Softwaretechnik

Sommer 2019: Teil 10: Projektmanagement und Vorgehensmodelle

Henning Schnoor

Institut für Informatik, Christian-Albrechts-Universität zu Kiel

Projektmanagement und Vorgehensmodelle

Softwareprojektmanagement

Es geht darum, sicherzustellen dass Software rechtzeitig, mit den verfügbaren Ressourcen und mit hoher Qualität ausgeliefert wird.

- Sowohl funktionale als auch nicht-funktionale Anforderungen müssen erfüllt werden.
- Organisatorische und wirtschaftliche Rahmenbedingungen spielen eine Rolle.
- Oft werden neuartige Softwaretechniken eingesetzt.
  Schwer vorhersagbar

Ziele dieser Lehreinheit:

- Es werden einige Managementtechniken präsentiert, allerdings:
  Management kann man nur richtig durch managen lernen
  - Z.B. im Softwareprojekt!

Übersicht

Übersicht

Zeitplanung

- Das Projekt wird in Einzelaufgaben aufgeteilt.
  - Für jede Aufgabe wird die erforderliche Zeit mit den verfügbaren Ressourcen geschätzt.
- Es ist angestrebt die Aufgaben nebenläufig zu organisieren,
  - damit die verfügbaren Mitarbeiter optimal eingesetzt werden können.
- Dabei ist es wichtig, die Abhängigkeiten zwischen Aufgaben zu minimieren,
  - um Verzögerungen durch Warten zu vermeiden.
- Generell ist hier die Erfahrung und Intuition der Projektmanager gefragt.

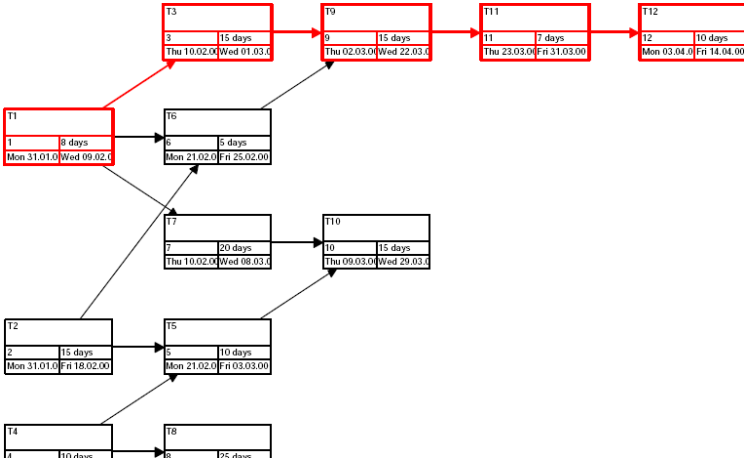
Netzpläne und Balkendiagramme

- Grafische Notationen, die den Projektablauf darstellen.
- Sie zeigen die Aufteilung in Teilaufgaben
- Netzpläne zeigen Abhängigkeiten zwischen Aktivitäten und kritische Pfade.
  - PERT Charts (Program Evaluation Review Technique)
- Balkendiagramme bilden die Aktivitäten auf Kalenderzeit ab.
  - Gantt Charts

Ein Beispiel für Aufgaben, deren Dauer und Abhängigkeiten

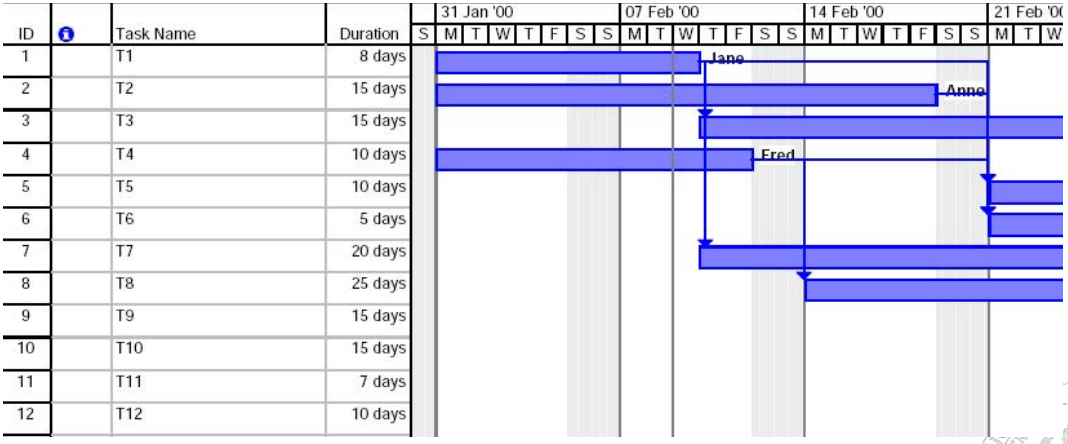
Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1
T4	10	
T5	10	T2, T4
T6	5	T1, T2
T7	20	T1
T8	25	T4
T9	15	T3, T6
T10	15	T5, T7
T11	7	T9
T12	10	T11

