Fallstudienarbeit

Hochschule:	Hochschule für Oekonomie & Management
Standort:	Düsseldorf
Studiengang:	Bachelor Wirtschaftsinformatik
Veranstaltung:	Fallstudie / Wissenschaftliches Arbeiten
Betreuer:	Prof. Dr. Uwe Kern
Typ:	Fallstudienarbeit
Themengebiet:	Agile Softwareentwicklung
Autor(en):	Muhammet Altindal, Frank Hinkel, Robert Zyla
Studienzeitmodell:	Abendstudium
Semesterbezeichnung:	SS11
Studiensemester:	2
Bearbeitungsstatus:	begutachtet
Prüfungstermin:	
Abgabetermin:	

Inhaltsverzeichnis

1 Abkürzungsverzeichnis

Abkürzung	Bedeutung
CRACK	Collaborative, Representative, Authorized, Committed, Knowledgeable performer
IEEE	Institute of Electrical and Electronics Engineers
TDD	Test-Driven Development
XP	eXtreme Programming

2 Einleitung

Klassische Verfahren der Softwareentwicklung wie das Wasserfallmodell, welches sich bereits in den 1970er Jahren durch W. Royce etablierte, begründen noch heute ihre Existenz in IT-Projekten[1].

Mit der Evolution der Informationstechnologien in Unternehmen entstanden weitere Vorgehensmodelle. Obwohl diese oft angepasst und in unterschiedlichen Varianten gelebt werden, liegt ihnen ein traditioneller prozess- oder planorientierter Ansatz zu Grunde. Klassische Verfahren umfassen also alle Vorgehensmodelle mit diesem Ansatz.

Unter agiler Softwareentwicklung sind jüngere Verfahren zu verstehen, welche scheinbar durch andere Philosophien und Ausrichtungen einen Traditionsbruch zu etablieren versuchen. Da grundsätzliche und altbewährte Methoden in Frage gestellt werden, ist ein wissenschaftlicher Vergleich hilfreich, um Misverständnissen entgegen zu wirken. Das Ausmaß des Traditionsbruchs kann folgende Aussage Eckerts verdeutlichen: demnach werde der Begriff Vorgehensmodell im deutschen Sprachgebrauch häufig synonym zu

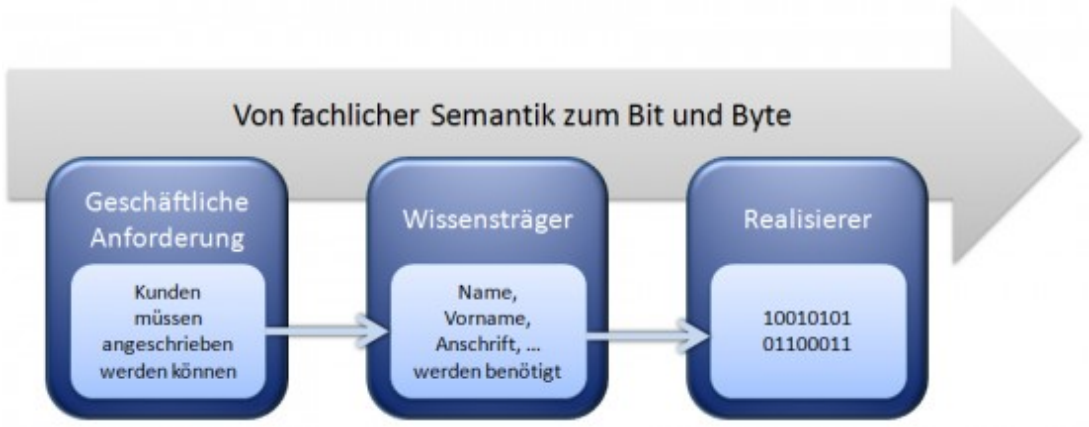
Prozessmodell verwendet und kontrastiert dadurch zu agilen - nicht prozessorientierten - Methoden[2].

Im Rahmen dieser Fallstudie werden Verfahren und deren Charakteristika von der agilen und klassischen Softwareentwicklung diskutiert und anschließend anhand unternehmensrelevanter Kriterien bewertet. Basierend auf den Ergebnissen einer Gegenüberstellung von Vor- und Nachteilen, können fundierte Entscheidungshilfen zur Ausgestaltung von IT-Projekten dargestellt werden.

3 Grundlagen

Zur fachlichen Näherung an die Thematik, soll in einem Grundlagenkapitel die grundsätzliche Problematik und Aufgabenstellung der Softwareentwicklung dargestellt werden, um anschließend den Sinn und Zweck von Vorgehensmodellen aufzuzeigen.

3.1 Aufgabe der Softwareentwicklung



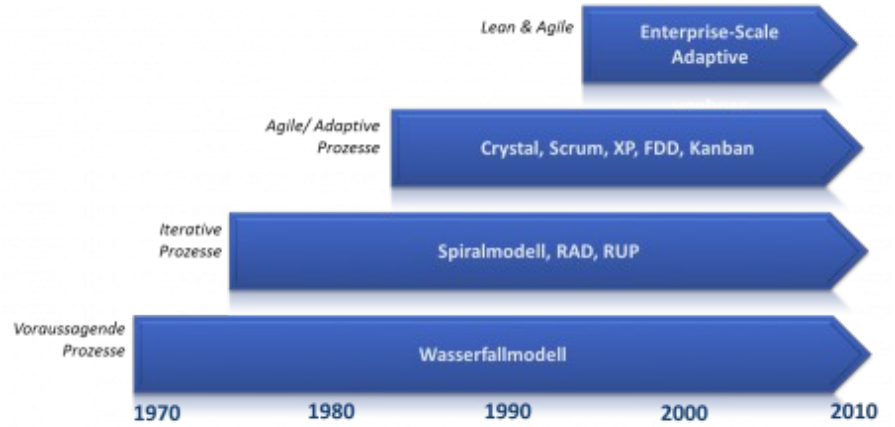
[Abb. 1] Vom Geschäftsprozess zum Bit

Eine ursprüngliche Aufgabe der Softwareentwicklung in Unternehmen ist es, fachliche Anforderungen und Problemstellungen durch Softwareprodukte zu lösen und abzubilden. Im Zentrum steht das Entwickeln einer Software, welche auf einer bereitgestellten Infrastruktur lauffähig sein muss. Dabei werden funktionale und nicht funktionale Anforderungen an das Softwareprodukt gestellt.

Umfangreiche fachliche Anforderungen, die beispielsweise durch komplexe Geschäftsprozesse der Unternehmen entstehen können, werden häufig von mehreren Personen bzw. Entwicklern umgesetzt. Neben den Entwicklern ist oftmals eine Vielzahl anderer Rollen an einem Softwareentwicklungsprozess beteiligt. Dabei spielen die fachlichen Experten (Fachbereiche) - als Quelle für das geschäftsrelevante Know-How - eine wichtige Rolle. Darüber hinaus werden ggf. weitere Kräfte, wie beispielsweise Administratoren oder Techniker in Anspruch genommen. Da in großen Unternehmen oft mehrere unterschiedliche Softwareprodukte aus unterschiedlichen Fachbereichen parallel umgesetzt werden, konkurrieren diese miteinander um Ressourcen. Dementsprechend sind auch verwaltende und steuernde Maßnahmen notwendig.

Häufig werden umfangreiche Softwareentwicklungen in IT-Projekten umgesetzt. Mit dem Ziel alle beteiligten Ressourcen wie beispielsweise Entwickler, Fachbereiche und Arbeitsmittel zu planen, koordinieren, auszurichten und anzuleiten, also in ein effizientes, zielgerichtetes und für Geldgeber und Entscheider transparentes Verhältnis zu setzen, etablierten sich unterschiedliche Verfahren: die Vorgehensmodelle.

3.2 Vorgehensmodelle und Methoden



[Abb. 2] In Anlehnung an: Leffingwell (2011), S. 4 **Verfahren in der Softwareentwicklung über die letzten Jahrzehnte**

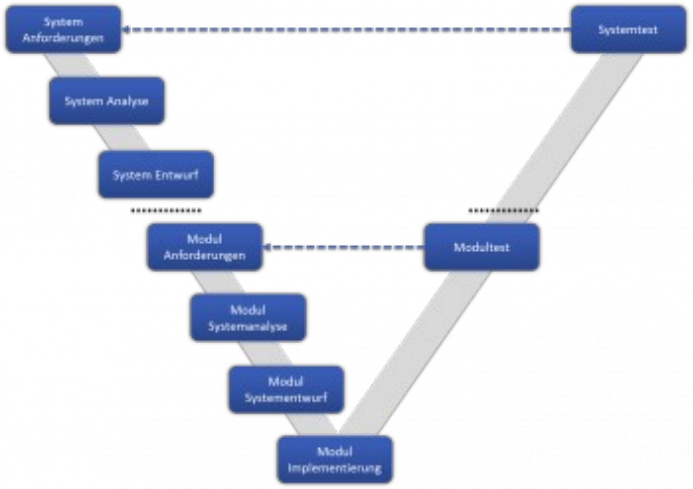
Frederick Brooks sieht in nicht eingehaltenen Terminen, gesprengten Budgets und fehlerhaften Produkten, die größten Gefahrenquellen welche sich Software-Projekte entgegen stellen müssen. Zur Begründung der potentiellen Gefahrenquellen äußert sich Brooks in seinem Artikel ?no silver bullet? detaillierter. Dabei betont er die nicht reduzierbare und unumgängliche Essenz von Komplexität, Konformität, Änderbarkeit und Unsichtbarkeit, welche in modernen Software Systemen enthalten seien[3]. Auf die Komplexität in der Softwareentwicklung wird auch im Buch ?Softwaretechnik ? Mit Fallbeispielen aus realen Entwicklungsprojekten? hingewiesen. In diesem Zusammenhang sei die Bedeutung der Effektivität von Vorgehensmodellen sowie deren praktische Umsetzung im Hinblick auf den Geschäftserfolg besonders hoch[4].

Somerville weist darauf hin, dass ein Vorgehensmodell eine vereinfachte Beschreibung eines Softwareprozesses sei[5]. Dies deutet darauf hin, dass es sich bei einem Vorgehensmodell um eine Abstraktion eines tatsächlichen Prozesses handelt. Wir unterscheiden im Rahmen dieser Fallstudie zwei wesentliche Gruppen von Vorgehensweisen: klassische bzw. traditionelle und agile.

