Analyse der Wirkung von Nutzer-Feedback auf die Entwicklungszeit komplexer Softwareprodukte

Inauguraldissertation
zur
Erlangung des Doktorgrades
der
Wirtschafts- und Sozialwissenschaftlichen Fakultät
der
Universität zu Köln

2011

vorgelegt
von

Diplom-Informatiker Lars Haferkamp

aus

Daun

Referent:	Prof. Dr. Werner Mellis
Koreferent:	Prof. Dr. Detlef Schoder
Tag der Promotion:	14.09.2011

Inhaltsverzeichnis

Kapitel 1: Einführung 1
1.1. Motivation..1
1.2. Forschungsbereich 4
1.3. Zielsetzung..6
1.4. Vorgehensweise 7
1.5. Aufbau der Arbeit 9
Kapitel 2: Prozesse zur Entwicklung von Softwareprodukten 11
2.1. Von Anforderungen zum Softwareprodukt 11
2.2. Prozessaktivitäten, Phasen und Rollen 16
2.3. Zyklische Entwicklung auf verschiedenen Ebenen 21
2.4. Plangetriebene Entwicklungsprozesse 26
2.5. Inkrementelle Entwicklungsprozesse 29
2.5.1. Grundlagen 29
2.5.2. Best practice: Die Rational-Unified-Process-Methodik............................31
2.5.3. Best practice: Die Extreme-Programming-Methodik 34
2.5.4. Best practice: Die Synch-and-Stabilize-Methodik...................................38
2.6. Steuerungsmöglichkeiten der Prozessplanung 41
Kapitel 3: Einflussfaktoren des Entwicklungsaufwands...45
3.1. Identifizierung der Einflussfaktoren 46
3.1.1. Faktoren aus der Produktentwicklung und Organisationstheorie 46
3.1.2. Faktoren aus Schätzmethoden des Implementierungsaufwands 48
3.1.3. Faktoren weiterer Prozessaktivitäten 50
3.2. Analyse der Anforderungsunsicherheit 53
3.2.1. Ausprägungen der Unsicherheit 53
3.2.2. Maßnahmen bei Unsicherheit..59
3.2.3. Ergebnis der Analyse der Anforderungsunsicherheit...............................61
3.3. Analyse der Sensitivität/Flexibilität in komplexen Systemen 63
3.3.1. Abhängigkeiten zwischen Anforderungen 65
3.3.2. Hierarchie und Modularität des Softwaresystems....................................69
3.3.3. Strukturen auf organisatorischer Ebene 73
3.4. Übersicht der in den Modellen betrachteten Einflussfaktoren 74
Kapitel 4: Kostenfunktionen für Prozesse mit zyklischem Nutzerfeedback...............78
4.1. Unterschiedliche Arten von Kostenfunktionen 78

4.2. Die Überarbeitungsdauer abhängig von Sensitivität und Unsicherheit 82
4.3. Bestimmung der Entwicklungszeit aus dem Entwicklungsaufwand 86
4.4. Skaleneffekte in der Softwareentwicklung 89
4.4.1. Positive, negative oder keine Skaleneffekte? 89
4.4.2. Skaleneffekte durch komplexe Modulstrukturen 94
4.5. Erweiterung der Kostenfunktionen durch die Modellierung von Zyklen 96
4.5.1. Modellierung des Überarbeitungsaufwand in den Zyklen 96
4.5.2. Modellierung der Fixkosten der Zyklen 99
Kapitel 5: Optimierung der Kosten unter Berücksichtigung von Feedbackeffekten 102
5.1. Allgemeine Modelle der Produktentwicklung 102
5.1.1. Vergleich der verschiedenen Modelle 102
5.1.2. Überlappung zwischen den Aktivitäten 105
5.2. Das Releaseplan-Modell 107
5.2.1. Modellierung von Feedback-Effekten 108
5.2.2. Modellierung der Releaseplanung 111
5.2.3. Reduzierung der Entwicklungszeit in Abhängigkeit der Prozessplanung 113
5.2.4. Zusammenfassung der Modellannahmen 118
5.3. Das Meetingplan-Modell 119
5.3.1. Modellierung der Überarbeitungsdauer 120
5.3.2. Reduzierung der Entwicklungszeit in Abhängigkeit der Prozessplanung 122
5.3.3. Diskussion der Modellannahmen 126
5.3.4. Erweiterung um Feedback durch Prototyping 127
Kapitel 6: Abschließende Betrachtung 137

Abbildungsverzeichnis

Abbildung 1-1: Hype Cycle for Emerging Technologies, 2009 2

Abbildung 1-2: Die Releasezyklen von Windows Vista 4

Abbildung 2-1: Zusammenhang von Rolle, Aufgaben und Artefakt am Beispiel von Aufgaben der Anforderungsanalyse 17

Abbildung 2-2: Detailliertes Flussdiagramm eines zyklischen SWE-Prozesses mit verschiedenen Entscheidungssituationen. 19

Abbildung 2-3: Verschiedene Ebenen von Entwicklungszyklen.....................................22

Abbildung 2-4: Mehrere Softwaregenerationen in Weiterentwicklungszyklen 23

Abbildung 2-5: Einordnung des On-Site-Customers 25

Abbildung 2-6: Beispielhafte Gewichtung der Entwicklungsaktivitäten in einem plangetriebenen Entwicklungsprozess 27

Abbildung 2-7: Beispielhafte Gewichtung der Entwicklungsaktivitäten in einem iterativen, inkrementellen Entwicklungsprozess 30

Abbildung 3-1: Ursachen für Anforderungsunsicherheit 54

Abbildung 3-2: Zusammenhang von Anforderungsunsicherheit und Maßnahmen...........62

Abbildung 3-3: Vorteil von mehreren Entwicklungszyklen bei Abhängigkeiten zwischen Systemanforderungen 68

Abbildung 4-1: Exemplarische Verteilung des Aufwands/Mitarbeitereinsatzes über die Zeit 88

Abbildung 4-2: Vergleich des Verlaufs der Kostenfunktion im Translog und im linearen Modell 90

Abbildung 4-3: Vergleich einer hochabhängigen mit einer hierarchischen Modulstruktur. 95

Abbildung 4-4: Lineare Regressionsgerade für den Aufwand in „enhancement releases“ nach (Basili et al. 1996) 100

Abbildung 5-1: Prozessflussdiagramm auf Ebene der Release-Zyklen..........................108

Abbildung 5-2: Reduktion des Änderungsumfangs durch Feedback-Effekte 110

Abbildung 5-3: Verlauf des Gesamtumfangs bei unterschiedlichen Releaseplanungen .112

Abbildung 5-4: Entwicklungszeit in Abhängigkeit der Anzahl der Releases und der Feedbackeffektivität...116

Abbildung 5-5: Entwicklungszeit in Abhängigkeit der Anzahl der Releases und der Releaseplanung 118

Abbildung 5-6: Prozessflussdiagramm des Prototypingplan-Modells 129

Abbildung 6-1: Windows Vista Timeline..161

Abkürzungsverzeichnis

AVN	Analogy With Virtual Neighbour
BCRM	Budget-Constrained Reliability-Maximization
CART	Classification And Regression Trees
CBR	Case Based Reasoning
CMM	Capability Maturity Model
Cocomo	Constructive Cost Model
DSM	Design Structure Matrix
Engl.	Englisch
FP	Function-Points
IBM	International Business Machines Corporation
IFPUG	International Function-Point Users Group
IIE	Iterativ-Inkrementelle Entwicklung
IKIWISI	„I'll know it when I see it“
IKT	Informations- und Kommunikationstechnologie
IOC	Initial Operational Capability
IT	Informationstechnologie
i. A.	Im Allgemeinen
JAD	Joint Application Design
LCA	Lifecycle Architecture
LCO	Lifecycle Objective
MA	Mitarbeiter
NESMA	Nederlandse Software Metrieken Associatie
OLS	Ordinary Least Squares
PD	Product Development

PR	Product Release
RCCM	Reliability-Constrained Cost-Minimization
REVL	Requirements Evolution and Volatility
SOA	Service-orientierte Architekturen
SLIM	Software Lifecycle Management
SU	Software Understanding
SW	Software
SWE	Softwareentwicklung
UNFM	Software Unfamiliarity
USA	United States of America
WBS	Work Break-Down Structure

Kapitel 1: Einführung

1.1. Motivation

Im Jahr 2005 setzten laut Statistischem Bundesamt[1] fast alle deutschen Unternehmen mit 20 Beschäftigten und mehr Computer – und damit auch Softwareprodukte – in ihrem Geschäftsablauf ein. Den sich daraus ergebenden IT-Markt, bedienen Hersteller von Standardsoftwareprodukten, wie die Microsoft Deutschland GmbH und die SAP Deutschland AG & CO. KG, Unternehmen aus dem Bereich IT-Beratung und Systemintegration, wie die IBM Deutschland Business Services GmbH und die Accenture GmbH, sowie IT-Service Unternehmen, wie die T-Systems International GmbH.[2]

Diese und andere Softwarehäuser stehen vor der Herausforderung, dass die Softwareentwicklung[3] in einem Umfeld stattfindet, welches durch schnell aufeinanderfolgende technologische Neuerungen und sich ändernde Marktbedingungen geprägt ist.[4] Die technologische Entwicklung des Umfelds verdeutlicht z. B. der „Hype Cycle for Emerging Technologies 2009“ der Analysten des Unternehmens Gartner Inc., wie in Abbildung 1-1 dargestellt.[5]

---

[1] Vgl. Statistisches Bundesamt - Pressestelle Februar 2006.

[2] Vgl. zu diesem Absatz Hossenfelder 2010.

[3] Im Folgenden wird der Begriff Softwareentwicklung verwendet, synonym dazu werden in der Literatur auch die Begriffe Softwaretechnik und Software Engineering verwendet.

[4] Vgl. u. a. MacCormack et al. 2001.

[5] Vgl. MacManus 2009 zitiert nach Gartner Inc. 2009.

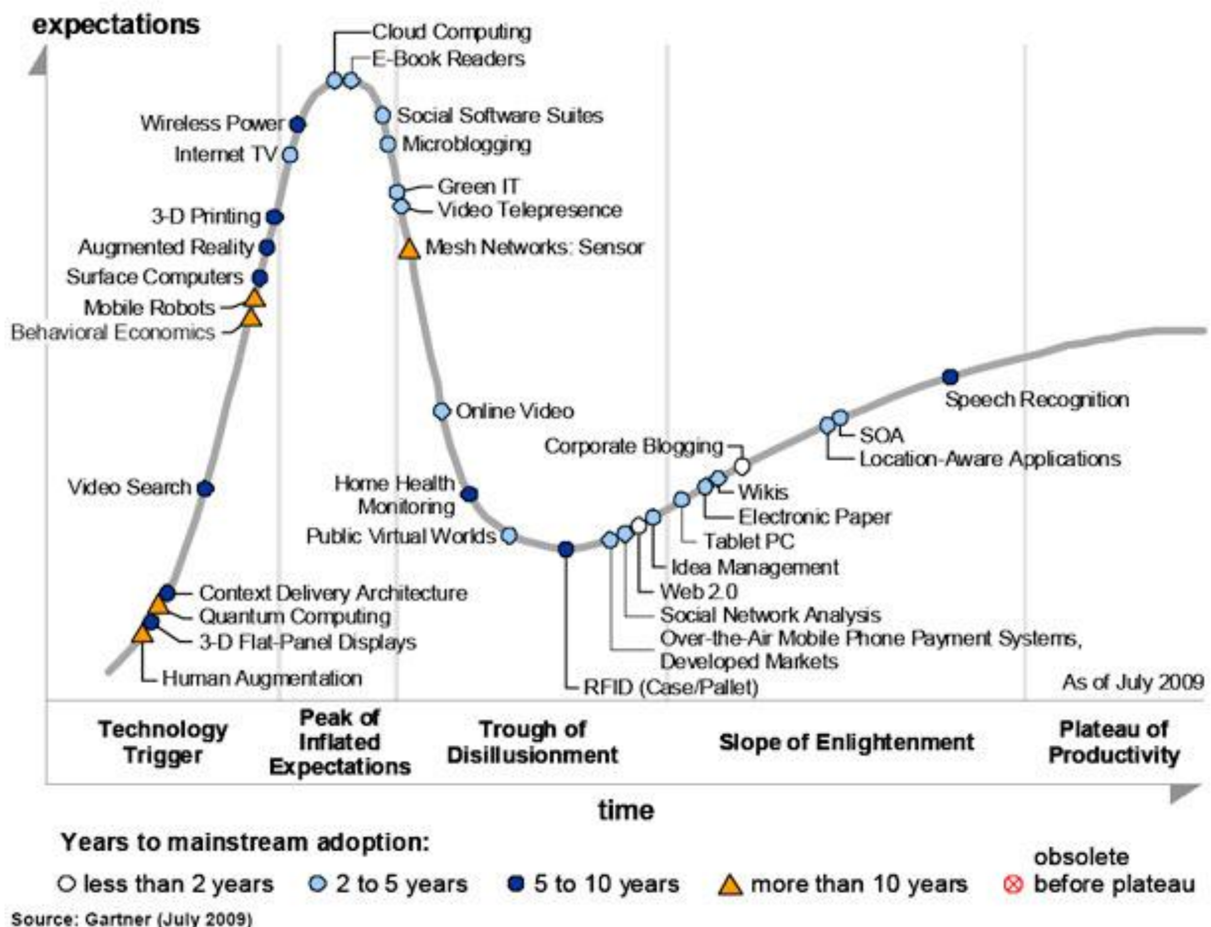


Abbildung 1-1: Hype Cycle for Emerging Technologies, 2009

Wie aus der Abbildung zu erkennen, wird in den nächsten fünf Jahren eine Vielzahl unterschiedlicher Informationstechnologien den Massenmarkt erreichen. Einen großen Markteinfluss ("*transformational*") werden dabei den Analysten zufolge u. a. *Web-2.0*, *Cloud-Computing*, und *Service-orientierte Architekturen* (*SOA*) haben.[6]

Zu diesem hoch dynamischen Umfeld kommt hinzu, dass *Softwaresysteme*[7] im Laufe der Zeit immer komplexer werden, wenn sie auf vergangenen Versionen und existierenden Technologien aufbauen, sowie unternehmensinterne und externe Software verbinden. Die wachsende unternehmerische Abhängigkeit von Software, führt nach (Lehman, Ramil

---

[6] Nach Oey et al. 2006 z. B. ist das Konzept der SOA mehr als „alter Wein in neuen Schläuchen“ – in Verdrehung eines Bibelzitats.

[7] In dieser Arbeit werden die Begriffe „Software“, „Softwaresystem“, „Softwareprodukt“ und „Informationssystem“ als Synonyme angesehen, z. T. wird für diese Begriffe die Abkürzung „SW“ verwendet.

2001a) und (Baskerville, Pries-Heje 2004) zu dem Bedürfnis einer kontinuierlichen und effektiven gemeinsamen Weiterentwicklung von Softwaresystemen und Unternehmen.

Unternehmen, wie Microsoft, haben aus diesem Grund Feedbackmechanismen entwickelt, durch die neue Anforderungen oder Änderungswünsche schon im Laufe der Produktentwicklung erkannt werden können.[8] Microsoft verteilt z. B. sogenannte „*Community Technology Previews*“ (*CTP*) an Tester, meist Entwickler aus Microsoft-Netzwerken[9]. Diese CTPs, sowie Beta-Versionen, sollen Kunden und Partner schon früh in die Entwicklung mit einbeziehen, um Feedback zu erhalten. Dieses Feedback soll dabei helfen, ein Produkt mit höchstmöglicher Qualität auszuliefern.

Während der Entwicklung des Betriebssystems Microsoft Windows Vista wurden z. B., innerhalb von zehn Monaten, zwei Beta-Versionen und vier CTPs veröffentlicht – wie in Abbildung 1-2 veranschaulicht.[10] Mit der *Beta 2* Version wurde eine vorläufige Version von Windows Vista für den „normalen“ Endnutzer zugänglich gemacht, so dass ab diesem Zeitpunkt die Verfügbarkeit des Betriebssystems verstärkt anstieg – und damit vermutlich neues Feedback eingefangen werden konnte.

---

[8] Vgl. zu diesem und dem folgenden Absatz Microsoft Corporation 2005, Wikipedia - Die freie Enzyklopädie 2010 und Abbildung 6-1 im Anhang sowie weitere mit der Entwicklung von Windows Vista zusammenhängende Pressemitteilungen der Microsoft Corporation (www.microsoft.com/presspass).

[9] Mit Microsoft-Netzwerken sind das *Windows Technical Beta Program* sowie das *Microsoft Developer Network* (*MSDN*) und *Microsoft TechNet* gemeint.

[10] Die Buildnummern stiegen dabei – bis auf einen Sprung zwischen *Beta 1* und dem zweiten CTP – relativ gleichmäßig an, was auf einen stabilen Entwicklungsprozess hindeutet. Der erste CTP wurde aus Gründen der Übersichtlichkeit nicht in der Abbildung dargestellt.

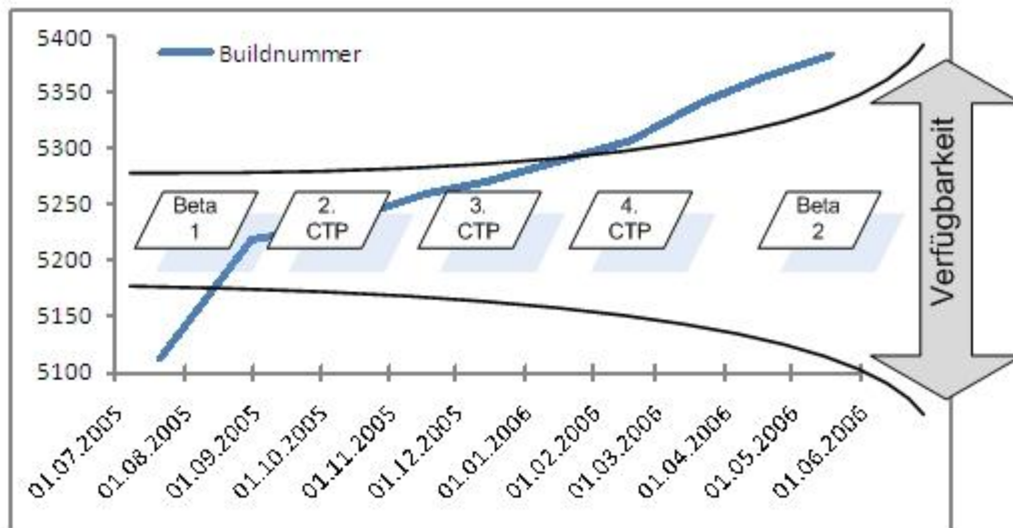


Abbildung 1-2: Die Releasezyklen von Windows Vista

1.2. Forschungsbereich

Die vorliegende Arbeit ist in der Forschung zu *inkrementellen* bzw. „agilen“[11] Softwareentwicklungsprozessen anzusiedeln. Inkrementelle Entwicklung bedeutet hier, dass – während des Entwicklungsprozesses – die noch nicht voll funktionsfähige Software (hier als Inkrement bezeichnet) dem Nutzer, z. B. als *Prototyp* oder *Beta-Version*, präsentiert wird, um Feedback einzufangen. So können Anforderungsänderungen oder neue Anforderungen frühzeitig erkannt und umgesetzt werden.

Verschiedene Autoren (u. a. Cusumano, Selby 1995; Kruchten 2007; Beck 2000; Beck et al. 2001; Cockburn 2007; Larman 2004; MacCormack et al. 2001) haben – auf Basis empirischer Untersuchungen im eingangs beschriebenen dynamischen Umfeld – inkrementelle Entwicklungsprozesse gefordert. Ergebnisse von Untersuchungen aus diesem Forschungsbereich fließen zum Teil in die Betrachtungen dieser Arbeit ein, zum Teil können die Annahmen, die diesen Ergebnissen zugrundeliegen, durch die vorliegende Arbeit kritisch analysiert werden.

[11] Die Bezeichnung „agil“ beruht auf dem „Agilen Manifest“ Beck et al. 2001. Agile Methodiken sind typischerweise gekennzeichnet durch die Unterstützung eines flexiblen, dynamischen und nutzerorientierten Entwicklungsprozesses.

Im Folgenden werden Aussagen und Untersuchungen zu Erfolgsfaktoren[12] in Entwicklungsprozessen aufgelistet und es wird auf die Motivation hinter der Verwendung inkrementeller Entwicklungsprozesse eingegangen.

A1. Frühzeitiges Feedback ist ein kritischer Erfolgsfaktor

Nach (Baskerville et al. 2002) führen inkrementelle kurze Entwicklungszyklen, und das damit verbundene frühzeitige Feedback, zu einem erhöhten Projekterfolg. Nach (Beck 2000) sollten die Anforderungen priorisiert und dann, in kurzen Entwicklungszyklen, umgesetzt werden. Diese Aussagen werden u. a. durch die empirische Untersuchung aus (Chow, Cao 2008) unterstützt. Die Autoren beobachten, dass eine regelmäßige Einführung der Software in die Nutzung, mit den wichtigsten Funktionalitäten in den ersten Inkrementen, ein kritischer Erfolgsfaktor in agilen Softwareprojekten ist. Die Untersuchung aus (MacCormack et al. 2001) zeigt, dass frühes Feedback durch die Nutzer die Softwarequalität steigert. Die obige Aussage wird durch weitere Untersuchungen verschiedener Autoren (u. a. MacCormack et al. 2001; Baskerville et al. 2002; Baskerville, Pries-Heje 2004) unterstützt.

Diese Aussage basiert u. a. darauf, dass, bei frühzeitiger Erkennung bisher unbekannter Anforderungen, das System noch nicht so komplex ist, wie zu einem späteren Zeitpunkt der Entwicklung (vgl. Mathiassen, Pedersen 2008, S. 491; Loch, Terwiesch 1998) und sich damit der Überarbeitungsaufwand reduziert.

A2. Ein inkrementeller Entwicklungsprozess ist ein Problemlösungsprozess.

Diese Aussage bezieht sich darauf, dass, durch die Nutzung von Prototypen bzw. SW-Inkrementen, Feedback eingefangen werden kann und dadurch bisher noch unbekannte Anforderungen erkannt werden (vgl. Mathiassen, Pedersen 2008, S. 491). Es wird angenommen, dass, bei Abhängigkeiten zwischen Anforderungen, eine bestimmte Anzahl von Feedbackzyklen nötig ist, um alle Anforderungen zu ermitteln.

A3. Ein inkrementeller Entwicklungsprozess führt zu einer verkürzten Reaktionszeit[13]

---

[12] Eine Definition des Begriffs „Projekterfolg“ wird hier nicht explizit gegeben, da am Seminar für Wirtschaftsinformatik und Systementwicklung zurzeit Untersuchungen zu diesem Thema laufen. Es scheint, dass eine Definition alleine auf Basis der Merkmale *Termineinhaltung*, *Budgeteinhaltung* und *Erfüllung der funktionalen Anforderungen* nicht hinreichend ist. Aus diesem Grund wird sich in dieser Arbeit auf den abstrakten Begriff beschränkt.

[13] Die Reaktionszeit entspricht der Dauer zwischen dem Auftreten einer Anforderung und der Umsetzung dieser in einem Softwareprodukt.

Im Vergleich zu traditionellen plangetriebenen Entwicklungsprozessen führt ein inkrementeller Entwicklungsprozess zu einer verkürzten Reaktionszeit (vgl. Iansiti, MacCormack 1997). Eine verkürzte Reaktionszeit bringt den Vorteil eines größeren Einführungszeitraums und die Chance auf Marktdominanz und Markterziehung[14] mit sich (vgl. Bratthall, Runeson 2000).

A4. Ein inkrementeller Prozess dient der Fortschritts- und Risikokontrolle.

Diese Aussage basiert auf der Ansicht, dass die Schätzung der Entwicklungskosten und die Planung durch Feedback nach jedem Zyklus angepasst werden kann, und diese somit im Laufe der Entwicklung immer genauer werden (vgl. Hearty et al. 2009; McConnell 1998; Kojima et al. 2008).[15]

A5. Der inkrementelle Prozess dient als Überzeugungsprozess.

Diese Aussage bedeutet, dass der Kunde, durch die frühzeitige Präsentation von Prototypen, von der Lösung seines Problems durch das SW-Produkt überzeugt wird – *„So funktioniert es!“* – (vgl. Mathiassen, Pedersen 2008, S. 491; Pew, Mavor 2007).

A6. Der inkrementelle Prozess führt zu einem erhöhten Koordinationsaufwand.

Dieser Koordinationsaufwand hat verschiedene Ursachen. Zum einen muss, während der Planung eines inkrementellen Prozesses, überlegt werden, wie der Funktionsumfang auf die Inkremente verteilt werden soll, um nützliches Feedback zu erhalten. Zum anderen führen die Kosten, die bei der Einführung der erstellten Software in die Nutzungsumgebung anfallen, zu einem erhöhten Gesamtaufwand bei einer erhöhten Anzahl von Zyklen (vgl. Mathiassen, Pedersen 2008, S. 491).

1.3. Zielsetzung

In der Softwareentwicklung verursacht der Entwicklungsaufwand, und die damit zusammenhängende Entwicklungsdauer[16], i. A. einen Großteil der Produktionskosten.[17] In einem unsicheren, dynamischen Umfeld hat die *Überarbeitungsdauer* einen großen Anteil an der Entwicklungszeit (vgl. Boehm, Papaccio 1988, S. 15; Cooper et al. 2002).

[14] D.h. Bedürfnisse werden durch das erste Produkt am Markt beeinflusst.

[15] Diese Beobachtung wird auch als „*Cone of Uncertainty*“ McConnell 1998 bezeichnet.

[16] Die Entwicklungsdauer wird in dieser Arbeit als proportional zum *Entwicklungsaufwand* angesehen. Zur Entwicklung zählen hier Analyse-, Implementierungs-, Test- und Integrationsaktivitäten.

[17] Vgl. Mellis 2004, S. 22.

Auf Basis dieser Beobachtung sollen in der vorliegenden Arbeit besonders die Einflussfaktoren der Überarbeitungsdauer inkrementeller Entwicklungsprozesse untersucht werden.

Ziel dieser Arbeit ist es

- die Struktur inkrementeller Softwareentwicklungsprozesse, die wiederholtes (zyklisches) Feedback erlauben, zu untersuchen
- Modelle zu entwickeln, anhand derer die Wirkungszusammenhänge der Einflussfaktoren der Entwicklungsdauer dieser Prozesse analysiert werden können
- auf Basis der ersten beiden Punkte, Empfehlungen zur Reduzierung der Entwicklungsdauer solcher Prozesse zu geben

Die Prozessstruktur soll auf der Ebene der Kommunikation und des Informationsflusses zwischen Analyse-, Implementierungs- und Nutzeraktivitäten betrachtet werden. Um Handlungsempfehlungen geben zu können, sollen verschiedene Planungsmöglichkeiten auf dieser Ebene in die, zu entwickelnden Modelle einfließen.

Dazu soll in den Modellen das Optimum der Entwicklungszeit, in Abhängigkeit verschiedener Einflussfaktoren und Planungsmöglichkeiten, bestimmt werden. Durch die Abbildung von Größen der Prozessstruktur und von Einflussfaktoren der Entwicklungszeit in einer Kostenfunktion, lassen sich Aussagen zur optimalen Prozessstruktur in Abhängigkeit der Einflussfaktoren herleiten. Das Ergebnis dieser Optimierung scheint nicht intuitiv vorhersehbar zu sein, wenn ein Faktor gleichzeitig positive und negative Einflüsse auf die Entwicklungszeit besitzt, und soll analytisch hergeleitet werden.

1.4. Vorgehensweise

Im Rahmen dieser Arbeit wird die Struktur inkrementeller Entwicklungsprozesse untersucht, und es werden Handlungsempfehlungen unter Berücksichtigung verschiedener Einflussfaktoren hergeleitet. Dabei werden Untersuchungen sowohl aus dem Bereich der Softwareentwicklung als auch aus dem Bereich der Produktentwicklung im Allgemeinen herangezogen.

Im Forschungsbereich der Produktentwicklung existieren verschiedene Modelle, auf Basis von Kostenfunktionen, für sogenannte nebenläufige bzw. überlappende

Entwicklungsprozesse (engl.: *concurrent engineering*), die strukturell den inkrementellen Prozessen der Softwareentwicklung gleichen. Ausgangspunkt der Literaturrecherche in diesem Bereich bildet für die vorliegende Arbeit der, in der Zeitschrift „Management Science" veröffentlichte, Artikel „Communication and Uncertainty in Concurrent Engineering" (Loch, Terwiesch 1998). Die Autoren stellen eine Kostenfunktion zur Optimierung der Planung von Entwicklungsprozessen auf und modellieren dabei die Konzepte der Evolution und Sensitivität des zu entwickelnden Produkts.[18] Diese Modellierungsansätze werden in der vorliegenden Arbeit auf eine Kostenfunktion der Softwareentwicklung übertragen und kritisch hinterfragt. In dem Entwicklungsprozess aus (Loch, Terwiesch 1998) ist der Informationsfluss einseitig gerichtet, und es wird kein Feedback aus der Nutzung – wie es in inkrementellen Entwicklungsmethodiken gefordert wird – berücksichtigt. Aus diesem Grund wird dieser Entwicklungsprozess um Prototyping und einen Feedbackzyklus erweitert. So können die Auswirkungen von regelmäßigem Feedback auf die Entwicklungskosten analysiert werden.

Um die Annahmen des Modells aus (Loch, Terwiesch 1998) vollständig zu überprüfen und mögliche weiterführende Forschungsbeiträge zu finden, werden alle, in diesem Artikel zitierten, Quellen untersucht und es wird zusätzlich eine Vorwärtssuche aller Artikel, die (Loch, Terwiesch 1998) zitieren, mit Hilfe von Literaturdatenbanken und Verlinkungen durchgeführt.[19]

Zur Identifizierung von Kostenfunktionen und Einflussfaktoren der Softwareentwicklung werden zwei Literaturreviews zur Kostenschätzung (Jörgensen, Shepperd 2007 und Boehm et al. 2000) als Grundlage herangezogen und die dort genannten Artikel weiter zielgerichtet untersucht. Es werden Faktoren mit signifikantem Einfluss auf die Entwicklungszeit analysiert. Mögliche Metriken für diese Faktoren stehen nicht im Mittelpunkt dieser Arbeit, sie werden aber dennoch betrachtet, da die Qualität der Planung des Entwicklungsprozesses von der Qualität der Metriken abhängt. Tom DeMarco schrieb dazu den viel zitierten Satz "you can't control what you can't measure" (DeMarco 1982, S. 6)[20]. Das Ergebnis der eigenen Untersuchung dient u. a. der Übertragung des Modells aus (Loch, Terwiesch 1998) auf die Softwareentwicklung.

---

[18] Diese Konzepte stammen aus Krishnan, Eppinger 1997.

[19] Hierzu wurde die Online-Literaturrecherche EBSCOhost mit den Datenbanken „Business Source Complete" und „Academic Search Complete" sowie Google Scholar verwendet.

[20] Übersetzt: Was man nicht messen kann, kann man nicht kontrollieren bzw. steuern.

Zudem werden in der vorliegenden Arbeit die Forderungen und Kritikpunkte zu Kostenfunktionen nach (Dolado 2001) berücksichtigt. Der Autor beschreibt, dass die Literatur zur Planung der Entwicklungsaktivitäten meist auf der Betrachtung des Produktumfangs und des Entwicklungsaufwands als Kostentreiber basiert. Eine Planung mit Hilfe von Kostenfunktionen alleine auf Basis dieser Faktoren, ist dem Autor zufolge nicht auf die Besonderheiten der Softwareentwicklung ausgerichtet. Seine Untersuchungen zeigen, dass die Genauigkeit dieser Schätzungen nicht das Niveau von Expertenschätzungen erreicht. In (Dolado 2001) wird kritisiert, dass die Bewertung alternativer Modelle alleine durch statistische Analysen durchgeführt wird. Der Autor fordert, dass auch die Annahmen, unter denen die Modelle erstellt wurden, untersucht werden müssen.

Des Weiteren wird die Literatur zur Softwareentwicklung anhand ausgewählter Schlagwörter und Zeitschriften nach Beschreibungen von Softwareentwicklungsprozessen durchsucht. Anhand dieser Artikel soll ein einheitliches Begriffssystem definiert werden, um einen konkreten Bezug zwischen der eigenen Arbeit und verschiedenen aus der Literatur bekannten Prozessmodellen zu schaffen.

Weitere nahestehende Forschungsarbeiten werden im weiteren Verlauf dieser Arbeit identifiziert und in die Betrachtungen aufgenommen, oder es wird eine Abgrenzung zu diesen Arbeiten aufgezeigt.

1.5. Aufbau der Arbeit

Die Arbeit ist in sechs Kapitel aufgeteilt. Auf ein allgemeines Kapitel zu „Software-Entwicklung“ und „Software-Architektur“ wird bewusst verzichtet, da der interessierte Leser hierzu ausreichend Fachliteratur (z. B. Sommerville 2007; Balzert 2001; Stahlknecht, Hasenkamp 2005; Bass et al. 2005) vorfindet.

In Anschluss an dieses Einführungskapitel werden – in Kapitel 2 – inkrementelle zyklische Prozesse zur Entwicklung neuer Softwareprodukte vorgestellt und diese u. a. mit plangetriebenen Entwicklungsprozessen verglichen. Zunächst werden SW-Entwicklungsprozesse im Allgemeinen und zugehörige Aktivitäten und Artefakte beschrieben. Dann werden verschiedene Zyklusebenen innerhalb dieser Prozesse näher untersucht. Als Fallbeispiele für zyklische Entwicklungsprozesse werden die Extreme-Programming-Methodik, die Rational-Unified-Process-Methodik sowie die Synch-and-

Stabilize-Methodik herangezogen. Am Ende des Kapitels werden, anhand der Prozessstruktur, Steuerungsmöglichkeiten der Prozessplanung betrachtet.

In Kapitel 3 werden die Einflussfaktoren des Entwicklungsaufwands identifiziert und detailliert analysiert. Hier wird die Anforderungsunsicherheit diskutiert und Ausprägungen dieser Unsicherheit sowie mögliche Maßnahmen zum Umgang mit verschiedenen Ausprägungen betrachtet. Weiterhin wird die Sensitivität bzw. die Flexibilität der Entwicklung komplexer Systeme analysiert. Die Komplexität wird unterteilt in die Abhängigkeiten zwischen Anforderungen, die Hierarchie und Modularität des Softwaresystems und die Strukturen auf organisatorischer Ebene. Am Ende des Kapitels wird eine Übersicht der, für diese Arbeit relevanten, Einflussfaktoren gegeben.

In Kapitel 4 werden Kostenfunktionen für Prozesse mit zyklischem Nutzerfeedback untersucht. Dazu werden zunächst verschiedene, allgemeine Kostenfunktionen zur Aufwandsschätzung in der Softwareentwicklung vorgestellt. In weiteren Abschnitten werden die Modellierung der identifizierten Einflussfaktoren und Steuerungsmöglichkeiten der Prozessplanung innerhalb dieser Kostenfunktionen analysiert. Die Modellierung von Skaleneffekten, d. h. Größenvor- und nachteilen, wird vertieft, da diese ein Unterscheidungsmerkmal der verschiedenen Kostenfunktionen darstellt.

In Kapitel 5 werden, durch die Optimierung von Kostenfunktionen, Handlungsempfehlungen zur Prozessplanung hergeleitet. Dazu wird zunächst die Übertragung verschiedener Annahmen und Ergebnisse aus allgemeinen Modellen der Produktentwicklung auf die Softwareentwicklung analysiert. Darauf aufbauend werden dann verschiedene Kostenfunktionen zur Planung von Entwicklungsprozessen mit zyklischem Nutzerfeedback definiert. Diese Kostenfunktionen unterscheiden sich u. a. in der Modellierung von Feedback-Effekten und der Berechnung des Überarbeitungsaufwands.

Im letzten Kapitel findet sich eine Zusammenfassung und Diskussion der Arbeit.

Im Anhang wird die analytische Herleitung und Optimierung der Kosten in den verschiedenen Modellen detailliert dargestellt.

Kapitel 2: Prozesse zur Entwicklung von Softwareprodukten

In diesem Kapitel wird die Struktur von SW-Entwicklungsprozessen i. A. analysiert und ein für diese Arbeit gültiges Begriffssystem definiert. Dazu werden die Anforderungen an eine Software, als Startpunkt des Entwicklungsprozesses, und das Softwareprodukt, als Ergebnis des Prozesses, betrachtet. Verschiedene Prozessmodelle aus der Literatur werden auf ihre Eignung in unsicheren Umgebungen untersucht. Diese Prozessmodelle werden auf einer Makroebene in *inkrementelle* und *plangetriebene* Prozessmodelle unterschieden. Aus der Kategorie der inkrementellen Modelle werden drei Methodiken ausgewählt und als „Best Practice" dargestellt: die *Rational-Unified-Process*-Methodik, die *Extreme-Programming*-Methodik und die *Synch-and-Stabilize*-Methodik.

Die beiden oben genannten Kategorien von Prozessmodellen können unabhängig vom Einsatz *schwergewichtiger* oder *agiler* Methoden betrachtet werden.[21] Plangetriebene Prozessmodelle werden – vermutlich aus historischen Gründen – oft mit schwergewichtigen Methoden gleichgesetzt. Agile Methoden sind aber auch in plangetriebenen Prozessen anwendbar, so z. B. die Methode „Priorisierung der Funktionalitäten nach Risiken" (vgl. Boehm 1984) oder die Methode „Testgetriebene Entwicklung" (vgl. Beck 2000). In (Beck 2000) wird ein kurzzyklisches inkrementelles Prozessmodell als agile Methode betrachtet.

2.1. Von Anforderungen zum Softwareprodukt

Zunächst sollen die Anforderungen an ein Softwareprodukt und das Softwareprodukt an sich kurz betrachtet werden, um die Aufgaben und Probleme des Entwicklungsprozesses zu verdeutlichen.

Anforderungen und Spezifikation

Anforderungen an ein Softwareprodukt werden, z. B. durch einen Auftraggeber, in einem *Lastenheft* formuliert.[22] Die Anforderungen beinhalten Aussagen über eine Anwendungsdomäne, die einen abgrenzbaren Problembereich darstellt. Die wichtigste

---

[21] Vgl. zu diesem Absatz auch Larman 2004, S. 173. Schwergewichtige Methoden werden dort auch unter dem Begriff „*ceremony*" betrachtet.

[22] Vgl. zu diesem Absatz Zave, Jackson 1997, S. 7.

Unterscheidung dieser Aussagen wird durch zwei grammatische Modi erfasst: Zum einen wird durch *indikative* Aussagen der Zustand der Anwendungsdomäne beschrieben. Diese Aussagen können als Annahmen oder Domänenwissen bezeichnet werden. Auf der anderen Seite wird durch „*optative*" Aussagen beschrieben, wie die Anwendungsdomäne sein sollte, insbesondere wenn das Softwaresystem in dieser Domäne eingesetzt wird. Optative Aussagen werden i. A. als Anforderungen bezeichnet.[23] Die Anforderungen sollten nach (Jackson 1995) als Beziehungen zwischen Eigenschaften der Anwendungsdomäne, unabhängig von den Eigenschaften eines Softwaresystems, formuliert werden. Die Anforderungen an ein Softwareprodukt lassen sich weiter in funktionale und nicht-funktionale (qualitative) Anforderungen unterteilen (z. B. nach Bass et al. 2005, S. 72).

Die Aufgabe der Anforderungsanalyse besteht darin diese Kundenanforderungen zu erfassen und, durch eine detaillierte Beschreibung der Systemanforderungen (die *Spezifikation*), zu ergänzen (vgl. z. B. Sommerville 2007, S. 130). Diese Spezifikation beschreibt nach (Jackson 1995) das externe Verhalten des Softwaresystems und seine Eigenschaften. Dieses externe Verhalten, als Schnittstelle zwischen Softwaresystem und Anwendungsdomäne, wird dadurch beschrieben, wie Ereignisse – meist durch den Nutzer – ausgelöst, durch das Softwaresystem verarbeitet werden (Reaktion) und welche Auswirkungen das Resultat wiederum auf die Anwendungsdomäne hat. Die Auswirkungen der Ereignisse können bei unterschiedlichen, durch die Software angesprochenen, Systemen wiederum unterschiedlich sein. Die Reaktion des Softwaresystems auf ein Ereignis geschieht durch die Ausführung verschiedener Systemfunktionen.

Aus diesen Betrachtungen folgt, dass Anwendungsdomäne und Softwaresystem an den Schnittstellen gemeinsame Eigenschaften haben, und im Softwaresystem ein Modell der Anwendungsdomäne abgebildet werden muss. Die Spezifikation wird im Pflichtenheft festgehalten und fließt zusammen mit dem Lastenheft in das Anforderungsdokument (engl.: *software requirements specification*, vgl. Sommerville 2007, S. 136) ein. Kundenanforderungen und Systemanforderungen werden in dieser Arbeit, der Einfachheit halber, oft unter dem Oberbegriff „Anforderungen" zusammengefasst. Der Unterschied sollte aber durch obige Erläuterungen bewusst gemacht werden.

---

[23] Diese Sichtweise vermeidet Probleme, die durch Einflüsse aus der Implementierung entstehen, da keine

Im Folgenden wird ein Beispiel für eine Anforderung vorgestellt, welches an verschiedenen Stellen dieser Arbeit wieder aufgegriffen wird.

Beispiel 2-1: Eine Anforderung mit zugehöriger Spezifikation

Die Anforderungen an die Webseite eines Online-Versandhauses beinhalten u. a. die Anforderung:

Ein Kunde kann einen Artikel ansehen. Ein Artikel kann ein Buch oder eine CD sein.

Diese Beschreibung muss durch eine detaillierte Spezifikation erweitert werden. Diese beinhaltet folgende Systemanforderungen:

a) Der Kunde kann Artikeldaten, Preis und Rezensionen ansehen

b) Der Kunde kann, durch Klick auf einen Button, den Artikel in den „Warenkorb“ legen

c) Der Kunde kann bei einem Buch das Inhaltsverzeichnis ansehen

d) Der Kunde kann bei einer CD die Songs anhören

Diese Systemanforderungen werden, abhängig von der Analysemethode, i. A. durch weitere Angaben, wie Vor- und Nachbedingungen, ergänzt. Aus Sicht des Kunden werden die implementierten Systemanforderungen als *Funktionalitäten* oder *Features* bezeichnet.

Im problemorientierten Ansatz nach (Jackson 1995) folgt: Wenn das Softwaresystem sich wie spezifiziert verhält, und die Anwendungsdomäne die vermuteten Eigenschaften hat, dann kann die Erfüllung der Anforderungen gefolgert werden. Das Wissen um die Anwendungsdomäne ist jedoch oft zu schwach ausgeprägt, um einen solchen problemorientierten Ansatz zu wählen (vgl. Curtis et al. 1988, S. 1271).

Methoden zur Anforderungsanalyse

Es existieren viele unterschiedliche Methoden zur Anforderungsanalyse (vgl. Sommerville 2007, S. 148–156), von denen eine Auswahl hier kurz vorgestellt werden soll.

Aussagen über das zu erstellende Softwaresystem an sich getroffen werden.

In der Extreme-Programming-Methodik (XP) nach (Beck 1999) werden Anforderungen durch sogenannte *Stories* beschrieben.[24] Eine Story besteht aus:

- den geschriebenen Anforderungen, die bewusst auf kleine 3x5-Zoll Indexkarten geschrieben werden, um die Menge der Informationen zu begrenzen;
- der Unterhaltung zwischen Entwickler- und Businessteams über die Anforderung, um die Anwendungsdomäne zu verstehen;
- dem Nutzerakzeptanztest auf den sich die *Stakeholder*[25] einigen, um die Erfüllung der Anforderungen später bestätigen zu können. Ein Akzeptanztest besteht meist aus verschiedenen Nutzereingaben und den dazugehörigen erwarteten Systemausgaben.

Die Verfeinerung von Stories wird in XP in *Tasks* niedergeschrieben. Ein Task kann auch mehrere Stories unterstützen (vgl. Beck 2000, S. 91), z. B. wenn es sich bei dem Task um die Implementierung einer Systemfunktion handelt.

Eine andere Möglichkeit besteht in der Erfassung der Anforderungen durch Methoden der *Unified-Modeling-Language* (*UML*).[26] In *Use-Case*-Diagrammen werden die, durch das System zu unterstützenden, Ereignisse aus Sicht der Nutzer beschrieben. Durch eine detaillierte Beschreibung der Use-Cases, u. a. durch Eingabedaten sowie Vor- und Nachbedingung, kann eine Spezifikation erstellt werden. In Beispiel 2-1 bildet jede der Systemanforderungen einen Use-Case. Die UML bietet weitere Methoden, wie Klassendiagramme, um basierend auf der Spezifikation den Feinentwurf des Systems durchzuführen.

Durch Methoden im Rahmen des sogenannten *Quality-Function-Deployment* (*QFD*) nach (Akao 1972) können die Zusammenhänge zwischen den Systemfunktionen und den Kundenanforderungen bestimmt und, durch Gewichtung der Anforderungen, eine Priorisierung der Funktionen für die Implementierung erreicht werden.

---

[24] Vgl. zu diesem Absatz Davies, Sharp 2006.

[25] Der Begriff Stakeholder wird verwendet, um sich auf alle Personen und Interessengruppen zu beziehen, die von dem Softwaresystem direkt oder indirekt betroffen sind (vgl. Sommerville 2007, S. 146). Interessengruppen sind z. B. die Gruppe der Endnutzer, der Entwickler oder der Auftraggeber.

[26] Vgl. zu diesem Absatz Balzert 2001. Dort finden sich weiterführende Informationen zur UML.

Das Softwareprodukt

Software, als Produkt des Softwareentwicklungsprozesses, beinhaltet hier die „Software im engeren Sinne“ (vgl. Mellis 2004, S. 2), als die „Gesamtheit der Programme, die zum Betreiben und Steuern einer bestimmten (Klasse von) Hardware und der Erfüllung einer bestimmten (Klasse von) Informationsverarbeitungsaufgabe(n) benutzt wird“[27].

Je nach betrachteter Ebene[28] des SW-Entwicklungsprozesses werden weitere Leistungen wie z. B. die Anforderungsbeschreibungen, Benutzerhandbücher und die Schulung der Nutzer dazu gezählt.[29]

An der Definition der Software im engeren Sinne ist eine spezielle Eigenschaft von Softwareprodukten zu erkennen: Ein Softwareprodukt wird i. d. R. nicht nur auf einem Endgerät als einzelne Installation genutzt, sondern auf verschiedenen Endgeräten installiert und kann dort nicht nur eine einzelne Eingabe, wie z. B. die Datei ‚Song.mp3‘, verarbeiten, sondern verschiedene Informationen einer Klasse, wie z. B. Musikdateien. Diese Möglichkeit birgt aber – aus Erfahrungen des Autors dieser Arbeit – auch die Schwierigkeit eine geeignete Softwarearchitektur zu finden, welche „so generell wie nötig und so speziell wie möglich ist“.

Die Eigenschaften eines Softwareprodukts können – analog zu den oben beschriebenen Anforderungen – in funktionale und nicht-funktionale Eigenschaften unterschieden werden. Die funktionalen Eigenschaften werden in dieser Arbeit auch als Funktionalität oder als Features bezeichnet, die nicht-funktionalen Eigenschaften als Qualität des Softwareprodukts. Jede implementierte Funktionalität einer Software besitzt eine bestimmte Qualität. Die Auswahl und Implementierung eines Algorithmus wirkt i. d. R. auf Qualitätsmerkmale wie Performance, Sicherheit, Bedienbarkeit und Änderbarkeit. Der Forschungsfortschritt der Informatik bestimmt, ob z. B. ein Algorithmus zur Implementierung einer bestimmten Funktionalität unter Einhaltung der Qualitätsanforderungen existiert.

---

[27] Vgl. Seibt 1972, S. 23 zitiert nach Mellis 2004, S. 2.

[28] Zur Beschreibung der verschiedenen Ebenen vgl. Abschnitt 2.3.

[29] Vgl. „Software im weiteren Sinne“ und „im umfassenden Sinne“ in Mellis 2004, S. 2–3.

2.2. Prozessaktivitäten, Phasen und Rollen

Nach (Kruchten 2007) beschreibt ein Prozess “wer was wie und wann macht“. Ein Prozess besteht dem Autor zufolge aus den *Rollen* (wer), den *Artefakten* (was), den *Aktivitäten* bestehend aus Aufgaben (wie) und dem *Workflow* (wann). Diese Arbeit fokussiert sich verstärkt auf die Aufgaben bzw. Aktivitäten und deren Ausführungszeitpunkte.

Zusammenhänge zwischen Prozesseigenschaften

Abbildung 2-1 zeigt beispielhaft den Zusammenhang zwischen diesen verschiedenen Prozesseigenschaften. In dieser Arbeit wird besonders der Zusammenhang zwischen Aktivitäten und Artefakten behandelt. Dazu werden Flussdiagramme verwendet, um den Zeitpunkt der Ausführung von Aktivitäten (z. B. die Reihenfolge), sowie den Ablauf von Feedback, Schleifen, Zyklen, komplexe Bedingungen zur Entscheidungsfindung, Eintritts- und Austrittskriterien darzustellen.[30] Die Aktivitäten selber werden als Blackbox dargestellt. Die Informationseinheiten, die durch die Aktivitäten erstellt bzw. verarbeitet und weitergegeben werden, beinhalten Daten, Artefakte, Produkte (Zwischen- und Endprodukte), und Gegenstände.

---

[30] Den Flussdiagrammen wird in Gomes, Joglekar 2008 die Transaktionssicht der Produktentwicklung gegenübergestellt, die eine Kombination aus Prozessflussdiagramm und organisatorischer Sicht darstellt und die Transaktionskosten auf organisatorischer Ebene untersucht.

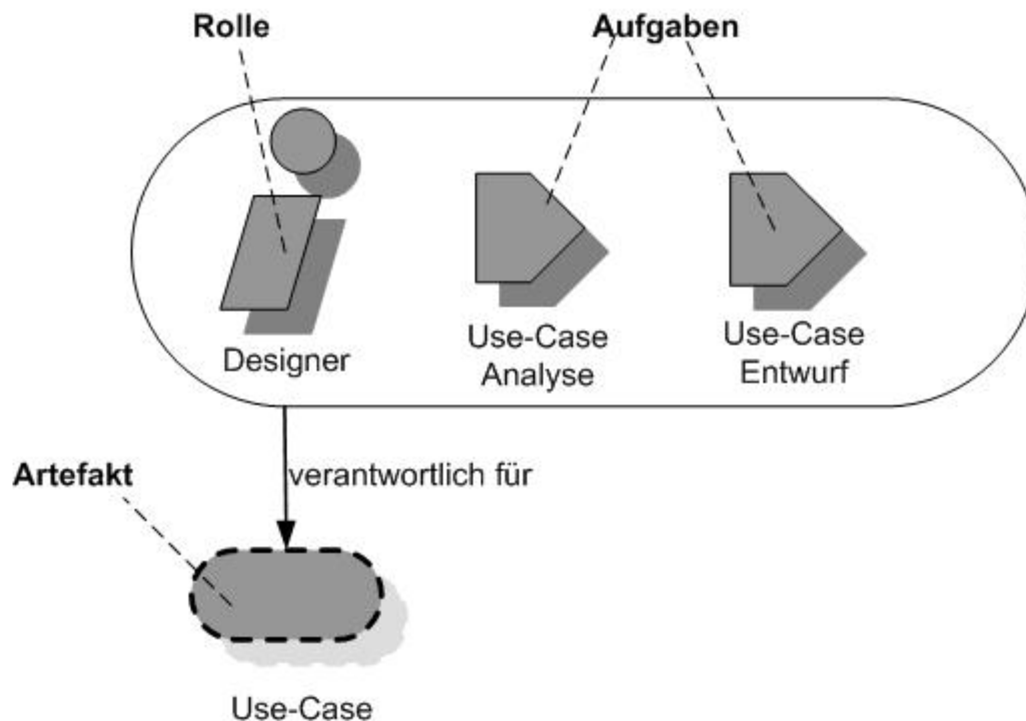


Abbildung 2-1: Zusammenhang von Rolle, Aufgaben und Artefakt am Beispiel von Aufgaben der Anforderungsanalyse

Aus organisatorischer Sicht wird in dieser Arbeit der Zusammenhang zwischen Aktivitäten und Rollen, sowie zwischen SW-Architektur und Teamstruktur, behandelt.

Aktivitäten

Ein Softwareentwicklungsprozess wird nach (ISO/IEC 12207) aus einer Menge von Aktivitäten entworfen, von denen jede wiederum aus einer Menge von Aufgaben entworfen wird. Eine Aufgabe ist eine Menge von elementaren Aktionen. Eine Aufgabe wandelt Eingaben (wie Anforderungen und Zeitbudget) in Ausgaben (wie Softwarefunktionen) um.

In dieser Arbeit werden die Aktivitäten innerhalb eines SWE-Prozesses in

- *Analyseaktivität*
- *Implementierungsaktivität*
- *Einführungsaktivität*

unterschieden. Weitere in der Literatur (z. B. in Kruchten 2007, S. 93) auch getrennt betrachtete SWE-Aktivitäten werden diesen untergeordnet. So werden die Projektplanung, die Anforderungsanalyse und der Einsatz von Feedbackmechanismen der

Analyseaktivität zugeordnet. Der architektonische Entwurf, das detaillierte implementierungsspezifische Design (Feinentwurf), die Codierung und Testaktivitäten werden der Implementierungsaktivität zugeordnet. Die Einführungsaktivität besteht aus der Überführung der Software in die Nutzung mit allen damit verbundenen Aufgaben, die in späteren Kapiteln noch gesondert analysiert werden. Tabelle 2-1 stellt die Aktivitäten mit ausgewählten zugehörigen Aufgaben übersichtlich dar.

