KARLSRUHER INSITUT FÜR TECHNOLOGIE

Softwaretechnik II
Zusammenfassung WS17/18

Manuel Lang

2. März 2018

1 VORGEHENSMODELLE

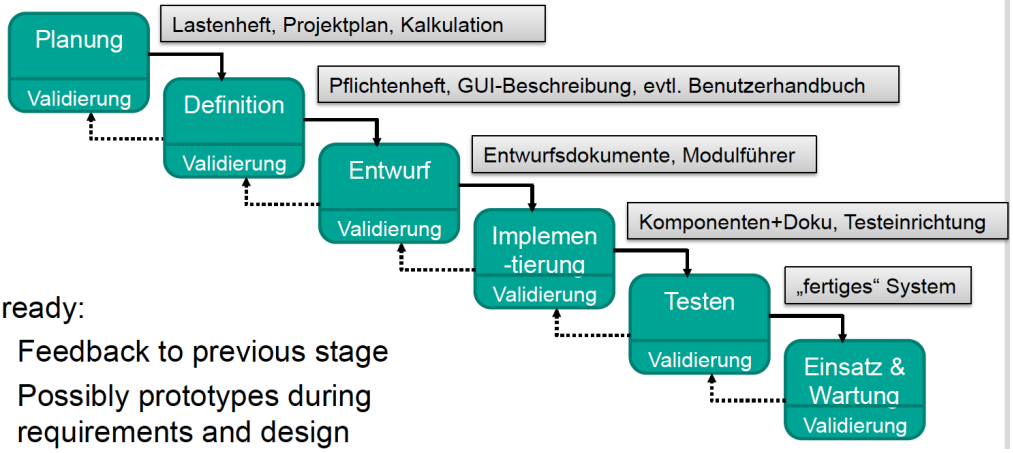
Vorteile strukturierter Vorgehensmodelle

- Reproduzierbarkeit (Erfahrungen für ähnliche Projekte)
- Skalierbarkeit (größere Komplexität schneller)
- Wiederverwendbarkeit (Code)
- Risikominimierung (Entiwcklung nach Plan)

Modelle

- Softwareentwicklung liefert nicht nur Code, sondern auch Deployment-Deskriptoren (Zusazudokumente), ursprüngliche Anforderungen (Code -> Anforderungen funktioniert nicht), welche Produkte wann und wo?
- Wasserfall-Modell

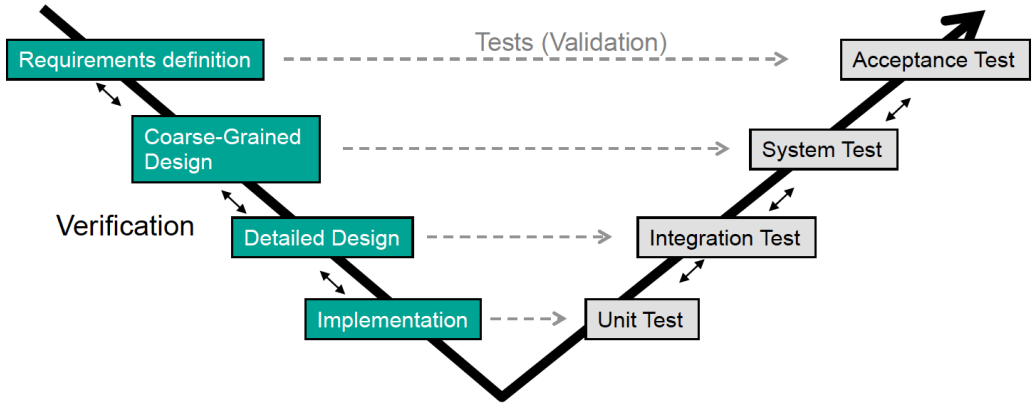
- The Waterfall Model as a sequential process model …



Already:

- Feedback to previous stage
- Possibly prototypes during requirements and design
  - Erstes Modell zur Definition der verschiedenen Phasen
  - überhaupt machbar?
  - Welche Stakeholder?
  - Lastenheft/Pflichtenheft
  - etc.
  - Problem: sehr steife Reihenfolge, klingt logisch, aber Phasenübergang ist in Realität unklar
  - Feedbackzyklen nötig, aber Unterscheidbarkeit der Phasen trotzdem schwierig
  - Zu langes Vorplanen (Airbus 10 Jahre?!) sehr schwierig, Was kann in der Zwischenzeit passieren? Niemand weiß das, daher kann Wasserfall langfristig nicht geplant werden (lange Planbarkeit nur sehr selten gegeben)

- V-Modell



➔ Can be seen as (just) an explanation how activites relate to each other.

