KARLSRUHER INSITUT FÜR TECHNOLOGIE

Softwaretechnik II
Zusammenfassung WS17/18

Manuel Lang

6. März 2018

INHALTSVERZEICHNIS

1 Vorgehensmodelle 2
2 Requirements Engineering 8
3 Software Architektur 13
4 Enterprise Application Patterns 19
5 Microservices 23
6 Objektorientierter Entwurf 24
7 Clean Code 25
8 Model Driven Development 29
9 Real-Time Design Patterns 33
10 Software Reliability 39
11 Software Security 42
12 Continuous Integration 46
13 Reviews 47

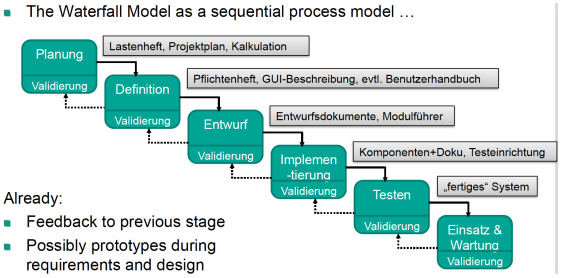
1 Vorgehensmodelle

Vorteile strukturierter Vorgehensmodelle

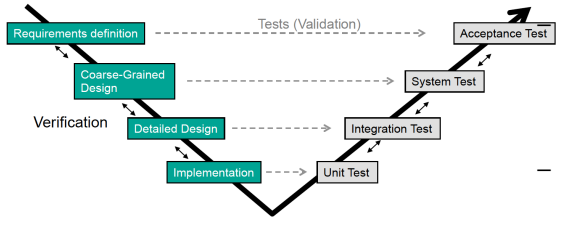
- Reproduzierbarkeit (Erfahrungen für ähnliche Projekte)
- Skalierbarkeit (größere Komplexität schneller)
- Wiederverwendbarkeit (Code)
- Risikominimierung (Entiwcklung nach Plan)

Modelle

- Softwareentwicklung liefert nicht nur Code, sondern auch Deployment-Deskriptoren (Zusazudokumente), ursprüngliche Anforderungen (Code -> Anforderungen funktioniert nicht), welche Produkte wann und wo?
- Wasserfall-Modell
  - Erstes Modell zur Definition der verschiedenen Phasen
  - überhaupt machbar?
  - Welche Stakeholder?
  - Lastenheft/Pflichtenheft
  - 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)



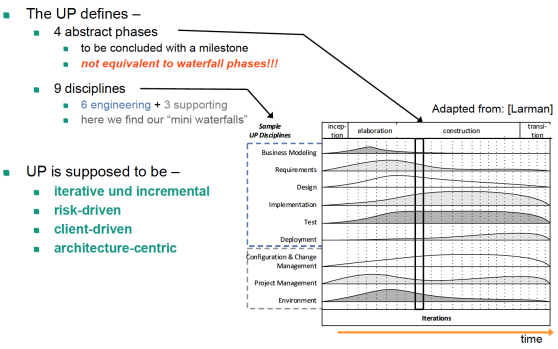
- V-Modell
  - sieht ähnlich aus wie Wasserfall
  - besagt welche Artefakte man hat + erklärt Zusammenhänge zwischen Dokumenten
  - überprüft bspw. Zusammenhang zwischen Implementierung und Design (Verifikation, zeigt Abwesenheit von Fehlern, All-Quantor funktioniert immer)
  - Z.B. mehrere Module zusammen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



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

- 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)

– 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

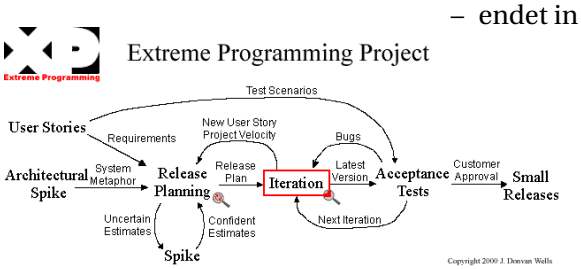


– Verschiedene Disziplinen wie früher Phasen, auch mit Deployment, insgesamt moderner, andere Disziplinen wie Änderungsmanagement, Projektmanagement, Environment (Entwicklungsumgebung) = Aufrechterhalten der Produktivumgebung + Zusammenspiel der Komponenten + Patches
– iterativ und inkrementell (streng 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

- zwischen Anforderungen und Code
- Entwurfsentscheidungen dokumentieren
- bei jedem Projekt anders
- viele Entscheidungen können vorbedingt sein
- Verteilung wichtig
- Einschränkungen des Entwurfsraums (z.B. objektorientierter vs prozeduraler Stil)
- Was evolvieren und was nicht? z.B. wartbar für welches Szenario? Was ist änderbar?
- Was wird selbst gemacht, was von außen? Was soll wiederverwendbar sein/werden?
- Design vs Architektur: Objektorientierung zwingt zu feingranularer Aufteilung, Architektur übersichtlicher (z.B. Komponentendiagramm)
- Verschiedene Sichten auf System (Requirements Engineer, Architekt, Operator, Tester, ...)
- Architekturbegriff setzt rationalen Planungsentwurf vorraus (laut Reussner)
- Grobe Struktur eines Systems
- Definition Reussner: Ergebnis einer Menge von Entwurfsentscheidungen und die betreffen die Komponenten und deren Beziehung untereinander und das Deployment

- Nicht ein Diagramm für alles, sondern verschiedene Sichten
- Ähnliche Sichten können in Viewpoint zusammengefasst werden (deutsch: Perspektive)
- Mindestens 3 Sichten werden benötigt: Aussage über Struktur (Komponenten, Abhängigkeiten), Was passierend während Ablauf (dynamisch, Abstraktion über Verhalten, Kommunikation der Komponenten untereinander oder was passiert bei Methodenaufrufen?), Abbildung auf verschiedene Ressourcen (Deployment View Point)
- Decision View Point: um auf View Points zu kommen müssen Entwurfsentscheidungen getroffen werden, warum bewusst Entscheidung getroffen
- Alternativ Kruchten 4+1 (logical, development, process, physical view und scenarios), verwendet in RUP, UP noch ausführlicher
- Komponentendiagramm gibt keine Aussage über Deployment (z.B. Thin Client vs Fat Client)
- Sequenzdiagramm Interaktion über Entitäten
- Aktivitätsdiagramm: Ablauf innerhalb von Komponenten
- Deployment View: Komponenten abbilden auf Ressourcen (3D, kann gestapelt werden)
- Vorteile explizit dokumentierter Architektur: Kommunikationsmedium (z.B. Namen für Komponenten), prüfen ob nichtfunktionale Anforderungen eingehalten werden können, Wiederverwendung, Aufwand von kleineren Komponenten leichter schätzbar, andere Faktoren können einbezogen werden
- getrieben von nicht-funktionalen Eigenschaften: Performance (sehr wichtig, Skalierbarkeit), Sicherheit (Architektur kann Exploits vorbeugen, auch wenn die meist durch Entwicklungsfehler entstehen), Verfügbarkeit (Überlastet, Abstürze, ...), Wartbarkeit, nicht sequentiell abarbeitbar, daher Architekturmuster
- Faktoren die die Architektur beinflussen: Anforderungen (nicht-funktional und Rahmenbedingungen), Wiederverwendbarkeit (Patterns) - von Vergangenheit aber auch dieses Projekt in Zukunft, Convey's Law: Architektur wird auch durch Form der Organisation der Software beeinflusst (Teamgröße, -erfahrung, Arbeit mit anderen Firmen und anderen Kompetenzen), Funktionale Anforderungen können mit jeder beliebigen Architektur realisiert werden
- Wiederverwendbarkeit: Referenz-Architektur (Architektur schon vorgegeben - neue Komponenten werden in Architektur eingebettet - nicht alles muss realisiert werden), Produktlinie

- Architekturpattern gröber als Designpattern, Designpattern 1-zu-1 Beziehnung zu Problem, Architekturpattern bieten dies nicht, eher Kompromiss mehrerer funktionaler Anforderungen
- Terminologie zur Wiederverwendung: konstanter Stil (nicht mischen), Muster kann man mischen, z.B. Übersetzer (Pipeline, Parser)
- Architekturpatterns: Schichtung (z.B. ISO/OSI), Enterprise Application Patterns, Domänenmodell, ...
- Design Prinzipien
  - Separation of concerns: Dinge die nicht zusammengehören entkoppeln, sonst viel Evaluationsdruck, in Praxis oft smart user interfaces (UI + Logik), z.B. MVC
  - Single Responsibility Prinzip: nur eine Verantwortlichkeit pro Klasse, folgt als Separation of concerns
  - Information Hiding: Auswirkung von Architekturentscheidungen lokal, kappseln in Modul
  - Principle of Least Knowledge (Law of Demeter): keine Annahmen über Entwurfsentscheidungen (z.B. nicht annehmen dass ein Auto einen Motor hat - auto.motor.anlassen() vermeiden, besser: auto.starten() ruft motor.anlassen() auf)
  - Vermeidung von Redundanzen
  - (YAGNI) - überleg ob du es brauchst? nicht alle Szenarien planen, Refactoring wenn nötig
- Observer Pattern
  - löst Problem von verschiedenen Sichten
  - z.B. verschiedene Sichten auf Datenbestand
  - Objekt zu beobachten
  - Beobachter
  - kann auch als Architekturpattern verwendet werden
- Model View Controller & Observer
  - Erweiterung des Observers
  - Controller und View an Observer, aktualisieren Model
  - Model aus Domänenebene
  - verschiedene Realisierungen, z.B. alle Kommunikation muss über Controller oder darf es direkte Kommunikation zwischen View und Model geben?
  - Applikation (App Controller ist nicht Controller, sondern Aufrufen von Domänenobjekten)

  - Pro Use Case einen Controller der diesen steuert (häufig als Fassade realisiert) als Daumenregel
  - bei kleinen Systemen kann Controller auch übersprungen werden
  - Fassaden: Ein Objekt, dass Zugriff auf andere Objekte steuert, ohne dass genaue Implementierung des zu steuernden Objekts bekannt sein muss
  - Daten Transfer Objekte: Sammlung von Objekten zusammengefasst
  - Domain Layer: Domänenmodel, Datenmodell + Funktionalitäten, Übernahme aus objektorientierter Analse

