Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Leon Fausten

Architektur-Diskussionen in Scrum Prozessen - Eine empirische Studie

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Leon Fausten

Architektur-Diskussionen in Scrum Prozessen - Eine empirische Studie

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Kai von Luck
Zweitgutachterin: Dr. Susanne Draheim

Eingereicht am: 08.12.2016

Leon Fausten

Thema der Arbeit
Architektur-Diskussionen in Scrum Prozessen - Eine empirische Studie

Stichworte
Softwarearchitektur, Architekturdebatten, Architektur-Diskussionen, Architekturerosion, Architekturreengineering, technische Schulden, agile Entwicklung, Scrum, Kanban

Kurzzusammenfassung
Diese Arbeit handelt von der Architekturentwicklung in agilen Prozessen. Im ersten Untersuchungsansatz wurde mithilfe von Reengineering geprüft, wie erfolgreich sich Strukturen aus vorhandenem Quellcode extrahieren lassen. Der zweite Ansatz hat sich mit vorhandenen theoretischen Aspekten zur Integration von Architekturdebatten in den agilen Prozess befasst. Im dritten Aspekt wurde mit acht Interviews in Unternehmen mit verschiedenen Geschäftsfeldern untersucht, wie diese versuchen das Problem zu lösen. Die Interviews wurden mithilfe verschiedener Themenschwerpunkte untersucht und verglichen. Aus einem Teil der Ergebnisse wurde ein Vorschlag für einen Scrum Prozess mit integrierten Architekturdebatten entwickelt.

Leon Fausten

Title of the paper
Architecture discussions in scrum processes - An empirical study

Keywords
software architecture, architecture debates, architecture discussions, architecture erosion, technical debts, architecture reengineering, agile development, scrum, kanban

Abstract
This thesis is about architecture development in agile processes. In the first research approach reengineering was used to examine how well structures can be extracted from existing source code. The second approach dealt with the existing theoretical aspects of integrating architectural debates into the agile development process. In the third approach eight interviews were conducted with companies from different business fields to examine how they try to solve this problem. The interviews were studied and compared by means of different topic focuses. A portion of the results were used to develop a scrum based process with integrated architectural debates.

Inhaltsverzeichnis

Abbildungsverzeichnis vii

1 Einleitung 1
- 1.1 Ziele . . . 2
- 1.2 Aufbau der Arbeit . . . 3

2 Analyse 5
- 2.1 Softwarearchitektur . . . 5
- 2.2 Relevanz von Softwarearchitekturen . . . 8
  - 2.2.1 Softwarelebenszyklen . . . 8
  - 2.2.2 Architekturdebatten . . . 9
  - 2.2.3 Accidental Architecture . . . 10
- 2.3 Technische Schulden und Architekturerosion . . . 10
  - 2.3.1 Architekturerosion . . . 11
  - 2.3.2 Ursachen und Symptome . . . 13
  - 2.3.3 Architectural Smells . . . 15
- 2.4 Architekturentwicklung in agilen Vorgehensmodellen . . . 16
  - 2.4.1 Agile Manifesto und das Framework Scrum . . . 17
  - 2.4.2 Refaktorisierung . . . 22
- 2.5 Sozio-technische Kongruenz . . . 25
- 2.6 Architekturerosion steuern . . . 26
- 2.7 Fazit . . . 26

3 Bottom-Up: Automatische Architektur Prüfung und Reengineering 28
- 3.1 Ziele . . . 28
- 3.2 Szenarien zur Wiederaufnahme von Projekten . . . 30
- 3.3 Analysetechniken . . . 31
- 3.4 Architekturanalysen . . . 33
  - 3.4.1 Bug Pattern Detection . . . 33
  - 3.4.2 Conformance Prüfung . . . 34
  - 3.4.3 Soll-Architektur Ermittlung . . . 38
  - 3.4.4 Architekturmonitoring . . . 38
- 3.5 Architekturanalysen in der Praxis . . . 39
- 3.6 Fazit . . . 41

4 Top-Down: Integration von Architekturdebatten in agilen Prozessen 44
4.1 Ziele . . . 44
4.2 Entscheidungsfindung . . . 45
4.3 Architekturmuster . . . 46
4.4 Agile Architektur-Prinzipien . . . 46
4.5 Architekturanforderungen . . . 49
4.6 Die Rolle eines Softwarearchitekten . . . 51
4.7 Architekturdokumentation . . . 53
4.7.1 Dokumentations Frameworks . . . 54
4.8 Vorgehen zur Architekturentwicklung in Scrum . . . 57
4.8.1 Ermittlung der Features . . . 57
4.8.2 Entwicklung der Softwarearchitektur . . . 58
4.8.3 Integration in Scrum . . . 60
4.9 Model-Driven Engineering . . . 61
4.9.1 Agile Model Driven Development . . . 63
4.10 Fazit . . . 65

5 Interviews zur Untersuchung der Architektur-Diskussionen 67
5.1 Vorgehen und Methodik . . . 67
5.1.1 Auswahl der Firmen und Projekte . . . 68
5.1.2 Experteninterviews . . . 69
5.1.3 Nachbereitung und Auswertung . . . 70
5.2 Themenauswahl . . . 71
5.2.1 Leitfaden . . . 71
5.3 Einzelauswertung . . . 73
5.3.1 Interview A . . . 73
5.3.2 Interview B . . . 74
5.3.3 Interview C . . . 76
5.3.4 Interview D . . . 77
5.3.5 Interview E . . . 78
5.3.6 Interview F . . . 79
5.3.7 Interview G . . . 80
5.3.8 Interview H . . . 82
5.4 Gesamtauswertung . . . 83
5.4.1 Projekt und Interview Kontext . . . 84
5.4.2 Diskussionsmöglichkeiten . . . 85
5.4.3 Management der Architekturerosion während der Entwicklung . . . 89
5.4.4 Softwarearchitekt . . . 95
5.4.5 Dokumentation . . . 97
5.5 Ableitungen zum Kontext der Unternehmen . . . 102
5.6 Abbildung in das Scrum Framework . . . 103
5.7 Fazit . . . 105

6 Schluss **108**
6.1 Zusammenfassung . . . 108
6.2 Fazit und Ausblick . . . 110

Literaturverzeichnis **112**

Anhang **123**
Interviewleitfaden Vorlage . . . 123
Interviewbericht Unternehmen A . . . 126
Interviewbericht Unternehmen B . . . 130
Interviewbericht Unternehmen C . . . 135
Interviewbericht Unternehmen D . . . 138
Interviewbericht Unternehmen E . . . 142
Interviewbericht Unternehmen F . . . 146
Interviewbericht Unternehmen G . . . 149
Interviewbericht Unternehmen H . . . 155
Kodierleitfaden . . . 158
Mindmap Architekturthemen . . . 159

Abbildungsverzeichnis

1 Einleitung **1**

2 Analyse **5**
2.1 Architektur Details . . . 7
2.2 Architekturerosion [55] . . . 12
2.3 Verbreitung einzelner agiler Vorgehen [81] . . . 18
2.4 Scrum Zyklus [16] . . . 19
2.5 Architektur-Refaktorisierung [20] . . . 24

3 Bottom-Up: Automatische Architektur Prüfung und Reengineering **28**
3.1 Dependency Structure Matrix [33] . . . 35
3.2 Generierung eines Reflexion Model [61] . . . 37

4 Top-Down: Integration von Architekturdebatten in agilen Prozessen **44**
4.1 SoftGoal Abhängigkeitsgraph [22] . . . 51
4.2 Entscheidungsbeziehungen als Graph [79] . . . 56
4.3 Plastic Partial Components [68] . . . 59
4.4 Integration einer Architekturentwicklung in Scrum [68] . . . 61
4.5 Agile Model Driven Development [5] . . . 63

5 Interviews zur Untersuchung der Architektur-Diskussionen **67**
5.1 Vorgehen mit Debatten, Abbildung angelehnt an [16] . . . 103

6 Schluss **108**

1 Einleitung

Die Softwareentwicklung hat sich im Laufe der Zeit stark verändert. In den traditionellen Modellen wird die Realisierung vor Beginn detailliert geplant und modelliert. Eine ausführliche, sehr feingranulare Softwarearchitektur wird zu diesem Zweck im Voraus entworfen. Nachdem die Architektur vollständig entwickelt ist, kann mit der Umsetzung begonnen werden. Die Architektur steht im Mittelpunkt der Entwicklung. Die Funktionalitäten werden um sie herum gebaut. Dies bringt einige Nachteile, wie einen sehr starren Entwicklungsprozess, mit sich und führt dadurch häufig zu einem Scheitern des Projekts [74]. Die agilen Vorgehensmodellen versprechen viele der Probleme zu lösen. Eine häufigere und deshalb schnellere Auslieferung wird erreicht. Als Folge entstehen kürzere Feedbackschleifen mit dem Kunden. Fachliche Fehler können frühzeitig aufgedeckt werden. Dies sorgt bei der Entwicklung zu einer Konzentration auf die Features [13]. Die Architektur gerät somit schnell aus dem Fokus. Als Folge werden die nichtfunktionalen Anforderungen allerdings schnell vernachlässigt. Dadurch können einige Qualitätsziele nicht erreicht werden und somit ebenfalls zu einem Scheitern des Softwareprojekts führen [32]. Bei den Qualitätszielen kann es sich exemplarisch um Anforderungen an die Wartbarkeit und Erweiterbarkeit handeln. Die agile Entwicklung führte aber dazu, dass keine oder nur noch wenig technische Planung stattfindet [40]. Dies bedeutet, dass keine gezielte Architektur entsteht. Gerade bei großen und langlebigen Anwendungen ist es allerdings erforderlich eine Architektur zielgerichtet zu realisieren und dauerhaft an die aktuellen Anforderungen anzupassen.
Bei einer ungeplanten Architektur entsteht diese zufällig. Dadurch kann es mit der Zeit dazu kommen, dass die eigentliche Entwicklung immer länger dauert, obwohl die Aufgaben nicht komplexer werden. Durch die vermehrte Aufnahme von technischen Schulden entsteht eine dramatische Architekturerosion. Probleme mit denen nicht gerechnet werden kann können auftreten. Dies können Nebeneffekte sein, die im ersten Blick häufig nicht mit der vorgenommenen Änderung in Zusammenhang stehen dürften. Eine Erosion entsteht aber auch bei der traditionellen Entwicklung, da die Architektur häufig nicht an geänderte Anforderungen angepasst wird. Ein weiterer Punkt ist, dass die Entwickler die Anwendung kennen müssen. Nur so kann eine Weiterentwicklung in einem angemessenem Zeitraum geschehen. Um dies zu

unterstützen müssen klare Architekturstile für die Anwendung festgelegt werden. Da in vielen Fällen keine aktuelle Dokumentation vorhanden ist, muss das Wissen händisch erarbeitet werden.
Eine Architektur entsteht bei jeder Entwicklung. Sie kann dabei gut strukturiert und übersichtlich oder auch ungewiss mit chaotischen Strukturen sein. Wie im Laufe dieser Arbeit erläutert wird, ist eine Erosion ein Zeichen für eine nicht gut strukturierte Architektur. Neben den fachlichen Aspekten ist es deshalb ebenfalls sinnvoll, eine technische Debatte zu führen. Wenn in dem Entwicklungsvorgehen keine Architektur-Diskussionen stattfinden, entsteht die Architektur unkontrolliert, meist mit vielen verschiedenen Stilen.
Eine strukturierte Architektur hilft zusätzlich bei der Kommunikation und Teilung der Verantwortlichkeiten bei mehrere Teams. Dies führt somit zu einer besser organisierbaren Entwicklung [39].

Es ist besonders wichtig, dass die Einführung von Architektur-Diskussionen die Agilität nicht beeinflusst. Die Vorteile der agilen Entwicklung dürfen nicht negativ beeinträchtigt werden. Es existiert ein Spannungsverhältnis zwischen der Agilität und der Einführung von Architektur-Diskussionen.