  - sieht ähnlich aus wie Wasserfall
  - besagt welche Artefakte man hat + erklärt Zusammenhänge zwischen Dokumenten
  - überprüft bspw. Zusammenhang zwischen Implementierung und Design (Verifikation, zeigt Abwesenheit von Fehlern, All-Quantor funktioniert immer)
  - Z.B. mehrere Module zusammen testen (Validierung, nicht Verifikation, Existenz-Quantor, Test-Fall der funktioniert)
  - Hauptbotschaft: kein notwendiger Wasserfall: Artefakte + Zusammenhänge (Validierung + Verifikation)

- 6 grundlegende Phasen in jedem SE-Projekt
  - Planung
  - Definition
  - Design/Entwurf
  - Implementierung
  - Testen
  - Betrieb
  - Wartung

- 3 Dinge über die ein SE-Modell Aussagen trifft
  - Welche Rollen?
  - Welche Aktivitäten?
  - Welche Produkte?

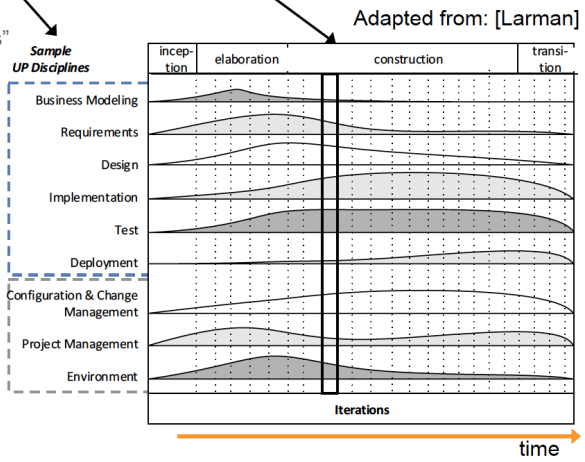
- Probleme des Wasserfall-Modells
  - Sehr lange Zeiträume
  - inflexibel (schlecht auf Änderungen reagierbar)

- Alternativen?
  - Inkrementeller Ansatz (nicht zwangsweise flexibel bei geänderten Anforderungen, sondern große Komplexität: mehrere Zyklen für kleinere Teilprojekte), viele kleine kürzere Wasserfälle
  - Evolutionär: Bei "Fertigstellungßurückspringen z.B. in Planungsphase, in der Praxis fast immer da z.B. langzeitig Änderungsbedarf auftritt
  - Gesamtsystem wird nach und nach gebaut
  - Integrationstests, immer neue Inkremente für Kunden für schnelles Feedback
  - Konzepte der agilen Entwicklung schon älter

- Spiralmodell

- Idee: 4 Quadranten, Zielfestlegung, Bewertung, Validierung, Planung, immer im Kreis um Irrglauben Software ist irgendwann fertig auszuräumen
- Nicht inkrementell, sondern evolutionär

- UP (Unified Process)
  - The UP defines –
    - 4 abstract phases
      - to be concluded with a milestone
      - *not equivalent to waterfall phases!!!*
    - 9 disciplines
      - 6 engineering + 3 supporting
      - here we find our "mini waterfalls"
  - UP is supposed to be –
    - **iterative und incremental**
    - **risk-driven**
    - **client-driven**
    - **architecture-centric**