- Layered Architecture
  - oben User Interface
  - Geschäftslogik (Anwendungslogik + fachliche Domäne)
  - unten Infrastruktur
  - neue User Interfaces entkoppelt (Schnittstellen)
  - Evaluation schwierig bei neuen Features (Änderung auf jeder Schicht)

- Referenzarchitektur hängt mit Layering zusammen, z.B. wird ISO/OSI auch geschichtet (7 Schichten)

- Software Komponenten
  - Definition: ein Baustein, der komponiert und adaptiert werden kann ohne dass die Innereien verstanden werden müssen. Man wird nicht gezwungen den Source-Code zu lesen bzw. zu verstehen, Widerspruch zu Vererbung (B müsste A kennen um sinnvoll zu überschreiben/erweitern, A weiß nicht ob B seine Methoden überschrieben hat)
  - Komponenten in Java? kein solcher Begriff "component", Schnittstelle kann implementiert (implements) werden aber Abhängigkeiten nach außen? Daher Dependancy Injection (required-Schnittstelle)

- Komponentenmodell
  - definiert: was ist eine Komponente? Welche Dienste werden angeboten? Wie werden Dienste angeboten? Wie wird kommuniziert? Beispiel Lego: Platform, Komponentenframework = Noppen (was passt drauf?), Komponenten = Bausteine, Repository = Bestand, Support Tool z.B. Stuhl
  - technische Realisierungen von CBSE: OMG way - CORBA Framework (Interfacebschreibungssprache zur Generierung von Proxy-Objekten), Sun way - Java, JavaBeans, EJB, Microsoft - .NET CLR, Probleme: Ansätze sind meisten objektorientiert statt komponentenbasiert, daher OSGi?
  - OSGi: Komponenteneigenschaft, Bundling von feingranularen Objekten, Lebenszyklusmodell

- Web Services: sind in einem gewissen Sinne Komponenten, aber Komponenten sind Softwarebausteine, Services können hiearchisch verschalten werden, laufen aber ab (daher Ergebnis kommt zurück, aber kein Baustein)
- Software-Oriented Architecture (SOA): Service ist schon deployed, also deployte Komponete
- SOFA (Software Appliance): definierte Schnittstelle, Komponente eingesteckt und läuft
- ROBOCOP: eigene Infrastruktur
- KobrA: architekturorientiert, UML-basiert
- Palladio: Architekturen können auch simuliert werden - nicht nur modelliert, Performanzvorhersage zur Design-Zeit, Unterstützung von CBSE, Zuverlässigkeitsvorhersagen
    * Palladio Component Model (PCM) ist Sprache um Komponenten zu beschreiben (domain specific modelling language (DSL))
    * Performancevorhersage: Architekturdiagramm (Modell der Software) liefert durch Simulation Vorhersage für Zeitverhalten (für nicht-echtzeitfähige Systeme)
    * Was beeinflusst Performance und Zuverlässigkeit einer Komponente? Algorithmus / Implementierung, System Umgebung (Hardware, OS), extern aufgerufene Dienste (z.B. Cloud-Dienste), aber auch Nutzerverhalten (z.B. Dateigröße bei Upload), drei der vier Faktoren kontext-abhängig (nicht von Komponente abhängig), werden alle in Palladio explizit modelliert
    * Allokationskontext: Komponente auf Umgebung abgebildet (Deployment)
    * Usage Kontext: Anwendungsprofil, Art der Anwendung der Komponente
    * Assembly Kontext: Verdrahtung mit extern angeschlossenen Komponenten
    * Simulator: Änderung der 3 Parameter, z.B. Änderung der Hardware (Allokation) zeigt Änderung der Performance, wie skaliert Sofrware mit mehr Anwendern (Usage)? Was bewirkt veränderte Architektur (Assembly)?
    * Wieso nicht einfach Prototyp messen? Nachteil: Kosten durch Programmierung, Umgebung benötigt, Deployment auch aufwendig, kann paar hundert tausend Euro kosten, Simulation schneller als Echtzeit, Hardware / Treiber / Last etc. vielleicht nicht verfügbar
    * Was benötigt man um Performance vorhersagen? Komponentenmodell, Struktur, Deployment Modell, Usage Modell
    * Was bekommt man: Antwortzeit, Ressourcenauslastung, Durchsatz (akkumuliert - wann erreiche ich x%)
    * strikt komponentenbasiert: werden auf Rollen abgebilet
        · Komponentenentwicklen

    - Software Architekt
    - Software Deployer: Abbildung auf Hardware-Ressourcen (häufig auch durch Architekt)
    - Domainexperte: Interaktion mit Benutzer oder anderen Systemen (nicht Teil der Architektur)
* ViewPoints in Palladio: Structural (Komponentenentwickler, Software Architekt), Behavioral (Komponentenentwickler und Software Architekt), Deployment (System Deployer)
* Beschreibung des Verhaltens bei Aufruf von Methoden von Komponenten: beschreibt Kontrollfluss, internal actions können sehr kompliziert sein, sind aber kein externer Aufruf (müssen daher nicht weiter modelliert werden)
* Service Effect Spezifizierung: SEFF Einflussfaktoren zusammenpacken
* Unabhängigkeit von externen Diensten
* Parametrisierung über Ressourcenumgebung
* Abhängigkeit von Benutzungsprofil (Parameter)
* Parameter kombiniert
* Komposition von Komponenten: Komponenten zusammenführen in neue (äußere) Komponente
* System Komposition: Aufrufe von außen nach innen abgebildet, Schnittstellen nach außen abgeleitet
* Delegation Connector (vertikal - verschachtelt) und Assembly Connector (horizontal - verdrahtet)
* Systemmodell: Schnittstelle nach außen, komponentenbasierte Architektur, kann Komponenten von verschiedenen Repositories beinhalten, nur als Einheit deploybar
* weitere Aufgaben des Softwarearchitekts: baut Architektur, trifft Entwurfsentscheidungen, Performanceanalyse, Feedback vom Deployment, delegiert die Entwicklungsaufgaben
* System Deployer: bildet Komponenten ab auf Ressourcen (können auch verschachtelt werden, Ressourcetypen: CPU, Netzwerk, Speicher (HD, RAM - noch nicht in Palladio))
* Domänenexperte: beschreibt Interaktion mit System an äußerster Systemschnittstelle, modelliert Anwendungsverhalten, nicht Komponenten, Benutzungsprofil ähnlich zu SEFF, aber kein Bezug auf Ressourcen oder Komponenten des Systems, Eingabeparameter, wie viele Requests?
* TODO Folie 56 einfügen
* Am Ende bekommt man Wahrscheinlichkeitsverteilung (Response Time - Wahrscheinlichkeit) oder akkumulierte Dichtefunktion (mit welcher Wahrscheinlichkeit bekommt man Antwort innerhalb gewisser Zeit)

    * Lesson learned: Kontexte, Einflussfaktoren auf Performance, komponentenbasierte Entwicklung, Trennung Architektur- und Komponentenentwicklung, Black Boxes für Komponenten, SEFF und Parametrisierung

4 Enterprise Application Patterns

- überwiegender Teil von Software
- Geschäftsprozesse sollen unterstützt werden
- Beispiele: Gehaltsabrechnungen, Online-Shops, Patienten-Daten, Nachverfolgung von Sendungen, Leasing-Systemen
- Gegenbeispiele: Telekomunnikationssysteme, Textverarbeitungsprogramme, Betriebssysteme, Anlagensteuerungen
- Eigenschaften: Daten sehr wichtig und müssen persistiert werden, große Datenmengen, hochgradig nebenläufiger Zugriff auf Daten, unterschiedliche UI-System (mobile, stationär), verschiedene User-Gruppen (Kunden, Personal, ...), hochgradig vernetzt, oft nicht anpassungsfähig, Komplexität meist durch sehr komplexe Systeme, verschiedene Anforderungen (Skalierbarkeit, Wartbarkeit, ...)
- Schichten von EA: Front End, Geschäftslogik, Daten (technologiegetrieben), hier hauptsächlich Domäne + Datenbank (kein Front End, da Technologie sehr schnelllebig)
- Patterns
  - Muster vs Stile: Stile können nicht gemischt werden, schrenken andere Stile aus, Pattern können kombiniert werden
  - mehrere Muster für individuelle Probleme
  - Muster können verschiedene Varianten haben und Implementierung muss häufig selbst gemacht werden
  - Geschäftslogik Patterns
    * Datenmodell existiert, wie strukturier ich Logik?
    * hohe Komplexität, testbar, wartbar, änderbar, ...
    * Beispiel: Versicherung macht Auszahlung nach Prämie, anders für jedes Versicherungsprodukt, richtiger Algorithmus für Summe
    * Transaction Script (Muster)
      · Was sind Anfragen die aus Benutzerschicht kommen?
      · Was wird dafür in der Geschäftslogik benötigt?
      · Skript macht alle dahinterliegenden Schritte
      · Gemeinsamkeiten rausfaktorisieren