1.1 Ziele

Ziel dieser Arbeit ist herauszufinden wie die agile Entwicklung mit der Architekturentwicklung vereint werden kann. Dabei soll besonderen Wert auf die Erhaltung der agilen Werte des Manifestos [43] gelegt werden. Die agilen Werte sind:

- *„Individuals and interactions over processes and tools.“*
- *„Working software over comprehensive documentation.“*
- *„Customer collaboration over contract negotiation.“*
- *„Responding to change over following a plan.“*

Dies bedeutet, dass die Einführung von Architekturdebatten keine Defizite in der Zusammenarbeit der Teammitglieder und mit dem Kunden bewirken dürfen. Aus diesem Grund ist erforderlich, dass die Architektur, wie auch die gesamte Anwendung, im Laufe des vollständigen Prozesses mitwächst und durchgehend an die aktuellen Bedürfnisse angepasst wird. Die Architektur darf nicht von einzelnen Personen vorgeschrieben werden. Das gesamte Team muss stattdessen an ihrer Entwicklung beteiligt sein.

Eine Architektur soll, z.B. durch einer dauerhafte Wartbarkeit dafür sorgen, dass die Lebensdauer einer Anwendung verlängert wird. Dem Ziel, ein Vorgehen zu finden, um eine Architektur zu entwickeln, soll sich mithilfe unterschiedlicher Teilziele angenähert werden. Ein Teilziel ist es, einen Überblick über die Möglichkeiten des Reengineerings-Ansatzes zu schaffen. Hier soll herausgefunden werden, in welchem Umfang eine Architektur aus vorhandenem Quellcode rekonstruiert werden kann. Dieser Ansatz soll außerdem herausfinden, wie eine dauerhaft gute Codequalität sichergestellt werden, kann indem potentielle Fehlerstellen automatisiert gefunden werden.
Ein zweites Subziel ist es herauszufinden, welche aktuellen Ideen existieren, um die agile Entwicklung und die Architekturentwicklung miteinander zu vereinen. Eine Übersicht über Kombinationsmöglichkeiten einer agilen Entwicklung mit einer Architekturentwicklung soll geschaffen werden.
Außerdem soll herausgefunden werden, wie die Entwicklung in der Praxis stattfindet. Die Untersuchungen in der Praxis sollen feststellen, welche der vorher erarbeiteten Punkte in der Realität eingesetzt werden. Als Schlussfolgerung soll ein Verfahren entwickelt werden, welches einen Scrum Prozess mit einer Architekturdebatte abbildet.

1.2 Aufbau der Arbeit

Diese Arbeit ist folgendermaßen aufgebaut. Zu Anfang wird in Kapitel 2 (Analyse) der Begriff Softwarearchitektur genauer erläutert. Es wird erklärt, wie der Begriff im Zusammenhang mit dieser Arbeit zu verstehen ist. Außerdem wird besprochen weshalb eine gut strukturierte Architektur wichtig ist. Folgen einer zufälligen Architektur, die ohne eine Debatte entsteht, werden vorgestellt. Vorhandene Aussagen und Ansätze des agilen Manifesto und konkreter agiler Vorgehen zu diesem Thema werden untersucht.
Im Kapitel 3 wird über den Bottom-Up Ansatz gesprochen. Das heißt, es werden Vorgehen und Tools vorgestellt die automatisierte Analysen des Quellcodes durchführen. Diese Analysen sollen unter anderem nach einer längeren Entwicklungspause eine Weiterentwicklung vereinfachen. Zu diesem Zweck wurden unterschiedliche Szenarien als Ausgangssituation entwickelt. Anschließend werden mehrere Analyseverfahren, um allgemeine Fehler zu finden vorgestellt. Zusätzlich wird der Vergleich einer Soll- und Ist-Architektur und ein halbautomatischer Vorgang zur Ermittlung der Soll-Architektur analysiert. Ein Vorgehen aus der Praxis um eine Art von Architekturreview durchzuführen wird außerdem präsentiert.
Anschließend wird im Kapitel 4 der Top-Down Ansatz untersucht. Zu diesem Zweck werden Vorschläge aus Arbeiten analysiert, die eine Architektur im Rahmen eines agilen Vorgehens

entwickeln. Es werden sowohl Teilaspekte, wie die Erstellung einer Dokumentation und die Rolle eines Softwarearchitekten, wie auch Vorschläge für vollständige Entwicklungsvorgehen betrachtet.
In Kapitel 5 wird ein realer Bezug zur Praxis hergestellt. Zu diesem Zweck wurden Interviews in Unternehmen durchgeführt, um deren Entwicklungsprozess zu erfassen. Die unterschiedlichen Prozesse werden auf Gemeinsamkeiten untereinander und in Bezug auf die vorherigen Kapiteln untersucht. Aus den gewonnenen Erkenntnissen wird ein Vorschlag für ein mögliches Verfahren auf Basis von Scrum entwickelt.
Zum Ende werden die Ergebnisse im Kapitel 6 zusammengefasst und ein allgemeines Fazit zur Arbeit gezogen. Zusätzlich wird ein Ausblick für eine mögliche Fortführung der Arbeit gegeben.

2 Analyse

In diesem Kapitel wird das Problemfeld vorgestellt. Zu diesem Zweck wird diskutiert, was eine Architektur ist und weshalb es wichtig ist diese schrittweise mithilfe von Debatten zu entwickeln. Auswirkungen von fehlenden Architekturdebatten werden besprochen. Dazu zählt die Aufnahme von technischen Schulden und der eng damit verbundenen Architekturerosion. Im nächsten Schritt werden das agile Manifesto und konkrete agile Vorgehen, wie Scrum auf Aussagen zur Architekturentwicklung hin untersucht. Der Einflussfaktor des Teams auf die Architektur wird anschließend vorgestellt. Zum Ende des Kapitels wird ein Fazit gezogen.

2.1 Softwarearchitektur

In der Informatik wird unter einer Softwarearchitektur der innere Aufbau einer Anwendung verstanden. Mit dem Begriff Architektur ist in der gesamten Arbeit die Softwarearchitektur gemeint.

Das Verständnis, was unter einer Softwarearchitektur verstanden wird unterscheidet sich zum Teil deutlich. Häufig existiert ein allgemeines Verständnis, im Rahmen von allgemeinen Mustern wie z.B. Schichtenarchitektur, bis hin zu Mustern die sich auf Details der Implementierung beziehen. Diese Sichtweisen unterscheiden sich sehr bei ihrer Detailliertheit und den darin enthaltenen Informationen.

Durch die Architektur sollen Qualitätsziele und nichtfunktionale Anforderungen erreicht werden. Dies kann z.B. die Sicherheit, Verlässlichkeit, Verfügbarkeit und viele weitere Eigenschaften betreffen. Diese Ziele können unterschiedlich erreicht werden, z.B. durch die Wahl von Sprachen, Frameworks oder z.B. Entwurfsmustern. In der Gesellschaft ist weitestgehend akzeptiert, dass die Umsetzung der Qualitätsziele eine hohe Relevanz hat und bei Vernachlässigung zu einer nicht nutzbaren Anwendung führen kann [23]. Die Qualitätsanforderungen hängen in vielen Fällen eng von den nichtfunktionalen Eigenschaften ab. Diese werden allerdings häufig während der Entwicklung vernachlässigt. [32]

Nord [63] sagt, dass eine Architektur die *High-Level* Strukturen einer Anwendung darstellt. Diese Beschreibung ist allerdings noch sehr allgemein. Unter High-Level Struktur können die allgemeinen Komponenten und deren Verbindungen verstanden werden. Wie sehr dies ins Detail geht ist durch den Begriff nicht eindeutig festgelegt. Coplien und Reenskaug [26] beschreiben Architektur als Form des Systems, welches durch bewusstes Entwerfen entstanden ist. Die Architektur stellt Strukturen innerhalb der realisierten Anwendung dar. Auf einem hohem abstraktem Level kann eine Architektur viele unterschiedliche Anwendungen gleichzeitig beschreiben. Das ist auch zutreffend, wenn einzelne Anwendungen andere Parameter und Funktionalitäten besitzen. Eine allgemeine Client-Server Architektur trifft z.B. auf viele Anwendungen zu. Solch eine Architektur enthält keine konkreten Details über das Innenleben der Komponenten und deren Funktionsweisen. Es ist nur bekannt, dass es zwei Komponenten gibt, die miteinander kommunizieren. Die Architektur muss die Evolution der Anwendung unterstützen, um bei geänderten Anforderungen reagieren zu können. [26]
Bertolino u.a. [11] beschreiben in ihrer Arbeit eine Softwarearchitektur (SA) ebenfalls als eine High-Level Struktur des Software-Systems. Die SA hat das Ziel die Komplexität der Komponenten zu managen und die Wiederverwendbarkeit zu verbessern. In den 90er Jahren war die Hauptaufgabe der SA die Komponenten, deren Verbindungen und das Systemverhalten zu beschreiben. Mit der Zeit wurden die Anforderungen an die Architektur immer größer. Die individuell relevanten Designentscheidungen wurden ein Teil der SA. [11]

Durch diese erweiterte Beschreibung stellt sich die Frage, was unter Architektur und Design zu verstehen ist. Nach Jansen und Bosch [52] basiert eine Architektur auf zahlreichen Designentscheidungen. Ihre Definition für Designentscheidungen ist folgende:

> *„A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture.“* [52]

Diese Definition ist rekursiv, da sie einerseits aussagt, dass eine Architektur aus zahlreichen Designentscheidungen besteht und Designentscheidungen wiederum die Architektur verfeinern. Bei dieser Definition besteht das Problem, dass klar sein muss, was die Architektur ist. Die Basis Architektur, die durch erste Designentscheidungen verfeinert wird, kann als reine Aufteilung in Komponenten und deren Kommunikation verstanden werden. Die Designentscheidungen füllen nach und nach das Innenleben der Komponenten und verfeinern diese dadurch.

Martin [58] hat eine sehr passende Beschreibung von Architektur, mit einer Aufteilung in verschiedene Ebenen aufgestellt. Die höchste und gröbste Ebene bei einer Architektur sind die

Architekturpattern, welche die Gesamtstrukturen der Anwendung beschreiben. Dies kann z.B. eine *3-Schichten Architektur* sein. Dieses grobe Muster ist den meisten Entwicklern geläufig, beinhaltet aber keine Aussagen über genauere Strukturen und trifft auf eine Vielzahl von Anwendungen zu. Eine Ebene tiefer befinden sich anwendungsspezifische Architektureigenschaften. Dies können Subsysteme und weitere gegebenen Eigenschaften sein, von denen die Anwendung abhängig ist und bei der Umsetzung zwingend berücksichtigt werden müssen. Ein Einfluss auf diese Eigenschaften ist häufig nicht möglich. In der nächsten Ebene befindet sich die konkrete Aufteilung in Komponenten und deren Kommunikation. Entwurfsmuster können sowohl die Aufteilung der Komponenten, wie auch deren Innenleben beschreiben. Aus diesem Grund sind diese auf gleicher Ebene, wie die Komponenten und auch eine Ebene höher wiederzufinden. Das *Factory* Entwurfsmusters macht z.B. Vorschläge zur genauen Objekterzeugung. Statt dem Konstruktor, werden Objekte mithilfe einer Methode erzeugt. Dies ist auf der Ebene der inneren Implementierung einer Klasse anzusehen. Das *Adapter* Pattern beschränkt sich im Vergleich dazu auf Empfehlungen für eine Komponente. Die einzelnen Stufen sind in Abbildung 2.1 nochmals visuell dargestellt.

Die zuletzt vorgestellte Beschreibung von Martin passt am besten für die Thematik, im Bezug auf diese Arbeit. Sie beschreibt die Architektur mit der Möglichkeit sie in unterschiedlicher Granularität zu betrachten. Zusammengefasst beschreibt eine SA sowohl die allgemeine Struktur im Rahmen eines allgemeinen Musters, sowie bei genauerer Betrachtung der einzelnen Bestandteile immer mehr Feinheiten.

Abbildung 2.1: Architektur Details

2.2 Relevanz von Softwarearchitekturen

Dieser Abschnitt soll feststellen, wie relevant eine Architektur für eine Anwendung ist und ob es durchaus zu akzeptablen Lösungen kommen kann, wenn keine Architekturdebatten stattfinden. Außerdem wird untersucht, ob dies bei jeder Art von Anwendung gleichbedeutend der Fall ist.

