefallen} = A(S2)$

In einigen Fällen muss man auch Komponenten behandeln, die nur dann "funktionieren", wenn alle ihre Eingänge korrekte Eingaben erhalten (und damit alle ihre Vorgängerkomponenten nicht ausgefallen sind). Wenn bei einer solchen Komponente eine Vorgängerkomponente ausfällt, fällt die Komponente selbst auch aus.

Alternative Berechnung der Ausfallwahrscheinlichkeit von n Versionen: n parallel geschaltete Versionen enthalten in etwa n-mal so viele Fehler wie eine Version, aber falls Wahrscheinlichkeit A für fehlerhafte Arbeitsweise einer Version v unabhängig vom Ausfall anderer Versionen ist, dann gilt:

- Richtiges Ergebnis werde berechnet, solange höchstens $\lfloor (n+1)/2 \rfloor$ Versionen fehlerhaft arbeiten (n ist sinnvoller Weise ungerade Zahl; ansonsten abrunden)

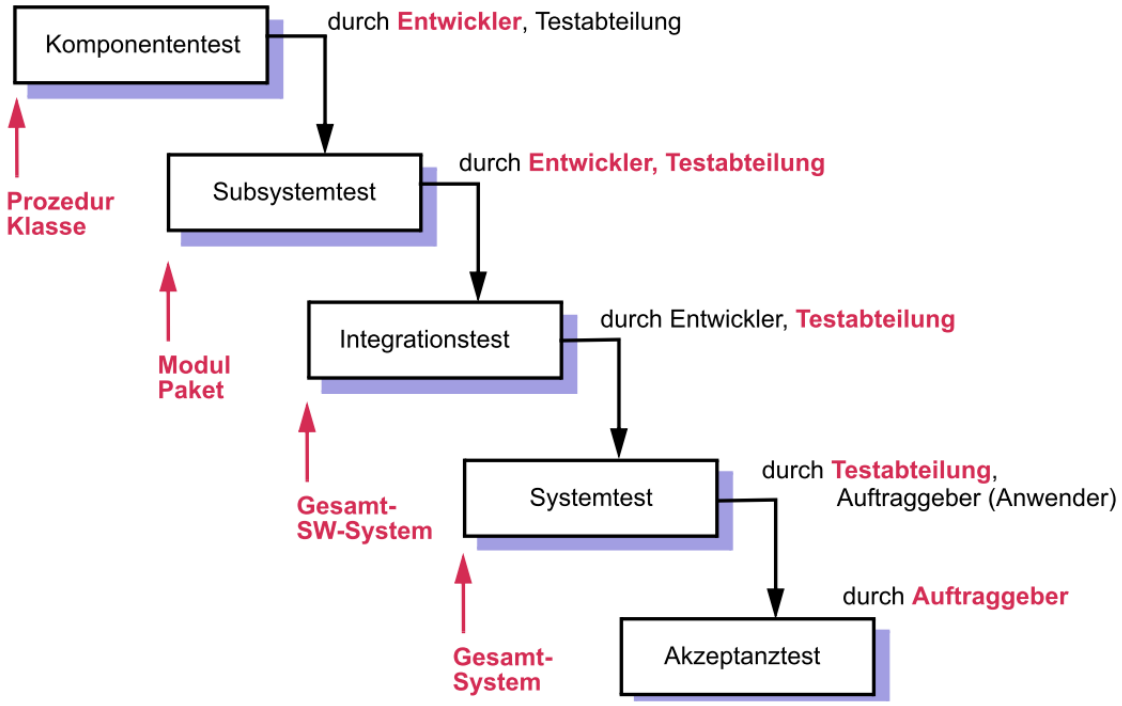


- Wahrscheinlichkeit für gleichzeitigen Ausfall von k unabhängigen Versionen:
$\left(\begin{array}{c} n \\ k \end{array} xA^k x(1-A)^{(n-k)} = \frac{\prod_{i=n-k+1}^{v} i}{\prod_{i=1}^{k} i} xA^k x(1-A)^{(n-k)} \right)$
- Wahrscheinlichkeit für **Ausfall des Gesamtsystems** mit n Versionen (bei n=3: Wahrscheinlichkeit für Ausfall von 2 oder 3 Versionen):
$\sum_{\lfloor (n+1)/2 \rfloor}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) xA^k x(1-A)^{(n-k)}$

Komponententest (Unit-Test) und Subsystemtest:

- jeweils ein **einzelner Softwarebaustein** wird überprüft, isoliert von anderen Softwarebausteinen des Systems
- die betrachtete Komponente (Unit) kann eine Klasse, Paket (Modul) sein
- **Subsystemtest** kann als Test einer besonders großen Komponente aufgefasst werden
- getestet wird gegen die Spezifikation der Schnittstelle der Komponente, dabei betrachtet werden funktionales Verhalten, Robustheit, Effizienz, ...
- Testziele sind das Aufdecken von Berechnungsfehlern, Kontrollflussfehlern, ...
- getestet wird in jedem Fall die Komponente für sich mit
  - **Teststummel** (Platzhalter, Dummies, Stubs) für benötigte Dienste anderer Komponenten
  - **Testtreibern**(Driver) für den (automatisierten) Test der Schnittstelle (für Eingabe von Parametern, Ausgabe von Parametern, ...)

Abbildung 15: Der Testprozess als Bestandteil des Softwareentwicklungsprozesses



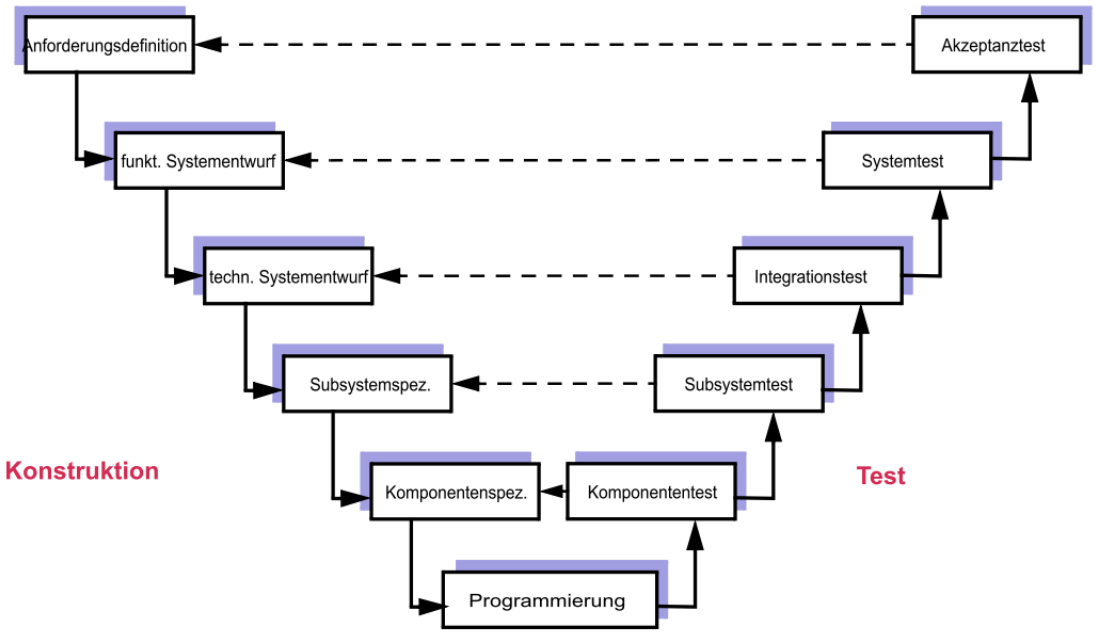
Integrationtest:

- das gesamte Software-System (oder ein abgeschlossenes Teilsystem) wird getestet; Schwerpunkt liegt dabei auf Test des **Zusammenspiels** der Einzelkomponenten
- normalerweise wird vorausgesetzt, dass Einzelkomponenten vorab bereits getestet wurden
- auch hier müssen wieder **Testtreiber** (Testwerkzeuge) verwendet werden, die die zu testende Komponente aufrufen bzw. steuern
- auf **Teststummel** kann meist verzichtet werden, da alle benötigten Teilsysteme zusammen getestet werden
- **Testziel** ist vor allem das Aufdecken von **Schnittstellenfehlern** und insbesondere Fehler beim Austausch von Daten

Gängige Integrationsteststrategien:

- **"Big Bang"-Strategie**: alle Teile sofort integrieren und nur als Gesamtheit testen
  - Lokalisierung von Fehlern schwierig

Abbildung 16: Der Testprozess als Bestandteil des Softwareentwicklungsprozesses



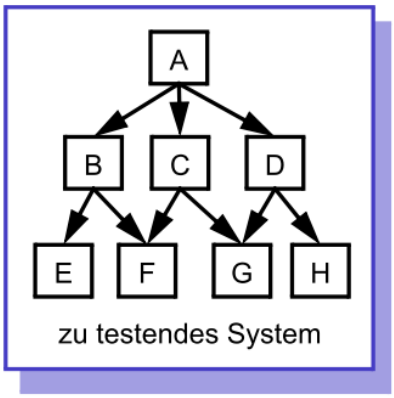
  - Arbeitsteilung kaum möglich
  - Testen beginnt zu spät

- **"Top-down"-Testverfahren:** zuerst A mit Dummies für B,C und D; dann B mit Dummies für E und F,...
  - Erstellung "vernünftiger" Dummies schwierig
  - Test der Basisschicht sehr spät
- **"Bottom-Up"-Testverfahren:** zuerst E,F,G und H mit Testtreibern, die Einbindung in B,C und D simulieren, dann B,C und D mit Testtreiber...
  - Test des Gesamtverhaltens des Systems gegen Lastenheft erst am Ende
  - Designfehler und Effizienzprobleme werden oft erst spät entdeckt
- **Ad-Hoc-Integration:** die Komponenten werden in der (zufälligen) Reihenfolge ihrer Fertigstellung integriert und getestet
- **Backbone-Integration (Inkrementelle Vorgehensweise):** zunächst wird Grundgerüst erstellt, weitere Komponenten werden stückweise hinzugefügt
  - wie erstellt und testet man Grundgerüst (z.B. Top-Down-Testen)

  - Hinzufügen von Komponenten kann bisherige Testergebnisse entwerten

- **Regressionstest (für inkrementelle Vorgehensweise):** da Änderungen neue Fehlerzustände in bereits getesteten Funktionen verursachen (oder bislang maskierte Fehlerzustände sichtbar machen) können werden
  - möglichst viele Tests automatisiert
  - bei jeder Änderung werden alle vorhandenen Tests durchgeführt
  - neue Testergebnisse mit alten Testergebnissen verglichen

Abbildung 17: Gängige Integrationsteststrategien



Systemtest - grundsätzliche Vorgehensweise:

- Nach abgeschlossenem Integrationstest und vor dem Abnahmetest erfolgt der Systemtest beim Softwareentwickler durch Kunden $\alpha - Test$
- Variante: Systemtests bei ausgewählten Pilotkunden vor Ort $\beta - Test$
- Systemtest überprüft aus der **Sicht des Kunden**, ob das Gesamtprodukt die an es gestellten Anforderungen erfüllt (nicht mehr aus Sicht des Entwicklers)
- anstelle von Testtreibern und Teststummeln kommen nun soweit möglich immer die realen (Hardware-)Komponenten zum Einsatz
- Systemtest sollte nicht beim Kunden in der Produktionsumgebung stattfinden, sondern in möglichst realitätsnaher Testumgebung durchgeführt werden
- beim Test sollten die **tatsächlichen Geschäftsprozesse** beim Kunden berücksichtigt werden, in das getestete System eingebettet wird
- dabei durchgeführt werden: Volumen- bzw. Lasttests (große Datenmengen), Stresstests (Überlastung), Test auf Sicherheit, Stabilität, Robustheit, ...

Systemtest - nichtfunktionale Anforderungen:

- **Lasttest**: Messung des Systemverhaltens bei steigender Systemlast
- **Performanztest**: Messung der Verarbeitungsgeschwindigkeit unter bestimmten Randbedingungen
- **Kompatibilität**: Verträglichkeit mit vorhandenen anderen Systemen, korrekter Import und Export externer Datenbestände, ...
- **Benutzungsfreundlichkeit**: übersichtliche Oberfläche, verständliche Fehlermeldungen, Hilfetexte, ... - für die jeweilige Benutzergruppe
- **Benutzerdokumentation**: Vollständigkeit, Verständlichkeit, ...
- **Änderbarkeit, Wartbarkeit**: modulare Systemstruktur, verständliche Entwicklerdokumentation, ...
- ...

Akzeptanztest (Abnahmetest):

- Es handelt sich um eine **spezielle Form des Systemtests**
  - der Kunde ist mit einbezogen bzw. führt den Test durch
  - der Test findet beim Kunden, aber in Testumgebung statt (Test in Produktionsumgebung zu gefährlich)
- auf Basis des Abnahmetests entscheidet Kunde, ob das bestellte Softwaresystem **mangelfrei** ist und die im Lastenheft festgelegten Anforderungen erfüllt
- die durchgeführten Testfälle sollten bereits im **Vertrag** mit dem Kunden spezifiziert sein
- im Rahmen des Abnahmetests wird geprüft, ob System von allen relevanten Anwendergruppen **akzeptiert** wird
- im Rahmen von sogenannten **Feldtests** wird darüber hinaus ggf. das System in verschiedenen Produktionsumgebungen getestet

Testen, testen, ... - wann ist Schluss damit?

- **nie** - nach jeder Programmänderung wird eine große Anzahl von Testfällen automatisch ausgeführt (siehe Regressionstest)
- Testbudget verbraucht bzw. Auslieferungszeitpunkt für Software erreicht (der Kunde testet unfreiwillig weiter ...)

- je Testfall (Zeiteinheit) gefundene Fehlerzahl sinkt unter gegebene Grenze (in der Hoffnung, dass die Anzahl der im Programm verbliebenen Fehler mit der Anzahl der pro Zeiteinheit gefundenen Fehler korreliert)
- n% absichtlich von einer Gruppe implantierter Fehler (seeded bugs) wurden von Testgruppe gefunden (siehe auch Mutationstestverfahren)
- gemäß systematischen Verfahren werden aus allen möglichen Eingabedatenkombinationen typische Repräsentanten ausgewählt und genau diese getestet
- Testfälle decken hinreichend viele (relevante) Programmdurchläufe ab

Die sieben Grundsätze des Testens nach [SL19]:

1. Testen zeigt die Anwesenheit von Fehlern (und nie die Abwesenheit)
2. Vollständiges Testen ist nicht möglich
3. Mit dem Testen frühzeitig beginnen
4. Häufung von Fehlern (in bestimmten Programmteilen)
5. Zunehmende Testresistenz (gegen existierende Tests)
6. Testen ist abhängig vom Umfeld
7. **Trugschluss**: Keine Fehler bedeutet ein brauchbares System

4.2 Laufzeit- und Speicherplatzverbrauchsmessungen

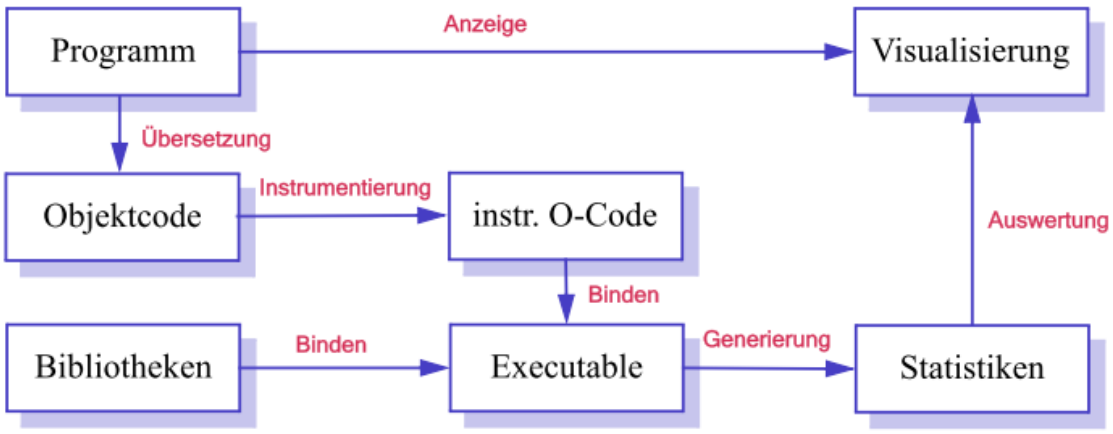
Gemeinsame Eigenschaft aller hier vorgestellten Werkzeuge/Verfahren ist:

- Objektcode für untersuchte Software wird vor Ausführung "instrumentiert" (um zusätzliche Anweisungen ergänzt)
- zusätzliche Anweisungen erzeugen während der Ausführung statistische Daten über Laufzeitverhalten, Speicherplatzverbrauch, ...

