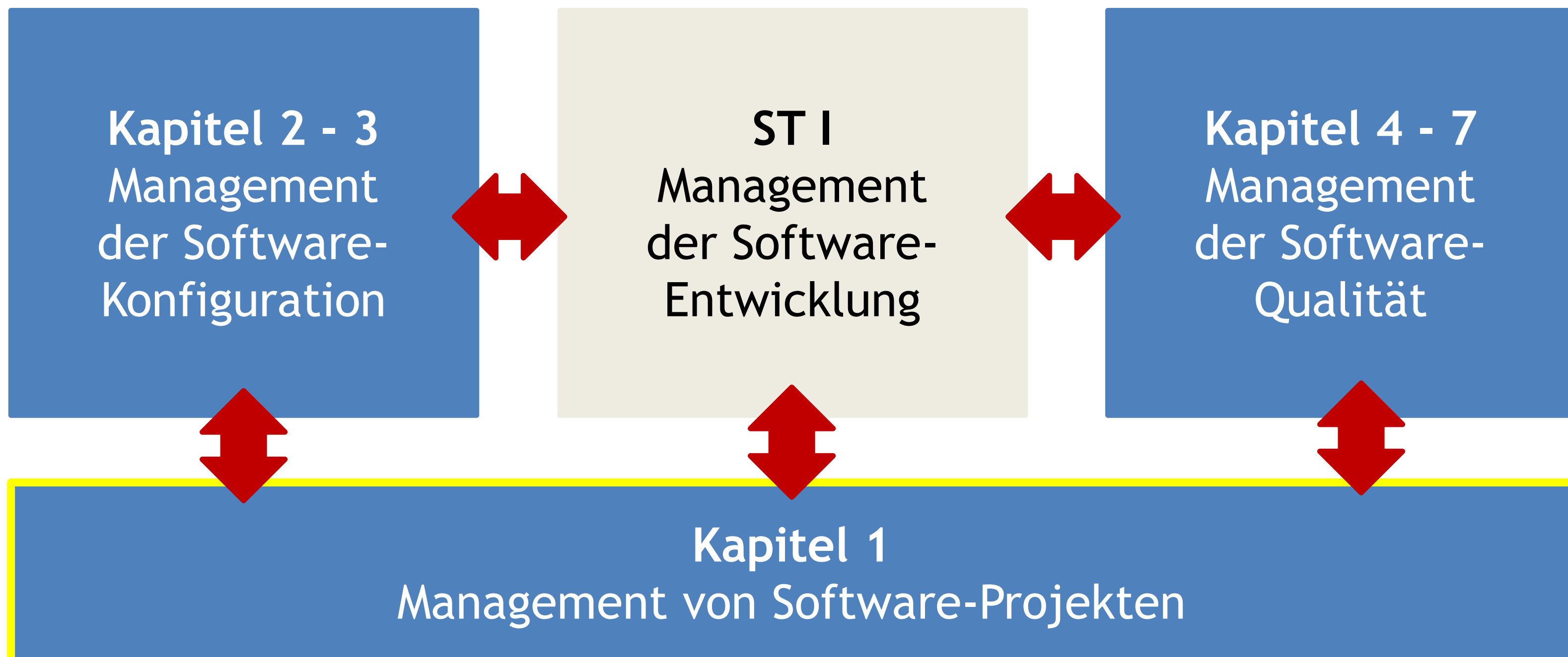
UNIVERSITÄT SIEGEN

Vorlesung Softwaretechnik II

Management von Software-Projekten: Grundlagen und Prozesse

UNIVERSITÄT SIEGEN

Aufbau der Vorlesung

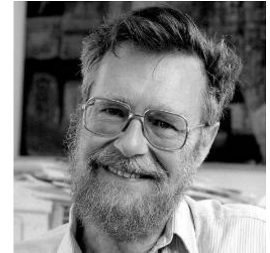


Inhalt

- Motivation und Grundbegriffe
- Iterative Softwareentwicklung, Forward-, Reverse- und Reengineering
- Vorgehensmodelle, Entwicklungsprozesse und Projektorganisation

Motivation und Grundbegriffe

Zitat



E.W. Dijkstra
(1930 - 2002)

„Als es noch keine Rechner gab, war auch das Programmieren noch kein Problem, als es dann ein paar leistungsschwache Rechner gab, war das Programmieren ein kleines Problem und nun, wo wir gigantische Rechner haben, ist auch das Programmieren zu einem gigantischen Problem geworden. In diesem Sinne hat die elektronische Industrie kein einziges Problem gelöst, sondern nur neue geschaffen. Sie hat das Problem geschaffen, ihre Produkte zu nutzen.“

Wozu Softwaretechnik?

Für die Entwicklung **großer Software-Produkte** mit Milliarden Zeilen Code (in Autos, Flugzeugen, Medizintechnikgeräten, …) braucht man andere Vorgehensweisen als für das Lösen kleiner Übungsaufgaben.

Einordnung der Softwaretechnik

- Das Gebiet der Softwaretechnik wird der praktischen Informatik zugeordnet, hat aber auch Wurzeln in der theoretischen Informatik.
- Kann als „querschneidende“ Disziplin angesehen werden.
- Informatikübergreifende Aspekte spielen eine wesentliche Rolle (wie Projektplanung, Organisation, Psychologie,…).

Geburtsstunde der Softwaretechnik

“Working Conference on Software Engineering”

7. - 10. Oktober 1968 in Garmisch

Eine kurze Geschichte der Softwaretechnik

- Der Auslöser für die Entstehung der wissenschaftlichen Disziplin **„Software-technik“** im Jahre 1968 war die sogenannte **„Software-Krise“**.
- Der Begriff “**Software Engineering**” wurde erstmals von F. L. Bauer im Rahmen einer „Study Group on Computer Science“ der NATO geprägt.

Die Software-Krise von 1968 bis …?

Apollo Guidance Computer (AGC)

Margaret Hamilton mit einem Ausdruck der AGC-Software

Ausgelöst vor allem durch die Raumfahrt, wurden Programmsysteme in den späten 60er Jahre sprunghaft komplexer, aber es gab

- Keine geeigneten (Programmier-)Sprachen.
- Keine geeigneten Methoden/Vorgehensweisen.
- Keine geeigneten Werkzeuge.

Und über 50 Jahre später?

Die *Standish Group* veröffentlicht in regelmäßigen Abständen den „Chaos Report“ (zuletzt 2015):

- 19% aller betrachteten IT-Projekte sind gescheitert (früher: 25%)
- 52% aller betrachteten IT-Projekte sind dabei, zu scheitern (früher: 50%)
- 29% aller betrachteten IT-Projekte sind erfolgreich (früher: 25%)

[http://www.standishgroup.com]

UNIVERSITÄT SIEGEN

Warum Scheitern IT-Projekte?

- Hauptgründe für das Scheitern von IT-Projekten: Irreparable Überschreitungen von Finanzbudget und Zeitrahmen.
- Hauptursachen für das Scheitern von IT-Projekten:
  1. Unklare **Anforderungen** und **Abhängigkeiten**
  2. Grundlegende Probleme beim **Qualitäts-** und **Änderungsmanagement**

2. ist Schwerpunkt dieser Vorlesung!

Worum ist Software-Erstellung so schwierig?

- **Kommunikationsprobleme** mit dem Anwender und wenig Anwendungswissen bei Entwicklern führen zu in Konflikt stehende Anforderungen.
- Software ist **immateriell** und wird nicht durch physikalische Gesetze begrenzt, deshalb **Modellbildung** für relevanten Weltausschnitt schwierig.
- Software lässt sich **leicht(er) modifizieren** als Hardware; Anforderungen ändern sich während Entwicklungszeit.
- Software unterliegt keinem **Verschleiß**, altert aber trotzdem, das führt zu **Wartungsproblemen** („Jahr 2000"-Problem, …).
- Software muss auf verschiedenen **Plattformen** laufen, das führt zu **Portabilitätsproblemen.**
- Software existiert in vielen **Varianten**, das führt zu Explosion der **Variantenvielfalt.**
- Software greift in bestehende **Arbeitsabläufe** ein, das führt zu Akzeptanzproblemen.
- Viele Software-Produkte sind **„einzigartig"** (keine Massenware).

Entwicklung des Umfangs von Software

Die folgenden Zahlen sind Schätzungen des Code-Umfangs (in Lines of Code - LOC) bekannter Softwareprodukte.

- Unix Version 1.0 (Stand 1971): 100.000 LOC
- Windows 3.1 (Stand 1992): 2 Millionen LOC
- Windows 7 (Stand 2009): 40 Millionen LOC
- Microsoft Office 2013: 45 Millionen LOC
- Windows 8: 50 bis 60 Millionen LOC
- Debian 5 Code Base: 65 Millionen LOC
- MAC OS X „Tiger“: 85 Millionen LOC
- Web-Site healthcare.com („Obama-Care“-Programm): 500 Millionen LOC

Ist LOC ein gutes Maß?