- zuerst UML: vereinheitlichte Notation, z.B. Symbole für Widerstände und Transistoren, Industriestandard
- Folge: vereinheitlichte Prozesse, also UP, aber komplett verschieden zu UML (einheitlich), aber es gibt nicht den einen richtigen Software-Prozess, hängt von vielen verschiedenen Faktoren ab (z.B. eingebettete Software), also Rahmenwerk für Prozesse
- Verschiedene Phasen: Inception, Elaboration, Konstruktion, Transition über Zeitachse
- Verschiedene Disziplinen wie früher Phasen, auch mit Deployment, insgesamt moderner, andere Disziplinen wie Änderungsmanagement, Projektmanagement, Environment (Entwicklungsumgebung) = Aufrechterhalten der Produktivumgebung + Zusammenspiel der Komponenten + Patches
- iterativ und inkrementell (streng nicht dasselbe, iterativ mehrere Schleifen durch Prozess, inkrementell heißt neues Teil), quasi Synonyme
- Risiko-getrieben, sehr früh auf Risiken reagierbar
- Client-driven: Was will der Auftraggeber?
- Architektur-zentriert: zentrales Dokument wovon andere Aktivitäten abhängen
  * Inception (Anfangsphase): Scope, Business cases?, machbar?, auf Markt?, Kostenschätzung (schwierig so früh, also sehr grob)

        * Elaboration (Ausarbeitung): Besonders risikobehaftete Teile werden zuerst betrachtet, Anforderungen verstehen, Übergang in Konstruktionsphase, klappt gut mit sehr erfahreren Entwicklern die Abstraktionsebenen einfach wechseln können, sonst problematisch
        * Konstruktionsphase
        * Übergabephase (Test, Deployment)
    - Disziplinen
        * Business Modelling: Technische Konzepte
        * Anforderungen: Anforderungsanalyse, Dokumentation
        * Entwurf: Lösungsorientiert, aber kleine klare Grenze zu Anforderungen, nicht nur zeitlich sondern auch konzeptionell überlappend
        * keine klaren Definitionen!

- Rational Unified Process (RUP)
    - 6 best practices (sehr generelle Aussagen): iterativ entwickeln, Anforderungen verwalten, Komponenten verwenden, Software grafisch visualisieren, Softwarequalität prüfen, Änderungen unter Kontrolle haben
    - Was konkret in welcher Phase wurde definiert, konkrete Aufgaben, konkret welche Artefakte
    - verschiedene Rollen für verschiedene Disziplinen definiert, Anpassungen für konkrete Projekte
    - prinzipiell genauere Ausarbeitung des UP

- Rollen
    - Verantwortung abgekoppelt von Person, Reussner Prof. + Vater + Vorstand
    - eine Person kann mehrere Rollen annehmen
    - kann aber zu viel Verschnitt führen (viele stehen rum und einer arbeitet, d.h. Gefahr, dass jeder in seiner Rolle bleibt, aber nicht flexibel ist)

- Zusammenfassung
    - iterativ
    - UP: Phasen + Disziplinen
    - RUP: Aktivitäten, Artefakte, Rollen, Rahmenwerk für Projekte
    - nicht ein gültiger Prozess

- Agile Methoden
    - kein Allheilmittel

  - Manifest für agile Softwareentwicklung: Individuals/Interactions > Prozesse, Software > Doku, Kundenzufriedenheit > Vertragsverhandlungen, Flexibilität > Plan folgen
  - Ist das tatsächlich nötig oder wird Feindbild aufgebaut? Leichter wogegen als wie besser, daher Kritik dass eh niemand so stur ist
  - Quintessenz: nicht beliebig planbar (Änderungen kommen immer), kontinuierliche Beobachtungen und schnelles Feedback (bau ich was Kunde will? ist Qualität gut?)

- Extreme Programming (XP)
  - Programmieren = zentral, alles andere außen rum
  - innen Entwickler: einfache Entwürfe für aktuelle Anforderung, Realisierung mit Pair Programming (verdoppelt Kosten, nicht immer gerechtfertigt, verbessert Qualität auch nicht besser als andere Review-Formen), testgetriebene Entwicklung (immer testbare Software + i.d.R. besser testbare Schnittstellen), Refactoring (Anpassen des Entwurfs für weitere Anforderungen)
  - außenrum Team: Continuous Integration (keine großen Aufwände bei Builds), Collective Ownership (jeder für alles verantwortlich, kann aber auch fehlschlagen, daher Gesamtprojektverantwortung), Coding Standard
  - äußerster Kreis: kleine Releases für schnelles Feedback, Kunden testen mit, Planungsspiele (Aufteilen der Inkremente)
  - Kritik: ad-hoc Prozess, schwer replizierbar, schlechte Doku, nicht wiederverwendbare Software, Kunden müssen eingebunden werden (will Kunde das?), vieles nicht wissenschaftlich validiert (z.B. Pair Programming), TDD kann problematisch sein
  - endet in Praxis oft in Code & Fix

