Der Software-Lebenszyklus

Stammvorlesung Softwaretechnik

Andreas Zeller
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

2005-10-20

Zur Vorlesung

Dozenten: Prof. Joachim Hertel und Prof. Andreas Zeller
Vorlesung: Mo 9–11, Do 9–11 hier
Übungsleiter: Silvia Breu und Valentin Dallmeier
Übung: nach Vereinbarung – Einteilung Montag
Prüfung: Klausur Mitte und Ende des Semesters

Inhalte

Die Vorlesung vertieft den Stoff aus dem Software-Praktikum:

- Moderne Programmierkonzepte
- Software-Prozesse
- Validierung
- Verifikation
- Software-Wartung
- Projektorganisation, Aufwands- und Zeitschätzungen
- Personalauswahl und -einsatzplanung, Kommunikation
- Metriken, Projektcontrolling
- Business Case und Change Request Management
- Risiko- und Qualitätsmanagement

Materialien

- *Lehrbuch der Software-Technik* von Balzert
- Weitere Bücher im Bibliotheksapparat
- Vorlesungsskript und -Folien

Fragestunde

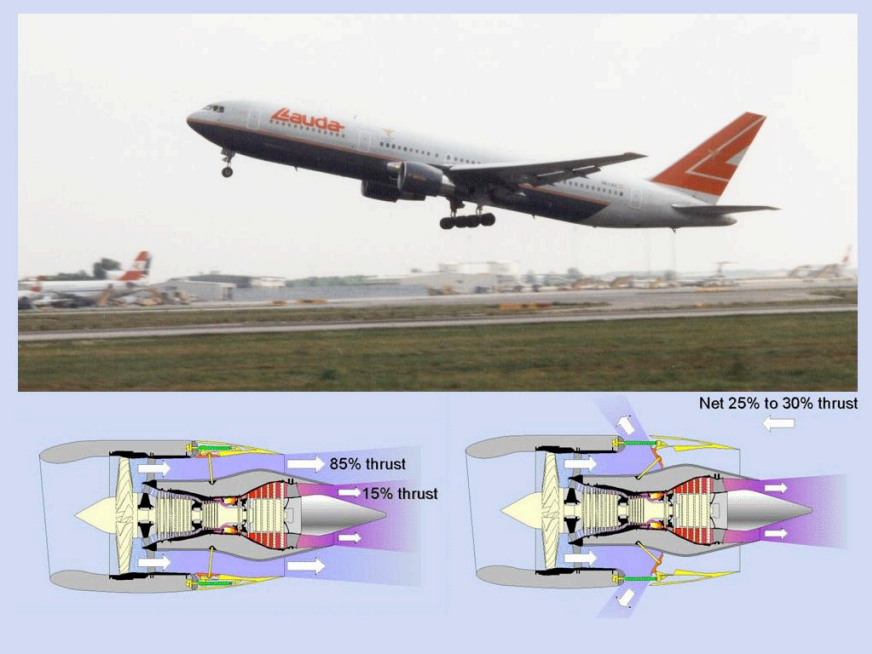
Was möchten Sie lernen?

Das Kampfflugzeug F-16

Der F-16 Flip-Flop

(Noch) intakte F-16 auf Landebahn

Das Rätsel der Schubumkehr



Der korrekte Airbus

15:33:56 Reverse auf?
15:33:57 Ja's voll
15:33:58 Clack
15:34:01 Hundert
15:34:02 Weiter bremsen.
15:34:05 Scheiße.
15:34:06 Was machen wir jetzt?
15:34:08 Tja, du kannst nix mehr machen.
15:34:10 Ich möcht nicht da gegen knallen.
15:34:11 Dreh'n weg.
15:34:12 Was?
15:34:12 Dreh ihn weg.
15:34:16 Scheiße.
15:34:17 (Noise of crash)

Der Rückruf-Roller

Fehler im Segway-Roller (2003):

