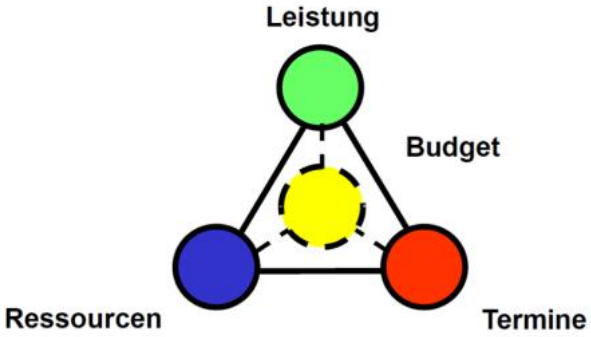
SEPM Zusammenfassung

B1 (Einführung in das Projektmanagement)
Definition Projekt:
Ein Projekt ist (im Gegensatz zum normalen Tagesgeschäft oder zur Produktion) ein einmaliges Vorhaben mit einem definierten Anfang, einem definierten Ende und mehreren beteiligten Personen.

Kennzeichnen von Projekten:

- Abgrenzbarkeit bezüglich Aufgabe/Erwartetes Ergebnis (Funktionsumfang, Leistungsumfang, Qualität)
- Ressourcen (Personaleinsatz, Geldmittel)
- Zeitrahmen (Start, Ende)
- Komplexität der Aufgabe
- Einmaligkeit der Aufgabe



Projektmanagement:

- Bedeutung: Die Gesamtheit der Methoden/Verhaltensweisen effizienteren Steuerung der Abwicklung von besonderen Aufgabenstellungen (Projekten). Im engeren Sinn ist das Projektmanagement die Projektleitung, d.h. die für die Führung/Steuerung eines Projekts verantwortliche Person/Stelle.
- Einsatz:
  - Theorie: Aufgabe lässt sich einfacher lösen/organisieren
  - Praxis: Projektorganisation oft falsch eingesetzt: z.B. Projektinflation, fehlendes Rollenverständnis

Risiken:
Problematik beim Risikomanagement:

- Menschen denken ziel-/ergebnisorientiert
- Gefährdet Interessen der Stakeholder
- Schwierigkeiten mit Wahrscheinlichkeiten
- Risikos sind binär (treten auf oder nicht)

Ursachen von Risiken:

- Mangelhafte Anforderungen:
  - – Die Anforderungen decken die Erfordernisse nicht ab
  - – Die Anforderungen sind unklar oder mehrdeutig
- Mangelhafte Umsetzung der Anforderungen
  - – Mangelndes Verständnis der Anforderungen
  - – Mängel in der technischen Umsetzung (Implementierung)
  - – Termin-Not und Ressourcenmangel
- Mängel im Projektmanagement
  - – Mängel in der Kommunikation
  - – Unrealistischer Termin- und Kostenrahmen
  - – Ressourcenmangel: Verfügbarkeit, Qualifikation, Erfahrung

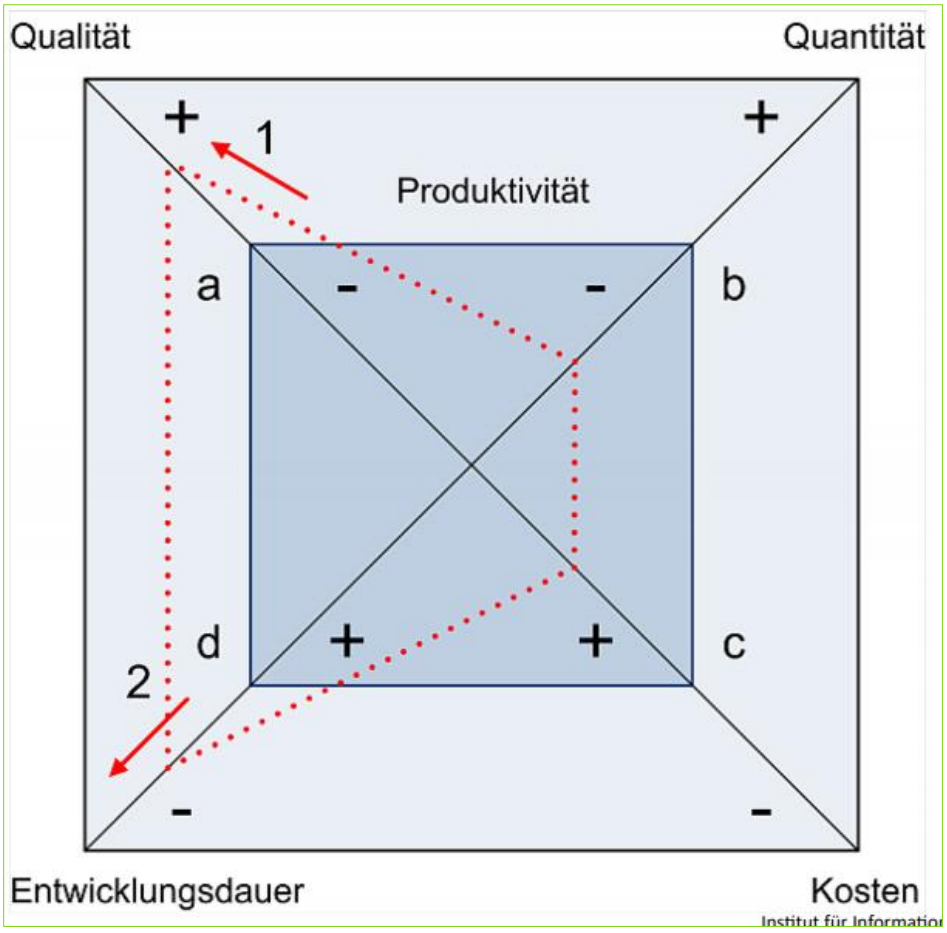
Risikoanalyse:

- Spezifische Risiken aus der Aufgabenstellung
- Generelle Risiken (Personalausfall)
- Eintrittswahrscheinlichkeit
- Schadenpotential

Risikomanagement:

- Vorbeugemaßnahmen (reduzieren das Risiko)
- Abhilfemaßnahmen (lindern die Folgen)
- Kalkulation möglicher Mehrkosten

Das Teufelsquadrat:



Die vier angegebenen Ziele des Quadrates konkurrieren um die verfügbare Produktivität, welche durch die Fläche des äußeren Vierecks dargestellt wird. Aufgrund der Begrenzungen der Ressourcen ergibt sich automatisch eine Begrenzung der Produktivität, symbolisiert durch die Fläche des inneren Vierecks. Man kann das Viereck in die eine oder andere Richtung strecken, muss dann allerdings einen geringerem Zielerfüllungsgrad auf der anderen Seite hinnehmen.

Aufgaben im Projektmanagement:

- Aufgabenstellung definieren
- Projekt planen
- Projekt(team) organisieren
- Aufgaben verteilen
- Fortschritt kontrollieren
- Bei Bedarf steuernd eingreifen
- Mit Risikos beschäftigen
- Entscheidungen vorbereiten/fällen
- Verwaltungskram
- etc.

Elemente des PM

- Projektablauf – was ist zu tun?
- Arbeitsstruktur – wer ist wofür zuständig?
- Informationswesen – wer informiert wen/worüber/wann/wie?
- Dabei zu beachten: Methodenunterstützung, psychologische Aspekte, Umfeldbedingungen

Projektstart:

1. Projektidee
2. Projektziel
3. Projektantrag
4. Genehmigung
5. Projektauftrag
6. Kickoff

Vorarbeiten für einen Projektstart:

- Evaluierung der Projektidee
- Vorstudien
- Ersteinschätzung des Ressourceneinsatzes
- Projektumfeldanalyse:
  - Projekttyp
  - Qualität
  - Mögliche Risiken

Die Ergebnisse der Vorarbeit sind relevant:

- Um zu entscheiden ob das Projekt wirklich durchgeführt wird
- Zur Konkretisierung einzelner Punkte aus dem Projektauftrag
- Zur Risikoreduktion
- Für die Forschung und Entwicklung

Das Projekt soll unter erwarteten Änderung dennoch erfolgsversprechend ablaufen.
Voraussetzungen dafür sind:

- Klare Prioritäten (Anforderungen)
- Planung muss Spielräume für Veränderungen übrig lassen
- Planungszyklen soll kürzer werden, um auf Änderungen geordnet reagieren zu können

Projektvorschlag:

- Projektbezeichnung und Entwicklerteam
- Ausgangssituation
- Projektbeschreibung
- Zielgruppe
- Ziele & Nicht-Ziele
- Typ des Projekts
- etc.

Rollen:

- Auftraggeber/Auftragnehmer
- Projektleiter
- Entwicklerteam:
  - Software Architekt
  - Dokumentationsbeauftragter
  - Testbeauftragter

Projektauftrag (Projektantrag, Projekterklärung, Projektdefinition oder Projektbeschreibung)

- schriftliche Zielvereinbarung zwischen Auftraggeber & Auftragnehmer, ein Projekt zu bestimmten Bedingungen durchzuführen
- Management-Unterlage – umfasst alle wichtigen Daten für Abwicklung
- Voraussetzung: Ausarbeitung von Projektvorschlag
- Bestandteile
  - Projektname
  - Projektbeschreibung
  - Rollen und Verantwortlichkeiten
  - Nicht-Ziele, Abgrenzung
  - Komponentendiagramm
  - Wiederverwendung von Komponenten
  - Technologiewahl
  - Lieferumfang und Abnahme
  - Nicht funktionale Anforderungen
  - Arbeitsstruktur
  - Projektplan (Zeit- und Kostenplan)
  - Informationswesen
  - Besonderheiten
  - Umfeld- und Risikoanalyse

Vorgehensweise (für die Erstellung des Projektauftrages)

1. Projektstrukturplan erarbeiten
2. Arbeitsplanung durch Arbeitspaketdefinition
3. Aufwandsschätzung
4. Ablauf und Terminplanung
5. Ressourcenplanung
6. Kostenplanung
7. Optimierung des Gesamtprojektes

B2 (Technik und Werkzeuge)
Source Code Management Systeme (SCM) – Aufgaben

- Gleichzeitige Entwicklung mehrerer Zweige
- Nachvollziehbarkeit
- Zugriff auf ältere Version
- Änderungen transparent machen
- Verwaltung von Artefakten

Versionierungssysteme (Unterkategorie von SCM?):

- Revision-Control-System (RCS): Basieren auf einzelne Dateien, wie Back-Ups
- Concurent-Version-System (CVS): Zentrales, serverbasiertes System
- Verteiltes SCM: jeder Entwickler hat ein eigenes Repository (lokal) – GIT

Zentralisierte Systeme:

- Server speichert Informationen.
- Single Point of Failure (-)
- Flaschenhals (-)
- Operationen (Push, Pull…) erfordern Netzwerkoperationen (-)
- Verwaltung einfach (+)

Verteilte Systeme:

- Repository wird geklont
  - Große Datenmenge, viele Dateien (-)
- Jede Komponente in einer eigenen Repository (+)
- Lokale Operationen/Lokale Versionierung (+)

Auswahl:

- Performance
- Verwaltungsaufwand (Zentral > Verteilt)
- Scope
- Branching
- Wichtigkeit des Offline-Modus‘

Konflikte: Strategien: Locking, Merging (bevorzugt)
Arbeitsablauf bei Konflikten:

- <u>Verteilte Systeme:</u>
  - Kopie des Hauptrepository
  - Lokale Änderungen durchführen
  - Lokale Commits
  - Merge
    - Push
    - Pull Request
- <u>Zentralisierte Systeme:</u>
  - Update zum Repository
  - Lokale Änderungen
  - Update zum Repository

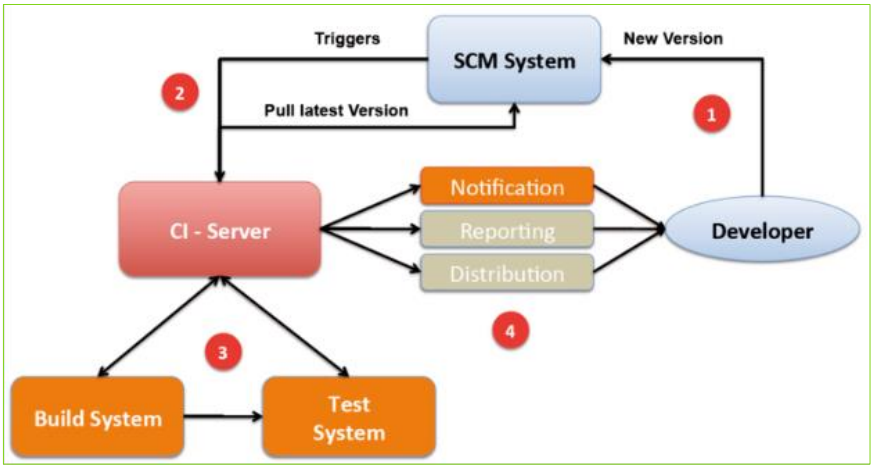
        - Änderungen remote und lokal
        - Merge
    - Commit

Automatisierung:

- Code Checks, Code Generierung, Testausführung, Kompilierung, Deployment, Validierung, Dependency Management etc.
- Beispiele: Build Management Systems (Maven)
- Convention over Configuration: Durch Einhaltung von Konventionen müssen keine Konfigurationen explizit definiert werden

Continuos Integration – regelmäßig(=kontinuierlich) Builds, Tests, Deployments auf einen zentralen Rechner ausführen lassen. Automatisierung beinhaltet:
Continuous Integration Werkzeuge unterstützen Server-basierte Integration und Ausführung von Tests. Diese Automatisierung beinhaltet:

- Ereignis-/Zeitgesteuerter Abruf
- Build Werkzeuge
- Ausführung von Tests
- Erstellen von Berichten
- Verschicken der Mitteilungen an die Zuständigen



