

# Entwicklung eines Prozessmodells für die evolutionäre Software- Entwicklung im Forschungsumfeld

Diplomarbeit am Fraunhofer IPT

Dieses Dokument ist leicht zensiert  
(»Geheimhaltung«). Die vollständige  
Version gibts auf Anfrage.

Prof. Dr. rer. nat. Horst Lichter  
Fakultät für Informatik  
RWTH Aachen

von

and. inform.

**Alexander Gran**

Matrikelnummer: 25 23 86

Erstprüfer: Prof. Dr. rer. nat. Horst Lichter

Zweitprüfer: Prof. Dr.-Ing. Dr.-Ing E.h. Fritz Klocke

Betreut von: Dipl. Inform. Dipl. Ing. Lothar Glasmacher

Tag der Anmeldung: 20. November 2007

Tag der Abgabe: 31. März 2008

---



---

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 31. März 2008



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Umfeld</b>	<b>5</b>
<b>3</b>	<b>Stand der Wissenschaft</b>	<b>11</b>
3.1	Ungeplante Entwicklung . . . . .	15
3.2	Klassische Methoden . . . . .	16
3.2.1	Wasserfallmodell . . . . .	16
3.2.2	Spiralmodell . . . . .	19
3.2.3	Rational Unified Process . . . . .	21
3.2.4	V-Modell (XT) . . . . .	24
3.3	Agile Methoden . . . . .	27
3.3.1	eXtreme Programming . . . . .	28
3.3.2	Evolutionary Development . . . . .	33
3.3.3	Feature Driven Development . . . . .	35
3.3.4	Adaptive Software Development . . . . .	36
3.3.5	Scrum . . . . .	38
<b>4</b>	<b>Zielsetzung der Arbeit</b>	<b>43</b>
4.1	Probleme der bekannten Software-Entwicklungsprozesse . . . . .	45
4.2	Zieldefinition . . . . .	50
<b>5</b>	<b>Software-Entwicklung als Fluss</b>	<b>51</b>
5.1	Generelles Vorgehen . . . . .	51
5.1.1	Motivation . . . . .	51
5.1.2	Idee der Software-Entwicklung als Fluss . . . . .	52
5.1.3	Entwicklungsablauf . . . . .	55
5.2	Aufwandsschätzung . . . . .	56
5.3	Priorisierung . . . . .	60
5.4	Projektstatus . . . . .	64
5.5	Release-Auskopplung . . . . .	68
5.6	Qualitätsmanagement . . . . .	69
5.7	Mitarbeitermanagement . . . . .	74

---

<b>6</b>	<b>Prozessdetails</b>	<b>79</b>
6.1	Große Umbauten . . . . .	79
6.2	Anforderungsanalyse . . . . .	81
6.3	Design . . . . .	82
6.4	Dokumentation . . . . .	83
6.5	Test . . . . .	86
6.6	Werkzeugunterstützung . . . . .	88
6.6.1	Versionsverwaltungssystem . . . . .	88
6.6.2	Change Request & Project Management . . . . .	90
6.6.3	Wiki . . . . .	91
6.6.4	Qualitätssicherung . . . . .	92
6.7	Prozesseinführung . . . . .	94
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>97</b>
	<b>Literaturverzeichnis</b>	<b>99</b>

# Abbildungsverzeichnis

2.1	Wechselwirkungen bei der Software-Entwicklung im Forschungsumfeld. . . . .	6
2.2	Release-Auskopplung aus Entwicklungszyklus. . . . .	8
3.1	Überblick über die CMMI Reifegrade von 1995 bis 2007. . . . .	14
3.2	Überblick der gebräuchlichsten Software-Entwicklungsprozesse .	15
3.3	Wasserfallmodell mit fünf Stufen. . . . .	16
3.4	Wasserfallmodell mit sieben Stufen. . . . .	17
3.5	Das Spiralmodell. . . . .	20
3.6	Der IBM Rational Unified Process. . . . .	22
3.7	Das V-Modell XT 1.2 im Überblick. . . . .	25
3.8	Ablaufdetails V-Modell XT. . . . .	26
3.9	Belady-Lehman Graph: Permanente Änderungen erhöhen Fehlerzahl. . . . .	27
3.10	Wasserfallmodell, inkrementelle und evolutionäre Entwicklung.	34
3.11	Feature Driven Development. . . . .	35
3.12	Der Lebenszyklus im Adaptive Software Development. . . . .	37
3.13	Der Scrum Entwicklungsprozess im Überblick. . . . .	39
3.14	Drei Stufen der Scrum Implementierung. . . . .	40
4.1	Anforderungserfüllung der Software-Entwicklungsprozesse. . .	49
5.1	Software-Entwicklung als Fluss im Großen. . . . .	53
5.2	Software-Entwicklung als Fluss im Kleinen. . . . .	54
5.3	Effort Points auf verschiedenen Stufen. . . . .	58
5.4	Der Restaufwand als erweitertes Burndown Chart. . . . .	58
5.5	Gegenüberstellung von Zeit, Ressourcen und Featuremenge. . .	60
5.6	Expected Cost of Change: Agile vs. Traditionelle Methoden. . .	61
5.7	Sinkender Expected Cost of Change. . . . .	63
5.8	Die sieben möglichen Projektstatusse. . . . .	66
5.9	Stetige Qualitätssicherung bei evolutionärer Entwicklung. . . .	70
5.10	Wichtige Qualitäten im Qualitätsbaum. . . . .	71
5.11	Metriken zur Überwachung der Fehlerbehebung. . . . .	73

6.1	Subversion Webinterface. . . . .	89
6.2	FogBugz mit Subversion Integration. . . . .	90
6.3	Schrittweise Adaption des Software-Entwicklungsprozesses. . .	94

# Kapitel 1

## Einleitung

Software wird in vielen verschiedenen Bereichen für die unterschiedlichsten Aufgaben entwickelt. Jeder Entwicklung liegt dabei bewusst oder unbewusst ein Modell zugrunde nach der sie abläuft. Dieses prägt mehr oder weniger stark den Entwicklungsprozess und steuert die beteiligten Entwickler. Dass es hier keinen Königsweg gibt, erkannte Brooks bereits 1987 in seinem bekannten Artikel »No silver bullet« [Bro87].

Für einige wenige Entwicklungsbereiche ist der Prozess der Software-Entwicklung derart gut verstanden, dass schon von Software-Produktion und nicht mehr von Entwicklung gesprochen werden kann, so z. B. die Compiler-Entwicklung. Für viele andere Bereiche gibt es ein zumindest grundlegendes Verständnis der beteiligten Akteure und der spezifischen Anforderungen, so dass entsprechende Prozessmodelle entstehen können und bereits entstanden sind. Ein Bereich, der in der Literatur bisher fast gar nicht betrachtet wurde, ist die Software-Entwicklung im Forschungsumfeld. Forschungsumfeld meint die Entwicklung von Software in fremden Bereichen. So wird in allen Wissenschaften heute Software für die verschiedensten Aufgaben eingesetzt. Dabei handelt es sich häufig nicht um Standardsoftware, die auch in anderen Anwendungen Verwendung findet, sondern es ist eine spezifisch für den entsprechenden Bereich entwickelte Sonderlösung.

Die Entwicklung dieser Software ist mit sehr spezifischen Problemen verbunden, die in dieser Arbeit betrachtet und zu den Lösungsmöglichkeiten aufgezeigt werden sollen. Erste Betrachtungen zu den spezifischen Problemen der Software-Entwicklung im Forschungsumfeld hat Segal gemacht

[Seg03, Seg05]: Der Software-Entwicklungsprozess in Forschung und Entwicklung unterliegt besonderen Anforderungen und Beschränkungen. Den klassischen Prozessmodellen der Software-Entwicklung ist gemein, dass zu Beginn der Entwicklung das Ziel – zumindest den Anwendern – grob bekannt ist. Dies ist bei der Entwicklung von Software im Forschungsumfeld im Allgemeinen nicht der Fall. Häufig handelt es sich um Simulations- und Optimierungssoftware, die grundsätzlichen Probleme tauchen aber auch bei jeder anderen Art auf. Es soll Software für ein Gebiet entwickelt werden, das bei den Anwendern nur unzureichend verstanden ist, schließlich wird genau deshalb aktiv geforscht.

Auch ist die Software nicht fertig, wenn sie das erste Mal in Betrieb ist, so dass sie dann in einer Wartungsphase weiter betreut werden könnte. Vielmehr werden sofort weitere Anforderungen aus den Ergebnissen erwachsen, die durchaus widersprüchlich zu den ursprünglichen Anforderungen sein können. Der Lebenszyklus der Software hat daher kein vorher festgelegtes Ende. Diese und einige weitere ungewohnte Anforderungen machen einen neuen Software-Entwicklungsprozess notwendig, der im Zuge dieser Arbeit entworfen wird.

Der Autor entwickelt am Fraunhofer IPT seit mehreren Jahren die Software NCProfiler zur Analyse und Optimierung von NC-Programmen für numerisch gesteuerte Arbeitsmaschinen. Dies ist gerade eines der soeben beschriebenen Spezialprogramme für die Forschung. Die Ergebnisse der Analysen haben teilweise direkten Einfluss auf die zu erfüllenden Anforderungen. Die während der Entwicklung dieser Software gemachten Beobachtungen fließen neben theoretischen Betrachtungen in diese Arbeit ein.

Die Arbeit beschreibt die »Software-Entwicklung als Fluss«, als mögliche Lösung der angesprochenen Probleme. Hierbei handelt es sich um einen evolutionäreren Ansatz, der es ermöglicht, mit stark fluktuierenden Anforderungen ebenso umzugehen, wie mit einem nicht endenden Lebenszyklus. Durch eine weitgehend unabhängige Arbeitsgestaltung ist es möglich, dass mehrere Entwickler parallel an verschiedenen Teilen arbeiten. Dennoch sind Maßnahmen vorgesehen, die eine Koordination ermöglichen, beispielsweise wenn ein Release an einen Kunden ausgeliefert werden muss.

Im Folgenden wird zunächst das Forschungsumfeld kurz beschrieben, dazu zählt auch eine kurze Einführung in den NCProfiler, damit die daraus

resultierenden Beispiele leichter verstanden werden können. Anschließend folgt in Kapitel 3 ein Überblick über den Stand der Wissenschaft, in dem die verschiedenen bekannten Software-Entwicklungsprozesse beschrieben werden. Kapitel 4 erörtert zunächst die Schwachstellen der bekannten Modelle bei einer Verwendung im Forschungsumfeld, anschließend werden konkrete Anforderungen an einen Software-Entwicklungsprozess definiert, die das Forschungsumfeld auferlegt. Die folgenden beiden Kapitel beschreiben den entwickelten neuen Software-Entwicklungsprozess, wobei Kapitel 5 einen Überblick über das Vorgehen gibt, und die Kernaspekte vorstellt. Kapitel 6 beschreibt einige komplexe Details und liefert einen Wegweiser zur schrittweisen Umsetzung.



# Kapitel 2

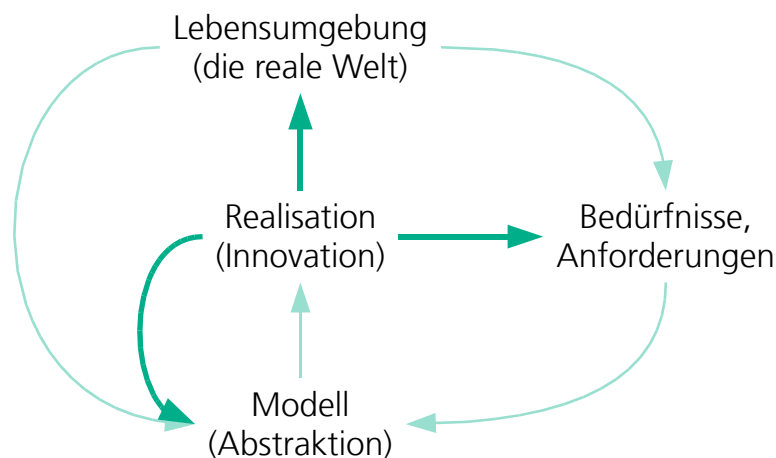
## Umfeld

In der Einleitung wurden bereits einige Eigenarten des Forschungsumfeldes beschrieben. Diese Beschreibung wird im Folgenden vertieft, ebenso wird die zusätzliche Möglichkeit in Betracht gezogen, dass Dritte die entwickelten Software-Produkte erwerben wollen.

Bei der Software-Entwicklung im Forschungsumfeld ist der Software-Entwickler Dienstleister für andere Forscher, diese Konstellation ist sowohl bei reinen Forschungseinrichtungen (z. B. Max-Planck oder Fraunhofer Institute, Universitäten), als auch in Entwicklungsabteilungen der Industrie vorhanden. Dabei sind mehrere Konstellationen denkbar, angefangen bei einer Vorlauf- oder Grundlagenforschung ohne Partner über bilaterale Entwicklungsprojekte bis hin zu spezialisierten internationalen Projekten mit verschiedenen Partnern aus mehreren der genannten Gruppen. In allen Fällen ist es dennoch möglich, die Auftraggeber der Software-Entwickler in zwei Gruppen einzuteilen: Forscher bzw. Anwender im eigenen Unternehmen sowie Externe.

Die Forscher haben Aufgaben im Bereich der Informationsverarbeitung, Simulation, Analyse oder Optimierung aus ihrem jeweiligen Forschungsgebiet, welche sie mit einer zu entwickelnden Software lösen wollen. Die zu entwickelnde Software im Forschungsumfeld dient daher häufig dazu, Teile der Realität zu simulieren oder bestehende Prozesse zu optimieren. Es liegt allerdings in der Natur der Forschung, dass die genauen Abläufe, die beispielsweise simuliert werden sollen, noch unbekannt sind. Die Software soll ja gerade zu deren Verständnis beitragen. Bei angeschlossenen Optimierungsaufgaben verhält es sich ähnlich: Oft ist lediglich ein grobes Ziel

bekannt (z. B. Prozess XY soll schneller werden). Wie dies erreicht werden kann ist aber gerade auch Gegenstand der Forschung. Die Ergebnisse der Software helfen dann dabei, die Umwelt zu verstehen, und erst sie ermöglichen Simulation und Optimierung. Daher erlauben erst die Ergebnisse der Software, die sie schon während ihrer Entstehung liefert, die genauere, weitergehende Spezifikation ihrer selbst. Dieses sehr intensive Feedback vom Ergebnis der Software zu Anforderung, Modell und Umgebung ist in Abbildung 2.1, nach [LL07], fett dargestellt.



**Abbildung 2.1: Wechselwirkungen bei der Software-Entwicklung im Forschungsumfeld.**

Der Entstehungsprozess hat aber noch eine weitere Besonderheit. So liegt es in der Natur der Forschung, dass diese niemals abgeschlossen ist. Jede Erkenntnis wirft auch neue Fragen auf, die es zu beantworten gilt. Daher muss auch der Lebenszyklus der verwendeten Software von vorneherein anders ausgelegt werden. Es gibt kein konkretes Ziel, das erreicht werden könnte, und ein Ende der Entwicklung markieren würde. Statt dessen wird das Modell der Realität von den Forschern permanent überarbeitet und verfeinert, und zieht daher Anpassungen im Softwaremodell nach sich. Eine klassische Wartungsphase wird daher gar nicht erreicht. Ist die Software einmal fertig, ist sie im selben Moment auch nutzlos.

Die Anforderung der klassischen Software-Entwicklungsprozesse, dass die Anforderungen so vollständig und korrekt wie möglich vorliegen sollen, bevor mit Design und Implementierung begonnen wird, sind widersprüchlich zu der Denkweise von Wissenschaftlern [Seg03]. Sie glauben immer, sie

---

könnten die Anforderungen bis zuletzt ändern, Software-Entwickler bevorzugen aber eine Festlegung und Priorisierung der Aufgaben vor Beginn der Entwicklung.

Testdaten und Sollwerte können im Forschungsumfeld häufig nicht einfach hergeleitet werden. Sie entstehen möglicherweise erst beim ersten Einsatz der Software [Seg05] oder sind in ihrer Berechnung derart aufwendig, dass sie manuell nicht praktikabel durchgeführt werden können. Dann können nur Messungen an realen Prozessen Zielwerte liefern.

Eine weitere Besonderheit ergibt sich aus den Rahmenbedingungen, die der Software-Entwicklung auferlegt werden. So ist die Forschung häufig universitätsnah angelegt. Dadurch kann es passieren, dass ein beachtlicher Teil der Arbeit im Zuge von Studienleistungen erbracht wird, beispielsweise als Diplom-, Master- oder Studienarbeit. Diese Teile müssen möglicherweise als abgeschlossene Blöcke erzeugt werden, und dennoch anschließend effektiv in die Software integriert werden. Damit einher geht auch eine ungewöhnliche Mitarbeiterstruktur, die sich zum Teil deutlich vom klassischen Festangestellten mit definierten Aufgaben und Fähigkeiten unterscheidet: So stammen die Mitarbeiter häufig aus der Universität, z. B. als Werksstudenten bzw. studentische Hilfskräfte (Hiwi). Diese sind dann auf Teilzeitbasis angestellt und stehen daher zum einen nicht permanent zur Verfügung. Zum anderen sind etwaige Schwankungen in der Wochenarbeitszeit, z. B. wegen Prüfungen, zu berücksichtigen. Die Anstellung als Hiwi bedingt wegen der universitären Strukturen häufig auch eine strikte und unflexible Tarifierung.

Auch wenn die Forschung nicht im Rahmen einer Universität stattfindet, sind Forschungsabteilungen wegen des hohen Kapitalbedarfs regelmäßig nur bei größeren Unternehmen anzutreffen. Beide Varianten induzieren nahezu immer recht unflexible (Verwaltungs-)Strukturen, die nicht einfach geändert werden können.

Zudem ändert sich der inhomogene Wissensstand der Studierenden einerseits bedeutend schneller und unregelmäßiger als bei gewöhnlichen Mitarbeitern, die sich nur nebenher fortbilden. Andererseits ist aber die Erfahrung mit der Entwicklung von Software bei ihnen sehr eingeschränkt. Sowohl die Studenten als auch – bei sehr universitätsnahen Einrichtungen – die anderen Kräfte der Software-Entwicklung sind nur wenige Jahre im

Betrieb aktiv, wohingegen die entwickelte Software möglicherweise deutlich länger besteht.

Auch muss bedacht werden dass möglicherweise Releases an Forschungspartner oder erste Kunden *ausgekoppelt* werden sollen. *Auskoppeln* meint dabei, einen aktuellen Erkenntnis- und damit Software-Stand für einen Kunden benutzbar zu veröffentlichen. Dieses kann natürlich auch bewusstes Ziel der Forschung sein, also einen Prozess hinreichend gut zu verstehen, um das entstandene Programm dann zu verkaufen. Für derartige Releases ist eine Auslieferung in vielen Iterationen, wie sie an die internen Partner erfolgen kann, nicht möglich. Das Auskoppeln ist in Abbildung 2.2 dargestellt.

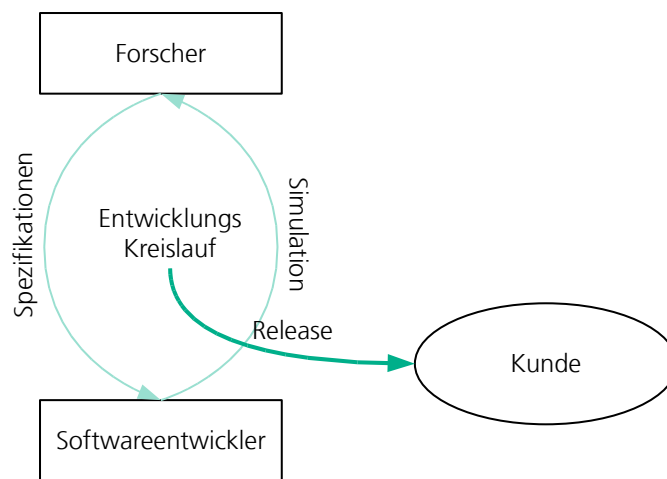


Abbildung 2.2: Release-Auskopplung aus Entwicklungszyklus.

Alternativ handelt es sich möglicherweise um Mischformen zwischen gezielter Entwicklung und Forschung. In diesem Fall besteht die zusätzliche Aufgabe darin, einen Preis für eine gewünschte Weiterentwicklung festlegen zu können. Möglicherweise ist sogar geplant die Software mittels einer eigens dafür zu gründenden Firma (Spin-off) in einem breiteren Markt zu etablieren [Moo95].

Auch die Ausstattung entspricht nicht der eines klassischen Software-Hauses. Einerseits werden teilweise teure Programme wegen des Forschungsumfeldes sehr kostengünstig vom Hersteller zur Verfügung gestellt, andererseits kann die Beschaffung neuer Ressourcen aus Verwaltungsgründen schwierig sein. Durch die verhältnismäßig lange Entwicklungszeit der Software, ist mit

einer deutlichen Weiterentwicklung der eingesetzten CASE<sup>1</sup> Werkzeuge zu rechnen. Zensierter Abschnitt

Zensierter Abschnitt

---

<sup>1</sup>CASE: Computer-Aided Software Engineering engl. für rechnergestützte Software-Entwicklung.



# Kapitel 3

## Stand der Wissenschaft

Ein *Software-Entwicklungsprozess* beschreibt den Prozess der Erzeugung von Software durch Entwickler. Außer bei *ungeplanter Entwicklung* (s.u.) bedienen sich die Software-Entwickler dabei eines *Prozessmodells*, das den Prozess in einzelne, definierte Abschnitte gliedert. Die Art der Unterteilung hängt vom Prozessmodell ab, und kann sowohl zeitlich, inhaltlich als auch personenbezogen sein. Auch etwaige Phasenübergänge werden teilweise vom Prozessmodell spezifiziert. Je nach Detailgrad wird auch vorgeschrieben, wie in diesen Teilen vorzugehen ist und welche Dokumente wann von wem zu erzeugen sind.

Genau wie die Software selbst werden auch die Software-Entwicklungsprozesse permanent weiterentwickelt. Durch Veränderungen der Möglichkeiten im Bereich von Hardware, Benutzern und Entwicklern werden hier permanent andere Software-Entwicklungsprozesse notwendig. Während in den Anfängen der Software-Entwicklung vor allem die Hardwareressourcen eingeschränkt waren, und keine Erfahrung mit Software-Entwicklungsprozessen vorlag, sind es heute vor allem die Humanressourcen, sowie die zur Verfügung stehende Zeit, die eingeschränkt sind. Mittlerweile sind verschiedenste Software-Entwicklungsprozesse bekannt und zum Teil auch gut verstanden. Es mangelt allerdings oft an detaillierten und vergleichbaren Erfahrungsberichten aus der Praxis.

Um Entwicklungsprozesse besser beurteilen zu können wurde das *Capability Maturity Model Integration* definiert [CKS03]. Es unterteilt vier Prozessgebiete: Projektmanagement, Entwicklung, Unterstützung und Prozessmanagement. In jedem dieser Gebiete gibt es weitere Unterteilungen

in einzelne Aufgaben. Um zu beurteilen wie gut ein Prozessmodell bei der Software-Entwicklung verwendet wird, werden *Fähigkeits-* sowie *Reifegrade* definiert. Die sechs Fähigkeitsgrade, nummeriert von 0 bis 5, beschreiben wie institutionalisiert ein Prozessgebiet ist. Sie bauen immer aufeinander auf, das heißt, dass z. B. ein Stufe 3 Prozessgebiet auch die Anforderungen der Stufen 0, 1 und 2 erfüllt. Im Einzelnen sind dies:

0. *Incomplete*

Der Prozess wird gar nicht oder nur in Teilen ausgeführt. Nicht alle Ziele werden erreicht, auch gibt es keine generischen Ziele auf dieser Stufe, da der Prozess nicht institutionalisiert, also definiert und niedergeschrieben, ist.

1. *Performed*

Die spezifischen Ziele des Prozessgebiets werden erreicht, dennoch sind sie nicht institutionalisiert und können daher mit der Zeit verloren gehen.

2. *Managed*

Es gibt eine elementare Unterstützung für den Prozess, die auch über längere Zeit eine Statuserhaltung ermöglicht. Der Prozess wird anhand seiner Vorschriften geplant, ausgeführt und überwacht. Die Mitarbeiter sind entsprechend geschult und die nötigen Ressourcen werden vorgehalten. Alle beteiligten Akteure werden beachtet.

3. *Defined*

Im Gegensatz zu einem Stufe 2 Prozess, wo Standards nur je Prozess definiert sind, existiert eine globale, d.h. unternehmensweite, Standarddefinition, die für die einzelnen Prozesse angepasst wird. Die Definitionen sind detailliert, sie spezifizieren Zweck, Eingaben und Ausgaben, Eingangsvoraussetzungen, Aktivitäten, Rollen, Prüfverfahren sowie Ausgangskriterien.

4. *Quantitatively Managed*

Sowohl Prozessqualität als auch Geschwindigkeit werden mit quantitativen oder statistischen Methoden ständig überwacht.

### 5. *Optimizing*

Veränderungen innerhalb des Prozesses werden verstanden und es wird versucht, den Prozess permanent zu optimieren. Dazu werden die Messwerte aus Stufe 4 eingesetzt.

Neben den sechs Fähigkeitsgraden werden noch fünf Reifegrade definiert. Diese Reifegrade beziehen sich auf die ganze Organisation, deren obere vier Stufen absichtlich den Fähigkeitsgraden für Prozesse entsprechen:

#### 1. *Initial*

Es gibt keine Anforderungen an das Vorgehen im Unternehmen. Diese Stufe wird immer erreicht.

#### 2. *Managed*

Die Prozessunterstützung kann von einem Projekt auf ein anderes erfolgreich übertragen werden.

#### 3. *Defined*

Es existiert eine unternehmensweite Standarddefinition für Projekte, die spezifisch angepasst werden kann.

#### 4. *Quantitatively Managed*

Die Prozessqualität und Geschwindigkeit aller Projekte werden mit quantitativen oder statistischen Methoden überwacht.

#### 5. *Optimizing*

Die Abläufe in den Prozessen sind im Detail verstanden, alle Entwicklungsprozesse werden permanent verbessert.

Das CMMI befasst sich also im Idealfall mit der permanenten Optimierung der Prozesse [CKS03] und hofft dadurch, ein besseres Produkt zu erzeugen. Kritik hierzu gibt es beispielsweise von Highsmith [Hig00], der Adaption höher bewertet als Optimierung.

Das Software Engineering Institute (SEI) an der Carnegie Mellon University in Pittsburgh USA, die Entwickler des CMMI, veröffentlichen permanent die Reifegrade von mehreren hundert Organisationen. Da die Auswahl durch regulatorische Vorgaben der US-Behörden und eine ansonsten freiwillige Meldung beeinflusst wird, können die Messungen nicht als repräsentativ

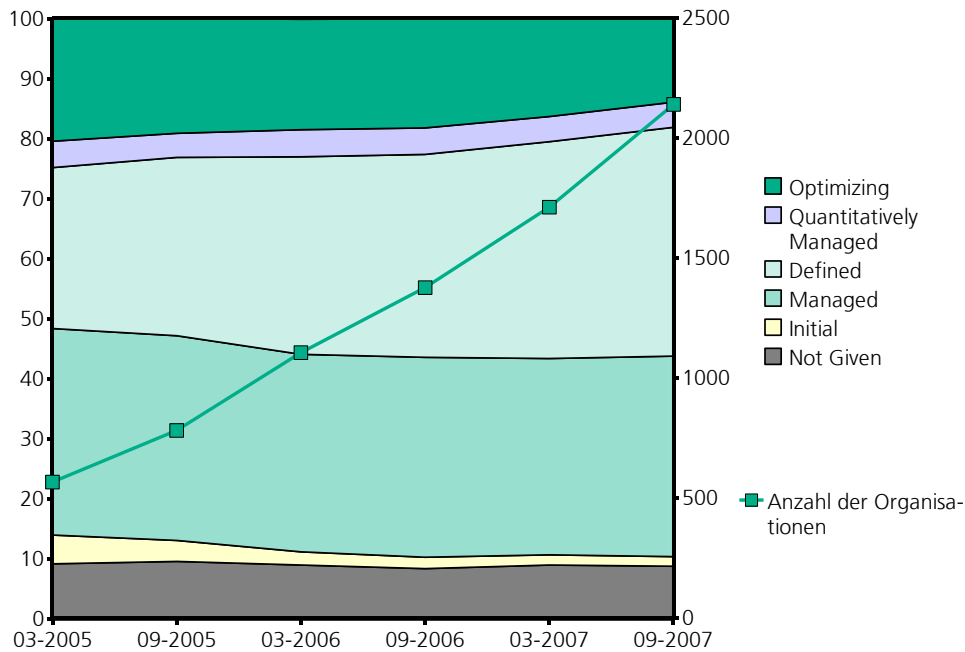


Abbildung 3.1: Überblick über die CMMI Reifegrade von 1995 bis 2007.

angesehen werden. Die Entwicklung von März 2005 bis September 2007 ist in Abbildung 3.1 dargestellt [Sei07].

Den einzigen umfassenden, aktuellen und wissenschaftlich brauchbaren Vergleich von Prozessmodellen liefern Benediktsson et al. [BDT05, BDT06]. Dabei entwickelten fünfzehn Teams zu je drei bis vier Entwicklern verschiedene Software unter Verwendung von V-Modell, Inkrementeller Entwicklung, Evolutionärer Entwicklung sowie eXtreme Programming. Die Ergebnisse sind nicht verwunderlich und belegen die Behauptungen der Verfechter agiler Software-Entwicklungsprozesse:

So brauchten die XP Teams nur sehr wenig Zeit für die Anforderungsanalyse und erzeugten die 3,5-fache Menge an Quellcode im Vergleich zu V-Modell Teams. Dabei waren sie auch 4,8-mal so produktiv wie die V-Modell Teams, 2,8-fach produktiver als inkrementelle Teams und noch 2,3-mal so produktiv wie die evolutionär arbeitenden Teams. Die Qualitätsunterschiede der einzelnen Prozessmodelle lagen unterhalb des statistisch Relevanten. Bemerkenswert ist allerdings, dass die meisten Entwickler, die inkrementell, evolutionär oder mit XP gearbeitet haben, mit dem Vorgehen zufrieden waren, wohingegen *alle* Entwickler, die das V-Modell einsetzen mussten, lieber ein anderes Vorgehen gewählt hätten.

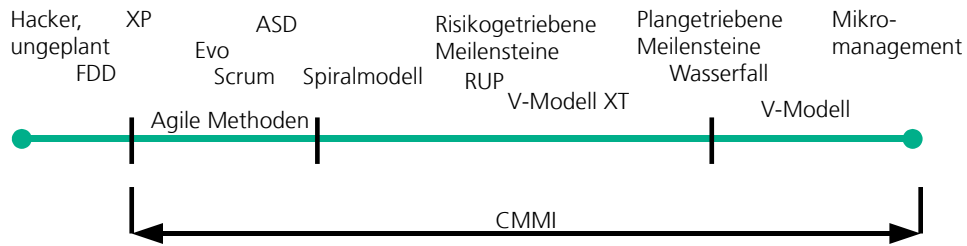


Abbildung 3.2: Überblick der gebräuchlichsten Software-Entwicklungsprozesse .

Abbildung 3.2, nach [Boe02], gibt einen Überblick über die im Folgenden behandelten, gebräuchlichsten Software-Entwicklungsprozesse. In den einzelnen Abschnitten werden auch die soweit verfügbaren Erfahrungsberichte aus der Praxis wiedergegeben.

### 3.1 Ungeplante Entwicklung

Eine Sonderstellung bei der Software-Entwicklung nimmt die ungeplante Entwicklung ein. Hier existiert kein Prozessmodell, es werden keine Phasen definiert, häufig wird ohne lange Überlegung mit der Implementierung der Software begonnen. Auf den ersten Blick unverständlich wird doch immer noch eine beachtliche Menge von Software entwickelt, ohne sich auf einen Software-Entwicklungsprozess festzulegen. Da die Software-Entwicklung planlos ist, sind auch verwertbare wissenschaftliche Berichte zu deren Erfolg knapp. Sicher kann angenommen werden, dass ein derartiges Vorgehen bei kleinen und überschaubaren Aufgaben erfolgreich sein kann, vor allem wenn nur ein Entwickler beteiligt ist. Sollten aber mehrere Entwickler über einen längeren Zeitraum zusammenarbeiten müssen, kann eine Planung nicht vernachlässigt werden.

Nicht weit von ungeplanter Entwicklung entfernt ist das sogenannte *Code and Fix* Vorgehen, in dem sich das Schreiben von Quellcode und das Beheben von Fehlern als einzige Arbeitsschritte permanent abwechseln. Die entstehende Software ist auch hier von minderer Qualität, eine Koordination liegt nicht vor.

## 3.2 Klassische Methoden

Im folgenden Kapitel werden einige klassische, das heißt vor allem nicht agile, Prozessmodelle beschrieben. Wasserfall- und Spiralmodell sind recht leichtgewichtige Vorgehensmodelle, wohingegen das V-Modell XT<sup>®</sup><sup>1</sup> und der Unified Process sehr stark dokumentierte und werkzeugunterstützte Verfahren sind.