2.2.1 Softwarelebenszyklen

Unter einer Architekturdebatte ist eine Diskussion des gesamten Teams über die technische Realisierung zu verstehen. Diese sollen gemeinsam Entscheidungen treffen, die die Weiterentwicklung der Anwendung betreffen.
Um festzustellen, ob eine ausgeklügelte Architekturdebatte wichtig ist, muss der geplante Lebenszyklus der Anwendung bekannt sein. Anhand des Lebenszyklus kann erkannt werden, ob nichtfunktionale Eigenschaften, wie Wartbarkeit und Erweiterbarkeit unterstützt werden müssen. Bei einer langlebigen Anwendung, dessen Ablösedatum nicht klar ist, ist es besonders wichtig diese Eigenschaften langfristig zu unterstützen. Im Gegensatz dazu ist es nicht notwendig, Zeit und Energie in die Architekturentwicklung einer sehr kurzlebigen Anwendung zu investieren. Ein Beispiel dafür ist eine Gewinnspielanwendung die nur für einen konkreten Zeitraum entwickelt und anschließend nicht wiederverwendet wird. Dieser Zeitraum sollte eine Spanne von ein paar Wochen oder Monaten nicht überschreiten. Ein weiteres Beispiel ist eine Anwendung die nur dafür gemacht wurde, um Daten von einem alten System in ein neues zu übertragen. Dieses Tool ist nur für eine sehr kurzzeitige Nutzung vorgesehen. Neben der Unterstützung der nichtfunktionalen Eigenschaften und Qualitätsziele, ermöglicht eine Architektur eine einfachere Aufteilung der Aufgaben auf verschiedene Teams und Komponenten [63]. Dadurch kann die Architektur gleichzeitig als zentrales Mittel zur Kommunikation dienen.
Für Firmen die hauptsächlich ein Produkt entwickeln und vermarkten ist es besonders wichtig, dass dieses auf Dauer wartbar bleibt. Bei starken Problemen geht die Firma im schlimmsten Fall insolvent, da sich ihr Produkt nicht kostendeckend weiterentwickeln lässt. Bei Firmen, die auf Projektbasis arbeiten, ist dieses Problem geringer. Diese haben nicht ihr gesamtes Kapital und Wissen in ein Produkt gesteckt. Ein gewisses Risiko besteht für das Unternehmen und dessen Kunden allerdings ebenfalls. Dieses Risiko unterscheidet sich sehr, je nach Größe und Laufzeit des Projektes. Neben dem finanziellen Schaden, beschädigen scheiternde Projekte den Ruf des Unternehmens und schrecken potentielle Neukunden ab. [19]

Ein weiterer Punkt bei einer langlebigen Anwendung ist, dass während dessen Lebensdauer neue Entwickler hinzu kommen und alte gehen. Die Architektur muss durch neue Personen verstanden werden können. Hierbei können klare Strukturen helfen, die sich nach bekannten Mustern orientieren. Besonders wichtig wird dies, wenn die ursprünglichen Entwickler nicht mehr erreichbar sind, um offene Fragen zu beantworten.

2.2.2 Architekturdebatten

Eine gute Architektur sollte mithilfe des gesamten Entwicklungsteams debattiert, erarbeitet und kontinuierlich verbessert werden. Eine Entwicklung abseits des Teams führt häufig zu einer Spaltung des Teams und einer geringen Beachtung der geplanten Architektur [80]. Die Debatten sollten so gestaltet werden, dass alle beteiligten Personen ein Mitspracherecht haben. Dabei können unter anderem im Folgenden beschriebene Fehler bei der Durchführung von Debatten gemacht werden. Der Einfluss von traditionelle Methoden bewirkt, dass unter einer Architekturentwicklung meist eine rein initiale Planung verstanden wird. Dieses *Big-Up-Front Design* stellt keine besonders erfolgversprechende Lösung dar. Viele der agilen Werte, wie die Möglichkeit schnell auf Änderungen reagieren zu können, gehen verloren. Um trotzdem auf Änderungen reagieren zu können, werden zuvor getroffene Entscheidungen ignoriert. Das Vorabdesign wird häufig durch Personen, die nicht an der eigentlichen Entwicklung beteiligt sind, entworfen [80]. Durch diese unterschiedlichen Zuständigkeiten und Phasen bei Planung und Entwicklung, entsteht eine Spaltung des Teams. Das genaue Gegenteil davon ist, dass keine Debatten und keine Planung stattfindet. Um die Anwendung lauffähig zu halten wird nachträglich durch Refaktorisierungen ausgebessert. Das Refaktorisieren geschieht dabei ebenfalls häufig in alleiniger Arbeit der Entwickler. Debatten in denen Änderungen besprochen und diskutiert werden, finden nicht statt. Der Erfolg der Änderungen und des gesamten Projektes hängt dann sehr von der Erfahrung des einzelnen Entwicklers ab. Dies ist hauptsächlich der Fall, weil in den Vorgehensmodellen keine Architekturdebatten vorgesehen sind (siehe Kapitel 2.4). Refaktorisieren ist ein wichtiger und sehr hilfreicher Bestandteil der agilen Methoden [54, 75]. Dieser sollte auch für die Architektur durchgeführt werden und ist auch in Projekten mit Architekturdebatten nicht wegzudenken. Eine Architektur allein durch Refaktorisieren zu erzeugen ist allerdings keine Lösung, wie in Kapitel 2.4.2 genauer beschrieben wird [9][1]. Vorschläge zur Durchführung einer konkreten Debatte werden in Kapitel 4 beschrieben.

[1] weitere Informationen und Beispiele, siehe Ausarbeitung des Grundprojekts: http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master14-15-gsm/fausten/bericht.pdf, Abgerufen: 15.08.2016

2.2.3 Accidental Architecture

Der Begriff *Accidental Architecture* wurde bereits im Jahr 2006 von Booch [12] eingeführt. Jede Anwendung besitzt eine Architektur. Allerdings sind viele davon eher zufällig entstanden. Booch unterscheidet dabei zwischen einer absichtlich (engl. intentional) und einer zufällig (engl. accidental) entwickelten Architektur. Eine absichtliche Architektur wird zuvor identifiziert und erst anschließend implementiert. Die zufällige Architektur entwickelt sich aus individuellen Designentscheidungen während der Entwicklung. Laut seiner Kenntnis sind die zufälligen Architekturen nicht zwingend schlecht, müssen aber sobald diese eingeführt werden bekannt gegeben werden. Nur so können diese im System verbleiben. Dadurch können zufälligen Architekturen, wenn diese mit der Zeit verbessert werden, zu beabsichtigten Architekturen werden. [12]
Zufällige Architekturen können auch als *Spaghetti-Architekturen* bezeichnet werden. Es existieren keine bewussten Strukturen, dadurch kann es z.B. passieren, dass viele Abhängigkeiten geschaffen werden. Dies macht Änderungen sehr mühselig. Durch verschiedene Entwickler, mit einem unterschiedlichem Wissensstand und unterschiedlichen Programmierstilen, werden viele verschiedene Stile in einer Anwendung gemixt. Nicht jede zufällig entstandene Architektur kann zu einer beabsichtigen Architektur wachsen. Dies ist unter anderem stark von der Größe der Anwendung abhängig. Die Verbesserungen werden durch die Refaktorisierungs-Technik durchgeführt. Weshalb dies nicht in jedem Fall eine zufriedenstellende Lösung ist, wird in Kapitel 2.4.2 beschrieben.

Unterschiedliche Stile, überflüssige Abhängigkeiten und weitere Eigenschaften können als technische Schulden bezeichnet werden. Dieser Begriff und dessen Ursachen werden im nächsten Abschnitt erläutert.

2.3 Technische Schulden und Architekturerosion

Eine genaue Definition von *Technische Schulden* ist nicht eindeutig festgelegt. Brown u.a. [14] beschreiben diese als Abstand zwischen dem aktuellem Stand der Entwicklung und einem hypothetisch idealem Stand des Systems. Die Schulden können z.B. sogenannte *Code Smells* sein, welche durch schlechte Implementierungen, wie z.B. einer Codeduplikation entstehen. Ein weiterer Punkt sind Design und Architekturschulden, die durch uneinheitliche Paketstrukturen oder z.B. Abweichungen von der geplanten Architektur entstehen. Nicht vorhandene Tests, sowie unzureichende Dokumentationen gehören ebenfalls zu den Schulden. [54]

Technische Schulden entstehen, wenn falsche oder suboptimale technische Entscheidungen getroffen werden. Für die Kunden und das Management sind die Schulden weitestgehend unsichtbar [14]. In den meisten Fällen sind diese nur im Code ersichtlich. Der Begriff wurde bereits 1992 durch Cunningham [27] eingeführt. Während der Entwicklung können die Entscheidungen zu einer einfacheren und schnelleren Lösung führen, bewirken zu einem späteren Zeitpunkt aber einen höheren Aufwand bei Wartung und Weiterentwicklung, schreibt Cunningham. Einzelne Schulden sind nicht problematisch, erst wenn sich diese anfangen zu häufen können Probleme entstehen [14, 72]. In manchen Fällen werden Schulden beabsichtigt aufgenommen, z.B. um neue Technologien (Frameworks, u.s.w.) auszutesten. [54]
Ein höherer Aufwand bedeutet, dass mehr Zeit benötigt wird. Dies verursacht direkte zusätzliche Kosten. Gartner hat die weltweiten Kosten für technische Schulden im Jahr 2010 auf 500 Milliarden US-Dollar, mit steigender Tendenz für die Zukunft (Verdopplung bis 2015), geschätzt [76]. Zum Zeitpunkt dieser Arbeit konnte leider nicht auf die aktuellen Zahlen zugegriffen werden.
Aus einer Menge von technischen Schulden entsteht schnell eine *Architekturerosion*.

2.3.1 Architekturerosion

Architekturerosion kann als Lücke zwischen der entwickelten Architektur und der geplanten Architektur betrachtet werden [29, 72, 75]. Dieser Architekturmissstand wird häufig auch als *Architektur Fäulnis* oder *Architektur Verfall* bezeichnet. Eine Architekturerosion entsteht im Laufe der Zeit mit der andauernden Aufnahme von neuen technischen Schulden. Es entsteht schnell ein unmanagebarer Monolith. Die Erosion hat negative Auswirkungen auf die Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit. [75]
Technische Schulden werden z.B. bewusst aufgenommen, um zum Zeitpunkt der Entwicklung, schneller Lösungen zu erreichen. Die Erosion wird auch erhöht, wenn sich die Anforderungen geändert haben und diese Änderungen in die Anwendung übernommen werden, die geplanten Architektur allerdings nicht aktualisiert wird. Dies gilt sowohl für neue Funktionalitäten, wie auch neue Ansprüche, z.B. an die Performance. Wenn diese Änderungen nicht im Einklang mit der geplanten Architektur stehen, weichen die Soll- und Ist-Architektur immer weiter voneinander ab. [10]
Ein weiterer Grund für die Entstehung einer Erosion ist, dass Entwickler die zugrundeliegenden Architekturentscheidungen nicht vollständig nachvollziehen konnten. Dadurch werden Änderungen vorgenommen, die den Entscheidungen und deren Qualitätszielen im Weg stehen. [23]

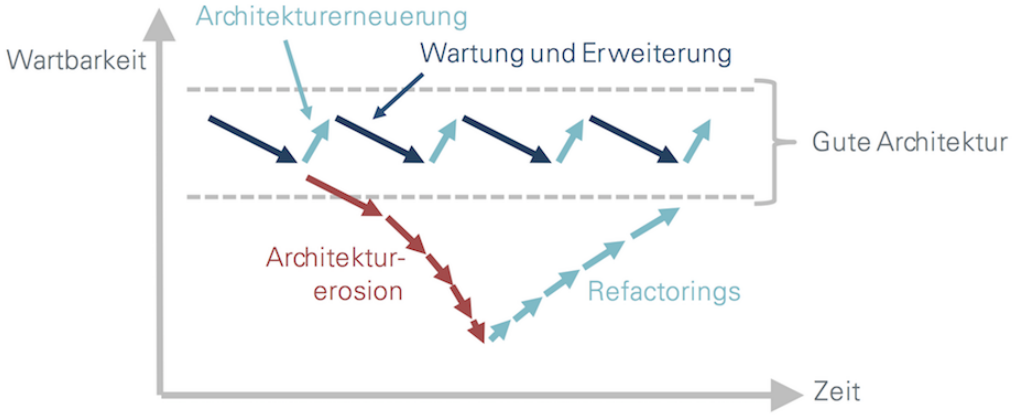


Abbildung 2.2: Architekturerosion [55]

In Abbildung 2.2 ist zu sehen, dass die Wartbarkeit direkt mit einer guten Architektur in Verbindung steht. Es ist möglich, dass die schlechte Qualität von außen, z.B. durch eine gute Performance, nicht sichtbar ist. Das gesamte Gerüst kann aber zusammenfallen, sobald Änderungen durchgeführt werden. [72]
Die dunkelblauen und roten Pfeile stellen die Wartung und Weiterentwicklung dar. Wenn dies ohne eine durchgehende Architekturerneuerung durchgeführt wird, verschlechtert sich die Wartbarkeit und Erweiterbarkeit drastisch. Die Entwicklung neuer Features wird immer schwieriger und benötigt erheblich mehr Zeit. Dies ist daran zu erkennen, dass die roten Pfeile immer kürzer werden und immer stärker sinken. Immer weniger Features können in der gleichen Zeit realisiert werden. Die neuen Features verringern erneut die Wartbarkeit, da zusätzliche Abhängigkeiten zu ihnen aufgebaut werden. [54, 72]
De Silva u.a. [72] haben durch Analysen von Fallstudien festgestellt, dass eine Erosion unvermeidbar ist. Dies liegt vor allem an der Komplexität der Software und den verwendeten Software-Engineering Praktiken. Ein Teil der Komplexität entsteht durch die dauerhafte Notwendigkeit die Anwendung an die aktuellen Anforderungen anzupassen. Deswegen wird statt nach einer vollständigen Vermeidung von Erosion, nach einer Möglichkeit gesucht, diese zu kontrollieren. Dadurch soll die Erosion auf einem akzeptablem Level gehalten werden.