Vorteile einer Build-Automatisierung

- Projekt schnell mit Archetypen aufsetzen
- Projekt ist portabel
- Abhängigkeiten leichter darstellbar und auflösbar
- Verbesserung der Projekt-Qualität durch Integration automatisierter Tests
- Automatisiertes Reporting

Kommunikation im Team:

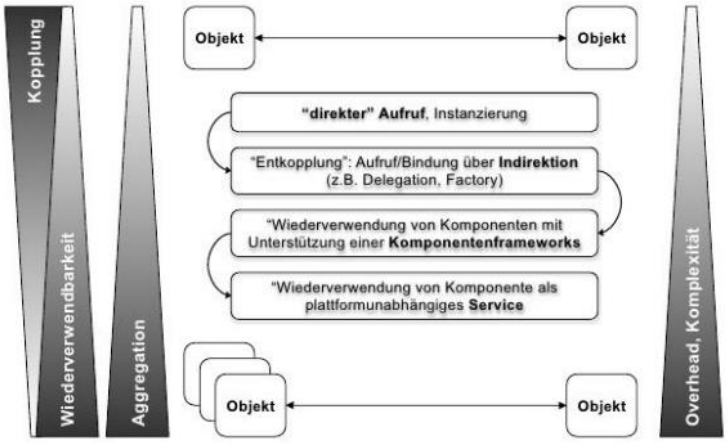
- Synchron:
    - Telefon, Face-To-Face, Team-Viewer
- Asynchron:
    - Mailingliste
    - Wiki
    - Email

- Issue Tracker (Bug Tracker, Projekt Tracker) – Erfassen von Fehlern/Änderungswünschen

Problemen bei globaler Kommunikation
- Andere Zeitzonen
- Andere Arbeitszeiten
- Große geographische Distanzen (→ kein Face-2-Face)

→ asynchrone Kommunikation

Komponentenorientierte Software Entwicklung
Begriffe
- Komponente
  - klare, stabile Schnittstelle
  - Höhere Granularität als eine Klasse
  - Wiederverwendbarkeit
- Service:
  - klar definierte Schnittstelle
  - plattformunabhängig
  - über ein Netzwerk angeboten
- Framework
  - Wiederverwendbar
  - Rahmenbedingungen für Komponenten (z.B. Persistenz-Framework)



Direkter Aufruf:
- Einfach zu implementieren
- Binding beim Kompillieren
- Starke Kopplung

Interface-Prinzip (Instanzierung):
- Einfach auszutauschen (Interface-Implementierung)
- Binding beim Kompillieren

Komponentenframework:

- Caller ruft Methoden des Frameworks auf, anstatt auf Komponenten einzeln zuzugreifen
- Keine Abhängigkeit beim Kompilieren

Inversion of Control

Abhängigkeiten werden von einem Container verwaltet, die Komponenten wissen nichts darüber (IOC ist transparent). Die Abhängigkeiten werden injiziert (siehe auch Dependency Injection DI). Vorteile von IOC sind:

- Abhängigkeiten vom Container verwaltet
- Abhängigkeit werden in Komponenten injiziert („Dependancy injection“ – DI)
- Vorteile:
  - Hohe Verwendbarkeit durch zentrale Verwaltung
  - Einfaches Austauschen der Implementierung
  - Verwalten von unterschiedlichen Konfigurationen (Deploy, Development)
  - Verteilen von Aufgaben (Testen, Implementieren…)

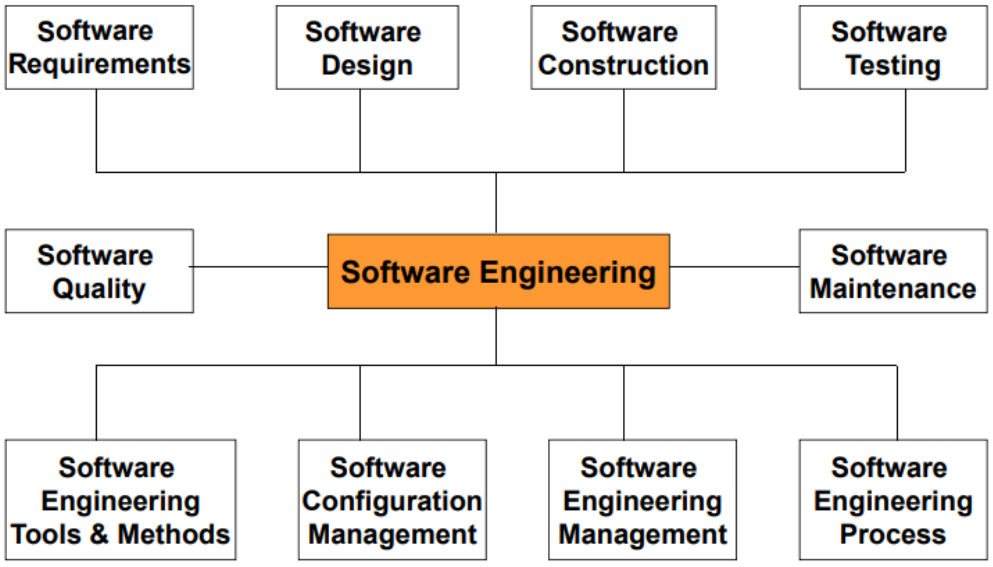
B3 – Einführung in Software Engineering
Begriff „No Surprise“ - Software – bezieht sich auf Software, die sich in der Anwendung korrekt und ohne Überraschungen verhält.
Software Engineering soll helfen, für große Software Systeme ähnliche Qualitätsmaßstäbe zu erreichen wie in klassischen Ingenieursdisziplinen.

- Kostengünstige Entwicklung
- Hohe Qualität
- Innerhalb des geplanten Zeitrahmens

Projekt und Produkttypen:

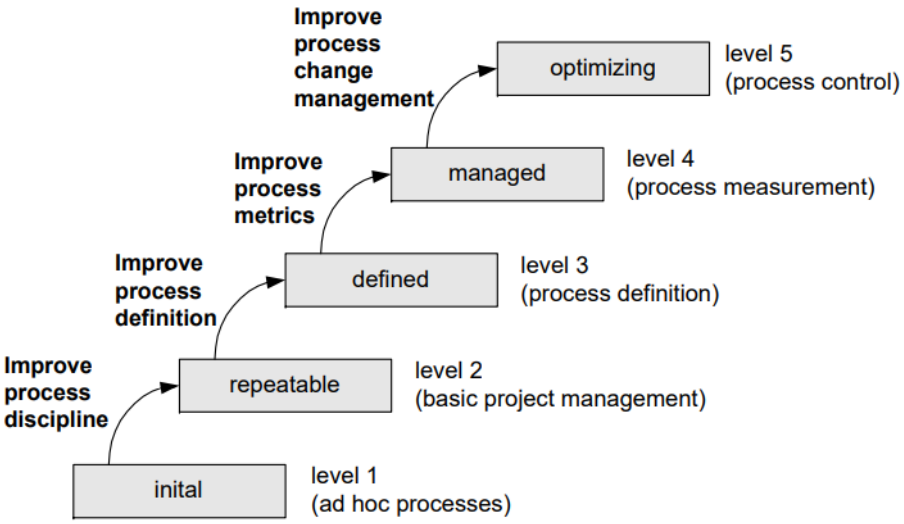
- Kommerzielle Software
  - Fokus: Benutzbarkeit, Verfügbarkeit, Support
- Eingebettete Systeme
  - Fokus: Zeigesteuert, Echtzeitanforderungen
- Wissenschaftliche Software
  - Fokus: Korrektheit, Rechengenauigkeit

Software Engineering Body of Knowledge:



Definition von „Reifegraden“ für Softwareentwicklungsorganisationen

Reifgrade der Software-Entwicklung
Capability Maturity Model Integrated CMMI



Stufe 1 (initial): Mindeststufe ohne besondere Software-Prozesse
Stufe 2 (repetable): Basic Project-Management, wiederholbare Software-Projekte

- Requirements Management, Projekt Planning, Projekt Tracking/Oversight

Stufe 3 (process definition): Definierte Prozesse als Basis für weitere Stufen der Prozessverbesserung verwenden

- Peer Reviews

Typische Probleme größerer SW-Projekte

- Zeitverzögerungen
- Überschreiten des Budgets
- Mangelhafte Qualität der Software
- Späteres Erkennen von Design-Fehlern
- Schwierige/Teure Wartung

Komplexitätstreiber:

Merkmal	Messbare Attribute	Beispiel
Grösse (PM)	Anzahl der beteiligten Personen	Personen: 50
Dauer und Aufwand (PM)	Anzahl Wochen bzw. Monate	Dauer: 52 Wochen Personen-Jahre: 3 PJ
Verwendete Technologien (SE)	Art, Anzahl und Alter der Technologie	Art: Scriping language, Alter: 4 J., Anzahl: 5
Komplexität (SE)	Anzahl Klassen, Module, Datenbanken; verwendete Technologien, Zeilen Code	Datenbanken: 2 Klassen: 42 Code: 30 000 Zeilen

Projekttypen – Größe eines Projekts

Größe	Kriterien	Beispiele
Klein	Bis zu 6 Personen Monate: 0-8 Anzahl Technologien: <5	Rechenprobleme, Algorithmen
Mittel	10-30 Personen Monate: 9-24 Anzahl Technologien: 5-12	Buchhaltung, Lagerverwaltung
Groß	50-100 Personen Monate: 25-45 Anzahl Technologien: 12-20	Compiler, Datenbank
Sehr groß	Ab 100 Personen Monate: 45-n Anzahl Technologien: >20	Raumfahrt, Atomkraftwerk, elektronische Börse, große Standardsoftware

Änderungen beim Erhöhen der Projektgröße:
Was ändert sich mit Zunahme der Größe:

- Komplexität (SE & PM)
- Bedarf an Flexibilität (SE & PM)
- Bedarf an Organisation/Plannung und Überblick (PM)
- Bedarf an Prozessorientierung (PM)
- Bedarf an Kompetenzen (PM)
- Anzahl der Kommunikationspfade (SE, PM)
- Testaufwand (SE)
- Bedarf an saubere Dokummentation (SE)
- Versionsverwaltung/Konfigurationsmanagement (SE)

Software-Beschaffung: Software-Architektur setzt sich aus mehreren Lösungselementen zusammen. Pro Element ist eine Entscheidung zu treffen

- Kauf, Abänderung (eines bereits existierenden Lösungsteils), Neuerstellung
- Abhängig von mehreren Faktoren: Entspricht Anforderungen, Änderbarkeit, Preis
- Hervorzuheben ist:
  - Neuerstellung ist verhältnismäßig teuer
  - Abänderung von bereits existierenden Lösungsteilen stößt an technische Grenzen
  - Neukauf enthält oft ungenutzte Funktionalität
  - Neukauf kann manchmal schwierig geändert werden, da die technischen Details nicht immer transparent sind
  -

	Kauf	**Abänderung**	**Neuerstellung**
Entspricht Anforde-rungen	Ungefähr (oft viele ungenützte Funktionen oder ein paar fehlende)	Oft ist keine exakte Anpassung möglich → Beschränkung durch technische Grenzen	Genau
Änderbarkeit	Schwierig, da technische Details oft nicht transparent sind	Ausgangsprodukt wurde selbst hergestellt: leicht änderbar, Durch andere hergestellt: Schwer änderbar	Gut möglich (Dokumentation der Entwicklung und somit techn. Transparenz vorhanden)
Preis	Nach Anforderung + Verbreitung	- durch eigene IT- Abteilung: kalkulierbar - durch andere: kostenintensiver	Teuer

Neukauf am besten wenn,

- billiger als Neuerstellung
- gewünschte Anforderungen sind vorhanden
- ausführliche Dokumentation
- Support

Vergleich: Embedded Systems/Kommerzielle Software

	Embedded Systems	Kommerzielle Software
Steuerung	Ereignisgesteuert, oft auch vollständig automatisiert	Benutzergesteuert
Kosten	Teuer, wegen Neuentwicklung oder Anpassung	Kaufen billiger als selber entwickeln
Zuverlässigkeit	Sehr wichtig	Oft nicht entscheidend
Wartung	Schwierig, z.T. Hardware-technisch unmöglich	Meist professioneller Support
Sicherheit	Müssen sicher sein (safety)	Unterschiedliche Wichtigkeit: Online-Banking, Datenbank Systeme vs. Photobearbeitung
Usability	Benutzerinterface rudimentär oder nicht vorhanden	Oft entscheidend, vor allem bei grosser Konkurrenz
Beispiele	Handysteuerung, Liftsteuerung, ABS-System, Ampel	Datenbanksystem, Web-Applikationen, Texteditor

Faktoren für ein Erfolgreiches Projekt:

1. Einbringung und Berücksichtigung der Anwender
2. Adäquates Projektmanagement: nicht zu viel aber auch nicht zu wenig
3. Anforderungen müssen eindeutig beschrieben, realisierbar und auch überprüfbar sein
4. Flexibler, realistischer Projektplan, der mögl. Verzögerungen berücksichtigt
5. Realistische Kostenschätzung und Budget, inkl. Risikoanalyse
6. Angemessene Ziele
7. Schlüsselteammitglieder haben genügend Projekterfahrung
8. Gute Teamarbeit, funktionierende Kommunikation im Team

Warum Projekte scheitern:

1. Unvollständige Anforderungen
2. Anwender nicht involviert
3. Zu wenig Ressourcen
4. Unrealistische Zeit- und Kostenpläne
5. Keine Management Unterstützung
6. Häufige Änderung der Anforderungen
7. Qualitätsmängel bei extern vergebenen Komponenten
8. Qualitätsmängel bei extern vergebenen Aufgaben
9. Fehlende Planung
10. Projekt wird nicht mehr benötigt.