### 3.2.1 Wasserfallmodell

Das *Wasserfallmodell* ist das älteste und einfachste Modell zur geplanten Software-Entwicklung, beschrieben bereits 1967 von Rosove [Ros67]. Es wurde entwickelt als erster Schritt zur strukturierten Software-Entwicklung und stellte daher einen bedeutsamen Fortschritt dar. Bereits 1970 erkannte Royce, dass die fehlenden Iterationen problematisch sind [Roy70]. Das Modell bildet daher nur dann den idealen Entwicklungsprozess ab, wenn zu Beginn die Anforderungen vollständig und fehlerfrei erkannt werden können.

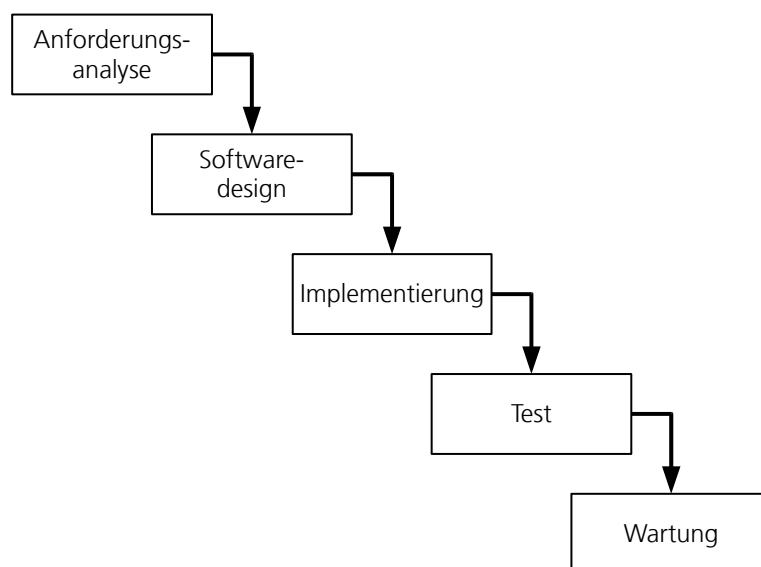


Abbildung 3.3: Wasserfallmodell mit fünf Stufen.

<sup>1</sup>V-Modell<sup>®</sup> ist eine geschützte Marke der Bundesrepublik Deutschland.

Es werden verschiedene Phasen durchlaufen, wobei die Reihenfolge fest vorgegeben ist. Dabei kann immer nur eine Phase weiter gesprungen werden. Einfache Varianten des Wasserfallmodells lassen auch ein Zurückspringen in die vorherige Phase zu, es ist aber niemals möglich in eine beliebige Phase zu springen. Die genaue Aufteilung und Definition der Phasen ist in der Literatur uneinheitlich. Abbildung 3.3 zeigt eine sehr einfache fünf-stufige Variante, Abbildung 3.4 eine komplexere mit sieben Stufen.

Aufgrund seiner Steifigkeit ist es hauptsächlich von theoretischem Interesse und kann nur bei sehr wenigen realen Projekten erfolgreich eingesetzt werden. Ein Beispiel für sinnvolle Einsatzgebiete ist die gut verstandene Entwicklung von Compilern.

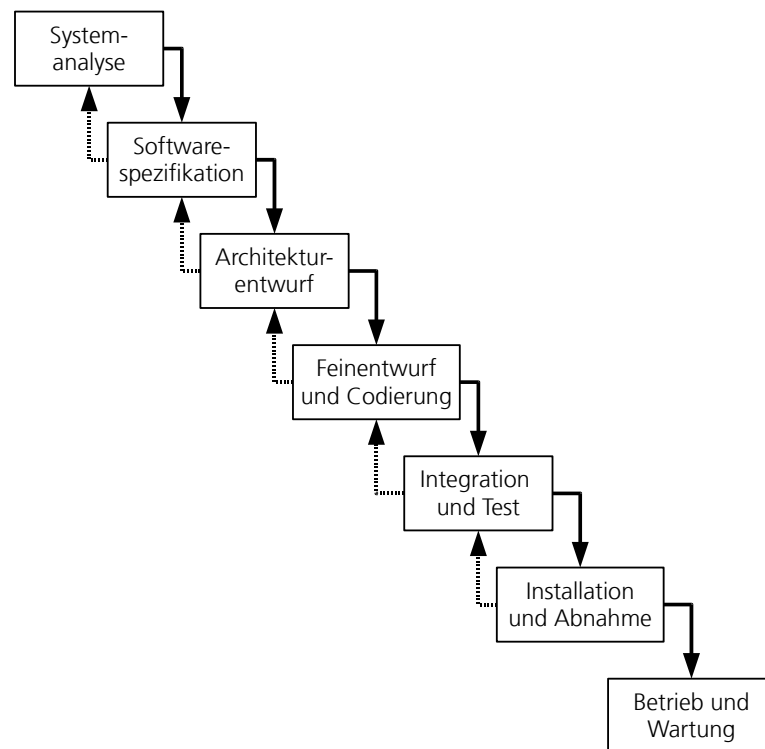


Abbildung 3.4: Wasserfallmodell mit sieben Stufen.

Die einzelnen Phasen finden sich in beinahe jedem anderen Modell abgewandelt wieder und seien im Folgenden kurz erläutert:

#### ■ Anforderungsanalyse

Bei der Anforderungsanalyse (engl. requirements analysis) versucht der Software-Entwickler die Anforderungen des Kunden – oder wenn

möglich des Endanwenders – zu verstehen und in geeigneter Form zu dokumentieren, um anschließend die Software nach diesen Vorgaben erstellen zu können. Da der Software-Entwickler sich für gewöhnlich einerseits nicht im fachlichen Umfeld des Anwenders auskennt, und andererseits der Anwender nicht die formalen Sprachen des Software-Entwicklers spricht, entsteht hier zwangsläufig Übersetzungsbedarf. Es muss versucht werden, Unklarheiten zwischen den beiden Parteien soweit wie möglich zu vermeiden. Allgemein wird verlangt, dass eine Anforderungsanalyse vollständig, eindeutig, verständlich, nachprüfbar und konsistent ist. Des Weiteren sollte die Form derart gewählt werden, dass die Anforderungen einzeln spezifiziert sind und eindeutig bezeichnet werden. Das Ergebnis der Anforderungsanalyse ist das Lastenheft. Die Anforderungsanalyse wird häufig eingebettet in die Anforderungserhebung (engl. requirements engineering). Damit ist zusätzlich noch die Strukturierung und Zuteilung sowie die Bewertung der Anforderungen gemeint. Ergebnis der Anforderungserhebung ist das Pflichtenheft.

#### ■ *Software-Design*

Im Software-Design wird ausgehend von Lasten- bzw. Pflichtenheft der Entwurf der Software erzeugt. Dabei kann zwischen Architektur-entwurf im Großen und Fein- bzw. Schnittstellenentwurf im Kleinen unterschieden werden. Es werden Komponenten festgelegt und ihre Beziehungen untereinander spezifiziert. Hier kommen verschiedene Notationsverfahren zum Einsatz, von formlosen Whiteboards über UML-Diagramme bis hin zu formal definierten Grammatiken.

#### ■ *Implementierung*

Bei der Implementierung wird anhand der vorhergehenden Entwürfe das Programm an sich erstellt. Dazu zählen auch (Anwender-)Dokumentation, Grafiken und sonstige Hilfsdokumente, also nicht nur der reine Quellcode. Das Erzeugen (engl. packaging) eines Release fällt je nach Autor ebenso in diese Phase.

#### ■ *Test*

In der Testphase wird geprüft, ob die Software den im Pflichtenheft dargestellten Anforderungen entspricht und daher als fehlerfrei be-

zeichnet werden kann<sup>2</sup>. Die Tests können sowohl von der entwickelnden Gruppe, vom Kunden, als auch von einem hierzu beauftragten Dritten durchgeführt werden.

#### ■ *Wartung*

In der Wartungsphase ist die Software beim Anwender in Betrieb. Es werden währenddessen gefundene Fehler behoben und neu hinzukommende Anforderungen umgesetzt. Dies ist üblicherweise die längste Phase im Lebenszyklus.

### 3.2.2 Spiralmodell

Das *Spiralmodell* ist ein von Boehm 1988 [Boe88] entwickeltes iteratives, generisches Vorgehensmodell zur Software-Entwicklung. Es sollte vor allem die damals bekannten Schwächen des Wasserfallmodells beheben. Abbildung 3.5 zeigt den Ablauf in der von Boehm vorgeschlagenen Darstellung [Dei03]. Dabei wird ein zweidimensionaler Ansatz verwendet: Zum einen die radiale Dimension, in der die kumulativen Kosten, welche die einzelnen Schritte nach sich ziehen, dargestellt werden, und zum anderen die winkelförmige Dimension, welche den erzielten Fortschritt beim Beenden jedes Umlaufs der Spirale darstellt.

Jede Iteration ist also ein Umlauf um die Spirale, und wird in vier Phasen eingeteilt: Begonnen wird mit der Zielbestimmung der Phase, Schaffen der dazu nötigen Voraussetzungen, Aufnahme der Randbedingungen inklusive Umfeldanalyse, Systemanalyse, Entwicklung und Beschreibung von Lösungsalternativen. Die zweite Phase widmet sich dem Risikomanagement. Es werden die Lösungsalternativen nach Risikogesichtspunkten bewertet und die risikominimale Lösung ausgewählt. Dazu wird möglicherweise bereits ein Prototyp der Software entwickelt. In der dritten Phase wird die ausgewählte Lösungsalternative konkret umgesetzt. Das Spiralmodell ist recht generisch und spezifiziert nicht, wie hierbei vorgegangen wird. In der letzten Phase werden schließlich die vorangegangenen Phasen bewertet, und

---

<sup>2</sup>Fehlerfrei wird Software nie sein, es gilt hier lediglich der Anforderung zu entsprechen.

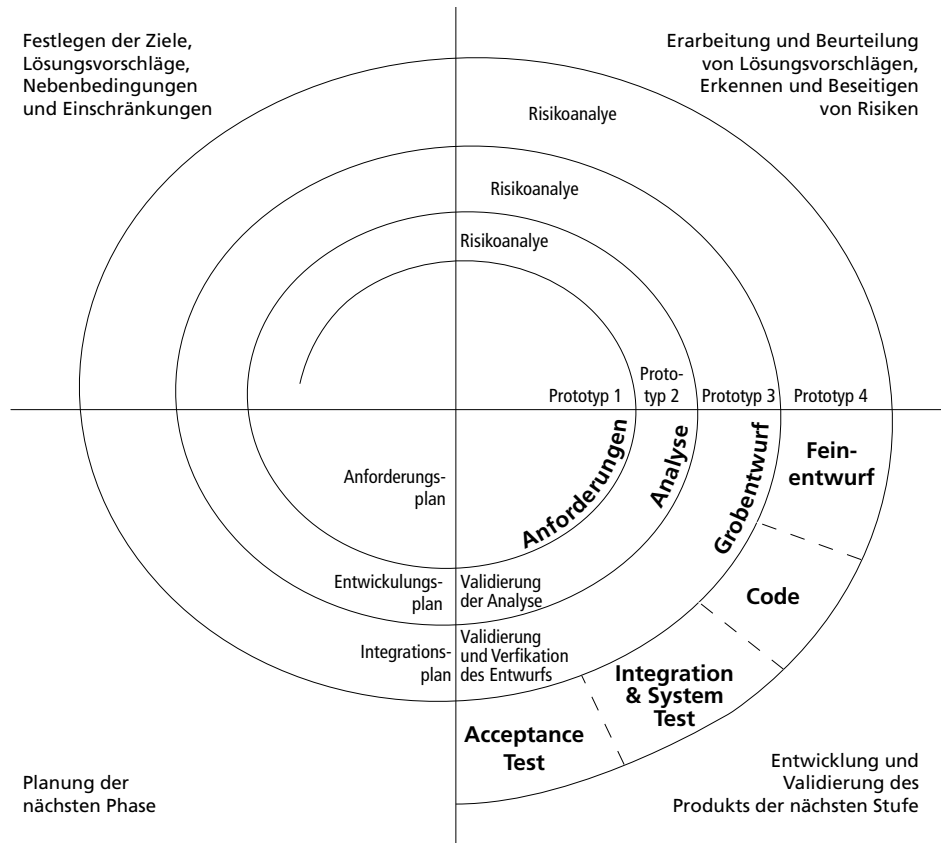


Abbildung 3.5: Das Spiralmodell.

die nächste Iteration geplant. Dazu wird die Einhaltung der festgelegten Einschränkungen, der Projektfortschritt sowie die Qualität überwacht.

Das Spiralmodell hat inzwischen verschiedene Weiterentwicklungen erfahren. Die Erste Abwandlung unter Einflüssen aus der Spieltheorie [BR89] führte zum *WinWin-Spiralmodell*. Hierbei definieren die Stakeholder<sup>3</sup> ihre Anforderungen anhand von *Win Conditions* und verhandeln diese über den gesamten Projektfortschritt. Es wird also die Interaktion der Stakeholder mehr in den Vordergrund gestellt. Boehm selber hat hiermit gute Erfahrungen gemacht: Die permanente Diskussion erhöhte die Flexibilität zur Reaktion auf Risiken und Unsicherheiten, die Iterationen erzeugten eine Disziplin zur Zieleinhaltung und die Stakeholder hatten ein höheres Vertrauen untereinander. Dennoch entstand ein hoher Wasserkopf durch die Dokumentenverwaltung aus mehreren Sichtweisen. Ebenso berichtet

<sup>3</sup>Stakeholder sind alle, die Interessen irgendeiner Art an dem Projekt haben.

er von anderen Einsatzfällen des Spiralmodells, allerdings ohne detaillierte Ergebnisse: C-Bridge's RAPID process, AT&T/Lucent/Telcordia, Xerox [BH00]. Alte Berichte über den Einsatz, die nicht ohne weiteres auf heutige Projekte übertragen werden können, finden sich bei [Boe88]. Auch der Unified Process basiert in Teilen auf dem Spiralmodell. Die NASA verwendet eine eigene Weiterentwicklung (TReK) und hat damit Erfolge erzielt, allerdings sind die Projekte noch nicht abgeschlossen [HS02].

### 3.2.3 Rational Unified Process

Der *Rational Unified Process* RUP ist ein von der Firma Rational (gehört inzwischen zum IBM Konzern) vertriebenes generisches Vorgehensmodell. Es handelt sich also um ein konkretes Produkt, das eigentlich zugrundeliegende Modell ist der Unified Process, der beispielsweise auch dem *Open Unified Process* OpenUP zugrunde liegt. Letzterer wird von der Eclipse Foundation als Open Source Produkt entwickelt [BaloJ, OpeoJc]. Auf den OpenUP wird hier nicht weiter eingegangen, da seine praktische Relevanz zur Zeit noch recht begrenzt ist. Ebenso verhält es sich mit den anderen Ausprägungen Agile Unified Process, Enterprise Unified Process und Essential Unified Process.

Als sehr generisches und umfangreiches Prozessmodell muss der RUP an die konkreten Projekte angepasst werden. Sowohl diese Anpassung (engl. tailoring) als auch die eigentliche Software-Entwicklung sind stark werkzeugunterstützt. Wegen der Komplexität nimmt die Anpassung dennoch im Idealfall ein sogenannter RUP Process Expert vor, der häufig auch die Entwickler schult. Es wird sehr stark auf UML und HTML zu Dokumentationszwecken gesetzt, Anforderungen werden mit Hilfe von Use Cases beschrieben. Die Entwicklung an sich ist eine Weiterentwicklung des Spiralmodells und wird als evolutionäres Prototyping bezeichnet: Es werden häufig Prototypen der Software, sogenannte Pilotsysteme, erzeugt, anhand derer der Projektfortschritt bewertet werden kann. So wird versucht, Risiken durch falsche Entwicklung zu minimieren.

Der RUP setzt, wie in Abbildung 3.6 dargestellt, auf einen zweidimensionalen Ansatz, um statische und dynamische Aspekte der Software-Entwicklung zu vereinen [IBM0J].

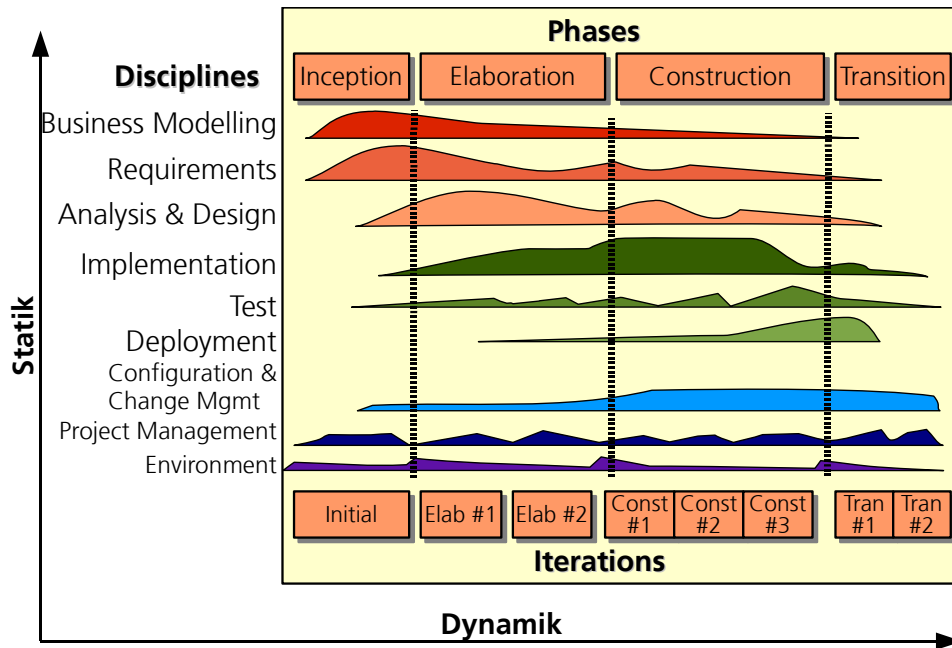


Abbildung 3.6: Der IBM Rational Unified Process.

Die Statik beschreibt die einzelnen Arbeitsschritte, welche in jeder Iteration mehr oder weniger intensiv durchgeführt werden: *Geschäftsprozessmodellierung* (engl. Business Modeling), *Anforderungsanalyse* (engl. Requirements), *Analyse & Design*, *Implementierung*, *Test* und *Auslieferung* (engl. Deployment) als Hauptarbeitsschritte, sowie Konfigurations- und Änderungsmanagement (engl. Configuration & Change Management), Projektmanagement und Umfeld (engl. Environment) als unterstützende Tätigkeiten. RUP versucht die Steifigkeit des Wasserfallmodells durch mehrere Iterationen je Phase zu lösen. Ein Zurückspringen ist dennoch nicht vorgesehen und soll unter anderem durch eine detaillierte Beschreibung der notwendigen Kriterien zum Meilenstein vermieden werden. Die Dynamik teilt die Iterationen in vier Phasen auf: Konzeption (engl. Inception), Entwurf (engl. Elaboration), Konstruktion (engl. Construction) und Übergabe (engl. Transition).

Je nach Phase ist die Arbeitslast in den Arbeitsschritten wie in Abbildung 3.6 dargestellt verschieden verteilt. In der Theorie benötigt die Konstruktionsphase dabei die meisten Ressourcen und die meiste Zeit, Konzeption und Übergabe benötigen am wenigsten. Der Entwurf liegt dazwischen. Am Ende jeder Phase sollte ein Meilenstein erreicht werden. Ist dieser nicht erreicht, muss die Phase wiederholt werden, wobei die Abläufe so anzupassen sind, dass der Meilenstein diesmal erreicht wird. Alternativ kann das Projekt

natürlich abgebrochen werden. Die Ergebnisse der Phasen sind denen aus Wasserfall- oder Spiralmodell ähnlich. Dennoch sei darauf hingewiesen, dass sie bei weitem weniger strikt sind als in den klassischen Modellen. Gewisse Unplanbarkeiten und Ungenauigkeiten sind bewusst vorgesehen:

#### ■ *Konzeption*

Die Konzeption entspricht in etwa der Anforderungserhebung des Wasserfallmodells, geht aber über diese hinaus. Zentrales Ergebnis ist die Vision des Projektes. Darin werden unter anderem die zentralen Erfolgskriterien definiert. Außerdem werden viele für die Projektüberwachung relevanten Dokumente erzeugt, die nachher permanent verfeinert werden können. Dazu zählt ein rudimentäres Modell des Anwendungsfalls, das die wesentliche Funktionalität beschreibt. Dafür wird versucht, alle Use Cases zu identifizieren, wobei die wichtigsten auch detaillierter ausgearbeitet werden. Vor allem bei größeren Projekten wird daraus ein Begriffslexikon abgeleitet. Des Weiteren werden sowohl die benötigten Ressourcen abgeschätzt, als auch der erwartete Gewinn bewertet. Eingeschlossen wird auch eine Analyse des Umfeldes, sowie die Identifikation und Bewertung der größten Risiken für das Projekt. Auf technischer Seite wird eine provisorische Architektur festgelegt.

#### ■ *Entwurf*

In der Entwurfsphase wird zum einen das Use Case Modell vervollständigt, wobei ein Abdeckungsgrad von 80 Prozent erreicht werden soll. Zum anderen wird auch die Architektur verfeinert, so dass am Ende ein Architekturprototyp entsteht, der alle für die Architektur relevanten Use Cases enthält. Die Phase entspricht etwa der Software-Design Phase des Wasserfallmodells. Die bereits in der Konzeptionsphase erzeugten Controlling Dokumente werden permanent weiter verfeinert. Alle wesentlichen technischen Risiken müssen in der Entwurfsphase minimiert werden, da sie sonst anschließend deutlich mehr Kosten verursachen würden.

#### ■ *Konstruktion*

Die Konstruktionsphase entspricht der Implementierungs- und Testphase des Wasserfallmodells. Es werden die Komponenten erzeugt, und die Use Cases nach und nach umgesetzt. Hier werden für gewöhnlich

die meisten Iterationsschritte notwendig. Am Ende ist das Produkt in einem Beta-Stadium, die erste Endbenutzerdokumentation liegt vor.

#### ■ *Übergabe*

Die Übergabephase hat keine direkte Entsprechung im Wasserfallmodell. Zwar ist auch Wartung in dieser Phase eingeschlossen, aber der Fokus liegt auf der Erzeugung eines Releases und der Übergabe an den Endanwender. Dazu müssen sowohl Endanwender als auch ein etwaiges Wartungsteam geschult werden. Außerdem wird eine abschließende Qualitätssicherung vorgenommen.

RUP erreicht, unterstützt von anderen ergänzenden Werkzeugen von Rational, CMMI Stufe 2. Dazu stellt Rational umfangreiche Dokumentation zur Verfügung [Rei03]. Auch CMM (Capability Maturity Model, der Vorläufer des CMMI) Stufe 3 ist mit Erweiterungen möglich [MP03]. Nach Aussagen von Rational kann RUP auch verwendet werden, um Legacy-Systeme weiter zu pflegen. So haben 80 Prozent der Kunden auch Legacy-Code zu verwalten [Kru01]. Die Adaption von RUP ist kein trivialer Prozess, vor allem wenn die Entwickler gewohnt sind, statisch zu entwickeln [LKB01]. Selbst erfahrene IT-Entwickler haben Schwierigkeiten, das Konzept von RUP anzuwenden und RUP nicht als bloße Werkzeugsammlung zu benutzen [MK02]. Viele vermuten, dass dieses Problem auch in der Vertragsgestaltung bei der Software-Entwicklung zu suchen ist: Zu feste Vorgaben in den Verträgen verhindern die notwendige Dynamik für eine wirkliche Adaption der iterativen oder inkrementellen Prozessmodelle [Lee06]. Mutschnig-Pitrik hat sehr gute Ergebnisse mit der RUP Einführung gemacht, weist aber ausdrücklich darauf hin, dass dies nur mit hinreichend geschulten Mitarbeitern und ohne starken Zeitdruck möglich war [MP02]. Andererseits ist RUP durchaus auch für kleinere Projekte erfolgreich einsetzbar, wenn initiale Anpassungsarbeiten erledigt und alle unnötigen Artefakte entfernt wurden [Hir02].

### 3.2.4 V-Modell (XT)

Die Grundidee zu einem V-förmigen Vorgehen bei der Software-Entwicklung hatte Barry Boehm bereits 1979 [Boe79]. Das Bundesverteidigungsministerium griff die Idee 1986 auf und lässt seitdem das V-Modell für den

behördlichen Einsatz in Deutschland entwickeln. Zur Zeit wird es von der Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung (KBSt) gepflegt. Das V-Modell wird permanent an neue Anforderungen in der Software-Entwicklung angepasst, die letzte Aktualisierung fand am 1. Februar 2006 auf Version 1.2 [Koo06] des *V-Modells XT* statt. Die XT (für *eXtreme Tailoring* = extreme Anpassbarkeit) Variante wurde eingeführt, um agilere Prozesse zuzulassen [BR05]. So steht mehr das entstehende Produkt im Mittelpunkt und nicht so sehr der Weg dorthin. Es werden mehrere mögliche Vorgehensmodelle beschrieben, wobei das Vorgehen nach Arbeitspaketen gegliedert ist, deren zeitliche Abfolge nicht immer vorgeschrieben ist. Dennoch kann man eine Abfolge festlegen, und damit das Modell beispielsweise auf das Wasserfallmodell abbilden. Außerdem können für kleinere Projekte Teile komplett weggelassen werden, um keinen übermäßigen Wasserkopf zu erzeugen.

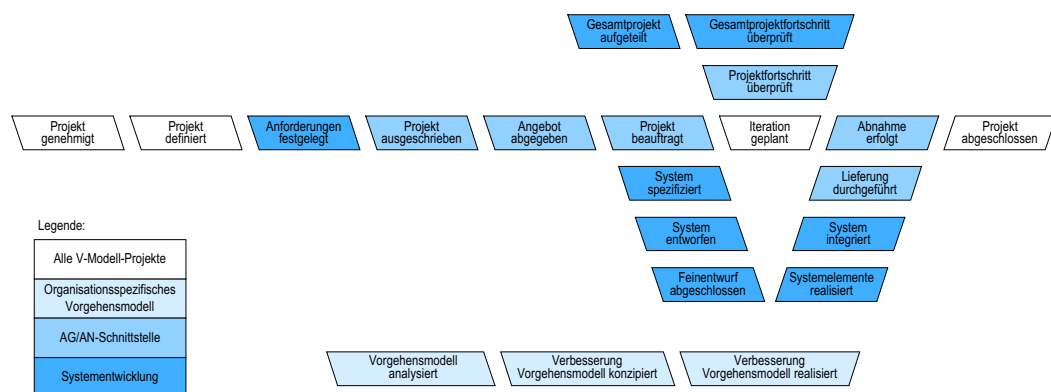


Abbildung 3.7: Das V-Modell XT 1.2 im Überblick.

Eine Übersicht über die Arbeitspakete findet sich in Abbildung 3.7, Details zum Ablauf in Abbildung 3.8<sup>4</sup>.

Das V-Modell XT trennt, im Gegensatz zu CMMI, strikt zwischen Auftraggeber und Auftragnehmerprojekten. Daher verteilen sich die Aktivitäten zwischen den beiden Parteien auf zwei Projekte. Die im Lastenheft festzuhaltenden Anforderungen werden im V-Modell vom Auftraggeber erzeugt, der Auftragnehmer leitet daraus seine technische Sicht ab, ergänzt diese

<sup>4</sup>Beide Abbildungen nach den V-Modell XT Quellen [Koo06].

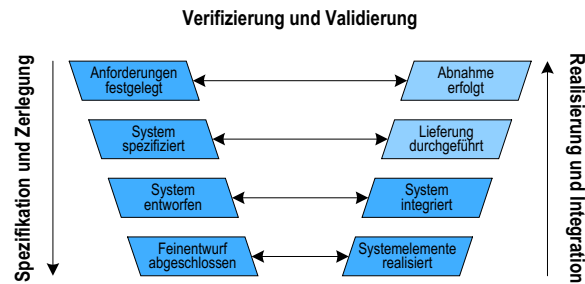


Abbildung 3.8: Ablaufdetails V-Modell XT.

eventuell durch weitere Anforderungen, die aus seiner Organisation erwachsen, und erzeugt daraus das Pflichtenheft als Gesamtsystemspezifikation. Eine einfache Bewertung welche der CMMI Stufen erreicht wird, ist somit nicht immer möglich.

Das V-Modell findet breiten Einsatz in der Praxis. Zunächst ist es grundsätzlich für alle deutschen IT-Projekte des Bundes vorgeschrieben und darf nur in begründeten Ausnahmefällen nicht eingesetzt werden. Außerdem ist bekannt, dass die Firmen Siemens, EADS, 4Soft, IABG, ESG, Witt-Weide und Funkwerk Kölleda das V-Modell (XT) verwenden [Koo07]. T-Systems führt auf breiter Basis (CMMI Level 3) das V-Modell ein und hat bisher positive Erfahrungen gemacht, wenn auch die Umstellung als sehr komplexer Prozess beschrieben wird [Mar07]. Das Fraunhofer Institut für Experimentelles Software Engineering (Fraunhofer IESE)<sup>5</sup> hat festgestellt, dass aufgrund des Umfangs des V-Modells die Adaption schwer fällt. Daher wurde Spearmint<sup>TM</sup> [BKBM<sup>+</sup>03]<sup>6</sup>, ein Tool zur Erzeugung einer webbasierten Prozessunterstützung, entworfen. Damit fällt die Adaption deutlich leichter [BKV99]. Kuhrmann, Niebuhr und Rausch haben gemischte Erfahrungen mit dem V-Modell gemacht. Neben den bereits erwähnten Anlaufschwierigkeiten waren vor allem im Modell nicht vorgesehene Abläufe problematisch, die von außen vorgegeben waren, aber dann keine Werkzeugunterstützung hatten [KNR06]. Das Tailoring des V-Modell 97 wird als schwieriges und fehlerbehaftetes Vorgehen beschrieben [MSV97]. Nach Aussagen der V-Modell Entwickler zeigte sich bereits in der Beta-Phase der Entwicklung der XT-Variante hier deutliche Besserung, allerdings ist ein Tailoring ohne Schulung

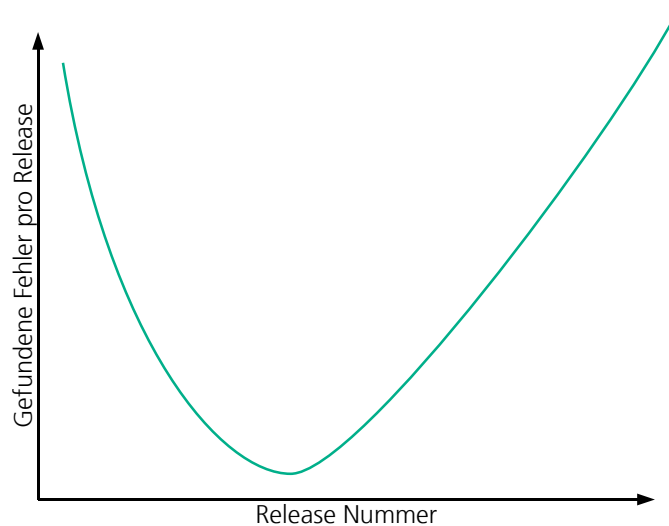
<sup>5</sup>[www.iese.fraunhofer.de](http://www.iese.fraunhofer.de)

<sup>6</sup>Spearmint = Software Process Elicitation, Analysis, Review, and Measurement in an Integrated Environment is a registered trademark of Fraunhofer IESE.

immer noch nicht sinnvoll möglich [KNR06]. Außerhalb Deutschlands ist das V-Modell eher von geringerer Bedeutung, es gibt wenig Berichte über den Einsatz.

### 3.3 Agile Methoden

Neuere Ansätze zur Software-Entwicklung entstanden aus den Erfahrungen beim Einsatz des Wasserfallmodells und anderer starrer Modelle: Die Anforderungsspezifikationen können häufig nicht zu Beginn vollständig erfasst werden und/oder ändern sich während der Entwicklung. Dabei wird versucht, nicht zuviel Aufwand in die Erhebung der Anforderungen zu investieren. Auch andere Prozessmängel werden versucht zu beheben. So wurde erkannt, dass mit fortschreitender Entwicklung die Zahl der gefundenen Fehler je Release in der Regel immer steigt. Dies ist zwar ein fundamentales Problem der Software-Entwicklung, das nicht verhindert werden kann [Ber02], dennoch wird versucht, neue Methoden zu finden,



**Abbildung 3.9: Belady-Lehman Graph: Permanente Änderungen erhöhen Fehlerzahl.**

die diesen Effekt wenigstens abmildern. Es hat sich der Begriff der agilen Software-Entwicklung eingebürgert. Dieser wurde nie formal definiert, nach einer umfassenden Betrachtung der als agil bezeichneten Prozesse kommen Abrahamsson et al. [ASRW02] aber dazu, sie wie folgt zu charakterisieren:

*Agile Software-Entwicklung* ist inkrementell (kleine Releases in schnellen Zyklen), kooperativ (Kunde und Entwickler arbeiten eng zusammen und kommunizieren permanent), geradlinig (die Entwicklungsmethode ist einfach zu lernen und zu verändern, gut dokumentiert) und anpassungsfähig (um auch in letzter Minute noch Änderungen unterzubringen). Diese Definition ist absichtlich recht grob gefasst, da es viele verschiedene Ausprägungen der agilen Software-Entwicklung gibt. Auf einige wird im Folgenden eingegangen.

