Software Engineering - Zusammenfassung

1. Einleitung

- Der Begriff Software-Engineering: Betrifft die Konstruktion von großen Programm-Systemen mit dem Fokus auf Komplexitätsbewältigung. Die regelmäßige Kooperation von Menschen ist ein weiterer integraler Bestandteil des SE. Einzuordnen ist SE in die praktische und angewandte Informatik mit starkem Anwendungsbezug.
- Besonderheiten: Software ist ein immaterielles Gut, allgemein anerkannte und bewährte Abstraktionen und Visualisierungen existieren noch nicht (erster Ansatz: UML). Projektkosten bestehen größtenteils aus Personalkosten, und die Qualität des Personals ist ein entscheidender Erfolgsfaktor.
- Benötigte Fähigkeiten eines Software Engineers: Kommunikation auf verschiedenen Abstraktionsebenen mit Personen mit unterschiedlichen Zielsetzungen/Vorstellungen, Erstellung und Verwendung von Modellen und Methoden, Arbeitsplanung und -koordination
- Software bestimmt einen Großteil der Entwicklungskosten von Fahrzeugen und Steuergeräten, 90% aller Innovationen sind getrieben von Elektronik/Software → Innovationstreiber.
- Entwicklung zum industriellen SE:
  60er Jahre: Spezialrechner mit Spezialsoftware 70er Jahre: Mikroelektronik
  80er Jahre: Software-Massenmarkt 90er Jahre: Komponenten-Markt, Migration
  2000++: Individuell anpassbare Massenprodukte basierend auf einer einheitlichen Plattform
- Phänomen 1: Qualität, Termin und Budget nicht gleichzeitig realisierbar, deshalb Konzentration auf 2 Kriterien
- Phänomen 2: Je größer das Projekt, desto größer die Abbrecherquote
- Phänomen 3: gestiegene SW-Kosten, SW-Entwicklung und Instandhaltung machen heutzutage einen Großteil der Kosten gegenüber Hardware-Kosten aus
- Gründe für die Phänomene: Viele Missverständnisse, häufig Projekte in neuen Branchen, relativ neue Wissenschaft (ca. 30 Jahre) → aber SE gewinnt immer mehr an Bedeutung
- 4-Sichten-Modell auf die Software-Entwicklung
  - **Produktionssicht** (P-Sicht)
    SE ähnelt der Entwicklung von Massengütern (Software Factory): Entwurf ist der kreative Anteil, anschließend beginnt der Produktionsprozess (welcher strikt arbeitsteilig organisiert wird), Kosten werden durch Produktionsmittel und Personalkosten bestimmt, Anpassung der Produkte nur über fest definierte Parameter

+ Spezialisten für einzelne Tätigkeiten + Trennung von Produktentwicklung und -produktion + klare Fortschrittskontrollen + Management-Beruhigung + Anwendungswissen wird nur zu Beginn benötigt	- Wissenstransfer über Phasengrenzen hinweg ist unvollständig - Fehlendes Anwendungswissen führt in späteren Phasen zu Fehlern

  - **Ingenieurssicht** (I-Sicht)
    SE ähnelt dem Bau von Brücken: Weniger strikte Trennung von Planung und Produktion (auch während des Baus noch (stark eingeschränkte) Entwurfsanpassungen möglich), Verwendung von allgemein akzeptierten Prinzipien, so gut wie keine Anpassungsmöglichkeiten nach Fertigstellung des Entwurfs

+ Handhabbarkeit + Eingeschränkte Flexibilität + Vorhersehbarkeit von Kosten	- Abstraktionen fehlen - Verführung zur beliebigen Änderung auf Grund der weichen Natur von Software - nicht geeignet für Massenproduktion

  - **Vertrags-/Juristensicht** (J-Sicht)
    SE ist so handzuhaben, dass sie auf Grund von Verträgen kontrollierbar ist: Aufteilung in klar überprüfbare Teilaufgaben, Verzug muss bewertbar sein und Konsequenzen nach sich ziehen, späte Anforderungen werden ausgeklammert

+ Juristische Eindeutigkeit	- Entspricht nicht dem Prozess des Erkenntnisgewinns während der Entwicklung - Produkt muss vor Vertragsvergabe vollständig geplant sein - führt nicht zu nützlicher Software

- **Kreativ-/Kommunikationssicht** (K-Sicht)
  Software als gemeinschaftliches Kunstwerk: keine Trennung zwischen Planung und Produktion, Kreativität muss in allen Phasen berücksichtigt werden, Kommunikation soll so wenig wie möglich reglementiert werden und muss beliebig breit möglich sein

+ Kommunikation verringert die Wahrscheinlichkeit von Missverständnissen **+** jeder hat die Chance, kreative Lösungen zu jeder Zeit einzubringen	**-** Ergebnis ist unvorhersehbar **-** völlige Unplanbarkeit **-** Unvorhersehbarkeit des Endes **-** Unwirtschaftlich **-** Produkte sind fast nicht wartbar

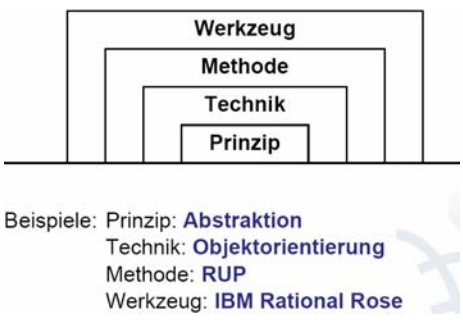
- Schlussfolgerung: Berücksichtigung von Aspekten aus allen vier Sichten; Festlegung welche Sicht welchen Stellenwert besitzt und welche Maßnahmen erforderlich sind, um diese Sicht im Projekt ausreichend zu berücksichtigen
  - Verträge müssen die Eckdaten (z.B. Kosten für die nächsten Phasen) festlegen, sie müssen Termine und Formate von Ergebnisdokumenten sowie die Mitwirkungspflicht des Anwenders definieren (J-Sicht, Juristen)
  - Grundlegende Entwurfsentscheidungen (spartenübergreifende Anwendung oder Spezialisierung) müssen zwischen allen Beteiligten abgestimmt werden (K-Sicht, Workshops und informale Kommunikationskanäle)
  - das Produkt muss ingenieurmäßig entworfen werden, z.B. kann nach Festlegung von Objekttypen die Datenbankzugriffsschicht an ein DB-Team weitergeleitet werden (I-Sicht, Software-Ingenieure und Domänenexperten)
  - Software muss anhand von Vorgaben stur programmiert und systematisch getestet werden (P-Sicht, Produktmanager und Marktstrategie)
- Rollenbasierte Software-Entwicklung: Monokausale Begründungen für die Probleme bei der Entwicklung von Software reichen nicht aus, deshalb Unterstützung der Einzeltätigkeiten durch passende Methoden, Werkzeuge und Sprachen unter Berücksichtigung von Organisation, Technologie, Psychologie und des Kontextes, in dem Einzeltätigkeiten stattfinden.
- Entwicklung von Software durch enge Interaktion von Personen mit unterschiedlichen Aufgaben und Zielsetzungen, Beziehungen zwischen diesen Personen und die Wechselwirkungen zwischen den Aufgaben sind vielfältig und tragen maßgeblich zum Erfolg/Misserfolg einer Software Entwicklung bei. Zielkonflikte zwischen den Aufgaben beteiligter Personen (Projektkaufmann ↔ Projektleiter, Projektleiter ↔ Software-Entwickler, QS-Verantwortlicher ↔ Projektkaufmann)
- Grundannahme: Projekte funktionieren, wenn die richtigen Leute unter Verwendung der geeigneten Methoden, Sprachen und Werkzeuge vernünftig miteinander arbeiten. Die Sicherstellung dieser Rahmenbedingung ist eine wesentliche Aufgabe in der Software Entwicklung.
- Rolle: beschreibt eine Menge von zusammengehörigen Aufgaben und Befugnissen/Qualifikationen, eine Rolle wird von Personen wahrgenommen (n-m-Beziehung), es treten in einer Software Entwicklung nicht immer alle Rollen auf