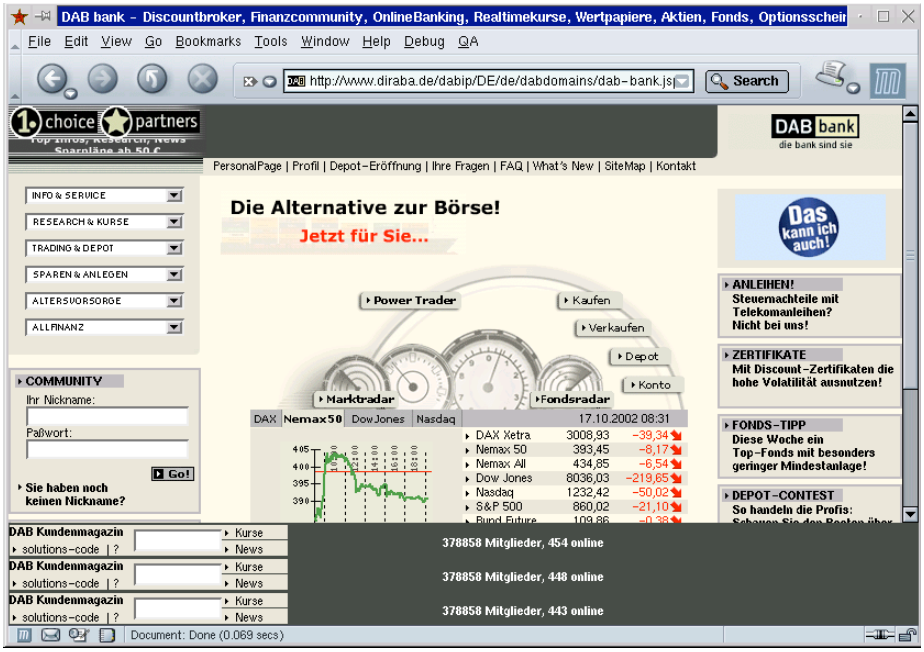
Bei schwächer werdenden Batterien halten die elektronischen Stabilisatoren den Roller nicht mehr aufrecht

Folge: drei Verletzte +
vorläufiger Rückzug des Produkts

50% aller Autopannen
passieren durch Softwarefehler!

Der Eudora-Knigge

Wer wird Millionär?



Code-and-Fix-Zyklus

geeignet für 1-Person-Projekte und Praktikumsaufgaben im ersten Semester

Ablauf:

1. Code schreiben und testen
2. Code „verbessern“ (Korrektur, Erweiterung, Effizienz…)
3. GOTO 1

Ist das Problem klar spezifiziert und kann eine Person die Implementierung allein bewältigen, ist wenig dagegen zu sagen.

Code-and-Fix-Zyklus (2)

Jedoch:

- Wartbarkeit und Zuverlässigkeit nehmen kontinuierlich ab („Entropie“)
- Wenn der Programmierer kündigt, ist alles vorbei
- Heutige Projekte umfassen -zig Personenjahre
- Wenn Entwickler und Anwender nicht identisch sind, gibt es oft Meinungsverschiedenheiten über den erwarteten/realisierten Funktionsumfang

⇒ Sogenannte *Software-Krise* (1968)

Prozessmodelle

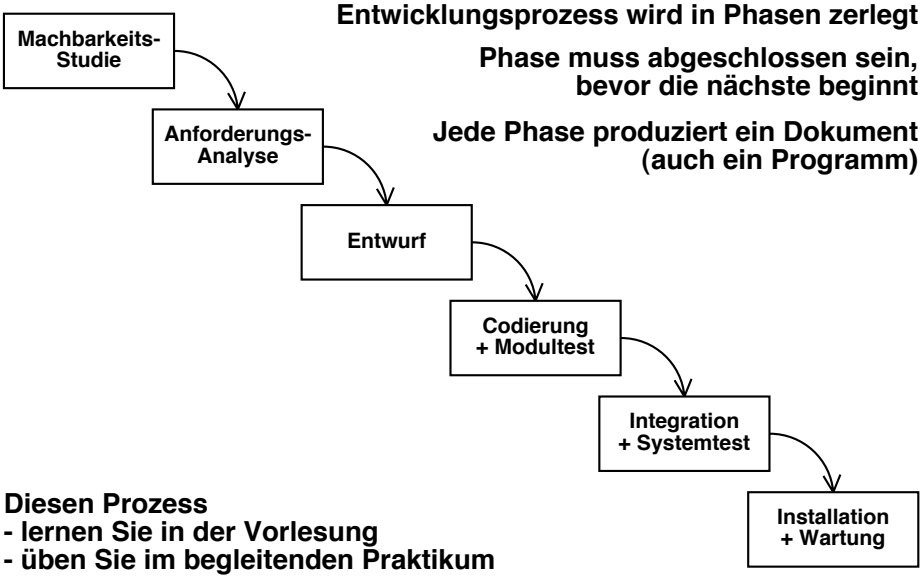
Die Software-Krise führte zur Entwicklung *strukturierter Prozessmodelle.*

Ein Prozessmodell bietet eine Anleitung für den Entwickler, welche Aktivitäten als nächstes kommen.