Die Zahl der Unternehmen, die agile Prozesse einsetzen, wächst ständig. Im Jahr 2004 ergab eine Umfrage von Forrester, dass zwar noch 60 Prozent Wasserfallmodelle einsetzen, aber 63 Prozent der Befragten bereits ein Prozessoptimierungsprogramm gestartet haben, und bereits 18 Prozent agile Methoden einsetzen [Bar05]. Eine detaillierte Übersicht über aktuelle agile Methoden liefern Abrahamsson et al. [ASRW02] und Zagrodnick [Zag05].

### 3.3.1 eXtreme Programming

*eXtreme Programming* (XP) ist die bekannteste agile Software-Entwicklungsmethode. Entworfen wurde sie 1999 von Beck [Bec99] bei Chrysler und dort auch erfolgreich eingesetzt (auch wenn das Projekt zwischenzeitlich eingestellt wurde) [ABB<sup>+</sup>98]. Sie ist ausgelegt für kleine Teams und sich permanent ändernde Anforderungen. Beck hatte erkannt: »Anforderungen sind am Anfang nie klar. Kunden können nie genau sagen, was sie wollen. Die Entwicklung einer Software ändert ihre eigene Anforderung. Sobald die Kunden das erste Release sehen, lernen sie, was sie in der zweiten haben wollen [...] oder was sie in der ersten wirklich wollten.« [Bec99].

XP ist zunächst eine Sammlung von schon länger bekannten Best-Practices, die aber extrem ausgelebt werden. Außerdem will XP fünf Werte fördern, die für eine erfolgreiche Software-Entwicklung unabdingbar sind: Kommunikation, Einfachheit, Feedback, Mut und – seit der zweiten Auflage – Respekt.

Während klassische Ansätze die zweifellos nötige *Kommunikation* unter den Entwicklern mittels ausreichender Dokumentation erledigen wollen, misst XP diesem Punkt deutlich mehr Bedeutung zu. Praktiken wie Pair Programming,

Metapher, Feedback und die starke Einbindung der Kunden (s.u.) sollen sehr schnell ein gemeinsames Verständnis des Problemumfeldes erzeugen.

Es wird extremer Wert auf *einfache* Lösungen gelegt. Der Entwickler soll immer mit der einfachsten Lösung beginnen, selbst wenn schon absehbar ist, dass diese nicht auf ewig erhalten bleiben kann. Da angenommen wird, dass sich die Anforderungen sowieso ändern, ist dies nicht schlimm, zusätzliche Funktionalität kann später hinzugefügt werden. Außerdem soll so die Qualität der Software steigen und die Kommunikation erleichtert werden.

*Feedback* kommt von verschiedenen Orten: Durch Unit- und Integrations-tests haben die Programmierer nach der Implementierung von Änderungen permanentes Feedback vom System. Akzeptanztests der Kunden liefern stets neue Informationen über die Lage der Anforderungen. Vom Team selber kommt Feedback durch die Anforderungsanalyse im sogenannten Planning Game und im morgendlichen Stand-Up Meeting.

Verschiedene Praktiken verlangen und bestärken *Mut*. So braucht es beispielsweise Mut, das Design nur für die Anforderungen von heute zu machen und nicht auch für die, die morgen vielleicht dazu kommen. Dieser Mut ermöglicht es auch, alten Code zu refactoren oder gar zu löschen, egal wieviel Arbeit hineingesteckt wurde.

XP verlangt von den Teammitgliedern aus mehreren Gründen *Respekt* untereinander. Zum einen soll niemand durch seine Änderungen den Gesamtfortschritt verzögern, also beispielsweise eine erfolgreiche Kompilierung verhindern, Unittests zum Scheitern bringen o.ä.. Zum anderen soll die Arbeit des Anderen zwar respektiert werden, dennoch aber nur die Lösung an sich im Mittelpunkt stehen, nicht der Autor.

Die bereits angesprochenen Best-Practices sollen helfen, diese Werte zu fördern. Im Einzelnen sind dies:

#### ■ *Pair Programming*

Pair Programming ist sicher die bekannteste Eigenschaft des XP, wiewohl es keine Erfindung des selben ist und schon vorher bekannt war. Es wird verlangt, dass immer zwei Menschen gleichzeitig vor einem Rechner sitzen und entwickeln. Während der eine das Gerät bedient und die manuelle Arbeit erledigt, denkt der andere quer und

unterzieht die erzeugte Software damit einem permanenten Review. Die Rollen werden regelmäßig getauscht, auch die Zusammenstellung der Paare soll häufig wechseln, damit viel Kommunikation entsteht, und jeder über möglichst viele Aspekte Bescheid weiß. Außerdem sollen die Entwickler so voneinander lernen.

#### ■ *User Stories*

Anforderungen der Kunden werden als User Stories festgehalten. Dies sind kurze, d.h. nicht mehr als eine Karteikarte umfassende, Spezifikationen der Anforderungen, welche erst bei Bedarf im Gespräch verfeinert werden. Der Kunde muss Akzeptanztests durchführen können, um ihre Einhaltung zu prüfen.

#### ■ *Test Driven Development*

XP fordert das möglichst umfassende Testen der Software, auch schon vor ihrer Entwicklung. Die Entwickler sollen also zuerst den (Unit-)Test schreiben und danach die zugehörige Implementierung. Dies soll anregen, bereits früh über mögliche Problem- und Randfälle nachzudenken.

#### ■ *Continuous Integration*

Continuous Integration soll Probleme durch Versionsunterschiede vermeiden und schnelle Zyklen ermöglichen. Die Teams sollen alle paar Stunden ihre Änderungen hochladen. Das schließt natürlich Integrationstests ein, die genau wie die Unit Tests möglichst werkzeugunterstützt sein müssen.

#### ■ *Simple Design*

Beim Entwurf wird ein möglichst einfaches Design gefordert. Es soll immer die minimal komplexe Möglichkeit gewählt werden, eine Anforderung zu implementieren. Weitergehende Funktionalität kann später noch eingefügt werden.

#### ■ *Design Improvement*

Damit die Software-Qualität nicht unter den permanenten Änderungen leidet, wird gefordert, das Design permanent zu verbessern, sobald eine unsaubere oder ungeschickte Lösung entdeckt wird. Hierzu hat sich der Begriff des *Code Smell* eingebürgert: Der Program-

mierer soll geradezu riechen, wenn etwas nicht elegant, einfach oder sonstwie geschickt gelöst ist, und es durch Refactoring verbessern.

#### ■ *System Metaphor*

Um das gemeinsame Verständnis zu fördern, wird eine Metapher beschrieben. Sie gibt ein Rahmenwerk für Namen von Klassen und Methoden, die ihren Zweck einfacher verständlich machen sollen. Dies reduziert auch den Overhead für die Dokumentation. Die Metapher wird sowohl für die Kommunikation mit den Kunden, als auch für das Design der Software verwendet.

#### ■ *Coding Standard*

Um den Dokumentationsaufwand weiter zu reduzieren und die Kommunikation zu beschleunigen, einigt man sich auf Coding Standards: Dadurch, dass sich alle Programmierer auf einen Regelsatz einigen, ist es leichter, Quelltext, der von anderen geschrieben wurde, zu verstehen.

#### ■ *Collective Code Ownership*

XP verlangt von jedem, sich für den gesamten Quelltext verantwortlich zu fühlen. Damit soll ein jeder ermutigt werden, beliebige Teile zu ändern. Um zu vermeiden, dass sich durch mangelndes Verständnis bei einer Änderung von fremdem Quelltext Fehler einschleichen, müssen die Unit Tests diese hinreichend gut abdecken.

#### ■ *Sustainable Pace*

Hauptsächlich außerhalb Europas ist eine gleichmäßige Entwicklungsgeschwindigkeit ein betonenswerter Aspekt. Die Programmierer sollen nicht mehr als 40 Stunden pro Woche arbeiten, da sonst die Qualität der Software leidet. In Europa sind höhere Wochenstundenzahlen nicht sonderlich verbreitet.

#### ■ *Small Releases*

Um ein schnelles Feedback zu erhalten und den Projektfortschritt messen zu können, sind häufige kleine Releases vorgesehen. Diese sind lediglich von Alpha Qualität, müssen also keineswegs fehlerfrei oder funktionsvollständig sein.

#### ■ *On-Site Customer*

Der Kunde entsendet (mindestens) einen Mitarbeiter, der permanent für Fragen zur Verfügung steht. Er nimmt an allen Sitzungen teil, fällt aber keine technischen Entscheidungen. Die Entwickler hingegen fällen keine fachlichen Entscheidungen. Dies soll das Erkennen von Mängeln beschleunigen und neue Anforderungen möglichst früh erkennen helfen.

#### ■ *Planning Game*

Die Iterationsplanung findet in einem Planning Game statt. Dazu weist der Kunde den User Stories Prioritäten zu, um festzulegen, welche davon in welcher Iteration abgeschlossen werden sein sollen. Er unterteilt hier nach Kritischem, Wichtigem aber nicht zwingend Notwendigem und Nützlichem. Gleichzeitig schätzen die Entwickler den Aufwand je User Story. Ist dies nicht hinreichend genau möglich, muss die User Story verschoben, oder in mehrere Teile aufgeteilt werden. Nachdem der Kunde den *Business Value* geschätzt hat, schätzen die Entwickler das (technische) Risiko. Je risikoreicher eine Story ist, desto früher sollte sie erledigt werden. Es wird versucht, die beiden Prioritäten im Dialog in Einklang zu bringen. Anschließend werden Teams gebildet, denen die Aufgaben zugeordnet werden.

#### ■ *Stand-Up Meeting*

Der Prozessüberwachung dient auch ein allmorgendliches Zusammenkommen, wo in einer Viertelstunde jeder Entwickler über seinen aktuellen Status berichtet. Um den zeitlichen Rahmen einzuhalten, werden fachliche Aspekte hierbei bewusst nicht geklärt.

#### ■ *Spikes*

Spikes werden verwendet, um Unsicherheiten zu klären. Es sind Wegwerfprototypen, die dazu dienen, einen einzelnen technischen Aspekt zu klären.

#### ■ *Tuning Workshop*

Zur Prozessverbesserung (siehe CMMI Stufe 5(!)) dient ein Workshop nach jeder Iteration. Dieser soll helfen, den Entwicklungsprozess zu verbessern. Es wird hier allerdings nicht auf festgelegte Metriken zurückgegriffen, auch ist das Vorgehen im Detail nicht spezifiziert.

Einen wissenschaftlichen Nachweis, dass Pair Programming Vorteile mit sich bringt, liefern Williams et al. [WKCJ00]: Bei ihren Messungen waren Teams 40 Prozent schneller als einzelne Entwickler, brauchten dabei nur 60 Prozent mehr Zeit, produzierten aber vor allem Code, der über 10 Prozentpunkte mehr Akzeptanztests bestand.

Der erste Erfolgsbericht über den Einsatz von XP (damals noch nicht unter diesem Namen bekannt) stammt von Cunningham [Cun92], es folgen Anderson [ABB<sup>+</sup>98] und Grenning [Gre01]. Dornberger liefert neben einer sehr detaillierten Übersicht über XP auch einen umfassenden Überblick über den Einsatz bei größeren Unternehmen [DH04a]: So setzte es die Ford Motor Company erfolgreich für ein Projekt mit sieben Mitarbeitern ein. Die Technische Universität München hat im Rahmen einer Studie 45 XP-Projekte betrachtet. Dabei wurden 98 Prozent als erfolgreich bewertet, und wichtige Projektkriterien wie Termintreue wurden eingehalten. Dennoch wurden verschiedene Forderungen von XP nicht immer umgesetzt: 40 Prozent verwenden Metapher nicht, 30 Prozent hatten mit dem On-Site Customer Schwierigkeiten. Dieselbe Metapher für die Kommunikation mit den Kunden und das Software-Design zu verwenden, kann zu Problemen führen [LL07]. Simons beweist, dass die Qualitätssicherung mit Hilfe von Regressionstests bei XP überbewertet wird: Durch Weiterentwicklungen und Veränderungen an Modulen werden die Regressionstests zunehmend schlechter und müssten eigentlich permanent neu entwickelt werden. Andernfalls sinkt ihre Abdeckung und damit ihre Sicherheit [Sim05].

### 3.3.2 Evolutionary Development

Die Evolutionäre Software-Entwicklung (kurz »Evo« genannt) wurde von Tom Gilb um 1980 benannt [Gil81, Gil88], eine weitere Einführung liefert [CDJSoJ]. Evo kann durchaus als Vorläufer vieler bekannter agiler Methoden betrachtet werden. Bei Evolutionärer Entwicklung wird die Software in sehr vielen kleinen, inkrementellen Schritten entwickelt, welche jeweils direkt an den Kunden ausgeliefert werden (Evolutionary Delivery). Dabei ist keine feste Richtung vorgegeben, diese ändert sich laufend während der Entwicklung anhand der Rückmeldungen der Anwender.

Abbildung 3.10 (nach [Mal06]) zeigt den Unterschied zur inkrementellen- und Wasserfall-Entwicklung: Da die Zielvorstellungen stetig konkreter wer-

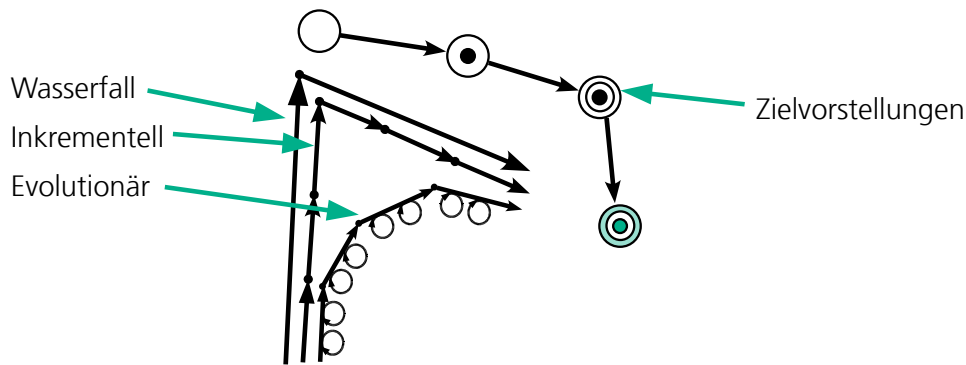


Abbildung 3.10: Wasserfallmodell, inkrementelle und evolutionäre Entwicklung.

den, wird auch die Zielvorgabe des Software-Entwicklungsprojektes an die sich ändernden Bedürfnisse der Kunden angepasst, und die Richtung der Entwicklung folgt sofort. Eine intensive und durchgehende Kundeneinbindung ist daher ebenso Kernbestandteil von Evo wie stetige Prozessverbesserung und die Fokussierung auf das Produkt statt auf den Prozess.

Gilb fordert für jedes Release, dass die funktionalen Komponenten, Qualitätsstufen, Tests, Spezifikation, Benutzerdokumentation und Hardwareanforderungen gemäß der Planung vollständig abgeschlossen sind. Wie das zu erreichen ist, oder was passieren soll, wenn ein Release dieser Anforderung einmal nicht genügt, wird nicht spezifiziert.

In diesem Umfeld sei auch noch das Modell der Evolutionären Objektorientierten Software-Entwicklung (EOS-Modell) nach Hesse beschrieben [Hes97]. Das Modell versucht die Eigenschaften der objektorientierten Software-Entwicklung im Prozessmodell abzubilden. Daher wird auch der Prozess komponentenweise ausgelegt, und hierarchisch nach einer Art Vererbung strukturiert. Die Entwicklung besteht aus vielen Zyklen, die wiederum in die Aktivitäten Analyse, Design, Implementierung und Operativer Einsatz zerfallen. Leider gibt es keine unabhängigen Berichte über den Einsatz des Modells.

Malotaux geht soweit, Evo in Anlehnung an Brooks »Silver Bullet« [Bro87] als »Magic Bullet« der Software-Entwicklung zu bezeichnen [Mal02]. Er spricht in diesem Zusammenhang von sehr vielen, sehr erfolgreichen Projekten, benennt aber keine haltbaren Quellen. Gilb selber berichtet vom erfolgreichen Einsatz bei IBM, wo bei einem vierjährigen, 200 Mannjahre

umfassenden Projekt alle 45 Auslieferungen im Zeit- und unter dem Budgetplan erfolgten, sowie von einem zehnjährigen NASA Programm, 7000 (!) Mannjahre umfassend, wo dies bei fast allen Auslieferungen erreicht werden konnte. In wie weit dort konkret evolutionäre Methoden eingesetzt wurden, wird nicht berichtet. Umfassende Arbeiten, hauptsächlich zum Einsatz im militärischen Bereich, finden sich bei [Nor03]. Im Experiment von Benediktsson et al. (siehe Seite 14) schneidet Evolutionäre Entwicklung im Mittelfeld ab.

### 3.3.3 Feature Driven Development

*Feature Driven Development* (FDD) rückt, wie der Name bereits andeutet, die einzelnen Features der Software in den Fokus. Ein Feature wird hierbei definiert als: »The features are small ›useful in the eyes of the client‹ results.« (Features sind kleine, in den Augen der Benutzer nützliche, Ergebnisse) [PEJ99] [Gyg03]. Außerdem darf ein Feature nicht mehr als zwei Wochen Entwicklungszeit benötigen. Würde es länger brauchen, müsste es in einzelne Teile zerlegt werden. Die Software erfährt daher spätestens alle zwei Wochen ein neue Release, das den Kunden präsentiert wird, um dann neue Features zu spezifizieren. Der Kunde ist also sehr stark in den Entwicklungsprozess eingebunden. FDD fasst etliche Best-Practices der agilen Software-Entwicklung zusammen und bildet daraus ein eigenständiges Prozessmodell. Der gesamte Ablauf ist in Abbildung 3.11 dargestellt und fünfstufig ausgelegt. Grundsätzlich werden die Teams, die in den einzelnen Phasen zusammenarbeiten, immer erst zu Beginn festgelegt und lösen sich nach Abschluss einer Phase wieder auf.

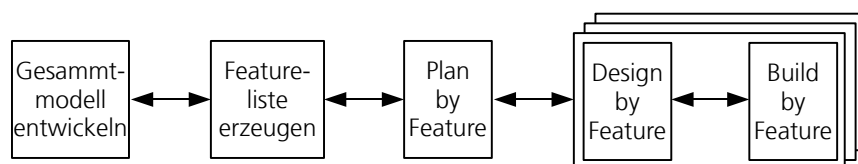


Abbildung 3.11: Feature Driven Development.

Zu Beginn wird versucht einen Überblick über die Fachmaterie zu gewinnen, indem vorhandene Dokumente dazu studiert werden. Dann werden

mittels UML-Diagrammen Teilmodelle erstellt, die anschließend zu einem großen Gesamtmodell zusammengesetzt werden. In der nächsten Phase wird die Liste der zu entwickelnden Features erstellt. Diese sind gruppiert in »Feature Sets« und »Major Feature Sets«, sowie in vier Prioritätsklassen eingeteilt: »must have«, »nice to have«, »add it if we can« und »future«. Hierbei entscheiden die Kunden über die Priorisierung. Außerdem wird geprüft, ob die Entwicklung eines Features die Zwei-Wochen-Frist einhält. Andernfalls wird es zu einem Feature Set zerteilt. In der dritten Phase wird der Entwicklungsablauf geplant. Dazu wird jeder Klasse des Gesamtmodells ein Verantwortlicher zugeteilt, und die Reihenfolge der Entwicklung wird festgelegt. Die beiden letzten Phasen, Design by Feature und Build by Feature, werden für jedes Feature einzeln durchlaufen und bilden so zu jedem Feature eine eigene Iteration. Sie umfassen alle aus dem Wasserfallmodell bekannten Stufen der Software-Entwicklung, die aber für jedes Feature einzeln angewendet werden. Der Entwicklungsstand eines Feature kann anhand definierter Meilensteine gemessen werden, um eine noch genauere Überwachung des Entwicklungsprozesses zu ermöglichen.

Der größte Aufwand bei FDD steckt erwartungsgemäß in den beiden letzten Phasen, die Autoren geben ihn mit 77 Prozent an. FDD ist kein umfassendes Modell und benötigt weitere Unterstützung für das Projektmanagement [ASRW02]. Die Einfachheit des Modells macht eine Adaption leicht möglich: Es ist bereits ausreichend, wenn 20 Prozent der Entwickler Feature Driven Development beherrschen [DL04]. Obschon FDD seit den späten 90er Jahren für große Projekte eingesetzt wird [Pal01], sind Erfahrungsberichte sowohl positiver als auch negativer Natur sehr knapp.

### 3.3.4 Adaptive Software Development

Das *Adaptive Software Development* (ASD) wurde von Highsmith 2000 eingeführt, um auf die sich immer schneller ändernden Anforderungen zu reagieren [Hig00]. Dabei verwendet er ganz bewusst keine neuen Praktiken, sondern bündelt, ähnlich wie beim eXtreme Programming, bestehende zu einem neuen Paket. ASD versucht den Prozess eben so schnell anzupassen, wie sich die Anforderungen ändern. Um sehr generisch zu sein, schreibt ASD keinen Teil als verbindlich und keine Regel als Zwang vor, alles kann weggelassen werden. Highsmith hofft hier auf ein gutes Team, dass diese

Entscheidungen fällt. Man unterscheidet drei Modelle die bei ASD zur Anwendung kommen: Adaptive Conceptual Model, Adaptive Development Model und Adaptive (Leadership-Collaboration) Management Model. Der Lebenszyklus ist in Abbildung 3.12 dargestellt [ASRW02].

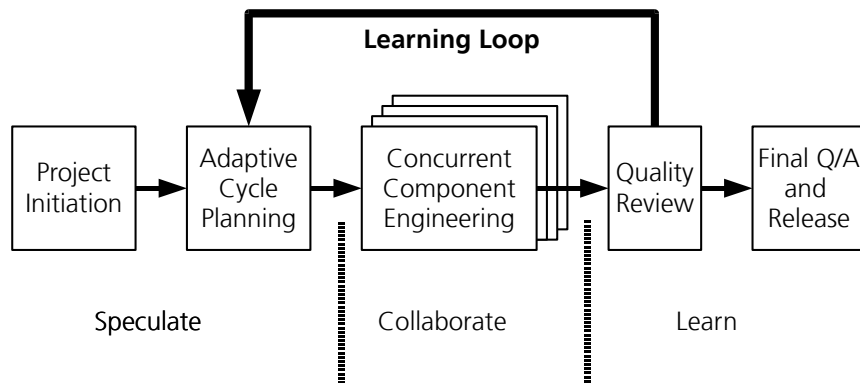


Abbildung 3.12: Der Lebenszyklus im Adaptive Software Development.

Das *Adaptive Conceptual Model* bildet das grundlegende Konzept ab. Hier wird zu Beginn ein recht grober Rahmen des Projektes aufgestellt, der zum einen die Mission enthält, zum anderen die Grenzen des Vorhabens vorgibt. Des Weiteren enthält das *Adaptive Conceptual Model* den eigentlichen Arbeitsablauf, den sogenannten *Adaptive Development Life Cycle*. Die Entwicklung an sich besteht aus mehreren Zyklen, die wiederum aus den Phasen *Spekulieren*, *Zusammenarbeiten* und *Lernen* bestehen. Die Phasen haben hier bewusst andere als die aus dem Wasserfallmodell bekannten Namen. So soll *Spekulieren* ausdrücken, dass während der Planung niemand wirklich sicher weiß, was passieren wird und Änderungen zu erwarten sind. *Zusammenarbeit* betont den Versuch, die Arbeit anhand von kollektiven Vorhersagen oder Adaptionen über Umgebung, Technologie, Anforderungen, Stakeholder etc. zu balancieren. Das *Lernen* will die Teilnehmer zwingen, erarbeitete Erkenntnisse aufzunehmen und zu verwenden sowie gemachte Fehler oder ungenaue Annahmen nicht zu wiederholen.

Im *Adaptive Development Model* wird auf Details der Zyklen eingegangen. So ist vorgeschrieben, dass, ähnlich wie bei FDD, nach jedem Zyklus ein vorführbares Stück Software entsteht. Außerdem hat jeder Zyklus einen festen Zeitrahmen (engl. *time boxing*). Wenn dieser nicht eingehalten werden soll, müssen entweder alle Beteiligten zustimmen, oder es muss bei Umfang,

Budget oder Qualität gespart werden. Um sich hier zu entscheiden, fordert ASD eine in der Projektmission festgelegte Priorisierung.

Das *Adaptive (Leadership-Collaboration) Management Model* wiederum will den TEAM<sup>7</sup>-Gedanken hervorheben. Da Software-Entwickler nach Studien von Highsmith tendenziell Individualbeschäftigung bevorzugen, benötigen sie daher besondere Führung (Leadership), um zusammenzuarbeiten (Collaboration). Deshalb fordert ASD explizit die Kommunikation untereinander, auch die nicht fachliche, um eine Teambildung zu unterstützen. Selbstverständlich soll gerade dieser Gedanke bei der Auswahl der Mitarbeiter im Vordergrund stehen.

Da ASD noch recht jung ist, gibt es bisher keine verwertbaren Erfahrungsberichte. Die Literatur kritisiert regelmäßig [Ebe03a] [Rie01] die Unschärfe von ASD: So besteht eine große Gefahr, dass das Team zu chaotischer, ungeplanter Software-Entwicklung zurückwechselt, da ASD nichts scharf vorschreibt. Auch wird ASD als Altbekanntes, ohne wirkliche Innovation, dargestellt: »Inhalt: 320 Seiten lang alter Wein in neuen Schläuchen. Man nehme: BWL Grundstudium + Bauknecht's IM + ein wenig Trivialpsychologie = Adaptive Software Development« [Ebe03b]

### 3.3.5 Scrum

Auch wenn immer wieder gefordert wird, Software solle nach den Methoden der Ingenieure entwickelt werden, um effizienter und planbarer zu entstehen, bleibt festzuhalten, dass auch die Ingenieure nicht von vorneherein effizient und planbar gearbeitet haben. In diesem Zusammenhang fällt Toyota eine besondere Bedeutung zu. In einer Krisensituation wurde dort zu Beginn der Neunziger Jahre das *Lean Manufacturing* als Toyota Production System entwickelt [WJR91, Gun99]. Toyota revolutionierte damit die bestehenden Produktionsmethoden. Dort wurde soweit irgendwie möglich auf alles verzichtet, was keinen direkten Kundennutzen erzeugt. Alle Abläufe wurden und werden permanent optimiert. Diese Prinzipien werden auch auf die Software-Entwicklung übertragen und sind als *Lean Development*

---

<sup>7</sup>Im Sinne von »Together everyone achieves more« und nicht »Toll ein anderer machts«.

bekannt [MFC05]. *Scrum* (engl. Gedränge) ist ein davon inspirierter, leichtgewichtiger, iterativer und adaptiver Software-Entwicklungsprozess. Abbildung 3.13 gibt eine Übersicht über die Abläufe bei einer Scrum Entwicklung [SB01].

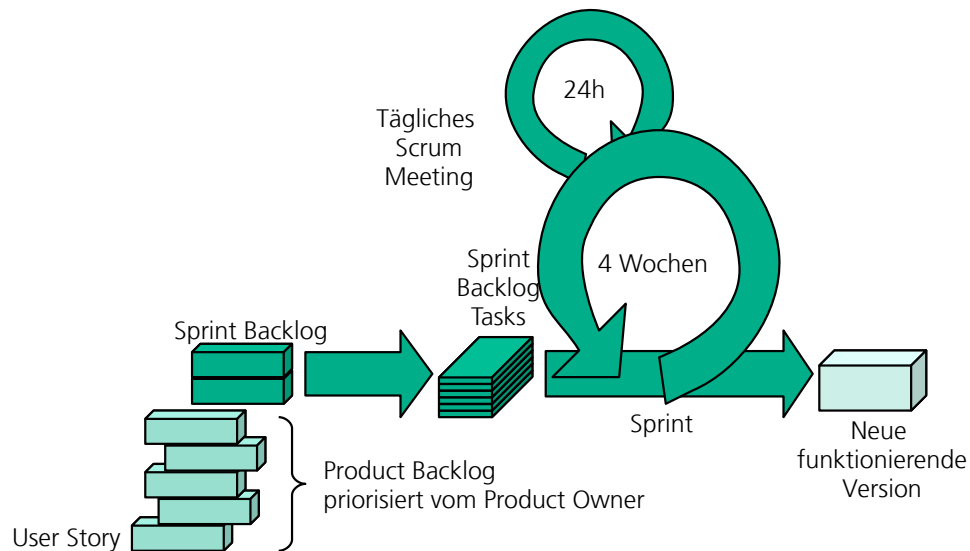


Abbildung 3.13: Der Scrum Entwicklungsprozess im Überblick.

Scrum verwendet für viele aus anderen Bereichen bekannte Dinge eine eigene Terminologie:

- *User Story*  
Aufgaben werden aus Benutzersicht aufgenommen und dann als User Story notiert. Eine Aufgabe darf nicht mehr als sechzehn Stunden Zeit in Anspruch nehmen, sonst wird sie zerteilt (vergleiche FDD).
- *Product Backlog*  
Das Product Backlog enthält alle noch zu erledigenden Aufgaben im Projekt. Es ist langfristig ausgerichtet, die Stories sind häufig noch wenig konkret.
- *Sprint*  
Eine Iteration heißt Sprint und stellt eine Time-Box (vergleiche ASD) für das Scrum Team dar. In der Regel werden zwei bis vier Wochen als Iterationslänge gewählt.

### ■ *Sprint Backlog*

Das Sprint Backlog ist das kurzfristige Pendant zum Product Backlog. Darin werden vom Scrum Team die im nächsten Sprint zu erledigenden Stories gesammelt.

### ■ *Burn Down Chart*

Zur Projektüberwachung dient ein Burn Down Chart, welches den Fortschritt misst. Es stellt eine Metrik der noch zu erledigenden Arbeit dar, und wird hier über die Zeit fortgeschrieben.

Scrum versucht initial nur wenig vorzuschreiben und definiert daher wenige Kernanforderungen. Dies macht eine stufenweise Implementierung möglich, wie sie Tabelle 3.14 zeigt: Je nach Größe des Unternehmens gibt es drei aufeinander aufbauende Stufen der Scrum Implementierung [Sut06].

	A-Flow	B-Pull	C-Innovate
Documents	Product Backlog Sprint Backlog Burndown Chart	Release Burndown Product Backlog Sprint Backlog Burndown Chart	Resource Plan Release Burndown Product Backlog Sprint Backlog Burndown Chart
Ceremonies	Sprint Planning Daily Meeting Sprint Review	Multi-level Planning Daily Meeting Scrum of Scrums Sprint Review	Multi-level Planning Scrum of Scrums Daily Meeting Sprint Review
Roles	Product Owner ScrumMaster Team	Chief Product Owner Product Owners Über ScrumMaster ScrumMasters Team	MetaScrum Chief Product Owner Product Owners Über ScrumMaster ScrumMasters Team

Abbildung 3.14: Drei Stufen der Scrum Implementierung.

Scrum fordert, neben einigen auch bedachten aber untergeordneten (in Scrum Terminologie »Chicken« genannt) Stakeholder Rollen, drei zentrale (»Pig«) Rollen: ScrumMaster, Product Owner und Scrum Team. Als agile Methode nimmt Scrum an, dass sich Projekte nicht im Detail planen lassen. Das entwickelnde Scrum Team soll sich daher während des Sprints weitgehend selbst organisieren. Damit dies sichergestellt ist, wird viel Wert auf eine strikte Aufgabentrennung gelegt:

#### ■ *ScrumMaster*

Der ScrumMaster ist verantwortlich für die Umsetzung von Scrum im Projekt. Dabei stellt er auch die Trennung der Aufgaben sicher. Der ScrumMaster arbeitet permanent eine Liste von Hindernissen (engl. Impediment List) ab, die vom Scrum Team erstellt wird. Durch das Ausräumen der für das Projekt hinderlichen Dinge steigt die Effizienz permanent an. Damit dies funktionieren kann, darf der ScrumMaster gleichzeitig weder als Product Owner noch als Teammitglied auftreten. Der ScrumMaster leitet tägliche Stand-Up Meetings.

#### ■ *Scrum Team*

Alle Entwickler und Tester sind Mitglieder des Scrum Teams. Es darf nicht zu groß werden, bei größeren Projekten werden statt dessen mehrere unabhängige Teams eingesetzt. Das Team schätzt den Arbeitsaufwand je Aufgabe ab und wählt selbstständig (anhand einer Priorisierung) aus, welche Aufgaben in der nächsten Iteration erledigt werden können. Dies wird als *Pull-Prinzip* bezeichnet.