2. Prinzipien

- Eigenschaften von Software
  - externe Eigenschaften
    im Fokus des Systemnutzers, sichtbar für den Nutzer des Systems
  - interne Eigenschaften
    verantwortlich für das Erreichen externer Eigenschaften
  - Prozesseigenschaften
    bestimmen die Qualität der Produkteigenschaften
  - Produkteigenschaften
    häufig Beseitigung schlechter Qualität durch Nachbesserung, besser wäre Ursache für schlechte Qualität herauszufinden und entsprechende Prozessverbesserungen durchzuführen
  - Korrektheit
    Übereinstimmung des Programms mit seiner funktionalen Spezifikation (wenn diese nicht vorhanden ist, dann Rückgriff auf andere Methoden wie Testen oder Reviews)

- Zuverlässigkeit
  dauerhafte Einsetzbarkeit; relatives Kriterium, da Einsatzzweck und Schadenspotential des Fehlers Einfluss auf Zuverlässigkeit haben; mangelnde Zuverlässigkeit wird oft noch toleriert
- Robustheit
  Toleranz gegenüber nicht spezifizierten Eingaben und Rahmenbedingungen (auch nicht robuste Software kann korrekt sein; nicht robuste Software kann leicht nutzlos werden, deshalb sollten wesentliche Rahmenbedingungen und Fehlerbehandlungen spezifiziert werden)
- Performanz
  Erfüllung der Anforderungen an Antwortzeitverhalten und ökonomischer Umgang mir Ressourcen (frühzeitiges Testen an Prototypen verhindert später komplettes Redesign)
- Benutzungsfreundlichkeit
  intuitive, fehlerrobuste Bedienung der Software (Gestaltung der Dialoge, einfache Konfigurierbarkeit des Interfaces, intuitiver Aufbau und Ablauf, einheitliches Look&Feel)
- Wartbarkeit
  Fähigkeit einer Software nach ihrer Auslieferung Anpassungen und Änderungen zu ermöglichen), hängt stark von der Strukturierung (Architektur) der Software ab, nimmt meist mit der Lebensdauer der Software ab
  - Korrektive Wartung: Beseitigung von Fehlern
  - Adaptive Wartung: nachträgliche Anpassung an geänderte Rahmenbedingungen (gesetzliche Änderungen, neue Organisation)
  - Perfektive Wartung: Verbesserung von nicht-funktionalen Anforderungen (Performanz, Ergonomie)
  - Enhansive Wartung: Hinzufügen von Funktionalität
- Wiederverwendbarkeit
  Einsatzfähigkeit der Software in einem anderen Kontext (oft komponentenbezogen), muss geplant werden (parametrisierbare Datenstrukturen, Klassenbibliotheken, Plattformen, Produktlinien)
  - Software-Produktlinien: Im Entwicklungsprozess werden die Kernfunktionalität und die Variabilität festgelegt, im Produktionsprozess werden die Kernfunktionalitäten dann wiederverwendet
- Portierbarkeit
  Aufwand (im Verhältnis zum Entwicklungsaufwand), der nötig ist, um eine Software auf einer anderen Plattform lauffähig zu machen; hoher Grad an Portierbarkeit durch Kapselung von Plattformabhängigkeiten sowie durch Berücksichtigung von Standards (z.B. Design Patterns)
- Interoperabilität
  Maß für die Fähigkeit eines Systems, mit anderen Systemen zu kooperieren (PC-Werkzeuge sind meist interoperabel, hochintegrierte Software-Systeme sind es meist weniger)
  - Plattform Integration: transparente Verteilung der Software auf verschiedene Systeme
  - Presentation Integration: einheitliches Look&Feel
  - Data Integration: XML als Datenaustauschformat
  - Control Integration: Benachrichtigung eines Werkzeugs über externe Ereignisse
  - Process Integration: Werkzeuge verhalten sich dem Prozess entsprechend (instanziieren ihn)

- Prinzipien



Werkzeuge: Rechnerunterstützung für Techniken und Methoden (z.B. SQS)

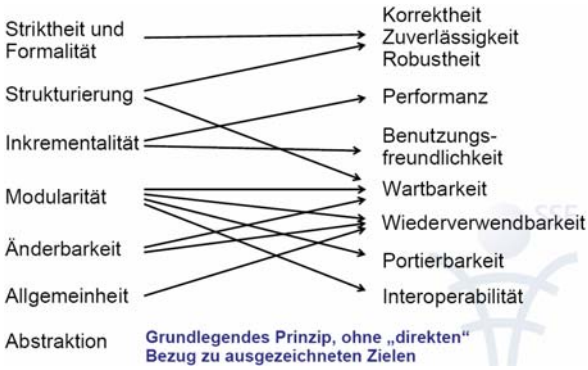
Methoden: planmäßige, begründete Bündelung von Techniken zur Erreichung vorgegebener Ziele - was ist wie unter welchen Rahmenbedingungen zu tun? Methoden beruhen auf einem oder mehreren Prinzipien (z.B. die Verwendung von Kontrollstrukuren zur Erfüllung des Prinzips der Strukturierten Programmierung)

Techniken: Vorschrift zur Durchführung einer Tätigkeit - was ist wie zu tun? (z.B. Backtracking innerhalb der Methode JSP)

Prinzipien: Grundsätze, die man seinem Handeln zugrundelegt (allgemeingültig und sehr abstrakt)

- Striktheit / Formalität
  - strikte Dokumentation des Software-Prozesses und seiner Ergebnisse
  - höchster Grad sind formale Beschreibungen (kritisch für das Verständnis, im Hinblick auf spätere Änderbarkeit und im Hinblick auf das Schadenspotential von Missverständnissen)
- Strukturierung
  - Unterteilung in Aspekte (aber Aspekte können auch untereinander abhängig sein), z.B. Performanz & Datenintegrität, Funktionalität & Zielplattform, Robustheit
  - Strukturierungsarten: zeitlich (Analyse, Entwurf, Implementierung, Test), qualitativ (Effizienz, Robustheit, Korrektheit), perspektivisch (Datenfluss, Kontrollfluss), Dekomposition
- Modularität
  - Unterteilung eines komplexen Systems in Komponenten als Ergebnis der Dekomposition
  - Voraussetzung für Wiederverwendung in kleinen Teilen und lokale Änderbarkeit
  - Ziele: hohe Kohäsion (enger Zusammenhalt) innerhalb eines Moduls, geringe Kopplung (wenig Wechselwirkungen) mit anderen Modulen
- Abstraktion
  - Trennung von wichtigen und unwichtigen Merkmalen zum Zweck der Konzentration auf das Wesentliche (z.B. abstrahiert ein Klassendiagramm von der gewählten Programmiersprache)
  - Wichtigkeit bzw. Unwichtigkeit sind relativ bzgl. des Zwecks der Abstraktion
- Änderbarkeit
  - Ursache für Wartungsintensität
  - Gewährleistung von konsistenten System trotz ständiger Änderungen
- Allgemeinheit
  - Einsatz von Software für verwandte Aufgaben und Zwecke → verursacht Aufwand
  - Anpassung an Rahmenbedingungen durch Parameter
- Inkrementalität
  - Gliederung einer Tätigkeit in Schritte, nach jedem Schritt Rückkopplung (Validierung)
  - basiert auf der Annahme, dass Änderungen nötig sind und Aufwand verursachen

