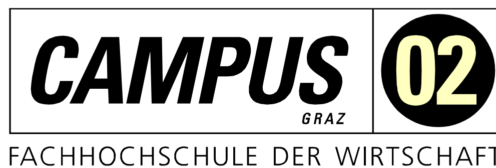
MASTERARBEIT

ENTWICKLUNG EINES VORGEHENSMODELLES ZUR ÜBERGABE UND WEITEREN BETREUUNG EINES BESTEHENDEN, AGIL UMGESETZTEN PROJEKTES IN EINEM NEUEN TEAM

ausgeführt am



Studiengang
Informationstechnologien und Wirtschaftsinformatik

Von: Daniel Steiner
Personenkennzeichen: 1510320021

Graz, am 31. März 2017

....
Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

....

Unterschrift

DANKSAGUNG

An dieser Stelle möchte ich mich bei allen in meinem Umfeld für die Unterstützung während der Erstellung dieser Arbeit bedanken. Ohne diese Unterstützung, die Motivationshilfen usw. hätte ich diese Arbeit nicht in dieser Form fertig stellen können. Besonders bedanken möchte ich mich bei meinem Betreuer Dr. Michael Amann-Langeder für die stets freundliche und fachliche Unterstützung und die sehr kurzen Antwortzeiten bzw. Telefonate. Zusätzlich statte ich den Lektoren des Campus 02 für die Vorbereitungen auf diese Arbeit Dank ab.

Ein weiterer besonderer Dank gilt meiner gesamten Familie, insbesondere meiner Frau Maria, die mich Tag und Nacht unterstützt hat und während dieser Zeit zudem sehr viel Aufwand getrieben hat, um mir den Rücken zu stärken. Zudem gilt mein Dank meiner Mutter und meinem Onkel Wolfgang, die mich zusätzlich bei der Korrektur dieser Arbeit tatkräftig unterstützt haben. Auch bedanken möchte ich mich bei meinen Studienkollegen, im Speziellen bei Patrick Lang und Gernot Waldhauser, sowie bei Freunden, die mich immer unterstützt haben und mir in Zeiten von Motivationsengpässen zur Seite gestanden sind.

Last but not least möchte ich mich noch bei meinem Arbeitgeber, der Firma Netconomy, insbesondere bei meinem direkten Vorgesetzten Martin Hechtl und meinem RSD1 Team, für die tolle Unterstützung in Form von flexiblen Arbeitszeiten, zur Verfügung gestellter Ressourcen bzw. Know-how und der mir bzw. der vorliegenden Arbeit gewidmeten Zeit bedanken.

KURZFASSUNG

Das Ziel dieser Arbeit ist die Schaffung eines neuen Vorgehensmodelles, um agil entwickelte Projekte einfach und effizient übergeben zu können. Daher beschäftigt sich die Arbeit mit der Beantwortung der Frage „Welche Aspekte sind als zentrale Einflussfaktoren zu berücksichtigen, wenn die Weiterentwicklung eines agil umgesetzten Projektes in einem anderen Team stattfinden soll, welches noch über keine bestehende Entwicklungsumgebung und nur begrenztes Know-how verfügt?“.

Um diese Frage beantworten zu können, wird zuerst ein grober Überblick über mögliche Modelle zur Entwicklung von Software gegeben. Der Fokus hierbei liegt auf den agilen Methoden. Ausgehend von dem agilen Manifest und den ausgewählten Praktiken wird ein allgemeiner Prozess anhand des Software-Lebenszyklus beschrieben. Aufbauend auf die Beschreibung der einzelnen Phasen des Software-Lebenszyklus wird der Ansatz der kontinuierlichen Integration von Software vorgestellt und beschrieben.

Mit Hilfe der theoretischen Grundlagen werden anschließend die Einflussfaktoren entlang des Software-Lebenszyklus identifiziert, kategorisiert und beschrieben. Die einzelnen Einflussfaktoren münden in eine Liste mit Risikobewertung und möglichen Werkzeugen und bilden damit das neue Vorgehensmodell.

Das neue Modell wird, nach einer kurzen allgemeinen Beschreibung des Unternehmens inklusive genereller Arbeitsweise, einem Praxistest unterzogen. Dies geschieht, indem ein bereits bestehendes, agil entwickeltes Projekt mit Hilfe des neuen Modells in ein neu geschaffenes Team übernommen wird. Im Zuge der praktischen Anwendung wird grob gezeigt wie das Modell eingesetzt werden kann und die Ergebnisse werden dokumentiert. Abschließend wird eine Übersicht der Ergebnisse in Form einer Tabelle gegeben, welche die einzeln getroffenen Entscheidungen zeigt, bewertet wie erfolgreich die einzelnen Einflussfaktoren im Zuge der Übernahme abgedeckt werden können und angibt, welche Folgeschritte geplant sind.

ABSTRACT

The aim of this thesis is to create a simple and efficient new approach to transferring agile-developed projects to other teams. This thesis addresses the question "What aspects should be considered as central influencing factors if the further development of an agile-developed project occurs in another team, with no existing development environment and only limited know-how?".

The thesis begins with a short overview of possible models for software development, particularly agile methods. Based on the agile manifesto and selected practices, a general process and understanding of the software lifecycle is described. Continuous integration is subsequently outlined.

With these theoretical foundations, the influencing factors along the software lifecycle are then identified, categorised and described. The individual influencing factors result in a new model, including a risk assessment and possible tools.

To test the new model, a practice run within the company is described. With the help of the new model, an existing project is transferred into a newly created team. The use of the model is broadly documented. Finally, an overview of the results is given as a table which shows the individual decisions taken, assesses how successfully the individual influencing factors can be covered during the takeover of the first project, and which follow-up steps are planned.

INHALTSVERZEICHNIS

1 EINLEITUNG 1

1.1 Ziele und Forschungsfrage 1
1.2 Motivation und Ausgangssituation 2
1.3 Vorgehensweise 2
1.4 Abgrenzung der Arbeit 3

2 DER WEG ZUR AGILEN SOFTWAREENTWICKLUNG 4

2.1 Vorgehensmodelle der Softwareentwicklung 4
2.2 Modellbegriff in der IT 5
2.3 Vorgehensmodelle in der IT 7
2.4 Klassische phasenorientierte Modelle 9
2.4.1 Wasserfallmodell 10
2.4.2 V-Modell 12
2.4.3 Spiralmodell 14

2.5 Agile Vorgehensweise 16
2.5.1 Das Agile Manifest 17
2.5.2 Extreme Programming (XP) 18
2.5.3 Scrum 20
2.5.4 Kanban 23

3 KONTINUIERLICHE AUSLIEFERUNG VON SOFTWARE 27

3.1 Software-Lebenszyklus 27
3.1.1 Identification 28
3.1.2 Inception 28
3.1.3 Initiation 29
3.1.4 Development and Release 30
3.1.5 Operation 32

3.2 Das Problem der regelmäßigen Auslieferung von Software 32
3.3 Kontinuierliche Integration von Software 37
3.3.1 Voraussetzungen für kontinuierliche Integration 38
3.3.2 Einsatz von kontinuierlicher Integration 41

3.4 Deployments und Releases von Applikationen 44
3.4.1 Release-Strategien und Release-Plan 45
3.4.2 Deployment einer Applikation 47
3.5 Kontinuierliches Deployment von Software 49
4 ENTWICKLUNG DES VORGEHENSMODELLS 53
4.1 Grundlagen für das neue Vorgehensmodell 53
4.2 Identifikation der Einflussgrößen 54
4.3 Entwicklung des Vorgehensmodells 61
5 ANWENDUNG DES NEUEN VORGEHENSMODELLS 64
5.1 Das Unternehmen 64
5.2 Allgemeiner Projektablauf 65
5.3 Das Projekt 67
5.4 Das Team 68
5.5 Die Projektübernahme 69
5.5.1 Vorgehensmodell, Verantwortung und Kommunikation 69
5.5.2 Jira und Confluence 71
5.5.3 Die Entwicklung 74
5.5.4 Das Ergebnis 78
6 CONCLUSIO UND OFFENE FRAGEN 82
ABKÜRZUNGSVERZEICHNIS 85
ABBILDUNGSVERZEICHNIS 86
TABELLENVERZEICHNIS 87
LITERATURVERZEICHNIS 88

1 EINLEITUNG

In dieser Einleitung werden folgende Punkte erläutert:

- Ziele und Forschungsfrage
- Motivation und Ausgangssituation
- Vorgehensweise
- Abgrenzung der Arbeit

1.1 Ziele und Forschungsfrage

Ziel dieser Arbeit ist die Entwicklung eines Vorgehensmodelles zur Weitergabe von abgeschlossenen Softwareentwicklungsprojekten an andere Teams. Dabei liegt der Fokus auf agil umgesetzten Projekten. Das Vorgehensmodell gibt Empfehlungen, wie Voraussetzungen für eine spätere Weitergabe geschaffen werden können und wie eine Übergabe erfolgen kann.

Das Vorgehensmodell wird anhand eines bestehenden E-Commerce Projektes der Firma Netconomy eingesetzt. In diesem Anwendungsfall wird ein Projekt aus der Produktentwicklung an die interne Abteilung „Customer Service und Support“ übergeben, die einerseits die Wartung, aber auch die Weiterentwicklung übernehmen soll. Für die Weiterentwicklung wird ein eigenes Team, fernab der üblichen Entwicklungsstrukturen im Unternehmen geschaffen werden.

In dieser Arbeit wird die Forschungsfrage „Welche Aspekte sind als zentrale Einflussfaktoren zu berücksichtigen, wenn die Weiterentwicklung eines agil umgesetzten Projektes in einem anderen Team stattfinden soll, welches noch über keine bestehende Entwicklungsumgebung und nur begrenztes Know-how verfügt?“ beantwortet.

Als Hypothesen sollen folgende dienen:

H1: Anhand des entwickelten Vorgehensmodelles ist die Übernahme eines bestehenden, agil umgesetzten Projektes in ein bestehendes oder neues Team ohne vorhandene Entwicklungsumgebung und mit begrenztem Know-how zu bewerkstelligen.

H0: Anhand des entwickelten Vorgehensmodelles ist die Übernahme eines bestehenden, agil umgesetzten Projektes in ein bestehendes oder neues Team ohne vorhandene Entwicklungsumgebung und mit begrenztem Know-how nicht zu bewerkstelligen.

Anhand der definierten Hypothesen soll abschließend kurz evaluiert werden, ob sich das entwickelte Vorgehensmodell für einen praktischen Einsatz eignet.

1.2 Motivation und Ausgangssituation

Die Motivation für dieses Thema kommt aus der derzeitigen Tätigkeit im Customer Service und den dort gesammelten Erfahrungen bzw. stetig wachsenden Anforderungen. Durch verbesserte Strukturen und zunehmend umfassenderes Know-how kann sich der Customer Service der Firma Netconomy deutlich von der Mitkonkurrenz abheben. Immer mehr Projekte werden von einem ständig wachsenden Team betreut und auch die Aufgaben und Kompetenzen verändern sich laufend.

Sehr viele Projekte werden auf Grund der Erfolge des Customer Service in der Vergangenheit nicht nur zu Zwecken der Wartung an den Customer Service übergeben. Die Entwicklung geht immer mehr in Richtung eigenständiger Weiterentwicklung bestehender Software direkt im Customer Service. Dies betrifft vor allem Projekte, die nach erfolgreicher Fertigstellung keine allzu große Weiterentwicklung anstreben. Daher wurde in der Vergangenheit ein eigenes kleines, agil arbeitendes Team innerhalb des Customer Service aufgestellt. Bis dato wurden im Customer Service aber nur sehr kleine und nicht komplexe Änderungen an der bestehenden Software vorgenommen, größere Anforderungen jedoch an bestehende oder jetzt nicht mehr bestehende Entwicklungsteams weitergeleitet. Geplant ist nun, mit diesem neuen Team mehrere Projekte gleichzeitig zu betreuen und diese weiterzuentwickeln, um eine zunehmend längerfristige Kundenbindung zu erreichen.

Die bisherig definierten Kriterien von Projekten, welche „nur“ zur Wartung in den Customer Service übergeben wurden, reichen hierfür nicht aus. Daher ist im Unternehmen die Anforderung entstanden, einen entsprechenden Prozess bzw. ein Modell zu entwickeln, anhand dessen die Übergabe von agil entwickelten Prozessen in den Customer Service vollzogen werden kann.

1.3 Vorgehensweise

Bevor die notwendigen Einflussfaktoren für das Vorgehensmodell evaluiert werden, soll zuerst ein grober Überblick über derzeit bestehende Phasenmodelle in der Softwareentwicklung sowie allgemeine Begrifflichkeiten gegeben werden.

Anschließend sollen der einschlägigen Literatur Theorien zur Fragestellung, welche Einflussfaktoren bei der Übergabe (Übernahme) eines agil umgesetzten Projektes zu beachten sind, entnommen und aufbereitet werden. Die identifizierten Einflussfaktoren sollen beschrieben werden und in ein Vorgehensmodell bzw. eine Best Practices-Checkliste münden. Außerdem sollen sie einer Werkzeugkategorie zugeordnet und eine mögliche Auswahl an Werkzeugen gezeigt werden.

In einem letzten Schritt soll der Aufbau einer Entwicklungsumgebung im Customer Service und Support der Netconomy anhand des entwickelten Vorgehensmodells bzw. der Übergabe eines bestehenden E-Commerce Projektes stattfinden und die Umsetzung dokumentiert werden.

1.4 Abgrenzung der Arbeit

Die Hauptziele dieser Arbeit sind die Identifikation der Einflussfaktoren zur Übergabe und Weiterentwicklung eines zuvor agil umgesetzten Projektes und die Konzeption eines entsprechenden Vorgehensmodelles. Das entwickelte Modell soll bei der Übergabe eines Projektes in den Customer Service Anwendung finden.

In dieser Arbeit nicht behandelt werden:

- Detaillierte Behandlung von technischen Aspekten wie Entwicklungsumgebung, Programmiersprachen usw.
- Es wird keine Messung des Erfolgs durchgeführt.
- Es werden nur beispielhaft Modelle beschrieben, aber keine Gesamtübersicht der Vorgehensmodelle gegeben.
- Die beschriebenen Vorgehensmodelle werden nicht hinsichtlich möglicher Einsetzbarkeit validiert oder miteinander in Beziehung gestellt.
- Das Vorgehensmodell wird eine Liste mit Werkzeugen enthalten. Auf eine detaillierte Beschreibung aller möglichen Werkzeuge wird auf Grund deren Umfangs verzichtet.
- Die Einsatzmöglichkeit des neuen Vorgehensmodells wird nicht hinsichtlich klassischer Modelle evaluiert.
- Die Dokumentation des Einsatzes beinhaltet keine detaillierte Beschreibung, wie das Modell angewandt wurde. Lediglich die Ergebnisse werden präsentiert.

2 DER WEG ZUR AGILEN SOFTWAREENTWICKLUNG

In den letzten knapp 30 Jahren hat sich agile Softwareentwicklung stetig weiterentwickelt und immer neue Methoden und Prozessgrundlagen geschaffen. Nicht nur große Unternehmen profitieren von diesen Weiterentwicklungen. Auch immer mehr kleinere Unternehmen können sich die Vorteile agiler Softwareentwicklung zu Nutzen machen. Durch agile Entwicklung von Software kann sehr oft die Produktivität gesteigert und Mitarbeitern/innen können neue Wege der Zusammenarbeit und Variabilität aufgezeigt werden. In daraus resultierenden Projekten können agile Softwareentwicklungsmethoden dazu beitragen, die benötigte Flexibilität zu erhalten und zudem eine Arbeitserleichterung herbeizuführen.

Der Einsatz agiler Softwaremodelle bringt aber nicht nur Vorteile mit sich, sondern stellt beteiligte Stakeholder[1] wie Kunden, Manager, Projektleiter und auch Softwareentwickler selbst vor die verschiedensten Herausforderungen. Es müssen daher immer auch ein entsprechender Aufwand bzw. Kosten für die Einführung von agilen Prozessen berücksichtigt werden. (Schütz, 2014)

Bevor im Detail die agilen Methoden beschrieben werden können, werden in den folgenden Kapiteln zuerst die dafür notwendigen Begrifflichkeiten erläutert.

2.1 Vorgehensmodelle der Softwareentwicklung

Bei der Umsetzung von Vorhaben der Informationstechnik (IT) sollten Unternehmen immer versuchen, standardisierte Abläufe in ihr Unternehmen zu integrieren, um diese dann zu verfolgen. Diese Abläufe müssen unabhängig vom Vorhaben sicherstellen, dass die Aufgaben ohne Überspringen von Teilschritten durchgeführt und abgeschlossen werden. (Wieczorrek & Mertens, 2011) Hierfür wurden im Laufe der Jahre in der IT etliche bereits dokumentierte und standardisierte Verläufe entwickelt. Einige dieser sogenannten Vorgehensmodelle sind sehr wissenschaftlich-theoretisch, andere bereits praxisnah erprobt. (Sandhaus, Berg, & Knott, 2014) Zusammengefasst durchläuft Software als Produkt sehr viele Phasen, die in der Praxis zusammenfassend als Softwarelebenszyklus bezeichnet werden. Dieser Lebenszyklus wird in Kapitel 3.1 näher beschrieben.

Zuvor soll jedoch geklärt werden, was unter einem Vorgehensmodell überhaupt im Detail verstanden wird. Zudem muss zuerst analysiert werden, was unter dem Begriff Modell zu verstehen ist und wie dieser Begriffsinhalt in der IT Anwendung findet.

[1] Als Stakeholder wird eine Person oder Gruppe bezeichnet, die ein berechtigtes Interesse am Verlauf oder Ergebnis eines Prozesses oder Projektes hat.

2.2 Modellbegriff in der IT

Seinen Ursprung hat der Begriff „Modell“ in der Kunst, wo darunter die Form, Beschaffenheit und Maßverhältnisse von Gegenständen verstanden werden. Im „Duden Fremdwörterbuch“ sind einige unterschiedliche Bedeutungen für den Begriff Modell zu finden. Unter anderem wird unter Modell ein Entwurf, Muster bzw. Vorlage zur serienweisen Herstellung oder auch die vereinfachte Darstellung der Funktion eines Gegenstandes oder des Ablaufes eines Sachverhalts verstanden. (Duden.de, 2016) Es gibt noch unzählige andere Definitionen für Bereiche wie Betriebswirtschaft, Physik oder auch Mathematik. Daher ist es schwierig, einen klaren Modellbegriff zu definieren.

Für IT-Systeme kann der Begriff Modell anhand von drei verschiedenen Merkmalen gekennzeichnet werden: (Stachowiak, 1973)

- Abbildung: Modelle sind immer ein Abbild von etwas und repräsentieren natürliche oder künstliche Originale, welche selbst auch wieder Modelle sein können.
- Verkürzung: In einem Modell müssen nicht zwangsweise alle Attribute des Originals vertreten sein. Es werden nur jene verwendet, welche für den Anwendungsfall als relevant erscheinen.
- Pragmatismus: Jedes Modell wird hinsichtlich eines Anwendungsfalles geschaffen. Ein Modell ist daher seinem Original nicht von Haus aus zugeordnet. Modelle erfüllen eine Ersetzungsfunktion hinsichtlich:
    - Subjekt (für wen?)
    - Zeitintervall (wann?)
    - Operation (wozu?)

Vereinfacht können die Anforderungen an ein Vorgehensmodell wie folgt dargestellt werden:

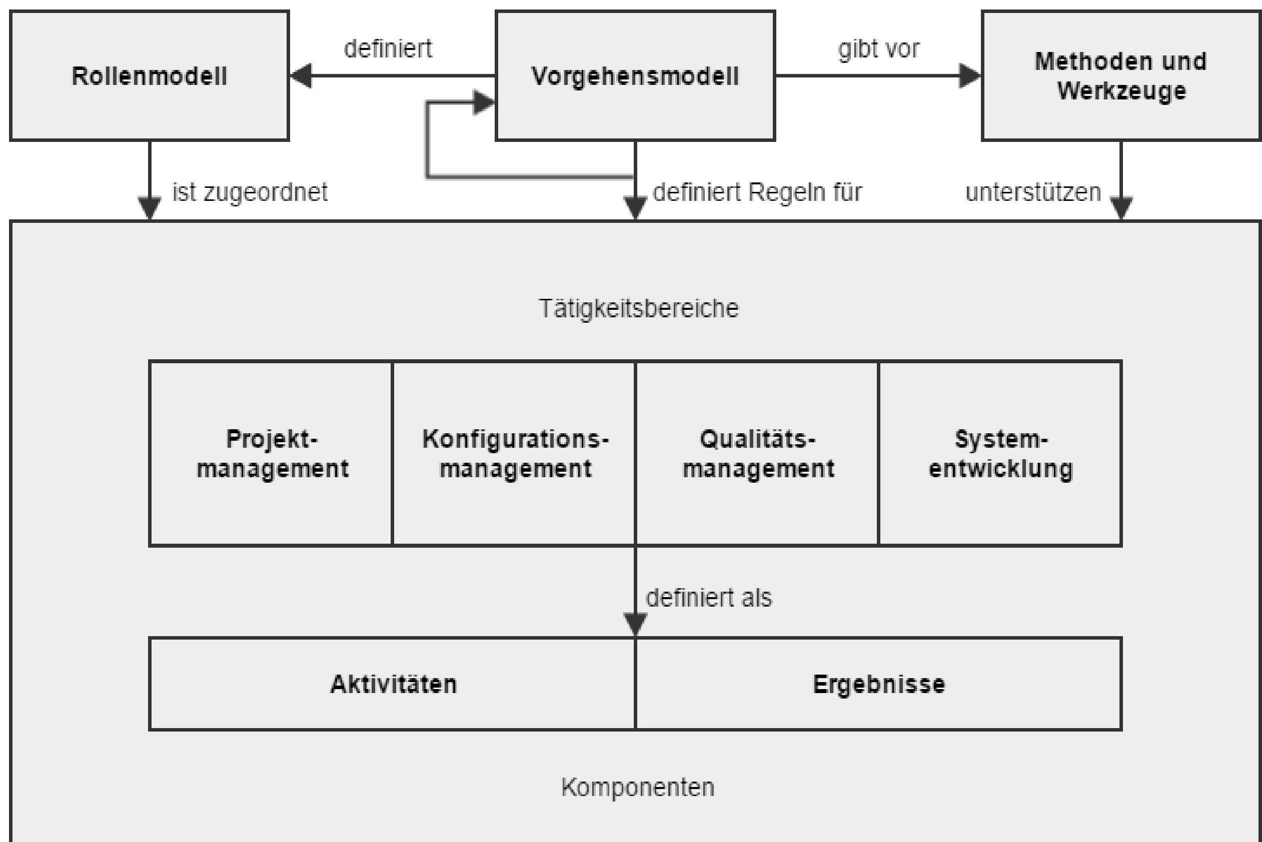


Abbildung 1 – Anforderungen an ein Vorgehensmodell (Balzert, 1998)

Wie in Abbildung 1 dargestellt, müssen Modelle gewisse Anforderungen abdecken. Ziel ist es natürlich, dass mit einem gewissen Input ein entsprechender Output generiert wird. Ein Vorgehensmodell gliedert sich also beispielsweise in Tätigkeitsbereiche, Rollen, Methoden und Werkzeuge usw. Die einzelnen Tätigkeitsbereiche können sich – ihrem Einsatz in unterschiedlichen Branchen entsprechend – enorm unterscheiden. In dieser Abbildung nur bedingt zu sehen ist, dass beispielsweise im Zuge der Entwicklung von Software die durchgeführten Aufgaben ständig mittels Rückkopplung Aufschluss darüber geben sollen, ob die gesteckten Ziele erreicht wurden.

Abgeleitet aus den Abläufen und Definitionen in Abbildung 1 lässt sich sagen, dass der mögliche Anwender eines Modelles auf jeden Fall in der Lage sein muss, das Modell als Ganzes zu verstehen, um es anschließend effizient anwenden zu können. Bezieht man dies nun auf den Prozess der Softwareentwicklung, so sollen Modelle helfen, die Komplexität der Realität zu reduzieren. Somit soll es möglich sein, einen konzentrierten Blick auf wichtige Prozesse und Methoden zu werfen. (Sandhaus, Berg, & Knott, 2014)

Nach Erläuterung des allgemeinen Begriffs eines Modelles in Bezug auf die IT wird im nächsten Kapitel geklärt, was unter einem Vorgehensmodell zu verstehen ist.

2.3 Vorgehensmodelle in der IT

Wie sehr viele natürliche bzw. künstliche Systeme hat auch Software einen gewissen Lebenszyklus. (Balzert, 2011) Begonnen wird dieser Zyklus mit einer Anforderungsanalyse, mithilfe derer festgestellt wird, welches Ergebnis mit der entwickelten Software erreicht werden soll. Aufbauend auf diese Anforderungen bzw. die daraus resultierenden Spezifikationen wird ein Entwurf vorgenommen. Das Ergebnis in Form einer Softwarearchitektur ist anschließend der Ausgangspunkt für die eigentliche Implementierung der Software mittels Programmcode, welcher nachfolgend durch Maßnahmen der Qualitätskontrolle geprüft und in letzter Instanz deployed[2] wird. Die anschließende Betriebsphase, in welcher die Software operativ genutzt wird, kann viele Jahre dauern. Während dieser Phase kann es zur mehr oder weniger ausgeprägten Weiterentwicklung an der bestehenden Software kommen. Das Ende findet der Lebenszyklus einer Software, wenn schlussendlich die Wartung eingestellt wird. Anschließend kann mit einem möglichen Folgeprojekt von vorne begonnen werden. (Rau, 2016)

Vom beschriebenen prototypischen Lebenszyklus einer Software wird in der Praxis vielfach mehr oder weniger stark abgewichen. Für die Strukturierung und Durchführung konkreter Softwareprojekte bietet der Lebenszyklus nur bedingt eine operationale Hilfestellung. (Rau, 2016) Abhilfe sollen hier nun Vorgehensmodelle (Prozessmodelle) schaffen.

Eines der Ziele eines Vorgehensmodelles ist die hierarchische Gliederung der Gesamtaufgabe. Darstellungsformen dieser Gliederung können entweder zerlegungsorientiert (Baum oder Hierarchie) oder ablauforientiert (Graph) sein. (Sandhaus, Berg, & Knott, 2014) Des Weiteren geben Vorgehensmodelle für spezifische Projekte einen Rahmen vor, welcher aber im Normalfall nicht 1:1 angewandt werden kann, sondern individuell angepasst werden muss. Im Kern beinhaltet dieser Rahmen das Vorgehensmodell, welches einen gezielten Projektablauf beschreibt. Zusätzlich gibt ein Vorgehensmodell auch noch organisatorische Regelungen wie zum Beispiel Rollen und Verantwortlichkeiten oder auch Festlegungen für ordentliche Dokumentation vor. (Rau, 2016)

Vorhaben der Anwendungsentwicklung durchlaufen prinzipiell immer Zyklen mit gewissen Phasen, welche häufig durch ein lineares Phasenmodell dargestellt werden. Das Grundprinzip dieser Phasenmodelle ist immer gleich und beinhaltet im Groben folgende Phasen: (Heinrich, 2001)

- Analyse und Definition
- Entwurf
- Umsetzung
- Qualitätssicherung und Tests
- Einsatz und Wartung

[2] Deployen beschreibt den Prozess der Installation von Software auf Anwender-PCs oder Servern.

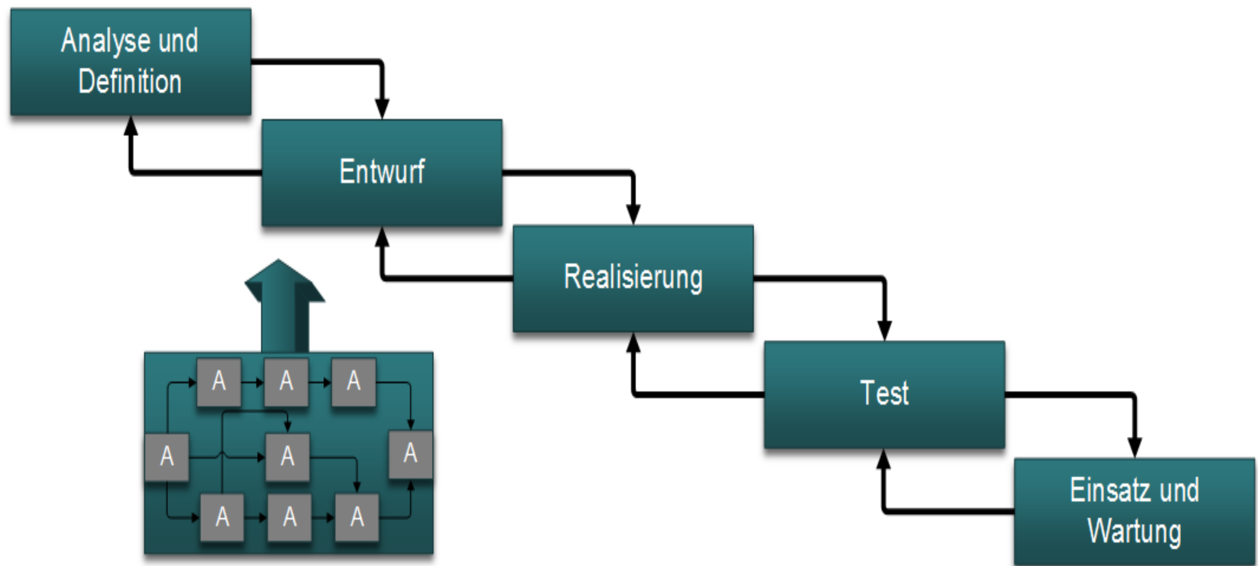


Abbildung 2 – Vereinfachte Darstellung eines Vorgehensmodells (Heinrich, 2001)

Wie in Abbildung 2 gezeigt, werden in klassischen Modellen wie beispielsweise dem Wasserfallmodell, das im Detail unter 2.4.1 beschrieben wird, die einzelnen Phasen sequentiell[3] abgearbeitet. Im Vergleich dazu werden in agilen Vorgehensmodellen die einzelnen Aufgaben iterativ[4] bzw. inkrementell[5] abgearbeitet.

Bei diesen phasenorientierten Vorgehensmodellen ist es essentiell, dass jede Phase zumindest einmal sequentiell durchlaufen wird. Änderungen der Anforderungen werden meistens erst in einem Nachfolgeprojekt berücksichtigt, welches auch wieder alle Phasen durchlaufen muss. Daher ist eines der größten Defizite dieser Modelle, dass es quasi unmöglich ist, auf geänderte Anforderungen während der Entwicklung rasch zu reagieren. (Grechenig, Bernhart, Breiteneder, & Kappel, 2010)

		Ablaufgestaltung	
		Sequentiell	**Iterativ**
Formalisierung	**Stark formalisiert**	Wasserfallmodell V-Modell V-Modell XT	Spiralmodell Prototyping OO Lifecycle Modell ...
	Wenig formal		Extreme Programming Object Engineering Process SCRUM Kanban

Abbildung 3 – Überblick Vorgehensmodelle (Krcmar, 2011)

[3] Von der Speicherung und Verarbeitung von Anweisungen eines Computerprogramms) fortlaufend, nacheinander erfolgend.

[4] Sich schrittweise in wiederholten Durchläufen der exakten Lösung nähernd.

[5] Schrittweise erfolgend, aufeinander aufbauend.

Wie in Abbildung 3 zu sehen ist, gibt es sehr viele unterschiedliche Vorgehensmodelle zur Entwicklung von Software. Im nächsten Kapitel werden nun einige der klassischen (phasenorientierten) Vorgehensmodelle grob beschrieben.