Tabelle 2-1: SWE-Aktivitäten mit ausgewählten zugehörigen Aufgaben

SWE-Aktivität	*Aufgaben*
Analyse	Projektplanung, Anforderungsanalyse, Einfangen von Feedback
Implementierung	Architekturmaßnahmen, Feinentwurf, Codierung, Test, Integration
Einführung[31]	Systemtests, Qualitätsoptimierung, Installation, Dokumentation

Abbildung 2-2 zeigt beispielhaft ein Flussdiagramm eines SWE-Prozesses mit verschiedenen Schleifen bzw. Zyklen. Diese Abbildung resultiert aus Betrachtungen aus (Beck 2000; Cockburn 2007; Kruchten 2007; Loch, Terwiesch 1998; Qi Feng et al. 2008; McConnell 1996, S. 427) und (Rauterberg 1992).

Wie in der Abbildung zu erkennen, werden zwischen den verschiedenen Aktivitäten Informationen weitergegeben. Zum Beispiel liefert die Anforderungsanalyse spezifizierte Anforderungen an die Implementierung, welche von dieser verarbeitet werden.[32] Die Implementierung erzeugt eine lauffähige neue Softwareversion (*Build*). Dieser Build, auch als internes Release bezeichnet, kann nun in die Nutzung überführt werden oder er dient „nur“ als Zwischenergebnis (z. B. zur Fortschrittskontrolle). Die Durchführung der Einführungsaktivität resultiert in einem (externen) *Release*. Durch Nutzung dieses Releases können wieder neue Anforderungen entstehen, die – in einem weiteren Releasezyklus – durch die Analyseaktivität ermittelt werden. Die dargestellten

[31] Die Einführungsaktivität ist in der englischsprachigen Literatur auch unter den Begriffen „*deployment*“, „*transition*“, „*productionizing*“ oder „*release management*“ zu finden (vgl. z. B. Beck 2000 oder Kruchten 2007).

[32] Ein – in den meisten Projekten anzunehmender – Informationsfluss von Implementierungs- zu Analyseaktivität ist in dieser Abbildung aus Gründen der Übersichtlichkeit nicht dargestellt.

Informationseinheiten bzw. Artefakte, welche durch die Aktivitäten bearbeitet werden, sowie weitere Einflussfaktoren werden in Kapitel 3 näher betrachtet.

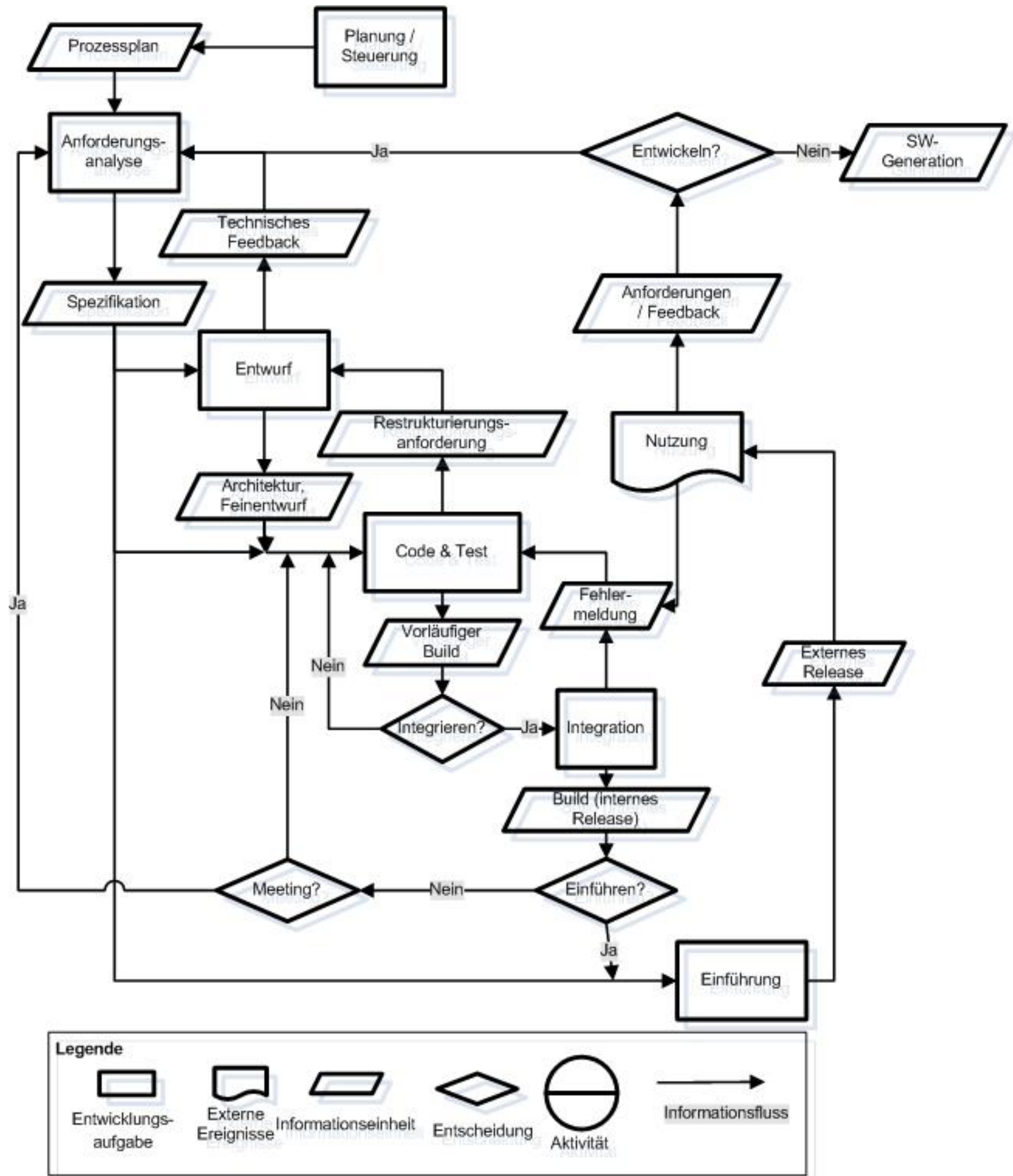


Abbildung 2-2: Detailliertes Flussdiagramm eines zyklischen SWE-Prozesses mit verschiedenen Entscheidungssituationen.

Des Weiteren sind in der Abbildung an den Entscheidungspunkten verschiedenen Handlungsmöglichkeiten dargestellt. Diese Handlungsmöglichkeiten werden durch die in dieser Arbeit betrachteten Modelle analysiert.

Phasen

Der Zeitraum von der erstmaligen Durchführung einer Aktivität im Entwicklungsprozess bis zur Abnahme der Software entspricht, im Verständnis dieser Arbeit, dem Entwicklungszyklus einer *SW-Generation* (vgl. Kruchten 2007, S. 90–91). Bei der Entwicklung einer SW-Generation werden verschiedene *Phasen* durchlaufen, die sich dadurch unterscheiden, dass, innerhalb dieser Phasen, die verschiedenen Aktivitäten unterschiedlich stark gewichtet sind, d. h. mit unterschiedlichem Aufwand und Priorität betrieben werden. Diese Gewichtung ist ein Merkmal zur Unterscheidung von plangetriebenen und inkrementellen Entwicklungsprozessen. Die Gewichtung innerhalb dieser Prozesse wird in den Abschnitten 2.4 und 2.5 näher betrachtet.

In dieser Arbeit wird der Entwicklungszyklus einer SW-Generation, ähnlich der Rational-Unified-Process-Methodik (RUP) nach (Kruchten 2007) wie sie in Abschnitt 2.5.2 dargestellt wird, in

- *Startphase*
- *Konstruktionsphase*
- *Finalisierungsphase*

unterteilt. In (Bennett, Rajlich 2000) untersuchen die Autoren die Ausprägung und Zusammenhänge der verschiedenen Phasen in Fallstudien.[33] In der Startphase liegt der Schwerpunkt auf der Analyseaktivität. Verschiebt sich der Schwerpunkt in Richtung Implementierungsaktivität beginnt die Konstruktionsphase, dies kann durch einen *Meilenstein* signalisiert werden.[34] In dieser Phase liegt das größte Gewicht auf der Implementierungsaktivität. Je nach Prozessmodell wird hier noch Anforderungsanalyse betrieben und es können schon Einführungsaktivitäten stattfinden. Die Konstruktionsphase und die Anzahl der Zyklen innerhalb dieser Phase sind Fokus der, in dieser Arbeit betrachteten, Modelle. Die letzte Phase, die Finalisierungsphase, beginnt, wenn der größte Arbeitsanteil auf der Behebung von Fehlern und der Einführung der

---

[33] Die Phase “Initial Development” aus Bennett, Rajlich 2000 kann der Startphase und die “Evolution” Phase der Konstruktionsphase zugeordnet werden. Das Ende einer SW-Generation wird in Bennett, Rajlich 2000 erreicht, wenn das SW-Produkt vom Hersteller nicht mehr unterstützt und vom Markt genommen wird.

[34] In Bennett, Rajlich 2000 endet die Startphase mit der Auslieferung der ersten lauffähigen SW-Version.

Software liegt. Die letzte, die aktuelle SW-Generation betreffende, Analysetätigkeit muss vor den letzten Implementierungs- und Einführungsaktivitäten stattfinden, da neue Anforderungen ansonsten nicht mehr umgesetzt werden.

Rollen

Eine *Rolle* definiert nach (Kruchten 2007) das Verhalten und die Verantwortlichkeiten eines Mitarbeiters oder einer zusammenarbeitenden Gruppe von Mitarbeitern. Jede Rolle ist mit einer Menge zusammenhängender Aufgaben verbunden. D. h. diese Aufgaben werden im Idealfall von einer Person durchgeführt. Jede Rolle ist verantwortlich für bestimmte Artefakte, die durch diese Rolle erstellt, verändert oder kontrolliert werden. Mitarbeitern können verschiedene Rollen zugeordnet werden.

Die Rollen in Entwicklungsprozessen orientieren sich an den Aktivitäten und Aufgaben. Ausgewählte Rollen sind in Tabelle 2-2 den Aktivitäten zugeordnet. Eine Beschreibung der Rollen in den Methodiken RUP und XP findet sich in folgenden Abschnitten.

Tabelle 2-2: Rollen in Softwarenentwicklungsprozessen

SWE-Aktivität	*Rolle*
Analyse	Systemanalyst, Prozessmanager
	Change Manager
Implementierung	Architekt, Designer, Entwickler, Testdesigner, Tester
Einführung	Tester, Technische Redakteure, Support

2.3. Zyklische Entwicklung auf verschiedenen Ebenen

Entwicklungszyklen (oder auch *Iterationen*) existieren auf verschiedenen Ebenen und können ineinander verschachtelt sein, so dass es zu Missverständnissen führen kann, nur von Zyklen zu sprechen, ohne den genauen Inhalt und Umfang eines Zyklus zu definieren. Der Begriff Zyklus bedeutet zunächst einmal, dass ein bestimmter Arbeitsablauf beliebig oft wiederholt werden kann. In (Cockburn 2007, S. 212) werden die verschiedenen *Ebenen* verschachtelter Zyklen diskutiert. Dort werden die Zyklen in

Projekt-, Auslieferungs-, Iterations-, Tages-, und Integrationsebene unterschieden.[35] Abbildung 2-3 zeigt die in verschiedenen Modellen dieser Arbeit untersuchten Zyklusebenen, die im Folgenden beschrieben werden.

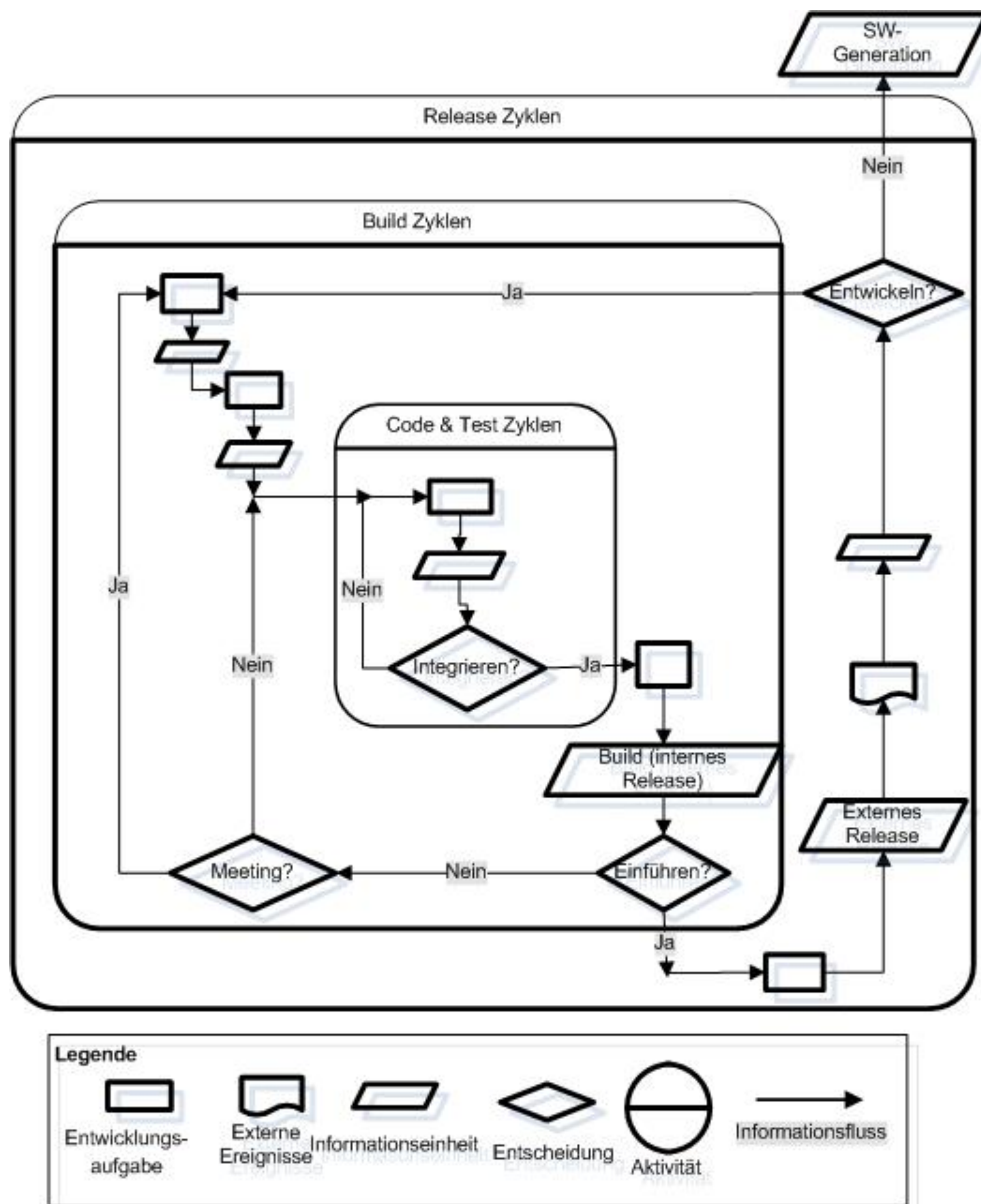


Abbildung 2-3: Verschiedene Ebenen von Entwicklungszyklen[36]

---

[35] Auf einigen Ebenen in Cockburn 2007, S. 212 richtet sich die Dauer und Anzahl der Zyklen nach dem Arbeitsrhythmus, d. h. nach Wochen und Tagen. Auf höheren Ebenen sind die Zyklen stärker planungsabhängig und erfolgskritisch.

[36] Eigene Abbildung in Anlehnung an Cockburn 2007, S. 212.

Die SW-Generation

Die oberste Ebene besteht hier in der Entwicklung einer SW-Generation. Der Entwicklungszyklus einer SW-Generation beginnt – wie in Abschnitt 2.2 beschrieben – mit der Startphase, besteht dann aus aufeinanderfolgenden Releasezyklen in der Konstruktionsphase und endet nach Abschluss der Finalisierungsphase. In Abbildung 2-3 endet der Zyklus einer SW-Generation, wenn die Frage „Entwickeln?" mit „Nein" beantwortet wurde, dies kann z. B. der Fall sein, wenn alle bekannten Anforderungen abgearbeitet wurden und keine neuen dazu gekommen sind.

In möglicherweise folgenden *Weiterentwicklungszyklen* werden die nächsten Softwaregenerationen entwickelt. Weiterentwicklungszyklen können nach (Kruchten 2007) durch Verbesserungswünsche der Nutzer, Änderungen in der Nutzungsumgebung, Änderungen in zugrundeliegenden Technologien oder Reaktionen auf Mitbewerber ausgelöst werden.

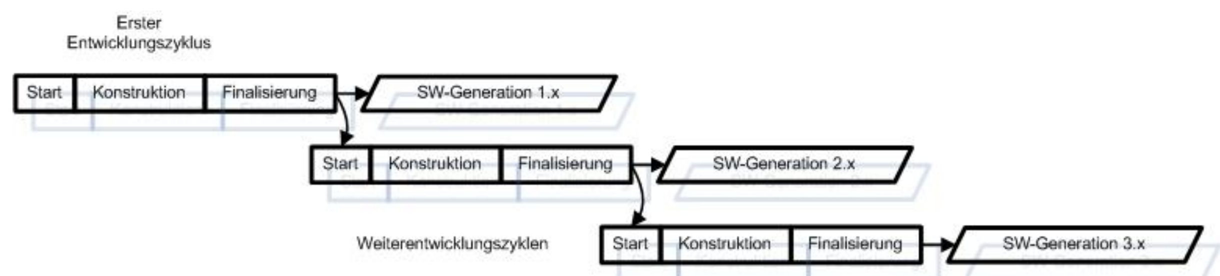


Abbildung 2-4: Mehrere Softwaregenerationen in Weiterentwicklungszyklen

Die Weiterentwicklungszyklen dürfen nach (Kruchten 2007) leicht überlappen, die Startphase darf während dem letzten Teil der Finalisierungsphase des vorangehenden Zyklus beginnen.

Auch das Prozessmodell aus (Bennett, Rajlich 2000) beschreibt die Entwicklung aufeinanderfolgender SW-Generationen[37]. Die Finalisierungsphase wird hier um die Phasen *Servicing*, *Phaseout* und *Closedown* erweitert: In der Servicing-Phase werden kleine Fehler behoben und einfache funktionale Änderungen vorgenommen; in der Phaseout-Phase wird das Servicing beendet und es wird so lange wie möglich Umsatz mit der Software generiert; in der Closedown-Phase wird das Produkt vom Markt genommen und dem Kunden wird ein Ersatzprodukt empfohlen (falls eins existiert). Mit der

[37] Im „*versioned staged model*" aus Bennett, Rajlich 2000 als „*evolution versions*" bezeichnet

Entwicklung der folgenden SW-Generation kann schon vor Beginn der Servicing-Phase begonnen werden.
Neue Softwaregenerationen werden z. B. bei der Microsoft Corporation nach (Cusumano, Selby 1995, S. 191) alle 12 oder 24 Monate entwickelt. Alle 24 Monate werden größere Verbesserungen und Architekturänderungen durchgeführt. Das gleiche Team, das das neue Produkt entwickelt, verbessert auch das alte. Die neue SW-Generation enthält auch Funktionalitäten, die aus Zeitgründen nicht mehr realisiert werden konnten. Auch in Lehman, Belady 1985, S. 36 entspricht diese SW-Generation der höchsten Abstraktionsebene. Mehrerer Produktgenerationen werden, dem Autor zufolge, nur durch langfristiges Feedback aus der Gesellschaft erreicht. In dieser Arbeit wird eine SW-Generation auch als Software oder Softwareprodukt bezeichnet.

Das Release

Die zweithöchste Ebene ist hier die Ebene des Release. Der Entwicklungszyklus eines Release beginnt mit der Analyse der Anforderungen, die auch aus dem Feedback vorangehender Releases stammen können. Dann werden ein oder mehrere aufeinanderfolgende Buildzyklen (auch interne Releases genannt) durchlaufen. Ein Releasezyklus endet mit der Einführung des letzten Builds in die Nutzung (externes Release). In Abbildung 2-3 ist dies der Fall, wenn die Fragen „Integrieren?“ und „Einführen?“ mit „Ja“ beantwortet wurden. Ein Releasezyklus im Verständnis dieser Arbeit entspricht dem der XP-Methodik (vgl. Beck 1999). In (Cockburn 2007, S. 212) wird dieser als Auslieferungszyklus und nach (Benediktsson et al. 2003) in der Literatur als Zyklus im Rahmen des *„incremental delivery“* bezeichnet.[38] In (Lehman, Belady 1985, S. 36) wird die zweithöchsten Abstraktionsebene als Releaseabfolge bezeichnet, d. h. dort entspricht ein Release einem Build im Verständnis dieser Arbeit.

Der Build

Die unterste betrachtete Ebene ist hier die Ebene des Builds. In einem Buildzyklus wird entwickelt und getestet. Ein Buildzyklus endet mit der Integration zum Teil noch unvollständiger Funktionalitäten einer Software, der Quellcode wird i. A. neukompiliert und es werden automatische *Regressionstests* ausgeführt.[39] Dies dient z. B. dazu Fehler durch Inkonsistenzen zu erkennen. Ein Buildzyklus nach Abbildung 2-3 endet, wenn die

---

[38] Benediktsson et al. 2003 selber verwenden hier die Bezeichnung „exogene“ Entwicklung.

[39] Vgl. zu diesem Absatz Cusumano, Selby 1997.

Frage „Integration?“ mit „Ja“ beantwortet wurde und die Fragen „Meeting?“ und „Einführen?“ mit „Nein“.

Des Weiteren dienen Builds der internen Kommunikation und Analyse (vgl. Benediktsson et al. 2003). Builds werden z. B. bei der Microsoft Corporation täglich durchgeführt (engl.: *daily builds*, vgl. Cusumano, Selby 1997). Nach (Benediktsson et al. 2003) wird ein Buildzyklus in der Literatur als Zyklus im Rahmen des *„incremental development“* betrachtet.[40]

Wie zu erkennen unterscheidet sich ein Releasezyklus durch den erhöhten Aufwand am Zyklusende von einem Buildzyklus, da am Ende eines Releases zusätzlich zu Maßnahmen der Qualitätssicherung auch Einführungsaktivitäten stattfinden. Daher wird i. d. R. nicht jeder Build zu einem Release.

On-Site-Customer

Zwischen Build- und Releasezyklus liegt die Möglichkeit, dass Arbeitsergebnisse einem Kunden, der vor Ort für Feedback zu Verfügung steht, präsentiert werden. Diese Einbindung des Kunden entspricht der *On-Site-Customer*-Methode nach (Beck 2000). Nach (Cusumano, Selby 1995) verfolgt die Microsoft Corporation mit den *„usability labs“* einen ähnlichen Ansatz zur Verbesserung der Bedienbarkeit. Abbildung 2-5 veranschaulicht die Einordnung der On-Site-Customer-Methode.[41]

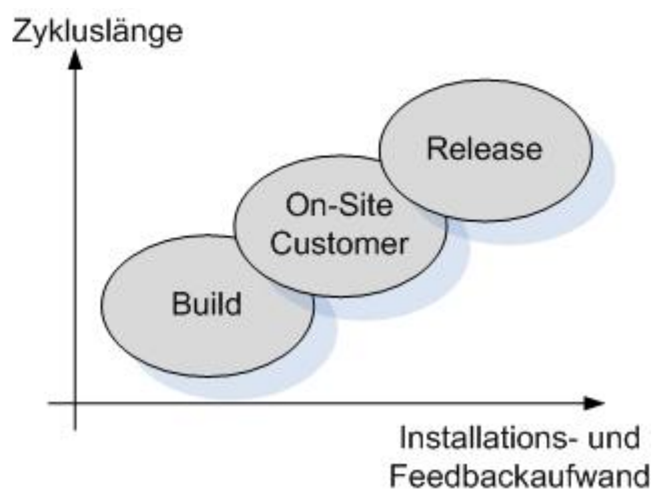


Abbildung 2-5: Einordnung des On-Site-Customers

[40] Benediktsson et al. 2003 selber verwenden hier die Bezeichnung „endogene“ Entwicklung.

Meetings

In Abbildung 2-3 wird außerdem der Fall berücksichtigt, dass innerhalb eines Buildzyklus weitere Informationen aus der Analyse in Meetings an die Implementierung weitergegeben werden können. Ein solches Meeting – welches wie in Abschnitt 3.1.3 beschrieben auch die Form eines Workshops haben kann – wird durchgeführt, wenn die Fragen „Integration?“ und „Meeting?“ mit „Ja“ beantwortet wurden und die Frage „Einführen?“ mit „Nein“.

2.4. Plangetriebene Entwicklungsprozesse

Das bekannteste Modell plangetriebener Entwicklungsprozesse ist das *Wasserfallmodell* bzw. *Stage-Gate-Modell*, das ursprünglich von (Royce 1970) entwickelt wurde. Hier wird der Entwicklungsprozess in strikte, durch Meilensteine getrennte, Phasen aufgeteilt. Das Wasserfallmodell wird als plangetriebene Methode bezeichnet (vgl. Boehm 2002), da in der ersten Phase unter hohem Aufwand ein Plan, d. h. eine detaillierte Spezifikation und Architektur, entwickelt wird. Dieser – zu anderen Prozessmodellen vergleichsweise hohe – Aufwand wird betrieben, damit der Plan später nicht mehr geändert werden muss, was hohen Überarbeitungsaufwand zur Folge haben kann. Im "vereinfachten" Wasserfallmodell sind die Phasen mit den Aktivitäten Anforderungsanalyse, Entwurf und Implementierung gleichzusetzen, die in dieser Reihenfolge durchlaufen werden. Dieses vereinfachte Modell hat Royce jedoch nie befürwortet, sondern immer zwei Durchläufe aller Phasen gefordert (vgl. Larman, Basili 2003). Das Wasserfallmodell wird in einer späteren Verfeinerung durch (Boehm 1976) beschrieben. Hier wird erlaubt, dass eine Aktivität mehrere Zyklen durchlaufen kann. Außerdem kann aus einer Phase wieder in eine vorangehende zurückgesprungen werden, falls z. B. durch Testaktivitäten Fehler entdeckt wurden. Abbildung 2-6 zeigt, durch die Anzahl der Mitarbeiter (MA) in den Aktivitäten, eine beispielhafte Gewichtung der verschiedenen Entwicklungsaktivitäten im Zeitverlauf. Hier wird eine strikte Trennung der Aktivitäten durch Meilensteine dargestellt, innerhalb der Phasen können Zyklen durchlaufen werden. Dieser Abbildung wird in Abschnitt 2.5.1 eine Abbildung eines inkrementellen Entwicklungsprozesses entgegengestellt.

---

[41] Einflussfaktoren des Installations- und Feedbackaufwands werden in Abschnitt 3.1.3 diskutiert.

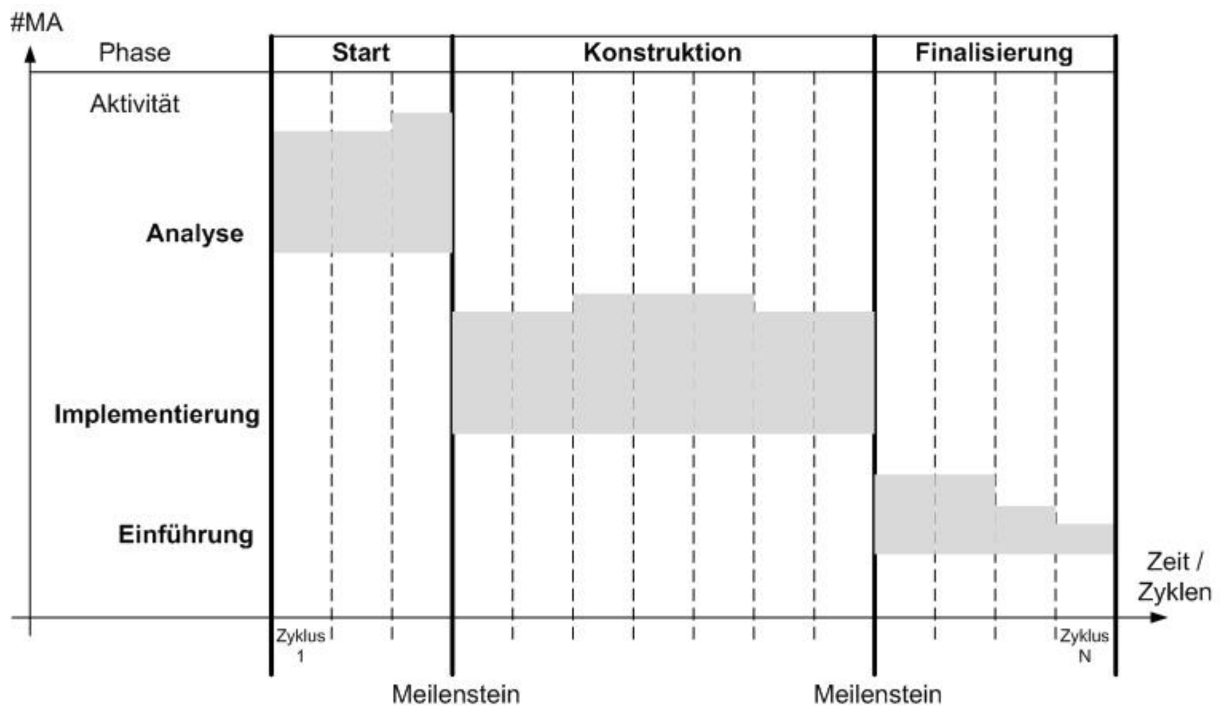


Abbildung 2-6: Beispielhafte Gewichtung der Entwicklungsaktivitäten in einem plangetriebenen Entwicklungsprozess[42]

Konkrete Entwicklungsprozesse, die dem Wasserfallmodell folgen, finden sich z. B. in (Cooper 2008) und (US National Aeronautics and Space Admin 2008). Das Wasserfallmodell wird oft als schwergewichtig (z. B. in Larman 2004) bezeichnet, da die Methodiken, die das Wasserfallmodell einsetzen meist zusätzlichen Aufwand durch eine detaillierte Dokumentation aller Entscheidungen und Arbeitsergebnisse fordern. Diese detaillierte Dokumentation hängt jedoch nicht direkt mit dem Prozess an sich zusammen, sondern kann auch in anderen Prozessmodellen gefordert werden.

Das vereinfachte Wasserfallmodell wurde in den 80ern durch das US-Verteidigungs-ministerium als Standard vorgeschrieben.[43] Doch dieses Modell führte zu einer Vielzahl von Misserfolgen, wie eine nachträgliche Untersuchung in (Jarzombek 1999) zeigt: 75% der Projekte scheiterten oder kamen nie zum Einsatz. So wurden 1987 iterative-inkrementelle Entwicklungsprozesse erlaubt und empfohlen. Die Einsicht entstand, dass die Softwareentwicklung Iterationen zwischen Designern und Benutzern benötigt, und darauf basierende Prozesse dem ursprünglichen Wasserfallmodell zu bevorzugen sind. Insbesondere folgt aus der Nicht-Analysierbarkeit von Anforderungen, wie sie in

[42] Eigene Abbildung in Anlehnung an Kruchten 2007, S. 93.

[43] Vergleiche zu diesem Absatz Larman, Basili 2003.

Abschnitt 3.2 beschrieben wird, eine wechselseitige Abhängigkeit[44] zwischen Analyse- und Implementierungsaktivitäten (vgl. dazu de Ven, Delbecq 1974, sowie Ha, Porteus 1995).

Nach (Boehm 2002, S. 66) gibt es aber auch Umgebungen in denen plangetriebene Prozesse von Vorteil sind. Dem Autor zufolge liegt der Schwerpunkt dieser Entwicklungsprozesse auf Projekten mit vollständigen, konsistenten, präzisen, testbaren und nachverfolgbaren Anforderungen. Solche Projekte finden sich z. B. in der Entwicklung stabiler, sicherheitskritischer hardwarenaher Software. Sind Anforderungsänderungen vorhersehbar, kann nach (Boehm 2002) auch ein hoher anfänglicher Architekturaufwand sinnvoll sein, wenn die Architektur für diese Änderungen ausgelegt ist. Allerdings entsteht dem Autor zufolge ein hohes Risiko, falls dieser Aufwand nicht den gewünschten Effekt hat: In einem Projekt an dem der Autor arbeitete wurde ein Großteil des Überarbeitungsaufwands durch „Architekturbrecher", wie Performance-, Verfügbarkeits-, oder Sicherheitsproblemen verursacht. Ein weiterer Vorteil, den der Autor plangetriebenen Prozessen zuordnet, ist deren Vorhersehbarkeit, Wiederholbarkeit und Optimierbarkeit.

Fließender Übergang zwischen den Prozessmodellen

Der Übergang zwischen plangetriebenen und inkrementellen Prozessen besteht im Grad der *Überlappung* zwischen Analyse- und Implementierungsaktivität. Überlappung zwischen beiden Aktivitäten bedeutet, dass die Implementierungsaktivität vor Abschluss der Analyseaktivität beginnt. Unter der Annahme, dass sich durch die Überlappung die Dauer beider Aktivitäten nicht ändert, hat dies eine Reduzierung der Entwicklungszeit zur Folge. Die Überlappung führt jedoch zu Überarbeitungsaufwand und es muss daher ein optimaler Trade-Off zwischen Überlappungsgrad und zusätzlichem Überarbeitungsaufwand gefunden werden (vgl. Loch, Terwiesch 1998). In den Modellen dieser Arbeit wird angenommen, dass eine positive Überlappung existiert. Diese Annahme ist auch Grundlage der in Abschnitt 2.5 vorgestellten inkrementellen Entwicklungsprozesse.

---

[44] Diese Art von Abhängigkeit findet sich in der Literatur auch unter den Stichwörtern „reziproke Abhängigkeit" oder „Interdependenz".

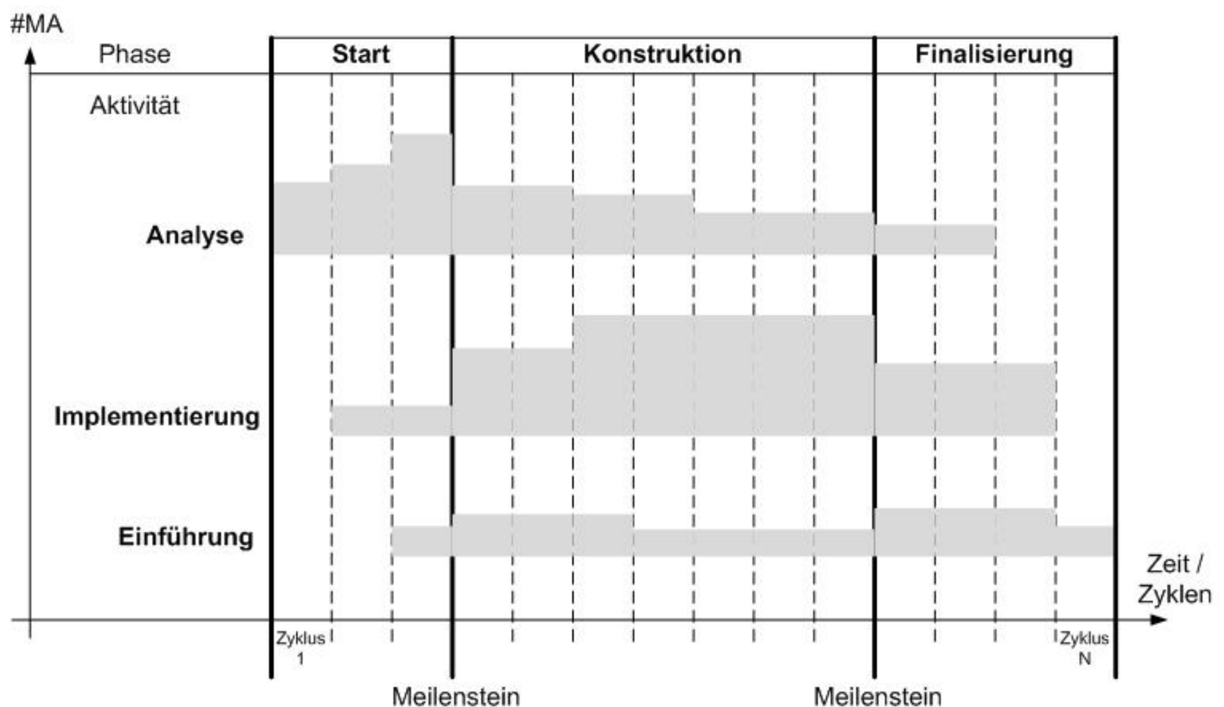
2.5. Inkrementelle Entwicklungsprozesse

2.5.1. Grundlagen

Die Anwendung von *inkrementellen Entwicklungsprozessen* fand schon in den 1950-ern statt. Methoden wie die Weiterentwicklung auf Grundlage von Nutzerfeedback wurden in der „IBM Federal Systems Division" entwickelt und bilden einen der Eckpunkte moderner agiler Entwicklungsmethodiken (vgl. Larman, Basili 2003, S. 47).

Inkrementelle Entwicklung bedeutet, dass der Umfang der Software auf Zyklen verteilt wird und die Software mit jedem Zyklus umfangreicher wird, bis im letzten Zyklus der volle Umfang erreicht wird. Ein *Inkrement* bezeichnet dabei eine in sich abgeschlossene lauffähige Einheit einer Software, mit allen zugehörigen Leistungen wie Anforderungs- und Entwurfsdokumentierung, Benutzerhandbüchern und Schulungen.[45]

In inkrementellen Entwicklungszyklen werden *jeweils* die Aktivitäten Analyse, Implementierung und Einführung durchgeführt. Abbildung 2-7 zeigt beispielhaft – in einer Art Gantt-Diagramm – mit welchem Aufwand die verschiedenen Aktivitäten über die Zyklen verteilt werden könnten.



[45] Benediktsson et al. 2003, S. 266 zitiert nach Graham 1989.

Abbildung 2-7: Beispielhafte Gewichtung der Entwicklungsaktivitäten in einem iterativen, inkrementellen Entwicklungsprozess[46]

Eine Betrachtung verschiedener Untersuchungen zu inkrementellen Prozessen lässt folgende Zusammenhänge vermuten: Der Wunsch ein frühes Kundenfeedback zu erhalten, setzt kurze Entwicklungszyklen und überlappende Phasen voraus (vgl. MacCormack et al. 2001; Baskerville et al. 2002). Dabei sollten erst die wichtigsten Features entwickelt werden (vgl. Beck 2000). Änderungen aufgrund des Kundenfeedbacks und die Forderung nach Skalierbarkeit aufgrund inkrementeller Entwicklungszyklen setzen eine flexible Architektur voraus (vgl. MacCormack et al. 2001; Baskerville et al. 2002). Eine große Generationenerfahrung hilft bei einer besseren Reaktion auf technologische Änderungen (vgl. MacCormack et al. 2001, Lang 2004). Häufiges Testen hilft dabei zum Großteil der Zeit ein lauffähiges System zu haben (vgl. Baskerville et al. 2002; Beck 2000).

Der inkrementelle Entwicklungsprozess auf Ebene der Releasezyklen wird auch als *exogener* iterativer inkrementeller Entwicklungsprozess beschrieben.[47] Ein exogener Einfluss entsteht dadurch, dass der Kunde in der inkrementellen Entwicklung durch die Nutzung des Systems Feedback bezüglich neuer oder geänderter Anforderungen gibt. Durch neue oder geänderte Anforderungen ergibt sich dann Überarbeitungsaufwand in der Implementierungsaktivität. Um eine gewisse „Stabilität“ zu erreichen wird in (Larman 2004, S. 14)[48] folgende Regel für einen Releasezyklus aufgestellt:

> *„Wenn die Anforderungen für einen Zyklus einmal feststehen und die Implementierung innerhalb der Zyklen begonnen hat, dann werden innerhalb dieser Zyklen keine Einflüsse durch externe Stakeholder mehr zugelassen.“*

Dem inkrementellen Prozessmodell können das Spiralmodell (vgl. Boehm 1986), die RUP-Methodik (vgl. Kruchten 2007, S. 90–91) und agile Methodiken wie Extreme Programming (vgl. Beck 1999) zugeordnet werden.

---

[46] Eigene Abbildung in Anlehnung an die Darstellung der RUP-Methodik aus Kruchten 2007, S. 93 und eine empirische Untersuchung aus Norden 1960 zitiert nach Putnam 1978.

[47] Vgl. Benediktsson et al. 2003, S. 267.

[48] In Larman 2004, S. 20 wird zwischen iterativer Entwicklung und inkrementeller Einführung unterschieden.

Diese Arbeit soll sich auf das inkrementelle Prozessmodell fokussieren und spezielle agile Methoden und Prinzipien – wie sie in (Beck et al. 2001) formuliert und in der Literatur scharf diskutiert werden (z. B. in Rakitin 2001) – nicht näher betrachten.

2.5.2. Best practice: Die Rational-Unified-Process-Methodik

Der Rational-Unified-Process ist eine Methodik, welche von der IBM Corporation mitentwickelt wurde und in der Praxis z. B. bei Unternehmen wie General Motors und Capgemini eingesetzt wird (vgl. IBM Corporation 2008). Ein weit verbreitetes Missverständnis ist es nach (Larman 2004, S. 194) die RUP-Methodik mit einem plangetriebenen Entwicklungsprozess – wie dem Wasserfallmodell – gleichzusetzen. RUP ist eine populäre Methodik mit einem iterativen, inkrementellen Entwicklungsprozess.[49] Walter Royce – der „Erfinder“ des Wasserfallmodells – trug als Angestellter der Rational Corporation auch zur Entwicklung der RUP-Methodik bei. Für diese Arbeit interessante Methoden, die im RUP enthalten sind und auf eine Beschreibung von Royce zurückgehen sind

- die besondere Beachtung hoher Risiken
- die Entwicklung einer Kernarchitektur in frühen Zyklen

Weiterhin interessant für diese Arbeit ist, dass die, in den Abschnitten 3.1.2 und 4.1 betrachtete, SW-Metrik nach Boehm 2000b einen nach RUP definierten Entwicklungsprozess zugrundelegt.

Phasen im RUP

Im RUP wird der Softwareentwicklungszyklus in die Phasen *Inception*, *Elaboration*, *Construction* und *Transition* unterteilt:[50]

- Die Inception-Phase ist die “gute Idee”, in der eine Vision des Endprodukts, sowie der Business Case und der Umfang des Produkts festgelegt werden. Die Inception-Phase endet mit dem Meilenstein „*lifecycle objective*“ (*LCO*).
- In der Elaboration-Phase werden die notwendigen Aktivitäten und geforderten Ressourcen geplant, die Anforderungen werden spezifiziert und die Architektur wird

[49] Vgl. zu diesem und den folgenden beiden Sätzen Larman, Basili 2003.

[50] Vgl. zu diesem Absatz Kruchten 2007, S. 90–91.

entworfen. Diese „Ausarbeitungsphase" endet mit dem Meilenstein „*lifecycle architecture*" (*LCA*).

- In der Construction-Phase wird das Softwareprodukt erstellt und es werden die *Vision*, die Architektur und die Pläne weiterentwickelt bis das Produkt – die vollständige Vision – fertig gestellt ist und dem Kunden ausgeliefert werden kann. Diese Konstruktionsphase endet mit dem Meilenstein „*initial operational capability*" (*IOC*).
- In der Transition-Phase wird das Produkt beim Kunden eingeführt, dies beinhaltet Produktion, Auslieferung, Training, Support und Instandhaltung des Produkts bis die Kunden zufrieden sind. Die Phase wird durch den Meilenstein „*product release*" (*PR*) abgeschlossen, der auch den Entwicklungszyklus abschließt.

Jede Phase besteht aus einem oder mehreren Zyklen und kann kleinere Meilensteine, sowie externe und interne Releases beinhalten.[51] Ein Release kann hier mit dem in Abschnitt 2.5 beschriebenen Inkrement gleichgesetzt werden. An den Meilensteinen zwischen den einzelnen Phasen und Zyklen nach RUP fallen möglicherweise Kosten an, z. B. Kommunikationsaufwand durch Meetings.
Die Einordnung der Phasen in das Begriffssystem dieser Arbeit wird in Tabelle 2-3 wiedergegeben. Dabei werden die RUP-Phasen Inception und Elaboration zusammengefasst und der Startphase zugeordnet.

Tabelle 2-3: Einordnung von Phasen im RUP in das Begriffssystem dieser Arbeit

Phasen im Begriffssystem dieser Arbeit	*Phasen im RUP (engl.)*
Startphase	*inception, elaboration*
Konstruktionsphase	*construction*
Finalisierungsphase	*transition*

Aktivitäten im RUP

In *jeder* dieser Phasen können im RUP die Prozessaktivitäten Planung, Anforderungsanalyse, Architekturentwurf, Feinentwurf, Implementierung, Integration, Test und Bewertung ausgeübt werden.[52] Die Einführungsaktivität wird in RUP nicht explizit benannt, aber unter den RUP Rollen findet sich ein „Deployment Manager", der

[51] Vgl. zu diesem Absatz Kruchten 2007.

[52] Vgl. zu diesem Absatz Kruchten 2007.

die Aufgaben in diesem Bereich übernimmt. Die Einordnung der Aktivitäten in das Begriffssystem dieser Arbeit wird in Tabelle 2-4 wiedergegeben.

Tabelle 2-4: Einordnung von Aktivitäten im RUP in das Begriffssystem dieser Arbeit

Aktivitäten im Begriffssystem dieser Arbeit	*Aktivitäten im RUP (engl.)*
Analyse	*planning, requirements*
Implementierung	*architecture, design, implementation, integration, test/assessment,*
Einführung	*Deployment*

Rollen im RUP

Nach (Kruchten 2007) werden im RUP die Rollen in die Kategorien Analyst, Entwickler, Tester, Manager und Produktion-/Support eingeteilt. Diesen Rollenkategorien werden 30 unterschiedliche Rollen zugeordnet. In Tabelle 2-5 werden die Rollen im RUP den verschiedenen Aktivitäten dieser Arbeit zugeordnet. Die Analysten-Rollen werden der Analyseaktivität, die Entwickler- und Tester-Rollen der Implementierungsaktivität und die Produktion-/Support-Rollen der Einführungs- und Nutzungsaktivität zugeordnet. Die verschiedenen Manager-Rollen werden auf die passenden Aktivitäten aufgeteilt, wobei hier nicht alle Manager-Rollen berücksichtigt wurden. Die genaue Beschreibung der Rollen findet sich in (Kruchten 2007).

Tabelle 2-5: Zuordnung von Rollen im RUP zu den Aktivitäten

Aktivitäten	*Rollen im RUP (engl.)*
Analyse	*process engineer, change control manager, business-process analyst, business designer, system analyst, requirements specifier, test analyst*
Implementierung	*software architect, designer, UI-designer, capsule designer, database designer, implementer, integrator, test manager, test designer, tester, system administrator*
Einführung	*deployment manager, technical writer, graphic artist, course developer*

2.5.3. Best practice: Die Extreme-Programming-Methodik

Die Extreme Programming Methodik nach (Beck 2000) ist die am häufigsten untersuchte und beschriebene agile Entwicklungsmethodik, obwohl auch andere agile Methodiken wie z. B. *Scrum* in der Industrie sehr populär sind.[53] Diese Methodik beinhaltet einen inkrementellen Prozess mit möglichst kurzen Zyklen.[54] Diese Zyklen werden in XP, wie auch in dieser Arbeit, als Releases bezeichnet. Der Kunde gibt im Idealfall nicht nur nach jedem Release Feedback, sondern es ist auch ein Endnutzer vor Ort und immer für Fragen bei unsicheren Anforderungen ansprechbar (vgl. On-Site-Customer-Methode in Abschnitt 2.3). Die Verteilung der Anforderungen auf die Releases ergibt sich daraus, dass die Anforderungen sortiert werden:[55] Zunächst werden die Anforderungen durch den Kunden[56] nach Geschäftswert sortiert, dann durch die Entwickler nach dem Risiko, mit dem sie die Entwicklungszeit abschätzen können. Der Umfang eines Release ergibt sich aus dem vom Kunden gewünschten Endzeitpunkt dieses Release oder umgekehrt.

---

[53] Vgl. Dyba, Dingsoyr 2008. Interessanterweise wurde bei Gesprächen des Autors dieser Arbeit mit Leitern der Softwareentwicklung verschiedener Unternehmen im Köln/Bonner Raum ausschließlich Scrum als eingesetzte agile Methodik genannt.

[54] Vgl. zu diesem und den folgenden beiden Sätzen Beck 2000.

[55] Vgl. zu diesem und den folgenden beiden Sätzen Beck 2000, S. 90.

[56] Die Kunden werden in der XP-Methodik auch unter den Begriff „Business“ gefasst.

In einem Extreme-Programming-Projekt werden neben dem Releasezyklus, wie in Abschnitt 2.3 beschrieben, verschiedene ineinander verschachtelten Zyklen betrachtet. (Beck 1999, S. 72) unterscheidet diese Zyklen in *Release*-, *Iteration*-, *Story*- und *Task*-Ebene. Die unterste Ebene bildet die Entwicklung von *Test-Cases*.

Phasen im XP

In (Beck 2000, S. 131–137) wird der Entwicklungsprozess einer SW-Generation in die Phasen *Exploration*, *Planning*, *Iterations-to-first-Release*, *Productionizing*, *Maintenance* und *Death* unterschieden. Die in (Beck 2000) als *Planning* bezeichnete Phase ist Teil des *Planning-Games* und nach Ansicht des Autors dieser Arbeit als Aktivität zu betrachten. Die anderen Phasen werden wie folgt beschrieben:[57]

- In der Exploration-Phase werden die Anforderungen für das erste Release festgelegt und die Entwickler experimentieren mit verschiedenen Technologien und Architekturen, um z. B. verschiedene Qualitätsmerkmale zu testen.
- In der Iteration-to-first-Release-Phase wird das erste Inkrement mit grundlegenden und einfachen Anforderungen entwickelt. Dies sollte dazu führen, dass das Grundgerüst der Architektur entwickelt wird.
- Die Maintenance-Phase ist der „normale“ Zustand eines XP-Projekts (vgl. Beck 2000, S. 135). Hier werden weitere Anforderungen umgesetzt und getestet, Releases erstellt, Defekte aus dem beim Kunden laufenden System behoben, und es wird *Refactoring* durchgeführt.
- Die Death-Phase startet, wenn der Kunde keine neuen Anforderungen mehr hat. In dieser Phase wird die Systemdokumentation erstellt. Eine andere Möglichkeit für den Eintritt in die Death-Phase ist, dass die Anforderungen nicht in angemessener Zeit erfüllt werden können oder die Fehlerrate zu hoch ist. In diesem Fall sollten die Ursachen des Scheiterns erforscht werden.

Die Zuordnung der Phasen in das Begriffssystem dieser Arbeit wird in Tabelle 2-6 wiedergegeben.

Tabelle 2-6: Zuordnung von Phasen in der XP-Methodik in das Begriffssystem dieser Arbeit

Phasen im Begriffssystem dieser Arbeit	*Phasen in der XP-Methodik (engl.)*

[57] Vgl. zu dieser Auflistung Beck 2000.

Startphase	*exploration*
Konstruktionsphase	*iterations to first release, maintenance*
Finalisierungsphase	*death*

Zwischen Aktivität und Phase ist die Verwendung des Begriffs *Productionizing* aus der XP Methodik einzuordnen. Productionizing bezeichnet nach (Beck 2000) eine Unterphase am Ende eines jeden Releases. In dieser Phase wird in verkürzten Zyklen entwickelt, die Testaktivitäten verstärkt und die Performance des Systems optimiert. In Zyklen dieser Phase werden weniger neue Anforderungen entwickelt als in vorangehenden Zyklen. Die zusätzlichen Aufgaben, die in dieser Phase durchgeführt werden, werden in dieser Arbeit als Teil der Einführungsaktivität aufgefasst.

Aktivitäten im XP

Analyse- und Planungsaktivitäten, zu Beginn eines jeden Releasezyklus, werden in der XP-Methodik im Planning-Game (auch *Release-Planning* genannt) sowie im *Iteration-Planning* durchgeführt.[58] Hier können auch neue Anforderungen durch den Kunden eingebracht werden. Diese Zusammenarbeit mit dem Kunden wird als *Steering* und *Listening* bezeichnet. Die weitere Entwicklung geschieht auf Basis des *Test-Driven-Developments*, d. h. es werden erst Tests geschrieben bevor mit der Implementierung begonnen wird. Das Schreiben der Tests beinhaltet implizit auch den Feinentwurf, da ein Entwurf der Objekte und sichtbaren Methoden nötig ist, um einen Test zu schreiben. Die Implementierung zielt dann darauf ab, die Tests fehlerfrei zum Laufen zu bekommen. Dann werden in einem Strukturverbesserungsschritt (dem *Refactoring*) architektonische Maßnahmen getroffen. Hier sollen Redundanzen entfernt und die Komplexität der Architektur verringert werden. Neu entwickelter Code wird kontinuierlich in das Gesamtsystem integriert (*Continuous-Integration*). Ist eine Anforderung umgesetzt, findet ein Akzeptanztest[59] statt, der zusammen mit dem Kunden definiert wurde.

Die Einordnung der Aktivitäten in das Begriffssystem dieser Arbeit wird in Tabelle 2-7 wiedergegeben. Die Einführungsaktivität besteht – wie in der Phasenbeschreibung erläutert – aus Aktivitäten die verstärkt im Productionizing stattfinden.

[58] Vgl. zu diesem und folgenden Absatz Beck 2000, S. 86–91.

[59] Dieser Akzeptanztest wird in Beck 2000 sowohl als „*acceptance test*“ als auch als „*functional test*“ bezeichnet.