Untersuchung des Laufzeitverhaltens eines Programms:

- wie oft wird jede Operation aufgerufen (oder Quellcodezeile durchlaufen)
- welche Operation ruft wie oft welche andere Operation (**descendants**) auf oder von welchen Operationen (**callers**) wird ein Programm wie oft gerufen
- wieviel Prozent der Gesamtlaufzeit wird mit Ausführung einer bestimmten Operation verbracht (ggf. aufgeteilt nach callers und descendants)
- ...

Abbildung 18: Laufzeit- und Speicherplatzverbrauchsmessungenn



Nutzen der ermittelten Daten:

- Operationen, die am meisten Laufzeit in Anspruch nehmen, können leicht identifiziert (und optimiert) werden
- tatsächliche Aufrufabhängigkeiten werden sofort sichtbar
- ...

Untersuchung des Speicherplatzverhaltens eines Programms:

- welche Operationen fordern wieviel Speicherplatz an (geben ihn frei)
- wo wird Freigabe von Speicherplatz vermutlich bzw. bestimmt vergessen (memory leak = Speicherloch):
  - bestimmt vergessen: Objekt lebt noch, kann aber nicht mehr erreicht werden (Garbage Collector von Java würde es entsorgen)
  - vermutlich vergessen: Objekt lebt noch und ist erreichbar, wird aber nicht mehr benutzt (Garbage Collector von Java kann es nicht freigeben)
- wo wird auf bereits freigegebenen Speicherplatz zugegriffen (nur für C++) bzw. wo wird Speicherplatz mehrfach freigegeben (nur für C++)
- wo wird auf nicht initialisierten Speicherplatz zugegriffen; anders als bei der statischen Programmanalyse wird für jede Feldkomponente (Speicherzelle) getrennt Buch darüber geführt
- wo finden Zugriffe jenseits der Grenzen von Arrays statt (Laufzeitfehler in guter Programmiersprache)

4.3 Funktionsorientierte Testverfahren (Blackbox)

Sie testen Implementierung gegen ihre Spezifikation und lassen die interne Programmstruktur unberücksichtigt (Programm wird als "Black-Box" behandelt):

- für **Abnahmetest** ohne Kenntnis des Quellcodes geeignet
- setzt (eigentlich) vollständige und widerspruchsfreie **Spezifikation** voraus (zur Auswahl von Testdaten und Interpretation von Testergebnissen)
- **repräsentative Eingabewerte** müssen ausgewählt werden (man kann im allgemeinen nicht alle Eingabekombinationen testen)
- man braucht "Orakel" für Überprüfung der Korrektheit der Ausgaben (braucht man allerdings bei allen Testverfahren)

Eingaben -> Testobjekt -> Ausgaben -> Orakel -> Ausgabe ok?

Kriterien für die Auswahl von Testdaten: An der Spezifikation orientierte **Äquivalenzklassenbildung**, so dass für alle Werte einer Äquivalenzklasse (Eingabewertklasse) sich das Softwareprodukt "gleich" verhält:

- Unterteilung in Klassen von Eingabewerten, für die das Programm sich laut Spezifikation gleich verhalten muss
- Alle Klassen von Eingabewerten zusammen müssen den ganzen möglichen Eingabewertebereich des betrachteten Programms abdecken
- Aus jeder Äquivalenzklasse wird mindestens ein repräsentativer Wert getestet
- Unterteilung in gültige und ungültige Eingabewerte (fehlerhafte Eingaben, ...) wird durchgeführt und später bei der Auswahl von Testwerten berücksichtigt
- Oft gibt es auch eine gesonderte Betrachtung von Äquivalenzklassen für besonders "große" oder besonders "kleine" gültige Eingaben (Lasttest)
- Hier nicht mit betrachtet wird die Problematik der Suche nach Eingabewerteklassen, die zu bestimmten (Klassen von) Ausgabewerten führen

Regeln für die Festlegung von Eingabewerteklassen:

- für geordnete Wertebereiche:
  - $]uv..ov[$ ist ein offenes Intervall aller Werte zwischen uv und ov (uv und ov selbst gehören nicht dazu)
  - $[uv..ov]$ ist ein geschlossenes Intervall aller Werte zwischen uv und ov (uv und ov selbst gehören dazu)

  - Mischformen $]uv...ov]$ und $[uv..ov[$ sind natürlich erlaubt

- für ganze Zahlen (Integer):
  - $[MinInt..ov]$ für Intervalle mit kleinster darstellbarer Integer-Zahl
  - $[uv..MaxInt]$ für Intervalle mit größter darstellbarer Integer-Zahl
  - offene Intervallgrenzen sind natürlich erlaubt
- für reelle Zahlen (Float) mit Voraussetzung kleinste/größte Zahl nicht darstellbar:
  - $]-\infty..ov]$ oder $]-\infty..ov[$ für nach unten offene Intervalle
  - $[uv..\infty[$ oder $]uv..\infty$ für nach oben offene Intervalle
  - alle Mischformen von Intervallen mit festen unteren und oberen Grenzen
- für Zeichenketten (String):
  - Definition über reguläre Ausdrücke oder Grammatiken
  - Aufzählung konkreter Werte (siehe nächster Punkt)
- für beliebige Wertebereiche:
  - v1 v2 v3 ... vn für Auswahl von genau n verschiedenen Werten
- für zusammengesetzte Wertebereiche:
  - Anwendung der obigen Prinzipien auf die einzelnen Wertebereiche
  - ggf. braucht man noch zusätzliche Einschränkungen (Constraints), die nur bestimmte Wertekombinationen für die Teilkomponenten zulassen
- Eingabewerteklassen, die aus genau einem Wert bestehen:

  v es ist aber auch die Repräsentation v oder v üblich

Auswahl von Testdaten aus Eingabewerteklassen:

- aus jeder Eingabewerteklasse wird mindestens ein Wert ausgewählt (Fehler- und Lasttestklassen werden später gesondert behandelt)
- gewählt werden meist nicht nur die Grenzen selbst, sondern auch die um eins größeren und kleineren Werte (siehe ISTQB-Prüfung; im Folgenden werden aus Platzgründen bei den Beispielen aber nur Grenzwerte selbst ausgewählt)
- als Intervalle dargestellte Eingabewerteklassen werden oft durch die so genannte **Grenzwertanalyse** nochmal in Unterklassen/Teilintervalle zerlegt:
  - $]uv..ov[$ wird zerlegt $[inc(uv)]]inc(uv)..dec(ov)[[dec(ov)]$

  - $[uv..ov]$ wird zerlegt in $[uv]]uv..ov[[ov]$
  - ...
  - $inc(uv)$ liefert den nächstgrößeren Wert zu uv
  - $dec(ov)$ liefert den nächstkleineren Wert zu ov
  - Achtung: bei Float muss für inc und dec die gewünschte Genauigkeit festgelegt werden, mit der aufeinanderfolgende Werte gewählt werden

Zusätzliche Wahl von Testdaten durch Grenzwertanalyse: Bilden Eingabewerteklassen einen Bereich (Intervall), so selektiert die Grenzwertanalyse also immer Werte um die Bereichsgrenzen herum:

- gewählt werden meist nicht nur die Grenzen selbst, sondern auch die um eins größeren und kleineren Werte (im Folgenden aus Platzgründen weggelassen)
- Idee dabei: oft werden Schleifen über Intervallen gebildet, die genau für die Grenzfälle falsch programmiert sind

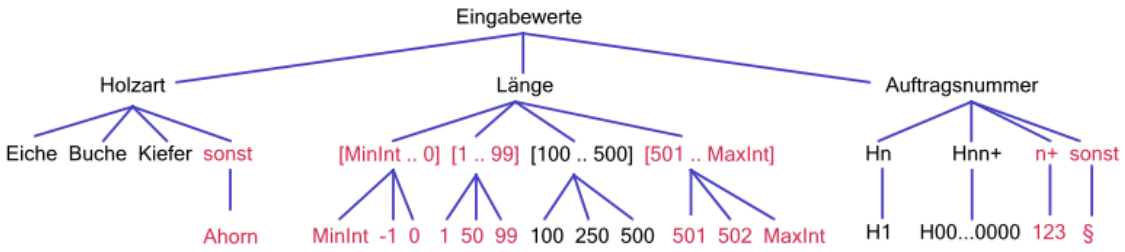
Auswahl von Testeingaben für countVowels (mit Zeichenketten!): s = sentence

- Klasse 1: s endet nicht mit einem Punkt (oh je, das geht schief)
- Klasse 2: s endet mit einem Punkt und enthält keine Vokale
  - s besteht nur aus einem Punkt
  - s besteht aus einem Konsonanten gefolgt von einem Punkt
  - s enthält sonstige Sonderzeichen
- Klasse 3: s endet mit einem Punkt und enthält einen Vokal:
  - 3a: s enthält ein a, e, i, o, u
  - 3b: s enthält ein A, E, I, O, U (oh je, dieser Fall wurde auch vergessen)
- Klasse 4: s enthält mehrere Vokale:
  - 4a: mehrere gleiche Vokale
  - 4b: mehrere verschiedene Vokale
- Klasse 5: Eingabe ist sehr lang und enthält ganz viele Vokale

Weitere Regeln für die Bildung von Äquivalenzklassen bei Intervallen:

- aus Äquivalenzklassen, die (geschlossene) Intervalle sind, werden jeweils die beiden Grenzwerte und ein weiterer Wert (z.B. aus der Mitte des Intervalls) ausgewählt (ohne platzraubenden Zwischenschritt der Zerlegung in drei Teilintervalle)
- einelementige Äquivalenzklasse und Wert aus dieser Äquivalenzklasse werden nicht unterschieden, also statt [v] schreiben wir gleich v
- reguläre Ausdrücke werden zur Definition v. String-Äquivalenzklassen eingesetzt

Abbildung 19: Grafische Darstellung von Äquivalenzklassen als Klassifikationsbaum

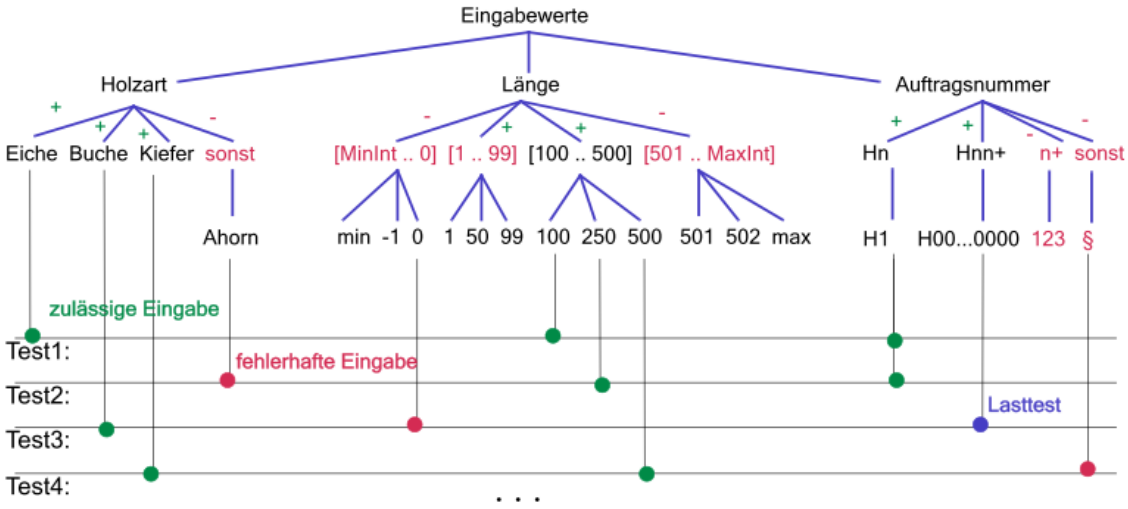


Grafische Darstellung von Äquivalenzklassen als Klassifikationsbaum:

- zunächst wird "Eingabewerte" in alle Eingabeparameter des Programms zerlegt
- zusammengesetzte Eingabeparameter werden weiter zerlegt
- schließlich wird der Wertebereich eines atomaren Eingabeparameters betrachtet
- der Wertebereich wird in Äquivalenzklassen „ähnlicher" Werte zerlegt
- dabei werden auch nicht erlaubte Eingabewerte (Fehlerklassen) betrachtet
- ebenso werden "extreme" erlaubte Werte (Lasttestklassen) betrachtet
- aus jedem Wertebereich werden Repräsentanten für den Test ausgewählt

Unvollständige/fehlerhafte Auswahl von Eingabewertekombinationen: Problem: trotz Bildung von Äquivalenzklassen bleiben (zu) viele mögliche Eingabewertekombinationen (im Beispiel sind 4 * 12 * 4 verschiedene Testläufe möglich).

Abbildung 20: Unvollständige/fehlerhafte Auswahl von Eingabewertekombinationen



Heuristiken für die Reduktion möglicher Eingabewertekombinationen:

- aus jeder **Äquivalenzklasse** wird mindestens einmal ein Wert ausgewählt; es gibt also jeweils mindestens einen Testfall, in dem ein Eingabewert der Äquivalenzklasse verwendet wird (bei Grenzwertanalyse werden drei Werte ausgewählt)
- bei **abhängigen Eingabeparametern** müssen Testfälle für **alle Kombinationen** ihrer jeweiligen "normalen" Äquivalenzklassen aufgestellt werden; Parameter sind abhängig, wenn sie gemeinsam das Verhalten des Programms steuern (und deshalb nicht unabhängig voneinander betrachtet werden können)
- der ausgewählte **Wert** einer **Fehleräquivalenzklasse** wird in genau einem Testfall verwendet (Fehleräquivalenzklassen sind solche Äquivalenzklassen, die unzulässige Eingabewerte zusammenfassen)
- der ausgewählte **Wert** einer **Lasttestklasse** wird ebenfalls genau einmal in einem Testfall verwendet (Lasttestklassen sind solche Äquivalenzklassen, die besonders "große/lange/..." zulässige/gültige Eingabewerte zusammenfassen)
- hat man mehrere Eingabeparameter, wird **höchstens einer** mit einem Wert aus einer Fehler- oder Lasttestklasse belegt (verhindert Verdeckung von Fehlern)

"Normierter" Aufbau von Klassifikationsbäumen für Übung/Klausur:

1. Ebene: alle Parameter(-namen) der betrachteten Funktion
2. Ebene: die Typen bzw. Wertebereiche der Parameter

3. Ebene: die Zerlegung der Wertebereiche in
   - Äquivalenzklassen für erlaubte Werte (mit "+" markiert)
   - Fehleräquivalenzklassen (mit "-" markiert)
4. Ebene: weitere Zerlegung der Wertebereiche mit Grenzwertanalyseverfahren
5. Ebene: konkrete Repräsentanten (Werte) der jeweiligen Äquivalenzklassen

Paarweiser Testansatz (Pairwise Testing): Die Praxis zeigt, dass ca. 80% aller von bestimmten Parameterwertkombinationen ausgelösten Softwarefehler bereits durch Wahl bestimmter Paarkombinationen beobachtet werden können. Also werden beim "paarweisen" Testen einer Funktion mit n Parametern nicht alle möglichen Kombinationen überprüft, sondern nur alle paarweisen Kombinationen.