Ab einem bestimmten Zeitpunkt der Entwicklung muss entschieden werden, ob die Anwendung vollständig neu entwickelt werden muss oder ob eine Refaktorisierung durchgeführt werden sollte [27]. Eine regelmäßige Architekturerneuerung durch refaktorisieren kann dies verhindern. Dies ist ebenfalls sinnvoll, wenn sich die Anwendung bereits außerhalb des Berei-

ches für eine gute Architektur befindet. [54]
Bei einer Neuentwicklung muss in vielen Fällen das alte System gleichzeitig weiterentwickelt und gewartet werden, bis es durch das Neue ersetzt wird. Das neue System muss, wie auch das Alte, durchgehend an die neuen Anforderungen angepasst werden. Dadurch entsteht trotz guter Designentscheidungen häufig erneut der Wunsch nach einer erneuten Neukonstruktion. [58]

Um eine Erosion feststellen zu können ist es notwendig dessen Symptome früh zu erkennen. Kenntnisse über die Ursachen sind zusätzlich notwendig, um eine Erosion soweit wie möglich von Beginn an zu verhindern.

2.3.2 Ursachen und Symptome

Lilienthal [54] konnte unterschiedliche Ursachen für die Entstehung von technischen Schulden und einer damit verbundenen Architekturerosion identifizieren. Diese kommen häufig in Kombination vor:

Entwickler ohne Erfahrung in Architekturdesign Das Personen programmieren können heißt nicht, dass diese ebenfalls Architekturen entwerfen können. Dieses Wissen muss zusätzlich angeeignet werden. Mithilfe verschiedener Projekte muss zusätzlich praktische Erfahrung gesammelt werden. Das Erlernen von theoretischen Konzepten und Ideen spielt dabei ebenfalls eine wichtige Rolle. Dies kann z.B. in einem Studium mit den passenden Fächern über Softwarearchitekturen und Mustern geschehen. Die praktischen Erfahrungen und das Hintergrundwissen aus der Theorie ermöglichen es erfolgreich Architekturen zu entwickeln. Häufig sind allerdings gerade ältere Entwickler Quereinsteiger, die sich vieles selber beigebracht haben. [54]

Erosion steigt unbemerkt Die Erosion kann unbemerkt steigen, wenn bei der Implementierung von den Vorgaben abgewichen wird. Dies kann bewusst, wie auch unbewusst geschehen. Es wird teilweise bewusst von zuvor getroffenen Entscheidungen abgewichen, weil erkannt wurde, dass diese die Anforderungen nicht erfüllen. Änderungen werden nur vereinzelt durchgeführt und nicht auf die gesamte Architektur übertragen. In einigen Fällen werden ohne einen vorherigen Entwurf Lösungen umgesetzt. Dies geschieht meist um Zeit einzusparen. Durch diese Art von Entwicklung erodiert die Architektur langsam, ohne dass dies direkt bemerkt wird.

Komplexität und Größe Eine Anwendung kann fachlich und technisch komplex sein. Dies erschwert das vollständige Verständnis. Außerdem sind aktuelle Softwaresysteme meist

zu groß um die gesamte Softwarearchitektur im Blick behalten zu können. Eine häufig fehlende aktuelle Dokumentation der Architektur erhöht die Komplexität zusätzlich [72]. Ein weiterer Grund ist, dass eine Verfolgung der Designentscheidungen schwierig ist. Viele dieser Entscheidungen sind schwer, bis unmöglich in dem reinen Quellcode ersichtlich [29]. Dadurch kann es vorkommen, dass bereits vorhandene Funktionalitäten mithilfe eines anderen Konzeptes erneut entwickelt werden. In der Anwendung entstehen dadurch verschiedene Strukturen, mit ähnlichen Funktionalitäten. Das Nachvollziehen dieser Strukturen wird zu einem späterem Zeitpunkt immer schwieriger.

Unverständnis des Kunden für Individualsoftwareentwicklung Der Kunde sieht meist keinen Vorteil in einer guten Architektur. Für ihn ist durch eine Verbesserung häufig kein Unterschied erkennbar [83]. Die Entwickler arbeiten unter Zeitdruck, wenn in dem Projekt feste Zeitbegrenzungen vorgegeben sind und keine Zeit für Reparaturen zur Verfügung gestellt wird. Statt einer ausgearbeiteten Architektur wird deshalb eine schnellere Lösung implementiert. [75]
Eine Architektur kann zu einem späterem Zeitpunkt auch aus Kundensicht wertvoll sein, diese ermöglicht es eine Wartung und Weiterentwicklung in akzeptabler Zeit durchzuführen. Dies erspart dem Kunden Folgekosten. [83]

Die Symptome einer Architekturerosion sind meist gleichzeitig dessen Folgen. Im Rahmen einer Untersuchung des *US Air Force Software Technology Support Centre* wurde die Entwicklung zweier Teams verglichen. Beide Teams mussten an der gleichen Software entwickeln. Der Unterschied war, dass das eine Team eine erodierte Version der Anwendung als Grundlage hatte und das andere eine nicht erodierte Version. Im direkten Vergleich wurde festgestellt, dass das Team mit der erodierten Version doppelt so lange für die Entwicklung von Features benötigte und bis zu einem achtfachen an Fehlern produzierte. [28]

Die Arbeiten von Martin [58] und Lilienthal [54] haben diese Symptome in fünf Kategorien unterteilt: Starrheit, Zerbrechlichkeit, Unbeweglichkeit, Zähigkeit und Hacks. Die *Starrheit* gegenüber Änderungen bedeutet, dass eigentlich kleine Änderungen viel Zeit in Anspruch nehmen. Dies kann unter anderem von vielen überflüssigen Abhängigkeiten und duplizierten Codestellen kommen, welche ebenfalls angepasst werden müssen. Ein zweiter Hinweis ist die *Zerbrechlichkeit*. Diese beschreibt das Auftreten von Fehlern an Stellen, welche eigentlich unabhängig von der Änderung sein sollten. Das Beheben der neuen Fehler, kann zu weiteren Fehlern führen. Probleme dieser Art können z.B. durch eine Verletzung der Schichten ausgelöst werden. Um eine *Unbeweglichkeit* handelt es sich, wenn es Stellen gibt die ein ähnliches Problem bereits lösen, diese aber nicht wiederverwendet werden können. Die vorhandenen

Lösungen sind so fest in das System integriert, welches eine Verallgemeinerung zu komplex und fehleranfällig machen würde. Eventuell würde eine erfolgreiche Verwendung ebenfalls die Schichten verletzen, dies weist auf eine fehlerhafte Aufteilung der Schichten hin. Eine Verschiebung der entsprechenden Methoden kann deshalb sinnvoll sein. Die *Zähigkeit* gegenüber Änderungen ist ein weiteres Symptom. Für Änderungen gibt es immer unterschiedliche Lösungen. Änderungen, welche dabei das Design zerstören, werden als *Hacks* bezeichnet. Wenn diese Hacks einfacher zu realisieren sind als die designerhaltenden Lösungen, ist die Anwendung zäh. [54, 58]
Ein starkes Auftreten dieser Symptome ist meist ein Hinweis dafür, dass die Architektur erneuert werden muss.

2.3.3 Architectural Smells

Für die zuvor beschriebenen Effekte auf die Entwicklung ist es möglich konkrete Gründe in der Implementierung zu finden. Diese können durch sogenannte *Architectural Smells* verursacht werden. Diese sind ähnlich zu den bekannten *Code Smells*. Stal [73] hat eine Liste mit ihm bekannten Architectural Smells ausgearbeitet. Da diese Liste sehr viele Möglichkeiten für Architectural Smells beinhaltet, wird hier nur eine gekürzte Auswahl in folgender Auflistung präsentiert.

Duplizierte Designartefakte Verschiedene Komponenten haben die gleichen Verantwortlichkeiten.

Komplexe Architektur Verursacht z.B. durch überflüssige Abstraktionen oder ausdruckslose Namensgebungen.

Überflüssige Individuallösungen Statt auf bewährte Lösungen und vorhandene Musters zu setzen, werden eigenständige Individuallösungen eingesetzt.

Zu generisches Design Zu viel Abstraktion und Generalisierung verringern die Wartbarkeit und das einfache Verständnis. Es sollten nur so wenige generische Lösungen eingesetzt werden, wie notwendig. Die Architektur sollte so genau wie möglich definiert sein.

Asymmetrische Strukturen Identische/ähnliche Probleme sollten immer durch die gleichen Muster gelöst werden und nicht durch unterschiedliche Lösungen realisiert werden.

Zyklische Abhängigkeiten Die Testbarkeit, Modifizierbarkeit und Aussagekraft wird verringert. Zum Modifizieren und Testen müssen meist alle Komponenten in der Reihe der Abhängigkeiten angepasst werden.

Unnötige und implizite Abhängigkeiten Um die Komplexität gering zu halten sollte die Anzahl der Abhängigkeiten möglichst gering gehalten werden. Das Hinzufügen von impliziten Abhängigkeiten sollte vermieden werden. Diese sind nicht in der entwickelten Architektur vorgesehen und werden häufig den anderen Entwicklern nicht mitgeteilt. Ersichtlich sind sie nur durch den Code. Dies kann bei Änderungen schnell die Lauffähigkeit der Anwendung beeinflussen. Ein Beispiel hierzu ist die Verwendung von globalen Variablen an unterschiedlichen Stellen.

Diese Liste stellt nur einen Ausschnitt möglicher Smells vor und lässt sich nahezu endlos verlängern. Die Smells müssen durch Analysen und Reviews gefunden und per Refaktorisierung (siehe Kapitel 2.4.2) beseitigt werden.

2.4 Architekturentwicklung in agilen Vorgehensmodellen

In diesem Abschnitt wird untersucht welche Aussagen das agile Manifesto und Vorgehen wie Scrum zur Architekturentwicklung machen. Ziel ist es herauszufinden inwiefern eine Architekturentwicklung bereits vorhanden ist oder wie diese erfolgreich integriert werden kann.
Anhänger der agilen Methoden sind häufig davon überzeugt, dass eine vorherige Architekturplanung und Evaluierung zu viel Zeit benötigt und keinen auslieferbaren Mehrwert für den Kunden bietet. Befürworter Architektur-zentrierter Methoden sind andererseits überzeugt, dass mit agilen Methoden nur kleine Anwendungen realisierbar sind. [1, 4, 53]
Wenn keine explizite Architektur entwickelt wird, entsteht eine zufällige und unstrukturierte Accidential Architecture [12].

