Traceability von Anforderungen und Tests in agilen Softwareentwicklungsprojekten

Thomas Wolfenstetter, Jonas Zitzelsberger, Markus Böhm, Helmut Krcmar

Lehrstuhl für Wirtschaftsinformatik
Technische Universität München
Boltzmannstr. 3
85748 Garching b. München
thomas.wolfenstetter@in.tum.de
jonas.zitzelsberger@in.tum.de
markus.boehm@in.tum.de
krcmar@in.tum.de

Abstract: Eine wesentliche Herausforderung bei Softwareentwicklungsprojekten besteht oft in der Notwendigkeit, mittels eines strukturierten Testmanagements zu überprüfen, ob die Anforderungen sämtlicher Stakeholder hinreichend erfüllt werden. Um dieses Ziel zu erreichen, muss transparent sein, wie Anforderungen, Lösungskomponenten und Tests miteinander in Beziehung stehen. Diese Eigenschaft wird allgemein als Traceability bezeichnet. Um Traceability sicherzustellen, bedarf es einer umfassenden Spezifikation und kontinuierlichen Aktualisierung der Abhängigkeiten zwischen den Artefakten. In der Praxis werden jedoch zunehmend agile Entwicklungsmethoden eingesetzt, bei welchen die dokumentierte Spezifikation zugunsten der flexiblen Kommunikation zwischen Entwicklern und Anwendern in den Hintergrund rückt. Bisherige Ansätze zur Sicherstellung der Traceability beziehen sich häufig auf traditionelle, phasenorientierte Projektvorgehensweisen und lassen sich daher nur eingeschränkt auf den agilen Kontext übertragen. Wie Traceability in agilen Projekten sichergestellt werden kann, wurde bisher nicht ausreichend betrachtet. In dieser Arbeit wird daher untersucht, wie Traceability in agilen Softwareprojekten gewährleistet werden kann. Dazu werden die besonderen Herausforderungen im agilen Kontext anhand einer Fallstudie aus der Industrie analysiert. Anschließend wird ein Datenmodell vorgestellt, welches die relevanten Artefakte abbildet und miteinander verknüpft.

1 Motivation

Bei der Erstellung von individueller oder der Anpassung standardisierter Unternehmenssoftware für spezifische Problemstellungen muss das zu entwickelnde System meist nahtlos in eine bestehende Systemlandschaft eingefügt werden können. Zu diesem Zweck müssen die Anforderungen vieler verschiedener Anwender und anderer Interessensgruppen berücksichtigt werden. Da es sich hierbei oft um relativ abstrakte Anforderungen handelt, können diese zu Beginn eines Entwicklungsprojekts oft nicht genau spezifiziert werden [Gr11]. Oftmals werden die spezifischen Anforderungen erst im Laufe des Projekts ersichtlich und sind dabei ständigen Änderungen unterworfen [Hi02]. Dane-

ben werden solche Anforderungen im Verlauf eines Entwicklungsprojekts immer vielfältiger, zunehmend komplexer und ändern sich in schnelleren Zyklen [Sc08]. Um flexibel auf solche Änderungen zu reagieren und auch spät identifizierte Anforderungen noch umsetzen zu können, werden bei der Softwareentwicklung agile Vorgehensmodelle immer beliebter [Gh08]

Im Gegensatz zum phasenorientierten Vorgehen werden im agilen Kontext Individuen und deren Interaktion statt Prozesse und Tools fokussiert. Funktionierende Software steht vor verständlicher Dokumentation und die Fähigkeit, auf Änderungen reagieren zu können, steht vor der Verfolgung eines straffen Zeitplans [WC03]. Abgesehen von den kurzen Iterationszyklen und der daraus resultierenden Flexibilität, unterscheidet sich die agile Softwareentwicklung in weiteren Punkten von der phasenorientierten Entwicklung. Statt einer individuellen Zuweisung spezieller Projektrollen, agieren im agilen Kontext sich selbst organisierende Entwicklungsteams, innerhalb derer Rollen zwischen den einzelnen Personen getauscht werden können [NMM05]. Eine weitere, zentrale Eigenschaft der agilen Entwicklung ist die kontinuierliche, enge Zusammenarbeit der Entwickler untereinander und mit externen Interessensgruppen. Diese Zusammenarbeit manifestiert sich in zahlreichen Meetings entlang des gesamten Entwicklungsprozesses. Im Gegensatz zu phasenorientierten Vorgehensmodellen, bei welchen die Anforderungsspezifikation bereits in einem frühen Projektstadium fixiert und anschließend nur noch in Ausnahmefällen geändert wird, finden bei agilen Methoden ständige Neuplanungen auf Basis der aktuellen Problemstellungen und Erkenntnisse statt, um so flexibel auf Änderungen oder neu identifizierte Anforderungen reagieren zu können [Hi02].

Um dies zu erreichen, wird weitgehend darauf verzichtet, umfangreiche Spezifikationsdokumente, wie sie aus phasenorientierten Vorgehensmodellen bekannt sind, anzulegen und zu pflegen. Stattdessen setzen agile Methoden auf bilaterale Kommunikation zwischen den Projektbeteiligten und damit einhergehend, auf informellen Wissenstransfer [CH01]. Genau hier zeigt sich jedoch eine mögliche Schwachstelle der agilen Entwicklung. Weil der Fokus auf der Kommunikation zwischen den Projektbeteiligten liegt, werden Ergebnisse oftmals nicht strukturiert oder nur sehr oberflächlich dokumentiert. Diese oberflächliche Dokumentation führt jedoch im weiteren Verlauf des Projekts häufig zu Mehrarbeit und kostet damit zusätzliche Ressourcen.

Gerade bei Software, die eine zentrale Rolle im Geschäftsbetrieb eines Unternehmens einnimmt und daher vom ersten Tag an tadellos funktionieren muss, gilt es sicherzustellen, dass alle essentiellen Anforderungen identifiziert und umgesetzt wurden, bevor das System in den laufenden Betrieb integriert wird. Dies ist durch Testfälle für sämtliche Anforderungen zu verifizieren. Hierzu muss jedoch transparent sein, wie Anforderungen, Code und Testfälle miteinander in Beziehung stehen. Diese Fähigkeit, nämlich den Lebenszyklus einer Anforderung und deren Beziehungen zu anderen Artefakten der Softwareentwicklung zu erfassen, und damit nachvollziehen zu können, wird als Traceability bezeichnet. Traceability sollte sowohl vorwärts als auch rückwärts entlang des Entwicklungsprozesses gegeben sein - das heißt sowohl von der Quelle einer Anforderung über die Detaillierung und Spezifikation zu der letztendlichen Umsetzung der Anforderung und den korrespondierenden Testfällen als auch in der entgegengesetzten Richtung [GF94].