- Scrum
  - Rahmenwerk mit notwendigen Anpassungen, aber ziemlich elaboriert
  - Name von Rugby
  - Product Owner verantwortlich für zu realisierende User Stories und landen als Sammlung in Product Backlog
  - Inkremente (Sprints): Planning Meeting Product Backlog in Sprint Backlog, Dauer 2-4 Wochen aber konstant
  - Daily Scrum: täglicher Projektfortschritt, Burn Down Chart (Liste der offenen TODOs abgearbeitet)
  - Scrum Master trainiert das Team, sorgt dafür, dass sich Entwickler auf das Entwickeln konzentrieren können (also eine stabile Arbeitsumgebung)
  - Sprint Review Meeting: Wie war die Qualität?
  - Retroperspective Meeting (Ende vom Projekt): Was hätte man besser machen können? Lehren?

- Rollen (pig rolls treiben voran - essentiell): Product Owner (Kundenstellvertreter, vergleichbar mit Architekt), Scrum Master (Hindernisse ausgeräumt, verantwortlich dass Vorgang läuft, damit Entwickler sich auf Rolle konzentrieren können), Team (selbstorganisiert, kümmert sich um Produkt, keine Hierarchie wie bei XP, etwa 7 Leute meist), (chicken rolls - nicht essentiell) Stakeholder, Manager, etc., dürfen Pigs nicht sagen wie sie ihre Arbeit machen
- oft TDD
- Product Backlog: Sammlung aller Anforderungen, dynamisches Dokument, Priorisierung der Features (z.B. wie stark wird Architektur beeinflusst, wie stark sind Risiken)
- Projektplanung: nach jedem Sprint kann etwas ausgeliefert werden, damit schnell Feedback erlangt werden kann, Planung auf 3 Ebenen: Release, Sprint, Arbeitstag
- Sprint Backlog: zerbröselter abstrakter Product Backlog für aktuellen Sprint in konkrete Arbeitsaufgabgen, Absprache mit Kunde vlt. nötig, Kategorien: to do, in progress, finished, aber Definition of done?, Sprint Backlog in Product Backlog ist nicht vorgesehen, aber möglich
- Sprint Backlog füllen: User Stories vom Product Backlog mit Product Owner und Team Membern diskutiert, oder vielleicht erst Prototyp nötig? Aktivität sollte nicht länger als 2 Tage sein, damit guter Überblick möglich, Abschätzen in Personenstunden, keine Puffer sondern präzise schätzen bspw. durch Time-Box (z.B. 2 Tage nehmen und unter Umständen neu planen)
- Wie viel ist genug? Geplant wird weniger, bspw.85% der Kapazität und zusätzlich etwa 25% Abzug (Meetings, Urlaub, Krankheit)
- Burn Down Chart (Abarbeitung des Product Backlogs über verschiedene Sprints, Infos über Arbeitsgeschwindigkeit zur besseren Planung)
- Kritische Bewertung
  * Personen/Rollen: Entwickler müssen nahe beisammen sein, kann nur gut klappen wenn sich alle Leute dran halten (benötigt viel Disziplin), effiziente Kommunikation ist wichtig, klappt aber nicht mit vielen Leuten
  * Artefakte: keine Dokumentation vorgesehen (nur wenn explizit geplant), Code + Testfälle (reicht oft nicht)
  * Dokumente: -
  * Aktivitäten: keine Phase zu Architekturentwurf (in Praxis oft Sprint 0 - Architektur)
  * Skalierbarkeit: sehr schlecht, weil nur für wenig Leute vorgesehen
  * Architektur: nicht per se vorgesehen
  * Qualitäts-kritische Software: per se kein Grund wieso nicht agil, aber sehr gute Qualitätssicherung für Sonderfälle sehr wichtig (z.B. Abschaltung Atomkraftwerk), die normalerweise nicht Teil von Scrum sind