Tabelle 2-7: Einordnung von Aktivitäten in der XP-Methodik in das Begriffssystem dieser Arbeit

Aktivitäten im Begriffssystem dieser Arbeit	*Aktivitäten in der XP-Methodik (engl.)*
Analyse	*planning game, iteration planning*
Implementierung	*refactoring, write tests, code, test, functional test*
Einführung	*productionizing*[60]

Rollen in der XP-Methodik

In der XP-Methodik gibt es nach (Beck 2000) keine detaillierten Rollen wie in der RUP-Methodik. Der Autor beschreibt folgende Rollen:

- *„Customer“*: Personen in dieser Rolle sind die Kunden und definieren die Anforderungen (Stories). Einige Kunden sollten auch am Entwicklungsort sein (On-Site-Customer).
- *„Programmer“*: Mitarbeiter in dieser Rolle sind für alle Implementierungsaufgaben zuständig. Sie sind aber auch für Aufgaben aus der Analyseaktivität zuständig. Sie halten Rücksprache mit den Kunden, wenn sie Anforderungen nicht verstehen und entwickeln die Architektur.
- *„Tester“*: Mitarbeiter in dieser Rolle erarbeiten zum einen, zusammen mit dem Kunden, die funktionalen Tests und sind zum anderen für die Durchführung der Tests verantwortlich. Daher wird diese Rolle unterschiedlichen Aktivitäten zugeordnet.
- *„Tracker“*: Mitarbeiter in dieser Rolle sammeln die Informationen über das Projekt und werten diese zur weiteren Steuerung aus.
- *„Coach“*: Mitarbeiter in dieser Rolle sind für die Steuerung des Entwicklungsprozesses verantwortlich.
- *„Consultant“*: Mitarbeiter in dieser Rolle sind Spezialisten in einer bestimmten Technologie und beraten die Programmierer.

---

[60] Wie beschrieben bezeichnet „*productionizing*“ in XP eine Phase, hier sind die speziellen Aktivitäten dieser Phase gemeint.

In Tabelle 2-8 ist zu erkennen, dass einer Rolle mehrere Aktivitäten zugeordnet werden können. Daraus kann gefolgert werden, dass die Rollendefinition in XP noch verfeinert werden müsste.

Tabelle 2-8: Zuordnung von Rollen in der XP-Methodik zu den Aktivitäten

Aktivitäten	*Rollen in der XP-Methodik (engl.)*
Analyse	*customer, tester, tracker, coach, programmer*
Implementierung	*programmer, tester, consultant*
Einführung	*tester, programmer, customer*

2.5.4. Best practice: Die Synch-and-Stabilize-Methodik

Die Synch-and-Stabilize-Methodik der Microsoft Corporation – wie sie in (Cusumano, Selby 1995) ausführlich beschrieben wird – beinhaltet einen iterativen, inkrementellen Entwicklungsprozess.[61] Die Methodik wird als Synch-and-Stabilize bezeichnet, da die Arbeit der Entwickler durch tägliche Builds synchronisiert und das Projekt durch einen zeitlicher Puffer für jede Iteration stabilisiert wird. Ziel der Entwicklung ist ein fixer Auslieferungszeitpunkt. Dabei wird der Entwicklungsprozess in die Phasen *Planning*, *Development* und *Stabilization* unterteilt, die in dieser Reihenfolge an den klar definierten Meilensteine „*functional specification and final schedule*", „*code complete release*" und „*product release to manufacturing*" enden. Tabelle 2-9 zeigt die Einordnung der Phasen in das Begriffssystem dieser Arbeit.

Tabelle 2-9: Einordnung der Phasen in Microsofts Synch-and-Stabilize in das Begriffssystem dieser Arbeit

Phasen im Begriffssystem dieser Arbeit	*Phasen in Synch-and-Stabilize (engl.)*
Startphase	*planning*
Konstruktionsphase	*development*
Finalisierungsphase	*stabilization*

[61] Vgl. zu diesem Absatz Cusumano, Selby 1995, S. 192–195.

In der Planungsphase wird mit extensiven Kundeneinsatz die Spezifikation erstellt, sowie architektonische Themen und Abhängigkeiten zwischen Komponenten geklärt.[62] D.h. in dieser Phase findet ein Großteil der Analysetätigkeit statt.

In der Entwicklungsphase werden nach (Cusumano, Selby 1995) die Anforderungen in drei bis vier Iterationen implementiert, integriert und getestet. Zusätzlich werden funktionale Details ausgearbeitet und Anforderungsänderungen berücksichtigt. Die Microsoft Corporation unterhält während der Entwicklung *„usability labs“* in denen Nutzer die täglich bereitgestellten neuen Funktionalitäten der Softwareprodukte testen. Die Änderungsrate kann bei 30% oder mehr liegen.

Der Entwicklungsprozess in dieser Phase verläuft spiralförmig, d. h. risikogetrieben. In der ersten Iteration werden zunächst die kritischsten Funktionalitäten, in den folgenden Iterationen weniger kritische und in der letzten Iteration werden die am wenigsten kritischen Funktionalitäten entwickelt. Die Beurteilung dieses Risikos basiert auf Nutzerfeedback zu den Funktionalitäten, sowie den Abhängigkeiten zwischen den Funktionalitäten auf Implementierungsebene. Jede Iteration entspricht einem internen Produkt-Release, der durch einen Meilenstein innerhalb dieser Phase gekennzeichnet ist.

In der Stabilisierungsphase werden die internen Testaktivitäten erhöht. Zusätzlich wird das Softwareprodukt in *Beta-Versionen* durch Endnutzer, bestimmte Hardwarehersteller (sogenannte *„original equipment manufacturer“* (OEM))[63] und unabhängige Softwarehersteller extern getestet.

Die Aktivitäten im Entwicklungsprozess von Microsoft-Office-Produkten werden unterteilt in *„program management“, „product planning“*, *„development“, „test“* und *„user education“* (vgl. Cusumano, Selby 1995, S. 49).[64] Die Einordnung dieser Aktivitäten in das Begriffssystem dieser Arbeit, sowie weiterer Aktivitäten aus Cusumano, Selby 1995, S. 196–197 sind in Tabelle 2-10 aufgeführt.

---

[62] Vgl. zu diesem Absatz und den folgenden drei Absätzen Cusumano, Selby 1997.

[63] Die Bindung von Microsofts Softwareprodukten an bestimmte Hardware wurde jedoch vom Bundesgerichtshof als nicht rechtskräftig zurückgewiesen (vgl. Bundesgerichtshof (BGH), Urteil vom 06.07.2000. JurPC Web-Dok.). Diese und weitere Entscheidungen betreffen auch den Streit um den Handel mit „gebrauchter“ Software.

[64] Vgl. zu diesem Absatz und der nachfolgenden Tabelle Cusumano, Selby 1995, S. 197.

Tabelle 2-10: Einordnung von Aktivitäten in Microsofts Synch-and-Stabilize in das Begriffssystem dieser Arbeit

Aktivitäten im Begriffssystem dieser Arbeit	*Aktivitäten in Synch-and-Stabilize (engl.)*
Analyse	*program management, product planning, usability lab tests*
Implementierung	*prototyping, UI-design, write and optimize code, testing, debugging*
Einführung	*user education, product support*

Die den Aktivitäten zugehörigen Rollen sind in der folgenden Tabelle dargestellt.[65] Auf einen Entwickler kommt laut (Cusumano, Selby 1995, S. 194) genau ein Tester. Die Mitarbeiter der Einführungsaktivität werden in (Cusumano, Selby 1995) nicht benannt und hier beispielhaft als Support-Engineer bezeichnet.

Tabelle 2-11: Zuordnung von Rollen in Microsofts Synch-and-Stabilize zu den Aktivitäten

Aktivitäten	*Rollen in Synch-and-Stabilize (engl.)*
Analyse	*product manager, program manager, usability lab tester*
Implementierung	*UI-designer, developer, tester*
Einführung	*support engineer*

Die Prozessplanung unterscheidet sich zwischen der Planung der Entwicklung von Anwendungen (wie der Tabellenkalkulation *Excel*) und der Entwicklung von Systemen (wie dem Betriebssystem *Windows*).[66] Die Anwendungsentwicklung hat kürzere Zeitpläne und genauere Lieferzeitpunkte als die Systementwicklung. Die Anwendungsentwicklung ist flexibler, da die Funktionen der Anwendungen klarer gekapselt sind und weniger Interdependenzen besitzen als die Funktionen der Systeme (vgl. Cusumano, Selby 1995, S. 189). Allerdings steigen die Abhängigkeiten zwischen Anwendungen mit der Anzahl gemeinsam verwendeter Komponenten, wie z. B. bei den Microsoft-Office-Produkten (vgl. Cusumano, Selby 1995, S. 190). Die Anforderungen von Systemen werden im Vergleich zu Anforderungen von Anwendungen früher im

[65] Vgl. Cusumano, Selby 1995, S. 196–197.

[66] Vgl. zu diesem Absatz Cusumano, Selby 1995.

Entwicklungsprozess detailliert und vollständig ausgearbeitet. Der Entwicklungsprozess von Systemen hat aufgrund erhöhter Stabilitäts- und Kompatibilitätsanforderungen an die Systeme längere Testperioden als der von Anwendungen.

2.6. Steuerungsmöglichkeiten der Prozessplanung

Die Zielsetzung dieser Arbeit beinhaltet die Untersuchung des Einflusses verschiedener Steuerungsmöglichkeiten der Prozessplanung auf den Entwicklungsaufwand und die Entwicklungszeit. Dieser Einfluss wird in verschiedenen Modellen, wie z. B. in (Ha, Porteus 1995; Krishnan, Eppinger 1997; Qi Feng et al. 2008; Loch, Terwiesch 1998) untersucht. Aus diesen Modellen sollen Ideen übernommen werden und in diese Arbeit einfließen.

Zur Betrachtung von Steuerungsmöglichkeiten der Prozessplanung eignen sich aktivitätsbasierte Modelle, welche die Zusammenhänge zwischen den Entwicklungsaktivitäten untersuchen. Solche Modelle finden sich in der Literatur zum Projektmanagement der Produktentwicklung. Darunter fällt auch der, für diese Arbeit besonders interessante, Forschungsbereich zur Entwicklung neuer bzw. innovativer Produkte (*New-Product-Development*) und zur nebenläufigen Entwicklung (*Concurrent-Engineering*)[67]. In (Browning, Ramasesh 2007) wird diese Literatur nach aktivitätsbasierten Modellen, die die Planung des Produktentwicklungsprozesses unterstützen, durchsucht.[68]

Planung der Prozessstruktur

Ein Ziel verschiedener aktivitätsbasierter Modelle betrifft nach (Browning, Ramasesh 2007) die Strukturierung des Entwicklungsprozesses. Nach (Browning, Ramasesh 2007) sind

- *Anzahl*
- *Größe/Umfang*
- *Zeitpunkt*

---

[67] Die Literatur zum Concurrent-Engineering bezieht sich auf die Nebenläufigkeit bzw. Überlappung verschiedener Aktivitäten mit dem Ziel der Reduzierung der Entwicklungs- oder Reaktionszeit.

[68] Diese Untersuchung beinhaltet 18 große Zeitschriften, wie „IEEE Transactions on Engineering Management“, „Journal of Operations Management“ und „Management Science“, im Zeitraum von 1994 bis 2005 und präsentiert die dort gefundenen Artikel nach Modellierungszielen geordnet.

von Reviews und Zyklen wichtige Steuerungsmöglichkeiten, um die „beste“ Makrostruktur festzulegen. Diese werden in verschiedenen Modellen wie in (Ha, Porteus 1995) betrachtet und optimiert. In dieser Arbeit wird die Planung der Struktur zyklischer Prozesse auf Ebene der Kommunikation zwischen Analyse- und Implementierungsaktivität, sowie der Kommunikation zwischen Analyse und Nutzung mittels Feedback betrachtet. Die Struktur, die den geringsten Entwicklungsaufwand zur Folge hat, wird hier als „beste“ Struktur angesehen.

Eine weitere Möglichkeit die Prozessstruktur zu beeinflussen, besteht in der Wahl des *Überlappungsgrads* zwischen den Aktivitäten (vgl. Abschnitt 2.4, sowie Browning, Ramasesh 2007). In dieser Arbeit wird, bei Betrachtung der Überlappung, auf einem Makrolevel zwischen plangetriebenen und inkrementellen Entwicklungsprozessen unterschieden. In der Literatur zum Concurrent-Engineering wird die Überlappung einer informationsliefernden Aktivität und einer informationsverarbeitenden Aktivität betrachtet (vgl. Krishnan, Eppinger 1997). In (Krishnan, Eppinger 1997) werden optimale Überlappungsstrategien basierend auf dem Verlauf der Unsicherheit der Informationen und der Verarbeitungsdauer neuer Informationen ermittelt (vgl. Abschnitt 3.1.1). Nach (Loch, Terwiesch 1998) hat der Überlappungsgrad direkten Einfluss auf die Unsicherheit.

Basierend u. a. auf (Ha, Porteus 1995) und (Krishnan, Eppinger 1997) werden in (Loch, Terwiesch 1998) zusätzlich Effekte durch Kommunikationskosten und Kommunikationsverzögerungen betrachtet und ein detailliertes Modell zur Berechnung der Entwicklungsdauer aufgestellt. Aus der Optimierung der Entwicklungsdauer ergibt sich in (Loch, Terwiesch 1998) der optimale Überlappungsgrad und die optimale *Kommunikationsrate*. Die Hypothesen aus diesem Modell werden in (Terwiesch, Loch 1999) empirisch untersucht und teilweise bestätigt. In Kapitel 5.3 werden die Modelle aus (Loch, Terwiesch 1998), (Krishnan, Eppinger 1997) und (Ha, Porteus 1995) ausführlich beschrieben.

In (Serich 2005) wird das Modell aus (Loch, Terwiesch 1998) auf die SW-Entwicklung übertragen. Dort wird der optimale *Übergangszeitpunkt zwischen Prototyping- und Analyseaktivität* so bestimmt, dass die Entwicklungsdauer minimal ist. Auch in (Thomke 1998) wird ein Modell aufgestellt, um den optimalen Übergangszeitpunkt zwischen Analyseaktivitäten (dort Simulation und Prototyping) zu bestimmen.

In (Qi Feng et al. 2008) wird ein Modell aufgestellt mit dem die optimalen *Zeitpunkte für eine koordinierte modulübergreifende Fehlerbehebung* im Softwareentwicklungsprozess berechnet werden können. Dieses Modell basiert auf (Loch, Terwiesch 1998) und dient der Planung und Steuerung von Entwicklungszyklen auf Implementierungs- und Testebene. Daraus ist zu erkennen, dass die Ergebnisse des Modells aus (Loch, Terwiesch 1998) auf verschiedene Ebenen des Entwicklungsprozesses übertragbar sind.

Releaseplanung

Ein weiterer Aspekt der Prozessplanung ist die Verteilung der Anforderungen auf die Entwicklungszyklen (*Releaseplanung*).

Boehm formalisiert mit seinem populären Spiralmodell das Konzept der Priorisierung von Entwicklungszyklen nach Risiken und fordert eine Risikoeinschätzung in jedem Zyklus (vgl. Boehm 1986; Larman, Basili 2003). Das Spiralmodell besagt, dass, in den ersten Zyklen, die Anforderungen mit der höchsten Unsicherheit umgesetzt werden sollen und in den letzten Zyklen die sichersten Anforderungen. Die Priorisierung der Anforderungen kann auch durch den Kunden geschehen, wie es Beck 1999 fordert. Auch in (Larman 2004) wird zwischen risikogetriebener („*risk-driven*“) und kundengetriebener („*client-driven*“) Priorisierung der funktionalen Anforderungen unterschieden. Eine kundengetriebene Priorisierung kann z. B. mit QFD-Methoden nach (Akao 1972) erreicht werden – wie in Abschnitt 2.1 kurz beschrieben.

In (Cohn 2006, S. 35) wird zur Planung des Aufwands des nächsten Zyklus, z. B. eines Releases, der geschätzte Aufwand für Design, Implementierung und Test („*ideal engineering days*“)[69] verwendet. Nach jedem Zyklus werden die Schätzungen für vervollständigte Anforderungen zusammengerechnet, woraus sich der produktive Aufwand eines Zyklus („*project velocity*“) ergibt.[70] Unter der Annahme, dass der nächste Zyklus genauso lange dauern soll, wählt der Kunde die unvollständigen Anforderungen mit der höchsten Priorität aus, für die in der Summe der gleiche Aufwand veranschlagt wird.

---

[69] D.h. ohne zusätzlichen Kommunikationsaufwand (vgl. Load-Factor in Abschnitt 3.1.2)

[70] Vergleichbar mit der Gesamtproduktivität aus Abschnitt 3.1.2.

Die hier kurz vorgestellten und in dieser Arbeit untersuchten Steuerungsmöglichkeiten der Prozessplanung sind Anzahl, Umfang und Zeitpunkt von (Release-)Zyklen, der Überlappungsgrad zwischen den Aktivitäten, sowie die Releaseplanung.

Kapitel 3: Einflussfaktoren des Entwicklungsaufwands

Nachdem in Kapitel 2 die Struktur von Entwicklungsprozessen in unsicheren Umgebungen analysiert wurde, ist das Ziel dieses Kapitels Einflussfaktoren des *Entwicklungsaufwands*[71] dieser Prozesse zu identifizieren und zu analysieren. Auf Basis dieser Faktoren und der in Abschnitt 2.6 beschriebenen Steuerungsmöglichkeiten sollen Aussagen über eine möglichst optimale Planung des Entwicklungsprozesses getroffen werden.

Ein Ziel dieser Arbeit ist es, die *Wirkung* der Einflussfaktoren auf den Entwicklungsaufwand zu untersuchen (Kausalprinzip). Es wird keine statistisch nachgewiesene signifikante Korrelation zwischen den Einflussfaktoren und dem Entwicklungsaufwand vorausgesetzt, auch wenn entsprechende Untersuchungen an verschiedenen Stellen zitiert werden.

In diesem Kapitel werden zunächst Einflussfaktoren durch Betrachtung von Modellen der Produktentwicklung und Organisationstheorie und von Metriken zur Kostenschätzung in der SW-Entwicklung sowie durch die Analyse weiterer Prozessaktivitäten identifiziert. Danach werden, der Zielsetzung entsprechend, die Einflussfaktoren *Anforderungsunsicherheit* und *Sensitivität/Flexibilität in komplexen Systemen* näher betrachtet, da diese den Überarbeitungsaufwand – wie er in Kapitel 4 modelliert wird – beeinflussen.

Auf Basis des, schon eingangs der Arbeit erwähnten, Prinzips „Was nicht gemessen werden kann, kann nicht kontrolliert bzw. gesteuert werden“ (vgl. DeMarco 1982, S. 6) werden in der SW-Entwicklung existierende Metriken der Einflussfaktoren betrachtet.

Eine nähere Analyse des Zusammenhangs zwischen Entwicklungsaufwand und *Entwicklungsdauer* wird in Abschnitt 4.3 durchgeführt. Es wird hier zunächst eine konstante Teamgröße und Gesamtproduktivität angenommen, wodurch sich die Entwicklungsdauer mit dem Aufwand erhöht.

---

[71] Gemessen z. B. in Personenmonaten.

3.1. Identifizierung der Einflussfaktoren

3.1.1. Faktoren aus der Produktentwicklung und Organisationstheorie

Die Literatur zur Produktentwicklung und Organisationstheorie liefert verschiedene allgemeine produktunabhängige Untersuchungen zu Einflussfaktoren des Entwicklungsaufwands.

In der Produktentwicklung lässt sich die Anpassungsgüte zwischen Produkt und Prozessparametern („*product / process fit*“, vgl. Adler 1995) mit der Differenz zwischen Anforderungen und Softwareprodukt vergleichen. Diese Differenz verursacht – neben einer möglicherweise verringerten Produktqualität – Überarbeitungsaufwand, der in großen Softwareentwicklungsprojekten über 50% der Gesamtkosten verursachen kann.[72]

Bei Anforderungsunsicherheit ist diese Anpassungsgüte unsicher und es werden nach (Adler 1995) verschiedenen Koordinationsmechanismen zwischen den Entwicklungsaktivitäten – in der SWE zwischen Analyse und Implementierung – gebraucht, um diese zu reduzieren. Diese Unsicherheit der Anpassungsgüte kann nach (Adler 1995)[73], (Loch, Terwiesch 1998) und (Krishnan, Eppinger 1997) in zwei Dimensionen unterteilt werden:

- Neuheitsgrad bzw. Innovationsgrad („*fit novelty*“ (vgl. Adler 1995) oder „*evolution*“ (vgl. Loch, Terwiesch 1998; Krishnan, Eppinger 1997))
- Erhöhter Aufwand bei der Umsetzung (hier Analyse *und* Implementierung) neuer Produktparameter bzw. Anforderungen („*fit analyzability*“ (vgl. Adler 1995) oder „*sensitivity*“ (vgl. Loch, Terwiesch 1998; Krishnan, Eppinger 1997))

Die erste Dimension wird in dieser Arbeit unter dem Aspekt der Unsicherheit aus nicht-vorhersehbaren Anforderungen betrachtet. Das Wachstum der Anforderungsmenge und damit auch des Umfangs unsicherer Anforderungen im Verlauf eines Projekts wird als Evolution bezeichnet. Eine unsichere Anpassungsgüte entsteht in der Softwareentwicklung nach (Curtis et al. 1988, S. 1271) dadurch, dass “tiefes anwendungsspezifisches Wissen, welches notwendig ist, um die meisten großen und komplexen Systeme zu entwickeln, in vielen Softwareentwicklungsteams schwach

[72] Vgl. Jones 1986 zitiert nach Boehm, Papaccio 1988, S. 1466.

[73] Basierend auf der Organisationstheorie, angefangen mit der vielzitierten Theorie aus Perrow 1967.

verteilt [ist]“. Eine genauere Analyse der Anforderungsunsicherheit folgt in Abschnitt 3.2.

Die zweite Dimension wird in dieser Arbeit, den Untersuchungen aus (Loch, Terwiesch 1998) und (Krishnan, Eppinger 1997) folgend, als *Sensitivität* bezeichnet. Wie in Abschnitt 3.3 näher erläutert wird entsteht aus der Komplexität des bestehenden Systems und aus Abhängigkeiten zwischen Anforderungen ein erhöhter Aufwand in der Analyse und Implementierung neuer Anforderungen.

Einflussfaktoren aktivitätsbasierter Modelle finden sich in der Literaturreview aus (Browning, Ramasesh 2007). Dort wird die Literatur zum Management der Produktentwicklung nach solchen Modellen durchsucht und kategorisiert. Eine Kategorie von Modellen hat den Autoren zufolge das Ziel – ähnlich den Modellen dieser Arbeit – verschiedene Schlüsselvariablen zu schätzen, zu optimieren und zu verbessern. Diese Schlüsselvariablen unterteilen sich nach (Browning, Ramasesh 2007) in die Kategorien

- Ressourcenanforderungen und -einschränkungen
- Qualität der Entwicklungsergebnisse
- Unsicherheit, Risiken und Chancen
- Robustheit, Flexibilität und Anpassungsfähigkeit

Den Autoren zufolge können diese Variablen wiederum Einflussfaktoren der Schlüsselvariablen Entwicklungszeit[74] sein. In (Browning, Ramasesh 2007) werden einige Modelle identifiziert, die solche Zusammenhänge berücksichtigen, wie die Untersuchung aus (Luh et al. 1999) zum Einfluss der Unsicherheit auf die Entwicklungszeit.

In der vorliegenden Arbeit wird die Kategorie „Ressourcenanforderungen und -einschränkungen“ und die zugehörigen Modelle (z. B. Adler et al. 1995) nicht berücksichtigt, da solche Modelle projektübergreifende Zusammenhänge betrachten, während hier nur einzelne Projekte betrachtet werden sollen.

Die Kategorie „Qualität der Entwicklungsergebnisse“ wird in dieser Arbeit differenziert betrachtet. Alle nicht-funktionalen Anforderungen an ein Softwareprodukt sind nach (Bass et al. 2005) gewünschte Ausprägungen eines Qualitätsmerkmals. Zu den

[74] Die, wie eingangs des Kapitels beschrieben, aus dem Entwicklungsaufwand resultiert.

Qualitätsmerkmalen zählen nach (Bass et al. 2005): Verfügbarkeit, Änderbarkeit, Performance, Sicherheit, Testbarkeit und Bedienbarkeit. Die Softwarequalität wird den Autoren zufolge durch die Softwarearchitektur bestimmt. Da die Entwicklung der Softwarearchitektur der Implementierungsaktivität im Entwicklungsprozess zugeordnet wird, beeinflussen die nicht-funktionalen Anforderungen diese Aktivität. Das Qualitätsmerkmal Änderbarkeit bzw. Anpassungsfähigkeit des entwickelten Softwareprodukts wird in dieser Arbeit getrennt betrachtet und auch in (Browning, Ramasesh 2007) einer gesonderten Kategorie zugeordnet.

Die Kategorie „Robustheit, Flexibilität und Anpassungsfähigkeit" kann mit der Sensitivität nach (Loch, Terwiesch 1998) und (Krishnan, Eppinger 1997)verglichen werden. Hier ist der Begriff Flexibilität ein Antonym des Begriffs Sensitivität – ein unflexibles System wird als sensitiv betrachtet.

Die Kategorie „Unsicherheit, Risiken und Chancen" wird in dieser Arbeit unter dem Aspekt der oben beschriebenen Anforderungsunsicherheit betrachtet.

3.1.2. Faktoren aus Schätzmethoden des Implementierungsaufwands

Um weitere, für die Entwicklungszeit relevante, Einflussfaktoren in der Softwareentwicklung zu identifizieren, wird die Forschung zur Schätzung des Implementierungsaufwands betrachtet. In (Jörgensen, Shepperd 2007) und (Boehm et al. 2000) werden verschiedene Methoden der Aufwands- und Kostenschätzung dargestellt und kategorisiert.

Ein Großteil dieser Methoden basiert auf der *Function-Point* (*FP*) -Metrik aus (Albrecht, Gafeney Jr. 1983). Die FP-Metrik dient dazu den *Umfang* der funktionalen Anforderungen zu schätzen. Dazu werden diese in verschiedene Transaktionen zerlegt wird.[75] Der Umfang einer Transaktion besteht aus dem Umfang der Eingabe-, Verarbeitungs- und Ausgabeprozesse von Daten. Der Verarbeitungsprozess von Daten kann aus dem Lesen, Verwalten oder Erstellen von Daten bestehen. Die Daten selber werden in verschiedenen FP-Metriken in Datentypen und Gruppen kategorisiert, um den funktionalen Umfang genauer abzuschätzen. Auf Basis der FPs und des Aufwands

[75] Vgl. bis zum Ende dieses Absatzes Onur Demirors, Cigdem Gencel 2009. Die Autoren stellen ein Modell auf, welches die verschiedenen FP-Metriken vereinen soll.

vergangener Projekte, liefern die verschiedenen Methoden Modelle, um den Aufwand neuer Projekte zu schätzen.

Auf der FP-Metrik basieren neue Metriken, wie die der *International Function-Points User Group* (*IFPUG)* (vgl. Brown et al. 2010), der *Netherlands Software Metrics Users Association* (*NESMA*) (vgl. NESMA 2009) und des *Common Software Measurement International Consortium* (*COSMIC*) (vgl. Abran et al. 2003). Diese neuen Metriken haben das Ziel die Anwendbarkeit der ursprünglichen FP-Metrik zu verbessern. Eine viel zitierte auf der FP- Metrik basierende Methode zur Kostenschätzung ist das „*Constructive Cost Model“* (*Cocomo)* nach Boehm 2000b. Im „*Cocomo II.2000 Post-Architecture Model*“ werden neben der FP-Metrik 22 weitere Einflussfaktoren berücksichtigt.

In der Untersuchung aus (Albrecht, Gafeney Jr. 1983) wird ein signifikanter Zusammenhang zwischen der FP-Metrik und dem Aufwand der Implementierungsaktivität nachgewiesen. Auch in der Untersuchung aus (Liu, Mintram 2005) auf Basis von 362 Projekten[76] stellen die Autoren eine signifikante Korrelation zwischen Aufwand und Umfang fest, allerdings keine signifikante Korrelation zwischen Umfang und Dauer. Der Umfang hat von allen untersuchten Variablen den größten Einfluss[77] auf den Aufwand. Auch nach (Boehm, Papaccio 1988, S. 1465) ist der Umfang der „einflussreichste“ Faktor für den Gesamtaufwand.

Eine zur FP-Metrik ähnliche Methode wird im Rahmen der Extreme-Programming-Methodik angewandt. Zunächst werden der Umfang der Anforderungen bzw. Stories (*Story-Points*), der implementierte Umfang pro Zeiteinheit (*Velocity*), sowie der Aufwand der, z. B. durch Meetings, zusätzlich zur Entwicklung dazu kommt (*Load-Factor*) geschätzt. Die *Produktivität* eines Entwicklers ergibt sich aus der Velocity abzüglich des Load-Factors. Die Gesamtproduktivität wird durch die Summierung der Produktivität aller, an der Story beteiligten, Entwickler berechnet. Das Produkt aus Gesamtproduktivität und Umfang ergibt die gesamte Entwicklungsdauer einer Story.

Diese Trennung der Einflussfaktoren in produktbezogene Faktoren (wie Umfang) und organisatorische Faktoren (wie Produktivität) soll in dieser Arbeit beibehalten werden,

---

[76] Aus einer bereinigten Teilmenge der Datenbank R9 der International Software Benchmarking Standards ISBSG Group (ISBSG).

[77] Diese Bewertung wird auf Basis des – durch eine Regressionsanalyse angepassten – Quadrats des Korrelationskoeffizienten *R* gemacht. Auf den Umfang folgen – in absteigender Reihenfolge – die

um Wirkungszusammenhänge zwischen diesen Faktoren zu modellieren und zu diskutieren. Die Berechnung des Entwicklungsaufwand aus dem Produkt von Umfang und Produktivität wird in Abschnitt 4.1 diskutiert, soll hier aber zunächst als Grundlage gelten.

Auf Basis dieser Unterteilung können die Produkt- und Plattformfaktoren der Cocomo-II-Metrik aus (Boehm 2000b, S. 41–46) dem Produktumfang zugeordnet werden und die Personal- und Projektfaktoren aus (Boehm 2000b, S. 47–51) der Produktivität.[78] Bei dieser Zuordnung fließen in den Produktumfang auch nicht-funktionale Anforderungen ein.

Die, auf Grundlage dieser Betrachtung, weder dem Umfang noch der Produktivität zuzuordnenden Skalenfaktoren aus Boehm 2000b, S. 30–36 werden im Zusammenhang mit der Sensitivität und Komplexität in Abschnitt 3.2, sowie – auf die Modellierung bezogen – in Abschnitt 4.4 gesondert betrachtet. Bei der getrennten Betrachtung von Umfang und Komplexität ist zu berücksichtigen, dass in bestimmten FP-Metriken, wie z. B. der Metrik der IFPUG, die Einschätzung der Komplexität in die Bestimmung des Umfangs mit einfließt (vgl. Jones 1998, S. 291).

3.1.3. Faktoren weiterer Prozessaktivitäten

Für die Optimierung und Planung des Entwicklungsprozesses sind nach (Loch, Terwiesch 1998) und (Ha, Porteus 1995) die *Fixkosten pro Zyklus*[79] ein wichtiger Einflussfaktor. Diese werden, je nach betrachteter Zyklenebene[80], durch verschiedene Faktoren beeinflusst. So kann auf Ebene der Releasezyklen die Einführungsaktivität einen Großteil der Fixkosten verursachen und auf Ebene der Meetingzyklen können dies die Kommunikationskosten sein.

Die Methoden zur Aufwandsschätzung aus Abschnitt 3.1.2 beschäftigen sich i. d. R. mit Einflussfaktoren des Aufwands der Implementierungsaktivität. Der in diesem Abschnitt betrachtete Aufwand der Analyse- und Einführungsaktivität wird z. B. im Cocomo-II-

---

Entwicklungsdauer, der Typ der Programmiersprache („3GL“, „4GL“, „APG“) und die Architektur (Client-Server, Multi-Tier, Stand-Alone)

[78] Weitere Faktoren aus Boehm 2000b, welche die Anforderungsunsicherheit betreffen werden in Abschnitt 3.3 untersucht.

[79] In Loch, Terwiesch 1998 als Zeitdauer modelliert.

[80] Die verschiedenen Ebenen von Entwicklungszyklen werden in Abschnitt 2.3 beschrieben.

Modell nicht geschätzt, da dieser je nach Projekt zu großen Variationen unterliegt.[81] Des Weiteren wird in diesem Abschnitt der Kommunikationsaufwand zwischen den Aktivitäten betrachtet, da angenommen wird, dass dieser Aufwand durch eine zyklische Entwicklung erhöht wird.

Analyseaufwand

Einen großen Einfluss auf den Aufwand der Analyseaktivität haben nach (Boehm 2000b, S. 309) die Komplexität aller zugehöriger Aufgaben, d. h. die Komplexität der Analyse der Anwendungsdomäne, der Unsicherheit, der Anforderungen, möglicher Architekturen, des zu verwendenden Prozessmodells, etc., und Änderungen in Rollen und Verantwortlichkeiten der Stakeholder. Der Aufwand der Teilaufgabe Anforderungsanalyse kann nach (Loch, Terwiesch 1998) Einfluss auf die Unsicherheit der Anforderungen haben. Der Anteil des Analyseaufwands in der ersten Prozessphase am gesamten Analyseaufwand unterscheidet plangetriebene SWE-Prozesse maßgeblich von inkrementellen Prozessen.

Einführungsaufwand

Nach (Boehm 2000b, S. 309) hat insbesondere die Anzahl unterschiedlicher Installationen einen großen Einfluss auf den Aufwand der Einführungsaktivität. Die Anzahl der Installationen des Produkts ist unabhängig von dem Produktumfang und bestimmt daher u. a. die Fixkosten eines Zyklus.

In (Coupaye, Estublier 2000) wird der gesamte Einführungsprozess einer Software in eine bestehende IT-Landschaft untersucht. Die Autoren unterscheiden zwischen der Perspektive des Softwareherstellers, des beauftragenden Unternehmens und der Benutzer. Sie definieren den Einführungsprozess als die Menge aller Aufgaben, die, vom Ende der Entwicklung selber bis zur tatsächlichen Installation und Wartung der Anwendung auf dem Computer des Benutzers, durchzuführen sind. Als Einflussfaktoren für den Aufwand des Einführungsprozesses in großen Projekten, nennen die Autoren die Anzahl der Komponenten sowie Versionen und Varianten der einzuführenden Anwendungen. Die Schnittstellen und die Verknüpfung mit dem bestehenden System resultieren nach (Coupaye, Estublier 2000) und (van der Hoek, Wolf 2003) in Aufwand während der

[81] Vgl. Boehm 2000b, S. 308. In Boehm 2000b wird, unter der Annahme eines sequenziellen Entwicklungsprozesses, die Analyseaktivität mit der Startphase, die Implementierungsaktivität mit der Konstruktionsphase und die Einführungsaktivität mit der Finalisierungsphase gleichgesetzt.

Einführung. Hier ist nach (Coupaye, Estublier 2000), seitens des benutzenden Unternehmen, besonders wichtig, wann und in welcher Reihenfolge die Software eingeführt wird und dass Risiken, wie die Unterbrechung des Geschäftsablaufs, vermieden werden.

Die Untersuchung aus (Coupaye, Estublier 2000) zeigt inwieweit sich der Einführungsprozess automatisieren lässt und welche Software-Tools dazu bereits existieren. Die Autoren kommen zu dem Schluss, dass die Erstellung und Einführung von benutzerspezifischen Varianten auf Unternehmensseite, eine der wichtigsten noch nicht automatisierten Aufgaben ist. Auch in (van der Hoek, Wolf 2003) wird die Automatisierung auf Nutzerseite als wichtiger Faktor angesehen. Hier stellt sich dem Autor dieser Arbeit allerdings die Frage, welche dieser zu automatisierenden Aufgabe bei Intranet-Anwendungen durchzuführen sind. Bei dieser Art von Anwendungen scheinen die Einführungskosten bedeutend niedriger zu sein.

Auf Basis dieser und weiterer Untersuchungen lassen sich verschiedene Aktivitäten und Aufwände der Einführungsaktivität hinzurechnen, wie z. B.

- Review- und Testaufwand (vgl. Basili et al. 1996)
- Dokumentationsaufwand (vgl. van der Hoek, Wolf 2003)
- Kompilierungsaufwand (vgl. Beck 2000)
- Installationsaufwand (vgl. Coupaye, Estublier 2000; Boehm 2000b; van der Hoek, Wolf 2003) – evtl. erhöht durch Inkompatibilitäten mit vorigen Versionen
- Schulungsaufwand (vgl. Cusumano, Selby 1995)
- Marketingaufwand (vgl. Cusumano, Selby 1995, S. 49)
- Supportaufwand (vgl. Cusumano, Selby 1995)

Dieser Einführungsaufwand scheint ein wichtiger Einflussfaktor der, in den Modellen dieser Arbeit betrachteten, Fixkosten auf Ebene der Releasezyklen zu sein.

Kommunikationsaufwand

Der Kommunikationsaufwand zwischen Analyse- und Implementierungsaktivität ist ein weiterer hier zu betrachtender Faktor. Die Kommunikation zwischen beiden Aktivitäten kann informell oder z. B. in *Joint-Application-Design* (*JAD*) -Workshops stattfinden (vgl. Davidson 1999). Es ist anzunehmen, dass das Domänenwissen der Entwickler – wie in

(Curtis et al. 1988) beschrieben – den Kommunikationsaufwand zwischen Analyse- und Implementierungsaktivität beeinflusst.

JAD-Workshops werden nach (Larman 2004, S. 284) in der zyklischen Softwareentwicklung eingesetzt. Das Ziel der Workshops ist es aus groben Kundenanforderungen eine Spezifikation, wie z. B. detaillierte Use-Cases, zu erstellen, mit der in der Implementierungsaktivität weiter gearbeitet werden kann. In diesen Workshops ist, neben Analysetätigkeiten, nach (Larman 2004, S. 286) auch Feedback aus der Implementierungsaktivität notwendig. In (Davidson 1999) wird die Verwendung von JAD-Workshops in der Praxis beschrieben. In diesen regelmäßigen JAD-Workshops wird die Spezifikation im Idealfall durch Kunden, Nutzer, Analysten und Entwickler gemeinsam (weiter-)entwickelt. Die Autoren berichten, dass die traditionelle Idee von intensiven JAD-Workshops (3 - 5 Tage) in 40% der untersuchten Projekte, welche die JAD-Methodik anwenden, verwendet wird. Sie berichten auch, dass in diesen Workshops meist mit analytischen Modellen (und weniger mit einer lauffähigen Software) gearbeitet wird. Als positives Ergebnis der Workshops wird in 30% der Projekte eine bessere Qualität der Anforderungen bezeichnet, was auf eine vorausgehende Anforderungsunsicherheit schließen lässt.

Der Kommunikationsaufwand, wie er durch JAD-Workshops entsteht, wird als wichtiger Einflussfaktor der Fixkosten des, in dieser Arbeit betrachteten, Modells zur Planung von Meetings angesehen.

3.2. Analyse der Anforderungsunsicherheit

Die Analyse der Anforderungsunsicherheit in der SW-Entwicklung soll dabei helfen, den Überarbeitungsaufwand abzuschätzen und verschiedene Handlungsalternativen zum Umgang mit Unsicherheit zu verstehen. Der Überarbeitungsaufwand hängt, wie im Folgenden erläutert, von der Art und möglicherweise dem Zeitpunkt des Auftretens von Änderungen in den Kundenanforderungen ab. Je größer die Unsicherheit ist desto größer ist i. d. R. auch der Überarbeitungsaufwand.[82]

3.2.1. Ausprägungen der Unsicherheit

Ursachen für Anforderungsunsicherheit

Die Ursachen für Anforderungsunsicherheit verdeutlicht Abbildung 3-1 nach (Curtis et al. 1988, S. 1275).

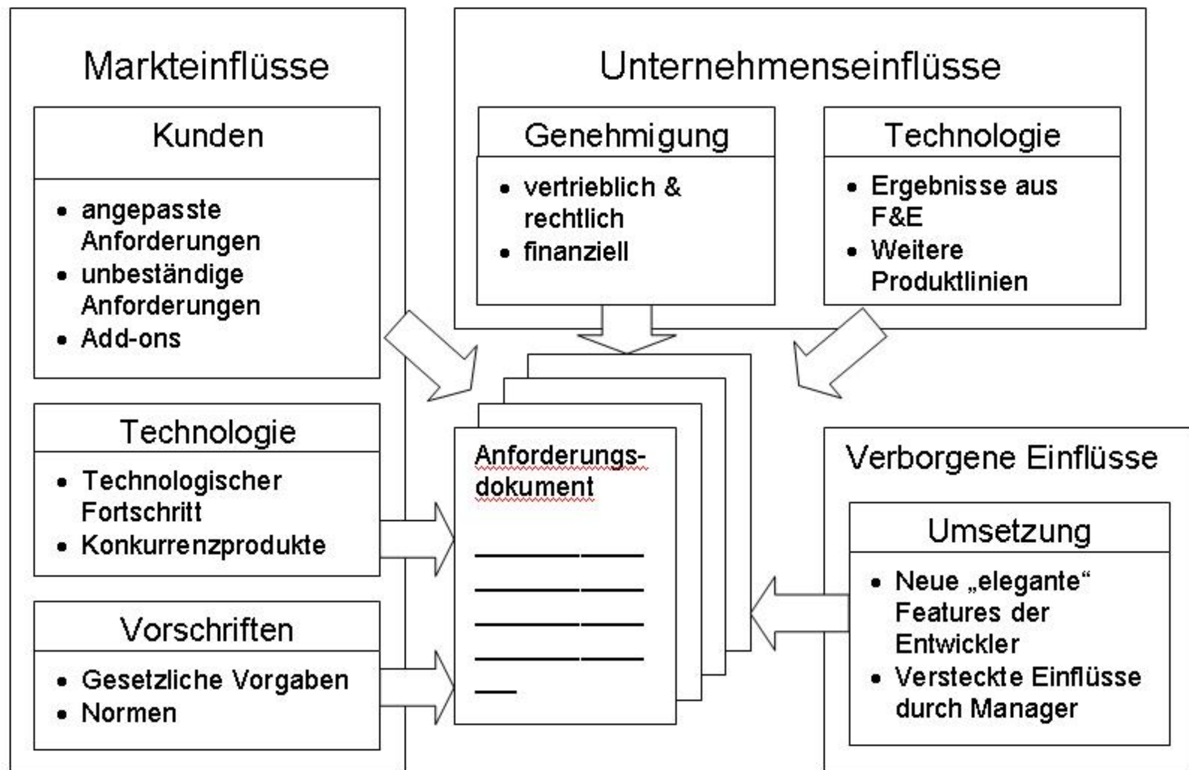


Abbildung 3-1: Ursachen für Anforderungsunsicherheit

Den Autoren zufolge werden die Ursachen für konfliktierende und fluktuierende Anforderungen in Markteinflüsse, Unternehmenseinflüsse und versteckte Einflüsse unterteilt. Produktanforderungen änderten sich in der Feldstudie nach (Curtis et al. 1988, S. 1276) am häufigsten, wenn verschiedene Nutzer verschiedene Bedürfnisse hatten oder wenn die Bedürfnisse eines Nutzers sich über die Zeit änderten. Bei der Entwicklung von Standardsoftware kam es den Autoren zufolge zu Konflikten, da sich die Anforderungen aus den Abteilungen Strategische Planung, Marketing und Produktplanung unterschieden. Der Einfluss dieser durch Nutzer verursachten Unsicherheiten auf die Prozessplanung wird in dieser Arbeit untersucht. Nach (Curtis et al. 1988) wird in evolutionären Prozessmodellen, wie dem Spiralmodell aus (Boehm 1986), ein vielversprechender Versuch gesehen, diese Konflikte auf einer Makroebene zu kontrollieren.

[82] Es sei denn Unsicherheit bedeutet, dass Anforderungen wegfallen können. Ist dies der Fall wurde aber ähnlich dem zusätzlichen Überarbeitungsaufwand anfangs zu viel Implementierungsaufwand betrieben.

Die Gründe für die in dieser Arbeit speziell betrachteten nicht-vorhersehbaren Anforderungen können unterschiedlich sein:[83]

- die Umgebung, in der die Software eingesetzt werden soll, hat sich verändert
- die Analysemethoden sind nicht geeignet
- die Nutzer stellen erst bei Verwendung der Software fest, was ihnen noch fehlt („I'll know it when I see it“ (*IKIWISI*), vgl. (Boehm 2000a))
- die Nutzer machen Verbesserungsvorschläge (vgl. u. a. Koch 2008a, S. 31)
- die Nutzer verstehen erst mit fortgeschrittener Implementierung die Details der Anforderungen (vgl. u. a. Walz et al. 1993, S. 64)
- es sind zunächst nur die Bedürfnisse des ersten Kunden bekannt und im Laufe der Zeit kommen andere Kunden und mit ihnen neue Anforderungen dazu (vgl. Curtis et al. 1988)

Durch dieses Nutzerfeedback kann zusätzlicher Nutzen und Innovation entstehen, wie Erfahrungen, die der Autor dieser Arbeit während der Entwicklung webbasierter Dienste gemacht hat, zeigen. Während der Entwicklung können, aufgrund von Nutzerfeedback, nicht vorhergesehene nützliche Verknüpfungen verschiedener Module erkannt werden.

Nach (Lehman, Belady 1985, S. 342) entsteht durch das Wachstum des Systems Unordnung und Komplexität, welche sich in falsch realisierten Spezifikationen, unentdeckten Fehlern, inadäquater und fehlerhafter Dokumentation und neuen internen Schnittstellen äußert. Auch diese Risiken sind im Detail nicht vorhersehbar.

Änderungen in der Anforderungsmenge

In der Literatur zur Softwareentwicklung wird das Auftreten von Änderungen in der Anforderungsmenge[84] teilweise unter einen Sammelbegriff gefasst („*requirements creep*“ (vgl. Jones 1998, S. 429) oder „*breakage*“ (vgl. Benediktsson et al. 2003)). Andere Autoren (z. B. Wang, Lai 2001) unterscheiden zwischen neuen, geänderten und entfernten Anforderungen. Der Umfang an neuen und geänderten Anforderungen wird in

[83] Vgl. zu diesem Absatz Parnas, Clements 1986a – zu diesem Artikel existiert eine Korrektur: Parnas, Clements 1986b.

[84] Unter Anforderungsmenge werden nicht die *Anzahl* der Anforderungen, sondern die gesammelten Anforderungen verstanden.

(Boehm 2000b, S. 28) zusammen mit einem Überarbeitungsfaktor[85] zur Berechnung des Überarbeitungsumfangs („*maintenance size*“) verwendet. Durch eine Art Unsicherheitsfaktor („*requirements evolution and volatility*“) wird in (Boehm 2000b, S. 25) der Prozentsatz des ausgelieferten Codes abgeschätzt, der durch Änderungen in den Anforderungen wahrscheinlich überflüssig wird.[86]

Des Weiteren können nicht-funktionale Anforderungen hinzukommen oder es kann das geforderte Qualitätsniveau gesenkt werden. Auch wenn ein bestimmtes Qualitätsmerkmal[87] nicht explizit in den Anforderungen genannt wird, besitzt die Software eine Ausprägung dieses Qualitätsmerkmals.[88] Ist das geforderte Qualitätsniveau allerdings so gering, dass diese nicht-funktionale Anforderung ohne spezifische architektonische Gestaltungsmaßnahmen erfüllt wird, so wird dies hier als das Nicht-Vorhandensein der nicht-funktionalen Anforderung gewertet.

Unsichere Anforderungen

Besteht die Möglichkeit, dass eine Eigenschaft einer Anforderung geändert wird, so gilt die Anforderung als unsicher. Die Art der Eigenschaften von Anforderungen und die Bestimmung der Unsicherheit können sehr unterschiedlich sein. Eigenschaften funktionaler Systemanforderungen können wie in Abschnitt 2.1 beschrieben u. a. Vor- und Nachbedingungen der Anforderung, aber auch die zugehörigen Systemfunktionen und Schnittstellen sein. Die Unsicherheit einer Schnittstelle wird z. B. nach (Jootar, Eppinger 2002) aus der Summe der Änderungswahrscheinlichkeiten der Funktionen, die mit dieser Schnittstelle zusammenhängen, berechnet.

Die Eigenschaften einer nicht-funktionalen Anforderung sind abhängig von der Art des Qualitätsmerkmals. Z. B. kann die Verfügbarkeit einer Software in Prozent gemessen werden. Hier kann die Unsicherheit z. B. daraus resultieren, dass die geforderte Verfügbarkeit nur in einem Intervall von „85% oder mehr“ angegeben wurde.

---

[85] Die Interpretation des „*maintenance adjustment factor*” (vgl. Boehm 2000b, S. 28) kann mit dem – in späteren Kapiteln vorgestellten – Konzept der Sensitivität verglichen werden. Nicht zu verwechseln ist dieser Faktor mit dem „*Reuse Model*“ aus Boehm 2000b, S. 22.

[86] Beträgt der Faktor „*requirements evolution and volatility*“ z. B. 20%, dann werden 120 KLOC produziert, aber nur 100 KLOC haben einen Nutzen und werden ausgeliefert.

[87] Qualitätsmerkmale werden auf Ebene der Anforderungen auch als nicht-funktionale Anforderungen bezeichnet (vgl. Abschnitt 2.1).

[88] Vgl. zu diesem Absatz Bass et al. 2005.

In Beispiel 3-1 werden mögliche unsichere Eigenschaften einer Systemanforderung dargestellt.

Beispiel 3-1: Eine unsichere Systemanforderung

Die Systemanforderung

Der Kunde kann bei einem Buch das Inhaltsverzeichnis ansehen

aus Beispiel 2-1 beinhaltet eine Systemfunktion zur

Erzeugung einer in einem Webbrowser lesbaren Darstellung des Inhaltsverzeichnisses.

Diese Systemfunktion kann eine Ausgabe im *Portable Document Format* (*PDF*) oder in der *Hypertext Markup Language* (*HTML*) erzeugen. Wird nur eine Ausgabemöglichkeit implementiert, so besteht die Unsicherheit z. B. darin, dass mit 50%-iger Wahrscheinlichkeit die andere Möglichkeit gefordert wird.

Die Unsicherheit kann auch darin bestehen, dass die Möglichkeit besteht, dass zusätzlich zum Inhaltsverzeichnis zwei zufällige Buchseiten angezeigt werden sollen. Tritt dieser Fall ein, muss die Systemfunktion angepasst werden.

Das Ausgabeformat der Systemfunktion aus Beispiel 3-1 kann auch den Qualitätsmerkmalen zugerechnet werden, da es die Bedienbarkeit beeinflusst. Daraus ist ersichtlich, dass ein starker Zusammenhang zwischen Anforderungen, Systemfunktionen und Qualitätsmerkmalen besteht und diese nicht unabhängig voneinander realisiert werden können.

Definition der Anforderungsunsicherheit

Auf Basis der obigen Betrachtungen wird im Folgenden die für diese Arbeit gültige Definition der Anforderungsunsicherheit gegeben, welche zwischen unsicheren Anforderungen und einer unsicheren Anforderungsmenge unterscheidet.

Definition 3-1: Anforderungsunsicherheit

> Anforderungsunsicherheit bedeutet hier, dass die Menge der Anforderungen selber oder deren Elemente unsicher sind. Eine Anforderung ist unsicher, wenn die Möglichkeit besteht, dass eine ihrer Eigenschaften im Laufe des Entwicklungsprozesses geändert wird. Die Anforderungsmenge ist unsicher, wenn neue Anforderungen hinzu kommen oder alte entfernt werden können.

Bekannte Unsicherheit

Die Entscheidungstheorie (vgl. Laux 2005, S. 23) differenziert bei Entscheidungen unter Unsicherheit zwischen

- Entscheidungen unter Risiko: Die Wahrscheinlichkeit für möglicherweise eintretende Umweltsituationen ist bekannt.
- Entscheidungen unter Ungewissheit: Man kennt zwar die möglicherweise eintretenden Umweltsituationen, allerdings nicht deren Eintrittswahrscheinlichkeiten. Diese Situation wird auch als Knight'sche Unsicherheit (vgl. Knight 1921) oder als Ambiguität[89] bezeichnet (vgl. Camerer, Weber 1992).

Dies bedeutet, dass die unsichere Variable und evtl. deren Wahrscheinlichkeitsverteilung – zumindest subjektiv (vgl. Savage 1972) – bekannt ist. Die unsichere Variable kann übertragen auf die Softwareentwicklung mit der Anforderungsmenge gleichgesetzt werden. Variabel sind das Vorhandensein bestimmter Anforderungen und die Ausprägungen der Anforderungseigenschaften. Um eine Wahrscheinlichkeit für eine bestimmte Ausprägung der Anforderungsmenge anzugeben, kann z. B., bei Unsicherheit aus differierenden Meinungen von Stakeholdern, der Einfluss der verschiedenen Stakeholder abgeschätzt werden.

Unbekannte Unsicherheit

Unbekannte Unsicherheit bedeutet, dass die (unsicheren) Anforderungen selber unbekannt sind oder bestimmte Eigenschaften einer Anforderung unbekannt sind oder die Möglichkeit, dass eine Anforderung entfernt wird, nicht in Betracht gezogen wird. Dieses

[89] Zu Missverständnissen kann die unterschiedliche Auffassung der Ambiguität führen. Z. B. wird sie in Pich et al. 2002 mit dem Konzept der unbekannten Variablen gleichgesetzt.

Konzept der unbekannten Unsicherheit findet sich in der Literatur auch unter dem Begriff „*unforeseeable uncertainty*“ (vgl. Loch et al. 2008) bzw. „*unknown unknowns*“ (vgl. Pich et al. 2002, S. 1013). In (Curtis et al. 1988) entspricht dies den fluktuierenden Anforderungen. In dieser Arbeit wird das Konzept der unbekannten Unsicherheit auch als Unsicherheit aus *noch unbekannten* oder *nicht vorhersehbaren* Anforderungen umschrieben. In Beispiel 2-1 kann z. B. die Systemanforderung „der Kunde kann bei einem Buch das Inhaltsverzeichnis ansehen“ zu Beginn der Implementierung noch unbekannt gewesen sein.