PERT chart in MS Project



Mit Visual Paradigm: `https://online.visual-paradigm.com/diagrams.jsp`

Gantt chart in MS Project



ID	Task Name	Duration
1	T1	8 days
2	T2	15 days
3	T3	15 days
4	T4	10 days
5	T5	10 days
6	T6	5 days
7	T7	20 days
8	T8	25 days
9	T9	15 days
10	T10	15 days
11	T11	7 days
12	T12	10 days

Übersicht

Übersicht

Typische Rollen im Projektteam

- Project Manager
- Domain Expert
- Domain Analyst
- Requirements Engineer
- System Architect
- Software Architect
- Entwerfer
- GUI Designer
- Bibliothekar
- Programmierer
- Technologieexperte
- Reviewer
- Tester
- Installation Engineer
- System Administrator
- Maintenance Engineer

Generell:

- Eine Person kann mehrere Rollen übernehmen.
- Eine Rolle kann durch mehrere Personen übernommen werden.
- Die Zuweisung von Rollen zu Personen kann sich während eines Projektes ändern.

Aufbau von Projektteams

Brooks'sches Gesetz

Adding manpower to a late software project makes it later

- Zusätzliche Mitarbeiter müssen von den vorhandenen geschult und eingearbeitet werden.
- Es muss eine weitere Aufgabenteilung und erneute Abstimmung vorgenommen werden.

$\Rightarrow$ Die Produktivität der alten Mitarbeiter sinkt.

$\Rightarrow$ Die optimale Mitarbeiterzahl muss rechtzeitig zur Verfügung stehen.

Übersicht

Übersicht

Elemente von Vorgehensmodellen

Ein Vorgehensmodell ...

- legt die Reihenfolge der Aktivitäten (Phasen) der Entwicklung und der Evolution von Software fest.
- legt die Übergangskriterien zwischen den Phasen fest.
- ist Grundlage für das Management von Projekten.

Grundlegende Elemente:

- Aktivitäten
- Dokumente
- Werkzeuge
- Rollen (für Personen)

Das 0. Vorgehensmodell der Software-Entwicklung

code & fix

1. Schreibe ein Programm!
2. Finde und behebe die Fehler in dem Programm!

Nachteile:

- Bei der Behebung von Fehlern wird das Programm häufig so umstrukturiert, dass es schlechter verständlich wird und weitere Fehlerbehebungen immer teurer werden.
- Dies führt zu der Erkenntnis, dass eine Entwurfsphase vor der Programmierung benötigt wird.

Erweiterte Vorgehensmodelle mit Entwurfsphase

Weiterhin Probleme:

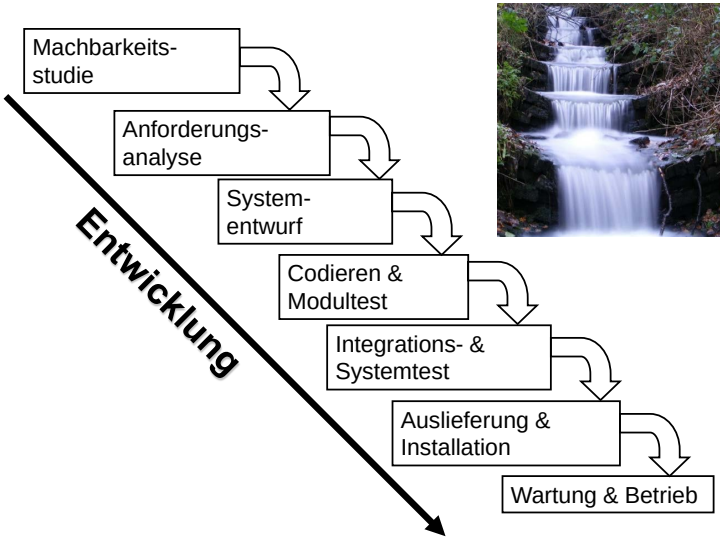
- Selbst gut entworfene Software wird vom Endbenutzer oft nicht akzeptiert.
  $\Rightarrow$ Dies führt zu der Erkenntnis, dass eine Anforderungsdefinition vor dem Entwurf benötigt wird.
- Fehler sind schwierig zu finden, da Tests schlecht vorbereitet und Änderungen unzureichend durchgeführt werden.
  $\Rightarrow$ Dies führt zu einer separaten (Integrations-) Testphase.
- Auch nach der Fertigstellung muss die Software weiterentwickelt werden.