- Prinzipien und Eigenschaften (Beispiele für positive Beeinflussung)

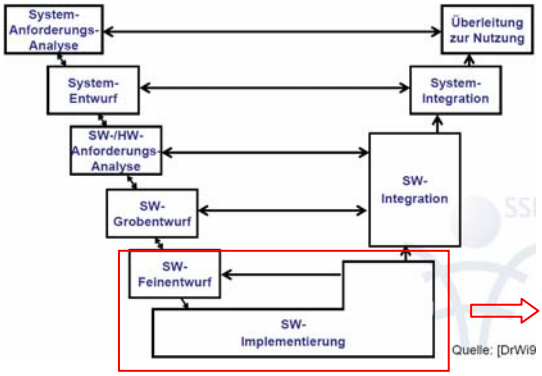


  - durch die Parametrisierung einer Funktion (Allgemeinheit) kann diese leichter wiederverwendet werden
  - eine gute Änderbarkeit / Strukturierung ermöglicht eine leichtere Wartung

3. Software-Prozesse

- Lebenszyklusmodell
  - Festlegung wesentlicher Entwicklungsaktivitäten und ihrer Abhängigkeiten
  - im Vordergrund steht die prinzipielle Vorgehensweise, weniger die formale Definition
  - ggfs. müssen während der Entwicklung Anpassungen am Lebenszyklusmodell vorgenommen werden
- Software-Prozessmodell
  - Definierung der Aktivitäten und ihrer Reihenfolge in einer Software Entwicklung

  - Werkzeuge und Methoden zur Unterstützung der Aktivitäten
  - Typen der Objekte, die erzeugt und verarbeitet werden
- Software-Prozess
  - Instanziierung eines Prozessmodells, entspricht der Durchführung einer Software Entwicklung - wer führt welche Aktivität durch, welche Objekte werden erzeugt bzw. verarbeitet
- Code and Fix Modell
  - Programm schreiben (code) und anschließend Fehler im Programm beheben (fix)
  - führt zu unstrukturiertem Code, keine Teamarbeit, Fehlerbehebung aufwändig und teuer
  - kein geeignetes Lebenszyklusmodell
- Wasserfallmodell
  - dokumentenzentriertes Vorgehen, jede Phase kennt nur die Ergebnisse der vorhergehenden Phase
  - an Bedürfnisse des Managements ausgerichtet, klare Ergebnisse nach Phasen, klare Verantwortungen für Phasen
  - Trennung der Phasen spiegelt aber nicht die Realität wieder: erkenntnisgetriebene Entwicklung, Entwicklung als Klärung von Anforderungen, Phasenüberlappungen
  - durch die fatale Überabstraktion nur als grobe Orientierungshilfe zu verwenden
- Spiralmodell
  - kein Lebenszyklusmodell im engeren Sinne, sondern ein Metamodell, das vorgibt, wie bestimmte Elemente konkreter Lebenszyklusmodelle verzahnt werden sollen
  - Ziel ist Risikominimierung
  - Iteration der folgenden vier Schritte:
    - Ziele, Randbedingungen und Alternativen identifizieren
    - Risiko-Management (Evaluation der Alternativen, Risiko-Einschätzung und -behandlung)
    - Entwicklung und Test von Teilergebnissen
    - Reviews der Resultate und Planung der nächsten Phase
  - Vorteil ist die Flexibilität (nicht ein Prozess-Modell für die ganze Entwicklung, Integration anderer Modelle als Spezialfälle), Fehler werden frühzeitig eliminiert
  - aber hoher Managementaufwand, eher für Großprojekte geeignet
- Prototyping Modell
  - Throw-away Prototypen: ausschließlich zur Sammlung von Erfahrungen und Klärung der Anforderungen, wird nicht zum Bestandteil des Produktes
  - Evolutionäre / Inkrementelle Prototypen: dient dem gleichen Zwecke wie der Throw-away Prototyp, wird aber zum Produkt weiterentwickelt; muss daher die Qualitätsanforderungen von vornherein berücksichtigen; Zielkonflikt: frühe Verfügbarkeit vs. Erweiterbarkeit
  - Horizontale Prototypen: realisiert nur spezifische Ebenen eines Systems, diese aber meist vollständig (z.B. Benutzungsschnittstellen, DB-Transaktionsebenen)
  - Vertikale Prototypen: implementiert ausgewählte Teile des Zielsystems vollständig durch alle Ebenen hindurch, dort geeignet, wo Funktionalitäts- und Implementierungsanforderungen noch offen sind
- V-Modell
  - gliedert sich in vier Submodelle auf:
    - Systemerstellung: Produkt entwickeln



Diese Aktivitäten werden zusehends ins Ausland verlagert.

        - Projektmanagement: Projekt planen und kontrollieren, Voraussetzungen und Software Entwicklungs-Umgebung bereitstellen
        - Qualitätssicherung: QS-Anforderungen vorgeben, Produkte prüfen
        - Konfigurationsmanagement: Produktstruktur planen, Produkte und Rechte verwalten
    - Probleme, die es zu vermeiden gilt: zu viele, nutzlose Dokumente, Fehlen wichtiger Dokumente
    - Lösung Tailoring: stellt sicher, dass der betriebene Aufwand dem Nutzen entspricht, Reduktion des generischen V-Modells auf die Elemente, die zur Durchführung des Projektes benötigt werden, Resultat: spezifisches V-Modell
    - Vorteile des V-Modells: Identifikation und Definition von Rollen und Verantwortungen, durchgängige Betrachtung der Qualitätssicherung, Integration von SE, KM, QS und PM
    - Nachteile: zu detailliert für ein Lebenszyklusmodell, zu unkonkret für ein Software-Prozessmodell, Art und Weise der Anwendung unklar, lange Trainingszeit
- Unified Process
    - Softwareprozess im Sinne der Entwickler der UML, Hauptaugenmerk auf Anforderungsmanagement, Analyse und Design
    - iterativ und inkrementell, dadurch Entschärfung von Risken und Handhabung sich ändernder Anforderungen
    - use-case-getrieben, um bedeutende Anforderungen zu identifizieren, eine einfache Erstellung und Validierung der Architektur zu ermöglichen und den Prozess voranzutreiben
    - Architekturzentriert, um Verständnis des Systems zu vertiefen, Entwicklung zu organisieren, Wiederverwendung zu fördern und die Weiterentwicklung des Systems zu unterstützen
    - besteht aus vier Phasen:
        - Inception (Konzeption): Festlegung des Geschäftsfelds und des Umfangs des Projekts sowie der zugehörigen Systemgrenzen
        - Elaboration (Entwurf): Planung der notwendigen Aktivitäten und Ressourcen sowie Spezifizierung der Funktionalität und Architektur
        - Construction (Konstruktion): Implementierung basierend auf der erstellten Architektur
        - Transition (Übergabe): Bereitstellung des Produkts für den Kunden sowie Konfiguration und Überprüfung der Einhaltung der Vorgaben durch Beta-Tests der Benutzer
    - diese Phasen werden iterativ (auch mehrmals) durchlaufen, wobei folgende Core Workflows in jeder Phase als Aktivitäten auftreten können
        - Requirements
        - Analysis
        - Design
        - Implementation
        - Test
    - eine konkrete Implementierung des Unified Process ist der Rational Unified Process
    - Vorteile: basiert auf der UML, Berücksichtigung der Wechselwirkung zwischen Anforderungen und Architektur, basiert auf Erfahrungen mehrerer Vorgehensweisen
    - Nachteile: insgesamt noch unausgereift und zu wenig erprobt, keine klaren Definitionen von Rollen, zu unkonkret für praktische Anwendungen