3.2.2. Maßnahmen bei Unsicherheit

In (Jones 1998, S. 429) wird berichtet, dass die durchschnittliche Änderungsrate in den USA bei 2% pro Monat liegt und durch verschiedene *Maßnahmen*, wie Prototyping, auf 0.5% pro Monat gesenkt werden kann.

In dieser Arbeit wird zwischen Maßnahmen bei unsicheren bekannten Anforderungen und Maßnahmen bei unbekannter Unsicherheit unterschieden. Auf bekannte Unsicherheiten kann – wie in diesem Abschnitt beschrieben wird – durch Maßnahmen der Implementierungsaktivität eingegangen werden, während zur Auflösung unbekannter Unsicherheiten Maßnahmen der Prozessplanung betrachtet werden.

Maßnahmen bei unsicheren bekannten Anforderungen

Wie wirkt sich die Kenntnis der Unsicherheit von Anforderungen auf die Maßnahmen in der Implementierungsaktivität aus? Ist eine konkrete Entscheidung, seitens der Stakeholder, bezüglich der genauen Anforderungen zunächst nicht möglich, so kann, je nach aktueller Situation, eine der folgenden Maßnahmen ausgewählt werden (vgl. Haferkamp et al. 2008[90]):

- Verfeinerung architektonischer Strukturen
- Flexibilisierung der Implementierungsaktivität
- Weiterentwicklung anderer, von der Entscheidung unabhängigen, Aktivitäten und warten auf die Entscheidung
- Implementierung aller möglichen Ausprägungen einer Variablen

[90] Die Autoren übertragen Handlungsempfehlungen aus der Produktentwicklung i. A. aus Terwiesch et al. 2002 auf die Softwareentwicklung.

Der Zusammenhang zwischen architektonischen Maßnahmen und dem durch Unsicherheiten verursachten Überarbeitungsaufwand wird z. B. in (Bahsoon, Emmerich W. 2003) untersucht.

Bei der unsicheren nicht-funktionalen Anforderung „Verfügbarkeit von 85% oder mehr" könnten z. B. folgende mögliche Maßnahmen existieren:

1. Bereitstellung *eines* Server mit 90%-iger Verfügbarkeit und Verfeinerung der architektonischen Strukturen, um unsichere Bereiche zu kapseln.
2. Bereitstellung eines *zweiten* Servers, so dass zusammen 99% Verfügbarkeit[91] erreicht werden können. Dies kommt der oben beschriebenen Implementierung aller möglichen Ausprägungen einer Variablen gleich.

Aus diesen Maßnahmen ergibt sich der Wertebereich der Handlungsvariablen. Dieser ist hier, im Gegensatz zum Wertebereich der Anforderungsvariablen „Verfügbarkeit", nicht stetig sondern binär. Hinzu kommen die unterschiedlichen Arten von Aufwänden bei Realisierung einer der Maßnahmen. Im ersten Fall besteht das Risiko, dass später ein zweiter Server gebraucht wird und es zu erhöhtem Überarbeitungsaufwand durch nachträgliche Integration von Mechanismen zur Synchronisierung beider Server kommt (vgl. Bass et al. 2005, S. 103). Im zweiten Fall werden diese Mechanismen von Anfang an implementiert und es wird später zu keinem Überarbeitungsaufwand kommen. Allerdings besteht das Risiko, dass dieser Aufwand „überflüssig" war.

Maßnahmen bei unbekannter Unsicherheit

Sind die Anforderungen unbekannt, so scheint es nicht sinnvoll spekulative architektonische Gestaltungsmaßnahmen zu ergreifen (vgl. Beck 2000). Die Implementierungsaktivität kann in dem Fall keine spezifischen, die unbekannten Anforderungen betreffenden, Gestaltungsmaßnahmen durchführen.

Um die Unsicherheit durch noch unbekannte Anforderungen aufzulösen, können Maßnahmen in der Prozessplanung ergriffen werden. Die vorgeschlagenen Maßnahmen nach (Mathiassen et al. 2007) zur Reduktion von Volatilitätsrisiken[92] sind die

---

[91] Diese 99% ergeben sich – bei stochastischer Unabhängigkeit – aus 100% minus der Ausfallwahrscheinlichkeit beider Server gleichzeitig (10% mal 10%).

[92] Die unbekannte Anforderungsidentität aus Mathiassen et al. 2007 scheint dem Begriff nach den nicht-vorhersehbaren Anforderungen am nächsten zu kommen. Die Maßnahmen, die zur Auflösung der dadurch entstehenden Risiken dienen, konzentrieren sich nach Mathiassen et al. 2007 allerdings weniger auf

„Priorisierung der Anforderungen“ und das „Experimentieren mit Anforderungen“. Letztere Maßnahme ist vergleichbar mit dem Prinzip „Lernen und Anpassen“, welches u. a. durch eine Vielzahl von Forschungsarbeiten zum Thema Innovation beschrieben wird (vgl. Loch et al. 2008, S. 32). Dieses Prinzip bedeutet, dass neue Anforderungen durch Kundenfeedback erkannt und dann in einem Überarbeitungsschritt umgesetzt werden.

Nach (Haferkamp et al. 2008) müssen „viele kleine Iterationen“ durchgeführt werden, „um eine Lösung der unbekannten Anforderungen zu finden“. „[A]uf eine spekulative Gestaltung der Architektur [wird] verzichtet“, „um Zeit zu sparen, und die Realisierung nur bei Bedarf an neue Anforderungen [anzupassen]“. „In der Softwareentwicklung wird dieser Ansatz durch XP verfolgt. Späterer Rework-Aufwand, u. a. durch Refactoring, wird bewusst in Kauf genommen. Die Annahme in XP ist, dass der erwartete Rework-Aufwand konstant über die Zeit und immer niedriger als der einer spekulativen Architekturgestaltung ist. Dagegen steht die Ansicht, dass der Rework-Aufwand mit der Komplexität des Systems steigt, wenn z. B. mehr als ein Modul betroffen ist [vgl. Shore 2004]. Zusätzliche Kosten können durch den Feedbackprozess, z. B. durch die mehrmalige Einführung der SW entstehen.“ (vgl. Haferkamp et al. 2008). Die Anzahl der Iterationen, z. B. der Releasezyklen, kann hier als Handlungsvariable betrachtet werden.

Die Maßnahme „Priorisierung der Anforderungen“ nach (Mathiassen et al. 2007) setzt die Bekanntheit der Anforderungen voraus und kann als zusätzlich mögliche Gestaltungsmaßnahme bei bekannten unsicheren oder noch unbekannten Anforderungen betrachtet werden. Als Maßnahme bei noch unbekannten Anforderungen ist die Priorisierung von Anforderungen wichtig, da der Umfang und der Inhalt der, in einem Release umgesetzten, Anforderungen auch Einfluss auf das Feedback hat.

Diese Maßnahmen entsprechen den, in Abschnitt 2.6 beschriebenen, Steuerungsmöglichkeiten der Releaseplanung und der Planung der Prozessstruktur.

3.2.3. Ergebnis der Analyse der Anforderungsunsicherheit

Für diese Arbeit ist die Einteilung der Anforderungsunsicherheit in *bekannt* und *unbekannt* interessant. In den Modellen dieser Arbeit werden Maßnahmen zur Auflösung

---

Feedback aus der Nutzung der Software, sondern mehr auf die persönliche Kommunikation zwischen Entwicklern und Nutzern.

unbekannter Unsicherheiten betrachtet. Diese Maßnahmen beinhalten – wie oben beschrieben – die Planung der Entwicklungszyklen und die Verteilung der Anforderungen auf diese. Andere Gestaltungsmaßnahmen, z. B. architektonische, zielen auf die Reduktion des Überarbeitungsaufwands bei bekannten Variablen. Diese Unterscheidung bildet zusammen mit Definition 3-1 das Ergebnis der Analyse der Anforderungsunsicherheit, welches Abbildung 3-2 veranschaulicht wird.

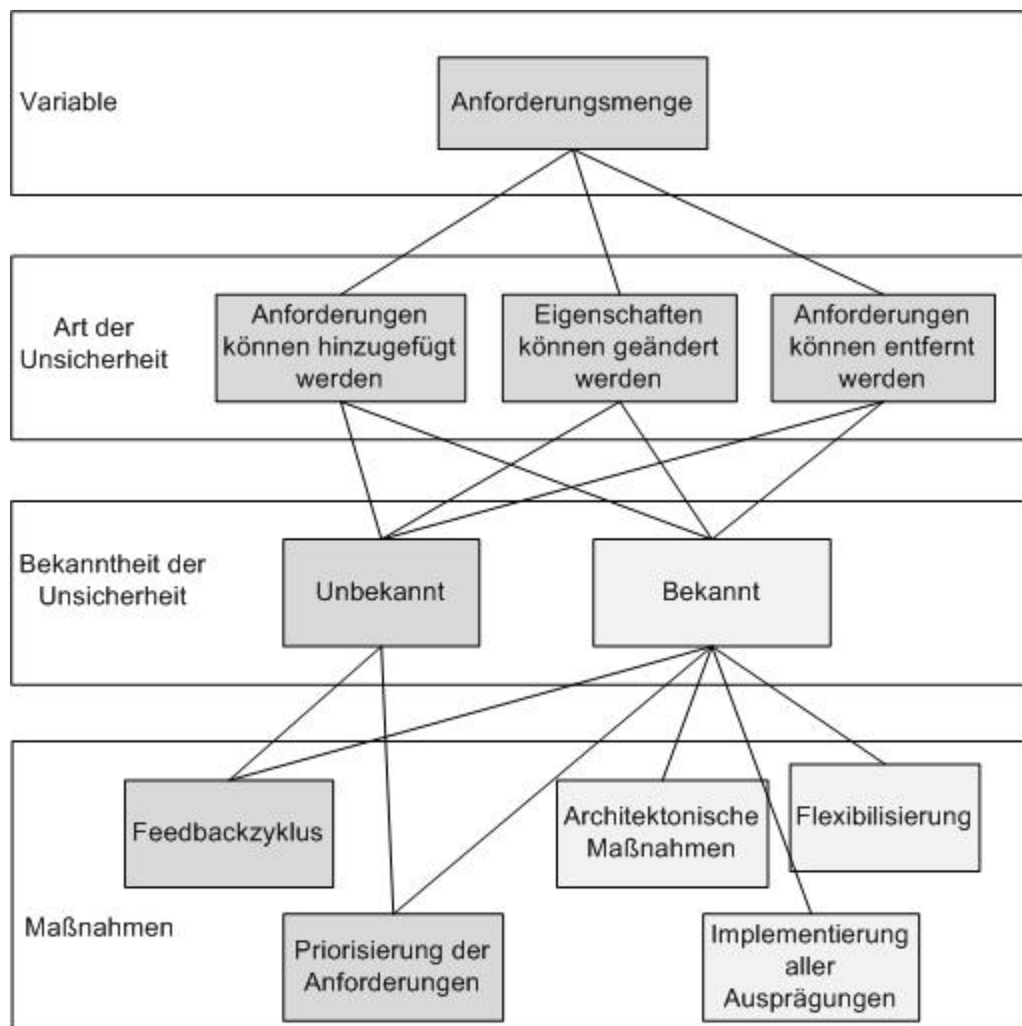


Abbildung 3-2: Zusammenhang von Anforderungsunsicherheit und Maßnahmen

Die in Abschnitt 3.2.2 beschriebene Maßnahme „Warten auf Entscheidung“ bei unsicheren bekannten Anforderungen wurde in Abbildung 3-2 der Maßnahme „Feedbackzyklus“ zugeordnet. Während eines Feedbackzyklus werden sichere Anforderungen entwickelt und die bestehende Unsicherheit am Ende des Zyklus geklärt, was zu Überarbeitungsaufwand führen kann.

Eine andere Kategorisierung der Anforderungsunsicherheit wird in (Mathiassen et al. 2007) beschrieben. Die Autoren kommen zu einer Kategorisierung nach „unbekannte

Anforderungsidentität", „Anforderungsvolatilität" und „Anforderungskomplexität". Die Anforderungsvolatilität ist vergleichbar mit dem Konzept der Unsicherheit durch unbekannte Anforderungen. Anforderungsvolatilität bedeutet nach (Mathiassen et al. 2007), dass sich die Anforderungen aufgrund dynamischer Umgebungen oder individuellem Lernen, d. h. dem Kennenlernen der eigenen Anforderungen, schnell ändern können. Die Kategorien nach (Mathiassen et al. 2007) scheinen allerdings nicht überschneidungsfrei. Diese Schlussfolgerung entsteht durch die Betrachtung der nicht überschneidungsfreien Maßnahmen aus (Mathiassen et al. 2007) zur Reduktion der Risiken, die aus diesen „verschiedenen" Arten von Unsicherheiten entstehen.

3.3. Analyse der Sensitivität/Flexibilität in komplexen Systemen

Aus der Analyse der Anforderungsunsicherheit in Abschnitt 3.2 ergibt sich die Frage: Wie groß ist der Überarbeitungsaufwand bei Änderungen in komplexen Systemen?

Der Überarbeitungsaufwand ergibt sich aus der Differenz zwischen dem „nominalen" Entwicklungsaufwand, d. h. dem Aufwand bei Auflösung aller Unsicherheiten vor Beginn der Entwicklung[93], und dem tatsächlichen Entwicklungsaufwand, der sich durch verzögerte Auflösung der Unsicherheiten erhöht. Das Verhältnis zwischen nominalem und tatsächlichem Entwicklungsaufwand wird hier als Sensitivität bezeichnet. Dieser Faktor wird durch die – in diesem Abschnitt beschriebene – Komplexität des Systems und der Anforderungen beeinflusst. Wie in Abschnitt 3.1 beschrieben, entspricht eine geringe Sensitivität einer hohen Flexibilität bzw. Anpassungsfähigkeit und ist auf das Softwaresystem bezogen ein Qualitätsmerkmal.

In dieser Arbeit wird die Komplexität der Entwicklungsaufgabe aufbauend auf (Jones 1998) unterteilt in

- Abhängigkeiten zwischen Anforderungen (Abschnitt 3.3.1)
- Hierarchie und Modularität des Softwaresystems (Abschnitt 3.3.2)
- Strukturen auf organisatorischer Ebene (Abschnitt 3.3.3)

Allerdings sind die Betrachtungen der drei Bereiche schwer voneinander zu trennen, denn die Komplexität einer Software hängt eng mit der Komplexität der Umgebung, der das

---

[93] Z. B. wenn alle nicht-vorhersehbaren Anforderungen vor Beginn der Entwicklung bekannt gewesen wären.

Programmverhalten angepasst werden soll, zusammen (vgl. Simon 1969). Die drei Bereiche werden hier jeweils unter den Komplexitätsaspekten *Hierarchie* und *Modularität* nach (Simon 1962) untersucht. Der hier als Modularität bezeichnete Komplexitätsaspekt wird in Simon 1969 als Aufteilung des Systems in Subsysteme und den daraus resultierenden Abhängigkeiten zwischen und innerhalb dieser Subsysteme verstanden.

Ein anderer Komplexitätsaspekt nach (Simon 1962) ist die Zeit, die ein System braucht, um sich weiterzuentwickeln und seinen endgültigen Zustand zu erreichen. Systeme, die stabile Zwischenformen erreichen, entwickeln sich schneller als Systeme ohne stabile Zwischenformen. Stabile Zwischenformen lassen sich mit Releases oder Builds in der Softwareentwicklung vergleichen, die daraus entstehenden Systeme sind nach (Simon 1962) hierarchisch geordnet.

Nach (Simon 1962) ist die Beschreibung und die Verständlichkeit des Systems ein weiterer Komplexitätsaspekt. Einfache Beschreibungen und Abstraktionen dienen der Reduktion der Komplexität. Diese Aspekte fließen z. B. in den „*maintenance adjustment factor*“ im Kostenmodell nach (Boehm 2000b, S. 28) ein. Dieser Faktor setzt sich nach (Boehm 2000b, S. 28) aus der Nicht-Vertrautheit des SW-Entwicklers mit dem Programmcode („*programmer unfamiliarity*“) und der Verständlichkeit des Programmcodes („*software understanding*“)[94] zusammen.

Die Verständlichkeit des Programmcodes ergibt sich nach (Boehm 2000b, S. 23) aus der Klarheit des Anwendungsbezug, der Selbstbeschreibung durch Kommentare, der Dokumentation, sowie aus der Codestrukturierung. Letztere entspricht der Modularisierung auf Codeebene, die u. a. auf den SW-Eigenschaften *Kohäsion* und *Information-Hiding* basiert. Allerdings existieren auch Studien, die zeigen, dass das Softwareverständnis mit einer erhöhten Modularisierung abnimmt (vgl. Woodfield et al. 1981).

In (Jones 1998, S. 288) wird eine Übersicht über 24 verschiedene Komplexitätsbegriffe in der Softwareentwicklung angegeben. Der Autor unterteilt die Komplexität in die drei Bereiche Problem-, Code- und Datenkomplexität, die sich zum Großteil auf die Komplexität auf Implementierungsebene beziehen. Er benennt aber auch die

[94] Die Verständlichkeit wird nur bewertet, falls ein Programmierer nicht mit dem Code vertraut ist (vgl. Boehm 2000b, S. 23–24)

organisatorische und semantische Komplexität, letztere bezieht sich auf die Komplexität der Anforderungen.

3.3.1. Abhängigkeiten zwischen Anforderungen

Die Komplexität der Anforderungen kann unter dem Aspekt hierarchischer und interdependenter Kundenanforderungen und Systemanforderungen betrachtet werden. Die Komplexität der Anforderungen ist dann mit der Komplexität nach (Simon 1962) vergleichbar.

Eine Art Hierarchie zwischen Anforderungen ist in Beispiel 2-1 aus Abschnitt 2.1 zu erkennen: Verschiedene Systemanforderungen gehören dort zu einer Kundenanforderung. Diese Systemanforderungen erben gemeinsame Eigenschaften aus der Kundenanforderung, wie z. B. den Akteur „Kunde". Ändert sich der Akteur, so müssen möglicherweise auch die Systemanforderungen angepasst werden.[95]

Eine Art Modularität ist bei der Zerlegung einer Anforderung in eine Ereignisfolge zu erkennen.[96] Hier entspricht ein Ereignis einem „Anforderungsmodul". Innerhalb des Moduls können Abhängigkeiten durch Interaktionen und zeitliche Abläufe von Systemfunktionen bestehen. Zwischen den Modulen ergeben sich Abhängigkeiten z. B. aus einer Ereignisfolge. Eine solche Ereignisfolge ist z. B. das Szenario „Kaufvorgang", welches aus folgenden sequenziell abhängigen Ereignissen besteht: „Artikel ansehen", „Artikel in den Warenkorb legen", „zur Kasse gehen", „Anschrift angeben", „Artikel bezahlen". Dieses Szenario ist dadurch gekennzeichnet, dass zwischen den Ereignissen Aktionen außerhalb der Kontrolle des Softwaresystems stattfinden. Dadurch können sich nicht-vorhersehbare Abhängigkeiten zwischen den Anforderungen ergeben. Mögliche Szenarios werden in einem Softwaresystem durch Zustandsautomaten modelliert. Durch diese Zustandsautomaten wird dargestellt in welchem Zustand ein System welche Ereignisse verarbeiten kann.

Die oben beschriebenen Abhängigkeiten zwischen Anforderungen ergeben sich daraus, dass ein Softwaresystem an der Schnittstelle zur Anwendungsdomäne direkt von den Eigenschaften dieser Anwendungsdomäne abhängig ist (vgl. Abschnitt 2.1). Ein weiteres

---

[95] Z. B. kann eine Änderung darin bestehen, dass der Akteur nicht mehr ein „normaler" Kunde ist sondern ein registrierter Kunde sein muss.

[96] Vgl. zu diesem Absatz auch die Modellierung durch Zustandsautomaten und Aktivitätsdiagramme nach Balzert 2001, S. 315–339 und den szenariobasierten Ansatz der Extreme-Programming-Methodik.

Beispiel hierzu findet sich in (Sellier et al. 2008). Die Autoren untersuchen ein SW-System, welches davon abhängig ist, ob ein Mobiltelefon eine Kamera hat oder nicht. Hat es eine Kamera, so muss die Software die Funktion „Aufnahme von Videos“ bieten. Von dieser Anforderung sind weitere Systemfunktionen abhängig, z. B. hat die Auflösung der Kamera durch den begrenzten Speicherplatz des Telefons Einfluss auf die maximale Länge eines Videos.

In (Bieman et al. 2003) werden die, in diesem Abschnitt betrachteten, Abhängigkeiten, durch die Betrachtung von Modulen, die häufig gleichzeitig Änderungen unterliegen, analysiert. Die Autoren kommen zu dem Ergebnis, dass Änderungsabhängigkeiten nicht unbedingt funktionale Zusammenhänge abbilden. Sie vermuten, dass diese Abhängigkeiten durch Änderungen in Qualitätsmerkmalen zustande kommen.

Maßnahmen bei Abhängigkeiten

In (Jootar, Eppinger 2002) zeigen die Autoren, dass unsichere Anforderungen in den ersten Entwicklungszyklen implementiert werden müssen, um den Überarbeitungsaufwand an den, von diesen Anforderungen abhängigen, Schnittstellen zu minimieren.[97] Diese Beobachtung betrifft funktionale und nicht-funktionale Anforderungen (vgl. Bieman et al. 2003) Durch eine frühzeitige Implementierung unsicherer Anforderungen, sinkt die Wahrscheinlichkeit, dass eine davon abhängige Schnittstelle in späteren Entwicklungszyklen überarbeitet werden muss. Implizit wird in (Jootar, Eppinger 2002) davon ausgegangen, dass, in der Phase der Risikoabschwächung („*risk mitigation phase*“), durch Feedback aus der Nutzung, alle Unsicherheiten des aktuellen Systems geklärt werden können. Diese Annahme wird auch in den später in dieser Arbeit vorgestellten Modellen getroffen. In (Jootar, Eppinger 2002) wird auch gezeigt, dass die Abhängigkeiten zwischen Schnittstellen und Funktionen durch geeignete (architektonische) Gestaltungsmaßnahmen stark reduziert werden können.

Abbildung 3-3 veranschaulicht, dass – unabhängig davon ob die unsicheren Variablen bekannt sind oder nicht – bei Abhängigkeiten zwischen Systemanforderungen (Features), eine erhöhte Zahl von Entwicklungszyklen und damit verbundenes frühzeitiges Feedback von Vorteil sind. Hier wird ein zwei-zyklischer mit einem vier-zyklischen

[97] Hier wird die Implementierung von Funktionen aus Jootar, Eppinger 2002 mit der Implementierung von Systemanforderungen gleichgesetzt. Das beschriebene Ergebnis erhalten die Autoren in einem ersten Schritt in dem Module mit Funktionen gleichgesetzt werden.

Entwicklungsprozess verglichen. Die Abbildung ist gemäß dem Zeitverlauf von links nach rechts zu lesen. In dieser Abbildung stellt jede Ellipse ein zu entwickelndes Feature dar. Die Abhängigkeit eines Features von einem anderen wird durch einen Pfeil, ausgehend vom abhängigen Feature, gekennzeichnet, d. h. das abhängige Feature sollte zeitlich später implementiert werden. Für jeden Zyklus wird das System mit allen, am Ende des Zyklus enthaltenen, Features dargestellt. Dargestellt ist auch das, zwischen jedem Zyklus stattfindende, Feedback, welches zu einer Änderung innerhalb eines Features, zur Forderung neuer Features oder zum Entfernen bisheriger Features führen kann. Solche Featureänderungen verursachen Überarbeitungsaufwand und werden grau dargestellt.

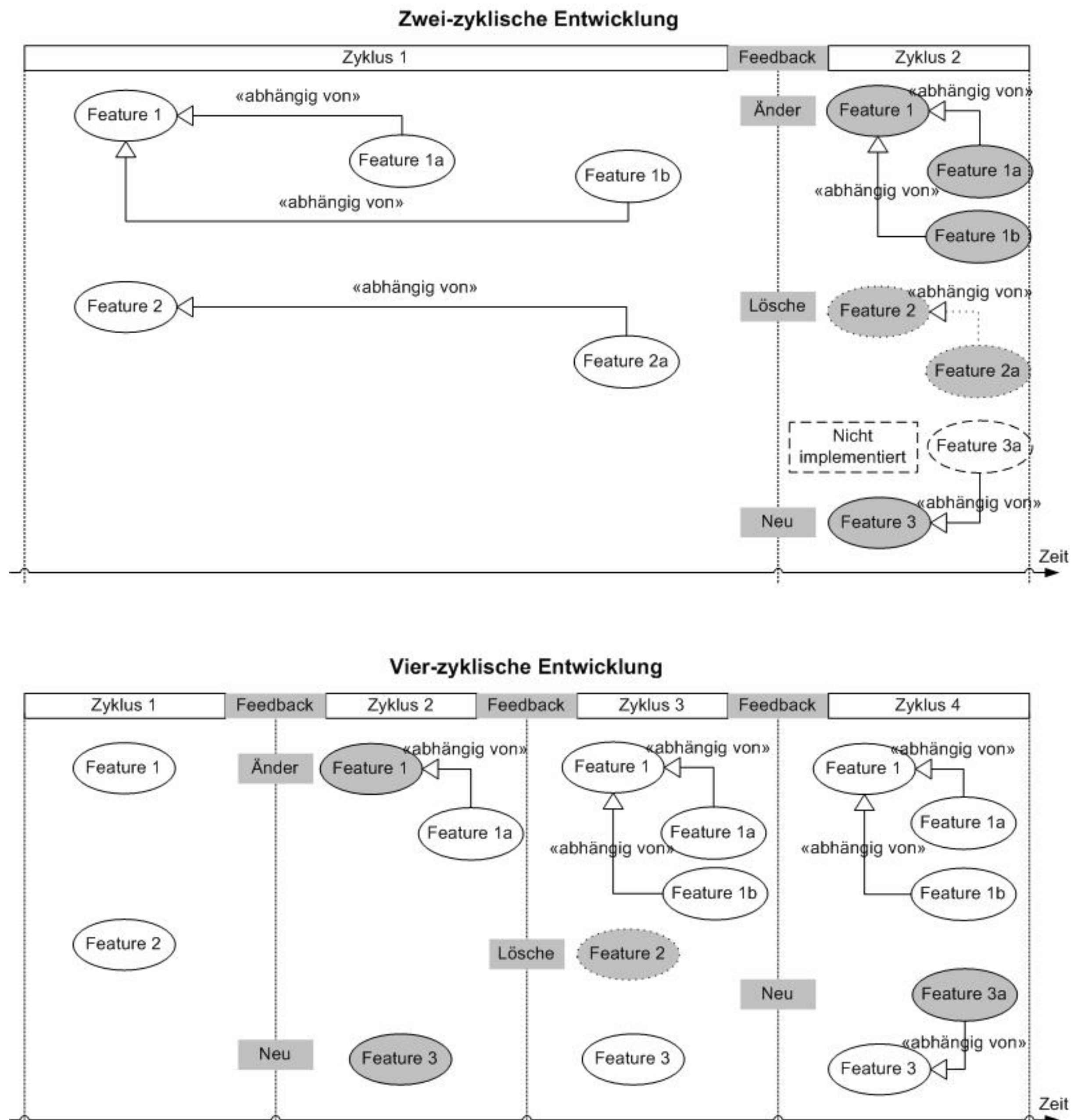


Abbildung 3-3: Vorteil von mehreren Entwicklungszyklen bei Abhängigkeiten zwischen Systemanforderungen

Auch wieder entfernte Features sind grau dargestellt, da sie überflüssigen Aufwand darstellen. Alle grauen Markierungen zusammen ergeben den gesamten Überarbeitungsaufwand.

Aus der Abbildung soll ersichtlich werden, dass in der vier-zyklischen Entwicklung durch Feedback, in dem die Unsicherheit von Features frühzeitig geklärt wird, weniger Aufwand entsteht als in der zwei-zyklischen Entwicklung. Dies ist daran zu erkennen, dass in der vier-zyklischen Entwicklung weniger Änderungen an abhängigen Features durchgeführt werden müssen, da die Features, von denen diese abhängen, nach Feedback

nicht mehr geändert werden. Zusätzlich wird dargestellt, dass durch Feedback zu einem neu erkannten und implementierten Feature weitere neue Features erkannt und implementiert werden können.

Wird der Aufwand für alle Features gleich berechnet, bedeutete dies: Im vier-zyklischen Entwicklungsprozess sind vier Überarbeitungen von Features bei fünf implementierten Features insgesamt zu erkennen. D. h. bei einer Gleichgewichtung von Überarbeitungs- und Implementierungsaufwand erhöht sich der nominalen Implementierungsaufwand um 80%. Dem gegenüber stehen, im zwei-zyklischen Entwicklungsprozess, sechs Überarbeitungen von Features bei vier implementierten Features insgesamt. D. h. der Aufwand erhöht sich um 150%. Dieser Vergleich zeigt beispielhaft den geringeren Überarbeitungsaufwand pro Feature, bei einer erhöhten Anzahl von Entwicklungszyklen.

3.3.2. Hierarchie und Modularität des Softwaresystems

In (Banker et al. 1998) wird der Einfluss der Softwarekomplexität auf die Sensitivität untersucht. Die Autoren kommen zu dem Schluss, dass größere Projekte eine höhere Komplexität und eine höhere Sensitivität aufweisen als kleinere Projekte. In (Gibson, Senn 1989) untersuchen die Autoren den Einfluss der Systemstruktur auf die Sensitivität. Sie kommen zu dem Schluss, dass strukturiertere Systeme eine geringere Sensitivität mit sich bringen.

Die Komplexität des Softwaresystems wird hier in die Komplexität innerhalb eines Moduls, d. h. auf Codeebene und die Komplexität der Modulstruktur (vgl. Bass et al. 2005, S. 36–37) der Software untergliedert.

Zur Bestimmung der Komplexität auf Codeebene existieren nach (Jones 1998, S. 288) verschiedene Metriken. Die bekannteste (vgl. Lehman, Belady 1985, S. 335) ist die zyklomatische und die, davon abgeleitete, essenzielle Komplexität nach (McCabe 1976), die auf dem Kontrollflussgraphen z. B. eines Algorithmus, basiert. Die Messung dieser Komplexität ist in modernen Entwicklungsumgebungen wie Microsofts Visual Studio .NET integriert. Daneben existieren nach (Jones 1998) weitere formale Komplexitätsmaße wie die algorithmische Komplexität und die Datenkomplexität, sowie subjektive Maße, wie die Code- und Problem-Komplexität, die zum Kalibrieren der vorangehenden Maße verwendet werden können. Die Komplexität auf Codeebene wird in (Boehm 2000b, S. 42) den Produktfaktoren zugeordnet und fließt damit wie in Abschnitt 3.1.2 beschrieben in die Berechnung des Produktumfangs ein.

In diesem Abschnitt soll besonders die Komplexität der Modulstruktur als Einflussfaktor der Sensitivität bzw. Flexibilität und damit des Überarbeitungsaufwands betrachtet werden (vgl. auch Bass et al. 2005). Auf Ebene der Modulstruktur, als Teil der Softwarearchitektur, existieren verschiedene Methoden zur Reduktion des Überarbeitungsaufwands. Diese sind u. a.

- Präventive Architekturgestaltung und Verwendung von Design Pattern (vgl. Bass et al. 2005)
- (Multiples) Refactoring (vgl. Beck 2000; Tokuda, Batory 2001), d. h. nachträgliche Architekturgestaltung
- Roundtrip-Engineering: Reverse Engineering und Modellbasierte Entwicklung (vgl. Frankel 2003)

Im Folgenden soll auf die ersten beiden Punkte eingegangen werden.

Architektonische Gestaltungsmaßnahmen

Nach (Boehm 2000b, S. 22) kann der Überarbeitungsaufwand durch eine Architektur, welche kohärente Module kapselt und somit die Abhängigkeiten zwischen Modulen reduziert, verringert werden.

Die Prinzipien der objektorientierten Programmierung, wie Vererbung und Datenkapselung, fördern eine, jedem objektorientierten System intrinsische, Hierarchie und Modularität (vgl. Beck 2000). Objektorientierte Datenkapselung bedeutet, dass Daten und Prozesse gemeinsam einem Objekt zugeordnet werden. Dem gegenüber steht die strukturierte Entwicklung, in der Objekte und Prozesse getrennt werden (vgl. Sircar et al. 2001).

Diese Prinzipien erhöhen nach (Bass et al. 2005, S. 105–110) die Anpassungsfähigkeit und Flexibilität der Software. In (Beck 2000, S. 56–57) wird der Nutzen eines hohen Architekturaufwands zu Beginn der Entwicklung in Frage gestellt. Der Autor vertritt die Meinung, dass Änderungen in den Anforderungen meist nicht vorhersehbar sind und daher auch keine spezifischen architektonischen Vorkehrungen getroffen werden können, um den späteren Überarbeitungsaufwand zu reduzieren. Er ist der Meinung, dass der Aufwand durch nachträgliche Architekturänderungen (mittels Refactoring) geringer ist.

In (MacCormack et al. 2001) kommen die Autoren aufgrund ihrer empirischen Untersuchung, in einem von Änderungen geprägten Umfeld, zu dem Schluss, dass

Projekte in denen ein vergleichsweise hoher Aufwand in den Entwurf der Architektur gesteckt wird, zu qualitativ höherwertigen Softwareprodukten führen. Die Autoren sind der Ansicht, dass ein hoher Aufwand zu tätigen ist, um eine Architektur zu erstellen, die einen flexiblen Entwicklungsprozess unterstützt. Die Architektur sollte so gestaltet werden, dass Systemfunktionen schon früh im Prozess lauffähig sind und weitere Funktionalitäten einfach hinzugefügt werden können. Den Autoren zufolge ergeben sich drei Anforderungen an die Systemarchitektur in unsicheren Umgebungen (vgl. MacCormack et al. 2001):

- Unsichere oder dynamische Anforderungen sollen durch (architektonische) Puffer von stabilen getrennt werden
- Kritische Elemente sollen demonstriert werden können, auch wenn andere Elemente noch nicht vollständig sind
- Neue Funktionalitäten sollen ohne großen Aufwand hinzugefügt werden können

Um die letzten zwei Punkte zu erfüllen wird in (Bass et al. 2005, S. 170) die Verwendung einer Schichtenarchitektur beschrieben. Die Autoren erläutern dort auch den Nutzen von vorgefertigten Schnittstellen (engl.: *stubs*). Solche Schnittstellen simulieren in anfänglichen Zyklen eine bestimmte Funktionalität, welche aber erst in späteren Zyklen implementiert wird. Dadurch reduziert sich der Überarbeitungsaufwand an diesen Schnittstellen.

Ein ähnlicher Ansatz wird in (Jootar, Eppinger 2002) vorgestellt. In ihrer Untersuchung gehen die Autoren davon aus, dass die Anzahl der, zu Schnittstellen gehörenden, noch nicht realisierten Funktionen im Laufe der Zyklen abgearbeitet werden muss und solange ein bestimmter Prozentsatz der Schnittstellen Änderungen unterliegt. D. h. der Überarbeitungsaufwand fällt bei jeder Änderung an, solange nicht alle Abhängigkeiten geklärt wurden.

Nach (Jootar, Eppinger 2002) sollte die Entwicklung so aufgeteilt werden, dass

- zuerst die Module mit hoher Unsicherheit realisiert werden.[98]
- zuerst die *Low-Level-Funktionen* (Kernfunktionen) und dann die *High-Level-Funktionen* realisiert werden.

[98] Unter der Annahme, dass die Module voneinander unabhängig sind.

- eine optimale Kombination der beiden vorangehenden Regeln erreicht wird.[99]

Low-Level-Funktionen werden von mehr Modulen verwendet als High-Level-Funktionen. Diese Low-Level-Funktionen können nach (Jones 1998, S. 250) durch die Betrachtung der *Fan-Komplexität*[100] ermittelt werden Ein hoher *Fan-In* bedeutet: Dies ist ein kritisches Kernmodul. Ein hoher *Fan-Out* bedeutet: Dieses Modul ist schwierig zu debuggen, da es viele andere Module aufruft, aus denen die Fehler stammen können.

Grenzen der Modularisierung

In (Lang 2004, S. 270–279) werden die Grenzen der Modularisierung aufgezeigt. Diese ergeben sich aus:

- der Intensität der Beziehungen: Zu große Module können problematisch sein, wenn z. B. in kleinen Teams entwickelt werden soll. Eine weitere Modularisierung kann sich aber negativ auswirken, da das System die Voraussetzung der Modularisierbarkeit nicht erfüllt und eine hohe intermodulare Komplexität zu einer höheren Fehlerzahl und Entwicklungskosten führt.
- dem Wissensstand. Bei neuartigen Entwicklungsvorhaben ist das Wissen über die Beziehungen zwischen den verschiedenen Modulen oft unzureichend. Das gleiche gilt für neuartige Technologien für die Wissen nur durch Reviews, Tests und Feedback erlangt werden kann. Die Festlegung von Schnittstellen kann unzweckmäßig sein, wenn sich durch die Änderung einer Schnittstelle viele andere Module ändern. Der Vorteil, dass durch die Modularisierung der Suchraum der Entwickler eingeschränkt wird, kann sich bei fehlerhaften Architektur-Entscheidungen negativ auswirken, da die Aufmerksamkeit der Entwickler möglicherweise in die falsche Richtung gelenkt wurde.
- der Zweckmäßigkeit: Eine hohe Modularisierung ist in Echtzeitsystemen nur bedingt sinnvoll, da dies zu schlechteren Reaktionszeiten und einer geringeren Effizienz der Speicherbelegung führen kann (vgl. auch Bratthall, Runeson 2000). Um dies zu vermeiden, können effizienzkritische Funktionalitäten allerdings auch in einem Subsystem gebündelt werden (vgl. Sommerville 2007, S. 218). Die

[99] Dieser letzte Punkt wurde von den Autoren noch nicht untersucht.

[100] Der Fan-In eines Moduls bezeichnet die Anzahl der Module durch die ein Modul aufgerufen wird (Anzahl der eingehenden Kanten). Der Fan-Out eines Moduls bezeichnet die Anzahl der Module, die ein Modul aufruft (Anzahl der ausgehenden Kanten).

Modularisierung kann weiterhin unzweckmäßig sein, wenn der Zeitaufwand des Architekturdesigns den später verringerten Überarbeitungsaufwand übertrifft. Bei wachsendem funktionalem Umfang des Systems muss die Architektur immer wieder angepasst werden und die Beziehungen zwischen Modulen werden möglicherweise immer komplexer. Allerdings kann eine modulare Architektur auch Vorteile bei der Integration neuer Funktionen bieten (*Skalierbarkeit*).

3.3.3. Strukturen auf organisatorischer Ebene

In (Gomes, Joglekar 2008) wird die Modularität von Aufgaben (z. B. der Implementierungsaktivität) auf Basis der *Design-Structure-Matrix* (*DSM*) -Methodik nach (Steward 1981) definiert und gemessen. Diese Methodik, aus der Engineering-Design-Literatur, bildet Abhängigkeiten bzw. Informationsflüsse zwischen Aufgaben ab. Die Autoren ermitteln die Modularität von Aufgaben durch verschiedene Maße: Abhängigkeiten, Sichtbarkeit und eine Kombination aus beidem.

Organisatorische Handlungsalternativen

Organisatorische Handlungsalternativen bzw. Handlungsvariablen dienen der Regelung der horizontalen/vertikalen Arbeitsteilung (Kompetenzsystem) und der horizontalen/vertikalen Informationsautonomie (Kommunikationssystem).[101]

Nach (Bass et al. 2005, S. 39,167-170) hängt die Aufgabenverteilung auf Entwicklerteams und die daraus resultierende *Teamstruktur* eng mit der Modulstruktur der Software zusammen. Die Teamstruktur sollte den Autoren zufolge den Teams Module zuweisen, so dass die Mitglieder eines Teams Expertenwissen für diese Module haben. Auch nach (Gomes, Joglekar 2008) sollten die Teams so aufgeteilt werden, dass die Vorteile der Spezialisierung voll genutzt werden können. Kommunikation darf im Idealfall nur zwischen Teams stattfinden, deren Module gemeinsame Schnittstellen haben. Mit der Komplexität der Modulstruktur steigt also die Komplexität der Teamstruktur und umgekehrt.

Aus diesem Zusammenhang folgt, dass die in Abschnitt 3.3.2 beschriebenen architektonische Gestaltungsmaßnahmen, wie die Modularisierung, zur Reduktion von Abhängigkeiten teilweise auf die Aufgabenverteilung übertragbar. Die Untersuchung aus (Gomes, Joglekar 2008) kommt zu dem Schluss, dass eine erhöhte Modularität den

[101] Vgl. Frese, Mensching 1986 zitiert nach Lang 2004, S. 115.

Entwicklungsaufwand und Koordinations- bzw. Kommunikationsaufwand reduziert. Eine „gute“ Modularisierung unterteilt zwischen sichtbaren Informationen an den Schnittstellen der Module und versteckten Informationen, die nur innerhalb eines Moduls bearbeitet werden (vgl. Baldwin, Clark 2000).

Es kann angenommen werden, dass auch die Reduktion der Komplexität der Modulstruktur durch eine hierarchische Strukturierung (vgl. Abschnitt 4.4.2) eine Rolle spielt. Längere Hierarchien scheinen zunächst weniger Kommunikationsaufwand zur Folge zu haben. Allerdings werden in (McAfee, McMillan 1995) lange Hierarchien in der Unternehmensorganisation als Ursache für erhöhte Kosten und Unternehmen mit solchen Hierarchien als nicht überlebensfähig in umkämpften Branchen angesehen. Im *Principal-Agent-Modell* der Autoren führen längere Hierarchien des Weiteren zu Verzerrungen der kommunizierten Informationen. In (McAfee, McMillan 1995) wird davon ausgegangen, dass die Kosten mit der Hierarchielänge überproportional steigen (negative Skaleneffekte).

Die Vermutung, dass negative Skaleneffekte in der Unternehmensorganisation existieren bestätigt ein – dem Autor dieser Arbeit gegenüber geäußertes – Zitat aus der Geschäftsführung eines IT-Service-Providers (ca. 1500 Mitarbeiter):

> *„Hab ich [...] gemerkt, dass ein Projekt von 80 statt 8 Mann nicht 10-mal so kompliziert ist, sondern eher 100-mal... Das denkt man nicht, aber die Komplexität der Probleme nimmt mit den Beteiligten vermutlich exponentiell zu!“*

3.4. Übersicht der in den Modellen betrachteten Einflussfaktoren

In dieser Arbeit sollen die Auswirkungen der Einflussfaktoren auf eine optimale Prozessplanung (vgl. Abschnitt 2.6) – mit dem Ziel die Entwicklungszeit zu minimieren – betrachtet werden. Die in diesem Kapitel identifizierten Faktoren mit Einfluss auf die Entwicklungszeit werden unterteilt in

- Eigenschaften der Informationseinheiten im Entwicklungsprozess (vgl. auch Abbildung 2-2)
- Eigenschaften der Organisation, sowie interner und externer Stakeholder, welche an den Aktivitäten im Entwicklungsprozess beteiligt sind (vgl. auch Tabelle 2-2).

Es folgt eine kurze Definition, zusammen mit den Variablenbezeichnungen, der in dieser Arbeit betrachteten Einflussfaktoren der Entwicklungszeit und der Steuerungsmöglichkeiten der Prozessplanung.

Eigenschaften der Informationseinheiten im Entwicklungsprozess

- *Erweiterter Produktumfang*

  Variablenbezeichnung: S (für **S**ize)

  Der erweiterte Produktumfang ergibt sich in dieser Arbeit aus der Anzahl und dem Inhalt spezifizierter bekannter funktionaler *und* nicht-funktionalen Anforderungen (mit Ausnahme der Flexibilitätsanforderung) interner und externer Stakeholder. Die Abschätzung des Umfangs zur Prozessplanung kann wie in Abschnitt 3.1.2 beschrieben z. B. durch eine Kombination aus Function-Points oder Story-Points und Cocomo-II-Metriken geschehen.

  Der Umfang des Quellcodes, als Resultat der Implementierung der spezifizierten Anforderungen, kann z.B. in Codezeilen (SLOC, engl.: *source lines of code*, vgl. Albrecht, Gafeney Jr. 1983) gemessen werden. Weitere Methoden für eine komplexere Umfangsmessung werden z. B. in (Allen et al. 2007) vorgeschlagen. In (Jones 2008, S. 110) wird eine Tabelle zur Umrechnung zwischen Function-Points und SLOC abhängig von der verwendeten Programmiersprache angegeben. Diese Umrechnung kann verwendet, um eine einheitliche Größe zu verwenden und z. B. die Planung auf Basis des aktuellen Produktumfangs anzupassen. Auch die Messung der Produktqualität, auf Basis der nicht-funktionalen Anforderungen, dient der Bestimmung des aktuellen erweiterten Produktumfangs.

- *Sensitivität/ Flexibilität*

  Variablenbezeichnung: c

  Wie oben beschrieben beinhaltet der erweiterte Produktumfang die Produktqualität mit Ausnahme der Flexibilität, da in dieser Arbeit speziell der Einfluss dieses Qualitätsmerkmals auf den Entwicklungsaufwand näher betrachtet wird.

  Das Verhältnis zwischen nominalen und dem – bei einer verzögerten Auflösung der Unsicherheiten – durch Abhängigkeiten erhöhte Entwicklungsaufwand wird

hier als Sensitivität bezeichnet. Eine geringe Sensitivität entspricht einer hohen Flexibilität oder Änderbarkeit.

- *Anforderungsunsicherheit / Änderungswahrscheinlichkeit*

  Variablenbezeichnung: q

  In dieser Arbeit wird der Anforderungsumfang als unsicher angesehen. Diese Unsicherheit kann durch Wahrscheinlichkeitsverteilungen und Erwartungswerte des Änderungsumfangs modelliert werden (vgl. Loch, Terwiesch 1998, S. 1035–1036). Der Änderungsumfang kann z. B. durch $q \cdot S$ berechnet werden. Die genauen Annahmen werden im jeweiligen Modell erläutert.

Eigenschaften der Organisation, sowie interner und externer Stakeholder

- *Gesamtproduktivität*

  Variablenbezeichnung: p_t

  In dieser Arbeit wird angenommen, dass sich die Eigenschaften des Personals und der Organisation auf den Einflussfaktor Produktivität auswirken und durch diesen abgebildet werden können. Hier wird die Produktivität des gesamten Projektteams – wie in Abschnitt 3.1.2 beschrieben – betrachtet. Diese wird, um die Entwicklungszeit zu berechnen, als Zeit (*t*) pro Umfangseinheit (z. B. Zeit pro Function-Point) definiert. Der Produktivität nicht zugeordnet wird die Feedbackeffektivität, die in dieser Arbeit gesondert betrachtet wird.

- *Feedbackeffektivität*

  Variablenbezeichnung: d

  Dieser Faktor ist ein Maß dafür, wie effektiv Anforderungsänderungen durch Feedback, auf Grundlage des aktuellen Systems, erkannt werden können. Voraussetzung für effektives Feedback sind Abhängigkeiten zwischen Anforderungen. Dieser Faktor wird in dem in Abschnitt 5.2 vorgestellten Modell berücksichtigt.

- *Umfangsunabhängiger Teil der Entwicklungsdauer (Fixkosten) eines Zyklus*

  Variablenbezeichnung: t_f

Dieser Faktor beinhaltet sämtlichen Aufwand, der unabhängig vom Produktumfang zur erbringen ist und damit auch nicht durch die oben genannte Produktivität beeinflusst wird. Dieser ergibt sich, wie in Abschnitt 3.1.3 beschrieben, z. B. aus der Anzahl der Installationen oder dem Kommunikationsaufwand. Dieser Faktor wird in Abschnitt 4.5.2 unter dem Aspekt der umfangsunabhängigen Entwicklungsdauer innerhalb eines Zyklus gesondert betrachtet.

Steuerungsmöglichkeiten der Prozessplanung

- *Anzahl der Zyklen*

  Variablenbezeichnung: n

  Die Anzahl der Zyklen entspricht je nach betrachteter Zyklusebene (vgl. Abschnitt 2.3) der Anzahl der Releases, Builds, Meetings oder Prototypingzyklen. Diese Variable wird in den Modellen dieser Arbeit bezüglich des Entwicklungsaufwands optimiert.

- *Evolutionsfaktor*

  Variablenbezeichnung: e

  Dieser Faktor dient der Releaseplanung, d. h. der Verteilung des Entwicklungsumfangs auf die Zyklen.

Diese beiden, letztgenannten Prozesseigenschaften beeinflussen Zeitpunkt, Art und Wirkung des Nutzerfeedbacks.

- *Dauer der Startphase*

  Variablenbezeichnung: T_S

  Die Dauer der Startphase wird auch unter dem Aspekt der Überlappung zwischen Analyse- und Implementierungsaktivität betrachtet.

Weitere in dieser Arbeit nicht betrachtete Faktoren finden sich z. B. in (Levary, Lin 1991) oder (Abdel-Hamid, Madnick 1989). In den weiteren Kapiteln sollen die Auswirkung der in diesem Abschnitt vorgestellten Faktoren auf den Entwicklungsaufwand und die Entwicklungsdauer betrachtet werden.

Kapitel 4: Kostenfunktionen für Prozesse mit zyklischem Nutzerfeedback

Nach dem Prinzip "you can't control what you can't measure" (DeMarco 1982, S. 6) ist eine Messung der Entwicklungszeit, z. B. durch *Kostenfunktionen*, Voraussetzung für die Prozessplanung und Steuerung. In diesem Kapitel werden die gemeinsamen Eigenschaften verschiedener Kostenfunktionen untersucht. Diese modellieren die Entwicklungszeit in Abhängigkeit der verschiedenen Einflussfaktoren, die in Kapitel 3 analysiert wurden. Durch Optimierung der Entwicklungszeit, in Bezug auf steuerbare Einflussfaktoren, wie die Anzahl der Zyklen, können die bestmöglichen Ausprägungen dieser erkannt und für die Prozessplanung verwendet werden. Durch die Kostenfunktionen kann ein Trade-Off zwischen Kosten und Nutzen berechnet werden. Kosten entstehen in den Modellen aus dem, in der Summe erhöhten, Zeitaufwand durch die Fixkosten der Zyklen. Der Nutzen entsteht durch die Vermeidung von Überarbeitungsaufwand durch eine Verkürzung der Zyklen.

4.1. Unterschiedliche Arten von Kostenfunktionen

Für die Prozessplanung ist eine Auswahl aus verschiedenen, strukturell unterschiedlichen Kostenfunktionen zu treffen. In (Hu 1997) wurde eine Übersicht verschiedener Produktionsfunktionen – wie sie in der Literatur zur Softwareentwicklung diskutiert werden – zusammengestellt.[102] (Hu 1997, S. 380) setzt die mathematische Dualität zwischen Produktionsfunktionen und Kostenfunktionen voraus.[103] Produktionsfunktionen beschäftigen sich, dem Autor nach, mit dem Prozess der Softwareentwicklung und sollten daher als Grundlage für die Untersuchung der Charakteristika der Softwareentwicklung dienen. Der Autor unterscheidet in seiner Übersicht zwischen vier Produktionsfunktionen. Hier werden die dazu äquivalenten vier Kostenfunktionen vorgestellt.

---

[102] Eine produktunabhängige Übersicht und tiefergehende Diskussion verschiedener Produktions- und Kostenfunktionen findet sich in Walters 1963.

[103] Hier bedeutet das Dualitätsprinzip, dass bei gegebener funktionaler Struktur die Minimierung der Kosten eng mit der Maximierung der Produktivität zusammenhängt.

Das *lineare Modell* der Kostenfunktion wird in Gleichung 4-1 dargestellt. Es entspricht der (einfachen) Regressionsgleichung zur Kostenschätzung in der Softwareentwicklung auf Basis der Function-Point-Metrik nach (Albrecht, Gafeney Jr. 1983).

$$E = a_0 + a_1 S$$

Gleichung 4-1: Lineares Modell der Kostenfunktion

Auf Basis des Produktumfangs *S* und des Entwicklungsaufwands *E* vergangener Projekte kann nach (Albrecht, Gafeney Jr. 1983) die Regressionsgleichung zur Prognose des Entwicklungsaufwands neuer Projekte bestimmt werden.[104] Hierbei resultieren a_1 und a_0 aus der Regression, d. h. es sind keine projektspezifisch ermittelten Einflussfaktoren. Sie können aber als durchschnittliche Produktivität (a_1) und Fixkosten (a_0) der Projekte eines Unternehmens angesehen werden. Zur Schätzung des Aufwands eines neuen Projekts reicht die Bestimmung des Produktumfangs, der dann in die Regressionsgleichung eingesetzt werden kann.

Das in Gleichung 4-2 dargestellte *quadratische Modell* stimmt am besten[105] mit den durch (Hu 1997) ermittelten Daten überein. Durch den quadratischen Term in dieser Gleichung werden negative Skaleneffekte, d. h. überproportional zur Produktgröße wachsende Nachteile, modelliert.

$$E = a_0 + a_1 S + a_2 S^2$$

Gleichung 4-2: Quadratisches Modell der Kostenfunktion

In (Pickard et al. 1999) wird die Untersuchung aus (Hu 1997) kritisiert und einer eigenen Datenanalyse folgend das in Gleichung 4-3 dargestellte *Cobb-Douglas-Modell* als genauer und robuster angesehen. Dieses Modell ist äquivalent zu einer Cobb-Douglas-Produktionsfunktion mit dem Aufwand *E* als einziger Produktionsressource.[106]

$$E = a_0 + a_1 S^{a_2}$$

Gleichung 4-3: Cobb-Douglas Modell der Kostenfunktion

[104] Ermittelt z. B. durch Regression mit Hilfe der minimalen Fehlerquadratmethode

[105] Auf Basis verschiedener Testkriterien, wie Goodness-of-fit und P-Test (vgl. Hu 1997).