2.4 Klassische phasenorientierte Modelle

Das erste phasenorientierte Vorgehensmodell wurde im Jahr 1956 von Herbert D. Benington geschaffen und zeitgleich mit der Veröffentlichung der ersten Hochsprachen[6] veröffentlicht. In diesem Modell schlug Benington eine phasenorientierte Vorgehensweise für die Entwicklung von Software vor. Jenes Vorgehensmodell enthielt bereits viele Elemente moderner Modelle und war später die Grundlage für das unter 2.4.1 beschriebene Wasserfallmodell. (Hofmann, 2008) Die Komplexität der einzelnen phasenorientierten Modelle soll durch die verschiedenen Entscheidungsstufen, wie zum Beispiel Meilensteinen während des kontinuierlichen Entscheidungsprozesses, reduziert werden. (Hansen & Neumann, 2009) Meilensteine sollen hierbei nicht nur die einzelnen Phasen voneinander abgrenzen. Zusätzlich legen sie fest, welche Ergebnisse erreicht werden müssen, welche Kriterien dafür ausschlaggebend sind und wer schlussendlich die Entscheidung fällt, ob die Phase abgeschlossen werden kann. Klassisch phasenorientierte Modelle kennzeichnet zusätzlich eine zeitliche Unterteilung der einzelnen Schritte zur Systementwicklung. (Sandhaus, Berg, & Knott, 2014)

Zusammengefasst lässt sich über phasenorientierte Modelle sagen, dass sie einen schrittweisen Weg, mit klar definierten und bindenden Phasen, von der Problemstellung bis zur Lösung vorschlagen. Dadurch soll die Entwicklung der gewünschten Software planbar, kontrollierbar und vorhersagbar sein. (Schütz, 2014)

[6] Unter Hochsprache ist in diesem Zusammenhang eine höhere Computersprache zu verstehen. Das Mindestmerkmal höherer Programmiersprachen ist, dass diese nicht unmittelbar von Mikroprozessoren verstanden und ausgeführt werden können. Befehle müssen durch Interpreter oder Compiler in Maschinensprache übersetzt werden. (wikipedia, 2016)

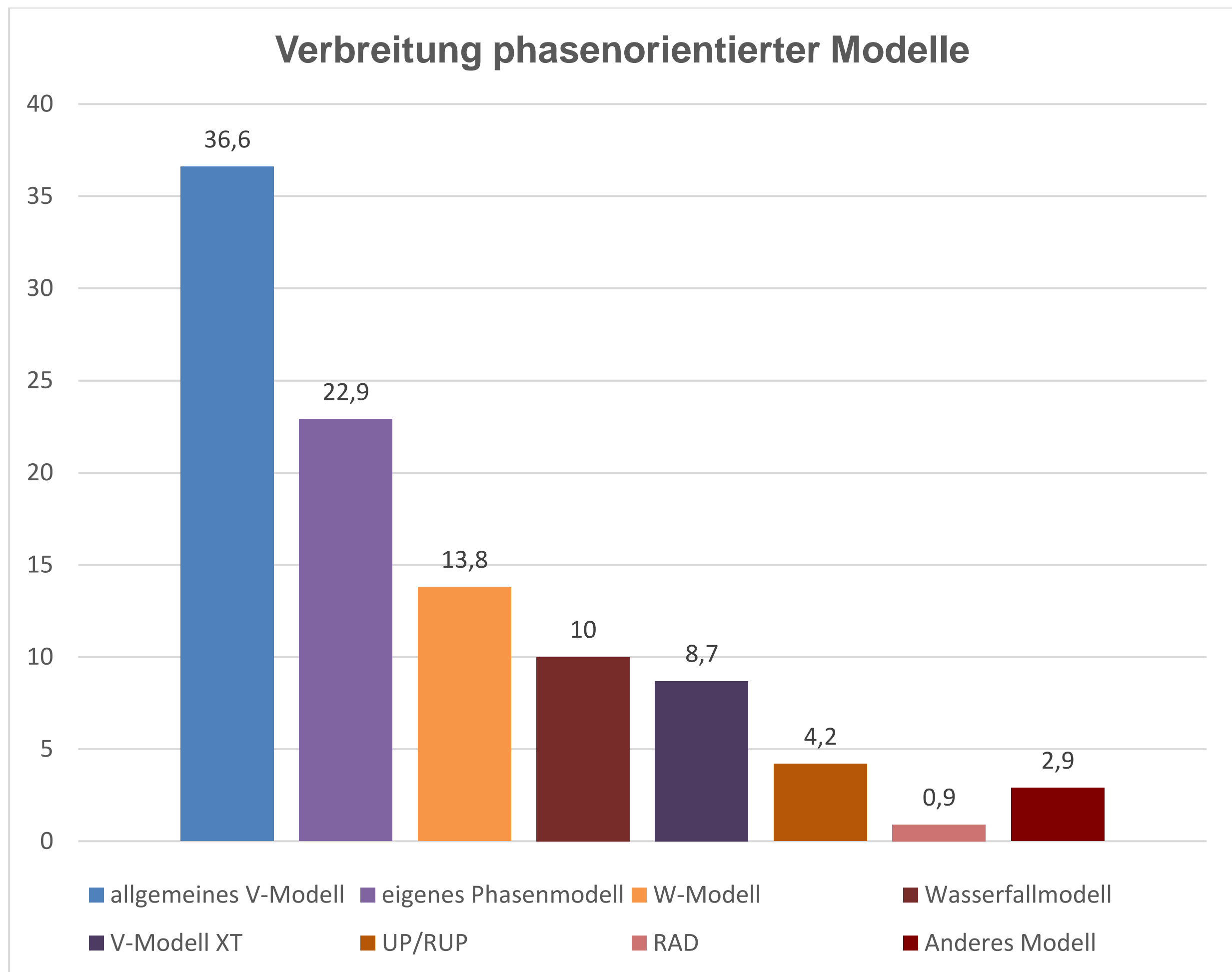


Abbildung 4 – Verbreitung phasenorientierter Modelle (Sandhaus, Berg, & Knott, 2014)

Sehr viele Unternehmen nutzen heutzutage immer noch diesen Ansatz. Abbildung 4 zeigt, welche konkreten Modelle in Unternehmen eingesetzt werden, wenn diese mit klassischen Modellen arbeiten. Im deutschsprachigen Raum werden vor allem das Wasserfall- und V-Modell eingesetzt, beide sollen in den nächsten Kapiteln beschrieben werden. (Sandhaus, Berg, & Knott, 2014)

2.4.1 Wasserfallmodell

Ausgehend von der „Code & Fix“[7] Herangehensweise der 1960er Jahre wurde in den 1970er Jahren das Wasserfallmodell geschaffen. Das ursprüngliche Modell wurde im Jahr 1970, als Weiterentwicklung des Softwarelebenszyklus-Modells von Winston Walker Royce[8], in dem Artikel „Managing the development of large Software Systems“ beschrieben, aber noch nicht als jetzt

[7] Bei „Code & Fix“ wird ohne vorherige Analyse, Spezifikation, usw. direkt vom Entwickler der Programmcode geschrieben, getestet und korrigiert. Dies passiert so lange bis das System den Vorstellungen des Entwicklers entspricht. (Krahe, 2009)

[8] War ein amerikanischer Computerwissenschaftler, Direktor bei Lockheed Software Technology Center und ein Pionier auf dem Gebiet der Software-Entwicklung

bekanntes Wasserfallmodel tituliert. 11 Jahre später wurde dann der Begriff „Wasserfallmodell" durch Barry W. Boehm[9] geprägt. (Rau, 2016) (Sandhaus, Berg, & Knott, 2014)

Das ursprüngliche Modell bestand aus den fünf Phasen Anforderungen, Analyse, Design, Implementierung und Test. Bei der Entwicklung einer Software muss jede dieser Phasen durchlaufen und ein gewisser Output (in Form von Dokumenten) generiert werden. Zudem können Meilensteine zur Überprüfung der Ergebnisse von einzelnen Phasen dienen. Am Ende jeder Phase wird entschieden, ob mit dem Projekt fortgefahren werden kann. Das ursprüngliche Modell sieht nicht vor, dass nach Abschluss einer Phase wieder in diese zurückgekehrt werden kann. Der Prozess der Entwicklung von Software kann als ein hochkomplexer angesehen werden, weswegen nicht garantiert werden kann, dass alle Phasen auf Anhieb erfolgreich abgeschlossen werden können. (Kleuker, 2009)

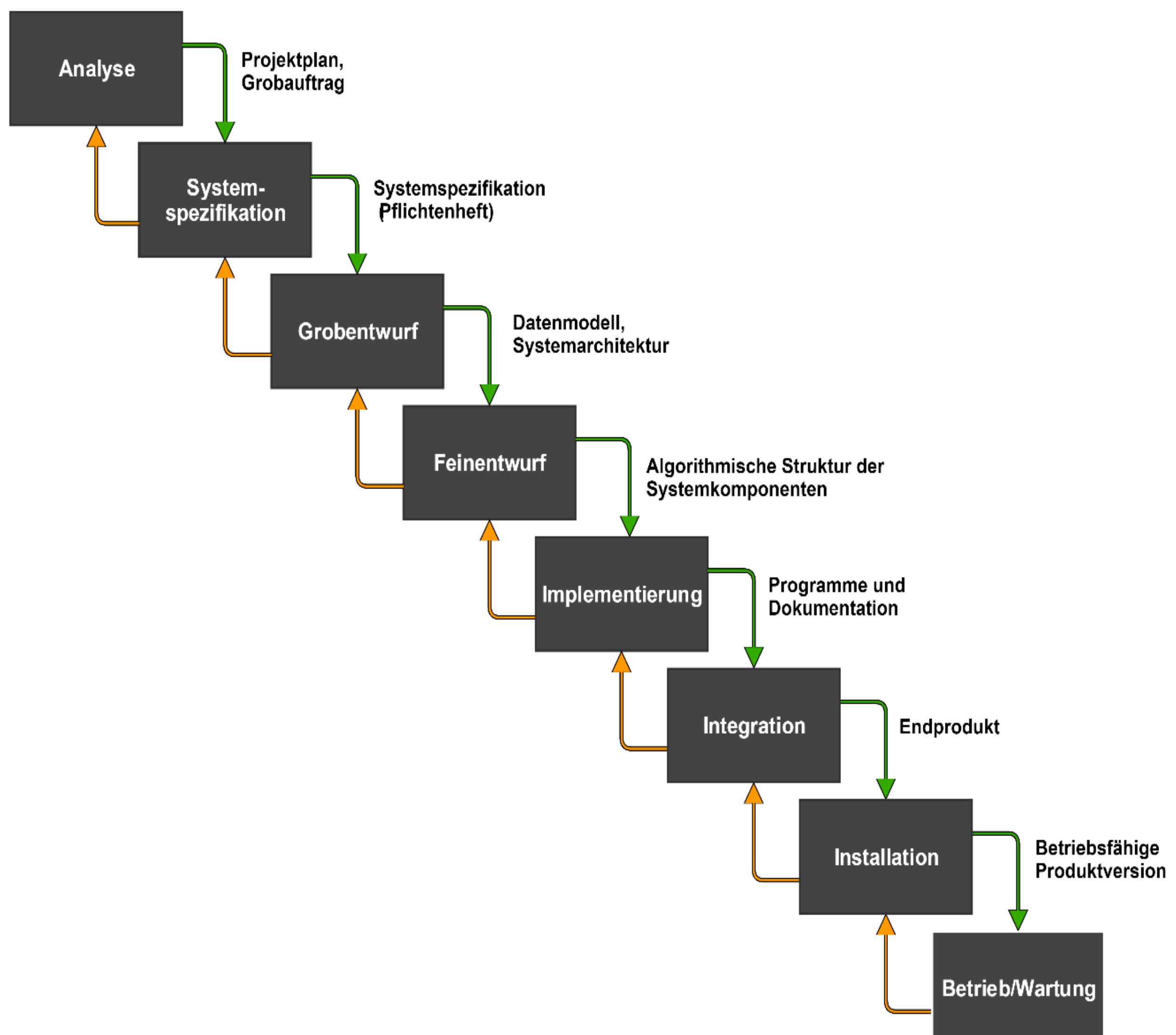


Abbildung 5 – Erweitertes Wasserfallmodell (Hansen & Neumann, 2009)

[9] Ist ein US-amerikanischer Software-Ingenieur und angesehener Professor der Informatik. Er ist bekannt für seine zahlreichen Beiträge auf dem Gebiet des Software Engineering.

Daher wurde das Wasserfallmodell, wie in Abbildung 5 zu sehen ist, um einige Phasen erweitert und lässt nun auch die Möglichkeit zu, in eine vorherige Phase zurückzukehren. Die einzelnen Phasen sehen wie folgt aus: (Sandhaus, Berg, & Knott, 2014)

1. Problemanalyse: Ein Konzept zur Umsetzung sowie zur Machbarkeit wird entwickelt. Nach erfolgreicher Validierung endet diese Phase mit einer groben Planung sowie einem Projektauftrag.
2. Systemspezifikation: In dieser Phase werden vom Auftragnehmer die einzelnen Anforderungen zum Beispiel in Form eines Pflichtenheftes, welches alle erforderlichen Funktionalitäten, die angestrebte Performance usw. enthält, definiert.
3. Grobentwurf: Stellt einen groben Entwurf der Hard- und Softwarearchitektur sowie Kontroll- und Datenstrukturen bereit.
4. Feinentwurf: Im Detail vollständiger, verifizierter Entwurf der einzelnen Spezifikationen und notwendigen Strukturen.
5. Implementierung: Eigentliche Entwicklung in Form von Programmcode basierend auf dem Feinentwurf.
6. Integration und Test: Einzelne entwickelte Komponenten werden zu einem System zusammengeführt und hinsichtlich definierter Qualitätskriterien getestet.
7. Installation: Deployen der Software. Unterstützung für den Anwender kann hier in Form von Handbüchern bzw. Schulungen geboten werden.
8. Betrieb und Wartung: Entwickelte Software wird genutzt. Während dieser Phase können kleine Korrekturen bzw. Weiterentwicklungen durchgeführt werden. Größere Änderungen führen zu einem Folgeprojekt.

Obwohl nun Rückführungen in vorherige Phasen möglich sind, müssen die einzelnen Phasen sequentiell abgearbeitet werden und es kann mit der nächsten Phase erst nach erfolgreichem Abschluss der vorherigen begonnen werden. In der Praxis hat sich oft gezeigt, dass es bei größeren Softwareprojekten praktisch unmöglich ist, alle Anforderungen zu einem gewissen Zeitpunkt zu kennen und zu formulieren. Ein Problem in einer früheren Phase wird daher oft erst sehr spät erkannt und kann schließlich zu einem kompletten Misserfolg des Projektes führen. (Kleuker, 2009)

Daher wurde 1979 von Barry W. Boehm ein weiteres Vorgehensmodell zur Entwicklung von Software, das V-Modell, das im nächsten Kapitel beschrieben wird, vorgestellt.

2.4.2 V-Modell

Das V-Modell von Boehm stellt in weiten Teilen eine Weiterentwicklung des beschriebenen Wasserfallmodelles dar und ist, wie Abbildung 4 zum Teil zeigt, in unterschiedlichen Ausprägungen existent. Zudem ist das V-Modell Basis für sehr viele aktuelle Entwicklungsmodelle wie zum Beispiel das V-Modell XT, das bereits Konzepte aus dem Bereich agiler Methoden integriert. Einsatz findet das V-Modell vor allem im öffentlichen Sektor wie zum

Beispiel in der deutschen Bundesverwaltung. (Grechenig, Bernhart, Breiteneder, & Kappel, 2010) Das Modell selbst ist kein abstraktes Modell und wird vor allem bei Großprojekten eingesetzt.

Unterteilt wird das V-Modell in Submodelle, welche sich mit Aufgaben der Systemerstellung, Qualitätssicherung, Konfiguration sowie dem Projektmanagement beschäftigen. Auf der linken Seite der V-förmigen Modellabbildung befinden sich alle für die Entwicklung einer Software benötigten Aktivitäten. Die rechte Seite zeigt Maßnahmen für die Integration und Qualitätssicherung. (Hansen & Neumann, 2009)

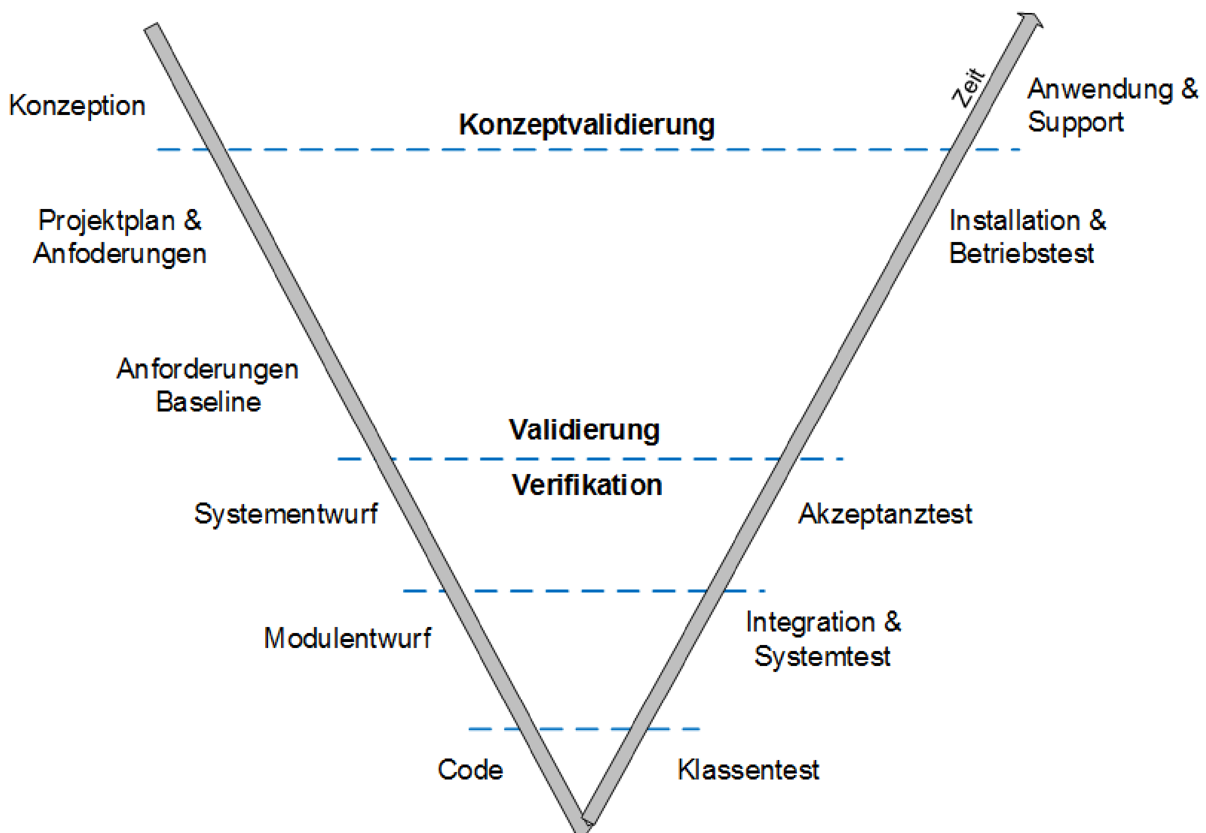


Abbildung 6 – Allgemeines V-Modell (Boehm, 1979)

Auch im V-Modell wird der Softwareentwicklungsprozess in Phasen organisiert. In diesem Modell werden jedoch die einzelnen Phasen des Wasserfallmodells um Mechanismen der Qualitätskontrolle erweitert. Hierbei wird unterschieden zwischen: (Sandhaus, Berg, & Knott, 2014)

- Validierung: Es wird überprüft, ob das „richtige“ System erstellt wurde.
- Verifizierung: Es wird überprüft, ob das System „richtig“, nach Plan erstellt wurde.

Wie in Abbildung 6 zu erkennen ist, hat das „V“ mehrere Bedeutungen. Es steht einerseits für das gesamte Vorgehen und die Modellformation entspricht dem Buchstaben „V“. Zusätzlich steht das „V“ für die bereits erwähnten qualitätssichernden Maßnahmen der Validierung und Verifizierung, die folgende Handlungen und Instrumente umfassen: (Osherove, 2009), (Boehm, 1979), (Sandhaus, Berg, & Knott, 2014)

- Klassen(Unit)-Tests: Mit diesen Tests werden zum Beispiel einzelne Methoden des Programmcodes hinsichtlich vorher definierter Kriterien auf deren Richtigkeit überprüft. Heutzutage passiert dies in der Regel bereits automatisiert.
- Integration & Systemtests: Diese testen mit Hilfe einer Reihe von Einzeltests die verschiedenen, zusammenhängenden Komponenten hinsichtlich ihres korrekten Zusammenspiels.
- Akzeptanztest: Dieser wird in der Regel zusammen mit dem Kunden durchgeführt und soll feststellen, ob die entwickelte Software den Anforderungen des Kunden entspricht.
- Installation & Betriebstest: Hier erfolgt die Validierung der ursprünglichen Anforderungen und es wird nachgewiesen, dass die Software ordentlich funktioniert.
- Anwendung & Support: Ermöglicht Hilfestellungen bei Fragen in der Anwendung sowie die erneute Validierung, ob das ursprüngliche Konzept korrekt ist.

Das V-Modell ist unter anderem wegen seiner sehr unterschiedlichen Ausprägungen eines der umfassendsten Vorgehensmodelle. Da es ein profundes Wissen voraussetzt, ist sein ordentlicher und erfolgreicher Einsatz jenseits von Großprojekten oftmals schwierig und problemanfällig. Ein Modell, das eher für Projekte mit einem hohen Anpassungsbedarf geeignet ist, wird im nächsten Kapitel beschrieben.

2.4.3 Spiralmodell

Das Spiralmodell ist ein ebenfalls von Barry W. Boehm im Jahre 1986 geschaffenes Vorgehensmodell zur Entwicklung von Software. Im Vergleich zu den anderen Modellen ist das Spiralmodell stark risikogetrieben und erlaubt es, verschiedene Vorgehensmodelle zu vermischen. (Kuhrmann, 2016)

Das Modell selbst gliedert sich, wie in Abbildung 7 zu sehen ist, in vier Phasen, welche wiederholt (iterativ) durchlaufen werden. (Vogel-Heuser, 2003)

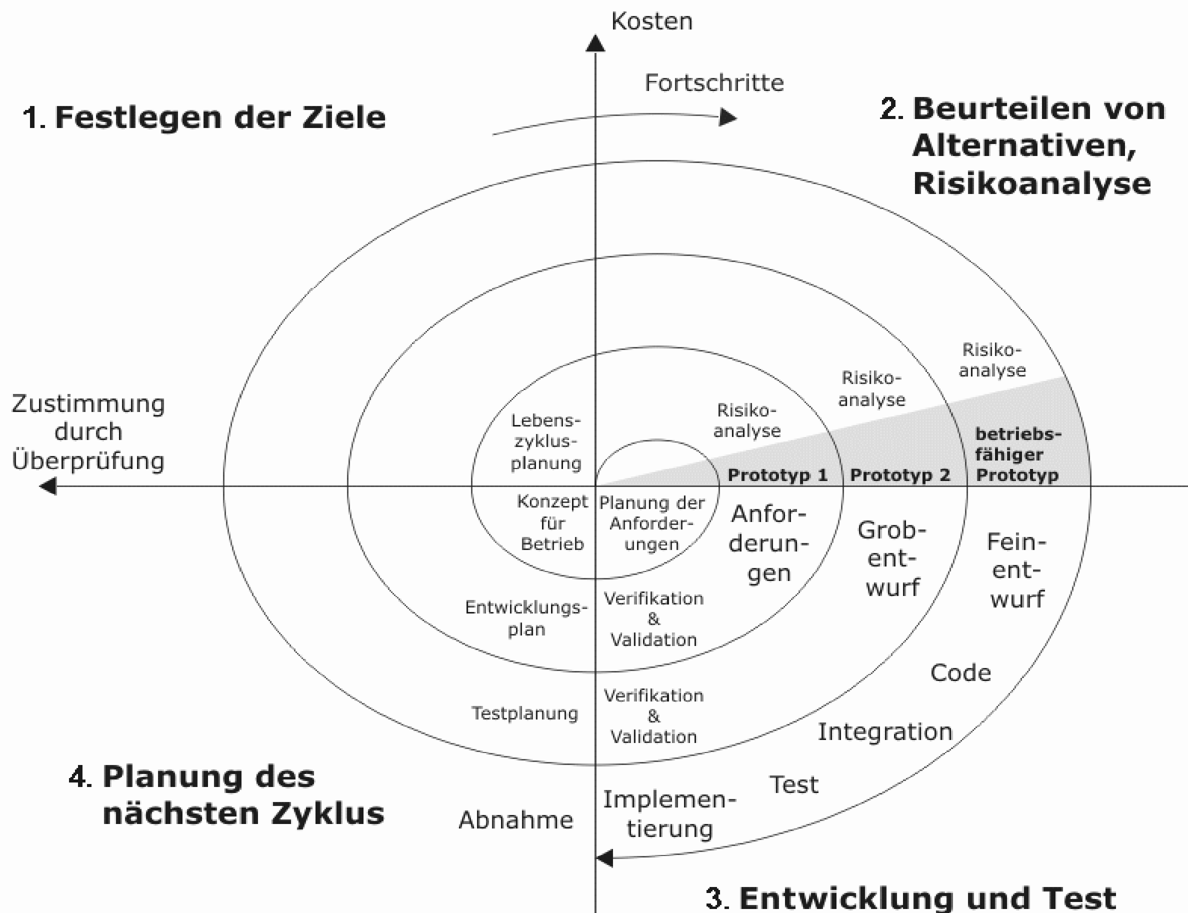


Abbildung 7 – Spiralmodell (Balzert, 2011)

Ebenfalls zu erkennen ist hier, dass einige der unter 2.4.1 bzw. unter 2.4.2 dargestellten und näher beschriebenen Elemente auch in diesem Modell Anwendung finden. Alle Phasen werden im Zuge einer evolutionären Entwicklung von Software mehrmals durchlaufen. Der Output jedes Durchlaufes basiert daher auf jenem der vorherigen Iteration und dient zudem als Grundlage für den nächsten Durchlauf. Ausgehend von den Basisanforderungen werden mit Hilfe des Spiralmodells sehr schnell Teillösungen entwickelt und mit jedem Zyklus schrittweise erweitert. Diese Teillösungen werden anschließend immer wieder in die Gesamtlösung eingearbeitet.

Die vier Phasen des Spiralmodells sehen wie folgt aus: (Grechenig, Bernhart, Breiteneder, & Kappel, 2010), (Pfetzing & Rohde, 2009)

1. Festlegen der Ziele: Jede Phase wird mit genauen Zielen und der Definition von Produkten begonnen. Zudem werden auch mögliche Alternativen evaluiert.
2. Beurteilung von Alternativen, Risikoanalyse: Die zuvor identifizierten Ziele und Alternativen werden bewertet und mögliche Risiken festgestellt. Für die identifizierten Risiken werden zudem Lösungsstrategien entwickelt. Hierfür können Prototypen, Simulationen, Modelle oder Benchmarks eingesetzt werden.
3. Entwicklung und Test: Entwicklung der eigentlichen Software mittels Programmcodes. Zudem findet hier die Qualitätskontrolle des entwickelten Teilproduktes mittels diverser Tests wie zum Beispiel Unit-Tests statt.

4. Planung des nächsten Zyklus: Abgeschlossen wir jeder Zyklus mit Reviews. Reviews sollen dazu dienen, Erkenntnisse aus dem abgeschlossenen Zyklus zu gewinnen, um diese in den nächsten Durchlauf einfließen lassen zu können.

Fehler und Risiken werden beim Spiralmodell sehr früh erkannt. Diesem Vorteil steht jedoch ein sehr hoher Managementaufwand gegenüber, wodurch das Modell für kleinere Projekte nicht wirklich zielführend ist.

Mit der Beschreibung des Spiralmodells findet die Aufzählung der gängigsten klassischen Modelle ihr Ende. Es gibt in der Praxis noch sehr viele weitere Modelle, die aber auf Grund des limitierten Umfanges dieser Arbeit hier nicht näher beschrieben werden können. Zusammenfassend kann über die dargestellten und erläuterten gängigsten klassischen Modelle gesagt werden, dass sie ihre Stärken und Vorteile insbesondere im Einsatz bei großen bzw. sehr großen Projekten haben und zur Geltung bringen. Bei der Entwicklung kleinerer bzw. mittelgroßer Projekte stoßen diese Modelle aber oft sehr schnell an ihre Grenzen. Hier empfiehlt es in der Regel, einen agilen Ansatz zur Softwareentwicklung zu verfolgen. Da diese Arbeit ein Vorgehensmodell für die Übergabe von Projekten in agile Teams beschreiben soll, wird auf die Charakteristik und Spezifika der agilen Softwareentwicklung in weiterer Folge detaillierter eingegangen.

2.5 Agile Vorgehensweise

Die in den vorherigen Kapiteln vorgestellten klassischen Modelle haben gemeinsam, dass sie sich zum Großteil nur für große bzw. sehr große Projekte eignen und zudem auf Grund ihrer Komplexität oftmals sehr abschreckend wirken. Des Weiteren ist es schwierig, bei Nutzung dieser Modelle auf Änderungen während der Entwicklung adäquat zu reagieren.

Abhilfe schaffen sollen hierbei die agilen Vorgehensweisen, die durch ihren ausgeprägt iterativen und inkrementellen Charakter deutlich mehr Flexibilität aufweisen. Im Vergleich zu den sehr statischen klassischen Modellen sind die agilen im Kern als eine Art Sammlung von Methoden und Best Practices zu verstehen. Um den Bezug zu den bereits beschriebenen phasenorientierten Vorgehensmodellen beizubehalten, wird weiter der Begriff Modell genutzt. (Sandhaus, Berg, & Knott, 2014)

Agile Modelle charakterisiert also, dass sie versuchen, den Prozess zur Entwicklung von Software flexibler und schlanker zu gestalten. Der Übergang zur agilen Entwicklung ist schon länger im Gange. Er hat in den späten 1980er bzw. zu Beginn der 1990er Jahre mit dem objektorientierten Ansatz begonnen, dessen Ziele es sind: (Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013)

- Die Produktivität durch Wiederverwendbarkeit zu steigern.
- Die Codemenge durch Reduktion und Vererbung zu reduzieren.
- Eine Erleichterung bei Codeänderungen durch kleinere, austauschbare Codebausteine zu erreichen.
- Die Fehlerauswirkungen durch Kapselung von Codebausteinen zu reduzieren.

Es gab weitere unterschiedliche Bestrebungen, agile Modelle in der Softwareentwicklung zu etablieren. Wirklich populär wurde die agile Softwareentwicklung ab dem Jahre 1999. In diesem Jahr veröffentliche Kent Beck das erste Buch zum Thema Extreme Programming (XP). Bis zum Jahre 2001, in dem siebzehn Autoren das „Agile Manifest" verfassten, war agile Entwicklung mit dem Label „leichtgewichtig" versehen. (Rau, 2016)

Im nächsten Kapitel werden nun das Agile Manifest, dessen Grundsätze und Werte im Detail beschrieben.

2.5.1 Das Agile Manifest

Im Frühjahr 2001 trafen Begründer vieler unterschiedlicher agiler Modelle zusammen und entwickelten, basierend auf ihren Erfahrungen und Überzeugungen, eine gemeinsame Basis für alle agilen Modelle. Dieser Leitfaden für agile Softwareentwicklung ist unter dem Namen „Agile Manifesto" (Agiles Manifest) bekannt. Das Agile Manifest beinhaltet vier Aussagen zu Werten und zwölf Prinzipien als Grundlage für alle agilen Praktiken. (Rau, 2016), (Sandhaus, Berg, & Knott, 2014), (Beck, et al., 2016)

Die vier Leitsätze zu agilen Werten im Überblick: (Baron & Hüttermann, 2010)

1. Individuen und Interaktionen mehr als Prozesse und Werkzeuge: Wird sehr häufig falsch verstanden. Sagt aus, dass Prozesse und Werkzeuge zwar von großer Bedeutung für Projekte sind, aber die Interaktion und Kommunikation mit den Stakeholdern (Kunden, Entwicklungsteam, usw.) noch viel wichtiger ist. Den Menschen machen die Projekte und nicht Werkzeuge und Maschinen.
2. Funktionierende Software mehr als umfassende Dokumentation: Die Dokumentation der Software ist zwar sehr wichtig, jedoch ist es wesentlich wichtiger, lauffähige Programme zu haben. Kunden ordern ja die Software. Dennoch darf die Dokumentation nicht komplett vernachlässigt werden.
3. Zusammenarbeit mit Kunden mehr als Vertragsverhandlung: Wenn am Ende eines Projektes nur mehr Diskussionen über den Vertrag aufkommen, ist es vermutlich gescheitert. Wichtiger ist eine ständige Interaktion mit Kunden. Denn im Zuge der Entwicklung können sich Anforderungen laufend ändern. Am Ende des Tages zählt immer das Resultat, nämlich eine gebrauchsfähige Software.
4. Reagieren auf Veränderung mehr als das Befolgen eines Plans: Es ist zwar wichtig, einen groben Fahrplan zu haben, jedoch darf dieser Plan nicht in Stein gemeißelt sein. Es ist viel wichtiger, flexibel und offen auf Änderungen zu reagieren.

Zusammengefasst heißt das: Obwohl die Werte auf der rechten Seite sehr wichtig sind, werden trotzdem jene auf der linken Seite höher eingeschätzt. Die daraus formulierten zwölf Prinzipien sollen helfen, die in den Leitsätzen formulierten Werte in der Praxis einzuhalten: (Beck, et al., 2016)

1. Unsere höchste Priorität ist es, Kunden durch frühe und kontinuierliche Auslieferung wertvoller Software zufrieden zu stellen.

2. Komplexe Anforderungsänderungen selbst spät in der Entwicklung sind willkommen. Agile Prozesse nutzen Veränderungen zum Wettbewerbsvorteil für Kunden.
3. Liefere funktionierende Software regelmäßig innerhalb weniger Wochen oder Monate und bevorzuge dabei die kürzere Zeitspanne.
4. Fachexperten und Entwickler müssen während des Projektes täglich zusammenarbeiten.
5. Errichte Projekte rund um motivierte Individuen. Gib ihnen das Umfeld und die Unterstützung, die sie benötigen und vertraue darauf, dass sie die Aufgabe erledigen.
6. Die effizienteste und effektivste Methode, Informationen an und innerhalb eines Entwicklungsteams zu übermitteln, ist das Gespräch von Angesicht zu Angesicht.
7. Funktionierende Software ist das wichtigste Fortschrittsmaß.
8. Agile Prozesse fördern nachhaltige Entwicklung. Die Auftraggeber, Entwickler und Benutzer sollten ein gleichmäßiges Tempo auf unbegrenzte Zeit halten können.
9. Ständiges Augenmerk auf technische Exzellenz und gutes Design fördert Agilität.
10. Einfachheit ist die Kunst, die Menge nicht getaner Arbeit zu maximieren. Diese Einfachheit ist essenziell.
11. Die besten Architekturen, Anforderungen und Entwürfe entstehen durch selbstorganisierte Teams.
12. In regelmäßigen Abständen reflektiert das Team, wie es effektiver werden kann und passt sein Verhalten entsprechend an.

Aus der eigenen Erfahrung heraus hat sich gezeigt, dass es nicht immer einfach ist, diese definierten Werte und Leitsätze einzuhalten. Viel schwieriger jedoch ist es oft, Kunden von dem agilen Weg zu überzeugen und ihnen diesen näher zu bringen. Denn häufig wurde in den betroffenen Unternehmen noch mit Hilfe klassischer Vorgehensmodelle gearbeitet.

Alle agilen Vorgehensmodelle bauen auf diese Aussagen zu Werten und Prinzipien als Grundlage für alle agilen Praktiken auf und bieten zudem Hilfestellungen, um alle in das „agile Boot" zu holen. In den folgenden Kapiteln werden einige der am häufigsten eingesetzten agilen Modelle beschrieben.

2.5.2 Extreme Programming (XP)

Eines der am häufigsten eingesetzten und bekanntesten agilen Vorgehensmodelle ist jenes des Extreme Programming (XP). Entstanden ist dieses agile Modell im Zuge von Arbeiten von Kent Beck für Chrysler. Es gab zu dieser Zeit kein geeignetes Vorgehensmodell, mit welchem auf sich ständig ändernde Anforderungen reagiert werden konnte. Im Jahre 1999 wurde das erste Werk von Kent Beck zum Thema XP veröffentlicht. (Rau, 2016)

Der Ansatz von XP lässt sich wie folgt charakterisieren:

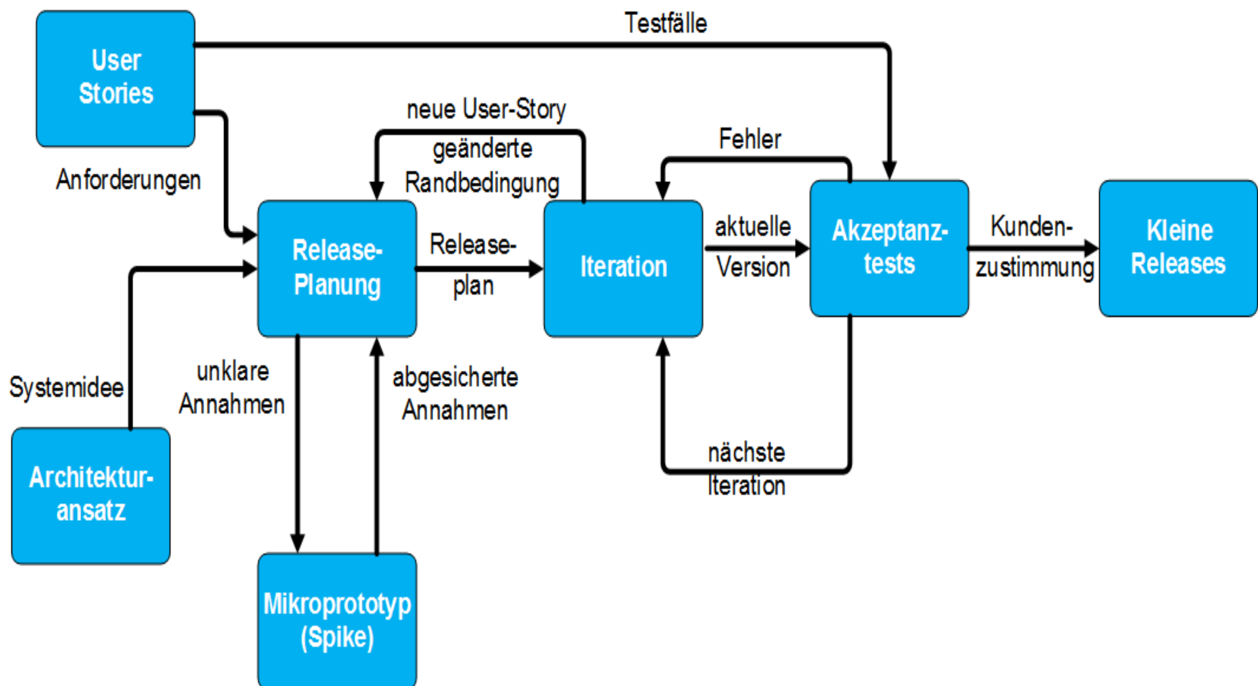


Abbildung 8 – Extreme Programming (XP) (Kleuker, 2009)

XP zeichnet sich durch das in Abbildung 8 grob dargestellte Phasenmodell aus. Statt der in den klassischen Modellen üblichen detaillierten Anforderungen werden hier User Stories, die gewünschte Arbeitsschritte der Software aus der Nutzersicht beschreiben, als Ausgangspunkt eingesetzt. Da es sich hier nicht um präzise formulierte Anforderungen handelt, ist der Aspekt, dass Kunden immer greifbar sein müssen, zentral. Mit Hilfe des Architekturansatzes wird die grobe Architektur der geplanten Software definiert. Aufbauend auf die formulierten User Stories werden die Aufwände geschätzt und zusammen mit den Kunden die Entwicklung anhand von Iterationen geplant. Änderungswünsche haben generell immer Vorrang. Am Ende einer jeden Iteration soll ein kleines Release (lauffähige Version einer Software), die mit Hilfe unterschiedlicher Tests validiert wird, zur Verfügung gestellt werden. Diese Methodik erlaubt es, schnell Ergebnisse zu präsentieren und die zuvor erstellten Anforderungen zu validieren bzw. anzupassen. Des Weiteren sind in XP folgende Techniken wichtig: (Kleuker, 2009)

- Fortschrittsmessung passiert anhand der abgearbeiteten und geschätzten User Stories.
- Projektteilnehmer arbeiten an unterschiedlichen Aufgaben. Jeder Teil des Programmcodes kann von jedem bearbeitet werden.
- Am Beginn jedes Arbeitstages findet ein sogenanntes Stand-Up-Meeting statt. Diese Meetings sollen in kurzer Zeit einen schnellen Überblick über die vorherigen und anstehenden Arbeiten sowie aktuelle Probleme geben.
- Eine schnelle Lösung ist immer zu bevorzugen. Optimierungen werden nach hinten verschoben. Erscheint eine Lösung in einem nächsten Schritt als nicht mehr optimal, wird sie durch einen Umgestaltungsprozess (Refactoring) angepasst.