        - Vorteile: leicht von Entwicklern verständlich, Transaktionsgrenzen leicht verständlich, einfach auf Datenquellen anwendbar
        - Probleme: skaliert schlecht mit komplexer Logik, oft Code-Duplikate
    * Domain Model (Muster)
        - Kappselung in Klassen
        - Objekt-orientierter Ansatz
        - Vorteil: gut mit komplexer Logik, skaliert gut
        - Nachteile: Objektorientierung kann schwierig für Entwickler sein
    * Table Module (Muster)
        - Funktion gliedern an Daten, möglichst wie Datenbankschema
        - Welche Geschäftsfunktionalität kann auf Daten ausgeführt werden?
        - Lösung z.B. über statische Methoden möglich
        - Implementierung oft vorgegeben z.B. JDBC
    * Wann verwendet man was? Orientierung nach Komplexität und Aufwand zur Erweiterung (Domainenmodell etwa linear, Transaktionsskript Vorteile bei einfachen Systemen, Table Module skaliert ab gewinner Größe nicht mehr so gut, also Table Module besser als Transaktions Skript (weniger Duplikationen), Transaktions Skript aber nahe an Terminologie des UI und daher guter und einfacher Einstieg an Software)
    * Komplexität als Entscheidungskriterium
    * Wie aufwändig ist es Pattern zu realisieren?
    * Wie gut können Entwickler Objektorientierung?
    * Wird Table Module von Middle Ware unterstützt?
    * Web-Shop hat ziemlich verschiedene Vorgänge mit selben Daten, Komplexität kann höher sein, Performance kann beim Domänenmodell leiden, Table Module könnte bequem sein, sonst auch Transaction Script möglich, da Code Duplikation eher unrealistisch scheint
    * Leasing System kann komplexe Domänenlogik haben, deshalb Domänenlogik
    * Expense Tracking System muss gut erweiterbar sein, aber nicht so komplex, muss schnell entwickelbar sein, Domänenmodell für Erweiterbarkeit, die anderen können später schwieriger geändert werden, Argument für Transaktionsskript geringer initialier Aufwand
- Data Source Architectural Patterns
    * Wie stelle ich objektorientiert Daten da, die aus nicht OO-Datenbank kommen
    * Auch bei NOSQL - muss auch OO Dargestellt werden

* Lösungen: Manuell (JDBC), Hibernate (oder andere OR-Mapper) oder XML
* Zur Wartbarkeit keine SQL-Statements im Code verstreuen!
* Record Set (Muster)
    - Daten in einer Tabelle als geordnete Liste im Speicher (in-memory) abgelegt
    - Eine Klasse des Record Sets mit Tabellen verknüpft
    - ResultSet im Code mit dem gearbeitet werden kann (z.B. AcetiveX Data Objects oder JDBC)
    - Zugriff über Strings (resultSet.getString("Fname");) nicht schön, Typfehler erst zur Laufzeit
    - schlechte Wartbarkeit - ist es wirklich das Set?
* Table Data Gateway (Muster)
    - Objekt einer Klasse, das Schnittstelle zur Datenbanktabelle ist
    - Instanz Zugriff auf alle Tabellen
    - CRUD Operationen durchführbar
    - Gateway-Klasse implementiert SQL-Statements
    - beißt sich mit Domänenmodell, da Daten stark nach DB gegliedert, anders als bei Domänenmodell, passt aber gut zu Transaktionsskript
    - Gateway oder Fasade: Fasade verschattet möglichst komplexe Struktur und bietet einfachen Zugriffsweg, Gateway kappselt bewusst wie auf Datenbank zugegriffen wird, Fasade wird von Entwickler zur Verfügung zur Verfügung stellt, Gateway wird von Nutzer zur Verfügung gestellt
* Active Record (Muster)
    - Objekt das eine Zeile der Datenbank beinhaltet, Geschäftslogik mit integriert
    - Passt sehr gut zur Domänenlogik
    - Gibt kein Record-Set, eine Instanz auf die CRUD-Operationen ausführbar sind
    - Nachteile: bei komplexerer Geschäftslogik können Funktionen benötigt werden, die auch Zugriff auf andere Objekte benötigen
    - Klappt nicht bei viel Verarbeitung - wie bei Oberklasse?
    - sehr stark an Datenbankschema orientiert
* Row Data Gateway (Muster)
    - Objekt einzelner Zugriff (Zeile), aber ohne Geschäftslogik
    - Vorteil: mehr Freiheitsgrade zur Strukturierung der Geschäftslogik
    - Gut für Code Generierung, Entkoppelung von DB

    * Identity Map (Muster)
        · Zugriff auf DB
        · Überprüfung ob Objekt schon da ist
        · Nur eine Instanz von Objekt, nicht mehrere Instanzen von "dem selben"
    * Data Mapper (Muster)
        · Schicht die sich darum küppert wie Daten in DB kommen
        · Entkoppelt durch Mapper
        · Felder, Arrays geklärt
        · Übergang von Objekt-Schema und relationes Schema
        · auch als DAO bekannt
        · Werkzeuge existieren, Larman: "Don't try this at home!"
        · OR-Mapper können einiges dieser Funktionalität
        · Wann verwenden: Legacy Systeme (Datenbestände da, aber System nicht mehr aktuell), Active Record ist nicht mehr ausreichend für Geschäftslogik, Datenbankschema und Objektmodell müssen unabhängig evolviert werden
    * Wann sollte man was verwenden? Geschäftslogik nach Transaktionslogik, dann Row Data Gateway oder Table Data Gateway (für record set), falls Domänenmodell für einfachen Code Active Record, bei komplexen Mappern Data Mapper, falls Table Module dann Table Data Gateway, Kombinationen möglich (hier eher da unterschiedliche Datenbestände möglich)
– Objekt-relationale Patterns
    * relationale Datenbank + objekttorientierte Struktur im Business Layer (mit Vererbung)
    * müssen sehr häufig nicht selbst implementiert werden
    * Single-Table Inheritance
        · alles in eine Tabelle
        · Vereinigung der Attribute in Tabelle
        · Namenskonflikte in OOP zu lösen
        · Vorteile: sehr einfach, keine Joins, Refactoring von Felder hoch und runter benötigt keine Änderungen der Datenbank
        · Nachteile: wenn irgendwo was geändert wird, muss immer Tabelle geändert werden, Tabelle kann sehr groß werden, für Basisklasse müssen alle Eigenschaften angelegt werden auch wenn diese nicht gebraucht werden
    * Class Table Inheritance
        · Jede Klasse bekommt eigene Tabelle

        - wieder 1:1 Mapping der Attribute
        - Vorteile: Felder werden sinnvoll verwendet (keine leeren), konzeptionell sehr einfach
        - Nachteile: viele Joins (Performanceprobleme möglich), häufiger Zugriff auf Super-Types
    * Concrete Table Inheritance
        - Attribute der Oberklasse werden nach unten mit kopiert (nur nicht abstrakte)
        - Vorteil: wenger Platzverschwendung, Joins vermieden
        - Nachteil: Felder werden kopiert, bei Änderung der Oberklasse müssen alle Tabellen geändert werden
    * Wann nehme ich was? Single Table: eher bei Instanzen der Unterklassen, Performance-Vorteil, Class Table: offensichtliche Struktur, Concrete Table: ohne Joins

- Java Persistence API
  - viele Muster implementiert
  - nicht zwangsweise Enterprise JavaBeans (EJB)
  - O/R Mapping in Hibernate mit -Annotationen
  - Single Table Inheritance als Beispiel

5 MICROSERVICES

- keine genaue Definition, aber Eigenschaften: Aufteilung in Services, Design for failure, enger Zusammenhang zur Automatisierung der Infrastruktur (CI/CD) für schnelleres Feedback ermöglicht kleine Inkremente, dezentralisierte unabhängige Komponenten, Produkt mehr im Vordergrund als Projekt
- Service ausführbar im Gegensatz zu Komponenten
- kompletter Stack wird durchgetrennt, oft mehrere Datenbanken um alles zu entkoppeln
- dezentralisierte Verwaltung der einzelnen Säulen
- Design for Failure ergibt sich durch Entkopplung (ein Auswahl führt nicht dazu, dass auch andere ausfallen)
- Evolution sehr leicht möglich (viele Releases möglich)
- Wenn CI/CD wichtig, dann sind Microservices spitze!
- aber auch Nachteile: z.B. schnell Redundanzen im Code
- Architekturstil (1x MS dann im ganzen Projekten)

- Geschäftslogik organisieren
- Wiederverwendung
- Funktionen zu Objekten zuweisen
- Kriterien: Jemand ist verantwortlich etwas zu tun oder jemand weiß etwas und daraus ergibt sich Verantwortung
- Aus Domänenmodell sind Methoden erahnbar
- In der agilen Welt häufig als Kommunikationsmittel gedacht
- Interaktionsdiagramme wichtig
- Design Class Diagram (DCD) - Fragment eines Klassendiagramms, beschreibt eine Klasse (dient zur Übersichtlichkeit bei vielen Klassen)
- Klassen identifizieren: auch Klassen aus dem technischen Entwurf die sich nicht aus dem Domänenmodell ergeben
- Methoden aus Sequenzdiagramm hinzufügen (Klassen aus Entwurf)
- Controller Pattern: Controller kappselt bestimmen Use-Case
- Pure Fabrication Pattern: Methode wird zurückgeliefert die nicht im Domänenmodell verfügbar ist (z.B. Data Mapper), gut wenn zu viel Evolutionsdruck auf selbe Klasse
- GRAS Patterns (GRASP)
  - Creator Pattern
    - Ich bin verantwortlich für Erzeugung von X wenn für Objekt C gilt, C contains oder aggregates X (X ist Teil von C), C verwendet X häufig (C ruft viele Aufrufe von X auf) oder C beinhaltet Initialisierungsdaten für X (je mehr desto besser für Pattern)
  - Controller Pattern
    - Grund 1: Zentralle Klasse soll Use-Cases bearbeiten
    - Composer Component: Fasaden-Muster (Fasaden-Controller) kann Use-Case beinhalten
  - Low Coupling & High Cohesion
    - Wenn 2 Sachen eng zusammenarbeiten, dann in eine Komponente (hohe Kohesion kappseln und niedrige Kopplung, geringe Kohesion aufsplitten)
    - Beispiel: Fasade Split (nicht so toll) - Fasaden oft Gott-Klassen
  - Polymorphie