Traceability schafft Transparenz über die Umsetzbarkeit von Anforderungen sowie über die Zusammenhänge zwischen den Anforderungen untereinander und weiteren Entwicklungsartefakten, wie beispielsweise Tests. Hierdurch verbessert sich meist auch die Qualität der Lösung und deren Anpassbarkeit an Änderungen [LGJ03]. Durch das explizite Erfassen der Beziehungen und Abhängigkeiten zwischen den Artefakten kann jederzeit nachvollzogen werden, welche Anforderungen bereits umgesetzt wurden oder zu welchem Grad diese umsetzbar sind [RJ01]. Bei Anforderungsänderungen können, auf Basis der durch Traceability verfügbaren Informationen, die betroffenen Entwicklungsartefakte, wie beispielsweise bestimmte Softwarefeatures oder Testfälle, identifiziert und entsprechend angepasst werden [KS98]. Weiterhin können diese Informationen genutzt werden, um Lücken oder unklar formulierte Anforderungen aufzudecken oder sie können für das Projektcontrolling herangezogen werden [RE93]. Da sich insgesamt die Transparenz verbessert, können auch weitere Personen leichter in das Projekt integriert werden [Gh08]. Diese Transparenz sollte auch bei agilen Softwareentwicklungsprojekten gewährleistet sein. Wie die im folgenden Abschnitt dargestellte Analyse der existierenden Publikationen auf diesem Gebiet zeigt, beziehen sich bisherige Ansätze zur Gewährleistung von Traceability jedoch überwiegend auf phasenorientierte Projektvorgehensweisen. Lediglich vereinzelte Arbeiten befassen sich im Ansatz mit Traceability im agilen Kontext.

2 Methodik

Zur Entwicklung eines möglichst praxistauglichen Ansatzes für Traceability im agilen Kontext ist es von zentraler Bedeutung, die in der Realität auftretenden Herausforderungen genau zu kennen. So gestalten sich beispielsweise reale Entwicklungsprojekte meist anders als ursprünglich in der Theorie vorgeschlagen. In der Praxis erfolgt meist eine Anpassung an die Spezifika des Unternehmens sowie des Projekts. Die in dieser Arbeit angewandte Methodik richtet sich daher nach dem Design Science Ansatz [He07]. Bei diesem Ansatz werden relevante Konzepte in der wissenschaftlichen Theorie identifiziert und auf Herausforderungen angewendet, welche in der Praxis bestehen. In mehreren, schrittweisen Iterationen kann so ein möglichst praxisnaher Lösungsansatz für eine bestimmte Problemstellung gefunden werden.

Zu diesem Zweck wurde ein konkretes Beispielprojekt eines deutschen Industriekonzerns (Alpha), bei welchem ein Product-Lifecycle-Management System in die bestehende Softwarelandschaft integriert wurde, untersucht. Um existierende Methoden und Werkzeuge zur Gewährleistung der Traceability auf den agilen Kontext übertragen zu können, müssen Beziehungen zwischen den generischen Entwicklungsartefakten sowie weitere Herausforderungen der agilen Vorgehensweise untersucht werden. Zu diesem Zweck wurde die relevante Literatur nach dem Vorgehen von Webster und Watson [WW02] identifiziert. In mehreren Iterationen wurden durch eine stichwortbasierte Literaturrecherche relevante Publikationen zum Thema Traceability in agilen Entwicklungsprojekten identifiziert und ausgewertet, um deren Kernkonzepte auf das Fallbeispiel übertragen zu können.

Um die zentralen Herausforderungen zu erheben und die angewendete Scrum-Vorgehensweise im Detail zu verstehen, wurden semi-strukturierte Experteninterviews mit sieben Vertretern des Entwicklungsteams und externer Fachbereiche bei Alpha geführt. Anschließend wurden die Interviewtransskripte nach den Richtlinien von Gläser und Laudel [GL06] inhaltlich qualitativ ausgewertet. Hierdurch konnten der Entwicklungsprozess, die dabei entstehenden Artefakte sowie Lücken in der Traceability identifiziert und nachvollzogen werden. Besonderer Fokus lag dabei auf der Frage, welche Artefakte zu welchem Zeitpunkt der Entwicklung aus welchen Prozessschritten resultieren und für welche Schritte diese Artefakte wiederum den Input darstellen.

Auf Basis der in der Praxis identifizierten Herausforderungen sowie der Erkenntnisse aus der Literaturanalyse wurde ein konzeptuelles Datenmodell entwickelt, welches die Artefakte in agilen Entwicklungsprojekten miteinander in Beziehung setzt und die erforderlichen Relationstypen (Trace Links) beschreibt. Dieses wurde in Abstimmung mit den befragten Experten von Alpha in mehreren, iterativen Schritten bewertet und optimiert.

3 Bestehende Ansätze zu agiler Traceability

Die Analyse der Literatur zum Thema Traceability zeigt, dass sich der Großteil der untersuchten Publikationen auf phasenorientierte Entwicklungsmodelle bezieht. Hinsichtlich der agilen Entwicklungsmethodik existieren hingegen bisher nur vereinzelte Ansätze aus einer relativ generischen Perspektive.

Bouillon et al. [Bo13a] beschreiben einen leichtgewichtigen Ansatz zur Gewährleistung von Traceability in agilen Entwicklungsprozessen. Der Fokus dieser Arbeit liegt in erster Linie darauf zu zeigen, dass Traceability auch im agilen Kontext wichtig ist sobald das Projekt eine kritische Komplexität aufweist. Hierbei werden wichtige Entwicklungsartefakte angeführt. Insgesamt gehen die Autoren jedoch nicht darauf ein, wie die Beziehungen zwischen diesen Artefakten im Detail aussehen und welche Informationen benötigt werden, um die dargestellten Abhängigkeiten von Anforderungen und Testfällen nachvollziehen zu können.

Ghazarian [Gh08] vertritt den Standpunkt, dass gewisse Arten von Anforderungen immer wieder zu ähnlichen Softwareentwurfsmustern führen. Vor diesem Hintergrund wird die Möglichkeit diskutiert, Traceability von Anforderungen dadurch sicherzustellen, dass die Struktur des Software Codes gewissen Regeln unterliegt. Dieser Ansatz eignet sich jedoch nur für sehr detaillierte Anforderungen, die bereits in der Sprache des Entwicklers beschrieben sind. Zudem wird Traceability zwischen Anforderungen und Testfällen durch diesen Ansatz nicht abgedeckt.