    * große Projekte: Brook's Law (spätes Hinzufügen von Leuten problematisch), daher eher zeitliches Aufsplitten von Team in mehrere Scrum-Teams oder direkt verschiedene Teilteams von Beginn und Scrum of Scrums als Gesamtmeeting
    * verteilte Entwicklung: Daily Scrum problematisch, Scrum Master immer nah beim Team, Product Owner, wenn sich Leute persönlich kennen auch hier vorteilhaft für Kommunikation, Verteilung während des Projekts kann funktionieren
    * keine silver bullet, passt bei vielen Projekten, funktioniert aber nicht immer (höchste Qualität, große Gruppen, ...), benötigt viel Arbeit und Disziplin (nicht ad-hoc!), Prozess ansich ist trivial
    * Distanz zu Code & fix wichtig

2 Requirements Engineering

- 48% aller gescheiterten Software Projekte wegen fehlerhaftem RE
- IEEE-Standard für Requirement: Fähigkeit die benötigt wird um ein Problem zu lösen
- Requirements werden so formuliert, dass...
  - prüfbar
  - präzise (Balance für Entwickler und Auftraggeber)
  - adäquat (treffen das was Kunde will)
  - widerspruchsfrei
  - vollständig (Problem: Wann vollständig?)
  - risikoabhängig (nicht alles beliebig tief, nur die Dinge die risikobehaftet sind)
  - eindeutig (adäquat?)
- 3 Arten von Anforderungen
  - funktional (Features)
  - nicht-funktional (bspw. Performanz)
  - Randbedingungen (z.B. gesetzliche Vorgeben oder Platformentscheidungen)
- Anfang und Endpunkt der Entwicklung, da Anforderungen auch Akzeptanztests bestimmen, Validierung: hab ich die Anforderungen verstanden? Geänderte Anforderungen ändern auch Akzeptanztests
- Requirements Engineering überlappt mit Architekturentwurf, da Rückwirkung auf RE von Architektur
- Anekdote über Echtzeit: nicht möglich, da Aufrufe immer anders verzögert, spezielle Hardware + OS nötig

- Aktivitäten im RE (iterativ): Eliciation (Was sind die Anforderungen?), Dokumentation (z.B. Use-Cases oder formal z.B. mathematische Verifikation), Übereinstimmung (Widersprüche/Konflikte finden und priorisieren, gemeinsamen Konsens finden), zusätzlich Validieren und Verwalten (Hand in Hand mit anderen Aktivitäten, Änderungen, andere Priosierung)
- Stakeholder: Benutzer des Systems (Anwenden), Betreiber des Systems (Deployen), Auftraggeber, Entwickler (und deren Kenntnisse), Architekten (wiederverwendbare Komponenten), Tester (frühe Testbarkeit hilft Validierung der Anforderungen), Botschaft: nicht nur der Anwender
- Rolle Requirements Engineer (Anforderungserheber): technisches Wissen ist notwendig, reicht aber nicht, Kommunikation, Konfliktlösung, etc.
- Techniken zur Gewinnung von Anforderung: Brainstorming (allein/Gruppe), User Stories zur Kommunikation mit Kunden, Beobachtungen (über die Schulter schauen), Fragebögen, Interviews, Vorgängersysteme? Vorschlagen von Ideen ggü. Kunde sinnvoll
- Funktional vs nicht-funktional vs constraint (Kind)
  - Verhalten von Software spezifizierbar (Turing Maschine): funktional, sonst nicht-funktional (Wartbarkeit, Sicherheit, Performanz, etc.)
  - System-Anforderungen (in dieser Vorlesung), sonst Projekt, Prozess
  - funktional: Funktionalität, System-Verhalten, Daten, ...
  - nicht-funktional: Performanz, Zuverlässigkeit, Usability
  - Rahmenbedingung: physisch, Gesetzt, etc.
- Andere Facetten
  - Satisfaction: Hard/Soft (need vs nice-to-have)
  - Role: Prescriptive - Wie soll das System sein - vorgeschrieben, Normativ - Umgebung des Systems, Assumptive - Annahme wie mit dem System interagiert wird
  - Repräsentation: Operational - Spezifikation der Daten, quantifizierbar - messbar, qualitativ - Ziele, deklarativ - rein beschreibend aber kein Umsetzungsgrad
- Klassifikationszusammenfassung
  - funktional oder nicht-funktional hat nichts mit Repräsentation zu tun
  - werden in natürlicher Sprache gegeben
  - Guidelines: kurze Sätze (1 Anforderung pro Satz), klare Formulierung, klar machen wer zuständig ist, schwache Sachen vermeiden (effektiv, Nutzer-freundlich... unprüfbar), Glossar hilft (aber nicht als Selbstzweck), aktive Sprache (Akteure?)
- Schablonen verwenden: z.B. The system must/should/will <whom?> <objects> <process word>, Wortwiederholungen nicht wie normal vermeiden, bewusst das selbe gemeint klarmachen!