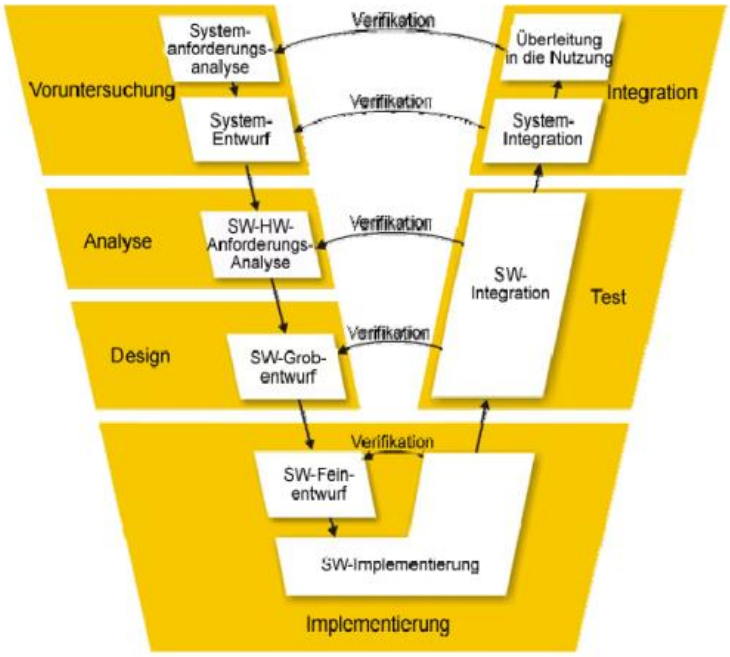
Verfolgen von Anforderungen
Vier Richtlininien der Produktqualitäten:

- **Change-driven requirements:**
  - Die Erweiterbarkeit und Anpassung wird oft vernachlässigt
- **Risk-driven requirements:**
  - Wie detailliert müssen Anforderungen sein? “If it’s risky to leave it out, put it in“
- **Value-driven requirements**:
  - Erfordert Geschäftsfall-Analyse
- **Shared-vision driven requirements:**
  - Stakeholder bewerten regelmäßig die Anforderungen → gesamtheitliche Sicht ist wichtiger als präzise Anforderungen

Prozessmodelltypen:
Charakteristika einiger Prozessmodelle:

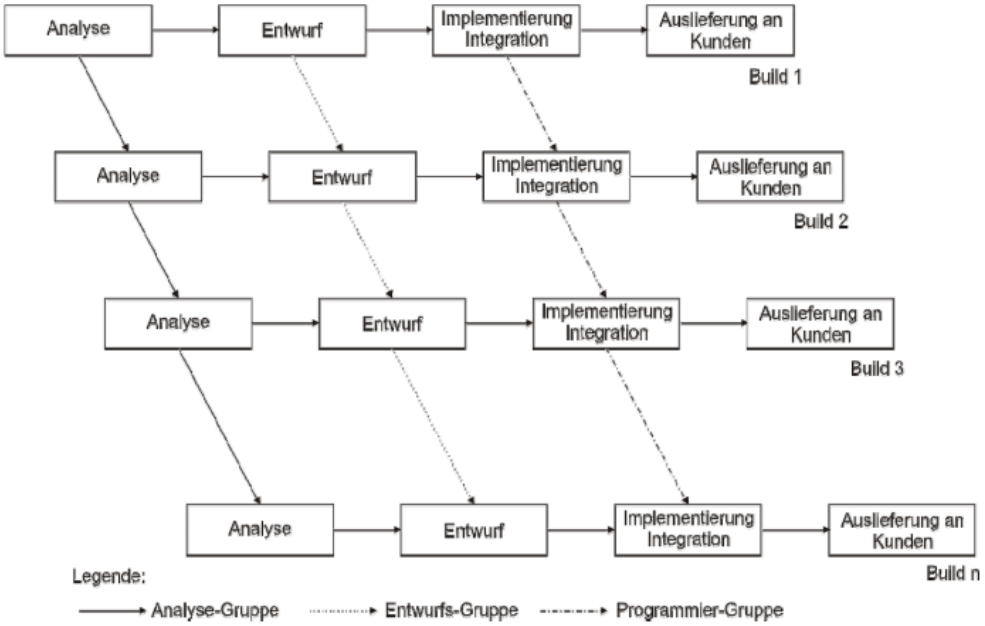
Modell	Phasen	Schwerpunkte
V-Modell	Voruntersuchung, Analyse, Design, Implementierung, Test, Integration	Dokumentation, Qualitätssicherung, Verbesserung der Kommunikation aller Beteiligten
Inkrementelles Modell	Analyse, Entwurf, Implementierung Integration, Auslieferung an Kunden	Minimale Entwicklungszeit, Risikominimierung, kurze Phasen
Iteratives Modell, z.B Unified Process	Etablierung, Entwurf, Konstruktion, Übergang	Architekturzentriert, Anwendungsfall gesteuert
Extreme Programming (XP)	Coding, Testing, Listening, Designing laufen dauern parallel ab und nicht in Phasen	Frühe Fehlererkennung, Minimale Entwicklungszeit, Schnelle Anpassung an sich ändernde Anforderungen

Prozessmodelle:
V-Modell



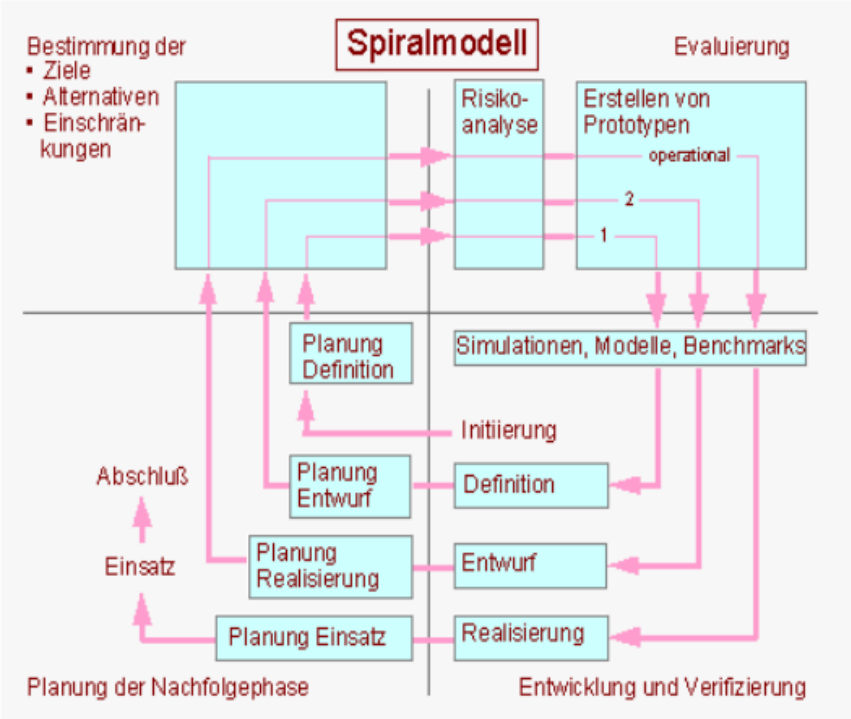
- Gesamtaufwand sinkt gemessen am Lebenszyklus einer IT-Anwendung (Kostenreduktion)
- Kürzere Einarbeitungszeiten für neue Mitarbeiter im Projekt
- Einfachere Kontrolle des Projektfortschritts (auch für den Auftraggeber)
- Projekte leicht vergleichbar -> genauere Aufwandsabschätzung zukünftiger Projekte
- 4 Submodelle: Systemerstellung, Qualitätssicherung, Konfigurationsmanagement, Projektmanagement
- Anwendung: für kleine Projekte zu detailliert, für große Projekte mit hohem Qualitätsanspruch geeignet (speziell Embedded Systems)

Inkrementelles Vorgehensmodell :



- Stufenweise Entwicklung des Projektes
- Kontinuierliche Integration, Kontinuierlich hohe Qualität
- Vorteile
  - Planung einfach
  - Planverfolgung einfach
  - Bei unklaren Anforderungen geeignet
- Einsatz: große, komplexe Systeme mit langer Entwicklungszeit, wenn das Basisprodukt schnell beim Kunden sein soll (und danach weiterentwickelt wird)

Spiralmodell:



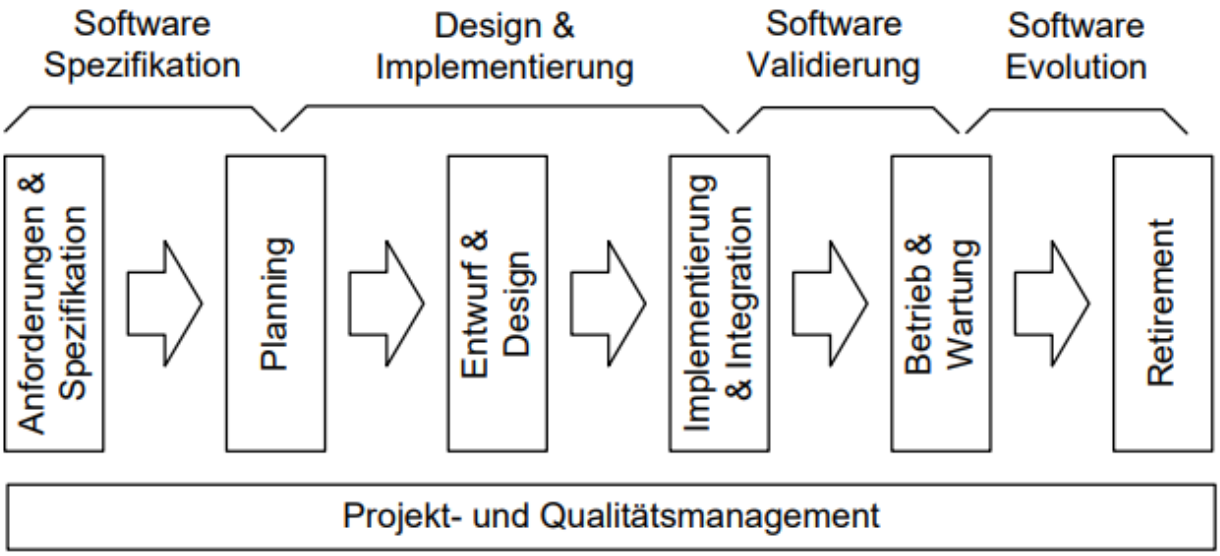
- Risikogetriebenes/Iteratives Vorgehensmodell
- Pro Teilprodukt/Verfeinerungsebene werden 4 zyklische Schritt durchlaufen
- Vorteile:
  - Sehr flexibel
  - Schrittweise Überprüfung des Wertes und der Risiken von zuerst kleinen, einfachen und zunehmend komplexen, realistischen Prototypen, um im Schlechtfall das Projekt rechtzeitig anpassen oder terminieren zu können.
- Risiken:
  - Overhead (viel Management-Aufwand)
  - Erfahrenes Management-Team
- Einsatz: große, risikoreiche Projekte

XP-Extreme Programming

- Iteratives Vorgehensmodell mit sehr kurzen Iterationen
- Agiler Ansatz
- Vorteile:
  - Schlanker, flexibler als andere Methoden
- Einsatz: Projekte mit schnell wachsenden oder vagen Anforderungen mit kleinen Entwicklerteams, zeitkritische Projekte

- Iteratives Vorgehensmodell, bei jeder Iteration wird mehr Funktionalität realisiert (sehr kurze Iterationen, Refactoring)
- Coding: kontinuierliche Integration, Programmieren in Paaren (der eine programmiert, der andere testet, Rollen können wechseln)
- Design: einfache Dinge nur einmal tun
- Dokumentation: selbstdokumentierend im Quelltext
- Test: Unit-Tests, Testfälle im Vornherein spezifizieren
- Integration: mehrmals täglich soll der Quellcode in die zentrale Code-Basis eingefügt werden, inklusive Tests
- Anwendung: Projekte mit schnell wechselnden oder vagen Anforderungen, mit kleinen Entwicklergruppen (bis 12 Personen), zeitkritische Projekte

B4: Software Engineering Phasen



Software-Phasen (kurz zusammengefasst):

1. Anforderungen – Wünsche des Kunden (user/customer view) – testbar
2. Spezifikation – Beschreibt das System aus technischer Sicht (engineering view)
3. Planung – Erstellung des Projektplans bezüglich Zeit, Kosten, Dauer
4. Entwurf/Design – technische Lösung der Systemanforderungen (Komponenten, Datenbankdesign…)
5. Implementieren und Testen
6. Integration und Testen – Zusammenfügen, Test der einzelnen Komponenten auf Architektur- und Systemebene
7. Operation und Wartung - Fehlerbehandlung, Unterstützung, Erweiterung des Softwareprodukts
8. Retirement - Nach der Einzelphase muss die Software kontrolliert aus dem Betrieb genommen werden

Anforderungen

- Qualitätskriterien
  - Testbar (überprüfbar)
  - Nachvollziehbar
  - Konsistent
  - Realistisch
  - Verständlich
- Arten:
  - Funktionale Anforderungen - Was bzw. welche Funktion soll umgesetzt werden?
  - Nicht-funktionale Anforderungen – sonstige Anforderungen, die nicht auf Funktionalität abzielen, diese aber beeinflussen
    - Leistung und Performance
    - Usability, menschliche Faktoren
    - Sicherheit
    - Erweiterbarkeit
    - Wartbarkeit

- Design-Bedingungen
  - worauf von technischer Sicht bei der Implementierung geachtet werden soll
- Prozessbedingungen
  - Rahmenbedingungen im Softwareprojekt z.B. Ressourcen/Dokumentation