[http://www.informationisbeautiful.net/visualizations/million-lines-of-code/]

Definition: Softwaretechnik

*„Die **Softwaretechnik** befasst sich mit der Entwicklung, Wartung und dem Einsatz **qualitativ hochwertiger Software** unter Verwendung von wissenschaftlichen Methoden, wirtschaftlichen Prinzipien, geplanten Vorgehensmodellen, Werkzeugen und quantifizierbaren Zielen.“*

[Kahlbrandt, 1998]

Ziele der Softwaretechnik

Ziel der Softwaretechnik ist demnach die effiziente Entwicklung von Software mit **messbarer Qualität**, die

- die gewünschte Funktionalität anbietet,
- eine benutzerfreundliche Oberfläche besitzt,
- korrekt bzw. zuverlässig arbeitet,
- ...

Software-Qualität

Ein wohldefinierter Qualitätsbegriff ist somit wesentlich für jede Disziplin, die sich ihre Zielsetzungen anhand ingenieursmäßiger Prinzipien definiert.

Merke: Die verschiedenen Qualitätsmerkmale eines Software-Produktes müssen nicht zwangsläufig „hochwertig“ sein, sondern ***nachvollziehbar einen zuvor vereinbarten Grad erreichen****.*

Definition: Qualität

„Qualität ist der Grad, in dem ein System, eine Komponente oder ein Prozess die Kundenerwartungen und Kundenbedürfnisse erfüllt.“

[IEEE 610]

Definition: Software-Qualität

„Softwarequalität ist die Gesamtheit der Funktionalitäten und Merkmale eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.“

[ISO 9126]

Beispiel: Software-Qualität

```
program SORT;
var a, b: file of integer;
      Feld: array [ 1 .. 10 ] of integer;
      i, j, k, l : integer
begin
      open ( a, '/usr/schuerr/Zahlen/Datei'); i := 1;
      while not eof ( a ) do
      begin
           read ( a, Feld [ i ] ); i := i + 1
      end;
      l := i - 1;
      open ( b, '/usr/schuerr/Zahlen/Datei');
      for i := 2 to l do
           begin
                for j := l downto i do
                     if Feld [ j - 1 ] > Feld [ j ] then
                     begin
                          k := Feld [j - 1]; Feld [j - 1] := Feld[ j ]; Feld [ j ] := k
                     end;
                     write ( b , Feld [ i - 1 ] )
           end
end SORT;
```

Aufgabe: Sie sollen die Qualität dieses einfachen Sortier-Programms, verfasst in einer Phantasie-Programmiersprache, bewerten…

Beispiel: Software-Qualität

```
program SORT;
var a, b: file of integer;
      Feld: array [ 1 .. 10 ] of integer;
      i, j, k, l : integer
begin
      open ( a, '/usr/schuerr/Zahlen/Datei'); i := 1;
      while not eof ( a ) do
      begin
            read ( a, Feld [ i ] ); i := i + 1
      end;
      l := i - 1;
      open ( b, '/usr/schuerr/Zahlen/Datei');
      for i := 2 to l do
            begin
                  for j := l downto i do
                        if Feld [ j - 1 ] > Feld [ j ] then
                        begin
                              k := Feld [j - 1]; Feld [j - 1] := Feld[ j ]; Feld [ j ] := k
                        end;
                        write ( b , Feld [ i - 1 ] )
            end
end SORT;
```

Nicht leicht verständlich, da Kommentare fehlen, Variablenbezeichner nichtssagend, …

Nicht korrekt (robust), da Programm für Eingabedateien mit mehr als 10 Elementen abstürzt.

Nicht effizient, da verwendeter Sortieralgorithmus quadratischen Aufwand hat.

Nicht portierbar, da Dateinamenskonventionen von Unix im Quelltext auftauchen und dieselbe Datei mehrfach geöffnet wird (geht nicht immer)

Nicht wiederverwendbar, da Dateinamen "fest verdrahtet" sind, ebenso maximale Dateilänge und zu sortierende Elemente.

Qualitätsmerkmale von Software

- Funktionalität (Functionality)
- Zuverlässigkeit (Reliability)
- Benutzbarkeit (Usability)
- Effizienz (Efficiency)
- Änderbarkeit (Maintainability)
- Übertragbarkeit (Portability)
- …

[ISO 9126]

1. Schritt: Qualitätszielbestimmung

Auftraggeber und Auftragnehmer legen vor Beginn der Software-Entwicklung ein gemeinsames Qualitätsziel für das Software-System mit nachprüfbarem Kriterienkatalog fest (als Bestandteil des abgeschlossenen Vertrags zur Software-Entwicklung).

Prinzipien der Qualitätssicherung

- **Quantitative Qualitätssicherung**: Einsatz automatisch ermittelbarer Metriken zur Qualitätsbestimmung (objektivierbare, ingenieursmäßige Vorgehensweise).
- **Konstruktive Qualitätssicherung**: Verwendung geeigneter Methoden, Sprachen und Werkzeuge (Versuch der aktiven Vermeidung von Qualitätsproblemen).

Prinzipien der Qualitätssicherung

- **Integrierte, frühzeitige analytische Qualitätssicherung**: Systematische Prüfung aller erzeugten Dokumente (Aufdeckung von Qualitätsproblemen).
- **Unabhängige Qualitätssicherung**: Entwicklungsprodukte werden durch eigenständige Qualitätssicherungsabteilung überprüft und abgenommen (verhindert z.B. „schlampige“ Tests zugunsten der Einhaltung des Zeitplans).

UNIVERSITÄT SIEGEN

Organisatorische Maßnahmen zur Qualitätssicherung

- Richtlinien (z.B. Gliederungsschema für Pflichtenheft, Programmierrichtlinien).
- Standards (z.B. für verwendete Sprachen, Dokumentformate, Management).
- Checklisten (z.B. „bei Ende einer Phase müssen folgende Dokumente vorliegen" oder "Softwareprodukt erfüllt alle Punkte des Lastenheftes").

Technische Maßnahmen zur Qualitätssicherung

- Verwendung sicherer Sprachen (z.B. Java) und zertifizierter Werkzeuge (z.B. verifizierte / verifizierende Compiler).
- Werkzeuge können darüber hinaus den Einsatz von Prinzipen und organisatorische Maßnahmen zur Qualitätssicherung erleichtern bzw. sogar erzwingen!

Iterative Softwareentwicklung

Forward-, Reverse- und Reengineering

Zitat

"Everything should be built top-down, except the first time."

[Perlisisms - "Epigrams in Programming" by Alan J. Perlis]

Vorgehensmodelle

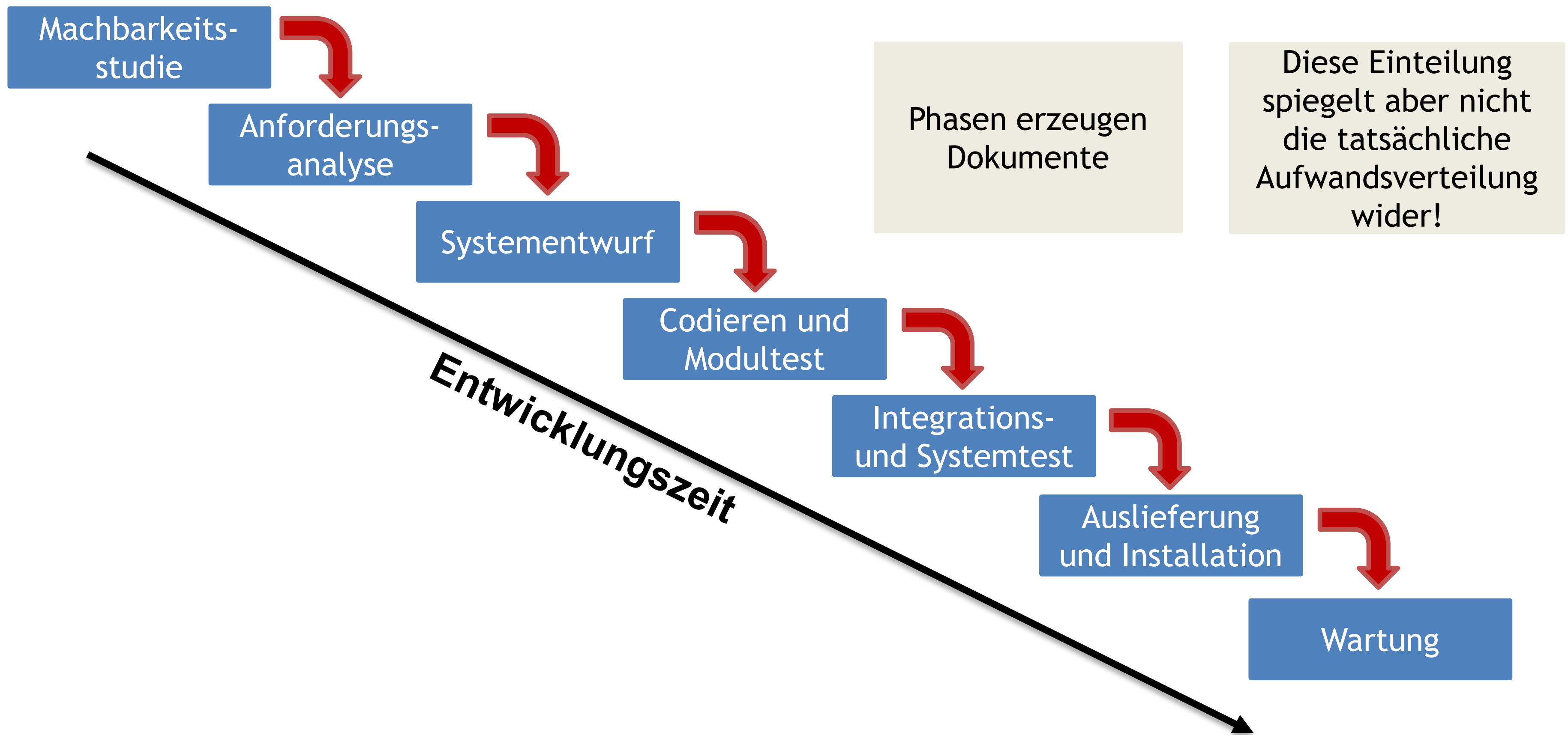
- Vorgehensmodelle sind Voraussetzung für den sinnvollen Einsatz von Methoden und Werkzeugen zur Software-Entwicklung und Qualitätssicherung.
- Aufteilung des Gesamtprozesses der Software-Erstellung und -Wartung in einzelne **Schritte**.
- Regelung von Verantwortlichkeiten beteiligter Personen im Software-Entwicklungsprozess von in Form von **Rollen**.