Ein weiterer Ansatz zur Gewährleistung von Traceability in agilen Projekten beschäftigt sich mit der Herausforderung, wie Anforderungen, die oft nur informell in Meetings diskutiert und nicht formal spezifiziert werden, explizit bestimmten Softwarefeatures zugeordnet werden können. Hierzu wird ein Werkzeug vorgestellt, mit dessen Hilfe Verknüpfungen zwischen Abschnitten aus Meeting Transskripten und zu erstellenden Software-

features angelegt und verwaltet werden können [LGJ03]. Dieser Ansatz ermöglicht jedoch lediglich die Identifikation zu implementierender Features und ist somit der Anforderungserhebung zuzuordnen. Im Gegensatz dazu liegt der Fokus dieser Arbeit auf der Abbildung des gesamten, agilen Entwicklungsprozesses und somit auf der Nachvollziehbarkeit ausgehend von Anforderungen über die Implementierung bis hin zu den Testfällen. Nur durch diese übergreifende Perspektive kann das volle Potenzial der Traceability realisiert werden [WGK13].

Espinoza und Garbajosa [EG11] beschreiben ein ausführliches, konzeptuelles Datenmodell zur Sicherstellung der Traceability in Abhängigkeit vom jeweiligen Projektkontext. Dabei wird jedoch nicht aufgezeigt, inwiefern die durch die Traceability gewonnenen Erkenntnisse verwendet werden können, um die Auswirkungen von Anforderungsänderungen zu antizipieren. Ebenso versuchen die Autoren, einen eher generischen, von der agilen Entwicklungsmethodik unabhängigen, Ansatz zu definieren, der zudem stark von phasenorientierten Entwicklungsansätzen beeinflusst wird. Hierbei werden Anforderungen, Code und Tests unabhängig voneinander betrachtet.

Die hier diskutierten Publikationen sollen den Fokus und die Perspektiven auf Traceability in agilen Projekten ausreichend widerspiegeln, um die offenen Herausforderungen aufzuzeigen. Ein Anspruch auf Vollständigkeit soll jedoch hier nicht erhoben werden. Ziel der vorliegenden Arbeit ist es, basierend auf der Analyse eines Fallbeispiels aus der Praxis, ein konzeptuelles Datenmodell für Traceability in agilen Projekten abzuleiten. Ein zentraler Punkt hierbei ist, zu analysieren, welche Artefakte und Informationen zu welchen Zeitpunkten während der Entwicklung entstehen [HDJ09] und wie diese genutzt werden können, um Änderungen zu antizipieren und Projektressourcen zu schonen. Dies betrifft beispielsweise die Detaillierung der groben Anwendungsfälle, aus denen detailliertere Anforderungen (User Stories) entstehen. Weiterhin wird in dieser Arbeit Traceability sehr nahe am agilen Entwicklungsprozess beschrieben, wodurch es beispielsweise möglich wird, für die Implementierung fehlende Anforderungen oder gewisse Inkonsistenzen zu identifizieren.

4 Fallstudie: Agile Softwareentwicklung bei Alpha

Inhalt des untersuchten, agilen Entwicklungsprojekts ist die Anpassung einer kommerziellen Product-Lifecycle-Management Systems eines Drittanbieters und dessen Integration in die bestehende Softwarelandschaft von Alpha. Alpha ist ein multinationaler Industriekonzern mit Sitz in Deutschland, der mechatronische Produkte entwickelt, produziert und vertreibt. Der betroffene Anwenderkreis umfasst mehrere tausend Personen aus verschiedenen Abteilungen und an unterschiedlichen Standorten.

4.1 Der agile Entwicklungsprozess

Im Unterschied zu den phasenorientierten Vorgehensmodellen, bei denen die eigentliche Implementierung erst nach einer detaillierten Spezifikationsphase beginnt, setzt der agile Entwicklungsprozess bei Alpha auf kontinuierliches Prototyping. Hierbei wird in kurzen

Iterationszyklen (Sprints) die zu entwickelnde Lösung schrittweise erweitert. Den Anwendern wird dabei regelmäßig ein aktueller, lauffähiger Prototyp zur Verfügung gestellt. So können diese prüfen, ob ihre Anforderungen richtig verstanden und umgesetzt wurden. Anschließend können gegebenenfalls Anforderungen weiter detailliert oder angepasst werden. Abbildung 1 illustriert den rekonstruierten Prozessablauf bei Alpha und zeigt, welche Artefakte durch die zyklisch ablaufenden Prozessaktivitäten entstehen und bei welchen Aktivitäten diese verwendet werden.

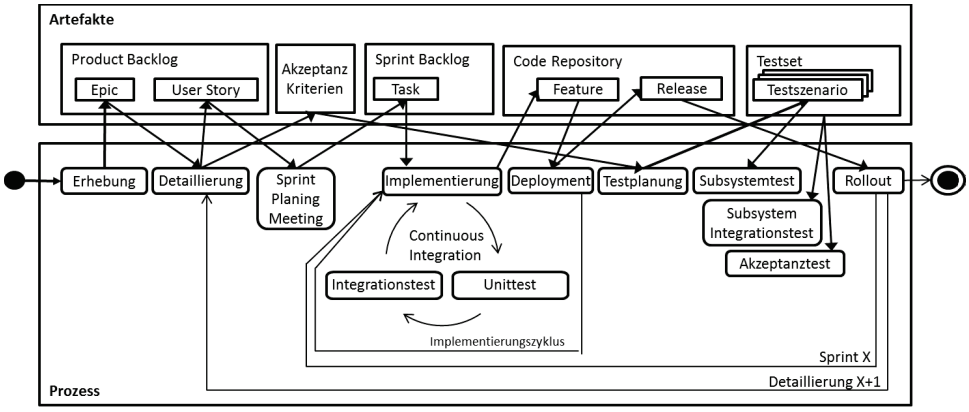

Abbildung 1: Schematischer Ablauf des agilen Entwicklungsprozesses bei Alpha

Zunächst werden übergreifende Anwendungsfälle (Epics) initial erhoben und im Product Backlog aufgelistet. Diese werden zu Beginn in sehr grober Form erhoben und nicht im Detail spezifiziert. Epics werden textuell, nach folgendem Muster definiert:

Als <*Benutzerrolle*> will ich <*das Ziel*>[, sodass <*Grund für das Ziel*>].

Beispiel: Als *Ingenieur* will ich *einen Testfall mit entsprechenden Werten hinterlegen*, so dass *ich spezielle Konfigurationen testen kann*.