Vorteil: Der Entwicklungsprozess wird (aus betriebswirtschaftlicher Sicht) *plan- und kontrollierbar*.

- Projektplanung
- Kostenkontrolle
- Abnahme von Zwischen- und Endprodukten
- …

Wasserfallmodell



Entwicklungsprozess wird in Phasen zerlegt

Phase muss abgeschlossen sein, bevor die nächste beginnt

Jede Phase produziert ein Dokument (auch ein Programm)

Diesen Prozess
- lernen Sie in der Vorlesung
- üben Sie im begleitenden Praktikum

Durchführbarkeitsstudie

Die Durchführbarkeitsstudie soll *Kosten und Ertrag* der geplanten Entwicklung abschätzen

Dazu (grobe) *Analyse* des Problems mit Lösungsvorschlägen, Alternativlösungen

Für jeden Lösungsvorschlag werden benötigte *Ressourcen, Entwicklungszeit und Kosten* geschätzt

Die Ergebnisse werden in einem Dokument festgehalten

Dieses Dokument ist häufig Grundlage eines *Angebotes* an einen potentiellen Kunden

Durchführbarkeitsstudie (2)

Problem: oft hoher Zeitdruck und begrenzte Ressourcen
⇒ ungenaue Schätzungen

Ist die Aufgabe exakt umrissen und das Umfeld bekannt, kann die Durchführbarkeitsstudie auch fehlen.

Ergebnis der Phase: *Durchführbarkeitsstudie*

Anforderungsanalyse

In der Anforderungsanalyse wird exakt festgelegt, *was* die Software leisten soll (aber nicht, *wie* diese Leistungsmerkmale erreicht werden).

Alle relevanten Funktions- und Qualitätsmerkmale müssen spezifiziert werden!

Diese Angaben werden im *Pflichtenheft* (auch *Lastenheft* genannt) dokumentiert.

Das Pflichtenheft ist Bestandteil des Vertrages.

Anforderungsanalyse (2)

Eine Anforderungsdefinition muß

verständlich sein, und zwar für Auftraggeber und Auftragnehmer

präzise sein, damit es hinterher nicht zu Streit um die Auslegung kommt

vollständig und konsistent sein, da Lücken und Widersprüche zu teuren Rückfragen führen – oder zu Katastrophen.

Anforderungsanalyse (3)

Am präzisesten sind *formale Spezifikationen*, die sich auch formal und (teil-)automatisch auf Vollständigkeit und Widerspruchsfreiheit überprüfen lassen.

Nachteil allerdings: Kunde versteht formale Spezifikationen nicht.

Deshalb häufig Verwendung von semiformalen Methoden, z.B. Entity-Relationship Diagramme, Datenflußdiagramme

oder sogar Prosa (mit standardisiertem Vokabular/Glossar)

Häufig ist die *Benutzeranleitung* (in vorläufiger Form) bereits Bestandteil des Pflichtenheftes.

Ergebnis der Phase: *Pflichtenheft*

Entwurf

Im Entwurf wird die *Systemarchitektur* festgelegt:

- Welche Module (Objekte, Klassen) gibt es?
- Welche Beziehungen bestehen zwischen ihnen?

Häufiges Vorgehen: *Top-Down Entwurf*

Zerlegung des Systems in Komponenten, Zerlegung der Komponenten in Unterkomponenten usw.

Oft ist die Verwendung spezieller Architektur- oder Systembeschreibungssprachen sinnvoll.

Ergebnis der Phase: *Entwurfsbeschreibung*, die die Systemarchitektur festhält.

Codierung und Modultest

Eigentliche Implementierungs- und Testphase

Codierung und Testen war früher die einzige Phase.