Stakeholder – Personen, „die aktiv an einem Projekt beteiligt sind, oder deren Interessen als Folge der Projektdurchführung oder des Projektabschlusses positiv oder negativ beeinflusst werden können“.
Beispiele:

- Kunden - bezahlen für die Software, haben Anforderungen (z.B. schnelle Lieferung)
- Anwender - müssen mit dem System arbeiten, haben Anforderungen (z.B. funktionale sowie nicht funktionale)
- Entwickler - erstellen die Software, wollen neueste Technik einsetzen
- Management: -Das Management beeinflusst die Anforderungen durch Ressourcenvergabe und Zeitplan.
- Konkurrent: -Der Konkurrent beeinflusst die Anforderungen, da der Kunde sich vermutlich von der Konkurrenz abheben will, um am Markt zu bestehen

Anforderungserhebung:

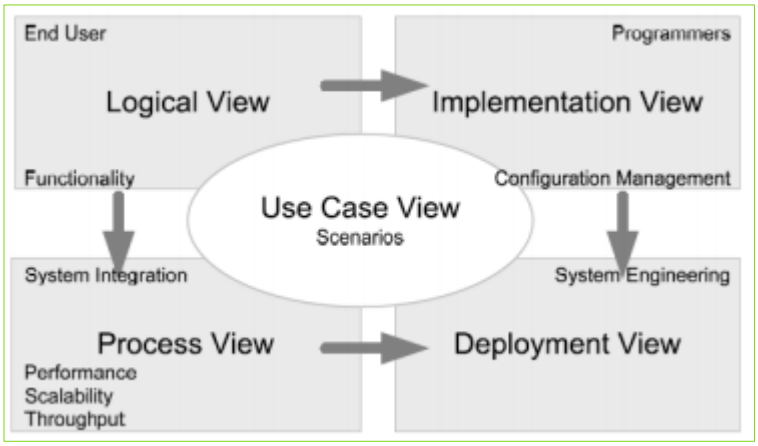
- Erhebung der Anforderungen aller Stakeholder (Zeitaufwändig und nicht trivial)
- Priorisieren
  - Hauptanforderungen (must-be)
  - Gewünschte Anforderungen (expected, nicht entscheidend)
    - Menüführung entspricht dem Standard
  - Optionale Anforderungen (nice-to-have)

Entwurf und Design – beinhaltet:

- Architekturdefinitionen & Evaluierungen zur systematischen Analyse von verschiedenen Architekturen
- Definition von Komponenten & Systemschnittstellen
- Domänen- und Datenbankmodelle
- User Interfaces

Das Software Design beinhaltet technische Anwendungsfälle der Anforderungen, aller ihrer Subsysteme (Komponenten) und Detailinformation, wie die Lösung aus technischer Sicht aussehen soll (auch Datenstrukturen, Datenflüsse und Algorithmen).

4+1 View Model of Architecture:



- Logical View
  - Funktionale Anforderungen, Enduser Sicht (UML Klassendiagramme) - Subsysteme, Klassen
- Implementation View
  - Statische Software Komponenten, Fokus auf Implementierung (UML-Komponentendiagramm) – Configuration Management, Source Code
- Prozess View
  - Nicht funktionale Anforderungen, Fokus auf Systemintegration (UML-Aktivitätsdiagramm) – Lastverteilung, Fehlertoleranz, Laufzeitbedingungen
- Deployment View
  - Ausführbare Applikation, Fokus auf System Engineers (UML-Verteilungsdiagramm) – Deployment, Installation, Performance
- Use-Case-View:
  - Anwendungsfälle und Aktivitäten (=Szenarien) werden abgebildet und in einen Zusammenhang gesetzt, gemeinsamer Nenner (UML-Anwendungsfalldiagramm)

Design Entscheidungen:

- Kopplung/Bindung – Coupling & Cohesion
  - Kopplung: Abhängigkeit **zwischen** den einzelnen Komponenten (z.B: Anzahl der Methodenaufrufe). Hohe Kopplung ↔ Hohe Abhängigkeit. Soll gering sein.
  - Bindung: Innere Zusammenhalt der Komponente d.h. wie gut und effizient arbeiten die Methoden/Variablen. Niedrige Cohesion ↔ sehr komplexe Komponente. Soll hoch sein, aber nicht allzuhoch.
  - Ziel: Gleichgewicht
- System Control
  - Stair – Verteilte Kontrolle, schrittweises Ausführen von Funktionen
    - Wiederverwendbarkeit der Methoden steigt
    - Spätere Wartung erschwert

- Fork – Zentrales Objekt übernimmt die Kontrolle
  - Wiederverwendbarkeit von Datenobjekten steigt
  - Wartbarkeit wird verbessert

Weitere Design Principles

- Abstraction – Ignorieren von Details zur Reduktion von Komplexität
- Modularisierung/Decomposition
- Encapsulation/Information Hiding
- Trennung Interface/Implementierung

Implementierung

- Standardisierung (extern) – übergreifende Standards (von IEEE, ISO)
  - Syntax von Programmiersprachen, Tools…
- Standardisierung (intern) – auf Unternehmensebene
  - Namenkonventionen (Code)
  - Formatierungsrichtlinien (Code)
  - Versionierung
  - Headerblock in jeder Komponente
- Unternehmensstandards
- Traceability – Nach- oder Rückverfolgung einer Information durch ihren gesamten Entwicklungszyklus und wird z.B. bei sicherheitskritischen Anwendungen gefordert.
  Sie umfasst auch die Änderungsverfolgung welche den Lebenszyklus einer Anforderung vom Ursprung über die verschiedenen Verfeinerungs- und Spezifikationsschritte bis zur Berücksichtigung in Entwicklungsartefakten nachvollziehbar macht. Mithilfe von Traceability können folgende typische Fragestellungen beantwortet werden:
  - Woher kommt eine Anforderung und wo wurde sie umgesetzt?
  - Welche Artefakte sind von einer Änderung der Anforderung betroffen?
  - Welche Anforderungen sind von einer Änderung der Umsetzung betroffen? Es gibt drei Arten von Traceability:

    - **Vertikale Traceability:** Beziehungen innerhalb einer Phase und eines ArtefaktTyps, z.B. System – Subsystsem – Komponente.

    - **Zeitliche Traceability**: Zeitliche Nachvollziehbarkeit unterschiedlicher Releases, z.B. durch Konfigurationsmanagement.

    - **Horizontale Traceability**: Beziehungen zwischen unterschiedlichen Entwicklungsartefakten, z.B. Anforderungen – Implementierung – Testfälle.

  - Vorteile:
    - Nachvollziehbarkeit der Information bei Änderungen
    - (Automatische) Benachrichtugung bei Änderungen

Integration – Die Integration unterschiedlicher Komponenten zu größeren Subsystemen

- **Big Bang** Integration - Alle Komponenten werden gleichzeitig integriert
  - VT: Kein zusätzlicher Testaufwand zur Simullierung von fehlenden Komponenten notwendig (Test-Stubs)

    - NT: Risiko bei der Integration sehr hoch, Fehler schwer zu lokalisieren
    - Einsatz: kleine, überschaubare Projekte
- **Top-Down**-Integration – Schrittweise Integration beginnend bei den Business Cases
    - VT: früh ausführbares Produkt-Framework, Prototypen für Demos, Framework für Testfälle
    - NT: Zusätzlicher (hoher) Aufwand für Test-Stubs, Integration von Hardware erfolgt erst spät (zusätzliches Risiko)
- **Bottom-Up**-Integration - Schrittweise Integration beginnend bei der Hardware
    - VT: Stabiles System (basierend auf HW Interfaces), Schrittweise Integration in Richtung Business Cases.
    - NT: Ausführbares Gesamtsystem spät verfügbar, zusätzlicher Aufwand für Prototypen, zusätzlicher Aufwand für Test-Drivers.
- **Build Integration** – Schrittweise Integration entsprechend den Business Cases (layerübergreifend)
    - VT: Verfügbarkeit von funktionalen Anforderungen, Prototyp und Demo, Berücksichtigung priorisierter Anforderungen möglich
    - NT: Wiederverwendung von Komponenten kann schwierig sein, Regressionstests erforderlich.

Qualitätssicherung/Testen – siehe vorletztes Unterkapitel

Wartung
Unterschiedliche Sichten auf die Wartung
Sichten:

- Activity View: Änderung der Software nach Auslieferung und Inbetriebnahme
- Process View: Schritte zur Durchführung einer Wartungsaufgabe
- Phase-Oriented-View: Die Wartungsphase beginnt mit der Auslieferung und Inbetriebnahme und endet mit der Stilllegung des Softwareproduktes

Kategorien:

- Reaktiv
  - Korrektive Wartung – Bug- und Fehlerkorrektur (Corrective)
  - Adaptive Wartung - Berücksichtigung geänderter externer Anforderungen (Hardware, Software) (Adaptive)
- Pro-aktiv
  - Produktpflege und Verbesserung – Erweiterungen (Perfective)
  - Vorbeugende Wartung – Ergänzung der Dokumentation (Preventive)

	Correction	Enhancement
Proactive	Preventive	Perfective
Reactive	Corrective	Adaptive

Techniken zur Wartung:

- **Herstellen des Produktverständnisses**: Das Verständnis „fremder“ Codestücke kann – speziell bei mangelnden Aufzeichnungen – sehr zeitaufwendig sein. KeyTools sind Code-Browsers und essentiell ist eine klare und präzise Dokumentation.

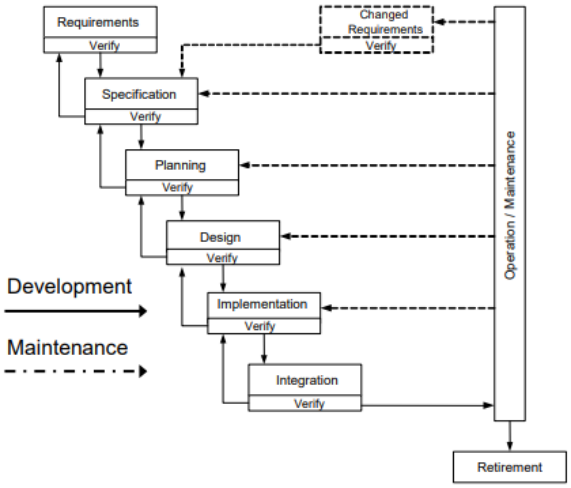
- **Reengineering:** Überprüfung und Überarbeitung des Software Codes. Stellt eine gravierende und teure Form der Änderung/Wartung dar.

- **Reverse Engineering**: Analyse der Software im Hinblick auf Komponenten und deren Zusammenhänge. Dabei hilft es, auf Basis des Software Codes, Modelle auf einem höheren Abstraktionsniveau zu erstellen z.B. UML-Modelle aus Code zu generieren.

Phase Retirement

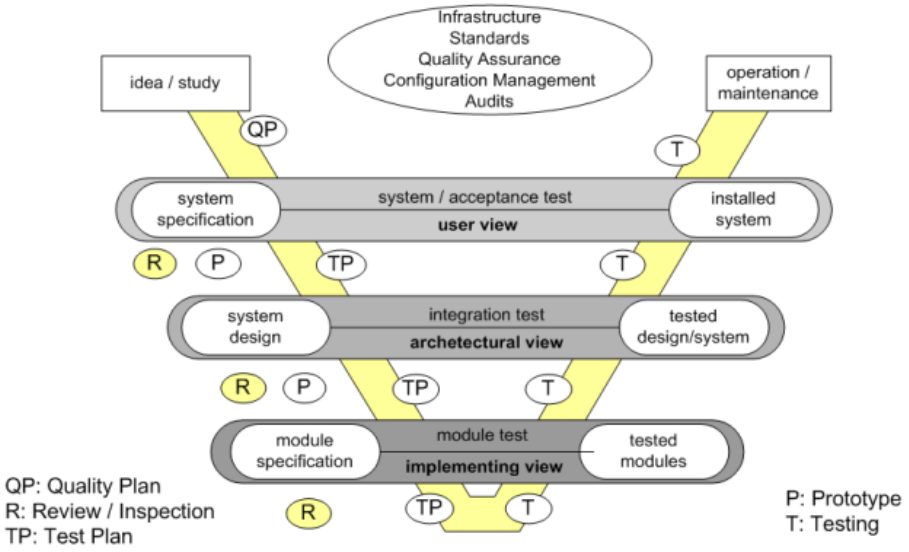
- Nach Betrieb und Wartung
- Kontrolliertes Außer-Betrieb-Setzen/störungsfreier Übergang zum Nachfolger
- Gründe für die Stilllegung einer Softwarelösung
  - Zahlreiche Änderungen in der Wartungsphase → komplettes Redesign
  - Laufzeitfehler durch Nebeneffekte
  - Inkonsistenzen durch Änderungen ohne Aktualisierung der Doku
  - Hardwareänderungen → Redesign/Neuprogrammierung

B5 – Prozesse, Vorgehensmodelle
„Ein Vorgehensmodell entspricht einer konkreten Strategie zur kontrollierten Durchführung eines spezifischen Projektes. “
Wasserfall-Modell:



- Vorteile:
  - Backtracking zu früheren Entwicklungsphasen
  - Risikominimierung durch Abschluss einer Phase
  - Bekannt
  - Strikte Trennung der Phasen
  - Unterstützt kleine Entwicklerteams
- Nachteile:
  - Alle Tasks einer Phase müssen abgschlossen sein, bevor es weitergeht
  - Starke Auswirkung von Fehlern in früheren Phasen
- Einsatz:
  - Gute Kenntnis der Anforderungsdomäne (No-Surprise Software)
  - Klar definiert (und vollständige) Anforderungen erforderlich

V-Modell



- Vorteile
  - Spezifikationsphase vs Realisieren und Testen
  - Kontext von Produkten und Tests
  - Verschiedene Abstraktionslevels
  - Fehlerbehandlung in früheren Phasen der Softwareentwicklung
- Nachteile
  - Klare Beschreibung der Systemanforderungen ist wichtig
  - Hoher Dokumentationsaufwand
  - Kritisch bei unklaren/sich ändernden Anforderungen
- Einsatz
  - Große Projekte im öffentlichen Bereich
  - Klar definierte Anforderungen

V-Modell XT

- Das V-Modell XT ist eine Weiterentwicklung des V-Modell 97.
- Veröffentlichung im Februar 2005.
- Laufende Weiterentwicklung (derzeit Version 1.3)
- Verpflichtendes Vorgehensmodell für IT Projekte im öffentlichen Bereich in Deutschland.