[106] Siehe auch Cobb, Douglas 1928.

Das Cobb-Douglas-Modell wird auch als loglineares Modell bezeichnet (z. B. in Banker, Kemerer 1989), da zur Schätzung der Parameter mittels einfacher Regression der Logarithmus auf beiden Seiten von Gleichung 4-3 genommen wird.

In (Banker, Kemerer 1989) wird das in Gleichung 4-4 dargestellte *Translog-Modell* vorgeschlagen. Dieses Modell ist, den Autoren zufolge, allgemeingültiger als das Cobb-Douglas-Modell. Es erlaubt, dass die durchschnittliche Produktivität ansteigt, wenn die Fixkosten über immer größere Projekte verteilt werden und nachdem die optimale Projektgröße („*most productive project size*“) erreicht wurde, diese durchschnittliche Produktivität wieder sinkt (vgl. Banker, Kemerer 1989). Eine sinkende Produktivität bei zu großen Projekten begründen die Autoren durch negative Skaleneffekte, wie eine überproportionale Zunahme der Kommunikationspfade. Der beschriebene Effekt wird in der Modellierung durch einen größenabhängigen Exponenten erreicht.

$$E = e^{a_0} S^{a_1} S^{a_2 \ln S}$$

Gleichung 4-4: Translog Modell der Kostenfunktion

Wie auch schon beim Einsatz des Cobb-Douglas-Modells wird zur Parameterschätzung der Logarithmus auf beiden Seiten von Gleichung 4-4 genommen. Dieses Modell wird in (Banker, Kemerer 1989) daher auch als logquadratisches Modell bezeichnet. Die geschätzten Parameter dieses Modell sind in den Untersuchungen der Autoren stark voneinander abhängig und instabil und können daher kaum interpretiert und zur Planung der Entwicklung verwendet werden.

Die Betrachtung der vier Arten der Kostenfunktion ist hier statistisch basiert. Die verschiedenen Autoren versuchen die Parameter dieser Funktionen so zu bestimmen, dass die Kosten bzw. der Aufwand der Softwareentwicklung für die, in den Untersuchungen verwendeten, Datensätze möglichst genau vorhergesagt wird. Dabei werden zunächst nur zwei Faktoren: Umfang und Zeit bzw. Aufwand betrachtet.

Barry Boehm verwendet in seiner Cocomo-II-Methode (vgl. Boehm 2000b) zur Kostenschätzung in der Softwareentwicklung zusätzlich zum Produktumfang spezifische Faktoren der SW-Entwicklung. Cocomo-II ist eine regressionsbasierte Methode, die im *Cocomo II.2000 Post-Architecture* Modell 22 Einflussfaktoren und den Softwareumfang nach der Function-Point Metrik verwendet. Die Kostenfunktion nach Cocomo-II ist in Gleichung 4-5 dargestellt und entspricht dem Cobb-Douglas-Modell aus Gleichung 4-3 bzw. dem loglinearen Modell nach (Banker, Kemerer 1989).

$$E = a_0 + a_1 \cdot EffortMultipliers \cdot S^{a_2 \cdot ScaleFactors} = a_0 + a_1 \cdot CocomoIIMetrik^{a_2}$$

Gleichung 4-5: Kostenfunktion nach Cocomo II, übertragen auf das Cobb-Douglas Modell

In Gleichung 4-5 sind die Parameter a_0, a_1, a_2 durch Regression zu schätzen. Die spezifischen Einflussfaktoren beeinflussen den Aufwand entweder proportional („*effort multipliers*“, wie z. B. die geforderte Zuverlässigkeit der Software) oder überproportional („*scale factors*“, wie z. B. die Prozessreife nach dem *Capability-Maturity-Modell* (*CMM*) aus (Paulk et al. 1993)).

Die Schätzung lässt sich nach (Boehm 2000b) unter Verwendung von bedingten Wahrscheinlichkeiten nach dem Bayes’schen Theorem (vgl. Bayes 1764) noch weiter kalibrieren, um die Probleme, die durch die (multiple) Regression entstehen zu verringern. Der Bayes’sche Ansatz erlaubt die Verwendung von Daten vergangener Projekte kombiniert mit a-priori-Wahrscheinlichkeiten aus Expertenurteilen zur besseren Gewichtung der Einflussfaktoren. So können a-posteriori-Wahrscheinlichkeiten der Einflussfaktoren berechnet werden, was zu signifikant besseren Ergebnissen führt (vgl. Chulani et al. 1999). Zu einer weiteren Verbesserung der Genauigkeit der Kostenschätzung gehört auch die adaptive Analyse der Einflussfaktoren (vgl. u. a. Hearty et al. 2009). Ebenso müssen geeignete Metriken ausgewählt werden, um die Werte der Einflussfaktoren zu ermitteln. Diese Verfeinerungen sind relevant für eine möglichst optimale Planung des SWE-Prozesses, haben aber keinen Einfluss auf die Untersuchungen dieser Arbeit.

Eine weitere Möglichkeit zur Modellierung von Kostenfunktionen sind *Neuronale Netze*, wie sie in der SWE-Literatur z. B. in Srinivasan, Fisher 1995 untersucht werden. Neuronale Netze können Kostenfunktionen aus mehreren gewichteten „Teilfunktionen“ berechnen. Ein Beispiel einer „Teilfunktion“ der Kosten o eines Knotens j eines neuronalen Netzes ist in Gleichung 4-6[107] dargestellt (vgl. Rumelhart et al. 1986). Auf der ersten Ebene berechnen sich die „Teilfunktionen“ z. B. aus gewichteten Einflussfaktoren, die in eine logistische Funktion einfließen.

$$o_{pj} = (1 + \exp[-\sum_i w_{ji} o_{pi} + \theta_j])^{-1}$$

Gleichung 4-6: "Teilkostenfunktion" eines neuronalen Netzes

[107] i ist hier der Index für den Vorgängerknoten und p der Index für den aktuellen „Layer“ des Netzes.

Ein Nachteil von Kostenfunktionen basierend auf neuronalen Netzen besteht in der größeren benötigten Datenmenge aus vergangenen Projekten gegenüber den anfangs dieses Abschnitts vorgestellten Kostenfunktionen, um einen akzeptablen mittleren Schätzfehler („*magnitude of relative error*") zu erreichen (vgl. Srinivasan, Fisher 1995).

Ein weiterer Nachteil besteht darin, dass sich durch die komplexen funktionalen Zusammenhänge, die in neuronalen Netzen abgebildet werden, der Einfluss einzelner Faktoren auf die Kosten nicht eindeutig herleiten lässt. Ein weiteres Problem in diesem Zusammenhang ist, dass die funktionale Form der Kostenfunktion – unabhängig von der genauen Ausprägung der Gewichte – mit der Wahl eines spezifischen neuronalen Netzes variiert. Somit sind Kostenfunktionen basierend auf neuronalen Netzen, für die vorliegende Arbeit, nicht zur Untersuchung einer Optimierung der Planung des Softwareentwicklungsprozesses geeignet.

Die Kostenfunktionen dieser Arbeit werden zunächst unter der Annahme des linearen Modells entwickelt und an verschiedenen Stellen werden die Auswirkungen von Skaleneffekten nach dem Cobb-Douglas Modell betrachtet.

4.2. Die Überarbeitungsdauer abhängig von Sensitivität und Unsicherheit

Die Überarbeitungsdauer hängt, wie in Kapitel 3 beschrieben, von der Unsicherheit und der Sensitivität komplexer Systeme ab. Der Überarbeitungsaufwand (engl.: *rework*) lässt sich aus dem Aufwand der Aufgaben, die bei einer Anforderungsänderung durchgeführt werden, herleiten. Der Ablauf dieser Aufgaben sieht auf Basis des „*Change-Mini-Cycle*"[108] wie folgt aus

1. Änderungsanfrage
2. Planung
    a. Programmverständnis
    b. Analyse der Änderungsauswirkungen
3. Implementierung der Änderung
    a. Restrukturierung

---

[108] Vgl. zu diesem Absatz Yau et al. 1978 zitiert nach Bennett, Rajlich 2000.

    b. Implementierung von Änderungsabhängigkeiten

4. Verifizierung und Validierung
5. Anpassung der Dokumentation

Der Zusammenhang des Überarbeitungsaufwands mit der *Anforderungsunsicherheit* ist hier in der Änderungsanfrage und der Zusammenhang mit der *Sensitivität* durch die Aufgaben Programmverständnis und Analyse der Änderungsabhängigkeiten zu erkennen.

Der Aufwand durch präventive architektonische Gestaltungsmaßnahmen wird hier nicht zum, durch Unsicherheiten erhöhten, Entwicklungsaufwand hinzugerechnet. Diese Maßnahmen werden, wie in Abschnitt 3.2.2 beschrieben, bei bekannten unsicheren Anforderungen getroffen und haben das Ziel, den erwarteten Überarbeitungsaufwand, bei Auflösung der Unsicherheit, zu reduzieren. Der Zusammenhang zwischen architektonischen und Überarbeitungsaufwand erscheint jedoch zu komplex (vgl. auch Bahsoon, Emmerich W. 2003), um in dieser Arbeit weiter verfolgt zu werden. Der präventive architektonische Aufwand wird hier dem nominalen Implementierungsaufwand hinzugerechnet.

Metriken zur Berechnung der Sensitivität

In (Boehm 2000b, S. 28) wird der Überarbeitungsaufwand auf Basis des „*maintenance adjustment factor*" (*MAF*) berechnet. Dieser, in Abschnitt 3.3 beschriebene, und mit der Sensitivität zu vergleichende Faktor berechnet sich aus dem Produkt der Faktoren „*programmer unfamiliarity*" (*UNFM*) und „*software understanding*" (*SU*), die vorher durch Experten zu bestimmen sind (vgl. Boehm 2000b, S. 23–24). In den Faktor SU fließt u. a. die Modularisierung des Softwaresystems ein.[109] Neben dem MAF wird in (Boehm 2000b, S. 332) der Faktor „*adaptation adjustment modifier*" (*AAM*) verwendet, der auch mit der hier dargestellten Sensitivität vergleichbar ist. Die Verwendung dieses Faktors im inkrementellen Cocomo-II-Modell wird in Abschnitt 4.5 kritisch betrachtet.

Das in Abschnitt 5.3 vorgestellte Modell, basierend auf (Loch, Terwiesch 1998), berücksichtigt auch einen solchen Faktor. Dieser Faktor wird dort – im Gegensatz zu den konstanten Faktoren MAF und AAM – zeitabhängig modelliert, um einen von der Entwicklungsdauer abhängigen Überarbeitungsaufwand abbilden zu können.

---

[109] Eine erhöhte Modularisierung kann wie in Abschnitt 3.3 angedeutet auch negative Auswirkungen auf das Softwareverständnis haben kann.

Die Sensitivität kann auch mit dem *Impact-Faktor* der NESMA-Methode zur Schätzung des Aufwands von SW-Weiterentwicklungsprojekten verglichen werden (vgl. NESMA 2009). Der Impact-Faktor wird für jede Daten- und Transaktionsfunktion, die durch eine Änderung beeinflusst wird bestimmt und als Grad der Veränderung dieser definiert. Der Überarbeitungsumfang („*enhancement function points*") ergibt sich dann aus dem Impact-Faktor multipliziert mit dem Umfang der Funktionen gemessen in Function-Points.

Eine weitere hier vorgestellte Idee ist es die Sensitivität (c) als „prozentualen Anteil unsicherer Schnittstellen am Produktumfang" nach Gleichung 4-7 zu berechnen. Daraus lässt sich zusammen mit dem Produktumfang der Überarbeitungsaufwand S_c an den Schnittstellen des aktuellen Systems berechnen.

$$c = q_M \frac{S_{I,M}}{S_M} \text{ und } S_c = cS$$

Gleichung 4-7: Beispielhafte Berechnung der Sensitivität und des Überarbeitungsumfangs an den Schnittstellen

In der Gleichung bezeichnet S_M den durchschnittlichen Modulumfang. $S_{I,M}$ bezeichnet den durchschnittlichen Umfang der Schnittstellen und der damit zusammenhängenden Funktionen eines Moduls. q_M ist ein prozentualer Faktor und beschreibt die Unsicherheit der Schnittstellen und somit die Risikokapselung durch die Architektur.

Die Berechnung des Überarbeitungsumfangs an den Schnittstellen gemäß Gleichung 4-7 ergibt $S_c = q_M S_{I,M} M$ (mit $S_M = S / M$ und M als Anzahl der Module). Daraus folgt ein erhöhter erwarteter Überarbeitungsaufwand bei

- einer erhöhten Unsicherheit der Schnittstellen
- einem erhöhte Schnittstellenumfang pro Modul
- einer erhöhten Anzahl von Modulen

Hierbei ist zu beachten, dass Modulumfang, Anzahl und Unsicherheit der Schnittstellen voneinander abhängig sein können.

Berechnung der erwarteten Überarbeitungsdauer

Bei unbekannten Anforderungsunsicherheiten kann zunächst keine konkrete Wahrscheinlichkeit für neue oder geänderte Anforderungen angegeben werden, aber durch Feedback und aus Erfahrungen vergangener Projekte lässt sich eine *erwartete*

Projektunsicherheit q bestimmen. So kann die Wahrscheinlichkeit anfänglich durch Expertenschätzung bestimmt und im Laufe der Zyklen angenähert werden.[110] Hier ist jedoch Vorsicht geboten, da z. B. agile Entwicklung selber zu einer erhöhten Unsicherheitswahrnehmung führen kann.[111] Diese Änderungswahrscheinlichkeit bezieht sich hier auf den gesamten Produktumfang S und sollte alle Faktoren, die in die Umfangsmessung einfließen, berücksichtigen. Das heißt die Änderungswahrscheinlichkeit sollte die mögliche Implementierung neuer Anforderungen, mögliche Änderungen in Eigenschaften bestehender Anforderungen und die Wahrscheinlichkeit für Änderungen durch Abhängigkeiten zwischen Anforderungen berücksichtigen.

Ein anderer Ansatz zur Bestimmung der Unsicherheit und eines Erwartungswertes wird im Meetingplan-Modell in Abschnitt 5.3.1 dargestellt.

Der erwartete Änderungsumfang berechnet sich gemäß der Modellannahmen dieser Arbeit aus dem Produkt aus Unsicherheit und Umfang ($S_q = qS$). Auf Basis dieses Änderungsumfangs wird ein zusätzlicher Arbeitsumfang durch die Integration dieser Änderungen in das bestehende System berechnet. Dieser von der Sensitivität abhängige zusätzliche Arbeitsumfang berechnet sich als $S_c = cS_q$. Der gesamte erwartete Überarbeitungsumfang durch noch unbekannte Anforderungen ergibt sich dann gemäß Gleichung 4-8.

$$S_{q,c} = qS + cqS$$

Gleichung 4-8: Erwarteter Arbeitsumfang durch nicht vorhersehbare Anforderungen

Die Überarbeitungsdauer ergibt sich aus diesem Überarbeitungsumfang und der Produktivität p_t und wird, unabhängig von der Prozessplanung, nach Gleichung 4-9 berechnet.

$$T_{q,c} = p_t(1 + c)qS$$

Gleichung 4-9: Prozessunabhängige Berechnung der gesamten Überarbeitungsdauer

---

[110] Zur adaptiven Analyse von Einflussfaktoren vgl. z. B. Kojima et al. 2008 und Hericko, Zivkovic 2008.

[111] Dies resultiert aus der verstärkten Wahrnehmung frustrierender Erlebnisse, wenn etwas wiederholt neuprogrammiert werden muss (vgl. Bratthall, Runeson 2000, S. 149).

Wird die Überarbeitungsdauer der einzelnen Zyklen betrachtet und die Prozessplanung berücksichtigt, so können feedback-abhängige Anforderungsunsicherheiten oder eine Verteilung der Anforderungen auf die Zyklen berücksichtigt werden. Durch die Funktion $f(q,S,i)$, in welche die aktuelle Zyklusnummer eingeht, können solche zyklusabhängigen Einflüsse berücksichtigt werden. Die Überarbeitungsdauer innerhalb des Zyklus Nummer *i* ergibt sich unter obigen Annahmen gemäß Gleichung 4-10.

$$T_{q,c}^{i} = p_t(1+c)\cdot f(q,S,i)$$

Gleichung 4-10: Überarbeitungsdauer innerhalb eines Zyklus

Auf die genaue Definition der Funktion *f* wird in den verschiedenen Modellen aus Kapitel 5 eingegangen. Durch diese Funktion kann ein feedback-abhängiger Änderungsumfang und/oder die Verteilung der Anforderungen auf die Releases modelliert werden. In letzterem Fall wird die Funktion so definiert, dass der gesamte Änderungsumfang unabhängig von der Anzahl der Zyklen ist. Des Weiteren ist zu beachten, dass, in dieser Arbeit, der durch die Sensitivität verursachte zusätzliche Arbeitsumfang eines Zyklus, abhängig von dem Änderungsumfang dieses Zyklus modelliert wird. Die Gründe für diese Modellierungsannahme werden in Abschnitt 4.5 diskutiert.

4.3. Bestimmung der Entwicklungszeit aus dem Entwicklungsaufwand

In den Kostenfunktionen dieser Arbeit wird, um eine Vergleichbarkeit zwischen verschiedenen Modellen zu erreichen, die Entwicklungszeit berechnet. Dazu kann, durch Betrachtung der Gesamtproduktivität, aus dem Entwicklungsaufwand die Entwicklungszeit ermittelt werden.

Das Verhältnis zwischen Produktumfang und Entwicklungsaufwand entspricht der Produktivität (gemessen z. B. in Personenmonaten pro Function-Point). Die Produktivität ist, im linearen Modell der Kostenfunktion aus Gleichung 4-1, unabhängig vom Produktumfang und wird durch den Faktor a_1 dargestellt. Dieser Faktor kann, wie beschrieben, durch Regression aus Vergangenheitsdaten geschätzt werden. In den anderen, dargestellten Kostenfunktionen ist die Produktivität vom Produktumfang abhängig und nicht eindeutig einem Faktor zuzuordnen.

Diese Produktivität, zur Bestimmung des Entwicklungsaufwands aus dem Umfang, bezieht sich auf die durchschnittliche Produktivität *eines* Entwicklers. Um die Entwicklungszeit zu berechnen, muss die *Gesamtproduktivität* (gemessen z. B. in Zeit pro Function-Point) bestimmt werden. Diese ergibt sich, im linearen Modell, aus der Produktivität eines Entwicklers dividiert durch die Anzahl der Entwickler. Alternativ kann bei Verwendung der anderen (komplexeren) Kostenfunktionen der Entwicklungsaufwand durch die Anzahl der Mitarbeiter dividiert werden, um die Entwicklungszeit zu bestimmen.

Aus diesen Betrachtungen folgt, dass der Zusammenhang zwischen Produktumfang und Zeit von organisatorischen Einflüssen, wie der Anzahl der Mitarbeiter und deren Produktivität abhängt. Dies wird auch durch die Untersuchung aus (Liu, Mintram 2005) mit Daten aus unterschiedlichen Unternehmen gestützt, in der keine (unternehmensunabhängige) signifikante Korrelation zwischen Produktumfang und Dauer nachgewiesen werden konnte. Dass der Zusammenhang zwischen Produktivität, Produktumfang und der Entwicklungszeit von organisatorischen Faktoren beeinflusst wird, zeigt z. B. Brooks 1975. Der Autor kommt zu dem Schluss, dass eine nachträgliche Erhöhung der Mitarbeiterzahl am Ende eines Projekts die Entwicklung nicht beschleunigt sondern u. a. wegen erhöhtem Einarbeitungsaufwand zu einer Verzögerung des Entwicklungsendes führen kann. Auch die Methode des *Pair-Programming* nach (Beck 2000) zeigt, dass sich die Gesamtproduktivität nicht alleine aus den summierten Function-Points pro Tag aller Entwickler ergibt. Der Einsatz dieser Methode bedeutet, dass sich zwei Entwickler einen Arbeitsplatz teilen: Ein Entwickler programmiert aktiv und der andere beobachtet und „denkt mit“. Diese Arbeitsweise, die zunächst den Anschein hat, Ressourcen zu verschwenden, soll nach (Beck 2000) die Produktivität und die Softwarequalität steigern.

Der allgemeine Arbeit von (Norden 1960) und der Erweiterung auf die SWE durch (Putnam 1978) folgend, ist nicht anzunehmen, dass der akkumulierte Aufwand *innerhalb* eines Projekts proportional zur Entwicklungszeit ansteigt. Dies resultiert aus empirischen Beobachtungen der Autoren, die zeigen, dass die aktuelle optimale Anzahl der Mitarbeiter während der Projektlaufzeit bis zu einem Maximum ansteigt und danach wieder sinkt.[112] Die aktuelle Anzahl der Mitarbeiter wird hier mit dem aktuellen Aufwand

[112] Hierbei ist zu beachten, dass ein Minimum an Mitarbeitern gebraucht wird und ein Maximum erreicht ist, wenn die Aufgaben nicht mehr teilbar sind.

gleichgesetzt und durch den Parameter $m(t)$ dargestellt. Der Verlauf von $m(t)$ wird durch das *Norden-Rayleigh-Modell* (Gleichung 4-11) dargestellt.

$$m(t) = 2E \cdot a \cdot t \cdot \exp(-at^2)$$

Gleichung 4-11: Norden-Rayleigh-Modell zur Verteilung des Aufwands/Mitarbeitereinsatzes über die Zeit

Abbildung 4-1 aus (Putnam 1978) zeigt exemplarisch den Verlauf der optimalen Anzahl der Mitarbeiter über die Zeit.

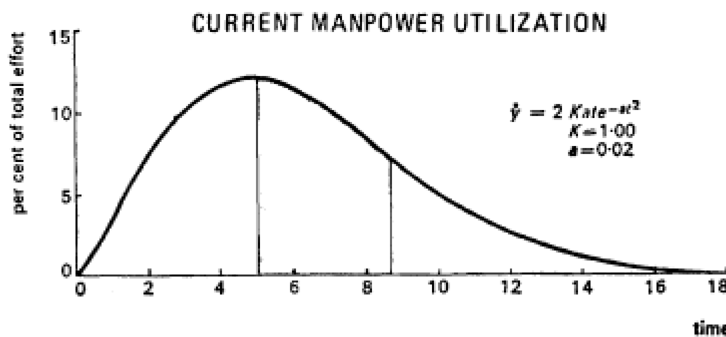


Abbildung 4-1: Exemplarische Verteilung des Aufwands/Mitarbeitereinsatzes über die Zeit

Die Fläche unter der Kurve entspricht dem Gesamtaufwand *E*.[113] Der Shape-Parameter a ist definiert als $a = 1/(2t_d^2)$. Der Parameter t_d spiegelt den Zeitpunkt wider, an dem das Maximum von *m(t)*, d. h. die maximale Anzahl der Mitarbeiter erreicht wird. Putnam zufolge zeigen empirische Untersuchungen, dass dieser Zeitpunkt, dem Punkt, ab dem das System einsatzbereit ist (dem Release), sehr nahe kommt. Der Aufwand nach diesem Zeitpunkt wird der Überarbeitung und Instandhaltung zugerechnet.

Ein Problem bei der Verwendung des Norden-Rayleigh-Modells entsteht nach (Koch 2008b) bei der Definition des Startzeitpunkts. Die Autoren sind dennoch beeindruckt von der Tatsache, dass dieses Modell, welches 1960 für kommerzielle Projekte entworfen wurde, eine gute Annäherung der Daten ihrer Untersuchung, aus dem Jahr 2008, zu Open-Source Projekten liefert. Die Autoren weisen darauf hin, dass dieses Modell die Grundlage für weitere Schätzmethoden, u. a. für die auch in dieser Arbeit vorgestellte Cocomo-II-Methodik liefert.

Bei der Planung der Mitarbeiter und der Verteilung der Entwicklungsaufgaben über die Zeit, ist zu beachten, dass Anforderungen, aufgrund von Abhängigkeiten untereinander,

---

[113] In der Abbildung durch *K* bezeichnet.

in einer bestimmten zeitlichen Reihenfolge bearbeitet werden müssen. Zusätzlich ist zu beachten, dass nicht jeder Mitarbeiter jede Aufgabe wahrnehmen kann und jeder Mitarbeiter eine andere Produktivität besitzt. Eine aus solchen Einschränkungen resultierende optimale Aufgabenverteilung wird unter dem Thema *Releaseplanung* in (Ngo-The, Ruhe 2009) untersucht und ist den Autoren zufolge besonders für die Planung inkrementeller Entwicklungsprozesse wichtig.

4.4. Skaleneffekte in der Softwareentwicklung

Die in Abschnitt 4.1 vorgestellten Kostenfunktionen unterscheiden sich unter anderem in der Modellierung von Skaleneffekten. *Negative Skaleneffekte* bedeuten überproportional zum Produktumfang wachsenden Aufwand, *positive Skaleneffekte* überproportional sinkenden Aufwand („*dis-/economies of scale*“, vgl. Banker, Kemerer 1989; Kitchenham 2002).[114] Während im linearen Modell keine Skaleneffekte angenommen werden, werden im quadratischen Modell negative Skaleneffekte vorausgesetzt. Mit dem Parameter a_2 im Cobb-Douglas-Modell lassen sich keine ($a_2 = 0$), negative ($a_2 > 0$) oder positive ($a_2 < 0$) Skaleneffekte darstellen. Das Translog-Modell (Gleichung 4-4) verwendet einen vom Produktumfang abhängigen Skalenparameter.

4.4.1. Positive, negative oder keine Skaleneffekte?

Projektbezogen existieren in der Literatur unterschiedliche Meinungen zum, möglicherweise exponentiellen, Zusammenhang zwischen aktuellem Produktumfang und Überarbeitungsaufwand bei Änderungen an einem bestehenden Produkt. In (Loch, Terwiesch 1998) gehen die Autoren davon aus, dass die Produktivität im Projektverlauf sinkt, da die Sensitivität[115] proportional mit dem Produktumfang (bzw. dort mit der Zeit) ansteigt, was zu einem überproportional ansteigenden Überarbeitungsaufwand führt. Dies entspricht auch der traditionellen Auffassung in der Softwareentwicklung.[116] (Beck 1999) dagegen geht davon aus, dass in der Softwareentwicklung die Sensitivität unabhängig vom Produktumfang ist und liefert damit die Begründung für den geringen Architekturaufwand in der Startphase seiner agilen Methodik.

---

[114] Positive Skaleneffekte werden auch in verschiedenen Zusammenhängen als Synergie-Effekte bezeichnet.

[115] Entspricht der Steigung der Überarbeitungsaufwandkurve

[116] Vgl. Boehm 1976. Die Ergebnisse der empirischen Untersuchung in Boehm 1976 zeigen, dass die relativen Kosten der Fehlerbehebung mit der Zeit ansteigen.

Dieser Unterschied kann anhand des Vergleichs des Translog-Modells (Gleichung 4-4) mit dem linearen Modell dargestellt werden. Das lineare Modell kann mit den Annahmen aus (Beck 2000) begründet werden und das Translog-Modell mit den Annahmen aus (Loch, Terwiesch 1998). In Abbildung 4-2 ist anhand der durchgezogenen Linie der Verlauf der Kostenfunktion im Translog-Modell mit $a_0 = 0, a_1 = 0, a_2 = 1$ zu sehen. Es ist zu erkennen, dass unter Verwendung dieser Modellannahmen in kleinen Projekten positive Skaleneffekte überwiegen. Ab einer bestimmten Projektgröße werden die positiven Skaleneffekte von den negativen dominiert werden. Dieser Punkt wird in Abbildung 4-2 bei *E=e* (ca. 2.6) erreicht, wenn die Kostenfunktion des Translog Modells (durchgezogene Linie) die Kostenfunktion des linearen Modells mit $a_0 = 0, a_1 = 1$ (gestrichelte Linie) schneidet.

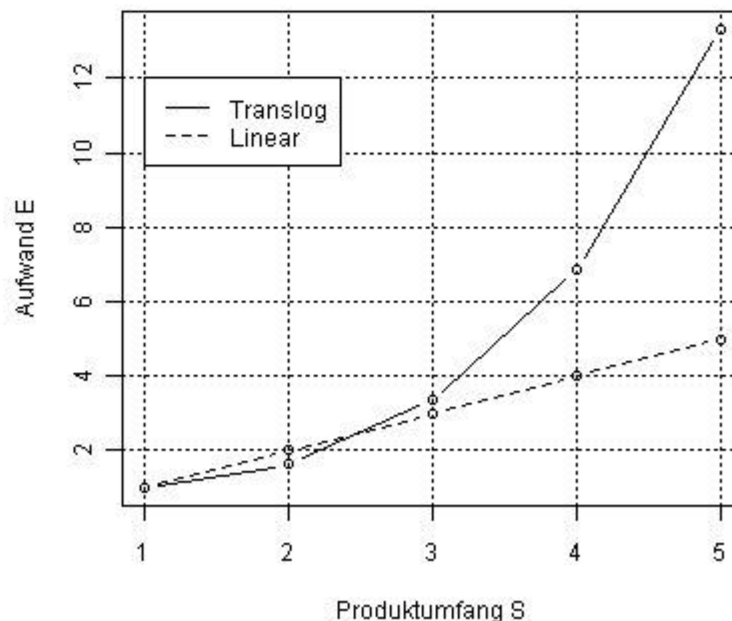


Abbildung 4-2: Vergleich des Verlaufs der Kostenfunktion im Translog und im linearen Modell

Es stellt sich die Frage,

- ob sich die positiven und negativen Skaleneffekte gegenseitig aufheben. Zu diesem Schluss kommen (Kitchenham 1992)[117] und (Dolado 2001). Diese Annahme ist Voraussetzung für das lineare Modell (Gleichung 4-1).

[117] Zitiert nach Hu 1997.

- ob negative oder positive Effekte überwiegen bzw. dominieren. Zu diesem Schluss kommen (Hu 1997) und (Pickard et al. 1999). Diese Annahme ist Voraussetzung für Gleichung 4-2 und Gleichung 4-3.
- oder ob – wie in (Banker, Kemerer 1989) vermutet – kleine Projekte Größenvorteile haben, während große Projekte Größennachteile haben. Diese Annahme kann durch das Translog-Modell unterstützt werden.

Die Untersuchungen in (Premraj et al. 2005) aus ca. 600 Projekten ergeben keine Skaleneffekte für Neuentwicklungen, aber Größenvorteile für Wartungsprojekte. In dieser Untersuchung zeigt sich, dass die Produktivität im Laufe der letzten Jahrzehnte angestiegen ist. In (Kitchenham 2002) werden die verschiedenen Artikel über Skaleneffekte untersucht und die Autorin kommt zu dem Schluss, dass die widersprüchlichen Aussagen in der Literatur, in unterschiedlichen Annahmen begründet sind und schlägt bestimmte Standards für weitere Untersuchungen dieser Art vor. Auch in (Dolado 2001) werden diese Widersprüche aufgedeckt und eine besseres, auf die Besonderheiten der Softwareentwicklung bezogenes, Modell als Grundlage für weitere Untersuchungen gefordert.

Es folgt eine Übersicht über die, in der Literatur gefundenen, Ursachen von Skaleneffekten. Für die meisten Faktoren liefern die Autoren keine Belege zur Einordnung als positiver oder negativer Skalenfaktor. Es wird in Tabelle 4-1 nur eine Auswahl der in der Literatur gefundenen Faktoren dargestellt, um diese kurz zu diskutieren und den verschiedenen, in dieser Arbeit verwendeten, Einflussfaktoren zuzuordnen.

Tabelle 4-1: Ursachen von Skaleneffekten und Modellzuordnung

Skalen-faktor	*Ursachen für Skaleneffekte*	*Zuordnung*
sf_1	Lerneffekte[118]	Einfluss auf Produktivität
sf_2	Automatisierung und Verwendung von Softwareentwicklungstools[119]	Einfluss auf Produktivität
sf_3	Anforderungen an die organisationsinterne und externe Zusammenarbeit[120], Anzahl und Ausprägung der Kommunikationswege zwischen Teammitglieder (bei wachsenden Teams)[121]	Einfluss auf Produktivität
sf_4	Aufwand in der Analyseaktivität (Anforderungsanalyse und Architekturdesign) während der Startphase[122]	Einfluss auf die Unsicherheit
sf_5	Technische und Anforderungsunsicherheit, sowie daraus resultierende Risiken[123]	Einfluss auf den Überarbeitungsaufwand
sf_6	Komplexe Schnittstellen zwischen Modulen[124]	Einfluss auf Sensitivität

Lerneffekte (sf_1) bedeuten, dass durch vertieftes Verständnis einer Aufgabe im Laufe der Entwicklung die Produktivität steigt. Zusammen mit der Annahme, dass größere Projekte länger dauern, folgt daraus ein positiver Skaleneffekt. Diese Lerneffekte sind solange zu beobachten, bis das autodidaktische Lernen seine Grenzen erreicht hat. Um weiteren Lernerfolg zu erzielen, können dann Schulungen durchgeführt werden.

---

[118] Vgl. Banker, Kemerer 1989, sowie die Faktoren *PREC* und *PMAT* aus Boehm 2000b.

[119] Vgl. Boehm 1984.

[120] Vgl. die Faktoren *TEAM* und *PMAT* aus Boehm 2000b.

[121] Vgl. Brooks 1975.

[122] Vgl. die Faktoren *RESL* und *PMAT* aus Boehm 2000b, sowie Jones 1986 zitiert nach Banker, Kemerer 1989.

[123] Vgl. die Faktoren *RESL*, *FLEX*, *PREC*, *TEAM*, *PMAT* aus Boehm 2000b.

[124] Vgl. Conte et al. 1986 zitiert nach Banker, Kemerer 1989, sowie den Faktor *SU* aus Boehm 2000b, der dort allerdings nicht den Skalenfaktoren zugeordnet wird.

Die Verwendung von Softwareentwicklungstools (sf_2) hat ebenso einen positiven Einfluss auf die Produktivität. Der Vorteil dieser Tools besteht in der Automatisierung bestimmter Aufgaben. Zusätzlich dienen diese Tools z. B. der Komplexitätsanalyse und Restrukturierung, d. h. sie helfen dabei negative Skaleneffekte zu vermindern. In der Cocomo-II-Methodik fließt die Verwendung von Tools im Gegensatz zu dem älteren Modell aus (Boehm 1984) nicht mehr als Skalenfaktor, sondern „nur noch" als multiplikativer Faktor ein.

Alle weiteren Skalenfaktoren, die einen Einfluss auf die Produktivität haben, werden hier in sf_3 zusammengefasst. sf_3 ist ein negativer Skalenfaktor unter der Annahme, dass größere Projekte von größeren Teams bewältigt werden und die Produktivität durch die komplexere Kommunikation (vgl. Abschnitt 3.3.3) überproportional stark abfällt.

Zwischen diesen Skalenfaktoren können auch Zusammenhänge existieren. So entsteht z. B. durch die oben genannten Lerneffekte spezialisiertes Personal, was möglicherweise eine verstärkte Kommunikation dieses Spezialwissens innerhalb des Teams zur Folge hat.

Werden diese Skalenfaktoren normiert und als unabhängig voneinander betrachtet, dann kann die Gesamtproduktivität durch $p_t^{sf_1+sf_2-sf_3}$ berechnet werden.

Der Aufwand in der Analyseaktivität während der Startphase (sf_4) kann einen Einfluss auf die Unsicherheit und damit dem Überarbeitungsaufwand haben. Werden – wie in dieser Arbeit beschrieben – Abhängigkeiten zwischen unsicheren Anforderungen angenommen und diese nicht durch Feedback geklärt, so kann der Überarbeitungsaufwand überproportional mit dem Umfang unsicherer Anforderungen steigen. Der Faktor sf_5 bezieht sich direkt auf die Unsicherheit und den damit verbundenen Überarbeitungsaufwand.

Komplexe Abhängigkeiten zwischen Modulen (sf_6) haben einen Einfluss auf die Sensitivität und sind ein weiterer Skalenfaktor. Die Komplexität der Modulstruktur – wie sie in Abschnitt 4.4.2 beschrieben wird – steht in engem Zusammenhang mit der Komplexität auf organisatorischer Ebene (vgl. Abschnitt 3.3.3) und der Komplexität der Anforderungen (vgl. Abschnitt 3.3.1). Im Cobb-Douglas-Modell aus Gleichung 4-3 kann der Skalenfaktor a_2 mit sf_6 gleichgesetzt werden. In (Boehm 2000b) ergibt sich a_2 aus

der Summe aller positiver und negativer Skalenfaktoren. Dieser Skalenfaktor soll daher in Abschnitt 4.4.2 näher untersucht werden.

4.4.2. Skaleneffekte durch komplexe Modulstrukturen

Zur Bestimmung des Skalenfaktors sf_6, welcher die Komplexität der Modulstruktur beschreibt wird in (Jones 1998) die Ein- und Ausgangsverzweigung (engl.: *fan-in, fan-out*) von Modulen und die Graph-Komplexität[125] auf Modulebene betrachtet.

Bezogen auf die Graphentheorie entsprechen die Module den Knoten und die Verbindungen bzw. Schnittstellen zwischen den Modulen den Kanten. Angenommen es gibt M Module, dann ist die Anzahl der Schnittstellen proportional zu $(M-1)M/2$. Dieser Zusammenhang wird auch in (Lehman, Ramil 2001b) betrachtet. Die Autoren sind der Auffassung, dass "die Anzahl möglicher Abhängigkeiten und Schnittstellen zwischen Elementen (Objekten, Modulen, Holonen, Subsystemen, etc.) proportional zum Quadrat der Anzahl [*M*] der Elemente ist.“ Sie folgern daraus, dass die Sensitivität im Laufe der Entwicklung proportional zum Quadrat des Systemumfangs *M* ansteigt: „… dann wächst der Arbeitsaufwand, um korrekte und adäquate Schnittstellen zwischen dem Neuen und dem Alten sicherzustellen, die Wahrscheinlichkeit für Fehler, Unterlassungen und Inkompatibilitäten zwischen Annahmen wie [M^2]“ (vgl. Lehman, Ramil 2001b). In einer ähnlichen Rechnung aus (Gerlich, Denskat 1994) ist die Anzahl der Schnittstellen, die bei einer Änderung überprüft werden müssen, wenn k von M Modulen geändert wurden, proportional zu $k(M-k)+k(k-1)/2$.[126]

Allerdings kann nach (Boehm 2000b, S. 22) eine hierarchische Strukturierung der Module die Anzahl der Schnittstellen, die bei einer Änderung überprüft werden müssen, reduzieren. Eine hierarchische Strukturierung hat zur Folge, dass – im Idealfall – Schnittstellen zwischen Modulen nur innerhalb der Hierarchie bestehen. Solche Hierarchien ergeben sich in der Softwareentwicklung durch das Vererbungsprinzip und der Definition von Ober- und Unterklassen, sowie durch die Verwendung einer strikten Schichtenarchitektur (vgl. Bass et al. 2005, S. 37). Des Weiteren ist zu beachten, dass bei konstantem Gesamtumfang und steigender Anzahl von Modulen, z. B. durch eine verfeinerte Hierarchie, der Umfang eines Moduls abnimmt.

---

[125] Die Graph-Komplexität ist auf Codeebene Teil der zyklomatischen Komplexität nach McCabe 1976.

Der Nutzen einer verfeinerten hierarchischen Strukturierung wird durch eine erhöhte Lerndauer reduziert, die daraus resultiert, dass alle Zusammenhänge in einem Submodul verstanden werden müssen.[127] In diesem Zusammenhang kann die Gesamtlänge aller Pfade von Modulen auf unterster Hierarchieebene bis zum obersten Modul als Maß dienen. Diese Gesamtlänge wächst exponentiell mit der Tiefe h der Hierarchie. Bei durchschnittlich k Submodulen eines Moduls ist die Gesamtlänge aller Pfade auf der untersten Hierarchieebene proportional zu hk^h.[128]

In Abbildung 4-3 werden ein hochabhängiges und ein hierarchisches System gegenübergestellt. In dem hochabhängigen System besteht zwischen allen Modulen eine direkte Verbindung woraus eine höhere Anzahl von Verbindungen resultiert. Im hierarchischen System besteht nur zwischen Ober- und Untermodulen eine Verbindung, wodurch allerdings länger Pfade zu weiter oben bzw. unten liegenden Modulen resultieren.

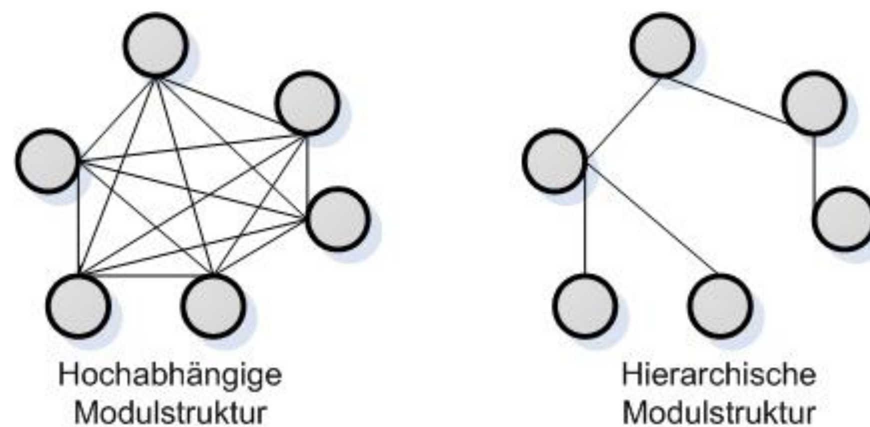


Abbildung 4-3: Vergleich einer hochabhängigen mit einer hierarchischen Modulstruktur.

Eine hochabhängige Modulstruktur wird auch als „*alien spider anti-pattern*“ (vgl. Kiefer et al. 2007) bezeichnet. Die Bezeichnung *Anti-Pattern* verdeutlicht den Gegensatz zur architektonischen Gestaltung durch *Design-Pattern*.

Aus diesen Beobachtungen lässt sich folgern, dass die, von einigen Autoren getroffene, Annahme einer mit dem Produktumfang steigenden Sensitivität nicht allgemeingültig ist.

---

[126] Vgl. Gerlich, Denskat 1994 zitiert nach Boehm 2000b, S. 21.

[127] Vgl. „*chunks*” und „*tracing”* in Cant et al. 1995.

[128] Die Herleitung dieser Formel ist nicht Kern dieser Arbeit und wird daher hier nicht explizit gegeben.

Durch hierarchische Systeme lassen sich die komplexen Abhängigkeiten reduzieren. Es bleibt die offene Frage, welchen Einfluss die negativen Folgen stark hierarchischer Systeme auf die Sensitivität des Systems haben.

4.5. Erweiterung der Kostenfunktionen durch die Modellierung von Zyklen

4.5.1. Modellierung des Überarbeitungsaufwand in den Zyklen

Die bisher vorgestellten Kostenfunktionen (vgl. Abschnitt 4.1) berücksichtigen nicht die Struktur des Softwareentwicklungsprozesses. Um die in dieser Arbeit beschriebenen Zyklen eines inkrementellen Entwicklungsprozesses in einer Kostenfunktion abzubilden, gibt es verschiedene Modellierungsansätze.

Eine Möglichkeit besteht darin die Kosten der einzelnen Zyklen, wie in (Benediktsson et al. 2003), zu summieren. Die Autoren verwenden eine Kostenfunktion, die, übertragen in die Modellierung dieser Arbeit, in Gleichung 4-12 dargestellt wird. T_S bezeichnet die Dauer der Startphase und $S_i = S / n$ den Produktumfang in Zyklus *i* bei *n* Zyklen.

$$T = T_S + \sum_{i=1}^{n} p_t (qS / n + S / n)^c$$

Gleichung 4-12: Kostenfunktion nach dem inkrementellen Modell aus (Benediktsson et al. 2003)

Der Arbeitsumfang eines Zyklus ergibt sich aus dem zu realisierenden Produktumfang des Zyklus und dem Überarbeitungsumfang des vorangehenden Zyklus. Beide werden addiert und mit dem Sensitivitätsfaktor *c* exponiert. In dieser Arbeit wird – wie in Abschnitt 4.3 beschrieben – zur Vergleichbarkeit zwischen den Modellen mit der Entwicklungszeit gerechnet und daher in Gleichung 4-12 die Gesamtproduktivität anstelle der Produktivität des ursprünglichen Modell von (Benediktsson et al. 2003) verwendet.

Aus der obigen Kostenfunktion leiten die Autoren eine Messgröße zum Vergleich des Entwicklungsaufwands zwischen Wasserfallmodell ($i=1$) und mehrzyklischen Entwicklungsprozessen her. Die Autoren unterscheiden in ihrem Modell nach Startphase, in der u. a. die Architektur entwickelt wird und Konstruktionsphase. Diese Phasen beeinflussen sich, wie z. B. auch in (Loch, Terwiesch 1998) angenommen, gegenseitig.

Gemäß (Benediktsson et al. 2003) ergibt sich die Entwicklungsdauer der Konstruktionsphase aus der linken Seite von Un-Gleichung 4-13 (mit $S_{q,i} = (1+q)S/n$). Auf der rechten Seite dieser Un-Gleichung ist die Entwicklungsdauer beim Exponieren des Gesamtumfangs – anstelle des Umfangs eines Zyklus – mit dem Sensitivitätsfaktor c zum Vergleich dargestellt.

$$p_t \sum_{i=1}^{n} S_{q,i}{}^{c} = p_t n S_{q,i}{}^{c} \leq p_t \left(\sum_{i=1}^{n} S_{q,i} \right)^{c} = p_t n^{c-1} n S_{q,i}{}^{c}$$

Un-Gleichung 4-13: Modellierung von Skaleneffekten unter der Berücksichtigung von Zyklen

Aus Un-Gleichung 4-13 ist zu erkennen, dass die Autoren in ihrem Modell annehmen, dass bei einer mehrzyklischen Entwicklung und negativen Skaleneffekten der Aufwand in der Konstruktionsphase um den Faktor n^{c-1} reduziert wird. Die Autoren erklären dies (implizit) mit der Annahme, dass der Architekturaufwand in der Startphase die Abhängigkeiten zwischen den Inkrementen – und damit des Faktors c – reduziert. Da der Architekturaufwand „nur“ linear mit der Anzahl der Zyklen ansteigt (vgl. Benediktsson et al. 2003, S. 272) wird dieser – bei negativen Skaleneffekten – durch Reduzierung des Aufwands der Konstruktionsphase in den meisten Simulationen der Autoren überkompensiert. Daraus folgt, dass der Nutzen der mehrzyklischen Entwicklung in der Konstruktionsphase aus einer Reduzierung der Komplexität durch erhöhten Architekturaufwand in der Startphase entsteht.

Die Annahme, dass der Architekturaufwand und damit die Komplexität von der Anzahl der Zyklen abhängen, ist nicht plausibel. Vorteilhafte architektonische Gestaltungsmaßnahmen sollten auch in ein-zyklischen Prozessen durchgeführt werden. Aus diesem Grund wird der Modellierungsansatz nach (Benediktsson et al. 2003) in dieser Arbeit nicht angewendet.

In (Boehm 2000b) wird ein weiteres Modell zur Bestimmung der Kosten der inkrementellen Entwicklung aufgestellt. Übertragen in die Notation dieser Arbeit wird die Entwicklungs- und Überarbeitungsdauer eines Zyklus durch Gleichung 4-14 bestimmt.

$$T_i = p_t \left(S_i + q(1+c) \cdot \sum_{j=1}^{i} S_j \right)^{b}$$

Gleichung 4-14: Kostenfunktion für einen Zyklus nach dem inkrementellen Modell aus Boehm

In dem Modell aus (Boehm 2000b, S. 332) wird der Überarbeitungsaufwand durch den Faktor „adaptation adjustment modifier" (*AAM*) beeinflusst, der in Gleichung 4-14 durch $q(1+c)$ ersetzt wird. Dieser Faktor wird in jedem Zyklus mit dem Umfang des bestehenden Systems $\sum_{j=1}^{i} S_j$ multipliziert und das Ergebnis zum Entwicklungsumfang des aktuellen Zyklus S_i addiert. Der daraus resultierende Umfang wird mit einem Skalenfaktor (b) exponiert und zur Berechnung der Entwicklungszeit mit der Gesamtproduktivität multipliziert.

Aus dieser Kostenfunktion folgt, dass der gesamte Überarbeitungsumfang mit der Anzahl der Zyklen um den Faktor $(n+1)/2$ steigt, da in jedem Zyklus das gesamte bestehende System überarbeitet werden muss. Die Berechnung des Überarbeitungsumfangs ist ohne Berücksichtigung von Skaleneffekten ($b=1$) und für gleichmäßig verteilte Zyklusumfänge $S_i = S/n$ in Gleichung 4-15 dargestellt.

$$\sum_{i=1}^{n} S_{q,c} = \sum_{i=1}^{n} q(1+c) \sum_{j=1}^{i} S_j = q(1+c)S(n+1)/2$$

Gleichung 4-15: Der Überarbeitungsumfang im inkrementellen Modell nach (Boehm 2000b)

Der gesamte Überarbeitungsumfang wird jedoch, wie in Abschnitt 4.2 beschrieben und ohne Berücksichtigung von Feedback-Effekten, von der Unsicherheit, der Sensitivität und dem Gesamtumfang des Systems beeinflusst und nicht von der Anzahl der Zyklen. Das Modell aus (Boehm 2000b) beruht auf der Annahme, dass einige Schnittstellen des bestehenden System in jedem Zyklus überarbeitet werden müssen. Dies kann z. B. bei qualitativ schlechtem und wechselndem Feedback aus der Nutzung, mangelhafter Analyse des Feedbacks oder in einem hochabhängigen System der Fall sein.

Hier widersprechen sich die Modelle aus (Benediktsson et al. 2003) und (Boehm 2000b, S. 332). Nach (Benediktsson et al. 2003) hat im Gegensatz zu (Boehm 2000b) eine höhere Anzahl von Zyklen kleinere Inkremente und eine geringere Komplexität zur Folge. Eine höhere Anzahl von Zyklen kann nach (Benediktsson et al. 2003) sogar im Gegenteil zu (Boehm 2000b, S. 332) einen geringeren gesamten Überarbeitungsaufwand zur Folge haben. Im Releaseplan-Modell dieser Arbeit wird im Gegensatz zu (Boehm 2000b) davon ausgegangen, dass die Inkremente nach einmaligem Feedback die Anforderungen erfüllen und nicht mehr verändert werden müssen.

In (Loch, Terwiesch 1998) wird eine zu (Boehm 2000b, S. 332) ähnliche Kostenfunktion aufgestellt. Ein Unterschied besteht darin, dass in (Loch, Terwiesch 1998) ein Integral über die Zyklusdauer anstelle der Summe über die Zyklusumfänge zur Modellierung der Zyklen verwendet wird. Unter der Annahme zeitunabhängiger Zyklusumfänge $S(t) = S/n$ und konstantem Verlauf des Änderungsumfangs (d. h. *e=0*) im Modell von (Loch, Terwiesch 1998) sowie gleichmäßig verteilten Zyklusumfängen und ohne Skaleneffekte (d. h. $S_i = S/n$ und *b=1*) im Modell von (Boehm 2000b, S. 332) wird die Berechnung der Überarbeitungsdauer $T_{q,c}$ beider Modelle in Gleichung 4-16 gegenübergestellt.

$$\int_{t=0}^{n} p_t \cdot (1+c) \cdot t \cdot q \cdot S(t)dt \approx \sum_{i=1}^{n} p_t \cdot (1+c) \cdot i \cdot q \cdot S_i$$

Gleichung 4-16: Vergleich von Integral- und Summenfunktion zur Berechnung der Überarbeitungsdauer

Das diskrete Modell, welches den Aufwand in den Zyklen summiert, liefert hier eine Approximation für die Integralbildung[129] über die Zeit, unter der Annahme, dass ein Zeitintervall *t* der Dauer eines Zyklus *i* entspricht.

Zur Modellierung von Feedback-Effekten und der Verteilung des zu entwickelnden Umfangs über die Zyklen wird im Rahmen des Releaseplan-Modells eine umfangsbasierte Summenfunktion verwendet. Zur Modellierung von Kommunikationsverzögerungen und eines über die Zeit wachsenden Systems wird im Rahmen des Meetingplan-Modells eine zeitbasierte Integralfunktion eingesetzt. Eine detaillierte Begründung hierfür wird in Abschnitt 5.3.1 geliefert.

4.5.2. Modellierung der Fixkosten der Zyklen

Eine Untersuchung aus (Basili et al. 1996) zeigt, dass, in dem von den Autoren untersuchten Projekt, jeder Release einer Weiterentwicklung gewisse *Fixkosten* verursacht, d. h. Kosten, die unabhängig vom Umfang der Änderungen sind. Die erhobenen Daten der Untersuchung werden von (Basili et al. 1996) in Abbildung 4-4 zusammen mit einer Regressionsgerade dargestellt. Die Regressionsgerade schneidet die Y-Achse bei einem Aufwand von 1040 Stunden, was die Autoren als durchschnittliche

[129] In diesem Fall entspricht das diskrete Modell (d.h. die Summenfunktion) der Obersumme des Integrals. Das Integral selber entspricht dem arithmetischen Mittel von Unter- und Obersumme (*Riemann-Integral*).

Fixkosten eines „*enhancement release*“ in den untersuchten Projekten interpretieren. Die Autoren bezeichnen es als wahrscheinlich, dass diese Fixkosten aus zyklischen Systemtests („*regression testing*“) und Einarbeitungszeiten entstehen, welche ihrer Meinung nach unabhängig von der Systemgröße sind.