Keine dieser beiden Ansichten stellt eine zufriedenstellende Aussage dar. Es muss eine Lösung erarbeitet werden, um die agile Entwicklung und die Architekturentwicklung erfolgreich in einem Verfahren zu vereinen. Dazu werden in diesem Abschnitt Aussagen und Hilfestellungen aus den vorhandenen Vorgehen analysiert.

Traditionelle Methoden sehen es vor, vor Beginn der Entwicklung ein vollständiges Design zu entwickeln. Die Architekturentwicklung findet in einer separaten Phase, unabhängig von der Entwicklung statt. Diese vollumfängliche Vorabplanung ist in dem agilen Denken vollständig abgeschafft. Das Entwerfen ist nicht explizit eingeplant. Bereits im Jahr 1968 hat Peter Naur auf einer der ersten Konferenzen zur Software Engineering (*NATO Software Engineering Conference*) gesagt, dass es keine bedeutenden Unterschiede im Vorgehen zwischen dem Designen und dem Implementieren gibt.

> *„In fact, there is no essential difference between design and production, since even the production will include decisions which will influence the performance of the software system, and thus properly belong in the design phase.“* [62]

Dies bedeutet, dass die Anwendung und die Architektur eigentlich gemeinsam entwickelt werden müssten und immer ein fester Bestandteil sein sollten. Das regelmäßige Feedback muss ebenfalls für eine gesteuerte Entwicklung der Architektur eingesetzt werden.

Dieser Abschnitt betrachtet Aussagen und Vorschläge, die das Agile Manifesto [43] und das Framework Scrum beinhalten. Es soll herausgefunden werden, ob es bereits einen Rahmen für Architekturdebatten gibt.

2.4.1 Agile Manifesto und das Framework Scrum

Unter einer Architekturentwicklung verstehen viele Entwickler ein großes Vorabdesign (*Big Up-Front Design - BUFD*) [1, 63, 85]. Dieses würde allerdings Prinzipien, wie *YAGNI - You Ain't Gonna Need It* widersprechen [53]. Deshalb wird häufig das Entwerfen, Evaluieren und Dokumentieren von Architekturen vernachlässigt. Architektur-zentrierte Ansätze werden häufig von Anhängern der agilen Vorgehen als Teil der traditionellen, Plan getriebenen Entwicklungsvorgehen angesehen. Begründet wird dies unter anderem dadurch, dass dies keine auslieferbaren Vorteile für den Kunden generiert. Gerade für große Softwareanwendungen ist es wichtig entsprechende Praktiken zu befolgen. Es wird meist davon ausgegangen, dass alle Designfragen durch Refaktorisierungen (siehe Abschnitt 2.4.2) gelöst werden können. [4]

Dieser Abschnitt soll darstellen, was dass agile Manifesto für Aussagen darüber trifft. Neben dem Scrum Framework werden im Folgenden weitere Entwicklungsvorgehen untersucht.

Agile Manifesto

Das *Agile Manifesto* macht keine konkreten Aussagen zur Architekturentwicklung. Wie im Grundprojekt[2] bereits festgestellt wurde widersprechen sich einige Punkte eher. Deshalb muss ein Gleichgewicht zwischen diesen gefunden werden. Aussagen, wie z.B. möglichst einfaches Design, durchgehende Beachtung von technischer Exzellenz und gutem Design fördern die Agilität, werden häufig so verstanden, dass keine Debatten notwendig sind. [43]

[2] http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master14-15-gsm/fausten/bericht.pdf, Abgerufen: 19.11.2016

Dies gilt ebenfalls für die Dokumentation einer Architektur. Diese wird gerne vernachlässigt. Der aktuelle Stand der Architektur ist dadurch häufig unbekannt. [37]

Scrum

Das Entwicklungsvorgehen Scrum setzt die Prinzipien des agilen Manifesto um. Es ist das am weitesten verbreitete agile Vorgehensmodell. 58% aller agilen Projekten werden mit Scrum durchgeführt. Weitere 10% setzten eine Mischung aus Scrum und XP ein. Insgesamt wird

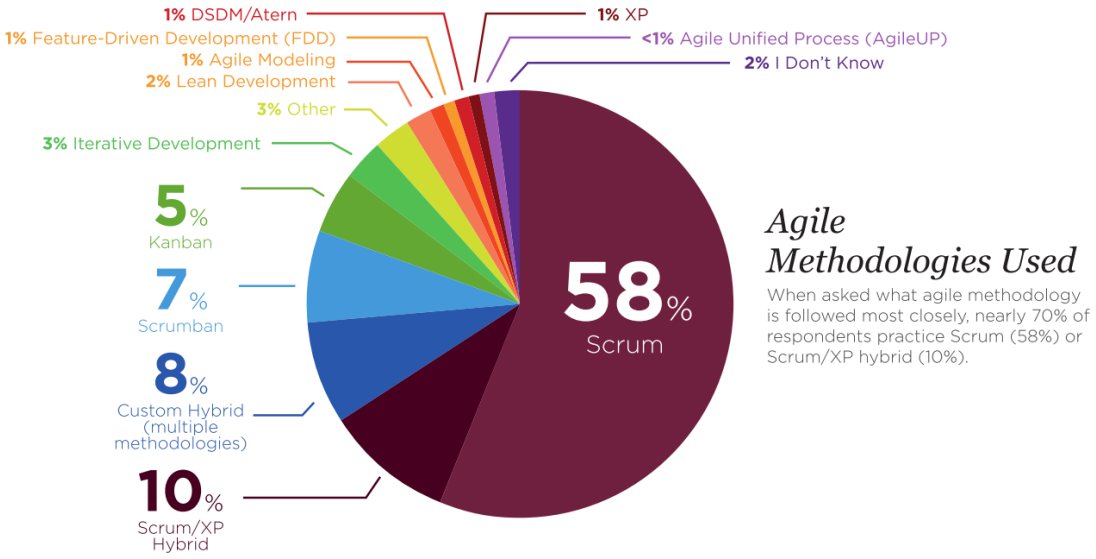


Abbildung 2.3: Verbreitung einzelner agiler Vorgehen [81]

zusammengefasst in circa 70% der Projekte Scrum eingesetzt. Die Verbreitung der Methoden ist in Abbildung 2.3 zu sehen. [81] Aufgrund der starken praktischen Präsens steht dieses Vorgehen im Fokus dieser Arbeit.

Die technische Entwicklung einer Architektur wird auch in einem konkreten Vorgehen wie Scrum wenig beachtet. In Abbildung 2.4 ist der Scrum Zyklus dargestellt. Bei genauer Betrachtung ist zu erkennen, dass Scrum rein Feature-basiert ist. Dies ist auch daran gut festzustellen, dass jede Story mit einem Ziel für den Benutzer geschrieben ist. Details zur Umsetzung werden bewusst ausgelassen.

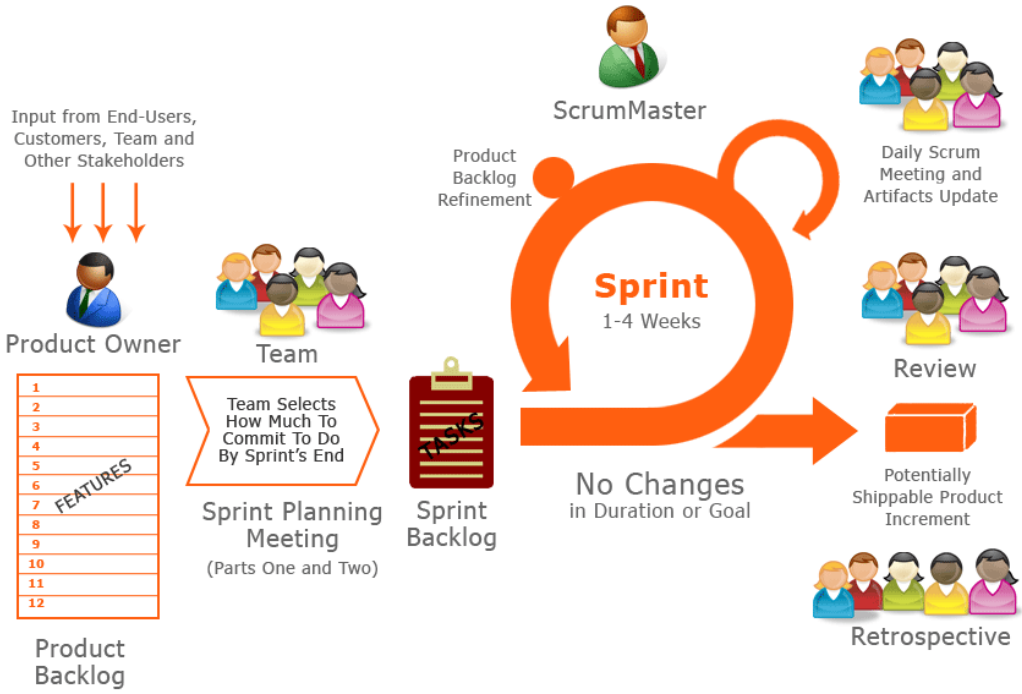


Abbildung 2.4: Scrum Zyklus [16]

Im *Scrum Guide* [71] ist vorgesehen, dass im zweiten Abschnitt des Sprint Plannings besprochen wird, wie die Arbeit erledigt wird. Dazu gehört ebenfalls ein Systementwurf: *„Das Entwicklungsteam beginnt normalerweise mit dem Systementwurf [...]“* [71]. Dieser kann bereits eine Basisarchitektur für den folgenden Sprint darstellen. In der Praxis ist dieser Abschnitt sehr unterschiedlich umgesetzt, wie in einem anschließendem Beispiel zu erkennen ist. Häufig werden die Stories hier in kleinere Teilaufgaben unterteilt. Dabei kann es sich auch um technische Aufgaben handeln. Ein Scrum Projekt kann aber auch vollständig ohne technische Debatten durchgeführt werden. Dies ist vor allem der Fall, wenn bereits eine Basisarchitektur vorhanden ist. Diese kann z.B. von einem bereits vorhandenem System stammen. Weitere technische Debatten sind nicht vorgesehen. Alle Themen, die die konkrete Realisierung betreffen, werden nicht exakt geplant und nicht im Nachhinein nochmals überprüft.

Ebenso ist dies personell, im Rahmen eines expliziten Softwarearchitekten nicht vorgesehen:

> *"Scrum kennt keine weiteren Unterteilungen innerhalb des Entwicklungsteams, ungeachtet von bestimmten zu adressierenden Domänen, wie „Test" oder „Analyse"."* [71]

Andererseits sagt der Scrum Guide: *„Individuelle Mitglieder des Entwicklungsteams können zwar spezialisierte Fähigkeiten oder Spezialgebiete haben, aber die Rechenschaftspflicht obliegt dem Team als Ganzem."* [71]. Dadurch wird es trotzdem möglich solch eine Rolle einzuführen. Diese Person darf keine anderen Rechte und Pflichten als die anderen Entwickler haben. Kann aber als beratende Person und als Leiter von Architekturdebatten eingeführt werden (siehe Kapitel 4.6).

Der Guide macht ebenfalls keinerlei Aussagen zu einer Dokumentation der Architektur. Dadurch wird der Ist-Zustand meist nur durch den Sourcecode beschrieben. Dies ist problematisch, weil viele Strukturen und Muster nicht direkt im Quellcode ersichtlich sind. Viele Architektureigenschaften werden z.B. nur durch die Kommunikation der einzelnen Komponenten dargestellt. Einen Überblick dabei zu behalten ist sehr schwer. Eine eindeutige Erkennung, der Komponenten ist ebenfalls, je nach Programmiersprache, nicht eindeutig möglich. Bei der Sprach PHP ist z.B. durch die dynamische Typisierung erst zur Laufzeit erkennbar, welche Objekte in eine Methode eingereicht werden. Das Gegenteil zu keiner Dokumentation ist eine sehr lange, komplexe Dokumentation. Diese ist ebenfalls nicht wünschenswert. Möglichkeiten zur Erstellung einer leichtgewichtigen Dokumentation werden in Kapitel 4.7 vorgestellt. Scrum sieht alleine vor, dass eine Architektur durch regelmäßige mündliche Kommunikation und Refaktorisierungen entsteht.