- Natürliche Sprache bietet Vorteile, da sie von allen Stakeholdern verstanden wird, Diagramme o.ä. dagegen nicht unbedingt
- Featurelisten: Requirements mit eindeutiger ID um verfolgbar zu sein, verwendbar für Entwurfsentscheidungen, Nachteil: unterschiedliches Abstraktionsniveau und manchmal Liste viel zu feingranular
- heutzutage User Stories (agil) oder Use Cases (modellbasiert)
- Validierung von Anforderungen: Baue ich das richtige System?
- Verifikation von Anforderungen: Baue ich das System richtig?
- keine formalen Modelle zur Verifikation
- Code Review: gemeinsames Fehlerfinden (Inspektion, Review, Walkthrough)
- Simulation: Performance-Eigenschaften simulieren
- Prototyping: Mock-Up die Interaktion mit dem System zeigt
- Aufstellen von System-Testfällen (zeigt ob Anforderungen verstanden wurden)
- Model Checking: formale Verifikationstechniken zum Untersuchen von Widersprüchen
- Wie weit RE? Fehlerbehebungskosten? Verweildauer von Bugs? Was sind Kosten einen Fehler zu entfernen ggü. Anforderung zu spezifizieren - 2 Kurven ergeben Wirtschaftlichkeit und Optimum dieser?
- Zusammenfassung: Was sind Anforderungen? Wie klassifizieren? Wie aufschreiben? Validierung? Kosten und Vorteile von RE
- Use Cases
  - Helfen geeignete Abstraktionsebene zu finden (auch durch Übung)
  - Art wie man Anforderungen aufschreibt
  - Use-Cases nur eine Notation!
  - Use-Case-Diagramm zeigt Beziehungen der Use-Cases
  - System oft als Blackbox (also nur Angaben über Schnittstelle)
  - UI wird in Use Case nicht beschrieben
  - aus der Sicht der Ziele eines Benutzers
  - System Boundary: Was wird entwickelt, was nicht?
  - System Kontext: Gesamte Umgebung die notwendig ist
  - Context Boundary: trennt System Kontext von irrelevanter Umgebung

  - Common Scopes: Business use case (Unternehmen ist Blackbox/Whitebox), System use case (System ist Blackbox/Whitebox), component use case (immer Whitebox)
  - Use Cases als Vorlage für Sequenzdiagramm
  - Elementary Business Process (EBP): Beschreibung wie Abläufe in Unternehmen funktionieren
  - Heuristiken: Boss Test (Vorstellung: Chef fragt was habe ich den ganzen Tag gemacht? z.B. Kundenkonten angelegt statt Felder ausgefüllt), Coffee Break Test (logischer Block zu Ende und dann Kaffee trinken), Größe Test (nur 1 Schritt zu wenig)
  - Verschiedene Ebenen: Summary - gesamter Geschäftsprozess, Ziele eines Benutzers - hier, Subfunktionen - wiederverwendbar in verschiedenen Use-Cases z.B. Anmeldung, Too low - Systemaufrufe