Dies führt uns zum Wasserfallmodell.

Übersicht

Übersicht

Das Wasserfallmodell



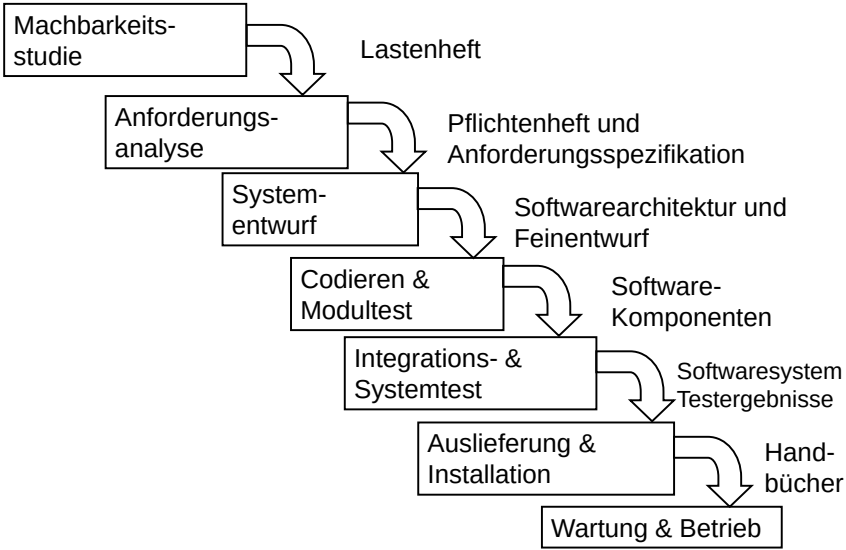
Das Wasserfallmodell

Das Wasserfall-Modell ist als erstes umfassendes Modell für das Software Engineering zu bezeichnen und basiert auf dem stagewise model [Ben56]. Weiterentwickelt wurde es u.a. durch Royce und Boehm, die die ursprüngliche lineare Abfolge von Aktivitäten um Rücksprungmöglichkeiten ergänzten und dem Modell den durch dessen Form begründeten und noch heute verwendeten Namen Wasserfall-Modell gaben [Roy87; Boe81]. Das Modell lässt sich durch die obige Abbildung beschreiben.

Charakteristika des Wasserfallmodells

- Software wird in sukzessiven Stufen entwickelt.
- Jede Aktivität ist in der richtigen Reihenfolge und in der vollen Breite vollständig durchzuführen.
- Der Entwicklungsablauf ist sequentiell.
  Jede Aktivität muss beendet sein, bevor die nächste anfängt.
- Rückkopplungsschleifen zwischen den Stufen sind begrenzt möglich.
- Nutzerbeteiligung ist nur in den ersten Phasen vorgesehen.
- Einfach, verständlich und benötigt relativ wenig Managementaufwand.
- Name Wasserfallmodell: Produkte einer Phase „fallen“ wie bei einem Wasserfall in die nächste Phase.

Das Wasserfallmodell: Produkte



Machbarkeitsstudie

- Prüfen der fachlichen und technischen Durchführbarkeit,
  - Softwaretechnische Realisierbarkeit,
  - Verfügbarkeit von Entwicklungs- und Zielmaschinen.
- Prüfen alternativer Lösungsvorschläge,
  - Beispiel: Kauf und Anpassung von Standardsoftware vs. Individualentwicklung.
- Prüfen der personellen Durchführbarkeit.
- Prüfen der ökonomischen Durchführbarkeit.
- Produkt u.a.: Das Lastenheft
  - repräsentiert die fachlichen, wirtschaftlichen, technischen und organisatorischen Erwartungen des Auftraggebers und
  - dient als Grundlage zur Erstellung des Pflichtenheftes durch den Auftragnehmer.