3.2.1 Klassische Verfahren

Unter den klassischen Vorgehensmodellen verstehen wir Vorhergehensweisen, welche sich prozessorientiert ausgestalten und dabei allen Beteiligten eines Projekts konkrete Arbeitsanweisungen zur Verfügung stellen. Der gesamte Vorgehensprozess ist in einzelne Phasen aufgeteilt, die ggf. öfter durchlaufen werden können. Dabei weisen die klassischen V-Modelle statisch an feste Vorgaben angelehnte Charakteristika auf.

3.2.1.1 V-Modell



[6] In Anlehnung an: Goll, Joachim (2011), S. 89 **Bild 3-6 Das V-Modell**
Das V-Modell wird von Goll als ein generalisierter Rohling deklariert, welcher auf ein konkretes Projekt angepasst werden kann. Anders als bei sonstigen klassischen Modellen werden im V-Modell Aktivitäten und Ergebnisse definiert. Auf eine zeitlich strikte Abfolge wird dabei verzichtet. Darüber hinaus fehlen die typischen Abnahmen, welche Phasenenden definieren. Dennoch ist es möglich, die Aktivitäten des V-Modells zum Beispiel auf ein Wasserfallmodell oder ein Spiralmodell abzubilden. Die Überführung einer V-Modell Vorlage in ein konkretes praktikables Projekthandbuch für ein entsprechendes Projekt wird als Tailoring bezeichnet[7]. Die Grundelemente des V-Modells bilden folgende Komponenten:

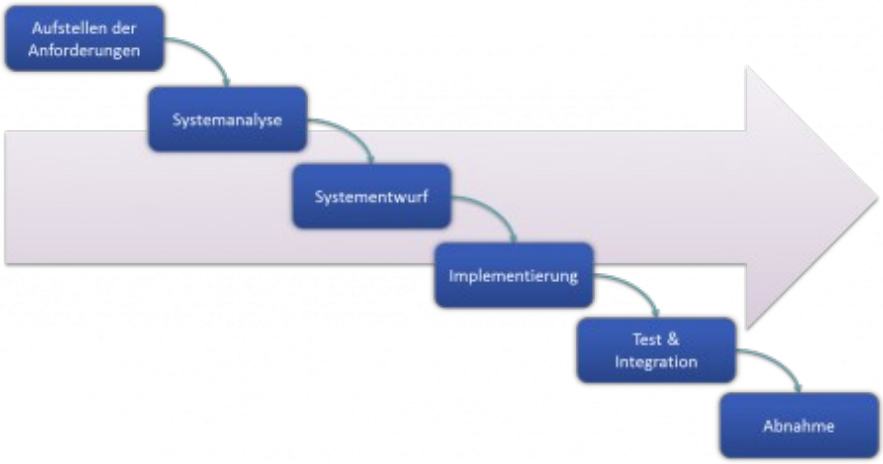
- Aktivitäten: Tätigkeiten im Software-Entwicklungsprozess
- Produkte: Artefakte der Aktivitäten

Werden komplexe Anforderungen an V-Modelle gestellt, können diese in Sub-Modelle untergliedert werden[8]:

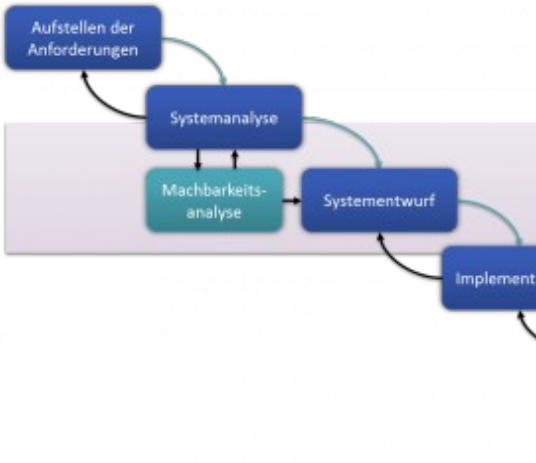
- Software-Erstellung
- Qualitätssicherung
- Konfigurationsmanagement und
- Projektmanagement

3.2.1.2 Wasserfallmodell

Das Wasserfallmodell setzt sich aus folgenden Schritten zusammen, welche in Abbildung 3 dargestellt sind. Jeder dieser Schritte wird im Wasserfallmodell als genau eine Phase abgebildet. Daher handelt es sich bei einem Wasserfallmodell um ein Phasenmodell. Jede Phase hat eine definierte Aufgabenstellung, die zu einem definierten Phasenergebnis führen soll[9].



[Abb. 3] In Anlehnung an: Goll, Joachim (2011), S. 84 **Bild 3-3 Grundform eines Wasserfallmodells ohne Machbarkeitsanalyse**



[Abb. 4] In Anlehnung an: Goll, Joachim (2011) **Wasserfallmodell mit Rückführungspfeilen**

Das Ergebnis einer Phase stellt die Eingabe und den Start der nächsten Phase dar[9]. Wenn die vorherige Phase mit einer Qualitätsprüfung freigegeben und abgeschlossen wurde, kann die nächste Phase begonnen werden. Bei entdeckten Fehlern oder Änderungswünschen während der Qualitätsprüfung, kann die nächste Phase nicht angestoßen werden. Erst die Freigabe - aufgrund einer ausreichenden Qualität - durch einen Verantwortlichen, führt zum Sprung in die nächste Phase. Werden an den Phasengrenzen der sequentiellen Entwicklungsschritte Qualitätsprüfungen durchgeführt, so kann das Wasserfallmodell als Baseline Management-Modell bezeichnet werden[10].

Wasserfallmodell mit Rückführschleifen

Die erweiterte Form des Wasserfallmodells beinhaltet Rückführschleifen, um auf veränderte Anforderungen oder Fehler reagieren zu können. Anders als bei der Baseline Management-Variante entfällt beim Wasserfallmodell mit Rückführschleifen die Zeitachse[11]. Rückführungen sind nicht zwingend in die vorherige Phase notwendig, sondern können auch in weit frühere Phasen erfolgen.

3.2.2 Agile Softwareentwicklung

Für Vorgehensweisen in der klassischen Softwareentwicklung wird grundsätzlich der Begriff Vorgehensmodell verwendet. In der agilen Softwareentwicklung ist die Verwendung des Begriffs Vorgehensmodell schwierig, da Schwerpunkte vielmehr auf Werte und Praktiken statt auf organisatorische Rahmen gelegt werden. Daher wird in diesem Zusammenhang der Begriff Methodik angewendet.

3.2.2.1 Das agile Manifest

Allen Methoden der agilen Softwareentwicklung liegt das Agile Manifest zugrunde. Die jeweiligen Vorgehensweisen und Konzepte basieren auf den dort erwähnten Werten. Im Detail ist das Agile Manifest eine öffentliche Erklärung von Absichten und Zielen durch Kent Beck, James Grenning und Robert C. Martin et al.:

Das Agile Manifest

"Wir erschließen bessere Wege, Software zu entwickeln,?

indem wir es selbst tun und anderen dabei helfen.?

Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt:

Individuen und Interaktionen *mehr als Prozesse und Werkzeuge*

Funktionierende Software *mehr als umfassende Dokumentation*?

Zusammenarbeit mit dem Kunden *mehr als Vertragsverhandlung?*

Reagieren auf Veränderung *mehr als das Befolgen eines Plans*

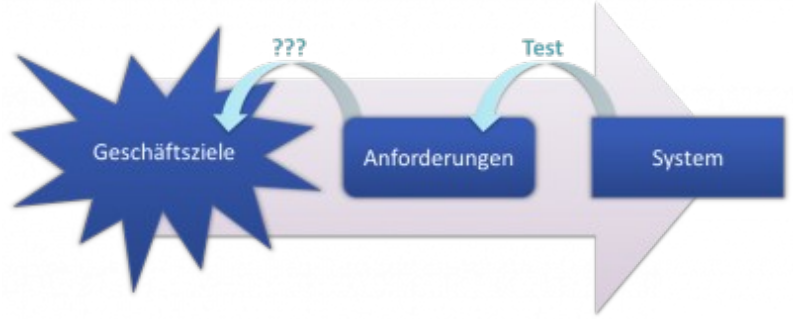
Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden,?

schätzen wir die Werte auf der linken Seite höher ein."[12]

3.2.2.2 Softwareentwicklung

Die Autoren Wolf und Roock beschreiben die agile Softwareentwicklung durch möglichst häufige Rückkopplungsprozesse und zyklisches (iteratives) Vorgehen auf allen Ebenen[13]. Mit Ebenen sind hier die Teams, das Management und die Programmierung gemeint. Das neu zu entwickelnde System, wird bei der agilen Softwareentwicklung nicht im Vorfeld detailliert geplant und dann in einem einzigen Lauf entwickelt. Die Entwicklungen wechseln zwischen kurzen Planungs- und Entwicklungsphasen miteinander ab. Wolf und Roock erklären, dass ein Plan für eine erste Version dann ausgearbeitet wird, wenn eine erste Vision, also die Ziele für das System, festgelegt und gewichtet worden sind[13]. Notwendige Anpassungen bezüglich der Entwicklungsgeschwindigkeit, den organisatorischen Rahmenbedingungen und den möglichen Problemen, werden erst nach diesem Schritt durchgeführt. Eine Basisversion mit den wichtigsten Features, soll möglichst schnell entwickelt werden, um diese gemeinsam mit dem Kunden zu bewerten, erforderliche Anpassungen vorzunehmen und nachfolgende Versionen besser optimieren zu können. Um diesen Prozess zu ermöglichen stellen Wolf und Roock folgende Anforderungen und Einstellungen in den Vordergrund der agilen Softwareentwicklung:

- Fehler dürfen gemacht werden und sind weniger als Schwäche, sondern viel mehr als Chance zu verstehen, um sich verbessern zu können.
- Jeder Projektbeteiligte ist mit seinen Stärken und Schwächen einzigartig und leistet damit seinen Beitrag für das Projekt.
- Transparenz schaffen um allen Projektbeteiligten Einblicke in die eigene Arbeit zu gewähren[14].