Ergebnisse der Phase:

- *Implementierungsbericht*, der Details der Implementierung beschreibt (etwa Abweichungen vom Entwurf, Abweichungen vom Zeitplan, Begründungen dazu)
- *Testbericht*, der die durchgeführten Tests und ihre Ergebnisse beschreibt
- getesteter Programmtext der einzelnen Module

Programmierrichtlinien

Firmen verwenden oft *Programmierrichtlinien*, z.B. Programmlayout, Namenskonventionen, Kommentierkonventionen

Das ist aber fragwürdig, denn veraltete Konventionen können die Qualität reduzieren:

> Folgendes Beispiel wurde uns glaubwürdig berichtet: In einer Abteilung einer bekannten Automobilfirma werden Variablennamen zentral vergeben und durchnumeriert, nach dem Schema „Für Modul x darfst du die Variablen VW1344 bis VW1576 verwenden“

Programmierrichtlinien (2)

Heute werden Programmierrichtlinien nicht nur von (benutzenden) Firmen, sondern gleich von den *Sprachautoren* veröffentlicht.
Vorteil: Richtlinien werden von Anfang an eingesetzt.

> Beispiel: Sun’s *Code Conventions for the Java Programming Language*

Integration und Systemtest

Die Module werden zu einem Programm zusammengebunden
Das Zusammenspiel der einzelnen Komponenten wird getestet
Kann mit der vorangegangenen Phase verschmolzen sein
Schließlich wird das gesamte System getestet:

- zunächst nur innerhalb der Entwicklungsorganisation (*Alpha-Test*)
- später bei ausgewählten Kunden (*Beta-Test*)

Ergebnisse der Phase:

- *Laufendes System*
- *Benutzeranleitung* in endgültiger Form

Installation und Wartung

Die Installation einer neuen Software findet häufig in zwei Phasen statt:

- Zunächst Auslieferung nur an ausgewählte, vertrauenswürdige Kunden. (*Beta-Test*)
- Dann Auslieferung an alle Kunden

Wartungskosten = 60% der gesamten Softwarekosten!
Davon wiederum

- 20% Fehlerbeseitigung
- 20% Adaption (z.B. Anpassung an neues Betriebssystem)
- 50% Perfektion (z.B. Umstellung von alphanumerischer auf graphische Schnittstelle)

Installation und Wartung (2)

Andere Quelle (Swanson 1980, Studie an 400 Softwareprojekten):

- 42% Änderungen der Anforderungsdefinition
- 17% Änderung der Dateiformate
- 12% Notfalldebugging
- 9% normales Debugging
- 6% Änderungen der Hardware
- 5% Verbesserungen der Dokumentation
- 4% Verbesserungen der Effizienz

Installation und Wartung (3)

Wartung ist um so teurer, je mehr die frühen Phasen davon betroffen sind:

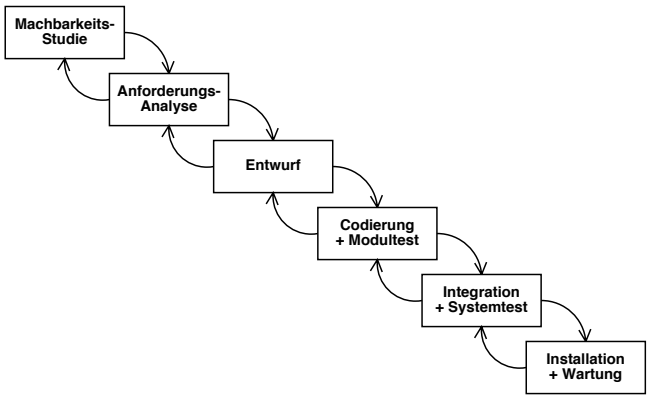
Nachträgliche Änderungen der Anforderungen *verzehnfachen* die Kosten pro Phase!

Ergebnis der Phase: keins, da kein Übergang in nächste Phase

Wasserfallmodell mit Rückkopplung

In der Praxis kann man nie strenge Trennung der Phasen erreichen; dies ist nur der Idealfall

Deshalb: Möglichkeit, in frühere Phasen zurückzukehren (etwa wenn Fehler oder Inkonsistenzen entdeckt wurden)



Schwächen des Wasserfallmodells

- Am Projektanfang sind nur *ungenaue* Schätzungen der Kosten und Ressourcen möglich.
- Ein Pflichtenheft, egal wie sorgfältig erstellt, kann nie den Umgang mit dem fertigen System ersetzen. Dies ist für viele Kunden ein Problem.
- Es gibt Kunden, mit denen man keine präzisen Pflichtenhefte erstellen kann,
  weil sie nicht wissen, was sie wollen.

  Beispiel: In Berlin sollte ein Programm zur Wohnungsvermittlung erstellt werden. Es sollte die Wohnungen nach sozialen Kriterien vergeben, aber auch verfügbare Wohnungen sofort vermitteln.