Zielsetzung der Entwicklung des V-Modell XT

- Verbesserung der Unterstützung von Anpassbarkeit, Anwendbarkeit, Skalierbarkeit und Änder- und Erweiterbarkeit des V-Modells.
- Berücksichtigung des neuesten Stand der Technik (Best-Practice).
- Kompatibilität zu formalen Richtlinien und Standards (z.B. ISO 9000 Standard, CMMI).
- Erweiterung des Anwendungsbereiches auf die Betrachtung des Systemlebenszyklus; Integration des Auftraggebers in das Projekt.
- Integration eines Prozessmodells zur “Einführung und Pflege eines organisations-spezifischen Vorgehensmodells).

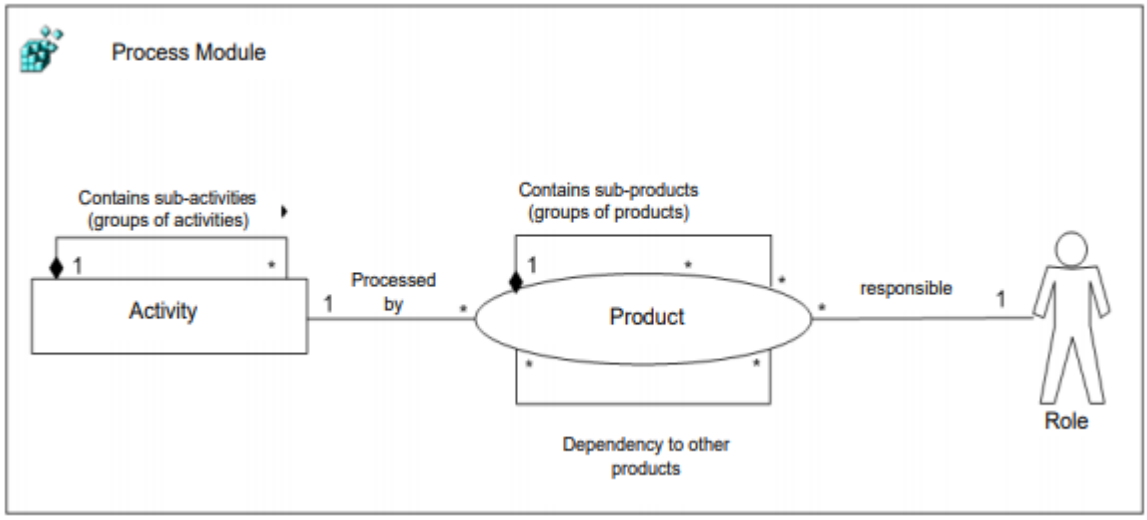
Grundkonzept:

- Produkte stehen im Mittelpunkt (=Projektergebnisse), für jedes Produkt gibt es definierte Rollen mit definierten Verantwortlichkeiten.
- Projektdurchführungsstrategien und Entscheidungspunkte geben die Reihenfolge der Produktfertigstellung und somit den Projektverlauf vor.
- **Vorgehensbausteine** sind die modularen Elemente des V-Modell XT.
  - – kapselt Rollen, Produkte und Aktivitäten.
  - – Kann als unabhängige Einheit eingesetzt werden.
  - – Ist eine Einheit, die unabhängig veränder- und aktualisierbar ist.

Komponenten des XT Modell:

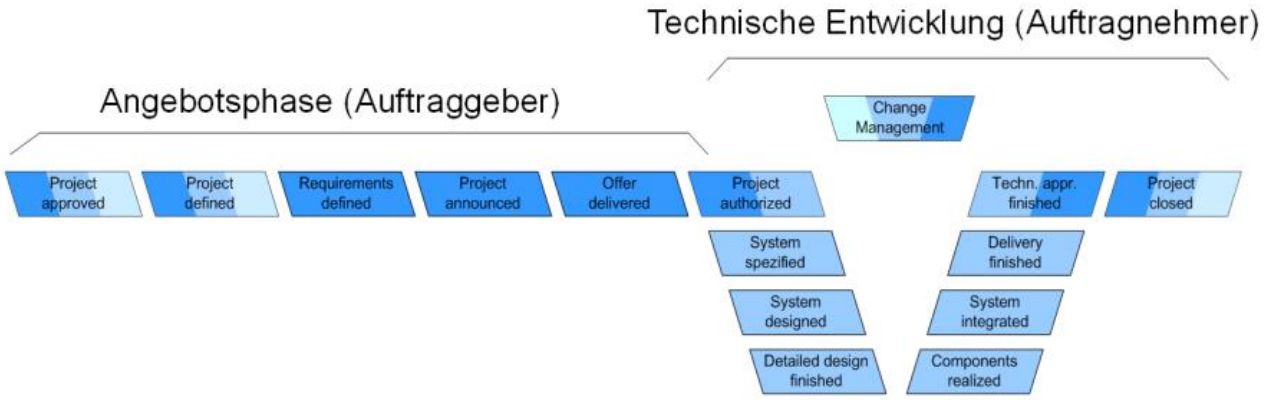
- Projekttypen vs. Projektgegenstand
- Vorgehensbausteine kapseln Produkte, Aktivitäten und Rollen
  - Verpflichtende Elemente (core elements)
  - Optionale Elemente (um individuelle Projektanforderungen erfüllen zu können)
- Unterstützung der Anpassbarkeit durch integrierte Tailoringmechanismen.

- Integrierte Methoden- und Toolunterstützung zur Erstellung von Produkten durch Aktivitäten und Rollen (verantwortlich für ein Produkt).
- Entscheidungspunkte (etwa Meilensteine) definieren einen Zeitpunkt, an dem eine Fortschrittsentscheidung getroffen wird.
- Projektdurchführungsstrategien definieren die Reihenfolge der im Projekt zu erreichenden Projektfortschrittsstufen (Sequenz von Entscheidungspunkten)
- Durch die Struktur des V-Modell XT ist eine Vergleichbarkeit zu herkömmlichen Prozessmodellen möglich (z.B. Konventionsabbildungen zu Prozessmodellen und Standards)

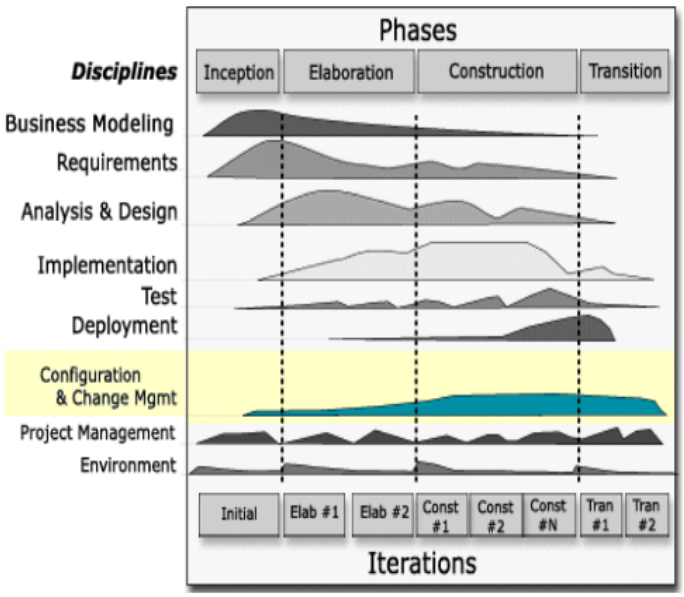


V-Modell XT in der Anwendung

- Flexible Anwendung des V-Modell XT durch Anpassung des Modells an unterschiedliche Projektgegebenheiten (Projekttyp, Projektmerkmale).
  - – Auswahl von benötigten Vorgehensbausteinen.
  - – Definition der passenden Projektdurchführungsstrategie.
- Werkzeugunterstützung (Open Source)
  - – V-Modell XT Projektassistent zur Anpassung des Modells an ein konkretes Projekt.
  - – Ergebnis ist eine angepasste Vorgehensweise und angepasste Templates für die Projektdokumentation.
  - – V-Modell XT Editor ermöglicht freie Konfigurationen des Vorgehensmodells, z.B. Anpassung auf ein Unternehmensmodell (Standard).
- Verpflichtendes Vorgehensmodell für öffentliche IT Projekte in Deutschland.
- Konventionsabbildungen ermöglichen die Kompatibilität zu Qualitätsmanagementstandards, wie CMMI und ISO 9000 sowie zu anderen Vorgehensmodellen, wie dem Rational Unified Process.



Rational Unified Process (RUP)



- Inkrementelle und iterative Vorgehensweise.

4 grundlegende Phasen:
- Inception (Beginn)
- Elaboration (Concept & Design)
- Construction
- Transition (Auslieferung)

Definierte Workflows und Disziplinen:
- 6 Engineering Workflows
- 3 Supporting Workflows
- Mehrere Iterationen innerhalb einer Phase.

- 4 grundlegende Phasen:
  - Beginn (Analyse, Anforderungen..)
  - Elaboration (Konzept & Design)
  - Construction (Implementierung)
  - Transition (Auslieferung)
- Beliebig viele Iterationen in jeder Phase – inkrementelle, iterative Vorgehensweise
- 6 Engineering Worklflows, 3 Supportig Workflows – eigene Disziplinien
- Vorteile
  - Real World Szenarien
  - Werkzeugunterstützung
  - Vordefinierte Liste mit erforderlichen Artefakten
- Nachteile
  - Hohe Komplexität
  - Hoher Dokumentationsaufwand
  - Anbieterabhängigkeit
- Einsatz
  - Große Projekte, die eine ganzheitliche Prozess-Sicht erfordern (inkl. Deployment).

Agile Prinzipien:

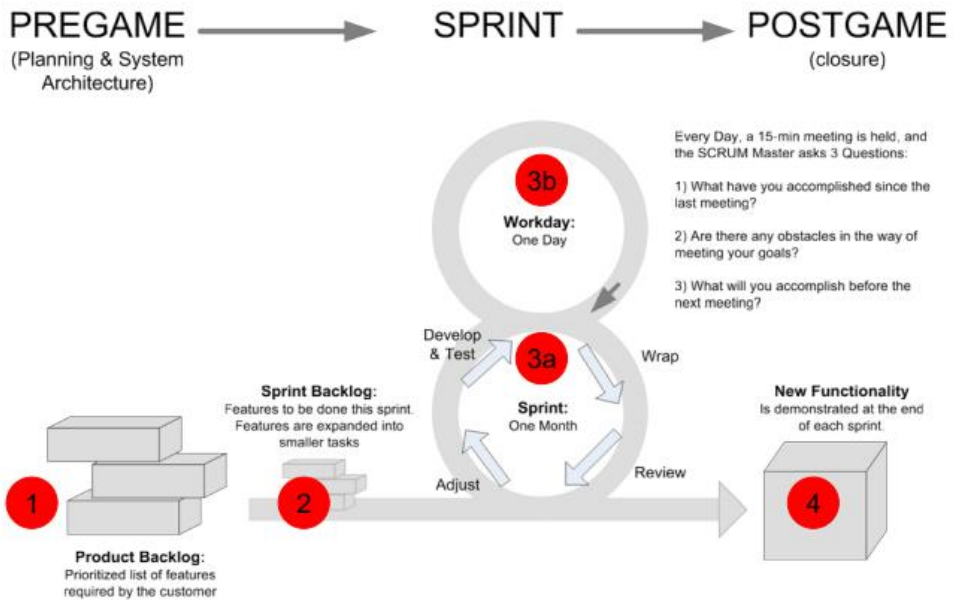
- 1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- 2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- 3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- 4. Business people and developers must work together daily throughout the project.
- 5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- 6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- 7. Working software is the primary measure of progress.
- 8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- 9. Continuous attention to technical excellence and good design enhances agility.
- 10. Simplicity--the art of maximizing the amount of work not done--is essential.
- 11. The best architectures, requirements, and designs emerge from self-organizing teams.
- 12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

SCRUM

- Flexibles Prozessmodell/Agiler Software Prozess aus Sicht des PM
- (Teil-)Produkte stehen dem Kunden frühzeitig zur Verfügung

Was leistet Scrum?

- Agiler Software Prozess aus Sicht des Projektmanagements (PM).
- Kleine aber hoch-effiziente Teams (auch mehrere Teams möglich).
- Flexibles Prozessmodell, um auf ändernde Anforderungen im Projektablauf reagieren zu können.
- (Teil-)Produkte stehen dem Kunden frühzeitig zur Verfügung.
- Das Projekt wird bestimmt durch Zeit, Wettbewerb, Kosten, und Funktionalität.
- Deliverables werden beeinflusst von Marktinformationen, Kundenkontakt und Skills der Entwickler.
- Hoher Bekanntheitsgrad in den letzten Jahren. § Hoher Erfüllungsgrad der agilen Prinzipien.