Vorgehensmodelle - gestern und heute

- Das berühmt-berüchtigte „**Wasserfallmodell**“ war lange Zeit das Standardvorgehensmodell zur idealisierten Erstentwicklung von Software („top-down Entwicklung von der grünen Wiese auf Grundlage perfekter Anforderungen“).
- Die heutige (Weiter-)Entwicklung moderner Software basiert hingegen zunehmend auf **iterativen und agilen Vorgehensmodellen** (z.B. V-Modell bis hin zu Scrum).

UNIVERSITÄT SIEGEN

Die Phasen des Wasserfallmodells



UNIVERSITÄT SIEGEN

Zitat

„Software-Systeme zu erstellen, die nicht geändert werden müssen, ist unmöglich. Wurde Software erst einmal in Betrieb genommen, entstehen neue Anforderungen und vorhandene Anforderungen ändern sich ... “

[Liggesmeyer, 2002]

Wartung (Maintenance)

- Nach der ersten Auslieferung der Software an die Kunden beginnt die „abschließende Phase“ der Software-Wartung.
- Aber: Diese Phase macht in der Praxis ca. 60-80% der gesamten Software-Kosten aus!

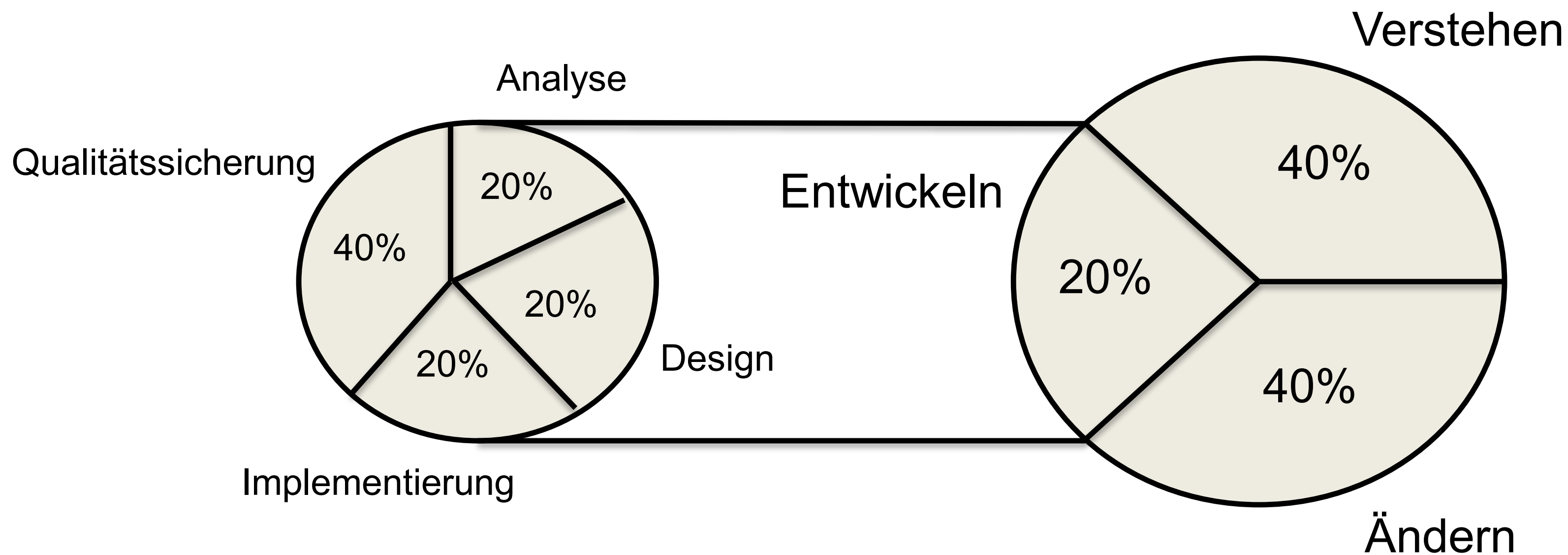
Aufgaben der Software-Wartung

- ca. 20% Fehler beheben (corrective maintenance).
- ca. 20% Anpassungen durchführen (adaptive maintenance).
- ca. 50% Verbesserungen vornehmen (perfective maintenance).

Ergebnis der Software-Wartung

- Software-Problemberichte (z.B. bug reports).
- Software-Änderungsvorschläge (z.B. feature requests).
- Neue Software-Versionen.

Aufwandsverteilung insgesamt

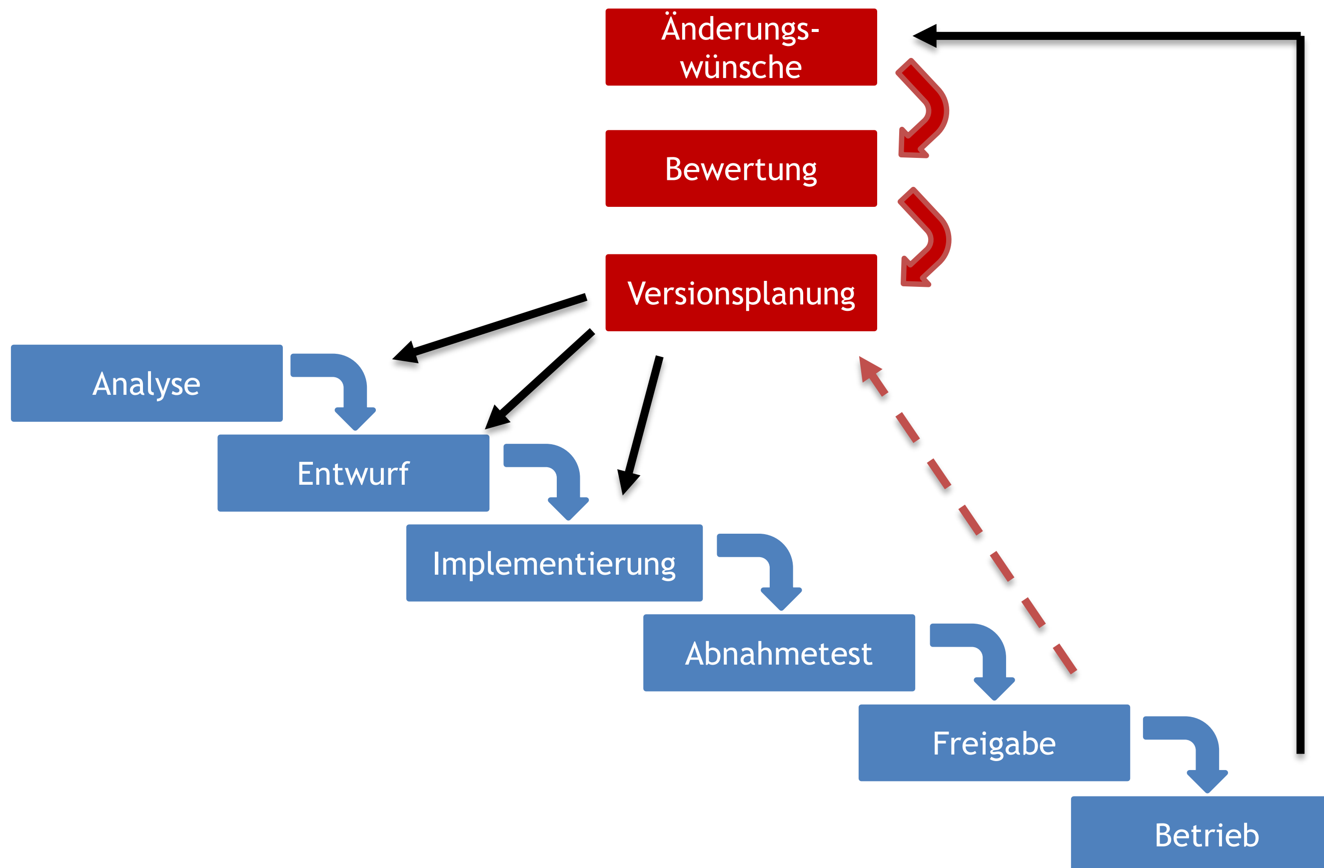


Häufige Probleme in der Wartungsphase

- Unerfahrenes Personal (nicht Entwicklungspersonal).
- Fehlerbehebung führt zu neuen Fehler.
- Stetige Verschlechterung der Programmstruktur.
- Zusammenhang zwischen Programm und (ursprünglicher) Dokumentation geht verloren.
- Zur Entwicklung eingesetzte Werkzeuge sterben aus.
- Benötigte Hardware steht nicht mehr zur Verfügung
- Ressourcenkonflikte zwischen Fehlerbehebung und Anpassung/Erweiterung.
- Völlig neue Ansprüche an Funktionalität und Benutzeroberfläche…