Anforderungsanalyse und Entwurf: Aktivitäten / Phasen

- Anforderungsanalyse → Pflichtenheft — Das „was“
  - basiert evtl. auf einer Machbarkeitsstudie,
  - produziert funktionale und nicht-funktionale Systembeschreibung, Zeitverhalten der einzelnen Funktionen, Schnittstellen, Ziel-Hardwareumgebung, etc.
  - Eine der schwierigsten Aufgaben in der Software-Entwicklung besteht darin, die tatsächlichen Anforderungen der Anwender zu ermitteln.
  - Interaktion zwischen Anwender und Entwickler ist nötig.
- Entwurf → Systemarchitektur — Das „wie“
  - produziert z.B. Spezifikation der Komponenten für die Implementierung, d.h. Schnittstellen mit Definition der Import- / Export-Beziehungen zwischen Komponenten
  - Der detaillierte Entwurf spezifiziert dann die Eigenschaften der einzelnen Komponenten.

Implementierung und Test: Aktivitäten / Phasen

- Implementierung → Software-Komponenten
  - Produziert Modulrümpfe in einer ausführbaren Sprache (Programmiersprache) nach Vorgabe der Systemarchitektur
  - Test einzelner Module (Unit-Tests)
    - Automatisieren!
  - Programmdokumentation
  - genannt: „programming-in-the-small“
- Test und Integration → Softwaresystem, Testpläne und Testergebnisse
  - Zusammenfügen der einzelnen Module zum System
  - Produziert Testpläne, Testtreiber, Testergebnisse für Gruppen von Modulen (Teilsysteme, Komponenten) und das Gesamtsystem
    - Automatisieren!
  - genannt: „programming-in-the-large“

Installation, Inbetriebnahme und Abnahme

- Installation, Inbetriebnahme und Abnahme
  $\rightarrow$ Technische Handbücher, Benutzungshandbücher
  - produziert Installation der lauffähigen Software auf der Zielmaschine, Installationsanleitung, Überführung vorhandener Daten, Schulung, Nutzungsbeschreibung
- Die Abnahmephase: Tätigkeiten
  - Mit der Übernahme verbunden ist im Allgemeinen ein Abnahmetest.
  - Innerhalb einer Abnahme-Testserie ist es auch sinnvoll, Belastungstests durchzuführen.
  - Ab diesem Zeitpunkt unterliegt das Produkt dann der Wartung & Pflege.

Betrieb, Wartung und Pflege: Aktivitäten / Phasen i

Wartung $\rightarrow$ verändertes Softwaresystem

- Produziert Veränderungen (neue Versionen) aller Ergebnisse aus allen vorigen Phasen je nach Anforderung,
- Zweierlei Arten der Wartung (Maintenance):

  progressiv Modifikation funktionaler und nicht-funktionaler Aspekte: Pflege

  korrektiv Fehlerbehebung, Optimierung

[Par94]
Software altert.

Betrieb, Wartung und Pflege: Aktivitäten / Phasen ii

[HRH09]

- Der kontinuierliche Betrieb von Softwaresystemen erfordert eine besondere Betrachtung und hat auch Einfluss auf den Entwurf der Software.
- Das Monitoring-Framework Kieker [Roh+08; Hoo+09] unterstützt den kontinuierlichen Betrieb von Softwaresystemen:
  `http://kieker-monitoring.net/`

[Gre+10]
Auch neuere Softwaretechnikbücher widmen dem Betrieb nur wenig Raum, z.B. nur eine Seite in [Gre+10, Abschnitt 8.2.3]. Selbst in [Bal11] nimmt der Betrieb nur einen kleinen Teil ein (Wartung und Pflege wird dort unter Betrieb gefasst).

Aufteilung der Wartungskosten [Kos15], https://wiki.uef.fi/display/tktWiki/Jussi+Koskinen

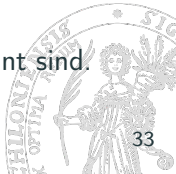
Year	Proportion of software maintenance costs	Definition	Reference
2000	>90%	Software cost devoted to system maintenance & evolution / total software costs	Erlikh (2000)
1993	75%	Software maintenance / information system budget (in Fortune 1000 companies)	Eastwood (1993)
1990	>90%	Software cost devoted to system maintenance & evolution / total software costs	Moad (1990)
1990	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Huff (1990)
1988	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Port (1988)
1984	65-75%	Effort spent on software maintenance / total available software engineering effort.	McKee (1984)
1981	>50%	Staff time spent on maintenance / total time (in 487 organizations)	Lientz & Swanson (1981)
1979	67%	Maintenance costs / total software costs	Zelkowitz *et al.* (1979)

Qualitätsmerkmale

- Produktnutzung
  - Die Qualitätsmerkmale Zuverlässigkeit, Verfügbarkeit, Effizienz, Korrektheit, Sicherheit (Safety und Security) und Ergonomie sind zentral.
  - Im Wesentlichen geht es hier um externe Qualitätsmerkmale.
- Wartung & Pflege
  - Die Merkmale Änderbarkeit, Modularität, Portierbarkeit, Testbarkeit und Verständlichkeit kommen hinzu.
  - Im Wesentlichen geht es hier um interne Qualitätsmerkmale.
- Verbesserung der Wartung durch Sanierung
  - Ziel: Verbesserung der internen Qualitätsmerkmale
    - Aufwand für Veränderungen verringern,
    - Wirtschaftlichkeit steigern.
  - Stichworte: Refactoring [Fow+99; Ker04; RL04] und Reengineering [DDN03].