- Phasen
  - Pregame
    - Plannung/Festlegung der Systemarchitektur (Fundament)
    - Festlegung des Product-Backlogs
  - Sprint
    - Aus 4 Phasen, iterative (1 Sprint ~ 2 Wochen)
  - Postgame
    - Ablieferung
- Begriffe:
  - Product-Backlog: beeinhaltet die Anforderungen
  - Sprint-Backlog: die zu implementierenden Anforderungen in einem Sprint
  - Daily-Scrum: tägliches Teammeeting (ca 15. min), um Fragen/Missstände zu klären
  - Burndown Chart: Visuelle Darstellung des Projekts

Prozesstailoring – Anpassung eines generischen Entwicklungsprozesses an ein spezielles Projekt

- Ersetzen einzelner Prozesschritte (durch passende alternative Lösungen)
- Wiederverwendung von Best Practices
- Individuelle Anpassung des Projektplans

Prozess-Customization – Anpassung an Unternehmensstandards, Domänenabhängige Anpassungen

- verallgemeinerte Variante des Tailoring
- Schaffung von „unternehmensspezifischen Prozessen“, die als Unternehmensstandards dienen

B7: Software Pattern
Definition Pattern: „Eine Idee die einmal nützlich in einem praktischen Kontext nützlich war und dementsprechend für andere Fälle genauso nützlich sein könnte“
Bestandteile:

- Ein sinnvoller Name
- Motivation/Problembeschreibung
- Context
- Lösung
  - Struktur
  - Beteiligte
  - Beispiele
  - Implementation..
  - Konsequenzen

Vorteile für die Softwareentwicklung:

- Gängige Namen → weniger Diskussion
- Helfen beim Aufbau komplexer Systeme
- Minimiert Entwicklungszeit und kosten
- Verbessert Dokumentation

Nachteile:

- Anpassung an den Code
- Pattern sind simple, keine Lösung für komplexere Probleme
- „Pattern Overload“

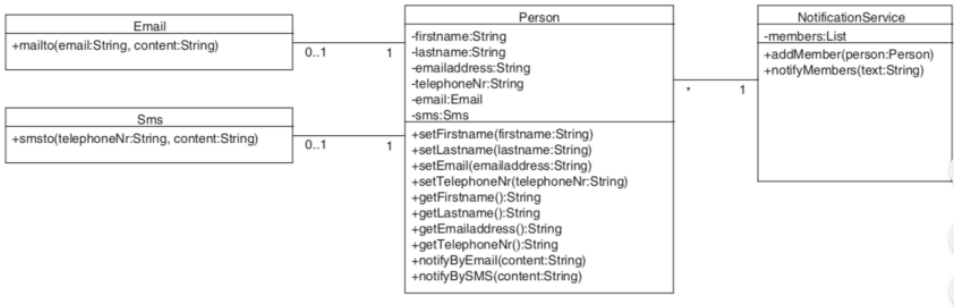
Einteilung(Klassifikation):

- Architektur-Patterns: Beschreiben die Struktur von Software-Systemen
- Design-Patterns: Beschreiben die Struktur/Beziehungen auf Klassenebene
- ProtoPatterns: Eine Lösung für einen Einzelfall, soll skaliert und für größere Anwendungen verwendet werden
- Antipatterns: Oft verwendet aber ineffektiv

Beispiele:
1) **Fundamental Patterns** – essentielle Konzepte der Software-Architektur
Interface Pattern: Trennung von Interface-Beschreibung und Implementierung
Delegation Pattern: Statt (mehrfaches) Vererben, extrige Klassen mit der Funktionalität

Delegation

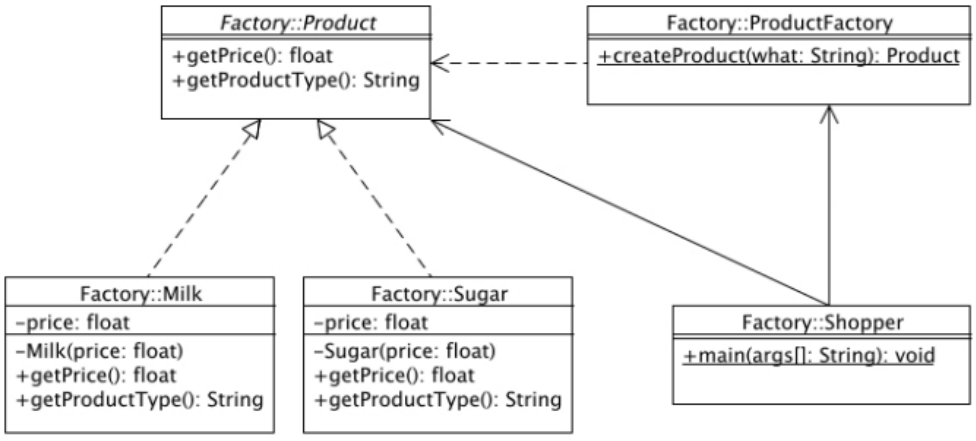
Outsource functionality into third class and use its instance via delegation



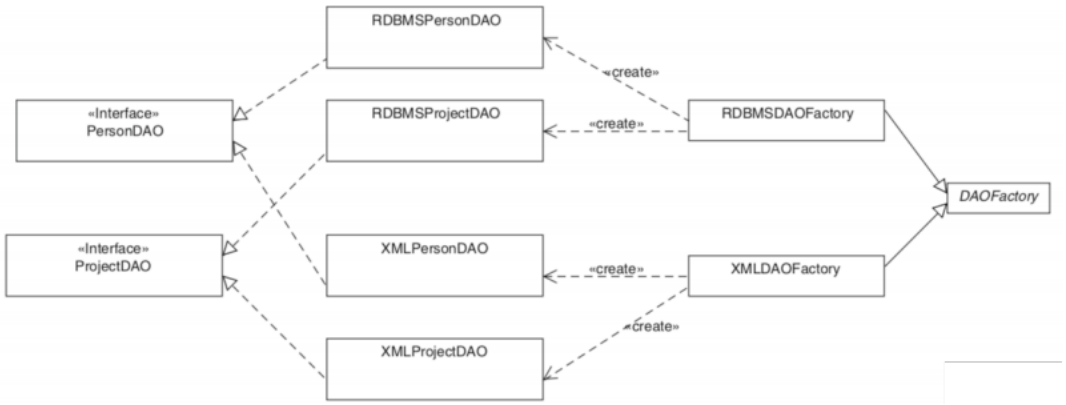
Immutable Pattern: Nach der Objekterzeugug soll das Objekt sich nicht mehr verändern (=Instanzvariablen)

2) **Erzeugerpatterns**(Creational Patterns) – Beschäftigen sich mit der Initialisierung und Konfigurierung von Klassen

Singleton: Nur eine Instanz der Klasse soll zur Laufzeit existieren
Factory: Objekterzeugung hängt von komplexen Anforderungen ab (z.B. weitere Subobjekte müssen initialisiert werden, komplexe Konfigurationsschritte)
- Objekterzeugung in eine andere Klasse delegieren

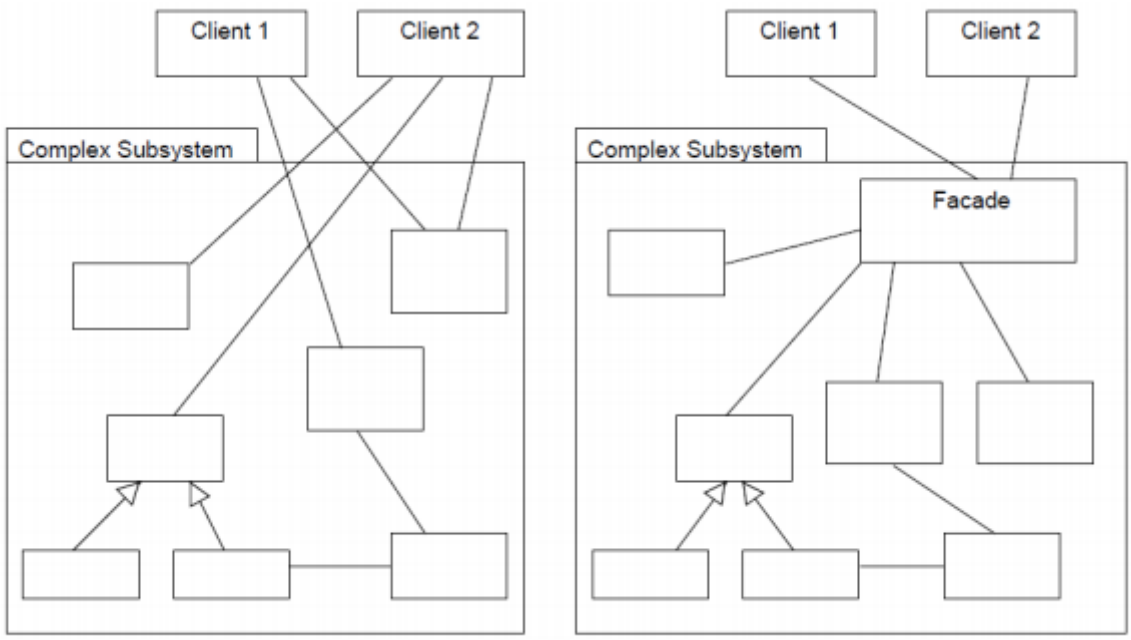


Abstract Factory: Funktioniert wie das Factory-Pattern, erreicht aber eine höhere Abstraktion indem beispielswesie für mehrere Factories mit einem ähnlichen Nutzen wiederum eine eigene, höher stehende Factory erzeugt wird. Factory für die Objekterzeugung ohne die konkrete Klasse zu definieren.

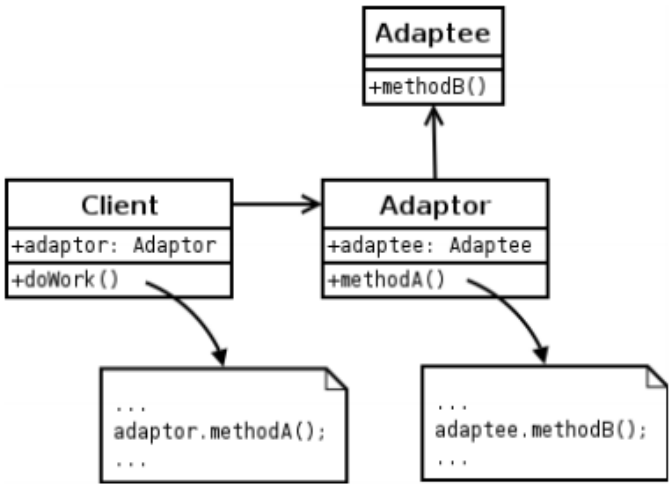

RDBMSPersonDAO
«create»
«Interface»
PersonDAO
RDBMSProjectDAO
«create»
RDBMSDAOFactory
DAOFactory
«Interface»
ProjectDAO
XMLPersonDAO
«create»
XMLDAOFactory
«create»
XMLProjectDAO

3) **Structual Patterns** – Wie können Klassen und Objekte zusammengesetzt werden, um größere Strukturen entstehen zu lassen

Facade(Facet): Die Facade realisiert Zugriffe auf das komplexe Subsystem. Die Clients müssen die innere Struktur des Systems nicht kennen → einfache Verwendung

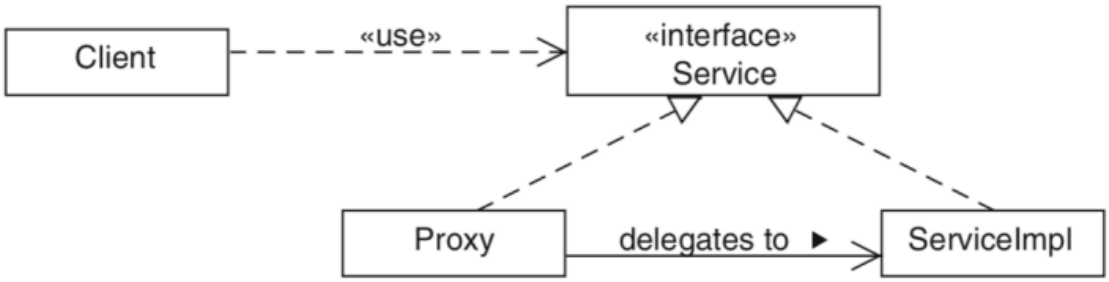


Adapter – Der Adapter ermöglicht es, externe inkompatible Funktionalität in das eigene System zu integrieren („Wrapper“). Transformiert (=Übersetzt) eine fremde Schnittstelle so, dass diese wie eine kompatible verwendet werden kann



Proxy – zusätzliche Funktionalität vor Methodenaufrufen

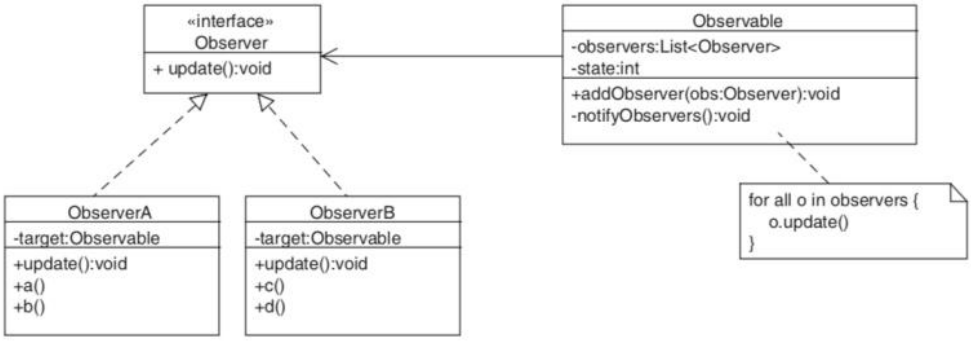
- Proxy agiert als repräsentatives Objekt und delegiert Methodenaufrufe an das konkrete Objekt