[Abb. 5] In Anlehnung an: Wolf, Henning; Roock, Arne (2011), S. 5 *Abb.2 Auf direktem Wege lässt sich nicht testen, ob definierte Anforderungen geeignet sind, die Geschäftsziele zu erreichen'*

Um diesen Anforderungen gerecht zu werden, soll der Kunde mit dem Management und den Entwicklern eng zusammenarbeiten und in Standup-Meetings gerade fertiggestellte und zeitnah anfallende Arbeiten kommunizieren. Die vom Kunden und Management gegebenen Geschäftsziele sollen mit jeder neuen Software in Anforderungen überführt werden, dem das neue System entsprechen soll.

Wolf und Roock sprechen hier von einem doppelten Übersetzungsprozess: also Geschäftsziele an Anforderungen und Anforderungen an das System[15]. Um diese Übersetzungsprozesse zu überprüfen werden Testverfahren eingeführt. Die Tests sollen so früh wie möglich mit einem früh eingeführten System erfolgen, um Fehlentwicklungen noch während der Projektlaufzeit entgegen zu wirken.

3.2.2.3 Beispiele für agile Methoden

Basierend auf den Grundkonzepten der Agilität, etablierten sich diverse agile Methoden. Folgende Tabelle soll eine kurze Übersicht über einige bekannte agile Methoden geben.

Agile Methode	Begründer	Kurzbeschreibung
eXtreme Programming (XP)	Kent Beck, Ward Cunningham, Ron Jeffries, Daimler Chrysler	Eine der beliebtesten agilen Methoden. Entstanden aus Erfahrungen, welche gewonnen wurden als bei Daimler Chrysler ein Informationssystem entwickelt wurde. XP ist sehr prinzipientreu. Alle in XP definierten Methoden sollten gemäß Lehrbuch angewendet werden. Praktiken wie User Stories, Pair Programming, Simple Design, Test First und Continuous Integration sind umzusetzen.
Adaptive Software Development (ASD)	Jim Highsmith	ASD entstand als Reaktion auf Turbolenzen der Industrie und mit dem Wunsch nach schneller Umsetzung von fachlichen Anforderungen. ASD bietet eine philosophische Basis und einen praxisorientierten Ansatz. Dabei spielten iterative Entwicklungsprozesse, Feature-Based Planning und eine Fokussierung auf Kundenanforderungen entscheidene Rollen.
Crystal	Alistair Cockburn	Eine Sammlung von Methoden, welche in unterschiedlichen Stufen verwendet werden können. Diese sind abhängig von der Größe des Teams und der Kritikalität des Projekts. Die Methoden reichen von agilen bis zu hin zu plangetrieben sowie psychologischen und oranisatorischen Entwicklungsforschungen.
Scrum	Ken Schwaber, Jeff Sutherland, Mike Beedle	Scrum richtet sich eher in Richtung Management aus. Dabei werden

		grundsätzlich Intervalle von 30 Tagen angelegt, in denen eine bestimmte Anzahl von Anforderungen abgearbeitet werden sollen. Diese Anforderungen wurden im vorhinein priorisiert (Backlog). Zusätzlich findet jeden Tag ein 15 minütiges Scrum Meeting statt, welches die hauptsächliche Koordination der Beteiligten regelt.
Feature-Driven Development (FDD)	Jeff DeLuca, Peter Coad	Ein sehr leichtgewichtiger, architekturbasierter Prozess, welches zu Beginn eine Gesamtübersicht zur Architektur und Features auflistet. Anschließend werden Methoden wie Design-By-Feature und Build-By-Feature angewendet. Zentrale Rollen sind hierbei der Chef Architekt und der Chef Entwickler. UML und andere Objektorientierte Design Methoden spielen eine zentrale Rolle.

[Tabelle 1]

4 Analyse

Sowohl die Komplexität von Softwareentwicklungen, als auch die existierende Artenvielfalt von Vorgehensmodellen, erschweren eine markante Abgrenzung zwischen agilen und klassischen Verfahren. Boehm und Turner haben aufgrund dieser Schwierigkeit wichtige Charakteristiken von Software-Projekten dokumentiert, welche bei einer Unterscheidung zwischen agil und klassisch helfen[16]. Ergänzend dazu stellt Schlimm in seinem Blog Eintrag essentielle Unterschiede zwischen agilen und klassischen Verfahren dar[17]. Folgende Zusammenstellung von Charakteristiken aus den beiden Quellen bilden die Grundlage für die darauf folgende Analyse und Abgrenzung der unterschiedlichen Verfahren:

- Primäre Ziele
- Umgebung/ Umfeld
- Kundenbeziehung
- Dokumentation
- Softwareentwicklung

4.1 Klassisch

4.1.1 Primäre Ziele

[Abb. 6] In Anlehnung an: Leffingwell (2011), S. 17 **Während Anforderungen fix sind, werden die Ressourcen und die Zeit geschätzt und können sich verändern.**
Boehm und Turner bezeichnen Vorhersagbarkeit, Stabilität und hohe Sicherheit als Ziele von traditionellen plangetriebenen Verfahren [18]. Beim V-Modell - als klassisches Verfahren - lassen sich diese Ziele wiederfinden. V-Modelle geben eine klar definierte Struktur für Software-Projekte vor. Voraussetzung für die Anwendung des V-Modells seien stabile und vollständige Anforderungen (Requirements). Die Anforderungen sollten sich während der Projektlaufzeit möglichst nicht verändern. Durch die Stabilität der Anforderungen bestehe die Möglichkeit einer guten Planung des Projekts [19].

4.1.2 Umgebung/ Umfeld

In klassischen Verfahren wie zum Beispiel dem Wasserfallmodell oder V-Modell, muss von einer sich wenig verändernden Umgebung ausgegangen werden, damit diese Verfahren effizient angewendet werden können. Verändert sich die Umgebung, so ist möglicherweise eine Anpassung an den bereits ermittelten Anforderungen erforderlich, welche bei klassischen Verfahren nicht vorgesehen ist.

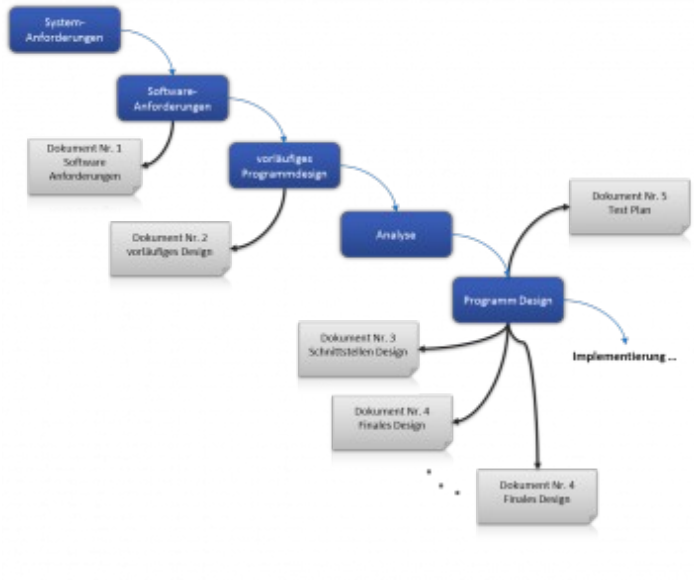
Diese Problemstellung verdeutlicht die sequentielle Phasenorientierung des Wasserfallmodells, welche in den Grundlagen dargestellt ist. Zentaler Aspekte bilden die Problematik von Fehlerbehandlungen und die Anpassung an Veränderungen.

Auch beim V-Modell zeige der Ansatz für den Umgang mit Fehlern und Änderungen ähnlich wie beim Wasserfallmodell nachteilige Wirkungen. Rückschritte in frühere Phasen seien durch Maßnahmen der Qualitätssicherung an jeder Phase nur bedingt möglich[19].

4.1.3 Kundenbeziehung

Traditionelle plangetriebene Verfahren weisen Abhängigkeiten von Verträgen sowie Spezifikationen auf. Dabei bilden Verträge die Basis zwischen Entwicklern und Kunden (Auftraggeber). Vereinbarungen zwischen Entwicklern und Kunden werden durch das Formalisieren von Lösungen zu vorhersehbaren Problemen ermittelt. Die Spezifikation und das genaue Festhalten der Lösungen alle erkennbaren Problemstellungen bildet einerseits die Arbeitsvorlage für Entweckler und anderseits wird Kunden im Voraus ein Bild davon vermittelt, wie das Produkt nach der Projektlaufzeit aussehen wird. Dies setzt voraus, dass der Vertrag sehr präzise und vollständig ist. Ist der Vertrag unvollständig oder unpräzise, so besteht die Gefahr von nicht erfüllbaren Erwartungen des Kunden. Damit ist das Vertrauen des Kunden zum Entwicklerteam gefährdet[20].

4.1.4 Dokumentation



[Abb. 7] In Anlehnung an: Royce (1970) **Wasserfallmodell mit den zu erstellenden Dokumenten**

Royce stellt in seinem Artikel "Managing the Development of Large Software Systems" das Wasserfallmodell vor und geht in diesem Zusammenhang näher auf die Thematik der Dokumentation ein. Demnach sei die oberste Regel der Verwaltung von Softwareentwicklungen die konsequente Durchsetzung des Bedarfs an Dokumentation. Die Notwendigkeit einer umfangreichen Dokumentation sei aufgrund der ungreifbaren und unbestimmbaren verbalen Kommunikation sehr hoch. Nach Royce sind die Begriffe Dokumentation, Spezifikation und Design abhängig voneinander. Den Wert der Dokumentation zeigt Royce beispielhaft anhand drei konkreter Situationen[21]:

- **Testphase**

  Die Dokumentation helfe bei der Testphase dabei, die Mitarbeiter auf Fehler im Programm zu konzentrieren.

- **Betriebsphase**

  Wenn die Software im Unternehmen eingesetzt wird, biete die Dokumentation eine Hilfestellung bei der Bedienung der Software.

- **Nachfolgende Änderungen**

  Effiziente Neugestaltung, Updates oder Nachrüsten werden durch eine gute Dokumetation ermöglicht. Ohne eine gute Dokumentation müsse die grundlegende Struktur der betriebenen Software auch bei geringen Veränderungen verworfen werden. Nur so könne sichergestellt werden, dass eine Neugestaltung oder Nachrüsten effizient umgesetzt werden kann.