- Use Case Diagramm
  - Überblick über die Use-Cases
  - Definieren Use-Cases nicht
  - Systemgrenzen werden gezeigt
  - Üblicherweise User Goal Use-Cases (nicht zwangsweise)
  - Aktoren üblicherweise links
  - Stakeholder beeinflusst/Bezug
  - Primary Aktoren lösen Interaktionen aus
  - Szenario: kann unterschieden werden zu Use-Case (z.B. PIN eingeben), also Instanz eines Use-Cases (z.B. 2x vertippt ... PIN ... etc.)
  - Wie Use-Cases finden? Systemgrenzen?, Primary Actors?, Welche Ziele (user goals) hat Aktor?, Welche Interaktion braucht man?
  - Beziehung zwischen Ziel und Szenario? Abhängigkeiten in beide Richtungen, z.B. Szenarien ergeben dass Ziele verfeinert werden müssen, bspw. durch Subziele, zyklische Abhängigkeiten
  - Iterativer Prozess zur Findung von Use Cases, lieber erst mal Überblick in breiter Suche bevor man zu weit in Tiefe geht, mit viel Erfahrung kann Abstraktionsebene schnell gewechselt werden
  - Use Case Definition: Aktor, Erfolgsszenario, Wann kein Erfolg? (Bedingungen), Welche Fehler können sinnvoll behoben werden
  - Aufsplitten oder Zusammenführen kann möglich sein
  - Use Case Schablonen, bekanntester "fully dressed": Context of us, scope, goal level, primary actor, stakeholder, pre- und post conditions, trigger, main success scenario, extensions, special requirements, technology and data variation lists

- Casual: verkürzt
- RUP: nahe zu Fully Dressed
- Wie Daten beschreiben? Level 1 Data nickname, z.B. PIN, reicht meist, sonst weiter als Typ oder noch weiter mit Länge und Validierung, möglichst keine If-Abfragen, eher trennen
- Fully Dressed Use Case sections
  * Vorspann bei fully dressed: Actor, Scope
  * Stakeholder und interessierte
  * Vorbedingungen: Annahmen über Systemzustand modellieren (sinnvoll, z.B. nicht Stromversorgung)
  * Nachbedingung: Erfolgsgarantie
  * Main Success Scenario: keine Verzweigungen
  * Erweiterung / Alternativen: bei Fehlern (genaues Handling nicht zwangsweise)
  * Annahmen über nichtfunktionale Anforderungen (z.b. Performanz)
  * Technologie und Daten
- Struktur von Requirements-Dokumenten
  * Einleitung (Kontext des Systems, Warum will man das System bauen? Extrem wichtig für Moral/Motivation, Lücken von Spezifikationen können evtl. umgangen werden)
  * Beschreibung des Systems: Umgebung, Architekturbeschreibung
  * Anforderungen
  * Anhang
  * Index
  * Beispiel: RUP
  * versioniert
- Werkzeuge für Requirements
  * damit nicht bspw. per Email geschickt werden
  * z.B. ROOM
  * Multi-user-access
  * Web-Frontend
  * Wiki kann auch gut sein (können viele lesen / bearbeiten, billig, aber wenig Generierungsfähigkeit, verschiedene Zugangslevel auch nicht möglich)
- Divide and Conquer
  * reale Welt

        * Analyse
        * Architektur/Design
        * Code
    - Zusammenfassung: verschiedene Abstraktionsebenen (summary, user goal), Scopes (business, user goal, component)

- Analyse und Entwurf hochgradig verzahnt (aber zwei verschiedene Tätigkeiten, Entwurf Gedanken für Losung)

- Operation Contracts
    - Anforderungen zu Sequenzdiagrammen / Entwurf?
    - Input aus Domänenmodell und Use Cases
    - Sequenz- und Klassendiagramme ableitbar
    - Design by contract: zwischen zwei Entitäten, aufrufender Kontext und Funktion, Contract für Methode sagt Vor- und Nachbedingung für Objekte und Rückgabewerte
    - Beschreiben Ablauf
    - Schema: Name der Operation, Referenzen auf Use Cases, Vor- und Nachbedingung
    - Nutzen in Business Prozessen, Use Cases, System Operationen oder auch bspw. Java Methoden

- Contracts vs Use Cases
    - Use Cases für Anforderungen
    - Contracts können zur weiteren Spezifikation erstellt werden, falls Verfeinerung notwendig

- Zusammenfassung Contracts: Operationen identifizieren, Vorbedingung und Nachbedingung, üblichster Fehler ist Vergessen von Assoziationen

3 Software Architektur