Bewertung der funktionalen Äquivalenzklassenbildung:

- Güte des Verfahrens hängt stark von **Güte der Spezifikation** ab (Aussagen über erwartete/zulässige Eingabewerte und Ergebnisse)
- Verwaltung von Äquivalenzklassen und Auswahl von Testfällen kann durch **Werkzeugunterstützung** vereinfacht werden
- **ausführbare Modelle** (z.B. in UML erstellt) können bei der Auswahl von Äquivalenzklassen, Testfällen helfen (und als Orakel verwendet werden)
- Test von Benutzeroberflächen, zeitlichen Beschränkungen, Speicherplatzbeschränkungen etc. (**nichtfunktionale Anforderungen**) wird kaum unterstützt
- mindestens **einmalige Ausführung** jeder Programmzeile wird nicht garantiert
- Test **zustandsbasierter Software** (Ausgabeverhalten hängt nicht nur von Eingabe, sondern auch von internem Zustand ab) geht so nicht
  - später Spezifikation und Testplanung mit Hilfe von Automaten/Statecharts
  - Grenzfall zwischen funktionalem und strukturorientiertem Softwaretest

Weitere Black-Box-Testverfahren

- **Erfahrungsbasierte Testverfahren:**
  - **Intuitives Testen (Error Guessing)**: ein Testansatz, bei dem Testfälle auf Basis des Wissens der Tester über frühere Fehler oder allgemeines Wissen über Fehlerwirkungen (irgendwie) abgeleitet werden
  - **Exploratives Testen**: ein Testansatz bei dem die Tester, basierend auf ihrem Wissen, der Erkundung des Testobjekts und dem Ergebnis früherer Tests, dynamisch neue Tests entwickeln

  - **Checklistenbasiertes Testen**: ein Testansatz bei dem erfahrene Tester eine Liste von Kontrollpunkten (Checkliste) bzw. Regeln oder Kriterien (für das Testobjekt) zur Steuerung des Testprozesses nutzen

- **Zufallstest**: wählt aus Menge der möglichen Werte eines Eingabedatums zufällig Repräsentanten aus (ggf. gemäß bekannter statistischer Verteilung)
  - zur Ergänzung gut geeignet, generiert oft "unerwartete" Testdaten
- **Smoke-Test:** es wird nur Robustheit des Testobjekts getestet, berechnete Ausgabewerte spielen keine Rolle (auf Tastatur hämmern, ...)
- **Syntax-Test**: ist für Eingabewerte der erlaubte syntaktische Aufbau bekannt (als Grammatik angegeben) kann man daraus systematisch Testfälle generieren; Beispiele:
  - syntaktisch korrekte Email-Adressen
  - zulässige Dateinamen, Verzeichnispfade
  - Aufbau von Zahlen (Integers, Floats)
  - ...
- **Zustandsbezogener Test**
- **Ursache-Wirkungs-Graph-Analyse / Entscheidungstabellenbasiertes Testen**
- **Anwendungsfallbasiertes Testen**

Ursache-Wirkungs-Graph-Analyse-Verfahren aus [SL19] Es handelt sich eigentlich um eine Kombination von zwei Verfahren. Die Basis bilden sogenannte **"Entscheidungstabellen"**, die Bedingungen an Eingaben und ausgelöste Aktionen eines Systems miteinander verknüpfen. Für die systematische Erstellung einer Entscheidungstabelle wird zunächst ein **"Ursache-Wirkungs-Graph"** erstellt.

Ein Ursache-Wirkungs-Graph verknüpft Eingaben = **Ursachen** / Bedingungen mit daraus resultierenden Ausgaben = **Wirkungen** / Aktionen (durch grafische Darstellung aussagenlogischer Ausdrücke). Die weitere Vorgehensweise ist wie folgt:

1. Eine Wirkung wird ausgewählt.
2. Zu der Wirkung werden alle Kombinationen von Ursachen gesucht, die diese Wirkung hervorrufen.
3. Für jede gefundene Ursachenkombination wird eine Spalte der Entscheidungstabelle erzeugt.
4. Die Spalten der Entscheidungstabelle entsprechen Testfällen.
5. Die Zeilen der Entscheidungstabelle entsprechen allen Ursachen und Wirkungen.

Anwendungsfallbasiertes Testen aus [SL19]: Ausgangspunkt für diese Testmethodik ist die Beschreibung von sogenannten **Anwendungsfällen** (Nutzungsszenarien, Geschäftsvorfällen) eines Systems im Zuge der Anforderungsanalyse. Dabei kommt in der Regel die "Unified Modeling Language" (UML) mit ihren Anwendungsfalldiagrammen zum Einsatz.

- Anwendungsfalldiagramme erlauben die Beschreibung von Nutzungsszenarien (und damit von Akzeptanztestfällen) auf sehr hohem Abstraktionsniveau sowie die Unterscheidung zwischen normalen Abläufen und Ausnahmen.
- Werden einzelne Anwendungsfälle informell / in natürlicher Sprache beschrieben, so werden die dazugehörigen Testfälle **"manuell"** erstellt.
- Werden für die Beschreibung einzelner Anwendungsfälle formalere Notationen wie Sequenz- oder Aktivitätsdiagramme der UML eingesetzt, so können Testwerkzeuge daraus **automatisch** Code für die Testfallausführung und -bewertung generieren.

Bewertung der Ursache-Wirkungs-Graph- und Anwendungsfall-Testens:

- Beide Methoden lassen sich sehr früh im Software-Lebenszyklus einsetzen und eignen sich insbesondere für die Erstellung von Akzeptanztestfällen.
- Die Ursache-Wirkungsgraph-Methode erlaubt die bei der Äquivalenzklassenbildung fehlende Verknüpfung von Eingaben und Ausgaben (Ursachen und Wirkungen).
- Das Anwendungsfallbasierte Testen erlaubt hingegen nicht nur die Beschreibung einzelner Testvektoren (Eingabewertkombinationen), sondern auch die Spezifikation ganzer Interaktionssequenzen zwischen Umgebung und zu testendem System.
- Insbesondere das Anwendungsfallbasierte Testen unterstützt aber nicht die systematische Identifikation fehlender Testfälle (für bestimmte Eingabetestvektoren).

Fazit: Alle vorgestellten "Black-Box"-Testmethoden (Funktionsorientierte Testverfahren) besitzen ihre Stärken und Schwächen und ergänzen einander!!!

4.4 Kontrollflussbasierte Testverfahren (Whitebox)

Grundideen des kontrollflussbasierten Testens:

- mit im Grunde zunächst beliebigen Verfahren werden **Testfälle festgelegt**
- diese Testfälle werden alle **ausgeführt** und dabei wird notiert, welche Teile des Programms durchlaufen wurden
- es gibt (meist) ein **Test-Orakel**, dass für jeden ausgeführten Testfall ermittelt, ob die berechnete Ausgabe (Verhalten des Programms) korrekt ist

- schließlich wird festgelegt, ob die vorhandenen Testfälle den Kontrollfluss des Programms hinreichend **überdecken**
- ggf. werden solange **neue Testfälle aufgestellt**, bis hinreichende Überdeckung des Quelltextes (Kontrollflusses) erreicht wurde
- ggf. werden **alte Testfälle gestrichen**, die dieselben Teile des Quelltextes (Kontrollflusses) überdecken

Kontrollflusstest - Anweisungsüberdeckung (C0-Test): Jeder Knoten des Kontrollflussgraphen muss mindestens einmal ausgeführt werden.

- Minimalkriterium, da nicht mal alle Kanten des Kontrollflussgraphen traversiert werden
- viele Fehler bleiben unentdeckt

Kontrollflusstest - Zweigüberdeckung (C1-Test): Jede Kanten des Kontrollflussgraphen muss mindestens einmal ausgeführt werden.

- realistisches Minimalkriterium
- umfasst Anweisungsüberdeckung
- Fehler bei Wiederholung oder anderer Kombination von Zweigen bleiben unentdeckt

Kontrollflusstest - Entscheidungs-/Bedingungsüberdeckung: Jede Teilbedingung einer Kontrollflussbedingung (z.B. von if- oder while-Anweisung) muss einmal den Wert true und einmal den Wert false annehmen.

atomare Bedingungsüberdeckung/-test:

- keine Anforderung an Gesamtbedingung
- umfasst nicht mal Anweisungsüberdeckung

minimale Mehrfachbedingungsüberdeckung/-test:

- jede Teil- und Gesamtbedingung ist einmal true und einmal false
- Orientierung an syntaktischer Struktur von Kontrollflussbedingungen
- Bedingung umfasst Zweigüberdeckung
- trotzdem werden viele Bedingungsfehler nicht entdeckt

Modifizierter Bedingungsüberdeckungstest (MCDC): Der **modifizierte Bedingungsüberdeckungstest** (Definierter Bedingungstest, Modified Condition Decision Coverage) benötigt wie die bisherigen Überdeckungskriterien eine linear mit der Anzahl der atomaren Teilbedingungen steigende Anzahl von Testfällen. Es werden aber i.A. "bessere" Testfälle als bei der minimalen Mehrfachbedingungsüberdeckung gewählt. Die Bedingungen sind:

- jeder atomaren Teilbedingung lassen sich zwei Testfälle zuordnen (verschiedene Teilbedingungen dürfen aber die selben Testfälle nutzen)
- die Werte aller Teilbedingungen, die für das Gesamtergebnis der Bedingung irrelevant sind und deshalb ggf. wegen "short circuit"-Evaluation des Compilers nicht ausgewertet werden, werden als irrelevant gekennzeichnet (mit Zeichen "-")
- die beiden Testfälle zu einer atomaren Teilbedingung setzen diese einmal auf true und einmal auf false und unterscheiden sich nur in der gerade betrachteten atomaren Teilbedingung (irrelevant kann mit true u. false gleichgesetzt werden)
- die beiden Testfälle zu einer atomaren Teilbedingung setzen die Gesamtbedingung einmal auf true und einmal auf false

Kontrollflusstest - Pfadüberdeckung (C-unendlich-Test): Jeder mögliche Pfad des Kontrollflussgraphen muss einmal durchlaufen werden.

- rein theoretisches Kriterium, sobald Programm Schleifen enthält (unendliche viele verschiedene Pfade = Programmdurchläufe möglich)
- dient als Vergleichsmaßstab für andere Testverfahren
- findet trotzdem nicht alle Fehler (z.B. Berechnungsfehler), da kein erschöpfender Test aller möglichen Eingabewerte
- davon abgeleitetete in der Praxis durchführbare Verfahren:
  - **boundary test**: alle Pfade auf denen Schleifen maximal einmal durchlaufen werden (ohne besondere praktische Bedeutung)
  - **boundary interior test**: alle Pfade auf denen Schleifen maximal zweimal (in direkter Folge) durchlaufen werden (Achtung: Anzahl Pfade explodiert bei geschachtelten Schleifen und vielen bedingten Anweisungen)
  - **modifizierter boundary interior test**: bei geschachtelten Schleifen wird beim Durchlauf einer äußeren Schleife die Anzahl der inneren Schleifendurchläufe nicht unterschieden

Bewertung der Kontrollflusstests:

- Anweisungsüberdeckung wird durch RTCA DO-178B-Standard für Software-Anwendungen in der Luftfahrt der **Kritikalitätsstufe C** gefordert (Software, deren Ausfall zu einer bedeutenden, aber nicht kritischen Fehlfunktion führen kann)
- Zweigüberdeckung wird durch RTCA DO-178B-Standard für Software-Anwendungen in der Luftfahrt der **Kritikalitätsstufe B** gefordert (Software, deren Ausfall zu schwerer aber noch nicht katastrophaler Systemfehlfunktion führen kann)
- modifizierte Bedingungsüberdeckung wird durch RTCA DO-178B-Standard für Software-Anwendungen in der Luftfahrt der **Kritikalitätsstufe A** gefordert (Software, deren Ausfall zu katastrophaler Systemfehlfunktion führen kann)
- Zweigüberdeckung sollte für uns Mindestanforderung beim Testen darstellen (Kontrollflussfehler/-Bedingungsfehler werden relativ gut gefunden, Datenflussfehler natürlich weniger gut)
- Einfache Variante von "modified boundary interior test" zur Ergänzung: für jede Schleife gibt es Testfälle/Pfade, die sie gar nicht, genau einmal und (mindestens) zweimal ausführen (die Randbedingung "alle Pfade" wird komplett aufgegeben)

4.5 Datenflussbasierte Testverfahren

Ausgangspunkt ist der Datenflussgraph einer Komponente bzw. der mit Datenflussattributen annotierte Kontrollflussgraph. Bei der Auswahl von Testfällen wird darauf geachtet, dass:

- für jede Zuweisung eines Wertes an eine Variable **mindestens eine** (berechnende, prädikative) Benutzung dieses Wertes getestet wird
- oder für jede Zuweisung eines Wertes an eine Variable **alle** (berechnenden, prädikativen) Benutzungen dieses Wertes getestet werden

Die datenflussbasierten Testverfahren haben folgende Vor- und Nachteile:

- Vorteile:
  - einige Verfahren enthalten die Zweigüberdeckung und finden sowohl Datenflussfehler **als auch** Kontrollflussfehler
  - besser geeignet für objektorientierte Programme mit oft einfachem Kontrollfluss aber komplexem Datenfluss
- Nachteile:
  - es gibt kaum Werkzeuge, die datenflussbasierte Testverfahren unterstützen

Kriterien für den Datenflusstest

- **all-defs-Kriterium**: für jede Definitionsstelle d(x) einer Variablen muss **ein** definitionsfreier Pfad zu **einer** Benutzung r(x) existieren (und getestet werden)
  - Kriterium kann statisch überprüft werden (entdeckt sinnlose Zuweisungen)
  - umfasst weder Zweig- noch Anweisungsüberdeckung
  - bei countVowels reichen Testbeispiele '.' und 'a.'
  - Verfahren findet einige Berechnungsfehler und kaum Kontrollflussfehler
- **all-p-uses-Kriterium**: für jede Definitionsstelle d(x) wird jeweils ein definitionsfreier Pfad zu **allen** (erreichbaren) prädikativen Benutzungen p(x) getestet
  - entdeckt vor allem Kontrollflussfehler
  - Berechnungsfehler bleiben oft unentdeckt
  - **Anmerkung**: manchmal wird auch Test **aller** definitionsfreien Pfade von d(x) zu allen p(x) gefordert, die Schleifen nicht mehrfach durchlaufen müssen (dann ist Zweigüberdeckung enthalten)
- **all-c-uses-Kriterium**: für jede Definitionsstelle d(x) wird jeweils **ein** definitionsfreier Pfad zu **allen** (erreichbaren) berechnenden Benutzungen c(x) getestet
  - entdeckt vor allem Berechnungsfehler
  - Kontrollflussfehler bleiben oft unentdeckt
  - lässt sich wegen Bedingungen oft nicht erzwingen
- **all-p-uses-some-c-uses-Kriterium**: für jede Definitionsstelle d(x) wird jeweils **ein** definitionsfreier Pfad zu **allen** (erreichbaren) prädikativen Benutzungen p(x) getestet; gibt es keine prädikate Benutzung p(x), so wird wenigstens ein Pfad zu einem berechnenden Zugriff c(x) betrachtet
  - entdeckt Kontrollfluss- und auch Berechnungsfehler
  - umfasst all-def- und all-p-uses-Kriterium
- **all-c-uses-some-p-uses-Kriterium**: ... (wird kaum benutzt)
- **all-uses-Kriterium**: all-p-uses- + all-c-uses-Kriterium (wird kaum benutzt)

Zusammenfassung des überdeckungsbasierten Testens:

- man wählt ein oder mehrere Überdeckungskriterien aus, die der "Kritikalität" der zu entwickelnden Software gerecht werden
- Kombination von einem kontrollflussbasierten und einem datenflussbasierten Überdeckungskriterium sinnvoll