- Vor der eigentlichen Software werden die notwendigen Tests geschrieben. Die Ergebnisse dieser Tests sind anschließend für alle Projektteilnehmer sichtbar.
- Die Entwicklung der Software findet mittels Pair-Programming statt. Pair-Programming heißt, dass zwei Entwickler gleichzeitig an der Lösung arbeiten.
- Es gibt keine Überstunden.

Nachdem einige der für XP typischen Techniken sowie einige der in Abbildung 8 dargestellten Begrifflichkeiten und Abläufe beschrieben wurden, wird im nächsten Kapitel ein weiteres agiles Vorgehensmodell beschrieben, nämlich Scrum.

2.5.3 Scrum

Ursprünglich kommt der Begriff „Scrum“ aus dem Rugby, wo er für das Gedränge der zwei Mannschaften steht, die versuchen den Ball zu erreichen und zu diesem Zweck die andere Mannschaft zurück in ihre Platzhälfte schieben müssen. Hierbei ist eine enge Zusammenarbeit aller Beteiligten notwendig. Als Begründer von Scrum als agiles Vorgehensmodell zur Entwicklung von Software gelten Ken Schwaber und Jeff Sutherland, die sich zu Beginn der 1990er Jahre damit beschäftigten, wie man Software schneller und flexibler entwickeln kann. (Kleuker, 2009), (Dräther, Koschek, & Sahling, 2013), (agilealliance.org, 2016)

Scrum ist kein Model mit konkreten Arbeitsanweisungen für Teams, sondern gibt lediglich einen gewissen Projektrahmen vor. Dabei definiert sich das Scrum Framework anhand von Regeln für fünf Aktivitäten, drei Artefakte und drei Rollen, welche zum Teil auch schon aus dem Ansatz des XP bekannt sind (vgl. Kapitel 2.5.2). Damit das Framework wirklich funktioniert, muss es durch Techniken für die Umsetzung der Aktivitäten, Artefakte und Rollen ergänzt werden. Alle genutzten Elemente in Scrum basieren auf denselben Werten: (Dräther, Koschek, & Sahling, 2013)

- Selbstverpflichtung: Eine Verpflichtung zu einem Ziel ist unbedingt notwendig. Durch Scrum erhält man die nötigen Befugnisse, diese Verpflichtungen auch zu erfüllen.
- Fokus: Es ist wichtig, sich auf die gegebenen Aufgaben zu konzentrieren. Alles rund herum soll einen nicht belasten.
- Offenheit: Alle Informationen werden transparent allen zur Verfügung gestellt.
- Respekt: Teammitglieder sind Individuen und unterscheiden sich durch ihre Erfahrungen und persönlichen Hintergründe. Es ist daher wichtig, die verschiedenen Menschen im Team zu respektieren.
- Mut: Es ist notwendig, alles offen anzusprechen und wichtig, zu seiner Meinung zu stehen. Auch wenn andere Teammitglieder etwas anders sehen. Mut ist daher besonders wichtig. Übermut sollte aber tunlichst vermieden werden.

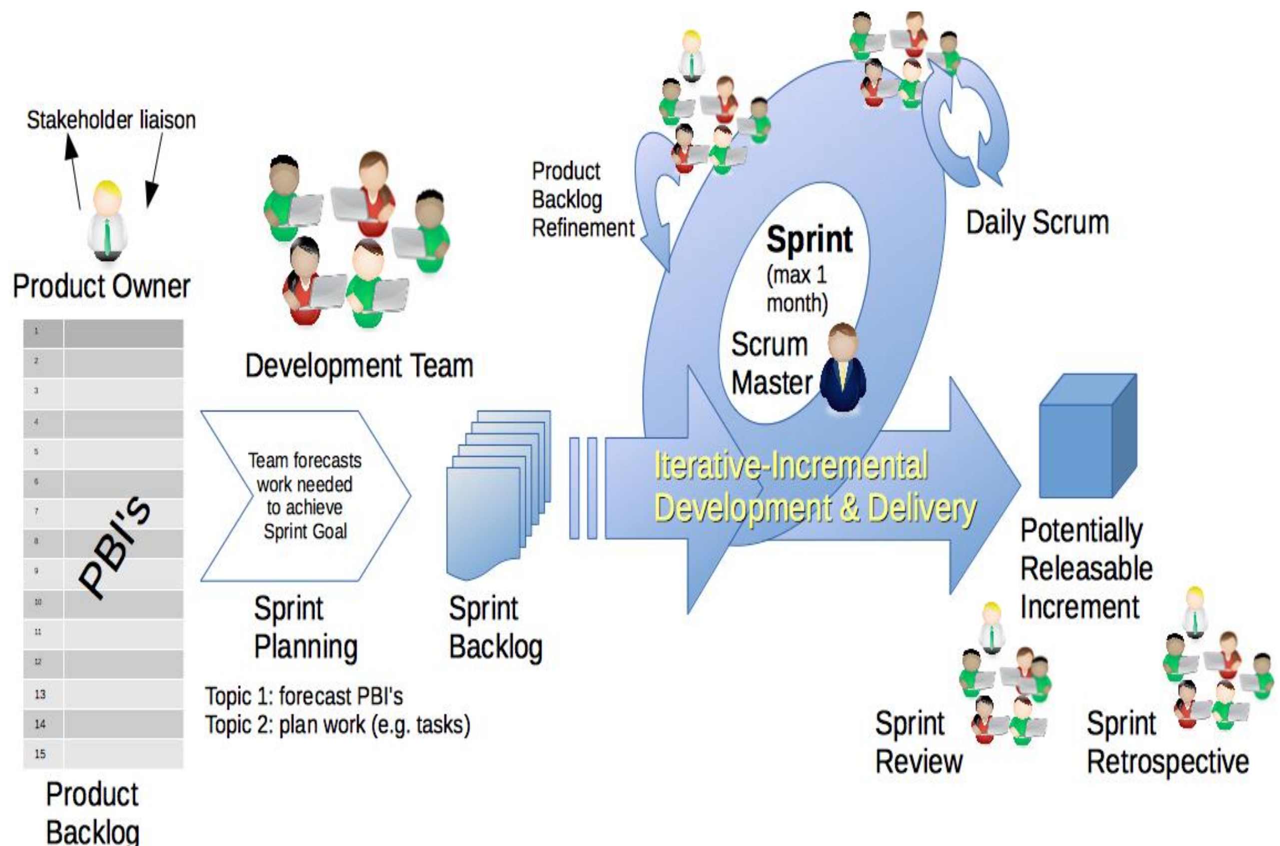


Abbildung 9 – Übersicht Scrum (Mitchell, 2016)

Abbildung 9 zeigt einen Überblick über die wichtigsten Mechanismen von Scrum sowie dessen empirischen[10], iterativen bzw. inkrementellen Ansatz. Aus der anfänglichen Vision einer Software werden konkrete Eigenschaften für diese abgeleitet. Diese werden anschließend vom Product Owner (PO) im Product Backlog organsiert und priorisiert. Der PO ist für den wirtschaftlichen Erfolg des Projektes verantwortlich und wird diesen Umstand bei der Priorisierung auch entsprechend einfließen lassen. Die Abarbeitung der Anforderungen erfolgt in Sprints, also Iterationen mit fester und immer gleicher Länge. Diese Sprints sollen nicht länger als 30 Tage sein. In der Praxis kann es durchaus vorkommen, Sprints immer wieder anzupassen, bis das optimale Setup gefunden wurde. Die eigentliche Umsetzung der Anforderungen wird anschließend vom Entwicklungsteam bewerkstelligt. (Roock & Wolf, 2016)

Die drei Rollen laut Scrum: (Roock & Wolf, 2016)

1. Product Owner (PO): Ist der fachliche Projektleiter und für den wirtschaftlichen Erfolg des Projektes verantwortlich. Der optimale Produktnutzen wird durch die Priorisierung hinsichtlich der Produkteigenschaften erreicht. Der PO ist zudem für das Product Backlog verantwortlich. Alle Anfragen in diesem müssen vom PO gepflegt, ordentlich formuliert, priorisiert und transparent für alle dargestellt werden.

[10] Erfahrungsgemäß. Auf der Erfahrung beruhend. In Bezug auf Scrum – Aus der vorherigen Iteration die Lehren ziehen und verbessern.

2. Entwicklungsteam: Besteht aus drei bis neun Mitgliedern. Es ist mit Mitarbeitern unterschiedlicher Fähigkeiten besetzt, um das Sprint Backlog in ein lieferbares Produktinkrement[11] umzuwandeln. Daher ist ein Scrum-Team nicht nur mit Entwicklern besetzt, sondern kann auch Experten im Bereich Testing, Design oder Customer Service umfassen. Im Vergleich zu anderen Modellen organisiert sich ein Scrum-Team selbst und trägt auch gemeinsam die Verantwortung.
3. Scrum Master: Ist für das korrekte Anwenden von Scrum verantwortlich. Scrum Master kümmern sich aber nicht nur um das „Scrum-konforme" Handeln des Teams, sondern schützen zudem das Team vor unnötigen Einflüssen von außen und beseitigen Hindernisse. Zudem können Scrum Master bei der Moderation der Meetings herangezogen werden und helfen dabei, Scrum zu verstehen. Kurz gesagt machen Scrum Master alles, um die Performance des Teams zu erhalten und zu verbessern.

Zusätzlich zu den beschriebenen Rollen kann ein Scrum-Team noch weitere Rollen wie Anwender, Kunden, Manager usw. beinhalten. (Dräther, Koschek, & Sahling, 2013) Die Aufgaben eines Scrum-Teams umfassen folgende fünf Aktivitäten: (Roock & Wolf, 2016)

1. Sprint Planning: Mit dem Sprint Planning wird vom gesamten Scrum-Team der Beginn eines neuen Sprints eingeläutet. Das Meeting selbst wird in zwei Teile gesplittet, welche immer am selben Ort zur selben Zeit stattfinden sollten. Im ersten Teil des Meetings wird geklärt, „Was" im nächsten Sprint umgesetzt werden soll. Voraussetzung hierfür ist ein ordentlich gepflegtes Product Backlog. Der zweite Teil beschäftigt sich mit den technischen Aspekten, also damit, „Wie" die Anforderungen umgesetzt werden. Am Ende kommt ein definiertes Sprint-Ziel heraus, das später in der Sprint Review evaluiert wird.
2. Daily Scrum: Ähnlich dem Stand-Up-Meeting von XP soll das Daily Scrum in knapp 15 Minuten abklären, was gestern gemacht wurde, was heute gemacht wird und welche Schwierigkeiten auftreten bzw. aufgetreten sind.
3. Sprint Review: Ist ein Meeting, in welchem das Team die Ergebnisse und Qualität der gelieferten Software präsentiert. Dies passiert meistens in einem erweiterten Rahmen. So sind bei Sprint Reviews oft Kunden und andere Stakeholder anwesend. Moderiert kann dieses Meeting von Scrum Mastern werden. In der Sprint Review werden zudem die definierten Sprint-Ziele evaluiert und Feedback gesammelt.
4. Sprint-Retrospektive: Ist ein Meeting des gesamten Scrum-Teams, im Normalfall ohne Beteiligung des Kunden. Hier werden die Arbeit bzw. die Zusammenarbeit des gesamten Teams im vorherigen Sprint evaluiert und Verbesserungen vorgeschlagen.
5. Product Backlog Refinement: Auch Backlog Grooming genannt, beschreibt die Pflege des Product Backlog mit Hilfe des gesamten Scrum-Teams. Aufgaben sind zum Beispiel das Ordnen, Detaillieren oder Schätzen von Anforderungen bzw. die Planung der Releases.

[11] Beschreibung siehe Seite 23 – Beschreibung der Artefakte von Scrum.

Ein Scrum-Prozess kann natürlich durch weiterführende Aktivitäten ergänzt werden, sollte aber so knapp wie möglich gehalten werden. Die drei definierten Artefakte von Scrum sind: (Roock & Wolf, 2016)

1. Product Backlog: Ist das zentrale Artefakt zu den Produktdefinitionen und wird von Product Ownern gepflegt und mit Blick auf den optimalen Produktnutzen priorisiert. Es beschreibt die von außen wahrnehmbaren Produkteigenschaften.
2. Sprint Backlog: Besteht aus den am höchsten priorisierten Items des Product Backlog, die für den Sprint zur Umsetzung ausgewählt wurden. Meistens werden die Items in kleinere Aufgaben (Tasks) gesplittet, in denen kein erkennbarer Nutzen für das Produkt erkennbar sein muss.
3. Product Increment: Ist das Ergebnis des Scrum-Teams nach jedem Sprint und den fertiggestellten Einträgen aus dem Product Backlog. Es stellt eine Erweiterung und oder Änderung des bisherigen Produkts dar. Das neue Inkrement muss nach einem Sprint in einem einsatzfähigen Zustand sein, auch wenn es vom Product Owner möglicherweise noch nicht ausgeliefert wird.

Neben den von Scrum vorgegebenen Rollen, Aktivitäten und Artefakten werden in der Praxis oftmals in Kombination noch weitere Techniken wie beispielsweise

- Definition of Done (DoD): Gemeinsames Verständnis, wann eine Anforderung als fertig angesehen kann.
- Planning Poker: Ist eine nützliche Methode, um gemeinsam Aufwände bzw. Komplexität von Anforderungen abzuschätzen.
- Taskboard: Visualisierung der Anforderungen in einem Board.
- Sprint Burndown Chart: Dient zur Messung des Entwicklungsfortschrittes.

eingesetzt. (Dräther, Koschek, & Sahling, 2013) Wie bereits beschrieben, ist Scrum kein striktes Vorgehensmodell, sondern gibt lediglich einen Rahmen vor. Daher kann, wenn notwendig, der Scrum-Prozess durch zahlreiche weitere Techniken ergänzt werden.

Nach Vorstellung des agilen Modells von Scrum in diesem Kapitel wird im nächsten Kapitel die Darstellung und Erläuterung der wichtigsten agilen Vorgehensmodelle mit der Beschreibung von Kanban vervollständigt.

2.5.4 Kanban

Kanban kommt aus dem Japanischen und bedeutet ursprünglich „Signalkarte“. Der Begriff steht für eine Technik aus dem Toyota-Produktionssystem, mit der ein gleichmäßiger Fluss in der Fertigung und eine Reduzierung der Lagerbestände erreicht werden sollte. Der Fokus liegt hierbei auf dem optimalen Fluss jedes einzelnen Produktes durch die Fertigung. Kanban in der IT übernimmt zwar den Namen, versucht aber keine direkte Übertragung einzelner Techniken aus

der Produktion. Als Begründer von Kanban als Vorgehensmodell für die IT gilt David Anderson[12], der das Gesamtkonzept 2007 veröffentlichte. Kanban besteht aus sechs Kernpraktiken, gibt aber keine Vorschriften für deren praktische Umsetzung vor, sondern unterbreitet diesbezüglich lediglich Vorschläge. Dabei geht Kanban davon aus, dass die bestehenden Prozesse bereits Potential aufweisen und selbst die Kraft haben, eine Weiterentwicklung zu erreichen. (Leopold & Kaltenecker, 2013)

Die sechs Kernpraktiken von Kanban sind: (Leopold & Kaltenecker, 2013)

1. Sichtbarmachen der Arbeit: Um einen Mehrwert bei Kunden generieren zu können, ist es essentiell, einen kontinuierlichen Arbeitsfluss zu etablieren. Kanban soll dabei helfen, sowohl die Abläufe als auch deren Probleme sichtbar zu machen. Wesentlicher Unterschied zu anderen Vorgehensmodellen ist, dass Kanban die Aufgaben nicht in den nächsten Arbeitsschritt weiterschiebt (Push-Prinzip), sondern sie aus einem vorgelagerten holt, wenn Kapazitäten verfügbar sind (Pull-Prinzip).

2. Limitierung des Work in Progress (WIP): Die Anzahl der Aufgaben, die gleichzeitig in einem Status bearbeitet werden, wird limitiert. Das heißt, wenn ein Limit von zwei etabliert wurde und sich gerade zwei Anfragen in Bearbeitung befinden, darf keine dritte Anfrage hinzukommen. Hiermit wird wieder das Pull-Prinzip gefördert.

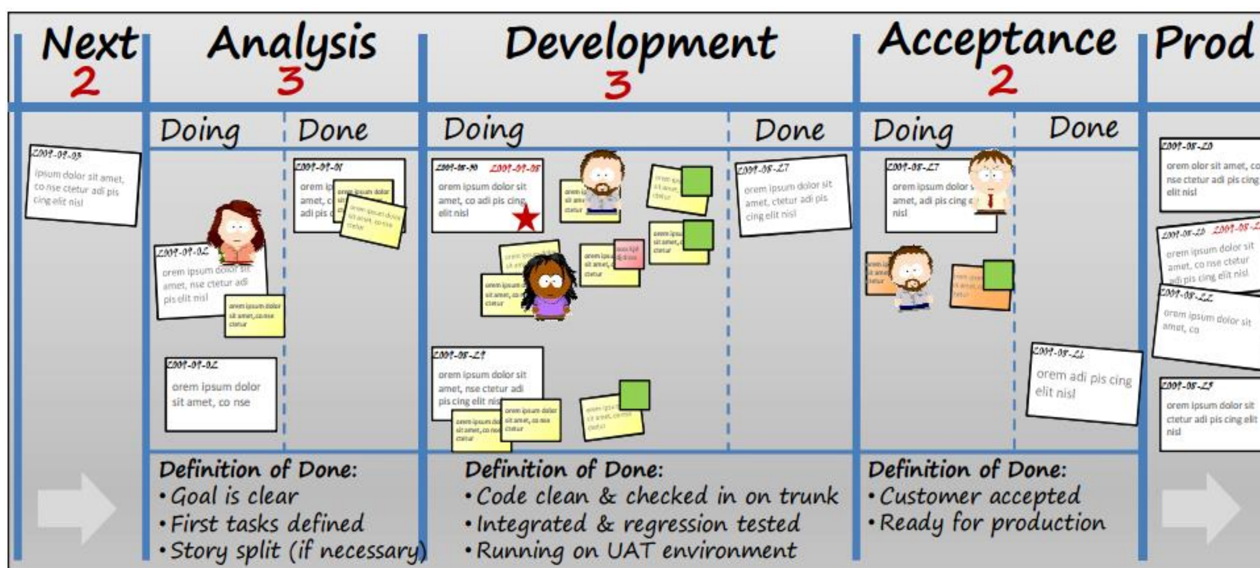
3. Messung und Steuerung des Flusses: Der Fokus wird auf den Arbeitsfluss gelegt, es wird alles unternommen, damit die Arbeit korrekt und wie vereinbart abgeliefert werden kann. Hierbei hilft zum Beispiel die Einführung von Serviceklassen als Grundlage für Service Level Agreements (SLA). Die Abläufe werden ständig gemessen, um Verbesserungen erreichen zu können.

4. Prozessregeln müssen explizit sein: Damit jeder am Prozess Beteiligte weiß, welche Annahmen und Regeln für die Arbeitsabläufe gelten, müssen diese explizit gemacht werden. Zu diesen gehören unter anderem detaillierte Beschreibungen sowie, ähnlich der DoD in Scrum, eine Definition dazu, wann eine Aufgabe fertig ist.

5. Installation von Feedback-Mechanismen: Ein weiterer Fokus von Kanban liegt auf der kontinuierlichen Verbesserung. Damit man sich verbessern kann, ist es notwendig, Feedback zu erhalten bzw. zu geben. Kanban bedient sich hier ähnlicher Mechanismen wie Scrum, also zum Beispiel Retrospektiven oder täglicher Meetings.

6. Gemeinsame Verbesserungen basierend auf Modellen und wissenschaftlichen Methoden: Für den erfolgreichen Umgang mit einer Vielzahl von Problemstellungen gibt es bereits entsprechende Modelle und Praktiken. Kanban sieht zwar vor, sich dieser Modelle zu bedienen, schreibt aber nicht vor, welche Modelle wann eingesetzt werden müssen.

[12] Ist ein US-amerikanischer Spezialist in den Bereichen effektiver Technologie-Entwicklung.



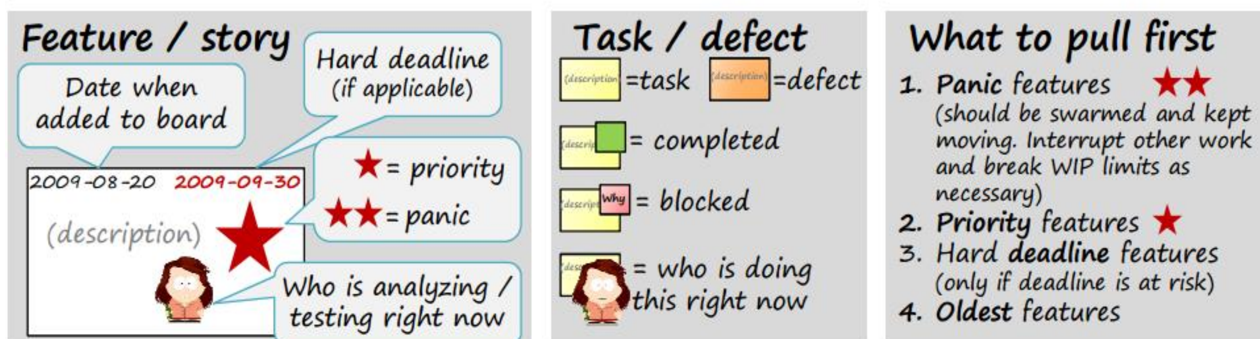


Abbildung 10 – Beispiel Kanban Board (Kniberg, 2016)

Abbildung 10 zeigt, wie die Umsetzung der zuvor beschriebenen Kernpraktiken von Kanban in Form eines Boards aussehen könnte. Zusammengefasst bietet Kanban also zwar einige Kernpraktiken, aber keine konkreten Anweisungen, wie diese umzusetzen sind. Diese Freiheit ist einerseits eine große Stärke von Kanban, ist man allerdings auf der Suche nach Best Practices, wird daraus eine der größten Schwächen dieses Modells. Mit Kanban steht also ein Diagnoseinstrument zur Verfügung, um Probleme sichtbar zu machen, es bietet selbst jedoch keine Methode zur Lösung der Probleme an. (Leopold & Kaltenecker, 2013) Der Unterschied zum in 2.5.3 beschriebenen Modell Scrum ist also, dass Kanban kein Vorgehensmodell im klassischen Sinne ist und daher u.a. auch keine Aufgaben, Ergebnistypen und Rollen vorgibt. Vielmehr geht es in Kanban vorwiegend darum, Arbeitsabläufe korrekt zu steuern und zu visualisieren. (Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013)

In der Zusammenschau lässt sich sagen, dass es in der Praxis sehr viele unterschiedliche Modelle zur Softwareentwicklung mit je eigenen Methoden und Herangehensweisen gibt, die alle ihre Daseinsberechtigung haben. Jedoch ist es wichtig, dass richtige Modell für das eigene Projekt zu wählen. Oftmals bedient man sich auch bei der Entwicklung von Software der Best Practices der einzelnen Modelle und schafft somit Mischformen. Scrumban zum Beispiel ist eine Mischform aus Scrum und Kanban und versucht, die Vorteile beider Modelle zu vereinen.

Was alle Modelle, seien sie nun klassisch, agil oder kombiniert, gemeinsam haben, ist, dass sie Projekte über den gesamten oder einen Teil des Software Life Cycle (Softwarelebenszyklus)

begleiten. Daher wird nun, bevor ein neues Modell entwickelt werden kann (siehe dazu Kapitel 40), im nächsten Kapitel kontinuierliche Auslieferung von Software beschrieben sowie der Software-Lebenszyklus vorgestellt und seine Spezifika erörtert.

3 KONTINUIERLICHE AUSLIEFERUNG VON SOFTWARE

Wie im vorherigen Kapitel beschrieben, gibt es sehr viele unterschiedliche Ansätze bzw. Modelle zur Softwareentwicklung. Diese Modelle unterscheiden sich in vielen Punkten und sind in der Praxis mit Blick auf die jeweils zu entwickelnde Software nicht alle gleich gut bzw. zielführend einsetzbar. Was diese Modelle jedoch gemeinsam haben, ist, dass sie die Entwicklung von Software bis zu einem bestimmten Punkt, also über eine oder mehrere Phasen des Software-Lebenszyklus hinweg begleiten. Wie zum Beispiel der menschliche Organismus hat auch Software einen bestimmten Lebenszyklus und durchläuft gewisse Phasen. Ziel muss es sein, kontinuierlich qualitativ hochwertige Software ausliefern zu können.

Bevor nun der Ansatz von kontinuierlicher Integration von Software in Kapitel 3.3 beschrieben wird, soll zuerst der allgemeine Software-Lebenszyklus definiert werden.

3.1 Software-Lebenszyklus

Jedes Projekt zur Entwicklung von Software ist unterschiedlich. Dennoch gibt es einige gemeinsame Elemente, aus welchen ein allgemeiner Software-Lebenszyklus abgeleitet werden kann. Dieser allgemeine Lebenszyklus lässt sich wie folgt darstellen:

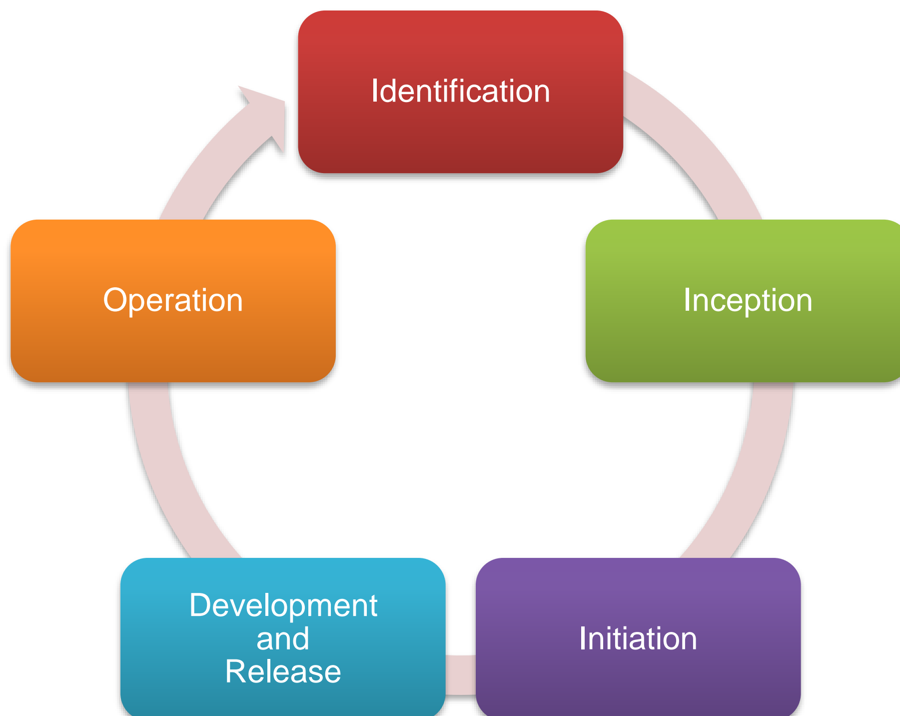


Abbildung 11 – Lebenszyklus einer Software (Humble & Farley, 2011)

Wie in Abbildung 11 dargestellt, beginnt der Lebenszyklus in der Regel mit der Ermittlung der Spezifikation und endet nach der Phase der Wartung, mit der Abschaffung, Ablösung oder Weiterentwicklung des Produktes (Balzert, 2011). Die einzelnen Phasen des Lebenszyklus werden in den nächsten Kapiteln grob beschrieben. Da die Anwendung des neuen

Vorgehensmodells für ein agil entwickeltes Projekt vorgesehen ist, werden die klassischen Vorgehensmodelle ab diesem Punkt nicht mehr berücksichtigt.

3.1.1 Identification

Im Normalfall folgen Unternehmen gewissen Strategien und entwickeln daraus das Bedürfnis für eine Software, welche die Erreichung der definierten Ziele unterstützen soll. In dieser Phase sollen diese Bedürfnisse definiert und am besten in einem Business Case[13] festgehalten werden.

Auch wenn es sehr oft vorkommt, dass die tatsächliche Software stark von diesem abweicht, ist ein Business Case für diese Phase essentiell. Ohne Business Case ist es anschließend sehr schwierig, die korrekten Anforderungen zu sammeln und objektiv zu priorisieren. Zusätzlich ist es in dieser Phase besonders wichtig, eine Liste mit allen Stakeholdern zu erstellen, welche von dem Projekt „betroffen“ sind. Nicht berücksichtigte Stakeholder können zu einem Projektabbruch führen! Stakeholder sind unter anderem auch das Operations-Team[14], Verkauf, Marketing, Customer Service, Entwicklungs- und Testteams, usw. Um nicht in politische Kämpfe innerhalb eines Projektes zu kommen, ist es wichtig, einen Hauptverantwortlichen für das Projekt zu definieren. In Scrum wird dies beispielsweise vom PO durchgeführt. Fragen, die unter anderem in dieser Phase evaluiert werden sollen, sind:

- Wer wird das System benutzen?
- Wie wird das System genutzt werden?
- Welche Daten sollen in das System importiert werden können?
- Welche Daten sollen aus dem System exportiert werden können?

Letztendlich soll ein gemeinsames Verständnis über das zu entwickelnde System in dieser Phase geschaffen werden, um anschließend in der nächsten Phase (Inception) alle Stakeholder ordentlich repräsentieren zu können. (Humble & Farley, 2011):

3.1.2 Inception

Kurz beschrieben ist dies die Phase, in der alle Voraussetzungen geschaffen werden, damit anschließend Software in Form von Programmcode entwickelt werden kann. In dieser Phase werden die Anforderungen gesammelt, analysiert, grob in den Anwendungsbereich übersetzt und anschließend geplant. Zudem werden sehr viele benötigte Artefakte generiert. Je nachdem welches Modell eingesetzt wird, können diese variieren. Was in dieser Phase aber auf jeden Fall als Artefakte herauskommen sollte, sind:

- Ein weiterer Business Case, welcher den erwarteten Wert des Projektes inkludiert.

---

[13] Wird im Vorfeld eines Projekts angewendet, um zum Beispiel die Rentabilität eines Geschäftsfalles oder um die Auswirkungen auf die Wirtschaftlichkeit zu untersuchen. Wichtige Bestandteile sind die Analyse von Kosten, Nutzen, Risiken usw. (Taschner, 2013)

[14] Zuständig für den operativen Betrieb, also Serverwartung, usw.

- Eine Liste mit high-level Funktionen und nicht funktionalen Anforderungen. Wichtig sind hier auch Anforderungen in Richtung Sicherheit, Verfügbarkeit, usw. Die Liste sollte detailliert genug sein, um die notwendige Arbeit abschätzen und einen Plan erstellen zu können.
- Eine Release und Test-Strategie.
- Eine Evaluierung der benötigten Architektur Diese führt anschließend zur Entscheidung, welche Plattform und welches Framework genutzt werden soll.
- Eine Beschreibung des angestrebten Entwicklungszyklus und ein Plan zur Abarbeitung der Listen.

Diese Artefakte sollten genug detailliert sein, damit mit der eigentlichen Arbeit am Projekt begonnen werden kann. Zudem müssen diese auf jeden Fall zentralisiert abgelegt werden. Da es sich um lebende Dokumente handelt, werden sich diese im Zuge der Entwicklung noch verändern und müssen daher auch aktuell gehalten werden. (Humble & Farley, 2011)

Eine der wichtigsten Tätigkeiten in dieser Phase ist zudem, alle zuvor definierten Stakeholder (Kunden, Entwickler, Management, Customer Service, Operations-Team, usw.) an einen Tisch zu holen. Diesen werden dann die entwickelten Artefakte vorgestellt, um anschließend ein gemeinsames Verständnis des Projekts zu schaffen. Dieser Schritt ist besonders wichtig, damit das Projekt überhaupt eine Chance auf Erfolg hat.

Anzumerken ist hier noch, dass jede Entscheidung in dieser Phase auf Spekulationen beruht und es sehr wahrscheinlich ist, dass sich im Laufe des Projektes noch einiges ändert. Daher muss der Aufwand für diese Phase im Rahmen bleiben. Detaillierte Planungen, Schätzungen oder Design in dieser Phase sind meistens eine Geld- und Zeitverschwendung. Es ist aber dennoch wichtig, alles so gewissenhaft wie möglich abzuarbeiten, um anschließend ordentlich auf etwaige Änderungen reagieren zu können. (Humble & Farley, 2011)

Mit Abschluss dieser Phase sollten alle Voraussetzungen für den Beginn der Umsetzung der eigentlichen Software geschaffen sein. Die Umsetzung beginnt in der nächsten Phase (Initiation) mit der Schaffung der notwendigen Projektinfrastrukturen.

3.1.3 Initiation

Im Anschluss an die Phase Inception, in welcher alle Voraussetzungen geschaffen werden, um mit der eigentlichen Umsetzung beginnen zu können, soll in der Phase Initiation die nötige Projektinfrastruktur geschaffen werden. Diese Phase dauert durchschnittlich zwischen einer und zwei Wochen und enthält grob folgende Tätigkeiten: (Humble & Farley, 2011)

- Es muss sichergestellt werden, dass das gesamte Team die Hardware und Software hat, um mit der Arbeit beginnen zu können.
- Zudem muss sichergestellt werden, dass die Basisinfrastruktur, etwa Verbindung zum Internet, benötige Arbeitsutensilien wie Papier und Stifte, usw. vorhanden ist.

- Alle benötigten E-Mail-Accounts müssen erstellt und die Zugriffe des Teams auf die benötigten Systeme eingerichtet werden.
- Die Versionskontrolle[15] muss eingerichtet und ein Setup für die kontinuierliche Integration von Software erstellt werden.
- Arrangements über die einzelnen Rollen, Verantwortlichkeiten, Arbeitszeiten und die einzelnen Meetings werden getroffen.
- Die Arbeit für die ersten Wochen wird anhand von Zielen vorbereitet. Wichtig ist, dass die Ziele keine Deadlines darstellen!
- Eine einfache Testumgebung mit Testdaten sollte aufgesetzt werden.
- Ein weiteres wichtiges Ziel dieser Phase ist es, einen detaillierten Blick auf das geplante System zu werfen. Hiermit sollen dessen Potentiale erkannt werden.
- Alle Analyse-, Entwicklungs- und Testrisiken müssen identifiziert und wenn möglich gleich gemindert werden.
- Das Story- bzw. Anforderungsbacklog sollte erstellt werden.
- Die eigentliche Projektstruktur sollte anhand der simpelsten Anforderungen, inklusive Build Scripts[16] und Tests erstellt werden.

Für die einzelnen Aufgaben in dieser Phase sollte genug Zeit aufgewendet werden. Dies ist wichtig, denn es ist sehr unproduktiv und demoralisierend, wenn die notwendigen Grundvoraussetzungen wie funktionierende Hardware nicht vorhanden sind.

Da diese Phase zur Schaffung der benötigten Projektstruktur und nicht als wirkliche Iteration der Entwicklung genutzt werden soll, ist es extrem hilfreich, die Arbeit mit wirklichen Aufgabenstellungen zu beginnen. Das Aufsetzen von zum Beispiel der Testumgebung, der Versionskontrolle usw. sollte also anhand der simpelsten Anforderung vonstattengehen. Damit kann sichergestellt werden, dass die geschaffene Infrastruktur den Anforderungen des zu entwickelnden Systems genügt. (Humble & Farley, 2011)

Sind alle notwendigen Voraussetzungen für das Projekt geschaffen, kann in der nächsten Phase mit der eigentlichen Entwicklung begonnen werden.

3.1.4 Development and Release

Außer bei extrem großen Projekten, in welche übermäßig viele Parteien involviert sind, eignen sich, wie auch unter 2.5 beschrieben, Modelle mit iterativen und inkrementellen Eigenschaften. Obwohl viele Teams den Vorteilen von iterativem Vorgehen zustimmen und der Meinung sind,

[15] Dient zur Verfolgung von Änderung und macht es möglich, frühere Versionen wiederherzustellen.