Probleme mit dem Wasserfallmodell

- Es ist nicht immer sinnvoll
  - alle Entwicklungsschritte in der vollen Breite und vollständig durchzuführen und
  - alle Entwicklungsschritte sequentiell durchzuführen.
- Häufig sind Rückschritte nötig weil:
  - Falsche Anforderungsdefinition: falsch verstanden oder gesagt, nicht realisierbar,
  - Tatsächliche Anforderungen können oft nur durch den praktischen Einsatz eines Systems überprüft und ermittelt werden.
- Die Anforderungen ändern sich häufig.
- I.A. kann eine Phase nicht komplett abgeschlossen werden, bevor die nächste beginnt.
- Bestenfalls geeignet, wenn die Anforderungen tatsächlich vollständig bekannt sind.



Probleme mit dem Wasserfallmodell (Forts.)

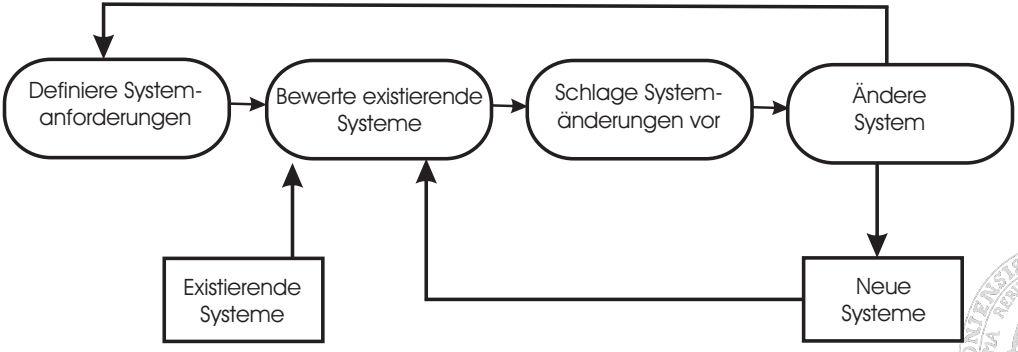
- Gefahr, dass die Dokumentation wichtiger wird als das eigentliche System,
- Risikofaktoren werden u.U. zu wenig berücksichtigt, da der einmal festgelegte Entwicklungsablauf durchgeführt wird.
  Nur die (optionale) Machbarkeitsstudie dient der einmaligen Risikoabschätzung.
- In der Praxis werden die einzelnen Phasen daher nicht streng sequentiell, sondern nebenläufig durchlaufen.
- Trotzdem: Manager lieben es!

Übersicht

Übersicht

Iterative, inkrementelle und evolutionäre Entwicklung als Verbesserung des Wasserfallmodells

Existierende Systeme müssen berücksichtigt werden:



Typische Probleme mit evolutionärer Entwicklung

Probleme:

- Der Prozess ist nicht wirklich „sichtbar“
  Systeme sind oft schlecht strukturiert.
- Vertragsgestaltung
  - Festpreise bei Vertragsabschluss (Werkverträge)
  - Dienstleistungsverträge können eine Lösung darstellen, erfordern aber eine gute Vertrauensbasis zwischen Auftraggeber und -nehmer.

Anwendbarkeit

- für kleine bis mittelgroße (interaktive) Systeme,
- für Teile von größeren Systemen (z.B. die Benutzungsschnittstelle),
- Für Systeme mit kurzer Lebensdauer
  $\rightarrow$ weiß man das vorher?

Übersicht

Übersicht

Definition (Evolutionäre Prototypen)

- Ziel ist, mit den Anwendern/Kunden von einer initialen Version per „Evolution“ zu einem nutzbaren System zu kommen.

Definition (Explorative Prototypen (Throw-away Prototyping))

- Ziel ist, (ausschließlich) die Anforderungen besser zu verstehen.

Definition (Experimentelle Prototypen)

- Dienen zum Erkunden der technischen Machbarkeit.

In der Praxis werden Prototypen häufig für mehrere Ziele genutzt [CH10].