4.1.5 Softwareentwicklung

- **Planung**

In der klassischen Softwareentwicklung bilden Planungen die Basis für Kommunikation und Koordination. Umfangreiche und genaue Planungen bringen die Sicherheit mit sich Chaos und hohen Kommunikationsaufwand zu vermeiden. Eine bessere Abschätzung der Entwicklungsdauer, sowie die Kontrolle umfangreicher Ergebnisse, seien weitere Vorteile von plangetriebenen Verfahren[22].

- **Entwicklung**

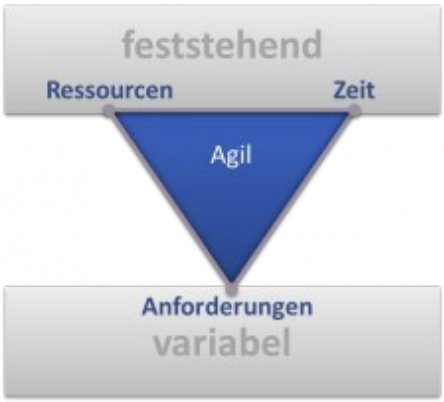
  In plangetriebenen Verfahren soll eine klare Rollenverteilung Mitarbeiter in ihrem Aufgabenbereich unterstützen[23]. Da in den Planungsphasen des Projekts die Software Anforderungen, Schnittstellen und Testpläne sehr detailliert dokumentiert sind, wird bei klassischen Ansätzen in der Softwareentwicklung ein diszipliniertes Vorgehen angestrebt. Ein wesentlicher Bestandteil ist die Befolgung von etablierten Prozessen[24].

- **Kommunikation**

  Nach Boehm und Turner werden in plangetriebenen Verfahren Informationen in Form von Prozessbeschreibungen oder Fortschrittsberichten in der Regel unidirektional kommuniziert. Zwar existiere in plangetriebenen Ansätzen zwischenmenschliche Kommunikation, jedoch spiele diese eine tendenziell untergeordnete Rolle und diene eher dazu das Verständnis der Intentionen und Bedeutungen der Dokumentationen herzustellen[25]. In der unidirektionalen Kommunikation hingegen, fließt die Information nur in eine Richtung. Dies bedeutet, dass grundsätzlich keine Rückmeldung vom Empfänger zum Sender der Information stattfindet.

4.2 Agil

4.2.1 Primäre Ziele



[Abb. 8] In Anlehnung an: Leffingwell (2011), S. 17 **Resourcen sowie die Zeit sind festgelegt, Anforderungen sind geschätzt und können sich verändern.**

Die wesentlichen Ziele der agilen Softwareentwicklung lassen sich im agilen Manifest sowie in den 12 Prinzipien des agilen Manifests wiederfinden. Das folgende agile Prinzip steht dabei an erster Stelle:

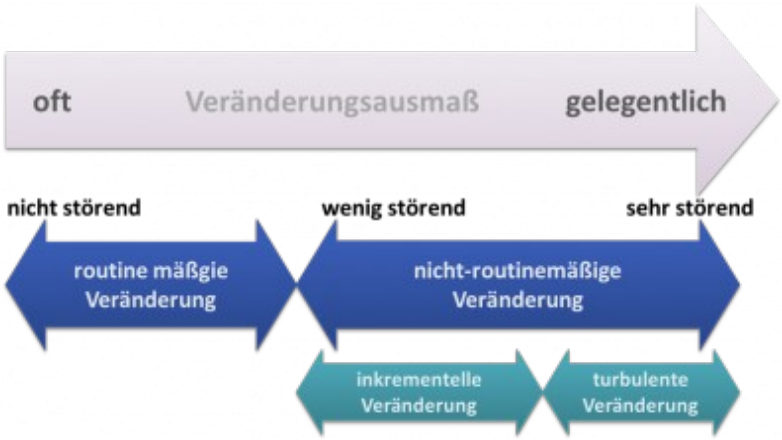
?Unsere höchste Priorität ist es, den Kunden durch frühe und kontinuierliche Auslieferung wertvoller Software zufrieden zu stellen.?[26]

Das Vertrauen des Kunden kann entscheidend dazu beitragen, dass Entwickler Aufträge von Kunden erhalten. Daher gehört zu den primären Zielen der agilen Softwareentwicklung auch die Kundenzufriedenheit, welche durch die Erstellung von wertvoller Software und deren kontinuierlichen Auslieferung erreicht werden kann. Auf die Wichtigkeit der Entwicklung von wertvoller Software wird im agilen Manifest gesondert hingewiesen:

?Funktionierende Software mehr als umfassende Dokumentation?[12]

Das Reagieren auf Veränderungen spielt in agilen Verfahren eine sehr wichtige Rolle in Bezug auf das Erstellen von wertvoller Software. Da sich beispielsweise Anforderungen des Kunden ändern können, wird das Reagieren auf Veränderung hoch eingeschätzt[27]. Als wesentliche Ziele der agilen Softwareentwicklung lassen sich Flexibilität, also das Reagieren auf Veränderung, funktionierende sowie wertvolle Software und die Kundenzufriedenheit festhalten.

4.2.2 Umgebung/ Umfeld



[Abb. 9] In Anlehnung an: Kelly (2008), S. 123 **Klassifizierung der Veränderung**
Allan Kelly geht in seinem Buch ?Changing Software Development? genauer auf die Thematik der Veränderung ein. Den Begriff der Veränderung unterteilt Kelly in zwei grundlegende Richtungen:

- Die regelmäßige, häufige, routinemäßigen und nicht störenden Veränderung
- Die gelegentlichen, nicht routinemäßigen und eher störenden Veränderung.

Kelly kommt zu dem Schluss, dass die agile Softwareentwicklung auf der Idee der inkrementellen bzw. schrittweisen Veränderungen aufgebaut sei. Dies reduziere das Risiko der Entwicklung im Projekt und eröffne Möglichkeiten für das organisatorische Lernen der Kunden und der Entwicklungsteams[28].

In der agilen Softwareentwicklung wird also von einem Umfeld ausgegangen, welches in der Lage ist sich regelmäßig inkrementell zu verändern. Ein agiles Prinzip weist daraufhin, dass Anforderungsänderungen selbst spät in der Entwicklung willkommen sind. Daher sind agile Ansätze grundsätzlich besonders bei einem turbulenten, sich ständig änderndem Umfeld anwendbar[29]. In agilen Verfahren soll das Reagieren auf Veränderung höher als das Befolgen eines Plans geschätzt werden. Jim Highsmith weist darauf hin, dass das Adaptive Software Development die Grundidee nahezubringen versucht, dass nichts beständiger sei als der Wandel. Daher solle nicht gegen den Wandel gekämpft werden, sondern der Wandel eher mit aufgenommen und zur Grundlage aller Handlungen gemacht werden[30]. Boehm und Turner weisen jedoch auf die Gefahr hin, dass durch das Reagieren auf Veränderung ohne angemessene Tests und Verifikationen erhebliche Fehler in der Entwicklung entstehen können[29]. Daher werden in agilen Verfahren häufig automatisierte Testläufe aufgesetzt.

Diese verfolgen das Ziel die Stabiltät der Anwendungen zu verifizieren, nachdem Änderungen an der Software vorgenommen wurden[31].

Beispiel aus der agilen Methode Scrum:

In Scrum werden alle Anforderungen in einem sogenannten Product Backlog gesammelt. Dabei repräsentiert das Product Backlog alle Wünsche und Bedürfnisse des Kunden an dem entsprechenden Produkt. Trotz des Umfangs weist das Backlog einen dynamischen Charakter auf. Mit dem Ziel die zu entwickelnden Produkte möglichst angemessen, wettbewerbsfähig und nützlich zu gestallten, können die Inhalte des Backlogs immer wieder modifiziert werden[32].

4.2.3 Kundenbeziehung

Die Kundenbeziehung trägt in der agilen Softwareentwicklung eine besondere Bedeutung. Nach dem agilen Manifest wird die Zusammenarbeit mit Kunden höher eingeschätzt als die Vertragsverhandlung[27]. Dies lockert die Bindung zu einmal festgelegten Anforderungen und eröffnet die Möglichkeit einer wandelbaren Bedürfnisgestalltung während der Projektlaufzeit.

Bei der Entwicklung von Software nach der Methode Scrum, nimmt der Kunde die Rolle des Product Owners ein. Nach Schwaber und Beedle hat einzig der Product Owner die Hoheit über Verwaltung und Steuerung des Product Backlogs. Damit sei der Product Owner offiziell verantwortlich für das Projekt. Damit sei der Product Owner offiziell verantwortlich für das Projekt. Im Rahmen dieser Zuständigkeit ist es Wichtig die Inhalt im Product Backlog transparent zu machen und zusätzlich bei Bedarf zu warten. Wenn das Product Backlog den Entwicklern transparent gemacht wird, dann wissen diese welche Anforderungen die höchsten Prioritäten haben und woran als Nächstes gearbeitet werden sollte. Damit der Product Owner erfolgreich in seiner Rolle sein kann sei das Respektieren der Entscheidungen notwendig[33].

In der Rolle des Product Owners ist die Einbeziehung des Kunden in die Entwicklung des Produkts deutlich zu erkennen. Dadurch bekommt der Kunde die Möglichkeit aktiv zur Entwicklung des Produkts beizutragen.