[16] Sind ausführbare Skripte mit welchen die Software dann „gebaut“ wird.

dass sie auch iterativ entwickeln, tun sie es dennoch nicht. Daher sollen an dieser Stelle die Basiskonditionen für ein iteratives Vorgehen darstellt werden: (Humble & Farley, 2011)

- Die Software funktioniert korrekt, wenn die Demonstration anhand der definierten Unit-, Komponenten und Akzeptanztests gelingt. Diese Tests laufen jedes Mal, wenn neue Änderungen bzw. Code ins Repository eingecheckt[17] werden.
- In jeder Iteration wird die funktionierende Software auf ein System deployed, welches ähnlich wie das spätere Life System[18] aufgesetzt ist. Auf diesem System wird die Software dann Usern vorgeführt. Dieser Schritt macht die Vorgehensweise inkrementell.
- Die Iterationen dauern nicht länger als zwei Wochen.

Die Gründe, um solche agilen Modelle zu nutzen, sind wie folgt: (Humble & Farley, 2011)

- Da die Anforderungen anhand von wirtschaftlichen Aspekten priorisiert und umgesetzt werden, wird Software oftmals vor dem eigentlichen Projektende als nützlich empfunden. Dadurch können auch Bedenken einiger Stakeholder oftmals sehr früh zerstreut werden.
- Durch regelmäßiges Feedback einzelner Stakeholder können laufend Anforderungen angepasst werden. Niemand kennt diese am Anfang eines Projektes genau. Auch wenn manche oft dieser Meinung sind!
- Dinge sind nur wirklich erledigt, wenn sie auch von Kunden abgesegnet worden sind. Durch regelmäßige Vorführungen der Fortschritte ist dies der einzige zuverlässige Weg.
- Dadurch, dass die Software für regelmäßige Vorführungszwecke funktionieren muss, wird das Team diszipliniert. Zudem muss nach jeder Iteration ein produktionsreifer Code zur Verfügung stehen. Dies führt zur einzigen wirklich sinnvollen Messung von Fortschritt in Softwareprojekten, die nur iterative Modelle bieten.

Da die Software nach jeder Iteration produktionsreif sein muss, müssen auch nicht funktionale Anforderungen wie Sicherheit, Verfügbarkeit usw. berücksichtig werden. Dies sollte anhand von automatisierten Tests, wie zum Beispiel Lasttests passieren. Die Schlüssel zu einer iterativen Entwicklung sind Priorisierung und Parallelisierung. Anforderungen werden anhand ihres Wertes priorisiert und es wird versucht, die Anzahl der Leute, die gleichzeitig an einer Aufgabe arbeiten, zu modifizieren. Dies führt zu einem sehr effizienten Entwicklungsprozess und soll Flaschenhälse verhindern. (Humble & Farley, 2011)

Nach Abschluss dieser Phase steht auf jeden Fall ein fertiges Release für den möglichen Einsatz auf einem Produktivsystem bzw. die letzte Phase des Lebenszyklus zur Verfügung.

---

[17] Repository ist der zentrale Ablagepunkt des Programmcodes. Einchecken beschreibt den Vorgang, wenn neuer Code hinzugefügt wird.

[18] Ist jenes System, auf dem die designierte Software von Kunden genutzt wird.

3.1.5 Operation

Im Normalfall ist das erste Release nicht das letzte. Was nun genau in dieser Phase passiert, hängt sehr stark vom Projekt ab. Es könnte zum Beispiel sein, dass die eigentliche Entwicklung mit voller Kraft weitergeführt wird, aber auch eine Reduzierung bzw. Erhöhung der Anzahl der Teammitglieder könnte die Folge sein.

Ein sehr interessanter Aspekt kontinuierlichen agilen Arbeitens ist, dass sich die operative Phase nicht von den eigentlichen Phasen der Entwicklung unterscheiden muss. So werden im Normallfall immer weiter neue Features entwickelt und weitere Releases gebaut. (Humble & Farley, 2011)

Während dieser Phase werden auch alle Tätigkeiten ausgeführt, die notwendig sind, um die Performance der Software aufzuzeichnen, die User zu unterstützen und um Probleme beim Auftreten zu beseitigen. Beispielsweise kann es notwendig sein, per Hotfix[19] den Stand der Software zu korrigieren. (Hallerstede, 2013)

Ausgehend von diesen Tätigkeiten werden immer wieder neue Features identifiziert und anschließend priorisiert, analysiert, entwickelt und getestet. Wie auch in den Phasen der regulären Entwicklung. Das regelmäßige Feedback von „echten“ Usern ist für die Nutzbarkeit, aber auch die Benutzbarkeit (Usability) sehr wichtig. Daraus lässt sich ableiten, dass die enge Verbindung dieser Phase mit jenen der Entwicklung sehr wichtig ist, um kontinuierlich Software ausliefern zu können und um zudem die Risiken zu minimieren. (Humble & Farley, 2011)

Mit der Phase Operation schließt sich der Kreis des Software-Lebenszyklus. Die Tätigkeiten und Inhalte der einzelnen Phasen können von Projekt zu Projekt unterschiedlich sein. Es wäre auch durchaus denkbar, dass einzelne Tätigkeitsfelder in andere Teams ausgelagert bzw. Projekte komplett an andere Teams übergeben werden. Unabhängig davon, wie die einzelnen Phasen gehandhabt werden, lässt sich sagen, dass es besonders wichtig ist, in regelmäßigen Abständen Software in hochwertiger Qualität auszuliefern.

Bevor der bisher noch nicht berücksichtige Aspekt der Übergabe eines bestehenden Projektes in ein anderes agil arbeitendes Team bzw. die dabei zu beachtenden Einflussfaktoren behandelt werden, wird im nächsten Kapitel der Ansatz der kontinuierlichen Auslieferung von Software näher beleuchtet und anschließend werden die ersten Schritte zur kontinuierlichen Auslieferung von Software anhand der Praxis der kontinuierlichen Integration vorgestellt.

3.2 Das Problem der regelmäßigen Auslieferung von Software

Das größte Problem, dem Softwarespezialisten gegenüberstehen, ist die Frage: Wie kann eine gute Idee schnell und in hochwertiger Qualität an die User geliefert werden? In den nächsten

[19] Ist eine Aktualisierung der Software, um akute Fehler zu beheben. Der Fehler ist dabei so gravierend, dass er sofort behoben werden muss und nicht erst in einem nächsten Release.

Kapiteln wird auf dieses Problem im Detail eingegangen und ein möglicher Ansatz vorgestellt. Dabei wird der Fokus auf Build-[20], Deploy-, Test- und Release-Prozesse gelegt. (Humble & Farley, 2011)

Es gibt, wie bereits in den Kapiteln zuvor beschrieben, sehr viele unterschiedliche Methoden, um Software zu entwickeln. Der Fokus dieser Methoden liegt zumeist im Management der Anforderungen und deren Einfluss auf den Erfolg der Entwicklung. Fragen zum Beispiel danach, was passiert, nachdem die Anforderungen identifiziert, eine Lösung konzipiert bzw. entwickelt und getestet wurde, oder wie diese Aktivitäten miteinander verbunden bzw. die Zusammenarbeit der einzelnen Teams verbessert werden, um den Prozess noch effizienter zu machen, werden oftmals nicht zur Genüge beantwortet. Daher sind Muster notwendig, um effektiv und effizient Software von der Entwicklung bis zum Deployment zu begleiten. (Humble & Farley, 2011)

Eines dieser Muster ist die sogenannte Deployment Pipeline. Die Deployment Pipeline ist eine automatisierte Implementierung der eigenen Build-, Deploy-, Test- und Release-Prozesse. Abhängig von dem Wertestrom jedes Unternehmens kann sich die Implementierung der Deployment Pipeline unterscheiden. Die Prinzipien variieren jedoch nicht.

Abbildung 12 – Deployment Pipeline (Humble & Farley, 2011)

Abbildung 12 zeigt, wie eine Deployment Pipeline in der Praxis aussehen könnte und soll verdeutlichen, dass jede durchgeführte Änderung an der Konfiguration, dem Programmcode, der Umgebung oder den Daten eine neue Instanz der Pipeline kreiert. Die ersten Schritte der Pipeline beschäftigen sich mit der Erstellung der Binärdateien[21] und Installer[22]. Im übrigen Teil der Pipeline wird eine Serie von Tests auf diese Binärdateien angewendet, um diese zu validieren und anschließend zu veröffentlichen. Jeder erfolgreiche Test steigert das Vertrauen in die Releasekandidaten und sagt aus, dass die Kombination von Binärcode, Konfiguration, Umgebung und Daten funktionieren wird. (Humble & Farley, 2011)

Ziele dieser Deployment Pipeline sind: (Humble & Farley, 2011)

1. Alle Teile der Build-, Deploy-, Test- und Release-Prozesse sollen, um die Zusammenarbeit zu fördern, für alle Beteiligten transparent dargestellt werden.

[20] Prozess zur Erstellung der Software.

[21] Ist eine Datei, deren Inhalt von einem Programm oder Hardwareprozessor interpretiert werden muss. Es muss vorher schon bekannt sein, wie die Datei formatiert ist. (Rouse, 2016)

[22] Installiert die gewünschte Software auf dem Zielsystem.

2. Feedback soll verbessert werden, um Probleme so früh wie möglich identifizieren und lösen zu können. Dies bedeutet aber auch, dass eine standardisierte Feedbackschleife eingeführt werden muss. Dies ist besonders wichtig, damit bei Auftritt eines Fehlers eine Benachrichtigung an die zuständigen Personen und eine Fehlerbehandlung erfolgt.
3. Das Team soll in der Lage sein, jede beliebe Version der Software in jede Umgebung zu deployen. Vorzugsweise durch einen automatisierten Prozess.

Aus diesen Zielen lassen sich nun drei wesentliche Werte für die kontinuierliche Auslieferung von Software ableiten: (Wolff, 2016)

1. **Regelmäßigkeit:** Alle Prozesse, die zur Auslieferung von Software notwendig sind, sollen kontinuierlich ausgeführt werden und nicht nur dann, wenn ein Release in Produktion gebracht werden muss. Beispielsweise müssen für die einzelnen Projekte die notwendigen Umgebungen, wie Test- oder Produktionsumgebung, sowie die notwendigen Tests geschaffen werden. Da dies im Normalfall nicht nur für ein Projekt durchgeführt werden muss, kann daraus ein automatisierter Prozess zur Bereitstellung der Umgebungen entwickelt werden. Regelmäßigkeit führt daher meistens zu Automatisierung.
2. **Nachvollziehbarkeit:** Alle an der Software und Infrastruktur durchgeführten Änderungen müssen nachvollziehbar sein. Jeder Stand von Software bzw. Infrastruktur muss daher rekonstruiert werden können. Dafür notwendig ist eine Versionierung, die nicht nur die Software umfasst. Damit kann im Fehlerfall einfach das passende System aufgebaut und Änderungen dokumentiert oder auditiert werden.
3. **Regression:** Das Risiko bei der Auslieferung von Software sollte minimiert werden. Daher sind ausreichende Tests notwendig und es muss, um Regression[23] zu vermeiden, die korrekte Funktion der neuen Features sichergestellt werden. Da der manuelle Weg hier sehr aufwendig ist, sind automatisierte Tests notwendig.

Auch wenn sehr viele Unternehmen einen Großteil der Tätigkeiten zur Auslieferung von Software bereits automatisiert haben, zeigt die Praxis, dass die Veröffentlichung (Deployment) meistens noch manuell vollzogen wird. Die Tage, an denen dies erfolgt, sind sehr oft besonders stressig und angespannt für die involvierten Parteien. Der Hauptgrund dafür ist, dass die manuelle Veröffentlichung von Software ein besonders hohes Risiko birgt und in vielen Softwareprojekten die manuell durchgeführten Prozessschritte sehr intensiv und aufwendig sind. (Humble & Farley, 2011)

Die diversen Umgebungen sind häufig individuell zusammengebaut und man ist von anderer Software (Third-party Software) abhängig. Ein gängiges Vorgehen ist es, die notwendigen Elemente wie Softwareartefakte, Konfigurationsinformationen und Daten auf das Produktivsystem zu kopieren. Anschließend wird die Applikation Schritt für Schritt gestartet. Der

[23] Mit Hilfe von Regressionstests sollen nicht nur die neuen Funktionen in sich überprüft werden, sondern auch der Rest des Systems. Damit soll sichergestellt werden, dass die neuen Änderungen nicht unerwartete Auswirkungen haben. (Liggesmeyer, 2002)

Grund, warum dies eine sehr stressige und fehleranfällige Vorgehensweise ist, sollte klar sein: Wenn irgendein Schritt nicht ordentlich genug ausgeführt wurde, wird die Applikation nicht korrekt funktionieren. An diesem Punkt ist es dann oftmals sehr schwierig herauszufinden, in welcher Prozessphase der Fehler gemacht wurde. (Humble & Farley, 2011) Der Aufwand bzw. die Kosten für die Behebung diverser Probleme können sehr beträchtlich sein und das System kann während des Deployment von Kunden nicht genutzt werden.

Zusammengefasst gesagt, kann der Prozess des manuellen Deployen ein sehr aufwendiger und risikoreicher sein. Menschliche Fehler können nicht ausgeschlossen werden. Zudem kommt es in der Praxis häufig vor, dass nicht nur „händisch" deployed wird. Sehr oft werden auch die benötigten Systeme, wie lokale Entwicklungsumgebung, Stage System (Spiegelung der Produktivumgebung) sowie Testsysteme manuell konfiguriert und die Änderungen nicht mittels Versionskontrolle verfolgt. Dies kann erstens massive Auswirkungen beim Deployen haben und des Weiteren sehr viele Folgeprobleme hervorrufen.

Damit das Ziel der Auslieferung von qualitativ hochwertiger und nützlicher Software in einer effizienten, schnellen und zuverlässigen Art und Weise erreicht werden kann, ist es daher notwendig, regelmäßige automatisierte Releases zu erzeugen: (Humble & Farley, 2011)

- Automatisiert: Wenn die einzelnen Build-, Deploy-, Test- und Release-Prozesse nicht automatisiert sind, ist es schwierig diese zu wiederholen. Denn jede Änderung an Software, Konfiguration, usw. schafft eine neue Situation. Durch die manuelle Durchführung der einzelnen Schritte ist es nicht möglich, im Detail zu rezensieren, was genau gemacht wurde und man erhält keine Kontrolle über den Release-Prozess.
- Regelmäßig: Wenn Software regelmäßig veröffentlich wird, sind die Unterschiede zur Vorgängerversion im Normallfall recht gering. Dies reduziert zum einen die Risiken und zum anderen verhilft es zu einem rascheren Feedback. Zudem ist es durch „kleinere" Änderungen ein Rollback[24] einfacher auszuführen.

Feedback ist eines der wichtigsten Dinge, wenn automatisiert und regelmäßig Software ausgeliefert werden soll und es sollte folgende drei Kriterien erfüllen: (Humble & Farley, 2011)

1. Egal welche Änderung durchgeführt wird, es muss immer der Feedbackprozess angestoßen werden: Eine funktionierende Softwareapplikation kann in die vier Teile ausführbarer Code, Konfiguration, Serverumgebung und Daten zerlegt werden. Ändert sich etwas in einer der vier Komponenten, kann das Auswirkungen auf die gesamte Applikation haben. Daher müssen diese Komponenten unter Kontrolle gehalten und eventuelle Änderungen verifiziert werden.
2. Das Feedback muss so schnell wie möglich an die zuständigen Personen gelangen: Der Schlüssel zu schnellem Feedback ist die Etablierung automatisierter Prozesse. Manuelle Prozesse generieren stets eine potentiell prekäre Abhängigkeit von den Leuten, die diese Aufgaben ausführen sollen.

[24] Rollback bedeutet die Rückkehr zu einer funktionierenden Vorgängerversion im Fehlerfall. (Murray, 2016)

3. Die Zielpersonen müssen das Feedback annehmen und darauf reagieren: Es ist essentiell, dass alle Personen, die in den Prozess der Auslieferung von Software involviert sind, auch in den Feedbackprozess miteingebunden werden. Dies schließt sowohl die eigentlichen Entwickler, Tester, aber auch den Bereich Operations, Infrastrukturspezialisten, Manager usw. mit ein. Letztlich ist Feedback nutzlos, solange nicht darauf reagiert wird. Dies erfordert Disziplin und detaillierte Planung.

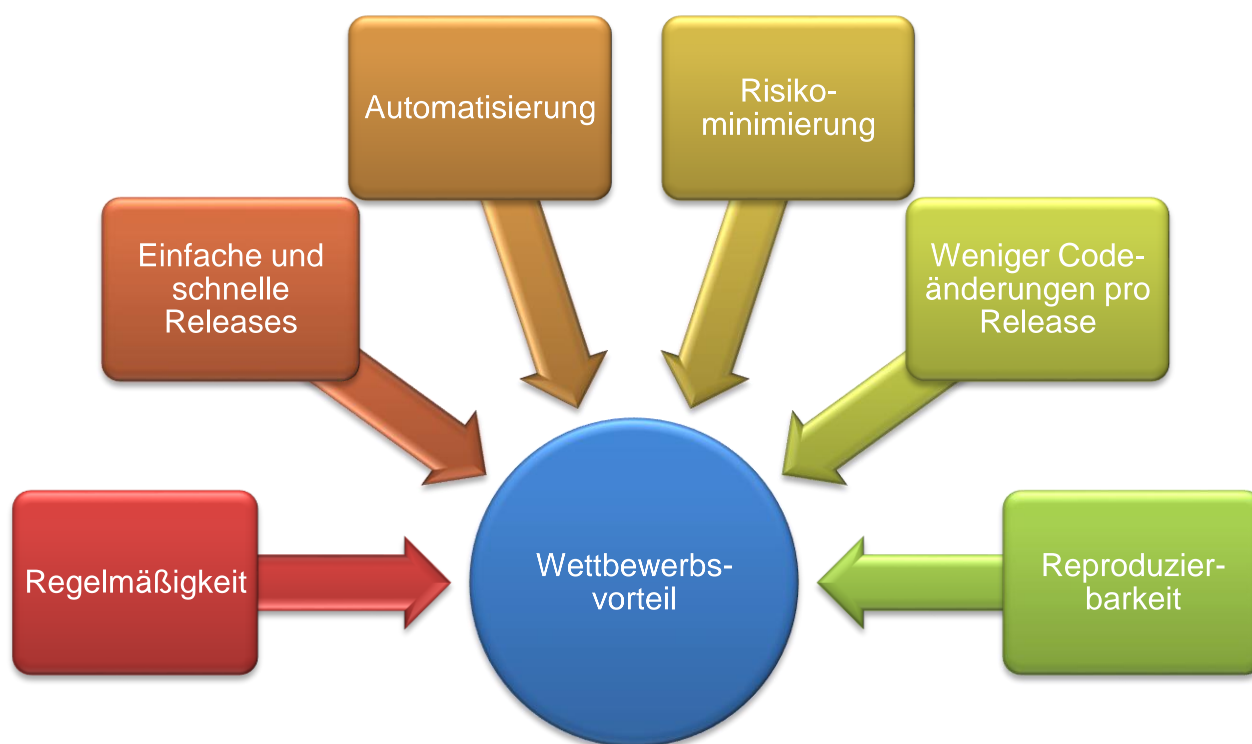


Abbildung 13 – Vorteile von kontinuierlicher Auslieferung von Software (Wolff, 2016)

Wenn man die Techniken für automatisierte und regelmäßige Build-, Deploy-, Test- und Release-Prozesse anwendet, erhält man, wie in Abbildung 13 gezeigt, sehr viele Vorteile und letztendlich kann dadurch ein Wettbewerbsvorteil generiert werden. Änderungen können nachvollzogen werden, Prozesse sind für unterschiedliche Umgebungen reproduzierbar, viele Fehlerquellen können ausgeschlossen bzw. das Risiko minimiert werden. Durch regelmäßiges und schnelles Feedback kann im Fehlerfall schnell reagiert werden, man erlangt schneller einen Nutzen und kann sich im Regelfall dadurch klar von der Konkurrenz abheben.

Wenn man nun solche automatisierten Prozesse im Unternehmen einführen möchte, sollten auch andere Best Practices wie Behavior Driven Development (BDD)[25] in Betracht gezogen werden. Insgesamt erhält man dadurch, neben den bereits erwähnten Vorteilen für die Software, auch mehr Freizeit und kann Stress wesentlich reduzieren. (Humble & Farley, 2011)

[25] Ist eine Technik der agilen Softwareentwicklung. Die evaluierten Ziele, Aufgaben usw. werden während der Anforderungsanalyse in Textform festgehalten, um anschließend darauf automatisierte Tests aufzubauen. (Hunt & Thomas, 2003)

Kontinuierliche Auslieferung von Software ist aber nichts, das von heute auf morgen in ein Unternehmen eingeführt werden kann. Um alle Tätigkeiten anhand der in Abbildung 12 gezeigten Deployment Pipeline automatisiert, regelmäßig und ordentlich ausführen zu können, müssen gewisse Voraussetzungen erfüllt sein. Begonnen wird hier mit einem Ansatz, der einen Großteil der zuvor beschriebenen automatisierten Prozesse von Build, Deploy, Test und Release behandelt und den Namen kontinuierliche Integration (Continuous Integration (CI)) trägt. Die allgemeinen Bestandteile eines CI-Systems werden im nächsten Kapitel beschrieben.

3.3 Kontinuierliche Integration von Software

Zum ersten Mal wurde von kontinuierlicher Integration (CI) im Jahre 1999 gesprochen. Dieser Begriff wurde im Zuge des agilen Modelles XP (vgl. Kapitel 2.5.2) von Kent Beck als Best Practice vorgestellt. Die Annahme dahinter war, dass wenn im Normalfall die Integration der Codebasis gut ist, warum solle sie das dann nicht die über gesamte Zeit sein. (Beck & Andres, 2005)

Dem Ansatz der CI liegt das seltsame, aber durchaus übliche Phänomen sehr vieler Softwareprojekte zugrunde, dass sich während eines sehr langen Zeitraumes des Entwicklungsprozesses die Applikation in einem nicht funktionierenden Zustand befindet. Das heißt, dass sehr viel Zeit respektive Aufwand für Software investiert wird, welche in einem unbenutzbaren Zustand ist. Der Grund dafür ist leicht gefunden. Niemand ist daran interessiert, die gesamte Applikation laufen zu lassen, bis sie fertig gestellt ist. Entwickler machen ihre Änderungen in Zweigen (Branches) und womöglich lassen sie sogar automatisierte Tests laufen, aber niemand lässt die gesamte Applikation laufen und checkt sie in einem produktionsähnlichen System. In vielen solchen Projekten sind die Integrationsphasen, in denen die eingecheckten Änderungen zusammengeführt und die Applikation anschließend zum Laufen gebracht wird, sehr lange und aufwendig. Noch viel schlimmer ist, dass vielfach erst in dieser Phase herausgefunden wird, dass die Software nicht den Zweck erfüllt. (Humble & Farley, 2011)

Es gibt aber auch Projekte, in welchen sich die Applikation nach letzten Änderungen nur wenige Minuten in einem nicht funktionierenden Zustand befindet. Der Grund dafür ist der Einsatz von kontinuierlicher Integration. Voraussetzung für CI ist, dass jedes Mal, wenn jemand Änderungen durchführt, die gesamte Applikation gebaut wird und eine umfassende Anzahl an automatisierten Tests erfolgreich durchlaufen wird. Wenn diese Tests fehlschlagen, sollte anschließend das Entwicklungsteam die aktuellen Tätigkeiten beenden, um das Problem zu lösen. Ziel von CI ist es schließlich, dass die Applikation zu jeder Zeit in einem funktionierenden Zustand ist. (Humble & Farley, 2011)

Ohne CI gilt die Software als nicht funktionsfähig und zwar so lange, bis das Gegenteil in der Test- oder Integrationsphase bewiesen wird. Mit CI wird die Software, unter der Voraussetzung von ausreichend automatisierten Tests, nach jeder getätigten Änderung als funktionsfähig angesehen und man weiß sofort, wann sich dieser Zustand ändert und kann entsprechend eingreifen. Teams, die sich CI zunutze machen, sind effektiver und schneller beim Liefern von Software und produzieren zudem weniger Fehler. Darüber hinaus werden die Fehler viel früher gefunden und die Kosten bzw. die aufzuwendende Zeit, um diese zu beheben, sind daher auch

wesentlich niedriger, als wenn sie erst am Ende der Deployment Pipeline gefunden werden. (Humble & Farley, 2011)

Zusammengefasst lässt sich sagen, dass es ohne CI sehr schwierig ist, kontinuierlich Software in hochwertiger Qualität und angemessener Zeit auszuliefern. Fehler erst in späteren Phasen zu beheben ist mit deutlich gestiegenen Kosten und größerem Zeitaufwand verbunden. Daher empfiehlt es sich, Mechanismen zu installieren, die sicherstellen, dass Software immer in einem nutzbaren fehlerfreien Zustand zur Verfügung steht. CI bietet solche Mechanismen und es ist ratsam, bestehende Projekte in ein CI-System zu integrieren.

Nachdem in diesem Kapitel beschrieben wurde, was unter dem Begriff CI zu verstehen ist und welche Vorteile mit deren Einsatz verbunden sind, wird im nächsten Kapitel gezeigt, welche Voraussetzungen dafür erfüllt werden müssen.

3.3.1 Voraussetzungen für kontinuierliche Integration

Wie in dem Kapitel zuvor beschrieben, ist CI entscheidend, wenn man regelmäßig Software in angemessener Zeit und Qualität liefern will. CI ist aber nicht etwa eine bestimmte Software, sondern eine essentielle Praxis für die gesamte Applikation (ausführbarer Code, Konfiguration, Serverumgebung und Daten), welche erfordert, dass die Teams gewissen Vorgehensweisen einhalten. Damit mit der Einführung von CI begonnen werden kann, müssen drei Voraussetzungen erfüllt sein: (Humble & Farley, 2011)

1. **Versionskontrolle:** Alles, was das Projekt betrifft, muss in ein einziges Repository zur Versionskontrolle eingecheckt werden. Dies beinhaltet zum einen sämtlichen Code, alle Tests, sowie alle Skripte für Datenbanken, zum Bauen und Deployen der Applikation und zum anderen alles Notwendige, um die Applikation erstellen, installieren, testen und am Ende laufen lassen zu können. Es mag einleuchtend klingen, in dieser Form Software zu entwickeln. In der Praxis kommt es jedoch sehr häufig vor, dass überhaupt keine Form von Versionskontrolle genutzt wird, wobei dahin gehend argumentiert wird, dass das vorliegende Projekt hierfür nicht groß genug sei. Es ist aber bei keinem noch so kleinen Projekt sinnvoll, ohne Versionskontrolle zu arbeiten. Denn selbst im Fall, dass man nur Code für sich selbst, die eigenen Bedürfnisse bzw. den eigenen Rechner schreibt, sollte eine Versionskontrolle eingesetzt werden.
2. **Automatisierter Build:** Man muss in der Lage sein, den Buildprozess mittels einer Kommandozeile starten zu können. Durchgeführt werden kann dies entweder mit einem Kommandozeilenprogramm, das der integrierten Entwicklungsumgebung (IDE)[26] die Anweisungen zum Bau der Software und Ausführen der Tests gibt, oder mit einer Sammlung mehrstufiger Build-Skripts, die sich nacheinander aufrufen. Wie auch immer

[26] Integrated Development Environment: bezeichnet eine Sammlung von Anwendungsprogrammen, um Software entwickeln zu können. Hier sind unter anderem die Editoren, Debugger (zur Fehlersuche), aber auch Komponenten zur Versionsverwaltung usw. enthalten. (Engels & Schäfer, 1989)

der Mechanismus genau ist, es muss auf jeden Fall für eine Person oder einen Computer möglich sein, den Prozess via Kommandozeile auszuführen. Mögliche Tools für IDEs oder CI werden immer komplexer und man kann die Applikation auch bauen bzw. testen, ohne auch nur in die Nähe einer Kommandozeile zu kommen. Dennoch ist es wichtig, die Möglichkeit zu haben, via Build-Skript und Kommandozeile die Aufgaben durchzuführen. Gründe dafür sind:

- Damit geprüft werden kann, wenn etwas nicht korrekt funktioniert, muss man in der Lage sein, den Build-Prozess automatisiert aus der CI-Umgebung starten zu können.
- Die Build-Skripte sollten wie der eigentliche Code behandelt und ständig überprüft bzw. angepasst werden, so dass sie ordnungsgemäß und einfach zu verstehen sind.
- Verständnis, Wartung und Fehlersuche des Build werden vereinfacht und die Zusammenarbeit mit anderen Teams wie Operations verbessert.

3. **Zustimmung der Teams:** CI ist eine Praxis und kein Werkzeug. Daher müssen sich die Teams dafür „verpflichten“ und anschließend entsprechende Disziplin an den Tag legen. Wirklich jeder muss regelmäßig kleine, inkrementelle Änderungen einchecken und der Hauptfokus muss darauf liegen, immer eine funktionierende Applikation zu haben. Jede Änderung, die einen Fehler hervorruft, muss daher umgehend korrigiert werden. Wenn sich die Leute nicht an diese Vorgaben halten bzw. die notwendige Disziplin an den Tag legen, kann mit CI eine Verbesserung der Qualität und der Durchlaufzeiten erreicht werden.

Durch den Einsatz von CI alleine ist es aber nicht möglich den Build-Prozess zu reparieren. Es kann daher sehr schmerzhaft sein, wenn man mit der Einführung in der Mitte eines Projektes startet. Bevor CI effektiv genutzt werden kann, sind noch folgende vier Praktiken zu berücksichtigen: (Humble & Farley, 2011)

1. Regelmäßiges Einchecken: Die wichtigste Praktik von CI ist das regelmäßige Einchecken von Code. Dies sollte zumindest ein paar Mal am Tag erfolgen. Das regelmäßige und oftmalige Einchecken von Code bringt noch weitere Vorteile. Die Änderungen werden kleiner und dadurch werden Builds weniger zerstört. Zudem hat man im Falle eines Fehlers eine kürzlich als gut befundene Version der Applikation, zu der zurückgekehrt werden kann. Des Weiteren kann dadurch verhindert werden, dass Änderungen an vielen Dateien mit anderen kollidieren.
2. Erstellung einer umfassenden, automatisierten Testsuite: Wenn keine entsprechend umfassende, automatisierte Testsuite vorhanden ist, bedeutet das, dass die Applikation nur möglicherweise funktioniert. Automatisierte Tests stärken das Vertrauen des Teams, dass die Applikation ordnungsgemäß funktioniert. Es gibt drei Arten von Tests, die automatisiert ausgeführt werden sollten:
    - Unit Tests: Werden geschrieben, um einen kleinen Teil unabhängig von der restlichen Applikation zu testen. Sie können gestartet werden, ohne dass die

gesamte Applikation läuft und betreffen nicht die Datenbank, das Dateisystem oder das Netzwerk.

- Component Tests: Mit diesen Tests wird das Verhalten sämtlicher Komponenten der Applikation getestet. Wie bei Unit Test, ist es nicht immer zwingend notwendig, die gesamte Applikation zu starten. Komponententests können Datenbank, Dateisystem oder Netzwerk betreffen.
- Acceptance Tests: Dienen zum Testen der vorgegebenen Akzeptanzkriterien. Diese beinhalten sowohl die eigentlichen Funktionalitäten als auch die Charakteristika der gesamten Applikation (Verfügbarkeit, Sicherheit, usw.). Für diese Tests muss die gesamte Applikation in einem produktionsähnlichen System laufen.

3. Build- und Testprozess kurzhalten: Wenn es zu lange dauert, den Code zu bauen und um die Tests auszuführen, treten folgende Probleme auf:
   - Die Leute werden vor dem Einchecken die Applikation nicht mehr komplett bauen und keine Tests ausführen. Die Folgen sind vermehrte fehlerhafte Builds.
   - Der Prozess der CI wird so lange dauern, dass bis zur Möglichkeit der Ausführung eines erneuten Builds sehr viele unterschiedliche Änderungen vorhanden sein werden. Anschließend wird es schwieriger herauszufinden, welche Änderung den Fehler verursacht hat.
   - Die Leute werden auf Grund langer Wartezeiten weniger oft Änderungen einchecken.

   Im Idealfall dauern die Kompilier- und Testprozesse vor dem eigentlichen Einchecken des Codes nur wenige Minuten. Sollte dies nicht der Fall sein, gilt es herauszufinden, welche Tests nicht optimal laufen. Diese sollten dann möglichst optimiert werden bzw. sollte eruiert werden, ob die selbe Deckung an Tests mit weniger Ausführungen erreicht werden kann. Diese Optimierungen sollten in regelmäßigen Abständen durchgeführt werden.
4. Managen der Entwicklungsumgebung: Es ist wichtig, dass die Entwicklungsumgebung sorgfältig gemanagt wird. Entwickler müssen in einer von ihnen kontrollierten Umgebung in der Lage sein, die Applikation zu bauen, die automatisierten Tests auszuführen und die Applikation zu deployen. Im Normalfall sollte das die lokale Entwicklungsumgebung auf dem eigenen Gerät sein. In dieser lokalen Umgebung sollten dieselben Vorbedingungen (automatisierte Prozesse, usw.) wie in der CI-, Test- und Produktivumgebung gelten. Hierfür ist es wichtig, alle notwendigen Informationen (Konfiguration, Testdaten, Build-Skripte, usw.) via Versionskontrolle zur Verfügung zu stellen. Zudem muss ein Konfigurationsmanagement für zum Beispiel Drittanbieterabhängigkeiten oder andere Komponenten durchgeführt werden. Denn es ist sicherzustellen, dass immer die korrekten Versionen verwendet werden.

Wenn diese drei Voraussetzungen erfüllt worden sind und die vier genannten Praktiken bereits im Unternehmen zelebriert werden, ist es möglich, ein Basissystem für kontinuierliche Integration zu installieren.

3.3.2 Einsatz von kontinuierlicher Integration

Nachdem die zuvor beschriebenen Voraussetzungen gegeben sind, wird für den eigentlichen Einsatz von kontinuierlicher Integration grundsätzlich keine spezielle Software benötigt. Denn CI ist, wie bereits beschrieben, kein Werkzeug, sondern eine notwendige Praxis.

Es gibt aber natürlich auch Software, kostenpflichtig oder nicht, die beim Einsatz von CI hilft. Auf die Auswahl der am besten geeigneten Software kann aus Platzgründen hier nicht eingegangen werden, außerdem spielt die für CI eingesetzte Software keine entscheidende Rolle. Für welche Lösung man sich letztendlich auch immer entscheidet, wesentlich ist, dass zum einen die zuvor erwähnten Voraussetzungen erfüllt sind und dass man zum anderen nach erfolgreicher Installation der Software in der Lage ist, in wenigen Minuten die Applikation zu verwalten. Unter „verwalten“ wird verstanden, dass jedes eingesetzte Werkzeug eine Konfiguration hat – wie Zugang zum Repository, zu Skripten und automatisierten Tests – und zudem festgelegt wird, wie und wer bei einem fehlerhaften Build benachrichtigt wird. (Humble & Farley, 2011)

Bei den ersten Ausführungen mittels CI-Tool kann es mitunter vorkommen, dass Fehler aufgrund fehlender Konfiguration usw. auftreten. Das Auftreten dieser Fehler sollte als Möglichkeit gesehen werden, um eine Vorlage für die Integration weiterer Projekte zu schaffen. Daher ist es besonders wichtig, jeden einzelnen getätigten Schritt zu dokumentieren und zentral abzulegen. Nachdem schließlich das CI-Tool ordnungsgemäß funktioniert, ist der nächste Schritt, dass jeder am Projekt Beteiligte damit zu arbeiten beginnt. Ein simpler Prozess unter der Annahme, dass die aktuell letzten Änderungen eingecheckt werden sollen, könnte wie folgt aussehen: (Humble & Farley, 2011)

1. Zuerst muss überprüft werden, ob bereits ein Buildprozess im Gange ist. Wenn ja, muss abgewartet werden, bis dieser beendet ist. Ist dieser fehlerhaft, muss das gesamte Team den Fehler beheben, bevor wieder etwas eingecheckt werden darf.
2. Wenn der Build erfolgreich abgeschlossen wurde und auch die notwendigen Tests durchlaufen sind, muss geprüft werden, ob eine aktuellere Version (im Repository der Versionskontrolle) als jene der Entwicklungsumgebung vorhanden ist und hat nötigenfalls ein Update durchgeführt zu werden.
3. Ausführung aller notwendigen Build-Skripte und Test auf dem lokalen Entwicklungsgerät. Dadurch soll sichergestellt werden, dass alles auch mit den aktuellen Änderungen noch ordnungsgemäß funktioniert.
4. Wenn die Tests und der Buildprozess erfolgreich durchlaufen sind, ist der neue Code in die Versionskontrolle einzuchecken.
5. Warten während das CI-Tool die Applikation mit den neuen Änderungen baut.