#### ■ *Product Owner*

Die Kundenwünsche werden vom Product Owner an das Team weitergegeben. Er kommuniziert dazu mit den entsprechenden Stakeholdern und priorisiert die Aufgaben. Über ihn wird ebenso das Budget zugeteilt.

Ein informelles Review nach jedem Sprint soll zusätzlich zur Prozessverbesserung beitragen. Da das konkrete Vorgehen bei der Entwicklung nicht festgelegt ist, können hier Module aus anderen Software-Entwicklungsprozessen verwendet werden, häufig wird XP eingesetzt. Steigt die Projektgröße an, ist es möglich, mit einem sogenannten *Scrum of Scrums* mehrere parallele Entwicklungsgruppen zu koordinieren.

Scrum wird in der Praxis sehr erfolgreich eingesetzt. Sutherland berichtet von über 200-prozentigen Effizienzsteigerungen beim Einsatz von Scrum, selbst bei CMMI Stufe 5 Unternehmen [Sut06]. Des Weiteren wurde es erfolgreich in Projekten verschiedener Größe (von 5 bis über 600 Entwickler) eingesetzt [Sut01]. Scrum skaliert mindestens linear, ist also theoretisch auch für beliebig große Projekte einsetzbar [Sut05]. Es erhöht die Kundenzufriedenheit und verbessert die Einhaltung von Zeitrahmen [MM05].



# Kapitel 4

## Zielsetzung der Arbeit

Im Rahmen dieser Diplomarbeit soll ein Software-Entwicklungsprozess konzipiert werden, der die Besonderheiten des Forschungsumfeldes, welche im Kapitel 2 herausgestellt wurden, berücksichtigt. Daraus ergeben sich folgende Anforderungen:

### ■ *Starke Änderungen der Anforderungen*

In der Beschreibung des Forschungsumfeldes wurde auf Seite 5 bereits auf einen grundlegenden Konflikt hingewiesen: Die Anforderungsspezifikation der Software entsteht erst anhand der Ergebnisse der Software selbst. Darum sind keine der Anforderungen als sicher oder genau anzunehmen. Hinzu kommt das nicht ungewöhnliche Problem, dass Wissenschaftler als Fachfremde zwar gewohnt sind, präzise und formal zu arbeiten, aber ungerne detaillierte Anforderungsdefinitionen erzeugen oder auch nur gegenlesen [Seg05]. Neue Forschungserkenntnisse können durchaus bisherige Annahmen ins Gegenteil verkehren, bisherige Optimierungskriterien ändern und Berechnungsvorschriften völlig neu gestalten. Der Software-Entwicklung darf durch diese sich häufig und stark ändernden Anforderungen nicht allzu negativ beeinflusst werden.

### ■ *Nicht endender Lebenszyklus*

Da die Software genau solange verwendet wird, wie die Forschung andauert, ist ein Ende der Entwicklung ex ante nicht abzusehen. Damit entfällt auch eine Wartungsphase, vielmehr befindet sich die Software nach kurzer Zeit in einem stetigen Weiterentwicklungsprozess. Der

wirklich wichtige Teil der Entwicklung beginnt daher erst, wenn die Software das erste Mal in der realen Umgebung läuft.

#### ■ *Release Planung*

Nehmen dritte Forschungspartner oder erste Kunden am Projekt teil, haben diese möglicherweise Interesse, eine Version der Software zu erwerben. Da sie nicht in der selben Art mit neuen Versionen versorgt werden können, wie die Forscher in der entwickelnden Einrichtung, muss hier eine detailliertere Planung erfolgen. Diese muss in der Lage sein, den Zeitpunkt zu dem ein gewisser Satz an Funktionen in der Software umgesetzt ist, vorherzusagen. Diese Vorhersage soll möglichst wenig Ambiguität enthalten, gleichzeitig sind Möglichkeiten vorzusehen, was zu tun ist, wenn der Zeitpunkt nicht eingehalten werden kann. Ebenso wie ein Release-Zeitpunkt müssen auch die jeweiligen Kosten beziffert werden können.

#### ■ *Einflechten von externen Arbeiten*

Größere, unabhängig entwickelte Blöcke müssen ohne allzu große Komplikationen und hohen Aufwand der Software hinzugefügt werden können.

#### ■ *Teilzeitmitarbeiter*

Teilzeitmitarbeiter stellen bei Arbeiten im Team ein Problem dar, da ihre Arbeitszeiten nicht immer frei zu vergeben sind. Der Software-Entwicklungsprozess darf daher keine dauernde Anwesenheit der Beteiligten voraussetzen, oder besser noch eine weitgehend unabhängige Entwicklung ermöglichen. Bei einem Wechsel der Mitarbeiter muss die Weitergabe der Arbeiten problemlos und ohne Informationsverluste erfolgen können.

#### ■ *Unerfahrene Mitarbeiter*

Die Einarbeitung in den Prozess soll schnell und ohne größere Anstrengungen möglich sein. Sind mehrere Manntage oder gar Mannwochen notwendig, um effektiv mit dem Software-Entwicklungsprozess arbeiten zu können, geht zu viel Zeit verloren. Das Verständnis des Forschungsbereiches durch die Software-Entwickler soll vom Prozess unterstützt werden.

### ■ Unflexible Strukturen

Die Strukturen größerer Einrichtungen müssen beachtet werden. So sind größere administrative Änderungen ebenso schlecht möglich, wie beispielsweise ein freier Einkauf oder beliebige Gehälter.

## 4.1 Probleme der bekannten Software-Entwicklungsprozesse

Im Kapitel 3 wurden die gebräuchlichsten Software-Entwicklungsprozesse vorgestellt. Dennoch ist keiner der Prozesse in der Lage, die skizzierten Anforderungen zu erfüllen. Dass ungeplante Entwicklung bei einem mehrjährigen Projekten mit mehreren verschiedenen Entwicklern ziemlich zuverlässig im Chaos endet, sollte augenscheinlich klar sein. Doch auch die anderen Modelle sind unzureichend:

Das Wasserfallmodell ist auf Grund seiner Steifigkeit denkbar ungeeignet, bei den sich ändernden Anforderungen die Vorgaben der Planung einzuhalten. Auch sieht der Lebenszyklus hier eine finale Wartungsphase vor, statt eine permanente Weiterentwicklung anzusetzen. Das Einflechten von externen Arbeiten ist nicht vorgesehen, da es sich nicht in die Stufensicht integrieren lässt. Immerhin wäre es mit dem Wasserfallmodell theoretisch möglich, für eine Release-Auskopplung Kosten und Aufwand zu schätzen. Die Erfahrungen zeigen jedoch, dass dies nicht zuverlässig funktioniert. Als Vorgehensmodell spezifiziert das Wasserfallmodell einige geforderte Dinge gar nicht. So entsteht auch mit der Mitarbeiterstituation kein Problem, da an die Mitarbeiter schlicht keine Anforderungen gestellt werden.

Als iterativer Prozess ist das Spiralmodell schon eher in der Lage, mit sich ändernden Anforderungen umzugehen. Dennoch ist es zu unflexibel, um radikale Änderungen anzunehmen. Der Lebenszyklus im Spiralmodell ist auch endlich, und daher nicht zu verwenden. Immerhin lässt die iterative Natur aber das Einflechten einer externen Arbeit zum Ende jeder Iteration als möglich erscheinen. Eine Release-Auskopplung ist allerdings nicht vorgesehen, dafür sind zum einen die Iterationen zu steif, und zum anderen die Anforderungen an das Ergebnis einer jeden Iteration zu ungenau. Dass die Risikominimierung in den Vordergrund gerückt wird, ist sicherlich

von Vorteil, da so versucht wird, die Änderungen der Anforderungen abzudämpfen.

Größtes Hindernis bei der Verwendung des RUP ist der große Aufwand bei der initialen Adaption auf Grund der Komplexität von RUP. Dies ist für tendenziell unerfahrene Software-Entwickler im Forschungsumfeld, die dazu noch eine recht begrenzte Wochenarbeitszeit haben, natürlich besonders problematisch. Dadurch, dass sehr viel Wert auf Dokumentation gelegt wird, entsteht im Fall einer geänderten Anforderung ein hohes Maß an Blindleistung, da dann nicht nur der erzeugte Quelltext, sondern auch alle mit hohem Aufwand erstellten Dokumente unbrauchbar werden. Das permanente Prototyping ermöglicht ein Auskoppeln eines solchen Prototypen als Release für einen Kunden, auch die Kostenplanung ist durch die Werkzeugunterstützung bei RUP theoretisch möglich. Der RUP zu Grunde liegende Lebenszyklus ist nicht ideal: Für einen neuen Satz von Anforderungen muss erst die aktuelle Iteration abgeschlossen werden, wobei die Iterationslänge bei RUP tendenziell eher hoch ist. Die Iterationen in den Phasen können lediglich kleinere Änderungen abfangen.

Seit der Einführung der XT Variante ist das V-Modell angeblich deutlich agiler geworden und lässt das Entfernen nicht benötigter Teile einfacher als bisher zu. Unabhängige Erfahrungsberichte, ob das die Adaption des V-Modells wirklich vereinfacht, liegen aber noch nicht vor. Da das V-Modell keine zeitliche Abfolge vorschreibt wäre damit der gewünschte Lebenszyklus theoretisch abbildbar. Allerdings wird dann die Release-Auskopplung einem Problem. Auch das Einarbeiten externer Arbeiten ist nicht vorgesehen. Die Dynamik der Anforderungen kann im V-Modell nicht hinreichend aufgefangen werden, da es immer noch die bekannten Phasen des Wasserfallmodells verwendet [Nor03]. Der Einsatz des V-Modells erfordert bisher ähnlich wie beim RUP einen extrem hohen Initialaufwand, der von Teilzeitmitarbeitern nur schwer geleistet werden kann. So umfasst die Einstiegsdokumentation ca. 600 DIN A4 Seiten, vorgefertigte kürzere Versionen für kleinere Projekte existieren nicht, sondern müssen im Einzelfall erzeugt werden. Da das Modell immer zwei Projekte, je eines auf Auftraggeber- und eines auf Auftragnehmerseite erzeugt, ist der Wissenstransfer zwischen Forscher und Entwickler recht komplex und bürokratisch.

Die agilen Ansätze sind von ihren Grundannahmen her bereits bedeutend besser geeignet als die eher steifen klassischen Modelle. So nimmt man

inzwischen an, dass sich bis zu 60 Prozent der Anforderungen während der Projektlaufzeit ändern, während es früher lediglich 10 Prozent waren [Sut06]. Im Forschungsbereich ist dies unter Umständen immernoch zu wenig, hier können sich vor allem Details der Anforderungen permanent ändern.

XP wird häufig als Allheilmittel für sehr agile Projekte beschrieben. Im Forschungsumfeld ist es dennoch nicht gut einsetzbar. So fordert man die Einhaltung aller Best Practices, damit das Gesamtkonzept funktioniert. Dies ist aber hier nicht möglich: Pair Programming scheitert an den verschiedenen Arbeitszeiten der (Teilzeit-)Mitarbeiter, oder erzeugt einen hohen Overhead zur Synchronisation der Arbeitszeiten. Selbiges gilt verstärkt für ein tägliches Stand-Up Meeting. Dadurch geht ein für XP essentieller Teil der Kommunikation verloren. Da XP weitestgehend auf Dokumentation verzichtet, entsteht bei einem Mitarbeiterwechsel ein zusätzliches Problem, da viel Wissen nun neu kommuniziert werden muss, damit es nicht beim nächsten Wechsel verloren geht. Die im eXtreme Programming geforderte gleichmäßige Entwicklungsgeschwindigkeit ist nur schwer einzuhalten. Zum einen schwankt die verfügbare Entwicklerkapazität, zum anderen können anstehende Releases Mehraufwand hervorrufen. Überhaupt ist die Release-Planung und Bewertung bei XP nicht sauber vorgesehen. Gleichwohl ermöglichen die permanenten kleinen Iterationsschritte eine permanente Kontrolle über die Anforderungserfüllung und damit auch eine Validierung der Anforderungen selbst. Der erhöhte Testaufwand soll durch Spikes reduziert werden, um nicht Tests für sich sofort wieder ändernden Code zu schreiben. Allerdings besteht wegen der teilweise großen Unsicherheit der Forscher die Gefahr, dass der Aufwand hierfür Überhand nimmt. Auch ist es häufig extrem schwer, an Testdaten zu kommen, für die Sollresultate vorliegen. Die restlichen XP Best Practices lassen sich im Forschungsumfeld jedoch gut anwenden, selbst die, die sonst Schwierigkeiten bereiten, wie ein On-Site Customer.

Feature Driven Development ist ein extrem leichtgewichtiges Modell, daher fehlen hier einige Bereiche des Projektmanagements. Der Ansatz, anhand von Features die Entwicklung zu planen, ist grundsätzlich anwendbar. Eine Planung einer Release-Auskopplung ist dadurch allerdings nicht möglich. Auch wird hier die Änderungsrate der Anforderungen zu wenig bedacht. Der Versuch, zu Beginn die Fachmaterie zu verstehen und dann bereits

große Modelle zu bilden, kann nicht funktionieren, da selbst die Spezialisten auf dem Gebiet, die jeweiligen Forscher, die Materie nicht abschließend verstanden haben.

Der Evolutionären Entwicklung mangelt es ebenso an konkreten Plänen zur Umsetzung. Hier ist also viel Eigeninitiative gefragt, was natürlich auch eine gewisse Flexibilität und Anpassbarkeit des Prozesses mit sich bringt. Schnell ändernde Anforderungen sind bei Evo explizit vorgesehen und stellen daher keine großen Hürden dar. Da Evo auch keinen Lebenszyklus vorweg spezifiziert, ist eine endlose Entwicklung möglich. Das Planen von Release-Zeitpunkt und Kosten wird nicht wirklich beschrieben, daher sind hier zusätzliche Methoden notwendig. Das Einflechten von externen Arbeiten ist wegen der kurzen Iterationen sehr schwierig. Unerfahrene Entwickler kommen mit dem evolutionären Ansatz, der fast keine Einarbeitung erfordert gut klar, starre Strukturen, die feste Pläne haben wollen, wiederum gar nicht. Die intensive Einbindung der Kunden gelingt wie bei XP problemlos und wirkt entsprechende effizienzsteigernd.

Adaptive Software Development scheint zunächst beinahe alle skizzierten Anforderungen zu erfüllen. Bei genauerer Betrachtung wird man allerdings feststellen, dass die Vorgaben, die ASD macht, derart vage sind, dass sie eigentlich jede Anforderung erfüllen können. Der Entwickler bekommt zu wenig konkrete Anweisungen an die Hand gegeben, so dass er große Gefahr läuft, gänzlich chaotisch zu entwickeln. Gerade die jungen und unerfahrenen Mitarbeiter sind damit vermutlich überfordert. Das Hauptwerk zu ASD [HH02] ist für eine kurze Einarbeitung viel zu umfangreich, enthält aber dennoch kapitelweise keine sinngebenden Aussagen.

Scrum besticht zunächst durch seine nachgewiesene Effizienz, auch sind Teile des Modells für Software-Entwicklung im Forschungsumfeld zu verwenden: Produkt und Sprint Backlog etwa, was die noch anstehenden Aufgaben enthält. Alleine betrachtet sind sie aber nicht mehr als Todo-Listen. Die festen Iterationen bei Scrum stellen ein Problem dar, da die Entwickler im Allgemeinen an verschiedenen komplexen Aufgaben arbeiten, die nicht simultan fertig werden. Für eine etwaige Release-Auskopplung ist das Konzept eines Sprints hingegen gut vorstellbar. Scrum hat keinen wirklich endenden Lebenszyklus, was positiv hervorzuheben ist. Da die Aufgaben einfach in einem Produkt Backlog gesammelt werden, ist eine permanente Weiterentwicklung der Software möglich. Allerdings stellt eine konstant

ansteigende Menge an Aufgaben eine Herausforderung für die Visualisierung des Projektfortschritts dar: Die Fortschrittüberwachung mit einem Burn Down Chart wäre nämlich auch zu verwenden, allerdings entstehen dabei Probleme, wenn neue Anforderungen hinzukommen. Die Selbstorganisation des Teams ist ähnlich wie bei ASD tendenziell gefährlich, da das Team eher unerfahren in der Software-Entwicklung ist. Auch ist ein Ausräumen der Hindernisse des Prozesses im strikten Gefüge einer größeren Einrichtung tendenziell eher schwierig.

	Extrem ändernde Anforderungen	Nicht endender Lebenszyklus	Release Planung	Einflechten von externer Arbeit	Teilzeitmitarbeiter	Unerfahrene Mitarbeiter	Unflexible Strukturen
Ungeplante Entwicklung	-	o	--	o	±	±	o
Wasserfallmodell	--	--	+	--	o	+	±±
Spiralmodell	o	-	--	--	o	+	o
Rational Unified Process	-	o	±±	-	--	--	+
V-Modell XT	-	o	±±	-	--	--	±±
eXtreme Programming	+	o	-	o	--	--	--
Feature Driven Development	o	+	-	o	+	+	-
Adaptive Software Development	+	+	o	o	--	--	-
Scrum	o	±±	+	o	-	-	--
Evolutionäre Entwicklung	±±	o	-	--	+	+	--

**Abbildung 4.1: Anforderungserfüllung der Software-Entwicklungsprozesse.**

Abbildung 4.1 gibt eine Übersicht, wie gut die einzelnen bekannten Software-Entwicklungsprozesse die gestellten Anforderungen an einen Prozess im Forschungsumfeld erfüllen. Keiner der angeführten Software-Entwicklungsprozesse plant einen quasi endlosen Lebenszyklus ein, bei dem in der Regel nicht auf konkrete Releases hingearbeitet wird. Am ehesten kann diese Forderung noch Scrum erfüllen, ohne dafür jedoch eine konkrete

Unterstützung zu liefern. Doch auch Scrum kann nicht direkt im Forschungsumfeld eingesetzt werden, außerdem benötigt es als sehr generisches Modell noch eine Konkretisierung der Entwicklung.

## 4.2 Zieldefinition

Der im Zuge dieser Arbeit zu entwickelnde Software-Entwicklungsprozess muss die verschiedenen, beschriebenen Anforderungen erfüllen. Die beiden wichtigsten sind der nicht endende Lebenszyklus der Software, sowie die starke Änderungsrate der Anforderungen. Als Nebenbedingungen soll es sowohl möglich sein, größere, externe Arbeiten, beispielsweise Diplomarbeiten, einzuflechten, als auch auf konkrete Releases für einen Kunden oder Forschungspartner hinzuarbeiten. Die Struktur der Mitarbeiter und Abläufe einer Forschungseinrichtung muss bedacht werden.

Es beinhaltet weniger Risiken iterativ zu entwickeln, als mit einem Wasserfallansatz, wie Tingey mit Mitteln der Informationstheorie allgemein nachgewiesen [Tin05] hat. Andererseits steigt das Risiko, wenn man ein Projekt in mehrere einzelne Teilprojekte zerteilt. Ebenso zeigen Benediktsson und Dalcher, dass inkrementelle Entwicklung den Gesamtaufwand reduziert [BD03]. Der Software-Entwicklungsprozess muss ermöglichen, dass mehrere Software-Entwickler an verschiedenen Projekten gleichzeitig arbeiten können. Ziel muss also sein, einen globalen, iterativen, inkrementellen und parallelisierbaren Ansatz zu finden.

# Kapitel 5

## Software-Entwicklung als Fluss

Diese Kapitel stellt die »Software-Entwicklung als Fluss« vor, und beschreibt die grundlegenden Gedanken und Abläufe, die zum Verständnis notwendig sind. Spezifischere Details werden in Kapitel 6 erörtert, ebenso findet sich dort eine Anleitung zum schrittweisen Umsetzen der beschriebenen Ideen (Abschnitt 6.7, Seite 94).

### 5.1 Generelles Vorgehen

Nach Highsmith [HH02] eignet sich agile Software-Entwicklung im Allgemeinen sehr gut für die von Moore im Zuge des »Tornado-Phänomens« definierten Phasen des Technologieakzeptanz-Lebenszyklus »Bowling Alley« und »Tornado« [Moo95]. Dies soll hier nicht weiter diskutiert werden, es sei allerdings angemerkt, dass sich der hier vorgestellte Software-Entwicklungsprozess sicher nicht für Tornado Phasen eignet. Die Normstrategie [Moo04] fordert hier eine Produktion von Infrastruktur und eine Generalisierung, weg von spezifischen Kundenbedürfnissen, hin zum Massenausstoß. Dies läuft den Intentionen der Software-Entwicklung im Forschungsumfeld entgegen, da der Hauptzweck Know-How-Gewinn in den Hintergrund tritt.

#### 5.1.1 Motivation

Durch die Analyse der bekannten Software-Entwicklungsprozesse in Kapitel 4.1 wurde herausgestellt, warum die bekannten Prozessmodelle gar nicht

oder nur schlecht anwendbar sind. Dabei fällt auf, dass eine starre Einteilung des Entwicklungsprozess in Phasen besonders hinderlich ist. Es fehlt zum einen die Flexibilität um schnell auf sich ändernde Anforderungen zu reagieren, zum anderen verzögert eine Einteilung die Entwicklung, da die Mitarbeiter für gewöhnlich nicht gleichzeitig in verschiedenen Bereichen und in verschiedenen Phasen tätig sein können. Scrum und FDD zeigen außerdem, dass bei sehr dynamischen Prozessen weniger Planung effektiver sein kann als der Versuch, möglichst viel im Vorhinein festzulegen.

Da die Anforderungen sowieso nicht en bloc auftreten, bietet sich eine featuregetriebene Entwicklung an. Das hier entwickelte Vorgehen ähnelt nicht zuletzt deshalb in gewisser Weise den Optional Scope Contracts von Beck [BC99]: Die zu entwickelnden Features werden erst mit der Zeit bekannt, was neben einer evolutionären Entwicklung auch eine inkrementelle Planung und Lieferung erfordert. Lediglich eine grobe Richtung, sowie etwaige Zeit- bzw. Kostenrahmen werden zu Beginn festgelegt. Dadurch ist es allerdings besonders wichtig, die Qualität der Software ständig zu überwachen und auf einem akzeptablen Niveau zu halten. Siehe dazu Abschnitt 5.6, Seite 69.

### 5.1.2 Idee der Software-Entwicklung als Fluss

Das hier entwickelte Prozessmodell verwendet also kein starres Phasenkonzept. Die evolutionäre Entwicklung von Gilb verwirklicht das durch sehr viele kleine Wasserfälle. Diese Idee wird hier noch weiter getrieben, und die Entwicklung wird als stetiger Fluss angesehen. Die Metapher eines Flusses passt in vielerlei Hinsicht und wird im Folgenden zunächst erläutert.

Ein Flusslauf läuft genau wie die Software-Entwicklung im Forschungsumfeld nicht geradlinig auf ein Ziel zu, sondern nimmt permanent unvorhersehbare Wendungen. Dabei hält er nur ein grobes Ziel ein: Der Fluss läuft immer bergab, der Software-Entwicklungsprozess versucht, Erkenntnisse in einem bestimmten Bereich zu erzeugen. Dies ist gerade der evolutionäre Aspekt bei der Software-Entwicklung als Fluss: Die konkret einzuschlagende Richtung entscheidet sich immer nur lokal und erweckt von außen leicht ein chaotisches Bild.

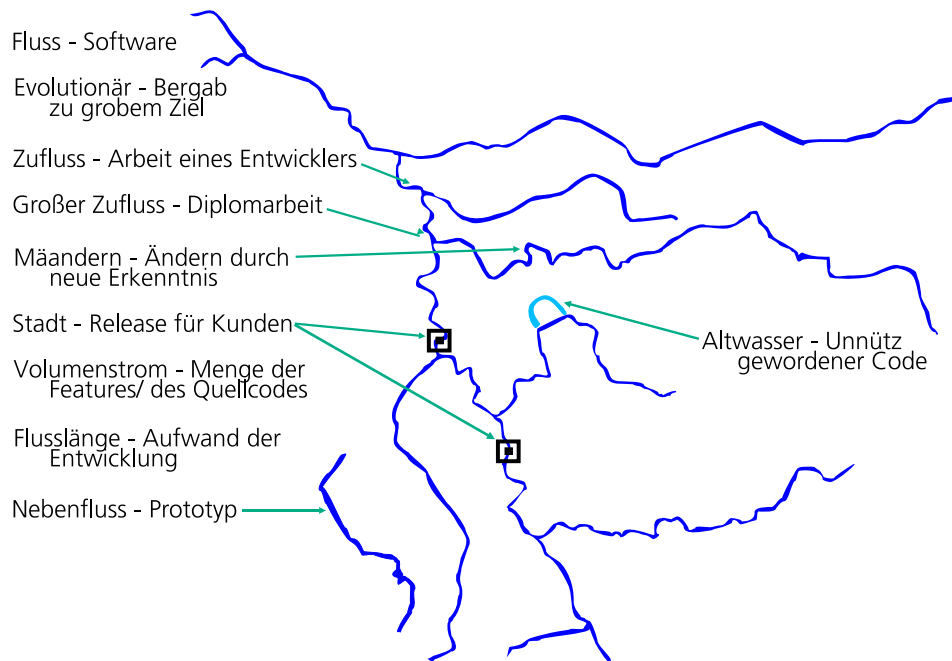


Abbildung 5.1: Software-Entwicklung als Fluss im Großen.

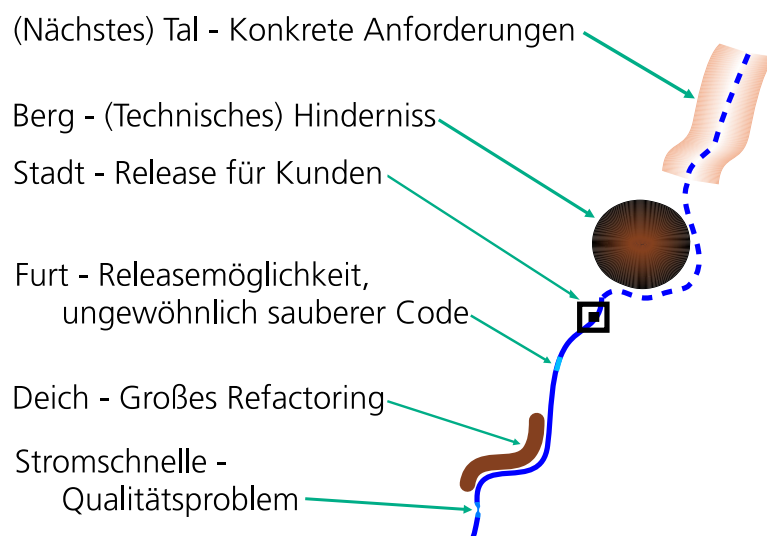
Durch Mäandern kann es passieren, dass alte Teile des Flusses zu Altwässern abgeschnitten werden. Sie werden nicht mehr gebraucht, da es inzwischen einen kürzeren Weg gibt. Dieses Phänomen kann in der Software-Entwicklung genauso beobachtet werden: Eine neue Erkenntnis macht beispielsweise alte Rechenwege unnötig, und der Code bleibt höchstens noch als Referenz im Versionsverwaltungssystem erhalten.

Die Integration eines neuen Software-Teils kann als Zufluss einer bestimmten Größe betrachtet werden, der dann auch dementsprechend Auswirkungen auf den Hauptstrom hat. Jeder Entwickler produziert diese Zuflüsse autark in seinem Gebiet, gesteuert von und unter den Beschränkungen der Umgebung. Die Erfassung des Entstehens dieser Zuströme und der Umwelt muss werkzeugunterstützt vorgenommen werden, Details dazu in Abschnitt 6.6, Seite 88 und Abschnitt 5.4, Seite 64.

Nebenflüsse, die keinen direkten Einfluss auf den Hauptstrom haben, sind Prototypen, die nur auf einem eingeschränkten Gebiet eine kurzfristige Erkenntnis erzeugen sollen. Einen derartigen Wegwerfprototypen dennoch in das Projekt zu integrieren erzeugt sehr wahrscheinlich Qualitätsprobleme [LL07].

Mit fortschreitender Flusslänge nimmt für gewöhnlich der Volumenstrom zu. Dies ist analog auf Entwicklungsaufwand und Menge des produzierten Quellcodes übertragbar. In beiden Fällen ist die Beziehung nicht zwingend gegeben, aber üblich.

An markanten Punkten der Entstehung wird der aktuelle Status als Release an einen Kunden geliefert. Dies kann man als Stadt am Flusslauf interpretieren. Die soeben dargestellte grobe Metaphorik ist in Abbildung 5.1 noch einmal zusammengefasst.



**Abbildung 5.2: Software-Entwicklung als Fluss im Kleinen.**

Auch in einem lokalen Kontext kann die Metaphorik weiterhin erhalten bleiben. So kann die Landhöhe als Maß für die Anforderungen bzw. Schwierigkeiten verstanden werden: Das nächste zu erreichende Tal ist dann eine konkrete Anforderung, während ein Berg ein (technisches) Hindernis darstellt, das entweder umgangen werden muss, oder mit erhöhtem Aufwand zu entfernen ist. Das Change Request Management System (CRM) übernimmt dann die Funktion einer Landkarte der Software-Landschaft.

Da Quelltextmenge und Entwicklungsgeschwindigkeit relativ einfach automatisiert messbar sind, versucht man häufig aus diesen und anderen Werten die Qualität der Software-Entstehung über Metriken zu messen. Im Bild des Flusses kann eine Stromschnelle bzw. ein See analog als Qualitätsproblem empfunden werden. Findet man am Fluss hingegen eine besonders seichte

Stelle, eine Furt, so ist die Software-Qualität besonders hoch, und es bietet sich an, hier eine Stadt, also ein Release, zu erzeugen.

Manchmal sind erzwungene, weitreichende Richtungsänderungen unumgänglich, auch wenn sowohl beim Flussmanagement als auch bei der Software-Entwicklung versucht wird, sie zu vermeiden. Beim Fluss ist hier ein Deich erforderlich, auf der Software-Ebene ein größeres Refactoring, wie es in Kapitel 6.1 im Detail behandelt wird. Abbildung 5.2 stellt diese feineren Details nochmals zusammen.

### 5.1.3 Entwicklungsablauf

Nachdem nun die Entwicklung in ein metaphorisches Rahmenwerk gesetzt wurde, wird im Folgenden auf den konkreten Entwicklungsprozess eingegangen. Die einzelnen Entwickler produzieren also – bevorzugt kleine – Zuflüsse, die dann in einem Hauptstrom vereinigt werden. Kleine Änderungen (Zuflüsse) sind aus mehreren Gründen zu bevorzugen: Zum einen ändern sie die Flussrichtung tendenziell nur wenig, was sich risikominimierend auf die weitere Planung auswirkt. Des Weiteren ist der Integrationsaufwand geringer und die Wahrscheinlichkeit von Konflikten (Turbulenzen) sinkt dadurch. Vor allem ist das Risiko, etwas Falsches entwickelt zu haben, durch ein schnelleres Feedback von Anwenderseite geringer. Die einzelnen Entwicklungsphasen, wie Anforderungsanalyse, Software-Design, Implementierung und Test, müssen dabei nicht zwingend streng konsekutiv ablaufen, sondern können durchaus bei mehreren Aufgaben und Entwicklern asynchron und parallel bearbeitet werden. Hier wird ein Unterschied zur klassischen evolutionären Entwicklung deutlich.

Da Entwickler die Software-Entwicklung gerne als einen kreativen Prozess verstehen möchten, soll diese möglichst zwangfrei erfolgen. Dies wird von administrativer Ebene ungern so betrachtet, hier wird viel mehr ein ingenieurmäßiges Vorgehen gewünscht [LL07]. In jedem Fall ist eine Steuerung von Nöten. Diese erfolgt auf verschiedene, möglichst integrierte und werkzeugunterstützte Arten: Zunächst erfasst ein Change Request Management System die anfallenden Features bzw. User Stories (Details in Abschnitt 5.2, Seite 56) und hält gleichzeitig die anzurechnende Arbeitszeit fest. Das Hinzufügen eines Zuflusses in den Hauptstrom wird von einem angekoppelten

Versionsverwaltungssystem nachgehalten. Es werden dazu verschiedene Statusse<sup>1</sup> definiert, die Regeln ob ein Hinzufügen – ein Commit – möglich ist oder nicht. Damit werden Komplikationen, zum Beispiel mit anstehenden Releases, verhindert; Details dazu in Abschnitt 5.4, Seite 64.

Bei größeren Zuflüssen, beispielsweise Diplomarbeiten oder Ähnlichem, ist eine permanente Integration in diesen kleinen Schritten ebenso zu bevorzugen. Wenn dies wegen externer Rahmenbedingungen nicht möglich ist, läuft die Integration ab wie ein großes Refactoring; siehe Abschnitt 6.1, Seite 79.