UNIVERSITÄT SIEGEN

(Naives) Prozessmodell für die Wartungsphase



Forward Engineering

- Beim **Forward Engineering** ist das fertige Software-System das Ergebnis des Entwicklungsprozesses.
- Ausgehend von der Anforderungsanalyse (Machbarkeitsstudie) wird ein **komplett neues** Software-System entwickelt.

Forward Engineering

Voraussetzungen/Annahmen:

- Vollständige Anforderungs- und Design-Dokumente für den Code existieren.
- Alle Dokumente sind vorhanden und untereinander konsistent.
- Die Entwickler des Codes bleiben verfügbar (z.B. zur Planung von Evolutionen).

Forward Engineering: Wunsch trifft auf Wirklichkeit

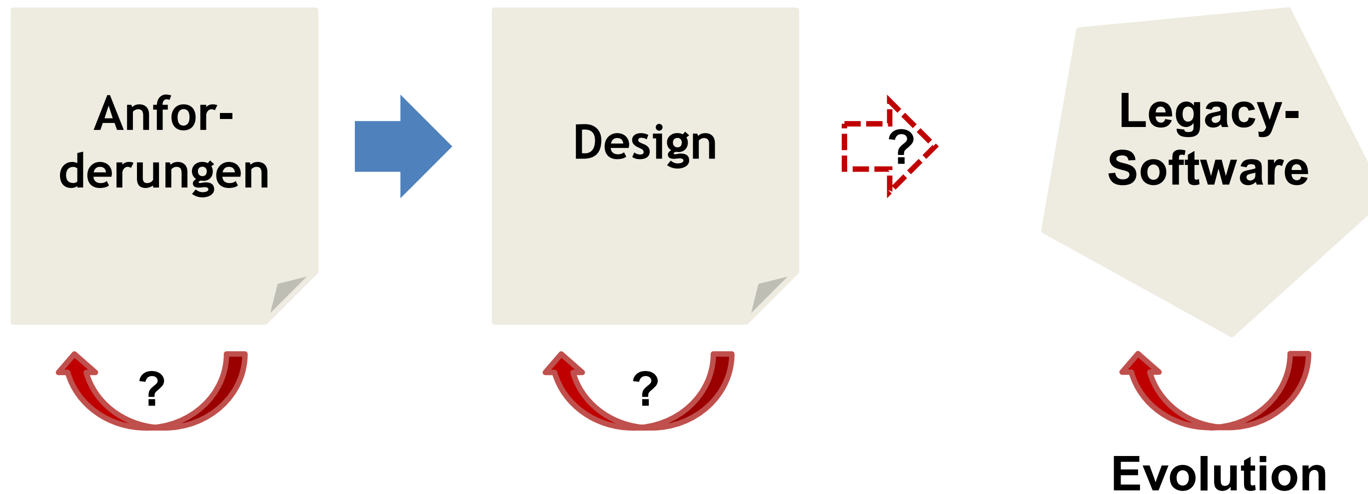
Software-Evolution - Wünsche:

- Wartungsmaßnahmen ändern Software kontrolliert ohne das Design zu zerstören.
- Konsistenz aller Dokumente bleibt erhalten.

Software-Evolution - Wirklichkeit:

- Die ursprüngliche Systemstruktur wird ignoriert.
- Dokumentation wird unvollständig oder unbrauchbar.
- Mitarbeiter (und ihr Wissen) verlassen Projekt.

Unkontrollierte Evolution

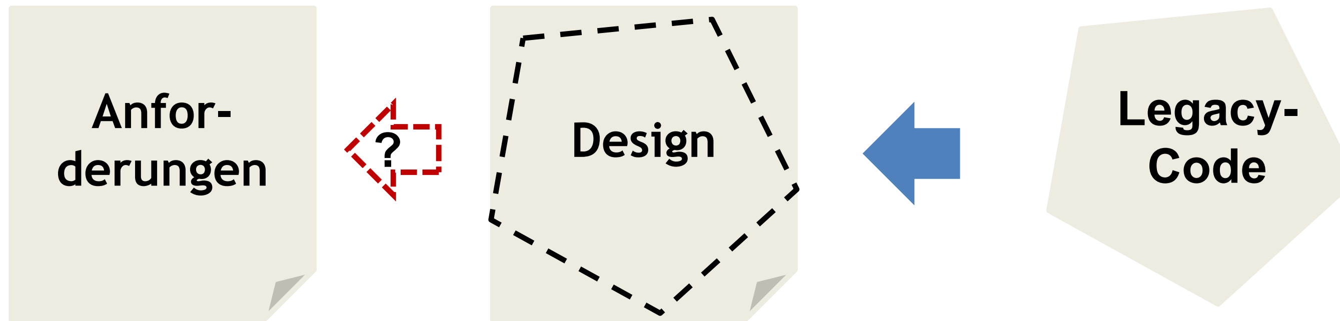


„Legacy“ = „wertvolles Erbe“

Ergebnis unkontrollierter Evolution

- Der Code ändert sich mit Sicherheit, seine Struktur bleibt meist erhalten (auch wenn sie neuen Anforderungen eigentlich nicht gewachsen ist).
- Das Design wird nach einer gewissen Zeit nicht mehr aktualisiert.
- Die Anforderungsdokumente werden erst recht nicht mehr gepflegt.

Reverse Engineering



Beim **Reverse Engineering** ist das vorhandene Software-System der Ausgangspunkt der Analyse.

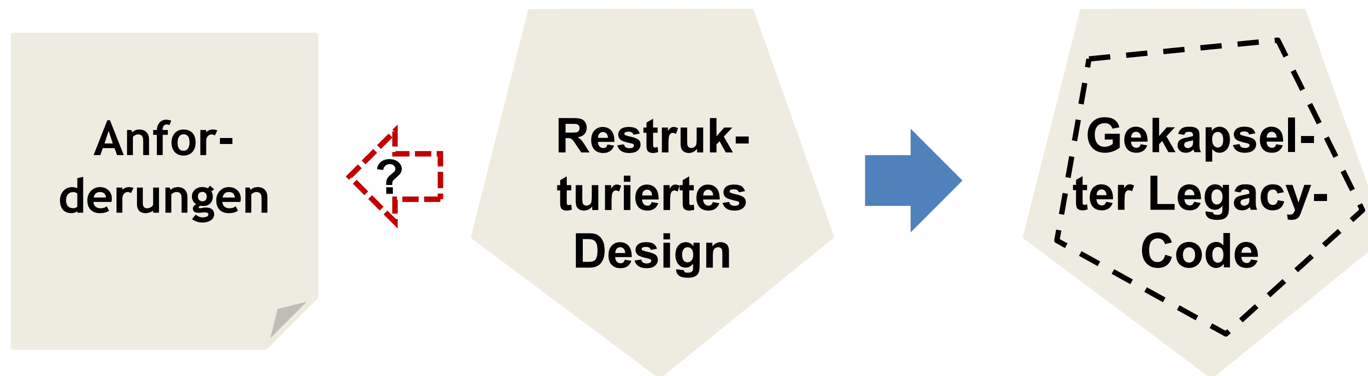
Reverse Engineering

- Das Design einer existierenden Implementierung wird meist „nur“ **rekonstruiert** und dokumentiert.
- Das betrachtete System wird (noch) **nicht modifiziert**.