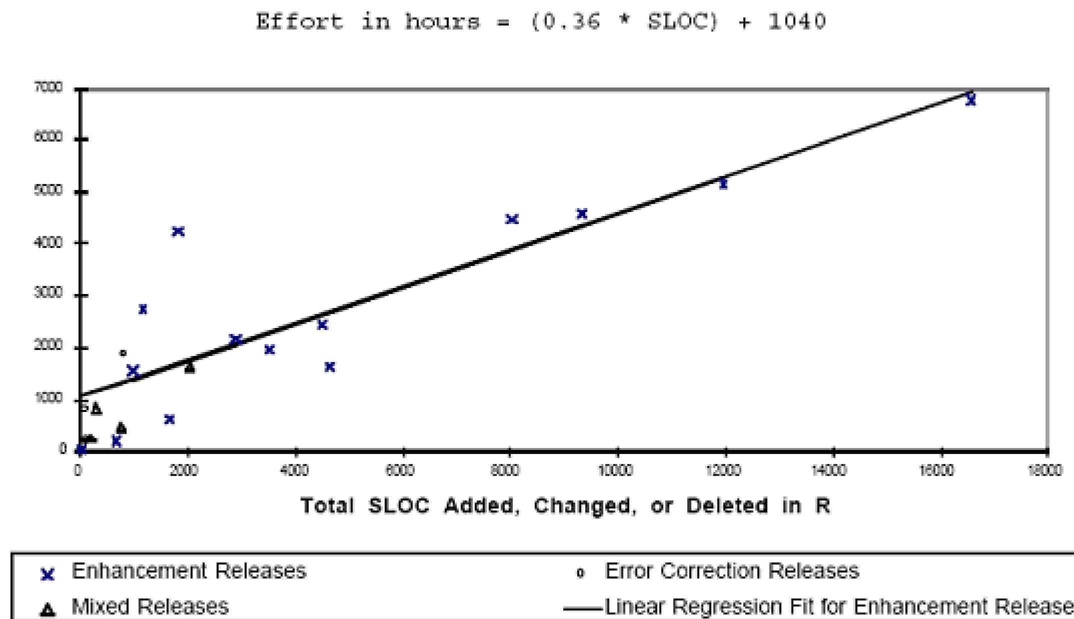


Abbildung 4-4: Lineare Regressionsgerade für den Aufwand in „enhancement releases“ nach (Basili et al. 1996)

Wie in Abschnitt 3.1.3 beschrieben ergeben sich die Fixkosten pro Zyklus zum Teil aus dem fixen Aufwand der Einführungsaktivität, z. B. durch die Anzahl der Installationen der Software. Es ist jedoch nicht anzunehmen, dass der gesamte Einführungsaufwand unabhängig von der Dauer und dem Umfang eines Zyklus ist. Z. B. erscheint es plausibel, dass, bei kürzeren Zyklen und damit voraussichtlich einer geringeren Anzahl neuer Features, der Schulungsaufwand für ein neues Release, gegenüber längeren Zyklen und einer erhöhten Zahl von Features, sinkt. Aber auch in den anderen Aktivitäten entstehen Fixkosten, z. B. durch den Kommunikationsaufwand an Meilensteinen beim Wechsel von der Analyse zur Implementierung, sowie durch Reports (vgl. Banker, Kemerer 1989).

Auch in (Mathiassen, Pedersen 2008, S. 491) werden erhöhte Kosten durch die zyklische Entwicklung beobachtet. In einer von den Autoren durchgeführten Fallstudie betrug der gesamte Aufwand für die Planung und die Durchführung von Akzeptanztests in der mehrzyklischen Entwicklung ca. 800 Stunden gegenüber einem Aufwand von 400-500 Stunden für den Abschluss in nicht-zyklischen Projekten.

Diese Betrachtungsweise der Fixkosten aus Sicht des Entwicklungsaufwands darf hier nicht mit dem in der Betriebswirtschaftslehre verwendeten Begriff der Fixkosten verwechselt werden.[130] Dort wird der gesamte Entwicklungsaufwand den Fixkosten zugerechnet, da dieser unabhängig von der Anzahl der produzierten Exemplare – hier der Anzahl der Installationen – ist. In Modellen der Betriebswirtschaftslehre wird der Vorteil einer erhöhten Anzahl von Exemplaren / Installationen darin gesehen, dass der Entwicklungsaufwand sich auf diese verteilt, was bei einer erhöhten Anzahl von Installationen die Stückkosten senkt und den Gewinn hebt.

In dieser Arbeit werden die Fixkosten dagegen, als die vom Produktumfang unabhängige Entwicklungsdauer betrachtet. Die Fixkosten pro Zyklus werden in den Modellen dieser Arbeit durch t_f gekennzeichnet und die Fixkosten von n Zyklen durch nt_f berechnet.

Die in diesem Kapitel durchgeführten Betrachtungen verschiedener Modellierungsansätze für Kostenfunktionen finden Eingang in die, im nächsten Kapitel betrachteten, Modelle zur Optimierung der Planung von Entwicklungsprozessen.

[130] Vgl. zu diesem Absatz z. B. Jung 2006, S. 1114.

Kapitel 5: Optimierung der Kosten unter Berücksichtigung von Feedbackeffekten

In diesem Kapitel werden zwei Modelle mit zeitbasierten Kostenfunktionen zur Optimierung der Planung von Prozessen mit zyklischem Nutzerfeedback betrachtet.

- Das *Releaseplan-Modell*: Ein Modell, welches der Optimierung der Planung der Entwicklung mehrerer Releasezyklen auf Ebene einer SW-Generation dient.
- Das *Meetingplan-Modell*: Ein Modell, welches der Optimierung der Planung von Meetings zur Steuerung der Implementierungsaktivität dient.
- Das *Prototypingplan-Modell*: Ein Modell basierend auf dem Meetingplan-Modell, welches der Optimierung der Planung der Entwicklung von Prototypen dient.

Die Kostenfunktion des Meetingplan-Modells kann, wie in Gleichung 4-16 gezeigt, mit der Kostenfunktion des Releaseplan-Modells verglichen werden.

Zunächst werden Annahmen aus allgemeine Modellen der Produktentwicklung vorgestellt, die z. T. auf die hier vorgestellten Modelle der Softwareentwicklung übertragen werden. Das Modell aus (Loch, Terwiesch 1998), welches für Produkte im Allgemeinen formuliert wurde, bildet, unter Berücksichtigung der, in den vorangegangenen Kapiteln beschriebenen, Besonderheiten der SW-Entwicklung, die Grundlage für das Meetingplan-Modell.

5.1. Allgemeine Modelle der Produktentwicklung

5.1.1. Vergleich der verschiedenen Modelle

Die hier vorgestellten Modelle beziehen sich auf die Forschung verschiedener Autoren (u. a. Loch, Terwiesch 1998; Krishnan, Eppinger 1997; Ha, Porteus 1995) zum New-Product-Development und zum Concurrent-Engineering – wie in Abschnitt 2.6 kurz beschrieben. Innerhalb dieses Forschungsbereichs werden zwei Entwicklungsaktivitäten betrachtet:

- eine informationsliefernde Aktivität („*upstream*” Krishnan, Eppinger 1997 bzw. „*product design*“ Ha, Porteus 1995)

- eine informationsverarbeitende Aktivität („*downstream*" Krishnan, Eppinger 1997 bzw. „*process design*" Ha, Porteus 1995)

Die informationsliefernde Aktivität kann mit der, in dieser Arbeit beschriebenen, Analyseaktivität und die informationsverarbeitende Aktivität mit der Implementierungsaktivität gleichgesetzt werden.[131]

In den hier vorgestellten allgemeinen Modellen zur Produktentwicklung wird der Umfang der Produktanforderungen nur implizit berücksichtigt. (Krishnan, Eppinger 1997) folgend kann eine nominale Dauer der Implementierungsaktivität angegeben werden und der Änderungsumfang kann unabhängig von dieser Zeitdauer durch die Intervallbreite eines Parameters modelliert werden. Dem Modell aus (Loch, Terwiesch 1998) folgend, steht die Dauer der Implementierungsaktivität fest und der Änderungsumfang ergibt sich durch die Änderungsrate und der damit verbundenen Anzahl von Änderungen während dieser Aktivität. Da der Produktumfang in der Softwareentwicklung, wie in Abschnitt 3.1.2 beschrieben, einen signifikanten Einfluss auf den Entwicklungsaufwand hat und Methoden zur Umfangsschätzung existieren, wird dieser in den Modellen dieser Arbeit verwendet und der Änderungsumfang abhängig vom Produktumfang betrachtet.

Die Modellierung der Unsicherheit ist in (Loch, Terwiesch 1998) und (Krishnan, Eppinger 1997) ähnlich, auch wenn die Annahmen unterschiedlich sind. In beiden Modellen wird die Unsicherheit auf einen Erwartungswert abgebildet. Der zu optimierende Überarbeitungsaufwand hängt in beiden Modellen vom erwarteten Änderungsumfang ab. Die Annahmen, die den Wahrscheinlichkeitsverteilungen der Erwartungswerte zugrunde liegen, beeinflussen die Ausgestaltung der Kostenfunktion nicht.

Die Kostenfunktion berechnet die Entwicklungszeit und ergibt sich in den allgemeinen Modellen zur Produktentwicklung aus der geplanten Dauer der informationsverarbeitenden Aktivität, der Überarbeitungsdauer, sowie der überlappungsfreien Dauer der informationsliefernden Aktivität. Eine Überlappung zwischen beiden Aktivitäten bedeutet, dass die Implementierungsaktivität beginnt bevor die Analyseaktivität beendet wurde.[132] Dies dient der Reduzierung der Entwicklungszeit und der Reaktionszeit („*response time*", vgl. MacCormack et al. 2001), d. h. der Zeit

[131] Diesen Vergleich zieht auch Loch, Terwiesch 1998, S. 1034.

[132] Vgl. auch den Abschnitt „Inkrementelle Entwicklungsprozesse".

zwischen Ermittlung der Anforderungen und Produktauslieferung. In Loch, Terwiesch 1998 wird dazu die Dauer der Startphase optimiert.[133] Findet nach Loch, Terwiesch 1998 keine Überlappung statt, kommt es nicht zu Überarbeitungsaufwand.

Eine weitere Möglichkeit der Reduzierung der Entwicklungszeit ergibt sich – dem Modell aus Loch, Terwiesch 1998 folgend – aus der Ermittlung der optimalen Kommunikationszeitpunkte zwischen Analyse- und Implementierungsaktivität. Die daraus resultierende Kommunikationsstrategie dient der Reduzierung der „erwarteten Kommunikationskosten“.

In (Krishnan, Eppinger 1997) wird unter dem Gesichtspunkt einer Reduzierung der Entwicklungszeit zusätzlich die Möglichkeit betrachtet, dass die informationsliefernde Aktivität frühzeitig beendet werden kann und ein Produkt mit geringerer Qualität entwickelt wird („*preemptive overlapping*“). Das Modell der Autoren dient der Bestimmung der optimalen Anzahl der Entwicklungszyklen, sowie des optimalen Endzeitpunkts der informationsliefernden Aktivität (*Finalisierung*). Voraussetzung für die Anwendung des Modells ist eine anfängliche Unsicherheit in der genauen Produktspezifikation.

In (Krishnan, Eppinger 1997) zeigen die Autoren an einer Fallstudie, dass bei einer erhöhten Sensitivität eine erhöhte Zahl von Zyklen geplant werden sollte. Dieses Ergebnis gilt unter der Voraussetzung negativer Skaleneffekte[134] und bei Verwendung einer zu Gleichung 4-12 ähnlichen Kostenfunktion. Diese Kostenfunktion wurde in Abschnitt 4.5.1 diskutiert und für die Modelle dieser Arbeit verworfen, da die Voraussetzungen sich nicht mit den Betrachtungen dieser Arbeit vereinen lassen.

Den Modellen aus (Krishnan, Eppinger 1997) und (Loch, Terwiesch 1998) folgend, wird im Meetingplan-Modell angenommen, dass die Implementierungsaktivität von Informationen aus der Analyseaktivität abhängig ist. Um einen wechselseitigen Informationsaustausch – wie er z. B. auch in (Ha, Porteus 1995) untersucht wird – zu modellieren, wird das, auf (Loch, Terwiesch 1998) basierende, Meetingplan-Modell um einen *Prototypingzyklus* erweitert. In (Loch, Terwiesch 1998) werden die Kosten dieses Informationsaustauschs in Meetings betrachtet, während in (Krishnan, Eppinger 1997)

---

[133] Kleinere Implementierungsaktivitäten können auch schon in der Startphase stattfinden, wie Abbildung 2-7 zeigt.

[134] Die in der erste Fallstudie aus Krishnan, Eppinger 1997 an einen Sprung in der Sensitivitätsfunktion zu erkennen sind.

keine Kosten durch den Informationsaustausch zwischen den beiden Aktivitäten modelliert werden.

In (Loch, Terwiesch 1998) und in den Modellen dieser Arbeit werden verschiedene Konzepte, wie Evolution und Sensitivität aus (Krishnan, Eppinger 1997) übernommen. (Krishnan, Eppinger 1997) hat den Autoren zufolge einen starken Einfluss auf Modelle aus vergleichbaren Forschungsbereichen und ähnelt der Arbeit aus (Loch, Terwiesch 1998). Drei Unterschiede zu (Krishnan, Eppinger 1997), die verschiedene Modellannahmen dieser Arbeit betreffen, werden hervorgehoben (vgl. Loch, Terwiesch 1998):

- Die Unsicherheit wird nicht durch ein Intervall möglicher Parameterausprägungen, sondern als Rate, mit der Änderungen („*engineering changes*") entstehen, modelliert.
- Diese Änderungen werden durch die Analyseaktivität gesammelt und zu einem bestimmten Zeitpunkt kommuniziert. Dieser Zeitpunkt hängt von der Sensitivität, der Evolution und den Fixkosten pro Zyklus ab und wird durch das Modell optimiert.
- Die Kommunikationszeitpunkte und die Überlappung der Aktivitäten werden *gleichzeitig* optimiert. Beides sind nach (Loch, Terwiesch 1998) Prozesseigenschaften, die bezüglich der Minimierung der Entwicklungszeit „auf eine grundlegende Art und Weise" interagieren.

5.1.2. Überlappung zwischen den Aktivitäten

Durch das Modell aus (Loch, Terwiesch 1998) wird die Überlappung der Aktivitäten in zweierlei Hinsicht gesteuert, jeweils mit der Absicht die Entwicklungszeit zu reduzieren. Zum einen wird die Länge der Startphase[135] („*precommunication*") optimiert. Diese anfängliche Phase hat den Schwerpunkt auf der Analyseaktivität und es sollte hier keine Überlappung stattfinden. Je länger die Startphase dauert desto geringer wird die Anforderungsunsicherheit im weiteren Entwicklungsprozess. Die Unsicherheit ist im Modell aus (Loch, Terwiesch 1998) abhängig von der Anzahl der Koordinationsmeetings in der Startphase, der inhärenten technischen Unsicherheit des Projekts und einem

[135] Die verschiedenen Phasen des Softwareentwicklungsprozesses werden in Abschnitt 2.2 ausführlich beschrieben und dort an Beispielen verdeutlicht.

unternehmensspezifischen Faktor. Der modellierte funktionale Zusammenhang dieser Faktoren ist konsistent mit Ergebnissen aus empirischen Untersuchungen in (Adler 1995).

Nach der Startphase beginnt die Konstruktionsphase. In dieser Phase wird – je nach Prozessmodell – der größte Teil der Implementierungsaktivität durchgeführt, es können auch noch Analyseaktivitäten stattfinden. Für diese Phase besteht nach (Loch, Terwiesch 1998) die Möglichkeit der Überlappung zwischen Analyse- und Implementierungsaktivität, durch die Zeit gespart werden soll. Änderungen entstehen nur während der Analyseaktivität und verursachen Überarbeitungsaufwand in der Implementierungsaktivität. Überlappen sich beide Phasen nicht entsteht kein Überarbeitungsaufwand.

In (Loch, Terwiesch 1998, S. 1042) kommen die Autoren zu dem Ergebnis, dass sich beide Aktivitäten entweder maximal überlappen oder keine Überlappung stattfinden sollte. Dieses Ergebnis basiert auf dem funktionalen Zusammenhang zwischen der Unsicherheit und der Dauer der Startphase. Dieser funktionale Zusammenhang modelliert einen sinkenden Grenznutzen, d. h. einer abnehmenden Reduktion der Unsicherheit, mit steigender Länge der Startphase. Bei verringerter Unsicherheit erhöht sich die optimale Überlappung, wodurch aber wiederum erhöhter Überarbeitungsaufwand entsteht. Dieser kann den Nutzen der Startphase und der Überlappung überkompensieren. D. h. die Startphase muss entweder ausreichend Nutzen generieren, dann wird vollständig überlappt oder sie ist überhaupt nicht nützlich und findet nicht statt. Die optimale Überlappung ohne Durchführung einer Startphase ergibt sich aus Gleichung 12 in (Loch, Terwiesch 1998, S. 1042). Ohne Startphase wird dort nur überlappt, falls die Unsicherheit unter einem bestimmten Schwellwert in Abhängigkeit von der Sensitivität und den fixen Kommunikationskosten liegt.

Aus diesen Beobachtungen und da in dieser Arbeit die Planung von zyklischen Entwicklungsprozessen betrachtet werden soll, wird im auf (Loch, Terwiesch 1998) basierenden Meetingplan-Modell eine vollständige Überlappung angenommen. Der Modell-Notation aus (Loch, Terwiesch 1998) folgend bedeutet dies: $T_1 = T_2, \lambda = 1$.[136]

---

[136] Die Dauer der Analyseaktivität ist dort T_1 und die der Implementierungsaktivität T_2. λ spiegelt den prozentualen Anteil von T_2 wider, der sich mit T_1 überlappt. $T_1 = T_2$ ist hier eine Annahme, die weitere Rechnungen vereinfacht. Dies hat keinen Einfluss auf die von der Überlappung unabhängigen Ergebnisse. Der Änderungsumfang kann durch die erwartete Anzahl von Änderungen pro Zeiteinheit angepasst werden kann.

5.2. Das Releaseplan-Modell

Durch Überführung eines lauffähigen Systems in die Nutzungsumgebung kann *Nutzerfeedback* eingefangen werden. Dieses Nutzerfeedback ist nötig, um – wie in Abschnitt 3.2 beschrieben – noch unbekannte Anforderungen zu erkennen. Durch das Releaseplan-Modell soll eine Grundlage geschaffen werden, auf Basis derer die Anzahl der Releasezyklen und damit die Frühzeitigkeit von Feedback geplant werden können. Des Weiteren soll durch das Modell eine optimale Releaseplanung, d. h. die Verteilung des Entwicklungsumfangs auf die Zyklen, ermittelt werden. Das Prozessflussdiagramm auf Ebene der Release-Zyklen ist in Abbildung 5-1 dargestellt.

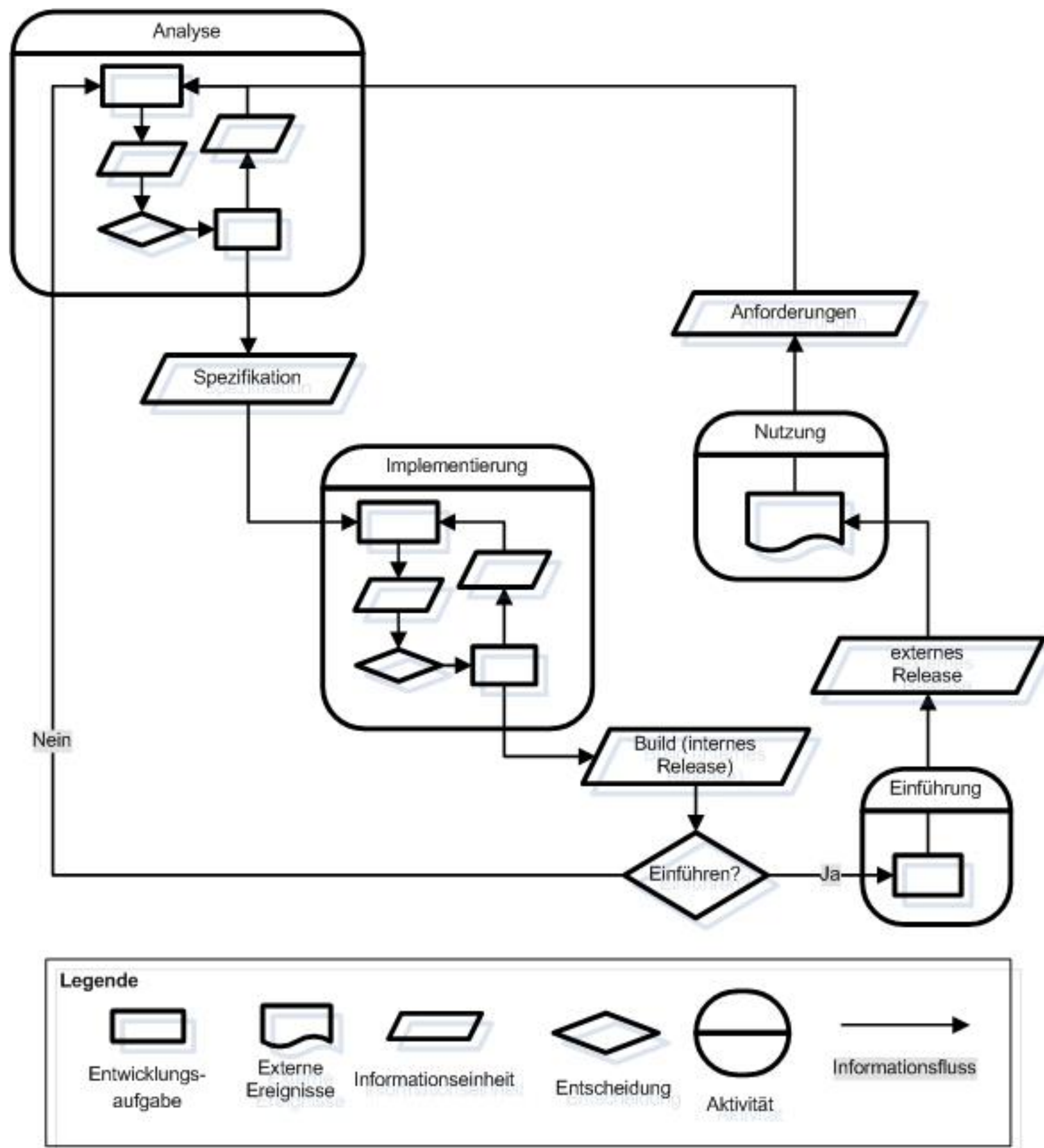


Abbildung 5-1: Prozessflussdiagramm auf Ebene der Release-Zyklen

5.2.1. Modellierung von Feedback-Effekten

Die Berechnung der Überarbeitungsdauer im Releaseplan-Modell entspricht der Berechnung der erwarteten Überarbeitungsdauer aus Abschnitt 4.2. In diesem Abschnitt wird die Funktion f_d zur Modellierung von Feedback-Effekten verwendet, die den Überarbeitungsumfangs beeinflussen.

Hier wird angenommen, dass durch Feedback, nach einem Release, vorher unbekannte Anforderungen ermittelt und die dadurch entstehenden Änderungen sofort umgesetzt

werden. Durch die sofortige Umsetzung dieser Änderungen reduziert sich der Überarbeitungsaufwand aller von diesen Änderungen abhängigen Anforderungen, die erst im weiteren Verlauf der Entwicklung implementiert werden sollen. Dieser Nutzenaspekt von frühzeitigem Feedback wurde bereits in Abbildung 3-3 veranschaulicht.

Der Feedback-Effekt wird in Gleichung 5-1 modelliert. Durch diese Berechnung des Änderungsumfangs wird modelliert, dass der Änderungsumfang qS/n eines Zyklus durch Feedback in *jedem* der darauf folgenden Releasezyklen proportional zur Anzahl der Zyklen sinkt. Der Faktor d spiegelt dabei die Effektivität von Feedback für die Analyse von Abhängigkeiten zwischen Änderungen und noch nicht implementierten Anforderungen wider. Liegt z. B. der Faktor d bei 100%, dann entspricht der Änderungsumfang – bei einer großen Anzahl von Zyklen – nach der Hälfte der Entwicklung nur noch ca. 50% des Änderungsumfangs eines Zyklus ohne Feedback und am Ende geht der Änderungsumfang gegen Null.

$$S_{q,i} = f_d(q, S/n, i) = (1 - d\frac{i-1}{n})qS/n \text{ mit } 0 \leq d < 1/(1-1/n) \leq 2, n \geq 2$$

Gleichung 5-1: Zyklusabhängiger Änderungsumfang bei Feedback-Effekten

Hier wird eine gleichmäßige Verteilung des Umfangs auf die Zyklen, d. h. ein Umfang von S/n pro Releases, angenommen. Der Änderungsumfang ist, wie in den Abschnitten 3.3.1 und 4.2 beschrieben, abhängig von der Wahrscheinlichkeit für Änderungen durch Abhängigkeiten bzw. der Komplexität der Anforderungen.

Abbildung 5-2 veranschaulicht die Reduktion des Änderungsumfangs durch Feedback. Hier ist zu erkennen, dass der gesamte Änderungsumfang bei obiger Modellierung in der zwei-zyklischen Entwicklung höher als in der vier-zyklischen Entwicklung ist. Feedback nach einem Zyklus reduziert den Änderungsumfang in *jedem* der darauf folgenden Zyklen um den Faktor $1/n$ (bei einer Feedbackeffektivität von $d = 1$).

In diesem Beispiel ist zu erkennen, dass der Änderungsumfang der zuletzt realisierten 25% der Anforderungen in der vier-zyklischen Entwicklung durch dreimaliges Feedback auf 25% des anfänglichen Änderungsumfangs sinkt. Dagegen sinkt der Änderungsumfang in der zwei-zyklischen Entwicklung durch Feedback nur einmal auf 50% des anfänglichen Umfangs.

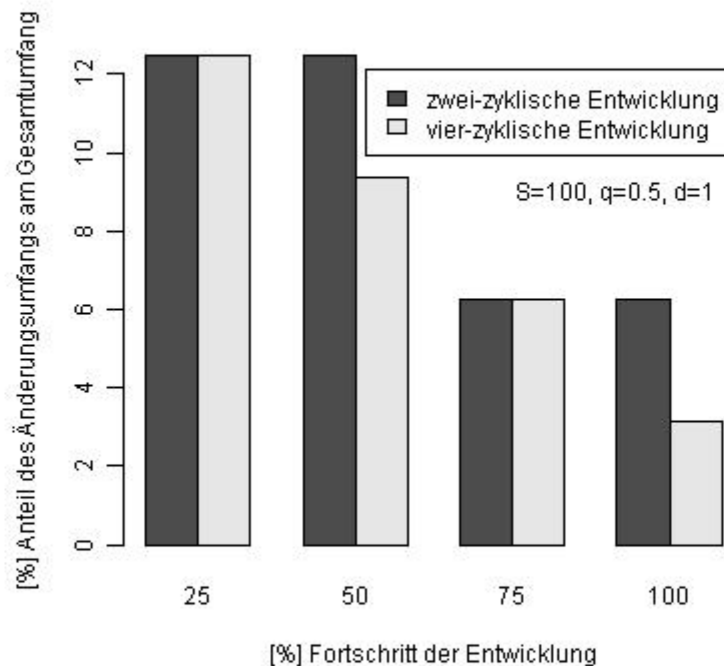


Abbildung 5-2: Reduktion des Änderungsumfangs durch Feedback-Effekte

Der gesamte Änderungsumfang ergibt sich, gemäß Gleichung 5-2, durch die Summierung der Änderungsumfänge aus Gleichung 5-1 über die Zyklen (Herleitung siehe Anhang A1).

$$S_q = \left(1 - \frac{1}{2} d (1 - \frac{1}{n})\right) \cdot qS$$

Gleichung 5-2: Gesamter Änderungsumfang bei Feedback-Effekten

Aus Gleichung 5-2 ist zu erkennen, dass der gesamte Änderungsumfang mit steigender Anzahl der Zyklen n und steigender Feedbackeffektivität d sinkt. Im obigen Beispiel ergibt sich in der vier-zyklischen Entwicklung ein gesamter Änderungsumfang in Höhe von 62,5% des Änderungsumfangs ohne zwischenzeitliches Feedback und in der zwei-zyklischen Entwicklung ein 75%-iger Änderungsumfang. Des weiteren ist zu erkennen, dass ein geringeres/erhöhtes d die Differenz der Änderungsumfänge bei einer unterschiedlichen Anzahl von Zyklen reduziert/erhöht. Aus diesem Grund wird dieser Parameter als Feedbackeffektivität bezeichnet.

Damit der gesamte Änderungsumfang nicht negativ wird, muss die Feedbackeffektivität die Bedingung gemäß Un-Gleichung 5-3 erfüllen (Herleitung siehe Anhang A2):

$$\frac{2}{1-1/n} \geq d$$

Un-Gleichung 5-3: Bedingung für die Feedbackeffektivität

5.2.2. Modellierung der Releaseplanung

Eine Steuerungsmöglichkeit, die in diesem Modell betrachtet werden soll, ist die Verteilung der Anforderungen auf die Releases (Releaseplanung) – wie in Abschnitt 2.6 beschrieben. Hier soll die Auswirkung der Releaseplanung auf den gesamten Überarbeitungsaufwand betrachtet werden. Nicht betrachtet wird hier der Zusammenhang der Releaseplanung mit einem möglicherweise frühzeitigen Ende der Entwicklung (vgl. Krishnan, Eppinger 1997), bei dem nicht alle Anforderungen umgesetzt wurden. In letzterem Fall scheint eine Verteilung der Anforderungen nach Kundenprioritäten sinnvoll, während hier eine risikogetriebene Verteilung untersucht wird.

Die in Gleichung 5-4 vorgestellte Funktion basiert auf der Evolutionsfunktion aus Loch, Terwiesch 1998. Diese wiederum basiert auf einem Modellierungsansatz aus (Krishnan, Eppinger 1997). Dort wird die Produktentwicklung i. A. betrachtet und die Unsicherheit durch die Veränderungen im Erwartungswert eines Produktparameters modelliert. Der Änderungsumfang eines Zyklus ist nach dem Modell aus (Krishnan, Eppinger 1997) abhängig von der Evolutionsfunktion und der Größe des ursprünglichen Intervalls möglicher Parameterausprägungen. Durch die Evolutionsfunktion kann nach (Krishnan, Eppinger 1997) modelliert werden, ob sich der Parameter schnell oder langsam dem endgültigen Wert annähert. Im hier vorgestellten Releaseplan-Modell wird dagegen von einem inkrementell wachsenden System ausgegangen. Bei einer Übertragung des Modells aus (Krishnan, Eppinger 1997) kann das Intervall möglicher Parameterausprägungen mit dem gesamten Änderungsumfang verglichen werden und die Evolutionsfunktion mit der Releaseplanung. Eine „schnelle Evolution“ nach (Krishnan, Eppinger 1997) kann hier mit anfänglichen großen Inkrementen gleichgesetzt werden.

$$S_i = f_e(S,i) = S/n + e(2i-1-n)/n$$

Gleichung 5-4: Funktion zur Verteilung der Anforderungen auf die Releases (Evolutionsfunktion)

Durch den *Evolutionsparameter* e kann der Umfang der Anforderungen eines Release bzw. Zyklus gesteuert werden. Die neu hinzugefügten Anforderungen werden als Inkrement bezeichnet. Je größer der Evolutionsparameter ist, desto kleiner sind die Inkremente in den ersten Zyklen. Ist der Parameter größer als Null, dann steigt der

Umfang der Inkremente im Laufe der Zyklen. Ist der Parameter kleiner Null, dann ist der Umfang der Inkremente fallend.

Diese Evolutionsfunktion ist so definiert, dass der Gesamtumfang der zu implementierenden Anforderungen unabhängig von der Releaseplanung ist, d. h. Gleichung 5-4 erfüllt die Voraussetzung: $\sum_{i=1}^{n} S_i = S$ (Herleitung siehe Anhang A4). Der Parameter e ist abhängig vom Gesamtumfang der Anforderungen. Dieser muss die Bedingung gemäß Un-Gleichung 5-5 erfüllen, damit der Umfang eines Inkrements nicht negativ wird (Herleitung siehe Anhang A5).

$$-S/n \leq e \leq S/n$$

Un-Gleichung 5-5: Bedingungen für den Parameter der Evolutionsfunktion

Abbildung 5-3 zeigt das Wachstum des Gesamtumfangs über die Zyklen bei unterschiedlichen Releaseplanungen gemäß obiger Modellierung – einmal bei steigendem Umfang pro Zyklus (*e=max.*) und einmal bei sinkendem Umfang pro Zyklus (*e=min.*). Diese Abbildung ist ähnlich der Abbildung aus (Krishnan, Eppinger 1997, S. 442).

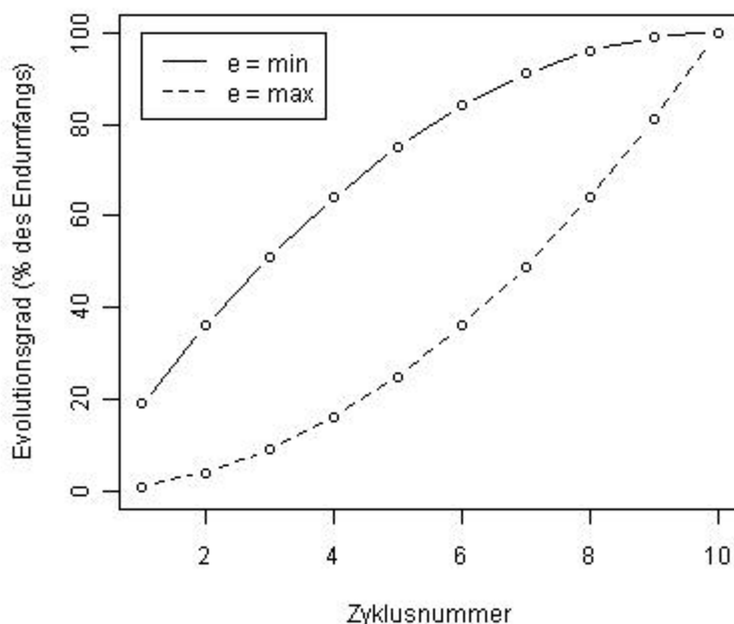


Abbildung 5-3: Verlauf des Gesamtumfangs bei unterschiedlichen Releaseplanungen

Aus dieser Abbildung ist zu erkennen, dass bei der Entwicklung anfänglich größerer Inkremente der Umfang nicht implementierter Anforderungen (der „Verlust“) bei Beendigung der Entwicklung in einem vorzeitigen Zyklus kleiner ist als bei anfänglich kleineren Inkrementen.

Dies ist nicht zu verwechseln mit dem Ergebnis aus (Krishnan, Eppinger 1997, S. 442). Dort kommen die Autoren zu dem Schluss, dass bei einer „schnellen Evolution“ ein früherer Finalisierungszeitpunkt gewählt werden sollte als bei einer „langsamen Evolution“, da in diesem Fall der „Verlust“ geringer ist. Im Releaseplan-Modell ist der Umfang nicht implementierter Anforderungen zu einem Zeitpunkt unabhängig von der Evolutionsfunktion, da diese den Umfang abhängig von der Zyklusnummer und nicht zeitabhängig steuert. Die Produktivität über die Zeit bleibt konstant. Das Ergebnis aus (Krishnan, Eppinger 1997, S. 442) ist übertragbar, wenn z. B. eine konstante Analysedauer für jeden Zyklus betrachtet wird: Bei einer schnellen Evolution kann diese Analysedauer mit einer fallenden Analyserate bzw. Produktivität erklärt werden, d. h. der Anteil gefundener Anforderungen pro Zeiteinheit sinkt. Damit sinkt auch der Anteil implementierter Änderungen pro Zeiteinheit über die Zyklen, was eine frühzeitige Finalisierung wie in (Krishnan, Eppinger 1997, S. 442) untersucht nützlicher macht.

5.2.3. Reduzierung der Entwicklungszeit in Abhängigkeit der Prozessplanung

Unter Verwendung der vorgestellten Modelle und Annahmen sollen nun die Parameter der Prozessplanung so bestimmt werden, dass die Entwicklungszeit reduziert wird.

Die Gesamtdauer der Entwicklung wird, wie in Kapitel 3 und 4 beschrieben, neben den im Rahmen des Releaseplan-Modells vorgestellten Faktoren bzw. Parametern, durch den Produktumfang, die Produktivität, die Sensitivität und die Unsicherheit beeinflusst.

Diese Entwicklungsdauer T wird zerlegt in die nominale Entwicklungsdauer und die erwartete Überarbeitungsdauer durch Unsicherheiten aus nicht vorhersehbaren Anforderungen ($T_{q,c}$), sowie die umfangsunabhängigen Fixkosten eines Zyklus (t_f). Die „nominale“ Entwicklungsdauer wird als Produkt aus Produktivität und Umfang ($p_t S$) unabhängig von der Zahl der Zyklen berechnet. Die Überarbeitungsdauer wird wie in 4.5.1 beschrieben abhängig von der Anzahl der Zyklen modelliert.

Die Zielfunktion im Releaseplan-Modell ergibt sich bei n Zyklen gemäß Gleichung 5-6 aus der Summe dieser Dauern.

$$T = p_t S + T_{q,c} + nt_f$$

Gleichung 5-6: Berechnung der Entwicklungszeit in der Zielfunktion des Releaseplan-Modells

Die gesamte Überarbeitungsdauer bei Feedback-Effekten

Wie oben erwähnt wird die Überarbeitungsdauer in Abhängigkeit der Anzahl der Zyklen und des Umfangs modelliert. In Abschnitt 5.2.1 wurde der Vorteil der mehrzyklischen Entwicklung durch Feedback-Effekte modelliert. Diese spiegeln sich in der gesamten Überarbeitungsdauer wider, die sich gemäß Gleichung 5-7 (Herleitung siehe Anhang A1) ergibt.

$$T_{q,c} = \sum_{i=1}^{n} p_t (1+c) \cdot qS / n \cdot (1 - d\frac{i-1}{n}) = p_t (1+c) \cdot qS \cdot \left(1 - \frac{1}{2} d (1 - \frac{1}{n})\right)$$

Gleichung 5-7: Die gesamte Überarbeitungsdauer bei Feedback-Effekten

Wie zu sehen, verringert sich die gesamte Überarbeitungsdauer mit der Erhöhung der Anzahl der Zyklen n. Wird nur ein Zyklus durchgeführt, so beträgt die gesamte Überarbeitungsdauer: $p_t(1+c)Sq$. Werden zehn Zyklen durchgeführt, so beträgt die gesamte Überarbeitungsdauer $d \cdot 45\%$ weniger.

Die optimale Anzahl von Zyklen bei Feedback-Effekten

Um die optimale Anzahl von Zyklen n zu bestimmen, müssen nun alle von n abhängigen Summanden der Zielfunktion aus Gleichung 5-6 betrachtet werden. Dies sind die Überarbeitungsdauer bei Feedback-Effekten und die gesamten Fixkosten wie in Gleichung 5-8 dargestellt.

$$T_{q,c} + nt_f = p_t (1+c) S \cdot q \left(1 - \frac{1}{2} d (1 - \frac{1}{n})\right) + nt_f$$

Gleichung 5-8: Zyklenzahlabhängige Summanden der Zielfunktion bei Feedback-Effekten

Durch Ableiten und Bestimmung der Nullstellen dieser Gleichung (Herleitung siehe Anhang A3) ergibt sich die optimale Anzahl der Zyklen n_d^* bei Feedback-Effekten gemäß Gleichung 5-9.

$$n_d^* = \sqrt{\frac{p_t(1+c)Sqd}{2t_f}}$$

Gleichung 5-9: Optimale Anzahl von Zyklen bei Feedback-Effekten im Releaseplan-Modell

Wie zu erkennen, steigt die optimale Anzahl von Zyklen mit:

- sinkenden Fixkosten
- erhöhter Unsicherheit
- erhöhter Feedbackeffektivität
- steigender Sensitivität des Systems
- steigendem Gesamtumfang.

In Abbildung 5-4 wird die Entwicklungszeit in Abhängigkeit der Anzahl der Releases und der Feedbackeffektivität dargestellt. Die Werte der Variablen der Kostenfunktion sind hier frei gewählt ($S = 100, p_t(1+c) = 1, q = 0.3, t_f = 1$) und die Feedbackeffektivität (*d*) wird variiert. Hier ist zu erkennen, dass eine optimale Anzahl von Releases, bei der die Entwicklungsdauer minimal ist, existiert und dieses Optimum mit steigender Feedbackeffektivität ansteigt. Bei einer Feedbackeffektivität von *d=0* ist ein Zyklus optimal, bei *d=0,5* sind drei Zyklen und bei *d=1* sind vier Zyklen optimal.

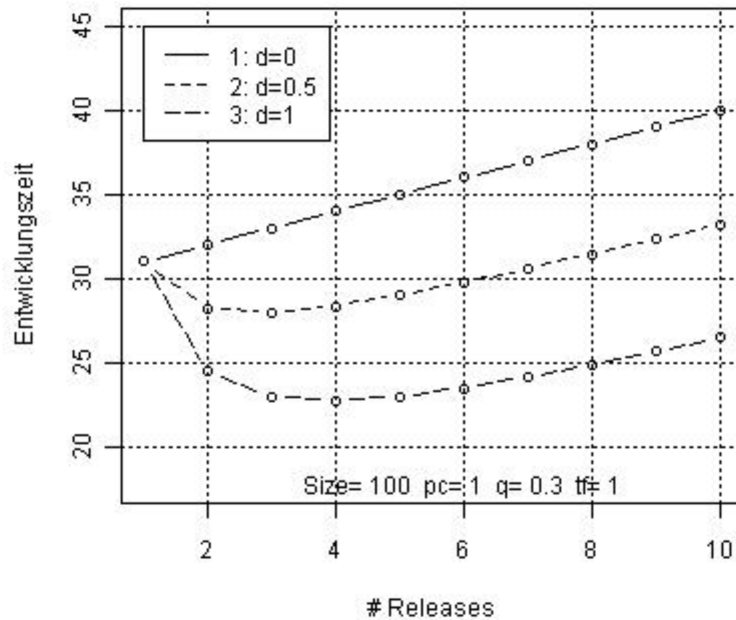


Abbildung 5-4: Entwicklungszeit in Abhängigkeit der Anzahl der Releases und der Feedbackeffektivität

Die optimale Anzahl von Zyklen in Abhängigkeit der Releaseplanung

Da die Releaseplanung in Gleichung 5-4 so modelliert wurde, dass der Gesamtumfang unabhängig von der Releaseplanung ist, wirkt sich die Releaseplanung nur in Zusammenhang mit Feedback-Effekten auf den Überarbeitungsaufwand aus. Hier wird angenommen, dass kleinere Inkremente den gleichen Feedback-Effekt haben wie größere Inkremente. Gleichung 5-10 stellt die Überarbeitungsdauer in Abhängigkeit der Releaseplanung und von Feedback-Effekten dar (Herleitung siehe Anhang A6).

$$
\begin{aligned}
T_{q,c} &= \sum_{i=1}^{n} p_t (1+c) f_d(q, f_e(S,i), i) \\
&= \sum_{i=1}^{n} p_t (1+c)(S/n + e(2i-1-n)/n) \cdot q(1 - d\frac{i-1}{n}) \\
&= p_t (1+c) q \left[S\left(2 - d(1-\frac{1}{n})\right)/2 - ed\left(n - \frac{1}{n}\right)/6 \right]
\end{aligned}
$$

Gleichung 5-10: Die Überarbeitungsdauer in Abhängigkeit der Releaseplanung und von Feedback-Effekten.

Aus Gleichung 5-10 ist zu erkennen, dass bei einer Erhöhung des Evolutionsparameters (e), welcher die Releaseplanung steuert, die gesamte Überarbeitungsdauer sinkt.[137] D. h. je kleiner die Inkremente in den ersten Zyklen desto geringer ist die gesamte Überarbeitungsdauer. Dies liegt daran, dass durch Feedback-Effekte die Unsicherheit in späteren Zyklen gesenkt wird und dadurch die Implementierung von größeren Inkrementen in späteren Zyklen weniger Überarbeitungsaufwand verursacht als in anfänglichen Zyklen.

Aus der obigen Annahme – die besagt, dass Feedback-Effekte unabhängig von der Inkrementgröße sind – folgt die Annahme, dass in den ersten Inkrementen die „kritischsten“, d. h. abhängigsten Anforderungen realisiert werden.[138] Dass dieses Vorgehen vorteilhaft ist, bestätigt die Untersuchung aus (Jootar, Eppinger 2002), sowie das Vorgehen risikogetriebener Entwicklungsprozesse wie sie z. B. in Boehms Spiralmodell (vgl. Boehm 1986) oder Microsofts Synch-and-Stabilize-Methodik (vgl. Cusumano, Selby 1995) verwendet werden.

Um die optimale Anzahl von Zyklen in Abhängigkeit der Releaseplanung zu bestimmen, wird die Überarbeitungsdauer aus Gleichung 5-11 in Gleichung 5-6 eingesetzt und die daraus resultierende Kostenfunktion optimiert. Die optimale Anzahl von Zyklen zur Reduzierung der Entwicklungszeit ergibt sich unter Beachtung der Fixkosten gemäß Gleichung 5-11 (Herleitung siehe Anhang A7).

$$n^*_{d,e} = \sqrt{\frac{p_t(1+c)Sqd + p_t(1+c)qde/3}{2t_f - p_t(1+c)qde/3}}$$

Gleichung 5-11: Optimalen Anzahl von Zyklen im Releaseplan-Modell in Abhängigkeit von der Releaseplanung

Anhand dieser Gleichung lässt sich bei gegebenen Annahmen schließen, dass mit steigendem Evolutionsparameter (e) die optimale Anzahl von Zyklen ansteigt, wenn positive Feedback-Effekte existieren ($d > 0$). D. h. je kleiner die anfänglichen Inkremente sind, und demzufolge je größer spätere Inkremente sind, desto mehr Zyklen sollten geplant werden. Dieses Ergebnis kann dadurch erklärt werden, dass hier Feedback-Effekte als unabhängig von der Inkrementgröße betrachtet werden, kleinere Inkremente

[137] Dies gilt bei zwei oder mehr Zyklen.

[138] Ohne diese Annahme müsste modelliert werden, dass bei kleineren Inkrementen weniger Unsicherheiten aus Abhängigkeiten reduziert werden.

aber geringeren Überarbeitungsaufwand verursachen, wodurch sich ein erhöhter Nutzen anfänglich kleinerer Inkremente ergibt. Dieser Effekt wird durch eine erhöhte Anzahl von Zyklen verstärkt genutzt, da so die durchschnittliche Inkrementgröße sinkt.

In Abbildung 5-5 wird die Entwicklungszeit in Abhängigkeit der Anzahl der Releases und der Releaseplanung dargestellt. Die Werte der Variablen der Kostenfunktion sind hier frei gewählt ($S = 100, p_t(1+c) = 1, q = 0.3, t_f = 1, d = 1$) und der Evolutionsparameters (*e*) wird variiert. Hier ist zu erkennen, dass die optimale Anzahl von Releases bei anfänglich kleineren Inkrementen (*e=10*) größer ist als bei anfänglich größeren Inkrementen (*e=-10*). Die gesamte Entwicklungszeit erhöht sich unabhängig von der Anzahl der Releases bei einer erhöhten Inkrementgröße in den ersten Zyklen.

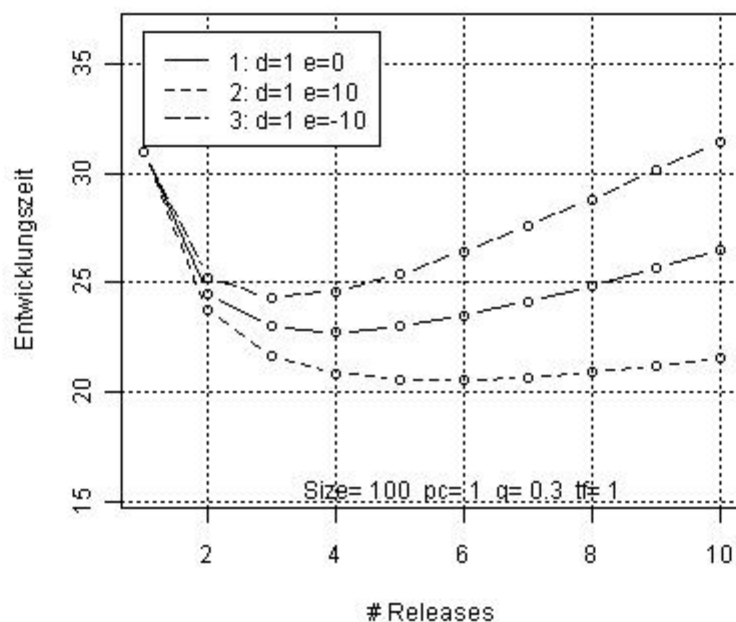


Abbildung 5-5: Entwicklungszeit in Abhängigkeit der Anzahl der Releases und der Releaseplanung

5.2.4. Zusammenfassung der Modellannahmen

Die Modellannahmen des Releaseplanmodells sollen an dieser Stelle noch einmal zusammengefasst werden, auch wenn sie an der zugehörigen Stelle in diesem Kapitel bereits diskutiert wurden.

A1. Durch Feedback nach einem Release werden noch unbekannte Unsicherheiten aus geänderten Anforderungen erkannt und die Änderungen sofort umgesetzt werden. Durch

Feedback auf Abhängigkeiten zu noch nicht implementierten Anforderungen wird die Unsicherheit in allen nachfolgenden Zyklen reduziert.

A2. Es treten keine Skaleneffekte auf bzw. positive und negative Skaleneffekte heben sich gegenseitig auf. Daraus folgt, dass die Kostenfunktion auf dem linearen Modell basiert.

A3. Die Sensitivität senkt die Produktivität.

A4. Die Anzahl der Zyklen beeinflusst nicht den Gesamtumfang, d. h. bei einer erhöhten Anzahl von Zyklen sinkt der Umfang eines jeden Zyklus.

A5. Unter der Voraussetzung einer risikogetriebenen Entwicklung wird angenommen, dass Inkremente zu Beginn der Entwicklung höhere Interdependenzen zu allen weiteren Inkrementen besitzen als später entwickelte Inkremente. Daraus folgt, dass anfängliches Feedback zu kleineren Inkrementen in gleichem Umfang zur Reduktion von Unsicherheiten beiträgt wie späteres Feedback zu umfangreicheren Inkrementen.

5.3. Das Meetingplan-Modell

Ein weiteres in dieser Arbeit entwickeltes Modell ist das *Meetingplan-Modell*, das auf dem Modell aus (Loch, Terwiesch 1998) basiert. Dieses Modell berücksichtigt zeitliche Aspekte wie Kommunikationsverzögerungen – im Gegensatz zum umfangsbasierten Ansatz des in Abschnitt 5.2 untersuchten Releaseplan-Modells. Um das Meetingplan-Modell für die Analyse von Feedback in inkrementellen Entwicklungsprozessen verwenden zu können, wird dieses Modell um einen Prototypingzyklus erweitert. Das erweiterte Modell wird als Prototypingplan-Modell bezeichnet und in Abschnitt 5.3.4 vorgestellt.

Das Meetingplan-Modell kann zur Planung von Meetings bei Optimierung der Entwicklungszeit verwendet werden. Die Planung geschieht durch Berechnung der optimalen Zeitpunkte von Meetings auf Basis einer Kostenfunktion. In diesen Meetings werden Informationen von der Analyse- an die Implementierungsaktivität weitergegeben.

Um eine Übertragung des Modells aus (Loch, Terwiesch 1998) auf die, in dieser Arbeit vorgestellten, Modellierungsansätze der Softwareentwicklung zu vereinfachen, wird die Zeiteinheit der dort verwendeten Variablen (t) mit der Dauer eines Releasezyklus gleichgesetzt und die Dauer eines solchen Zyklus als konstant angesehen.

5.3.1. Modellierung der Überarbeitungsdauer

Unsicherheit und Evolutionsfunktion

Der Änderungsumfang innerhalb eines Zyklus ist abhängig von der Evolutionsfunktion. Dem Modell aus (Loch, Terwiesch 1998) folgend, wird im Meetingplan-Modell angenommen, dass die Anforderungen von Beginn an präzise definiert sind[139] und im Laufe der Analyseaktivität Änderungen generiert werden. Hier entsprechen diese Änderungen neuen oder geänderten Anforderungen. Die Generierung von Änderungen kann, wie in (Loch, Terwiesch 1998), durch einen stochastischen Prozess – dort ein inhomogener Poisson-Prozess – mit zeitabhängigem Parameter modelliert werden. Dies entspricht dem, in dieser Arbeit vorgestellten, Konzept der Unsicherheit durch nicht-vorhersehbare Anforderungen, d. h. es ist nicht bekannt was sich ändert, aber es wird eine bestimmte Änderungswahrscheinlichkeit angenommen.

Durch die Evolutionsfunktion kann der Parameterwert des Poisson-Prozesses für ein bestimmtes Zeitintervall (hier die Dauer eines Release) berechnet werden. Der Parameterwert $\mu_\alpha(t)$ nach (Loch, Terwiesch 1998) entspricht der erwarteten Anzahl von Änderungen $qS(t)$ in diesem Zeitintervall im Meetingplan-Modell. Die Evolutionsfunktion der Autoren ist eine lineare Funktion mit dem Evolutionsparameter e, die so definiert ist, dass die Gesamtzahl der Änderungen unabhängig von e konstant bleibt. Ein negativer Evolutionsparameter hat z. B. die Folge, dass die Änderungsrate zunächst höher als der Mittelwert (hier: qS/n), zur Hälfte der Analysedauer gleich diesem und am Ende niedriger ist. Das Integral über diese lineare Funktion entspricht der Evolutionsfunktion aus (Krishnan, Eppinger 1997). Durch den Evolutionsparameter kann damit eine schnelle oder eine langsame Evolution modelliert werden.