Gilt es auf ein spezifisches Release hinzuarbeiten, also einen bestimmten Satz von Anforderungen zu erreichen, können diese separat entwickelt werden und dann hinreichend lange vor dem Zieltermin zusammengeführt werden. Dies ist über die bekannten Felder des Zielrelease im CRM System möglich, kollidiert aber möglicherweise mit der in Kapitel 5.3 vorgeschlagenen Priorisierung der Features. Hier muss dann im Einzelfall entschieden werden. Es sollten aber auf jeden Fall die erkenntnisserzeugenden Features eines Zielrelease zuerst entwickelt werden, um das Risiko minimal zu halten. Zur späteren Nachhaltung und Wartung des Release gibt Abschnitt 5.5, Seite 68 weitere Hinweise.

## 5.2 Aufwandsschätzung

Besonders für die Kosten- und Zeitplanung von Releases, aber auch zur allgemeinen Projektüberwachung ist es notwendig, den Aufwand und damit die Kosten eines Features schätzen zu können. Die Erfahrungen mit dem Wasserfallmodell zeigen, dass naive Ansätze zuverlässig nicht funktionieren. Fragt man also eine Gruppe Entwickler, wie lange sie für dieses oder jenes Feature brauchen würden und gibt ihnen keine Bewertungsanleitung an die Hand, werden die Planzeiten beinahe nie erreicht. Roock schlägt daher vor, dass die Entwickler den Aufwand relativ bewerten [RW04]. Dies deckt sich mit psychologischen Erkenntnissen, wonach Menschen sowieso immer relativ entscheiden [KT79, LP92, LT89] und daher eine externe Kalibrierung

---

<sup>1</sup>In dieser Arbeit wird die Pluralbildung »Statusse« nach Mackensen der lateinischen Form »Statūs« vorgezogen.

benötigen [vN02]. Dazu werden Subsysteme in Features und die Features wiederum in Stories zerlegt. Auf unterster Ebene werden also User Stories statt Anwendungsfällen verwendet, was sich bei Unsicherheit risikominierend auswirkt [C+04]. Jede Ebene bekommt eigene Aufwandspunkte zugewiesen: Subsystem-Effort-Points (SEP), Feature-Effort-Point (FEP), Story-Effort-Point (STEP). Diese werden dann von den Entwicklern bei neuen Aufgaben relativ zu bekannten vorangegangenen geschätzt. Die so erzeugten Schätzungen sind bedeutend präziser, als solche, die aus einer direkten Nachfrage nach anzusetzenden Stunden Entwicklungszeit einer Änderung folgen.

Während der Arbeit hält der Entwickler nach, mit welcher Story er beschäftigt ist. Diese Aufwandserfassung sollte in einem integrierten Werkzeug erfolgen (Abschnitt 6.6, Seite 88). Nach einer anfänglichen Kalibrierungszeit ist die Dauer für einen Step hinreichend gut bekannt, ebenso die Anzahl der jeweiligen Aufwandspunkte je Hirarchieebene. Dann kann relativ präzise zwischen den einzelnen Aufwandspunkten umgerechnet werden, und Gesamt- sowie Restaufwand angegeben werden. Dazu müssen die Konstanten  $n$  und  $m$  errechnet werden, die die Umrechnung zwischen den Ebenen zulassen. Die Errechnung der Konstanten kann mit bekannten Verfahren der Statistik erfolgen, z. B. der Methode der kleinsten Fehlerquadrate nach Gauß [DS67]:

$$\begin{aligned}1\text{SEP} &= n \cdot \text{FEP} \\1\text{FEP} &= m \cdot \text{STEP} \\ \Rightarrow 1\text{SEP} &= n \cdot m \cdot \text{STEP}\end{aligned}$$

Der Zusammenhang ist in Abbildung 5.3 zusammengefasst.

Ebenso kann aus der Aufwandserfassung die durchschnittliche Entwicklungsgeschwindigkeit eines Entwicklers  $s_i$  in STEP pro Stunde ermittelt werden. Hierbei ist es sehr wichtig, den beteiligten Entwicklern zu kommunizieren, dass dies nicht zur ihrer Überwachung oder Bewertung verwendet wird. Dagegen sprechen mehrere Gründe: Zum einen werden die Entwickler die Messungen unterlaufen, wenn sie Nachteile befürchten müssen. Außerdem sind manche Entwickler damit beschäftigt, andere zu unterstützen. Dies ist schwierig zu erfassen, variiert aber die  $c_i$  Werte im Einzelfall sehr deutlich. Eine Änderung über die Zeit kann aber wirksam veröffentlicht werden, wenn zusätzlich auch Metriken der Fehlerbeseitigung/Rücklaufquote

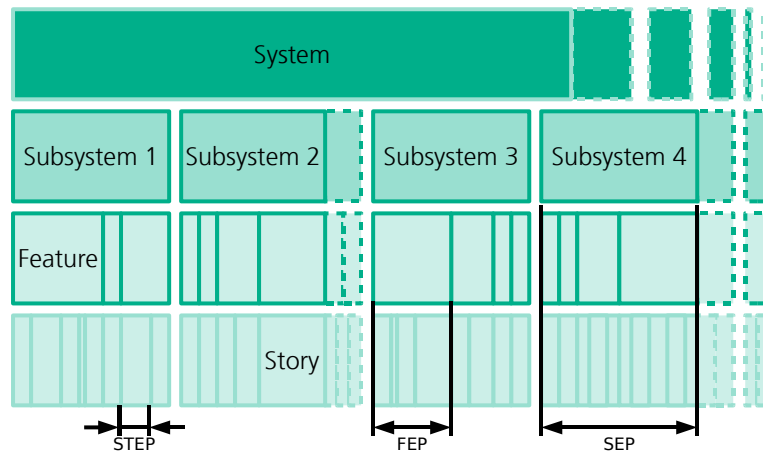


Abbildung 5.3: Effort Points auf verschiedenen Stufen.

betrachtet werden. Dies spornt die Entwickler an, sich dort besser zu positionieren [Coh02]. Andernfalls wird diese Metrik sicherlich unterlaufen werden. Im besten Fall verhindert die eingesetzte Software daher das direkte Auslesen der Werte. Beachtet werden muss, dass der  $c_i$  Wert in *Effort Time* angesetzt wird, also in realer Arbeitszeit. Zur Umrechnung in *Lead Time*, also verstrichener Gesamtzeit, muss die Arbeitszeit je Echtzeit bedacht werden (Siehe dazu auch [Mal0J]).

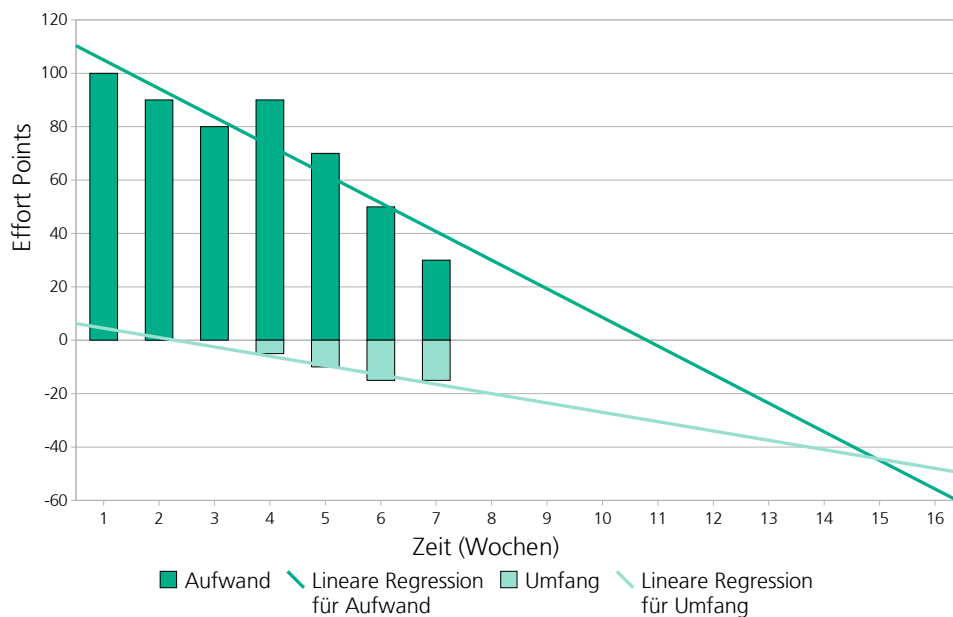


Abbildung 5.4: Der Restaufwand als erweitertes Burndown Chart.

Zur Visualisierung des Restaufwandes bietet sich eine modifizierte Variante der Scrum Burndown Charts an, wie von [Coh05] vorgeschlagen und in Abbildung 5.4 dargestellt. Hier wird neben dem noch verbleibenden Restaufwand auch jede Änderung des Umfangs separat notiert. Dies ist gerade bei den extremen Schwankungen der Anforderungen notwendig. Die Y-Achse im Diagramm stellt den Restaufwand dar, wobei die Höhe der Säulen den gesamten Restaufwand visualisiert. Der von Entwicklern erledigte Aufwand wird immer von oben entfernt, ebenso wird dort der von *ihnen* festgestellte Zusatzaufwand, beispielsweise durch technische Schwierigkeiten, hinzugefügt. An der unteren Kante wird von *außen* hinzugefügter bzw. entfernter Aufwand notiert, hierbei handelt es sich also um Änderungen im Umfang. Kommen zu einem sich gerade in der Entwicklung befindenden Release neue Anforderungen hinzu, werden diese unten angefügt. Läuft ein Chart dann längere Zeit, wird es sich permanent nach unten verschieben, was aber durch Ausdehnung der Skalen kein Problem darstellt. Die Aktualisierung des Charts kann nach jeder abgeschlossenen Story erfolgen oder aber in festen Abständen, etwa wöchentlich.

Anhand der vorgeschlagenen Visualisierung kann auch permanent die zu erwartende Restlaufzeit errechnet werden. Mittels einfacher Techniken, wie einer linearen Regressionsanalyse, kann eine stets aktuelle Abschätzung getroffen werden, wann alle Effort Points erledigt sein werden, und das Projekt damit abgeschlossen ist. Dieser Wert ist allerdings wegen der schwankenden und ambiguitätsbehafteten Wochenarbeitszeit im Forschungsumfeld mit Vorsicht zu genießen. Der Prozessüberwacher (Siehe 5.7) kann versuchen, dies hinreichend detailliert zu erfassen, um für die nahe Zukunft eine genaue Prognose zu erreichen. Langfristig sollten sich die Schwankungen ausgleichen, so dass sich dennoch eine einfache Trendfortschreibung anbietet. Über lange Sicht lässt sich aus den Differenzen zwischen der ursprünglichen Annahme des Fertigstellungszeitpunktes und des wegen Änderungen im Umfang und wegen zusätzlicher Hindernisse erreichten Zeitpunktes ein Sicherheitsfaktor für zukünftige Planungen ermitteln.

Werden größere, abgeschlossene Blöcke in Studien- bzw. Diplomarbeiten ausgelagert, bei denen keine fortwährende Aufwandserfassung möglich ist, so sollte versucht werden, nachträglich die Zerlegung in Effort Points sowie

die Aufwandsbetrachtung vorzunehmen. Dies vergrößert zum einen die zur Verfügung stehende Datenbasis und ermöglicht zum anderen relative Schätzungen gegen eben diese Teile. Dabei muss besonders beachtet werden, nicht einem Hindsight Bias zu erliegen, und die Dinge im Nachhinein zu positiv zu bewerten [vN02]. Es ist keine »gefälschte Entstehungsgeschichte« [LL07] gefragt, sondern reale Angaben.

### 5.3 Priorisierung

Wie bereits in der Motivation herausgestellt, ist es aufgrund des sehr ungenauen Wissens der Anwendungsdomäne nicht möglich, von Anfang an eine vollständige Liste der Anforderungen zu erzeugen. Daraus folgt, dass sich auch die Priorisierung der User Stories nicht von Anfang an festlegen lässt. Diese werden erst mit der Zeit bekannt, und müssen daher auch jedes Mal neu priorisiert werden. Da Zeit und Ressourcen für gewöhnlich schon beschränkt sind, ist es darum die Menge der realisierten Features, die variabel ist, wie in Abbildung 5.5 nach [Mal02] dargestellt. Variabel

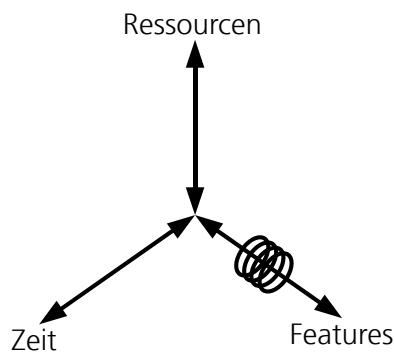


Abbildung 5.5: Gegenüberstellung von Zeit, Ressourcen und Featuremenge.

heißt hier natürlich, dass vermutlich nicht alle Anforderungen aus einem Projektabschnitt realisiert werden können. Langfristig sollten sich natürlich die zur Verfügung stehenden Ressourcen gemäß dem Ausgleichsgesetz der Planung [Gut72] an die Zeit und Aufwandsbedürfnisse anpassen. In der klassischen evolutionären Entwicklung wird versucht, die Features mit dem höchsten Kundennutzen zuerst zu realisieren. Dies wird hier ein wenig modifiziert: Da das Projekt längerfristig angelegt ist, und der

Hauptzweck Know-How-Gewinn in der Anwendungsdomäne ist, wird Wissensgewinn höher bewertet als üblich. Auch muss versucht werden, die Risiken der Änderungen zu minimieren, damit sie nicht wie in Abbildung 5.6 exponentiell steigen [LH06].

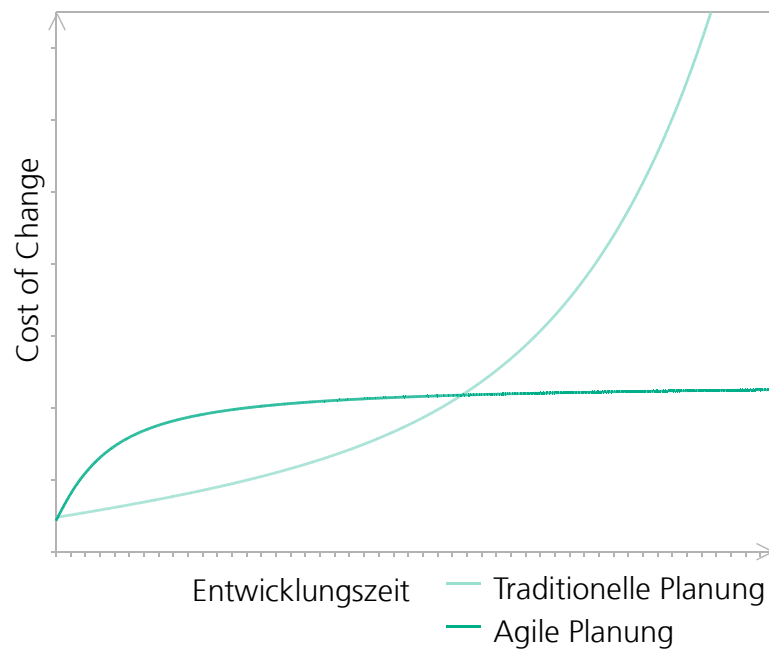


Abbildung 5.6: Expected Cost of Change: Agile vs. Traditionelle Methoden.

Es ergeben sich folgende Einflussfaktoren auf die Priorität einer User Story bzw. eines Features:

- *Expected Cost of Change*  
Die Kosten einer späteren Änderung eines Features, gewichtet nach der Wahrscheinlichkeit der Änderung.
- *Erkenntnisgewinn*  
Durch die Realisation eines Features kann zusätzliches Wissen, sowohl auf technischer, als auch auf der wichtigeren Anwenderseite, gewonnen werden.
- *Externe Priorisierung*  
Gemeint sind Vorgaben z. B. durch Realeaseanforderungen oder Management. Vom Kunden beanstandete Fehler sind hier besonders hervorzuheben.

### ■ *Entwicklungsbeschleunigung*

Manche Dinge nutzen lediglich den Entwicklern, um ihre Arbeit zu erleichtern. Dies können sowohl der Einsatz zusätzlicher CASE Werkzeuge, als auch Änderungen in der Software selber sein.

Die hier genannten Faktoren sind selbstverständlich nicht unabhängig voneinander zu betrachten. Es kann daher keine klare Hierarchie definiert werden. So ist es immer denkbar, dass zum Beispiel eine externe Vorgabe jetzt umgesetzt werden muss, auch wenn alle internen Überlegungen dagegen sprechen. Auf der anderen Seite kann es sinnvoll sein, eine Woche Arbeit nur in Entwicklungsbeschleunigung zu investieren. Daraus entsteht zwar kein direkter Kundennutzen, der kann aber im Folgenden deutlich schneller erzeugt werden. Es sollte daher stets sinnvoll abgewogen werden, wie stark der Einfluss der einzelnen Faktoren zur Zeit ist.

Zu den Faktoren im Detail: Der Expected Cost of Change (ECC) eines Features ist das arithmetische Produkt aus Änderungswahrscheinlichkeit und den Kosten für eine Änderung [HC06b]. Die beiden Faktoren Änderungswahrscheinlichkeit und Kosten bzw. Aufwand einer Änderung sind jeweils zu schätzen. Für die Wahrscheinlichkeit dienen hauptsächlich die Erfahrungen der Schätzenden als Grundlage, wohingegen bei der Schätzung der Auswirkungen durch Werkzeuge (IDE, Sotoarc) ein Überblick der betroffenen Teile gewonnen werden kann. Hierbei sind die üblichen Vorsichtsmaßnahmen bei Gruppenentscheidungen vorzunehmen: Risky Shift vermeiden, Kurzsichtigkeit beachten (siehe auch [vN02]). Bei einer werkzeugunterstützten Schätzung bietet sich daher eine Delphi-Methode an, die diese Probleme von vornherein vermeidet.

Features die eine hohe ECC aufweisen sind möglichst spät zu realisieren. Da der ECC sich aber durch neue Erkenntnisse verändert, werden Features, die den ECC reduzieren, möglichst früh realisiert. Dies sind gerade erkenntnisgewinnende Features, beispielsweise kurze Proof-of-Concept-Blöcke. Sie sind zu bevorzugen, da sie Unsicherheiten reduzieren und damit risikominimierend wirken. Ein Proof-of-Concept, der als Nebenfluss realisiert wird, hat dabei sogar ein ECC von null, da der Code danach weggeworfen wird und damit keiner Gefahr unterliegt, geändert werden zu müssen. Abbildung verdeutlicht dies: Die Linien stellen die nach ECC sortierten Features dar, wobei solche mit geringem ECC-Wert zuerst zu realisieren sind. Die gesamte

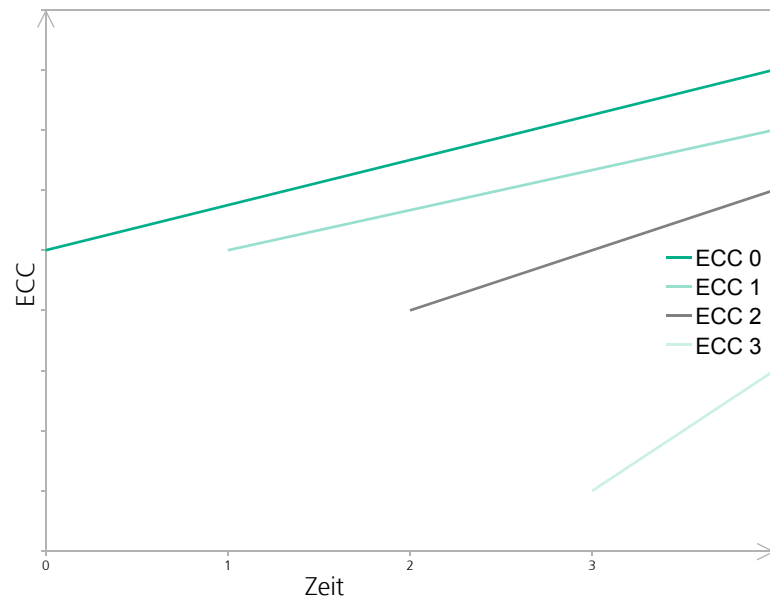


Abbildung 5.7: Sinkender Expected Cost of Change.

Menge an zu erwartenden Änderungen ist dann die Fläche unter den Kurven. Mit fortschreitender Entwicklung werden Erkenntnisse gewonnen, die die Kurve insgesamt nach unten verschieben, wobei die einzelnen Features durchaus umsortiert werden können (In der Abbildung nicht dargestellt). Die Steigung der Kurve ändert sich natürlich ebenso mit jeder neuen Erkenntnis.

Die externe Priorisierung wird von außen vorgegeben und muss beachtet werden. Es kann versucht werden, diese zu Gunsten der anderen Faktoren zu verändern, da sie für den Projektfortschritt an sich besser sind. Wenn aber beispielsweise ein Kundenwunsch sofort erledigt werden muss, und sei er noch so riskant, haben die Entwickler häufig wenig Einfluss. Dies erklärt sich nicht zuletzt mit dem möglicherweise sehr schnell zu realisierenden Gewinn, der auf eine solche Änderung folgt.

Arbeiten zur Entwicklungsbeschleunigung sind alle Tätigkeiten, die weder mit der entwickelten Software noch mit dem Anwendungsbereich direkt in Verbindung stehen. Dazu zählen so profane Dinge wie Kaffee kochen, aber auch die Installation und Wartung von Werkzeugen zur Entwicklung, beispielsweise die Implementierung eines neuen Change Request Management Systems. Der Anteil an der Gesamtarbeitszeit von nicht direkt produktiver Arbeit sollte nicht unterschätzt werden, die Literatur gibt ihn mit bis zu 30-50 Prozent an [Mal02, Bro75]. Andererseits können auch Änderungen in der Software die Entwicklung beschleunigen. Beispielsweise führt die Erstellung

von generischen Debugroutinen nicht direkt zu Kundennutzen, ermöglicht später aber eine schnellere Fehlerbehebung. Zensierter Abschnitt

Gemeldete Fehler in der Software haben in diesem Zusammenhang eine Sonderstellung. Sie sind grundsätzlich früh zu analysieren und auf ihre Auswirkungen hinsichtlich der Unsicherheiten zu bewerten, da sie oftmals Unklarheiten aufzeigen. Wenn die Ursache klar ist, kann entschieden werden, ob eine Behebung des Fehlers große Auswirkungen auf die ECCs hat und entsprechend hoch priorisiert wird. Ist hingegen zwar ein größerer Umbau erforderlich, es werden aber daraus wenig neue Erkenntnisse gewonnen, ist eine niedrige Priorität möglich. Wer den Fehler festgestellt hat, ist zunächst unerheblich, wobei dies natürlich Auswirkungen auf die externe Priorisierung des selbigen haben kann [Zan07]. Möglicherweise macht es auch Sinn, vor der Behebung eines Fehlers erst gewisse Unsicherheiten in der Domäne zu senken, indem ein anderes Feature vorher bearbeitet wird. Hier überträgt sich dann die Priorität des Fehlers auf das andere Feature, welches die ECC des Fehlers senkt.

Da die Auswirkungen des Lernprozesses unklar sind, sollte nach Möglichkeit immer nur geplant werden, welche Arbeiten sofort und welche eben nicht sofort angegangen werden. Detailliertere Zukunftsplanung ist unnötig, denn tritt eine Änderung ein - und davon wird grundsätzlich ausgegangen - war sie nur unnötiger Arbeitsaufwand [HC06a].

## 5.4 Projektstatus

Da die Entwickler weitgehend unabhängig arbeiten, ist es notwendig, die Mündung des jeweiligen Zufluss, also das Einfügen in den Hauptzweig (Commit), derart zu koordinieren, dass keine anderen Entwickler oder gar Anwender gestört werden. Vor allem muss sichergestellt werden, dass sich die Software anschließend noch in einem benutzbaren Zustand befindet. Dazu werden verschiedene Projekt- und Zuflussstatusse definiert, mit denen dies dann sichergestellt werden kann:

Zunächst bekommt jeder Change Request, und damit jeder Zufluss, ein zusätzliches Attribut, welches seinen Typ angibt. Die einzelnen Typen sind:

1. *Non Switchable Breaker*

Ein Non Switchable Breaker ist Code, der beim Commit dazu führt, dass das Projekt in einen nicht benutzbaren Zustand gerät, zu dem auch kein Workaround existiert. Der problematische Teil des Codes lässt sich insbesondere auch nicht abschalten.

2. *Switchable Breaker*

Hier funktioniert die Software auch nach dem Commit noch weiterhin, da die problembehafteten Stücke abgeschaltet werden können, bzw. extra hinzugeschaltet werden müssen. Der Entwickler erwartet allerdings Probleme beim Hinzuschalten, daher ist die Voreinstellung natürlich, dass der neue Teil abgeschaltet ist.

3. *Non breaker*

Hierbei geht der Entwickler, bzw. ein anderer Verantwortlicher (siehe Kapitel 5.7), nicht davon aus, dass die Software-Funktionalität gefährdet ist. Dies ist natürlich nie sicher festzustellen, wichtig ist hier, dass es nicht zu erwarten ist.

4. *Trivial Bugfix*

Ein Trivial Bugfix ist derart simpel, dass mit nahezu hundertprozentiger Sicherheit festgestellt werden kann, dass keine Funktionalität beeinträchtigt wird. Dazu zählt beispielsweise das Beheben eines Nullpointers, nicht aber das Beheben eines Multithreading-Problems.

5. *Documentation & String changes*

In diesem Fall entstehen überhaupt keine Änderungen an der Funktionalität. Es sind nur Änderungen in der Dokumentation, gleich welcher Art, oder sonstigen Texten zulässig.

6. *No Change*

Ein No Change Zufluss ändert zunächst überhaupt nichts. Dieser Zustand ist nötig für alle Change Requests, die noch nicht untersucht sind, und auch für solche Fehlerberichte, die sich selbst als fehlerhaft herausstellen.

Entsprechend zu den oben beschriebenen Zuflusstypen erhält auch der Hauptfluss einen Zustand. Dieser beschreibt die aktuell zulässigen Zuflüsse nach einem Schema analog zu den Zuflusstypen. Die verschiedenen

Beschränkungen sind notwendig, wenn beispielsweise auf ein Release hingearbeitet wird. So wird zwar die Flexibilität der Zuströme und damit auch die Entwicklungsgeschwindigkeit in den höheren Stufen zum Teil deutlich eingeschränkt, dafür steigt aber auch die Sicherheit der Planung für eine Release-Auskopplung. Der Zusammenhang zwischen Änderbarkeit und Sicherheit ist in Abbildung 5.8 dargestellt.

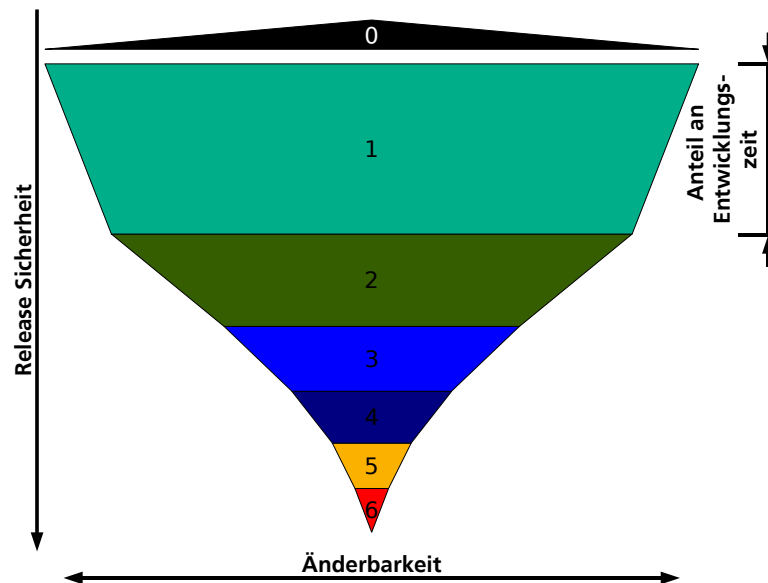


Abbildung 5.8: Die sieben möglichen Projektstatusse.

Die Zustände im Einzelnen sind:

#### 0. *Already Broken*

Im Zustand *Already Broken* ist die Software in der Regel von einem Typ 1 Zufluss betroffen und zur Zeit nicht funktionsfähig. Im schlimmsten Fall compiled der Code nicht. Der Status 0 ist grundsätzlich zu vermeiden und taucht nie in der Planung auf. Zuflüsse vom Typ 1 sind daher nicht zulässig, und müssen nötigenfalls aufgeteilt werden, wie in Abschnitt 6.1, Seite 79 beschrieben. Dennoch sind auch hier nur Zuflüsse der Typen 2-5 ohne vorherige Rücksprache möglich.

#### 1. *Normal*

Dies ist der grundsätzlich zu bevorzugende Zustand. Hier sind Zuflüsse der Typen 2-5 ohne vorherige Rücksprache möglich, Zuflüsse vom Typ 1 (non switchable breaker) sind nach wie vor unerwünscht und

erfordern wenigstens eine gesonderte Rücksprache mit dem gesamten Team.

2. *Switchable Breakers Only*

In diesem Fall besteht eine erste Einschränkung darin, dass Zuflüsse vom Typ 1 gänzlich verboten sind. Hier ist die erreichbare Entwicklungsgeschwindigkeit noch relativ hoch, ohne das eine gänzliche Unsicherheit über die Lauffähigkeit der Software besteht.

3. *Non Breakers Only*

Im diesem Zustand dürfen selbst abschaltbare Probleme nicht mehr hinzugefügt werden. Dies verhindert beispielsweise dass ein Kunde diese aus Versehen wieder einschaltet. Dies stellt eine deutlichere Einschränkung der Geschwindigkeit dar, verhilft aber zu einem nur geringen Sicherheitsgewinn im Vergleich zu switchable breakers only.

4. *Trivial Bugfixes*

Trivial Bugfixes erlaubt nur noch Zuflüsse der Typen 4 und 5. Dies macht eine Weiterentwicklung der Software in Bezug auf neue Features gänzlich unmöglich, gibt aber auch eine schon sehr hohe Sicherheit.

5. *Documentation & String Changes*

Dies ist der letzte Produktive Zustand. Hier sind nur noch kosmetische Änderungen zulässig. Dies gefährdet die Software-Qualität sicher nicht, verhindert aber auch jede positive funktionale Änderung.

6. *None*

Der Zustand ist nur notwendig, wenn beispielsweise während eines aufwendigen Build-Prozesses keine Änderungen zulässig sind. Die Entwicklung des Hauptflusses steht hier gänzlich still.

Es empfiehlt sich, den aktuellen Status in der Vorlage der Commit-Message des Versionsverwaltungssystems zu hinterlegen. Dadurch sieht der Entwickler spätestens dann, ob sein Commit zulässig ist oder nicht. Sind keine (non) switchable breaker Teil des aktuellen Hauptstromes, befindet sich das Projekt an einer Furt, und es wird daher leichter, ein Release auszukoppeln. Aus Effizienzgründen sollte der Anteil an der Entwicklungszeit der Zustände 1 und 2 mit Abstand am größten sein, und daher die Typen 3-6 nur für sehr kurze Zeitintervalle festgelegt werden.

## 5.5 Release-Auskopplung

Wie ein Release zu planen ist, wurde bereits in den vorhergehenden Kapiteln angerissen: Zunächst wird, falls nötig, die Menge der neuen User Stories definiert. Wenn diese Schwankungen unterliegt, muss festgelegt werden, ob die Menge der User Stories oder der Release-Zeitpunkt geändert werden kann. Beides einhalten zu wollen ist im Allgemeinen illusorisch, wie hinreichend in der Literatur belegt (z. B. [Bro75]): An der Metapher lässt sich dieses Dilemma erklären: Man kann beim Fluss entweder den Abstand zur Quelle (=Zeitpunkt) oder den Volumenstrom (=Featuremenge) wählen. Ändert man das Eine, muss das Andere auch geändert werden.

Sind keine besonderen Features gewünscht, sondern lediglich eine hinreichende Stabilität gefordert, kann dieser erste Block übersprungen werden. Nach der Definition werden die User Stories wie in Abschnitt 5.2, Seite 56 beschrieben auf ihren Aufwand hin untersucht und in einem Release Burndown Chart addiert. Dieser liefert dann auf Basis der vergangenen Werte auch einen möglichen Release-Zeitpunkt. Sind alle Stories verarbeitet wird das Projekt schrittweise in einen höheren Status versetzt und dort je nach gewünschter Testintensität gehalten. Falls Akzeptanztests definiert worden sind, werden diese nun durchgeführt. Je nach Leidenschaft und technischer Versiertheit des Kunden kann es nötig sein, alle switchable breakers zu entfernen, bevor ausgeliefert wird. Sind keine vorhanden, befindet man sich sowieso an einer Furt, und das Release kann dementsprechend schneller erfolgen.

Nach erfolgreichem Abschluss aller Tests kann die Software ausgeliefert werden. Dabei wird dokumentiert: Kunde, Anforderungen inklusive Akzeptanztests, aktuelle Versionsnummer<sup>2</sup> aus dem Versionsverwaltungssystem sowie der aktuelle Stand der Dokumentation. Diese Notiz sollte werkzeuggestützt erfolgen und als eine Einheit abrufbar sein, beispielsweise als Wiki Snapshot, Details dazu in Abschnitt 6.6, Seite 88.