Reverse Engineering: Wunsch trifft Wirklichkeit

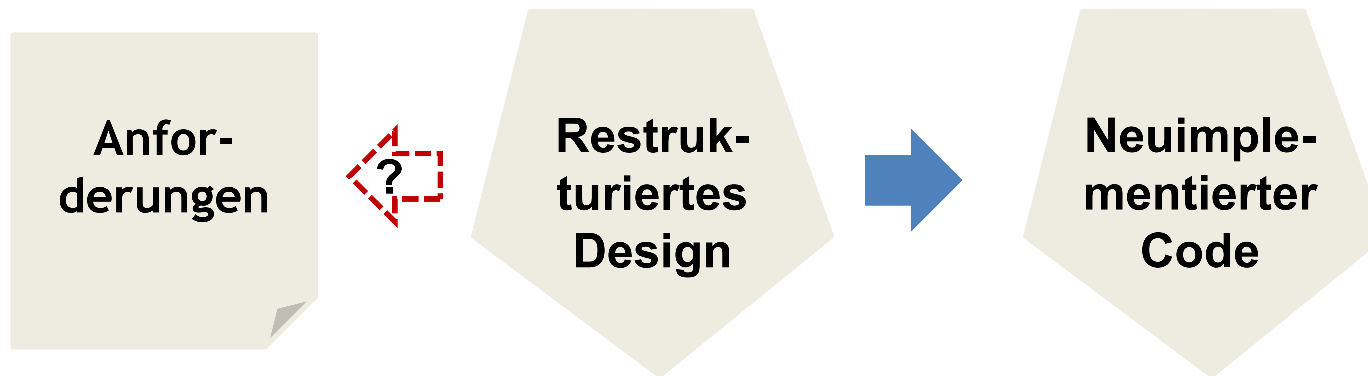
- Lässt sich das Software-System überhaupt noch restrukturieren/sanieren?
- Oder kann/muss man sein Innenleben „verkapseln“, um es (zunächst) weiterzuverwenden?
- Ansonsten: Reengineering.

Reengineering durch Kapselung



- Das **Reengineering** befasst sich mit der Sanierung eines vorhandenen Software-Systems bzw. seiner Neuimplementierung.
- Ausgangspunkt sind die Ergebnisse des Reverse Engineerings.

Reengineering durch Neuimplementierung



- Das **Reengineering** befasst sich mit der Sanierung eines vorhandenen Software-Systems bzw. seiner Neuimplementierung.
- Ausgangspunkt sind die Ergebnisse des Reverse Engineerings.

Iterative Software-Entwicklung

Roundtrip-Engineering:

- Initiale Iterationen durch Forward Engineering.
- Spätere Iterationen durch Reverse- /Reengineering.

Vorgehensmodelle, Entwicklungsprozesse und Projektorganisation

Management der Software-Entwicklung

Die folgenden Überlegungen sind nicht nur für **Software**-Entwicklungsprozesse geeignet, sondern werden für die Steuerung beliebiger komplexer technischer Entwicklungsprozesse eingesetzt:

- Projektmanagement.
- Bessere/modernere Prozessmodelle.
- Verbesserung/Qualität von Softwareprozess-Modellen.

Zielsetzung des Projektmanagements

Merke: Hauptziel des Projektmanagements ist die **Erhöhung der Produktivität!**

Definition: Produktivität

Produktivität = Produktwert / Aufwand
= Leistung / Aufwand

Produktivität in der Software-Entwicklung

Produktivität = Größe der Software / geleistete Mitarbeitertage

Diskussion: Produktivität

- Sinnvolles Maß für die „Größe“ von Software?
- Berücksichtigung der Produktqualität?
- Aufwand = Mitarbeitertage?
- Nutzen (Return Of Investment) = Größe der Software?

Aufgaben des Projektmanagements

Planungsaktivitäten:

- Ziele definieren.
- Vorgehensweisen auswählen.
- Termine festlegen.
- Budgets vorbereiten, …

(Für Softwaretechniker: Vorgehensmodelle, Kostenschätzung, Projektpläne aufstellen)

Aufgaben des Projektmanagements

Organisationsaktivitäten:

- Strukturieren von Aufgaben.
- Festlegung organisatorischer Strukturen.
- Definition von Qualifikationsprofilen für Positionen, …

(Für Softwaretechniker: Rollenmodelle, Team-Modelle, Projektpläne aufstellen)

Aufgaben des Projektmanagements

Personalaktivitäten:

- Positionen besetzen.
- Mitarbeiter beurteilen.
- Mitarbeiter weiterbilden, …

(Für Softwaretechniker: nicht Thema dieser Vorlesung)

Aufgaben des Projektmanagements

Leitungsaktivitäten:

- Mitarbeiter führen.
- Mitarbeiter motivieren.
- Mitarbeiter koordinieren, …

(Für Softwaretechniker: nicht Thema dieser Vorlesung)

Aufgaben des Projektmanagements

Kontrollaktivitäten:

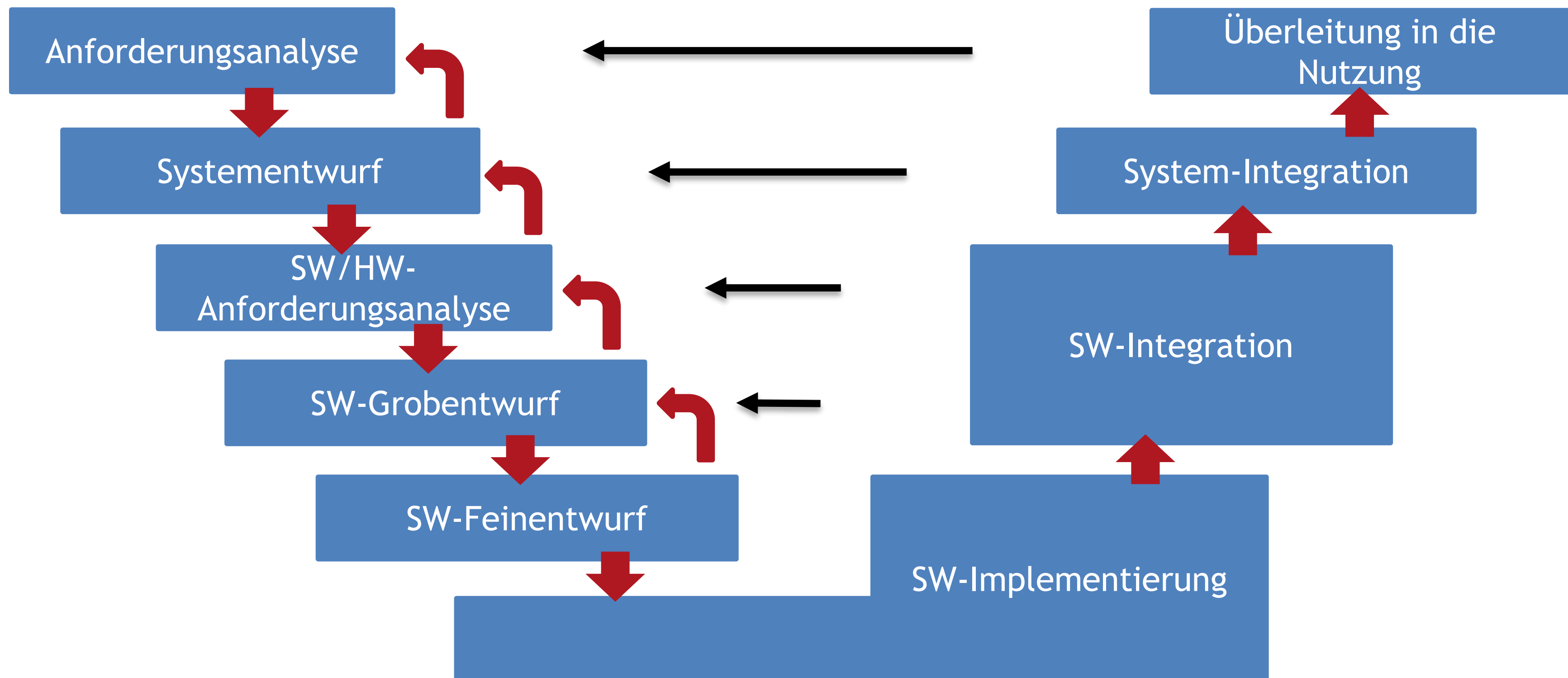
- Prozess- und Produktstandards entwickeln.
- Berichts- und Kontrollwesen etablieren.
- Prozesse und Produkte vermessen.
- Korrekturen vornehmen, …

(Für Softwaretechniker: Qualitätsmanagement, insbesondere für Software-Entwicklungsprozesse)

Moderne Prozess-/Vorgehensmodelle (Auswahl)

- V-Modell
- Rapid Prototyping
- Rational Unified Process (RUP)
- Extreme Programming (XP)
- Agile Prozessmodelle (SCRUM)

V-Modell (Standard in Bundesbehörden)



Diskussion: V-Modell

- In verschiedenstes Ausprägungen weit verbreitet.
- Erlaubt iterativeres Vorgehen als das Wasserfallmodell.
- Trennung von Entwicklung und Qualitätssicherung.
- Aber: Der Weg zur ersten lauffähigen Software-Version ist immer noch sehr lang und „schwergewichtig“.

Rapid Prototyping (Throw-Away-Prototyping)

- Mit GUI-Generatoren, ausführbaren Spezifikationssprachen, Skriptsprachen etc. wird ein **Prototyp** des Systems (bzw. seiner Benutzeroberfläche) mit der notwendigen „Rumpffunktionalität“ realisiert.
- Hauptsächlich zum Zwecke der **Demonstration** beim Management, (potentiellen) Kunden,…
- Anschließend zumeist weggeschmissen und/oder neu implementiert (bzw. verworfen).