Die spezifizierten Epics werden in einem Product Backlog aufgelistet. Für die Reihenfolge der Einträge in dem Product Backlog ist der Product Owner verantwortlich. Der Product Owner vertritt die fachliche Sicht auf die Software und ist sowohl für das Entwicklungsteam, als auch für das Management des Product Backlogs verantwortlich. Er entscheidet ebenso, welche User Stories in das Sprint Backlog übernommen werden. Bevor dies geschieht, müssen die Epics detailliert werden. Hieraus ergeben sich User Stories, welche ebenfalls textuell, durch das oben beschriebene Muster, allerdings deutlich detaillierter spezifiziert, und durch Akzeptanzkriterien konkretisiert werden. Anhand dieser, kann am Ende eines Sprints beurteilt werden, ob eine User Story erfolgreich umgesetzt wurde. Die Akzeptanzkriterien, welche zudem die Testfälle einer User Story darstellen, werden nach der Methodik „Specification By Example“ [Ad11] definiert.

In sogenannten Sprint Planning Meetings werden die, durch den Product Owner priorisierten und detaillierten, User Stories durch das Entwicklungsteam in feingranulare

Tasks zerlegt, welche im Sprint Backlog aufgelistet werden. Diese Tasks stellen die wesentlichen Arbeitspakete dar, welche anschließend durch das Entwicklungsteam in einem Sprint umgesetzt werden. Ein Sprint dauert maximal 30 Kalendertage und stellt den Kern der agilen Scrum-Methode bei Alpha dar.

Der Entwickler implementiert einen spezifischen Task und führt anschließend einen Unittest durch. Tritt ein Defect auf, muss der Entwickler den Fehler suchen und den Softwarecode anpassen. Dies wird so lange wiederholt, bis im Unittest kein Defect mehr auftritt. Bei der Implementierung entsteht so auf Basis des Tasks ein erstes Softwarefeature.

Nachdem der Unittest fehlerfrei verlaufen ist, führt der Entwickler den Integrationstest durch. Hierbei wird versucht, den soeben implementierten Task mit den bereits vorher umgesetzten Tasks zu integrieren. Hierbei wird das Zusammenspiel der isoliert voneinander fehlerfreien Tasks getestet. Tritt ein Defect auf, muss der zugehörige Code angepasst und auf dessen Basis ein erneuter Unittest durchgeführt werden. Erst danach kann der Task wieder integriert werden.

Nachdem auch im Integrationstest kein Defekt mehr auftritt, kann der Entwickler die implementierten Features in die Testumgebung ausrollen. Dieser Implementierungszyklus wird so lange wiederholt, bis keine Tasks mehr vorhanden sind. Die implementierten Softwarefeatures werden anschließend schrittweise in der Testumgebung zu einem Release integriert. Das kontinuierliche Zusammenfügen der einzelnen Features zu einem Release bzw. das automatische Testen des ganzen Systems bis zum aktuellen Entwicklungsstand, entspricht dem Prozess der „Continuous Integration“ [Fo06]. Dieser wird durch das Einbinden des Codefragments in das Repository automatisch ausgelöst.

Nach Abschluss des Implementierungszyklus sowie der Entwicklertests werden in der Testumgebung, durch ausgewiesene Tester, manuelle Tests durchgeführt. Hierzu werden Akzeptanzkriterien zu Testszenarien gebündelt, auf deren Basis Subsystemtests durchgeführt werden. Eine weitere Testebene stellen die Subsystem Integrationstests dar. Hierfür werden mehrere Testszenarien zu einem Testset gebündelt. Ebenso werden auf Basis der Testsets Akzeptanztests für die letztendliche Abnahme in den laufenden Betrieb durchgeführt. Nachdem diese Tests erfolgreich abgeschlossen sind, kann das Release in die Produktivumgebung ausgerollt werden.

Sind nach dem Ausrollen noch User Stories in dem Product Backlog vorhanden, können diese in einem weiteren Sprint X implementiert werden. Damit jedoch immer ausreichend detaillierte User Stories vorhanden sind, muss die Detaillierung der initial erhobenen Epics immer einen weiteren Sprint (X+1) vor deren Umsetzung stattfinden.

4.2 Herausforderungen für Traceability im agilen Entwicklungsprozess

Viele der in der Literatur existierenden Traceability Ansätze basieren auf detaillierten Anforderungsspezifikationsdokumenten. Diese Dokumente sind charakteristisch für phasenorientierte Vorgehensmodelle. Aus diesem Grund eignen sich existierende Traceability Ansätze vor allem für solche Entwicklungsmethoden. Abbildung 2 illustriert, welche

Trace Links (1-7) zwischen den Artefakten im Fallbeispiel des agilen Entwicklungsprozesses bei Alpha nicht dokumentiert werden. Wie sich aus den durchgeführten Experteninterviews zeigt, ergeben sich dadurch die nachfolgend diskutierten Herausforderungen.

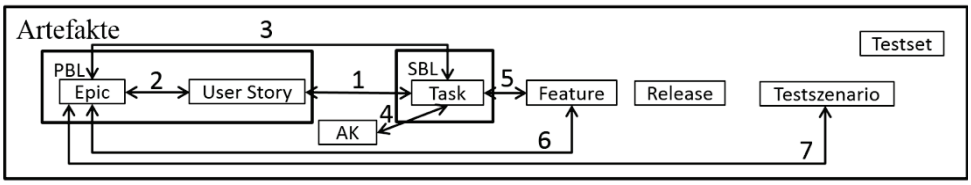

Abbildung 2: Herausforderungen für Traceability im agilen Entwicklungsprozess

Durch die informelle Anforderungsspezifikation im agilen Kontext, welche die direkte Kommunikation zwischen Fachbereich (Anwender bzw. Anforderungssteller) und Entwickler statt einer detaillierten Anforderungsspezifikation vorsieht, lassen sich bisherige Ansätze nur eingeschränkt auf agile Modelle anwenden. Dies betrifft vor allem den Prozessschritt der Detaillierung, bei dem die initial erhobenen Epics in einem Workshop zwischen Fachbereich und Anforderungsmanager verfeinert werden. Die Anforderungsmanager haben anschließend eine detaillierte Vorstellung über die Bedürfnisse des Fachbereichs und können auf dieser Basis die Akzeptanzkriterien für jede User Story festlegen. Dies geschieht oftmals in einem Workshop mit dem Entwicklungsteam. Durch diese Workshops und die direkte Kommunikation zwischen den beteiligten Rollen, wird eine detaillierte Dokumentation auf den unterschiedlichen Abstraktionsebenen oft als überflüssig erachtet (2).