    * z.B. viele verschiedene Payments, dann abstrakte Klasse Payment die Authorisierungs-Methode vorgeben (häufig in Praxis instance of Typabfrage in Hauptklasse - grausam, da nicht neue Unterklassen hinzugefügt werden können und Oberklasse soll nicht Unterklassen kennen)
    * Vererbung ist Whitebox-Technik, Komposition dagegen Blackbox-Technik (Interna nicht relevant), Beispiel: RubberDuck erbt von Duck, aber Duck definiert fly(), was passiert in Unterklasse - Exception?
    * Mehrfachvererbung in C++ könnte helfen mehrere Superklassen zu definieren
    * In Java dagegen Strategiemuster: Ente, Schnittstellen die FlyBehaviour und QuackyBehavior definieren (Komponenten statt Vererbung)
  - Pure Fabrication
    * Datenbankevolution und Geschäftsmodellevolution in selber Klasse
    * Daher Entity Manager oder PersistentStorage erzeugt Methode zur Datenbankankopplung
  - Indirection
    * Aufgaben werden an andere Klassen delegiert
    * wegen zu hohem Evolutionsdruck
    * Pure Fabrication ist Spezialfall von Indirection
  - Law of Demeter
    * Eine Klasse soll nicht zu beliebig anderen Klassen Kontakt haben
    * Methode soll nur mit Objekten der gleichen Klasse, Methoden der Methode, Attribute ihren Attributen oder in der Methode erzeugten Objekten kommunizieren
  - Wichtig: Diagramme auf Patterns anpassen, sofern diese nicht von vornherein geplant sind

7 Clean Code

- Es gibt immer Code
- Schreibe Code immer für den nächsten Entwickler, nicht für den Compiler / Interpreter
- Schlechter Code verringert die Produktivität exponentiell
- Lehman's first law: A system that is used will be changed
- Lehman's second law: An evolving system increases its complexity unless work is done to reduce it
- Der Aufwand der für eine Änderung im System auftritt, erhöht sich je später sie auftritt

- Die Kurve kann durch agile und iterative Methoden flach gehalten werden
- Was ist "Clean Code"?
  - elegant und effizient
  - offensichtliche Logik
  - minimale Abhängigkeiten
  - macht eine Sache gut
  - einfach und direkt
  - liest sich wie gut geschriebenes Prosa
  - verdeckt niemals die Intention des Designers
  - leicht erweiterbar
  - Code, um den sich gekümmert wurde
  - Guter Code: "Weniger WTFs pro Minute"
- Objektorientiertes Design
  - GRASP aus der vorherigen Vorlesung
  - Die 5 SOLID Prinzipien
    * Single Responsibility Principle (SRP)
      · "There should never be more than one reason for a class to change."
      · Jede Verantwortung behandelt ein Belangen
      · Schelcht: große Klassen (über 200 LOC, über 15 Methoden/Felder)
      · Refactoring: Klasse verkleinern
      · Nutzen: Code leichter verständlich, Hinzufügen oder Modifizieren sollte nur wenig Klassen beeinflussen, Risiko den Code zu zerstören wird minimiert
      · Einschub - Command-Query-Separation: Aktionen und Aufrufe trennen, damit Nebeneffekte vermieden werden
    * Open Closed Principle (OCP)
      · SSoftware entities (classes, modules, funktions, etc.) should be open for extensions, but closed for modification"
      · Verhalten sollte durch neuen Code verändert werden, nicht dadurch alten Code zu ändern
      · Sehr stark verbunden mit dem Ïnformation Hiding Principle"
    * Liskov Substitution Principle (LSP)
      · Funktionen die Pointer oder Referenzen zu Basis-Klassen verwenden müssen Objekte der Basis-Klasse verwenden können

        - Verbunden zu Design by Contract: "When redefining a routine [in a derivative], you may only repalce its precondition by a weaker one, and its postcondition by a stronger one."
    * Interface Segregation Principle (ISP)
        - "Clients should not be forced to depend upon intercaes that they do not use."
        - Steckdosen-Metapher: es kann nur ein Gerät eingesteckt werden
        - Interfaces sollten so schlank wie möglich gehalten werden, hohe Cohesion - nur ein Konzept, sollten nicht von anderen Interfaces abhängen nur weil eine Unterklasse beide benötigt
        - getrennt wenn sie von verschiedenen Clients verwendet werden
    * Dependancy Inversion Principle (DIP)
        - Ä. High level modules should not depend upon low level modules. Both should depend upon abstractions. B. Abstractions should not depend upon details. Details should depend upon abstractions."
        - Designproblem: Generischen Algorithmus von detailliertem Kontext trennen
        - Template Method Pattern: Inheritance, Binding zu Compile-Zeit, hängt stark von Algorithmus ab und verletzt DIP
        - Strategy Pattern: Parametrisierung mit Konstruktor oder Setter
        - Dependancy Injection: Technologie um abstrakte Abhängigkeiten mit konkreten Objekten zur Laufzeit zu instanziieren
        - Inversion of Control (IoC) Container
        - vereinfacht Testen
        - entkoppelt Boilerplate Code
- Law of Demeter ("Don't talk to strangers"): Eine Methode m der Klasse C sollte nur Methoden der Klasse C, von m erstellte Objekte, als Argument an m übergebene Argumente oder Objekte von Instanzvariablen von C aufrufen.
- Boy Scout Rule: "Leae the campground cleaner than you found it! aufrichtig sein ggü. Code, Kollegen, zu sich selbst, agile Werte (XP, Scrum): Aufräumen ist Teil von Conecpt of Done
- Principle of Least Surprise: Äny function or class should implement the behaviours that another programmer could reasonably expect.", wenn offensichtliches Verhalten nicht implementiert wird, können Leser und Anwender nicht ihrer Intuition vertrauen und müssen Internals lesen
- Code Conventions

    * Naming: standardisiert, aussagekröftig (für jeden klar) - Hinweis über Kontext (AccountName in Account-Klasse), Hinweise über Typen (nameString, accountList), bestimmte prefixes (m_name, IName), können durch IDE bereitgestellt werden, aussagelose Namen Vermeiden (außer Sonderfälle wie i als Schleifeninvariante)
    * Kommentare: nicht schlechten Code kommentieren, besser neuschreiben, gute Kommentare sind aussagekräftig (z.B. Test case takes ages), warnend (X used here isn't threadsafe), informativ (TODO: .. Will take care of missing..), aber aussagekräftiger Code ist besser als viele Kommentare
  - Formatierung: Kohesionslevel visuell repräsentieren, vertikaler Abstand zwischen Konzepten (z.B. nach Imports oder Methoden), horizontaler Abstand um Operatoren oder Parameter zu betonen - verdeutlichen Elemente, häufig unterstützen IDEs Auto-Formatierung
  - Don't Repeat Yourself (DRY) - copy & paste reduziert Wartbarkeit, Verständllichkeit und Evolvierbarkeit, Code-Duplikation fördert Fehler und Inkonsistenzen
  - Keep It Simple, Stupid (KISS) - "Make everything as simple as possible, but not simpler"(Albert Einstein), guter Code ist für jeden leicht verständlich, guter Code adressiert das Problem adäquat, z.B. wenn IEnumerable ausreicht, verwende keine ICollection oder eine IList, Techniken die helfen dass Code für andere verständlich ist: Code Reviews, Pair Programming
  - You Ain't Gonna Need It (YAGNI): Features sind teuer (+Testen/Dokumentation) und machen das System schlechter wartbar da sie komplex werden (KISS), Vorsicht vor Optimierungen (kann teuer werden)
  - Single Level of Abstraction (SLA): Abstraktion wie Zeitung - TODO

- Clean Architecture Patterns
  - verschiedene Designpatterns: heagonale Architektur ("Ports and Adapters", bessere Testbarkeit des Kerns und unabhängig von äußeren Services die leicht ersetzbar sind), Zwiebel Architektur (kontrolliert Coupling, da es richtung Zentrum geht), Boundary/Control/Entity(BCE): Varation des MVC, Verteilung der Verantwortungen
  - Clean Architecture Patterns führen zu Unabhängigkeit von Frameworks, testbaren Systemen, Unabhängigkeiten vom UI, Unabhängigkeit von Datenbanken, Unabhängkeite von externen Gegebenheiten
  - "Clean Architecture": kreisförmige Schichten repräsentieren verschiedene Bereich von Software (weiter innen = höheres Level der Software, außen Mechanismen, innen Policies), "The Dependancy Rule": Source Code Abhängigkeiten zeigen immer nach innen
  - Entities: Kappseln Business Rules, Objekt mit Methoden, Datenstruktset mit Funktionen, kann als Businessobjekt für Anwendungen verwendet werden, operationale Änderungen sollten die Entity nicht ändern

  - Use Cases: beinhaltet architekturabhängige Business Rules, implementiert Use Cases des Systems, Schicht sollte sich nicht ändern bei Änderung der Entitites oder DB, UI oder anderen Externas, Schichte sollte sich ändern bei operationalen Änderungen der Anwendung
  - Interface Adapters: Betrifft Datenaustausch zwischen Schichten, beinhaltet MVC Architektur eines GUI, Modelle sind Datenstrukturen
  - Grenzen überschreiten: Controller und Presenter kommunizieren in Use-Cases, Source Code Abhängigkeiten zeigen nach innen (meist mit Dependancy Inversion)
  - Refactoring: Ïf it strinks, change it."(Grandma Beck), Beispiele: Methoden/Klassen extrahieren, Methoden/Felder verschieden, Inline Klassen, ABER: Refactoring nur mit Tests, Bad Smells: lange Methoden, duplizierter Code, Aufruf Methoden anderer Klassen, Datenklassen, riesige Klassen / Gottklasse, ..., Format des Refactorings: Name, Zusammenfassung, Motivation, Mechanismus, Beispiel, Wann refactorn: wenn "Bad Smellsäuftauchen, Limitationen: Performance, Persistenz, Interfaces

- Zusammenfassung: Empfehlungen wie Code zu strukturieren, Refactoring für saubereren Code, Design Philosophien und Prinzipien

8 Model Driven Development

- Was ist ein Modell?
  - Abbildungsmerkmal: Modelle sind immer Modell von etwas, d.h. Abbildung von natürlichen oder künstlichen Originalen, die auch Modelle sein können
  - Verkürzungsmerkmal: Modelle beschreiben nicht alle Atriibute des Originals, nur die als für Ersteller und Anwender wichtig erarchteten
  - Pragmatisches Merkmal: Modelle sind ihren Originalen nicht eindeutig zugeordnet. Sie erfüllen ihre Substitutionsfunktion a) für bestimmte Subjekte, erkennen oder handeln, mit Hilfe von Modellen, b) in bestimmten Zeitabständen und c) unter Einschränkung auf bestimmte mentale oder tatsächliche Operationen.
  - Original kann durch eine Vielzahl von Modellen beschrieben werden
- Metamodelle (griechisch meta = nach): Modelle die Modellierung beschreiben, beschreibt die Struktur von Modellen (Kontrukte der Modellierungssprache, Assoziationen zwischen Elementen, Constraints, Modeling Rules)
- Beschreibung von Metamodellen
  - meist in formaler Sprache beschrieben
  - bisher (VL + Zusammenfassung) nur UML Klassendiagramme
  - Klassendiagramme verwendendet für modellbasierte Software-Entwicklung und hauptsächlich für Kommunikation designed