Diskussion: Rapid Prototyping

Pluspunkte:

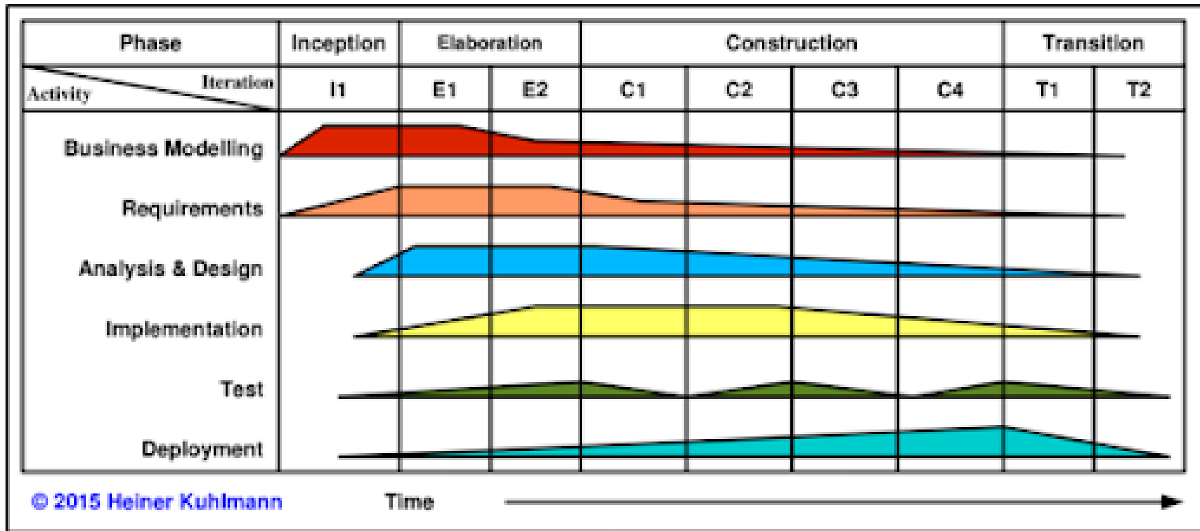
- Erlaubt schnelle Klärung der Funktionalität und Risikominimierung.
- Vermeiden von Missverständnissen zwischen Entwickler und Auftraggeber (frühzeitig User-Stories durchspielen).
- Früher Test der Benutzerschnittstelle.

Diskussion: Rapid Prototyping

Minuspunkte:

- Nicht wirklich ein Vorgehensmodell, sondern eher ein „Add-on“ für andere Modelle.
- Gefahr der Weiterverwendung des („hübschen“ aber häufig unausgegorenen) Prototypen (= ungeplantes evolutionäres Modell).
- Und/oder: es entsteht erheblicher Mehraufwand durch Wegwerfen und Neu-Implementierung.

Rational Unified Process (RUP)



- Von IBM/Rational forcierter Ansatz (seit 1998).
- Verwendet massiv UML-Modelle.

[http://www.rational.net/rupcenter]

Diskussion: RUP

Pluspunkte:

- Manager hat grobe Sicht (“Inception-Elaboration-Construction-Transition”).
- Entwickler hat zusätzlich feinere Sicht („Aktivitäten“).
- Wartung ist eine Abfolge zu entwickelnder Produktgenerationen.
- Es wird endgültig die Illusion aufgegeben, dass Analyse, Design, … zeitlich begrenzte, strikt aufeinander folgende Phasen sind.
- Es gibt den „Open Unified Process“ im Eclipse-Umfeld (http://epf.eclipse.org/wikis/openup/)

Diskussion: RUP

Minuspunkte:

- Sehr umfangreiches Vorgehensmodell, zugeschnitten auf modellbasierte SW-Entwicklung.
- Nicht richtig mit Behördenstandards (V-Modell) integriert.
- Qualitätssicherung ist kein eigener Aktivitätsbereich.

Schwergewichtige Vorgehensmodelle

Häufige Kritik an bisher betrachteten Vorgehensmodellen (V-Modell, RUP):

- Sehr starr und unflexibel („schwergewichtig“), insbesondere im Falle von (unvorhersehbaren) Änderungen.
- Produzieren Unmengen an Dokumenten, bevor auch nur eine Zeile Produktiv-Code entsteht.
- Bindet frühzeitig viele Arbeitskräfte, ohne ein sichtbares Ergebnis zu produzieren.

Leichtgewichtige Vorgehensmodelle

Leichtgewichtige Prozessmodelle (**light-weight processes**) sollen sinnlosen „bürokratischen" Overhead vermeiden:

- **Extreme Programming (XP)**: radikale Abkehr von bisherigen Vorgehensmodellen und Ersatz durch „best practices" aus der Programmierung.
- **Agile Prozessmodelle**: Bisherige Prozessmodelle auf das unbedingt notwendige zurückschneiden und situationsbedingt flexibel und schnell (agil) voranschreiten.

Manifest der Agilen Software-Entwicklung

- Individuen und Interaktionen wichtiger als Prozesse und Werkzeuge.
- Funktionierende Software wichtiger als umfassende Dokumentation.
- Zusammenarbeit mit Kunden wichtiger als Vertragsverhandlung .
- Reagieren auf Veränderung wichtiger als das Befolgen eines Plans.

[http://agilemanifesto.org/iso/de/]

Prinzipien der Agilen Entwicklung

- Requirements are user scenarios
- Let the team self-organize
- Tests first
- Put the Customer at the center
- Develop only code and tests
- Develop iteratively
- Work at sustainable pace
- Produce only the product requested
- Accept change
- Produce Product with minimal functionality

[B. Meyer, 2014]

Grundideen von Extreme Programming (XP)

Was bei XP **keinesfalls** gemacht wird:

- Unterteilung der Softwareentwicklung in Phasen, Arbeitsbereiche, …
- Eigenständige Analyse- oder Designaktivitäten vor der Codierung.
- Erstellung vom Code getrennter Dokumentationsdokumente, Modelle, …

Grundideen von Extreme Programming (XP)

Was bei XP **unbedingt** gemacht werden soll:

- Gut kommentierter Code, dient als vollwertige Dokumentation.
- Sorgfältige Testplanung: Testfälle werden immer zusammen mit (oder gar vor!) dem Code geschrieben (und können auch als Dokumentation angesehen werden).

Grundideen von Extreme Programming (XP)

Was XP **agil** machen soll:

- Der Auftraggeber soll permanent neue Anforderungen aufstellen und priorisieren („one action a day"…).
- Die Entwicklung erfolgt in sehr kurzen Iterationszyklen mit häufigen Releases.
- Nach jeder Änderung werden alle Tests wiederholt durchlaufen (Automatisierte Regressionstests).

Grundideen von Extreme Programming (XP)

Was XP sonst noch **propagiert**:

- Unbedingtes Einhalten der 40-Stunden-Woche, möglichst in Vollzeit.
- „Pair Programming“, …

Randbedingungen von XP

- Beschränkt auf relativ kleine Projekte mit maximal 5 bis 10 Programmierern.
- Sehr intensive Kommunikation zwischen Programmierern und Kunden notwendig; erschwert räumliche Verteilung von Teams.
- Der Kunde muss bereit sein, auf eine separate Dokumentation der Software zugunsten ausführlicher Tests zu verzichten.

Randbedingungen von XP

- Programmierer versuchen nicht, bei der Realisierung des aktuellen Releases bereits künftige Releases mit zu „antizipieren“.
- Programmierer sind aber zum ständigen Umbau (Redesgin, Refactoring) der bereits erstellten Software bereit.

Diskussion: XP

Pluspunkte:

- Kompromiss zwischen der gelebten Wirklichkeit der Programmentwicklung und sehr aufwendigen Vorgehensmodellen (systematisches „Hacking“).
- Betonung der menschlichen Komponente: 40-Stunden-Woche, intensive Kommunikation, Pair Programming, Empfehlungen für die Arbeitsumgebung.
- Agilität durch systematisches evolutionäres Rapid Prototyping sowie permanente Modifikationen in Kombination mit Regressionstesten und Refactoring.

Diskussion: XP

Minuspunkte:

- Endprodukt besitzt keine Dokumentation (in vielen Bereichen aber sogar gesetzlich zwingend!).
- Entwicklung des benötigten Gesamtsystems durch schnelle kleine Releases nicht unbedingt sehr zielgerichtet.
- Grundannahmen (z.B. leichte Änderbarkeit des Codes über gesamte Projektlaufzeit) nicht hinreichend empirisch belegt.
- Nur für kleine, risikoarme Projekte in nicht sicherheitskritischen Anwendungsbereichen.

Was ist Scrum?