Das *Fraunhofer-Institut für Experimentelles Software Engineering* in Kaiserslautern hat in einer Untersuchung mithilfe des Standard Scrum festgestellt, dass diese zweite Phase zum Entwerfen nicht ausreicht. Das Forschungsteam bestand aus drei Wissenschaftlern und drei erfahrenen Entwicklern. Zu Forschungszwecken wurde eine Anwendung entwickelt. Im Laufe der Zeit dauerten neue Stories und Features erheblich länger und erforderten komplexe Änderungen an immer mehr unterschiedlichen Stellen im Code. Dies hatte außerdem zur Folge, dass die Aufwandsschätzungen immer ungenauer wurden. [83] Diese Symptome können, wie zuvor beschrieben (Kapitel 2.3.2) der Architekturerosion zugeordnet werden.

Weitere Vorgehen

Neben Scrum existieren zahlreiche weitere agile Entwicklungsvorgehen. Im Rahmen dieses Abschnittes werden Aussagen ausgewählter, weiterer Vorgehen zusammengefasst vorgestellt.

Nach Scrum und dem Scrum/XP Hybrid sind individuelle Hybride (8% Verbreitung), Scrumban (7% Verbreitung) und Kanban (5% Verbreitung) am meisten in der Praxis im Einsatz. In diesem Abschnitt werden aus Gründen der Verbreitung XP und Kanban genauer betrachtet. Extreme Programming (XP) alleine hat nur eine Verbreitung von 1%, soll aber wegen der häufigen Kombination mit Scrum zusätzlich untersucht werden. [81]

Extreme Programming Bei Extreme Programming, bzw. XP ist ebenfalls der Ansatz des möglichst einfachen Designs, wie bei Scrum, gewählt worden. Dieses Vorgehen arbeitet auch mit Feature-basierten Stories. Vorgaben, wie etwas umgesetzt werden soll, existieren deshalb nicht. Der Entwurf der Anwendung soll durch häufige Refaktorisierungen erzeugt und verbessert werden. [4]
Indem nach potenziell schlechten Codestellen gesucht wird, soll die Architektur durch ein tägliches designen entstehen. Die Codestellen sollen, wenn notwendig ausgebessert werden. Eine gute Architektur soll dadurch nicht durch einen geleiteten Prozess, sondern durch das entwickelte System selbst entstehen. [20]
XP bevorzugt die direkte Kommunikation, statt der Kommunikation über Dokumente. Dadurch wird eine vollständige Dokumentation der Architektur vernachlässigt. Ebenfalls werden Anforderungen an Stories meist nicht niedergeschrieben, sondern sind nur in den Köpfen der Entwickler vorhanden. Dies führt schnell zu Missverständnissen und macht es schwierig Entscheidungen für die Umsetzung nachvollziehen zu können. Besonders zutreffend ist dies für Entwickler die nicht an der Umsetzung beteiligt waren. [77]

Zu den XP Prinzipien gehört es eine *System Metaphor* zu wählen. Damit ist gemeint, dass ein einfaches Design mit verschiedenen Qualitätsansprüchen festgelegt wird. Es soll eine Struktur gewählt werden, die es neuen Personen ermöglicht sich schnell ohne große Dokumente einzuarbeiten. Außerdem sollen einheitliche Namensgebungen für Klassen und Methoden festgelegt werden. [84]
Dieses Prinzip ist allerdings nicht verpflichtend bei der Verwendung von XP. Häufig ist es schwer eine einfache System Metaphor zu finden und wird deshalb gerne ausgelassen. [77]

Kanban Kanban ist darauf ausgelegt den Arbeitsprozess zu visualisieren, um ihn anschließend zu optimieren. Kanban macht ansonsten keine weiteren Vorgaben, die die Architekturentwicklung betreffen. Stattdessen ist dieses Vorgehen vollständig auf eine schrittweise Optimierung des Arbeitsprozesses ausgelegt.

Weitere Bei weiterer Betrachtung anderer agiler Vorgehen ist festzustellen, dass diese ebenfalls wenig, bis keine Angaben zur Architektur machen. In den agilen Prozessen stehen die Entwickler im Vordergrund. Dies entspricht dem Prinzip *„individuals and interactions over processes and tools“* [43].
Die Architektur ist eher eine Nebenbetrachtung, die indirekt mit der entstehenden Software wächst und dessen Qualität durch Refaktorisierungen erreicht werden soll. Auf die Möglichkeiten und Einschränkungen von Refaktorisierungen wird im nächsten Abschnitt genauer eingegangen.

2.4.2 Refaktorisierung

Refaktorisierung (engl. refactoring) ist ein fester und wichtiger Bestandteil in den agilen Vorgehensmodellen. Es beschreibt die Änderung von Code, ohne dessen funktionales Verhalten nach außen zu ändern. Es ist hilfreich um ein unkontrolliertes Wachsen der Anwendung und seiner Architektur zu kontrollieren und zu korrigieren. Ein Wachstum entsteht durch ungeplante Änderungen, wie z.B. geänderte Anforderungen, geänderte Infrastruktur, gefundene Bugs oder fehlerhaft getroffene Entscheidungen. Durch die Änderungen wird die Anwendung an nicht vorgesehenen Stellen erweitert. Solche Änderungen sind in den agilen Vorgehen vorgesehen und stellen keine Ausnahme dar, sollten aber immer anschließende Refaktorisierungen zur Folge haben. [73]
Um die entstehenden Probleme möglichst gering zu halten sollten Prinzipien, wie kein Vorausplanen beachtet werden. Dies bedeutet, dass keine Vorhersagen für die Zukunft gemacht werden sollten. Die Entwicklung sollte auf aktuellem Wissen beruhen und dadurch möglichst simpel gehalten werden. Die agilen Prinzipien werden in Kapitel 4.4 genauer beschrieben.

Es ist allerdings wichtig, zwischen Code- und Architektur-Refaktorisierung zu unterscheiden. Die Code-Refaktorisierung kann ebenfalls die Komplexität verringern und die Wartbarkeit erhöhen [73]. Um allerdings Eigenschaften, die die gesamte Skalierbarkeit, Performance, Testbarkeit oder Robustheit betreffen zu beeinflussen, muss eine Architektur-Refaktorisierung durchgeführt werden. Diese kann die unterschiedlichsten Ebenen der Anwendung einbeziehen. Eine Architektur-Refaktorisierung kann ebenfalls für die Wartbarkeit und Erweiterbarkeit notwendig sein. Code-Refaktorisierung betrifft z.B. das Optimieren oder Auslagern einzelner Methoden und dem Korrigieren von Fehlern. Bei einer Refaktorisierung der Architektur müssen meist breite Teile der Anwendung mit einbezogen werden. Dies kann sowohl mehrere Komponenten, den Kontrollfluss, wie auch unterschiedliche Schichten betreffen. Buschmann hat dies passenderweise folgendermaßen formuliert:

> *"Refactoring isn't limited to code detail but can range up to the larger scale of a system's software architecture."* [17]

Das kann zur Folge haben, dass während der Durchführung nicht direkt beteiligte Entwickler blockiert werden. Es kommt dadurch nicht zu einem Stillstand der Entwicklung. Ein Fortschritt des Projektes ist dabei aus Anwendersicht aber nicht festzustellen, da keine neuen Features entwickelt werden. [9, 31]
Eine Architektur-Refaktorisierung beinhaltet im Normalfall eine Code-Refaktorisierung. Der Kunde und das Management sehen keinen direkten Fortschritt, deshalb werden diese schnell unzufrieden. Ein Grund dafür kann ein falsches Verständnis, wie z.B. die Unterscheidung zwischen Code- und Architektur-Refaktorisierung sein [73].

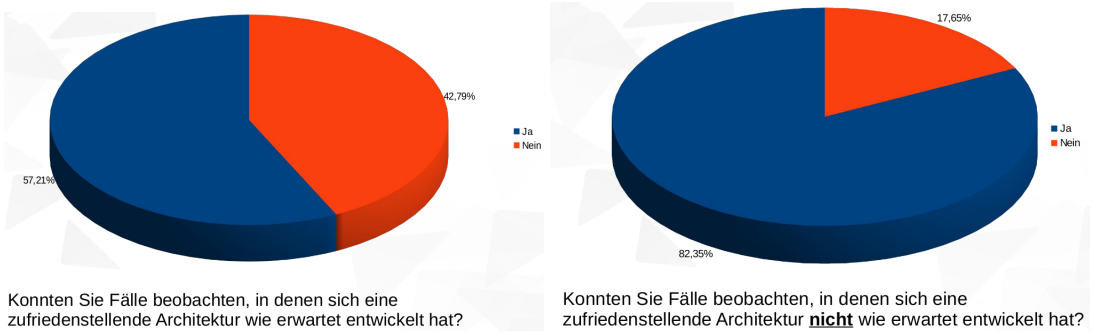
Ein bereits erwähntes Problem ist, dass unter Architekturplanung und Architekturdebatten häufig ein großes Vorabdesign verstanden wird. Dies ist nicht im Einklang mit den agilen Werten und Prinzipien (z.B. Einfachheit und kleine Schritte). Da weder das Manifesto, noch Beschreibungen der Vorgehen (z.B. Scrum Guide) dieses Thema behandeln, sind einige agile Befürworter davon überzeugt, dass ein alleiniges Refaktorisieren ausreicht, um eine akzeptable Architektur zu entwickeln. [20, 63]

Die Praxis zeigt allerdings häufig das Gegenteil. Forscher und Nutzer der agilen Methoden stehen dieser Einstellung ebenfalls sehr skeptisch gegenüber. Meyer sagt dazu: *"refactored junk is still junk"* [60]. Refaktorisierung ist eine wichtige Technik der agilen Entwicklung, ersetzt aber nicht ein vorheriges Designen. Diese Problematik hat Dijkstra bereits 1968 erkannt:

> *„However, I am convinced that the quality of the product can never be established afterwards."* [62]

Diese frühe Erwähnung zeigt, dass dies eigentlich kein neues Problem ist und schon vor der agilen Bewegung (circa 1990, [42]) ein Thema bei der Entwicklung war. Die erste Version des Manifesto wurde zum Vergleich 2001 veröffentlicht. Im Rahmen der traditionellen Methoden wurde viel Wert auf Architektur gelegt, welcher sich in den agilen Methoden verringert hat. Beim Wasserfallmodell war die Entwurfsphase zum Planen der Architektur vorgesehen.

In der Praxis existieren viele Projekte bei denen es zu Problemen aufgrund von zu wenig Beachtung der Architektur gekommen ist. Im Rahmen einer Forschung haben Chen u.a. [20] eine Umfrage mit 102 Personen durchgeführt, um deren Erfahrungen in Bezug auf reines Refaktorisieren ohne Debatten zu ermitteln. Bei der Auswahl der Teilnehmer wurde darauf geachtet, dass diese bereits ausreichend Erfahrung mit der agilen und der Architektur-Entwicklung gesammelt haben. Außerdem wurde darauf geachtet, dass die Personen nicht das gleiche Arbeitsfeld



(a) (b)

Abbildung 2.5: Architektur-Refaktorisierung [20]

haben. Aus insgesamt 13 Ländern wurden Personen aus Firmen unterschiedlicher Größe ausgewählt. Ihnen wurden hauptsächlich zwei Fragen gestellt. Durch diese sollte ermittelt werden, ob die Personen Projekte erlebt hatten, in denen sich eine zufriedenstellende Architektur allein durch Refaktorisierung entwickelt hat. Im zweiten Teil wurde die Fragestellung umgedreht und nach Projekten gefragt, die nur durch Refaktorisierung keine zufriedenstellende Architektur entwickelt hatten. In Abbildung 2.5 sind beiden Fragen und die Antworten nochmals visuell dargestellt.

57,21% konnten erfolgreich von einem Projekt berichten in dem die Architektur nur durch Refaktorisierung entstanden ist. Allerdings konnten auch 82,2% von Erfahrung aus einem Projekt berichten in dem dies nicht der Fall war. Bei der kontextuellen Betrachtung der Fragen konnte festgestellt werden, dass 59% der Befragten beide Fälle erlebt haben. 23% konnten von keinem erfolgreichem Fall berichten. 9% kannten nur Erfolgsfälle. Das bedeutet, dass es unter Umständen möglich ist eine erfolgreiche Architektur durch alleiniges Refaktorisieren zu entwickeln. In mehr als 80% der Fälle funktioniert dies allerdings nicht. Es wird ein hohes Risiko eingegangen, welches die wenigsten Personen und Firmen eingehen möchten.