  - Spezifikation von Metamodellen
    * Abstrakte Syntax: beschreibt die Konstrukte des Modells und die Eigenschaften und Beziehungen zwischen ihnen, Beschreibung ist unabhängig von der konkreten Darstellung dieser Konstrukte, Eigenschaften und Beziehungen.
    * Konkrete Syntax: beschreibt die Darstellung von Konstrukten, Eigenschaften und Beziehungen, die in der abstrakten Syntax spezifiziert sind. Für die abstrakte Syntax eines Metamodells muss mindestens eine konkrete Syntax angegeben werden, und beliebig viele können angegeben werden.
    * Statische Semantik: beschreibt die Modellierungsregeln und -einschränkungen, die in der abstrakten Syntax nicht ausgedrückt werden können. Für die Definition von Restriktionen als Constraints gibt es spezialisierte Sprachen (z.B. OCL).
    * Dynamische Semantik:beschreibt die Bedeutung seiner Konstrukte. Die dynamische Semantik wird oft nicht formal festgelegt, sondern durch Texte in natürlicher Sprache. Durch die Erstellung eines Mappings des Metamodells auf eine Sprache mit genau definierter Semantik (z.B. Petri-Netze) kann die dynamische Semantik formal spezifiziert werden.

- Representation / Instatation
  - Modell repräsentiert Original
  - Wenn das Modell preskriptiv (vor dem Original) erstellt wurde, dann Instanziiert das Original das Modell
  - Metamodelle werden normalerweise im vorraus erstellt, damit Modelle als Instanzen erstellt werden können
  - häufig Modellierung in vier Schichten

- Selbstbeschreibende Modelle
  - Modelle der abstraktesten Schicht werden oft ßelf-descriptive"genannt

- Model-Driven Software Development (MDSD)
  - Model-Driven Engineering: Kombination von Domain-specific modelling languages (DSML) - Deklarative Beschreibung der Anwendungsstruktur, Verhalten und Anforderungen, Transformation Motoren und Generatoren, Ziel: Reduktion der Plattform Komplexität
  - Model-Driven Software Development: Anwendung von MDE für Software-Entwicklung, andere Anwendungen: Hardware-Entwicklung, Software-intensive Systeme, Laufzeit-Modellierung
  - Model-Driven Architecture: Markenzeichen der Object Management Group (OMG), spezieller Prozess für MDSD, verwendet OMG Standards (UML, MOF, XMI, EDOC, SPEM, CWM), definiert einen Prozess für den Modelltypen CIM, PIM and PSM

  - Code ist nur ein Modell: Er beschreibt das Verhalten und die Struktur eines Systems
  - Model-Driven vs. Model-Based
    * Model-Based: manche Modelle sind sekundäre Artefakte, Modelle werden für Dokumentation und Kommunikation verwendet, manuelle Analyse und Evaluation
    * Model-Driven: Modelle sind primäre Artefakte, wesentlicher Teil des Systems während der Entwicklung, können nicht weggelassen werden, explizit spezifiziert, entwickelt, versioniert, etc., Analyse durch Modelltransformationen
  - Ziele von MDSD
    * bessere Plattform-Unabhängigkeit und Zusammenarbeitsfähigkeit
    * verbesserte Entwicklungsgeschwindigkeit mit Code-Generierung
    * bessere Software-Qualität
    * Wiederverwendung von MDSD Infrastruktur in SOftware Product Lines (SPL)
    * Optimierte Trennung von Belangen mit verschiedenen Modellen
    * Einfachere Wartung durch Vermeidung von Redundanzen
    * Managen von technologischen Änderungen
    * Managen von Komplexität durch Abstraktion
  - Nutzen von MDSD
    * Kostenreduktion
    * kürzere "time-to-market"
    * Variablität durch Verwendung von Software Product Lines
    * Verwendung von Domänenwissen in Modellen
    * Höherere Software-Qualität

- Model-Driven Architecture (MDA)
  - bekannte OMG Standards: UML, CORBA, MOF, XMI, QVT, MDA
  - MDA bietet einen Ansatz und Tools zur Spezifizierung eines System unabhängig von der Zielplattform, zur Spezifizierung der Plattformen, zur Wahl von Plattform und System und zur Transformation der Systemspezifikation zu einer bestimmten Plattform. 3 Hauptziele: Portabilität, Interoparibilität und Wiederverwendbarkeit, alle Standards basieren auf Meta-Object-Facility (MOF)
  - Computation-Independent Model (CIM) beschreibt die Anforderung für System und Umgebung, Details der Struktur und Verhalten sind verdeckt oder unbestimmt
  - Platform-Independant Model (PIM) fokussiert auf Anwendung eines Systems, aber verbirgt die für eine Plattform notwendigen Details, zeigt den Teil der kompletten Spezifikation der sich bei Änderung der Plattform nicht ändert

  - Platform-Specific Model kombiniert den platformunabhängigen Viewpoint mit zusätzlichem Fokus auf das Detail der Anwendung einer spezifischen Plattform eines Systems

- Im idealen MSDS sind keine Programmierer nötig, Domänenexperte entwickelt Software über DSLs mit Code-Generierung und Technologie-Experte hilft bei Modellierung der Plattformen, Transformationen und Meta-Modellen

- Model-Transformationen
  - Transformation ist die automatische Generierung eines Zielmodells von einem Quell-Modell abhängig von Transformationsdefinitionen
  - Transformationsdefinition ist ein Set von Transformationsregeln die zusammen beschreibung wie ein Modell in Quell-Sprache in ein Modell in Ziel-Sprache transformiert wird
  - Eine Transformationsregel ist eine Beschreibung wie ein or mehrere Konstrukte in der Quell-Sprache in ein oder mehrere Konstrukte der Ziel-Sprache transformiert werden
  - Mechanismen für Transformations-Sprachen
    * Deklarativ: fokussiert auf dem "WasÄspekt, was in was transformiert wird ist als eine Relation zwischen Quell- und Ziel-Modell definiert, funktionale / logische Programmierung
    * Operational / Imperativ: fokussiert auf dem "WieÄspektt, Spezifikation der Schritte die benötigt werden um die Zielmodelle aus den Quellmodellen abzuleiten
    * Beispiele: QVT-R, QVT-O, ATL, Xtend

- Sprachen und Tools
  - General Purpose Languages (GPL)
    * z.B. UML, Java, C#
    * "lingua franca"
    * sehr ausdrucksvoll
    * nicht kompakt und präzise
    * leichte Kommunikation über Domänengrenzen
  - Domain-specific Languages (DSL)
    * e.g. BPMN, PCM
    * optimiert für spezielle Anwendungen
    * weniger ausdrucksvoll
    * kompakter und präziser
    * "Tower of Babylon"Problem

  - Textuelle Modellierung
    * Nachteile von Diagrammen: graphisch konkretex Syntaxen behandelten oft nur ein Teil der abstrakten Syntax, zusätzliche Eigenschaften nötig, Layout-Informationen benötigt um Modelle zu verstehen, Versionierungs-Probleme, schwierige Navigation für sehr große Modelle
    * Vorteile von textueller Modellierung: Wiederverwendbarkeit von textuellen Tools (copy/paste, diff/merge, patches, ...), Auto-Completion, Syntax Highlighting, Fehler Hinweise
    * Xtekt: Framework zur Erstellung von textuellen Sprachen, Open Source
    * Model-to-Text Transformationen (xpand): Template-Engine

- Zusammenfassung: Modellierung von Sprachen muss auch durch Modelle beschrieben werden um von Maschinen verarbeitbar zu sein sodass Modelle in andere Modelle oder Code umgewandelt werden können, weitere Teile von MDSD beinhalten die Object Constraint Language, Domain-Specific Languages, Transformation Languages

9 Real-Time Design Patterns

- Echtzeitsysteme
  - meistens Systeme die Umwelt aufzeichnen und steuern (gewöhnlich eingebettete Systeme, aber nicht alle eingebetteten System sind Echtzeitsysteme)
  - Unvermeidlich assoziiert mit Hardware (Sensoren: sammeln Daten aus der Umwelt des Systems, Aktoren: Verändern die Umwelt des Systems)
  - Zeit ist kritisch: Echtzeitsysteme müssen in einer definierten Zeit antworten, typischermüssen müssen Daten schneller verarbeitet werden als sie gesammelt werden, jedoch ist dies nicht immer möglich, weshalb Buffer verwendet werden
  - Logische vs temporale Korrektheit: Ein Airbag muss in einer bestimmten Zeit voll ausgefahren sein (logisch und zeitlich korrekt)
  - Ein Echtzeitsystem ist ein Softwaresystem das von der Funktionsweise von korrekten Ergebnisse sowohl der Zeit wann diese Ergebnisse produziert werden abhängt.
  - Ein weiches Echtzeitsystem ist ein System bei dem die Operation vemrindert wird wenn Ergebnisse nicht in der geforderten Zeit geliefert werden
  - Bei einem harten Echtzeitsystem wird solche eine Operation als inkorrekt betrachtet.
  - Antwortzeiten des Systems müssen früh im Deisgn betrachtet werden (oft durch möglich Anreize, die erwartete Antwort und Zeitbedingungen)
  - Anreiz/Antwort Systeme: unter einem gegebenen Anreiz muss das System innerhalb einer bestimmten Zeit eine Antwort liefern - 2 Fälle: Periodischer Anreiz (z.B.