"Traditionally, Scrum project management is a software agile development process. Scrum models allow projects to progress via a series of iterations called agile ***sprints****. Each sprint is typically two to four weeks, and sprint planning in the agile methodology and Scrum process is essential. While the agile Scrum methodology can be used for managing any project, the Scrum agile process is ideally suited for projects with rapidly changing or highly emergent requirements like software"*

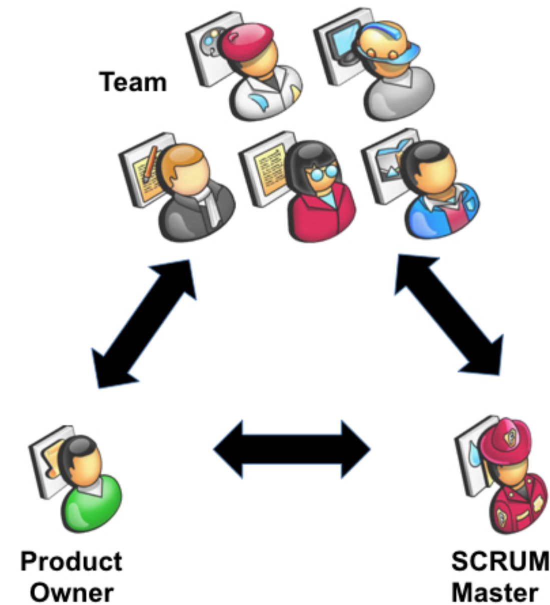
[http://www.mountaingoatsoftware.com/topics/scrum]

Grundwerte von Scrum

- **Commitment:** Wenn ich mir eine Aufgabe aussuche, nehme ich auch die Verantwortung an, diese Aufgabe umzusetzen.
- **Fokus:** Kurze, gekapselte Iterationszyklen bieten klare, für die Entwickler schnell erreichbare Ziele, auf die sie sich voll und ganz konzentrieren können.
- **Offenheit:** Offen gegenüber neuen Techniken und Denkweisen aber auch Transparenz in Bezug auf Konflikte, Anforderungen und Informationen
- **Respekt:** Respektvoller Umgang miteinander – Jeder hat etwas zum Team beizutragen.
- **Mut:** Probleme offen ansprechen, neue Denkweisen vorstellen, Kollaborationen anstoßen.

Rollen in Scrum

- Product Owner.
- Scrum-Master
- Das Team.



Rolle: Product Owner

- Definiert die Eigenschaften des Produktes.
- Bestimmt Datum der Fertigstellung und Inhalt.
- Verantwortlich für die Wirtschaftlichkeit (ROI).
- Priorisiert Eigenschaften (Iterativ, wenn nötig in jedem Entwicklungszyklus neu).
- Akzeptiert die Arbeitsprodukte oder lehnt sie ab.

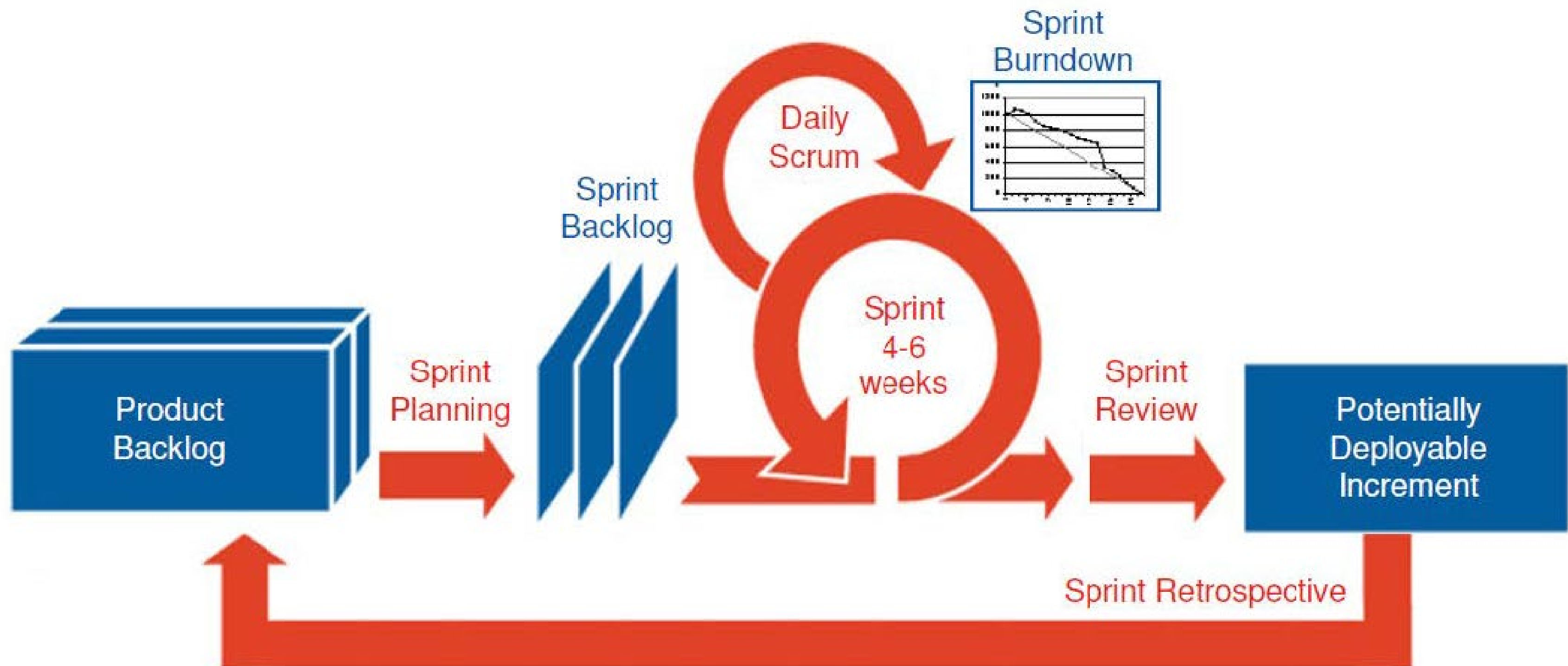
Rolle: Scrum-Master

- Agiert als Stellvertreter des Teams und „schützt“ das Team vor externen Einflüssen.
- Verantwortlich dafür, dass Scrum-Praktiken gepflegt und umgesetzt werden.
- Sorgt dafür, dass „Störfaktoren“ aus dem Weg geräumt werden.
- Sorgt dafür, dass das Team funktionsfähig und produktiv sein kann.

Rolle: Das Team

- Normalerweise zwischen 5-9 Personen.
- Übergreifende Funktionalitäten: Programmierer, Tester, UI-Entwickler, …
- Wissenstransfer mittels Pair-Programming o.ä.
- Jeder sollte Vollzeit arbeiten.
- Teams sind selbst-verwaltend: Jeder darf sich seine Aufgaben selbst aussuchen.
- …

Der Scrum-Prozess



Backlog

- Product-Backlog: Sammlung an Anforderungen/Spezifika aus Sicht des Product-Owner.
- Sprint-Backlog: Sammlung an und Verfolgung von Arbeitseinheiten zur Umsetzung für den aktuellen Sprint.
- Teammitglieder suchen sich ihre Aufgaben heraus und schätzt den Aufwand in Arbeitsstunden.
- Jedes Teammitglied darf den Sprint-Backlog anpassen.
- Wenn Spezifikation unklar ist (z.B. User Story), dann darf dies in mehrere Sprint-Backlog-Items aufgeteilt werden, zur Umsetzung im nächsten Sprint.

User Stories

Anforderungen an das Produkt im Backlog werden **ausschließlich** anhand von „User Stories“ formuliert.

Beispiel: Software für Urlaubsbuchung.

User Story: „As a vacation planner, I want to see photos of the hotel. I first want to see only a few interesting photos to get a first impression and optionally many more of them…”

Sprint

- Eigentliche Entwicklung des Produktes: Umsetzung der Anforderungen aus dem Backlog.
- Ziel eines Sprints ist eine Aussage, auf was sich der Sprint fokussieren wird (z.B. Weiterentwicklung einer Simulation, ...)
- Tägliche Updates:
  - Geschätzter Restaufwand.
  - Spezifika, sofern Neues bekannt wird.

Tägliches Scrum-Meeting (Daily Scrum)

- Tägliches, 15-minütiges Treffen.
- Aufstehen und vor der Gruppe Statusbericht geben.
- Dient **nicht** zum Lösen der Probleme.
- Jeder kann dazu kommen, nur Projektmitglieder dürfen sprechen.
- Andere Meetings sollen damit unnötig werden.