- Einsatz: Caching, Logging, Security

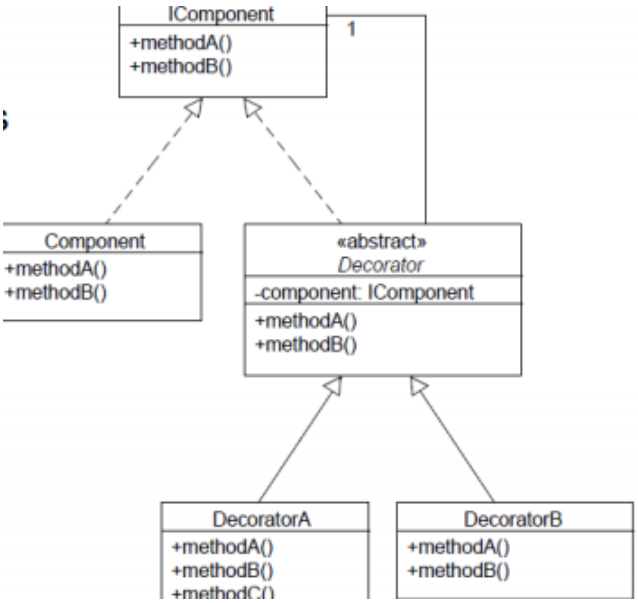
4) **Behavioral Patterns** – Umgang mit dynamischen Interaktionen zwischen Objekten

Observer – Auf Objektzustandsänderungen reagieren. Wenn Objekt A in den Zustand B wechselt soll eine entsprechend Aktion durchgeführt werden



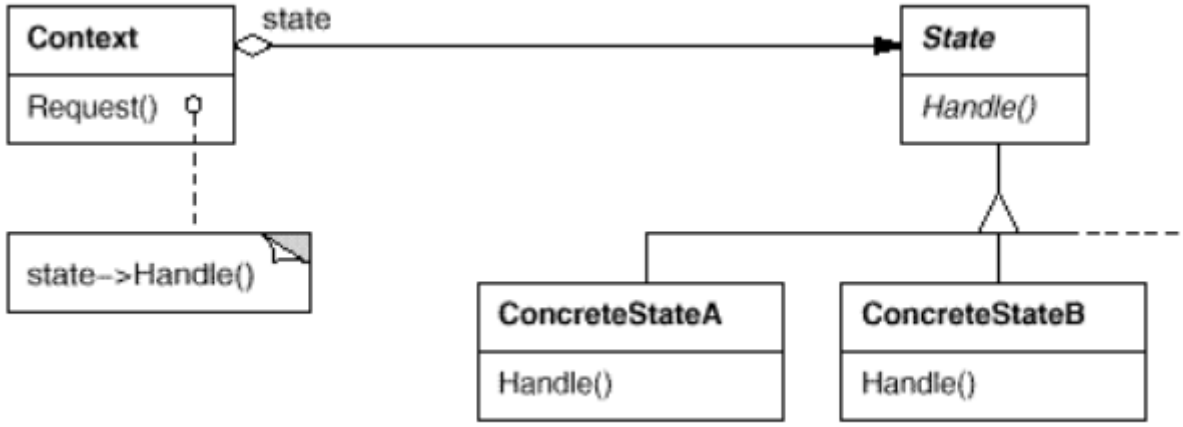
Decorator – Objektfunktionalität soll zur Laufzeit dynamisch erweitert werden.

- Problem: Testen, schwierig umzusetzen mit Proxy?



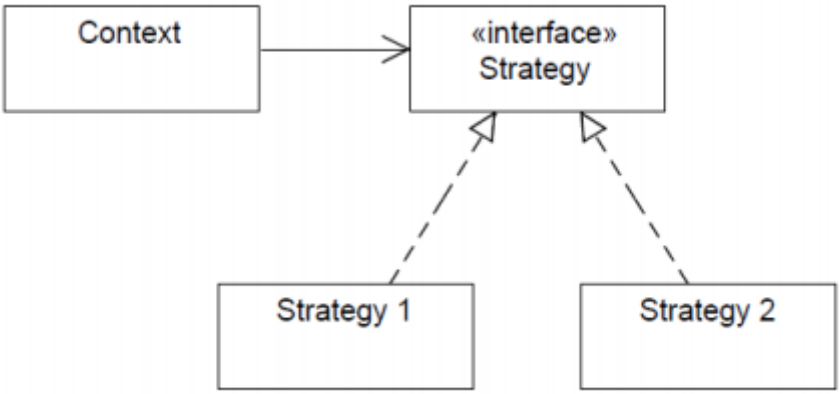
State – Objektverhalten (=Funktionen) soll aufgrund des aktuellen Zustands geändert werden

- Zustandsänderungen existieren demnach explizit
- Eventuell viele Subklassen (=Zustände)



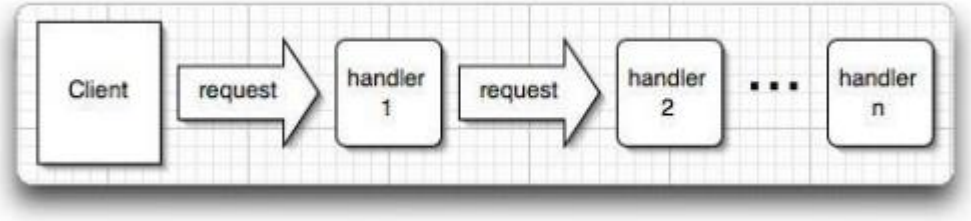
Strategy – Zur dynamischen Änderung von Strategien zur Laufzeit(bsp. Notification strategy)

- Algorithmen austauschen, hinzufügen



Chain of Responsability – Dient zur Verstärkung der losen Kopplung, die entsteht, wenn mehrere Processing-Operationen nacheinander durchgeführt werden.

- Besteht aus sogenannten:
  - Command Objekts (Anfragen)
  - Processing Objects (Filter)



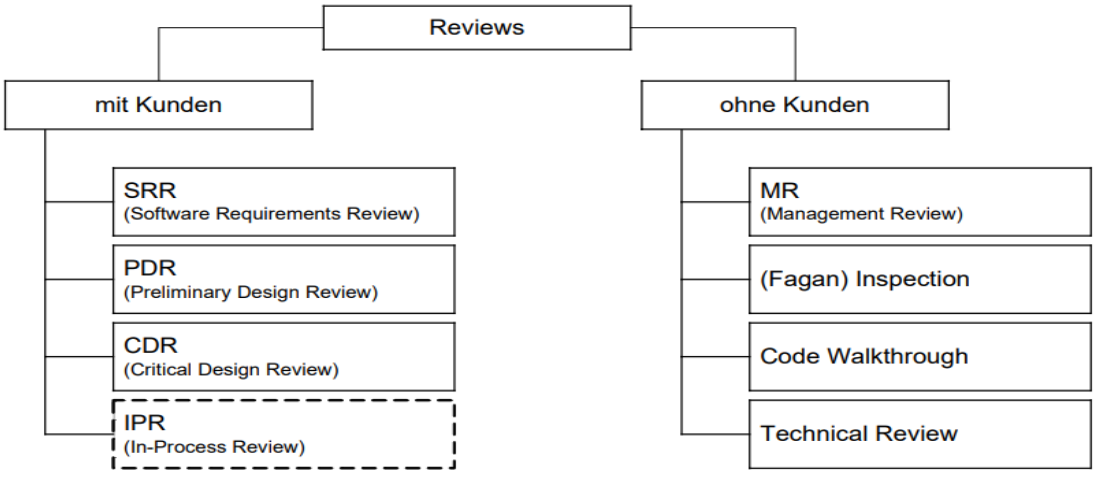
B8:
Qualitätssicherung und Testen:

- Qualität: die Eignung zur Erfüllung vordefinierter Anforderungen.
  - Muss während der Entwicklung sichergestellt werden
  - Ist keine Eigenschaft, die später hinzugefügt werden kann
- Qualitätssicherung: besteht in der Durchführung von Verifikation und Validierung in jeder Phase der Software-Entwicklung
  - Verifikation: Spezifikation vs. Umsetzung („Wurde das Produkt richtig entwickelt?“)
    - Beispiel: Komponententests - Prüfung gegen die technische Spezifikation
  - Validierung: Erwartung des Kunden vs. Umsetzung („Wurde das richtige Produkt entwickelt?“)
    - Beispiel: Akzeptanz- und Abnahmetests - Prüfung gegen Anforderungen.

Statische Methoden der QS:

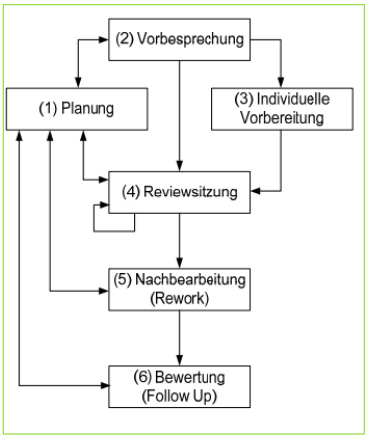
Ein **Review** ist ein formal geplanter und strukturierter Analyse- und Bewertungsprozess, in dem Projektergebnisse einem Team von Gutachtern präsentiert und von diesem kommentiert oder genehmigt werden

- Dient der qualitativen Bewertung von Produkten und Prozessen, die schwer quantitativ beurteilt werden können
- Tätigkeiten
  - Inspektion: Leser ist nicht Autor, dient zur Behebung von konkreten Mängel
  - Walkthrough: Leser ist Autor & Modera tor, dient zur Behebung von konkreten Mängel
  - Audit: Ziel: Überblickende Kontrolle, Projektmitarbeiter nur als Informationslieferanten
- Arten:
  - Mit Kunden
    - Software Requirements Review
    - Critical Design Review
  - Ohne Kunden
    - Management Review
    - Technical Review



-

- Rollen (Reviewgröße 3 bis 6 Personen)
  - Moderator (Leiter)
  - Leser
  - Gutachter (Kommentierung)
  - Schreiber (Protokoll)
  - Autor (keine Kommentierung und Rechtfertigung)
- Ablauf:
  - Planung: Objekt, Prüfziele, Teilnehmer usw. festlegen
  - Vorbesprechung: Vorstellung des Prüfobjektes
  - Intensive Einzelarbeit
  - Durchführung: Gemeinsames Lesen, Aufzeigen von Mängeln, keine Korrektur
  - Nachbearbeitung: aufgezeigte Mängel korrigieren
  - Berichtserstattung



- Richtlinien – technische Reviews
  1. Das Produkt reviewen, nicht den Autor.
  2. Verwenden Sie einen Arbeitsplan.
  3. Diskussionen sachlich und kurz halten.
  4. Problembereiche identifizieren, aber nicht jedes Problem gleich lösen.
  5. Schriftliche Aufzeichnungen führen.
  6. Anzahl der Teilnehmer begrenzen; gute Vorbereitung aller Teilnehmer.
  7. Für jedes Produkt eine passende Checkliste verwenden.
  8. Ausreichende Ressourcen und Zeitbudget zur Verfügung stellen.
  9. Vorab Training für alle Reviewer.
  10. Reviews im Nachhinein beurteilen für Verbesserung der Reviews.

Überprüfung der Anforderungen

- Kriterien für brauchbare Anforderungen
  - Notwenige Kriterien
    - Testbarkeit: Erfüllung einer Anforderung durch ein System muss eindeutig testbar sein (Messvorschrift: was ist wie zu messen? Z.B. Dauer eines Ablaufs)
  - Erstrebenswerte Kriterien
    - Klare, knappe Beschreibung
    - Umsetzbarkeit
    - Konsistent mit bekannten Interessen der Stakeholder(richtig, komplett, relevant)
- Wichtig: Darstellung, die eine systematische Bearbeitung unterstützt
  - Vor der weiteren Verwendung (z.B. von Prosatext) kann eine Transformation die Verwendung der Anforderungen erleichtern; z.B. Herstellung einer Liste von Anforderungen.

Überprüfung statischer Aspekte

- Anwendungsfälle
  - Software Requirement Review
  - Jeder Anwendungsfall muss in den Anforderungen beschrieben werden
  - Akteure + Operationen in den Anforderungen führen zu den Anwendungsfällen
- Klassendiagramme

- Preliminary Design Review
- Die Daten welche in den Anforderungen zu finden sind werden von Model-Klassen gekapselt
- Operationen + Entities in den Anforderungen führen zu Klassen
- Aber auch: GUI Klassen, Support-Klassen für Internationalisierung, ...

Überprüfung Datenbankrelation

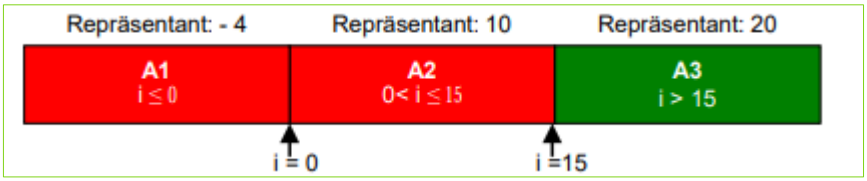
- Anforderung strukturiert darstellen, z.B. als Liste
- Anforderungen mit ER-Model reviewen und konsistent machen
- ER-Modell systematisch durchgehen
  - Entitäten und Beziehungen finden
  - Schlüsselattribute auf Eindeutigkeit überprüfen
  - Attribute: Typen und Optionen überprüfen
  - Integritätsbedingungen zu Attributen überprüfen
- Datenbanktabellen systematisch durchgehen
  - Nicht markierte Elemente auf Sinnhaftigkeit überprüfen

Testansätze, Test-Driven-Development
Testfälle bestimmen(spezifizieren):

- Ziel aus dem Testplan bestimmen:
  - – Z.B. Normal-, Sonder- und Fehlerfälle in einem Anwendungsszenario abdecken
  - – Z.B. Alle Knoten und Kanten in einem Kontrollflussgraphen abdecken
- Fälle bestimmen:
  - Eingabeparameter
  - Entscheidungen im Ablauf
- Erwartetes Ergebnis bestimmen

Ausgewählte Testtechniken:

- Äquivalenzklassen – Für Eingabedaten aus einem bestimmten Wertebereich wird, zur Reduktion der Testanzahl, ein repräsentativer Wert ausgesucht. Dieser soll den gesamten Wertebereich abdecken.