Temparatur-Sensor liefert 10 Daten pro Sekunde) oder unregelmäßige Anreize (z.B. Stromunterbrechnung ausgelöst)

  - Interrupts: Kontrolle wird automatisch an einen definierten Speicherbereich transferriert, weitere Interrupts werden blockiert, Interrupt Services müsse kurz, einfach und schnell sein
  - Periodische Prozesse: werden auch in System existieren jeweils mit verschiedenen Perioden, Ausführungszeiten und Deadlines

- Typen von Echtzeitsystemen
  - Monitoring & Control Systems sind wichtige Klassen von Echtzeitsystemen
  - Monitoring Systeme führen Aktionen as wenn außergewöhnliche Sensordaten auftreten
  - Control Systeme kontrollieren kontinuierlich Hardware Aktoren abhängig der Sensordaten
  - Data Acqusition Systems: dritte Klasse von Echtzeitsystemen, sammeln Sensordate für nachfolgende Verarbeitung (Erfassung und Verarbeitung können unterschiedliche Perioden und Deadlines haben)
  - einfaches Sensor/Aktor Schema: Sensoren generieren typischerweise Anreize die dann vom System zu einer Antwort ausgewertet werden und an den Aktor gesendet werden (TODO Grafik Folie 11)
  - Designentscheidungen: Da die Timinganforderungen von verschiedenen Anreizen/Antworten ausgehen sollte jedes Sensor/Aktor paar seine eigenen Prozesse haben,

- Echtzeit-Betriebssysteme
  - herkömmliche Betriebssystem tragen viel Overhead wie Datei- und Netzwerkmanagement, UI-Unterstützung, etc. und sind daher nicht echtzeitfähig bzw. nicht für Echtzeitsysteme ausgelegt.
  - Echtzeitbetriebssysteme sind spezialisiert für Echtzeitsysteme und beinhaltet die folgenden Komponenten
    * Echtzeituhr (stellt Information für das Scheduling von Prozessen zur Verfügung)
    * Interrupt Handler (managet die unregelmäßigen Aufrufe des Services)
    * Scheduler (wählt den nächsten auszuführen Prozess aus)
    * Ressourcenmanager (reserviert Speicher und Prozessorresourcen)
    * Dispatcher (starter die Ausführung von Prozessen)
    * TODO Komponentenübersicht Folie 15

- Management von Prozessen: beschäftigt sich damit, das Set von Prozessen zu verwalten, Echtzeitbetriebssystem Scheduler verwendet die Echtzeituhr um zu bestimmen wann ein Prozess ausgeführt werden soll, müssen zwei Prioritätenlevel unterst+tzen (Interrupt Level Priorität, Clock Level Priorität - periodische Prozesse)
- Periodisches Management von Prozessen
  1. Scheduler wählt den nächsten Prozess aus
  2. Ressourcenmanager reserviert Speicher und Prozessor
  3. Dispatcher nimmt den Prozess von der Liste, lädt ihn auf einen Prozessor und startet Ausführung
- Scheduling Strategien
  * First In First Out (FIFO) - dynamisch, nicht präventiv, ohne Priorisierung
  * Fixed Priorities - dynamisch, statische Priotäten
  * Earlierst-Deadline-First-Scheduling(EDF) - dynamisch, dynamische Priorisierung
  * Least-Laxity-First-Scheduling (LLF) - dynamisch, dynamische Priorisierung (Laxität = Deadline - jetzt - verbleibende Prozessdauer)
  * Time-Slice-Scheduling - dynamisch, präventiv, ohne Prioriserung
- Java als Echtzeitsprache?
  * Harte Echtzeitsysteme sollten in Assembly implementiert werden, damit sichergestellt werden kann, dass Deadlines eingehalten werden
  * Standard Java hat Probleme mit Echtzeit-Ausführung - Wan soll ein Thread gestartet werden? Automatische Garbage-Collection kann Threads interrupten, dynamisches Laden und Linking von Klassen ist problematisch, JIT Compiling, ..., aber es existieren spezifische Java-Spezifikationen z.B. JAVA/RT
  * JamaicaVM: harte Echtzeit-Ausführung, Garbage Collection in Echtzeit

- System Design
  - oft muss Hardware und Software designed werden
  - Designentscheidungen sollten auf Basis von nichtfunktionalen Anforderungen getroffen werden (z.B. Performanz, Änderbarkeit), spezielle Hardware hat bessere Performance aber schlechtere Änderbarkeit (Tradeoff)
  - Timing-Constraints könne Simulation und Prototyping benötigen
  - Der Echtzeit Systemdesign Prozess
    1. Identifiziere Anreize und zugehörige Antworten
    2. Definiere die zugehörigen Timing-Constraints
    3. Wähle die Ausführungs-Plattform aus (Hardware + RTOS)

4. Fasse Anreize und Antworten in andauernde Prozesse auf
5. Designe Algorithmen zum Verarbeiten der Anreize und Generieren der Antworten
6. Designe ein Scheduling System welches sicherstellt, dass Prozesse immer vor der Deadline ablaufen

– Threads und Prozesse
  * Mehrere Threads können mit dem selben Prozess existieren, die Ressourcen wie Speicher teilen
  * Verschiedene Prozesse teilen keine Ressourcen
  * Die Threads eines Prozess teilen seine Anweisungen (Code) und seinen Context (die Zustände von Variablen zu einem bestimmten Zeitpunkt)
  * In UML sind Threads Grundeinheit von Nebenläufigkeiten (Lightweight Threads entsprechen der obigen Definition von Threads sofern Speicher geteilt wird und heavyweight Threads entsprechen der obigen Definition von Prozessen sofern kein Speicher geteilt wird)
  * In dieser VL: Prozesse und Threads nur als Einheit der Nebenläufigkeiten
  * Eigenschaften von Threads
    · Grundeinheit von Nebenläufigkeitenin UML
    · Eingangspattern kann periodisch oder unregelmäßig sein
    · Ausführungszeit (für Modellierung wird meist Wahrscheinlichkeitsverteilung benötigt)
  * Thread Kommunikation
    · Kommunikation kann über Speicher (gesharete Variablen) oder Nachrichten (synchron/asynchron, blockend/nicht-blockend) erfolgen
    · Kommunikation über gesharete Variablen
      · schneller als über Nachrichten
      · benötigt Zugriffskontrolle um typische Fehler (lesen von alten Werten, Daten überschreiben, inkonsistente Daten) zu vermeiden
    · Kommunikation über Nachrichten
      · Verhalten von Sende-Operationen blockend oder nicht-blockend (Komplettierung muss überprüft werden, IDs für Nachrichten benötigt)
      · Zustand des Empfängers während der Kommunikation kann synchron (Sender und Empfänger bekommen Operation) oder asynchron (Empfänger muss nicht Operation erhalten - Buffer nötig) erfolgen
  * Petri Netze
    · traditionelle Modellierung von Nebenläufigkeiten

            - basieren auf Graphentheorie, deshalb präzise Semantik
            - Erweiterungen wie Timed Petri Nets oder Stachstic Petri Nets existieren
            - Elemente: Places (Kreis) - beschreiben die aktuellen Zustände eines Systems, Transitions (Rechtecke) - stehen für Events die das System in einen neuen Zustand überführen, Arcs (Pfeile) - verbinden Places und Transitions, Tokens - Bedingung ist wahr
        * Analyse von Echtzeitsystemen
            - Petri-Netze können zur Analyse des Systemflusses verwendet werden
            - Jedoch ist es auch wichtig, die Einhaltung der Timing Constraints zu analysieren
            - Kann aufdecken, dass Timing Contraints nicht eingehalten werden können

- Dimensionen von Dependability
    - Availability (the ability of the system to deliver services when requested)
    - Reliability (the ability of the system to deliver services as specified)
        * Wahrscheinlichkeit oder Zeitdauer von fehlerfreier Bedienung
        * ob ein Fehler in Gefahr resultiert hängt von der Systemumgebung ab
        * eingebettete Systeme versuchen öfters Realibility durch Redundanzen zu erhöhen
    - Safety (the ability of the system to operate without catastrophic failure)
        * Abwesendheit von Gefahr für Menschen und Umgebung (katastophalen Fehlern, nicht verwechseln mich Sicherheit)
    - Security (the ability of the system to protect itself against accidental or deliberate intrusion)

- Fault-Error-Failure Chain
    - Fault
        * Defekt in einem System
        * kann zu einem Failure führen, muss aber nicht
        * z.B. Fehler im Code, Eingabe und Zustandsbedingungen können bedingen, dass Fehlercode nicht ausgeführt wird
        * gewöhnlich als Bug bezeichnet
    - Error
        * Diskrepanz zwischen dem beabsichtigten und dem eigentlichen Verhalten des Systems
        * tritt zur Laufzeit auf

  - Failure
    * Zeitinstanz wann ein System unerwartetes Verhalten zeigt
  - Ein Error muss nicht notwendigerweise der Grund für einen Failure sein, z.B. kann eine Exception von einem System geworfen werden
  - Typen von Fehlern
    * systematisch (auch Design Faults) - Fehler die zur Design- oder Build-Zeit gemacht werden
    * zufällig - Fehler treten bei etwas auf, das zu einer anderen Zeit funktioniert hat (transient: verschwinden nach einiger Zeit, persistent: bleiben bis Intervention)

- RT-Patterns (Safety und Realiability)
  - Architectural Pattern: Channel
    * Channel: pipe die sequentiell Daten von Eingabe in Ausgabe-Wert transformiert
    * Architekturpattern für Echtzeitsysteme
    * Mehrere Channels um Qualität zu erhöhen: höherer Durchsatz - bessere Performance, Fehlertoleranz
  - Protected Single Channel
    * billigere Erhöhung von Safety ohne Redundanz durch Fehlerdetektion in Channel
  - Homogeneous Redundancy (Switch to Backup)
    * Schutz gegen zufällige Faults, fehlertoleranter Zustand wird nicht angenommen, daher Backup
  - Triple Modular Redundancy
    * Schutz gegen zufällige Faults ohne fehlertoleranten Zustand
    * Operationen können weitergeführt werden
    * keine Validierung benötigt
  - Heterogeneous Redundancy
    * ähnlich zu Homogeneous Redundancy aber unabhängiges Design und Implementierung der Channel
    * Schutz gegen zufällige und systematische Faults
    * Garantiert Fortlauft der Operation ohne einen fehlertoleranten Zustand
  - Monitor-Actuator Pattern
    * Schutz gegen zufällige und systematische Faults in einem fehltertoleranten Zustand