Bei phasenorientierten Entwicklungsmethoden ist das Resultat der Spezifikationsphase ein vollständiges Anforderungsprofil. Im Gegensatz zu diesem Vorgehensmodell, existiert im agilen Kontext ein solches Anforderungsprofil nicht. Erst nachdem das Projekt abgeschlossen ist und alle Anforderungen in das Softwareprodukt umgesetzt worden sind, existiert eine vollständige, detaillierte Spezifikation der Anforderungen. Der Grund hierfür ist, dass bei der agilen Entwicklung immer nur die Anforderungen detailliert werden, welche anschließend in einem Sprint implementiert werden. Somit ist Traceability, wenn auch nur eingeschränkt, innerhalb der Sprints und zu den bereits umgesetzten Anforderungen sichergestellt, nicht jedoch übergreifend über alle bzw. zu den noch ausstehenden Anforderungen des Softwareprojekts. Trace Links zu Anforderungen, welche zwar bereits erhoben, aber noch nicht detailliert wurden, existieren nicht. Bei industriellen Großprojekten mit einer langen Laufzeit besteht die Gefahr, dass Inkonsistenzen zwischen den Anforderungen der einzelnen Sprints auftreten. Dieses Problem betrifft auch phasenorientierte Entwicklungsmethoden, bei denen in der Spezifikationsphase Anforderungen seitenweise detailliert werden, wodurch die Übersichtlichkeit meist leidet. Dennoch ist bei solchen Entwicklungsmethoden, aufgrund der detaillierteren Dokumentation, Traceability gewährleistet. So ist es im Nachhinein möglich, Redundanzen, hinsichtlich einzelner User Stories oder Tasks bzw. Inkonsistenzen, zu erkennen (1, 2, 3).

Abgesehen von Redundanzen erschwert eine übermäßige Detaillierung, den Überblick über alle feingranularen Tasks zu behalten. In der Praxis ist es den Projektmitgliedern

aufgrund des Zeitdrucks oft nicht möglich, für jeden einzelnen Task, Trace Links manuell zu dokumentieren und diese fortlaufend zu aktualisieren (2, 3).

Aufgrund fehlender Trace Links zwischen einzelnen Artefakten, die in unterschiedlichen Tools dokumentiert und verwaltet werden, ergeben sich meist weitere Probleme. Im untersuchten Fall werden beispielsweise die initial erhobenen Epics und die zugehörigen detaillierten User Stories in Form von Use Case Diagrammen sowie die von einer User Story abgeleiteten Tasks in unterschiedlichen Tools verwaltet. Aufgrund dieses Bruches kann nach Änderungen nur durch manuelle Prüfung nachvollzogen werden, welche Tasks zu einer bestimmten Epic gehören. Wegen fehlender Schnittstellen zwischen diesen Tools, kommt es bei Änderungen somit häufig zu Inkonsistenzen (1, 3). In der Folge bedarf es für die Sicherstellung übergreifender Traceability eines erheblichen, manuellen Aufwands.

Akzeptanzkriterien werden für User Stories definiert und stellen zugleich die Testfälle für diese dar. Diese Testfälle besitzen jedoch keine Beziehungen zu den abgeleiteten Tasks, welche die konkreten Arbeitspakete für die Implementierung darstellen. Zur Validierung spezifischer Akzeptanzkriterien können die dafür benötigten Tasks nicht identifiziert werden (4). Wie bereits angesprochen, werden die Tasks in einem separaten Tool für die agile Entwicklung dokumentiert. Nach der Implementierung eines Tasks, muss jedoch, beim Einbinden in das Coderepository, die zugehörige User Story angegeben werden. Somit kann nicht eindeutig nachvollzogen werden, welches Feature aus welchem Task resultiert (5). Nach der Implementierung muss manuell geprüft werden, ob bestimmte Epics vollständig umgesetzt sind. Dennoch kann nur schwer nachvollzogen werden, welche spezifischen Features, bzw. welche Codefragmente zu einer Epic gehören (6). Die Testszenarien für eine User Story werden in einem Testmanagementtool spezifiziert. Innerhalb dieses Tools können jedoch die einzelnen User Stories nicht zu übergreifenden Epics zusammengefasst werden. Somit kann nicht nachvollzogen werden, welche Testszenarien, welche Epics adressieren, bzw. welche Testszenarien zur Validierung einer spezifischen Epic benötigt werden (7).

5 Ein konzeptuelles Datenmodell für Traceability in agilen Projekten

Um den in Abschnitt 4.2 diskutierten Herausforderungen zu begegnen, ist es notwendig, die bei der agilen Entwicklung entstehenden Artefakte, sowie die notwendigen Trace Links zwischen diesen, strukturiert abzubilden. Hierzu wurden die in der Literatur identifizierten Konzepte auf den, in der Fallstudie analysierten, agilen Entwicklungsprozess übertragen und in fünf Iterationsschritten durch einen intensiven Austausch mit Alpha evaluiert und erweitert. Das in Abbildung 3 dargestellte Klassendiagramm stellt das resultierende, konzeptionelle Datenmodell für Traceability in agilen Projekten dar.

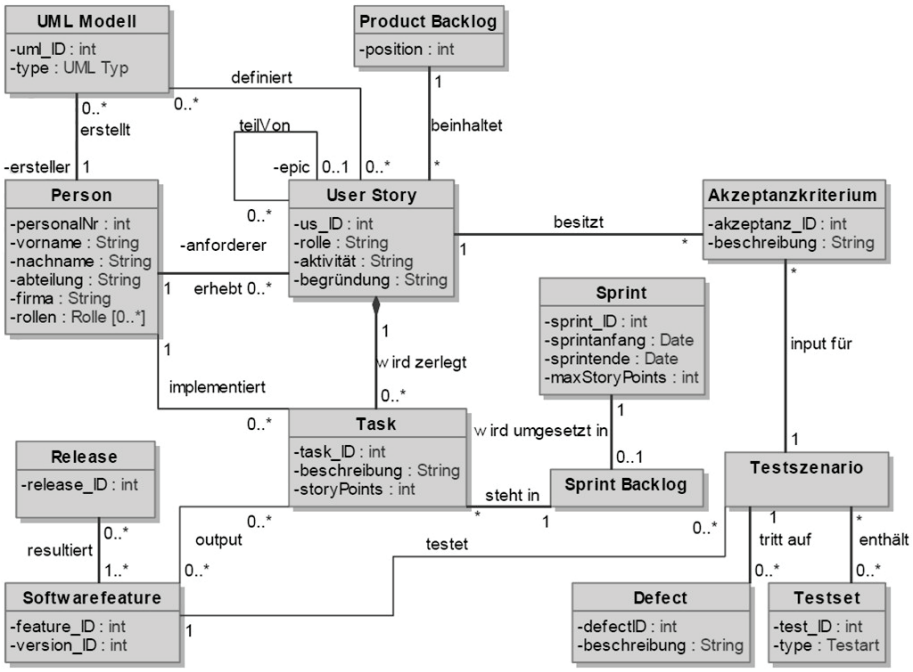

Abbildung 3: Konzeptuelles Datenmodell für Traceability in agilen Projekten

Eine *User Story* wird von einer *Person*, welche in dieser Beziehung die Rolle eines Anforderungsstellers einnimmt, erhoben. Eine *Person* kann dabei auch mehrere Rollen, annehmen. Durch die Beziehung *erhebt* kann nachvollzogen werden, welche Person, welche User Stories erhoben hat.