Horizontale vs. vertikale Prototypen

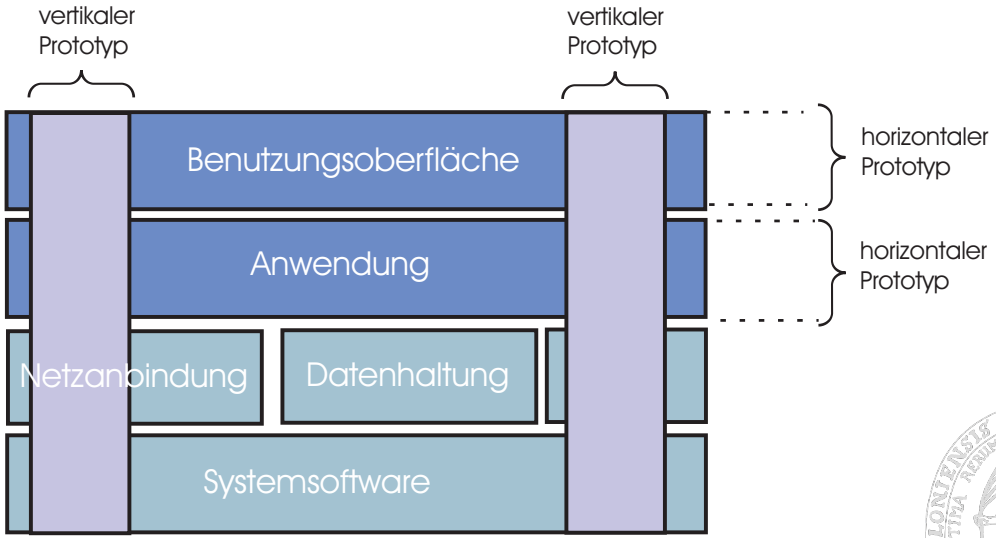
Definition (Horizontaler Prototyp)

- Realisiert nur spezifische Ebenen des Systems.
- Die betreffende Ebene wird möglichst vollständig realisiert.

Definition (Vertikaler Prototyp)

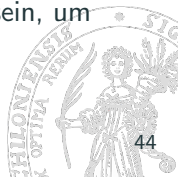
- Implementiert ausgewählte Teile des Zielsystems vollständig durch alle Ebenen hindurch.
- Dort geeignet, wo die Funktionalitäts- und Implementierungsoptionen noch offen sind.

Horizontaler und vertikaler Prototyp



Horizontaler und vertikaler Prototyp i

Unterscheiden lassen sich Prototypen in diesem Modell in horizontale und vertikale Prototypen. In einem horizontalen Prototypen werden nur spezifische Ebenen der zukünftigen Software realisiert, bspw. die Benutzeroberfläche. Die entsprechende Ebene wird dabei möglichst vollständig realisiert, die mit dieser Ebene verknüpften Funktionalitäten sind jedoch nicht vorhanden. Durch diese Möglichkeit eignen sich horizontale Prototypen besonders für Demonstrationszwecke, bei denen das optische Erscheinungsbild der Software im Vordergrund steht. Bezogen auf die Benutzerschnittstelle bedeutet dies, dass die Programmoberfläche sichtbar ist, die Funktionalität jedoch fehlt. In einem vertikalen Prototypen werden dagegen einzelne Aspekte der Software über alle Ebenen hinweg realisiert. Dies kann notwendig sein, um die Umsetzbarkeit bestimmter technischer Anforderungen zu testen. Vertikale



Horizontaler und vertikaler Prototyp ii

Prototypen sind daher für Labormuster oder die Entwicklung von Pilotsystemen geeignet. Vorstellbar sind auch Kombinationen beider Typen, die begrenzte Ausschnitte des Zielsystems über mehrere Ebenen hinweg enthalten.

Software-Prototyp vs. Konventioneller Prototyp

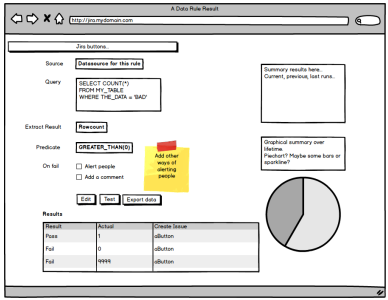
- Ein Software-Prototyp ist nicht das erste Muster einer Produktserie
  - Gegenbeispiel: Massenproduktion in der Autoindustrie
- Ein Software-Prototyp zeigt ausgewählte Eigenschaften des Zielproduktes im praktischen Einsatz.
  - Analoges Beispiel: Pappmodell einer Brücke
- Prototypen dienen primär zur Klärung von Problemen
  »I can't tell you what I want, but I'll know it when I see it«

Herausforderungen mit Prototypen

Anwender haben oft Schwierigkeiten, zwischen Prototyp und Produktionssystem zu unterschieden.