4. Rollenübersicht

- Projektmanager (PM-Bereich)
    - übergreifend über alle Phasen des Projektes
    - planerische Vorgaben an QS, SE und KM (Projektorganisation, Kostenschätzung, Erstellung eines Projektplans, Projektkoordination)
    - kontrolliert Einhaltung der Vorgaben (Produktverfolgung, Planüberprüfung, Produktivitätsüberwachung)
    - trifft wesentliche Entscheidungen, verantwortlich für eine erfolgreiche Projektdurchführung
- Risikomanager (zwischen allen Bereichen, oft Bestandteil des PM)
    - übergeordnete Rolle, Einschätzung von Risiken und Strategien zu deren Minimierung

- Qualitätsmanager (QS-Bereich)
  - projektbegleitend, verantwortlich für die Qualität des Produkts
  - Vorgaben und Überprüfung durch PM
  - Koordination der QS-Maßnahmen des Projekts, Erstellung eins überprüfbaren QS-Plans, Verabschiedung einer Qualitätspolitik und von Qualitätsrichtlinien, Inkraftsetzen von Verfahrensanweisungen
  - Sicherstellung (Prüfung der Übereinstimmung mit Anforderungen, Sicherstellen der Kundenzufriedenheit) und Überprüfung (Erfassung von Messdaten, Tests, Reviews und Walkthroughs) der Systemqualität
- Konfigurationsmanager (KM-Bereich)
  - projektbegleitende, übergreifende Rollen
  - Koordination der KM-Maßnahmen, verantwortlich für Produktstruktur, Versions- und Konfigurationsmanagement (Konfigurationsidentifikation, -kontrolle und Statusverfolgung)
  - Festlegung der Systemkonfiguration
  - Nachvollziehbarkeit und Konsistenz: Sicherstellung von Sichtbarkeit, Verfolgbarkeit und Kontrollierbarkeit eines Produktes und seiner Teile im Lebenszyklus
- Anforderungsanalytiker (SE-Bereich, Analyse und Entwurf)
  - Vorgaben von und Überprüfung durch PM, QS und KM
  - Ermitteln, Erkennen und Dokumentieren von Anforderungen an das System
  - Machbarkeitsstudie, Integration unterschiedlicher Sichten/Ziele
  - Erstellung eines Anforderungsdokuments
- Spezifizierer (SE-Bereich, Analyse und Entwurf)
  - Vorgaben von und Überprüfung durch PM, QS und KM
  - bekommt Ergebnisse des Anforderungsanalytikers
  - implementierungsnahe Spezifikation der Anforderungen an das Softwaresystem
  - Modellierung der Anforderungen aus Implementierungssicht (Kommunikation mit Designer und Programmierer)
  - Definition der Qualitätsanforderungen an das Softwaresystem
  - Vorbereitung der Qualitätssicherung: Dokumentation der Anwendungsfälle, Definition von Testfalldesigns
- Designer (SE-Bereich, Grob- und Feinentwurf)
  - Vorgaben von und Überprüfung durch PM, QS und KM
  - bekommt Ergebnisse des Anforderungsanalytikers
  - Vorbereitung der Implementierung, Entwurf der Architektur des Systems
  - Mapping von Anforderungen auf Komponenten/Subsysteme
  - Einhaltung von Qualitätsattributen
  - Vorbereitung der Systemintegration: Dokumentation der Systemarchitektur, Definition von Testfällen für den Integrationstest
- Programmierer/Entwickler (SE-Bereich, Implementierung)
  - Vorgaben von und Überprüfung durch PM, QS und KM
  - bekommt Ergebnisse des Designers und führt das Design aus
  - eigentliche Programmierung und Vorbereitung der Integration zum Gesamtsystem
  - Dokumentation des Programmcodes, Definition der modulspezifischen Algorithmen
  - Beachtung von Standards und Zusicherungen, Pre-Tests der Module gemäß der QS-Anforderungen
  - Vorbereitung des Modultests, Walkthroughs, Reviews
- Tester (SE- und QS-Bereich)
  - Vorgaben von und Überprüfung durch PM, QS und KM
  - bekommt u.a. die Ergebnisse des Programmierers
  - Durchführung der Modul-, System- und Integrationstests
  - Detaillierung von Testplan, -entwurf, -verfahrensspezifikation und -fällen
  - Auswahl von Testverfahren und Dokumentation der Testbedingungen und -ergebnisse

    - Überprüfung der Zusammenarbeit der Komponenten und des Gesamtsystems
- Systemtechniker (SE- und KM-Bereich, Analyse & Entwurf und Integration & Abnahme)
    - Mitarbeit von Projektanfang bis -ende, Systemintegration und Überleitung zur Nutzung
    - Bereitstellung der Systemkonfiguration, Ermittlung der Zielkonfiguration bei Projektstart, Beurteilung auf Machbarkeit
    - Integration des Systems in die Zielumgebung, Erstellung von System- und Integrationstests/-szenarien während der Anforderungsermittlung sowie projektbegleitend
    - Nutzbarmachung des Systems, Installation der Soft- und Hardware, Ermittlung des Schulungsbedarfs und ggfs. Durchführung der Schulungen
- Technologieberater (PM- und KM-Bereich, Anforderungsanalyse und Überleitung zur Nutzung)
    - Beurteilung des Einsatzes neuer Technologien aus wirtschaftlichen Aspekten (Beobachtung aktueller Entwicklungen, konsequente Verfolgung neuer Technologien)
    - welche Technologie wird in diesem Projekt benötigt, welche Technologie sollte in der Produktion langfristig eingesetzt werden?
    - präzise Unterscheidung von kurz-, mittel- und langfristigem return-on-investment
- Wartungsexperte (vergrößerter SE-Bereich)
    - Wartung und Pflege der Software nach Inbetriebnahme unter Berücksichtigung von QS, SE und KM
    - Rückmeldung von Fehlern an QS → Prozessverbesserung
    - Abwägung zwischen Wartung, Sanierung und Neuentwicklung
    - Kontrolle über Änderungen behalten: Änderungsmanagement, Release-Planung
- Datensammler und -bewerter (alle Bereiche)
    - projektbegleitende Unterstützung des Managements
    - projektübergreifende Messungen für Voraussage nächster Projekte
    - Kontrolle des Software-Prozesses durch Messung des Ist-Zustandes (verbrauchte Zeit/Budget/Auslastung, Erstellung fairer Metriken zur Bewertung von Produktivität und Qualität, Verdichtung der Metrikerkenntnisse und Kommunikation an PM und QS)
    - genauere Vorhersage zukünftiger Produktivität/Kosten
    - Schwachstellensuche zur Prozessverbesserung
- Software-Prozessverbesserer (alle Bereiche)
    - übergeordnete Rolle, Überblick über mehrere Projekte
    - Einschätzung (Beobachtung vergangener Projekte, Einschätzung der Qualität der beobachteten Prozesse) und Verbesserung des Softwareprozesses
- Wiederverwender (vergrößerter PM-Bereich)
    - projektübergreifende Tätigkeit
    - Reduktion von Entwicklungsaufwänden: Bereitstellung einer entsprechenden Infrastruktur, Incentive-Systems als organisatorische Maßnahme (Belohnung für die Bereitstellung, Wiederverwendung und Verbesserung von wiederverwendbaren Komponenten)
    - Qualitätsverbesserungen, Verbesserung des Entwicklungsprozesses
- Chief Executive Officer
    - Vorsitzender der Geschäftsführung
    - Ziele messen und ggfs. sanktionieren
- Chief Information Officer
    - oberster Informationstechnologieverantwortlicher
    - Sicherstellung der internen DV (Koordination der Entwicklungsinfrastruktur, Verfügbarkeit von Plattformen, strategische Ausrichtung der Entwicklungsplattform, Festlegung der Entwicklungswerkzeuge)
    - in Software-Häusern meist nicht vertreten, Rolle wird von Entwicklern und PL übernommen
    - in großen Unternehmen, die Software für die eigene Benutzung entwickeln, in aller Regel vertreten, zwischen den Abteilungen und dem CIO herrscht häufig ein Spannungsverhältnis
- Chief Financial Officer
    - kaufmännischer Leiter
    - Sicherstellung der Wirtschaftlichkeit der Software-Entwicklung auf Firmenebene