Schwächen des Wasserfallmodells (2)

- Oft werden die endgültigen Anforderungen erst nach einer gewissen *Experimentierphase* deutlich.
- Antizipation des Wandels (von Anforderungen) wird überhaupt nicht unterstützt; stattdessen werden die Anforderungen frühzeitig eingefroren.
- Am Ende jeder Phase muß ein *Dokument* abgeliefert werden. Dies kann zu einer Papierflut und überbordender Bürokratie führen, ohne daß die Qualität profitiert.

Evolutionäres Modell

Alternative *Evolutionäres Vorgehen*:
Ein Produkt wird als Folge von *Approximationen* realisiert.

Jede Approximation entsteht durch Änderungen/Erweiterungen aus der vorangegangenen Version.

Einige oder sogar alle Phasen des Lebenszyklus können evolutionär realisiert werden.

Prototyping

Software-Prototypen sind *Approximationen* an das endgültige System mit

- reduziertem Funktionsumfang
- reduzierter Benutzerschnittstelle
- reduzierter Leistung

Wir betrachten folgende *Spezialfälle*:

- Rapid Prototyping
- Horizontaler Prototyp
- Vertikaler Prototyp

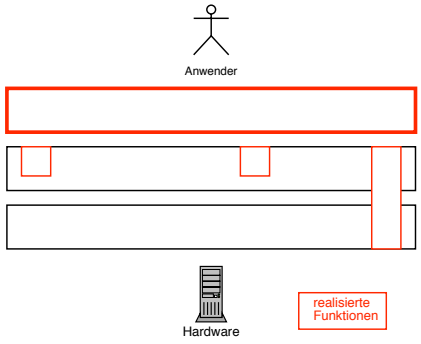
Rapid Prototyping

Verwendung von Generatoren, ausführbaren Spezifikationssprachen oder Skriptsprachen

- Vorteil: sehr schnelle Realisierung, frühzeitige Validierung der funktionalen Spezifikation
- Nachteil: u.U. ineffizient, schlechte Benutzerschnittstelle, bei Skriptsprachen: schlechte Systemarchitektur

Horizontaler Prototyp

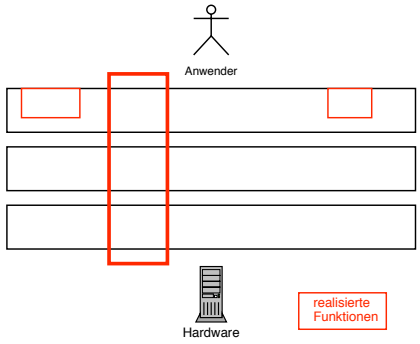
Nur eine Systemschicht, z.B. hohle Benutzerschnittstelle ohne echte Funktionalität



- Vorteil: frühzeitiges Einbinden der zukünftigen Benutzer; Validierung der Spezifikation
- Nachteil: nur für Demos benutzbar

Vertikaler Prototyp

Abgemagerter Funktionsumfang wird durch alle Ebenen implementiert



- Vorteil: frühzeitige Validierung technischer Eigenschaften; frühzeitige Auslieferung eines Kernsystems
- Nachteil: teuer

Probleme des Prototypings

Problem: Den Übergang vom Prototypen zum Endprodukt bewältigen

- Kunde sieht Prototyp und wundert sich, daß er weggeworfen werden soll
- Prototypen können nur begrenzt zu einem vollwertigen Programm ausgebaut werden

Prototyping kombiniert

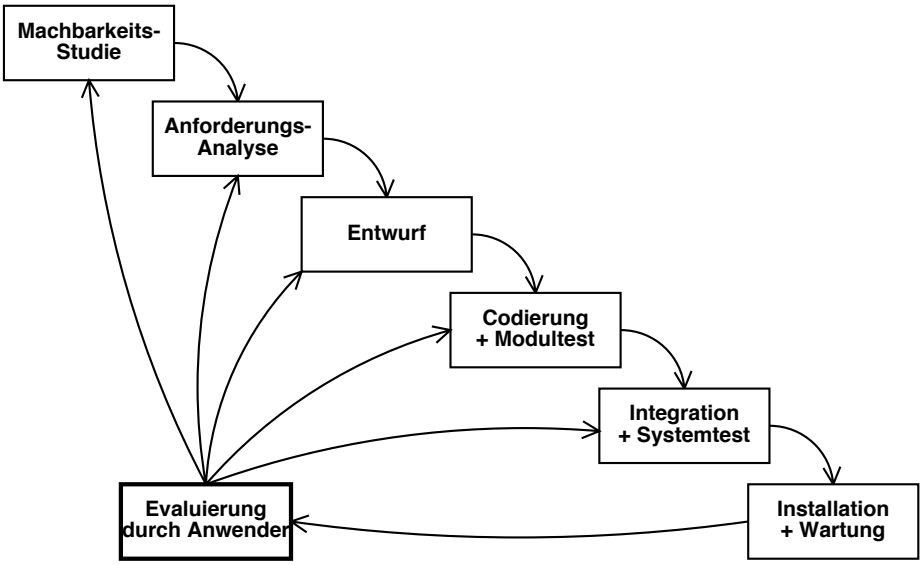
Heutiger Trend:

- Benutzerschnittstellen werden mit Skriptsprachen (z.B. *Tcl/Tk*, *Python* oder *Visual Basic*) oder Webseiten prototypisch realisiert; die Benutzerschnittstelle wird so Teil des Endprodukts
- Die einzelnen Funktionalitäten werden zunächst ebenfalls mit Skriptsprachen prototypisch realisiert und im Endprodukt durch vollwertige, herkömmlich realisierte Funktionen ausgetauscht.

Ratschläge für evolutionäre Entwicklung

- Das System entwickelt sich um frühzeitig entwickelte *Schlüsselkomponenten*, um die andere Systemteile herumgebaut werden (Beispiel: Programmierumgebung entwickelt sich um Editor herum)
- Stets muß ein *vorführbarer Prototyp* vorhanden sein, und sei er noch so vorläufig
- Man sollte eine gute *Entwicklungsinfrastruktur* aufbauen – Werkzeuge, High-Level-Sprachen und Generatoren
- Die Systemarchitektur sollte in besonderer Weise *für zukünftige Optionen offen* sein
- Da es meistens ähnliche Projekte anderswo gibt, ist es wichtig, *gut informiert* zu sein.

Allgemeines evolutionäres Modell



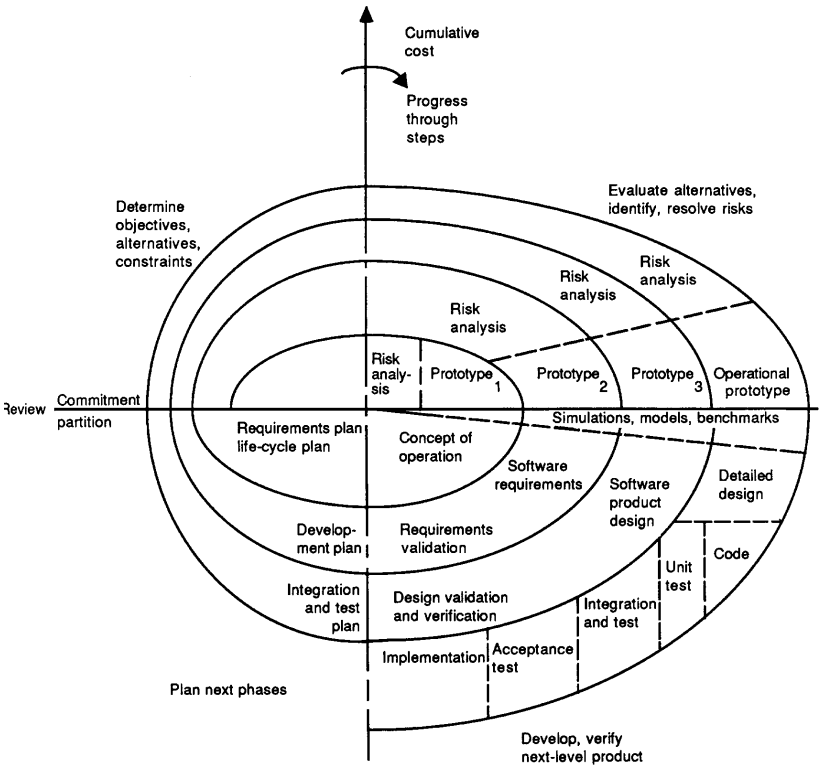
Spiralmodell

Das Spiralmodell ist ein *Metamodell*, das evolutionäre Aspekte und Risikobewertung umfaßt.