6. Tritt hier ein Fehler auf, muss das Problem umgehend identifiziert und behoben werden. Anschließend wieder zurück zu Schritt 3.
7. Wenn der Build funktioniert, darf gejubelt und anschließend mit der nächsten Aufgabe begonnen werden.

Wenn sich jeder aus dem Team an diese sieben Schritte hält, weiß man, dass nach jeder Änderung an der Applikation diese auf jedem System konfiguriert wie die CI-Box läuft. (Humble & Farley, 2011)

Der Großteil des bis dato Beschriebenen beinhaltete die Automatisierung der verschiedenen Prozesse, die zur Auslieferung von Software notwendig sind. Diese Automatisierungen existieren in einer Umgebung von menschlichen Prozessen und manuelle Eingriffe müssen berücksichtig werden. CI ist kein Werkzeug, sondern eine Praxis, deren Effektivität von der Disziplin der einzelnen Beteiligten abhängt. Ziel des CI-Systems ist es, die Software über die gesamte Dauer lauffähig zu halten. Um dies bewerkstelligen zu können, sind folgende Praktiken mit Nachdruck durchzusetzen: (Humble & Farley, 2011)

- Kein Einchecken, wenn ein Build fehlerhaft ist: Die größte Sünde bei der Nutzung von CI wäre es, in einen fehlerhaften Build neuen Code einzuchecken. Wenn ein Build einen Fehler wirft, sind die verantwortlichen Entwickler gefragt, um den Fehler so schnell wie möglich zu beheben. Weitere Codeänderungen erschweren das Finden des Fehlers und der Aufwand wird immer größer werden.
- Alle Commit Tests müssen lokal ausgeführt werden, bevor ein Commit durchgeführt wird: Wie bereits erwähnt, sollte ein Commit die Erstellung eines neuen Releasekandidaten anstoßen. Daher soll die Ausführung aller Commit Tests eine Art Sicherheitscheck darstellen und zudem dafür sorgen, dass all das, von dem wir annehmen, dass es ordnungsgemäß funktioniert, auch tatsächlich getan wird. Zudem muss darauf geachtet werden, dass die zuvor beschriebenen sieben Schritte ordentlich ausgeführt werden.
- Warten bis die Commit Tests abgeschlossen sind: Das CI-System stellt eine geteilte Ressource für das gesamte Team dar. Wenn das gesamte Team CI effektiv nutzt und regelmäßig Änderungen eincheckt, ist jeder daraus entstehende Fehler für das Team und das Gesamtprojekt bloß ein kleiner Fehltritt. Fehler beim Build sind normal und werden als erwartbar angesehen. Entscheidend ist, die Fehler anschließend schnell zu finden und zu beseitigen. Entwickler müssen daher warten, bis der Build tatsächlich abgeschlossen ist, bevor sie mit einer neuen Aufgabe beginnen oder beispielsweise an einem Meeting teilnehmen. Erst wenn der Build erfolgreich war, dürfen sich Entwickler neuen Aufgaben widmen oder zum Beispiel Mittagspause machen.
- Niemals nach Hause gehen, wenn ein Build einen Fehler hat: Es ist wichtig, dass niemals nach Hause gegangen wird, bevor der Buildprozess erfolgreich abgeschlossen wurde. Entweder man versucht, den Fehler doch noch zu beheben, oder man setzt seine Änderungen zurück. Da es nicht der Normalfall sein sollte, dass ein Mitarbeiter Überstunden machen muss, sollte folgende Regel befolgt werden: Änderungen werden spätestens eine Stunde vor Ende der Arbeitszeit eingecheckt. Dies sollte genug Zeit

bieten, um mögliche Fehler zu beheben. Im schlimmsten Fall empfiehlt es sich, um am nächsten Tag mit der Behebung des Fehlers weitermachen zu können, die Änderungen rückgängig zu machen und sich diese lokal zur Verfügung zu halten.

- Immer darauf vorbereitet sein, eine vorherige Version wiederherzustellen: Wie bereits beschrieben ist es normal, dass Fehler passieren und so kann auch erwartet werden, dass jeder einmal einen fehlerhaften Build produziert. Im Normalfall sollte der Fehler schnell gefunden und behoben werden. Es kann jedoch vorkommen, dass es nicht in angemessener Zeit möglich ist, den Fehler zu beheben. Da die Applikation immer in einem lauffähigen Zustand sein sollte, kann es daher notwendig sein, zu einer Vorgängerversion zurückzukehren. Anschließend sollte der Fehler auf einem lokalen Gerät behoben werden. Dies ist einer der Gründe, warum eine Versionskontrolle im Softwarebereich so wichtig ist.
- Festlegen einer Timebox: Das Team braucht eine Regel, die vorgibt, wie viel Zeit für das Beheben eines Fehlers aufgewendet werden sollte, bevor zu einer vorherigen Version zurückgekehrt wird. Beispielweise sollten zehn bis maximal fünfzehn Minuten für die Fehlerbehebung aufgewendet werden, wenn ein Build gebrochen ist. Sollte es nicht möglich sein, den Fehler innerhalb dieser Zeitspanne zu beheben, wird zur letzten funktionierenden Version zurückgekehrt.
- Kein Auskommentieren von fehlerhaften Tests: Wenn man die zuvor beschriebene Regel mit Nachdruck ausführen lässt, geschieht es sehr häufig, dass Entwickler kurzerhand Tests auskommentieren, um ihre Änderungen einchecken zu können. Sollten dann Test über einen längeren Zeitraum fehlschlagen, ist es oftmals sehr schwierig, den Grund dafür herauszufinden. Daher empfiehlt es sich, die Tests regelmäßig auf Relevanz zu überprüfen und sie im Fall der Fälle zu entfernen.
- Verantwortung für die eigenen Änderungen übernehmen: Wenn Änderungen durchgeführt sowie alle selbst geschriebenen Tests erfolgreich ausgeführt werden, andere aber werfen einen Fehler, liegt es dennoch in der eigenen Verantwortung, diese Fehler zu behandeln. Im Normalfall bedeutet dies, dass die eigenen Änderungen eine Auswirkung auf andere Teile des Codes gehabt haben und man zudem Zugriff auf den gesamten Code benötigt. Um CI effektiv betreiben zu können, ist es daher essentiell, dass jeder Zugriff auf die gesamte Codebasis hat.
- Test-Driven Development (TDD): Wie bereits beschreiben, ist eine umfassende, automatisierte Testsuite für CI essentiell. Schnelles Feedback ist nur möglich, wenn eine ausreichende Abdeckung durch Tests erreicht wird. Die Erfahrung zeigt, dass dies nur dann möglich ist, wenn Code testgesteuert entwickelt wird. Zudem ist diese Praxis für die kontinuierliche Auslieferung von Software unabdingbar. Hinter Test-Driven Development steckt die Idee, dass die Tests zur Verifizierung des gewünschten Verhaltens erstellt werden, bevor neuer Code entwickelt oder ein Fehler behoben wird. Diese Tests können sowohl als Regression Tests als auch als Dokumentation dienen.

Werden die in Kapitel 3.3.1 ausgeführten Voraussetzungen für kontinuierliche Integration erfüllt und zudem der in diesem Kapitel beschriebene Prozess zur Durchführung von Änderungen bzw. die notwendigen Praktiken mit Nachdruck eingesetzt, sollte es problemlos möglich sein, CI effektiv und effizient für das eigene Projekt nutzen zu können.

Zusammengefasst lässt sich daher sagen, dass CI mehr als nur ein Werkzeug ist. Es ist beinahe schon einer Philosophie gleichzusetzen, denn es verlangt zusätzlich zu den Prämissen für Build-, Deploy-, Test- und Release Prozesse, dass jeder im Team diese Praxis aktiv unterstützt und sie „lebt". Es setzt also – wie jeder erfolgreiche Prozess – eine hohe Teamdisziplin voraus. Und im gegenständlichen Fall ist sofort zu erkennen, wenn es an Disziplin fehlt. Bleiben die Builds erfolgreich, kann im Normalfall davon ausgegangen werden, dass diese auch ordnungsgemäß funktionieren. Natürlich kann es auch sein, dass zu wenig Fokus auf die automatisierten Tests gelegt wird und die Builds daher immer erfolgreich sind. Hier bieten viele CI-Systeme aber bereits einfache Möglichkeiten zur Nachbesserung. (Humble & Farley, 2011)

Solange man kein CI für die eigenen Projekte einsetzt, muss davon ausgegangen werden, dass die Applikation fehlerhaft ist, bis das Gegenteil bewiesen wird. Mit CI hat man permanent eine lauffähige Applikation und ist gezwungen, ein ordentliches Management der Konfiguration zu betreiben und automatisierte Prozesse aufzubauen und zu warten. Die Grundidee von CI lässt sich aber nicht ausschließlich auf die Entwicklung von Software selbst anwenden. Ein bestehendes CI-System kann vielmehr auch als Vorlage für weitere Infrastruktur und Prozesse herangezogen werden.

Es gibt zudem noch wesentlich mehr empfohlene Praktiken zur Softwareentwicklung, beispielsweise für verteilte Teams, die bei kontinuierlicher Integration Einsatz finden. Aufgrund der großen Bandbreite dieser Thematik können diese hier aber nicht näher ausgeführt werden. Auch Maßnahmen zur Einführung von Testing-Strategien können nicht beschrieben werden.

Für das angestrebte Vorgehensmodell zur Übergabe und weiteren Betreuung von agil umgesetzten Projekten reicht das in diesem Kapitel geschaffene Verständnis. Bisher wurde anhand von CI gezeigt, wie ein Großteil der Schritte der in Abbildung 12 gezeigten Deployment Pipeline automatisiert werden können. Es ist logisch, dass es nicht möglich sein wird, die gesamten Tätigkeiten innerhalb des Prozesses, wie zum Beispiel manuelles Testen, zu automatisieren. Nicht automatisierte Tätigkeiten werden immer bei der Auslieferung von Software durchgeführt werden müssen. Die Frage, die sich an diesem Punkt jedoch stellt, ist: Wenn ohnehin der Großteil der Schritte anhand der Deployment Pipeline automatisiert sind, warum dann nicht das bestehende CI-System erweitern, um auch die Anwendung vollautomatisch in Produktion zu deployen? Um diese Frage beantworten zu können, werden im nächsten Kapitel Deployments und Releases näher beleuchtet.

3.4 Deployments und Releases von Applikationen

Es gibt Unterschiede zwischen der Veröffentlichung bzw. dem Deployen von Software in eine Produktions- respektive in eine Testumgebung, zumindest was die damit jeweils einhergehende Anspannung und Stressentwicklung anlangt. Technisch gesehen sollte es nur wenig Unterschied

machen, auf welches System die Software deployed wird. Es sollte also für jedes Deployment derselbe Prozess angewendet werden können und die Unterschiede sollten durch Konfigurationsdateien verschachtelt werden. Im Prinzip sollte es also ausreichen, die Version und das System zu wählen, um anschließend mit einem einzigen Knopfdruck die gewünschte Version auf das gewählte System deployen zu können. Der gleiche Prozess sollte dann für alle nachfolgenden Deployments und Releases angewendet werden. Der Unterschied zwischen einem Release und einem Deployment liegt in der Eignung bzw. Möglichkeit, ein Rollback auszuführen. (Humble & Farley, 2011)

Alle diese Prozesse, wie ein Deployment in Test- bzw. Produktionsumgebung oder auch Rollbacks, müssen in der Deployment Pipeline berücksichtigt werden (vgl. Abbildung 12). Eine Liste der verfügbaren Builds für ein Deployment und die entsprechenden Umgebungen (Produktion, Test, usw.) sollte sichtbar sein, um den automatisierten Deploymentprozess mittels Knopfdruck starten zu können. Dies sollte der einzige Weg sein, wie Änderungen an einem System durchgeführt werden können. Eingeschlossen sind sämtliche Änderungen an der Konfiguration und der Drittanbietersoftware. Durch Festhalten an diese Vorgehensweise sollte es möglich sein zu sehen, welche Version auf welchem System derzeit läuft, wer das Deployment genehmigt hat und welche Änderungen seit dem letzten Deployment durchgeführt wurden. (Humble & Farley, 2011)

Da, wie bereits beschrieben, derselbe Prozess für Deployments und Releases verwendet werden soll, ist es wichtig, einen klaren Release-Plan zu haben und diesem zu folgen. Im nächsten Kapitel wird daher in Ansätzen erläutert, was für eine praktikable Release-Strategie notwendig ist.

3.4.1 Release-Strategien und Release-Plan

Das Wichtigste beim Erstellen einer Release-Strategie ist es, alle an der Applikation beteiligten Stakeholder für den Prozess der Projektplanung mit ins Boot zu holen. Der Hauptgrund für das Zusammentreffen aller Stakeholder sollte die Schaffung eines gemeinsamen Verständnisses für Deployments und Wartung der Applikation während der gesamten Phasen des Lebenszyklus sein (vgl. Abbildung 11). Dieses gemeinsame Verständnis sollte in eine Release-Strategie münden. Das erstellte Dokument ist von den Stakeholdern laufend zu aktualisieren und zu warten. Bei Erstellung der Erstversion der Release-Strategie für das Projekt sind folgende Punkte zu beachten: (Humble & Farley, 2011)

- Erstellung einer Vermögensstrategie sowie einer Strategie für das Konfigurationsmanagement.
- Definition der für die Deployments und Releases für jede Umgebung Verantwortlichen.
- Erstellung einer Beschreibung der Technologien, die für die Deployments eingesetzt werden sollen. Diesen Technologien müssen sowohl das Operations- als auch das Entwicklungsteam zustimmen.
- Aufstellung eines Plans zur Implementierung der Deployment Pipeline.

- Aufzählung der für Akzeptanz-, Kapazitäten-, Integrations- und Benutzerakzeptanztests zu Verfügung stehenden Umgebungen sowie Aufzeichnung des Buildprozesses.
- Beschreibung der festgelegten Prozesse für Deployment in Test- bzw. Produktionsumgebung. Zusätzlich muss beschrieben werden, welche Voraussetzungen für Change Requests (CR)[27] gelten und welche Bestätigungen diese vor ihrer Umsetzung brauchen. Des Weiteren muss festgelegt werden, wie mit Fehlern in der Produktivumgebung umgegangen wird.
- Aufzeichnung aller Anforderungen, inklusive notwendiger Programmierschnittstellen[28] (API), die notwendig sind, um die Applikation zu überwachen bzw. um das Operations Team zu benachrichtigen.
- Beschreibung der Integration mit externen Systemen. Wie und wann werden diese als Bestandteil des gesamten Releases getestet? Wie wird mit dem Provider im Fehlerfall kommuniziert?
- Festlegung, wie die Aufzeichnungen (Logging) vonstatten gehen sollen. Das Operations Team muss in der Lage sein, jederzeit den Status der Applikation herauszufinden.
- Erstellung eines Wiederherstellungsplans, sodass die Applikation im Fehlerfall wiederhergestellt werden kann.
- Definition einer SLA-Strategie: Wann muss wie auf welches Ereignis reagiert werden?
- Dimensionierung der Systeme: Welche Leistung oder wieviel Speicherplatz benötigen sie?
- Festlegung dahingehend, wie das initiale Deployment vonstatten gehen bzw. funktionieren soll.
- Definition der Strategie zur Archivierung nicht mehr benötigter Daten. Diese sollten weiterhin zum Zwecke von Support und Audits zur Verfügung stehen.
- Festlegung dahingehend, wie Upgrades des Produktivsystems inklusive Datenmigration durchgeführt werden sollen.
- Aufstellung darüber, wie der Support der Applikation gehandhabt werden soll.

Der Akt der Erstellung der Release-Strategie ist sehr wertvoll, stellt er im Normalfall doch die Ressource für funktionale wie nicht funktionale Anforderungen an sowohl die Softwareentwicklung als auch an Design, Konfiguration und Hardwareanforderungen dar. Die Erstellung der Release-Strategie ist natürlich nur der Anfang. Diese wird sich im Laufe der Zeit

[27] Als Change Requests (CR) werden neue Anforderungen verstanden, mithilfe derer die Funktionalität der Applikation erweitert werden soll. (Schmidt & Dohle, 2009)

[28] Eine Programmierschnittstelle, meistens nur API genannt, ist eine Schnittstelle, die von der Software zur Anbindung anderer Software zur Verfügung gestellt wird. (Gamma, Helm, Johnson, Vlissides, & Booch, 1994)

immer wieder verändern und angepasst werden. Ein wichtiges Element der Release-Strategie ist der Release-Plan. (Humble & Farley, 2011)

Im Normalfall ist das erste Release immer das schwierigste und jenes mit dem größten Risiko. Daher ist es wichtig, dieses besonders gut zu planen. Das Ergebnis dieser Planung könnten automatisierte Skripts, Dokumentationen oder andere Artefakte für wiederholbare Deployments in die Produktivumgebung sein. Zusätzlich zu den erwähnten Punkten der Release-Strategie sollte der Release-Plan Folgendes enthalten: (Humble & Farley, 2011)

- Alle notwendigen Schritte, um die Applikation das erste Mal deployen zu können.
- Eine Strategie, um die Applikation und alle Services via Smoke Tests[29] prüfen zu können.
- Ein Plan zur Wiederherstellung der Applikation oder zum Wechsel zu einer Vorgängerversion im Fehlerfall. Zudem alle nötigen Schritte, um ein fehlerhaftes Deployment erneut starten zu können.
- Alle Schritte, die notwendig sind, um die Applikation upgraden zu können, ohne diese zu zerstören.
- Eine Beschreibung, in welcher Form und wo die Logs abgelegt werden bzw. welche Informationen diese enthalten.
- Alle notwendigen Schritte, um Daten im Zuge eines Releases migrieren zu können.
- Eine Beschreibung des Monitorings der Applikation.
- Eine Aufzeichnung der bei vorherigen Deployments aufgetretenen Probleme.

Es gibt noch andere Informationen, die in einem Release-Plan gepflegt werden können. In der Praxis kann ein Release-Plan zum Beispiel auch eine grobe Übersicht der geplanten Deployments über das gesamte Jahr hinweg enthalten. Der Release-Plan selbst sollte gleich der Release-Strategie ständig aktuell gehalten werden. Nachdem nun eine Release-Strategie sowie ein erster Release-Plan vorhanden sind, wird im nächsten Kapitel kursorisch auf das Thema Deployment eingegangen, bevor anschließend der Ansatz des kontinuierlichen Deployment beschrieben wird.

3.4.2 Deployment einer Applikation

Die Schlüssel zu verlässlichen und einheitlichen Deployments sind zum einen die im vorherigen Kapitel erläuterten Release-Strategie bzw. der Release-Plan und zum anderen die Nutzung eines einheitlichen Prozesses für alle Umgebungen. Die Automatisierung von Deployments sollte daher mit dem ersten Deployment in eine Testumgebung starten. Anstatt die Applikation händisch auf

[29] Smoke Tests sind eine Sammlung von System- und Businesskritischen Testfällen. Mit diesen sollen die Hauptfunktionalitäten der Applikation abgedeckt werden. (Dustin, Rashka, & Paul, 1999)

die vorgesehenen Server zu kopieren, sollte ein Skript geschrieben werden, das die notwendigen Aufgaben ausführt. (Humble & Farley, 2011)

Das erste Deployment sollte nach der ersten Iteration, wenn erste demonstrierbare Teile, der ersten User Stories bzw. Kundenanforderungen dargestellt werden können, stattfinden. Hieraus sollten die höchst priorisierten Teile herausgesucht werden, die anschließend auch einfach lieferbar sind. Diese sollen dann als Vorlage für das Deployment in eine Produktionsumgebung und dazu dienen, die ersten Schritte der Deployment Pipeline zum Laufen zu bringen. Daraus resultiert eine der wenigen Situationen, in welcher eine technische Priorisierung jener nach Geschäftswert vorzuziehen ist. Diese Strategie kann als Ankurbelung des Entwicklungsprozesses gesehen werden. Am Ende dieser ersten Iteration sollten die folgenden Punkte verfügbar sein: (Humble & Farley, 2011)

- Eine funktionierende Commit Stage-Phase der Deployment Pipeline.
- Eine produktionsähnliche Umgebung, auf die deployed werden kann.
- Ein automatisierter Prozess, welcher die notwendigen Daten auf die Systeme deployen kann.
- Ein einfacher Smoke Test, der die Funktionalität des getätigten Deployment sowie der Applikation verifizieren kann.

Das erste Deployment sollte nicht allzu schwierig sein, denn schließlich wird an der Applikation erst wenige Tage aktiv gearbeitet. Es wurde wiederholt von produktionsähnlichen Umgebungen gesprochen, aber noch nicht geklärt, was darunter zu verstehen ist. Sehr viele unterschiedliche Aspekte wie Betriebssystem, Cluster, usw. sind zu berücksichtigen. Eine detaillierte Aufstellung darüber würde aber den Rahmen sprengen. Egal wie die Systeme letzten Endes konfiguriert sind, sollte Folgendes berücksichtigt werden: (Humble & Farley, 2011)

- Das Betriebssystem sollte auf allen Systemen dasselbe sein.
- Auf dem Produktionssystem sollte die selbe Software in den selben Versionen wie in den anderen Umgebungen installiert sein. Es sollten aber keine Werkzeuge der eigentlichen Entwicklung (Compiler, usw.) installiert werden.
- Die Umgebung sollte, soweit möglich und sinnvoll, auf die gleiche Weise gemanagt werden wie die übrigen Systeme.

Nachdem die Applikation stetig wachsen und immer komplexer werden wird, wird sich auch die Implementierung der Deployment Pipeline (vgl. Abbildung 12) ändern. Unter anderem muss festgelegt werden, welche Phasen eine Release durchlaufen muss oder was notwendig ist, damit dieses genehmigt werden kann.

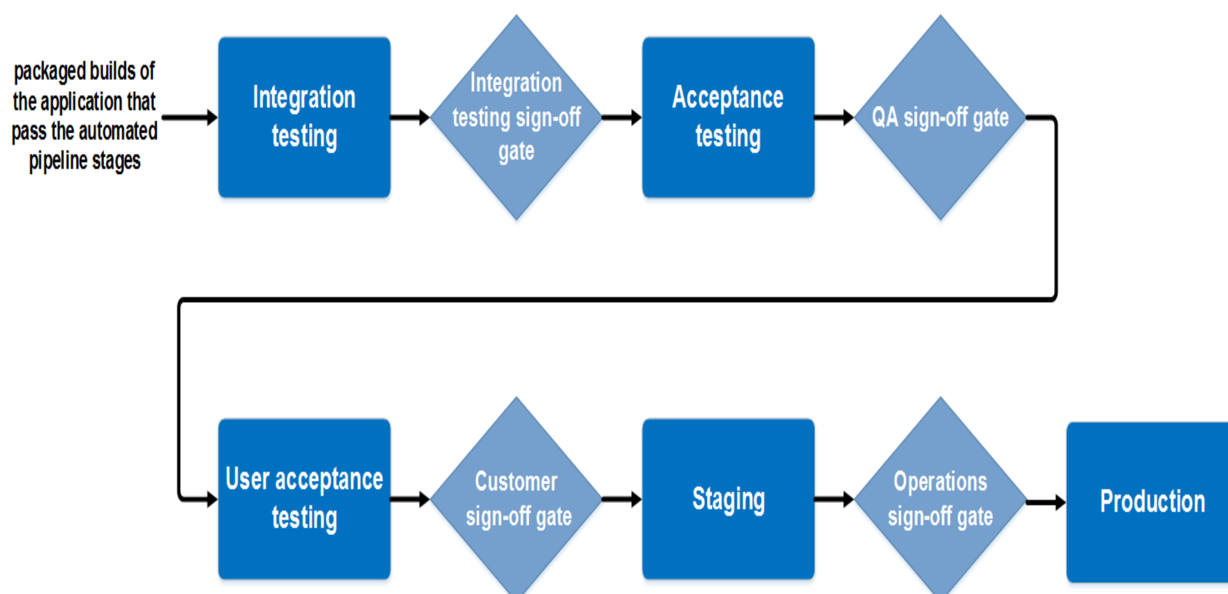


Abbildung 14 – Erweiterte Deployment Pipeline (Humble & Farley, 2011)

Am Ende könnte eine erweiterte Implementierung der Deployment Pipeline wie in Abbildung 14 aussehen. Wie auch immer die Deployment Pipeline am Ende aussieht, sie muss die in Kapitel 3.2 Das Problem der regelmäßigen Auslieferung von Softwarebeschriebenen Aufgaben erfüllen.

Es gibt unterschiedliche Ansätze dafür, wie die Deployments am Ende durchgeführt werden könnten. Beispielsweise wäre es möglich Deployments in Form von Zero-Downtime Deployments, also Deployments ohne Ausfallzeit, oder auch Canary Releasing, bei welchem am Anfang nur ein kleiner Anteil an Usern die neue Version zur Verfügung gestellt bekommen, durchzuführen. Es gibt noch viele weitere solcher Methoden für Deployments. Auf Grund des Umfanges und der Zweckmäßigkeit für das angestrebte Vorgehensmodell können diese nicht im Detail dargestellt werden. Stattdessen wird im nächsten Kapitel der nächste Schritt von kontinuierlicher Auslieferung von Software, das kontinuierliche Deployment kurz vorgestellt.

3.5 Kontinuierliches Deployment von Software

Nachdem nun schon quasi die gesamten Schritte der Deployment Pipeline weitestgehend automatisiert sind, könnte man eigentlich anstreben, auch die Anwendung vollautomatisiert zu deployen.

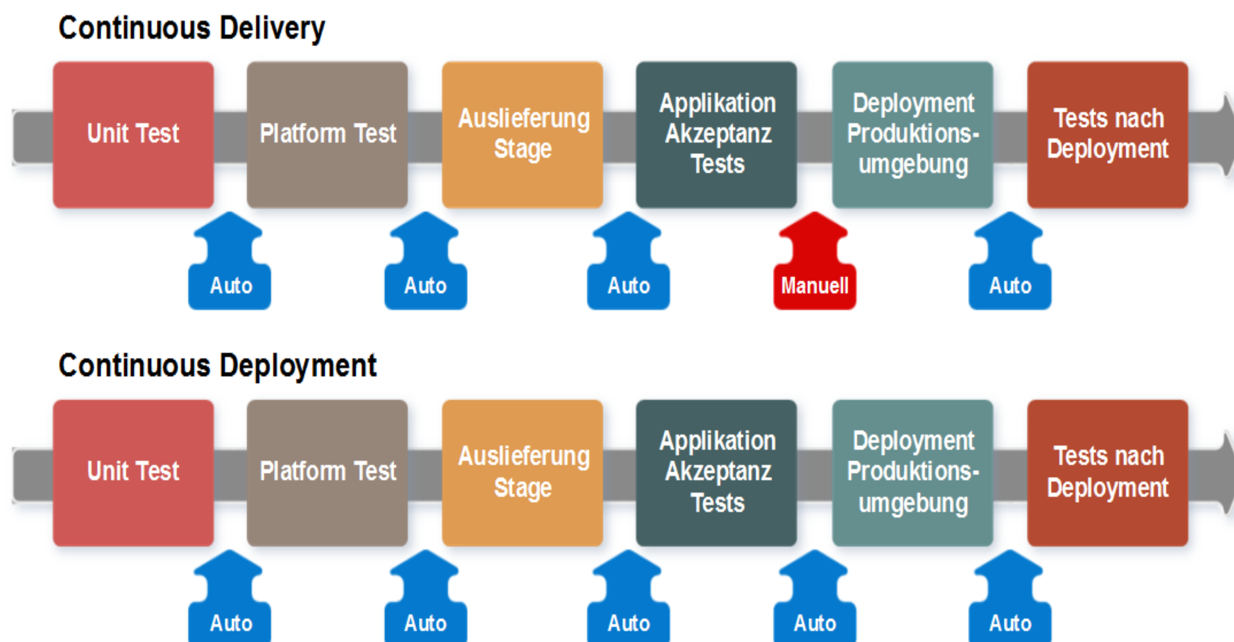


Abbildung 15 – Continuous Delivery vs. Continuous Deployment (Caum, 2016)

In Abbildung 15 soll der aktuelle Stand noch mal verdeutlicht werden. Wenn von automatisiertem Deployment die Rede ist, meint man kontinuierliche Deployments (Continuous Deployment). Der kritische Punkt ist ein automatisiertes Deployment in die Produktionsumgebung. (Humble & Farley, 2011)

Im Prinzip muss dabei bloß auch noch der letzte Schritt der Deployment Pipeline (vgl. Abbildung 14), das Deployment in die Produktionsumgebung, automatisiert vollzogen werden. Das bedeutet, dass nach erfolgreichem Durchlauf aller notwendigen Tests die Änderung direkt auf das Produktionssystem deployed wird und somit den Benutzern sofort uneingeschränkt zur Verfügung steht. Damit dies funktioniert ohne das System dadurch zu schädigen, müssen die automatisierten Tests fantastisch sein. Die Unit-, Komponenten- und Akzeptanz-Tests (funktional und nicht funktional) müssen in der Lage sein, die gesamte Applikation abzudecken. Hier empfiehlt es sich, die in Kapitel 3.2 vorgestellte Praxis des Test-Driven Development anzuwenden und alle notwendigen Tests schon vor der Entwicklung des Codes zu schreiben. Wenn eine User Story abgeschlossen ist, muss sie alle diese Tests bestehen. (Humble & Farley, 2011)

Der Ansatz, dem hier nachgegangen wird, scheint auf den ersten Blick sehr radikal und realistischerweise kaum umsetzbar zu sein und weicht extrem stark von allem klassischen Vorgehen ab. Damit Kunden möglichst nichts von den Deployments oder den daraus resultierenden Problemen merken und um die Risiken zu minimieren, kann Continuous Deployment auch mit Techniken wie Canary Releasing kombiniert werden, um damit die Änderungen am Anfang nur einer kleinen Anzahl von Usern zur Verfügung zu stellen. Durch den kleinen Anteil an manuellen Tests und die letzten Endes manuelle Freigabe der Änderung lässt sich das Risiko zudem nochmal reduzieren. Manuelle Schritte sind aber nicht zwingend erforderlich und können durchaus auch wegfallen. Darüber hinaus können organisatorische Maßnahmen die Risiken noch weiter reduzieren. Eine Vorgabe kann zum Beispiel lauten, dass Deployments nur in Bürozeiten durchgeführt werden dürfen, um so im Fehlerfall durch Monitoring

und Feedback an die Entwicklung schnellstmöglich eingreifen zu können. Den größten Anteil an der Risikominimierung hat jedoch die Tatsache, dass beim permanenten Deployment von Änderungen in die Produktivumgebung diese Änderungen jeweils meist sehr geringgradig sind und deren Fehlerpotential dementsprechend niedrig ist. Kontinuierliche Deployments kleiner Änderungen lassen sich also nicht mit klassischen Releases vergleichen und bieten einen guten Weg, die Risiken zu minimieren. (Wolff, 2016)

Wenn man das Potential von Continuous Deployment tatsächlich ausschöpfen möchte, ist man gezwungen, die richtigen Dinge zu tun. Ohne weitestgehende Automatisierung der Build-, Deploy-, Test- und Release-Prozesse, umfassende und zuverlässige automatisierte Tests und ohne System-Tests für die Produktionsumgebung kann diese Methode nicht erfolgreich eingesetzt werden. (Humble & Farley, 2011) Ein verantwortungsvoller und effizienter Einsatz dieser Praxis setzt eine sehr stark optimierte Deployment Pipeline (vgl. Abbildung 14) voraus. Das damit verbundene möglicherweise signifikante Investment stellt wohl den größten Nachteil von Continuous Deployment dar. (Wolff, 2016)

Der Einsatz von Continuous Deployment bringt jedoch auch klare Vorteile. Da die Änderungen sehr klein ausfallen, verringert sich auch die Fehlerbehebungsdauer deutlich. Im Problemfall muss anschließend lediglich eine neue Version der Software die Deployment Pipeline durchlaufen und geht damit automatisch auf das Produktionssystem. Der übliche Ansatz zur Behebung akuter Probleme, das Einspielen eines Hotfix, das der raschen Fehlerbehebung willen standardmäßig an den übrigen Prozessen vorbeigeht, kann hier entfallen. Ein Prozess zur Sonderbehandlung von Hotfixes fällt somit auch komplett weg. Ein weiterer Vorteil ist, dass die Verantwortung der Teams steigt. Da jede Änderung „sofort“ in der Produktion landet, sind Entwickler dazu angehalten, disziplinierter zu arbeiten. Teams müssen sich sicher sein, dass durchgeführte Änderungen keine Probleme in der Produktionsumgebung auslösen. Um dies zu gewährleisten, werden sie entlang der Deployment Pipeline entsprechende Absicherungen einbauen. Dies kann sehr positive Auswirkungen auf die Qualität der Software haben. Zudem bieten kleine Änderungen, die schnell in die Produktionsumgebung gelangen, eine erhöhte Flexibilität. Neue Features können dadurch schneller auf Kundenreaktionen und das Verhalten in der Produktion hin untersucht und abgetestet werden. (Wolff, 2016)

Continuous Deployment kann darüber hinaus aufgrund des Umstands, dass möglichst immer nur kleine Teile der Software ausgetauscht werden, sehr positive Auswirkungen auf die Architektur der Software haben. Im gleichen Atemzug bedeutet dies aber auch, dass eine nicht auf kontinuierliche Auslieferung ausgelegte Architektur ein derartiges Vorgehen praktisch unmöglich macht. (Wolff, 2016)

Zusammengefasst bieten die Ansätze der kontinuierlichen Auslieferung von Software sehr viele Vorteile, gleichzeitig bedarf es aber auch eines gewissen Aufwands, gründlicher Vorbereitung, hoher Disziplin, usw., um sie in Projekte zu integrieren. Am Ende sollte es möglich sein, jede gewünschte „fertige“, das heißt ordentlich und ausreichend getestete, qualitativ hochwertige, mit geringem Fehlerrisiko behaftete Softwareversion in wenigen Schritten auf jedes gewünschte System zur Verfügung zu stellen. Zudem sollte jeder im Team (Entwickler, Operations, Kunde, usw.) in der Lage sein, schnellstmöglich zu eruieren, welche Version auf welchem System gerade

installiert ist und welche Änderungen durchgeführt wurden. Letztendlich können alle beteiligten Stakeholder durch den Einsatz von kontinuierlicher Auslieferung und Integration sowie kontinuierlichem Deployment profitieren.