4.2.4 Dokumentation

Aus dem agilen Manifest geht hervor, dass in der agilen Softwareentwicklung eine umfassende Dokumentation gegenüber einer funktionierenden Software geringer geschätzt wird[27]. Vielmehr geht es um die Angemessenheit der Dokumentation. Ergänzend dazu werden in dem Buch ?Agility kompakt? folgende Grundsätze für agile Projekte empfohlen:

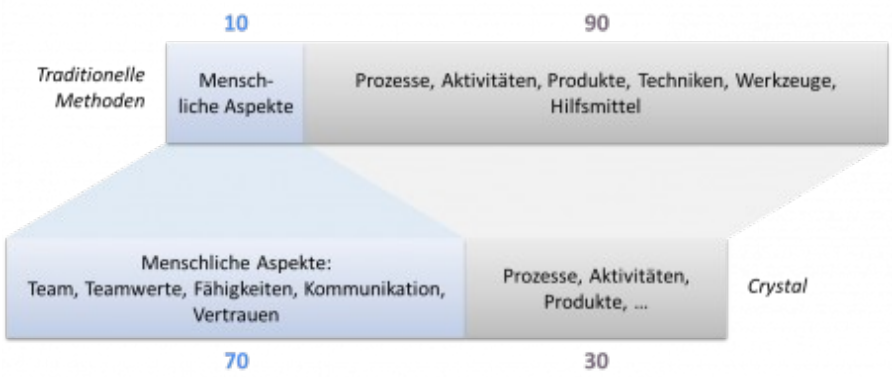
- ?So wenig Dokumentation wie möglich, aber soviel wie nötig.
- Warum statt wie.
- Dokumentieren Sie redundanzfrei: One-fact, one-place.
- Dokumentieren Sie aus Lesersicht.?[34]

Code Dokumentation

Robert C. Martin, Mitgründer des agilen Manifests, unterscheidet in seinem Buch ?Clean Code ? A Handbook of Agile Software Craftsmanship? zwischen zwei Gruppen von Kommentaren: die zu vermeidenden Kommentartypen und die zu nutzenden Kommentartypen. Martin zählt unter anderem redundante und irreführende Kommentare zu den zu vermeidenden Kommentartypen[35]. Auf der anderen Seite werden informative und klarstellende Kommentare zur Code Dokumentation empfohlen[36].

4.2.5 Softwareentwicklung

Selbstorganisierende Teams



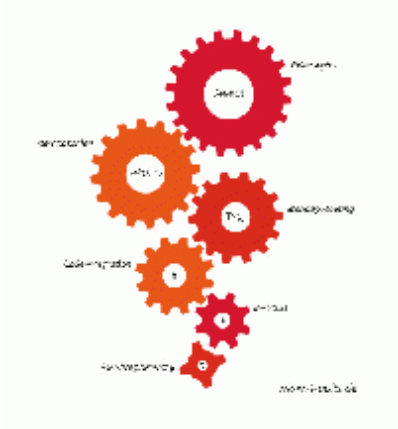
[Abb. 10] In Anlehnung an: Starke et al. (2009), S. 54 **Darstellung der Wichtigkeit menschlicher Werte anhand der Crystal Methode im Vergleich zu klassischen Verfahren**

In Scrum gibt es klar definierte Rollen und eine einfache Struktur. Zu jeder Rolle sind Aufgaben definiert. Beispielsweise entspricht die Rolle des Scrum Masters der des Methodenfachmanns, welcher den Entwicklungsprozess aufrecht erhalten soll. Der Product Owner ist verantwortlich für das Produkt und hat die Aufgabe Anforderungen zu definieren und zu priorisieren. Mit Hilfe dieser Struktur, sollen selbstorganisierende und selbststeuernde Teams ermöglicht werden[37].

Die Praktiken des XP wie pair programming, daily standup, meetings, team planning führen zum Zusammenkommen der Entwickler und des Kunden. Dies sorgt für eine gemeinsames Wissensbasis sowie ähnlichen Erfahrungsschatz[38]. Die gemeinsamen Erfahrungen sowie das gemeinsame Wissen kann unterstützend für die Kommunikation im Team sein. Es kann von einem schnelleren Informationsaustausch und weniger auftretenden Missverständnissen ausgegengen werden. So können entstehende Veränderungen aufgrund des gemeinsam genutztes Wissen schneller gelöst werden[39].

Simple design

In der agilen Softwareentwicklung wird einfaches, simples Design bevorzugt. Einer der fünf Werte des XP ist die Einfachheit. Shore und Warden beschreiben in diesem Zusammenhang, dass YAGNI (you aren?t going to need it) Prinzip. Dieses Prinzip fasse einen wichtigen Aspekt von einfachem Design zusammen. Damit ist gemeint, dass das spekulative Programmieren vermieden werden soll. Sobald etwas Neues zum Design hinzugefügt wird, müsse überprüft werden, ob es die Anforderungen des Kunden unterstützt[40].



[Abb. 11] it-agile **XP-Zyklen**

Iterativ inkrementell

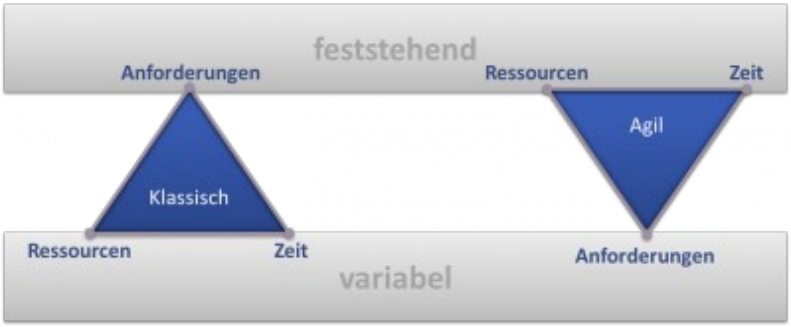
Kurze Zyklen und die iterative Entwicklung sind wesentliche Merkmale der agilen Softwareentwicklung. Für die Offenheit von Änderungen eigne sich die schrittweise Entwicklung eines Produktes. In Agile Methoden wie Scrum und XP wird inkrementelle Entwicklung angewendet. In Scrum wird das Produkt für den Kunden in mehreren Sprints, in denen jeweils eine Auswahl von Anforderungen realisiert werden, fertiggestellt[41]. Eine der 13 Praktiken von XP beschreibt den inkrementellen Entwurf. Dies bedeutet, dass die Softwareentwicklung schrittweise - nach den Anforderungen des Kunden - erfolgen soll[42]. Die kontinuierliche Integration wird hier benötigt um die bestehende Stabilität und Qualität der Anwendung transparent zu machen. Dieses Verfahren findet bei XP in der Regel stündlich statt. Jede Änderungen an Teilen des Systems, führt zu einer Überprüfung des Gesamtsystems. Dabei werden programmierte Testfälle ausgeführt um funktionale Stabilität zu verifizieren. Darüber hinaus können nicht funktionale Anforderungen über Mechanismen zur statischen Qualitätskontrolle, wie beispielsweise die maximale Anzahl an erlaubten Zeilen pro Funktion, untersucht werden.

Test-Driven Development (TDD)

Die Test-getriebene Entwicklung ist eine zentrale Praktik der agilen Methode XP. Im TDD schreiben Entwickler als erstes Testfälle, bevor sie fachliche Logik ausprogrammieren. Das Prinzip des Unit-Testings wird hier angewendet. Dabei soll eine kleine atomare Einheit für sich abgetestet werden. Bestenfalls werden alle möglichen Pfade durch eine Funktion oder Methode in allen möglichen Variationen durchlaufen. Komplexe Testfälle wie Integrations- oder Oberflächentest spielen in dem ersten Entwicklungsschritt eine untergeordnete Rolle. Diese Vorgehensweise wird auch als Test-First Ansatz verstanden. Dabei wird in drei aufeinanderfolgenden Schritten vorgegangen:

- Red - Zunächst wird ein Testfall geschrieben, welcher fehlschlägt, da die fachliche Logik noch nicht implementiert wurde.
- Green - Dann wird die eigentliche Logik ausprogrammiert, solange bis der Testfall erfolgreich also grün ist.
- Refactor - Anschließend soll das Design des Codes durch Refaktorisierung optimiert werden. Ein erneutes Ausführen des Testfalls, soll abschließend verifizieren, dass die fachliche Logik nach der Codeoptimierung immer noch korrekt arbeitet. [43]. Insbesondere bei der kontinuierlichen Integration seien automatisierte Tests sehr hilfreich, um möglichst schnell Sicherstellen zu können, ob das gesamte Produkt nachdem weitere Funktionen hinzugekommen sind weiterhin stabil funktioniert[44].

5 Gegenüberstellung



[Abb. 12] In Anlehnung an: Leffingwell (2011), S. 17 **Abgrenzung der Parameter agiler und klassischer Verfahren**

Folgende Gegenüberstellung zeigt zunächst die markanten Stärken und Schwächen von klassischen Verfahren und der agilen Softwareentwicklung. Anschließend wird die Betrachtungsweise von Beohm und Turner aufgegriffen, welche agile und klassische Charakteristika gegenüberstellt. Ziel ist die Dokumentation der entscheidungsrelevanten Faktoren bei der Auswahl einer Vorgehensweise für die Realisierung eines Softwareprodukts.

5.1 Stärken von klassischen Verfahren

- **Fundierte Erfahrungswerte und geringe Hemmschwelle**

  Die klassische Softwareentwicklung begann 1970 und hat sich bis heute weiterentwickelt. Insbesondere Projektleiter, Softwareentwickler und Auftraggeber haben sich an die Vorgehensweise der plangetriebenen Verfahren gewöhnt. Neben dieser fehlenden Hemmschwelle ein solches Verfahren anzuwenden, verfügen sie zusätzlich über einen reichhaltigen Erfahrungsschatz, kennen potentielle Gefahren, Schwachstellen und sonstige Stolpersteine. Damit bewegen sich die Beteiligten nicht auf unbekanntem Terrain und fühlen sich sicherer.

  Zusätzlich haben Unternehmen bekannte Vorgehensmodelle für ihre Bedürfnisse modfiziert. Damit sind diese Verfahren über viele Jahre auf die jeweilige Unternehmenspolitik, Personal- und Ressourcenplanung und IT-Strategie ausgerichtet worden. Eine weitere Softwareentwicklung in einem bereits etablierten Verfahren umzusetzen, welche schon andere Softwareprodukte ans Ziel geführt hat, ist bei einem Fehlschlag des Projekts durchaus leichter zu rechtfertigen als ein Fehlschlag mit völlig neuen, traditionsbrechenden Methoden.

- **Steuerung komplexer Strukturen**

  Alle klassischen Verfahren legen großen Wert auf eine umfrangreiche Analyse, Dokumentation, Planung und Sicherheit. Dabei soll ein hohes Maß an Ordnung erzeugt, Chaos und Unvorhersehbarkeiten strikt vermieden werden. Klassische Verfahren spielen also ihre Stärke in Projekten aus, welche über einen Planungszeitraum von mehreren Jahren angelegt sind, an denen viele Mitarbeiter beteiligt sind und das Resultat als geschäftskritisch angesehen wird[45].