Diese Notiz ist extrem wichtig, wenn der Kunde später Fehler in der Software meldet. So ist der Snapshot der reinen Software selbstverständlich

---

<sup>2</sup>Bei Versionsverwaltungssystemen mit einer globalen Versionsnummer reichte diese, ansonsten muss ein Tag gesetzt werden.

notwendig, um den Fehler nachzuvollziehen. Aber auch die Dokumentation, hier vor allem die Architekturdokumentation, ist notwendig wenn die Version des Kunden schon älter ist, und ein systematischer Fehler in der Architektur nachvollzogen werden soll. Nachdem der Fehler analysiert worden ist, kann, wie in Kapitel 5.3 beschrieben, entschieden werden, ob und gegebenenfalls wann er behoben wird. Es ist auf jeden Fall zu bevorzugen, den Fehler in der neuesten Version der Software zu beheben. Die extreme Dynamik in der Entwicklung macht es sehr schwer, mehrere parallele Versionen zu halten und zu pflegen, vor allem da viele Fehler Modellfehler sind, die sich nicht einfach in die alte Version des Kunden backporten lassen [Leh96]. An dieser Stelle kommen die Akzeptanztests erneut ins Spiel: Bevor der Kunde die aktuelle Version der Software bekommt muss sichergestellt werden, dass auch seine Anforderungen damit noch erfüllt werden. Die Tests werden also erneut benötigt. Ansonsten verläuft der Release-Prozess wie oben beschrieben ohne neue User Stories ab.

Ist es unvermeidlich, dass eine parallel gehaltene Version an den Kunden im Fehlerfalle ausgeliefert wird, und es stellt sich heraus, dass der Kunde auf einen inzwischen unbewusst behobenen Fehler gestoßen ist, so kann mittels einer einfachen Bisektionssuche im Versionsverwaltungssystem die entsprechende Änderung gefunden werden. Ebenso kann damit auch die Ursache eines nicht mehr erfolgreichen Akzeptanztests gefunden werden. Wäre allerdings ein Modellfehler zu beheben, kann der Aufwand eines Backport schnell unverhältnismäßig hoch werden. Um dies zu vermeiden, empfiehlt es sich daher, die Akzeptanztests des Kunden regelmäßig während der Entwicklung ab dem Zeitpunkt seines Releases durchzuführen. Dies erhöht natürlich die Kosten des Releases und sollte entsprechend bedacht werden. Andererseits trägt der zusätzliche Testaufwand zur Erhöhung der Software-Qualität bei.

## 5.6 Qualitätsmanagement

Der vorgestellte Entwicklungsprozess hat eine sehr hohe Dynamik und lässt dem einzelnen Entwickler relativ viel Freiheit. Daher ist es notwendig, ein entsprechendes Qualitätsmanagementsystem zu verwenden, das permanent für die Einhaltung der Qualitätsanforderungen sorgt. Der gewünschte

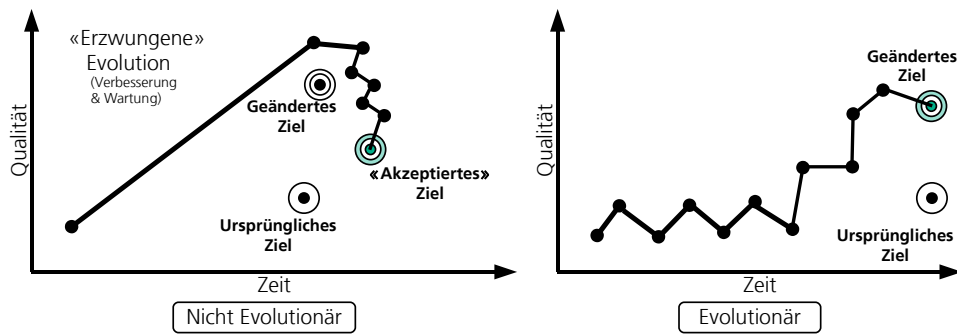


Abbildung 5.9: Stetige Qualitätssicherung bei evolutionärer Entwicklung.

Effekt ist in Abbildung 5.9 dargestellt [Gil85]: So wird die Qualität ständig verbessert, und soll nicht wie bei einem Wasserfallansatz einmal erreicht werden, um dann permanent schlechter zu werden.

Um die Anforderungen zunächst festzulegen ist in Abbildung 5.10 ein Qualitätsbaum [BBL76, Lic07] dargestellt, in dem die im Forschungsumfeld besonders wichtigen Software-Qualitäten hervorgehoben sind. Im Einzelnen sind dies, in der Reihenfolge ihrer Bedeutung:

- *Know-How-Gewinn*

Wichtigstes Qualitätskriterium ist der Know-How-Gewinn, hier allerdings im nicht softwaretechnischen Bereich, der während der Entwicklung entsteht. Dies ist unmittelbar einsichtig, da genau aus diesem Grund die Software entwickelt wird.

- *Bausteingewinn*

Der Bausteingewinn ist für eine nachhaltige Entwicklung, die auch in unbestimmte Richtungen verlaufen soll, wichtig. Denn nur wenn die Komponenten der Software entsprechend flexibel verwendbar sind, wird wenig unnötiger Entwicklungsaufwand eingesetzt.

- *Termineinhaltung*

In Bezug auf Releases an Kunden ist es wichtig, geplante Termine auch einzuhalten. Selbstverständlich ist im Einzelfall zu bewerten, ob überhaupt und wenn wie wichtig dem Kunden der Termin ist.

- *Prozesstransparenz*

Damit die Mitarbeiter schnell in den Prozess hineinfinden, ist eine hohe Transparenz wünschenswert. Undurchsichtige Strukturen verhin-

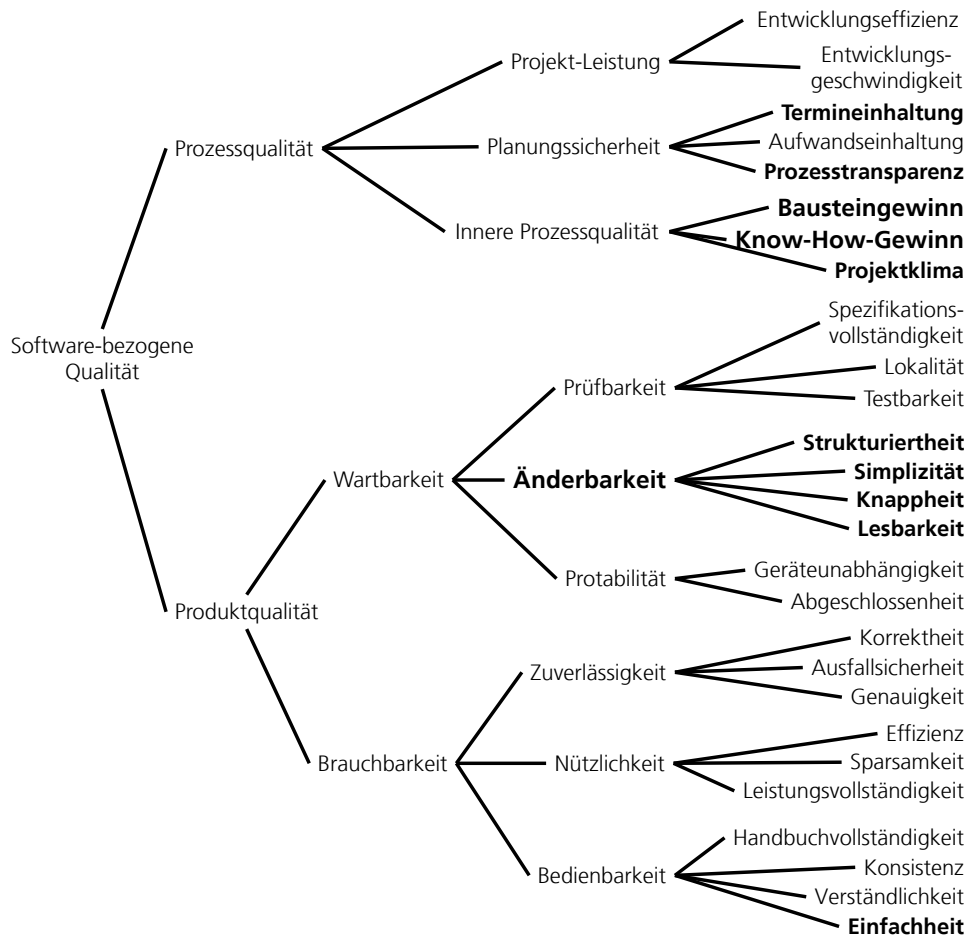


Abbildung 5.10: Wichtige Qualitäten im Qualitätsbaum.

dern eine schnelle Adaption des Prozesses und damit eine effiziente Entwicklung. Außerdem kann nur ein transparenter Prozess bewertet und optimiert werden.

#### ■ Änderbarkeit

Auf der technischen Seite ist die Änderbarkeit fundamental wichtig. Nur wenn die Software schnell, effizient und ohne große Nebenwirkungen zu ändern ist, kann auf sich ändernde Anforderungen angemessen reagiert werden.

- Kernaspekt der Änderbarkeit, vor allem auf lange Sicht, ist die *Strukturiertheit*. Eine gute Struktur verhindert im Falle einer Änderung größere Komplikationen.
- Die *Lesbarkeit* der Software ermöglicht insbesondere bei Mitarbeiterwechseln das schnelle Zurechtfinden der Neuen, aber

auch das Ändern von fremden Code wird hierdurch erleichtert. Dies reduziert wiederum die Zeit, die benötigt wird um auf eine geänderte Anforderung zu reagieren.

- Nach Möglichkeit sollte die Software über eine hinreichende *Simplizität* verfügen. Einfache Lösungen sind komplizierteren vorzuziehen. Eine Ausnahme existiert lediglich dann, wenn eine kompliziertere Lösung spätere Änderungen vereinfacht. Hier muss allerdings zwischen einer rein technisch vorgesehenen Änderbarkeit und einer solchen, die auch die Anwendungsebene mit einbezieht, unterschieden werden. Letzteres ist nicht wünschenswert, da nicht praktikabel. Zensierter Abschnitt
- Wo möglich ist auch hohe *Knappheit* zu erzielen. Was kurz formuliert ist, kann schneller erfasst werden. Allerdings dürfen Lesbarkeit und Simplizität hierunter auf keinen Fall leiden.

#### ■ *Einfachheit der Bedienung*

Aus Sicht der Benutzer wird nur eine einfache Bedienung gefordert. Dies sollte im Allgemeinen genügen, um ein Arbeiten mit der Software zu ermöglichen, ohne zuviel zusätzlichen Aufwand zu erfordern. Des Weiteren wies Khramov nach, dass Einfachheit für einen Produkterfolg wichtiger ist als etwa Code-Qualität [Khr06].

#### ■ *Projektklima*

Für einen jeden nachhaltigen Projekterfolg sind die Mitarbeiter zentrales Element. Daher muss das Projektklima besonders gepflegt werden, hierauf geht Abschnitt 5.7, Seite 74 im Detail ein.

Die anderen Qualitätskriterien sind entweder für die Software-Entwicklung im Forschungsumfeld nicht so wichtig, so z.B. Ausfallsicherheit und Genauigkeit. Oder aber sie sind schlicht unter vertretbarem Aufwand nicht zu erreichen, z.B. Verständlichkeit und Aufwandseinhaltung.

Zur Qualitätssicherung ist es notwendig, einen Überblick über die Qualität des Projektes zu erhalten. Im Kontext des Flusses braucht man also ein Art Cockpit von dem aus man eine Karte des Flusses bekommt, auf der mögliche Problemstellen schnell zu erkennen sind. Bei nicht technischen Qualitätskriterien, wie Know-How-Gewinn oder Projektklima ist dies natürlich nicht mit einfachen Werkzeugen möglich. Die Termineinhaltung kann

im Burndown Chart direkt abgelesen und auch damit gesteuert werden. Die Prozesstransparenz sollte weitgehend direkt aus dieser Arbeit folgen und wird auch durch den Prozessüberwacher (Abschnitt 5.7, Seite 76) kommuniziert. Ob die Bedienung hinreichend einfach ist, kann von den eng eingebundenen Anwendern direkt bewertet werden. Zusätzlich sollten aber regelmäßig mit der Software nicht vertraute Dritte hinzugezogen werden, um die Bedienbarkeit zu bewerten.

Die Änderbarkeit als zentrale Anforderung an die Software-Qualität ist wiederum relativ gut messbar. Über die Jahre wurden verschiedene Metriken vorgeschlagen, die zusammengefasst hiervon ein Bild liefern können. Grundsätzlich ist anzumerken, dass immer nur die Änderung einer Metrik über den Verlauf der Zeit ein Indiz liefern kann, und auch nie zwingend kausal mit einer Veränderung des zu Messenden zusammenhängt. Die Korrelation zwischen Metrik und Gemessenem, hier also der Änderbarkeit, ist zwar gegeben, darf aber auch nicht überstrapaziert werden. Eine manuelle Untersuchung ist immer notwendig. Zur Überwachung der Änderbarkeit wird hier der Maintainability Index vorgeschlagen. Zu seiner Berechnung siehe Abschnitt 6.6.4, Seite 92.

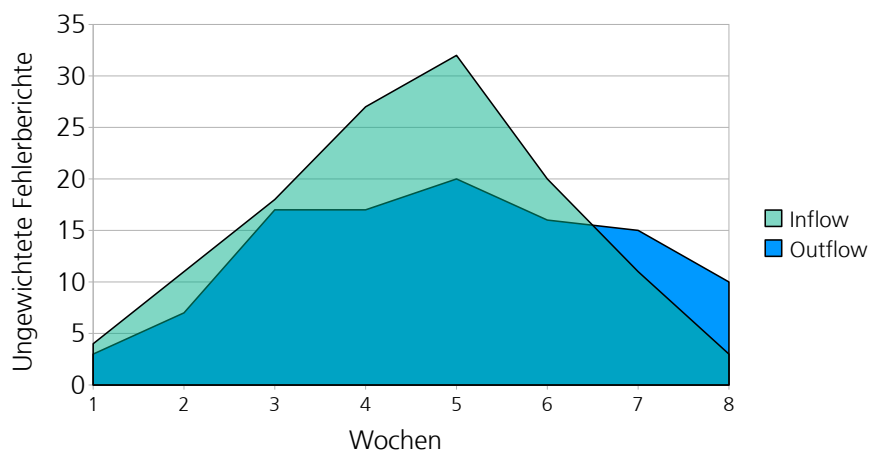


Abbildung 5.11: Metriken zur Überwachung der Fehlerbehebung.

Eine permanente Überwachung der Architektur kann mit Software-Tomographie-Werkzeugen erreicht werden, z. B. dem Sotograph [Sofol]. Zur weiteren Überwachung der Qualität bietet es sich an Metriken der Fehlerberichte zu erheben: Zum einen die Rate der neu hinzugekommenen und der erledigten

Bugs, zum anderen die Rückfälligkeit [Coh02]. Dabei sind auch wieder weniger die absoluten Werte, als vielmehr eine Entwicklung über die Zeit von Interesse.

Des Weiteren sollen permanente Reviews dazu dienen, die Software-Qualität einzuhalten. Jedes Modul hat dazu einen Verantwortlichen, der alle Änderungen der anderen Entwickler an diesem Modul reviewed. Außerdem bietet es sich an, die Entwickler reihum zusätzliche Reviews machen zu lassen. Auf detaillierte Tests, vor allem im Unit Bereich, wird hier weitgehend verzichtet. Wie im Detail zu verfahren ist, wird in Abschnitt 6.5, Seite 86 dargestellt.

Um technische Risiken einschätzen zu können, oder die Machbarkeit festzustellen, werden gerne Proof-of-Concept-Stücke, beispielsweise die aus XP bekannten Spikes, eingesetzt. Bei diesen Wegwerfprototypen muss die Qualität der Lösung zurückstecken, hier investierte Zeit wäre nutzlos und verleitet den Programmierer möglicherweise sogar dazu, Teile dieses Wegwerfcodes nacher weiter zu verwenden.

## 5.7 Mitarbeitermanagement

Mitarbeiter sind wohl das wichtigste Erfolgskriterium eines Projektes, allem die Kommunikation untereinander ist extrem kritisch [For95, DSS04]. Auch der Kunde muss zwingend eng in die Kommunikation eingebunden werden, wenn das Projekt erfolgreich sein soll. Dies ist im Forschungsumfeld einfach, da der Kunde, hier also der Anwender im Unternehmen, beinahe immer greifbar ist, so dass etwaige Unklarheiten in den Anforderungen, zumindest wenn sie nicht auf Unwissenheit des Anwender beruhen, schnell geklärt werden können. Bei externen Kunden ist die Problematik natürlich dennoch gegeben. Hier kann nur erneut auf eine möglichst intensive Kommunikation hingewiesen werden.

Die Selbstverpflichtung der Mitarbeiter zum Projekt, neudeutsch Commitment genannt, ist ein kritischer Faktor für den Erfolg eines Software-Entwicklungsprojektes. Nur ein motivierter Mitarbeiter mit einem hohen Commitment verhindert schlechten Code oder, um den XP Begriff zu verwenden, Code Smell [PW06]. Aus der Summe aller Commitments folgt

ein entsprechendes Projektklima, das in Abschnitt 5.6, Seite 69 schon als wichtig herausgestellt wurde. Da die Möglichkeiten, beispielsweise über Gehaltsboni, eine Mitarbeitermotivation zu erzeugen möglicherweise im Forschungsumfeld eingeschränkt sind, müssen anderen Methoden besonders in Erwägung gezogen werden. Eine Unterstützung in der Karriereplanung oder gemeinsame gesellschaftliche Aktivitäten bieten sich hier als nicht technische Möglichkeiten an. Auf technischer Seite ist es beispielsweise möglich, für jeden kritischen Aufgabenbereich einen verantwortlichen Entwickler zu ernennen, der neben seiner normalen Tätigkeit noch die Arbeiten in einem speziellen Bereich überwacht. Dubinsky und Hazzan haben mit einem derartigem Vorgehen gute Erfahrungen gemacht [DH04b]. Dieses Vorgehen stärkt die vom Mitarbeiter empfundene Verantwortung, und diese erhöht das Commitment [vN02]. Außerdem ist es grundsätzlich vorzuziehen, Probleme von Einzelnen lösen zu lassen, wenn nicht zwingend eine Gruppe erforderlich ist. Dies reduziert die Gefahr des Social Loafing bei unklaren Zuständigkeiten und die Risikofreude [vN02].

Ob jemand die in seiner Rolle spezifizierten Aufgaben selber erledigt oder dies delegiert und nur die Prüfung überwacht, hängt sicher von der Projektgröße ab. Beides ist denkbar. Es bieten sich folgende Rollen an:

■ *End-User*

Der End-User-Verantwortliche prüft vor der Auslieferung eines Releases, ob erzeugte Lösungen auch den von den Anwendern spezifizierten Wünschen entsprechen. Außerdem koordiniert er die Kommunikation mit externen Kunden und sorgt dafür, dass interne wie externe Anwender stets mit aktuellen Versionen versorgt werden.

■ *Dokumentation*

Auch das Erzeugen der geforderten Dokumentation muss überwacht und koordiniert werden. Das erledigt der Dokumentationsverantwortliche nach den in Abschnitt 6.4, Seite 83 geforderten Leitlinien.

■ *Code-Standards*

Die Einhaltung von vereinbarten Code-Standards überwacht, für gewöhnlich mit entsprechenden Werkzeugen oder durch Überwachung von Reviews, ein weiterer Verantwortlicher. Außerdem pflegt er die entsprechende Dokumentation und weist neue Mitarbeiter ein.

### ■ *Effizienz*

Ein Effizienzverantwortlicher prüft regelmäßig die Geschwindigkeit der Applikation und weist bei Bedarf die jeweiligen Entwickler an, Programmteile zu beschleunigen. Die Geschwindigkeitsmessungen können auch bei den sowieso regelmäßig, also täglich oder nach jedem Commit, ablaufenden Tests erfolgen. Die Arbeitsgeschwindigkeit der Benutzeroberfläche muss natürlich dennoch von Hand getestet werden, insofern bei der zu entwickelnden Software in diesem Bereich Probleme auftreten können.

### ■ *Installation*

Der Installationsverantwortliche sorgt dafür, dass neue Teile in den Installationsprozess der Software aufgenommen werden. Und dafür, dass nichts mitinstalliert wird, was bei einem Endkunden nicht ankommen darf.

### ■ *Prozess Überwachung*

Ein Prozessüberwacher prüft den Entwicklungsprozess auf seine Geschwindigkeit und meldet etwaige Probleme. Er koordiniert die schwankende Arbeitszeit, um bei einer Besprechung die notwendige Planungssicherheit zu haben. Er muss die Mitarbeiter dazu anhalten, die Erfassung der Arbeitszeit je Change Request gewissenhaft durchzuführen. Dazu muss kommuniziert werden, dass eine lange Bearbeitungszeit nichts Negatives ist, und die Messung nicht zur Bewertung der Mitarbeiter eingesetzt wird.

### ■ *Gesamtarchitektur*

Von zentraler Bedeutung ist ein Verantwortlicher für die Gesamtarchitektur. Ebenso wie der Integrationsverantwortliche muss er einen sehr guten Überblick haben. Er sorgt für die Einhaltung der obersten Architekturschicht, die über den einzelnen Modulen steht, und bringt damit eine Konsistenz in das Gesamtsystem. Des Weiteren unterstützt er die Modulverantwortlichen in Architekturfragen. Um diese Aufgaben erledigen zu können, ist neben einer guten Übersicht über die Software auch eine recht aufwendige Einarbeitung in die entsprechenden Werkzeuge (Siehe Abschnitt 6.6.4, Seite 92) nötig. Daher ist es sinnvoll diese Position langfristig zu besetzen.

### ■ *Integration*

Ein Integrationsverantwortlicher koordiniert den Zufluss der einzelnen Teilflüsse. Dazu benötigt er einen entsprechenden Überblick über die Gesamtarchitektur. Denkbar ist auch, vor allem bei größeren Projekten, dass nicht jeder Entwickler berechtigt ist zu commiten, sondern diese Aufgabe vom Integrationsverantwortlichen an ausgewählte Entwickler delegiert wird. Hierbei kann auch gleichzeitig ein weiteres Review des Codes und der Architektur vorgenommen werden.

Darüber hinaus macht es Sinn, jedem *Modul* des Projektes einen Verantwortlichen zuzuteilen. Ein Modul kann eine Klasse oder mehrere Packages umfassen, je nach Bedarf. Wichtig ist eine einfache Ermittlung des Zuständigen. Dennoch ist es selbstverständlich wünschenswert, dass andere Entwickler fremden Code ändern. Der Zuständige prüft aber grundsätzlich *alle* Änderungen an seinem Code bei seinem nächste Update aus dem Versionsverwaltungssystem. Dies erzeugt Verantwortung des Mitarbeiters gegenüber dem spezifischen Modul und erhöht so sein Commitment [vN02] und seine Motivation [BD97]. Nebenbei wirkt sich diese recht strikte Modularisierung positiv auf die Änderbarkeit der Software aus. Ein Collective Ownership wie bei XP gefordert wird sich im Forschungsumfeld nicht durchsetzen lassen, da das Verhältnis von Quelltextmenge zu Entwicklern dafür zu schlecht ist.

Offensichtlich werden die Entwickler häufig mehrere Rollen gleichzeitig ausfüllen. Diese sollten dann so gewählt sein, dass sie nicht kollidieren, sondern im besten Fall noch Synergieeffekte erzielen. Die zusätzliche, übergeordnete Rolle eines *Vorgesetzten* gibt für gewöhnlich die Betriebsstruktur vor. Seine Aufgaben ergeben sich ebenso. An ihn sind des Weiteren alle nicht im Team lösbaren Probleme zu melden. Der Vorgesetzte sollte bestenfalls im Sinne eines Scrum Masters bemüht sein, alle Hindernisse (Impediments), die den Entwicklern im Wege stehen, zu beseitigen.

Wenn Menschen zusammenarbeiten sollen, ist es vorteilhaft Temperamente und Persönlichkeitstypen zu mischen. Sfetso et al. haben nachgewiesen, dass die Menge der Kommunikation zwischen den Mitarbeitern nur dann positiv mit ihrer Produktivität korreliert, wenn sie verschiedene Persönlichkeitstypen haben[SSAD06]. Ihre Analyse kann auch bei gemeinsamen

gesellschaftlichen Veranstaltungen vorgenommen werden. Wie wichtig diese Aspekte sein können zeigt u.a. das Beispiel Google [Goo08a, Goo08b]

# Kapitel 6

## Prozessdetails

### 6.1 Große Umbauten

Unter gewissen Umständen lassen sich größere Änderungen an der Software nicht vermeiden. Im Folgenden sei eine Änderung groß, wenn sie sich nicht binnen eines Tages realisieren lässt. Der Zeitraum, der für wirklich große Änderungen nötig ist, kann natürlich leicht auch Wochen oder Monate umspannen. Sie können nötig werden, weil größere Änderungen an der Architektur nötig sind, oder auch überall verwendete Komponenten eine Änderung erfahren. Derart große Umbauten sind konjunktive Problemlösungsaufgaben [vN02], daher ist eine Problemlösung in der Gruppe vorzuziehen und notwendig: Sie betreffen viele Entwickler und daher bedarf es hier einer besonderen Koordination. Bildlich gesprochen müssen auch die nächsten Zuflüsse angepasst werden, wenn ein Deich den Hauptstrom umlenkt.

Solche Änderungen können entweder nicht direkt en bloc in die Versionsverwaltung übernommen werden, oder führen dazu, dass die Software in den Status 0, also defekt, wechselt. Dies ist natürlich zu vermeiden. Ein nicht Committen bremst die weitere Entwicklung massiv aus und führt ziemlich sicher zu Konflikten beim einem abschließenden, großen Commit. Es muss daher versucht werden, auch große Umbauten in den täglichen Entwicklungsprozess mit einzubeziehen. Gerade wenn diese Umbauten Kernbestandteile der Software außer Betrieb setzen, stellt dies ein Problem dar. Um den Konflikt zu lösen, ist es notwendig, die Änderungen und Umbauten

zu planen und in der Gruppe zu kommunizieren. Je umfangreicher dabei eine Änderung ist, desto größer muss sie kommuniziert und geplant werden. Lippert führt dazu das Konzept eines elektronischen Refactoring Plans ein [Lip04]:

Die anstehenden Änderungen werden zunächst in der Gruppe diskutiert und in kleinere Teile zerlegt. Ziel muss hier sein, dass jeder Teil innerhalb eines Tages zu erledigen ist und die Software den Status 0 nie erreicht, sondern immer lauffähig bleibt. Dies erzeugt zwar einen gewissen Overhead beim Refactoring, vermeidet aber Reibungsverluste zwischen den Entwicklern. Des Weiteren werden die geplanten Änderungen in einem – vorzugsweise elektronisch ausgeführten und für alle Beteiligten leicht einsehbar – Plan dokumentiert. Hier tragen die Beteiligten auch während des Refactorings den aktuellen Fortschritt ein, damit die betroffenen Entwickler stets informiert sind, ob eine Änderung bereits durchgeführt ist, bzw. wo sie aktuell eine gewünschte Funktionalität finden. Änderungen am Plan sind natürlich – nach Absprache mit allen Beteiligten – möglich und unvermeidlich. Der Plan kann beispielsweise in dem Wiki System geführt werden, welches auch die Dokumentation enthält, siehe Abschnitt 6.6, Seite 88, sowie Abschnitt 6.4, Seite 83. Der Ablauf ist dann im Detail wie folgt:

1. Die Gruppe diskutiert und zerlegt den Umbau in kleine Arbeitsschritte, diese werden geeignet notiert, zum Beispiel im Wiki.
2. Danach wird im Wiki notiert, dass sich ein Abschnitt zur Zeit in Bearbeitung befindet. Die anderen Entwickler können dann vor Arbeitsbeginn dort nachschauen, um an den entsprechenden Stellen keine (kollidierenden) Änderungen vorzunehmen. Die Aktualisierung des Eintrages findet immer nur vor oder nach der eigentlich Arbeitszeit statt. Bei geographisch verteilten Teams sind hier genaue Zeiten festzulegen.
3. Die Änderungen des Arbeitsschritts werden lokal vorgenommen und anschließend commitet.
4. Ist dies erfolgreich und die Software noch lauffähig, wird im Wiki der Abschnitt als abgeschlossen markiert. Der Prozess beginnt erneut bei Schritt 2.

5. War der Commit nicht erfolgreich, hat ein anderer Entwickler kollidierende Änderungen vorgenommen. Neben der obligatorischen Behebung des Konflikts sollte sichergestellt werden, dass das nicht noch einmal passiert, da hier deutliche Verzögerungen entstehen können. Sollte die Software nicht mehr lauffähig sein, wird versucht diesen Umstand binnen Tagesfrist zu beheben. Ist dies nicht möglich, wechselt der Software-Status (Abschnitt 5.4, Seite 64) auf 0 und das Problem wird kurz im Wiki notiert. Alternativ kann auch der Commit zurückgenommen werden.
6. Sollte die Software im Status 0 sein, wird dies am nächsten Tag schnellstmöglich behoben. Die Behebung hat Priorität vor allen anderen Änderungen.
7. In jedem Fall muss das Problem in der Gruppe diskutiert werden und vermutlich der Refactoring Plan überarbeitet werden. Der Prozess beginnt von vorne.

## 6.2 Anforderungsanalyse

Verschiedene Rahmenbedingungen gestalten die Anforderungsanalyse im Forschungsumfeld zunächst relativ einfach: Da die Anwender selber Forscher sind, sind sie es gewohnt, präzise zu arbeiten und möglicherweise auch nicht natürlichsprachliche Anforderungen zu formulieren. Außerdem sind sie für Rückfragen etc. leicht zu erreichen, da sie im selben Unternehmen arbeiten. Dennoch besteht hier natürlich das Problem, dass das Wissen der Anwender in der Problemdomäne beschränkt ist, da hier ja konkret geforscht wird. Dies löst unter anderem die permanenten Änderungen der Anforderungen aus.

Eine weitere Anforderungsquelle sind externe Kunden. Hier treten alle bekannten Probleme der Anforderungsanalyse auf, da diese für gewöhnlich nicht permanent erreichbar sind, ihre Anforderungen unklar spezifizieren etc.. Da aber noch sehr spezifische Lösungen konstruiert werden – das Produkt befindet sich in der Einführungs- oder höchstens Bowling Alley Phase – ist auch ihre Bereitschaft zur Kooperation entsprechend hoch. Dies sollte für eine intensive Kommunikation und ein häufiges Feedback genutzt

werden. Im besten Fall können sie sich autark neue Versionen der Software besorgen und haben Zugriff auf das CRM-System, so dass sie ihre Wünsche und Probleme ohne viele Umwege kommunizieren können. Das macht ein hinreichend einfaches Release- und Build-System nötig.

Eine dritte Quelle von Anforderungen können auch die Software-Entwickler selber sein. So ist es denkbar, dass sie beispielsweise eine Möglichkeit zur Fehlersuche implementieren, die von den Anwendern nachher auch dankbar für andere Zwecke angenommen wird und sich so als eigenständiges Feature durchsetzt. Zensierter Abschnitt

### 6.3 Design

Das Software-Design muss vor allem die Anforderung nach einer hohen Änderbarkeit umsetzen. Das heißt, dass die Modularisierung besonders wichtig ist, eine Kopplungsreduktion ist immer günstig. Es greift Parnas Law: »Only what is hidden can be changed without risk« [Par72]. Hier ist also Strukturiertheit wichtiger als Einfachheit. Es muss darauf geachtet werden, dass nicht überstrukturiert wird: Häufig wird versucht, Schnittstellen sehr generisch auszulegen. Fast immer wird dabei allerdings genau der Aspekt vergessen, der später als Änderung notwendig wird. Dadurch wird die Schnittstelle später doch neu geschrieben werden. Hier ist daher eine getrennte Betrachtung angebracht: Wenn sich die Generalität gänzlich auf die technische Seite bezieht, kann hier mehr Aufwand investiert werden. Sind allerdings Elemente der Anwendungsdomäne mit einbezogen, ist dies potentiell gefährlich.

Zensierter Abschnitt

Auch bietet es sich an, Teile der Realität, die als sicher gelten (das heißt im Allgemeinen, dass sie nicht Gegenstand der Forschung sind) derart zu modellieren, dass die anderen, risikobehafteten Teile austauschbar bleiben. Die Literatur [Fow02, GHJ04, BMR<sup>+</sup>00] kennt hier entsprechende Architektur- und Entwurfsmuster um z.B. Algorithmen oder allgemein das Verhalten von Modulen leicht wechseln zu können. Dies kommt auch der in Abschnitt 5.4, Seite 64 häufig geforderten Abschaltbarkeit einer Änderung entgegen: Wenn ein Algorithmus leicht austauschbar ist, dann wird dieses

Austauschen auch derart konfigurierbar, dass mehrere Lösungen parallel in der Software zur Verfügung stehen, und daher die Funktionalität nicht beeinträchtigt wird. Eine Änderung kann also leichter als switchable breaker realisiert werden.