Bei der Klasse *User Story* handelt es sich um fachliche Rollen, wie beispielsweise einen Werkstattmeister. Diese dürfen nicht mit den Rollen der Klasse *Person* verwechselt werden. Eine *User Story* wird durch das textuelle Muster beschrieben und stellt somit eine Metaebene dar. Die Beziehung *teilVon* stellt die Granularität von User Stories dar. Initial erhobene User Stories werden durch grobe Epics repräsentiert, welche bei der Detaillierung in mehrere, kleinere User Stories zerfallen. Anhand dieser Beziehung kann nachvollzogen werden, welche detaillierten *User Stories* welchem *Epic* zugeordnet werden können.

Die exakte Definition einer *User Story* findet in einem *UML Modell* statt. Beispielsweise können in einem *Use Case Diagramm* mehrere *User Stories* definiert werden und eine *User Story* kann durch mehrere Diagrammtypen definiert sein. Durch die Beziehung *erstellt*, kann diejenige *Person* identifiziert werden, welche das *UML Modell* erstellt hat. Anhand der Beziehung *definiert* kann nachvollzogen werden, welche *User Story*, in welchem *UML Modell* definiert ist. Alle *User Stories* werden im *Product Backlog* dokumentiert. Das Attribut *Position* bezieht sich auf die Reihenfolge, in welcher die *User Stories* im *Product Backlog* stehen.

Bei der Detaillierung werden für eine *User Story* mehrere *Akzeptanzkriterien* festgelegt. Aufgrund der Methodik Specification By Example stellen die *Akzeptanzkriterien* zugleich die Testfälle einer *User Story* dar. Durch den Trace Link *besitzt* können die, zu einer spezifischen *User Story* zugehörigen, *Akzeptanzkriterien* identifiziert werden.

Das Entwicklungsteam zerlegt eine *User Story* in mehrere, noch detailliertere *Tasks*. Ein *Task*, welcher eine gewisse Anzahl an Story Points aufweist, wird von einer *Person* mit der Rolle Entwickler implementiert. Durch den Trace Link *wird zerlegt* können die, zu einer spezifischen *User Story* zugehörigen, *Tasks* identifiziert werden.

Die *Tasks* stehen in einem *Sprint Backlog*, das in einem *Sprint* abgearbeitet bzw. umgesetzt wird. Pro *Sprint* kann eine bestimmte Anzahl an Story Points umgesetzt werden. Über das zwischengeordnete Artefakt *Sprint Backlog* kann implizit nachvollzogen werden, welche *Tasks*, und somit welche *User Stories*, in welchem *Sprint* umgesetzt werden.

Nach der Implementierung eines *Tasks* können als Output mehrere *Softwarefeatures* resultieren. Durch die Versionierung können Änderungen innerhalb eines Features, anhand der verschiedenen Versionen nachvollzogen werden. Anhand des Trace Links *output*, welcher die Implementierung darstellt, kann nachvollzogen werden, welches *Softwarefeature* welche *Tasks* adressiert. Ein *Softwarefeature* resultiert in mindestens einem *Release*. Durch die Beziehung *resultiert* ist die Nachvollziehbarkeit der *Softwarefeatures* in das resultierende *Release* gewährleistet.

Ein *Softwarefeature* kann von mehreren *Testszenarien* getestet werden, welche von einer *Person* mit der Rolle Tester durchgeführt werden. Ein *Testszenario* hat als Input mehrere *Akzeptanzkriterien*. Durch den Trace Link *Input für* kann nachvollzogen werden, welche einzelnen *Akzeptanzkriterien* in welchem *Testszenario* gebündelt werden. Ein *Testszenario* kann in keinem, aber auch in mehreren *Testsets* enthalten sein. Ein *Testszenario* wird sowohl für das *Testset* des System Integrationstests, als auch für das *Testset* der Akzeptanztests benötigt. Hingegen enthält ein Testset meistens mehrere Testszenarien. Durch die Beziehung *testet* kann nachvollzogen werden, welches *Testszenario* welches spezifische *Softwarefeature* testet.

Zu einem Testszenario können mehrere *Defects* auftreten. Anhand des Trace Links *tritt auf* kann nachvollzogen werden, zu welchem spezifischen *Testszenario* ein *Defect* aufgetreten ist.

6 Diskussion

Etliche der in Abschnitt 4.2 angesprochenen Probleme resultieren aufgrund von Brüchen in der zugrundeliegenden Toollandschaft. Diese Tools sind auf Unternehmensentscheidungen für die einzelnen Bereiche, wie beispielsweise das Anforderungsmanagement oder das Testmanagement, zurückzuführen. Hierbei wäre es für agile Projekte besser, wenn solche Brüche zwischen den einzelnen Tools, welche während der Entwicklung eingesetzt werden, vermieden werden. Innerhalb der einzelnen Tools ist Traceability sichergestellt - nicht jedoch toolübergreifend. Somit lassen sich Anforderungen nicht über

deren kompletten Lebenszyklus hinweg konsistent nachverfolgen. Bei geeigneten Schnittstellen könnten auch toolübergreifende Trace Links identifiziert werden. Diese globale Nachvollziehbarkeit ist, vor allem bei Änderungen in großen, unübersichtlichen Projekten, von Vorteil, da aufgrund dieser Links, alle weiteren Entwicklungskomponenten angepasst werden können [LQ10].

Durch die Sicherstellung von Traceability zu den atomaren Tasks können Redundanzen bereits vor deren Umsetzung erkannt werden. Zudem wird, durch die Rückverfolgung der Trace Links zu redundanten, aber bereits implementierten Tasks, die Wiederverwendung bereits implementierter Features möglich. Ein bereits implementierter Task kann komplett oder zum Großteil für den neuen Task wiederverwendet werden. Somit wird es durch Traceability möglich, den Implementierungsaufwand zu verringern und dadurch schneller und kosteneffizienter das Sprint- bzw. Projektziel zu erreichen. Abgesehen hiervon kann mit Hilfe des in Kapitel 5 beschriebenen konzeptionellen Datenmodells und des resultierenden Beziehungswissens der agile Entwicklungsprozess in weiteren Punkten optimiert werden. Diese Optimierung findet auf Basis geeigneter Anwendungsszenarien für Traceability aus der Praxis statt [Bo13b].