  - Sinnvolle Kostenverteilung der einzelnen Aktivitäten, Budgetanteile für die unvermeidlichen späten Anforderungen, für die Phase zwischen Auslieferung und Bezahlung und für Tests und Reparaturen
  - Kalkulation der externen Aufwände
- Projektmanagement Board (Verantwortlicher Projektmanager)
  - Zweck: gemeinschaftliches Feststellung des Projektfortschritts, Identifikation der aktuellen Risiken, Analyse potentieller Konflikte zwischen Rollen und Identifikation von Maßnahmen zu ihrer Überwindung
  - Teilnehmer: PM, QS, KM, teilweise SE-Verantwortlicher, Risikomanager, GF-Vorsitzender
- Change Management Board (Verantwortlicher Projektmanager)
  - Zweck: Priorisierung von Änderungsanforderungen, Zusammenfassung der Änderungsanforderungen zu Update-Einheiten
  - Teilnehmer: PM, Kundenbetreuer, Designer, KM, QS, teilweise Anforderungsmanager
- Risiko Management Board (Verantwortlicher Risiko-Manager)
  - Zweck: Kommunikation der Risiken in den einzelnen Projekten, Identifikation von Gegenmaßnahmen
  - Teilnehmer: PL, Risiko-Manager, teilweise GF-Vorsitzender, Kundenbetreuer
- Software Process Group (Verantwortlicher CIO, oder Qualitätsmanager)
  - Zweck: Validierung und Weiterentwicklung des vorgegebenen Software-Prozessmodells
  - Teilnehmer: PL, QM, Software-Prozessverbesserer, CIO
- Messgruppe (Verantwortlicher Datensammler und -bewerter)
  - Zweck: Austausch zwischen den einzelnen Messprogrammen, Entwicklung einer Strategie für ein integriertes Messprogramm
  - Teilnehmer: alle am Aufsetzen von Messprogrammen beteiligte Personen, QM

5. Konfigurationsmanager

- Probleme:
  - Produktevolution: es werden eine Vielzahl von Versionen erstellt und getrennt weiterentwickelt
  - Produktvarianten: es werden verschiedene Varianten parallel weiterentwickelt
  - Gruppenunterstützung: die Vergabe und Verwaltung von Zugriffsrechten sowie die Zugriffskontrolle sind Aufgaben des KM
- Ziele
  - Sicherstellung der Sichtbarkeit, Verfolgbarkeit und Kontrollierbarkeit eines Produktes und seiner Elemente im Lebenszyklus
  - Überwachung der Konfigurationen, so dass Zusammenhänge und Unterschiede zwischen früheren Konfigurationen und den aktuellen jederzeit erkennbar sind
  - Sicherstellung, dass jederzeit auf vorangegangene Versionen zurückgegriffen werden kann, damit Änderungen nachvollziehbar und überprüfbar sind
  - Unterstützung der Teamarbeit an einer bestimmten Konfiguration, um die Koordination von Tätigkeiten an identischen Objekten zu gewährleisten
- Begriffe
  - Revision: Version, die ihren Vorgänger ersetzt (Evolution über die Zeit)
  - Variante: Version, die gleichzeitig mit alternativen Versionen existiert
  - Kooperierende Version: eine von vielen parallel existierenden Versionen, die das gleichzeitige Arbeiten ermöglichen sollen
  - Release: eine Version, die an einen Kunden ausgeliefert wird
  - Invariante: Eigenschaft, die von allen Versionen eines Gegenstandes geteilt wird
  - Externe Sicht: nur eine Version ist sichtbar, um die Auswahl einer Version für den Anwender zu vereinfachen
  - Interne Sicht: auch Komponenten werden versioniert
- Funktionen des KM:

- Identifikation: eindeutige Benennung der zu verwaltenden Softwareobjekte
- Herstellung: Verwaltung der Entwicklungsartefakte während der Produkterstellung
- Kontrolle: Überwachung des Produkts und dessen Änderungen während der Entwicklung
- Teamwork: Koordination der Arbeit zwischen mehreren Anwendern
- Statuserfassung: Erfassung und Dokumentation der Stati von Komponenten und Änderungsanfragen
- Prozessmanagement: Sicherstellung der Vorgehensweisen der Firma, der Vorschriften und des Lebenszyklusmodells

- Zugriffskontrolle auf Konfigurationseinheiten, wird typischerweise durch Check-Out/-In realisiert, besteht aus Versionsbibliothek (Repository), Arbeitsbereichen (Workspaces) und versionierten Objekte (Konfigurationseinheiten)
  - pessimistische Zugriffskontrolle: beim Check-Out wird Konfigurationseinheit gesperrt, eine Einheit kann nur einmal gleichzeitig ausgecheckt werden, ausgecheckte Einheiten können von anderen Projektmitarbeitern nicht gleichzeitig bearbeitet werden
  - optimistische Zugriffskontrolle: beim Check-Out wird Konfigurationseinheit nicht gesperrt, gleichzeitiges Bearbeiten möglich, Zusammenschmelzen (Merge) der unterschiedlichen Änderungen wird erforderlich
- Speicherung von Revisionen/Varianten: Nutzung der Deltatechnik um Speicherplatzbedarf und Identifikationsprobleme zu minimieren
  - Vorwärtsdeltatechnik: Differenz zum Vorgänger, geringer Speicherbedarf, Berechnungsaufwand erforderlich, Rekonstruktionsaufwand erforderlich
  - Rückwartsdeltatechnik: Speicherung des aktuellen Dokuments als vollständiges Dokument, Differenz zum Vorgänger, Rekonstruktionsaufwand oft geringer als bei Vorwärtsdeltatechnik
  - Überlappende Speicherung: jede Zeile der Rekonstruktionsdatei wird gekennzeichnet, in welcher Version sie vorkommt → Gesamtgröße der Datei steigt bei jedem Durchlauf
- Anforderungen an Werkzeuge für das KM:
  - Zugriffskontrolle (Check-In/Check-Out)
  - Effektive Speicherung von Versionen (Delta-Ermittlung)
  - folgende Fragen müssen beantwortet werden können:
    - Welcher Kunde hat eine spezielle Version des Produkts erhalten?
    - Welche Hardware- und Software-Konfigurationen werden für eine geplante Produktversion benötigt?
    - Wie viele Versionen des Produktes wurden wann und von wem erzeugt?
    - Welche Versionen sind von der Änderung einer Konfigurationseinheit betroffen?