- man wählt nach beliebiger Methodik initiale Menge von Testfällen aus
- dann wird (durch Code-Überdeckungsanalyse-Werkzeug) überprüft, zu wieviel Prozent die gewählten Kriterien erfüllt sind
- es werden solange Testfälle hinzugefügt, bis eine vorab festgelegte Prozentzahl für alle gewählten Überdeckungskriterien erfüllt ist (90% oder ...)
- Achtung: 100% lässt sich in vielen Fällen nicht erreichen (wegen Anomalien

4.6 Testen objektorientierter Programme (zustandsbez. Testen)

Prinzipiell lassen sich in objektorientierten Sprachen geschriebene Programme wie alle anderen Programme testen. Allerdings gibt es einige Besonderheiten, die das Testen sowohl erschweren als auch erleichtern (können):

- Positiv
  - die Datenkapselung konzentriert Zugriffsoperationen auf Daten an einer Stelle und erleichtert damit das Testen (Einbau von Konsistenzprüfungen)
  - die Vererbung mit Wiederverwendung bereits getesteten Codes reduziert die Menge an neu geschriebenem zu testenden Code
- Negativ
  - die Datenkapselung erschwert das Schreiben von Testtreibern, die auf interne Zustände von Objekten zugreifen müssen
  - die Vererbung ist eine der Hauptfehlerquellen (falsche Interaktion mit geerbten Code bzw. falsche Redefinition von geerbten Methoden)
  - dynamisches Binden erschwert die Definition sinnvoller Überdeckungsmetriken ungemein (beim White-Box-Test)
  - Verhalten von Objektmethoden ist (fast) immer zustandsabhängig

Prinzipien beim Tests objektorientierter Programme:

- beim **"Black-Box"-Test** wie bisher vorgehen (Objekte als Eingabeparameter werden gemäß interner Zustände verschiedenen Äquivalenzklassen zugeordnet)
- beim **"White-Box"-Test** werden Kontrollflussgraphen erweitert, um so Effekte des dynamischen Bindens mit zu berücksichtigen (wird hier nicht weiter verfolgt)
- **Zustandsautomaten** werden zusätzlich zu Kontrollflussgraphen zur Testplanung herangezogen (dieses Verfahren wird meist dem "Black-Box"-Test zugeordnet)

- Einbau von **Konsistenzüberprüfungen** (Plausibilitätsüberprüfungen, assert in Java), in Methoden (zu Beginn und nach Abarbeitung von Methodencode)
- **defensive Programmierung**: Code fängt alle erkennbaren Inkonsistenzen ab
- **geerbter Code** wird wie neu geschriebener Code behandelt und immer vollständig im Kontext der erbenden Klasse neu getestet (Variation des Regressionstests)
- inkrementelles Testen mit **Regressionstests** unter Verwendung von Frameworks wie JUnit

Prinzipien beim Test einzelner Klassen

1. **Nicht-modale Klassen**: Methoden der Klasse können immer (zu beliebigen Zeitpunkten) aufgerufen werden; interner Zustand der Objekte spielt dabei keine Rolle
   - Methoden können isoliert für sich getestet werden; bei der Auswahl der Testfälle muss Objektzustand nicht mit berücksichtigt werden
   - Beispiel: Gerätesteuerung mit setStatus-Methode u. getStatus-Methode, die Zustand liefert (Beispiel ist Grenzfall; besser Klasse ohne Attribute)
2. **Uni-modale Klasse**: Methoden können nur - unabhängig vom internen Zustand der Objekte - in einer bestimmten Reihenfolge aufgerufen werden (warum?)
   - Testfälle müssen alle zulässigen und nicht zulässigen Reihenfolgen von Methodenaufrufen durchprobieren; interne Objektzustände nicht relevant (Automaten mit zulässigen Methodenaufrufen als Transitionen werden zur Testplanung herangezogen)
   - Beispiel: Gerätesteuerung mit init-, setStatus- und getStatus-Methoden; init-Methode muss zuerst aufgerufen werden
3. **Quasi-modale Klasse**: Zustand der Objekte bestimmt Zulässigkeit von Methodenaufrufen (und nur dieser)
   - Methoden werden isoliert getestet, aber für alle zu unterscheidenden Äquivalenzklassen des internen Objektzustandes (Automaten mit Objektzuständen werden zur Testplanung herangezogen)
   - Beispiel: Gerätesteuerung mit setStatus-Methode und getStatus-Methode, die nur 100.000 Status-Wechsel zulässt und dann den Dienst verweigert (nach Wartung verlangt)
4. Modale Klasse: Methoden können nur in fest vorgegebenen Reihenfolgen aufgerufen werden; zusätzlich hat Objektzustand Einfluss auf Zulässigkeit von Aufrufen
   - Kombination der Testmethoden für uni-modale und quasi-modale Klassen notwendig; Testplanung mit Automaten
   - Beispiel: Gerätesteuerung mit init-, setStatus- und getStatus-Methode mit zusätzlicher Beschränkung auf 100.000 Status-Wechsel

Abbildung 21: Tabellarische Übersicht über verschiedene Arten von Klassen beim Testen:

	Zustand bestimmt nicht Methodenaufrufbarkeit	**Zustand bestimmt Methodenaufrufbarkeit**
Aufrufreihenfolge der Methoden flexibel	**nicht modal**: Methoden isoliert testen	**quasi-modal**: Methoden isoliert testen für alle Zustandsäquivalenz-klassen
Aufrufreihenfolge der Methoden fest	**uni-modal**: alle zulässigen und verbote-nen Reihenfolgen testen	**modal**: alle zulässigen/verbotenen Reihenfolgen testen für alle Zustandsäquivalenzklassen

Vorüberlegungen zur Realisierung von Invarianten,...:

- die teuere Überprüfung von Invarianten, ... muss durch "Compile"-Flag abschaltbar sein (entweder systemweit oder je Subsystem)
- die Überprüfungen dürfen das Verhalten des ausführbaren Programms (abgesehen von Laufzeit und Speicherplatzverbrauch) nicht veränder
- bei abgeschalteten Überprüfungen sollte soweit möglich der Code für diese Überprüfungen nicht Bestandteil des ausführbaren Programms sein
- die Reaktion auf fehlgeschlagene Überprüfungen muss an einer Stelle (veränderbar) festgelegt werden (Programmabbruch, Ausnahmeerweckung, ...)
- die Überprüfungen sollten möglichst lesbar niedergeschrieben werden
- Invarianten werden auch vor der Ausführung einer Methode und nach der Ausführung von "Observer"-Methoden überprüft (um illegale Objektzugriffe entdecken zu können)

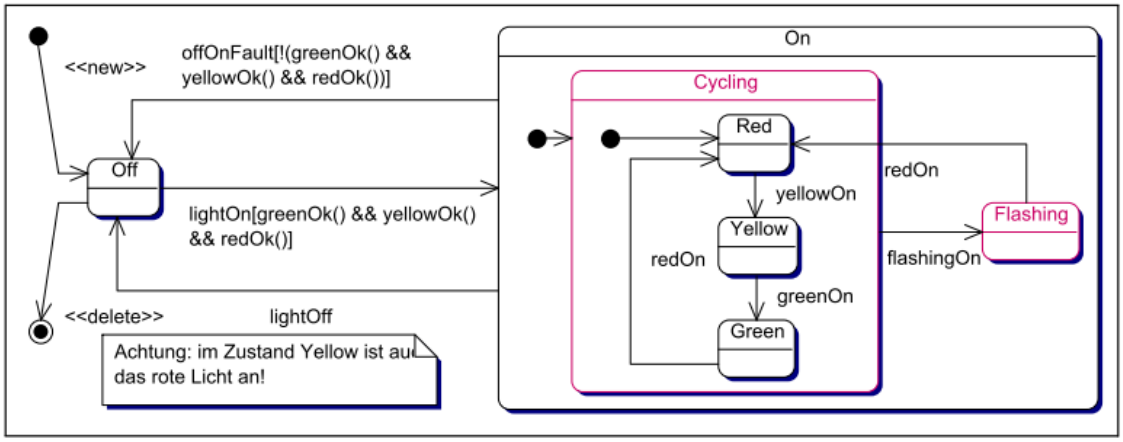
Zusicherungen (Assertions) in Java:

- Java besitzt ab Version 1.4 „assert"-Statement, das für den Einbau von Überprüfungen (Vor-/Nachbedingungen, Invarianten) genutzt wird
- die Überprüfung von "assert"-Statements kann durch Runtime-Flags generell oder klassenweise an- bzw. abgeschaltet werden ("-enableassertions" = "-ea" und ".-disableassertions" = "-da")
- die Überprüfungen sollten das Verhalten des ausführbaren Programms (abgesehen von Laufzeit und Speicherplatzverbrauch) nicht verändern (der Programmierer muss das sicherstellen)

- soll der Code für "assert"-Statements nicht Bestandteil des ausführbaren Programms sein, so muss der Compiler davon "überzeugt" werden, dass der Code wegoptimiert werden kann
- "assert"-Verletzungen können als "AssertionError"-Ausnahmen abgefangen werden (und damit im Code festgelegte Reaktionen auslösen)

Einsatz von Automaten (Statecharts) zur Testplanung: Oft lassen sich die Vorbedingungen für den Aufruf von Methoden besser durch ein Statechart (hierarchischer Automat, siehe Software Eng. - Einführung) darstellen. Ein solches Statechart kann dann - ähnlich wie ein Kontrollflussgraph - zur Planung von Testfällen zur Berechnung von Testüberdeckungsmetriken herangezogen werden.

Abbildung 22: Einsatz von Automaten (Statecharts) zur Testplanung



Präzisierung der vier verschiedenen Arten von Klassen:

- Nicht modale Klasse:
  - keine Methode der Klasse hat eine Vorbedingung über Attributen
  - der Zustandsautomat der Klasse besteht aus einem Zustand
- Quasimodale Klasse:
  - mindestens eine Methode hat eine Vorbedingung über Attributen
  - der Zustandsautomat der Klasse besteht aus einem Zustand
- Unimodale Klasse:
  - keine Methode der Klasse hat eine Vorbedingung über Attributen
  - der Zustandsautomat der Klasse besteht aus mehreren Zuständen

- Modale Klasse:
  - mindestens eine Methode hat eine Vorbedingung über Attributen
  - der Zustandsautomat der Klasse besteht aus mehreren Zuständen

Definition von Testüberdeckungsmetriken für Statecharts:

- Tests müssen garantieren, dass alle Zustände mindestens einmal erreicht werden
- Tests müssen garantieren, dass jede Transition mindestens einmal ausgeführt wird
- Tests müssen garantieren, dass jede Transition mit allen sich wesentlich unterscheidenden Belegungen ihrer Bedingung ausgeführt wird
- Tests müssen alle möglichen Pfade durch Statechart (bis zu einer vorgegebenen Länge oder vorgegebenen Anzahl von Zyklen) ausführen
- zusätzlich zum Test aller explizit aufgeführten Transitionen werden für jeden Zustand alle sonst möglichen Ereignisse (Methodenaufrufe) ausgeführt

Testplanung mit Transitionsbaum (Übergangsbaum) aus [Bi00]:

1. das gegebene Statechart wird in einen flachen Automaten übersetzt
2. Transitionen mit komplexen Boole'schen Bedingungen werden in mehrere Transitionen mit Konjunktion atomarer Bedingungen übersetzt (Transition mit [(a1 && a2) || (b1 && b2)] wird ersetzt durch Transition mit [a1 && a2] und Transition mit [b1 && b2]
3. ein Baum wird erzeugt, der
   - initialen Zustand als Wurzelknoten (ersten, obersten Knoten) besitzt
   - Zustandsknoten im Baum werden expandiert, indem alle Transitionen zu anderen Zuständen (und sich selbst) als Kindknoten hinzugefügt werden
   - jeder Zustand wird nur einmal als Knoten im Transitionsbaum expandiert
4. jeder Pfad in dem Baum (von Wurzel zu einem Blatt) entspricht einer Testsequenz
5. zusätzlich werden in jedem Zustand alle Ereignisse ausgelöst, die nicht im Transitionsbaum aufgeführt sind (spezifikationsverletzende Transitionen)

Behandlung ereignisloser Transitionen: Automatenmodelle für die Verhaltensbeschreibung wie UML-Statecharts erlauben oft auch die Definition ereignisloser Transitionen. Diese werden wie folgt beim Testen behandelt:

- Transition **ohne Ereignis mit Bedingung** [b]: sobald die Bedingung erfüllt ist, schaltet die Transition; beim Aufstellen von Testsequenzen sind zwei Fälle zu unterscheiden (zwei Unterbäume im Transitionsbaum):
  - Bedingung b ist bereits erfüllt, wenn Startzustand der Transition betreten wird (Transition wird mit der Bedingung "[b == true]" markiert und schaltet sofort bei der Testausführung)
  - Bedingung ist nicht erfüllt, wenn Startzustand betreten wird; wird später aber erfüllt (Transition wird mit Ereignis "b -> true" markiert und schaltet sobald die Bedingung b erfüllt ist)
- Transition **ohne Ereignis und ohne Bedingung**: die Transition schaltet, sobald der Startzustand betreten wird: der Startzustand erhält im Transitionsbaum eine ausgehende Kante/Transition ohne Markierung

Ausführung der Testsequenzen:

- **Testvorbereitung**: Objekt muss in initialen Zustand (zurück-)versetzt werden
- **Testausführung** einer Sequenz von Methodenaufrufen (Testvektor): es wird unterschieden
  - **white-box-Sicht**: es gibt Zugriffsoperationen für Abfrage des internen Zustands; man kann also am Ende einer Testsequenz abfragen, ob richtiger Zustand erreicht wurde (interne Zustände der Implementierung müssen mit "extern" definierten Zuständen korrespondieren)
  - **black-box-Sicht**: Aussenverhalten muss überprüft werden
- **Testbeendigung**: ggf. wird Sequenz von Methodenaufrufen so vervollständigt, dass am Ende der Ausführung Objekt sich in einem "terminalen" Zustand befindet
- **Kombination** von Testsequenzen: um Aufwand für Initialisierung zu reduzieren, werden möglichst lange Testsequenzen generiert bzw. kombiniert

Abschließende Checkliste für Klassentest nach [Bi00]:

- jede Methode (auch die geerbten) wird mindestens einmal ausgeführt
- alle Methodenparameter und alle nach aussen sichtbaren Attribute werden mit geeigneter Äquivalenzklassenbildung durchgetestet
- alle auslösbaren (ausgehenden) Ausnahmen werden mindestens einmal ausgelöst
- alle von gerufenen Methoden auslösbaren (eingehenden) Ausnahmen werden mindestens einmal behandelt (oder durchgereicht)

- alle identifizierten Objektzustände (auch hier Äquivalenzklassenbildung) werden beim Testen erreicht
- jede zustandsabhängige Methode wird in jedem Zustand ausgeführt (auch in den Zuständen, in denen ihr Aufruf nicht zulässig ist)
- alle möglichen Zustandsübergänge (mit allen Kombinationen von Bedingungen an den Übergängen) werden aktiviert
- zusätzlich werden die üblichen Performanz-, Last-, ... -Tests durchgeführt

4.7 Mutationsbasierte Testverfahren

Hypothesen

- Programmierer erstellen (oft) annährend korrekte Programme (Competent Programmer Hypothesis)
- Komplexe Fehler bedingen häufig die Existenz von simplen Fehlern (Coupling Effect)

Schlussfolgerungen

- Typische Programmierfehler lassen sich oft auf kleine syntaktische Änderungen (Mutationen) eines korrekten Programmes zurückführen
- Solche Programmmutationen lassen sich automatisiert durchführen
- Testfälle sind "gut", die bei gleichen Eingaben zu unterschiedlichen Ausgaben (unterschiedlichem Verhalten) eines Programms und eines seiner Mutanten führen

Anforderungen an (stark-)mutationserkennende Testfälle Der gesuchte Testfall soll bei seiner Ausführung

- die fehlerhafte Stelle erreichen (**Reachability Condition**),
- der Programmzustand soll infiziert werden (**Infection Condition**), also während der Ausführung zu veränderten Variablenbelegungen führen
- und der infizierte Programmzustand soll in das Ergebnis propagiert werden (**Propagation Condition**), dieses also verändern.