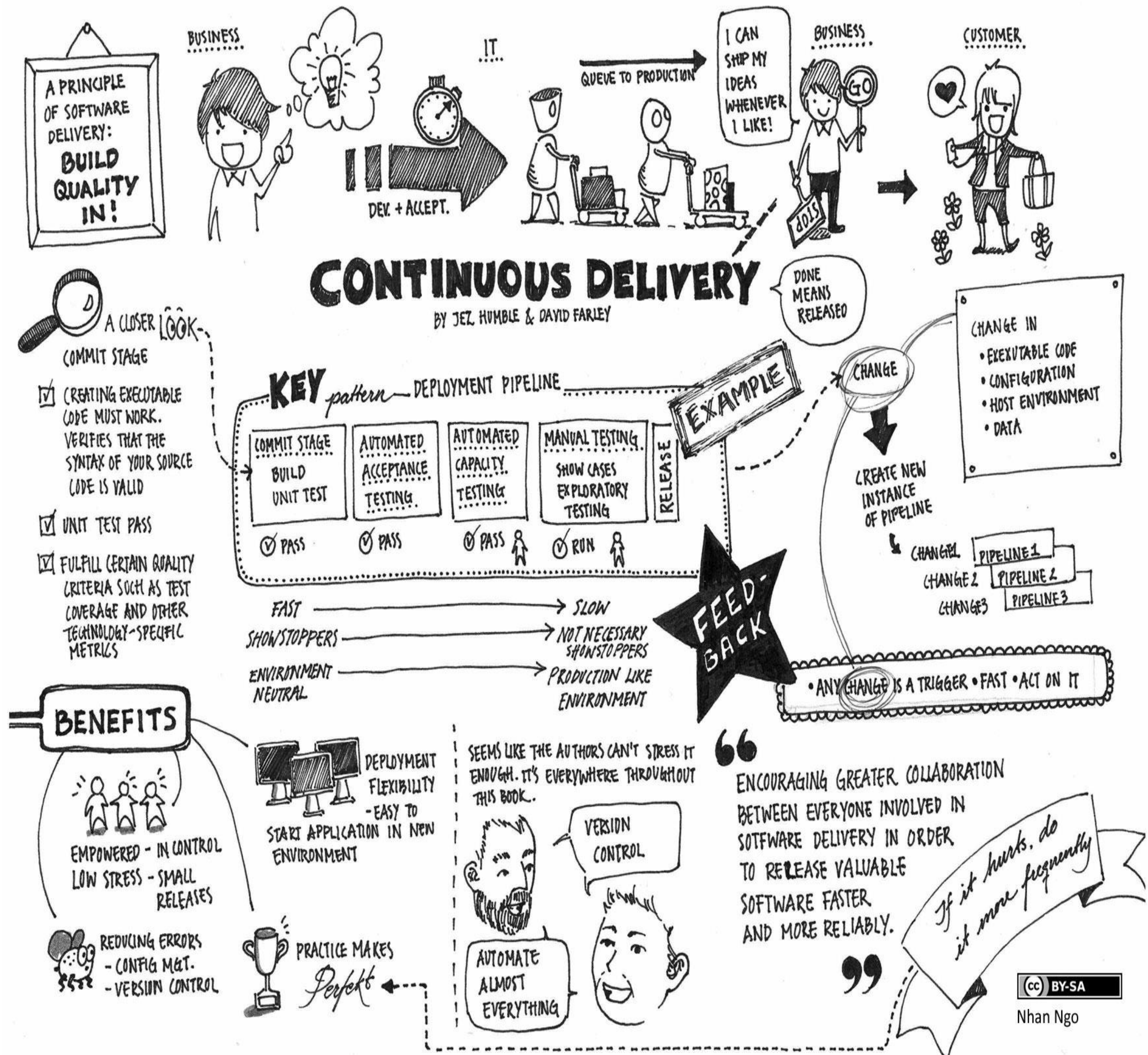


Abbildung 16 – Continuous Delivery (Humble & Farley, 2011)

Abbildung 16 gibt eine Übersicht über die in Kapitel 3 näher ausgeführten Praktiken kontinuierlicher Auslieferung von Software, die die Grundlage für die im nächsten Kapitel präsentierten und evaluierten Einflussfaktoren bei der Übergabe von agil entwickelten Projekten in andere Teams schaffen. Es ist aber nicht zwingend notwendig, dass bereits kontinuierliche Auslieferung mit allen Ausprägungen im Unternehmen betrieben wird.

4 ENTWICKLUNG DES VORGEHENSMODELLS

In den vorherigen Kapiteln wurden unterschiedliche Ansätze – klassisch oder agil – vorgestellt und der Softwarelebenszyklus sowie seine gesamtheitliche Begleitung mithilfe von kontinuierlicher Auslieferung beschrieben. Welches Modell für das eigene Projekt eingesetzt wird, hängt unter anderem vom Projektumfang, den jeweiligen Gegebenheiten, vom Team usw. ab. In diesem Kapitel sollen die Einflussfaktoren identifiziert werden, die bei der Übergabe eines agil entwickelten Projektes in ein anderes Team zu beachten sind. Der Fokus liegt hier zwar eindeutig auf agilen Modellen, die entwickelte Vorgangsweise aber ebenso gut für klassische Modelle eingesetzt werden. Alternative Einsatzmöglichkeiten des neuen Modelles werden in dieser Arbeit nicht evaluiert.

Ehe die einzelnen Faktoren entlang des Softwarelebenszyklus (vgl. Abbildung 11) identifiziert und beschrieben werden, soll in einem ersten Schritt kurz erläutert werden, wie der Aufbau des abschließenden Vorgehensmodells ausschauen soll.

4.1 Grundlagen für das neue Vorgehensmodell

Wie bereits in Abbildung 1 veranschaulicht wurde, muss ein Vorgehensmodell gewisse Anforderungen abdecken. Das in Kapitel 4.3 entwickelte neue Modell zur Übergabe agil entwickelter Projekte wird sich grob an diesen Anforderungen orientieren.

Das Vorgehensmodell selbst wird in folgende Kategorien (Tätigkeitsfelder) unterteilt:

1. Projektmanagement (PM): Beinhaltet alle Faktoren, die zum Managen bzw. Betreuen oder auch Überwachen des Projektes notwendig sind, wie zum Beispiel das gewählte Vorgehensmodell.
2. Konfigurationsmanagement (KM): Alle Faktoren, die notwendig sind, um die genutzten Systeme, Software, usw. konfigurieren zu können. Dies beinhaltet nicht die eingesetzten Programmiersprachen.
3. Qualitätsmanagement (QM): Inkludiert alle notwendigen Maßnahmen, um kontinuierlich qualitativ hochwertige Software ausliefern zu können. Zudem sollen in dieser Kategorie alle Faktoren gesammelt werden, die Einfluss auf die Prüfung der Systeme haben.
4. Systementwicklung (SE): Alle Faktoren, welche die eigentliche Entwicklung der Applikation beeinflussen oder für die Entwicklung notwendig sind.
5. Andere (A): Beinhaltet jene Faktoren, die nicht in die vorherigen Kategorien eingeordnet werden können.

Die im nächsten Kapitel evaluierten Einflussfaktoren entlang des Softwarelebenszyklus werden diesen Kategorien zugeteilt. Eine Mehrfachzuordnung ist möglich. Die Einflussfaktoren sollen zudem kursorisch beschrieben und anschließend möglichen Werkzeugkategorien zugeordnet werden. Das Vorgehensmodell wird keine Gesamtaufzeichnung aller möglichen Werkzeuge und

Methoden enthalten, da diese zum Teil sehr stark vom gewählten agilen Vorgehensmodell abhängen können und zudem der Umfang dieser Arbeit überschritten werden würde. Das Modell wird jedoch die Grundlagen schaffen und kann bei Bedarf durch weitere Methoden und Einflussfaktoren ergänzt werden.

Nachdem nun kurz beschrieben wurde, welche Anforderungen das neue Modell abdecken soll bzw. wie es aufgebaut sein wird, werden im nächsten Kapitel die Einflussfaktoren für die Übergabe von agil entwickelten Projekten identifiziert.

4.2 Identifikation der Einflussgrößen

In diesem Kapitel werden nun die einzelnen Einflussfaktoren entlang des Softwarelebenszyklus (vgl. Abbildung 11) identifiziert und knapp erläutert werden. An die Beschreibung jedes Faktors schließt sich jeweils eine Empfehlung für Methoden bzw. Werkzeuge. Wegen ihres großen Umfangs können die einzelnen Werkzeuge nicht im Detail beschrieben werden. Schließlich werden die identifizierten Einflussfaktoren nach ihrem Beitrag zum Erfolg des Vorhabens bewertet.

Die identifizierten Einflussfaktoren im Überblick:

1. **Vorgehensmodell:** Beinhaltet je nach Art des Projektes die möglichen Vorgehensmodelle, welche auszugsweise in Kapitel 2 beschrieben wurden. Natürlich können auch Mischformen oder nur Teile eines Vorgehensmodells eingesetzt werden.

   Die Wahl des „richtigen“ Vorgehensmodells hat sehr hohen Einfluss auf den Erfolg des Projektes. Bei nicht korrekter Wahl, unbeherztem Einsatz oder dem kompletten Weglassen des Vorgehensmodells fehlen wichtige Vorgehensweisen, Werkzeuge, Rollendefinitionen usw. Mögliche bereits bestehende Prozesse, Tools, usw. müssen vor der Wahl des Modells auf deren Kompatibilität geprüft werden. Das Vorgehensmodell stellt daher das Herz des Projektes dar und mit dessen Wahl steht und fällt die Unternehmung.

2. **Verantwortung:** Je nachdem, welches Vorgehensmodell zum Einsatz kommt, kann sich die Verantwortlichkeit entsprechend unterscheiden. Verstanden wird darunter unter anderem die Gesamtverantwortung für den korrekten Ablauf des Projektes, aber auch Verantwortlichkeiten für einzelne Phasen, Rollen usw. Die Verantwortung geht daher sehr stark Hand in Hand mit dem Vorgehensmodell.

   Auf den Erfolg hat die Verantwortung mittelgroßen Einfluss, da im Normalfall dem Kunden gegenüber immer das Unternehmen als Ganzes geradestehen muss und weniger das jeweilige Team. Innerhalb des Unternehmens sind Verantwortlichkeiten jedoch sehr wichtig, um unter anderem eine hochwertige Qualität zu liefern und den Prozess stetig verbessern zu können. Werden die Verantwortungen nicht klar definiert, kann es zu erheblichen Verzögerungen und zu Qualitätseinbußen kommen.

3. **Plattform und Infrastruktur:** Hinter diesem Einflussfaktor verbergen sich alle eingesetzten Systeme. Nicht umfasst sind aber zum Beispiel die eingesetzte Software für

die Versionskontrolle, die Nachverfolgung der Applikation durch den Software-Lebenszyklus, usw., da diese auf Grund ihrer besonderen Relevanz gesondert behandelt werden. Unter Plattform und Infrastruktur werden primär die für das Projekt zur Verfügung stehenden Hardwaregegebenheiten und deren Konfiguration verstanden.

Auf den Erfolg hat dieser Faktor nur geringen Einfluss. Welche Konfigurationskombinationen auch immer bereits für andere Projekte eingesetzt werden, neue bzw. übernommene Projekte sollten prinzipiell auch auf diese aufgebaut werden. Neue Plattformen bzw. Infrastruktur bzw. das damit möglicherweise einhergehende mangelhafte Know-how bergen ein zusätzliches Risiko. Gibt es bei der Übergabe des Projektes noch keine entsprechende Plattform und Infrastruktur, ist hier mit erhöhtem Aufwand und in der Folge mit Verzögerungen beim eigentlichen Projekt zu rechnen.

4. **Verträge:** Beinhaltet die vertraglich festgehaltenen Rahmenbedingungen für das Projekt. Diese können schon vorgegeben sein oder bei der Projektübergabe neu verhandelt werden. Besondere Beachtung verdienen in diesem Zusammenhang eventuell vereinbarte SLA Zeiten oder die Pönaleregelung im Fall des Nichteinhaltens des Vertrages,

   Da die Verträge im Normalfall bei Übergabe eines Projektes bereits vorhanden sind, haben diese nur geringen Einfluss auf den Erfolg, können im schlimmsten Fall aber dennoch für ein schlechtes Projektergebnis sorgen. Sollten bei der Projektübergabe noch keine Verträge existieren bzw. die bestehenden neu verhandelt werden müssen, sollte damit unbedingt so früh wie möglich begonnen werden. Projekte ohne klare Vertragslage weiterzuentwickeln macht nicht wirklich Sinn und kann zu großen Problemen führen.

5. **Know-how:** Beinhaltet das gesamte Wissen zu dem Projekt, eingesetzten Tools, angebundenen Fremdsystemen, Konfiguration usw. Es ist wichtig, ausreichend Know-how in seinem Team zu haben und dieses auch für neue Mitglieder bzw. andere Teams zur Verfügung zu stellen. Hierbei im Fokus steht der Know-how-Transfer im Team bzw. mit anderen Teams, vor allem, wenn diese nicht am selben Standort tätig sind.

   Das vorhandene bzw. nicht vorhandene Know-how hat sehr großen Einfluss auf den Erfolg des Projektes. Fehlendes oder nicht übermitteltes Know-how kann sehr schnell zu Engpässen und im schlimmsten Fall zum Scheitern des Projektes führen. Vor Übergabe des Projektes ist sicherzustellen, dass im übernehmenden Team der notwendige Wissensaufbau abgeschlossen und das entsprechende Know-how bereits verfüg- und einsetzbar ist.

6. **CI-Tools:** Darunter werden die Werkzeuge zu Überwachung und Betreuung des Projektes verstanden. CI-Tools haben die wichtige Funktion, unter anderem Anforderungen zu sammeln und zu bearbeiten bzw. jederzeit den aktuellen Status des Projektes zum Beispiel in Form von einzelnen Aufgaben widerzuspiegeln. Sie bilden die Prozesse bzw. den Software-Lebenszyklus ab und stehen auch den Kunden zur Verfügung.

   Die Wahl der richtigen Tools kann den Projektalltag erheblich erleichtern, hat aber auf den Erfolg nur geringen Einfluss. Fehlt das entsprechende Tool bei der Übergabe des

Projektes oder sind die Arbeitsschritte nicht korrekt abgebildet, wird etwas Einarbeitungszeit notwendig sein. Das Projekt selbst sollte aber nicht gefährdet sein.

7. **Versionskontrolle:** Wie bereits in Kapitel 3.3 beschrieben, ist im Rahmen der Softwareentwicklung die Versionierung besonders wichtig, um regelmäßig Software in hochwertiger Qualität ausliefern zu können. Bei der Wahl der Versionskontrolle ist entscheidend, dass sie mehrere lauffähige Versionen der Software zur Verfügung stellt, diese ordentlich durch Tests abdeckt und ein einfaches Procedere zur Abarbeitung der Deployment Pipeline erlaubt. Zudem sollte gewährleistet sein, dass die einzelnen Builds automatisch erzeugt werden.

   Die falsche Wahl der Versionskontrolle kann die Arbeit zwar erheblich erschweren, auf den Erfolg hat sie letztendlich nur geringen Einfluss. Fehlt die Versionskontrolle aber komplett, ist dies sehr gefährlich und ein Rollback im Fehlerfall nur sehr schwer möglich. Es empfiehlt sich daher unbedingt, mit Versionskontrolle zu arbeiten.

8. **Entwicklung:** Dieser Einflussfaktor fasst alle Komponenten zusammen, die für die eigentliche Entwicklung der Software notwendig sind. Dazu zählen:

   a. Entwicklungsumgebung: Gute Umgebungen bieten unter anderem die Möglichkeit, Tests zu integrieren und Software lokal einfach zum Laufen zu bringen. Auch eine Integration mit der eingesetzten Versionskontrolle ist wichtig. Die Entwicklungsumgebung kann vom Unternehmen vorgegeben sein, im Normalfall aber hängt ihre Wahl von den Präferenzen des Entwicklers ab. Auf das Projekt hat die Entwicklungsumgebung nur sehr geringen Einfluss. Möglicherweise entsteht beim Wechsel kurzzeitig erhöhter Aufwand, der sich aber in Grenzen halten sollte. Das komplette Fehlen von Entwicklungsumgebungen ist im Regelfall ausgeschlossen

   b. Programmiersprachen: Hier geht es darum, welche Programmiersprachen im Unternehmen für das zu übergebende Projekt bereits eingesetzt werden und wieviel Know-how im Team vorhanden ist. Der Wechsel der Sprache kann mit sehr viel Aufwand und hohem Risiko verbunden sein. Daher empfiehlt es sich, die eingesetzten Sprachen tunlichst beizubehalten und ggf. das Wissen des Teams aufzubessern. Denn verfügt das neue Team bei der Übergabe des Projektes noch nicht über die benötigten Kenntnisse, kann dies immense negative Auswirkungen auf das Projekt haben. Muss erst Know-how aufgebaut werden, um ein Projekt überhaupt betreuen zu können, wird dies im besten Fall nur zu Verzögerungen, im schlimmsten Fall aber zum Scheitern des Projektes führen.

   c. Datenbanken: Ähnlich der Frage nach den eingesetzten Programmiersprachen ist es auch von Bedeutung, welche Datenbanken für ähnliche Projekte bereits eingesetzt werden und welches Know-how vorhanden ist. Ein Wechsel der Datenbank birgt aber in der Regel kein so hohes Risiko wie der Wechsel der Programmiersprachen. Dennoch sollte der Aufwand hierfür nicht unterschätzt werden. Fehlt es bei der Projektübergabe an Wissen im Bereich Datenbanken,

kann es zu Verzögerungen bei der Weiterentwicklung des Projektes kommen. In den meisten Fällen reicht jedoch mittleres Wissen in diesem Bereich aus.

Insgesamt hat dieser Faktor keinen hohen Einfluss auf die Übergabe des Projektes, solange nicht ein Wechsel der Programmiersprache in Betracht gezogen wird. In der Praxis ist dies aber als eher unwahrscheinlich anzusehen.

9. **Feedback:** Wie in dieser Arbeit bereits angesprochen und betont, ist es sehr wichtig, dass entlang der Prozesse regelmäßig Feedback in alle Richtungen gegeben wird. Denn nur regelmäßiges und angemessenes Feedback erlaubt es, den Prozess zu verbessern und mögliche Missverständnisse auszuräumen. Um das zu erreichen, empfiehlt es sich, gewisse Mechanismen zu installieren. Dies können zum Beispiel regelmäßige Meetings sein, die durch Tools unterstützt werden. Wichtig ist, dass Feedback auch entsprechend verarbeitet wird.

   Auf den langfristigen Erfolg hat regelmäßiges und ordentliches Feedback großen Einfluss, Die Wahl der entsprechenden Mechanismen eher geringen. Fehlen Feedbackmechanismen bei der Übergabe des Projektes, ist dies zu Beginn zwar nicht kritisch. Es sollten aber so schnell wie möglich zielführende Mechanismen etabliert werden, um den langfristigen Erfolg des Projektes zu gewährleisten.

10. **Testing:** Um regelmäßig Software in hochwertiger Qualität ausliefern zu können, ist es wichtig, diese auch entsprechend zu testen. Eine Teststrategie ist für das Projekt unabdingbar. Ist diese bei Übergabe bereits existent, muss Know-how dafür im Team vorhanden sein, widrigenfalls aber aufgebaut werden. Die einzelnen Builds sind wie in Kapitel 3.3.1 beschrieben, entweder lauffähig oder nicht, solange sie nicht ordentlich getestet sind. (Crispin & Gergory, 2009)

    Die Thematik des Testens hat einen sehr hohen Einfluss auf den Erfolg, denn je später Fehler gefunden werden, umso schwieriger und aufwendiger ist die ihre Behebung. Zudem wird in dieser Arbeit davon ausgegangen, dass früher oder später der in Kapitel 3.5 beschriebene Ansatz eingesetzt werden soll. Hierfür ist eine ansprechende Teststrategie unerlässlich. Fehlen Testmechanismen komplett, so ist dies zum einen mit sehr hohem manuellen Aufwand des Testens verbunden zum anderen können sehr viele Fehler unentdeckt bleiben und in weiterer Folge sehr hohen Aufwand und gravierende Probleme verursachen.

11. **Softwarearchitektur:** Darunter versteht man die Organisation des Systems, mit den einzelnen Komponenten, deren Zusammenspiel, den einzelnen Umgebungen, Design usw. Sie spiegelt daher die Summe der getroffenen Entscheidungen wider, wie die Auswahl der Strukturelemente und deren Schnittstellen, den hierarchischen Aufbau oder auch die Organisation des Softwaresystems. (sei.cmu.edu, 2017) Zudem sollen auch die einzelnen Einflussfaktoren sowie Risiken daraus hervorgehen und die Architektur abschließend ordentlich dokumentiert, kommuniziert und laufend angepasst werden. (Fairbanks & Garlan, 2010)

Die Architektur wird bei der Übernahme eines Projektes im Normalfall nicht geändert werden. Der Einfluss der Architektur im Gesamten ist dennoch als hoch einzustufen, denn es ist wichtig, dass alle Teammitglieder die Architektur verstehen und damit arbeiten können. Heikler wird das Ganze, wenn die Architektur nicht ordentlich dokumentiert wurde. Fehlt diese Dokumentation komplett, ist die weitere Betreuung des Projektes sehr schwierig und es können massive Probleme entstehen. Ist die Dokumentation jedoch vorhanden, so ist der Einfluss auf den Erfolg der Projektübergabe eher gering.

12. **Style Guide:** Ist eine Sammlung von Richtlinien, welche die Gestaltung des Projektes beschreiben. Also zum Beispiel welche Schriftarten in welcher Größe verwendet werden, welche Auflösung die Bilder haben, wie die Navigation aufgebaut wird, welche Abstände eingehalten werden müssen usw. Der Auftritt ist das öffentliche Gesicht des Unternehmens und definiert, wie Kunden, Partner, usw. das Unternehmen wahrnehmen. (executionists.com, 2017), (pc-magazin.de, 2017) Für das Team, das das Projekt übernimmt, ist es wichtig, dass die getroffenen Entscheidungen hinsichtlich des Styles entsprechend dokumentiert und zugänglich sind, damit nicht zusätzlicher Aufwand entsteht.

    Auf den Erfolg hat der Style Guide nur geringen Einfluss. Fehlt dieser bei der Übergabe, sollte versucht werden, ihn während der laufenden Entwicklung und Anpassungen nachzuholen.

13. **Coding Standard:** Damit der Quellcode stetig nach den vorgegebenen Regeln erstellt wird, ist ein Coding Standard notwendig. Dieser trägt auch zur Qualität der Software bei und fördert die regelmäßige Auslieferung von Software in hochwertiger Qualität. Zudem hilft er beim Verständnis des Codes und macht ihn wartbar. Ist ein solcher Standard erst einmal definiert, muss dieser laufend, am besten automatisiert überprüft werden. (Bergin, 2001)

    Auf den Erfolg hat ein solcher Standard langfristig sehr großen Einfluss, da das Projekt gefährdet ist, wenn seine Qualität sinkt. Bei der eigentlichen Übergabe ist der Einfluss auf den Erfolg aber eher nur gering, da im Normalfall im Unternehmen für alle Teams ähnliche Standards gelten. Fehlen solche Standards bei der Übergabe des Projektes und gibt es im neuen Team nicht bereits ähnliches, sollte vor der Übergabe ein solcher definiert werden, um in gleich hochwertiger Qualität auszuliefern. Ohne Standards erhöht sich das Risiko für Fehler und damit einhergehenden Aufwand und höhere Kosten merklich.

14. **Dokumentation:** Auch, wenn das von den eigentlichen Entwicklern oft anders gesehen wird, ist Dokumentation eines der essentiellsten Dinge, wenn es um die Entwicklung von Software geht. Eine ordentliche Dokumentation von zum Beispiel der Softwarearchitektur, dem Style Guide, einzelnen Releases, Features oder User Stories ist sehr wichtig, um unter anderem später nachvollziehen zu können, warum gewisse Entscheidungen in der bestimmten Form getroffen wurden. Zudem hilft es bei der Übergabe von Projekten ungemein, wenn diese ordentlich dokumentiert und die Informationen zentral abgelegt werden. Welche Form der Dokumentation gewählt wird, kann von den im Unternehmen eingesetzten Tools und Richtlinien abhängen. Besonders wichtig ist, dass die

Dokumentation in welcher Form auch immer zu jederzeit und für alle zentral zugänglich ist. (Hruschka, Rupp, & Starke, 2009)

Auf den Erfolg hat die Dokumentation einen hohen Einfluss, denn wenn diese bei der Übergabe schon in ansprechender Form vorhanden und zugänglich ist, kann enorm viel Zeit und Aufwand eingespart werden und die Weiterentwicklung des Projektes gestaltet sich wesentlich einfacher. Fehlt diese komplett oder ist sie nicht auf dem neuesten Stand, können massive Verzögerungen bei der Entwicklung auftreten und der Aufwand dafür, diese nachzureichen, steigt umso weiter an, je länger man damit wartet.

15. **Definition of Ready (DOR):** Bei diesem Einflussfaktor handelt es sich um ein Element, das vor allem im agilen Umfeld zur ordentlichen Verwaltung des Product Backlog Anwendung findet. Es ist wichtig, vorab zu definieren, wann ein neu angelegtes Arbeitspaket zur eigentlichen Umsetzung bereit ist. Fehlt diese Definition, kann es zu erhöhtem Aufwand für die Umsetzung kommen. (Rubin, 2013) Wichtig ist auch hier, dass diese Definition festgehalten und für alle verfügbar zentral abgelegt wird.

    Auf den Erfolg hat die DOR nur geringen Einfluss. Ist diese bei der Projektübergabe nicht vorhanden, kann sich der Aufwand für die Weiterentwicklung zwar erhöhen, das Projekt sollte aber nicht gefährdet sein, solange die Definition schnell nachgeholt wird. Zudem vereinfacht es die Abarbeitung von neuen Features erheblich, wenn diese gleich in richtiger Form, mit allen notwendigen Informationen das Team erreichen.

16. **Definition of Bugs (DOB):** Um mit Bugs schnell und ordentlich umgehen zu können, ist es wichtig, ein Konzept zu haben und zu definieren, was ein Bug genau ist. Also zum Beispiel muss definiert werden, welche Informationen bei der Erstellung eines Bugs übermittelt werden müssen. Zudem muss geklärt werden, wann und wie mit der Behebung des Bugs gestartet wird. (Tian, 2005) Welches Tool hier eingesetzt wird, ist eher nebensächlich. Wichtig ist wieder, dass die Definition kommuniziert, zentral abgelegt und ständig aktuell gehalten wird.

    Auf den Erfolg hat die DOB nur geringen Einfluss, da es bei ihrem Fehlen zwar zu erhöhtem Aufwand kommen kann, aber der ordnungsgemäße Ablauf des Projektes nicht gefährdet ist. Sind diese bei der Übergabe des Projektes nicht vorhanden, so gilt dasselbe wie für die DOR.

17. **Definition of Done (DOD):** Diese Definition enthält eine Liste von Fertigstellungskriterien, die das gesamte Team zur Fertigstellung des Produktes zu beachten hat. Es ist also auch eine Einigung des agilen Teams darauf, was getan werden muss, damit eine Aufgabe als fertig angesehen werden kann. (Lacey, 2016) Dies beinhaltet unter anderem, das zum Beispiel die Builds fehlerfrei erstellt werden können, die Tests ordentlich durchlaufen oder auch, dass eine entsprechende Testanleitung für die Software Development Quality Assurance (SDQA) erstellt wird. Die DOD sind für agile Teams auch als Kontrolle der erledigten Arbeit sehr wichtig, wurden im Normalfall auch zusammen mit dem Kunden entwickelt und können zudem noch sehr viel mehr Einflussfaktoren abdecken. Wichtig ist wieder, dass diese ordentlich dokumentiert, zentral für jeden zugänglich abgelegt und ständig aktuell gehalten werden.

Für die Projektübergabe haben die DOD einen mittleren Einfluss auf den Erfolg, denn wenn solche Definitionen bereits existieren, müssen sie auch nach der Übernahme eingehalten werden. Falls diese fehlen, kann es durchaus vorkommen, dass Aufgaben nicht zur vollen Zufriedenheit aller Stakeholder erledigt werden. Daher empfiehlt es sich auch hier, sie so schnell wie möglich zu erstellen.

18. **Requirements Engineering (RQE):** Beschäftigt sich mit dem „richtigen" Managen von Anforderungen. In der Praxis hat sich gezeigt, dass ein und dieselbe Anforderung von diversen Stakeholdern unterschiedlich verstanden und interpretiert wird. Daher ist es zur exakt der gewünschten Funktionalität entsprechenden Entwicklung der Features vonnöten, bereits bei der Formulierung bzw. dem Managen der Anforderungen anzusetzen. Je später ein Missverständnis aufgedeckt wird, umso aufwendiger und teurer wird es, dies zu korrigieren. (Bray, 2002)

    Richtiges RQE ist daher für einen langfristig erfolgreichen Projektablauf unbedingt notwendig. Sofern im Unternehmen noch keine ähnlichen Bestrebungen existieren, sollten diese so schnell wie möglich in Angriff genommen werden. Auf die Übergabe des Projektes hat RQE nur geringen Einfluss: im Fall, dass die Anforderungen bis dato in angemessener Form eingebracht und verarbeitet werden konnten, ist davon auszugehen, dass dies auch in den ersten Phasen nach der Übergabe möglich sein sollte. Nichtsdestotrotz kann durch den Einsatz von RQE langfristig einiges an Aufwand und Kosten eingespart werden und er sollte daher auf keinen Fall vernachlässigt werden.

19. **Security:** Bei der Entwicklung von Software spielt Sicherheit eine immer größere Rolle. Hierbei geht es zum einen darum, dass der entwickelte Code den Sicherheitsrichtlinien entspricht und zum anderen darum, dass alle angebundenen Systeme sowie auch die verwendete Software von Drittanbietern bzw. auch der Weg zu den Systemen sicher sind.

    Auf die eigentliche Projektübergabe hat Security in dem Fall, dass im Unternehmen bereits Standards usw. zur Verfügung stehen, nur einen geringen Einfluss. Sollte dies bei der Übergabe des Projektes nicht der Fall sein, sollte dies unverzüglich nachgeholt werden. Ansonsten kann das massive Auswirkungen auf den Erfolg des gesamten Projektes haben. Sobald die Übergabe des Projektes durchgeführt worden ist, empfiehlt es sich, die vorhandenen Security-Gegebenheiten zum Beispiel mit OpenSamm[30] genauer zu untersuchen und festzuhalten.

Die zuvor beschriebenen Einflussfaktoren wurden in Bezug auf die in Kapitel 5.1 beschriebenen allgemeinen Gegebenheiten sowie unter Berücksichtigung der in Kapitel 5.2 dargestellten allgemeinen Projektabläufe im Unternehmen identifiziert. Je nach Arbeitsweise des Unternehmens, Handhabung und Aufbau des zu übergebenden Projektes sowie Interessen der einzelnen Stakeholder können noch weitere Einflussfaktoren auftreten.

[30] Ist ein freies Framework zur Evaluierung und Einführung einer Security-Strategie. Vorteil ist der einfache Einsatz, ohne dass sehr detaillierte Vorkenntnisse notwendig sind. Bei dem Einsatz selbst sollte trotzdem Rat von entsprechenden Spezialisten eingeholt werden. (opensamm.org, 2017)

Im nächsten Kapitel werden die bereits identifizierten Einflussfaktoren in eine Tabelle eingetragen, Kategorien zugeordnet und mit Werkzeugen ergänzt.

4.3 Entwicklung des Vorgehensmodells

Wie dem vorigen Kapitel zu entnehmen ist, gibt es bei der Übergabe von Projekten in ein anderes Team eine ganze Reihe von Einflussfaktoren, die zu berücksichtigen sind. Nicht jeder dieser Einflussfaktoren hat im gleichen Maße Einfluss auf den Erfolg der Übergabe. Der Einfluss auf den Erfolg der eigentlichen Projektübergabe wird mit „niedrig", „mittel" und „hoch" bewertet werden. Durch die Kategorisierung der Einflussfaktoren soll unter anderem eine Hilfestellung bei der Vergabe diverser Aufgaben im Zuge der Projektübergabe ans Rollenmodell gegeben und die Auswahl der Werkzeuge erleichtert werden.

Die Einflussfaktoren im Überblick:

<table>
<tr><th>Nr.</th><th>Einflussfaktor</th><th>Wichtigkeit</th><th>Kategorie</th><th>Werkzeuge</th></tr>
<tr><td>1</td><td>Vorgehensmodell</td><td>Hoch</td><td>PM</td><td rowspan="2">• Scrum
• Kanban
• Scrumban</td></tr>
<tr><td>2</td><td>Verantwortung</td><td>Mittel</td><td>PM</td></tr>
<tr><td>3</td><td>Plattform und Infrastruktur</td><td>Niedrig</td><td>KM</td><td>• Puppet
• Salt
• Rundeck</td></tr>
<tr><td>4</td><td>Verträge</td><td>Niedrig</td><td>PM</td><td>• SAP Ariba
• Word
• Jira</td></tr>
<tr><td>5</td><td>Know-how</td><td>Hoch</td><td>PM, QM, KM, SE</td><td>• Jira
• Servicenow
• Trello</td></tr>
<tr><td>6</td><td>CI-Tools</td><td>Niedrig</td><td>PM, QM</td><td>• Jira
• Servicenow
• IBM Jazz</td></tr>
<tr><td>7</td><td>Versionskontrolle</td><td>Mittel</td><td>QM</td><td>Versionskontrolle:
• Jira (Bamboo)</td></tr>
</table>

				• Jenkins • Codeship **Builds:** • Ant • Maven • Travis CI
8	Entwicklung	Niedrig	SE	**Entwicklungsumgebung:** • Eclipse • NetBeans • IntelliJ **Programmiersprachen:** • Java • JavaScript • HTML / CSS • C# • C / C++ **Datenbanken:** • Microsoft SQL • MySQL • Oracle • SAP Hana
9	Feedback	Hoch	A	• Culture Amp • RoundPegg • Office Vibe
10	Testing	Hoch	QM	• Selenium • Gatling • Sonar
11	Software-architektur	Mittel	KM, SE	• Arc42 • Atam

<table>
<tr><td></td><td></td><td></td><td></td><td>• Arid</td></tr>
<tr><td>12</td><td>Style Guide</td><td>Niedrig</td><td>SE</td><td>• JavaDoc
• Jira
• Docstore</td></tr>
<tr><td>13</td><td>Coding Standard</td><td>Mittel</td><td>KM, SE</td><td>• Sonar
• Squale
• PMD</td></tr>
<tr><td>14</td><td>Dokumentation</td><td>Hoch</td><td>PM, QM, KM, SE</td><td>• JavaDoc
• Jira
• Docstore</td></tr>
<tr><td>15</td><td>DOR</td><td>Niedrig</td><td>PM, QM, SE</td><td rowspan="4">• Jira
• Servicenow
• Excel
• Trello
• Pivotal Tracker
• Omnitracker
• IBM Jazz</td></tr>
<tr><td>16</td><td>DOB</td><td>Niedrig</td><td>PM, QM, SE</td></tr>
<tr><td>17</td><td>DOD</td><td>Mittel</td><td>PM, QM, SE</td></tr>
<tr><td>18</td><td>RQE</td><td>Niedrig</td><td>PM, QM</td></tr>
<tr><td>19</td><td>Security</td><td>Niedrig</td><td>KM, QM, SE</td><td>• Logstash
• Graylog
• sPlunk</td></tr>
</table>

Tabelle 1 – Einflussfaktoren entlang des Software-Lebenszyklus (xebialabs.com, 2017), (Toll, 2017), (resources.workable.com, 2017), (Wirdemann, 2011)

Alle entlang des Software-Lebenszyklus identifizierten Einflussfaktoren wurden in Tabelle 1 übersichtlich dargestellt und mit Empfehlungen für diverse Werkzeuge ergänzt. Da sich Projekte und Unternehmen sehr stark unterscheiden, kann diese Aufstellung nicht als vollständig angesehen werden. Jedoch sollte es mit ihrer Hilfe problemlos möglich sein, ein Projekt zu übernehmen. Dies wird im nächsten Kapitel anhand eines praktischen Einsatzes des Modelles evaluiert.

5 ANWENDUNG DES NEUEN VORGEHENSMODELLS

Nachdem in den vorherigen Kapiteln dafür die Grundlagen geschaffen wurden, wird in diesem Kapitel nun das in Tabelle 1 entwickelte neue Vorgehensmodell seine praktische Anwendung finden. Zuvor wird noch kurz das Unternehmen vorgestellt und der allgemeine Projektablauf im Unternehmen erörtert.

5.1 Das Unternehmen

Netconomy ist einer der führenden Dienstleister im Bereich Hybris im deutschsprachigen Raum und beschäftigt derzeit rund 235 Mitarbeiter an Standorten wie Graz, Wien, Zürich und Dortmund. Als einer der ersten Hybris Partner (Partnerstatus Platinum) pflegt Netconomy seit dem Jahr 2003 eine intensive Partnerschaft mit dem Hersteller.

Zusätzlich ist Netconomy der erste Partner aus dem Hybris[31]-Ecosystem, der zugleich auch eine Partnerschaft mit SAP etabliert hat. Ziel ist die Erstellung von nutzbringenden Anwendungen, die Kunden den zielgerichteten Einsatz neuer Technologien ermöglichen. Im Fokus der Betrachtungen stehen daher neben den technischen und funktionalen Kriterien die langfristigen Nutzenaspekte für die Kunden.