Der Unterschied zum Releaseplan-Modell ist hier, dass die Änderungen an den Anforderungen alleine von der Analyseaktivität ermittelt werden. Diese Änderungen werden dann in Meetings spezifiziert (vgl. JAD-Workshops aus Abschnitt 3.1.3) und zur Implementierung freigegeben. Änderungen an Anforderungen, die erst durch Feedback aus vorangehenden Releases erhoben werden können, werden so nicht berücksichtigt. Somit können Änderungen, wie oben beschrieben, zeitabhängig modelliert werden und sind nicht von der Dauer der Implementierungsaktivität abhängig.

---

[139] Dies entspricht einem der Unterschiede zu Krishnan, Eppinger 1997.

Die in Gleichung 5-12 dargestellte Evolutionsfunktion basiert auf (Loch, Terwiesch 1998) und dient der Berechnung des Änderungsumfangs eines Releasezyklus.[140]

$$qS(t) = q\frac{S}{n}[1 + e(2t/n - 1)]$$

Gleichung 5-12: Die Evolutionsfunktion im Meetingplan-Modell

Wie eingangs beschrieben wird die Zeiteinheit t mit der konstanten Dauer eines Release gleichgesetzt und daher der Implementierungsumfang S/n in einem Release (bei n Releases) als konstant angesehen. Daraus folgt, dass die Gesamtdauer der Entwicklung n Zeiteinheiten entspricht.[141]

Sensitivität

Durch die Sensitivitätsfunktion wird die Überarbeitungsdauer in Abhängigkeit des Änderungsumfangs modelliert. Dem Modell aus (Loch, Terwiesch 1998) folgend, wird im Meetingplan-Modell angenommen, dass die Überarbeitungsdauer i. A. linear und nicht-fallend mit dem Änderungsumfang wächst. D. h. die Überarbeitungsdauer wird abhängig vom aktuellen Umfang der Entwicklung berechnet. Der Zeitaufwand pro Umfangseinheit in einem Release ist dann: $p_t(1+c)\cdot t$. Aus dieser Annahme und einer vollständigen Überlappung der Analyse- und Implementierungsaktivität (vgl. Abschnitt 5.1.2) ergibt sich die Überarbeitungsdauer gemäß Gleichung 5-13 (Herleitung siehe Anhang B1).

$$T_{q,c} = \int_{t=0}^{n} p_t(1+c)t \cdot qS(t)dt = p_t(1+c)qSn(\frac{1}{2} + \frac{1}{6}e)$$

Gleichung 5-13: Die gesamte erwartete Überarbeitungsdauer im Meetingplan-Modell

Hier ist zu erkennen, dass die Überarbeitungsdauer mit wachsendem Evolutionsparameter (e) und wachsender Gesamtdauer ansteigt. Daraus folgt, dass der Änderungsumfang in den ersten Releases möglichst klein sein sollte. Dies ist vergleichbar mit den Ergebnissen des Releaseplan-Modells, die besagen, dass anfänglich möglichst kleine Inkremente von Vorteil sind.

---

[140] Die funktionale Form ähnelt der Evolutionsfunktion im Releaseplan-Modell.

[141] Die Gesamtdauer wird in Loch, Terwiesch 1998 durch die Konstante T dargestellt.

Im Vergleich zu Gleichung 5-13 wird im Releaseplan-Modell die Überarbeitungsdauer jedoch gemäß Abschnitt 4.2 modelliert. Wird im Releaseplan-Modell eine Integralfunktion anstellte einer Summenfunktion eingesetzt, so resultieren Änderungen zusammen mit der obigen Evolutionsfunktion in einer gesamten erwarteten Überarbeitungsdauer gemäß Gleichung 5-14

$$T_{q,c} = \int_{t=0}^{n} p_t (1+c) q S(t) dt = p_t (1+c) q S$$

Gleichung 5-14: Vergleichende Darstellung der gesamten erwarteten Überarbeitungsdauer bei Verwendung einer Integralfunktion im Releaseplan-Modell.

Aus dieser vergleichenden Darstellung ist zu erkennen, dass die Überarbeitungsdauer im Releaseplan-Modell im Gegensatz zum Meetingplan-Modell unabhängig von der Anzahl der Zyklen bzw. der Entwicklungszeit und der Evolution ist.[142]

In (Loch, Terwiesch 1998) wird erwähnt (allerdings nicht in den Simulationen berücksichtigt), dass die Sensitivität ansteigt, wenn Änderungen nicht nur Überarbeitungsaufwand verursachen, sondern auch einen Teil des bereits entwickelten Produkts überflüssig machen. Auf der anderen Seite sinkt die Sensitivität mit sinkendem Produktumfang und bei negativen Skaleneffekten. Daher nehmen die Autoren wahrscheinlich an, dass dieser überflüssigen Teil ersetzt werden muss und dadurch neuen Implementierungsaufwand verursacht.

(Loch, Terwiesch 1998) folgend wird im Meetingplan-Modell angenommen, dass Überarbeitungsaufwand nur während der Überlappung entsteht. Bei einer Betrachtung der optimalen Überlappung unter Berücksichtigung der Sensitivität resultiert dies den Autoren zufolge darin, dass eine erhöhte Sensitivität – unter bestimmten Optimierungsbedingungen – den Überlappungsgrad reduziert.

5.3.2. Reduzierung der Entwicklungszeit in Abhängigkeit der Prozessplanung

Zerlegung der Gesamtdauer der Entwicklung

Ähnlich den Annahmen im Releaseplan-Modell (vgl. Abschnitt 5.2.3) wird die Gesamtdauer der Entwicklung im Meetingplan-Modell zerlegt in:

[142] Dies gilt für das Releaseplan-Modell ohne Berücksichtigung von Feedback-Effekten.

- die Dauer der Startphase T_S (entspricht $\alpha\tau_1$ in Loch, Terwiesch 1998)
- die nominale Implementierungsdauer T_I (entspricht T_2 in Loch, Terwiesch 1998)
- die erwartete Überarbeitungsdauer durch nicht-vorhersehbare Anforderungen $T_{q,c}$ (entspricht $ER(\alpha, \sigma(t), \lambda)$ in Loch, Terwiesch 1998)
- die Fixkosten eines Zyklus $t_f(t)$ (entspricht $\beta(t)\tau_2$ in Loch, Terwiesch 1998)
- die zusätzliche Überarbeitungsdauer aus Kommunikationsverzögerungen $t_{q,c}^{delay}$ innerhalb eines Zyklus

Die letzten beiden Zeitdauern entsprechen zusammen den erwarteten Kommunikationskosten $EC(\alpha, \sigma(t), \lambda)$ nach Loch, Terwiesch 1998. Die Analysedauer (entspricht T_1 in Loch, Terwiesch 1998) wird hier nicht betrachtet, da eine vollständige Überlappung angenommen wird. Die Implementierungsdauer wird in (Loch, Terwiesch 1998) vorher geschätzt bzw. festgelegt, dabei wird angenommen, dass ein fest vorgegebenes Qualitätslevel am Ende erreicht werden muss.

Fixkosten durch Meetings

Die Fixkosten eines Zyklus (vgl. Abschnitt 4.5.2) ergeben sich nach (Loch, Terwiesch 1998) aus der Anzahl der Meetings innerhalb eines Releases (Kommunikationsrate) und der Dauer („*setup time*“) pro Meeting. Diese Anzahl berechnet sich aus der Änderungsrate und einem Schwellenwert ($s(t)$), der den Umfang der gesammelten Änderungen angibt, bei dem ein Meeting durchgeführt wird. Die Kommunikationskosten durch alle Meetings in einem Releasezyklus ergeben sich gemäß Gleichung 5-15.

$$t_f(t) = t_{setup} \cdot qS(t) / s(t)$$

Gleichung 5-15: Fixe Kommunikationskosten in einem Releasezyklus des Meetingplan-Modells

Diese Meetings können als JAD-Workshops (vgl. Abschnitt 3.1.3) verstanden werden. In diesen Workshops wird meist mit analytischen Modellen (und, im Vergleich zu Releases, weniger mit einer lauffähigen Software) gearbeitet, um aus groben Anforderungen eine detaillierte Spezifikation zu erstellen.

Zusätzliche Überarbeitungsdauer durch Kommunikationsverzögerungen

Durch verspätete Kommunikation („*communication delay*“) von Änderungen in Meetings erhöht sich die Überarbeitungsdauer, da das System, wie oben beschrieben, über die Zeit

stetig wächst und angenommen wird, dass eine spätere Umsetzung von Änderungen in ein größeres System zusätzliche Überarbeitungsdauer bedeutet (vgl. Loch, Terwiesch 1998).

Änderungen werden (Loch, Terwiesch 1998) folgend in $\beta(t) = qS(t)/s(t)$ Meetings pro Release kommuniziert. Wie oben beschrieben bezeichnet $qS(t)$ den Änderungsumfang pro Release und $s(t)$ den Änderungsumfang pro Meeting.

Bei einer konstanten Änderungsrate zwischen zwei Meetings, ergibt sich die mittlere Verzögerung der Kommunikation einer Änderung aus der Hälfte der Zeitspanne zwischen zwei Meetings: $t^{delay}(t) = \frac{1}{2} \cdot \frac{1}{n\beta(t)}$.

Der, durch diese Verzögerungen erhöhte, Überarbeitungsaufwand pro Meeting ergibt sich aus der Sensitivitätsfunktion und der Anzahl der Änderungen pro Meeting: $t_{q,c}^{delay,M}(t) = p_t(1+c) \cdot t^{delay}(t) \cdot s(t)$

Multipliziert mit der Anzahl der Meetings pro Zyklus ergibt sich die zusätzliche Überarbeitungsdauer durch Kommunikationsverzögerungen in einem Releasezyklus gemäß Gleichung 5-16.

$$t_{q,c}^{delay}(t) = \beta(t) \cdot t_{q,c}^{delay,M}(t) = p_t(1+c) \cdot \frac{s(t)}{2n}$$

Gleichung 5-16: Zusätzliche Überarbeitungsdauer durch Kommunikationsverzögerungen in einem Releasezyklus des Meetingplan-Modells

Die Berechnung der zusätzlichen Überarbeitungsdauer weicht hier von der Berechnung in Lemma 3 im Anhang von (Loch, Terwiesch 1998, S. 1044) ab, da dort die mittlere Anzahl nicht abgearbeiteter Änderungen in einer Zeiteinheit (hier die Dauer eines Zyklus bzw. Release) *fälschlicherweise* durch die mittlere Anzahl von nicht abgearbeiteten Änderungen zwischen zwei Meetings modelliert wird. Die Dauer zwischen zwei Meetings entspricht auch in den Annahmen aus (Loch, Terwiesch 1998) nicht einer Zeiteinheit.[143]

---

[143] Des Weiteren wurde im Gegensatz zu Loch, Terwiesch 1998 die „letzte Änderung“ vor einem Meeting im Überarbeitungsaufwand mit berücksichtigt, da hier eine stetige Skala für den Änderungsumfang angenommen wird. Diese Modelländerung hat aber keinen Einfluss auf die Ergebnisse.

Bestimmung der optimalen Anzahl von Meetings bei vorgegebener Startphase

Es wird hier, wie in Abschnitt 5.1.2 beschrieben, angenommen, dass sich Analyse- und Implementierungsaktivität vollständig überlappen und die Implementierungsdauer größer als die Analysedauer ist. Dann ergibt sich die Berechnung der Gesamtdauer im Meetingplan-Modell gemäß Gleichung 5-17.

$$T = T_S + T_I + T_{q,c} + \int_{t=0}^{n} t_{q,c}^{delay}(t) + t_f(t) \partial t$$

Gleichung 5-17: Berechnung der Gesamtdauer im Meetingplan-Modell

Bei vorgegebener Länge der Startphase, geschieht die Optimierung der Gesamtdauer durch Optimierung der von der Kommunikationsstrategie abhängigen Entwicklungsdauer. Letztere ergibt sich aus der erwarteten Kostenrate, d. h. hier aus der, durch Meetings und Kommunikationsverzögerungen entstehenden, zusätzlichen Entwicklungsdauer pro Release (vgl. Lemma 3 im Anhang von Loch, Terwiesch 1998) und berechnet sich gemäß Gleichung 5-18.[144]

$$t_f(t) + t_{q,c}^{delay}(t) = t_{setup} \cdot qS(t)/s(t) + p_t(1+c) \cdot \frac{s(t)}{2n}$$

Gleichung 5-18: Zu minimierende Kommunikationskosten pro Release im Meetingplan-Modell

Die Anzahl der Meetings pro Release $\beta(t)$ ergibt sich aus dem Umfang erwarteter Änderungen dividiert durch den Änderungsumfang ($s(t)$), der in jedem Meeting kommuniziert wird. Die optimale Anzahl von Meetings wird durch Optimierung von Gleichung 5-18 in Abhängigkeit des „kritischen" Änderungsumfangs pro Meeting bestimmt (vgl. Loch, Terwiesch 1998) und ergibt sich gemäß Gleichung 5-19 (Herleitung siehe Anhang B2).

$$\beta(t) = qS(t)/s(t) \underset{=\!=}{opt.} \sqrt{\frac{p_t(1+c)qS(t)}{2nt_{setup}}}$$

Gleichung 5-19: Optimale Anzahl von Meetings innerhalb eines Releases im Meetingplan-Modell

Aus Gleichung 5-19 ist zu erkennen, dass aus einem erhöhten Änderungsumfang in einem Release ($qS(t)$) und einer erhöhten Sensitivität (c) eine erhöhte Kommunikationsrate

[144] Die Kostenrate wurde hier gegenüber Loch, Terwiesch 1998 modifiziert (vgl. Abschnitt 5.3.1).

resultiert. Diese sinkt mit steigender durchschnittlicher Meetingdauer (t_{setup}) und steigender Gesamtdauer (n). Die optimale Kommunikationsrate kann für jedes Release neu bestimmt werden, z. B. falls sich der Änderungsumfang ändert.

Die funktionale Form von Gleichung 5-19 des Meetingplan-Modells ähnelt der von Gleichung 5-9 zur Bestimmung der optimalen Anzahl von Releasezyklen im Releaseplan-Modell.

5.3.3. Diskussion der Modellannahmen

Im Meetingplan-Modell wird durch die Überlappung der beiden Aktivitäten Analyse und Implementierung zunächst einmal die Entwicklungszeit reduziert. Die Überlappung der Aktivitäten hat eine verspätete Kommunikation von Änderungen zur Folge. Durch diese Kommunikationsverzögerung erhöht sich der Überarbeitungsaufwand, da das System über die Zeit stetig wächst und angenommen wird, dass eine spätere Umsetzung von Änderungen in ein größeres System einen erhöhten Aufwand bedeutet (steigende Sensitivität). Durch eine erhöhte Kommunikationsrate lassen sich diese Kommunikationsverzögerung und ihre negativen Folgen reduzieren und somit ein Nutzen aus dieser verstärkten Kommunikation zwischen Analyse- und Implementierungsaktivität erzielen.

Die optimale Kommunikationsrate ist u. a. abhängig von den Kommunikationskosten und den Kommunikationsverzögerungen und wird hier in Meetings pro Release gemessen. Durch eine geringere Kommunikationsrate verringern sich die Kommunikationskosten, aber erhöhen sich bei steigender Sensitivität die Kosten durch verspätete Überarbeitung. Unter Abwägung dieser verschiedenen Kostenaspekte kann eine optimale Kommunikationsrate berechnet werden (vgl. Gleichung 5-19). Diese optimale Kommunikationsrate sollte für jedes Release neu bestimmt werden.

Es wird ein Problem der, auf Loch, Terwiesch 1998 basierenden, Modellannahmen beobachtet: Die Annahmen des Modells implizieren, dass die u. a. durch Kommunikationsverzögerungen verursachten Überarbeitungen sofort nach einem Meeting durchgeführt werden, da der Überarbeitungsaufwand für den Zeitpunkt des Meetings berechnet wurde. Würde die Überarbeitungsaktivität erst am Ende der Entwicklungszeit stattfinden (wie in Abbildung 1 in Loch, Terwiesch 1998 dargestellt), hätte sich der Überarbeitungsaufwand durch die ansteigende Sensitivität weiter erhöht.

Daraus folgt implizit die Annahme, dass während der Kommunikations- und Überarbeitungsaktivität keine weiteren Änderungen auftreten dürfen, da sich ansonsten rekursiv weitere Kommunikationsverzögerungen ergeben. D. h. nur die Implementierung der ursprünglichen Anforderungen darf parallel zur Analyse laufen. Diese Folgerung passt jedoch nicht zur Annahme, dass Änderungen, von der Implementierung unabhängig, durch einen stochastischen zeitabhängigen Prozess generiert werden. Eine Möglichkeit wäre es, die durch Überarbeitungsaktivitäten wiederum entstehenden Kommunikationsverzögerungen z. B. durch Differentialgleichungen zu modellieren, was das Modell komplexer machen würde. Eine andere Möglichkeit wäre es den Überarbeitungsaufwand als parallelen zusätzlichen Personalaufwand und nicht als zeitlichen Zusatzaufwand zu modellieren, wodurch sich die Zielfunktion ändern würde. Diese Änderungen des Modells und mögliche Auswirkungen auf die Ergebnisse sind nicht Teil der Betrachtungen dieser Arbeit.

Nach (Loch, Terwiesch 1998) besteht eine wesentliche Einschränkung des Meetingplan-Modells in der einseitig gerichteten Kommunikation zwischen den Aktivitäten – im Gegensatz zu einem möglichen wechselseitigen Informationsaustausch. Das Meetingplan-Modell wird daher in Abschnitt 5.3.4 durch einen Prototypingzyklus um Feedback aus der Nutzung erweitert.

5.3.4. Erweiterung um Feedback durch Prototyping

Ein Prototyp stellt ein Modell des zu entwickelnden Produkts dar. Er kann in der Nutzungsumgebung eingesetzt werden und erlaubt „reales“ Experimentieren (vgl. Thomke 1998). Zum Beispiel kann durch einen Prototypen die Benutzeroberfläche einer Software auch ohne geforderte Serveranbindung auf Akzeptanz getestet werden. Durch diese Prototypen sollen zusätzlich Informationen gewonnen werden, die nicht durch die „kalte“ Anforderungsanalyse ermittelt werden können, sondern durch Feedback eingefangen werden müssen. In Abschnitt 3.2.1 werden Ursachen solcher unvorhersehbaren Anforderungen detailliert beschrieben.

Im Meetingplan-Modell wird angenommen, dass die Unsicherheit der Anforderungen durch die Analyse alleine geklärt werden kann. Um auch die Möglichkeit zu modellieren, dass Anforderungen aus Nutzerfeedback erkannt werden, wird das auf (Loch, Terwiesch 1998) basierende Meetingplan-Modell im Folgenden um einen Prototypingzyklus erweitert. Diese Erweiterung ist nicht zu verwechseln mit dem in (Serich 2005)

vorgeschlagenen „*upfront prototyping*“, welches auch auf dem Modell aus (Loch, Terwiesch 1998) aufbaut. In (Serich 2005) wird der Analyseaktivität eine Prototyping-Aktivität vorangestellt, die sich von dieser alleine durch die Änderungsrate und den Evolutionsparameter unterscheidet. Es wird dort der optimale Übergangszeitpunkt zwischen beiden Aktivitäten bestimmt.

Der Prototypingzyklus

Es wird angenommen, dass der Zyklus von der Prototyp-Anforderung bis zur Reaktion auf das Feedback folgende Schritte durchläuft:

1. Die Analyseaktivität fordert die Entwicklung von Prototypen in der Implementierungsaktivität.[145]
2. Die Implementierungsaktivität erstellt die Prototypen gemäß den Anforderungen (Prototyping).
3. Die implementierten Prototypen werden gesammelt und an einem zu optimierenden Zeitpunkt in die Nutzung überführt.[146]
4. Aus der Nutzung der Prototypen entstehen neue Anforderungen (Feedback), die von der Analyseaktivität ermittelt werden und wieder in den änderungen-generierenden Prozess einfließen
5. Die Änderungen werden an die Implementierungsaktivität übermittelt und dort umgesetzt.

Abbildung 5-6 stellt den Prototypingzyklus schematisch dar, die Nummern entsprechen der Nummerierung der Schritte in der obigen Beschreibung.

[145] D.h. diese Anforderungen stammen aus einer prozessinternen Aktivität, im Gegensatz zu Anforderungsänderungen aus der externen Nutzung.

[146] Dadurch ist die Implementierungsaktivität keine reine informationsverarbeitende Aktivität mehr.

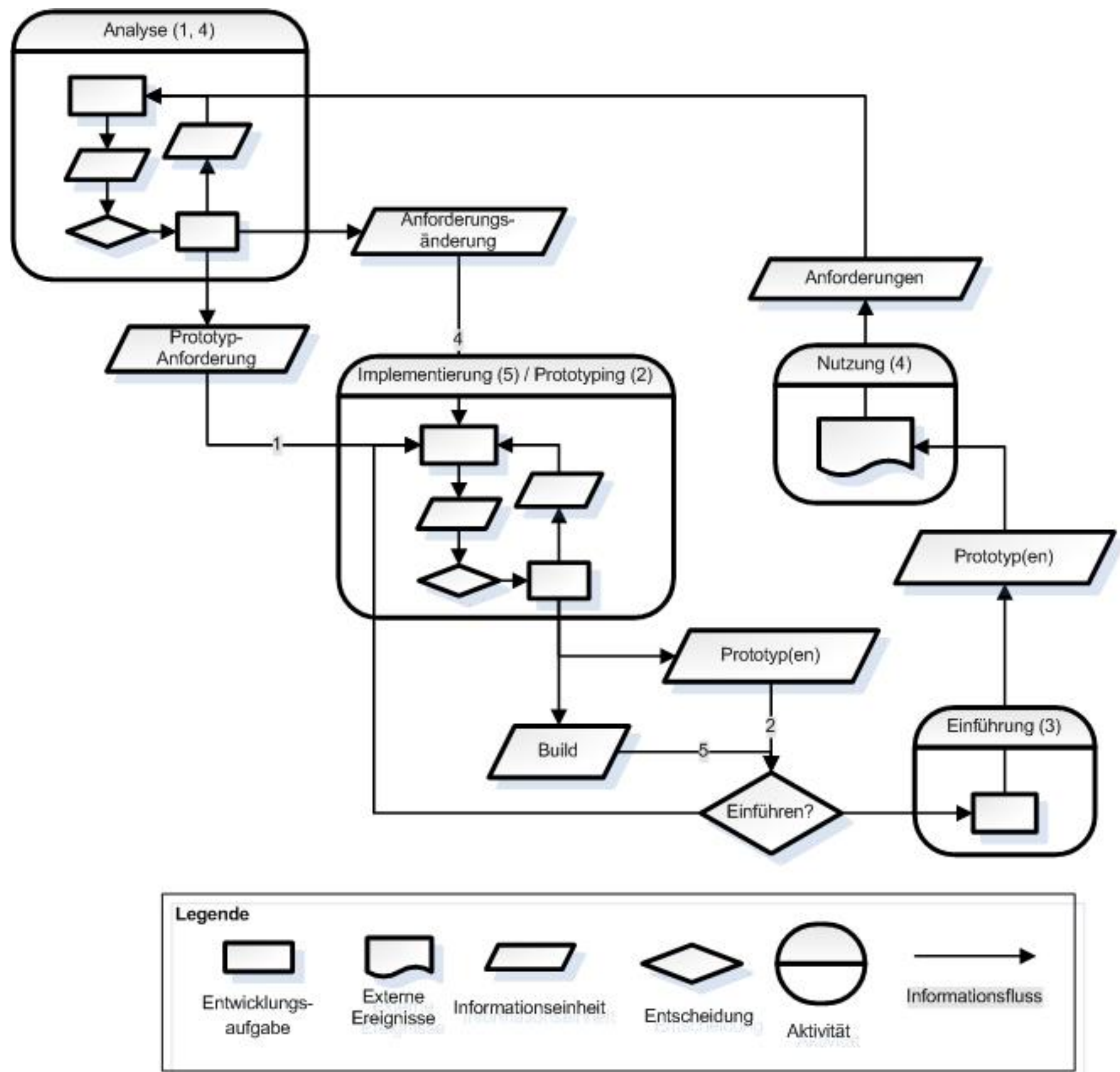


Abbildung 5-6: Prozessflussdiagramm des Prototypingplan-Modells

Modellannahmen

Es wird angenommen, dass eine bestimmte Anzahl von Prototypen notwendig ist, um alle noch unbekannten Anforderungen zu ermitteln. Analog zur Änderungsrate des Meetingplan-Modells wird hier durch die Analyseaktivität eine bestimmte Anzahl von Prototypen gefordert (vgl. Schritt eins). Der Prototypumfang pro Releasezyklus, d. h. die Prototyp-Anforderungsrate, wird abhängig vom Gesamtumfang S durch $q_P S(t)$ dargestellt.

Die gesamte Implementierungsdauer der Prototypen und die Analysedauer für Feedback aus Prototypen werden hier als unabhängig von der Prozessplanung betrachtet und gehen daher nicht in die Optimierung ein.

Vergleichbar der Kommunikationskosten im Meetingplan-Modell kann angenommen werden, dass durch die Kommunikation der Prototyp-Anforderungen zusätzliche Kosten entstehen. Die Kommunikationskosten werden hier jedoch nur abhängig vom Gesamtumfang der Prototypanforderungen angesehen, welcher unabhängig von der Prozessplanung ist. D. h. an dieser Stelle wird im Gegensatz zum Meetingplan-Modell keine optimale Kommunikationsrate berechnet. Es wird hier angenommen, dass die Prototyp-Anforderungen sofort an die Implementierungsaktivität weitergegeben werden[147], da dort die Entscheidung gefällt werden soll, wann eine entsprechende Architektur entwickelt und diese Prototypen umgesetzt werden.

Zur Planung der Einführung der Prototypen in die Nutzung (Schritt drei), wird ein Schwellwert für den Prototypumfang ($s_P(t)$), bei dem die Prototypen in die Nutzung überführt werden, definiert. Des Weiteren werden die Einführungskosten der entwickelten Prototypen als zyklisch auftretende Fixkosten (t_{setup}^{P}) modelliert.

Damit das Modell praktikabel ist, sollte die Dauer eines Zyklus – innerhalb dem die Kosten durch das Modell minimiert werden – so groß gewählt werden, dass mindestens eine Einführung innerhalb eines Releasezyklus stattfindet. Bei der letzten Einführung innerhalb eines Zeitraums muss dann über eine neue Einführungsstrategie entschieden werden. Aus diesen Annahmen wird hier eine, dem Meetingplan-Modells ähnliche, Kostenfunktion erstellt.

Die Planung von Schritt fünf – die Weitergabe von durch Feedback zu Prototypen ermittelten Änderungen an die Implementierung – kann wieder mit Hilfe des Meetingplan-Modells durchgeführt werden. Der durch diese Änderungen verursachte Überarbeitungsaufwand fällt in der gleichen Implementierungsaktivität wie im Meetingplan-Modell an. Daher wird, wie im Meetingplan-Modell, von einer steigenden Sensitivität bei fortschreitendem Implementierungsstand ausgegangen.

Da der Überarbeitungsaufwand durch die, mit Hilfe von Prototypen erfassten, Kundenanforderungen beeinflusst wird, sollen auch die Wirkungszusammenhänge zwischen der Effektivität der Prototypen und dem Überarbeitungsaufwand untersucht werden. Dazu wird angenommen, dass eine Prototypanforderung eine bestimmte Anzahl

[147] Alternativ könnte auch angenommen werden, dass die Analyseaktivität die Prototypanforderungen sammelt und dann einer optimalen Kommunikationsstrategie folgend (wie auch bei der Kommunikation von Anforderungsänderungen im Meetingplan-Modell) diese an die Implementierungsaktivität weitergibt.

von Änderungen in den Kundenanforderungen generiert. Diese „Prototyp-Effektivität“ *g* wird durch das Verhältnis zwischen Prototypumfang und Änderungsumfang gemessen.

Die optimale Prototyp-Einführungsrate

Die Anzahl von Prototyp-Einführungen pro Releasezyklus ergibt sich aus dem Schwellwert bei dem die Prototypen in die Nutzung überführt werden und der Prototyp-Anforderungsrate. Diese Prototyp-Einführungsrate wird daher durch $q_P S(t) / s_P(t)$ berechnet. Die Einführungskosten eines Zyklus ergeben sich, gemäß Gleichung 5-20, aus den Fixkosten einer Einführung multipliziert mit der Anzahl der Einführungen.

$$\beta_P(t) = t_{setup}^P \cdot q_P S(t) / s_P(t)$$

Gleichung 5-20: Berechnung der Prototyp-Einführungsrate

Dadurch, dass die Prototypen nicht unmittelbar implementiert und eingeführt werden, sondern erst ab einem bestimmten Schwellwert, verzögert sich das Feedback und die Implementierung der durch Prototypen generierten Anforderungsänderungen. Analog dem Meetingplan-Modell ergibt sich, bei einer konstanten Prototyp-Anforderungsrate zwischen zwei Einführungen, die mittlere Verzögerung der Implementierung aus der Hälfte der Zeitspanne zwischen zwei Einführungen: $t_P^{delay}(t) = \frac{1}{2} \cdot \frac{1}{n\beta_P(t)}$.

Der, durch diese Verzögerungen erhöhte, Überarbeitungsaufwand pro Einführung ergibt sich aus der Sensitivitätsfunktion und der Anzahl der Änderungen pro Einführung gemäß Gleichung 5-21 – bei gegebener Produktivität (p_t) und Sensitivität (*c*). Unter Berücksichtigung der Prototyp-Effektivität (*g*) werden durch Feedback zu Prototypen *einer* Einführung Anforderungsänderungen im Umfang von $g \cdot s_P(t)$ generiert.

$$t_{q,c,P}^{delay,M}(t) = p_t(1+c) \cdot t_P^{delay}(t) \cdot g \cdot s_P(t)$$

Gleichung 5-21: Zusätzliche Überarbeitungsdauer pro Einführung durch Verzögerungen im Prototyping

Multipliziert mit der Anzahl der Einführungen pro Releasezyklus ($\beta_P(t)$) ergibt sich die zusätzliche Überarbeitungsdauer durch Verzögerungen im Prototyping in einem Zyklus gemäß Gleichung 5-22.

$$t_{q,c,P}^{delay}(t) = \beta_P(t) \cdot t_{q,c}^{delay,P}(t) = p_t(1+c) \cdot g \cdot \frac{s_P(t)}{2n}$$

Gleichung 5-22: Zusätzliche Überarbeitungsdauer durch Verzögerungen im Prototyping in einem Releasezyklus des Prototypingplan-Modells

Eine erhöhte Überarbeitungsdauer durch verspätete Kommunikation von Prototypanforderungen oder verspätete Analyse von Anforderungsänderungen wird hier nicht betrachtet.

Die erwarteten zu minimierenden Kosten ergeben sich aus den Einführungskosten und der zusätzlichen Überarbeitungsdauer pro Releasezyklus gemäß Gleichung 5-23 (ähnlich Lemma 3 im Anhang von Loch, Terwiesch 1998).

$$t_{setup}^{P} \cdot q_P S(t) / s_P(t) + p_t(1+c) \cdot g \cdot \frac{s_P(t)}{2n}$$

Gleichung 5-23: Zu minimierende Kosten pro Release im Prototypingplan-Modell

Wird Gleichung 5-23 in Abhängigkeit des Schwellwerts für die Einführung von Prototypen ($s_P(t)$) minimiert, so berechnet sich die optimale Prototyp-Einführungsrate $\beta_P^*(t)$ gemäß Gleichung 5-24 (Herleitung analog zu Anhang B2, zu beachten ist, dass der Faktor *g* hinzukommt).

$$\beta_P^*(t) = q_P S(t) / s_P^*(t) \underset{=\!=}{opt.} \sqrt{\frac{p_t(1+c) g q_P S(t)}{2n t_{setup}^{P}}}$$

Gleichung 5-24: Optimale Anzahl von Prototyp-Einführungen pro Release im Prototypingplan-Modell

Aus Gleichung 5-24 ist zu erkennen, dass die optimale Einführungsrate sich erhöht durch

- eine erhöhte Prototyp-Anforderungsrate
- eine sinkende Produktivität und erhöhte Sensitivität
- einer erhöhten Prototyp-Effektivität (*g*)

Die optimale Einführungsrate sinkt mit steigender durchschnittlicher Einführungsdauer und der gesamten Entwicklungsdauer (*n*).

Die relative Auswirkung von Veränderungen in den Faktoren auf die Einführungsrate sinkt mit steigender Größe des Terms unter der Wurzel in Gleichung 5-24. D. h. bei

einem hohen Prototyp-Umfang pro Zyklus hat eine weitere Erhöhung des Prototyp-Umfangs geringere Auswirkungen auf die Einführungsrate als die gleiche Erhöhung bei einem niedrigen Prototyp-Umfang.

Um das Modell allgemein zu halten, wurde bis hierhin eine zeitabhängige Prototyp-Anforderungsrate verwendet. Um das Modell zu vereinfachen, wird in den weiteren Betrachtungen eine durchschnittliche Prototypanforderungsrate $q_P S / n$ bei n Releasezyklen angenommen.

Überlappung zwischen den Aktivitäten Analyse, Prototyping und Implementierung

Zur Optimierung der Überlappung zwischen Prototyping und Analyseaktivität wird die Dauer t_P zwischen Prototyp-Anforderung und Änderungen in den Kundenanforderungen abhängig von der optimalen Einführungsrate berechnet. Diese ergibt sich aus der durchschnittlichen Zeit zwischen zwei Einführungen ($1/(n\beta_P)$), der Implementierungsdauer der einzuführenden Prototypen ($s_P \cdot p_t^P$, bei einer Prototyping-Produktivität von: p_t^P) und der fixen Einführungsdauer der Prototypen (t_{setup}^P). Unter Einsetzung der optimalen Prototypeinführungsrate aus Gleichung 5-24 ergibt sich diese Dauer gemäß Gleichung 5-25 (Herleitung siehe Anhang B3).

$$t_P^* = \sqrt{\frac{2t_{setup}^P}{p_t(1+c)gq_P S}}(1 + q_P S \cdot p_t^P) + t_{setup}^P$$

Gleichung 5-25: Dauer zwischen Prototyp-Anforderung und Änderungen in den Kundenanforderungen bei optimaler Prototyp-Einführungsrate

Es wird angenommen wird, dass bei früherem Beginn des Prototypings keine zusätzlichen Überarbeitungskosten entstehen, sondern nur Nutzen durch früher entdeckte Änderungen. Daraus folgt, dass die Prototyping-Aktivität t_P^* Zeiteinheiten vor Beginn der Analyse von Änderungen in den Kundenanforderungen starten sollte. Prototypanforderungen müssen entsprechend früher gestellt werden. Gemäß Gleichung 5-25 wird die optimale überlappungsfreie Zeit t_P^* kleiner und somit die Überlappung größer, wenn

- die Einführungskosten der Prototypen sinken
- die Prototyping-Produktivität steigt (d. h. p_t^P sinkt)
- die Produktivität in der Implementierung sinkt und die Sensitivität steigt

- die Prototyp-Effektivität steigt

Durch eine erhöhte Überlappung reduziert sich die Entwicklungszeit und die Änderungen in den Kundenanforderungen werden früher erkannt, was einen geringeren Überarbeitungsaufwand zur Folge hat (vgl. Meetingplan-Modell).

Unter der Annahme, dass zwischen Prototyping-Aktivitäten und „normalen" Implementierungsaktivitäten unterschieden werden kann, kann ähnlich dem Modell aus (Loch, Terwiesch 1998) die optimale Überlappung zwischen Prototyping und Implementierung berechnet werden. Aus einer erhöhten Überlappung entsteht ein erhöhter Überarbeitungsaufwand, der aus einer erhöhten Sensitivität gegenüber Änderungen bei einer durch die Überlappung fortgeschrittenen Implementierung resultiert. Aus den Ergebnissen aus (Loch, Terwiesch 1998) folgt[148], dass eine erhöhte Sensitivität und/oder eine erhöhte Unsicherheit (hier Prototyp-Anforderungsrate) den optimalen Überlappungsgrad reduzieren. In Abschnitt 5.1.2 wird jedoch gezeigt, dass Implementierungs- und Analyseaktivität sich in den Modellen dieser Arbeit vollständig überlappen sollten. Daher sollte die Implementierungsaktivität genauso wie die Analyseaktivität t_P^* Zeiteinheiten nach der Prototypingaktivität beginnen.

Auswirkungen auf die Kommunikationsrate im Meetingplan-Modell

Die Kommunikationsrate nach dem Meetingplan-Modell wird von der Prototyping-Strategie und der daraus resultierenden Änderungsrate beeinflusst. Es wird hier das oben beschriebene vereinfachte Modell mit zeitunabhängiger Prototyp-Anforderungsrate ($q_P S/n$) verwendet. Die neue Änderungsrate der Kundenanforderungen ergibt sich gemäß Gleichung 5-26.

$$q^+ S/n = qS/n + g \cdot q_P S/n$$

Gleichung 5-26: Die Änderungsrate in Abhängigkeit der Prototyping-Strategie

Diese neue Änderungsrate kann in die Gleichungen des Meetingplan-Modells eingesetzt werden und eine optimale Kommunikationsrate in Abhängigkeit der Prototyping-Strategie bestimmt werden. Ist die Analysedauer fest vorgegeben, so werden im gleichen Zeitraum mehr Anforderungen erkannt und die Qualität des Produkts steigt.

[148] Falls Überarbeitungsaufwand nur während der Überlappung entsteht.

Zeitersparnis durch Prototyping bei konstantem Änderungsumfang

Auf dieser erhöhten Änderungsrate basierend kann ein weiterer Nutzen durch Feedback aus Prototypen untersucht werden. Unter der Voraussetzung, dass der Gesamtumfang der Änderungen im Meetingplan-Modell (qS) unabhängig vom Einsatz von Prototypen ist, kann aus einer erhöhten Änderungsrate gemäß Gleichung 5-26 gefolgert werden, dass durch Feedback aus Prototypen Änderungen in den Anforderungen früher erkannt und kommuniziert werden. Hier wird angenommen, dass eine zyklusunabhängige Prototypanforderungsrate und Änderungsrate vorliegt (*e=0*). Dann ergibt die neue Analysedauer (T_A^P), unter Einsatz von Prototyping, multipliziert mit der neuen Änderungsrate den gesamten Änderungsumfang ohne Prototyping und es folgt gemäß Gleichung 5-27.

$$q^+ S / n \cdot T_A^P = qS$$
$$\Leftrightarrow T_A^P = n \cdot q / (q + g q_P)$$

Gleichung 5-27: Die Analysedauer abhängig vom Prototyping bei konstantem Änderungsumfang

Wie zu erkennen, ist diese Analysedauer verkürzt gegenüber der Zeit ohne Prototyping (dort $T_A = n$ Zeiteinheiten), da der Term $q / (q + g q_P)$ unter den gegebenen Annahmen kleiner als Eins ist. Die Verkürzung der Analysedauer ergibt sich gemäß Gleichung 5-28.

$$T_A - T_A^P = n \cdot (1 - q / (q + g q_P))$$

Gleichung 5-28: Die Verkürzung der Analysedauer des Meetingplan-Modells durch Prototyping

Aus dieser verkürzten Analysedauer ergibt sich eine verringerte Überarbeitungsdauer aus Kommunikationsverzögerungen im Meetingplan-Modell gemäß Gleichung 5-29.

$$\int_0^{T_{A1} - T_A^P} t_{q,c}^{delay} \partial t = \int_0^{T_{A1} - T_A^P} p_t (1 + c) \cdot \frac{qS}{2n^2 \beta} \partial t$$
$$= p_t (1 + c) \cdot \frac{qS}{2n^2 \beta} (T_A - T_A^P)$$
$$= p_t (1 + c) \frac{qS}{2n\beta} \cdot (1 - \frac{q}{q + g q_P})$$

Gleichung 5-29: Verkürzte Überarbeitungsdauer durch Prototyping

Wie aus Gleichung 5-29 zu erkennen, ergeben sich erhöhte Zeiteinsparungen bei

- einer erhöhten Änderungsrate

- einer geringeren Kommunikationsrate von Anforderungsänderungen
- einer erhöhten Sensitivität
- geringerer Gesamtdauer der Entwicklung (n)
- steigender Prototyp-Anforderungsrate
- steigender Prototyp-Effektivität

Zusammenfassend lässt sich, durch die Erweiterung des Meetingplan-Modells, um Prototyping, feststellen, dass aus einer erhöhten Prototyp-Anforderungsrate und dem dadurch erhöhten Feedback durch Prototypen eine erhöhte Kommunikationsrate resultiert und Überarbeitungsaufwand aus Kommunikationsverzögerungen eingespart wird.

Kapitel 6: Abschließende Betrachtung

Die Ergebnisse dieser Arbeit unterstützen Projektmanager bei der Planung von Softwareentwicklungsprozessen in unsicheren Umgebungen unter Berücksichtigung von zyklischem Nutzerfeedback. In dieser Arbeit werden dazu die Struktur inkrementeller Entwicklungsprozesse und die Wirkungszusammenhänge zwischen Planungsmöglichkeiten, wie Zeitpunkt und Umfang von Feedback, und Einflussfaktoren, wie die Sensitivität gegenüber Änderungen, dieser Prozesse untersucht. Anhand der, aus diesen Untersuchungen resultierenden, Modellen kann die Prozessplanung hinsichtlich einer Reduzierung der Entwicklungszeit optimiert werden.

Das anfänglich definierte, einheitliche *Begriffssystem* dient dazu, einen Übergang zwischen verschiedenen populären Methodiken und Begriffen der Softwareentwicklung und den hier entwickelten Modellen zu finden. Dieses Begriffssystem wird auf einer Makroebene des Softwareentwicklungsprozesses definiert, so dass die Begriffe verschiedener spezifischer Methodiken, wie dem „Extreme Programming“, dem „Rational Unified Process“ und Microsofts „Synch-and-Stabilize“, diesem zugeordnet werden können. Der Definitionsprozess dieses Begriffssystems kann als Top-Down/Bottom-Up-Ansatz bezeichnet werden, da es aus der Betrachtung allgemeiner Prozessmodelle und verschiedener spezifischer Methodiken resultiert.

Die Schwierigkeit und die Notwendigkeit, ein solches Begriffssystem zu definieren, resultiert aus der Verwendung gleichnamiger Begriffe für unterschiedliche Konzepte in der Literatur. So bezieht sich der Begriff der „Iteration“ je nach Betrachtungsweise des Autors auf die wiederholte Entwicklung von (internen) Builds *oder* (externen) Releases, wobei auch der Release-Begriff selber nicht einheitlich verwendet wird. Aus diesem Grund werden in dieser Arbeit die verschiedenen *Zyklusebenen* klar voneinander abgegrenzt. Die in dieser Arbeit untersuchte Releaseebene bezieht sich auf die zyklische Entwicklung von Softwareinkrementen und die Einführung dieser Inkremente in die Nutzungsumgebung.

Weiterhin liefert diese Arbeit eine Differenzierung der, in wissenschaftlicher Literatur zu Softwareentwicklungsprozessen teilweise synonym verwendeten, Begriffe der *Phase* und der *Aktivität*. Diese synonyme Verwendung stammt vermutlich historisch aus der Beschreibung plangetriebener Prozessmodelle, wie dem Wasserfallmodell, in denen jeder Phase genau eine Aktivität zugeordnet werden kann. Eine Unterscheidung von Phasen

und Aktivitäten ist jedoch für die Betrachtung inkrementeller Prozessmodelle notwendig. In inkrementellen (bzw. „überlappenden“ oder „nebenläufigen“) Prozessen können in jeder Phase verschiedene Aktivitäten stattfinden. Die Modelle dieser Arbeit dienen der Planung der Aktivitäten in der, hier definierten, *Konstruktionsphase*.

In dieser Arbeit werden diverse Planungsmöglichkeiten inkrementeller Entwicklungsprozesse identifiziert, wie die Planung der *Anzahl* von Release- und Prototypingzyklen, der *Zeitpunkte* von Meetings und Feedback oder die sogenannte *Releaseplanung* (die Planung der Verteilung der Anforderungen auf die Entwicklungszyklen). Diese Planungsmöglichkeiten werden hinsichtlich einer Optimierung der Entwicklungszeit untersucht. Weitere hier nicht betrachtete Planungsmöglichkeiten können sich, neben der Releaseebene, auch auf andere Ebenen beziehen, z. B. die Planung der Integrationszeitpunkte in Buildzyklen. Die Ergebnisse der Modelle dieser Arbeit können möglicherweise auf andere Ebenen übertragen werden, diese Möglichkeit sollte jedoch an anderer Stelle untersucht werden.

Zur Planung der Entwicklung in unsicheren Umgebungen werden in Kapitel 3 verschiedene Einflussfaktoren des *Entwicklungsaufwands* und der *Entwicklungszeit* untersucht. Der *Umfang* des Softwaresystems, dessen Messung in der Softwareentwicklung durch verschiedene Metriken unterstützt wird, wird als signifikanter Einflussfaktor des Entwicklungsaufwands identifiziert. Je nach Fortschritt des Entwicklungsprozesses können unterschiedliche Metriken zur Messung dieses Umfangs angewendet werden, so dass eine Planung anhand umfangsbasierter Modelle praktikabel erscheint.

Des Weiteren wird in dieser Arbeit das Verhältnis zwischen *Überarbeitungsaufwand* und *Änderungsumfang*, welches hier als *Sensitivität* bezeichnet wird, und die Abhängigkeit dieses Faktors von der Systemgröße als zentral für die Planung der Entwicklung in unsicheren Umgebungen identifiziert. Aus den hier vorgestellten Untersuchungen folgt, dass, in *komplexen Systemen*, frühzeitiges *Feedback* aus der Nutzung die Anforderungsunsicherheit und damit den Entwicklungsaufwand reduziert. Es wird gezeigt, dass aus Anforderungsabhängigkeiten resultierende Folgeunsicherheiten und Folgefehler durch Feedback reduziert werden können.

Die Betrachtung verschiedener Arten von in der SW-Entwicklung untersuchten *Kostenfunktionen* in Kapitel 4 zeigt, dass die Zusammenhänge zwischen den Faktoren

Produktumfang, Produktivität und Entwicklungsaufwand nicht eindeutig sind. Als wesentliches Unterscheidungsmerkmal zwischen den Kostenfunktionen wird die Modellierung von *Skaleneffekten* identifiziert und es werden verschiedene Argumente aus der Literatur sowohl für positive als auch für negative Skaleneffekte in der SW-Entwicklung vorgestellt und analysiert.

Die, in dieser Arbeit durchgeführte, Analyse von in der Literatur verwendeten Kostenfunktionen zur Modellierung zyklischer inkrementeller Prozesse, zeigt die Schwächen und Stärken verschiedener Modelle. Kritisch betrachtet wird hier z. B. die Behandlung von Skaleneffekten durch die Kostenfunktion aus (Benediktsson et al. 2003). Die dieser Kostenfunktion für inkrementelle Prozesse zugrundeliegende Annahme, dass der Architekturaufwand, welcher negative Skaleneffekte reduziert, von der Anzahl der Zyklen abhängt, scheint nicht plausibel. Vorteilhafte architektonische Gestaltungsmaßnahmen sollten auch in ein-zyklischen Prozessen durchgeführt werden.

In dieser Arbeit werden dagegen nur Vor- und Nachteile inkrementeller Prozesse betrachtet, die unmittelbar mit deren Planungsmöglichkeiten zusammenhängen. Die in den Modellen dieser Arbeit getroffene Annahme, dass jeder Zyklus vom Produktumfang unabhängige Kosten (*Fixkosten*) verursacht, führt z. B. dazu, dass in jedem Zyklus dem Nutzen durch Feedback – durch die Reduzierung von Folgeunsicherheiten – auch Feedbackkosten entgegenstehen. Diese Kosten-Nutzen-Abwägung kann durch die Modelle dieser Arbeit durchgeführt werden und es kann ein *Optimum* für die Anzahl der Zyklen hinsichtlich einer Reduzierung der Entwicklungszeit gefunden werden.

Aus der Analyse der Wirkungszusammenhänge zwischen verschiedenen Einflussfaktoren in der Kostenfunktion des *Releaseplan-Modells* folgt, dass die optimale Anzahl von Zyklen mit erhöhter Feedbackeffektivität, Unsicherheit, Sensitivität und erhöhtem Produktumfang ansteigt und dieses Optimum mit steigenden Fixkosten sinkt.

Neben Feedback-Effekten kann im umfangsbasierten *Releaseplan-Modell* die optimale Verteilung des Anforderungsumfangs auf die Releasezyklen – die Releaseplanung – untersucht werden. Diese Untersuchung führt zu dem Schluss, dass durch die Releaseplanung die Entwicklungsdauer gesenkt wird, wenn in den ersten Zyklen kleinere Inkremente als in den späteren Zyklen entwickelt werden. Diese Aussage gilt unter der Annahme, dass – vergleichbar dem Spiralmodell aus (Boehm 1986) – die „kritischsten“

Anforderungen zuerst entwickelt werden und unter der Voraussetzung eines von der Releaseplanung unabhängigen Gesamtumfangs.

Mit Hilfe des zeitbasierten *Meetingplan-Modells* werden Wirkungszusammenhänge zwischen *Verzögerungen* in der Kommunikation von Anforderungen und der Überarbeitungdauer analysiert. Diese Analyse zeigt, dass die Kostenreduktion durch eine erhöhte *Kommunikationsrate* eine, mit der Entwicklungsdauer steigende, Sensitivität des (wachsenden) Systems gegenüber Änderungen voraussetzt. Ohne eine steigende Sensitivität könnten alle Änderungen in den Anforderungen während der Entwicklung gesammelt und am Ende der Konstruktionsphase *einmal* ohne Erhöhung des Überarbeitungsaufwands kommuniziert und umgesetzt werden. Der Vorteil dieser Kommunikationsstrategie wäre, dass nur einmal Kosten durch Kommunikation dieser Änderungen – z. B. in JAD-Workshops – entstehen.

Die Erweiterung des Meetingplan-Modells um einen *Prototypingzyklus*, führt dazu, dass die Auswirkungen von frühzeitigem Feedback aus Prototypen auf die Entwicklungszeit betrachtet werden kann. Dabei werden – ähnlich dem Releaseplan-Modell – die Feedbackkosten, die durch die Einführung der Prototypen in die Nutzungsumgebung entstehen, den Kosten, die durch eine Verzögerung des Feedbacks entstehen, entgegengestellt. Es wird angenommen, dass eine Verzögerung des Feedbacks bei steigender Sensitivität einen erhöhten Überarbeitungsaufwand zur Folge hat.

Durch die Modellierung eines Prototypingzyklus werden verschiedene Wirkungszusammenhänge zwischen Prototyping-Intensität, Prototyp-Effektivität und Entwicklungsdauer beobachtet. Durch eine erhöhte Prototyping-Intensität, d. h. eine erhöhte Anzahl von Prototypen pro Zyklus und eine erhöhte Prototyp-Effektivität, d. h. eine erhöhte Anzahl von Anforderungsänderungen pro Prototyp, sinkt die Analysedauer.[149] Aus einer verkürzten intensiveren Analyseaktivität folgt, wie oben bereits beschrieben, bei steigender Sensitivität ein verringerter Überarbeitungsaufwand.

Schlussfolgerung

In der agilen XP-Methodik nach (Beck 1999) wird davon ausgegangen, dass die Sensitivität mit wachsendem System konstant bleibt. Unter dieser Annahme ist im Meetingplan-Modell *kein* Vorteil eines frühzeitigen, zyklischen Feedbacks ersichtlich,

[149] Dies gilt aus Gründen der Vergleichbarkeit unter der Voraussetzung, dass die Gesamtzahl der Änderungen unabhängig von der Durchführung des Prototypings ist.

wie es jedoch in (Beck 1999) gefordert wird. Im Meetingplan-Modell basiert der Vorteil eines frühzeitigen Feedbacks auf einer steigenden Sensitivität. Daraus folgte in dieser Arbeit die Frage, warum die von (Beck 1999) geforderten kurzen (Feedback-)Zyklen durchgeführt werden sollten. Grund dafür können – wie im Releaseplan-Modell gezeigt – Abhängigkeiten zwischen Anforderungen sein.

Ein Vergleich der Modelle zeigt, dass im Releaseplan-Modell die Wirkung von Feedback auf den Änderungsumfang modelliert wird und durch Feedback, bei Anforderungsabhängigkeiten, der Änderungsumfang in späteren Inkrementen reduziert werden kann. Im Meetingplan-Modell wird der Zeitpunkt des Feedbacks und der darauf basierenden Überarbeitungsaktivität modelliert. Frühzeitiges Feedback reduziert den Überarbeitungsaufwand, falls der Überarbeitungsaufwand durch die Systemkomplexität im Laufe der Entwicklung überproportional steigt.

Aus der Analyse verschiedener Kostenfunktionen und eigenen Modellierungsansätzen wird ersichtlich, welche Annahmen getroffen werden müssen, um zu bestimmten, scheinbar intuitiven, Aussagen zu gelangen. Diese Annahmen basieren zum Teil auf der bewussten Vernachlässigung von Zusammenhängen, um analytisch, eindeutige Handlungsempfehlungen herleiten zu können. Die Diskussion dieser Annahmen erlaubt es, die hier entwickelten Modelle und andere Modelle der Literatur kritisch zu hinterfragen. Die den Modellen zugrundeliegenden Annahmen werden argumentativ oder mit Untersuchungen anderer Autoren begründet. Eine, auf dieser Arbeit basierende, spezifische empirische Untersuchung der Modellannahmen kann dabei helfen, die gefolgerten Aussagen zu validieren.

Eine Planung auf Basis der hier vorgestellten Modelle ist statisch, d. h. es wird nicht berücksichtigt, ob die Planung im Laufe der Entwicklung angepasst werden muss. Nach (McConnell 1998) kann die Genauigkeit der Modelle und die Planung *während* der Entwicklung durch eine adaptiven Analyse der Einflussfaktoren (vgl. Hearty et al. 2009, Hericko, Zivkovic 2008, Kojima et al. 2008) verbessert werden. Basierend auf den hier vorgestellten Modellen können Schwellwerte definiert werden, um *steuernd* in den Entwicklungsprozess einzugreifen. Ein solcher Ansatz zur Steuerung der Integrationszeitpunkte von Builds findet sich in dem Modell aus (Qi Feng et al. 2008), welches ebenso wie das Meetingplan-Modell auf (Loch, Terwiesch 1998) basiert.