6. Software-Architekt

- Problem: Da ein Software-System aus mehreren Millionen Zeilen Quellcode bestehen kann, ist eine gute, angemessene Strukturierung wichtig. Es gibt gute Gründe, nicht den gesamten Code in eine Datei zu schreiben:
  - technisch: selbst kleine Änderungen erfordern eine komplette Re-Compilation
  - konzeptionell: die Beherrschung der Komplexität erfordert Strukturierung in übersichtliche Einheiten
- Änderbarkeit, Weiterentwicklungsfähigkeit und Portierbarkeit hängen direkt von der Qualität der Softwarearchitektur ab
- durch die Trennung von Eigenschaften (Prinzip der Strukturierung) erhält man verschiedene Sichten auf die Softwarearchitektur (Komponenten, Konfigurationen, Verhalten, Daten, etc.) und Regeln, die bestimmen, wie Strukturelemente des Softwaresystems gebildet werden und wie diese miteinander interagieren können
- Ziel: Softwarearchitektur als Vermittler zwischen Analyse, Entwurf und Implementierung
- formale Schnittstellen als Schlüsselrollen
- eine Komponente bildet eine abgeschlossene Einheit, in der die Leistungen Ex- und Import explizit definiert werden

- Je nach Zweck der Softwarearchitektur werden unterschiedliche Sichten (Funktions-, Datensicht, Code, Nebenläufigkeit) eingeführt. Sichten können separat entwickelt werden, müssen aber zu bestimmten Zeitpunkten untereinander konsistent sein.
- Softwarearchitektur durch Design by Contract
  - Funktionssicht: jede Komponente erfüllt einen (!) bestimmten, eng umgrenzten Zweck, Verbindungen zwischen Komponenten werden durch eine Art Vertrag charakterisiert
  - Kontrakte basieren auf Leistungen, die zwischen zwei Akteuren vereinbart werden
  - die Spezifikation einer Leistung besteht aus der Definition der exportierten Leistungen, den Erwartungen an importierte Leistungen und den inneren Werten, die diese Leistungen implementieren
  - Invarianten beschreiben innere Werte, die vor und nach jedem Programmschritt gelten
  - Vor- und Nachbedingungen sind gegenseitige Garantien
    - was die Liste als Vorbedingung erwartet, muss der Nutzer garantieren
    - der Nutzer darf das erwarten, was die Liste als Nachbedingung und Invariante garantiert
  - Kontrakte zwischen zwei Klassen werden auf deren Subklassen mit folgenden Einschränkungen vererbt:
    - alle Invarianten der Oberklasse müssen in der Unterklasse erfüllt sein
    - eine Vorbedingung einer redefinierten Methode darf die ursprüngliche Vorbedingung nicht verschärfen
    - eine Nachbedingung einer redefinierten Methode darf die ursprüngliche Nachbedingung nicht aufweichen

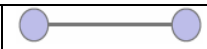
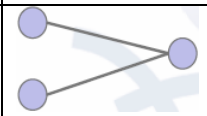
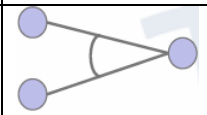
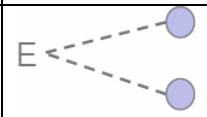
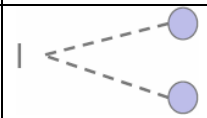
7. Tester

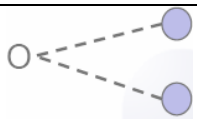
- Als Testen bezeichnen wir jede einzelne Ausführung des Testobjekts unter spezifizierten Bedingungen zum Zwecke des Überprüfens der beobachteten Ergebnisse im Hinblick auf gewissen gewünschte Eigenschaften. Testen ist eine Kontrollfunktion und dient der Aufdeckung von Fehlerwirkungen. Testen umfasst nicht die Fehlerkorrektur (→ Debuggen).
- Gründe zum Testen:
  - Kostenreduktion: je später ein Fehler erkannt wird, umso teurer die Beseitigung
  - Fehler führen zur Kundenunzufriedenheit
  - bestimmte Produkte benötigen eine bestimmte Qualität
  - entdeckte Fehler ermöglichen eine Analyse und somit eine Prozessverbesserung
- Ziel des Testens ist es, Fehler zu entdecken. Somit ist ein Test erfolgreich, wenn mindestens ein Fehler entdeckt wurde.
- Zur Fehlererkennung benötigt man Soll-Ergebnisse: ein Test-Orakel. Das Test-Orakel darf nicht das Programm sein, was getestet wird, und sollte idealerweise generiert werden (sind jedoch typischerweise Menschen). Aber Vorsicht: Auch Sollvorgaben können Fehler haben!
- Klassifikation von Tests:
  - Laufversuch: Entwickler übersetzt das Programm und versucht es zu starten, irgendwann ist es ausführbar → Ergebnisse sind nicht offensichtlich falsch
  - Wegwerf-Test: das Programm wird ausgeführt und Daten werden eingegeben, in einigen Fällen werden Fehler erkannt → Problem: Programm-Änderung (Regressionstest), Nachvollziehbarkeit
  - Systematischer Test: Ableitung von Testfällen aus der Spezifikation, Vergleich von Ist- und Soll-Resultaten, Analyse der Resultate, Dokumentation der Testdaten und der Ist-Situation
- drei Kategorien von Fehlern (allgemein Abweichung des tatsächlichen Zustands zu einem erwarteten Soll-Zustand):
  - wrong: falsche Umsetzung eines Aspekts der Spezifikation
  - missing: Aspekt der Spezifikation nicht umgesetzt
  - extra: Aspekt in der Spezifikation nicht beschrieben
- Vorteile von Tests: objektives Prüfverfahren, i.d.R. reproduzierbar, Systemreaktion bzw. -verhalten wird sichtbar gemacht