Jeder Umlauf der Spirale entspricht dem nächsten Prototyp;
Winkel $\simeq$ Zeit, Radius $\simeq$ Kosten

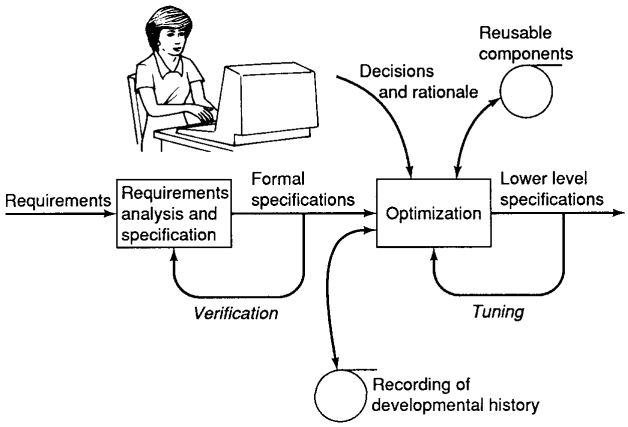
Jeder Umlauf zerfällt in 4 Hauptphasen (Quadranten):

1. Anforderungsdefinition
2. Bewertung von Alternativen und Risiken
3. Entwurf, Implementierung, Test
4. Auswertung und Planung des nächsten Umlaufs



Transformationsmodell

Im Transformationsmodell wird das fertige Programm (halb-) automatisch aus einer formalen Spezifikation erzeugt



Transformationsmodell (2)

Korrektheitserhaltende Transformationen + Maschinenunterstützung

1. Ausgangspunkt ist *nichtdeterministische funktionale Spezifikation*
2. Zusätzliche *Entwurfsentscheidungen* beseitigen Nichtdeterminismus
3. Transformation in *tailrekursive Form*
4. Transformation in *prozedurales Programm*

Vorteil: Korrektheit garantiert

Nachteil: anspruchsvoll, nur für kleine Beispiele brauchbar

Extreme Programming

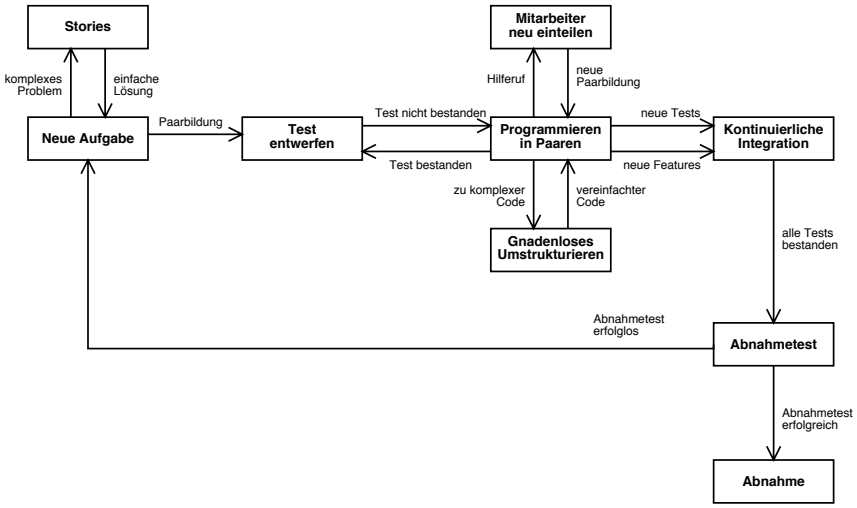
Extreme Programming („XP“) ist ein Prozess zum *schnellen Programmerstellen in unsicherem Umfeld.*

Antwort auf übermäßige *Prozessmodellierung* der 90er.

Eigenschaften:

- Arbeiten in kleinen Gruppen (max. 10 Personen)
- Kurzfristige Planung (bis zum nächsten Termin)
- Konzentration auf das, was gemacht werden muß
- Flexibilität ist Trumpf!

Das XP-Vorgehensmodell



Stories

Im Extreme Programming beschreibt der Auftraggeber die Anforderungen über *Stories*.

Jede Story beschreibt einen konkreten Vorfall, der für den Betrieb des Produktes notwendig ist.

Beispiel – eine „Parkhaus“-Story:

- Der Fahrer bleibt an der geschlossenen Schranke stehen und fordert einen Parkschein an.
- Hat er diesen entnommen, öffnet sich die Schranke
- Der Fahrer passiert die Schranke, die sich wieder schließt.

Weitere Situationen (volles Parkhaus, Stromausfall…) werden ebenfalls durch eigene Stories beschrieben.