Bei der initialen Erhebung weist die grobe User Story den Status einer Epic auf. Bei der späteren Detaillierung werden die Epics in detailliertere User Stories zerlegt. Durch den Trace Link zwischen den verschiedenen Versionen einer User Story kann nachvollzogen werden, welche detaillierten User Stories aus welcher Epic resultieren.

Da ein Task nur von einem Entwickler umgesetzt wird, kann durch Traceability nachvollzogen werden, welcher Task von welchem Entwickler implementiert wird. Parallel zur Implementierung eines spezifischen Tasks in einem Sprint kann der Product Owner, mit Hilfe der Trace Links, den Implementierungsstatus dieses Tasks prüfen. Dies ist aufgrund der Beziehungen zwischen Entwickler, Task und Sprint Backlog möglich. Der Product Owner kann nachverfolgen, welcher Entwickler, zu welchem Zeitpunkt des Sprints einen spezifischen Task aus dem Sprint Backlog implementiert.

Nachdem einige User Stories bzw. die zugehörigen Tasks erfolgreich in ein Release umgesetzt wurden, kann der Product Owner den Projektfortschritt anhand dieser implementierten User Stories ableiten. Dies ermöglichen die Beziehungen zwischen den User Stories und dem Product Backlog, in dem alle Anforderungen aufgelistet sind. Bei der Implementierung werden die zugehörigen User Stories von dem Product Backlog in das Sprint Backlog übernommen, wodurch es dem Product Owner ermöglicht wird, anhand der restlichen User Stories im Product Backlog den Projektfortschritt abzuleiten.

Hat sich eine bereits implementierte User Story geändert, können aufgrund des Trace Links zwischen User Story und Task, die zu der geänderten User Story zugehörigen Tasks identifiziert und entsprechend modifiziert werden.

Für die Navigation mittels Trace Links zwischen Spezifikation, Design, Code und Test müssen diverse Mappings der eindeutigen IDs stattfinden. Die exakte Spezifikation einer textuellen User Story, findet in einem UML Modell statt. Hierzu muss zur Gewährleistung der Nachvollziehbarkeit zwischen Spezifikation und Design Artefakten die ID des

Modells mit der textuell spezifizierten User Story verknüpft werden. Für die Nachvollziehbarkeit zwischen Design und Code muss die ID der modellierten User Stories mit den zugehörigen Tasks verknüpft werden. Diese werden von einem Entwickler in Softwarefeatures umgesetzt. Somit ergibt sich über zugehörigen Tasks implizit Traceability zwischen Designartefakten und Code. Nach der Implementierung eines Tasks werden die resultierenden Softwarefeatures in das Repository übertragen, wobei jeweils eine neue Version entsteht. Zur Gewährleistung von Traceability zwischen Code und Test muss nun die neu entstandene Versions ID mit der ID der Entwicklertests verknüpft werden. Dadurch kann nachvollzogen werden, bei welcher Version eines Softwarefeatures, der zugehörige Test erfolgreich bestanden wurde.

Nachdem eine User Story an den Anfang des Product Backlog platziert wurde, prüft der Product Owner, ob diese mit einer bereits implementierten User Story in Konflikt steht. Diese Prüfung wird durch die Traceability einer User Story über die Tasks hin zu den Softwarefeatures möglich. Der Product Owner kann somit nachvollziehen, ob für die spezifische User Story bereits Softwarefeatures existieren. Besteht ein Konflikt, kann anhand der Trace Links der zugehörigen Stakeholder, welcher die User Story erhoben hat, identifiziert und über den Konflikt benachrichtigt werden.

Besteht kein Konflikt, kann der Product Owner die spezifische User Story bewerten. Hierzu bedient er sich Trace Links zu ähnlichen, bereits umgesetzten User Stories, wodurch er den Umfang für die aktuelle User Story abschätzen kann. Ist sie zu umfangreich, wird sie in detailliertere User Stories zerlegt. Nach diesem Prozess, bei dem jede zerlegte User Story versioniert wird, können dadurch diese zu übergeordneten User Stories bzw. Epics zurückverfolgt werden. Zusätzlich wird es durch die Zerlegung und der Trace Links zu bereits umgesetzten Softwarefeatures möglich, redundante User Stories zu identifizieren und diese zu entfernen.

Kann ein Defect eigenständig behoben werden, lokalisiert der Entwickler anhand der Versions ID der zuvor eingecheckten Softwarefeatures, den zugehörigen Code und kann diesen modifizieren. Kann der Defect aufgrund der Komplexität nicht eigenständig behoben werden, muss zur Unterstützung der Stakeholder herangezogen werden, der die zugehörige User Story erhoben hat. Diese Rückverfolgung ist mittels der Trace Links zwischen Softwarefeature und Task, zwischen Task und User Story, sowie zwischen User Story und Person gewährleistet.

Durch Traceability kann überprüft werden, ob alle Anforderungen der Stakeholder in dem Softwareprodukt vorhanden sind, bzw. ob diese deren Vorstellungen entsprechen. Auch können redundante Testfälle identifiziert werden, wodurch Ressourcen und Zeit im Testmanagement eingespart werden können. Sind alle Testfälle abgedeckt und erfolgreich bestanden, wirkt sich dies positiv auf die Qualität der Software und dies wiederum positiv auf die Zufriedenheit der Stakeholder aus.

Für zukünftige Softwareentwicklungsprojekte können Informationen bezüglich der Trace Links zwischen Anforderungen, Code und Testfällen vorausgegangener Projekte genutzt werden, sodass mögliche Problemstellungen des neuen Projekts frühzeitig identifiziert werden können.

7 Limitationen und Ausblick

Obwohl die, in dieser Arbeit vorgestellten Ergebnisse sich auf eine spezifische Fallstudie beziehen, gehen wir davon aus, dass sich die Ergebnisse problemlos bzw. mit geringen Modifikationen auf andere Unternehmen übertragen lassen. Zum einen befanden sich unter den befragten Personen auch Entwickler externer Dienstleister, zum anderen zeigt ein Vergleich des rekonstruierten Vorgehensmodells bei Alpha mit Vorgehensmodellen aus der Literatur weitreichende Übereinstimmung. Somit können abgesehen von den unternehmensspezifischen Teststufen, die dargestellten Artefakte und deren Zusammenhänge auf Anwendungsfälle und agile Entwicklungsprozesse anderer Unternehmen übertragen werden.