  Da in großen Unternehmen häufig mehrere umfangreiche Softwareentwicklungen in parallel laufenden Projekten umgesetzt werden und diese Parallelentwicklungen ggf. Abhängigkeiten voneinander haben können, ist hier eine langfristig angelegte Planung hilfreich. Klassische Werte wie Ordnung und

Sicherheit unterstützen den Abstimmungsprozess unter den verschiedenen Entwicklungsprojekten.

- **Weniger Experten notwendig**

  Nach Boehm und Turner gilt folgender Erfahrungswert: "Plan-driven methods can succeed with a smaller percentage of talented people"[46]. Diese Eigenschaft eignet sich besonders für große Projekte. Bei einer hohen Anzahl an Projektbeteiligten wäre der Anspruch, überwiegend talentierte und erfahrene Experten einzusetzen tendenziell unrealistisch. Dieser Zustand resultiert vor allem aus der prozessgetriebenen Detailplanung. Im Idealfall arbeiten Entwickler ein statisches Konzept ab.

5.2 Schwächen von klassischen Verfahren

Sowohl Wolf und Roock als auch Schlimm sehen das Planen eines gesamtes Prozesses im Voraus als eine der größten Problemstellungen der klassischen Ansätze[47]. Das Potential dieser Aussage lässt sich unter verschiedenen Aspekten betrachten:

- **Fehlende Details bei der Bündelung von fachlichen Anforderungen**

  Bei der Erstellung der Dokumentation und der Planung des Projekts muss Wissen über bestehende fachliche Anforderungen gebündelt werden. Dabei kann es vorkommen, dass nicht alle benötigten Details miteinfließen. Möglicherweise ist der entsprechende Experte nicht an dem Analyseprozess beteiligt oder übersieht schlichtweg Informationen. Dies kann zu einer unvollständigen oder fehlerhaften Dokumentation führen, welche die grundlegene Basis für den weiteren Prozess bildet[48].

- **Änderungen der Anforderungen zur Projektlaufzeit**

  Unvorhersehbare Veränderungen in der Unternehmsumwelt, welche bei der ursprünglichen Planung nicht berücksichtigt werden konnten, wirken sich möglicherweise auf die Ausgestaltung der Anforderungen auf die zu entwickelnde Software aus.

5.3 Stärken der agilen Softwareentwicklung

Die Stärken der agilen Softwareentwicklung lassen sich aus Blickwinkeln von unterschiedlichen Rollen in Entwicklungsprojekten betrachten.

- Collaborative, Representative, Authorized, Committed, Knowledgeable performer (CRACK)
- Entwickler

5.3.1 Vorteile für CRACK

- **Sofortstart**

  Da kein Pflichtheft erstellt wird und nur wenig Basisfunktionalitäten zu Beginn vorgegeben werden müssen, kann ein neues Projekt zeitnah und mit wenig Verzögerung starten[49].

- **Flexibilität**

Unerwartete Anpassungen der funktionalen und nicht funktionalen Anforderungen an die zu entwickelnde Software können verschiedene Ursachen haben[49]. Diese lassen sich im Wesentlichen nach externen Rahmenbedingungen und internen Anforderungen aufteilen.

Unternehmensexterne Rahmenbedingungen	Beschreibung	Beispiel anhand eines Versicherungsunternehmens
Wettbewerbssituation	Durch neue Produkte von Mitbewerbern auf dem Markt, kann das aktuelle Projektziel an Nutzen verlieren und muss ggf. neu ausgerichtet oder im Detail angepasst werden.	Ein konkurrierendes Versicherungsunternehmen bringt neue Nicht-Raucher-Tarife auf den Markt und erzielt damit durchschlagenden Erfolg. Aufgrund dieses Erfolges, beschließen andere Versicherer in ihre aktuelle - in Entwicklung befindlichen - Versicherungsbestands-Systeme, ebenfalls Nicht-Raucher-Varianten, zu integrieren.
Gesetzliche Vorgaben	Ändern sich legislative Vorgaben während der Durchführung einer umfangreichen Softwareentwicklung oder Durchführung eines IT-Projekts und betrifft diese Änderung die fachlichen Anforderungen an die zu entwickelnde Software, muss dies berücksichtigt und umgesetzt werden.	Neue steuerliche Vorgaben an die mathematische Berechnung von Kapitalversicherungen, erzwingen die Erfassung von weiteren personenbezogenen Daten.

[Tabelle 2]

Unternehemensinterne Anforderungen	Beschreibung	Beispiel anhand eines Versicherungsunternehmens
Interne Kollaboration/Integration	In Unternehmen werden häufig verschiedene Softwareprodukte parallel neu oder weiterentwickelt. Existieren hier Anforderungen an Integration oder Kollaboration, müssen laufende Fortschritte aus Projekten in den jeweils anderen berücksichtigt und verarbeitet werden.	Das zentrale Inkassosystem, welches in einem Projekt A entwickelt wird, hat sich dazu entschieden neuerdings auch Lastschrift als Inkassoverfahren zu verwenden. In Projekt B wird ein neues Antragserfassungssystem entwickelt und muss sich einer neuen fachlichen Anforderung stellen. Demnach soll auf der Oberfläche die Auswahlmöglichkeit des Lastschriftverfahrens mit der entsprechenden Geschäftslogik, bis hin zur

		Übergabe der Einverständniserklärung an das zentrale Inkassosystem erfolgen.
Sonstige Kundenwünsche	Ob entweder Anforderungen von den Entwicklern fehlinterpretiert wurden oder der Kunde den ursprünglich konzipierten Ablauf für ineffizient oder unpraktikabel deklariert; letztendlich kann mit Anpassungen gerechnet werden, sobald der Kunde mit dem neuen System in Berührung kommt.	Der Kunde hat bei der Spezifikation der Anforderungen die Anzahl der Anrufe von Kunden unterschätzt, welche Auskünfte über ihre Risikodaten haben möchten. Der Weg zu diesen Informationen ist allerdings umständlich, da über mehrere Seiten navigiert werden muss. Hier wird eine Lösung gesucht, sodass mit einem Arbeitsschritt z.B. Click auf einen Button die Risikodaten angezeigt werden können.

[Tabelle 3]

- **Kontrolle**

  Die Auftraggeber können sich jederzeit den aktuellen Entwicklungsstand vorführen lassen und haben somit Gewissheit, ob sich die Software in die richtige Richtung entwickelt. Falls nicht, kann sofort kommuniziert werden, um beispielsweise Misverständnissen entgegen zu wirken und die Umsetzung wieder mit den Anforderungen zu synchronisieren[50].

- **Geringe Time-To-Market**

  Da die wichtigsten Features und Basisfunktionalitäten mit dem größten Geschäftswert zuerst entwickelt werden, kann nach kurzer Zeit eine Basisversion des Softwareprodukts produktiv gestellt und verwendet werden. Dies führt zu einer geringen Time-To-Market und bestenfalls zu einem selbstfinanzierenden Projekt, welches seine Kosten selbst erwirtschaftet[49].

5.3.2 Vorteile für Entwickler

- **Selbstverwirklichung**

  Entwickler in agilen Projekten schätzen Aufwände selbst, besprechen Anforderungen mit dem Kunden und können eigene Ideen und Vorschläge miteinfließen lassen. Das steigert die Identifizierung mit dem zu erstellenden Softwareprodukt[47].

- **Lernprozess in Aufwandsschätzungen**

  Aufwandsschätzungen existieren sowohl in klassischen als auch in agilen Verfahren. Das zyklische Verfahren der agilen Methoden erlaubt jedoch ursprüngliche Schätzungen zeitnah zu verifizieren. Hat sich der Entwickler grob verschätzt, kann er aus dieser Erfahrung lernen und nach der Ursache suchen. Daraus kann ggf. eine Verbesserung der Fähigkeit Aufwände zu schätzen, also ein Lernzprozess

entstehen[47].

- **Lernprozess in der Programmierung**

  Methoden wie Pair Programming bringen Entwickler dazu, über das Vorhaben zu diskutieren und sich während der Programmierung auszutauschen. Dabei können sowohl Tricks und Kniffe von erfahreneren Entwicklern weitergegeben als auch aktives Coaching und Wissentransfer betrieben werden[47].

- **Gewissheit des Nutzens**

  Durch die intensive Kommunikation mit den Anwendern können Entwickler besser verstehen was diese eigentlich für eine Software benötigen. Sie haben also die Gewissheit, dass ihre aktuelle Tätigkeit einen direkten Nutzen erzielt und wen dieser adressiert. Dabei hat der Entwicklungsprozess einen unterstützenden und helfenden Charakter, welcher den Entwicklern ein zusätzlich gutes Gefühl geben kann[47].

- **Kooperation in Abweichungen**

  Verschätzt sich ein Entwickler in Aufwänden, kann der Anwender tendenziell mehr Verständnis dafür aufbringen als bei klassischen Verfahren, da dieser im direkten und regelmäßigen Kontakt mit dem Entwickler steht und sich besser in die Lage des Anderen versetzen kann. Ebenso kann der Entwickler besser verstehen, falls der Anwender seine ursprünglichen Anforderungen abändern möchte[47].

5.4 Schwächen der agilen Softwareentwicklung

5.4.1 Überzeugungskraft

Auftrag- bzw. Kapitalgeber müssen vor einer Investition erst einmal davon überzeugt werden, dass diese für sie von Vorteil ist. Das heißt, sie benötigen möglichst genaue Informationen über Rentabilität, Risiko, Dauer und Nutzen. Da in agilen Methoden Anforderungen iterativ ausgearbeitet werden und im Vergleich zu den klassischen Verfahren ein ungenauer Vorgehensplan existiert[51], kann es schwieriger sein, Kapitalgeber - welche in der Regel risikoavers sind - von der Investition zu überzeugen.