Das Dienstleistungsspektrum reicht von der lösungsnahen Beratung über Konzepterstellung und Projektmanagement bis hin zur technischen Implementierung und laufenden Wartung bzw. Weiterentwicklung sowie Betriebsführung (Operations). Um den laufenden Betrieb zu gewährleisten, werden bei Bedarf ergänzend abschließende sowie weiterführende Schulungen durchgeführt.

Das Unternehmen verfolgt eine konsequente Qualitätsstrategie und investiert laufend in die Aus- und Weiterbildung der Mitarbeiter sowie in die Verbesserung der Abläufe und Prozesse. Hybris stellt die zentrale Lösungsplattform für Netconomy dar, wobei eine flächendeckende Produktzertifizierung aller Mitarbeiter angestrebt wird.

Innerhalb der Organisation von Netconomy werden Projektteams, Customer Service / Support und Betriebsführung strikt getrennt, um Auswirkungen notwendiger Unterstützungsleistungen und Problemlösungen im laufenden Betrieb möglichst gering zu halten. Durch diese Trennung können sowohl der Support und Betrieb wie auch die laufende Weiterentwicklung optimal bedient werden.

[31] Hybris gehört zu den führenden Anbietern im Bereich Unternehmenssoftware SAP (Hybris, 2016). Die Produkte basieren auf dem Spring Framework, welches ein umfangreiches Programmier- und Konfigurationsmodell für auf Java basierende Applikationen bereitstellt (Spring.io, 2016).

Zu den Kunden von Netconomy zählen unter anderem die Andritz AG, BayWa AG, Migros, die XXXLutz Gruppe, die Ricardo Gruppe, PayLife Bank GmbH und Leder & Schuh AG. (Netconomy, 2016)

5.2 Allgemeiner Projektablauf

Nachdem – in der Regel nach längeren Phasen der Ausschreibung bzw. Verhandlungen – ein neuer Kunde bzw. ein neues Projekt für das Unternehmen gewonnen werden konnte, beginnt die Phase der ersten Planung. In dieser Phase wird, sofern noch nicht vorhanden, ein entsprechendes Projektteam (Entwicklung, Quality Management, PO, usw.) aufgestellt. Mit Hilfe dieses Teams wird anschließend mit Unterstützung diverser anderer Rollen des Unternehmens, wie zum Beispiel Solution Architects (Solar), die für den Entwurf der Applikation (Softwarearchitektur) inklusive Serverinfrastruktur herangezogen werden, das Projekt in groben Zügen geplant.

Als Vorgehensmodell wird im Unternehmen zum Großteil das unter Kapitel 2.5.3 beschriebene Scrum verwendet. Daher werden im Zuge der Planung alle notwendigen Artefakte wie Sprint Backlog, Release Plan usw. geschaffen, um anschließend mit der Entwicklung beginnen zu können. Zudem werden vor dem eigentlichen Systemstart (bevor die erste Codezeile geschrieben wird) alle konfigurations- und systemspezifischen Voraussetzungen geschaffen. Dies beinhaltet einige der zuvor in Kapitel 3 beschriebenen Elemente, wie zum Beispiel die Versionskontrolle oder alle notwendigen Systeme (Produktions-, Test-, Entwicklungssystem). Diese Phase ist im Normalfall von sehr vielen Meetings geprägt. Letztendlich schließt diese Phase mit einem Projektsetup inklusive aller Meetings ab.

Nach Erfüllung aller Vorbedingungen wird mit der eigentlichen Entwicklung begonnen. Die Entwicklung wird analog dem Vorgehensmodell Scrum durchgeführt, also als regelmäßige Iteration in Form von Sprints, die jeweils mit Reviews bzw. Retrospektiven enden. Auch andere bereits beschriebene Artefakte werden angewendet und am Ende wird immer ein fertiges Release mit zusätzlichen Funktionen oder behobenen Fehlern geliefert. Die Sprintzyklen haben im Unternehmen keine fixe Länge und können je nach Projekt variieren. Andere Teams wie zum Beispiel Operations, die für die Konfiguration, Infrastruktur usw. verantwortlich sind, arbeiten nach dem agilen Ansatz Kanban.

Nach „vollendeter“ Entwicklung, also wenn eine Applikation einen Status erreicht, der den Einsatz in einer Produktionsumgebung erlaubt, können Kunden zusätzlich eine Wartungs- und Servicevereinbarung bzw. Vereinbarung für den Betrieb eingehen. Die Entscheidung liegt hier beim Kunden und es sind verschiedenste Kombinationen möglich. Zudem gibt es noch die Möglichkeit, eine Vereinbarung zu Bereitschaften außerhalb der Geschäftszeiten einzugehen. In

den ersten Wochen nach dem Go-Live[32] wird die Applikation in den meisten Fällen noch vom Entwicklungsteam hinsichtlich Wartung usw. betreut.

Während dieser Phase wird das Projekt, sofern die entsprechenden Vereinbarungen getroffen wurden, an den Customer Service und Support (CSS) übergeben. Die Übergabe bedeutet, dass alle notwendigen Artefakte wie Serverinfrastruktur, Zugänge, Dokumentationen usw. zur Verfügung gestellt werden. Zudem wird die Übergabe von mindestens einem Entwickler begleitet, der das Projekt inklusive wichtigste Funktionen, bereits bekannte Fehler, usw. vorstellt. Die Mitarbeiter des CSS erhalten zudem alle notwendigen Informationen hinsichtlich Versionskontrolle, Repository, Build der Applikation usw. Nach der Übergabe muss es möglich sein, zur Suche möglicher Fehler die Applikation lokal laufen zu lassen. Dies funktioniert in der Theorie sehr gut. In der Praxis hat sich gezeigt, dass bei der Übergabe von Projekten selbst zu reinen Wartungszwecken die bisher eingesetzten Methoden nur bedingt funktionieren. Das in dieser Arbeit entwickelte Vorgehensmodell könnte in absehbarer Zukunft zum Teil auch für die Übergabe von Projekten in den CSS im Unternehmen eingesetzt werden. Wobei hier zu erwähnen ist, das der CSS derzeit kein agiles Vorgehensmodell einsetzt, sondern lediglich versucht, so agil wie möglich zu agieren. Dennoch ist durch den Einsatz diverser agiler Methoden im Unternehmen ein großes Wissen im CSS notwendig, um mit den Teams effizient zusammenarbeiten zu können.

Während sich Projekte nun in einer sogenannten Wartungsphase befinden, wird im Normalfall die Software laufend um neue Features erweitert und bestehende Fehler ausgebessert. Bei vielen dieser Projekte hat sich in der Praxis jedoch gezeigt, dass sich nach dem erfolgreichen Go-Live und einer weiteren Phase wesentlicher Weiterentwicklungen ein eigenes Entwicklungsteam sowohl für Kunden als auch für das Unternehmen nicht mehr „lohnt“. Dies betrifft vor allem kleine bzw. mittelgroße Projekte. Große bzw. sehr große Projekte sind von diesem Umstand nicht betroffen. Die Mitglieder des ursprünglichen Teams werden im Regelfall anderen Teams (Projekten) zugeordnet und das Projekt befindet sich über eine längere Phase in einem quasi unveränderten Zustand. Fehler werden zwar weiterhin behoben, aber die Funktionalität der Applikation bleibt weitestgehend dieselbe. Befindet sich ein Projekt einmal in dieser Phase, sind Weiterentwicklungen sehr schwierig, da meistens kein Team mehr verfügbar ist. Dieser Umstand hat sehr oft dazu geführt, dass wesentliche Veränderungen innerhalb der Teams notwendig waren, um ein solches Projekt wieder betreuen und weiterentwickeln zu können. Teilweise wurden dringende Änderungen im CSS durchgeführt, welcher aber nur bedingt über das Know-how zur Entwicklung von Software verfügt.

Ausgehend von dieser Tatsache, soll nun ein neues Team mit dem Namen Rapid Solution Development Team 1 (RSD 1) gegründet werden. Dieses Team wird nicht wie alle anderen Entwicklungsteams organisatorisch eingeteilt, sondern direkt im CSS angesiedelt sein. Da alle anderen Entwicklungsteams auf denselben Ressourcenpool (Entwickler, SDQA, usw.) zugreifen, wurde bewusst entschieden, dieses Team davon abzukapseln. Denn mit Hilfe dieses Teams

[32] Go-Live ist der Tag, an welchem die Applikation am Produktionssystem ausgeführt wird und den Kunden zur Verfügung steht.

sollen in Zukunft all jene kleinen bzw. mittelgroßen Projekte betreut und weiterentwickelt werden, bei welchen sich der Einsatz eines separaten Teams nicht „lohnt“. Zudem wird gerade an einer Lösung für Klein- und Mittelbetriebe gearbeitet, die zur Gänze im RSD Team[33] umgesetzt werden soll. Um so viele Projekte gleichzeitig und auf Dauer managen zu können, ist es besonders wichtig, dass der Kern des Teams sich nicht ständig ändert. Zum Beginn soll das RSD1 Team aus mehreren Entwicklern (mindestens einem Senior), einem Verantwortlichen für die Qualität sowie notwendigen POs bestehen. Ein erstes Projekt soll nun in dieses Team übernommen und dort weiter betreut werden.

Nachdem in diesem Kapitel die Ausgangslage skizziert wurde, wird im nächsten Kapitel das erste Projekt übergeben werden.

5.3 Das Projekt

Aus unternehmerischen Gründen wird bei der praktischen Umsetzung auf den Namen des Projektes verzichtet und dieses in weiterer Folge E-Commerce Project X (EPX) genannt. Der eigentliche Projektname wird auch in den weiter hinten abgebildeten Screenshots unkenntlich gemacht. Ebenso werden die einzelnen Mitglieder des neuen RSD1 Teams mit Ausnahme des Autors aus Datenschutzgründen nicht namentlich genannt werden.

Das Unternehmen, dessen Projekt weiterentwickelt werden soll, hat mehrere Tausend Mitarbeiter, ist weltweit tätig und macht einen Umsatz im Milliardenbereich. Bei der Software selbst handelt es sich um ein B2B Shopsystem mit Anbindung zu dem führenden[34] SAP System, das in verschiedene Portfolios unterteilt ist. Die Abwicklung des Projektes ist wie in Kapitel 5.2 beschrieben durchgeführt worden.

Vor der Übergabe ist die Applikation bereits in einem bedingt lauffähigen Zustand. Es gab im Vorfeld einige Probleme, die im Zuge diverser Eskalationsmeetings der Geschäftsführungen aus dem Weg geräumt wurden. Die derzeitige Situation ist jedoch für beide Seiten nicht gerade von Vorteil und die Mitglieder des ursprünglichen Entwicklungsteams wurden bereits anderen Teams zugeteilt. Daher wurde mit dem Kunden vereinbart, ein neues Team auf die Beine zu stellen, in welchem das Projekt erfolgreich zum Abschluss gebracht und weiterentwickelt werden soll. Begonnen wird hier mit der Fertigstellung der zum jetzigen Zeitpunkt nahezu Go-Live-fähigen Applikation.

Für einen erfolgreichen Go-Live müssen nach der Übergabe ins RSD1 Team noch einige Funktionen ergänzt sowie einige Go-Live-kritische Fehler behoben werden. Die Übergabe wird

[33] Werden diese Projekte im RSD Team umgesetzt, wird dieses um einige weitere Teams ergänzt werden. Daher ist es besonders wichtig, dass die erste Projektübernahme erfolgreich ist und die notwendigen Gegebenheiten für weitere Projekte geschaffen werden.

[34] Führend bedeutet in diesem Fall, dass Hybris nicht alleine das Hauptsystem ist. Informationen wie Preise, Lieferzeiten, Berechtigungen usw. kommen aus einem angebundenen SAP. Informationen werden daher nur zur Darstellung im Frontend gespeichert. Die Wahrheit liegt jedoch im SAP-System des Kunden.

von dem ehemaligen Team unterstützt, welches bis zum Go-Live bedingt auch für weiterführende Anliegen zur Verfügung steht.

Nachdem nun die allgemeine Beschreibung des Projektes abgeschlossen ist, wird im nächsten Kapitel der Aufbau des neuen Teams beschrieben.

5.4 Das Team

Wie bereits in Kapitel 5.2 beschrieben, soll das neue RSD1 Team aus mehreren Entwicklern, einem SDQA-Verantwortlichen (SDQA Manager) sowie notwendigen POs bestehen. Ein Teil des neuen Teams wird sich aus bestehenden Mitarbeitern des CSS zusammensetzen, die zum Teil eine Doppelrolle ausführen werden. Hauptaugenmerk liegt klar auf der Fertigstellung des übernommenen Projektes. Das Team wird durch einen ehemaligen PO als SDQA Manager und zusätzliche Entwickler aus einem Joint Venture komplettiert. Die Rolle des neuen PO wird doppelt besetzt: Sie wird zum einen vom Autor dieser Arbeit, der selbst für den CSS tätig ist, ausgeübt werden, zum anderen wird ein weiterer PO (ebenfalls aus dem CSS) das Team unterstützen. Die Doppelbesetzung dieser Rolle erfolgt vor dem Hintergrund, dass in Zukunft mehrere weitere Projekte betreut werden sollen.

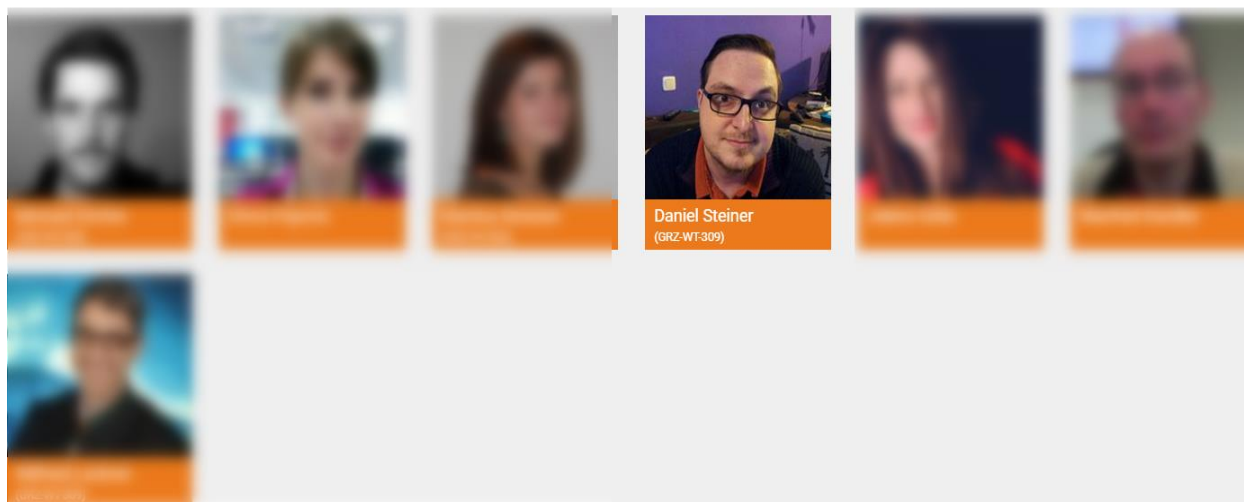


Abbildung 17 – RSD1 Team

Abbildung 17 zeigt die initiale Zusammenstellung des Teams, das über mehrere Standorte in Österreich, Deutschland und Serbien verteilt tätig sein wird. Sollten auf den erfolgreichen Abschluss des ersten Projektes wie geplant weitere Projekte folgen, wird das Team durch diverse Entwickler verstärkt werden. Die Erfahrungsstufen der derzeitigen Entwickler sind sehr unterschiedlich und bei einigen ist es sogar schwierig, diese vorab einzuschätzen.

Nachdem in diesem Kapitel das neue Team kurz vorgestellt wurde, wird im Folgenden die eigentliche Übergabe des ersten Projektes EPX nachgezeichnet werden.

5.5 Die Projektübernahme

Nachdem in vorigen Kapiteln alle notwendigen Voraussetzungen für eine erfolgreiche Projektübernahme geschaffen und ein neues Team auf die Beine gestellt wurde, wird nun mit der eigentlichen Projektübernahme in das RSD1 Team begonnen. Die in Tabelle 1 durchnummeriert aufgelisteten Faktoren werden abgearbeitet und die Ergebnisse zum Teil beschrieben. Die Reihenfolge der Aufstellung in Tabelle 1 wird dabei nicht eingehalten. Um eine einfachere Übernahme des Projektes zu gewährleisten, wird das über mehrere Standorte verteilte Team für diese Phase an einem Standort tätig sein.

5.5.1 Vorgehensmodell, Verantwortung und Kommunikation

Als Erstes wird im gesamten Team evaluiert, welches Vorgehensmodell für die weitere Betreuung des Projektes eingesetzt werden soll. Da das gesamte Team bereits sehr viel Erfahrung mit Scrum hat, wird entschieden, dies erstmals auch für dieses Projekt einzusetzen. Sobald mehrere Projekte betreut werden, wird das Model erneut evaluiert werden. In weiterer Folge ist der Einsatz von Kanban oder einer Mischform aus verschiedensten agilen Methoden durchaus wahrscheinlich.

Die Verantwortung ergibt sich aus dem Rollenmodell von Scrum. Die Gesamtverantwortung für das Projekt liegt beim PO. Alle Belange betreffend Testing, Qualität usw. gehen in die Verantwortung des SDQA Managers. Die Entwickler selbst sind mit Unterstützung der übrigen Rollen für die regelmäßige Ablieferung der Software in hoher Qualität verantwortlich. Die Aufgaben des Scrum Master werden zu Beginn von PO und SDQA-Verantwortlichem übernommen.

Der konkrete Ablauf eines Sprints sieht wie folgt aus:

- Fixierte Sprintlänge von 21 Tagen: 14 Tage für die eigentliche Entwicklung und 7 Tage für die Abnahme des Kunden. Am letzten Entwicklungstag erfolgt auch das Deployment auf das Abnahmesystem (Q System) des Kunden, von welchem die Aufgaben dann abgenommen werden.
- Regelmäßiges Backlog Grooming mit dem Kunden zur Priorisierung der einzelnen Aufgaben.
- Sprint Planning 1 zur Abklärung des geplanten Sprints mit dem gesamten Team. Vorstellung der einzelnen Aufgaben durch den PO. Etwaige Fragen sollen im Zuge dieses Meetings geklärt werden. Außerdem werden in diesem Meeting zusätzliche Aufgaben wie Deployment Master vergeben.
- Sprint Planning 2: Sind alle Aufgaben für alle Teilnehmer verständlich, ziehen sich die Entwickler zurück und besprechen die einzelnen Tasks hinsichtlich ihres Aufwands. Am Ende des Meetings verpflichtet sich das Team die Aufgaben des Sprints betreffend.

- Während der Abnahmephase arbeitet das Team in dem nächsten Sprint weiter. Werden im Zuge der Abnahme Release Blocker[35] gefunden, werden diese sofort im aktuellen Sprint für das letzte Release erledigt. Alle anderen Defects/Bugs werden, sofern nicht anders besprochen, im Backlog Grooming gemeinsam mit dem Kunden priorisiert und für die nächsten Sprints vorbereitet.
- Vor dem Q Deployment gibt es ein Review Meeting mit allen Stakeholdern, in welchem alle durchgeführten Aufgaben des Sprints kurz präsentiert werden.
- Jeder Sprint schließt mit einer Sprint Retro ab, in welcher der Sprint evaluiert wird. Dadurch soll es möglich sein, regelmäßig den Output sowie die Arbeitsweise zu verbessern. Auch Probleme, für die während der regulären Entwicklung möglicherweise keine Zeit ist, sollen hier angesprochen und es soll nach Lösungen gesucht werden.

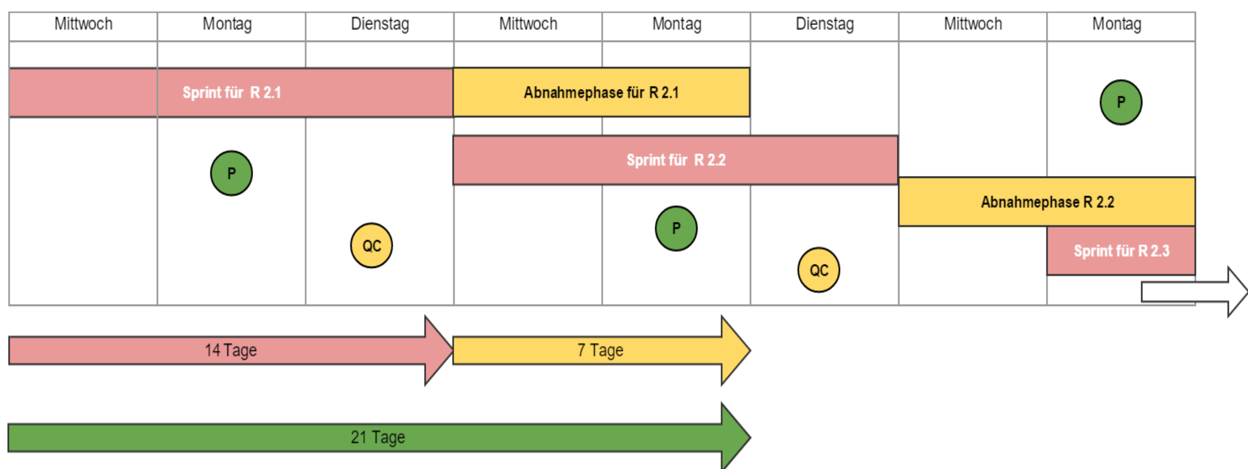


Abbildung 18 – Release Plan EPX

Abbildung 18 zeigt beispielhaft, wie zukünftig mit Releases umgegangen wird. Hier wurde bewusst auf die Festlegung des genauen Starttermins verzichtet, da mit dem Kunden vereinbart ist, dass alle notwendigen Arbeiten für den ersten Go-Live in einem Sprint mit flexibler Laufzeit erledigt werden. Sobald dieser abgeschlossen ist, werden die Sprints wie oben beschrieben aufgebaut. Während dieses „finalen" Sprints wird es laufend Statusaktualisierungen in Form von Review Meetings geben.

Durch die Verteilung des Teams auf mehrere Standorte ist der Einsatz diverser Kommunikationstools unumgänglich. Als Hauptkommunikationsmittel kommt Slack[36], welches im Unternehmen bereits genutzt wird, zum Einsatz. Dies bietet zum einen die Möglichkeit, das gesamte Team ständig an allen Meetings visuell teilhaben zu lassen und zum anderen können auch Kunden in die Kommunikation miteingebunden werden.

[35] Beschreibt ein Fehlverhalten der Software welches extreme technische oder geschäftskritische Auswirkungen hat.

[36] Kommunikationstool in welches viele externe Tools wie Google Hangout (Videokonferenz) einfach integriert werden können.

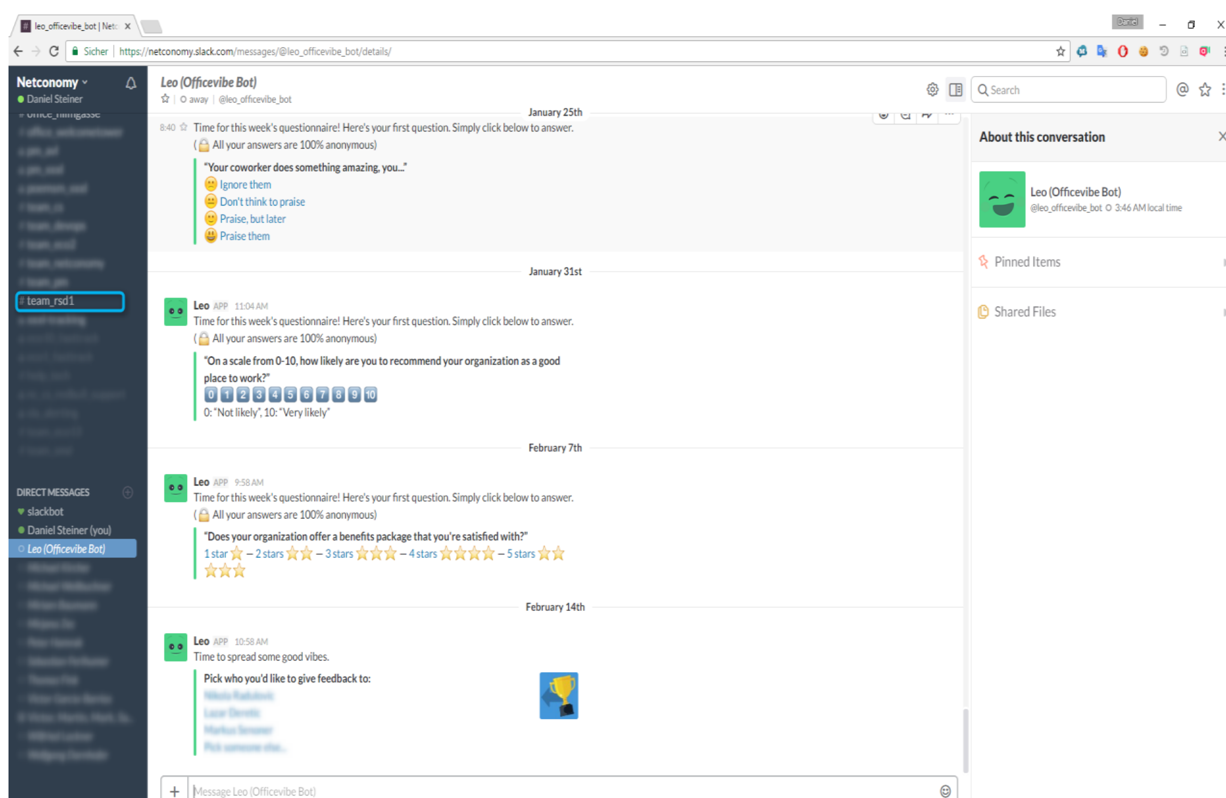


Abbildung 19 – Slack RSD1

Wie in Abbildung 19 zu sehen ist, wird die gesamte Kommunikation in einem eigenen Channel durchgeführt werden. Kommunikation via E-Mail wird auf ein Minimum reduziert. Slack soll aber nicht die Dokumentation ersetzen!

Aufbauend auf die regelmäßig durchgeführten Sprint Retro Meetings, werden zur Verbesserung der Teamperformance vierteljährlich Feedbackgespräche mit dem Team durchgeführt werden. Um regelmäßiges Feedback zu erhalten, wird Office Vibe[37] mit einer Slack-Integration zum Einsatz kommen (vgl. Abbildung 19). Dies bietet eine einfache Möglichkeit, in regelmäßigen Abständen vom gesamten Team das gewünschte Feedback einzuholen. Die bereits erwähnten Feedbackgespräche im Rahmen regelmäßiger vierteljährlicher Workshops werden jedes Mal an einem anderen Standort stattfinden. Mithilfe dieser Maßnahme soll der Zusammenhalt des Teams erhalten bzw. gestärkt werden. Wenn sich in absehbarer Zukunft das RSD Team vergrößert bzw. mehrere RSD Teams zum Einsatz kommen, werden diese Maßnahmen noch entsprechend erweitert.

5.5.2 Jira und Confluence

Nachdem die grundlegenden Fragen nach Vorgehensmodell, Verantwortung und Kommunikation geklärt sind, wird als nächstes nach einem Weg gesucht, dies in Form von Prozessen usw.

[37] Software zur einfachen Messung von Mitarbeiterzufriedenhalt und zum Sammeln von Feedback.

abzubilden. Da für das Projektmanagement im Unternehmen bereits Jira[38] eingesetzt wird, wird es auch für das gegenständliche Projekt zur Anwendung kommen. Die derzeit für ähnliche Entwicklungsprojekte definierten Prozesse wurden nur minimal verändert. Zum besseren Verständnis seitens aller Beteiligten wurden anschließend die aktuellen Prozesse im Confluence (Wiki Software zu Jira) dokumentiert. Zudem wurde mit allen Stakeholdern vereinbart, dass die Kommunikation zum Großteil mittels Slack, Jira und Confluence stattfinden muss und der Mailverkehr so gering wie möglich gehalten werden soll. Alle relevanten Informationen müssen in den entsprechenden Tasks vorhanden sein.

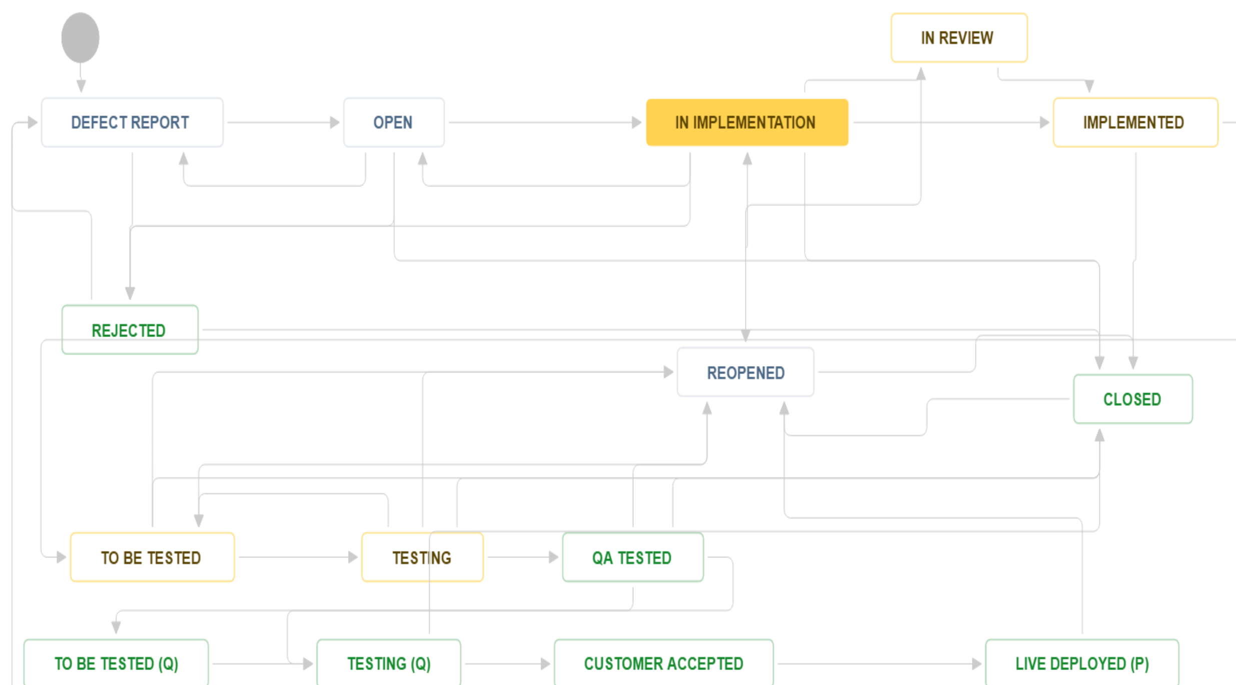


Abbildung 20 – Workflow RSD1

Als Beispiel für diese Dokumentation soll der in Abbildung 20 gezeigte Workflow dienen. Die Visualisierung der einzelnen Arbeitsschritte wurde mit detaillierten Beschreibungen und Definitionen von Verantwortlichkeiten ergänzt. Zusätzlich wurden im Zuge dieser Tätigkeiten die davor noch nicht existenten DOR, DOB und DOD in Abstimmung mit dem gesamten Team nachgeholt.

[38] Jira von Atlassian ist ein CI-Tool zum Managen von Projekten. Es besitzt viele Integrationen wie ein eigenes Wiki (Confluence). Im weiteren Verlauf dieser Arbeit wird, wenn von Jira gesprochen wird, das gesamte System inklusive aller genutzten Integrationen verstanden.

Dashboard Netconomy Teams Locations Projekte IT Knowledge Clans Suche

RSD 1 Erstellen

Diesen Space durchsuchen

- Estimation Meetings
- Releaseplan
- RSD 1 Startseite
  - Holiday Planning
  - Projektmanagement New
    - JIRA Workflow
    - Definition of Defect Priority
    - Description of Bugs and Defects
    - **Definition of Done**
    - Definition of Ready
    - Notes

Seiten / RSD 1 Startseite / Projektmanagement New Bearbeiten

Definition of Done

Angelegt von Daniel Steiner, zuletzt geändert am Mär 06, 2017

General Definition:

- All acceptance criteria assigned to this user story (JIRA Issue) are correctly implemented.
- There are no open related Sub-Tasks or Sub-Bugs (These are created during code reviews, testing or sprint reviews).
- Code compiles on the CI build - All releated builds are green
- All automated tests pass
- Code style fits the internal agreement
- Test documentation according to the SDQA standard is created.
- Code is merged to the appropriate branch (no open Pull Requests)

Team Extension:

- Code is reviewed and passed passed by another developer
- Code reviewed & passed (Pull Requests)
- Test is done on the stage system and **Developer documentation is created (i.e. how to setup, configuration params added, etc.).**
- Release notes are written
- Successful 2nd Level Test - QA tests the functionality
- User Story is PO accepted
- Written Release Notes have been created in CFL and sent out

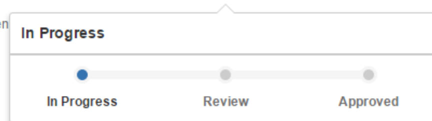
Abbildung 21 – Dokumentation RSD1

Da Jira mit Confluence eine ansprechende Möglichkeit zur Dokumentation bietet, wie Abbildung 21 zeigt, wurde entschieden, dies für die gesamte projektrelevante Dokumentation zu nutzen. Während der Übergabe werden die bisher definierten Dokumentationsrichtlinien beibehalten. Nach dem ersten Go-Live werden als Erstes mögliche alternative Tools zur Code-Dokumentation bzw. zur einfacheren Erstellung von Testdokumentationen evaluiert werden.

Auch die Einflussfaktoren Verträge, Know-how, Softwarearchitektur, Style Guide und auch RQE werden mittels den Werkzeugen die Jira und Confluence bilden abgedeckt.

2e. EPX System - Set up a democase IN PROGRESS

Angelegt von Daniel Steiner, zuletzt geändert vor 4 Minuten



User Story

As a EPX shop manager I want to set up a demo case with one product as a proof of concept so that I can be sure the integration with the external system and hybris will work for my customers.

Jira-Links

Additional Links

Acceptance Criteria

1. The EPX Product 00001 is used as the demo product.
2. The demo products media page can be accessed via the category navigation.
3. The demo products media page can be accessed via full text search.
4. A "Configure" button is available at the media page (3a. Adapt media page).
5. No detailed price is shown on the media page.

Abbildung 22 – RQE RSD1

Abbildung 22 ist die beispielhafte Darstellung einer User Story inklusive des Workflows, bis diese so definiert ist, dass sie in Form von Tasks umgesetzt werden kann.

Wie bereits zuvor beschrieben, kann durch Jira in Kombination mit Confluence ein großer Teil der Einflussfaktoren abgedeckt werden. Dies alleine reicht aber zum Beispiel im Falle des Faktors Know-how nicht aus. Im Unternehmen finden unter anderem bereits regelmäßige Meetings zum Know-how-Austausch zwischen den Teams statt. Das neue RSD1 Team wird nun auch an diesen Meetings teilnehmen, um auf diese Weise sein Know-how stetig zu verbessern. Daneben wird ein intensiver Austausch der anderen Scrum-Rollen im Unternehmen stattfinden.

5.5.3 Die Entwicklung

Da das ursprüngliche Projekt bereits in Java umgesetzt wurde, wird diese Programmiersprache auch beibehalten. Auch alle anderen eingesetzten Sprachen sowie die Datenbank (Oracle) werden beibehalten. Auch die bestehende Infrastruktur wird weiterverwendet. Diese sieht für das aktuelle Projekt überblicksmäßig wie folgt aus:

- Internes Testsystem: Wird intern vom Team verwaltet. Änderungen werden mittels Mock Files[39] getestet. Es besteht keine Anbindung an Fremdsysteme des Kunden.