Für die Praxis interessant dürften die hier gewonnenen Einblicke in die Wirkungszusammenhänge zwischen Anforderungsunsicherheit, Komplexität, Kommunikationskosten, Überarbeitungsaufwand und Feedback sein. Die, durch die Modelle hergeleiteten, Empfehlungen zur Planung von Entwicklungszyklen können dabei helfen, die Entwicklungszeit zu reduzieren. Manager könnten analysieren – sofern ein inkrementeller Entwicklungsprozess möglich ist – auf welcher Ebene (Release, Build, etc.) im Entwicklungsprozess zyklisch Kosten entstehen, die sich zunächst nicht weiter reduzieren lassen und so hoch sind, dass es sich lohnt Metriken einzuführen, um die Planung des Entwicklungsprozesses zu optimieren.[150]

---

[150] Unter Berücksichtigung der, den Metriken inhärenten Unsicherheiten.

Literaturverzeichnis

Abdel-Hamid, Tarek K.; Madnick, Stuart E. (1989) Lessons learned from modeling the dynamics of software development. In: Communications of the ACM, Jg. 32, H. 12, 1989, S. 1426–1438. Online verfügbar unter http://doi.acm.org/10.1145/76380.76383.

Abran, Alain; Desharnais, Jean-Marc; Oligny, Serge; St-Pierre, Denis; Symons, Charles (2003) Cosmic-FFP Measurement Manual. The Cosmic Implementation Guide For ISO/IEC 19761. Version 2.2. Online verfügbar unter http://www.ifpug.org/discus/messages/1751/full_function_points_new-4835.pdf, zuletzt geprüft am 29.04.2010.

Adler, Paul S. (1995) Interdepartmental Interdependence and Coordination: The Case of the Design/Manufacturing Interface. In: Organization Science, Jg. 6, H. 2, 1995, S. 147–167. Online verfügbar unter http://www.jstor.org/stable/2635119.

Adler, Paul S.; Mandelbaum, Avi; Nguyen, Vien; Schwerer, Elizabeth (1995) From Project to Process Management: An Empirically-Based Framework for Analyzing Product Development Time. In: Management Science, Jg. 41, H. 3, 1995, S. 458–484. Online verfügbar unter http://www.jstor.org/stable/2632976.

Akao, Y. (1972) New product development and quality assurance: quality deployment system. (in Japanese). In: Standardization and Quality Control, Jg. 25, H. 4, 1972, S. 7–14.

Albrecht, Allan J.; Gafeney Jr., John E. (1983) Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. In: IEEE Transactions on Software Engineering, Jg. 9, H. 6, 1983, S. 639–648.

Allen, Edward; Gottipati, Sampath; Govindarajan, Rajiv (2007) Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. In: Software Quality Journal, Jg. 15, H. 2, 2007, S. 179–212.

Bahsoon, R.; Emmerich W. (2003) Evaluating software architectures: development, stability, and evolution. In: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications. Tunis, Tunesia, 14 - 18 July 2003. Piscataway, NJ, USA: IEEE Operations Center, S. 47–51.

Baldwin, Carliss Y.; Clark, Kim B (2000) The power of modularity. Cambridge, Mass., USA, 2000: MIT Press (Design rules, 1).

Balzert, Helmut (2001) Lehrbuch der Software-Technik. 2. Aufl. Heidelberg, 2001: Spektrum Akademischer Verlag (Lehrbücher der Informatik).

Banker, Rajiv D.; Davis, Gordon B.; Slaughter, Sandra A. (1998) Software development practices, software complexity, and software maintenance performance: A field study. In: Management Science, Jg. 44, H. 4, 1998, S. 433.

Banker, Rajiv D.; Kemerer, Chris F. (1989) Scale Economies in New Software Development. In: IEEE Transactions on Software Engineering, Jg. 15, H. 10, 1989, S. 1199.

Basili, Victor R.; Briand, Lionel C.; Condon, S.; Kim, Yong-Mi; Melo, W. L.; Valen, J. D. (1996) Understanding and predicting the process of software maintenance releases. In: Proceedings of the 18th International Conference on Software Engineering. March 25 - 29, 1996, Berlin, Germany. Los Alamitos, Calif., USA: IEEE Computer Soc. Press, S. 464-464.

Baskerville, Richard; Levine, Linda; Pries-Heje, Jan; Ramesh, Balasubramaniam; Slaughter, Sandra A. (2002) Balancing Quality and Agility in Internet Speed Software Development. In: Proceedings of the Twenty-Third International Conference on Information Systems (ICIS). Barcelona, Spain, S. 859–864.

Baskerville, Richard; Pries-Heje, Jan (2004) Short cycle time systems development. In: Information Systems Journal, Jg. 14, H. 3, 2004, S. 237–264.

Bass, Len; Clements, Paul; Kazman, Rick (2005) Software architecture in practice. 2. Aufl. Boston, Munich, 2005: Addison-Wesley (SEI series in software engineering).

Bayes, Thomas (1764) An Essay Toward Solving a Problem in the Doctrine of Chances. In: Philosophical Transactions of the Royal Society of London, Jg. 53, 1764, S. 370–418.

Beck, Kent (1999) Embracing Change with Extreme Programming. In: Computer, Jg. 32, H. 10, 1999, S. 70–77.

Beck, Kent (2000) Extreme programming explained. Embrace change. Reading, Mass., 2000: Addison-Wesley (The XP series).

Beck, Kent; Cockburn, Alistair; Cunningham, Ward; Fowler, Martin; Highsmith, Jim (2001)Manifesto for Agile Software Development. Online verfügbar unter http://agilemanifesto.org, zuletzt aktualisiert am 2001, zuletzt geprüft am 04.09.2009.

Benediktsson, Oddur; Dalcher, Darren; Reed, Karl; Woodman, Mark (2003) COCOMO-Based Effort Estimation for Iterative and Incremental Software Development. In: Software Quality Journal, Jg. 11, H. 4, 2003, S. 265–281.

Bennett, Keith H.; Rajlich, Vaclav T. (2000) Software maintenance and evolution: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering. Limerick, Ireland: ACM, S. 73–87.

Bieman, J. M.; Andrews, A. A.; Yang, H. J. (2003) Understanding change-proneness in OO software through visualization. In: Proceedings of the 11th IEEE International Workshop on Program Comprehension. Los Alamitos, Calif.: IEEE Computer Society, S. 44–53.

Boehm, Barry W. (1976) Software Engineering. In: IEEE Transactions on Computers, Jg. 25, H. 12, 1976, S. 1226–1241.

Boehm, Barry W. (1984) Software Engineering Economics. In: IEEE Transactions on Software Engineering, Jg. 10, H. 1, 1984, S. 4–21.

Boehm, Barry W. (1986) A spiral model of software development and enhancement. In: SIGSOFT Software Engineering Notes, Jg. 11, H. 4, 1986, S. 14–24. Online verfügbar unter http://doi.acm.org/10.1145/12944.12948.

Boehm, Barry W. (2000a) Requirements that Handle IKIWISI, COTS, and Rapid Change. In: Computer, Jg. 33, H. 7, 2000, S. 99–102. Online verfügbar unter http://dx.doi.org/10.1109/2.869384.

Boehm, Barry W. (2000b) Software cost estimation with COCOMO II. Upper Saddle River, NJ, 2000: Prentice Hall PTR.

Boehm, Barry W. (2002) Get Ready for Agile Methods, with Care. In: Computer, Jg. 35, H. 1, 2002, S. 64–69. Online verfügbar unter http://doi.ieeecomputersociety.org/10.1109/2.976920.

Boehm, Barry W.; Abts, Chris; Chulani, Sunita (2000) Software development cost estimation approaches - A survey. In: Annals of Software Engineering, Jg. 10, H. 1-4, 2000, S. 177–205.

Boehm, Barry W.; Papaccio, P. N. (1988) Understanding and Controlling Software Costs. In: IEEE Transactions on Software Engineering, Jg. 14, H. 10, 1988, S. 1462–1477.

Bratthall, L.; Runeson, P. (2000) A survey of lead-time challenges in the development and evolution of distributed real-time systems. In: Information & Software Technology, Jg. 42, H. 13, 2000, S. 947.

Brooks, Frederick P. (1975) The mythical man-month. Essays on software engineering. Reading, Mass., 1975: Addison-Wesley.

Brown, Bonny S.; Edwards, Royce; Fischer, E. Jay; Gamus, David; Russac, Janet; Timp, Adri; Thomas, Peter (2010) Function Point Counting Practices Manual. Release 4.3.1. Herausgegeben von International Function Points Users Group (IFPUG). NJ, USA. Online verfügbar unter http://www.ifpug.org/publications/CPM%204.3.1%20TOC%20Excerpts.pdf, zuletzt geprüft am 29.04.2010.

Browning, Tyson R.; Ramasesh, Ranga V. (2007) A Survey of Activity Network-Based Process Models for Managing Product Development Projects. In: Production & Operations Management, Jg. 16, H. 2, 2007, S. 217–240.

Bundesgerichtshof (BGH): Urteil vom 06.07.2000, Aktenzeichen I ZR 244/97. In: JurPC Web-Dok., Jg. 220, 1-36.

Camerer, Colin; Weber, Martin (1992) Recent Developments in Modeling Preferences: Uncertainty and Ambiguity. In: Journal of Risk & Uncertainty, Jg. 5, H. 4, 1992, S. 325–370.

Cant, S. N.; Jeffery, D. R.; Henderson-Sellers, B. (1995) A conceptual model of cognitive complexity of elements of the programming process. In: Information and Software Technology, Jg. 37, H. 7, 1995, S. 351–362. Online verfügbar unter http://www.sciencedirect.com/science/article/B6V0B-3YS2CDK-12/2/22bf23a9a547b16f9a0316e5379672ba.

Chow, Tsun; Cao, Dac-Buu (2008) A survey study of critical success factors in agile software projects. Agile Product Line Engineering. In: Journal of Systems and Software, Jg. 81, H. 6, 2008, S. 961–971. Online verfügbar unter http://www.sciencedirect.com/science/article/B6V0N-4PHBWP9-2/2/bb076ac0abc4145c48db822da0a8da52.

Chulani, Sunita; Boehm, Barry W.; Steece, Bert (1999) Bayesian Analysis of Empirical Software Engineering Cost Models. In: IEEE Transactions on Software Engineering, Jg. 25, H. 4, 1999, S. 573.

Cobb, Charles W.; Douglas, Paul H. (1928) A THEORY OF PRODUCTION. In: American Economic Review, Jg. 18, 1928, S. 139.

Cockburn, Alistair (2007) Agile software development. The cooperative game. 2. Aufl. Upper Saddle River, NJ, 2007: Addison-Wesley (The Agile software development series).

Cohn, Mike (2006) Agile estimating and planning. Upper Saddle River, NJ, 2006: Prentice Hall PTR (Robert C. Martin series).

Conte, Samuel Daniel; Dunsmore, H. E.; Shen, V. Y. (1986) Software engineering metrics and models. Menlo Park, Calif., 1986: Benjamin/Cummings (Benjamin/Cummings series in software engineering).

Cooper, Kenneth G.; Lyneis, James M.; Bryant, Benjamin J. (2002) Learning to learn, from past to future. In: International Journal of Project Management, Jg. 20, H. 3, 2002, S. 213.

Cooper, Robert G. (2008) Winning at new products. Accelerating the process from idea to launch. 3. ed, repr. New York, 2008: Basic Books.

Coupaye, T.; Estublier, J. (2000) Foundations of enterprise software deployment. In: Ebert, Jürgen (Hg.): Proceedings of the Fourth European Conference on Software Maintenance and Reengineering. Reengineering Week Zurich, University of Zurich, Switzerland, February 29 - March 3, 2000. Los Alamitos, Calif.: IEEE Computer Society, S. 65–73.

Curtis, Bill; Krasner, Herb; Iscoe, Neil (1988) A field study of the software design process for large systems. In: Communications of the ACM, Jg. 31, H. 11, 1988, S. 1268–1287. Online verfügbar unter http://doi.acm.org/10.1145/50087.50089.

Cusumano, Michael A.; Selby, Richard W. (1995) Microsoft secrets. How the world's most powerful software company creates technology, shapes markets, and manages people. New York, 1995: Simon & Schuster (A Touchstone book).

Cusumano, Michael A.; Selby, Richard W. (1997) How Microsoft Builds Software. In: Communications of the ACM, Jg. 40, H. 6, 1997, S. 53–61.

Davidson, E. J. (1999) Joint application design (JAD) in practice. In: Journal of Systems and Software, Jg. 45, H. 3, 1999, S. 215–223. Online verfügbar unter http://www.sciencedirect.com/science/article/B6V0N-3VXYV1G-5/2/7d3d8df4409c689deb96db64b8a7b3ca.

Davies, Rachel; Sharp, Helen (2006) Early and Often: Elaborating Agile Requirements. In: Cutter IT Journal, Jg. 19, H. 7, 2006, S. 6–11.

DeMarco, Tom (1982) Controlling software projects. Management, measurement & estimation. Englewood Cliffs, N.J., 1982: Yourdon (Yourdon computing series).

Dolado, J. J. (2001) On the problem of the software cost function. In: Information & Software Technology, Jg. 43, H. 1, 2001, S. 61.

Dyba, Tore; Dingsoyr, Torgeir (2008) Empirical studies of agile software development: A systematic review. In: Information & Software Technology, Jg. 50, H. 9/10, 2008, S. 833.

Frankel, David S (2003) Model driven architecture. Applying MDA to enterprise computing. Indianapolis, Ind., 2003: Wiley (OMG Press).

Frese, Erich; Mensching, Helmut (1986) Unternehmungsführung. Landsberg am Lech, 1986: Verl. Moderne Industrie (Studienbibliothek Betriebswirtschaft).

Gartner Inc. (Hg.) (2009) Hype Cycle for Emerging Technologies, 2009. Unter Mitarbeit von Jackie Fenn, William Clark und Yefim V. Natis et al. Online verfügbar unter http://www.gartner.com/DisplayDocument?doc_cd=169368&ref=g_fromdoc.

Gerlich, R.; Denskat, U. (1994) A cost estimation model for maintenance and high reuse. In: Proceedings of the European Software Cost Modeling Conference (ESCOM). Ivrea, Italy .

Gibson, Virginia R.; Senn, James A. (1989) System structure and software maintenance performance. In: Communications of the ACM, Jg. 32, H. 3, 1989, S. 347–358. Online verfügbar unter http://doi.acm.org/10.1145/62065.62073.

Gomes, Paulo J.; Joglekar, Nitin R. (2008) Linking modularity with problem solving and coordination efforts. In: Managerial & Decision Economics, Jg. 29, H. 5, 2008, S. 443–457.

Graham (1989) Incremental development: review of nonmonolithic life-cycle development models. In: Information and Software Technology, Jg. 31, H. 1, 1989, S. 7–20.

Statistisches Bundesamt - PressestelleStatistisches Bundesamt, Informationstechnologie in Unternehmen und Haushalten 2005. Bestellnummer: 0000121-05700-1. Pressemitteilung vom Februar 2006. Wiesbaden.

Ha, Albert Y.; Porteus, Evan L. (1995) Optimal Timing of Reviews in Concurrent Design for Manufacturability. In: Management Science, Jg. 41, H. 9, 1995, S. 1431–1448.

Haferkamp, Lars; Keutel, Marcus; Mellis, Werner (2008) Alternative Reactions of the Design and Coding Process to Preliminary Information. Arbeitspapier. Seminar für Wirtschaftsinformatik und Systementwicklung, Universität zu Köln.

Hearty, Peter; Fenton, Norman; Marquez, David; Neil, Martin (2009) Predicting Project Velocity in XP Using a Learning Dynamic Bayesian Network Model. In: IEEE Transactions on Software Engineering, Jg. 35, H. 1, 2009, S. 124–137.

Hericko, Marjan; Zivkovic, Ales (2008) The size and effort estimates in iterative development. In: Information & Software Technology, Jg. 50, H. 7/8, 2008, S. 772–781.

Hossenfelder, Jörg (2010)Lünendonk- MARKTFORSCHUNG- LÜNENDONK LISTEN- IT-MARKT. Herausgegeben von Lünendonk GmbH. Online verfügbar unter http://www.luenendonk.de/IT_Markt_Liste.php, zuletzt aktualisiert am 16.03.2010, zuletzt geprüft am 06.05.2010.

Hu, Qing (1997) Evaluating Alternative Software Production Functions. In: IEEE Transactions on Software Engineering, Jg. 23, 1997, S. 379–387. Online verfügbar unter http://doi.ieeecomputersociety.org/10.1109/32.601078.

Iansiti, Marco; MacCormack, Alan (1997) DEVELOPING PRODUCTS ON INTERNET TIME. In: Harvard Business Review, Jg. 75, H. 5, 1997, S. 108–117.

IBM Corporation (2008) Case Study: IBM Business Partner Unified Process Mentors helps clients increase predictability and consistency using IBM Rational solutions. Online verfügbar unter ftp://ftp.software.ibm.com/software/rational/web/casestudy/RAC14022-USEN-00UPM.pdf, zuletzt geprüft am 28.04.2010.

ISO/IEC, 12207 (1995): Information Technology - Software Lifecycle Processes, International Standards Organisation and International Electrotechnical Committee 1995.

Jackson, Michael (1995) Problems and requirements. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering. March 27 - 29, 1995, York, England: IEEE Computer Society, S. 2–8.

Jarzombek, S. J. (1999) k.A. In: Joint Aerospace Weapon System Support, Sensors and Simulation Symposium. San Diego, California, June 13-18, 1999 .

Jones, Capers (1986) Programming Productivity. Steps Toward a Science. New York, 1986: McGraw-Hill (McGraw-Hill Series in software engineering and technology).

Jones, Capers (1998) Estimating software costs. New York, 1998: McGraw-Hill.

Jones, Capers (2008) Applied software measurement. Global analysis of productivity and quality. 3. Aufl. New York, NY, USA, 2008: McGraw-Hill.

Jootar, Jay; Eppinger, Steven D. (2002) A System Architecture-based Model for Planning Iterative Development Processes. General Model Formulation and Analysis of Special Cases. (Innovation in Manufacturing Systems and Technology (IMST)). Online verfügbar unter http://hdl.handle.net/1721.1/4042.

Jörgensen, Magne; Shepperd, Martin (2007) A Systematic Review of Software Development Cost Estimation Studies. In: IEEE Transactions on Software Engineering, Jg. 33, H. 1, 2007, S. 33–53.

Jung, Hans (2006) Allgemeine Betriebswirtschaftslehre. 10. Aufl. München, 2006: Oldenbourg.

Kiefer, Christoph; Bernstein, Abraham; Tappolet, Jonas (2007) Mining Software Repositories with iSPAROL and a Software Evolution Ontology. In: Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR 2007). Minneapolis, MN, USA, May 19-20, 2007: IEEE Computer Society, S. 10–17.

Kitchenham, Barbara A. (1992) Empirical studies of assumptions that underlie software cost-estimation models. In: Information and Software Technology, Jg. 34, H. 4, 1992, S. 211–218.

Kitchenham, Barbara A. (2002) The question of scale economies in software--why cannot researchers agree? In: Information & Software Technology, Jg. 44, H. 1, 2002, S. 13.

Knight, Frank H. (1921) Risk, uncertainty and profit. NY, Cornell University, Dissertation, Ithaca, 1916. Boston, New York, 1921: Houghton Mifflin (The London School of Economics and Political Science, 16).

Koch, Oliver (2008a) Strategieorientierte Einführung komplexer Softwaresysteme. Vorgehensmodell zur Sicherung von Wettbewerbsvorteilen und zum TCO-optimierenden Projektmanagement. 1. Aufl. Kassel, 2008: Witec-Verlag.

Koch, Stefan (2008b) Effort modeling and programmer participation in open source software projects. Empirical Issues in Open Source Software. In: Information Economics and Policy, Jg. 20, H. 4, 2008, S. 345–355.

Kojima, Tsutomu; Hasegawa, Toru; Misumi, Munechika; Nakamura, Tsuyoshi (2008) Risk analysis of software process measurements. In: Software Quality Journal, Jg. 16, H. 3, 2008, S. 361–376.

Krishnan, Viswanathan; Eppinger, Steven D. (1997) A model-based framework to overlap product development activities. In: Management Science, Jg. 43, H. 4, 1997, S. 437.

Kruchten, Philippe (2007) The rational unified process. An introduction. 3. Aufl. Upper Saddler River, NJ, 2007: Addison-Wesley (The Addison-Wesley object technology series).

Lang, Carsten (2004) Organisation der Software-Entwicklung. Probleme, Konzepte, Lösungen. Universität zu Köln, Dissertation, 2003. 1. Aufl. Wiesbaden, 2004: Deutscher Universitäts-Verlag (Gabler Edition Wissenschaft).

Larman, Craig (2004) Agile and iterative development. A manager's guide. Boston, Mass., 2004: Addison-Wesley (Agile software development series).

Larman, Craig; Basili, Victor R. (2003) Iterative and Incremental Development: A Brief History. In: Computer, Jg. 36, H. 6, 2003, S. 47–56.

Laux, Helmut (2005)Entscheidungstheorie. (Springer-Lehrbuch). Online verfügbar unter http://dx.doi.org/10.1007/b138945.

Lehman, M. M.; Ramil, J. F. (2001a) An approach to a theory of software evolution. In: Proceedings of the 4th International Workshop on Principles of Software Evolution. Vienna, Austria: ACM, S. 70–74.

Lehman, Meir M.; Belady, Laszlo A. (1985) Program evolution. Processes of software change. London, 1985: Academic Press (APIC studies in data processing, 27).

Lehman, Meir M.; Ramil, Juan F. (2001b) Rules and Tools for Software Evolution Planning and Management. In: Annals of Software Engineering, Jg. 11, H. 1, 2001.

Levary, Reuven R.; Lin, Chi Y. (1991) Modelling the software development process using an expert simulation system having fuzzy logic. In: Software: Practice and Experience, Jg. 21, H. 2, 1991, S. 133–148. Online verfügbar unter http://dx.doi.org/10.1002/spe.4380210203.

Liu, Qin; Mintram, Robert C. (2005) Preliminary Data Analysis Methods in Software Estimation. In: Software Quality Journal, Jg. 13, H. 1, 2005, S. 91–115.

Loch, Christoph H.; Solt, Michael E.; Bailey, Elaine M. (2008) Diagnosing Unforeseeable Uncertainty in a New Venture. In: Journal of Product Innovation Management, Jg. 25, H. 1, 2008, S. 28–46.

Loch, Christoph H.; Terwiesch, Christian (1998) Communication and Uncertainty in Concurrent Engineering. In: Management Science, Jg. 44, H. 8, 1998, S. 1032–1048.

Luh, Peter B.; Feng Liu; Moser, Bryan (1999) Scheduling of design projects with uncertain number of iterations. In: European Journal of Operational Research, Jg. 113, H. 3, 1999, S. 575–592.

MacCormack, Alan; Verganti, Roberto; Iansiti, Marco (2001) Developing Products on "Internet Time": The Anatomy of a Flexible Development Process. In: Management Science, Jg. 47, H. 1, 2001, S. 133–150.

MacManus, Richard (2009)Gartner Hype Cycle 2009: Web 2.0 Trending Up, Twitter Down. Online verfügbar unter http://www.readwriteweb.com/archives/gartner_hype_cycle_2009.php, zuletzt aktualisiert am August 11, 2009, zuletzt geprüft am August 12, 2009.

Mathiassen, Lars; Pedersen, Keld (2008) Managing Uncertainty in Organic Development Projects. In: Communications of AIS, Jg. 2008, H. 23, 2008, S. 483–500.

Mathiassen, Lars; Saarinen, Timo; Tuunanen, Tuure; Rossi, Matti (2007) A Contigency Model for Requirements Development. In: Journal of the Association for Information Systems, Jg. 8, H. 11, 2007, S. 569–597.

McAfee, R. Preston; McMillan, John (1995) Organizational Diseconomies of Scale. In: Journal of Economics & Management Strategy, Jg. 4, H. 3, 1995, S. 399–426. Online verfügbar unter http://ideas.repec.org/a/bla/jemstr/v4y1995i3p399-426.html.

McCabe, Thomas J. (1976) A Complexity Measure. In: IEEE Transactions on Software Engineering, Jg. 2, H. 4, 1976, S. 308–320.

McConnell, Steve (1996) Rapid development. Taming wild software schedules. Redmond, Wash., 1996: Microsoft Press.

McConnell, Steve (1998) Software project survival guide. How to be sure your first important project isn't your last. Redmond, Wash., 1998: Microsoft Press.

Mellis, Werner (2004) Projektmanagement der SW-Entwicklung. Eine umfassende und fundierte Einführung. 1. Aufl. Wiesbaden, 2004: Vieweg (Aus dem Bereich IT erfolgreich gestalten).

Microsoft Corporation (2005)Microsoft Windows Vista October Community Technology Preview Fact Sheet: October 2005. Online verfügbar unter http://www.microsoft.com/presspass/newsroom/winxp/WinVistaCTPFS.mspx, zuletzt geprüft am 28.04.2010.

NESMA (2009) Function Point Analysis for Software Enhancement. Guidelines. Version 2.2.1. Netherlands. Online verfügbar unter http://www.nesma.nl/download/boeken_NESMA/N13_FPA_for_Software_Enhancement_(v2.2.1).pdf.

Ngo-The, A.; Ruhe, G. (2009) Optimized Resource Allocation for Software Release Planning. In: IEEE Transactions on Software Engineering, Jg. 35, H. 1, 2009, S. 109–123.

Norden, P. V. (1960) On the anatomy of development projects. In: IRE Transactions on Engineering Management, Jg. 7, H. 1, 1960, S. 34–42.

Oey, Kai; Wagner, Holger; Rehbach, Simon; Bachmann, Andrea (2006) Mehr als alter Wein in neuen Schläuchen. Eine einführende Darstellung des Konzepts der serviceorientierten Architekturen. In: Aier, Stephan (Hg.): Unternehmensarchitekturen und Systemintegration. 2. Aufl. Berlin: GITO-Verlag (Enterprise Architecture, 3).

Onur Demirors; Cigdem Gencel (2009) Conceptual Association of Functional Size Measurement Methods. In: IEEE Software, Jg. 26, H. 3, 2009, S. 71–78.

Parnas, David Lorge; Clements, Paul C. (1986a) A Rational Design Process: How and Why to Fake It. In: IEEE Transactions on Software Engineering, Jg. 12, H. 2, 1986, S. 251–257.

Parnas, David Lorge; Clements, Paul C. (1986b) Correction to "A Rational Design Process: How and Why to Fake It". In: IEEE Transactions on Software Engineering, Jg. 12, H. 8, 1986, S. 874-874.

Paulk, Mark C.; Curtis, Bill; Chrissis, Mary Beth; Weber, Charles V. (1993) Capability Maturity Model, Version 1.1. In: IEEE Software, Jg. 10, H. 4, 1993, S. 18.

Perrow, Charles (1967) A Framework for the Comparative Analysis of Organizations. In: American Sociological Review, Jg. 32, H. 2, 1967, S. 194–208. Online verfügbar unter http://www.jstor.org/stable/2091811.

Pew, Richard W.; Mavor, Anne S. (2007) Human-system integration in the system development process. A new look. Washington, DC, 2007: National Academics Press.

Pich, Michael T.; Loch, Christoph H.; Meyer, Arnoud de (2002) On Uncertainty, Ambiguity, and Complexity in Project Management. In: Management Science, Jg. 48, H. 8, 2002, S. 1008–1023.

Pickard, Lesley; Kitchenham, Barbara; Jones, Peter (1999) Comments on: Evaluating Alternative Software Production Functions. In: IEEE Transactions on Software Engineering, Jg. 25, H. 2, 1999, S. 282–285.

Premraj, Rahul; Shepperd, Martin; Kitchenham, Barbara A.; Forselius, Pekka (2005) An Empirical Analysis of Software Productivity over Time. In: Proceedings of the 11th IEEE International Software Metrics Symposium: IEEE Computer Society, S. 37.

Putnam, Lawrence H. (1978) A General Empirical Solution to the Macro Software Sizing and Estimating Problem. In: IEEE Transactions on Software Engineering, Jg. 4, H. 4, 1978, S. 345–361.

Qi Feng; Mookerjee, Vijay S.; Sethi, Suresh P. (2008) Application Development Using Fault Data. In: Production & Operations Management, Jg. 17, H. 2, 2008, S. 162–174.

Rakitin, Steven R. (2001) Letters. Manifesto Elicits Cynicism. In: Computer, Jg. 34, H. 12, 2001, S. 4.

Rauterberg, Matthias (1992) AN ITERATIVE-CYCLIC SOFTWARE PROCESS MODEL. In: Proceedings of the Fourth International Conference on Software Engineering and Knowledge Engineering. SEKE 92, June 15 - 20, 1992, Europa Palace Hotel, Capri, Italy. Los Alamitos, Calif., USA: IEEE Computer Society Press, S. 600–607.

Royce, W. W. (1970) Managing the development of large software systems: concepts and techniques. In: Proceedings of the IEEE WESCON. Los Angeles, August 1970. Los Alamitos, United States, S. 1–9.

Rumelhart, D. E.; Hinton, G. E.; Williams, R. J. (1986) Learning internal representations by error propagation. In: Rumelhart, D. E.; McClelland, James L.; the PDP Research Group (Hg.): Parallel distributed processing: explorations in the microstructure of cognition: MIT Press (1 - foundations), S. 318–362.

Savage, Leonard J. (1972) The foundations of statistics. 2. Aufl. New York, NY, USA, 1972: Dover Publications.

Seibt, Dietrich (1972) Organisation von Software-Systemen. Wiesbaden, 1972: Gabler (Betriebswirtschaftliche Beiträge zur Organisation und Automation, 18).

Sellier, David; Mannion, Mike; Mansell, Jason Xabier (2008) Managing requirements inter-dependency for software product line derivation. In: Requirements Engineering, Jg. 13, H. 4, 2008, S. 299–313.

Serich, Scott (2005) Prototype Stopping Rules in Software Development Projects. In: IEEE Transactions on Engineering Management, Jg. 52, H. 4, 2005, S. 8p.

Shore, Jim (2004) Continuous Design. In: IEEE Software, Jg. 21, H. 1, 2004, S. 20–22.

Simon, Herbert A. (1962) The Architecture of Complexity. In: Proceedings of the American Philosophical Society, Jg. 106, H. 6, 1962, S. 467–482. Online verfügbar unter http://www.jstor.org/stable/985254.

Simon, Herbert A. (1969) The science of the artificial. Cambridge, Mass., USA, 1969: MIT Press.

Sircar, Sumit; Nerur, Sridhar P.; Mahapatra, Radhakanta (2001) Revolution or Evolution? A Comparison of Object-Oriented and Structured Systems Development Methods. In: MIS Quarterly, Jg. 25, H. 4, 2001, S. 457–471. Online verfügbar unter http://www.jstor.org/stable/3250991.

Sommerville, Ian (2007) Software engineering. 8. Aufl. Harlow, England, 2007: Addison-Wesley (International computer science series).

Srinivasan, Krishnamoorthy; Fisher, Douglas (1995) Machine Learning Approaches to Estimating Software Development Effort. In: IEEE Transactions on Software Engineering, Jg. 21, H. 2, 1995, S. 126.

Stahlknecht, Peter; Hasenkamp, Ulrich (2005) Einführung in die Wirtschaftsinformatik. 11. Aufl. Berlin [u.a.], 2005: Springer (Springer-Lehrbuch).

Steward, D. V. (1981) The Design Structure System: A Method for Managing the Design of Complex Systems. In: IEEE Transactions on Engineering Management, Jg. 28, H. 3, 1981, S. 71–74.

Terwiesch, Christian; Loch, Christoph H. (1999) Measuring the Effectiveness of Overlapping Development Activities. In: Management Science, Jg. 45, H. 4, 1999, S. 455–465.

Terwiesch, Christian; Loch, Christoph H.; Meyer, Arnoud de (2002) Exchanging Preliminary Information in Concurrent Engineering: Alternative Coordination Strategies. In: Organization Science, Jg. 13, H. 4, 2002, S. 402–419.

Thomke, Stefan H. (1998) Managing Experimentation in the Design of New Products. In: Management Science, Jg. 44, H. 6, 1998, S. 743–762.

Tokuda, Lance; Batory, Don (2001) Evolving Object-Oriented Designs with Refactorings. In: Automated Software Engineering, Jg. 8, H. 1, 2001, S. 89–120.

US National Aeronautics and Space Admin (2008) NASA Systems Engineering Handbook, 2008.

van der Hoek, Andre; Wolf, Alexander L. (2003) Software release management for component-based software. In: Software - Practice and Experience, Jg. 33, H. 1, 2003, S. 77–98.

Ven, Andrew H. Van de; Delbecq, Andre L. (1974) A Task Contingent Model of Work-Unit Structure. In: Administrative Science Quarterly, Jg. 19, H. 2, 1974, S. 183–197. Online verfügbar unter http://www.jstor.org/stable/2393888.

Walters, A. A. (1963) Production and Cost Functions: An Econometric Survey. In: Econometrica, Jg. 31, H. 1/2, 1963, S. 1–66. Online verfügbar unter http://www.jstor.org/stable/1910949.

Walz, Diane B.; Elam, Joyce J.; Curtis, Bill (1993) Inside a software design team: knowledge acquisition, sharing, and integration. In: Commun. ACM, Jg. 36, H. 10, 1993, S. 63–77. Online verfügbar unter http://doi.acm.org/10.1145/163430.163447.

Wang, Qing; Lai, Xufang (2001) Requirements management for the incremental development model. In: Yu, Y T (Hg.): Proceedings. Second Asia-Pacific Conference on Quality Software: 10 - 11 December 2001, Hong Kong. Los Alamitos, Calif.: IEEE Computer Society, S. 295–301.

Wikipedia - Die freie Enzyklopädie (Hg.) (2010)Development of Windows Vista. Wikimedia Foundation. Online verfügbar unter http://en.wikipedia.org/wiki/Development_of_Windows_Vista, zuletzt aktualisiert am 26.04.2010, zuletzt geprüft am 28.04.2010.

Woodfield, S. N.; Dunsmore, H. E.; Shen, V. Y. (1981) The effect of modularization and comments on program comprehension. In: Proceedings of the 5th international

conference on Software engineering. San Diego, California, United States: IEEE Press, S. 215–223.

Yau, S. S.; Collofello, J. S.; MacGregor, T. (1978) Ripple effect analysis of software maintenance. In: Computer Software and Applications Conference. COMPSAC '78. The IEEE Computer Society's Second International, 1978, S. 60–65.

Zave, Pamela; Jackson, Michael (1997) Four dark corners of requirements engineering. In: ACM Transactions on Software Engineering and Methodology (TOSEM), Jg. 6, H. 1, 1997, S. 1–30. Online verfügbar unter http://doi.acm.org/10.1145/237432.237434.

Anhang

A. Rechnungen zum Releaseplan-Modell

A1. Herleitung des gesamten Überarbeitungsumfangs bei Feedback-Effekten

Der gesamte Überarbeitungsumfang bei Feedback-Effekten ergibt sich aus der Summierung der Änderungsumfänge aus Gleichung 5-1 über die Zyklen.

$$
\begin{aligned}
& qS/n\sum_{i=1}^{n}(1-d\frac{i-1}{n}) \\
& = qS/n\cdot\left(n+d-\frac{d}{n}\sum_{i=1}^{n}i\right) \\
& = qS/n\cdot\left(n+d-\frac{d}{n}\frac{(n+1)n}{2}\right) \\
& = qS/n\cdot\left(n(1-\frac{1}{2}d)+\frac{1}{2}d\right) \\
& = qS\cdot\left(1-\frac{1}{2}d(1-\frac{1}{n})\right)
\end{aligned}
$$

A2. Herleitung der Bedingung für die Feedbackeffektivität

Aus der Bedingung, dass der gesamte Änderungsumfang nicht negativ werden darf, lässt sich eine Bedingung für die Feedbackeffektivität *d* herleiten.

$$
\begin{aligned}
& qS\cdot\left(1-\frac{1}{2}d(1-\frac{1}{n})\right)\geq 0 \\
& \Leftrightarrow 1\geq\frac{1}{2}d(1-\frac{1}{n}) \\
& \Leftrightarrow \frac{2}{1-1/n}\geq d \\
& \Rightarrow 4\geq d, n\geq 2
\end{aligned}
$$

A3. Herleitung der optimalen Anzahl von Zyklen bei Feedback-Effekten

Die Optimierung der von der Anzahl der Zyklen abhängigen Entwicklungszeit

$$T_{q,c} + nt_f = p_t(1+c)S \cdot q\left(1 - \frac{1}{2}d(1 - \frac{1}{n})\right) + nt_f$$

bezüglich der Anzahl der Zyklen n ergibt

$$\left(p_t(1+c)S \cdot q \cdot \left(1 - \frac{1}{2}d(1 - \frac{1}{n})\right) + nt_f \right)\partial / \partial n \overset{!}{=} 0$$

$$\Leftrightarrow \left(p_t(1+c)S \cdot \left(q - \frac{1}{2}dq + \frac{1}{2}dq\frac{1}{n})\right) + nt_f \right)\partial / \partial n \overset{!}{=} 0$$

$$\Leftrightarrow -p_t(1+c)S \cdot \frac{1}{2}dq\frac{1}{n^2} + t_f = 0$$

$$\Leftrightarrow \frac{1}{n^2} = t_f\left(p_t(1+c)S \cdot qd\frac{1}{2}\right)^{-1}$$

$$\Leftrightarrow n = \sqrt{\frac{p_t(1+c)Sqd}{2t_f}}$$

A4. Herleitung des Gesamtumfangs in Abhängigkeit der Releaseplanung (Evolutionsfunktion)

Die Summe über die Zyklen, bei zyklusabhängiger Evolutionsfunktion, ergibt immer den Gesamtumfang der zu implementierenden Anforderungen S, unabhängig der, durch den Evolutionsparameter e gesteuerten, Releaseplanung.

$$\sum_{i=1}^{n} S/n + e(\frac{2i-1}{n} - 1)$$

$$= n(S/n) + \sum_{i=1}^{n}(2e/n)i - e/n - e$$

$$= S + (2e/n)n(n+1)/2 - e(1+n)$$

$$= S$$

Anmerkung: Die von dem Parameter e unabhängige Summierung auf den Gesamtumfang S wird durch den Summanden $-1 \cdot e$ der Summenfunktion erreicht. Dadurch gleichen sich kleine und große Inkremente gegenseitig aus.

A5. Herleitung der Bedingungen für den Evolutionsparameter der Releaseplanung

Damit der Umfang eines Inkrements nicht negativ wird, müssen Bedingungen für den Evolutionsparameter der Releaseplanung hergeleitet werden.

$$S/n + e(\frac{2i-1}{n} - 1) > 0, e > 0$$

$$\Leftrightarrow S/n + e(\frac{1}{n} - 1) > 0, e > 0$$

$$\Leftrightarrow e < S/(n-1), e > 0$$

$$\Rightarrow -S/n \leq e \leq S/n$$

A6. Herleitung der gesamten Überarbeitungsdauer bei Feedback-Effekten in Abhängigkeit der Releaseplanung

$$\sum_{i=1}^{n} p_t(1+c)(S/n + e(2i-1-n)/n)q(1 - d\frac{i-1}{n})$$

$$= p_t(1+c)q \sum_{i=1}^{n} ((S/n - e/n - e) + (2e/n)i)(1 - d\frac{i-1}{n})$$

$$= p_t(1+c)q\left[(S/n - e/n - e)\cdot\left(n - \frac{d}{n}\sum_{i=1}^{n}(i-1)\right) + (2e/n)\left(\frac{n(n+1)}{2} - \frac{d}{n}\sum_{i=1}^{n} i(i-1)\right)\right]$$

$$= p_t(1+c)q\left[\begin{array}{l}(S/n - e/n - e)\cdot\left(n - \frac{d}{n}(n(n+1)/2 - n)\right) \\ + (2e/n)\left(\frac{n(n+1)}{2} - \frac{d}{n}\cdot(n(n+1)(2n+1)/6 - n(n+1)/2)\right)\end{array}\right]$$

$$= p_t(1+c)q\cdot[(S/n - e/n - e)\cdot(n(1-d/2) + d/2) + (2e/n)\cdot((1/2 - d/3)n^2 + n/2 + d/3)]$$

$$= p_t(1+c)q\cdot[S\cdot((1-d/2) + d/(2n)) - e(1 + (1-d/2)n + d/(2n)) + e\cdot((1-2d/3)n + 1 + 2d/(3n))]$$

$$= p_t(1+c)q\left[S\left(2 - d(1-\frac{1}{n})\right)/2 - ed\cdot\left(n - \frac{1}{n}\right)/6\right]$$

A7. Herleitung der optimalen Anzahl von Zyklen im Releaseplan-Modell in Abhängigkeit von der Releaseplanung

Die Optimierung von

$$T_{q,c} + nt_f = p_t(1+c)q\left[S\left(2 - d(1-\frac{1}{n})\right)/2 - ed\left(n - \frac{1}{n}\right)/6\right] + nt_f$$

bezüglich der Anzahl der Zyklen n ergibt

$$\left(p_t(1+c)q\left[S\left(2-d(1-\frac{1}{n})\right)/2+ed\left(n-\frac{1}{n}\right)/6\right]+nt_f\right)\partial/\partial n\overset{!}{=}0$$

$$\Leftrightarrow\left(p_t(1+c)q\left[\frac{1}{2}Sd\frac{1}{n}+\frac{1}{6}ed\frac{1}{n}-\frac{1}{6}edn\right]+nt_f\right)\partial/\partial n\overset{!}{=}0$$

$$\Leftrightarrow-\frac{1}{n^2}p_t(1+c)qd\left(\frac{1}{2}S+\frac{1}{6}e\right)-p_t(1+c)qd\frac{1}{6}e+t_f\ =0$$

$$\Leftrightarrow\frac{1}{n^2}=\left(t_f-p_t(1+c)qd\frac{1}{6}e\right)\Big/\left(p_t(1+c)qd\left(\frac{1}{2}S+\frac{1}{6}e\right)\right)$$

$$\Leftrightarrow n=\sqrt{\frac{p_t(1+c)Sqd+p_t(1+c)qde/3}{2t_f-p_t(1+c)qde/3}}$$

Aus den Bedingungen für den Parameter der Releaseplanung und die Feedbackeffektivität folgt $-S/2\le e\le S/2$ und $d\le 4$, wenn $n\ge 2$. Daraus folgt $-2S\le ed\le 2S$.

Die folgende Bedingung für *e>0* muss erfüllt damit der Bruch unter der obigen Wurzel nicht negativ und dessen Nenner nicht Null wird.

$$t_f > p_t(1+c)qS/3$$

B. Rechnungen zum Meetingplan-Modell

B1. Herleitung der gesamten erwartete Überarbeitungsdauer

Die gesamte Überarbeitungsdauer ist u. a. abhängig von der Gesamtdauer *n* der Entwicklung und dem Evolutionsparameter *e*.

$$T_{q,c}=\int_{t=0}^{n}p_t(1+c)t\cdot qS(t)dt$$
$$=p_t(1+c)qS/n(\frac{1}{2}n^2+\frac{1}{3}2\frac{e}{n}n^3-\frac{1}{2}en^2)$$
$$=p_t(1+c)qSn(\frac{1}{2}+\frac{1}{6}e)$$

Im Vergleich dazu die äquivalente Herleitung in der Notation des Modells aus Loch, Terwiesch 1998. Die erwartete Überarbeitungsdauer berechnet sich aus:

$$ER(\alpha,\sigma(t),\lambda)=\int_{0}^{\lambda T_2}kt\mu_\alpha(t+T_1-\lambda T_2)dt$$

Hier bezeichnet k den Sensitivitätsfaktor und $\mu_\alpha(t)$ die erwartete Änderungsrate. Für $e = 0$ ergibt sich

$$ER(\alpha,\sigma(t),\lambda)=\frac{1}{2}(\lambda T_2)^2 k\mu_\alpha$$

Hier entsprechen $\lambda T_2\mu_\alpha$ der Anzahl der Änderungen (Engineering Changes, ECs) und $\frac{1}{2}\lambda T_2 k$ der durchschnittlichen Überarbeitungsdauer pro EC, jeweils im Überlappungszeitraum. Bei vollständiger Überlappung mit $T_1 = T_2$ und $\lambda = 1$ ergibt sich

$$ER(\alpha,\sigma(t),\lambda)=T_2^2 k\mu_\alpha\left(\frac{1}{2}-\frac{1}{6}e\right)$$

Hier entsprechen $T_2\mu_\alpha$ der Anzahl der ECs und $\left(\frac{1}{2}-\frac{1}{6}e\right)T_2 k$ der durchschnittlichen Überarbeitungsdauer pro EC.

B2. Herleitung der optimalen Anzahl von Meetings (Kommunikationsrate) innerhalb eines Releases

Die optimale Anzahl von Meetings ergibt sich aus der Ableitung der Kommunikationskosten nach dem Schwellenwert für den Änderungsumfang ($s(t)$), der in einem Meeting kommuniziert werden soll.

$$\partial(p_t(1+c)s(t)/(2n)+qS(t)/s(t)t_{setup})/\partial s(t)=0$$
$$\Leftrightarrow p_t(1+c)/(2n)-qS(t)/s(t)^2 t_{setup}=0$$
$$\Leftrightarrow s^*(t)=\sqrt{2nqS(t)t_{setup}/p_t(1+c)}$$

Mit $\beta(t)=qS(t)/s(t)$ ergibt sich

$$\beta^*(t)=\sqrt{\frac{p_t(1+c)qS(t)}{2nt_{setup}}}$$

B3. Herleitung der Dauer zwischen Prototyp-Anforderung und Änderungen in den Kundenanforderungen bei optimaler Prototyp-Einführungsrate

Die optimale Prototyp-Einführungsrate berechnet sich wie folgt:

$$\beta_P^*(t) = \sqrt{\frac{p_t(1+c)gq_P S(t)}{2nt_{setup}^P}}$$

Unter Verwendung dieser Einführungsrate und einer Prototyping-Produktivität von p_t^P kann die Dauer t_P^* zwischen Prototyp-Anforderung und Änderung in den Kundenanforderungen wie folgt hergeleitet werden

$$\begin{aligned}
t_P^* &= 1/(n\beta_P^*) + s_P^* p_t^P + t_{setup}^P \\
&= 1/(n\beta_P^*) + q_P S/(n\beta_P^*) p_t^P + t_{setup}^P \\
&= \sqrt{\frac{2nt_{setup}^P}{p_t(1+c)gq_P S/n}}\left(\frac{1}{n} + q_P S/n \cdot p_t^P\right) + t_{setup}^P \\
&= \sqrt{\frac{2t_{setup}^P}{p_t(1+c)gq_P S}}(1 + q_P S \cdot p_t^P) + t_{setup}^P
\end{aligned}$$

C. Rechnungen zum Modell aus Loch, Terwiesch 1998

Zum vertieften Verständnis des Modells aus (Loch, Terwiesch 1998) wurden die dort durchgeführten Berechnungen nachvollzogen und sind im Folgenden aufgeführt.

C1. Herleitung der Entwicklungsdauer basierend auf der Kommunikationsstrategie

$$\begin{aligned}
&EC(\alpha, \sigma(t), \lambda) \\
&= \int_0^{\lambda T_2} k(n(t)-1)/2 + \mu_\alpha(t)/n(t)\tau_2 \\
&\underline{\underline{opt.}} \int_0^{\lambda T_2} \sqrt{2k\tau_2\mu_\alpha} - k/2 \\
&= \lambda T_2(\sqrt{2k\tau_2\mu_\alpha} - k/2)
\end{aligned}$$

Der erste Summand des anfänglichen Integrals entspricht den Kommunikationsdauer durch Meetings, der zweite Summand der zusätzlichen Überarbeitungsdauer durch Kommunikationsverzögerungen. Der Schritt unter Einsetzen der optimalen Kommunikationsstrategie *n*(t)* ist im Folgenden nachvollzogen:

$$
\begin{aligned}
& k(n^*(t)-1)/2+\mu_\alpha(t)/n^*(t)\tau_2 \\
&= k(\sqrt{2\mu_\alpha(t)\tau_2/k}-1)/2+\mu_\alpha(t)/\sqrt{2\mu_\alpha(t)\tau_2/k}\tau_2 \\
&= \sqrt{\mu_\alpha(t)\tau_2 k/2}-k/2+\sqrt{\mu_\alpha(t)k\tau_2/2} \\
&= \sqrt{2\mu_\alpha(t)\tau_2 k}-k/2
\end{aligned}
$$

C2. Herleitung der gesamten Zielfunktion

$$
\begin{aligned}
& ET(\alpha,\sigma(t),\lambda) \\
&= T_1+(1-\lambda)T_2+\alpha\tau_1+\lambda T_2(\sqrt{2k\tau_2\mu_\alpha}-k/2+\tfrac{1}{2}\lambda T_2 k\mu_\alpha) \\
&= T_1+(1-\lambda)T_2+\alpha\tau_1+\lambda T_2 k(\sqrt{2\tau_2\mu_\alpha/k}-1/2+\lambda T_2\mu_\alpha/2)
\end{aligned}
$$

C3. Herleitung der optimalen Überlappung

$$
\begin{aligned}
& \partial(T_1+(1-\lambda)T_2+\alpha\tau_1+\lambda T_2 k(\sqrt{2\tau_2\mu_\alpha/k}-1/2+\lambda T_2\mu_\alpha/2))/\partial\lambda \\
&= -T_2+T_2 k(\sqrt{2\tau_2\mu_\alpha/k}-1/2)+k\lambda T_2^2\mu_\alpha/4 \\
&= T_2(-1+k(\sqrt{2\tau_2\mu_\alpha/k}-1/2)+k\lambda T_2\mu_\alpha/4 \overset{!}{=} 0 \\
&\Leftrightarrow \lambda T_2 = \frac{4(1-k(\sqrt{2\tau_2\mu_\alpha/k}-1/2))}{k\mu_\alpha} \\
&\Leftrightarrow \lambda T_2 = \frac{4(1/k-\sqrt{2\tau_2\mu_\alpha/k}-1/2)}{\mu_\alpha}
\end{aligned}
$$

Damit reduziert sich die Überlappung mit steigender Unsicherheit und (bei $0 < k < 1$) mit steigender Sensitivität.

D. Windows Vista Timeline

Abbildung 6-1 zeigt den Verlauf der Entwicklung von Microsofts Windows Vista, da der Weblink zu dieser Abbildung nicht mehr verfügbar ist.

Abbildung 6-1: Windows Vista Timeline

Diese Zeitangaben in dieser Abbildung wurden auf Basis eigener Nachforschungen angepasst.

Lebenslauf (03.2011)

Dipl.-Inform. Lars Haferkamp

Severinusstr. 19, 50354 Hürth
Mobil: +49 177 9378946
E-Mail: LHaferkamp@Googlemail.com

Angaben zur Person

Geburtsdatum, -ort	07.04.1979, Daun
Familienstand	Ledig
Staatsangehörigkeit	Deutsch

Schule, Grundwehrdienst und Studium

Seit 10/2006	Promotionsstudium der Wirtschaftsinformatik an der Universität zu Köln
10/1999 - 05/2005	Studium der Informatik an der RWTH Aachen mit Abschluss Diplom
09/2001 - 07/2002	Auslandsstudium an der NTNU Trondheim in Norwegen
09/1998 - 08/1999	Grundwehrdienst in der Sportfördergruppe Köln-Wahn
09/1989 - 07/1998	Besuch des Geschwister Scholl Gymnasium in Daun mit Abschluss Abitur

Beruflicher Werdegang

05/2010 – heute	Senior Consultant Softwareentwicklung bei der Cyber:con GmbH in Bonn: Softwareentwicklung und 3rd-Level Support in der Telekommunikationbranche.
10/2006 - 03/2010	Wissenschaftlicher Mitarbeiter am Seminar für Wirtschaftsinformatik und Systementwicklung der Universität zu Köln: Leitung von Programmierkursen- und praktika, Betreuung der Vorlesung „Information Systems Architecture“, Forschung im Bereich „Agile Softwareentwicklung“
01/2006 - 12/2007	Selbständige Tätigkeit in der Sportbranche gefördert durch das Bundesministerium für Wirtschaft und Technologie: Entwicklung einer interaktiven Website, Erstellung eines Businessplans
05/2005 - 09/2006	Wissenschaftliche Hilfskraft am Seminar für Wirtschafts- und Sozialstatistik der Universität zu Köln: Entwicklung statistischer Algorithmen und Simulationen, Systemadministration
04/2003 - 04/2005	Studentische Hilfskraft am Lehrstuhl für Informatik 6 der RWTH Aachen: Entwicklung eines automatischen Spracherkennungssystems

Veröffentlichung, Konferenzen

Mosler, K., Haferkamp, L. (2010): “A Comparison of Recent Procedures in Weibull Mixture Testing.” Veröffentlicht in: Advances in Data Analysis, Statistics for Industry and Technology, 5, 211-218.

Haferkamp, L. (2007): "Social Software in Business". Vortrag im Rahmen der VENUS Summer School 2007, The Use of Social Software and Web 2.0 in Business and Education, Universität zu Köln.

Mosler, K.; Haferkamp, L. (2007): “Size and Power of Recent Tests for Homogeneity in Exponential Mixtures”. Veröffentlicht in: Communications in Statistics - Simulation and Computation, 36, 3, 493 – 504.

Macherey, W.; Haferkamp, L.; Schlüter, R.; Ney, H. (2005): "Investigations on error minimizing training criteria for discriminative training in automatic speech recognition". In Interspeech 2005 - Eurospeech, Lissabon, Portugal, 2133-2136.