    * Überwachen von Channel und Aktor
    * Aktor Monitor Sensor muss ein unabhängiger Sensor sein
  - Sanity Check Pattern
    * Lightweight Schutz gegen zufällige und systematische Faults in einem fehlertoleranten Zustand
    * Abgeleitet von Monitor-Actuator Pattern
    * nur eine Approximation des Ergebnisses
  - Watchdog Pattern
    * Sehr leichtgewichtiger Schutz gegen zeitbasierte Faults und Detektion von Deadlocks in einem fehlertoleranten Zustand
    * einfache Fehlerdetektion in Channel
    * keine Überwachung vin Aktor
    * Channel sendet liveness Nachrichten an Watchdog
  - Safety Executive Pattern
    * Safety für komplexe System mit nicht-trivialen Mechanismen um fehlertoleranten Zustand zu erreichen

10 Software Reliability

- Software Reliability ist definiert als die Wahrscheinlichkeit von fehlerfreier Bedienung eines Computer Codes für eine bestimmte Missionszeit in einer bestimmten Eingabeumgebung
- ähnlich zu Hardware Reliability
- Fehlerfreie Bedienung: Code produziert die erwartete Ausgabe
- 1 - reliability = probability of failure on demand (POFOD)
- Rate of occurrence of failures (ROCOF): Anzahl an Fehlern in gegebener Zeit, Kehrwert ist mean time between failures (MTBF)
- Mission time - Zeit während der die Software bedienbar und für Eingaben bereit ist
- Availability beschreibt die Zeit wann sie verfügbar ist: Availability = (MTBF/(MTBF+MTTR))
- Software Reliability vs Hardware Reliability
  - Hardware leidet unter einer Verschlechterung der Nutzung und ist daher im Laufe der Zeit anfällig für Ausfälle
  - Software-Fehler entstehen durch Bugs im Code, Hardware dagegen durch Material-Defekte

  - Es ist theoretisch möglich einen Bug-freien Code zu haben und deshalb werden nie Software-Fehler auftreten, bei Hardware ist dies nicht möglich

- Fault-Error-Failure Chain (Wiederholung von vorheriger Vorlesung)
- Übersicht Software Testen
  - Functional Testing: Alle Nutzer-Funktionen der Software testen
  - Coverage Testing: Alle Pfade der Software testen
  - Random Testing: Testfälle werden zufällig aus Eingaberaum gewählt
  - Partition Testing: Eingaberaum wird in SStrataäufgeteilt und Testfälle werden zufällig entnommen
  - Statistical Testing: "formal random experimental paradigmßum zufälligen Testen
- Reliability berechnen
  - Bugfixes produzieren in etwa 7% aller Fälle neue Bugs
  - Bugs werden nicht direkt gefixt
  - daher keine 100 prozentige Reliability
- Testen ist Samplen
  - meisten existiert für Software eine unendliche Anzahl an Tests
  - Nur endliches Sample davon können getestet werden
  - Wie sollte Sample selektiert werden?
  - Was sollen wir aus dem Sample ableiten?
  - Statistisches Experient - wie modellieren?
- Statistisches Testen
  - Testen mit ausgewählten Testfällen bringt nur anekdotische Belege - statistisches Testen ist benötigt um wissenschaftlich die Reliability von Software zu bestimmen
  - Um die Reliability vorherzusagen wird ein Modell für die erwartete operational use der Software, eine Testumgebung die das operational environment simuliert und ein Protokoll um die Testdaten zu analysieren und statistisch valide Inferenz über die Reality zu machen benötigt
  - Im statistischen Testen wird ein Modell entwickelt um die Anzahl an Nutzen der Software zu charakterisieren und ein Modell verwendet um ein statistisch korrektes Sample von allen Nutzen der Software zu generieren
- Erwartete Anzahl an Failures
  - Jeder Test kann als Bernoulli Trial betrachtet werden, also entweder funktioniert die Software korrekt oder nicht
  - Man kann k Failures auf n Trials in $\binom{n}{k}$ verschiedenen Möglichkeiten aufteilen

- Realiability samplen
  - $\hat{R}$ ist berechnet als $\hat{R} = \frac{n-k}{n}$
  - Differenz zwischen einem und 10 erfolgreichen Tests ist Konfidenz
  - Reliability Modelle
    * da $\hat{R}$ eine zufällig Variable ist $C[\hat{R}]$:= Konfidenz von $\hat{R}$
    * Konfidenz ist Wahrscheinlichkeit, dass richtige Reliability R mindestend $\hat{R}$ ist.
    * $\rightarrow C[\hat{R}] = Pr[R \geq \hat{R}] = \beta$
    * Eine Frage des Vetrauens
      · Wie konfident können wir über die gemessene Reliability sein? Statistische Sampling-Theorie sollte helfen
      · Es ermöglicht eine obere Grenze $\tilde{p}$ für den Fehlerwahrscheinlichkeit und ein Konfidenz-Level zu ermitteln
    * Usage-Based Statistical Testing
      · 3 Kern-Bedingungen
        · Statistische Tests werden anhand eines probabilistischen Modells generiert
        · Das Resultat eines Tests ist statistisch unabhängig von den Resultaten der anderen
        · Existenz eines Orakels zur Bestimmung von erfolgreichen und fehlgeschlagen Tests
      · Statistical Usage Model
        · Knoten sind SStates of Use"(bspw. Komponenten oder Websites einer WebApp)
        · Pfeile sind gelabelt mit Nutzer-Events (z.B. Nutzer betätigt Button)
        · Wahrscheinlichkeiten basieren auf Wissen oder Erwartungen (z.B. Logs / Expertenschätzungen)
        · Mehr als ein Exit-State möglich
        · Nutzen: Testgenerierung kann automatisiert werden (z.B. durch random walk - Tests abhängig der Wahrscheinlichkeit generiert oder coverage set - minimales coverage set für baseline test), statistische Berechnungen (Analyse des Modells - Analyse der Testergebnisse), Requirement Analyse
        · Andere Vorteile: Usage Profile wird vom Modell berechnet, Statistiken können verwendet werden um Annahmen über Use zu validieren
        · Modell zu konstruieren ist Aufgabe der Requirement Analyse

        - Modell zu konstruieren führt zu Diskussionen über die Anwendung des Systems durch Kunden
        - Anwendungen von Ideen zur statistischen Prozesskontrolle (z.B. Abbruchkriterien)
        - Modell von Use hilft für Seperation der Concerns
    - Zusammenfassung
        - Modell wird entwickelt um die Population an Uses der Software zu charakterisieren
        - Modell wird verwendet um ein statistisch korrektes Sample zu generieren
        - Resultate wird als Basis für Konklusion verwendet
        - Erlaubt Reliability zu messen
        - Resultate indizieren dass zufälliges Testen besser als Partition und Coverage Testing sein kann, kosteneffizienter ist und dem Tester realistischere Schätzungen liefert
        - Limitationen: Exception-Bedingungen, zustandsabhängige Fehler, Usage-Modell benötigt, daher Kombination mit anderen Test-Ansätzen

- Testen und Debuggen allein lässt keine Aussagen über Software-Reliability schließen, Markov-Ketten liefern nützlichen Ansatz für statistisches Testen

11 Software Security

- Wieso Software Security in Betracht ziehen? - Unsichere Software kostet Geld wegen DOwntimes, Zeit und Kosten Probleme zu beheben, Datenlecks, zusätzliche Sicherheitsmaßnahmen (Firewalls, Anti Viren Software, Spam Filter)
- Software ohne Secure kann auch Safety Problem weren
- Ziele von Software Security
    - Hauptziele: Vertraulichkeit (persönliche Daten - Privacy, Unternehmensdaten), Availability und Integrity von Daten und System
    - Unterstützende Ziele: Nutzer-Authentifikation, Rückverfolgbarkeit & Revision, Monitoring, Anonymität
    - Wie Testen: Die Abstinenz von Bugs/Sicherheitslücken kann nicht gezeigt werden
- Wieso wird Software unsicher?
    - oft durch Manangement-Vorgaben (keine Zeit/Geld zum Testen, Änderung der Anforderungen, System wird in verschiedenem Kontext verwendet, Leute nicht genügend ausgebildet)

  - Schlechtes Design
  - Im Gegensatz zu Safety oder Reliability Problemen werden Security Probleme durch intelligente Angreifer hervorgerufen
  - Implementierungsfehler (häufig: Buffer Overflows bei Eingaben, Race Conditions, Pseudo Randomness)
- Potentielle Angriffe
  - Netzwerk-Sniffer und Proxies
  - Viren, Würmer, Trojaner
  - Rootkits, Port Scanner
  - Insider Attacks
  - Diebstahl
  - Einbruch
  - Social Engineering Angriffe
- Security während Entwicklungs-Lifecycle
  - Security ist nicht-funktionale Anforderung (so früh wie möglich in Betracht ziehen)
  - Software Security Management = Risiko Management (Trade-Off zwischen Funktionalität, Performanz, Usability und Time-To-Market)
  - Design for Security (Security in jeder Phase der Entwicklung berücksichtigen, auch Entwicklungs-Umgebung schützen)
- Finden von Security-Requirements
  - Ziehe Security Ziele in Betracht (jedoch meist zu generell)
  - Versuche potentielle "Misues Casesßu finden (basierende auf Use Cases), z.B. "Kreditkartendaten lesen darf nicht für Angreifer möglich sein
  - Ziehe System Kontext in Betracht
  - Bedrohungsmodellierung mit Checklisten
- Formulieren von Security-Requirements
  - Vermeide generelle Aussagen und identifiziere was geschützt werden soll (vor wem? wieso?)
  - Klassifiziere Schutz-Anforderungen für Risiko- und Prioritätsmanagement
- In der Design Phase
  - leite potentielle Maßnahmen von den Sicherheits-Anforderungen und Misuse Cases ab
  - Evaluiere verschiedene Design-Alternativen