Achtung:

- Ist die "Propagation Condition" erfüllt, dann ist auch die "Infection Condition" zwangsläufig erfüllt.
- Ist die "Infection Condition" erfüllt, dann ist auch die "Reachability Condition" zwangsläufig erfüllt.
- Bislang für die Auswahl von Testfällen benutzte Programmüberdeckungs kriterien konzentrieren sich auf die "Reachability Conditions"

Mutationsbasierte Testsuite-Bewertung:

- gegeben ist ein potentiell fehlerhaftes Programm p und eine **Test-Suite** TS, die aus einer Menge von Testfällen $tc1, tc2, ...$ besteht
- zunächst wird eine Menge von **Mutanten** $M = m1, m2, ...$ erzeugt, die sich in der Regel jeweils nur an einer Programmstelle vom Originalprogramm p unterscheiden (Programme mi mit mehreren Unterschieden gegenüber p werden **Mutanten höherer Ordnung** genannt)
- dann werden alle Testfälle der Test-Suite TS auf dem Programm p und der Menge seiner Mutanten M ausgeführt
- die Test-Suite bzw. ein Testfall tötet einen Mutanten m_i , falls die Ausführung von p und m_i unterschiedliche Ausgaben erzeugen
- **Effektivität** einer Test-Suite: Anzahl der getöteten Mutanten
- **Effizienz** einer Test-Suite: Anzahl der benötigten Testfälle
- Erhöhung der Effizienz einer Test-Suite ohne Reduktion ihrer Effektivität: Testfälle, die keinen Mutanten töten, werden eliminiert; gleiches gilt für Teilmengen von Testfällen, die alle den selben Mutanten töten)

Mutationsbasierte Testsuite-Generierung: Die werkzeuggestützte Erzeugung von Testfällen, die bestimmte Mutanten töten, ist ein "schwieriges" Problem (i.A. nicht berechenbar).

- naiver Ansatz: zufallsgesteuerte Erzeugung von Testfällen
- verifikationsbasierter Ansatz: das Problem der Erzeugung eines (fehlenden) Testfalles, der einen bestimmten Mutanten m eines Programms p tötet, wird in ein Verifikationsproblem übersetzt:
  - erzeugt wird ein neues Programm, das p und m hintereinander ausführt und die Ausgaben der Ausführung von p und m in verschiedenen Variablen speichert
  - verifiziert wird dann die Eigenschaft des neuen Programms, für alle möglichen Eingaben bei der Ausführung von p und m immer die gleichen Ausgaben zu produzieren
  - jedes von einem Verifikationswerkzeug produzierte Gegenbeispiel zu dieser Eigenschaft legt einen Testfall fest, der den Mutanten m tötet

Bewertung des Ansatzes [ABL05]:

- Ähnlichkeit zwischen mechanisch erzeugten Mutanten und realen Fehlern ist größer als zwischen manuell eingebauten Fehlern und realen Fehlern
- bei manuell eingebauten Fehlern wird die Erkennungsrate einer Test-Suite unterschätzt (manuell eingebaute Fehler sind oft schwerer zu detektieren als mechanisch erzeugte Mutationen)
- (ausgewählte) Mutanten sind nicht einfacher oder schwerer zu detektieren als reale Fehler

4.8 Testmanagement und Testwerkzeuge

Testen wird nach [SL19] immer wie folgt durchgeführt:

- **Testplanung**: es werden die zum Einsatz kommenden Methoden und Werkzeuge sowie die zu testenden Objekte in einem Testkonzept festgelegt
- **Testüberwachung und Teststeuerung**: Durchführung, Protokollierung und Bewertung der folgenden Aktivitäten inklusive Entscheidung über Testende
- **Testanalyse**: es wird festgelegt was genau zu testen ist durch Überprüfung vorhandener Anforderungspezifikationen, Testobjekte, ...
- **Testentwurf**: es wird festgelegt wie getestet wird; dabei werden abstrakte (mit Bedingungen für Eingabewerte) oder konkrete Testfälle (mit konkreten Eingabewerten) spezifiziert
- **Testrealisierung**: es werden die spezifizierten Testfälle implementiert
- **Testdurchführung**: die ausgewählten Testfälle werden ausgeführt
- **Testabschluss**: es werden die Ergebnisse der vorangegangenen Aktivitäten zusammengetragen und konsolidiert

Aufgaben, Qualifikationen und Rollen nach [SL19]:

- **Testmanager** (Leiter): ist für Management der Testaktivitäten, der Testressourcen und die Bewertung des Testobjekts (gegenüber Projektmanager) verantwortlich
- **Testdesigner** (Analyst): erstellt Testspezifikationen und ermittelt Testdaten [Ergänzung: könnte beim modellbasierten Testen auch für die Erstellung von Testmodellen und Festlegung von Überdeckungskriterien zuständig sein]
- **Testfallgenerator**: werden Testfälle aus Modellen bzw. Spezifikationen automatisch erzeugt, so bedarf es einer weiteren Rolle, die die Testfallgenerierung mit Werkzeugunterstützung durchführt
- **Testautomatisierer**: realisiert die automatisierte Durchführung der spezifizierten Testfälle durch ausgewählte Testwerkzeuge
- **Testadministrator**: stellt die Testumgebung mit ausgewählten Testwerkzeugen zur Verfügung (zusammen Systemadministratoren etc.)
- **Tester**: ist für Testdurchführung, -protokollkierung und -auswertung zuständig (entspricht "Certified Tester Foundation Level"-Kompetenzen)

Weitere Aspekte des Testmanagements nach [SL19]:

- Betrachtung von **Kosten- und Wirtschaftlichkeitsaspekten**
  - Ermittlung und AbschätzunTeststrategieg von Fehlerkosten
  - Ermittlung und Abschätzung von Testkosten / Testaufwand
- Wahl einer
  - proaktiv vs. reaktiv (Testmanagement startet mit Projektbeginn vs. Testaktivitäten werden erst nach der Erstellung der Software gestartet)
  - Testerstellungsansatz / Überdeckungskriterien / ...
  - Orientierung an Standards für Vorgehensmodelle
- Testen und **Risiko**
  - Risiken beim Testen (Ausfall von Personal, ...)
  - Risikobasiertes Testen (Fokus auf Minimierung von Produktrisiken)

Meldung von Fehlern/Fehlermanagement: Es müssen alle Informationen erfasst werden, die für das Reproduzieren eines Fehlers notwendig sind. Eine Fehlermeldung kann wie folgt aufgebaut sein:

- **Status**: Bearbeitungsfortschritt der Meldung (Neu, Offen, Analyse, Abgewiesen, Korrektur, Test, Erledigt)
- **Klasse**: Klassifzierung der Schwere des Problems (Menschenleben in Gefahr, Systemabsturz mit Datenverlust, ... , Schönheitsfehler)
- **Priorität**: Festlegung der Dringlichkeit, mit der Fehler behoben werden muss (sofort, da Arbeitsablauf blockiert beim Anwender, ...)
- **Anforderung**: die Stelle(n) in der Anforderungsspezifikation, auf die sich der Fehler bezieht
- **Fehlerquelle**: in welcher Softwareentwicklungsphase wurde der Fehler begangen
- **Fehlerart**: Berechnungsfehler, ...
- **Testfall**: genaue Beschreibung des Testfalls, der Fehler auslöst
- ...

Arten von Testwerkzeugen: Testwerkzeuge werden oft "**Computer-Aided Software Test**"-Werkzeuge genannt (**CAST-Tools**). Man unterscheidet folgene Arten von CAST-Tools:

- Werkzeuge zum **Testmanagement** und zur Testplanung: Erfassen von Testfällen, Abgleich von Testfällen mit Anforderungen, Verwaltung und (statistische) Auswertung von Fehlermeldungen, ...
- Werkzeuge zur **Testspezifikation**: Testdaten und Soll-Werte für zugehörige Ergebnisse werden (manuell) festgelegt und verwaltet
- **Testdatengeneratoren**: aus Softwaremodellen, Code, Grammatiken, ... werden automatisch Testdaten generiert
- **Testtreiber**: passend zu Schnittstellen von Testobjekten werden Testtreiber bzw. Testrahmen zur Verfügung gestellt, die die Aufruf von Tests mit Übergabe von Eingabewerten, Auswertung der Ergebnisse etc. abwickeln
- **Simulatoren**: bilden möglichst realitätsnah Produktionsumgebung nach (mit Simulation von anderen Systemen, Hardware, ...)
- **Testroboter** (Capture & Replay-Werkzeug): zeichnet interaktive Benutzung einer Bedienungsoberfläche auf und kann Dialog wieder abspielen (solange sich Oberfläche nicht zu stark verändert)
- Werkzeuge für Last- und **Performanztests** (dynamische Analysen): Laufzeitmessungen, Speicherplatzverbrauch, ...
- **Komparatoren**: vergleichen erwartete mit tatsächlichen Ergebnissen, filtern unwesentliche Details, können ggf. auch Zustand von Benutzeroberflächen prüfen
- **Code-Überdeckungsanalysatoren**: für gewählte Überdeckungsmetriken wird Buch darüber geführt, wieviel Prozent Überdeckung erreicht ist, welche Programmausschnitte noch nicht überdeckt sind
- Werkzeuge für **statische Analysen**: Berechnung von Metriken, Kontroll- und Datenflussanomalien, ...

4.9 Zusammenfassung

Dynamische Programmanalysen und das "traditionelle" Testen sind die wichtigsten Mittel der analytischen Qualitätssicherung. Mindestens folgende Maßnahmen sollten immer durchgeführt werden:

- Suche nach Speicherlecks und Zugriff auf nichtinitialisierte Speicherbereiche (oder falsche Speicherbereiche) mit geeigneten Werkzeugen
- automatische Regressions-Testdurchführung mit entsprechenden Frameworks zur Testautomatisierung

- Überprüfung der Zweigüberdeckung durch Testfälle anhand von Kontrollflussgraph (durch Werkzeuge)
- Verwendung von "Black-Box"-Testverfahren mit Äquivalenzklassenbildung für Eingabeparameter (Objekte) zur Bestimmung von Testfällen
- Einbau möglichst vieler abschaltbarer Konsistenzprüfungen in Code (Vor- und Nachbedingungen) und ggf. defensive Programmierung

5 Management der Software-Entwicklung

Themen dieses Kapitels

- bessere/modernere Prozessmodelle
- Verbesserung/Qualität von Softwareprozessmodellen
- Projektmanagement(-werkzeuge)
- optional: Kostenschätzung für Softwareprojekte

Viele im folgenden vorgestellten Überlegungen sind nicht ausschließlich für **Software** Entwicklungsprozesse geeignet, sondern werden ganz allgemein für die Steuerung komplexer technischer Entwicklungsprozesse eingesetzt.

Aufgaben des Managements:

- **Planungsaktivitäten**: Ziele definieren, Vorgehensweisen auswählen, Termine fest legen, Budgets vorbereiten, ...
  - Vorgehensmodelle, Kostenschätzung, Projektpläne
- **Organisationsaktivitäten**: Strukturieren von Aufgaben, Festlegung organisatorischer Strukturen, Definition von Qualifikationsprofilen für Positionen, ...
  - Rollenmodelle, Team-Modelle, Projektpläne
- **Personalaktivitäten**: Positionen besetzen, Mitarbeiter beurteilen, weiterbilden, ...
  - nicht Thema dieser Vorlesung
- **Leitungsaktivitäten**: Mitarbeiter führen, motivieren, koordinieren, ...
  - nicht Thema dieser Vorlesung
- **Kontrollaktivitäten**: Prozess- und Produktstandards entwickeln, Berichts- und Kontrollwesen etablieren, Prozesse und Produkte vermessen, Korrekturen, ...
  - Qualititätsmanagement, insbesondere für Software-Entwicklungsprozesse

Ziele des Managements Hauptziel des Projektmanagements ist die **Erhöhung der Produktivität**!

Allgemeine Definition von Produktivität: Produktivität = Produktwert / Aufwand (oder: Leistung / Aufwand)

Für Software-Entwicklung oft verwendete Definition: Produktivität = Größe der Software / geleistete Mitarbeitertage

- Maß für Größe der Software
- Berücksichtigung der Produktqualität
- Aufwand = Mitarbeitertage ?
- Nutzen (Return Of Investment) = Größe der Software ?

Einflussfaktoren für Produktivität [ACF97]: Angabe der Form "+ 1 : X" steht für Produktivitätssteigerung um maximal Faktor X
Angabe der Form "- 1 : Y" steht für Produktivitätsminderung um maximal Faktor Y.

- Werkzeug-Einsatz: + 1 : 1,6
- Geeignete Methoden: + 1 : 1,9
- Produktkomplexität: - 1 : 2,4
- Hohe Zuverlässigkeitsanforderung: - 1 : 1,9
- Firmenkultur (corporate identity): + 1 : 11
- Arbeitsumgebung (eigenes Büro, abschaltbares Telefon, ...): + 1 : ???
- Begabung der Mitarbeiter: + 1 : 10
- Erfahrung im Anwendungsgebiet: + 1 : 1,6
- Bezahlung, Berufserfahrung: kein messbarer Einfluss

5.1 "Neuere" Vorgehensmodelle

Die naheliegendste Idee zur Verbesserung des Wasserfallmodells ergibt sich durch die Einführung von **Zyklen** bzw. **Rückgriffen**. Sie erlauben Wiederaufnehmen früherer Phasen, wenn in späteren Phasen Probleme auftreten.

Weitere Vorgehensmodelle:

- das V-Modell (umgeklapptes Wasserfallmodell)
- das evolutionäre Modell (iteriertes Wasserfallmodell)
- Rapid Prototyping (Throw-Away-Prototyping)

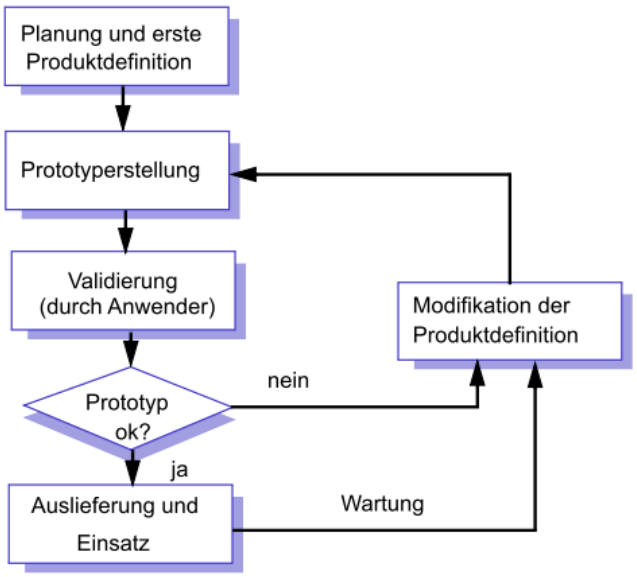
Probleme mit dem Wasserfallmodell insgesamt:

- Wartung mit ca. 60% des Gesamtaufwandes ist eine Phase
  - andere Prozessmodelle mit Wartung als eigener Entwicklungsprozess
- zu Projektbeginn sind nur ungenaue Kosten- und Ressourcenschätzungen möglich
  - Methoden zur Kostenschätzung anhand von Lastenheft (Pflichtenheft)
- ein Pflichtenheft kann nie den Umgang mit dem fertigen System ersetzen, das erst sehr spät entsteht (Risikomaximierung)
  - andere Prozessmodelle mit Erstellung von Prototypen, ...
- Anforderungen werden früh eingefroren, notwendiger Wandel (aufgrund organisatorischer, politischer, technischer, ... Änderungen) nicht eingeplant
  - andere Prozessmodelle mit evolutionärer Software-Entwicklung
- strikte Phaseneinteilung ist unrealistisch (Rückgriffe sind notwendig)
  - andere Prozessmodelle mit iterativer Vorgehensweise

Bewertung des evolutionären Modells:

- **Vorteile:**
  - es ist sehr früh ein (durch Kunden) evaluierbarer Prototyp da
  - Kosten und Leistungsumfang des gesamten Softwaresystems müssen nicht zu Beginn des Projekts vollständig festgelegt werden
  - Projektplanung vereinfacht sich durch überschaubarere Teilprojekte
  - Systemarchitektur muss auf Erweiterbarkeit angelegt sein
- **Nachteile:**
  - es ist schwer, Systemarchitektur des ersten Prototypen so zu gestalten, dass sie alle später notwendigen Erweiterungen erlaubt

Abbildung 23: Evolutionäres Modell (evolutionäres Prototyping)



- Prozess der Prototyperstellung nicht festgelegt: Spiralmodell von Berry Böhm integriert Phasen des Wasserfallmodells
- evolutionäre Entwicklung der Anforderungsdefinition birgt Gefahr in sich, dass bereits realisierte Funktionen hinfällig werden
- Endresultat sieht ggf. wie Software nach 10 Jahren Wartung aus

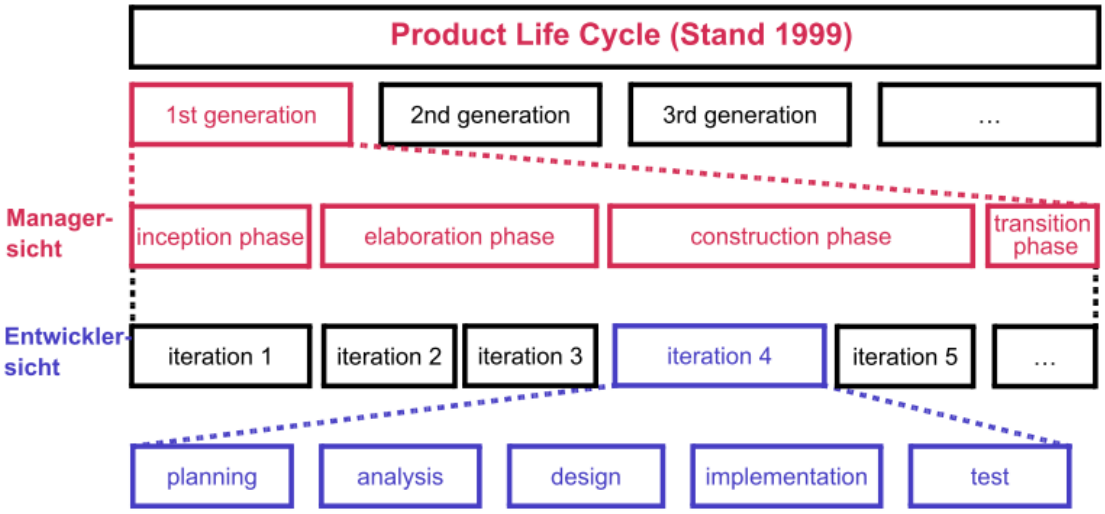
Rapid Prototyping (Throw-Away-Prototyping): Mit Generatoren, ausführbaren Spezifikationssprachen, Skriptsprachen etc. wird:

- Prototyp des Systems (seiner Benutzeroberfläche) realisiert
- dem Kunden demonstriert
- und anschließend weggeschmissen

Bewertung:

- erlaubt schnelle Klärung der Funktionalität und Risikominimierung
- Vermeidung von Missverständnissen zwischen Entwickler und Auftraggeber
- früher Test der Benutzerschnittstelle

Abbildung 24: Firma IBM (ehemals Rational) dominiert(e) Entwicklung der Standard-OO-Modellierungssprache UML und des zugehörigen Vorgehensmodells.



5.2 Rational Unified Process für UML

Phasen der Lebenszyklusgeneration

- **Inception (Vorbereitung)**: Definition des Problembereichs und Projektziels für Produktgeneration mit Anwendungsbereichsanalyse (Domain Analysis) und Machbarkeitsstudie (für erste Generation aufwändiger)
  - bei Erfolg weiter zu ...
- **Elaboration (Entwurf)**: erste Anforderungsdefinition für Produktgeneration mit grober Softwarearchitektur und Projektplan (ggf. mit Rapid Prototyping)
  - bei Erfolg weiter zu ...
- **Construction (Konstruktion)**: Entwicklung der neuen Produktgeneration als eine Abfolge von Iterationen mit Detailanalyse, -design, ... (wie beim evolutionären Vorgehensmodell)
  - bei Erfolg weiter zu ...
- **Transition (Einführung)**: Auslieferung des Systems an den Anwender (inklusive Marketing, Support, Dokumentation, Schulung, ...)

Eigenschaften des Rational Unified Prozesses (RUP):

- **modellbasiert**: für die einzelnen Schritte des Prozesses ist festgelegt, welche Modelle (Dokumente) des Produkts zu erzeugen sind
- **prozessorientiert**: die Arbeit ist in eine genau definierte Abfolge von Aktivitäten unterteilt, die von anderen Teams in anderen Projekten wiederholt werden können.
- **iterativ und inkrementell**: die Arbeit ist in eine Vielzahl von Iterationen unterteilt, das Produkt wird inkrementell entwickelt.
- **risikobewusst**: Aktivitäten mit hohem Risiko werden identifiziert und in frühen Iterationen in Angriff genommen.
- **zyklisch**: die Produktentwicklung erfolgt in Zyklen (Generationen). Jeder Zyklus liefert eine neue als kommerzielles Produkt ausgelieferte Systemgeneration.
- **ergebnisorientiert**: jede Phase (Iteration) ist mit der Ablieferung eines definierten Ergebnisses meist zu einem konkreten Zeitpunkt (Meilenstein) verbunden

Faustregeln für die Ausgestaltung eines Entwicklungsprozesses:

- die Entwicklung einer **Produktgeneration** dauert höchstens 18 Monate
- eine **Vorbereitungsphase** dauert 3-6 Wochen und besteht aus einer Iteration
- eine **Entwurfsphase** dauert 1-3 Monate und besteht aus bis zu 2 Iterationen
- eine **Konstruktionsphase** dauert 1-9 Monate und besteht aus bis zu 7 Iterationen
- eine **Einführungsphase** dauert 4-8 Wochen und besteht aus einer Iteration
- jede **Iteration** dauert 4-8 Wochen (ggf. exklusive Vorbereitungs- und Nachbereitungszeiten, die mit anderen Iterationen überlappen dürfen)
- das gewünschte **Ergebnis** (Software-Release) einer Iteration ist spätestens bei ihrem Beginn festgelegt (oft Abhängigkeit von Ergebnissen vorheriger Iterationen)
- die **geplante Zeit** für eine Iteration wird nie (höchstens um 50%) überschritten
- innerhalb der Konstruktionsphase wird mindestens im wöchentlichen Abstand ein **internes Software-Release** erstellt
- mindestens 40% **Reserve** an Projektlaufzeit für unbekannte Anforderungen, ...

Anmerkungen zu den Arbeitsbereichen (Workflows) des RUP:

- **Business Modeling** befasst sich damit, das Umfeld des zu erstellenden Softwaresystems zu erfassen (Geschäftsvorfälle, Abläufe, ...)
- **Requirements (Capture)** befasst sich damit die Anforderungen an ein Softwaresystem noch sehr informell zu erfassen
- **Analysis/Design** präzisiert mit grafischen Sprachen (Klassendiagramme etc.) die Anforderungen und liefert Systemarchitektur
- **Implementation/Test** entspricht den Aktivitäten in den Phasen "Codierung bis Integrationstest" des Wasserfallmodells
- **Deployment** entspricht "Auslieferung und Installation" des Wasserfallmodells
- **Configuration Management** befasst sich mit der Verwaltung von Softwareversionen und -varianten
- **Project Management** mit der Steuerung des Entwicklungsprozesses selbst
- **Environment** bezeichnet die Aktivitäten zur Bereitstellung benötigter Ressourcen (Rechner, Werkzeuge, ...)

Bewertung des (Rational) Unified Prozesses:

- Vorteile
  - Manager hat die grobe "Inception-Elaboration-Construction-Transition"-Sicht
  - Entwickler hat zusätzlich die feinere arbeitsbereichsorientierte Sicht
  - Wartung ist eine Abfolge zu entwickelnder Produktgenerationen
  - es wird endgültig die Illusion aufgegeben, dass Analyse, Design, ... zeitlich begrenzte strikt aufeinander folgende Phasen sind
  - es gibt "Open Unified Process" im Eclipse Umfeld
- Nachteile
  - sehr komplexes Vorgehensmodell für modellbasierte SW-Entwicklung
  - nicht mit Behördenstandards (V-Modell, ...) richtig integriert
  - Qualitätssicherung ist kein eigener Aktivitätsbereich

5.3 Leichtgewichtige Prozessmodelle

Herkömmlichen Standards für Vorgehensmodelle zur Softwareentwicklung (V-Modell, RUP) wird vorgeworfen, dass

- sie sehr starr sind
- Unmengen an Papier produzieren
- und nutzlos Arbeitskräfte binden

Deshalb werden seit einiger Zeit sogenannte "leichtgewichtige" Prozessmodelle (**light-weight processes**) unter dem Schlagwort "Agile Prozessmodelle" propagiert, die sinnlosen bürokratischen Overhead vermeiden wollen.

- siehe separater Foliensatz zu dieser Thematik

Verbesserung der Prozessqualität Ausgangspunkt der hier vorgestellten Ansätze sind folgende Überlegungen:

- Softwareentwicklungsprozesse sind selbst Produkte, deren Qualität überwacht und verbessert werden muss
- bei der Softwareentwicklung sind bestimmte Standards einzuhalten (Entwicklungsprozess muss dokumentiert und nachvollziehbar sein)
- es bedarf kontinuierlicher Anstregungen, um die Schwächen von Entwicklungsprozessen zu identifizieren und zu eliminieren

Hier vorgestellte Ansätze:

- **ISO 9000 Normenwerk** (Int. Standard für die Softwareindustrie)
- **Capability Maturity Model** (CMM/CMMI) des Software Engineering Institutes (SEI) an der Carnegie Mellon University
- ISO-Norm **SPiCE** integriert und vereinheitlicht CMM und ISO 9000

Qualitätssicherung mimt der ISO 9000: Das **ISO 9000 Normenwerk** legt für das Auftraggeber-Lieferantenverhältnis einen allgemeinen organisatorischen Rahmen zur Qualitätssicherung fest.

Das ISO 9000 Zertifikat bestätigt, dass die Verfahren eines Unternehmens der ISO 9000 Norm entsprechen.

Wichtige Teile:

- **ISO 9000-1**: allgemeine Einführung und Überblick
- **ISO 9000-3**: Anwendung von ISO 9001 auf Softwareproduktion
- **ISO 9001**: Modelle der Qualitätssicherung in Design/Entwicklung, Produktion, Montage und Kundendienst
- **ISO 9004**: Aufbau und Verbesserung eines Qualitätsmanagementsystems

Von ISO 9000-3 vorgeschriebene Dokumente:

- **Vertrag Auftraggeber - Lieferant:** Tätigkeiten des Auftraggebers, Behandlung von Anforderungsänderungen, Annahmekriterien (Abnahmetest), ...
- **Spezifikation:** funktionale Anforderungen, Ausfallsicherheit, Schnittstellen, ... des Softwareprodukts
- **Entwicklungsplan:** Zielfestlegung, Projektmittel, Entwicklungsphasen, Management, eingesetzte Methoden und Werkzeuge, ...
- **Qualitätssicherungsplan:** Qualitätsziele (messbare Größen), Kriterien für Ergebnisse v. Entwicklungsphasen, Planung von Tests, Verifikation, Inspektionen
- **Testplan:** Teststrategie (für Integrationstest), Testfälle, Testwerkzeuge, Kriterien für Testvollständigkeit/Testende
- **Wartungsplan:** Umfang, Art der Unterstützung, ...
- **Konfigurationsmanagement:** Plan für Verwaltung von Softwareversionen und Softwarevarianten

Von ISO 9000-3 vorgeschriebene Tätigkeiten:

- **Konfigurationsmanagement** für Identifikation und Rückverfolgung von Änderungen, Verwaltung parallel existierender Varianten
- **Dokumentenmanagement** für geordnete Ablage und Verwaltung aller bei der Softwareentwicklung erzeugten Dokumente
- **Qualitätsaufzeichungen** (Fehleranzahl oder Metriken) für Verbesserungen am Produkt und Prozess
- **Festlegung** von Regeln, Praktiken und Übereinkommen für ein Qualitätssicherungssystem
- **Schulung** aller Mitarbeiter sowie Verfahren zur Ermittlung des Schulungsbedarfes

Zertifizierung: Die Einhaltung der Richtlinien der Norm wird von unabhängigen Zertifizierungsstellen im jährlichen Rhythmus überprüft.

Bewertung von ISO 9000:

- Vorteile
  - lenkt die Aufmerksamkeit des Managements auf Qualitätssicherung
  - ist ein gutes Marketing-Instrument
  - reduziert das Produkthaftungsrisiko (Nachvollziehbarkeit von Entscheidungen)
- Nachteile
  - Nachvollziehbarkeit und Dokumentation von Prozessen reicht aus
  - keine Aussage über Qualität von Prozessen und Produkten
  - (für kleine Firmen) nicht bezahlbarer bürokratischer Aufwand
  - Qualifikation der Zertifizierungsstellen umstritten
  - oft große Abweichungen zwischen zertifiziertem Prozess und realem Prozess

Das Capability Maturity Model (CMM): Referenzmodell zur Beurteilung von Softwarelieferanten, vom Software Engineering Institute entwickelt

- Softwareentwicklungsprozesse werden in **5 Reifegrade** unterteilt
- Reifegrad (**maturity**) entspricht Qualitätsstufe der Softwareentwicklung
- höhere Stufe beinhaltet Anforderungen der tieferen Stufen

Stufe 1 - chaotischer initialer Prozess (ihr Stand vor dieser Vorlesung):

- Prozess-Charakteristika:
  - unvorhersehbare Entwicklungskosten, -zeit und -qualität
  - kein Projektmanagement, nur "Künstler" am Werke
- notwendige Aktionen:
  - Planung mit Kosten- und Zeitschätzung einführen
  - Änderungs- und Qualitätssicherungsmanagement

Stufe 2 - wiederholbarer intuitiver Prozess (Stand nach dieser Vorlesung?):

- Prozess-Charakteristika:
  - Kosten und Qualität schwanken, gute Terminkontrolle
  - Know-How einzelner Personen entscheidend
- notwendige Aktionen:

  - Prozessstandards entwickeln
  - Methoden (für Analyse, Entwurf, Testen, ...) einführen

Stufe 3 - definierter qualitativer Prozess (Stand der US-Industrie 1989?):

- Prozess-Charakteristika:
  - zuverlässige Kosten- und Terminkontrolle, schwankende Qualität
  - institutionalisierter Prozess, unabhängig von Individuen
- notwendige Aktionen:
  - Prozesse vermessen und analysieren
  - quantitative Qualitätssicherung

Stufe 4 - gesteuerter/geleiteter quantitativer Prozess:

- Prozess-Charakteristika:
  - gute statistische Kontrolle über Produktqualität
  - Prozesse durch Metriken gesteuert
- notwendige Aktionen:
  - instrumentierte Prozessumgebung (mit Überwachung)
  - ökonomisch gerechtfertigte Investitionen in neue Technologien

Stufe 5 - optimierender rückgekoppelter Prozess:

- Prozess-Charakteristika:
  - quantitative Basis für Kapitalinvestitionen in Prozessautomatisierung und -verbesserung
- notwendige Aktionen:
  - kontinuierlicher Schwerpunkt auf Prozessvermessung und -verbesserung (zur Fehlervermeidung)

ISO 9000 und CMM im Vergleich

- Schwerpunkt der **ISO 9001** Zertifizierung liegt auf Nachweis eines Qualitätsmanagementsystems im Sinne der Norm
  - allgemein für Produktionsabläufe geeignet
  - genau ein Reifegrad wird zertifiziert
- **CMM** konzentriert sich auf Qualitäts- und Produktivitätssteigerung des gesamten Softwareentwicklungsprozesses

  - auf Softwareentwicklung zugeschnitten
  - dynamisches Modell mit kontinuierlichem Verbesserungsdruck
- ISO-Norm **SPiCE** integriert und vereinheitlicht CMM und ISO 9000 (als ISO/IEC 15504)

SPiCE = Software Process Improvement and Capability dEtermination: Internationale Norm für **Prozessbewertung** (und Verbesserung). Sie bildet einheitlichen Rahmen für Bewertung der Leistungsfähigkeit von Organisationen, deren Aufgabe Entwicklung oder Erwerb, Lieferung, Einführung und Betreuung von Software-Systemen ist. Norm legt Evaluierungsprozess und Darstellung der Evaluierungsergebnisse fest.