Sind Änderungen an der Architektur nötig, entscheidet darüber der jeweilige Modulverantwortliche, bzw. der Verantwortliche für die Gesamtarchitektur (Abschnitt 5.7, Seite 76). Ob ein Zufluss unvorhergesehen die Architektur ändert, überwacht der Integrationsverantwortliche, außerdem natürlich jeder Modulverantwortliche bei seinem Review einer fremden Änderung.

Bei der Entwicklung von Prototypen gibt es zwei große Gruppen: Solche die nur genau einmal eingesetzt und dann nicht weiter entwickelt werden (Wegwerfprototypen), und solche die auch später noch eine Modifikation erfahren. Da Änderungen schon häufig genug vorkommen, und die Anwender möglicherweise prototypische Funktionalitäten annehmen werden, wird hier empfohlen, auf die Entwicklung von Prototypen möglichst zu verzichten, und wenn dann nur Wegwerfprototypen einzusetzen. Bei einem derartigem Nebenfluss kann auf sauberes Design und Architektur verzichtet werden. Hier ist Entwicklungsgeschwindigkeit und Funktion oberste Maxime. Sollte das gewünschte Resultat erreicht werden, kann überlegt werden, wie die Erkenntnisse vernünftig in der Software umgesetzt werden können – dann auch unter dem Aspekt der Änderungsantizipation. Wichtig ist, den Prototyp nicht weiterzuentwickeln. Wenn das angedacht wird, ist es ein sicheres Zeichen dafür, dass die Funktionalität eigentlich in den Hauptstrom gehören sollte.

## 6.4 Dokumentation

Da Know-How-Gewinn oberstes Ziel der Software-Entwicklung im Forschungsumfeld ist, ist es besonders wichtig, das gewonnene Know-How entsprechend zu dokumentieren. Dies ist natürlich zum größten Teil separate Dokumentation, muss aber eng zusammen mit der Software gepflegt werden. Daneben fällt, wie bei jeder Software-Entwicklung, weitere Dokumentation an, die sich nur auf die Software selber bezieht.

Durch die hohe Volatilität der Anforderungen ist es allerdings ungeschickt, Dinge sofort zu dokumentieren, die sich mit hoher Wahrscheinlichkeit sehr zeitnah wieder ändern. Andererseits muss verhindert werden, dass die Dokumentation vergessen oder vernachlässigt wird. Im Folgenden werden dazu zunächst vier Dokumentationsarten unterschieden:

■ *Benutzer*

Die Benutzerdokumentation erklärt die Funktionalität der Software für einen Anwender. Dabei wird weder im Detail auf innere Abläufe der Software, noch auf technische Zusammenhänge der Anwendungsdomäne eingegangen.

■ *Anwendungsdomäne*

Die Dokumentation der Anwendungsdomäne ist das niedergeschriebene Wissen, was mittels der Software entsteht oder aus dritten Quellen stammt. Dazu zählen vor allem auch die Informationen, welche für ein Verständnis des Anwendungsgebietes, und damit natürlich auch zur Entwicklung der Software notwendig sind. Wichtigster Teil dieser Dokumentationsart ist das Begriffsmodell.

■ *Integrierte Dokumentation*

Diese Dokumentation enthält Kommentare direkt im Quellcode, die zur Verwendung desselben notwendig sind. Im weiteren Sinne zählen daher hier auch Bezeichner, Methoden- und Klassennamen usw. dazu. Ebenso eingeschlossen sind Kommentare in den Methoden und Klassen, die zum lokalen Verständnis notwendig sind: Beispielsweise bei Ausdrücken, die wie Fehler aussehen, aber keine sind, oder komplexeren Konstrukten wie etwa ein Duffs Device [Hol05].

■ *Innere Dokumentation*

Hier werden die Abläufe in der Software selber, welche nicht Teil der integrierten Dokumentation sind, beschrieben. Hier wird ein globaleres Verständnis der Zusammenhänge vermittelt. Dazu zählt auch Architektur- und (Software-)Design-Dokumentation, Ablauf- und Automatengraphen sowie sonstige Diagramme.

Die integrierte Dokumentation erfolgt grundsätzlich simultan schon während der Entstehung des Quellcodes. Ohne sie ist ein Stück Code häufig

unbrauchbar, da schon Tage später selbst der Entwickler selber nicht mehr weiß, was hier gemeint war. Ebenso kann innere Dokumentation häufig zeitnah erzeugt werden: Entwirft der Software-Architekt beispielsweise das System in UML, können die Diagramme direkt als zusätzliches Dokumentationsartefakt hinterlegt werden.

Der Aufwand, Benutzer- und Anwendungsdokumentation zu erstellen, kann relativ hoch sein, wohingegen die Nutzung der Dokumentation für gewöhnlich nicht sofort, sondern einige Zeit in der Zukunft erfolgen wird. So werden die Benutzer für eine gänzlich neue Funktionalität, die vielleicht noch nicht einmal geprüft wurde, sicher nicht die Dokumentation zu Rate ziehen. Viel eher wird der jeweilige Entwickler dabei sein, wenn die ersten Tests ablaufen, um etwaige Probleme direkt anzugehen. Stellen sich dann größere Änderungen in der Anforderung heraus, wäre möglicherweise ein großer Teil der Dokumentationsarbeit umsonst gewesen.

Aus den genannten Gründen wird hier für diese Dokumentationsarten eine verzögerte Erstellung vorgeschlagen: Die Entwickler reservieren sich dafür einen bestimmten Anteil der Wochenarbeitszeit, der Anteil sollte je nach Anforderung an die Menge der Dokumentation im Projekt gewählt werden, beispielsweise 20 Prozent. Während dieser Zeit wird die Dokumentation zu dem Code erstellt, der vier Wochen (diese Zeit ist auch nicht fix, sondern kann natürlich angepasst werden) vorher erstellt und seitdem nicht geändert wurde. So wird versucht, keine Arbeit in die Dokumentation von noch sehr volatilem Code zu stecken, und gleichzeitig sicherzustellen, dass die Dokumentation dennoch nicht zu kurz kommt. Das Versionsverwaltungssystem macht es leicht, die entsprechenden Teile zu identifizieren.

Das Begriffsmodell ist für gewöhnlich zum Verständnis sehr wichtig. Für eine hohe Effizienz sollte es im Wiki geführt und mit dem Quelltext verzeigert sein: Wird zum Beispiel ein Begriff durch eine Klasse in der Software repräsentiert, bietet es sich an, vom Wiki einen Hyperlink auf die entsprechende Website des Versionsverwaltungssystems zu setzen, ebenso in der integrierten Dokumentation einen Link auf das Wiki zu setzen.

Die integrierte Dokumentation ist als Quelltext sowieso Teil der Versionsverwaltung. Die innere Dokumentation sollte in einem Wiki oder wenigstens aber in einem Versionsverwaltungssystem abgelegt werden. Hier kann

dasselbe verwendet werden, welches auch für die sonstigen, bisher spezifizierten Aufgaben verwendet wird.

Im Idealfall wird die innere Dokumentation von der Quelltextdokumentation aus verzeigert. So ist es denkbar, im Javadoc einer Klasse einen Hyperlink auf die zugehörigen Diagramme im Wiki zu erzeugen und im Wiki wiederum Hyperlinks auf die Weboberfläche des Versionsverwaltungssystems zu setzen. Diese direkte Verzeigerung erleichtert vor allem neuen Mitarbeitern den Einstieg in die Software erheblich. Für Benutzer- und Anwendungsdokumentation sollte hingegen ein separater Teil des Wiki oder sogar ein zusätzliches System verwendet werden: Will man später den Dokumentationsstand eines Zeitpunktes  $t$  haben, ist ja die entsprechende Benutzer- und Anwendungsdokumentation erst vier Wochen später, zum Zeitpunkt  $t + 4$  entstanden. Es muss daher einfach möglich sein, auch diese zu erhalten.

## 6.5 Test

Wie bereits in Abschnitt 5.6 angedeutet, wird auf detaillierte, granulare Tests verzichtet. Das hat mehrere Gründe:

Zunächst ist das häufig geforderte Testen aller (Rand)-Fälle unter Gegebenheiten, wie sie auch im Forschungsumfeld vorherrschen, zum Erreichen einer hohen Software-Qualität nicht notwendig[Khr06]. Des Weiteren ist es im Forschungsumfeld extrem schwierig, sinnvolle Solldaten gezielt zu generieren: Soll zum Beispiel ein Teil der Realität simuliert werden, sind die einzig sicher richtigen Solldaten natürlich die realen Vorgänge. Wirklich erreichen werden Simulationen – auf Grund der Vereinfachungen, die im Modell vorgenommen wurden – die Werte aber nie. Außerdem kann die Erfassung der Daten und Überführung in Testfälle ein nicht trivialer Prozess sein. Die Software wird nun laufend angepasst, damit ihre Ergebnisse sich möglichst genau diesen realen Werten annähern. Daher ändern sich ihre Ergebnisse aber auch fortlaufend, allerdings nicht zwingend in jedem Fall direkt näher zum Optimum. Dies macht einfache Soll-Ist Abgleiche schwierig. Häufig ist die Komplexität der Berechnung auch derart hoch, dass es nicht wirtschaftlich ist, die Sollergebnisse händisch auszurechnen. Zensierter Abschnitt

Fein granulierte Tests, vor allem Unittests, erzeugen daher relativ viel Entwicklungsaufwand, gerade wenn zusätzlich versucht wird, eine hohe Pfadüberdeckung zu erreichen. Dies wirkt sich wegen der hohen Dynamik in den Anforderungen sehr negativ auf die Entwicklungsgeschwindigkeit aus. Außerdem wird es passieren, dass noch gar nicht alle erreichbaren Pfade von Anwenderseite spezifiziert sind – und auch nicht in näherer Zukunft spezifiziert werden. Das macht zusätzlich eine sinnvolle Bestimmung eines nachprüfbar Testauswahl-Kriteriums schwierig.

Deshalb werden nur Regressionstests verlangt, die grobe Fehler abfangen sollen. Es wird dabei nicht gefordert, ein bestimmtes Sollergebnis zu erreichen, sondern es wird eine Bandbreite von Ergebnissen zugelassen. Braucht ein simulierter Vorgang in der Realität z.B. 10 Sekunden, nimmt man etwa alle Ergebnisse der Software zwischen 8 und 12 Sekunden als korrekt an. Damit werden zwar nur grobe Fehler festgestellt, aber der Testfall muss auch nicht permanent angepasst werden. Die ganz groben Fehler, wie Ausnahmen oder gänzlich falsche Werte, z.B. von einer Division durch null erzeugte Not-a-Number-Werte werden dadurch detektiert. Bestenfalls laufen diese Tests automatisch nach jedem Commit, wenigstens aber täglich [CS95] ab. Der klare Vorteil ist hier, dass auch reale Eingangsdaten verwendet werden können, wobei die Sollergebnisse einfach aus einmalig manuell überprüften Resultaten der Software übernommen werden können.

Eine gewisse Ausnahme bilden natürlich die Akzeptanztests für ein Release. Diese werden, falls nicht automatisierbar, manuell durchgeführt. Andernfalls bietet es sich geradezu an, diese in die automatisch ablaufenden Tests aufzunehmen. Alle Testfälle sollten in der Versionsverwaltung gehalten werden.

Zur Überwachung von Code-Standards sowie der Einhaltung der festgelegten Architektur gibt es entsprechende Werkzeuge, die in Abschnitt 6.6.4, Seite 92 näher beschrieben sind. Grundsätzlich können die Tests natürlich von jedem Entwickler durchgeführt werden, sei es permanent während der Arbeit durch die Testintegration in die Entwicklungsumgebung, oder aber auch nur bei Bedarf. Beispielsweise bietet es sich, an bei einer vermuteten Architekturverletzung diese mit Sotoarc/Sotograph nachzuprüfen. Zusätzlich sind der Verantwortliche für die Gesamtarchitektur sowie der Code-Standards-Verantwortliche besonders in der Pflicht, die Einhaltung der Richtlinien permanent zu überwachen.

## 6.6 Werkzeugunterstützung

Der in dieser Arbeit beschriebene Software-Entwicklungsprozess setzt massiv auf die Unterstützung der Entwicklung durch verschiedene Software-Werkzeuge (CASE). Diese werden in diesem Kapitel zusammengefasst. Zur Erstellung von Dokumenten aller Art werden verschiedenste Editoren oder auch integrierte Entwicklungsumgebungen (IDE) verwendet. Welche konkret zur Anwendung kommen, ist für den Software-Entwicklungsprozess nicht weiter von Bedeutung und wird hier auch nicht spezifiziert. Die Werkzeuge sollten so gewählt werden, dass der Benutzer in seiner Arbeit unterstützt wird, und die Einarbeitung schnell möglich ist. Eine Verzahnung mit den anderen hier beschriebenen Werkzeugen ist natürlich wünschenswert. Der Autor hat gute Erfahrungen mit der Netbeans IDE [Bou02] gemacht, sicher können auch andere Entwicklungsumgebungen erfolgreich eingesetzt werden. Daneben sind weitere unterstützende Werkzeuge denkbar, etwa ein Profiling-Werkzeug zur Messung der Geschwindigkeit und des Speicherverbrauchs (siehe Abschnitt 5.7, Seite 74) oder ein UML-Werkzeug.

Der Autor hat mit den nachstehend genannten Programmen gute Erfahrungen gemacht und empfiehlt sie daher für den jeweiligen Zweck. Das schließt nicht aus, dass andere Lösungen möglicherweise leistungsfähiger sind, es bietet sich daher immer an, den schnelllebigen Markt nach entsprechenden Produkten zu durchforsten. Sicherlich schränkt das Budget die Verfügbarkeit von kommerziellen Lösungen ein. Eine zwar etwas alte aber dennoch sehr lesenwerte Übersicht vieler freier Werkzeuge liefert [ZK03].

### 6.6.1 Versionsverwaltungssystem

Zum wohl wichtigsten Werkzeug bei der Erstellung der Dokumente gehört neben den jeweiligen Editoren das Versionsverwaltungssystem. Auf die vielen Gründe, warum ein Versionsverwaltungssystem einzusetzen ist, wird hier nicht mehr näher eingegangen. Neben den üblichen Funktionen, mehrere Versionen von Dateien zu verwalten, und Änderungen nachzuvollziehen, ist es notwendig, dass die Versionsverwaltung in der Lage ist, eine Konfiguration mit einem eindeutigen Bezeichner zu versehen. Dies erleichtert das spätere Auffinden, wenn ein Release an einen Kunde geliefert

wurde. Außerdem sollte für die in Abschnitt 6.4, Seite 83 abgesprochene Verzeigerung eine Webschnittstelle zur Verfügung stehen. Des Weiteren sollte es möglich sein, die Vorlagen der Commitmessages zu ändern, so dass dort der aktuelle Projektstatus (siehe Abschnitt 5.4, Seite 64) als Information für den Entwickler abgelegt werden kann. Schließlich bietet es sich an, das System an das Change Request Management zu koppeln, so dass dort nachgehalten werden kann, zu welchem Change Request eine Änderung gehört.

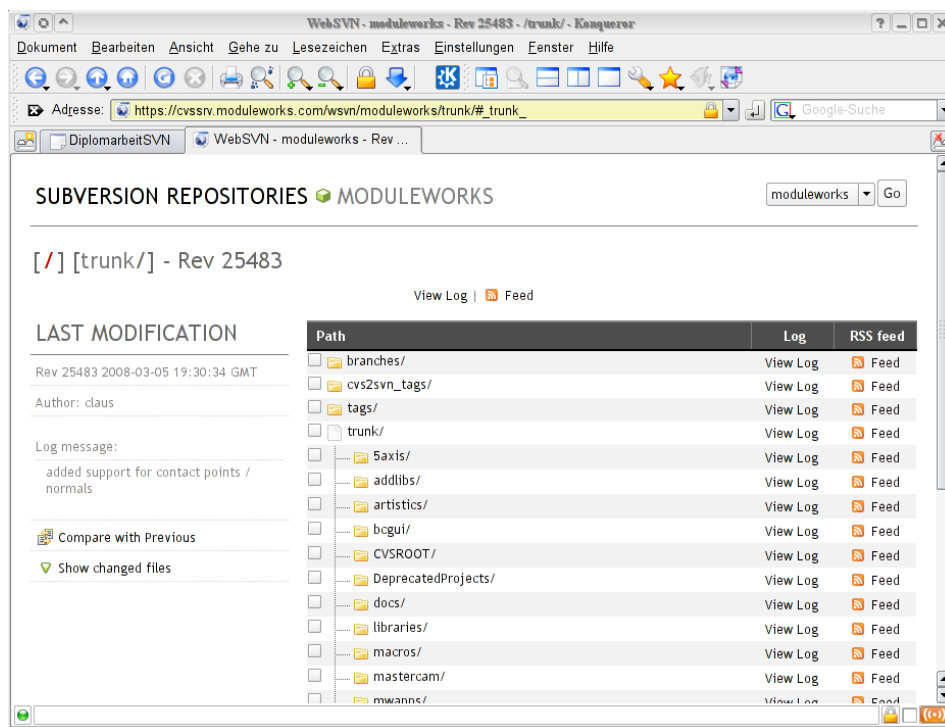


Abbildung 6.1: Subversion Webinterface.

Das bekannteste System dieser Gattung ist wohl CVS [OpeoJa], was wegen verschiedener Schwächen aber nicht mehr empfohlen werden kann. Der ebenfalls freie Nachfolger Subversion [OpeoJd, CSFP04] ist für alle neuen Projekte zu bevorzugen. Zentrales Merkmal ist, dass hier jede Konfiguration der Dokumente eine eindeutige Nummer enthält, so dass häufiges Tagging wie bei CVS entfällt. Das Anpassen der Nachrichtenvorlagen ist hier ebenso möglich wie das Ankoppeln an andere Systeme, da Subversion hierfür entsprechende Hooks bietet. Die standardmäßig mitinstallierte Weboberfläche (Siehe Abbildung 6.1) ermöglicht das Verzeigern aus anderen Anwendungen heraus.

## 6.6.2 Change Request & Project Management

Das Change Request Management System verwaltet nicht nur die Fehlerberichte der Kunden, sondern auch alle sonstigen Änderungen und Anforderungen, die an der Software verwirklicht werden sollen. Daneben muss es möglich sein, die darauf verwendete Arbeitszeit zu erfassen (siehe Abschnitt 5.2, Seite 56). Ebenso wünschenswert ist eine enge Verzahnung mit Versionsverwaltung und Wiki, um Änderungen schneller nachzuvollziehen und neue Mitarbeiter schnellstmöglich einarbeiten zu können. Eine Schnittstelle zum Export statistischer Daten ermöglicht die Erhebung von Metriken über die Fehlerberichte, wie bereits in Abschnitt 5.6 beschrieben.

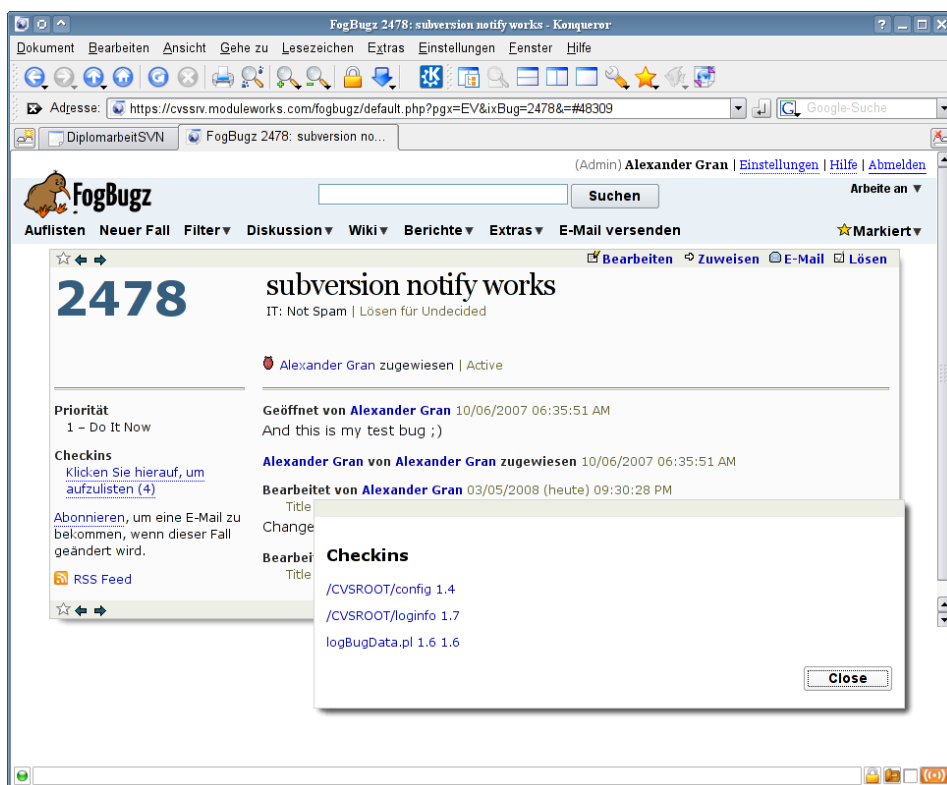


Abbildung 6.2: FogBugz mit Subversion Integration.

Ein Werkzeug, welches die angesprochenen Möglichkeiten bietet, ist FogBugz [Gun07, Fog08]. Es ist in der Lage, neben der Erfassung von Arbeitszeit und Change Request, auch die angesprochene Verzahnung vorzunehmen. So findet sich, wie in Abbildung 6.2 zu erkennen ist, bei

jedem Change Request die zugehörigen Dateiänderungen in der Versionsverwaltung. Die offene und dokumentierte Datenbankstruktur ermöglicht eine relativ einfache Extraktion der enthaltenen Daten zur weiteren Verwendung. So kann beispielsweise das Burndown Chart daraus erstellt werden (siehe Abschnitt 5.2, Seite 56), daneben verwendet FogBugz einen Monte-Carlo-Algorithmus, »Evidence-Based Scheduling« [Spo07] genannt, der möglicherweise<sup>1</sup> alternativ zur Bestimmung des Release-Zeitpunkts verwendet werden kann. Da zusätzlich noch ein Wiki integriert ist, können mit einem einzigen Werkzeug relativ viele Bereiche abgedeckt werden.

### 6.6.3 Wiki

Im Gegensatz zu einer in der Versionsverwaltung gehalten Menge an Dokumenten ist es bei einem Wiki möglich, die Dokumente untereinander effizient zu verzeigern. Dadurch kann das enthaltene Wissen sehr schnell durchforstet werden. Durch die einheitliche Semantik der Informationen ist auch Durchsuchen möglich. Da beinahe jedes Wiki eine Versionsverwaltung bietet, ist es zudem immer möglich, alte Informationen einfach zu erhalten. Damit ähnelt ein Wiki sehr stark Quelltext, der in einer Entwicklungsumgebung bearbeitet wird: Auch hier ist für gewöhnlich eine starke Verzeigerung impliziert durch die Programmiersprache gegeben. Diese Möglichkeiten prädestinieren ein Wiki für die Erfassung der Dokumentation bei der Software-Entwicklung. Hierin wird daher wenigstens die Dokumentation der inneren Abläufe der Software sowie ihre Architektur beschrieben. Die Daten werden möglicherweise auch noch mit zusätzlichen Programmen bearbeitet, deren Dokumente dann natürlich auch einer Versionsverwaltung unterliegen müssen. Außerdem wird das Wiki zur Verwaltung von größeren Refactorings verwendet (siehe Abschnitt 6.1, Seite 79). Zensierter Abschnitt

Die bekannteste Wiki-Software ist sicherlich (Semantic-)Mediawiki [MTA<sup>+</sup>05, WikoJa], für die dank der großen Popularität durch den Einsatz bei Wikipedia mehrere hundert Erweiterungen für beinahe jeden Zweck zur Verfügung stehen. Daneben gibt es zahlreiche weitere Wiki-Software, die größte Übersicht findet sich, wie nicht anders zu erwarten, bei Wikipedia [WikoJb]. Im

---

<sup>1</sup>Software und Algorithmus sind zu neu, um im Detail für diese Arbeit geprüft worden zu sein.

Idealfall ist das Wiki in die bereits bestehende Software integriert, wie es zum Beispiel bei FogBugz oder Trac [Edg07] der Fall ist.

Da im Forschungsumfeld für gewöhnlich nicht sonderlich viele (externe) Kunden existieren, kann das Wiki eingeschränkt auch als Customer Relations Management System eingesetzt werden, falls nicht sowieso ein derartiges (unabhängig von der Software-Entwicklung) existiert. Ein dediziertes System ist natürlich vorzuziehen, nur für ein kleines Software-Entwicklungsprojekt aber wohl übertrieben. Wichtig ist, dass auf jeden Fall die Releases des Kunden incl. Akzeptanztests usw. (siehe Abschnitt 5.5, Seite 68) notiert werden.

#### 6.6.4 Qualitätssicherung

Zur Qualitätssicherung ist es erforderlich, die in 5.6 spezifizierten Qualitätsmerkmale messen zu können. Die Verantwortlichen für Gesamtarchitektur und Code-Standards brauchen Möglichkeiten, um möglichst schnell und übersichtlich, aus einer Art Cockpit, die aktuelle Lage der Qualität zu erfassen und, wenn nötig korrigierende Maßnahmen einzuleiten.

Auf der untersten Schicht existieren inzwischen ausgereifte und in die Entwicklungsumgebungen integrierte Werkzeuge, wie PMD [Cop05, OpeoJb], die Verletzungen von simplen Code-Standards anhand von Regeln aufzeigen und zum Teil sogar automatisch beseitigen können. Metriken können im allgemeinen nicht verwendet werden, um die Fehlerwahrscheinlichkeit von Software vorherzusagen, wohl aber für die Wartbarkeit [ER03, MA07, SGK07]. Daher wurde vorgeschlagen, zur Messung der Wartbarkeit den Maintainability Index *MI* (Ursprünglich [OH92], später [PODB95]) zu erheben. Er wurde auch von Moser erfolgreich zur Qualitätssicherung eingesetzt [MSS07], und ist in der Lage, wie ein Frühwarnsystem schnell auf Refactoringbedarf hinzuweisen. Dieser errechnet sich wie folgt [Van04]:

$$MI := 171 - 5,2 * \ln(\overline{HV}) - 0,23 * \overline{V(g')} - 16,2 * \ln(\overline{LOC}) + 50 * \sin(\sqrt{2,4 * \overline{CM}})$$

Die verwendeten Faktoren wurde empirisch durch mehreren Studien bestätigt [PODB95, OH92]. Es werden folgende Metriken verwendet:

- $\overline{HV}$  durchschnittliches Halstead Volume [Hal77, Van97] der Klassen.
- $\overline{V(g')}$  durchschnittliche erweiterte zyklomatische Komplexität [McC76, MW94, Van00] der Klassen.
- $\overline{LOC}$  Lines of Code, durchschnittliche Anzahl der Quellcodezeilen je Klasse.
- $\overline{CM}$  durchschnittlicher prozentualer Anteil der Kommentarzeilen je Klasse.

Die Metriken werden jeweils als im arithmetisches Mittel über alle Klassen berechnet. Maintainability Index Werte unter 65 gelten als schlecht wartbar, Werte über 85 als gut wartbar. Der *MI* kann sowohl für die gesamte Software, als auch für einzelne Subsysteme errechnet werden. Daher ist es damit möglich, vor allem mit einer Trendanalyse, frühzeitig Problemstellen in der Software zu identifizieren und dann entsprechend zu beheben. Visual Studio ist das einzige bekannte Entwicklungswerkzeug, was eine direkte Unterstützung des *MI* bietet. Mittels erweiterbarer Metrik-Plugins ist aber eine Realisation auch in gängigen anderen Entwicklungsumgebungen möglich.

Zensierter Abschnitt Der *MI* ist relativ nah an der Implementierung angesiedelt, daher können Probleme oder gar Verletzungen der Architektur damit nicht festgestellt werden. Dazu braucht es ein Werkzeug, welches auf einer höheren Abstraktionsebene arbeitet. Ein Beispiel dafür sind Sotoarc und Sotograph [SofoJ, BKL04, Hec07]. Dort kann die Architektur hinreichend abstrakt spezifiziert werden, so dass dann Verletzungen vom Werkzeug erkannt werden können, wie in Abbildung ?? zu erkennen. Sotograph bietet außerdem durch seine offene SQL-Schnittstelle zur Metrikdefinition auch eine Möglichkeit, den *MI* zu berechnen und über die Zeit zu überwachen. Dazu kommt die Möglichkeit, unzählige weitere Metriken zu überwachen und Probleme in der Architektur aufzuzeigen: Zensierter Abschnitt Das Programm erfordert allerdings eine relativ hohe Einarbeitungszeit, so dass es erst bei größeren Projekten lohnenswert erscheint. Mit ConQAT entsteht z.Z. ein Werkzeug, das versucht alle die hier skizzierten verschiedenen Aspekte der Qualitätssicherung zu vereinen [DPS06]. Zensierter Abschnitt Die automatisiert ablaufenden Tests lassen sich für gewöhnlich mit den Bordmitteln des Betriebssystems abwickeln. So kann ein einfaches Shellskript die Software aus der Versionsverwaltung auschecken, den Buildprozess einleiten

und entsprechende Testläufe durchführen. Dafür ist es natürlich notwendig, dass die Software entsprechend steuerbar ist; dies sollte aber leicht umzusetzen sein. Die Benachrichtigung über fehlgeschlagene Testfälle kann per E-Mail erfolgen, wie in Abbildung ?? dargestellt.

## 6.7 Prozesseinführung

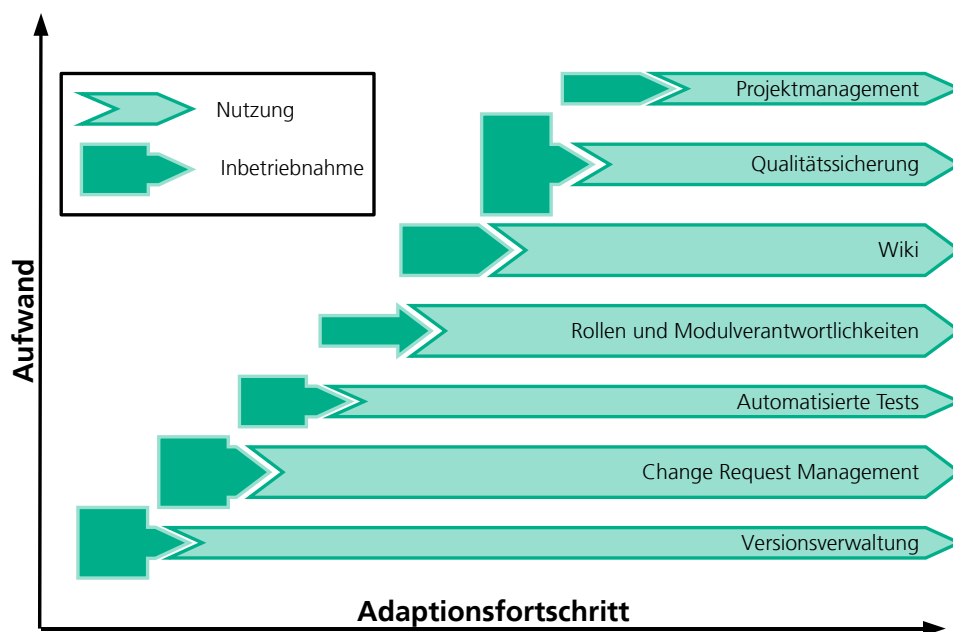


Abbildung 6.3: Schrittweise Adaption des Software-Entwicklungsprozesses.

Da Software-Entwicklungsprozesse auch im Forschungsumfeld für gewöhnlich nicht von Anfang an verwendet werden, sondern vielmehr mit einem Code-and-Fix-Vorgehen begonnen wird, werden in diesem Abschnitt Hilfestellungen gegeben, wie der Software-Entwicklungsprozess schrittweise eingeführt werden kann. Da der Prozess diverse Werkzeuge benötigt, müssen diese natürlich zuerst in Verwendung sein, bevor der Prozess vollständig umgesetzt werden kann. Die Einführung ist in Abbildung 6.3 zusammengefasst: Die Breite der Pfeile gibt jeweils den Aufwand für die Inbetriebnahme bzw. den Betrieb eines Aspektes an, dementsprechend ergibt sich der Gesamtaufwand als Summe der Pfeilbreiten.

Sofern noch nicht erfolgt, bietet es sich zunächst an, ein Versionsverwaltungssystem zu etablieren. Es ermöglicht, alle weiteren Änderungen an der Software nachzuvollziehen und ist daher extrem wichtig, wenn das Projekt wächst. Dies erzeugt bei der Inbetriebnahme relativ viel Einstellungsarbeit, bis alle Entwickler es in ihre Entwicklungsumgebung integriert haben und der Zugriff geregelt ist.

Im Anschluss sollte das Change Request Management in Betrieb genommen werden. Hier können dann mit und mit die Anforderungen eingetragen werden: Immer wenn der Entwickler Arbeitszeit erfassen will und feststellt, dass es keinen entsprechenden Punkt gibt, liegt es nahe, diesen einzutragen. Auch die Anwender im Unternehmen sowie externe Kunden können bereits schrittweise angebunden werden. Durch den möglichst frühen Beginn der Arbeitszeiterfassung ist später die Datenbasis zur Aufwands- und Zeitschätzung größer.