Weiterhin Gefahr, dass in nachfolgenden Versionen die komplette Systemarchitektur überarbeitet werden muss.

$\Rightarrow$ Refactoring / Sanierung (siehe oben).



https://balsamiq.com/

Übersicht

Übersicht

Betrachtete Vorgehensmodelle

In LE 10:

- Das Wasserfallmodell: Dokumentengetrieben
- Iterative, Inkrementelle und Evolutionäre Entwicklung: Anforderungsgetrieben

Weitere interessante Modelle:

- Rational Unified Process: Architektur- und Anwendungsfallgetrieben
- Das V-Modell: Qualitätssicherungsgetrieben
- Spiral-Modell: Risikogetrieben
- b-Modell: Wartungsgetrieben
- Agile Softwareentwicklung u.a. mit SCRUM und Kanban betrachten wir im Softwareprojekt.

Let's read [Rup10]!

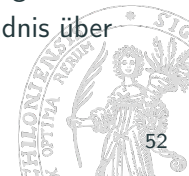
Das Spiral-Modell i

Das Spiral-Modell wurde von Boehm beschrieben [Boe88; Boe+98]. Es handelt sich um ein Vorgehensmodell, dessen Konzept die Entwicklung von Software in Teilprodukte und Verfeinerungsebenen zerlegt. In jeder Verfeinerungsebene sind dabei vier Schritte zu durchlaufen:

1. Identifikation der Ziele, die für die Erstellung des Teilprodukts erforderlich sind. Entscheidung über alternative Lösungsmöglichkeiten zur Erstellung des Teilprodukts. Ermittlung von Randbedingungen, die bei der Wahl der Alternativen zu beachten sind.

Das Spiral-Modell ii

2. Evaluation der Alternativen unter Berücksichtigung der Ziele und Randbedingungen. Sofern Risiken für die Durchführung von Alternativen identifiziert wurden, sind Strategien zur Überwindung der Risken zu entwickeln, z.B. Prototypenerstellung, Simulationen oder Benutzerbefragungen.
3. Je nach verbliebenem Risiko wird ein geeignetes Vorgehensmodell für die Durchführung dieses Schrittes gewählt, z.B. Wasserfall-Modell, V-Modell oder Rapid Prototyping. Zur Risikominimierung ist auch die Kombination mehrerer Vorgehensmodelle möglich.
4. Überprüfung der vorhergehenden drei Schritte. Planung von Zielen und Ressourcen für den nächsten Zyklus. Darin enthalten kann auch eine Aufteilung der Ergebnisse in Teilprodukte sein, die dann separat weiterentwickelt werden. Einverständnis über den nächsten Zyklus sicherstellen.

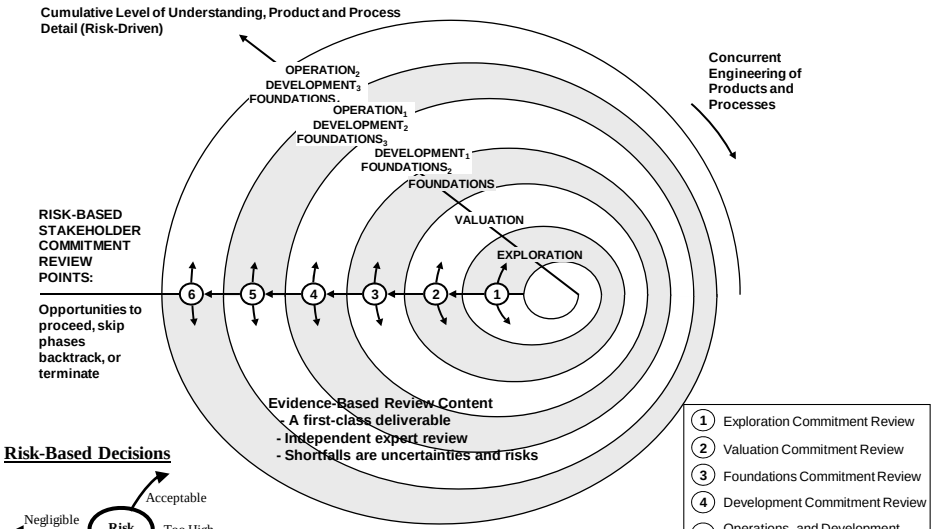


Das Spiral-Modell iii

Das mehrfache Durchlaufen dieser vier Schritte wird als Spirale dargestellt.

