

Softwaretechnik II

Zusammenfassung WS17/18

Manuel Lang

1. März 2018

1 VORGEHENSMODELLE

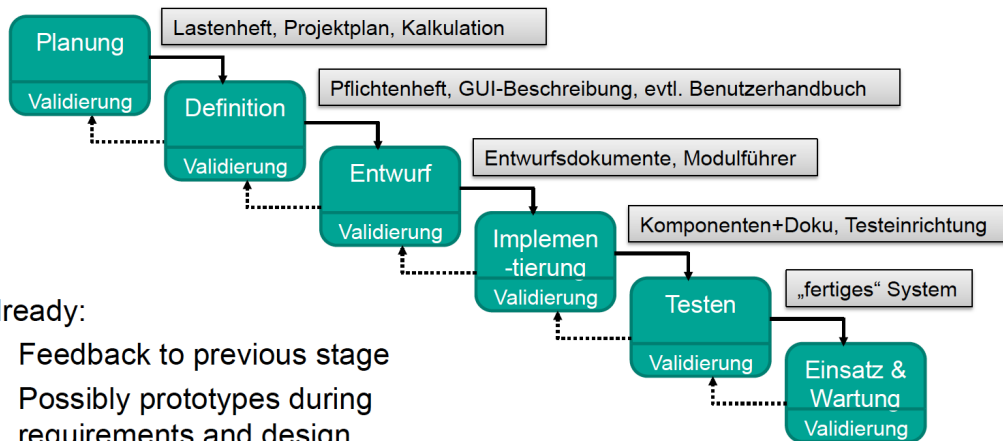
Vorteile strukturierter Vorgehensmodelle

- Reproduzierbarkeit (Erfahrungen für ähnliche Projekte)
- Skalierbarkeit (größere Komplexität schneller)
- Wiederverwendbarkeit (Code)
- Risikominimierung (Entwicklung 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

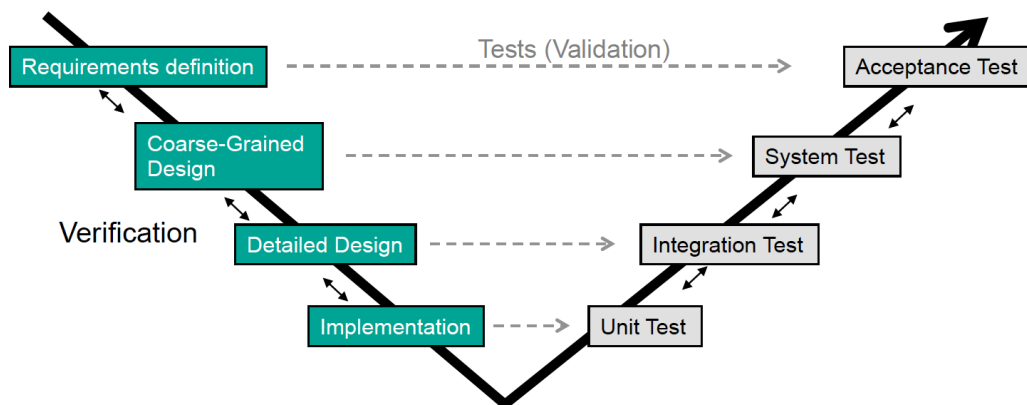
- The Waterfall Model as a sequential process model ...



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 activities 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 "Fertigstellung"urückspringen z.B. in Planungsphase, in der Praxis fast immer da z.B. langfristig 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

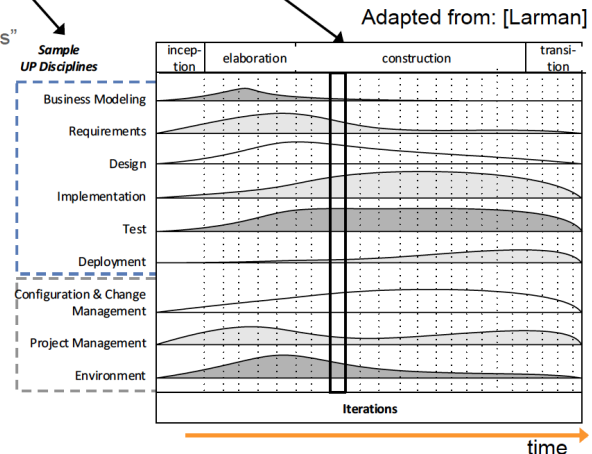
- UP (Unified Process)

- 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 and 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 (einhellig), 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 erfahrenen 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 TO-DOs 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?
 - Retrospective 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 Arbeitsaufgaben, 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 Mitgliedern diskutiert, oder vielleicht erst Prototyp nötig? Aktivität sollte nicht länger als 2 Tage sein, damit guter Überblick möglich, Abschätzen in Personestunden, 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 Gesamt-meeting
- * 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
 - nicht-funktional
 - Randbedingungen (z.B. gesetzliche Vorgaben oder Platformentscheidungen)