- Nachteile: Ergebnisse werden oft überinterpretiert, nicht alle Eigenschaften von Software und nicht alle Anwendungssituationen sind testbar, keine Erkennung der Fehlerursachen
- Testen erfordert Unabhängigkeit, Entwickler sollten ihre Programme nicht selbst testen
- vollständiges Testen ist nicht möglich, Nichtexistenz von Fehlern ist nicht nachweisbar
- ein Test ist nur so gut wie die Testfälle: Tests müssen geplant werden, Definierung von gültigen und ungültigen Eingaben, zu jedem Testfall gehört ein Soll-Ergebnis, Vermeidung von Wegwerf-Testfällen
- Teststrategien:
  - Top-Down: erst Test des Gesamtsystems, dann konkretere Komponenten
  - Bottom-Up: erst atomare Komponenten, anschließend werden zusammengesetzte Komponenten getestet, zum Schluss das Gesamtsystem
  - Thread: nur für Systeme mit mehreren parallelen Prozessen
  - Stress: Test des Systems unter Extrembelastung (z.B. Funktionstest unter 100 % CPU-Auslastung)
  - Back-to-Back: Test von unterschiedlichen Systemversionen sowie Vergleich der Ergebnisse
- Testarten
  - Modultest: isoliertes Testen einzelner Module/Komponenten; benötigt oft zusätzliche Software (Testumgebung); kann schon früh (parallel zur Entwicklung) durchgeführt werden; gewährleistet, dass beim Integrationstest die Module korrekt sind
  - Integrationstest: testet das Zusammenspiel von Komponenten; stellt sicher, dass die Komponenten zusammenpassen und die gewünschte Funktionalität und Qualität erfüllen; wird i.d.R. stufenweise durchgeführt
  - Systemtest: Testen der Systemfunktionalität, -benutzbarkeit und -qualität; Integration mit externen Systemen
  - Akzeptanztest (alpha test): testet, ob der Kunde das System akzeptiert; dabei werden oft Missverständnisse bzw. in der Spezifikation fehlende Anforderungen aufgedeckt
  - Installationstest: testet das System in der Umgebung des Kunden; prüft korrekte Interaktion mit anderen System
  - Abnahmetest (beta test): dient meist zur Endabnahme des Systems vom Kunden; nach erfolgreichem Abnahmetest gilt das Projekt als abgeschlossen
- Spezifikationsbasierte Technikarten (Black-Box-Test)
  - Ausgangspunkt für Testfälle ist die Spezifikation, fehlt diese, stellt sich die Frage nach den Sollergebnissen
  - Ziel und Vorteil ist die möglichst umfassende Prüfung der spezifizierten Funktionalität/Qualität
  - Nachteile: die konkrete Implementierung wird nicht geeignet berücksichtigt, häufig wird durch die Ausführung eines reinen Black-Box-Test nicht die Minimalanforderung von White-Box-Tests erzielt (z.B. 100 % Zweigüberdeckung)
  - funktionale Äquivalenzklassenbildung
    - Spezifikation des Testobjekts durch Abbildungsfunktion, die Eingabedaten aus der Menge aller möglichen Eingaben in Ausgabedaten aus der Menge aller möglichen Ausgaben abbildet
    - es ist unmöglich, sämtliche Kombinationen der Eingabewerte zu testen; Lösung: Unterteilung der Eingaben in Gruppen ähnlicher Werte → Äquivalenzklassen
    - Definition Äquivalenzklasse: Teilmenge von Elementen aus einer Gesamtmenge, die untereinander in einer Äquivalenzrelation stehen
    - sämtliche Eingaben einer ÄK führen zu dem selben Testergebnis → Test eines Repräsentanten einer ÄK reicht aus
    - Zerlegung aller möglichen Eingabedaten in gültige und ungültige (erfordert Fehlerbehandlung durch das Prüfobjekt) ÄKs
    - Zerlegung in ÄKs mit Hilfe von spezifizierten Gültigkeitsbereichen und spezifizierten oder vermuteten Sonderbehandlungen (Äquivalenz ist hypothetisch, Klassen werden z.T. nach Erfahrungen und Intuition gebildet)
    - Vorgehen beim Äquivalenzklassentest
      - Definitionsbereich für jede Eingabevariable ermitteln

        - ÄKs für Eingaben bilden, d.h. gültige und ungültige Klassen bilden
            - ÄKs schrittweise weiter unterteilen
        - ÄKs identifizieren (z.B. gültige ÄKs werden identifiziert durch 1, 2,…; und jede dazugehörige ungültige ÄKs wird davon abgeleitet identifiziert: 1a, 1b, 2a, …)
        - Testfälle für gültige ÄKs definieren (Testfälle werden so ausgewählt, dass die Eingaben möglichst viele bisher noch nicht abgedeckte gültige ÄKs abdecken → Wiederholung, bis alle gültige ÄKs abgedeckt sind)
        - Testfälle für ungültige ÄKs definieren (Testfälle werden so ausgewählt, dass diese nur genau eine bisher noch nicht abgedeckte ungültige ÄK abdecken → dadurch wird vermieden, dass ein Programm bei der Entdeckung einer fehlerhaften Eingabe weitere fehlerhafte Eingaben nicht mehr verarbeitet; werden mehrere ungültige ÄKs mit einem Testfall abgedeckt, ist nicht mehr transparent, welche falsche Eingabe die Fehlerbehandlung auslöst)
        - Optional: Ausgabeäquivalenzklassen bilden, jeweils erforderliche Eingabewerte zurückrechnen und Eingabewerte ermitteln (kann aufwändig sein)

      → Ableitung von ÄKs aus der Spezifikation ist nicht trivial
    - trotz Äquivalenzklassenbildung hohe Anzahl von Testfällen bei vielen Eingabeparametern → Reduzierung der Menge gültiger Testfälle notwendig, z.B. über Nutzungsprofile
    - Güte der Testfälle abhängig von der Aussagekraft der Spezifikation und nicht nur von den Eingabeparametern
- Grenzwertanalyse
    - Problem: Annahme, dass alle Werte innerhalb einer ÄK zum Auffinden von Fehlern gleich gut geeignet sind, ist nicht realistisch
    - Erfahrung: Fehler treten oft in den Grenzbereichen der ÄK auf
    - Heuristik: bei gültigen ÄK, die Intervalle umfassen, wird für jede Grenze jeweils ein Testfall abgeleitet für den exakten Grenzwert und die beiden benachbarten Werte (kleinstmögliche Inkrement wählen)
    - weitere Auswahl: besondere Werte wie Null, positive und negative Werte
- Ursache-Wirkungs-Analyse
    - Problem bei Tests auf Basis von ÄK ist, dass nur einzelne Eingaben betrachtet werden, jedoch keine Wechselwirkungen und Abhängigkeiten
    - Lösungsansatz: systematische Auswahl von Eingabekombinationen → Ursache-Wirkungs-Graph (UWG)
    - Vorgehensweise: Ursachen und Wirkungen aus der (Teil-)Spezifikation identifizieren
        - Ursache: einzelne Eingangsbedingung (oder ÄK mehrere Bedingungen)
        - Wirkung: Ausgangsbedingung oder Systemtransformation
    - Ursachen und Wirkungen im Graphen werden auf Grund der Bedeutung der Spezifikation entsprechend verbunden

Identität	Ursache & Wirken besitzen stets den selben booleschen Wert	
Negation	Wirkung tritt auf, falls Ursache nicht vorhanden ist (und andersrum)	
ODER-Beziehung	Wirkung tritt auf, falls mindestens eine Ursache erfüllt ist	
UND-Beziehung	Wirkung tritt auf, falls mehrere Ursachen gleichzeitig erfüllt sind	
Exklusive Abhängigkeit	höchstens eines kann den Zustand 1 haben	
Inklusive Abhängigkeit	mindestens eines muss den Zustand 1 haben	