[39] Sind Platzhalter für echte Objekte innerhalt eines Tests. Mit ihnen können zum Beispiel Schnittstellen ohne echte Anbindung an die Fremdsysteme getestet werden. (Freeman, 2017)

- Externes Testsystem: Einmal in der Nacht und nötigenfalls auch manuell wird der Codestand auf das externe System des Kunden deployed. Hier existiert bereits eine Anbindung an die einzelnen Fremdsysteme.
- Externes Qualitativsystem: Auf diesem System passiert abschließend die Abnahme der umgesetzten Aufgaben. Dieses System verfügt über eine Anbindung zu denselben Systemen wie das Produktivsystem. Zusätzlich ist der Datenstand ähnlich dem Produktivsystem.
- Produktivsystem: Jenes System, auf welches dann alle Kunden Zugriff haben. Da der Betrieb nicht von der Netconomy durchgeführt wird, ist der Zugriff für das RSD1 Team hier beschränkt.

Alle Systeme außer dem internen Testsystem werden von Kunden bzw. deren Partner verwaltet und konfiguriert. Das interne Testsystem wird vom Netconomy Operations Team verwaltet, welches die Systeme mit Puppet und Salt konfiguriert. Da die Konfiguration der Systeme in der Gesamtheit nicht vom RSD1 gemanagt wird, ist es hier zunächst nicht notwendig, Anpassungen vorzunehmen. Insgesamt verfügt das gesamte Team über ausreichend Know-how in den einzelnen Bereichen, sodass mögliche Änderungen an Sprachen, der Datenbank, usw. von vornherein verworfen wurden.

Die Wahl der Entwicklungsumgebung stand den Entwicklern frei. Es haben sich jedoch alle für den Einsatz von IntelliJ entschieden. Als Coding Standard wurde der unternehmensweit definierte Standard herangezogen.

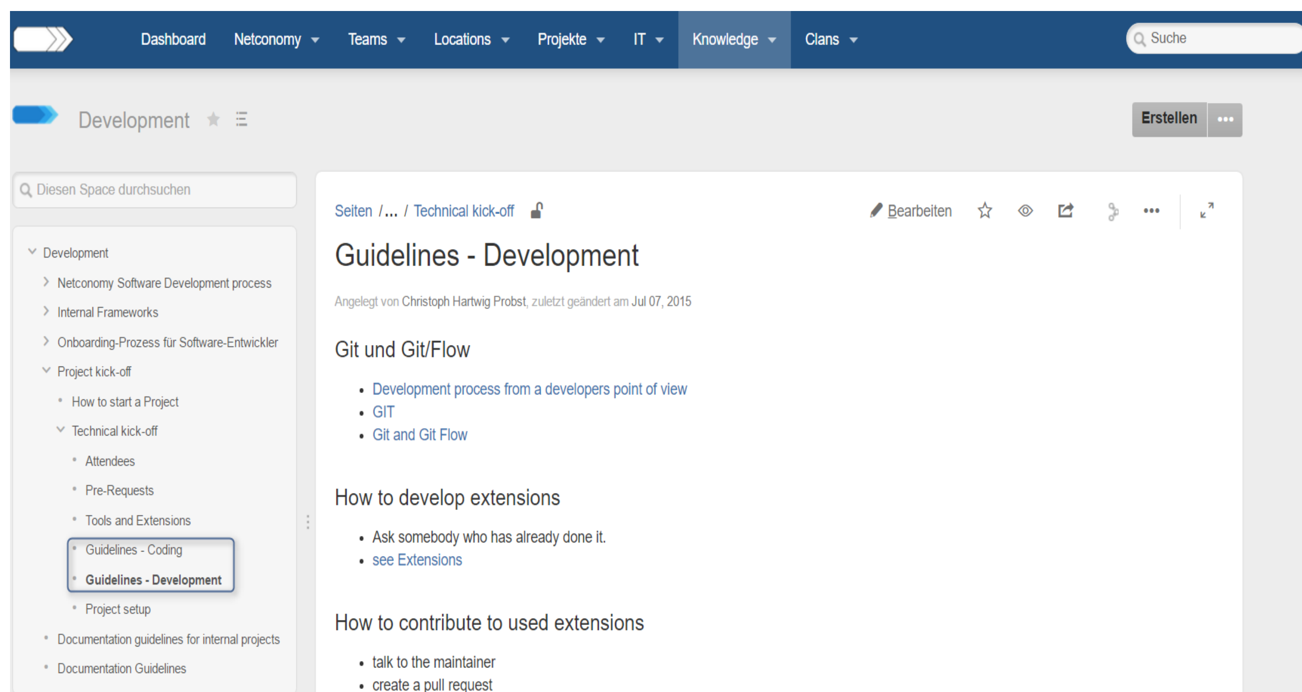
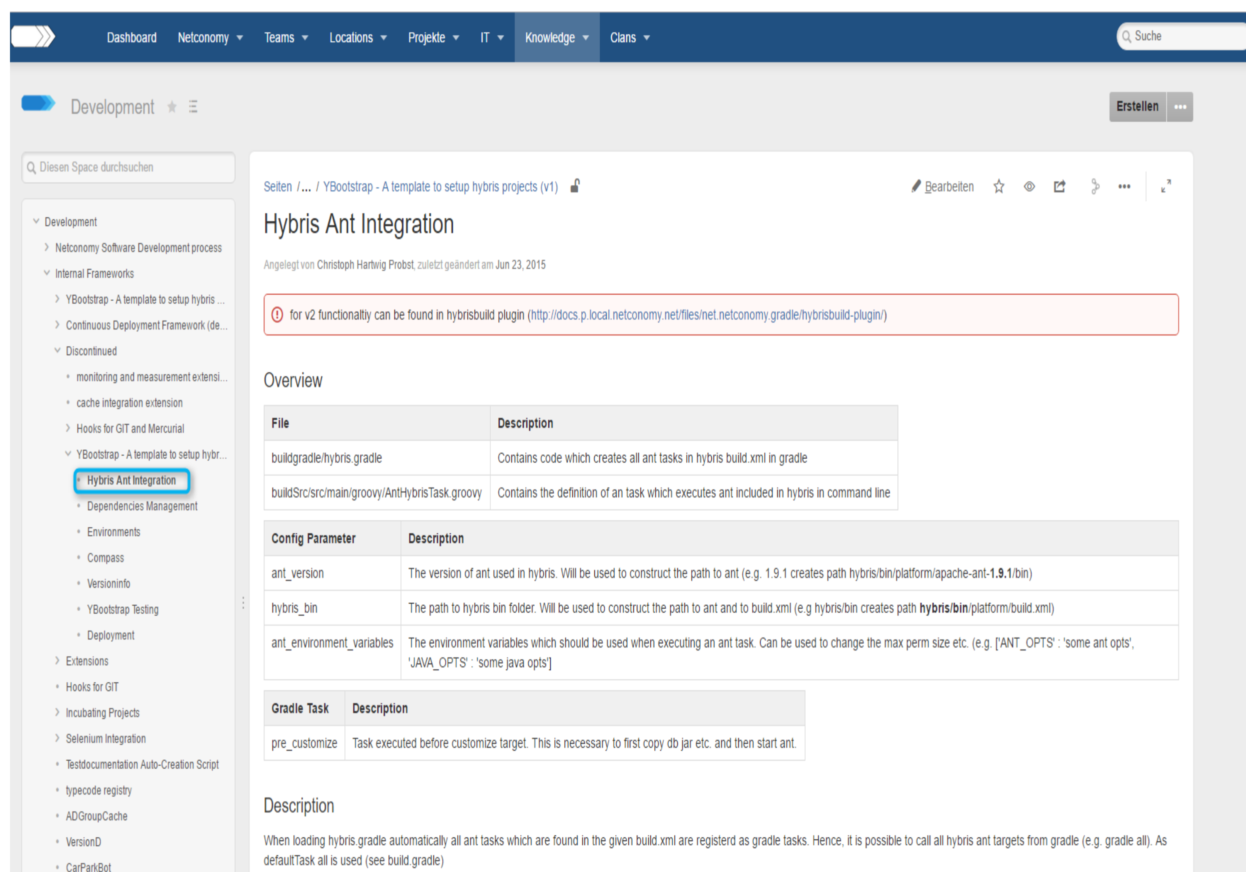


Abbildung 23 – Guidelines Development RSD1

Abbildung 23 bietet eine Übersicht der definierten Standards, welche auch für das RSD1 Team zum Tragen kommen. Diese Standards werden schon jetzt mittels Sonar überprüft, das wird daher auch beibehalten.

Im Normalfall werden die Applikationen von Hybris mit dem Build-Werkzeug Ant gebaut. Innerhalb des Unternehmens wird dies aber mit Gradle durchgeführt. Da dies unternehmensweit einheitlich ist, wird hier im Zuge der Projektübernahme auch nichts daran geändert.



Hybris Ant Integration

Angelegt von Christoph Hartwig Probst, zuletzt geändert am Jun 23, 2015

for v2 functionaltiy can be found in hybrisbuild plugin (http://docs.p.local.netconomy.net/files/net.netconomy.gradle/hybrisbuild-plugin/)

Overview

File	Description
buildgradle/hybris.gradle	Contains code which creates all ant tasks in hybris build.xml in gradle
buildSrc/src/main/groovy/AntHybrisTask.groovy	Contains the definition of an task which executes ant included in hybris in command line

Config Parameter	Description
ant_version	The version of ant used in hybris. Will be used to construct the path to ant (e.g. 1.9.1 creates path hybris/bin/platform/apache-ant-**1.9.1**/bin)
hybris_bin	The path to hybris bin folder. Will be used to construct the path to ant and to build.xml (e.g hybris/bin creates path **hybris/bin**/platform/build.xml)
ant_environment_variables	The environment variables which should be used when executing an ant task. Can be used to change the max perm size etc. (e.g. ['ANT_OPTS' : 'some ant opts', 'JAVA_OPTS' : 'some java opts']

Gradle Task	Description
pre_customize	Task executed before customize target. This is necessary to first copy db jar etc. and then start ant.

Description

When loading hybris.gradle automatically all ant tasks which are found in the given build.xml are registerd as gradle tasks. Hence, it is possible to call all hybris ant targets from gradle (e.g. gradle all). As defaultTask all is used (see build.gradle)

Abbildung 24 – Hybris Build RSD1

Abbildung 24 ist überblicksmäßig zu entnehmen, wie das Builden der Software mittels Gradle anstatt Ant im Unternehmen funktioniert. Die einzelnen Builds werden mittels Jenkins, das eine Integration zu Jira hat, verwaltet. Mittels SourceTree und Git werden die einzelnen Änderungen des Teams durchgeführt. Die Sicherheit der Applikationen wird derzeit nur sehr rudimentär mittels Graylog überprüft.

Abbildung 25 – Jenkins RSD1

Abbildung 25 zeigt, wie die einzelnen Builds für das Projekt zum Zeitpunkt des Abschlusses dieser Arbeit aussehen. Die aktuell bestehenden Tests hinsichtlich Funktionsfähigkeit der Software werden 1:1 übernommen. Nach dem erfolgreichen ersten Go-Live wird in Zusammenarbeit mit dem SDQA Manager und einem Teil des Entwicklungsteams diese Strategie erweitert werden. Auch beibehalten wird die derzeit mittels Selenium umgesetzte Testingstrategie, die aber in einem nächsten Schritt erheblich erweitert werden soll, um einen möglichen Einsatz von Test-Driven Development vorzubereiten. Auch im Bereich Sicherheit sollen die bisherigen Umsetzungen mittels Graylog evaluiert und angepasst werden. Nachdem alle relevanten Informationen, Tools, Code, usw. von dem bisherigen Entwicklungsteam übergeben wurden, standen als letzter Punkt nur noch die lokalen Entwicklungsumgebungen des Teams auf dem Plan. Vor dem Hintergrund der bereits geleisteten Vorarbeit und des gemeinsamen Verständnisses stellte dies nur mehr einen vergleichsweise kleinen Schritt zur Vollendung der Projektübernahme dar.

5.5.4 Das Ergebnis

Nach Abarbeitung der einzelnen Einflussfaktoren aus Tabelle 1 sieht die Situation im RSD 1 Team nach Übergabe des Projektes EPX in der Zusammenfassung wie folgt aus:

Nr.	Einflussfaktor	Wichtigkeit	Werkzeuge	Anmerkung
1	Vorgehensmodell	Hoch	• Scrum • Jira	Erneute Evaluierung ab der Betreuung von drei Projekten.
2	Verantwortung	Mittel		
3	Plattform und Infrastruktur	Niedrig	• Puppet	Betreuung durch Operationsteam. Prozess zur Verbesserung der Zusammenarbeit mit dem Operationsteam gerade im Gange.
4	Verträge	Niedrig	• Jira • Cloud	Ablage im Wiki und in der Cloud ausreichend.
5	Know-how	Hoch	• Jira • Know-how Meetings	Evaluierung zusätzlicher Möglichkeiten zu Dokumentation und Know-how-Austausch im Sommer 2017.
6	CI-Tools	Niedrig	• Jira	Update der Jira Version birgt ein geringes Risiko, da die im Unternehmen eingesetzte Version um einige Versionen hinterherhinkt.
7	Versionskontrolle	Mittel	**Versionskontrolle:** • Jenkins **Builds:** • Ant (Gradle) • SourceTree	Vom Team übernommen. Builds müssen noch angepasst und erweitert werden. Derzeit sind zum Beispiel einige Builds fehlerhaft und werden nicht mehr benötigt.

8	Entwicklung	Niedrig	**Entwicklungs-umgebung:** • IntelliJ **Programmier-sprachen:** • Java • HTML / CSS • JavaScript **Datenbanken:** • Oracle	Es zeichnet sich derzeit kein Einsatz nicht bekannter Sprachen oder Datenbanken ab.
9	Feedback	Hoch	• Slack • Quartals-meetings	Evaluierung der Methoden mit Ende des Jahres 2017.
10	Testing	Hoch	• Selenium • Sonar	Zu wenig Tests bei der Übergabe vorhanden. TDD wird angestrebt.
11	Software Architektur	Mittel	• Jira	Dokumentation derzeit nur im Jira ersichtlich. Neue Dokumentation mittels Arc42 im Sommer 2017. Anschließend ist eine Evaluierung der Architektur geplant.
12	Style Guide	Niedrig	• Jira	Vorhanden und nicht komplex.
13	Coding Standard	Mittel	• Sonar	Erweiterungen in Bezug auf das Thema Sicherheit mittelbar notwendig.
14	Dokumentation	Hoch	• Jira	Weitere Möglichkeiten zu einfacherer Dokumentation werden gerade

				unternehmensweit evaluiert.
15	DOR	Niedrig	• Jira	Thema RQE wird unternehmensweit gerade neu betrachtet. Nach Präsentation der Ergebnisse Anpassungen im eigenen Team notwendig. Derzeitige Darstellung und Abarbeitung mittels Jira unzureichend. Vor allem in Bezug auf die angestrebten neuen Projekte ist die derzeitige Handhabung unzureichend.
16	DOB	Niedrig		
17	DOD	Mittel		
18	RQE	Niedrig		
19	Security	Niedrig	• Graylog	Thema noch zu wenig ausgeprägt. Know-how muss schnellstmöglich aufgebaut werden. Ausblick: Erweiterung der Testingstrategie und Coding Standards. Aufbau von Know-how für alle Rollen im Projekt.

Tabelle 2 – Projekt EPX im RSD1 Team

Zur besseren Darstellung der Gegebenheiten wird in Tabelle 2 auf die Zuordnung zu Kategorien verzichtet. Stattdessen wird die Tabelle um die Spalte „Anmerkungen" ergänzt. Diese Spalte enthält zusätzliche Informationen zu den Gegebenheiten der einzelnen Einflussfaktoren nach der Übergabe des Projektes EPX in das RSD 1 Team und ist farblich hinterlegt. Mit dieser farblichen Darstellung soll die Abdeckung der Einflussfaktoren nach Übernahme des Projektes veranschaulicht werden. Jene Einflussfaktoren, die rot markiert sind, konnten bei Übergabe nicht

ausreichend abgedeckt werden. Im Gegensatz dazu sind grün markierte Einflussfaktoren vollständig abgedeckt und weiterer Handlungsbedarf ist vorderhand nicht notwendig.

Zusammengefasst lässt sich sagen, dass die Projektübernahme anhand des entwickelten und in Tabelle 1 zusammengefassten Vorgehensmodelles problemlos möglich und erfolgreich war und zudem, wie aus Tabelle 2 ersichtlich, alle Einflussfaktoren abgedeckt werden konnten. Bei einigen dieser Einflussfaktoren ist jedoch mittelbar Handlungsbedarf erkennbar und es bleibt abzuwarten, welche Herausforderungen während der ersten Phasen der eigenständigen Entwicklung noch zu bewältigen sind.

Mit der erfolgreichen Übernahme eines bestehenden, agil umgesetzten Projektes in das RSD1 Team ist die eigentliche Forschungsarbeit abgeschlossen. Das abschließende Kapitel 6 beinhaltet eine Kurzzusammenfassung der zentralen Ergebnisse der vorliegenden Arbeit samt Conclusio sowie einen Hinweis und Ausblick auf offen gebliebene Fragen.

6 CONCLUSIO UND OFFENE FRAGEN

In dieser Arbeit mit dem Titel „Entwicklung eines Vorgehensmodelles zur Übergabe und weiteren Betreuung eines bestehenden, agil umgesetzten Projektes in einem neuen Team“ wird nach einer kurzen Einleitung zuerst ein allgemeiner Überblick über Vorgehensmodelle zur Entwicklung von Software sowie ein kurzer Abriss zu den klassischen bzw. agilen Methoden gegeben. Da der Fokus auf agilen Methoden und weiterführenden Techniken liegt, werden die agilen Methoden etwas detaillierter beschrieben und letztendlich der Ansatz von kontinuierlicher Auslieferung von Software ins Zentrum gerückt. Im Zuge dieser Beschreibung wird außerdem ein typischer Software-Lebenszyklus mit all seinen Phasen vorgestellt.

Aufbauend auf diesen theoretischen Grundlagen und deren Verständnis wird das neue Vorgehensmodell mittels Identifikation und Beschreibung der Einflussfaktoren anhand des Software-Lebenszyklus entwickelt und eine Risikobewertung der einzelnen Faktoren durchgeführt. Letztendlich wird das auf diese Weise generierte Vorgehensmodell umfassend und detailliert in einer übersichtlichen Tabelle dargestellt, die sämtliche identifizierten Einflussfaktoren inklusive Kategorisierung und möglicher Werkzeuge beinhaltet.

Daran anschließend wird nach knapper Beschreibung der aktuellen Gegebenheiten im Unternehmen Netconomy das neu entwickelte Vorgehensmodell für eine erste Übernahme eines Projektes herangezogen. Die Übernahme des Projektes wird entsprechend vorbereitet und die einzelnen Einflussfaktoren werden berücksichtigt. Durch den Einsatz dieses Modelles wird abschließend das erste Projekt erfolgreich in das neue RSD1 Team übernommen werden und anschließend weiterentwickelt und betreut.

Diese Arbeit beschäftigt sich darüber hinaus mit der Beantwortung der Forschungsfrage „Welche Aspekte sind als zentrale Einflussfaktoren zu berücksichtigen, wenn die Weiterentwicklung eines agil umgesetzten Projektes in einem anderen Team stattfinden soll, welches noch über keine bestehende Entwicklungsumgebung und nur begrenztes Know-how verfügt?“. Die in Kapitel 1.1 dazu formulierte Hypothese H1 kann nunmehr erfolgreich mit Ja beantwortet werden.

Nach erfolgreicher Projektübernahme lassen sich folgende Schlüsse ziehen:

- Es gibt zahlreiche Einflussfaktoren, die bei Projektübergaben berücksichtig werden müssen. Diese können sich je nach Unternehmung zusätzlich unterscheiden.
- Projektübergaben, die unter Außerachtlassung dieser Einflussfaktoren erfolgen, können funktionieren und haben in der Vergangenheit bereits funktioniert. Der weiterführende Erfolg des Projektes bleibt meistens jedoch aus und das Risiko des Scheiterns steigt.
- Bei der Übernahme eines Projektes ist man sehr stark auf das bereits bestehende Umfeld im Unternehmen bzw. des zu übernehmenden Projektes angewiesen. Eine Veränderung diverser Komponenten wie Vorgehensmodell, Programmiersprachen, usw. zum Zeitpunkt der Übernahme kann das Unternehmen zum Scheitern bringen.

- Diverse Risiken bei der Übernahme reduzieren sich durch bereits im Unternehmen bestehende Gegebenheiten. Weiterführend können diese aber zu einem höheren Risiko führen. Die Übernahme muss daher gut vorbereitet und durchdacht werden.
- Anhand des entwickelten Vorgehensmodells ist es einfach möglich, ein bestehendes, agil entwickeltes Projekt in ein neues Team zu übernehmen. Bereits vorhandenes Know-how erleichtert diese Übernahme enorm. Das Team selbst hat auf die erfolgreiche Übernahme ebenfalls entscheidenden Einfluss. Es empfiehlt sich daher, die Übernahme nicht auf eigene Faust zu versuchen, sondern laufend das gesamte Team miteinzubeziehen.
- Nicht alle Einflussfaktoren müssen gleich bei der Übergabe komplett abgedeckt werden. Es ist jedoch wichtig zu identifizieren und evaluieren, welche dieser Faktoren besonders großen Einfluss auf den erfolgreichen Weiterverlauf des Projektes haben.
- Auch wenn das neue Modell mit Blick auf die mögliche Vielfalt und Ausdifferenzierung unternehmerischer Gegebenheiten keine gesamthafte Aufstellung möglicher Einflussfaktoren liefern kann, so sichert doch schon die entsprechende Berücksichtigung der hier identifizierten Faktoren eine erfolgreiche Projektübergabe.

Offene Fragen

Alle in Kapitel 1.1 definierten Ziele dieser Arbeit konnten erreicht werden. Die erzielten Ergebnisse stellen eine solide Grundlage für eine erfolgreiche praktische Übernahme eines Projektes dar. Auf Grund ihrer immensen thematischen Bandbreite konnten zum Beispiel die Vorgehensmodelle sowie Werkzeuge im Rahmen dieser Arbeit zwar nicht im kompletten Umfang beschrieben oder gegenübergestellt werden. Zudem bietet das entwickelte neue Vorgehensmodell auch keine komplette Aufstellung aller möglichen Einflussfaktoren und einsetzbaren Werkzeuge. Die letztlich erstellte Liste bietet aber definitiv eine gute Grundlage und kann nach Bedarf jederzeit einfach erweitert werden.

Des Weiteren fehlt in dieser Arbeit, obwohl ein erstes Projekt bereits erfolgreich übernommen werden konnte, eine detaillierte Erfolgsmessung bzw. Risikoanalyse. Weitere Einsatzmöglichkeiten wie zum Beispiel für klassische Modelle wurde nicht in Erwägung gezogen.

Bei der Entwicklung des Modelles wurde von der Übernahme eines Projektes innerhalb des eigenen Unternehmens ausgegangen. Eine Einsatzmöglichkeit bei der Übernahme eines externen Projektes wurde fürs Erste bewusst ignoriert.

Ausblick

In einem halben Jahr wird das übernommene Projekt hinsichtlich Performance, Zufriedenheit der einzelnen Stakeholder, usw. untersucht werden. Zusätzlich wird bei der Übernahme weiterer Projekte die Einsatzmöglichkeit anderer agiler Vorgehensmodelle wie zum Beispiel Kanban untersucht werden. Die Liste mit identifizierten Einflussfaktoren und Werkzeugen wird dabei ständig aktualisiert und angepasst werden. Das bereits entwickelte Modell wird zudem für einen weiteren Einsatz, zum Beispiel die Übernahme von Projekten in den CSS vorbereitet. Darüber hinaus wird es um die detailliertere Anleitung zu einzelnen Werkzeugen erweitert werden.

Letztendlich soll das Vorgehensmodell nicht mehr als zwei im Unternehmen eingesetzte Werkzeuge umfassen, für die detaillierte Anleitungen vorzusehen sind.

Das bestehende Projekt soll noch im Jahr 2017 in Richtung kontinuierliche Auslieferung von Software erweitert werden. Ziel dieser Umstellung ist unter anderem, mehrmals in der Woche Software auf das Produktivsystem des Kunden ohne Downtime zu deployen. Für die notwendigen Anpassungen soll das bestehende Vorgehensmodell herangezogen und auf seine Erweiterbarkeit hin überprüft werden. Am Ende soll es möglich sein, ein Projekt einfach zu übernehmen und kontinuierlich Software auszuliefern.

Abschließendes Ziel ist es, mit diesem Vorgehensmodell auch Projekte, die nicht bereits im Unternehmen betreut werden, erfolgreich zu übernehmen. Ein erstes Projekt ist hier bereits in der Pipeline. Glückt die Übernahme des Projektes, wird das Vorgehensmodell unternehmensweit als Standard ausgerollt und auch für die Übernahme der Lösung für Klein- und Mittelbetriebe herangezogen.

ABKÜRZUNGSVERZEICHNIS

A	Andere
API	Programmierschnittstelle
BDD	Behavior Driven Development
CI	Continuous Integration
CR	Change Requests
CSS	Customer Service und Support
DOB	Definition of Bugs
DoD	Definition of Done
DOD	Definition of Done
DOR	Definition of Ready
IDE	Integrated Development Environment
IT	Informationstechnik, Informationstechnologie
KM	Konfigurationsmanagement
PM	Projektmanagement
PO	Product Owner
QM	Qualitätsmanagement
RQE	Requirements Engineering
RSD	Rapid Solution Development
SDQA	Software Development Quality Assurance
SE	Systementwicklung
SLA	Service Level Agreements
Solar	Solution Architects
TDD	Test-Driven Development
WIP	Work in Progress
XP	Extreme Programming

ABBILDUNGSVERZEICHNIS

Abbildung 1 – Anforderungen an ein Vorgehensmodell (Balzert, 1998) 6
Abbildung 2 – Vereinfachte Darstellung eines Vorgehensmodells (Heinrich, 2001) 8
Abbildung 3 – Überblick Vorgehensmodelle (Krcmar, 2011) 8
Abbildung 4 – Verbreitung phasenorientierter Modelle (Sandhaus, Berg, & Knott, 2014) 10
Abbildung 5 – Erweitertes Wasserfallmodell (Hansen & Neumann, 2009) 11
Abbildung 6 – Allgemeines V-Modell (Boehm, 1979) 13
Abbildung 7 – Spiralmodell (Balzert, 2011) 15
Abbildung 8 – Extreme Programming (XP) (Kleuker, 2009) 19
Abbildung 9 – Übersicht Scrum (Mitchell, 2016) 21
Abbildung 10 – Beispiel Kanban Board (Kniberg, 2016) 25
Abbildung 11 – Lebenszyklus einer Software (Humble & Farley, 2011) 27
Abbildung 12 – Deployment Pipeline (Humble & Farley, 2011) 33
Abbildung 13 – Vorteile von kontinuierlicher Auslieferung von Software (Wolff, 2016) 36
Abbildung 14 – Erweiterte Deployment Pipeline (Humble & Farley, 2011) 49
Abbildung 15 – Continuous Delivery vs. Continuous Deployment (Caum, 2016) 50
Abbildung 16 – Continuous Delivery (Humble & Farley, 2011) 52
Abbildung 17 – RSD1 Team 68
Abbildung 18 – Release Plan EPX 70
Abbildung 19 – Slack RSD1 71
Abbildung 20 – Workflow RSD1 72
Abbildung 21 – Dokumentation RSD1 73
Abbildung 22 – RQE RSD1 74
Abbildung 23 – Guidelines Development RSD1 75
Abbildung 24 – Hybris Build RSD1 76
Abbildung 25 – Jenkins RSD1 77

TABELLENVERZEICHNIS

Tabelle 1 – Einflussfaktoren entlang des Software-Lebenszyklus (xebialabs.com, 2017), (Toll, 2017), (resources.workable.com, 2017), (Wirdemann, 2011) 63
Tabelle 2 – Projekt EPX im RSD1 Team 80

LITERATURVERZEICHNIS

agilealliance.org. (06. Oktober 2016). *www.agilealliance.org*. Von https://www.agilealliance.org/agile101/practices-timeline/ abgerufen

Balzert, H. (1998). *Lehrbuch der Software-Technik: Software-Management, Software- Qualitätssicherung, Unternehmensmodellierung.* Spektrum.

Balzert, H. (2011). *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb.* Spektrum.

Baron, P., & Hüttermann, M. (2010). *Fragile Agile: Agile Softwareentwicklung richtig verstehen und leben.* Hanser.

Baumgartner, M., Klonk, M., Pichler, H., Seidl, R., & Tanczos, S. (2013). *Agile Testing. Der agile Weg zur Qualität.* Hanser.

Beck, K., & Andres, C. (2005). *Extreme Programming Explained - Second Edition.* Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., . . . Thomas, D. (06. Oktober 2016). *http://agilemanifesto.org/*. Von http://agilemanifesto.org/ abgerufen

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., . . . Thomas, D. (06. Oktober 2016). *http://agilemanifesto.org/*. Von http://agilemanifesto.org/iso/de/principles.html abgerufen

Bergin, J. (2001). *Coding at the Lowest Level. Coding Patterns for Java Beginners.* Pace University.

Boehm, B. W. (25. September 1979). Guidelines for Verifying and Validating Software Requirements and Design Specifications. *EURO IFIP 1979: Proceeding of the European Conference on Applied Information Technology of the Internet.*, S. 711-719.

Bray, I. (2002). *An Introduction to Requirements Engineering.* Addison-Wesley.

Caum, C. (09. November 2016). *https://puppet.com/*. Von https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff abgerufen

Crispin, L., & Gergory, J. (2009). *Agile Testing - A Practical Guide for Testers and Agile Teams.* Addison-Wesley Professional.

Dräther, R., Koschek, H., & Sahling, C. (2013). *Scrum - kurz & gut.* O´Reilly Verlag.

Duden.de. (02. Oktober 2016). *http://www.duden.de/*. Von http://www.duden.de/rechtschreibung/Modell abgerufen

Dustin, E., Rashka, J., & Paul, J. (1999). *Automated Software Testing - Introduction, Management and Performance.* Addison-Wesley Professional.

Engels, G., & Schäfer, W. (1989). *Programmentwicklungsumgebungen: Konzepte und Realisierung.* Vieweg+Teubner Verlag.

executionists.com. (01. Februar 2017). *www.executionists.com*. Von http://www.executionists.com/website-brand-style-guide/ abgerufen

Fairbanks, G., & Garlan, D. (2010). *Just Enough Software Architecture: A Risk-Driven Approach.* Marshall & Brainerd.

Freeman, S. (12. Februar 2017). *mockobjects.com*. Von http://www.mockobjects.com/ abgerufen

Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Booch, G. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

Grechenig, T., Bernhart, M., Breiteneder, R., & Kappel, K. (2010). *Softwaretechnik*. Pearson Studium.

Gregory, J., & Crispin, L. (2014). *More Agile Testing: Learning Journeys for the Whole Team.* Addison-Wesley Professional.

Hallerstede, S. H. (2013). *Managing the Lifecycle of Open Innovation Platforms.* Springer Gabler.

Hansen, H. R., & Neumann, G. (2009). *Wirtschaftsinformatik 1 - Grundlagen und Anwendung.* Lucius & Lucius Verlagsgesellschaft, UTB.

Heinrich, A. (2001). *Management von Softwareprojekten* . Oldenbourg Wissenschaftsverlag.

Hofmann, D. W. (2008). *Software-Qualität.* Springer.

Hruschka, P., Rupp, C., & Starke, G. (2009). *Agility kompakt - Tipps für erfolgreiche Systementwicklung.* Springer Spektrum.

Humble, J., & Farley, D. (2011). *Continuous Delivery.* Addison-Wesley Professional.

Hunt, A., & Thomas, D. (2003). *Der Pragmatische Programmierer.* Hanser.

Hybris. (20. November 2016). *https://www.hybris.com*. Von https://www.hybris.com/de/about-us abgerufen

Kleuker, S. (2009). *Grundkurs Software-Engineering mit UML.* Vieweg + Teubner.

Kniberg, H. (17. Oktober 2016). *https://www.crisp.se/.* Von https://www.crisp.se/gratis-material-och-guider/kanban abgerufen

Krahe, J. (2009). *Der „Duct Tape Programmer" - Mentalität als Einflussgröße des Software Engineerings.* Fachhochschule für Ökonomie und Management.

Krcmar, H. (2011). *Informationsmanagement.* Springer.

Kuhrmann, M. (05. Oktober 2016). *http://www.enzyklopaedie-der-wirtschaftsinformatik.de.* Von http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Vorgehensmodell/Spiralmodell abgerufen

Lacey, M. (2016). *The Scrum Field Guide: Agile Advice for Your First Year and Beyond.* Addison-Wesley.

Leopold, K., & Kaltenecker, S. (2013). *Kanban in der IT. Eine Kultur der kontinuierlichen Verbesserung schaffen.* Hanser.

Liggesmeyer, P. (2002). *Software-Qualität: Testen, Analyisieren und Verififzieren von Software.* Spektrum.

Mitchell, I. (11. November 2016). *https://en.wikipedia.org.* Von https://en.wikipedia.org/wiki/File:Scrum_Framework.png abgerufen

Murray, A. P. (2016). *The Complete Software Project Manager.* Wiley.

Netconomy. (20. November 2016). *https://www.netconomy.net.* Von https://www.netconomy.net/uber-uns/ abgerufen

opensamm.org. (12. Februar 2017). *http://www.opensamm.org/.* Von http://www.opensamm.org/ abgerufen

Osherove, R. (2009). *The Art of Unit Testing.* Manning Publications.

pc-magazin.de. (01. Februar 2017). Von http://www.pc-magazin.de/business-it/styleguide-entwickeln-erscheinungsbild-website-webseite-design-ci-2160694.html abgerufen

Pfetzing, K., & Rohde, A. (2009). *Ganzheitliches Projektmanagement.* Dr. Götz Schmidt.

Rau, K.-H. (2016). *Agile objektorientierte Software-Entwicklung.* Springer.

resources.workable.com. (12. Februar 2017). *https://resources.workable.com/.* Von https://resources.workable.com/blog/tracking-employee-morale abgerufen

Roock, S., & Wolf, H. (2016). *Scrum - verstehen und erfolgreich einsetzen.* dpunkt verlag.

Rouse, M. (29. Oktober 2016). *http://whatis.techtarget.com/.* Von http://whatis.techtarget.com/definition/binary-file abgerufen

Rubin, K. S. (2013). *Essential Scrum - A Practial Guide to the Most Popular Agile Process.* Addison-Wesley.

Sandhaus, G., Berg, B., & Knott, P. (2014). *Hybride Softwareentwicklung.* Springer.

Schmidt, R., & Dohle, H. (2009). *ITIL V3 umsetzen - Gestaltung, Steuerung und Verbesserung von IT-Services.* Symposion Publishing GmbH.

Schütz, A. (2014). *Agile Softwareentwicklung von Anfang an.* Wegtam UG.

sei.cmu.edu. (01. Februar 2017). *http://www.sei.cmu.edu.* Von http://www.sei.cmu.edu/architecture/start/glossary/community.cfm abgerufen

Spring.io. (20. November 2016). *https://projects.spring.io/.* Von https://projects.spring.io/spring-framework/ abgerufen

Stachowiak, H. (1973). *Allgemeine Modelltheorie.* Springer.

Taschner, A. (2013). *Business Cases. Ein anwendungsorientierter Leitfaden.* Springer.

Tian, J. (2005). *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement.* John Wiley & Sons.

Toll, W. (05. Februar 2017). *blog.profitbricks.com.* Von https://blog.profitbricks.com/top-integrated-developer-environments-ides/ abgerufen

Vogel, O., Arnold Ingo, Chughtai, A., Ihler, E., Kehrer, T., Mehlig, U., & Zdun, U. (2009). *Software-Architektur - Grundlagen - Konzepte - Praxis.* Springer.

Vogel-Heuser, B. (2003). *Systems Software Engineering.* Oldenbourg Industrieverlag.

Wieczorrek, H. W., & Mertens, P. (2011). *Management von IT-Projekten.* Springer.

wikipedia. (03. Oktober 2016). *https://de.wikipedia.org.* Von https://de.wikipedia.org: https://de.wikipedia.org/wiki/H%C3%B6here_Programmiersprache abgerufen

Wirdemann, R. (2011). *Scrum mit User Stories.* Hanser.

Wolff, E. (2016). *Continuous Delivery - Der pragmatische Einstieg.* dpunkt.verlag.

xebialabs.com. (05. Februar 2017). *www.xebialabs.com.* Von https://xebialabs.com/periodic-table-of-devops-tools/ abgerufen