Daily Scrum

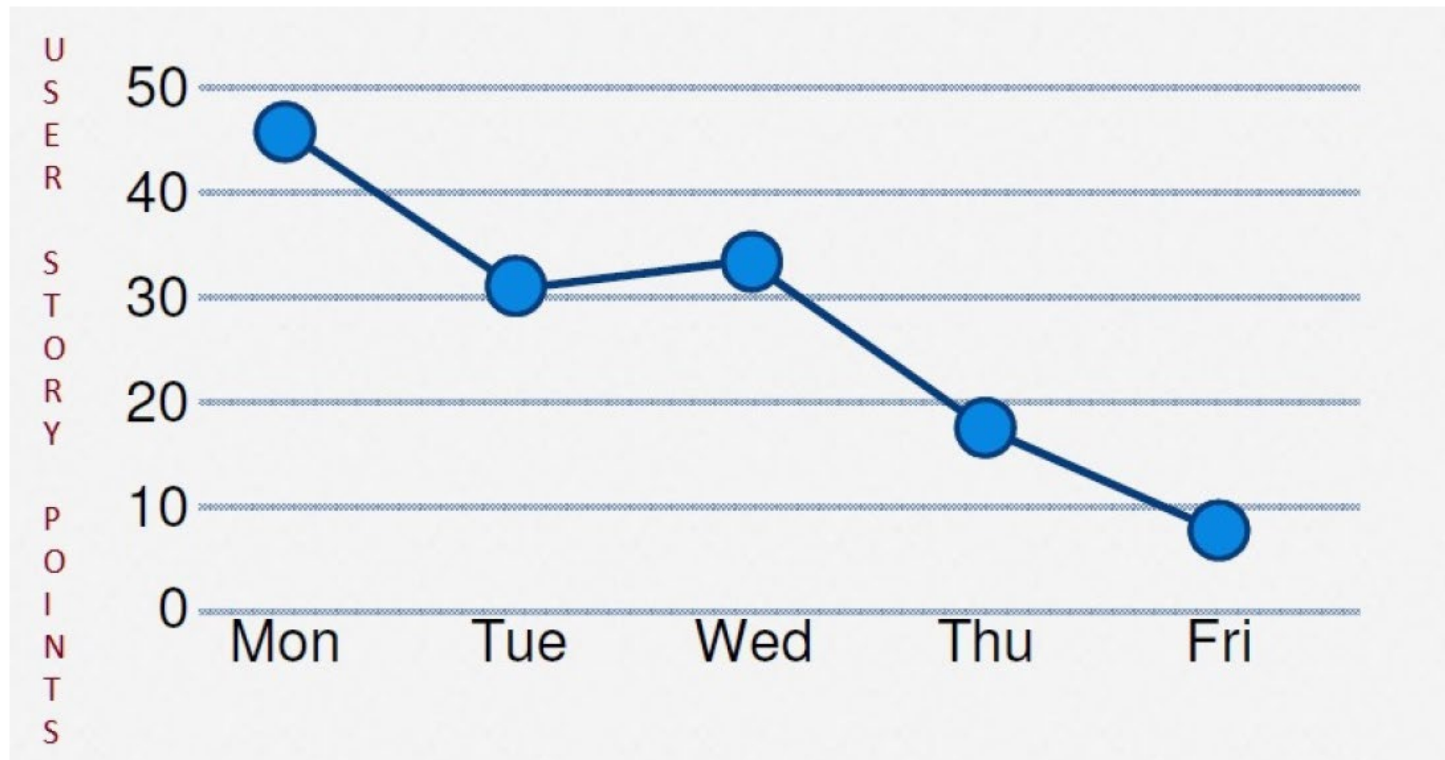
Jedes Projektmitglied hat folgende drei Fragen zu beantworten:

- *Was habe ich gestern gemacht?*
- *Was werde ich heute machen?*
- *Steht uns irgend etwas im Weg?*

Der Statusbericht ist nicht für den Scrum-Master gedacht, sondern für das ganze Team.

UNIVERSITÄT SIEGEN

Burndown Charts mit verbrauchten Stunden



Sprint-Review

- Abnahme eines (ausführbaren) Inkrements zusammen mit Product-Owner:
  - Testen der gesetzten Ziele gemäß Sprint-Backlog.
- Feedback einholen anhand dem erarbeiteten Inkrement:
  - Was wird abgenommen? Streichen aus dem Product-Backlog.
  - Was muss geändert werden? Neuer Change Request ins Product-Backlog.
  - Was wird abgelehnt? Anforderungen wandern wieder ins Product-Backlog.

Sprint-Review

- Review dient als Möglichkeit zur Diskussion, wie sich das Projekt weiter entwickeln soll.
- Scrum-Master leitet und notiert sich neue Anforderungen.
- Später dann Abgleich, was wirklich umgesetzt werden soll (gemäß ROI).

Sprint-Retrospektive

- Dient zur Auswertung des **letzten** Sprints und somit als Input für den **nächsten** Sprint.
  - Was ist schlecht gelaufen?
  - Was kann im nächsten Sprint besser gemacht werden?
- Aufteilen der Probleme:
  - Team-intern: Wird vom Team gelöst.
  - Externe Kunden-spezifische Probleme: Scrum-Master.

Sprint-Retrospektive

- Probleme und Lösungsansätze werden in der nächsten Sprint-Planung zeitlich geschätzt und bewertet.
- Muss von **jedem** vorbereitet werden.

Diskussion: Scrum

- Scrum = XP + „Modernes Projektmanagement“!?
- Fokus von Scrum ist Projektmanagement, nicht der eigentliche Entwicklungsprozess / Werkzeuge / Technologien /…!
- Damit Scrum funktioniert, braucht es weiterhin: *Daily Builds*, *Testautomatisierung*, *Testgetriebene Entwicklung*, *Pair-Programming Richtlinien zur Programmierung und Dokumentation*, … **(siehe folgende Kapitel!)**

Agile - Ein abschließende Polemik

The Good

- Promoting (code) refactoring
- Short daily meetings
- Team Communication

…

The Brilliant

- Short time-boxed iterations
- Closed window rule (frozen functionality during iteration)
- Continuous integration & regression testing
- Clearly defined product owner
- Delivering working software
- Associating a test with every piece of software

…

[Agile! - The Good, the Hype and the Ugly, 2014]

Agile – Ein abschließende Polemik

The Ugly / Bad

- User Stories as a basis for requirements
- Rejection of upfront generalization
- Rejection of traditional manager tasks
- Deprecation of documents

…

The Hyped

- Pair programming
- Self-organizing teams
- Working at a sustainable pace
- Producing minimal functionality
- Collective code ownership

…

[Agile! - The Good, the Hype and the Ugly, 2014]

Qualitätssicherung nach ISO 9000

- Das **ISO 9000 Normenwerk** legt für das Auftraggeber-Lieferantenverhältnis einen allgemeinen organisatorischen Rahmen zur Qualitätssicherung fest.
- Das **ISO 9000 Zertifikat** bestätigt, dass die Verfahren eines Unternehmens der ISO 9000 Norm entsprechen.
- Darüber hinaus kann es je nach Anwendungs-domäne spezifische Standards geben (z.B. ISO 26262 im Automotive-Bereich).

[http://www.iso-9000.co.uk/]

Wichtige Bestandteile der ISO 9000

- **ISO 9000-1**: allgemeine Einführung und Überblick.
- **ISO 9000-3**: Anwendung von ISO 9001 auf Softwareproduktion.
- **ISO 9001**: Modelle der Qualitätssicherung in Design/Entwicklung, Produktion, Montage und Kundendienst.
- **ISO 9004**: Aufbau und Verbesserung eines Qualitätsmanagementsystems.

CMMI

Das Capability Maturity Model Integration (CMMI) integriert verschiedene Qualitäts-Modelle für unterschiedliche Entwicklungs-Disziplinen (z.B. für Software-Entwicklung oder System-Entwicklung) in einem modularen Modell.

CMMI: Fähigkeitsgrade für Prozessgebiete

0 - Incomplete: Ausgangszustand, keine Anforderungen.

1 - Performed: die spezifischen Ziele des Prozessgebiets werden erreicht.

2 - Managed: der Prozess wird gemanagt.

3 - Defined: der Prozess wird auf Basis eines angepassten Standard-Prozesses gemanagt und verbessert.

4 - Quantitatively Managed: der Prozess steht unter statistischer Prozesskontrolle.

5 - Optimizing: der Prozess wird mit Daten aus der statistischen Prozesskontrolle verbessert.

UNIVERSITÄT SIEGEN

CMMI: Reifegrade erforderter Fähigkeitsgrade

1- Initial: keine Anforderungen, diesen Reifegrad hat jede Organisation automatisch.

2 - Managed: die Projekte werden gemanagt durchgeführt und ein ähnliches Projekt kann erfolgreich wiederholt werden.

3 - Defined: die Projekte werden nach einem angepassten Standard-Prozess durchgeführt, und es gibt eine kontinuierliche Prozessverbesserung.

4 - Quantitatively Managed: es wird eine statistische Prozesskontrolle durchgeführt.

5 - Optimizing:die Prozesse werden mit Daten aus statistischen Prozesskontrolle verbessert.

Prüfungsstoff

- Grundbegriffe zu Entwicklungsprozessen und Qualitätssicherung kennen.
- Probleme mit der Pflege „langlebiger“ Software erläutern können.
- Die Begriffe „Software-Wartung und -Evolution“ sinnvoll definieren können.
- Forward-, Reverse- und Reengineering unterscheiden können.
- Wichtige Prozessmodelle kurz skizzieren.

Literatur

- P. Liggesmeyer: ***Software-Qualität: Testen, Analysieren und Verifizieren von Software***, Spektrum Akademischer Verlag (2009).
- B. Kahlbrandt: ***Software-Engineering: Objektorientierte Software-Entwicklung mit der Unified Modeling Language***, Springer Verlag (1998)
- B. Meyer: ***Agile! The Good, the Hype and the Ugly,*** Springer-Verlag (2014)