One-and-Only-One Abhängigkeit	eines und nur eines muss den Zustand 1 haben	

- Graph wird in eine Entscheidungstabelle überführt
  - Auswahl einer Wirkung
  - Durchsuchen des Graphen nach Kombinationen von Ursachen, die den Eintritt der Wirkung hervorrufen bzw. nicht hervorrufen (dies erfolgt ausgehend von der Wirkung in Richtung der Ursachen)
  - Erzeugung jeweils einer Spalte der Entscheidungstabelle für alle gefundenen Ursachenkombinationen und die verursachten Zustände der übrigen Wirkungen
  - Eliminieren doppelter Einträge in der Entscheidungstabelle
- Spalten der Entscheidungstabelle werden in Testfälle konvertiert
- Regeln zur Ermittlung der Ursachenkombinationen
  - ODER-Verknüpfung mit Ergebnis 1: eine Ursache = 1, Rest 0
  - ODER-Verknüpfung mit Ergebnis 0: alle Ursachen = 0
  - UND-Verknüpfung mit Ergebnis 1: alle Ursachen = 1
  - UND-Verknüpfung mit Ergebnis 0: eine Ursache = 0, Rest 1

- Quellcodebasierte Techniken (White-Box-Test)
  - Testreferenz ist Quellcode des Prüfobjekts
  - Bewertung der Vollständigkeit der Testfälle stützt sich auf Quellcode, Bewertung der Korrektheit der Testergebnisse auf Basis der Spezifikation!
  - Vorteil: es lassen sich formale Testauswahlkriterien definieren
  - Nachteil: nicht implementierte Funktionalitäten, die in der Spezifikation beschrieben sind, werden nicht erkannt
  - Kontrollflussbasierter Test
    - Programm wird als Kontrollflussgraph betrachtet
    - Es existieren verschiedene Testverfahren, die unterschiedliche Abdeckungen des Graphen verfolgen:
      - Anweisungsüberdeckung (C_0-Test, einfache kontrollflussbasierte Testmethode)
        - Eine Testfallmenge T erfüllt das C_0-Kriterium, wenn es für jede Anweisung A des Programms P einen Testfall gibt, der die Anweisung A ausführt.
        - Überdeckungsgrad: $\frac{\text{ausgeführte Anweisungen}}{\text{Anzahl aller Anweisungen}}$
        - funktionale Testfälle erreichen i.d.R 60 - 70 % Anweisungsüberdeckung
        - Fehleridentifizierungsquote 18 %
        - sehr schwaches Kriterium: reicht nicht als Mindestanforderung an einen Test aus
        - häufig auftretende Fehler werden nicht zuverlässig erkannt, dafür jedoch dead-code
      - Zweigüberdeckung (C_1-Test, dynamische kontrollflussbasierte Testmethode)
        - Eine Testfallmenge T erfüllt das C_1-Kriterium, wenn es für jede Kante k im Kontrollflussgraph von P einen Weg in Wege (T,P) gibt, zu dem k gehört, d.h. wenn alle Entscheidungskanten ausgeführt werden
        - Überdeckungsgrad: $\frac{\text{ausgeführte Zweige}}{\text{Anzahl aller Zweige}}$
        - funktionale Testfälle erreichen i.d.R 80 % Anweisungsüberdeckung
        - Problem ist die Beziehung zwischen Testfallanzahl und Überdeckungsrate: bereits wenige Testfälle erzielen hohe Überdeckungsraten, mit jedem weiteren Testfall werden nur noch wenige, bisher nicht abgedeckte Zweige durchlaufen, Überdeckungsrate steigt nur langsam weiter an → Vortäuschung falscher Sicherheit
        - Abhilfe: Zweige nicht mitzählen, die immer dann ausgeführt werden, wenn auch ein anderer Zweig ausgeführt wird, verbleibende Zweige = primitive Zweige, Ausführung aller primitiven Zweige = Ausführung aller Zweige → verbessertes Abdeckungskriterium mit linearem Verhalten

        - gilt als minimales Testkriterium und bildet die Grundlage für weitere strukturbasierte Techniken, subsumiert den C_0-Test
        - Schleifen werden nicht zuverlässig getestet (müssen nur einmal durchlaufen werden, um das Kriterium zu erfüllen), zusammengesetzte Entscheidungen werden nicht zuverlässig getestet
    - Bedingungsüberdeckung
        - Ziel: zuverlässiges Testen zusammengesetzter Entscheidungen
        - bei Zweigentscheidung ist nur von Bedeutung, dass die Gesamtentscheidung jeweils mindestens einmal die Werte true und false annimmt, bei der Bedingungsüberdeckung müssen die Testfälle auch die Teilentscheidungen berücksichtigen
        - einfache Bedingungsüberdeckung
            - Test aller atomaren Entscheidungen gegen true/false, subsumiert i.A. nicht den C_1-Test
        - minimale Mehrfach-Bedingungsüberdeckung
            - neben atomaren Entscheidungen müssen auch alle Teilentscheidungen gegen true/false geprüft werden
        - modifizierte Bedingungs-/Entscheidungsüberdeckung
            - Prüfung, ob atomare Entscheidungen den Wahrheitswert der Gesamtentscheidung unabhängig von andere Teilentscheidungen beeinflusst
        - Mehrfach-Bedingungsüberdeckung
            - Test aller Wahrheitswertekombinationen der atomaren Entscheidungen
            - subsumiert alle anderen Bedingungsüberdeckungstest → hoher Aufwand
    - minimale Termüberdeckung
        - Ziel: jeder atomare Term einer Bedingung bestimmt wenigstens einmal mit true und false das Gesamtergebnis der Bedingung
    - Pfadüberdeckung (C_7-Test, intensivste kontrollflussorientierte Testmethode)
        - Ziel: alle unterschiedlichen Pfade sollen einmal ausgeführt werden
        - Schleifen müssen für einen vollständigen C_7-Test mit jeder möglichen Anzahl von Durchläufen getestet werden
        - für den praktischen Einsatz nur eingeschränkt tauglich, Anzahl der Möglichkeiten können unendlich groß sein, Kontrollflussgraph beinhaltet nicht gesamte Information des Programmcodes
    - eingeschränkte Pfadüberdeckung
        - nicht alle Pfade durch Schleifen werden berücksichtigt, Beschränkung auf fünf Aspekte:
            - 0 Iterationen: die Schleife wird nicht betreten
            - 1 Iteration: zeigt häufig Initialisierungsfehler
            - 2 Iterationen: testet den Rumpf der Schleife
            - typische Anzahl Iterationen: Prüfung des Normalfalls
            - maximale Anzahl Iterationen: zeigt Fehler beim Abbruchkriterium
- Datenflussbasierter Test
    - zwei Klassen von Anweisungen: Steuerung des Ablaufs und Manipulation der Daten
    - Zugriff auf die Variablen ist Testmittelpunkt
    - schreibender Zugriff auf Variablen: def x (z.B. int x = 0)
    - lesender Zugriff auf Variablen:
        - Eingabewerte für eine Berechnung: c-use x (z.B. y = 2 * x)
        - Ermittlung des Wahrheitswertes einer Entscheidung: p-use x (z.B. if (x > 2))
- Zusammenfassung
    - Funktionstest beinhalten Methoden zur Testfallableitung aus der Spezifikation
    - Strukturtests geben Testvollständigkeitskriterien auf Basis des Programmcodes an
    - Kombination der Strukturtests mit Methoden zur Testfallableitung: Tester definiert nach einer gewählten Methode Testfälle, führt die Testfälle mit Hilfe eines Werkzeugs aus, welches die Testabdeckung ermittelt, Tester analysiert Abdeckung und ergänzt ggfs. Testfälle