Insgesamt 23 Faktoren, die die Architekturentwicklung direkt, wie auch indirekt beeinflussen können, haben sie identifizieren. Die Einflussfaktoren wurden in fünf Kategorien unterteilt:

Projekt z.B. Anzahl der Änderungen, Größe, System Alter, [...]

Unternehmen z.B. Kultur, Management Unterstützung, [...]

Team z.B. Erfahrung, Größe, Verteilung, [...]

Praktiken Continious Integration, Design Prinzipien, Sicherheitsnetz

Bei einer genauen Betrachtung der Faktoren kann festgestellt werden, dass diese nur teilweise durch die Entwickler selbst beeinflussbar sind. Der Bereich in dem sie den größten Einfluss haben, sind die Praktiken. Aber auch hier muss dies vom Management und der Projektleitung (oder anderen Verantwortlichen) unterstützt werden. Durch den Einsatz von Continious Intergration (CI) ist eine hohe Testabdeckung notwendig (empfohlen sind über 90%). Die hohe Testabdeckung stellt das Sicherheitsnetz für die Entwickler dar. Wenn nach einer Änderung die Tests nach wie vor erfolgreich durchlaufen, können sich die Entwickler einigermaßen sicher sein, dass ihr Refaktorisierungsschritt keine neuen fachlichen Probleme verursacht hat. [20] Eine genauere Untersuchung der Faktoren für die Refaktorisierung wird in dieser Arbeit nicht durchgeführt.

2.5 Sozio-technische Kongruenz

Es existieren verschiedene Einflussfaktoren auf die Architekturentwicklung. Ein sehr großer Faktor ist das Entwicklungsteam. Das Team kann die Architektur sowohl positiv, wie auch negativ beeinflussen.
Die Architektur ermöglicht es die Entwicklung in Teams aufzuteilen. Dadurch entstehen z.B. Infrastruktur und Feature Teams. Die Hauptidee für die *Sozio-technische Kongruenz (STC - Socio-Technical Congruence)* stammt bereits von 1968 von Conway [25] und wird als *Conway's Law* bezeichnet. Diese sagt aus, dass die Struktur einer entwickelten Anwendung meist die Organisationsstruktur des Unternehmens abbildet. Wie im Namen zu erkennen ist, besteht diese Thematik aus einer technischen und einer sozialen Komponente. Die soziale Komponente betrachtetet die Organisation und die Individuen, die mit dem Entwicklungsprozess in Kontakt stehen. Hierzu zählt neben dem Entwicklerteam und dem gesamtem Unternehmen auch der Kunde. Alle Gruppen können einen Einfluss auf den Aufbau einer Anwendung haben. Die technische Seite behandelt unter anderem die Entwicklungsaufgaben und Technologien. Diese beiden Komponenten müssen für ein erfolgreiches Projekt im Einklang sein. [18]

Die STC kann als Untersuchung der Koordination und der damit verbundenen Kommunikation betrachtet werden. Durch Untersuchungen dieser Art soll herausgefunden werden, wie die Koordination der Teams verbessert werden kann. [69]

Konflikte werden verringert, wenn eine hohe STC besteht. Die Teams sind dadurch voneinander unabhängiger. Dies unterstützt die agile Arbeitsweise [63]. In der Theorie wird außerdem eine Steigerung der Produktivität und der Qualität erwartet. Andererseits können aber auch

erhöhte Risiken und größere Kosten die Folgen sein [69]. Dies erfordert, dass eine gute Balance zwischen den Vor- und Nachteilen gefunden wird.

2.6 Architekturerosion steuern

Damit die Architektur zu einer langlebigen Architektur wird, sollte die Entwicklung dauerhaft überwacht werden. Dazu muss herausgefunden werden, wie automatisiert eine Erosion gefunden werden kann. Um die Erosion erkennen zu können, muss der aktuelle Zustand bekannt sein. Dies kann durch den Ansatz des automatisierten Reengineerings angegangen werden. Ein Vergleich des Soll- und Ist-Zustandes soll eine mögliche Lücke zwischen ihnen aufdecken. Allgemeine, nicht anwendungsspezifische Probleme können zum Großteil bereits durch allgemeine Metriken erkannt werden. Für diese Erkennung ist die Ist-Architektur nicht erforderlich. [75]

Ansätze zum Reengineering mit dem Ziel eine Architekturdokumentation zu erstellen und ein Monitoring zu realisieren, werden in Kapitel 3 genauer vorgestellt.

Kapitel 4 diskutiert Möglichkeiten, um während der Entwicklung die Architekturerosion möglichst gering zu halten. Um eine Erosion von vorn herein gering zu halten ist es notwendig einen Raum zur Diskussion über die Architektur zu schaffen. Dazu werden Ansätze für eine Integration von technischen Architekturdebatten in agilen Prozessen diskutiert.

Sobald eine Erosion erkannt wurde, muss eine Refaktorisierung durchgeführt werden (siehe Kapitel 2.4.2). Je später eine notwendige Refaktorisierung erkannt wird, desto komplexer kann diese werden.

2.7 Fazit

Die dargestellte Problematik der fehlenden Architekturdebatten und damit verbundenen Auswirkungen macht deutlich, dass dies ein relevantes Problem innerhalb der agilen Entwicklung ist. Es konnte festgestellt werden, dass zuerst ein gemeinsames Verständnis für Softwarearchitekturen geschaffen werden muss. Dies ermöglicht eine gleiche Kommunikationsbasis für alle Teammitglieder. Außerdem muss vor Beginn eines Projekts geklärt werden, wie relevant eine Architektur für das konkrete Projekt ist.
Für ein dauerhaft erfolgreiches Projekt, mit einer langlebigen Architektur, ist es wichtig die Architektur zielgerichtet, mithilfe von Debatten zu entwickeln und dauerhaft zu verbessern.

Wenn keine explizite Architektur vorhanden ist, kann es schnell zu unvorhersehbaren Problemen und einem Scheitern des Projektes kommen. Die mit der Zeit immer komplexer werdende Erosion stellt ein hohes Risiko für ein Scheitern dar. Besonders die Weiterentwicklung und Wartung einer Anwendung ist davon betroffen. Dies kann ernsthafte Folgen für die beteiligten Unternehmen haben.
Vorhandene agile Vorgehen und das Manifesto machen keine konkreten Aussagen, wie eine Architektur innerhalb des Prozesses entwickelt werden kann. Diese sind eher auf die Optimierung der Arbeitsprozesse ausgelegt. Vorgaben, wie die konkrete Realisierung aussehen oder wie ein Plan dafür entwickelt werden kann, werden bewusst ausgelassen. Ein Rahmen für Verbesserungen wird durch die agilen Prozesse angeboten, ohne deren Prinzipien zu widersprechen. Refaktorisierungen sind ebenfalls bei Projekten mit Architekturdebatten zwingend erforderlich. Nur so kann auf Änderungen reagiert und trotzdem eine hohe Codequalität erreicht werden. Die Architektur hängt sehr von der Struktur und Organisation der Entwicklung und des Unternehmens ab (Kapitel 2.5). Aus diesem Grund ist es eine sehr wichtige Bedingung, dass das Unternehmen die Entwicklung vollständig unterstützt und entsprechende Strukturen anbietet. Die Strukturen dürfen nicht zu starr sein und sollten eventuell individuell an die jeweiligen Projekte angepasst werden können.
Es konnte allerdings festgestellt werden, dass es auch ohne eine Architekturdebatte möglich ist, ein Projekt erfolgreich abzuschließen. Die eigentliche Entwicklung läuft dabei ohne größere Probleme ab. Im Anschluss, während der Wartung und Weiterentwicklung, entstehen allerdings häufig Probleme. Diese verzögern und erschweren die Entwicklung unnötig. Im nächsten Kapitel wird untersucht, wie Analysen auf vorhandenem Quellcode durchgeführt werden können. Diese sollen Strukturen sichtbar machen und mögliche Probleme finden. Die Weiterentwicklung soll dadurch unterstützt werden.

3 Bottom-Up: Automatische Architektur Prüfung und Reengineering

Dieses Kapitel beschreibt Analysemöglichkeiten auf Basis des vorhandenen Quellcodes. Der Quellcode wird verwendet, um Strukturen zu rekonstruieren und Metriken zu berechnen. Soweit wie möglich sollen die Strukturen und Metriken mithilfe von automatisierten Werkzeugen berechnet werden. Die Metriken können auf vorhandene Defekte hinweisen. Bei den Defekten kann es sich um unbekannte Architekturerosionen handeln. Durch die Erkennung und Auflösung wird die Softwarequalität deutlich angehoben.

Zu Beginn dieses Kapitels werden die allgemeinen Ziele dieses Ansatzes erläutert. Da dieser Ansatz die Weiterentwicklung und Wartung als wichtiges Einsatzgebiet hat, werden mögliche Szenarien für die Arbeit an einer vorhandenen Software vorgestellt. Im folgenden Abschnitt werden unterschiedliche Techniken und deren Einsatzmöglichkeiten zur Durchführung von Analysen präsentiert. Während der Entwicklung kann ein Monitoring die Architekturentwicklung unterstützen. Dies wird im anschließendem Ansatz erläutert. Der nächste Abschnitt beschreibt ein Beispiel aus der Praxis, in dem eine Architektur Ermittlung mit Conformance Check durchgeführt wird.
Am Ende dieses Kapitels wird ein Fazit zu den gewonnen Erkenntnissen gezogen.

3.1 Ziele

Der Bottom-Up Ansatz hat das Ziel aus vorhandenem Quellcode eine Architektur zu reengineeren. Strukturen sollen sichtbar und Probleme erkannt werden können.

Dies ist sinnvoll, da in vielen Projekten keine aktuelle Architekturdokumentation vorhanden ist [70]. Der Ist-Stand ist deshalb häufig unbekannt. Eine Anwendung muss während ihres Lebenszyklus dauerhaft gewartet werden. Dazu ist erforderlich den aktuellen Stand und Aufbau zu kennen. Ansätze, die die Strukturen einer Anwendung automatisch oder halbautomatisch extrahieren, können das Verständnis über die Anwendung deutlich erhöhen. Dieser Ansatz

wird immer wichtiger, weil die meisten Anwendungen immer länger eingesetzt werden und immer komplexer werden [7]. [30]
Die Wunschvorstellung ist, dass die reengineerte Architektur als Ersatz für eine fehlende oder veraltete manuelle Dokumentation dienen kann. Die Gründe dafür, dass keine aktuelle Dokumentation vorhanden ist, können vielseitig sein. Diese muss immer manuell und zeitnah zu den Änderungen gepflegt und weiterentwickelt werden, ansonsten geraten Details und Gründe schnell in Vergessenheit. Die Anfertigung ist meistens mühselig und eine unbeliebte Aufgabe des Teams. Oft ist nicht bekannt, wie solch eine Dokumentation aussehen soll und liefert im Moment der Anfertigung keinen merkbaren Mehrwert.