Das Spiralmodell ist ein risikogetriebenes Modell, das als Hauptziel die Minimierung von Risiken definiert. Die Ziele für einen Zyklus werden aus den Ergebnissen des vorherigen Zyklus abgeleitet. Ein weiteres Ziel ist die Reduzierung der Entwicklungskosten. Dafür wird die Entwicklung zunächst mit den grundlegenden Konzepten begonnen und die Erstellung der Software mit möglichst eng aneinander liegenden Spiralzyklen durchgeführt. Die Fertigstellung der Entwicklung wird durch Analysen des erreichten Produkts definiert. Durch die kritische Betrachtung von Zielsetzung und Anforderungen kann die teure Entwicklung unnötiger Funktionen vermieden werden.

The Incremental Commitment Spiral Model



References i

- Balzert, Helmut (2011). Lehrbuch der Softwaretechnik: Entwurf, Impementierung, Installation und Betrieb. 3. Spektrum Akademischer Verlag.
- Benington, H.D. (1956). "Production of Large Computer Programs". In: Proceedings of the ONR Symposium on Advanced Program Methods for Digital Computers.
- Boehm, B.W. (1981). Software Engineering Economics. Prentice-Hall.
- — (1988). "A Spiral Model of Software Development and Enhancement". In: IEEE Computer 21.5, S. 61–72.
- Boehm, B.W., A. Egyed, J. Kwan, D. Port, A. Shah und R. Madachy (1998). "Using the WinWin spiral model: a case study". In: IEEE Computer 31.7, S. 33–44.

References ii

- Christensen, Henrik Baerbak und Klaus Marius Hansen (Jan. 2010). "An empirical investigation of architectural prototyping". In: Journal of Software and Systems 83.1, S. 133–142. DOI: 10.1016/j.jss.2009.07.049.
- Demeyer, Serge, Stéphane Ducasse und Oscar Nierstrasz (2003). Object-Oriented Reengineering Patterns. Morgan Kaufmann.
- Fowler, Martin, Kent Beck, John Brant, William Opdyke und Don Roberts (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. ISBN: 0201485672.
- Grechenig, Thomas, Mario Bernhart, Roland Breiteneder und Karin Kappel (2010). Softwaretechnik. Pearson Studium.

References iii

- Hoorn, André van, Matthias Rohr und Wilhelm Hasselbring (Okt. 2009). "Engineering and Continuously Operating Self-Adaptive Software Systems: Required Design Decisions". In: Design for Future – Langlebige Softwaresysteme. CEUR Workshop Proceedings, S. 52–63.
- Hoorn, André van, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey und Dennis Kieselhorst (Nov. 2009). Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Techn. Ber. TR-0921. Department of Computer Science, University of Kiel, Germany. URL: http://eprints.uni-kiel.de/14459/.
- Kerievsky, Joshua (2004). Refactoring to Patterns. Addison-Wesley Professional. ISBN: 0321213351.

References iv

- Koskinen, Jussi (Apr. 2015). Software Maintenance Costs. Techn. Ber. https://wiki.uef.fi/display/tktWiki/Jussi+Koskinen. School of Computing, University of Eastern Finland, Joensuu, Finland.
- Parnas, David Lorge (1994). "Software aging". In: Proceedings of the 16th international conference on Software engineering (ICSE 1994). Sorrento, Italy: IEEE Computer Society Press, S. 279–287. ISBN: 0-8186-5855-X.
- Rohr, Matthias, André van Hoorn, Jasminka Matevska, Nils Sommer, Lena Stoever, Simon Giesecke und Wilhelm Hasselbring (Feb. 2008). "Kieker: Continuous Monitoring and on demand Visualization of Java Software Behavior". In: Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08). Hrsg. von Claus Pahl. Innsbruck, Austria, S. 80–85. ISBN: 978-0-88986-715-4.

- Roock, Stefan und Martin Lippert (2004). Refactorings in großen Softwareprojekten: Komplexe Restrukturierungen erfolgreich durchführen. dpunkt Verlag. ISBN: 13 978-3-89864-207-1.
- Royce, W.W. (1987). "Managing the development of large software systems: concepts and techniques". In: Proceedings of the Ninth International Conference on Software Engineering, S. 328–338.
- Ruparelia, Nayan B. (2010). "Software development lifecycle models". In: SIGSOFT Softw. Eng. Notes 35.3, S. 8–13. DOI: 10.1145/1764810.1764814.