5.4.2 Verzettelung in nicht funktionalen Anforderungen

In dem Artikel "Quantitativer und qualitativer Vegleich von Anforderungen bei agilen und konventionellen Softwareprojekten"[52] werden Studienergebnisse der Leibniz Hannover Universität dargestellt, welche sowohl klassische als auch agile studentische Softwareentwicklungsprojekte durchführen und unter folgenden Gesichtspunkten untersuchen:

- Das Verhältnis funktionaler zu nicht-funktionaler Anforderungen
- Anteil von Anforderungen an die Bedienbarkeit
- Granularität von Anforderungen

Bei der Betrachtung der Ergebnisse muss berücksichtigt werden, dass die Erkenntnisgewinne nicht auf Softwareentwicklungen in Unternehmen beruhen, sondern ausschließlich studentischen Softwareprojekten. Die dort aufgestellten Thesen zur agilen Softwareentwicklung werden weitesgehend bestätigt:

- Mehr nicht-funktionale Anforderungen mit agilen Methoden
- Usability-Fokus der Anforderungen steigt
- Details und niedrige Abstraktion in der Anforderungsbeschreibung

Da der Kunde zu jeder Zeit mit in die Entwicklung einbezogen ist und ggf. das neue System auch schon mit Basisfunktionalitäten verwendet, kann dieser viel mehr Einfluss auf nicht funktionale Anforderungen, insbesondere das Look-and-Feel der Anwendung nehmen. Hier kann das Interesse von dem Kunden, welcher die Anwendung verwendet, von dem des Auftraggeber bzw. Investors abweichen, der mit einer pragmatischen und stabilen Lösung zufrieden wäre. Es besteht die Gefahr, dass sich Anwender in Detailfragen verzetteln[51] und Entwickler den Fokus auf die wichtigen Aspekte verlieren.

5.5 Zusammenfassende Gegenüberstellung

Folgende Übersicht ist, basierend auf den Erfahrungswerten von Beohm und Turner, eine knappe Gegenüberstellung von klassischen und agilen Verfahren und bestärkt die Erkenntnisse von Wolf und Rook:

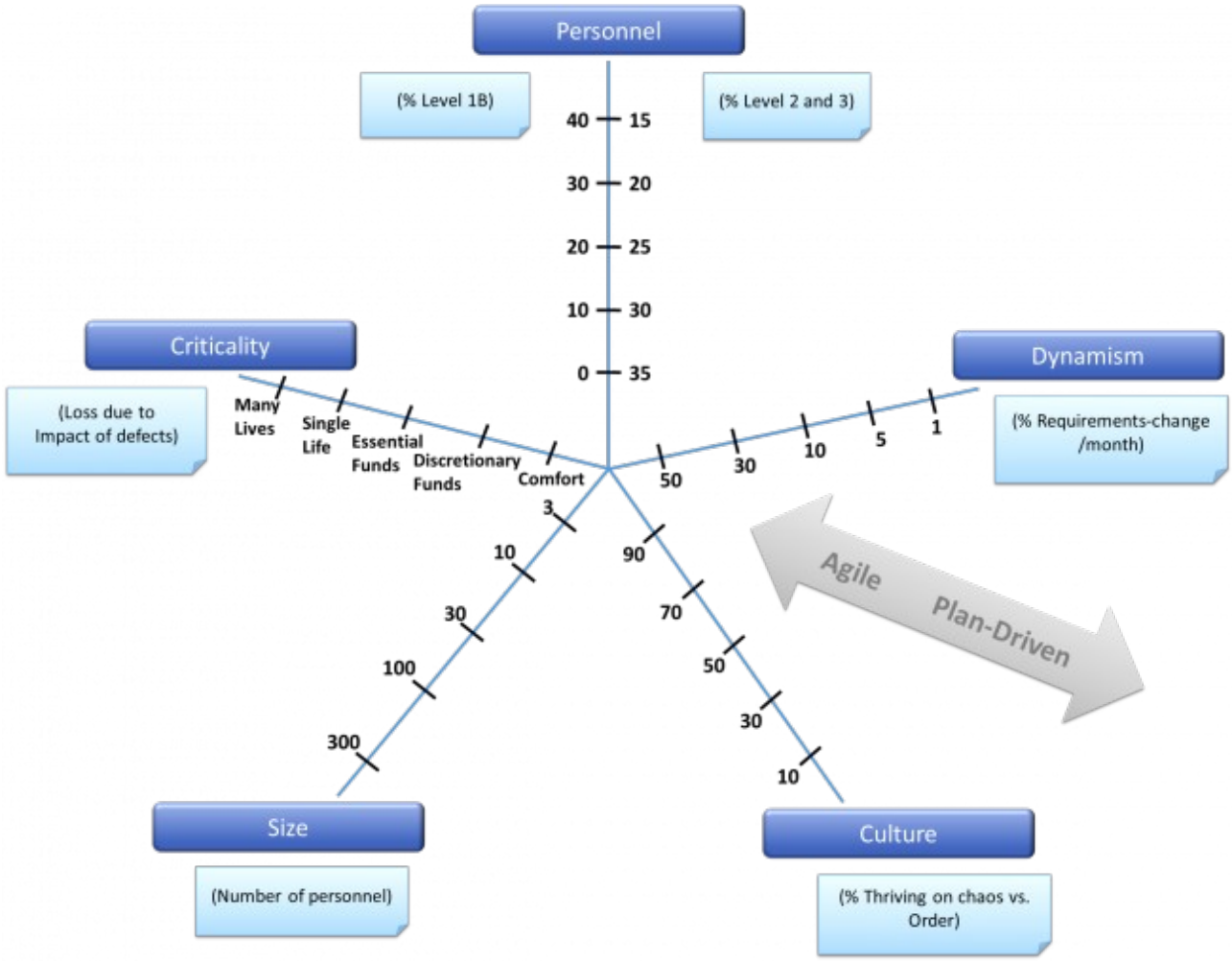
Characteristics	Agile	Plan-Driven
Application		
Primary Goals	Rapid value; responding to change	Predictability, stability, high assurance
Size	Smaller teams and projects	Larger teams and projects
Environment	Turbulent; high change; project-focused	Stable; low-change; project/organization focused
Management		
Customer Relations	Dedicated on-site customers; focused on prioritized increments	As-needed customer interactions; focused on contract provisions
Planning and Control	Internalized plans; qualitative control	Documented plans, quantitative control
Communication	Tacit interpersonal knowledge	Explicit documented knowledge
Technical		
Requirements	Prioritized informal stories and test cases; undergoing unforeseeable change	Formalized project, capability, interface, quality, foreseeable evolution requirements
Development	Simple design; short increments; refactoring assumed expensive	Extensive design; longer increments; refactoring assumed inexpensive
Testing	Executable test cases define requirements	Documented test plans and procedures
Personnel		
Customers	Dedicated, collocated [Collaborative, Representative, Authorized, Committed, Knowledgeable] performers	[Collaborative, Representative, Authorized, Committed, Knowledgeable] performers, not always collocated

Developers	At least 30% full-time Cockburn Level 2 and 3 experts; no Level 1B or -1 personnel [These numbers will particularly vary with the complexity of the application]	50% Cockburn Level 3s early; 10% throughout; 30% Level 1Bs workable; no Level -1s [These numbers will particularly vary with the complexity of the application]
Culture	Comfort and empowerment via many degrees of freedom (thriving on chaos)	Comfort and empowerment via framework of policies and procedures (thriving on order)

[Tabelle 4]

Diese Gegenüberstellung legt für Boehm und Turner den Grundstein für das Polardiagramm in Abbildung 13. Dieses ist eine graphische Darstellung von entscheidungsrelevanten Faktoren bezüglich der Auswahl von anzuwendenden Softwareentwicklungsverfahren. Dabei werden folgende Parameter berücksichtigt:

- Anzahl der beteiligten Personen
- Kritikalität
- Dynamik der Anforderungen
- Unternehmenskultur
- Expertise der beteiligten Personen



[Abb. 13] In Anlehnung an: Boehm, Turner (2004), S. 150 **Home Ground Polar Chart**

Interpretation des Polardiagramms

Bei der Positionierung der Parameter in dem Diagramm weisen Werte im zentralen Bereich eine agile Tendenz auf. Je weiter die Parameter an den äußeren Enden des Diagramms positioniert werden, desto eher tendieren diese zu plangetriebenen bzw. klassischen Verfahren. Dieser Ansatz kann je nach Ausprägung der Werte im Polardiagramm unterschiedliche Handlungsempfehlungen aussprechen.

Folgende Tabelle beschreibt im Detail die Achsen des Diagramms von Boehm und Turner:

Factor	Agility Discriminators	Plan-Driven Discriminators
Size	Well-matched to small products and teams. Reliance on tacit knowledge limits scalability.	Methods evolved to handle large products and teams. Hard to tailor down to small projects.
Criticality	Untested on safety-critical products. Potential difficulties with simple design and lack of documentation.	Methods evolved to handle highly critical products. Hard to tailor down to low-criticality products.
Dynamism	Simple design and continuous refactoring are excellent for highly dynamic environments, but a source of potentially expensive rework for highly stable environments.	Detailed plans and Big Design Up Front excellent for highly stable environment, but a source of expensive rework for highly dynamic environments.
Personnel	Requires continuous presence of a critical mass of scarce Cockburn Level 2 or 3 experts. Risky to use non-agile Level 1B people.	Needs a critical mass of scarce Cockburn Level 2 and 3 experts during project definition, but can work with fewer later in the project ?unless the environment is highly dynamic. Can usually accommodate some Level 1B people.
Culture	Thrives in a culture where people feel comfortable and empowered by having many degrees of freedom. (Thriving on chaos)	Thrives in a culture where people feel comfortable and empowered by having their roles defined by clear policies and procedures. (Thriving on order)

[Tabelle 5]

Auffällig hierbei ist die Achse zur Expertise der beteiligten Personen. Mit den Angaben Level 1B, 2 und 3 beziehen sich Boehm und Turner auf die strukturierte Einteilung von Fähigkeiten nach Cockburn, welche in folgender Tabelle kurz dargestellt ist:

Level	Characteristics
3	Able to revise a method (break its rules) to fit an unprecedented new situation

2	Able to tailor a method to fit a precedented new situation
1A	With training, able to perform discretionary method steps (e.g., sizing stories to fit increments, composing patterns, compound refactoring, complex COTS integration). With experience, can become Level 2.
1B	With training, able to perform procedural method steps (e.g., coding a simple method, simple refactoring, following coding standards and CM procedures, running tests). With experience, can master some Level 1A skills.
?1	May have technical skills, but unable or unwilling to collaborate or follow shared methods.