Wenn das CRM bereit ist Fehlerberichte aufzunehmen, sollten die automatischen Tests in Betrieb genommen werden. Je nach verwendetem System können diese Fehler auch direkt an das CRM melden. Im nächsten Schritt sollten die Zuständigkeiten der Mitarbeiter für Module und Aufgaben verteilt werden, damit sie sich dort einfinden können. Natürlich sind, solange noch nicht alle Werkzeuge zusammenarbeiten, nicht alle Arbeiten sinnvoll durchführbar. Dieser Schritt sollte nicht vor der CRM-Einführung erfolgen, weil sonst anstehende Arbeiten nicht vernünftig erfasst und delegiert werden können.

Wird als nächstes das Wiki eingerichtet, kann dort Schritt für Schritt die Dokumentation eingefügt werden. Außerdem sind nun alle Komponenten aktiv, um den Projektstatus festzuschreiben und die Entwickler dadurch besser zu koordinieren. Anschließend können die Maßnahmen zur Qualitätssicherung eingeleitet werden, da deren Ergebnisse auch im CRM eingetragen werden und die zuständigen Mitarbeiter definiert sind. Nach einer gewissen Anlaufzeit, bedingt durch die Einarbeitung in die QS-Werkzeuge, ist zu erwarten, dass größerer Refactoringbedarf entsteht. Um diesen umzusetzen sind aber bereits alle Komponenten in Betrieb.

Als letztes kann das Project Management angegangen werden, also die Werte zur Aufwandsschätzung errechnet werden und vielleicht schon ein Release-Zeitpunkt festgelegt werden.



# Kapitel 7

## Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde nach einer Evaluation der Besonderheiten der Software-Entwicklung im Forschungsumfeld und den daraus resultierenden Anforderungen an einen Software-Entwicklungsprozess die »Software-Entwicklung als Fluss« präsentiert. Dabei handelt es sich um eine evolutionäre, parallelisierte Entwicklung, die ganz bewusst vermeidet, detaillierte und beständige Anforderungsspezifikationen zu erfordern. Vielmehr sind alle Bereiche darauf ausgerichtet, durch sich ändernde Anforderungen keinen unnötigen Zusatzaufwand zu erzeugen. Dies wird durch eine verzögerte aber dennoch sichergestellte Erzeugung der Dokumentation und eine umfassende Werkzeugunterstützung mit dem Maintainability Index, Fehlerberichtmetriken und Software-Tomographie zur Qualitätssicherung erreicht. Außerdem ergeben ein gänzlicher Verzicht auf Phasen in der Entwicklung, sowie eine stark verzeigerte Dokumentation die Möglichkeit zur endlosen Entwicklung mit mehreren Entwicklern, ohne dabei eine größere Synchronisierung zu erfordern. Die Möglichkeit zur Release-Auskopplung wird über einen zusätzlich definierten Projektstatus und eine werkzeuggestützte Aufwandserfassung erreicht. Es handelt sich daher um einen grundsätzlich adaptiven Planungsprozess [Lar04], der im Normalfall lediglich die nächste Iteration im Detail plant. Wenn ein Release ansteht, können auch mehrere Iterationen geplant werden, so dass die Planung etwas prädiktiver abläuft. Durch eine relativ geringe Komplexität des Prozesses mit wenigen definierten Rollen und einer effektiven Dokumentation der Software ist es möglich, aus neuen Mitarbeitern schnell produktive Entwickler zu machen.

In weiteren Arbeiten wäre es interessant die praktische Verwendbarkeit des präsentierten Ansatzes zu untersuchen. Auch sind die angesprochenen Werkzeuge bisher noch nicht optimal vernetzt. Hier wären weitere Schritte wünschenswert, so dass der Entwickler nur mit einer einzigen, durchgängigen Umgebung arbeiten kann. Im Falle der Ausweitung der Software auf einen breiten Markt oder gar bei einer eigens dafür stattfindenden Gründung eines Spin-offs, ist die Software-Entwicklung als Fluss nicht mehr anwendbar. Hier bedarf es einer neuen Evaluation: Kommt ein bekannter Software-Entwicklungsprozess in Frage, oder ist eine Neuentwicklung nötig?

# Literaturverzeichnis

- [ABB<sup>+</sup>98] **Anderson, A., R. Beattie, K. Beck, D. Bryant, M. DeArment, M. Fowler, M. Franczak, R. Garzaniti, D. Gore, B. Hacker et al.:** *Chrysler Goes to Extremes*. Distributed Computing, 1(10):25–28, 1998.
- [ASRW02] **Abrahamsson, P., O. Salo, J. Ronkainen und J. Warsta:** *Agile Software Development Methods: Review and Analysis*. VTT Publications, 2002.
- [Bal0] **Balduino, R.:** *Introduction to OpenUP (Open Unified Process)*, o.J. Von <http://www.eclipse.org/epf/general/OpenUP.pdf>, besucht am 14.03.2008.
- [Bar05] **Barnett, L.:** *Big changes coming for rational unified process*, July 2005. Von <http://www.forrester.com/Research/Document/Excerpt/0>, besucht am 10.12.2007.
- [BBL76] **Boehm, B.W., J.R. Brown und M. Lipow:** *Quantitative evaluation of software quality*. In: *2nd International Conference on Software Engineering*, Seiten 592–605. IEEE Computer Society Press Los Alamitos, CA, USA, 1976.
- [BC99] **Beck, K. und D. Cleal:** *Optional Scope Contracts*. White Paper, Three Rivers Institute, 1999.
- [BD97] **Boehm, B.W. und T. DeMarco:** *Software risk management*. IEEE Software, 14(3):17–19, 1997.
- [BD03] **Benediktsson, O. und D. Dalcher:** *Effort estimation in incremental software development*. In: *Software*, Band 150, Seiten 351–357. IEE Proceedings, 2003.
- [BDT05] **Benediktsson, O., D. Dalcher und H. Thorbergsson:** *Working with Alternative Development Life Cycles: A Multiproject Experiment*. In: *Thirteenth International Conference on Information Systems Development-ISD, Advances in Theory, Practice and Education*. Springer, 2005.

- [BDT06] **Benediktsson, O., D. Dalcher** und **H. Thorbergsson**: *Comparison of software development life cycles: a multiproject experiment*. In: *Software*, Band 153, Seiten 87–101. IEE Proceedings, 2006.
- [Bec99] **Beck, K.**: *Extreme Programming Explained*. Addison-Wesley Professional, 1999.
- [Ber02] **Berry, D.M.**: *The Inevitable Pain of Software Development, Including of Extreme Programming, Caused by Requirements Volatility*. In: *Radical Innovations of Software and Systems Engineering in the Future*, Canada, 2002. University of Waterloo.
- [BH00] **Boehm, B.W.** und **W.J. Hansen**: *Spiral Development: Experience, Principles, and Refinements*. Carnegie Mellon University, Software Engineering Institute, 2000.
- [BKBM<sup>+</sup>03] **Becker-Kornstaedt, U., F. Bella, J. Münch, H. Neu, A. Ocampo** und **J. Zettel**: *SPEARMINT 7 – User Manual*. Technischer Bericht, Fraunhofer Institut für Experimentelles Software Engineering, 2003.
- [BKL04] **Bischofberger, W., J. Kuhl** und **S. Loffler**: *Sotograph – A Pragmatic Approach to Source Code Architecture Conformance Checking*. In: *LNCS 3047: Software Architecture: First European Workshop*. Springer, 2004.
- [BKV99] **Becker-Kornstaedt, U.** und **M. Verlage**: *The V-Modell guide: experience with a web-based approach for process support*. In: *Software Technology and Engineering Practice*, Seiten 161–168, 1999.
- [BMR<sup>+</sup>00] **Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, M. Stal** und **C. Löckenhoff**: *Pattern-orientierte Softwarearchitektur*. Addison Wesley in Pearson Education Deutschland, 2000.
- [Boe79] **Boehm, B.W.**: *Guidelines for Verifying and Validating Software Requirements and Design Specifications*. In: *European Conference on Applied Information Technology of the International Federation for Information Processing*, Band 79, Seiten 711–719, London, 1979. North Holland Publishing.
- [Boe88] **Boehm, B. W.**: *A spiral model of software development and enhancement*. In: *IEEE Computer*, Band 21, Seiten 61–72. IEEE Computer Society Press Los Alamitos, CA, USA, 1988.
- [Boe02] **Boehm, B.W.**: *Get ready for agile methods, with care*. *IEEE Computer*, 35(1):64–69, 2002.

- [Bou02] **Boudreau, T.**: *NetBeans: The Definitive Guide*. O'Reilly, 2002. [www.netbeans.org](http://www.netbeans.org).
- [BR89] **Boehm, B.W.** und **R. Ross**: *Theory-W software project management principles and examples*. In: *IEEE Transactions on Software Engineering*, Band 15, Seiten 902–916. IEEE Computer Society Press Los Alamitos, CA, USA, 1989.
- [BR05] **Broy, M.** und **A. Rausch**: *Das neue V-Modell® XT – Ein anpassbares Vorgehensmodell für Software und System Engineering*. In: *Informatik-Spektrum*, Band 28, Seiten 220–229. Springer, 2005.
- [Bro75] **Brooks, FP**: *The Mythical Man Month*. Addison-Wesley, 1975.
- [Bro87] **Brooks, F.P.**: *No Silver Bullet*. In: *IEEE Computer*, Band 20, Seiten 10–19. IEEE Computer Society Press Los Alamitos, CA, USA, 1987.
- [C+04] **Cohn, M.** et al.: *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004.
- [CDJS0] **Chiwenda, M., D.V. Delano, F. Jonson** und **H.E.T. Scott**: *Evolutionary development model – An Introduction and Analysis*, o.J. von <https://idenet.bth.se/servlet/download/news/18332/EVO.pdf>, besucht am 12.3.2008, Jahr und Art der Veröffentlichung unbekannt.
- [CKS03] **Chrissis, M.B., M. Konrad** und **S. Shrum**: *CMMI: Guidelines for Process Integration and Product Improvement*. Addison-Wesley Professional, 2003.
- [Coh02] **Cohn, M.**: *A Measured Response*. Better Software Magazine, STQE Publishing, Seiten 21–25, 2002.
- [Coh05] **Cohn, M.**: *Agile Estimating and Planning*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.
- [Cop05] **Copeland, T.**: *PMD Applied*. Centennial Books, 2005.
- [CS95] **Cusumano, M.A.** und **R.W. Selby**: *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press New York, NY, USA, 1995.
- [CSFP04] **Collins-Sussman, B., B.W. Fitzpatrick** und **C.M. Pilato**: *Version Control with Subversion*. O'Reilly Media, Inc., 2004. Permant aktualisierte onlineversion unter <http://svnbook.red-bean.com/>, besucht am 3.3.2008.

- [Cun92] **Cunningham, W.:** *The WyCash portfolio management system.* In: *Conference on Object Oriented Programming Systems Languages and Applications*, Seiten 29–30. ACM Press New York, NY, USA, 1992.
- [Dei03] **Deißenböck, F.:** *DHP-FAQ*, 2003. Von <http://www.deissenboeck.de/faqs>, besucht am 24.01.2008.
- [DH04a] **Dornberger, R.** und **T. Habegger:** *Extreme Programming: Eine Übersicht und Bewertung*, 2004. Diskussion Paper, Fachhochschule Solothurn Nordwestschweiz.
- [DH04b] **Dubinsky, Y.** und **O. Hazzan:** *Roles in Agile Software Development Teams.* In: *LNCS 3092: Extreme Programming and Agile Processes in Software Engineering*, Band 5, Seiten 157–165. Springer, 2004.
- [DL04] **De Luca, J.:** *What percentage of people are required to be experienced?*, 2004. Von <http://www.featuredrivendevelopment.com/node/635> besucht am 12.12.2007.
- [DPS06] **Deissenboeck, F., M. Pizka** und **T. Seifert:** *Tool Support for Continuous Quality Assessment.* Workshop on Software Technology and Engineering Practice (STEP). IEEE Computer Society, 2006.
- [DS67] **Draper, N.R.** und **H. Smith:** *Applied Regression Analysis.* New York, 1967.
- [DSS04] **Dall’Agnol, M., A. Sillitti** und **G. Succi:** *Project Management and Agile Methodologies: A Survey.* In: *LNCS 3092: Extreme Programming and Agile Processes in Software Engineering*, Band 5, Seiten 223–226. Springer, 2004.
- [Ebe03a] **Eberle, C.:** *Adaptive Software Development.* Technischer Bericht, Institut für Informatik, Universität Zürich, 2003. Seminar: Agile vs. klassische Methoden der Software-Entwicklung.
- [Ebe03b] **Eberle, C.:** *Adaptive Software Development*, 2003. Institut für Informatik, Universität Zürich, Seminar: Agile vs. klassische Methoden der Software-Entwicklung, Seminar Foliensatz.
- [Edg07] **Edgwall Software:** *Trac Project*, 2007. Von <http://trac.edgwall.org/>, besucht am 4.3.2008.
- [ER03] **Endres, A.** und **H.D. Rombach:** *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories.* Addison Wesley Longman, 2003.

- [Fog08] **Fog Creek Software:** *FogBugz - Project Management Software*, 2000 – 2008. Von <http://www.fogcreek.com/FogBugz/>, besucht am 3.3.2008.
- [For95] **Forsythe, C.:** *Human factors in agile manufacturing*. Human Factors and Ergonomics Society, 1995.
- [Fow02] **Fowler, M.:** *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
- [GHJ04] **Gamma, E., R. Helm und R. Johnson:** *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison Wesley in Pearson Education Deutschland, 2004.
- [Gil81] **Gilb, T.:** *Evolutionary development*. In: *ACM SIGSOFT Software Engineering Notes*, Band 6, Seiten 17–17. ACM Press New York, NY, USA, 1981.
- [Gil85] **Gilb, T.:** *Evolutionary Delivery versus the "waterfall" model*. In: *ACM SIGSOFT Software Engineering Notes*, Band 10, Seiten 49–61. ACM Press New York, NY, USA, 1985.
- [Gil88] **Gilb, T.:** *Principles of software engineering management*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1988.
- [Goo08a] **Google:** *The Google Culture*, 2008. Von <http://www.google.com/corporate/culture.html>, besucht am 3.3.2008.
- [Goo08b] **Google:** *Life at Google*, 2008. Von <http://www.google.com/support/jobs/bin/static.py?page=about.html>, besucht am 3.3.2008.
- [Gre01] **Grenning, J.:** *Launching extreme programming at a process-intensive company*. IEEE Software, 18(6):27–33, 2001.
- [Gun99] **Gunasekaran, A.:** *Agile manufacturing: a framework for research and development*. In: *International Journal of Production Economics*, Band 62, Seiten 87–105. Elsevier, 1999.
- [Gun07] **Gunderloy, M.:** *Painless Project Management with FogBugz*. Apress Berkely, CA, USA, 2007.
- [Gut72] **Gutenberg, E.:** *Grundlagen der Betriebswirtschaftslehre. 1. Die Produktion*. Springer, 1972.
- [Gyg03] **Gyger, D.:** *Feature-Driven Development*. Technischer Bericht, Institut für Informatik, Universität Zürich, 2003. Seminar: Agile vs. klassische Methoden der Software-Entwicklung.

- [Hal77] **Halstead, M.H.:** *Elements of Software Science, Operating, and Programming Systems Series*. Band 7. Elsevier New York, 1977.
- [HC06a] **Harris, S.** und **M. Cohn:** *Incorporating Learning and Expected Cost of Change in Prioritizing Features on Agile Projects*. In: *LNCS 4044: Extreme Programming and Agile Processes in Software Engineering*, Band 7, Seiten 175–180. Springer, 2006.
- [HC06b] **Harris, S.** und **M. Cohn:** *The Role Of Learning And Expected Cost Of Change In Prioritizing Features On Agile Projects*. In: *LNCS 4044: Extreme Programming and Agile Processes in Software Engineering*, Band 7, Seiten 180–190. Springer, 2006.
- [Hec07] **Heck, Jan P.:** *Architekturanalyse mit Sotograph*. Technischer Bericht, Institut für Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster, 2007. Seminar: Qualitätssicherung in der Software-Entwicklung.
- [Hes97] **Hesse, W.:** *Wie evolutionär sind die objektorientierten Analysemethoden? Ein kritischer Vergleich*. Informatik-Spektrum, 20(1):21–28, 1997.
- [HH02] **Highsmith, J.A.** und **J. Highsmith:** *Agile Software Development Ecosystems*. Addison-Wesley Professional, 2002.
- [Hig00] **Highsmith, J. A.:** *Adaptive software development: a collaborative approach to managing complex systems*. Dorset House Publishing Co., Inc., New York, NY, USA, 2000.
- [Hir02] **Hirsch, M.:** *Making RUP agile*. In: *Conference on Object Oriented Programming Systems Languages and Applications*. ACM Press New York, NY, USA, 2002.
- [Hol05] **Holly, R.:** *A reusable Duff device*. Dr. Dobbs Journal, 30(8):73–74, 2005.
- [HS02] **Hendrix, T.D.** und **M.P. Schneider:** *NASAs TReK Project: A Case Study in Using the Spiral Model of Software Development*. Communications of the ACM, 45(4ve):153, 2002.
- [IBMoJ] **IBM:** *Rational Unified Process Data Sheet*, o.J. Von [ftp://ftp.software.ibm.com/software/rational/web/datasheets/RUP\\_DS.pdf](ftp://ftp.software.ibm.com/software/rational/web/datasheets/RUP_DS.pdf) besucht am 4.12.2007.
- [Khr06] **Khramov, Y.:** *The cost of code quality*. Agile Conference, Seite 7, 2006.
- [KNR06] **Kuhrmann, M., D. Niebuhr** und **A. Rausch:** *Application of the V-Modell XT – Report from A Pilot Project*. In: *LNCS 3840:*

- Unifying the Software Process Spectrum, International Software Process Workshop, SPW*, Seiten 463–473. Springer, 2006.
- [Koo06] **Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung: Die V-Modell XT Quellen**, 2006. Von <ftp://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/1.2.1/install-v-modell-xt-1.2.1.jar>, besucht am 3.12.2007.
- [Koo07] **Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung: Offenes V-Modell XT Forum: Nutzung V-Modell XT**, 2007. Von [http://www.kbst.bund.de/kbst\\_forum/showthread.php?t=937,2027](http://www.kbst.bund.de/kbst_forum/showthread.php?t=937,2027). 11.07, besucht am 29.11.2007.
- [Kru01] **Kruchten, P.**: *Using the RUP to Evolve a Legacy System*. The Rational Edge, 2001.
- [KT79] **Kahneman, D. und A. Tversky**: *Prospect Theory: An Analysis of Decision under Risk*. Econometrica, 1979.
- [Lar04] **Larman, C.**: *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2004.
- [Lee06] **Lee, K.**: *An Empirical Development Case of a Software-Intensive System Based on the Rational Unified Process*. In: LNCS: *Computational Science and Its Applications – ICCSA*, Band 3984, Seite 877. Springer, 2006.
- [Leh96] **Lehman, M.M.**: *Laws of Software Evolution Revisited*. European Workshop on Software Process Technology, Seiten 108–124, 1996.
- [LH06] **Llopis, N. und S. Houghton**: *Backwards Is Forward: Making Better Games with Test-Driven Development*, 2006. presented at Game Developers Conference.
- [Lic07] **Lichter, H.**: *Software-Qualitätssicherung und Projektmanagement*, 2007. Foliensatz zur Vorlesung.
- [Lip04] **Lippert, M.**: *Towards a Proper Integration of Large Refactorings in Agile Software Development*. In: LNCS 3092: *Extreme Programming and Agile Processes in Software Engineering*, Band 5, Seiten 113–122. Springer, 2004.
- [LKB01] **Larman, C., P. Kruchten und K. Bittner**: *How to Fail with the Rational Unified Process: Seven Steps to Pain and Suffering*. Valtech Technologies, 2001.

- [LL07] **Lichter, H.** und **J Ludewig**: *Software Engineering – Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag Heidelberg, 2007.
- [LP92] **Loewenstein, G.** und **D. Prelec**: *Anomalies in Intertemporal Choice: Evidence and an Interpretation*. In: *The Quarterly Journal of Economics*, Band 107, Seiten 573–597. JSTOR, 1992.
- [LT89] **Loewenstein, G.** und **R.H. Thaler**: *Anomalies: Intertemporal Choice*. Band 3, Seiten 181–193. JSTOR, 1989.
- [MA07] **Marchenko, A.** und **P. Abrahamsson**: *Predicting Software Defect Density: A Case Study on Automated Static Code Analysis*. In: *LNCS 4536: Extreme Programming and Agile Processes in Software Engineering*, Band 8, Seiten 137–140. Springer, 2007.
- [Mal02] **Malotaux, N.**: *Evolutionary Project Management Methods*, 2002. Von <http://www.malotaux.nl/nrm/pdf/MxEvo.pdf> besucht am 28.2.2008.
- [Mal06] **Malotaux, N.**: *Evolutionary Project Management Methods – Slash Project Time with Evolutionary Methods*, 2006. Von <http://www.malotaux.nl/nrm/pdf/EvoWorkshop.pdf> besucht am 14.3.2008.
- [MaloJ] **Malotaux, N.**: *The difference between effort and lead-time*, o.J. Von <http://www.malotaux.nl/nrm/Evo/EffLead.htm> besucht am 9.3.2008.
- [Mar07] **Marschall, F.**: *Experiences with Introducing a new Software Process Model at T-Systems*. In: *Software Engineering and Advanced Applications, 33rd EUROMICRO Conference on*, Seiten 281–288. IEEE Computer Society Press Los Alamitos, CA, USA, 2007.
- [McC76] **McCabe, TJ**: *A Complexity Measure*. In: *IEEE Transactions on Software Engineering*, Band 2, Seiten 308–320. IEEE Press Piscataway, NJ, USA, 1976.
- [MFC05] **Middelton, P., A. Flaxel** und **A. Cookson**: *Lean Software Management Case Study: Timberline Inc.* In: *LNCS 3556: Extreme Programming and Agile Processes in Software Engineering*, Band 6, Seiten 1–9. Springer, 2005.
- [MK02] **Madsen, S.** und **K. Kautz**: *Applying System Development Methods in Practice – The RUP Example*. Information Systems Development: Advances in Methodologies, Components and Management, Seiten 267–278, 2002.

- [MM05] **Mann, C.** und **F. Maurer**: *A Case Study on the Impact of Scrum on Overtime and Customer Satisfaction*. In: *Proceedings of the Agile Development Conference*, Seiten 70–79. IEEE Computer Society Press Los Alamitos, CA, USA, 2005.
- [Moo95] **Moore, G.A.**: *Inside The Tornado: Marketing Strategies From Silicon Valley's Cutting Edge*. Harper Business New York, 1995.
- [Moo04] **Moore, G.A.**: *Inside the Tornado: Strategies for Developing, Leveraging, and Surviving Hypergrowth Markets*. Harper Business New York, 2004.
- [MP02] **Motschnig-Pitrik, R.**: *Employing the Unified Process for Developing a Web-Based Application-A Case-Study*. In: *Practical Aspects of Knowledge Management: 4th International Conference, PAKM*, Vienna, Austria, December 2002. Springer.
- [MP03] **Manzoni, L.V.** und **R.T. Price**: *Identifying extensions required by RUP (rational unified process) to comply with CMM (capability maturity model) levels 2 and 3*. In: *IEEE Transactions on Software Engineering*, Band 29, Seiten 181–192. IEEE Computer Society Press Los Alamitos, CA, USA, 2003.
- [MSS07] **Moser, R., M. Scotto, A. Sillitii** und **G. Succi**: *Does XP Deliver Quality and Maintainable Code?* In: *LNCS 4536: Extreme Programming and Agile Processes in Software Engineering*, Band 8, Seiten 105–114. Springer, 2007.
- [MSV97] **Münch, J., M. Schmitz** und **M. Verlage**: *Tailoring großer Prozeßmodelle auf der Basis von MVP-L*. In: *Vorgehensmodelle – Einführung, betrieblicher Einsatz, Werkzeug-Unterstützung und Migration*, Band 4, Seiten 63–72, 1997.
- [MTA<sup>+</sup>05] **Muljadi, H., H. Takeda, J. Araki, S. Kawamoto, S. Kobayashi, Y. Mizuta, S.M. Demiya, S. Suzuki, A. Kitamoto, Y. Shirai** et al.: *Semantic MediaWiki: a user-oriented system for integrated content and metadata management system*. Proceedings of the IADIS International Conference WWW/Internet, Seiten 19–22, 2005.
- [MW94] **McCabe, T.J.** und **A.H. Watson**: *Software Complexity*. Crosstalk, Journal of Defense Software Engineering, 7(12):5–9, 1994.
- [Nor03] **North Atlantic Treaty Organisation, Research and Technology Organisation**: *Technology for Evolutionary Software Development*, 2003. Papers presented at the Information Systems Technology Panel (IST) Symposium held in Bonn, Germany, 23-24 September 2002.

- [OH92] **Oman, P.** und **J. Hagemeister**: *Construction and Validation of Polynomials for Predicting Software Maintainability*. SETL Report, Seiten 92–06, 1992.
- [OpeoJa] **Open Source Software: CVS - Concurrent Versions System**, o.J. Von <http://www.nongnu.org/cvs/>, besucht am 3.3.2008.
- [OpeoJb] **Open Source Software: PMD**, o.J. Von <http://pmd.sourceforge.net/>, besucht am 5.3.2008.
- [OpeoJc] **Open Source Software, Eclipse Foundation: Eclipse Process Framework Project (EPF), OpenUP**, o.J. Von <http://www.eclipse.org/epf/>, besucht am 14.3.2008.
- [OpeoJd] **Open Source Software, Tigris.org: Subversion**, o.J. Von <http://subversion.tigris.org/>, besucht am 3.3.2008.
- [Pal01] **Palmer, S.R.**: *A Practical Guide to Feature-Driven Development*. Pearson Education, 2001.
- [Par72] **Parnas, D.**: *On the Criteria for Decomposing Systems into Modules*. Communications of the ACM, 15(12):1053–1058, 1972.
- [PEJ99] **P., Coad, Lefebvre E.** und **De Luca J.**: *Java Modeling in Color with UML: Enterprise Components and Process*. Prentice Hall, 1999.
- [PODB95] **Pearse, T., P. Oman, P. Div** und **I.D. Boise**: *Maintainability measurements on industrial source code maintenance activities*. In: *Software Maintenance, International Conference on*, Seiten 295–303, 1995.
- [PW06] **Pietrzak, B.** und **B. Walter**: *Leveraging Code Smell Detection with Inter-smell Relations*. In: *LNCS 4044: Extreme Programming and Agile Processes in Software Engineering*, Band 7, Seiten 75–84. Springer, 2006.
- [Rei03] **Reitzig, R.W.**: *Using Rational Software Solutions to Achieve CMMI Level 2*. The Rational Edge, 2003.
- [Rie01] **Riehle, D.**: *A comparison of the value systems of Adaptive Software Development and Extreme Programming: How methodologies may learn from each other*. In: *Extreme Programming Explained*, Seiten 35–50. Addison-Wesley, Boston, USA, 2001.
- [Ros67] **Rosove, P.E.**: *Developing Computer-based Information Systems*. John Wiley & Sons Inc, 1967.

- [Roy70] **Royce, W.:** *Managing the Development of Complex Software Systems: Concepts and Techniques*. In: *9th international conference on Software Engineering*, Monterey, California, United States, 1970. IEEE Computer Society Press Los Alamitos, CA, USA.
- [RW04] **Roock, S.** und **H. Wolf:** *Agile Project Controlling*. In: *LNCS 3092: Extreme Programming and Agile Processes in Software Engineering*, Band 5, Seiten 202–209. Springer, 2004.
- [SB01] **Schwaber, K.** und **M. Beedle:** *Agile Software Development with Scrum*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.
- [Seg03] **Segal, J.:** *When software engineers met research scientists: a field study*. Technischer Bericht, Department of Computing, Faculty of Mathematics and Computing, The Open University, Walton Hall, Milton Keynes, UK, 2003.
- [Seg05] **Segal, J.:** *When Software Engineers Met Research Scientists: A Case Study*. In: *Empirical Software Engineering*, Band 10, Seiten 517–536, Hingham, MA, USA, 2005. Kluwer Academic Publishers.
- [Sei07] *CMMI SCAMPISM Class A Appraisal Results, 2005–2007*. Onlinedaten von <http://www.sei.cmu.edu/appraisal-program/profile/profile.html>, besucht am 18.03.2008.
- [SGK07] **Sato, D., A. Goldman** und **F. Kon:** *Tracking the Evolution of Object-Oriented Quality Metrics on Agile Projects*. In: *LNCS 4536: Extreme Programming and Agile Processes in Software Engineering*, Band 8, Seiten 84–92. Springer, 2007.
- [Sim05] **Simons, A.J.H.:** *Testing with Guarantees and the Failure of Regression Testing in extreme Programming*. In: *Extreme Programming And Agile Processes in Software Engineering: 6th International Conference, XP*, Sheffield, UK, June 2005. Springer.
- [Sof0] **Software-Tomography:** *Sotograph*, o.J. Website <http://www.software-tomography.de/html/sotograph.html>, besucht am 5.3.2008.
- [Spo07] **Spolsky, J.:** *Evidence Based Scheduling*, 2007. Aus «Joel on Software», Von <http://www.joelonsoftware.com/items/2007/10/26.html>, besucht am 20.01.2008.
- [SSAD06] **Sfetsos, P., I. Stamelos, L. Angelis** und **I.S. Deligiannis:** *Investigating the Impact of Personality Types on Communication*

- and Collaboration-Viability in Pair Programming*. In: *LNCS 4044: Extreme Programming and Agile Processes in Software Engineering*, Band 7, Seiten 43–52. Springer, 2006.
- [Sut01] **Sutherland, J.**: *Agile Can Scale: Inventing and Reinventing SCRUM in Five Companies*. Cutter IT Journal, 14(12):5–11, 2001.
- [Sut05] **Sutherland, J.**: *Future of Scrum: Parallel Pipelining of Sprints in Complex Projects*. Agile 2005 Conference, 2005.
- [Sut06] **Sutherland, J.**: *Scrum Tuning: Lessons learned from Scrum implementation at Google*, 2006. Google Tech Talks, Video at <http://video.google.com/videoplay?docid=8795214308797356840>.
- [Tin05] **Tingey, F.**: *Quantifying Requirements Risk*. In: *Extreme Programming And Agile Processes in Software Engineering: 6th International Conference XP*, Sheffield, UK, June 2005. Springer.
- [Van97] **VanDoren, E.**: *Halstead Complexity Measures*, 1997. Von [http://www.sei.cmu.edu/str/descriptions/halstead\\_body.html](http://www.sei.cmu.edu/str/descriptions/halstead_body.html), besucht am 10.3.2008, Carnegie Mellon University, Software Engineering Institute.
- [Van00] **VanDoren, E.**: *Cyclomatic Complexity*, 2000. Von <http://www.sei.cmu.edu/str/descriptions/cyclomatic.html>, besucht am 10.3.2008, Carnegie Mellon University, Software Engineering Institute.
- [Van04] **VanDoren, E.**: *Maintainability Index Technique for Measuring Program Maintainability*, 2004. Von <http://www.sei.cmu.edu/str/descriptions/mitmpm.html>, besucht am 2.3.2008, Carnegie Mellon University, Software Engineering Institute.
- [vN02] **Nitzsch, R. von**: *Entscheidungslehre*. Schäfer-Poeschel Verlag Stuttgart, 2002.
- [WikoJa] **Wikimedia Foundation**: *MediaWiki*, o.J. Von <http://www.mediawiki.org/wiki/MediaWiki>, besucht am 4.3.2008.
- [WikoJb] **Wikipedia**: *Comparison of wiki software*, o.J. Von [http://en.wikipedia.org/wiki/Comparison\\_of\\_wiki\\_software](http://en.wikipedia.org/wiki/Comparison_of_wiki_software), besucht am 4.3.2008.
- [WJR91] **Womack, J.P., D.T. Jones** und **D. Roos**: *The Machine that Changed the World: The Story of Lean Production*. HarperPerennial, 1991.

- [WKCJ00] **Williams, L., RR Kessler, W. Cunningham und R. Jeffries:** *Strengthening the case for pair programming.* IEEE Software, 17(4):19–25, 2000.
- [Zag05] **Zagrodnick, C.:** *Agile Development in Small and Medium Sized Projects.* Diplomarbeit, HS Anhalt, 2005.
- [Zan07] **Zang, J.:** *Financial Organization Transformation Strategy.* In: *LNCS 4536: Extreme Programming and Agile Processes in SoftwareEngineering*, Band 8, Seiten 188–192. Springer, 2007.
- [ZK03] **Zeller, A. und J. Krinke:** *Open-Source-Programmierwerkzeuge – Versionskontrolle - Konstruktion - Testen - Fehlersuche.* dpunkt.verlag Heidelberg, 2003.