Für die Sicherstellung von Traceability müssen alle Beziehungen von den Anforderungen zu den spezifischen Entwicklungskomponenten, wie beispielsweise Tests, initial erstellt und im Laufe des Projekts fortlaufend aktualisiert werden. In komplexen Projekten ist dies fehleranfällig und oftmals mit großem Aufwand für die Projektbeteiligten verbunden. Für denjenigen, der diese Links dokumentiert, ergeben sich zunächst keine Vorteile. Sind jedoch Trace Links zwischen den verschiedenen Artefakten umfassend dokumentiert, resultieren zahlreiche Vorteile hinsichtlich des Projektmanagements oder einer Verbesserung der Entwicklungsqualität (siehe Kapitel 1). Während der Entwicklung können, beispielsweise im Fall von Anforderungsänderungen, alle betroffenen Komponenten identifiziert und entsprechend modifiziert werden. Wenn diese Trace Links jedoch nicht korrekt erstellt bzw. laufend aktualisiert werden, können auf dieser Basis fehlerhafte Artefakte erzeugt oder falsche Managemententscheidungen getroffen werden. Aufgrund fehlerhafter Trace Links kann es weiterhin zu falschen Designentscheidungen oder Prioritäten kommen, was wiederum zu Unzufriedenheit der Stakeholder führen kann [LGJ03].

Auch die Ergebnisse der Experteninterviews spiegeln diese Problematik wider. Um Projektverantwortliche und Entwickler in der Praxis von den Vorteilen zu überzeugen und den Aufwand für die Dokumentation und Pflege der Traceability Links möglichst gering zu halten, bedarf es daher Traceability Ansätze, die an den Bedürfnissen der Praxis ausgerichtet sind sowie geeigneter IT-Unterstützung. Das in dieser Arbeit vorgestellte konzeptionelle Datenmodell kann hierbei als Grundlage für ein Softwaretool zur Sicherstellung von Traceability im agilen Entwicklungskontext dienen.

Danksagung

Diese Veröffentlichung entstand im Rahmen des Sonderforschungsbereichs 768 „Zyklenmanagement von Innovationsprozessen – Verzahnte Entwicklung von Leistungsbündeln auf Basis technischer Produkte“. Das Forschungsvorhaben wird gefördert durch die Deutsche Forschungsgemeinschaft (DFG).

Literaturverzeichnis

[Ad11] Adzic, G.: Specification by Example: How Successful Teams Deliver the Right Software. Verlag Manning, 1. Auflage, 2011.

[Bo13a] Bouillon, E. et al.: Leichtgewichtige Traceability im agilen Entwicklungsprozess am Beispiel von Scrum. Softwaretechnik-Trends, Ausgabe 01/2013. Köllen Druck & Verlag GmbH, 2013; S. 29-30.

[Bo13b] Bouillon, E. et al.: A Survey on Usage Scenarios for Requirements Traceability in Practice Requirements Engineering. Foundation for Software Quality Lecture Notes in Computer Science, Vol. 7830, 2013; S.158-173.

[CH01] Cockburn, A.; Highsmith, J.: Agile Software Development: The People Factor. IEEE Computer Society, Vol. 34, No. 11, 2001; S. 131-133.

[EG11] Espinoza, A.; Garbajosa, J.: A study to support agile methods more effectively through Traceability. Innovations in Systems and Software Engineering, Vol.7, No. 1, 2011; S. 53-69.

[Fo06] Fowler, M.: Continuous Integration. 2006, http://www.thoughtworks.com/ Continuous Integration.pdf (Stand: 09.09.2014)

[GF94] Gotel, O.; Finkelstein, A.: An Analysis of the Requirements Traceability Problem. Proceedings of the First International Conference on Requirements Engineering, Colorado Springs 1994; S. 94-101.

[Gh08] Ghazarian, A.: Traceability Patterns: An Approach to Requirement- Component Traceability in Agile Software Development. Proceedings of the 8th conference on Applied Computer Science, Wisconsin 2008; S. 236-241.

[GL06] Gläser, J.; Laudel, G.: Experteninterviews und qualitative Inhaltsanalyse: als Instrumente rekonstruierender Untersuchungen. VS Verlag für Sozialwissenschaften, 2. Auflage, 2006.

[Gr11] Grande, M.: 100 Minuten für Anforderungsmanagement. Vieweg+Teubner Verlag, Springer Fachmedien Wiesbaden GmbH, 2011; S. 9-20.

[HDJ09] Hayes, J. H.; Dekhtyar, A.; Janzen, D. S.: Towards Traceable Test- Driven Development. ICSE Workshop on Traceability in Emerging Forms of Software Engineering, Vancouver 2009; S. 26-30.

[He07] Hevner, A. R.: A Three Cycle View of Design Science Research. Scandinavian Journal of Information Systems, Vol. 19, No. 2, Article 4, 2007.

[Hi02] Highsmith, J.: What Is Agile Software Development?. Cross Talk, The Journal of Defense Software Engineering, Vol. 15, No. 10, 2002; S. 4-9.

[KS98] Kotonya, G.; Sommerville, I.: Requirements Engineering - Processes and Techniques. John Wiley & Sons, New York, 1998.

[LGJ03] Lee, C.; Guadagno, L.; Jia, X.: An Agile Approach to Capturing Requirements and Traceability. Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering, 2003; S. 17-23.

[LQ10] Lucia, D. A.; Qusef, A.: Requirements Engineering in Agile Software Development. Journal of Emerging Technologies n Web Intelligence, Vol. 2, No. 3, 2010; S. 212-220.

[NMM05] Nerur, S.; Mahapatra, R.; Mangalaraj, G.: Challenges of Migrating to Agile Methodologies. Communications of the ACM – Adaptive complex enterprises, Vol. 48, No. 5, 2005; S. 72-78.

[RE93] Ramesh, B.; Edwards, M.: Issues in the Development of a Requirements Traceability Model. Proceedings of IEEE International Symposium on Requirements Engineering, San Diego, 1993; S. 256-259.

[RJ01] Ramesh, B.; Jarke, M.: Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering, Vol. 27, No. 1, 2001; S. 58-93.

[Sc08] Schaffry, A.: Manufacturing Execution Systeme, 2008; http://www.cio.de/strategien/methoden/847558/index2.html (Stand: 18.08.2014)

[WC03] Williams, L.; Cockburn, A.: Agile Software Development: It's about Feedback and Change. IEEE Computer Society, Vol. 36, No. 6, 2003; S. 39-43.
[WGK13] Wolfenstetter, T.; Goswami, S.; Krcmar, H.: Herausforderungen auf dem Weg zu einer zyklengerechten Requirements Traceability für Produkt-Service-Systeme. Zyklenmanagement Aktuell – Innovationen Gestalten, No. 3, 2013; S. 7-9.
[WW02] Webster, J.; Watson, R. T.: Analyzing the past to prepare for the future: writing a literature review. MIS Quarterly, Vol. 26, No. 2, 2002; S. 13-23.