Pair Programming

Pair Programming (Programmieren in Paaren) sorgt für ständiges Gegenlesen durch den Partner

- Häufiger, ständiger Rollenwechsel (Programme schreiben/lesen)
- Teams werden stets neu zusammengestellt
- Auch der Kunde kann so beteiligt sein

Automatisches Testen

Ständiges automatisches Testen sorgt dafür, daß Funktionalität erhalten bleibt

- Für jede neue Funktionalität gibt es einen Testfall
- Testfälle werden *vor* dem Programm geschrieben

Umstrukturierung

Jeder kann jederzeit die Software *umstrukturieren* (*refactoring*)

- Software muß entsprechend wandlungsfähig sein
- Testfälle müssen weiterhin erfüllt werden
- Code-Qualität muß auf hohem Niveau bleiben

XP kennt keinen Entwurf

Extreme Programming kennt keine eigenen Entwurfsphasen:

- Die Funktionalität des Systems wird in Stories zusammengefaßt
- Jede Story wird 1:1 in einen Testfall umgesetzt
- Man nehme den *einfachsten Entwurf, der die Testfälle besteht* – und implementiere ihn
- Die Implementierung ist abgeschlossen, wenn alle Testfälle bestanden sind
- Treten bei der Abnahme weitere Fragen auf, gibt es *neue Stories* – und neue zu erfüllende Testfälle

Der Kunde ist bei der gesamten Entwicklung dabei!

Einsatzbedingungen

Extreme Programming ist *geeignet…*

- ✔ wenn die Anforderungen *vage* sind
- ✔ wenn die Anforderungen *sich schnell ändern*
- ✔ wenn das Projekt klein bis mittelgroß ist (< 10–12 Programmierer)

Extreme Programming ist *ungeeignet…*

- ✘ wenn es auf *beweisbare* Programmeigenschaften ankommt
- ✘ wenn späte Änderungen zu teuer werden
- ✘ wenn häufiges Testen zu teuer ist
- ✘ wenn das Team zu groß oder nicht an einem Ort ist

XP – Vor- und Nachteile

- ✔ Testfälle vor dem Codieren schreiben
- ✔ Programmieren in Paaren

- Kein Entwurf
- Keine externe Dokumentation

- ✘ Erschwerte Wiederverwendung
- ✘ Nur für kleine, hochqualifizierte Teams geeignet
- ✘ Erst wenig Erfahrung vorhanden

Eine Kette von Entscheidungen

Alle Prozessmodelle haben eins gemeinsam: den Übergang von einer *vagen Idee* zu einem *konkreten Produkt*

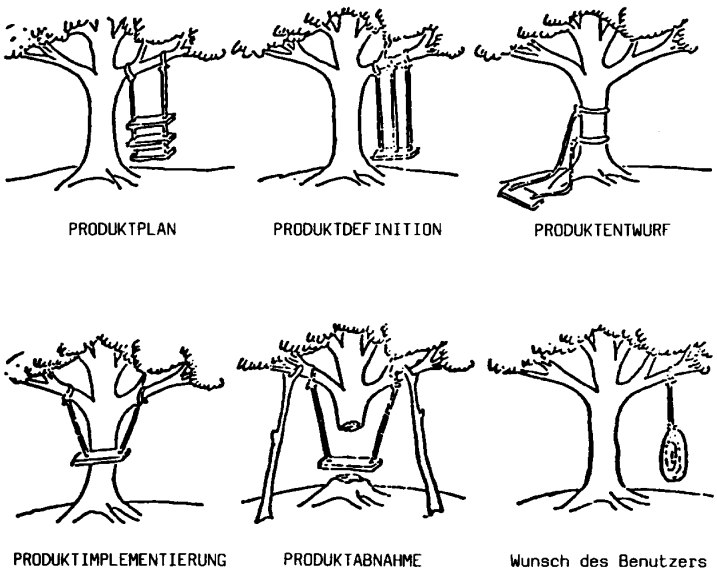
Hierfür müssen Sie Unmengen von Entscheidungen treffen!

Ziel ist, die Entscheidungen so zu treffen,

- dass *Ihre frühere Arbeit* nicht unnütz wird
- dass *spätere Entscheidungen* möglichst offen bleiben können

Sie kommen nicht umhin, alle Entscheidungen zu treffen, die Sie treffen müssen. Dabei müssen Sie so stetig und risikolos wie möglich ins Ziel gelangen!

Eine alternative Sicht



Quelle unbekannt