Unterschiede zu CMM:

- orthogonale Betrachtung von Reifegraden und Aktivitätsbereichen
- deshalb andere Definition der 5 Reifegrade (z.B. "1" = alle Aktivitäten eines Bereiches sind vorhanden, Qualität der Aktivitäten noch unerheblich, ...)
- jedem Aktivitätsbereich oder Unterbereich kann ein anderer Reifegrad zugeordnet werden

Aktivitätsbereich von SPiCE:

- **Customer-Supplier-Bereich:**
  - Aquisition eines Projektes (Angebotserstellung, ...)
  - ...
- **Engineering-Bereich:**
  - Software-Entwicklung (Anforderungsanalyse, ... , Systemintegration)
  - Software-Wartung
- **Support-Bereich:**
  - Qualitätssicherung
  - ...
- **Management-Bereich:**
  - Projekt-Management
  - ...
- **Organisations-Bereich:**
  - Prozess-Verbesserung
  - ...

CMMI = Capability Maturity Model Integration (neue Version von CMM): CMMI ist die **neue Version des Software Capability Maturity Model**. Es ersetzt nicht nur verschiedene Qualitäts-Modelle für unterschiedliche Entwicklungs-Disziplinen (z.B. für Software-Entwicklung oder System-Entwicklung), sondern integriert diese in einem neuen, modularen Modell. Dieses modulare Konzept ermöglicht zum einen die Integration weiterer Entwicklungs-Disziplinen (z.B. Hardware-Entwicklung), und zum anderen auch die Anwendung des Qualitätsmodells in übergreifenden Disziplinen (z.B. Entwicklung von Chips mit Software).

Geschichte von CMM und CMMI:

- 1991 wird Capability Maturity Model 1.0 herausgegeben
- 1993 wird CMM überarbeitet und in der Version 1.1 bereitgestellt
- 1997 wird CMM 2.0 kurz vor Verabschiedung vom DoD zurückgezogen
- 2000 wird CMMI als Pilotversion 1.0 herausgegeben
- 2002 wird CMMI freigegeben
- Ende 2003 ist die Unterstützung von CMM ausgelaufen

Eigenschaften von CMMI: Es gibt **Fähigkeitsgrade** für einzelne Prozessgebiete (ähnlich zu SPiCE):

- **0 - Incomplete:**
  - Ausgangszustand, keine Anforderungen
- **1 - Performed:**
  - die spezifischen Ziele des Prozessgebiets werden erreicht
- **2 - Managed:**
  - der Prozess wird gemanagt
- **3 - Defined:**
  - der Prozess wird auf Basis eines angepassten Standard-Prozesses gemanagt und verbessert
- **4 - Quantitatively Managed:**
  - der Prozess steht unter statistischer Prozesskontrolle
- **5 - Optimizing:**
  - der Prozess wird mit Daten aus der statistischen Prozesskontrolle verbessert

Eigenschaften von CMMI-Fortsetzung: Es gibt **Reifegrade**, die Fähigkeitsgrade auf bestimmten Prozessgebieten erfordern (ähnlich zu CMM):

- **1- Initial:**
  - keine Anforderungen, diesen Reifegrad hat jede Organisation automatisch
- **2 - Managed:**
  - die Projekte werden gemanagt durchgeführt und ein ähnliches Projekt kann erfolgreich wiederholt werden
- **3 - Defined:**
  - die Projekte werden nach einem angepassten Standard-Prozess durchgeführt, und es gibt eine kontinuierliche Prozessverbesserung
- **4 - Quantitatively Managed:**
  - es wird eine statistische Prozesskontrolle durchgeführt
- **5 - Optimizing:**
  - die Prozesse werden mit Daten aus statistischen Prozesskontrolle verbessert

Konsequenzen für die "eigene" Software-Entwicklung: Im Rahmen von Studienarbeiten, Diplomarbeiten, ... können Sie keinen CMM(I)-Level-5- oder SPiCE-Software-Entwicklungsprozess verwenden, aber:

- Einsatz von Werkzeugen für **Anforderungsanalyse und Modellierung**
  - in der Vorlesung "Software-Engineering - Einführung" behandelt
- Einsatz von **Konfigurations- und Versionsmanagement-Software**
  - wird in dieser Vorlesung behandelt
- Einsatz von Werkzeugen für systematisches **Testen, Messen** der Produktqualität
  - wird in dieser Vorlesung behandelt
- Ergänzender Einsatz von "**Extreme Programming**"-Techniken (z.B. "Test first")
- Einsatz von Techniken zur Verbesserung des "persönlichen" Vorgehensmodells
  - siehe [Hu96] über den "**Personal Software Process**"

5.4 Projektpläne und Projektorganisation

Am Ende der Machbarkeitsstudie steht die Erstellung eines Projektplans mit

- Identifikation der einzelnen **Arbeitspakete**
- **Terminplanung** (zeitliche Aufeinanderfolge der Pakete)
- **Ressourcenplanung** (Zuordnung von Personen zu Paketen, ...)

Hier wird am deutlichsten, dass eine Machbarkeitsstudie ohne ein grobes Design der zu erstellenden Software nicht durchführbar ist, da:

- Arbeitspakete ergeben sich aus der Struktur der Software
- Abhängigkeiten und Umfang der Pakete ebenso
- Realisierungsart der Pakete bestimmt benötigte Ressourcen

Konsequenz: Projektplanung und -organisation ist ein fortlaufender Prozess. Zu Projektbeginn hat man nur einen groben Plan, der sukzessive verfeinert wird.

Terminologie:

- **Prozessarchitektur** = grundsätzliche Vorgehensweise einer Firma für die Beschreibung von Software-Entwicklungsprozessen (Notation, Werkzeuge)
- **Prozessmodell** = Vorgehensmodell = von einer Firma gewählter Entwicklungsprozess (Wasserfallmodell oder RUP oder ...)
- **Projektplan** = an einem Prozessmodell sich orientierender Plan für die Durchführung eines konkreten Projektes
- **Vorgang** = Aufgabe = Arbeitspaket = abgeschlossene Aktivität in Projektplan, die
  - bestimmte Eingaben (Vorbedingungen) benötigt und Ausgaben produziert
  - Personal und (sonstige) Betriebsmittel für Ausführung braucht
  - eine bestimmte Zeitdauer in Anspruch nimmt
  - und Kosten verursacht und/oder Einnahmen bringt
- **Phase** = Zusammenfassung mehrerer zusammengehöriger Vorgänge zu einem globalen Arbeitsschritt
- **Meilenstein** = Ende einer Gruppe von Vorgängen (Phase) mit besonderer Bedeutung (für die Projektüberwachung) und wohldefinierten Ergebnissen

5.5 Aufwands- und Kostenschätzung

Die **Kosten** eines Softwareproduktes und die **Entwicklungsdauer** werden im wesentlichen durch den personellen Aufwand bestimmt. Bislang haben wir vorausgesetzt, dass der personelle Aufwand bekannt ist, hier werden wir uns mit seiner Berechnung bzw. Schätzung befassen.

Der **personelle Aufwand** für die Erstellung eines Softwareproduktes ergibt sich aus

- dem "**Umfang**" des zu erstellenden Softwareprodukts
- der geforderten **Qualität** für das Produkt

Übliches Maß für Personalaufwand: Mitarbeitermonate(MM) oder Mitarbeiterjahre (MJ): 1MJ = 10MM (wegen Urlaub, Krankheit,...)
Übliches Maß für Produktumfang: "LOC"

Schätzverfahren im Überblick:

- **Analogiemethode:** Experte vergleicht neues Projekt mit bereits abgeschlossenen ähnlichen Projekten und schätzt Kosten "gefühlsmäßig" ab
  - Expertenwissen lässt sich schwer vermitteln und nachvollziehen
- **Prozentsatzmethode:** aus abgeschlossenen Projekten wird Aufwandsverteilung auf Phasen ermittelt; anhand beendeter Phasen wird Projektrestlaufzeit geschätzt
  - funktioniert allenfalls nach Abschluss der Analysephase
- **Parkinsons Gesetz:** die Arbeit ist beendet, wenn alle Vorräte aufgebraucht sind
  - praxisnah, realistisch und wenig hilfreich ...
- **Price to Win:** die Software-Kosten werden auf das Budget des Kundens geschätzt
  - andere Formulierung von "Parkinsons Gesetz", führt in den Ruin ...
- **Gewichtungsmethode:** Bestimmung vieler Faktoren (Erfahrung der Mitarbeiter, verwendete Sprachen, ...) und Verknüpfung durch mathematische Formel
  - LOC-basierter Vertreter: COnstructive COst MOdel (COCOMO)
  - FP-basierte Vertreter: Function-Point-Methoden in vielen Varianten

	Analyse	Design	Codierung	Test	Sonstiges
C	3 Wochen	5 Wochen	8 Wochen	10 Wochen	2 Wochen
Smalltalk	3 Wochen	5 Wochen	2 Wochen	6 Wochen	2 Wochen

Tabelle 1: Einfluss von Programmiersprache auf Produktivität:

	Programmgröße	Aufwand	Produktivität
C	3 2.000 LOC	28 Wochen	70 LOC/Woche
Smalltalk	500 LOC	18 Wochen	27 LOC/Wochen

Tabelle 2: Konsequenzen für die Gesamtproduktivität:

Softwareumfang = Lines of Code? Die "**Lines of Code**" als Ausgangsbasis für die Projektplanung (und damit auch zur Überwachung der Produktivität von Mitarbeitern) zu verwenden ist fragwürdig, da:

- Codeumfang erst mit Abschluss der Implementierungsphase bekannt ist
- selbst Architektur auf Teilsystemebene noch unbekannt ist
- Wiederverwendung mit geringeren LOC-Zahlen bestraft wird
- gründliche Analyse, Design, Testen, ... zu geringerer Produktivität führt
- Anzahl von Codezeilen abhängig vom persönlichen Programmierstil ist
- Handbücher schreiben, ... ungenügend berücksichtigt wird

Achtung: Die starke Abhängigkeit der LOC-Zahlen von einer Programmiersprache ist zulässig, da Programmiersprachenwahl (großen) Einfluss auf Produktivität hat.

Fazit:

- Produktivität kann **nicht** in "Lines Of Code pro Zeiteinheit" sinnvoll gemessen werden (sonst wäre Programmieren in Assembler die beste Lösung)
- also: Vorsicht mit Einsatz von Maßzahlen (keine sozialistische Planwirtschaft)

Softwareumfang = Function Points! Bei der **Function-Point-Methode** zur Kostenschätzung wird der Softwareumfang anhand der Produktanforderungen aus dem Lastenheft geschätzt. Es gibt inzwischen einige Spielarten; hier wird (weitgehend) der Ansatz der **International Function Point Users Group (IFPUG)** vorgestellt.

Jede Anforderung wird gemäß IFPUG einer von 5 Kategorien zugeordnet [ACF97]:

1. **Eingabedaten** (über Tastatur, CD, externe Schnittstellen, ...)

Interne Entitäten	Anzahl Attribute <= 19	19 < Anzahl Attribute <= 50	Anzahl Attribute > 50
Klassen+Assoz. <= 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
2 <= Klassen+Assoz. <= 5	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Klassen+Assoz. > 5	mittlere Komplexität	hohe Komplexität	hohe Komplexität

Tabelle 3: Interne Entitäten

2. **Ausgabedaten** (auf Bildschirm, Papier, externe Schnittstelle, ...)
3. **Abfragen** (z.B. SQL-Queries auf einem internen Datenbestand)
4. **Datenbestände** (sich ändernde interne Datenbankinhalte)
5. **Referenzdateien** (im wesentlichen unveränderliche Daten)

Dann werden **Function-Points (FPs)** berechnet, bewertet, ...

Datenbestände = Internal Logical File (ILF) = Interne Entitäten: Unter einer **internen (Geschäfts-)Entität** definiert die IFPUG eine aus Anwendersicht logisch zusammengehörige Gruppe vom Softwaresystem verwalteter Daten, also z.B.:

- eine Gruppe von Produktdaten des Lastenheftes in der Machbarkeitsstudie
- Klassen mit Attributen u. Beziehungen eines Paketes aus Modellen im Pflichtenheft in der Analysephase

Es werden Datenelementtypen (Attribute) sowie Entitätstypen (Klassen, Sätze) und zusätzlich Beziehungstypen (Assoziationen) gezählt. Anhand dieser Zählung wird Komplexität eines Datenbestandes wie folgt bestimmt:
einfach = 7 FPs, mittel = 10 FPs oder komplex = 15 FPs

Referenzdateien = External Interface File = (EIF) = Externe Entitäten: Unter einer **externen (Geschäfts-)Entität** definiert die IFPUG eine aus Anwendersicht logisch zusammengehörige Gruppe vom System benutzter aber nicht selbst verwalteter Daten.

Wieder werden Datenelementtypen (Attribute) sowie Entitätstypen (Klassen, Sätze) und zusätzlich Beziehungstypen (Assoziationen) gezählt. Anhand dieser Zählung wird Komplexität eines Datenbestandes wie folgt bestimmt:
einfach = 5 FPs, mittel = 7 FPs oder komplex = 10 FPs

Es werden weniger FPs als bei internen Entitäten vergeben, da die betrachteten Datenbestände nur eingelesen aber nicht verwaltet werden müssen.

Externe Entitäten	Anzahl Attribute <= 19	19 < Anzahl Attribute <= 50	Anzahl Attribute > 50
Klassen+Assoz. <= 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
2 <= Klassen+Assoz. <= 5	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Klassen+Assoz. > 5	mittlere Komplexität	hohe Komplexität	hohe Komplexität

Tabelle 4: Externe Entitäten

Externe Eingabe	Anzahl Attribute <= 4	4 < Anzahl Attribute <= 15	Anzahl Attribute > 15
Anzahl Klassen <= 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
Anzahl Klassen = 2	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Anzahl Klassen > 2	mittlere Komplexität	hohe Komplexität	hohe Komplexität

Tabelle 5: Externe Eingaben

(Externe) Eingabedaten = External Input (EI): Eingabedaten für **Elementarprozess**, der Daten oder Steuerinformationen des Anwenders verarbeitet, aber keine Ausgabedaten liefert. Es handelt sich dabei um den kleinsten selbständigen Arbeitsschritt in der Arbeitsfolge eines Anwenders, als etwa:

- Produktfunktionen des Lastenheftes in der Machbarkeitsstudie
- "Use Cases" aus Pflichtenheft in der Analysephase

Gezählt werden für jeden Elementarprozess die Anzahl seiner als Eingabe verwendeten Entitätstypen (Klassen, Sätze) und deren Datenelementtypen (Attribute, Felder). Anhand dieser Zählung wird Komplexität des Elementarprozesses wie folgt bestimmt:
einfach = 3 FPs, mittel = 4 FPs oder komplex = 6 FPs

Externe Ausgaben = External Output (EO) Ausgabedaten eines **Elementarprozesses** (Produktfunktion, Use Case), der Anwender Daten oder Steuerinformationen liefert. Achtung: der Elementarprozess darf keine Eingabedaten benötigen; ansonsten handelt es sich um eine "Externe Abfrage" oder ... .

Gezählt werden für jeden Elementarprozess die Anzahl seiner als Ausgabe verwendeten Entitätstypen (Klassen, Sätze) und deren Datenelementtypen (Attribute, Felder). Anhand dieser Zählung wird Komplexität des Elementarprozesses wie folgt bestimmt: **einfach = 4 FPs, mittel = 5 FPs oder komplex = 7 FPs**

Externe Abfragen = External Inquiry (EQ): Betrachtet werden **Elementarprozesse** (Produktfunktion, Use Case), die anhand von Eingaben Daten des internen Datenbestandes ausgeben (ohne auf diesen Daten komplexe Berechnungen durchzuführen).