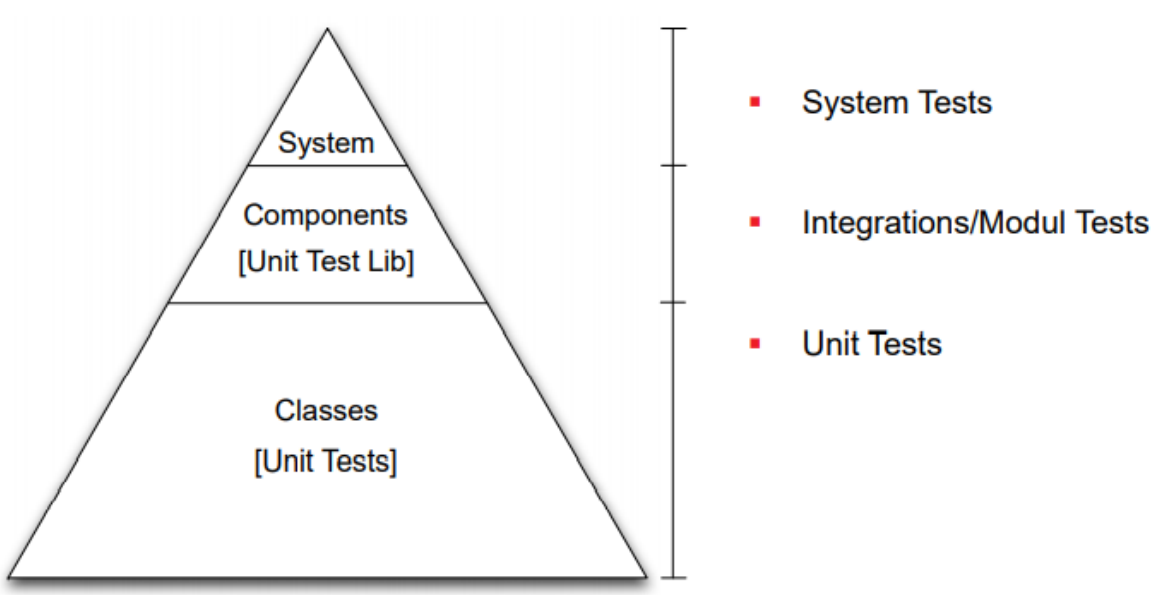
- Grenzwertanalyse: Erweiterung der Äquivalenzklassenzerlegung für bessere Überdeckung. Grenzwerte werden als Klassenrepräsentanten ausgewählt. Je Klassengrenze ein Testfall

Value Based Testing:
Aufgrund von unterschiedlicher Nutzung eines Features, ist ein Feature mehr „wert“ als das andere, somit ist dieses eher zu testen

- Requirements-Based Testing:

    - Welche Anforderungen bringen dem Kunden die meisten Nutzen?
- Risk-Based-Testing:
    - Welche Risiken gefährden das Nutzen?
- Test case selection techniques:
    - Welche Testfälle adressieren dieses Risiko?

Testen – Gesamtsystem
Die Fläche des Pyramidenabschnitts ist proportional zu der Anzahl der jeweiligen Tests welche bei Auslieferung vorhanden sein sollten.



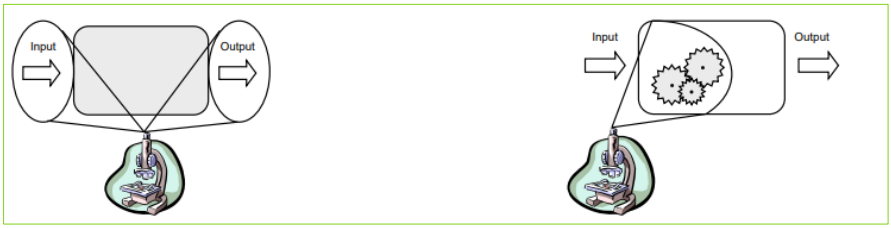
Testfallbeschreibung (Testfalldokumentation):
- Nr
- Typ: NF, SF, FF
- Beschreibung: Komponente im System und eine Funktion
- Vorbedingungen
- Eingabewerte: konkrete Parameterbeschreibung
- Aktionen: Aktivitäten zur Eingabe der Werte
- Erwartete Ergebnisse: Zustände/Ausgabeparameter
- Ergebnisse: Abweichungen zum tatsächlichen Ergebnis
- OK/NOK

Testarten
- **Unit Test**: Fokus auf Komponenten und Prüfung auf Übereinstimmung zwischen der Umsetzung der Komponente und der technischer Spezifikation.
    - Erlauben eine genaue Lokalisierung und frühe Erkennung von Fehlern
    - Automatisierung essentiell
- **Modultests**: Fokus auf die Überprüfung eines Moduls gegen die technische Spezifikation. Sonst ähnlich wie die Unit-Tests

- **Integrationstests:** Fokus auf Interaktion zwischen Modulen, zusammenführen von Modulen durch Big-bang, etc. Inkrementelles testen ist vorzuziehen
- **Systemtest**: Fokus auf Übereinstimmung zwischen funktionalen und technischen Bedingungen des Gesamtsystems (nahe an der Zielplattform).
- **Regressionstest:** Testen von geänderten Komponenten. Dadurch sollen Fehler, die durch Änderungen, z.B. auch durch Fehlerkorrektur, entstanden sind, verhindert werden.
- **Akzeptanztes**t: Übereinstimmung der festgelegten Anforderungen in der Zielumgebung des Kunden.
- **Installationstest**: Identifizieren von Fehlern während der Installation.

Teststrategien:

- **Black Box Tests:**
  - Anforderungen/Spezifikation als Grundlage
  - Unabhängig von der Realisierung der Module
  - Data-driven
  - „Anforderungsüberdeckung“
  - Keine genaue Fehlerortung möglich
  - Äquivalenzklassen von Eingabedaten?
- **White Box(Glass Box) Tests:**
  - Software Code als Grundlage
  - Wissen über den Aufbau notwendig
  - Logikgesteuert
  - „Kontrollflussüberdeckung“
  - Ermöglicht Fehlererkennung und -lokalisierung.
  - Äquivalenzklassen von internen Schleifen/Verzweigungen

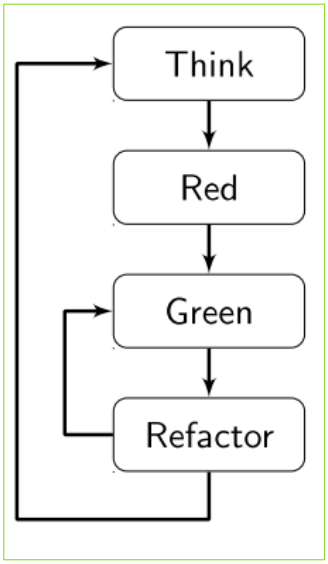


Testfalldokumentation – Zweck:

- Information für den Entwickler
- Berichtserstattung (Einschätzung der Produktqualität)
- Wiederholbarkeit
- Testfälle als Kommunikationswerkzeug

Test-Driven Development:
Prozess:

- Think: Spezifikation des Tests
- Red: Implementierung des Tests → Test schlägt fehl
- Green: Implementierung der zu testenden Klasse/Komponente → Test ist erfolgreich
- Refactor: Veränderung der Implementation → Test sollte nie wieder fehlschlagen



Best Practices:

- Gute Rollenverteilung im Team
- Viele Unit-Tests → Vereinfachen Integration Tests
- Testen der Objekte so, wie sie im Programm verwendet werden

B10 – People Management

Motivation:

- extrinsische – durch äußere Zwänge verursachte Motivation
- intrinsische – durch die Anreize einer Aufgabe

Prozesstheorien – geben eine Erklärung, wie das Verhalten gesteuert wird
Individuelle Einflussfaktoren des Motivationsprozesses:

- Anspruchsniveau
  - Erwartung an die eigene Leistung und deren Belohnung
  - Wenn oft erreicht → Niveau wird angehoben (von sich selbst)
- Persönliche Einstellung
  - Grundhaltung gegenüber der Organisation, der Problemstellung, dem Umfeld usw.
  - Negative Einstellungen → führen oft zur Frustration
- Subjektive Wahrscheinlichkeit
  - Abschätzung der Chancen, Erfolg zu haben

Folgerungen der Prozesstheorien:

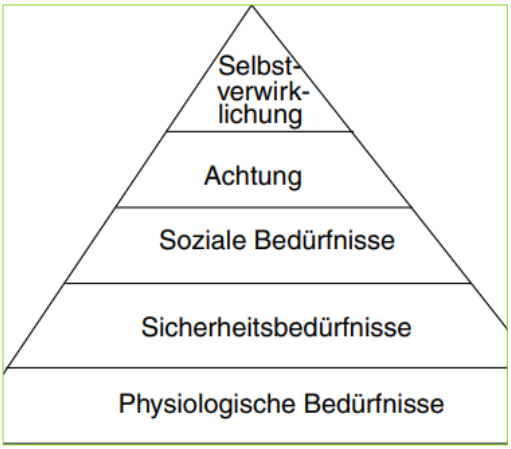
- Jeder Mitarbeiter = Individuum
- Mitarbeiter nicht 1zu1 austauschbar
- Der Umgang mit den Personen soll die Tagesverfassung berücksichtigen
- Mitarbeiter lernen aus Erfahrungen und richten ihr zukünftiges Verhalten danach

Inhaltstheorien – Beschreiben Bedürfnisse, Antriebe und Ziele der Menschen

- Monothematische Theorien
  - Ein einzelnes Motiv ist ausschlaggebend für zielgerichtete Dynamik des Menschen (Bsp.: Freud: Sexualtrieb, Adler: Minderwertigkeitskomplex)
  - In Bezug auf die Arbeitszufriendenheit heute nicht mehr brauchbar
- Polythematische Theorien
  - Mehrere Grundmotive beeinflussen und bestimmen das Verhalten des Menschen
  - Unterschiedliche Modelle (Maslow-Pyramide, ERG-Modell, zweidimensionale Arbeitszufriendenheitstheorie von Herzberg etc.)

Maslow Pyramide: Wenn eine Stufe in einem gewissen Maß befriedigt ist, wird der Fokus auf die nächste Stufe gelegt

(1) – Lebensnotwendige Bedürfnisse (Schlaf..)
(2) – Absicherung der Person hinsichtlich Schutz des Lebens, Schutz vor Arbeitslosigkeit etc.
(3) – Die Eingliederung in eine Gruppe und deren soziale Akzeptanz
(4) – Selbstachtung/Fremdachtung
(5) – Das Verlangen sich zu verwirklichen d.h. potentielle Fähigkeiten entfalten



Produktivität

- Die Produktivität wird üblicherweise als Verhältnis von erbrachter Leistung zu aufgewendeter Zeit gemessen

- Beliebte Maßnahmen zur Produktivitätssteigerung sind:
  - Personal zu mehr Arbeit anhalten
  - Prozess der Produktentwicklung automatisieren
  - Qualitätsmaßstäbe für das Produkt modifizieren (i.e. reduzieren)
  - Vorgehensweisen standardisieren

- Diese Maßnahmen sind (langfristig) kontraproduktiv
  - Sie führen zu verringerter Arbeitszufriedenheit
  - Die Folge: erhöhte Fluktuation

Produktivität:
Faktoren ohne Einfluss auf die Produktivität:

- Programmiersprache
- Berufserfahrung
- Anzahl der Fehler
- Gehalt

Faktoren mit Einfluss auf die Produktivität:

- Teampartner
- Arbeitsplatz (Qualität)

Der U-Faktor: Ruhe am Arbeitsplatz
„In Fahrt" – Der Zustand, in dem man voll und ganz auf die Arbeit konzentriert ist
Umweltfaktor = ungestörte Stunden / Stunden körperlicher Anwesenheit

Auswahl von Mitarbeitern:

1. Job Specification – Anforderungen des Jobs
2. Job holder profile – Anforderungen an die Person (Qualifikationen etc.)
3. Bewerbungen einholen
4. Auswertung der Bewerbungen – Ungeeignete Kandidaten frühzeitig ausscheiden
5. Auswahl der Mitarbeiter
   - Portfolio – Produkte ihrer bisheriger Tätigkeiten
   - Eignungstests
   - Anhörungen – Aspekte ihrer bisheriger Tätigkeiten erzählen
   - Interviews

Teambildung verhindern:

- Defensives Management: Entscheidungen trifft nur das Management
- Bürokratie: sinnloses Produzieren von Projektdokumenten, hält die Teammitglieder auf
- Physikalische Trennung: räumliche Trennung
- Zersplitterung der Zeit der Mitarbeiter: Aufteilung der Mitarbeiter auf mehrere Projekte
- Scheintermine: unrealistische Termine → geringe Identifikation mit dem Projekt
- Cliquenkontrolle: Veränderung der Teamstruktur

- Qualitätsreduktion der Produkte: (wegen Konstenreduktion) bewirkt dass Teammitglieder eine geringe Identifikation mit dem Projekt aufweisen

Formen der Teamorganisation:

- Klassisch hierarchisch
- Matrix-Organisation
- Chef-Programmierer Team
- Offen strukturierte Teams
- SWAP
- XP

Richtlinien zur Teamorganisation:

- Von Leuten lösen, die nicht ins Team passen
- Aufgaben verteilen, die an die Fähigkeiten der Mitglieder anpasst sind
- Mittelfristig auf die persönliche Entwicklung der Mitarbeiter achten
- Balancierte/Harmonische Mischung
- Aufgaben verteilen, die an die Interessen der Mitglieder anpasst sind