  - Finde angebrachten Trade-Off für widersprüchliche Anforderungen
  - Weitere Maßnahmen: verwende etablierte Prinzipien und Patterns, halte das Design einfach

- Prinzipien zum Erstellen von securer Software
  1. Sichere die schwächste Verbindung
  2. Übe Verteidigung in Tiefe aus, falls ein Fehler nicht von einer Verteidigung abgefangen wird, dann eben von einer anderen
  3. Faile sicher: sicheres Exception-Handling, keine System-Details herausgeben
  4. Secure by default: Alle Security Einstellungen sollte per Default aktiviert sein
  5. Folge dem least privilege Prinzip: vergebe nur minimale benötige Rechte
  6. Keine Security durch Obscurity (Kerkhoff's Prinzip): keine verstecken oder magischen URLs, Logins oder Cheats um Reverse Engineering zu vermeiden
  7. Minimiere die Angriffsfläche: weniger Code ist besser da er die Auftrittswahrscheinlichkeit von Bugs minimiert (vermeide Duplikate), nicht unnötig viele Interfaces
  8. Vertraulicher Kern: isolierter Code mit Sicherheits-Privilegien
  9. Eingabe Validierung und Ausgabe Encoding (Eingabe über Whitelists überprüfen, Ausgabe annotieren, dass sie nicht ausgeführt werden kann)
  10. Mische nicht Daten und Code: Buffer Overflows können erlauben auf diese zuzugreifen, alle Daten extern

- Weitere nützliche Security Patterns / Idiome
  1. Verwende federated identity management (LDAP) wenn möglich
  2. Wenn Passwörter benötigt werden, überprüfe Password-Stärke, niemals im Client validieren, Hash + Salt auf Server
  3. Session Handling ist kritisch in Web Apps

- Implementierung
  - Bugs oder ungepflegte Implementierung kann zu Security Issues führen: dokumentiere Security-relevanten Code gut, weise Programmieren dahingehend hin, verwende Techniken für secure Entwicklung
  - Beinhalte die kritischten Software-Schwächen im Kopf

- Guidelines zum Testen
  1. Am effektivsten: Fokus darauf Risiken zu identifizieren (aber: nur identifizierte Risken werden getesten und es gibt keine Garantie dass nicht andere Schwachstellen existieren)
  2. Verwende Code Coverage Metrik
  3. Code und System Reviews

4. Versuche Security in frühen Phasen zu evaluieren
5. Nutze verschiedene Quellen für Testfälle
6. Verwende Standard Security Testing Umgebung die grundlegende Probleme überprüft
7. Lass Security Experten das System als Black Box oder White Box überprüfen

- Evaluation von Security: Third Party Evaluation mit einem Security Zertifikat
- Security Issues
  - Race conditions
    * treten auf wo Threads interagieren
    * Annahme muss für bestimmte Zeit gehalten werden
    * Beispiel: Time-of-Check vs Time-of-Use
    * Zwischen Prüfung Zugriffsrechte und Operation kann Unterbrechung auftreten
  - Buffer Overflows
    * Mehr Daten in Speicherbereich schreiben als vorgesehen
    * Manipulation von Rücksprungadresse
    * Beispiele C-Methode gets(...) hat keine Längen-Abfrage, besser fgets
  - SQL Injection
    * Wenn Abfrage als String verknüpft wird, kann diese so manipuliert werden, dass String beenedet wird und zusätzlich nach der Query noch weitere Operationen ausgeführt werden
    * Lösung: Verwendung von PreparedStatements
  - Krpytographie
    * machen immer Annahmen über Art der Verwendung
    * Beispiel Zufallszahlengenerator
      · jedoch sind Zufallszahlen i.d.R. Pseudo-Zufall, daher nicht Zufall sondern deterministisch wegen gleichbleibendem Seed, besser: dynamischer Seed (Maus-Bewegung, Hintergrundgeräusche, Netzwerk-Traffic)
      · richtig mit physikalischem Gerät und Schwingung gibt echten Zufall
    * Message: Bibliotheksfunktionen machen Annahmen
    * anderes Beispiel: RSA mit zu kleinen Schlüssel ist unsicher, auch wenn Algorithmus an sich sicher ist.
- noch viel mehr möglich, Vorlesung nur Beispiele

12 Continuous Integration

- Problem: Ön my machine it worksöder es funktionierte gestern - Entwicklungsumgebung ist anders als Kundenumgebung
- Testtreiber für Methoden sinnvoll
- ohne CI meist lokales Testen, nebenläufige Arbeiten, seltene Integrationstests (häufig erst bei Releases)
- CI: viel stärker integrierter Prozess anstatt nur in lokalen Prozessen
- Wichtige Punkte: Ansatz (kein Tool), mehrere Integrationen am Tag (oder noch deutlich häufiger), Automation macht es leichtgewichtiger, für Reussner Fortsetzung der agilen Prinzipien für spätere Phasen (Integration und Deployment), Hauptgrund: schnelle Feedback-Zyklen
- CI Entwicklungs-Prozess: manuell arbeiten und lokal testen, commit, Code gesetzt und gelinkt, dann automatisierte Integrationstests und Feedback, Fehlerbehebung
- Vorteile: Verringerung von Risiken, manuelle Prozesse werden automatisiert, erlaubt schnelles Deployment der Software - Continuous Delivery
- Ist extrem hohe Anzahl von Deployments sinnvoll? nicht für alle gleich beglückend - abhängig von Kontext
- Was wird benötigt für CI? Werkzeuge für Integration - Maven, Ant, make, ... - Versionierung, Test-Automatisierung, Prozess und Leute die sich daran halten
- Best Practices
  - Versionierung
  - Modellierung der Artefakte (welche versioniert? welche generiert und daher nicht versioniert?)
  - Konfigurationsdateien versionierbar
  - Builden für jede Umgebung
  - Builds für Subkomponenten (weniger Abhängigkeiten im Team)
- Tools: Jenkins (frühes Tool, viele Plugins), Travis CI (moderner, yml statt xml - leichter lesbar)
- Continuous Deliver
- Canary Release: erst CD für wenige User, dann später für alle sofern positives Feedback

13 Reviews

- Erinnerung: V-Modell - Reviews: informelle Verifikationstechnik - Menschen überprüfen Artefakte auf Fehler
- Review: Meeting von Menschen zur Überprüfung von Dokumenten - laufend im Projekt
- Retrospektive: War Prozess sinnvoll am Ende vom Projekt?
- Begriffe: Inspection, Team Review, Walkthrough, Pair Programming, Pass-around, Adhoc review
- keine ausführbaren Artefakte benötigt
- Reviews sind fast eine Silberkugel
- werden nicht oft verwendet - erhöht Kosten am Anfang vom Prozess
- Gefahren von Reviews: Testen wird vernachlässigt
- Gefahr für Reviews: werden gerne gestrichen bei Knappheiten, Leute nicht gut vorbereitet, ohne gute Fehlerkultur versuchen guten Code zu fabrizieren damit keine Fehler gefunden werden
- Nutzen von Reviews: besserer Überblick, weniger Single-Points-of-Failure (z.B. eine Person krank), für Anfänger ganz gut um von erfahreneren Kollegen zu lernen, in der Regel bessere Lesbarkeit des Codes (da Entwickler wissen, dass es Reviews geben wird)
- verschiedene Phasen in einem Review
  - Planungsphase - was sind Artefakte
  - Überblick - wann ist was bereit? Wer macht mit?
  - Vorbereitung - Dokumente werden geschickt (oder Spezifikationen)
  - Meeting - zusammensitzen
  - Rework - Autor setzt Dinge um
  - Follow-Up - falls nochmal gemeinsam drübergeschaut werden soll
  - Causual Analysis - Retrospektives Meeting (Warum gab es Probleme? Was sind häufige Fehler)
- Rollen in Review-Meeting
  - Author: Ersteller des Dokuments
  - Moderator: Ablauf des Meetings
  - Leser
  - Recorder: schreibt Protokoll

  - Verifier: überprüft ob Sachen sinnvoll umgesetzt sind
  - Wie viele Reader? Mehr als 5 nicht nötig.
- Inspektionsprozess
  - relativ naheliegend, geschwungen Dokumente und Prozessschritte
  - zeigt welche Rollen wo involviert sind
  - interessant z.B. wo Tippfehler hingehen (Rework)
- Techniken zum Lesen von Code
  - zeilenweise nicht unbedingt sinnvoll
  - besser Hypothesen-basiert (Wozu ist diese Schleife da? Macht sie das auch? If - ist Fehlerbehandlung? If in Schleife - Extraelement oder früherer Abbruch?)
  - falls kein Code: architectural reading: unklar, Pattern durch Hypothesen erkennen, Was sind Strukturen/Stile?
- Verschiedene Review-Typen (TODO Folie 15 einfügen), Pair Programming nicht beste Technik
- Psychologische Interaktions Patterns (Anti-Patterns)
  - The alcoholic pattern: Jemand mit schlechter Angewohnheit, versucht immer zu schauen wie weit er es treiben kann, bevor es klappt (nicht vorbereitet, zu spät)
  - The "now-I-have-got-you"pattern: unter die Nase reiben, dass Fehler gefunden wurde
  - The ßee-what-you-made-me"pattern: Ich hätte das erreichen können, wenn du nicht damit gekommen wärst...
  - The hurried pattern: überarbeitet, zeigt es aber nicht und bekommt zusätzliche Last aufgebrummt
  - The ïf-it-were-not-for-you"pattern: Wenn es nicht für dich wäre, hätte ich das erreicht...
  - The "Look-how-hard-I-tried"pattern: Leute dokumentieren wie hart sie arbeiten, Projekt scheitert aber ihnen liegt es nicht
  - The SSchlemiel"pattern: Verursacher ist ganz woanders, aber anderer steht als Verantwortlicher dar
  - The "yes-but"pattern: Vorschlag gemacht, dann ja aber... (typische Abwehrhaltung)
  - The "wouldn't-it-be-nice-if"pattern: Gegenteil zu yes but, etwas implementiert und dann kommt wäre es aber nicht schön? ja aber...