[Tabelle 6]

6 Schlussbetrachtung

Boehm und Turner kommen zu dem Schluss, dass es bei der Entscheidung zwischen agil oder klassisch keine Universallösung gebe. Vielmehr spielen der Kontext und die Umgebungsparameter wie Projektgröße und Kritikalität eine Rolle, um die individuell benötigten Methoden auszuwählen. Obwohl der Übergang von klassisch zu agil fließend ist, dominieren die klassischen Verfahren in den äußereren und die agilen in den inneren Bereichen des Polardiagramms[53]. Boehm und Turner untersuchen Softwareentwicklungen unterschiedlicher Größenordnung und schlagen dabei vor, wie ein Auswahlprozess einer geeigneten Methode aussehen kann. Im Rahmen dieser Auswahl werden systematisch Projektrisiken identifiziert, analysiert und sich schrittweise der gesuchten Methoden genähert[54].

Boehm und Turner machen den Bedarf an agilen Methoden deutlich und positionieren sich mit den folgenden Aussagen:

- "Future Applications Will Need Both Agility and Discipline"[55]
- "Large projects can no longer count on low rates of change, and their extensive process and product plans will become expensive sources of rework and delay."[55]

Demnach ist die Fähigkeit, die projektrelevanten Methoden auszuwählen, eine wichtige Kompetenz. Die Vorschläge zur Auswahl von geeigneten Methoden sind nach Boehm und Turner keine Universalvorlage und es wird damit gerechnet, dass sich weitere Methoden etablieren werden, um den Balanceakt zu unterstützen.

Die abschließende Erkenntnis von Boehm und Turner geht jedoch in eine andere Richtung:

- "Focus Less on Methods?More on People, Values, Communication, and Expectations Management."[56]
- "The agilists have it right in valuing individuals and interactions over process and tools."[56]

Damit appellieren Boehm und Turner daran, nicht den Fokus auf jene Werte zu verlieren, welche die ursprüngliche Basis aller agilen Methoden sind, sondern vielmehr, diese Aspekte in den Mittelpunkt zu stellen. Diese Aussage wird von Wolf und Roock deutlich untermauert indem sie betonen:

"Insgesamt sind eine gute Zusammenarbeit zwischen Entwicklern und Kunden sowie eine gemeinsame Vertrauensbasis von unschätzbarem Wert für jedes Projekt."[57]

7 Abbildungsverzeichnis

8 Tabellenverzeichnis

9 Literatur- und Quellenverzeichnis

9.1 Monographien

Verweis	Literatur / Quelle
Östreicher et al. (2010)	Schatten, Alexander; Demolsky, Markus; Winkler, Dietmar; Biffl, Stefan; Gostischa-Franta, Erik; Östreicher, Thomas: Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen, Springer Akademischer Verlag, Heidelberg 2010
Eckert (2010)	Hanser, Eckert: Agile Prozesse: Von XP über Scrum bis MAP, Springer Verlag, Heidelberg 2010
Balzert (2009)	Balzert, Helmut: Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering, 3. Auflage, Spektrum Akademischer Verlag, Heidelberg 2009
Goll (2011)	Goll, Joachim: Methoden und Architekturen der Softwaretechnik, 1. Auflage, Vieweg + Teuber Verlag, Wiesbaden 2011
	Sommerville, Ian: Software Engineering, 6.Auflage, Pearson Studium, München 2001

Sommerville (2001)	
Grechenig et al. (2010)	Grechenig, Thomas; Bernhart, Mario; Breiteneder, Roland; Kappel, Karin: Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten, Pearson Studium, München 2010
Boehm, Turner (2004)	Boehm, Barry; Turner, Richard: Balancing Agility and Discipline: A Guide for the Perplexed, Pearson Education, 2004
Wolf, Roock (2011)	Wolf, Henning; Roock, Arne: Agile Softwareentwicklung: Ein Überblick, 3. Auflage, dpunkt.verlag, Heidelberg 2011
Schwaber, Beedle (2002)	Schwaber, Ken; Beedle, Mike: Agile Software Development with Scrum, Prentice Hall, 2002
Starke et al. (2009)	Hruschka, Peter; Rupp, Chris; Starke, Gernot: Agility kompakt: Tipps für erfolgreiche Systementwicklung, 2. Auflage, Spektrum Akademischer Verlag, Heidelberg 2009
Kelly (2008)	Kelly, Allan: Changing Software Development: Learning to become agile, John Wiley & Sons, Ltd., England 2008
Martin (2009)	Martin, Robert C.: Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, 2009
Cockburn (2003)	Cockburn, Alistair: Agile Software-Entwicklung, 1. Auflage, verlag moderne industrie buch AG & CO., 2003
Leffingwell (2011)	Leffingwell, Dean: Agile Software Requirements: Lean Requirements Practices for teams, Programs, and the enterprise, Pearson Education, 2011

9.2 Zeitschriften

Verweis	Literatur / Quelle
Hennemann, Knauss (2011)	Hennemann, Melanie; Knauss, Eric: Quantitativer und qualitativer Vergleich von Anforderungen bei agilen und konventionellen Softwareprojekten, in: GI Softwaretechnik-Trends 2011, Heft Nr. 31

9.3 Internetquellen

Verweis	Literatur / Quelle
Brooks (1987)	Brooks, Frederick Phillips: No Silver Bullet ? Essence and Accidents of Software Engineering, April 1987, http://people.eecs.ku.edu/~saiedian/Teaching/Sp08/816/Papers/Background-Papers/no-silver-bullet.pdf (14:36, 4. Jun. 2011 (CEST))
Agiles Manifest	Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas: Manifest für Agile Softwareentwicklung, 2001, http://agilemanifesto.org/iso/de/ (14:30, 4. Jun. 2011 (CEST))
Agile Prinzipien	Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas: Prinzipien hinter dem agilen Manifest, http://agilemanifesto.org/iso/de/principles.html (14:29, 12. Jun. 2011 (CEST))

Royce (1970)	Royce, Winston W.: Managing the Development of Large Software Systems, August 1970, http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf (15:15, 12. Jun. 2011 (CEST))
Schlimm (2011)	Schlimm, Niklas: Essential differences in agile and traditional software development processes, Januar 2011, http://niklasschlimm.blogspot.com/2011/01/essential-differences-in-agile-and.html (18:01, 4. Jun. 2011 (CEST))

10 Fußnoten

1. ? Vgl. Östreicher (2010), S. 48
2. ? Vgl. Eckert (2010), S. 3
3. ? Vgl. Brooks (1987)
4. ? Vgl. Grechenig et al. (2010), S. 372
5. ? Vgl. Sommerville (2001), S. 24
6. ? Das V-Modell
7. ? Vgl. Goll (2011), S. 89
8. ? Vgl. Goll (2011), S. 91
9. ? 9,0 9,1 Vgl. Goll (2011), S. 84
10. ? Vgl. Goll (2011), S. 85
11. ? Vgl. Goll (2011), S. 86
12. ? 12,0 12,1 Agiles Manifest (2001)
13. ? 13,0 13,1 Vgl. Wolf, Roock (2011), S. 3
14. ? Vgl. Wolf, Roock (2011), S. 4
15. ? Vgl. Wolf, Roock (2011), S. 4 f.
16. ? Vgl. Boehm, Turner (2004), S. 25
17. ? Vgl. Schlimm (2011)
18. ? Vgl. Boehm, Turner (2004), S. 27
19. ? 19,0 19,1 Vgl. Östreicher et al. (2010), S. 51
20. ? Vgl. Boehm, Turner (2004), S. 32
21. ? Vgl. Royce (1970)
22. ? Vgl. Boehm, Turner (2004), S.34
23. ? Vgl. Boehm, Turner (2004), S. 49
24. ? Vgl. Starke et al. (2009), S. 97
25. ? Vgl. Boehm, Turner (2004), S. 36
26. ? Agile Prinzipien
27. ? 27,0 27,1 27,2 Vgl. Agiles Manifest
28. ? Vgl. Kelly (2008), S. 140
29. ? 29,0 29,1 Vgl. Boehm, Turner (2004), S. 29
30. ? Vgl. Starke et al. (2009), S. 59
31. ? Vgl. Hunt, Subramaniam (2006), S. 79 ff.
32. ? Vgl. Schwaber, Beedle (2002), S. 33
33. ? Vgl. Schwaber, Beedle (2002), S. 34
34. ? Starke et al. (2009), S. 33
35. ? Vgl. Martin (2009), S. 55 ff.
36. ? Vgl. Martin (2009), S. 59 ff.
37. ? Vgl. Wolf, Roock (2011), S. 6
38. ? Vgl. Boehm, Turner (2004), S. 33 f.
39. ? Vgl. Cockburn (2003), S. 28 f.
40. ? Vgl. Shore, Warden (2007), S. 315

41. ? Vgl. Schwaber, Beedle (2002), S. 29 f.
42. ? Vgl. Wolf, Roock (2011), S. 13
43. ? Vgl. Starke et al. (2009), S. 86
44. ? Vgl. Starke et al. (2009), S. 88
45. ? Vgl. Boehm, Turner (2004), S. 55
46. ? Boehm, Turner (2004), S. 53
47. ? 47,0 47,1 47,2 47,3 47,4 47,5 Vgl. Wolf, Roock (2011), S. 30
48. ? Vgl. Schlimm (2011)
49. ? 49,0 49,1 49,2 Vgl. Wolf, Roock (2011), S. 28
50. ? Vgl. Wolf, Roock (2011), S. 29
51. ? 51,0 51,1 Vgl. Hennemann, Knauss (2011), S. 4 f.
52. ? Hennemann, Knauss (2011), S. 4 f.
53. ? Vgl. Boehm, Turner (2004), S. 148 ff.
54. ? Vgl. Boehm, Turner (2004), S. 99 f.
55. ? 55,0 55,1 Boehm, Turner (2004), S. 151
56. ? 56,0 56,1 Boehm, Turner (2004), S. 152
57. ? Wolf, Roock (2011), S. 30