Nach den Regeln für "Externe Eingaben" werden die Eingabedaten bewertet, nach den Regeln für "Externe Ausgaben" die Ausgabedaten; anschließend wird die höhere Komplexität übernommen und wie folgt

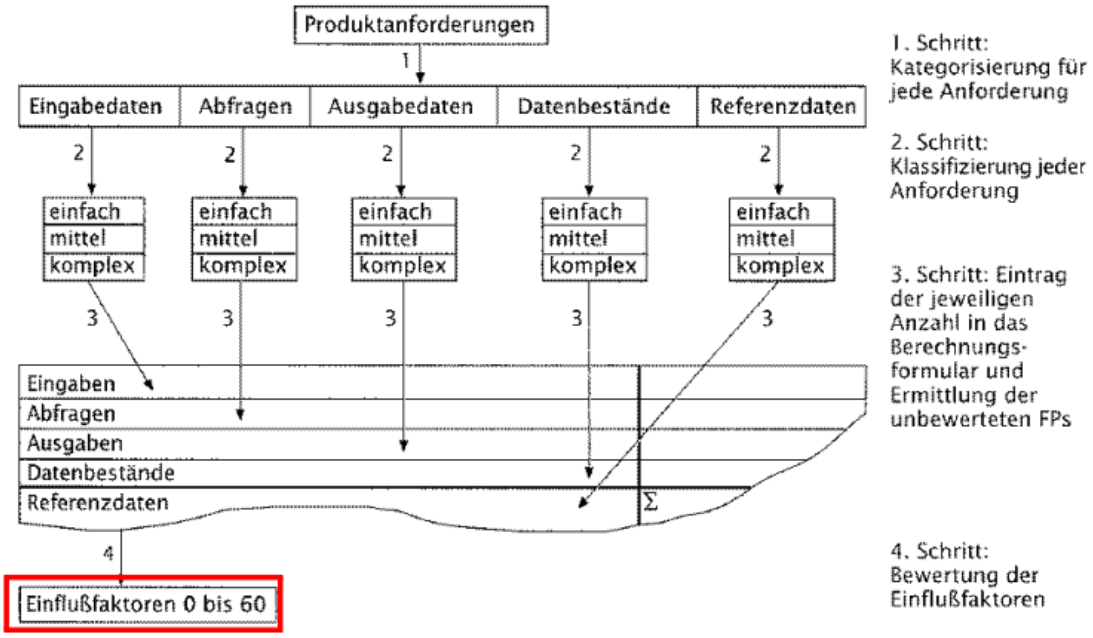
Externe Ausgaben	Anzahl Attribute <= 5	5 < Anzahl Attribute <= 19	Anzahl Attribute > 19
Anzahl Klassen <= 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
2 <= Anzahl Klassen <= 3	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Anzahl Klassen > 3	mittlere Komplexität	hohe Komplexität	hohe Komplexität

Tabelle 6: Externe Ausgaben

umgerechnet:
einfach = 3 FPs, mittel = 4 FPs oder komplex = 6 FPs

Achtung: ein Elementarprozess, der Eingabedaten zur Suche nach intern gespeicherten Daten benötigt und vor der Ausgabe **komplexe Berechnungen** durchführt, wird anders behandelt. In diesem Fall wird nicht das Maximum gebildet, sondern die **Summe** der FPs von "Externe Eingabe" und "Externe Ausgabe".

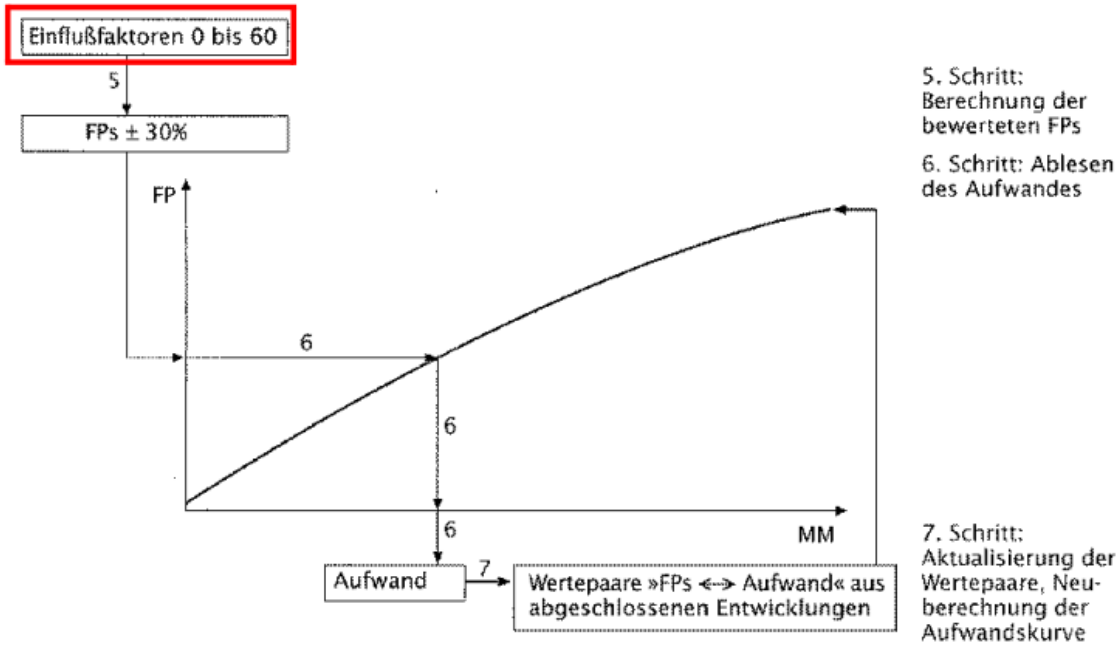
Abbildung 25: Überblick über die FP-Methode - 1 [Ba98]:



Zusätzliche Einflussfaktoren: Die vorige Tabelle unterscheidet sieben Einflussfaktoren; andere Quellen nennen 14 bzw. **19 verschiedene Faktoren**, die Werte von 0 bis 5 erhalten (siehe [Hu99]):

1. Komplexität der Datenkommunikation
2. Grad der verteilten Datenverarbeitung

Abbildung 26: Überblick über die FP-Methode - 2:



3. geforderte Leistungsfähigkeit
4. Komplexität der Konfiguration (Zielplattform)
5. Anforderung an Transaktionsrate
6. Prozentsatz interaktiver Dateneingaben
7. geforderte Benutzerfreundlichkeit
8. interaktive bzw. Online-Pflege des internen Datenbestandes
9. Komplexität der Verarbeitungslogik
10. geforderter Grad der Wiederverwendbarkeit
11. benötigte Installationshilfen
12. leichte Bedienbarkeit (Grad der Automatisierung der Bedienung)
13. Mehrfachinstallationen (auf verschiedenen Zielplattformen)
14. Grad der gefoderten Änderungsfreundlichkeit

15. Randbedingungen anderer Anwendungen
16. Sicherheit, Datenschutz, Prüfbarkeit
17. Anwenderschulung
18. Datenaustausch mit anderen Anwendungen
19. Dokumentation

Ermittlung der FP-Aufwandskurve:

- beim ersten Projekt muss man auf **bekannte Kurven** (für ähnliche Projekte) zurückgreifen (IBM-Kurve mit $FP = 26 * MM^{0,8}8$, VW AG Kurve, ...)
- alternativ kann man eigene abgeschlossene Projekte **nachkalkulieren**, allerdings:
  - Nachkalkulationen sind aufwändig
  - Dokumentation von Altprojekten oft unvollständig
  - oft gibt es nur noch den Quellcode (keine Lasten- oder Pflichtenhefte)
  - Kosten (Personenmonate) alter Projekte oft unklar (wurden Überstunden berücksichtigt, welche Aktivitäten wurden mitgezählt, ...)
- das Verhältnis von MM zu FP bei abgeschlossenen eigenen Projekten wird zur nachträglichen "**Kalibrierung**" der Kurve benutzt:
  - neues Wertepaar wird hinzugefügt oder neues Wertepaar ersetzt ältestes Wertepaar
  - Frage: was für eine Funktion benutzt man für Interpolation von Zwischenwerten (meist nicht linear, sondern eher quadratisch oder gar exponentiell)

Nachkalkulation von Projekten mit "Backfiring"-Methode: Bei alten Projekten gibt es oft nur noch den Quellcode und keine Lasten- oder Pflichtenhefte, aus denen FPs errechnet werden können. In solchen Fällen versucht man FPs aus Quellcode rückzurechnen.
Achtung: LOC in Programmiersprache X pro MM Codierung angeblich nahezu konstant -> damit ist z.B. Produktivität beim Codieren in Smalltalk 4 mal höher als in C

Vorgehensweise bei Kostenschätzung (mit FP-Methode):

1. Festlegung des verwendeten **Ansatzes**, Bereitstellung von Unterlagen
2. **Systemgrenzen** festlegen (was gehört zum Softwaresystem dazu)
3. **Umfang** des zu erstellenden Softwaresystems (in FPs) **messen**
4. Umrechnung von Umfang (FPs) in **Aufwand** (MM) mit Aufwandskurve

5. **Zuschläge** einplanen (für unvorhergesehene Dinge, Schätzungenauigkeit)
6. Aufwand auf Phasen bzw. Iterationen **verteilen**
7. Umsetzung in **Projektplan** mit Festlegung von **Teamgröße**
8. Aufwandschätzung **prüfen** und dokumentieren
9. Aufwandschätzung für Projekt während Laufzeit regelmäßig **aktualisieren**
10. Datenbasis für eingesetztes Schätzverfahren aktualisieren, Verfahren **verbessern**

Einplanung von Zuschlägen (Faustformel nach Augustin): Korrekturfaktor $K = 1,8/(1+0,8F^3)$ für Zuschläge mit F als geschätzter Fertigstellungsgrad der Software.

Problem mit der Berechnung des Fertigstellungsgrades der Software: Der Wert F ist vor Projektende unbekannt, muss also selbst geschätzt werden als **F = bisheriger Aufwand / (bisheriger Aufwand + geschätzter Restaufwand)**

Modifizierte Formel für korrigierte Aufwandsschätzung: $MM_g = MM_e + MM_k = MM_e + MM_r * 1,8/(1 + 0,8(MM_e/(MM_e + MM_r))^3)$

MM_g = korrigierter geschätzter Gesamtaufwand in Mitarbeitermonaten
MM_e = bisher erbrachter Aufwand in Mitarbeitermonaten
MM_k = korrigierter geschätzter Restaufwand in Mitarbeitermonaten
MM_r = geschätzter Restaufwand in Mitarbeitermonaten

Erläuterungen zu Korrekturfaktor für Kostenschätzung:

- zu **Projektbeginn** ist F = 0, da noch kein Aufwand erbracht wurde; damit wird der geschätzte Aufwand um 80% nach oben korrigiert
- am **Projektende** ist F = 1, da spätestens dann die aktuelle Schätzung mit tatsächlichem Wert übereinstimmen sollte, es gibt also keinen Aufschlag mehr
- Unsicherheiten in der Schätzung nehmen nicht **nicht linear** ab, da Wissenszuwachs über zu realisierende Softwarefunktionalität und technische Schwierigkeiten im Projektverlauf keinesfalls linear ist
- im Laufe des Projektes wird Fertigstellungsgrad F nicht immer zunehmen, sondern ggf. auch abnehmen, wenn Schätzungen sich als **zu optimistisch** erwiesen haben

- Auftraggeber wird mit Zuschlag von 80% auf geschätzte Kosten nicht zufrieden sein, deshalb werden inzwischen manchmal Verträge geschlossen, bei denen nur **Preis je realisiertem FP** vereinbart wird:
  - Risiko für zu niedrige Schätzung von FPs liegt bei Auftraggeber
  - Risiko für zu niedrige Umrechnung v. FPs in MM liegt bei Auftragnehmer

Aufwandsverteilung auf Phasen bzw. Entwicklungsaktivitäten: Hat man den Gesamtaufwand für ein Softwareentwicklungsprojekt geschätzt, muss man selbst bei einer ersten Grobplanung schon die ungefähre **Länge einzelner Phasen** oder Iterationen festlegen:

- für die Aufteilung des Aufwandes auf Phasen bzw. Aktivitätsbereiche gibt es die **Prozentsatzmethode**, hier in der Hewlett-Packard-Variante aus [Ba98]:
  - Analyseaktivitäten: 18% des Gesamtaufwandes
  - Entwurfsaktivitäten: 19% des Gesamtaufwandes
  - Codierungsaktivitäten: 34% des Gesamtaufwandes
  - Testaktivitäten: 29% des Gesamtaufwandes
- für die Aufwandsberechnung **einzelner Iterationen** einer Phase wird die Zuordnung von FPs zu diesen Iterationen herangezogen oder es wird bei festgelegter Projektlänge und fester Länge von Iterationen (z.B. 4 Wochen) die Anzahl der FPs, die in einer Iteration zu behandeln sind, festgelegt

Bestimmung optimaler Entwicklungsdauer (Faustformel nach Jones): für geringen Kommunikationsoverhead und hohen Parallelisierungsgrad:
$Dauer = 2,5 * (AufwandinMM)^S$ mit
s = 0,38 für Stapel-Systeme
s = 0,35 für Online-Systeme
s = 0,32 für Echtzeit-Systeme
durchschnittliche **Teamgröße = Aufwand/Dauer**

Überlegungen zu obiger Formel:

- Anzahl der maximal sinnvoll parallel arbeitenden Mitarbeiter hängt ab von Projektart
- große Projekte dürfen nicht endlos lange laufen (also mehr Mitarbeiter)
- mit der Anzahl der Mitarbeiter wächst aber der Kommunikations- und der Verwaltungsaufwand überproportional (also weniger Mitarbeiter)
- Anzahl sinnvoll parallel zu beschäftigender Mitarbeiter während Projektlaufzeit (Putnam-Kurve)

Bewertung der FP-Methode:

- lange Zeit wurde LOC-basierte Vorgehensweise propagiert
- inzwischen: FP-Methode ist wohl einziges halbwegs funktionierendes Schätzverfahren
- Abweichungen trotzdem groß (insbesondere bei Einsatz "fremder" Kurven)
- Anpassung an OO-Vorgehensmodelle, moderne Benutzeroberflächen notwendig
- moderne Varianten in Form von "**Object-Point-Methode**", ... sind noch nicht standardisiert und haben sich wohl noch nicht durchgesetzt
- Schätzungsfehler in der Machbarkeitsstudie sind nicht immer auf fehlerhafte Schätzmethode zurückzuführen, sondern ggf. auch auf **nicht** im Lastenheft **vereinbarte** aber **realisierte Funktionen** oder zusätzliche Umbaumaßnahmen
- bisher geschilderte Vorgehensweise **nur für Neuentwicklungen** geeignet (ohne umfangreiche Umbaumaßnahmen im Zuge iterativer Vorgehensweise)

Problematik der FP-Berechnung bei iterativer Vorgehensweise: Bei Projekten zur **Sanierung oder Erweiterung** von Softwaresystemen bzw. bei einer stark iterativ geprägten Vorgehensweise (mit Umbaumaßnahmen) werden einem System nicht nur Funktionen hinzugefügt, sondern auch Funktionen verändert bzw. entfernt. Damit ergibt sich der Aufwand für Projektdurchführung aus:
Aufwand in MM = Aufwand für hinzugefügte Funktionen + Aufwand für gelöschte Funktionen + Aufwand für geänderte Funktionen

Vorgehensweise:

- man benötigt modifizierte Regeln für die Berechnung von FPs für **gelöschte** Funktionen (Löschen etwas einfacher als Hinzufügen, deshalb weniger FPs?)

man benötigt modifizierte Regeln für die Berechnung von FPs für **geänderte** Funktionen (Ändern = Löschen + Hinzufügen?)