Die reengineerte Architekturdokumentation kann dazu dienen Punkte in der Anwendung zu identifizieren, an denen für eine erfolgreiche Weiterentwicklung oder Wartung angesetzt werden muss. Vorteil einer digitalen automatisierten Dokumentation ist, dass diese immer auf dem aktuellen Stand beruht und auch im Nachhinein noch generiert werden kann. Ein weiterer Punkt ist, dass von einer sehr abstrakten Sicht mit Blick auf die gesamte Anwendung, bis hin zu direkten Implementierungsdetails hineingeschaut werden kann. Dies ermöglicht es von einer feinen Ansicht, mit vielen Details, zu einer groben Ansicht, mit allgemeinen Strukturen zu wechseln. Dies kann so realisiert sein, dass in der allgemeinen Ansicht die Basiskomponenten sichtbar sind. Im zweiten Schritt sind z.B. die Strukturen und Komponenten innerhalb einer Basiskomponente sichtbar. In der feinsten Auflösung wird der Quellcode angezeigt.
Das gesamte Verfahren ist allerdings nicht ohne zusätzliche manuelle Arbeit möglich, wie dies für eine Architekturdokumentation erforderlich wäre. Im Rahmen des Grundprojektes[1] wurden zahlreiche Anwendungen und Tools vorgestellt, die versuchen einen Teil dieses Problems zu lösen. Anschließend konnte in einem Selbstversuchen im Hauptprojekt[2] festgestellt werden, dass gerade das Finden von Ansatzpunkten für Erweiterungen nur durch die generierte Dokumentation, bzw. Tool-Artefakte problematisch ist. Es müssen trotzdem weiterhin grobe, bis detaillierte Kenntnisse über die Anwendung und dessen Aufbau vorhanden sein. Viele Tools bieten allerdings bereits funktionierende Möglichkeiten, um Metriken aus einer Anwendung zu extrahieren und zu visualisieren. Einige der Tools konnten automatisiert Entwurfsmuster erkennen. Diese mussten allerdings zuvor manuell definiert werden. Die Definition dieser Muster kann komplex sein. Des Weiteren war die Erkennung häufig fehlerhaft. Es konnten nicht alle Muster erkannt werden. Zusätzliche wurden Muster fälschlicherweise erkannt.

[1] http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2015-proj/fausten.pdf, Abgerufen: 06.08.2016
[2] http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2016-proj/fausten.pdf, Abgerufen: 06.08.2016

Ein weiteres Ziel ist ein Einsatz zum Monitoring. Das Monitoring kann bereits von Beginn an während der Entwicklung eingesetzt werden. Dadurch soll eine dauerhaft gute Codequalität sichergestellt werden. Unbewusste Verletzungen, z.B. beim Codestyle oder Schichtenverletzungen, sollen erkannt werden. Ein Monitoring mit einer einfachen Visualisierung ermöglicht es, dass Änderungen direkt sichtbar werden und alle Teammitglieder ein besseres Bewusstsein für den aktuellen Stand haben.

3.2 Szenarien zur Wiederaufnahme von Projekten

In vielen Fällen müssen zu späteren Zeitpunkten Änderungen an bereits existierenden Anwendungen vorgenommen werden. Um Änderungen erfolgreich durchzuführen ist es notwendig, dass der Aufbau der Anwendung bekannt ist. Für die Weiterentwicklung einer seit längerer Zeit nicht veränderten Anwendung wurden drei vorstellbare, unterschiedliche Szenarien entwickelt. Die Szenarien stellen die Wiederaufnahme eines alten Projekts nach einer längeren Entwicklungspause dar.

Szenario A: Der Entwickler hat früher selbst an der Anwendung gearbeitet. Er kennt die Strukturen und den Aufbau allerdings nur noch grob. Ihm ist es wahrscheinlich schnell möglich sich wieder einzuarbeiten und sich an frühere Beweggründe zu erinnern.

Szenario B: Dem Entwickler ist die Anwendung fremd. Er hat aber die Möglichkeit die ursprünglichen Entwickler um Rat zu fragen. Dadurch ist ihm eine, durch die ansprechbaren Entwickler geleitete, Einarbeitung möglich. Während der Entwicklung kann er bei Fragen um Unterstützung bitten. Hierbei ist es von Vorteil, wenn die alten Entwickler in der räumlich gleichen Umgebung wie die neuen arbeiten.

Szenario C: Dem Entwickler ist die Anwendung fremd. Er hat allerdings nicht die Möglichkeit die ursprünglichen Entwickler um Rat zu fragen. Diese können z.B. nicht mehr erreicht werden, weil diese in anderen Firmen arbeiten oder unbekannt sind. Der Entwickler ist vollständig auf sich gestellt.

In jedem der Fälle ist es notwendig die Architektur der Anwendung zu kennen, um die Stellen zu finden an denen für bestimmte Weiterentwicklungen angesetzt werden muss. Durch die Kenntnis der Architektur kann außerdem verhindert werden, dass Entscheidungen getroffen werden, die die Architektur verletzen und dadurch die Erosion erhöhen. Wie bereits zuvor festgestellt, können diese Probleme ein Unternehmen in eine Krise führen, wenn eine Weiterentwicklung nicht mehr unter realistischen Umständen möglich ist. Diese Problematik kann

durch Wissen über die aktuelle Architektur verhindert werden. Ein Wissenstransfer an alle beteiligten Personen ist deshalb notwendig. Wenn keine aktuelle oder nur eine unvollständige Dokumentation dieser existiert, ist der Sourcecode die einzige Beschreibung der Architektur. Einige Strukturen sind allerdings nicht direkt im Code ersichtlich und müssen dadurch mühsam analysiert werden. Um diese Problematik zu verringern werden im nächsten Schritt Analyseverfahren vorgestellt, die in den darauf folgenden Analysemöglichkeiten verwendet werden.

3.3 Analysetechniken

Bei den Analysetechniken muss grundsätzliche zwischen statischen und dynamischen Verfahren unterschieden werden. Statische Analysen werden auf der reinen Codebasis, den kompilierten Dateien und eventuell vorhandenen Metadaten durchgeführt. Es kommt dabei zu keiner Ausführung der Anwendung. Eine automatisierte Ausführung ist keine Voraussetzung für diese Art von Analysen. Manuelle Reviews gehören z.B. ebenfalls zu den statischen Analysen [46]. Da es zu keiner Ausführung der Anwendung kommt, sind statische Analysen sehr performant und deshalb in kurzer Zeit durchführbar [46]. Beim aktivem Einsatz während der Entwicklung ermöglicht die schnelle Durchführung früh Feedback zu erhalten [29]. Der Großteil der existierenden Reengineering-Anwendungen verwendet das statische Vorgehen.
Bei dynamischen Analysen wird die Anwendung zur Laufzeit betrachtet. Dies ist erforderlich, wenn Abhängigkeiten erst zur Laufzeit gebunden werden und sich durch verschiedene Ausführungsstränge unterscheiden können. Implizite Kommunikation oder veränderbare Referenzen können dies ermöglichen [2]. Dynamische Analysen sind nochmals komplexer als statische Analysen, weil der Daten- und Kontrollfluss der Anwendung beobachtet werden muss. Dadurch erhalten diese Analysen eine zusätzliche Zeitkomponente. Für solch eine Analyse muss jeder potentielle Ausführungspfad durchlaufen werden. Dieses Verfahren weist dadurch starke Performance Defizite auf und ist deshalb nicht gut skalierbar. [36]

Um eine Architektur zu erfassen kann zusätzlich auf unterschiedliche Arten vorgegangen werden. Zwei Varianten sind das Cluster-basierte und das Pattern-basierte Reengineering. Beide Ansätze schließen sich nicht gegenseitig aus. Beim Cluster-basierten Ansatz werden Basiselemente, wie z.B. Klassen, anhand der Werte von speziellen Codemetriken gruppiert. Die Metriken werden in der Regel durch statische Analysen berechnet. Eine Metrik kann z.B. die Kopplung sein. Dadurch kann eine Reihe von Komponenten, Subkomponenten und deren Verbindungen erkannt werden. Die Berechnung der Metriken ist meist recht performant

und eignet sich deshalb auch für große Systeme. Ein Nachteil ist, dass einige komplexe Informationen nicht durch Metriken ausgedrückt werden können. Hierbei kann es sich z.B. um Bindungen handeln, die erst zur Laufzeit feststehen. Gründe für diese Strukturen können nicht automatisch erfasst werden und müssen deshalb manuell nachgetragen werden. [30] Diese Variante wird von einem Großteil der Werkzeuge umgesetzt.
Der Pattern-basierte Ansatz sucht nach zuvor definierten Struktur- und Verhaltensmustern. Dies können z.B. Entwurfsmuster sein. Der Vorteil bei der Erkennung von Mustern ist, dass diese bereits eine Intention besitzen. Dadurch kann deren Einsatzzweck schneller erkannt werden und ist deshalb nützlicher bei einer fehlenden oder unvollständigen Dokumentation. Neben der Suche nach gewollten Pattern, kann auch nach Anti-Pattern gesucht werden, um unerwünschte Strukturen zu finden. Für die Suche nach Pattern muss häufig der Kontroll- und Datenfluss bekannt sein. Dies ist nur durch dynamische Analysen möglich. Reine Strukturpattern können auch durch statische Analysen gefunden werden, Verhaltensmuster allerdings nicht [7]. Im Normalfall werden zahlreiche Ergebnisse gefunden, da auf einer sehr niedrigen Abstraktionsebene gearbeitet wird. Zusätzlich ist die Erkennung häufig fehlerhaft und nicht alle bekannten Muster der *Gang of Four* können erkannt werden [7]. Dies macht deren Bearbeitung mühselig. Dieser Ansatz ist erheblich langsamer als der Cluster-basierte Ansatz. [30]
Die meisten Tools zur Architektur Rekonstruktion beherrschen keine Entwurfsmustererkennung [7]. Allgemein existieren nicht viele Anwendungen, die die Mustererkennung unterstützen. Die meisten Anwendungen, die dies unterstützen sind nur zu Forschungszwecken entwickelt worden. Eine Weiterentwicklung und Optimierung fand meistens nicht statt, wie im Grundprojekt und in [7] festgestellt werden konnte.
Von Detten und Becker [30] beschreiben einen Reengineering Prozess, um beide Verfahren miteinander zu kombinieren. In ihrem Ansatz führen sie ein Cluster-basiertes Reengineering durch, um Komponenten zu identifizieren. Anschließend führen sie auf den einzelnen Komponenten, den Pattern-basierten Prozess durch. Dadurch wird die Anzahl der Ergebnisse verringert. Ein weiterer positiver Effekt ist, dass es möglich wird, nur Teile der Anwendung zu analysieren. Zeit für die Berechnung nicht geänderter Komponenten wird bei einer Neuberechnung eingespart. Außerdem können bereits nach der Cluster-basierten Analyse Refaktorisierungen, welche Probleme beheben und eventuell ein Verkleinern der gefundenen Komponenten bewirken, durchgeführt werden.
Eine Auflistung einiger Tools zur Entwurfsmusteranalyse und der Architektur Rekonstruktion ist in der Ausarbeitung des Grundprojekts und in der Arbeit von Arcelli Fontana und Zanoni [7] zu finden. Der folgende Abschnitt stellt eine Auswahl an Analysemöglichkeiten vor, die mit

den hier vorgestellten Techniken arbeiten. Anschließend wird ein Vorgehen um im Nachhinein eine Soll-Architektur zu ermitteln präsentiert.

3.4 Architekturanalysen

In diesem Abschnitt werden unterschiedliche Ansätze und Tools zur Analyse von Strukturen vorgestellt. Viele der Tools bieten weit mehr Einsatzzwecke an, die nur indirekt die Architektur betreffen. Dies kann z.B. eine automatische Erkennung von dupliziertem Code sein. Dabei handelt es sich ebenfalls um technische Schulden, die die Entwicklung behindern können und deshalb behoben werden sollten. Neben der Möglichkeit allgemeine Probleme zu finden, wird vorgestellt, wie eine Soll-Architektur mit einer Ist-Architektur verglichen werden kann. Wenn keine Soll-Architektur vorhanden ist, muss diese zuvor ermittelt werden. Hierzu wird ein halbautomatisches Vorgehen beschrieben. Dieses Vorgehen bezieht sich nicht auf eine spezielle Technik oder ein spezielles Tool, ist für eine Soll/Ist Prüfungen allerdings zwingend erforderlich.

Bei Literaturrecherchen konnte kein bedeutender Einsatz von dynamischen Techniken und der Entwurfsmustererkennung gefunden werden. Aus diesem Grund setzten die im Folgenden beschriebenen Analysen den statischen Ansatz ein. Bei den meisten dieser Ansätze und Tools kann es sehr nützlich sein, diese in den automatischen Build-Prozess zu integrieren. Dies kann im Rahmen von *Continuous Integration (CI)* geschehen. Analysen werden dadurch automatisch gestartet. Richtlinien können integriert werden, welche beim Finden von Problemen verhindern, dass der neu entwickelte Quellcode in die Anwendung integriert werden darf. Dies fördert die frühzeitige Findung von Problemen und stellt eine einheitliche Qualität sicher.

3.4.1 Bug Pattern Detection

Eine Möglichkeit allgemeine Probleme der Anwendung zu finden ist die *Bug Pattern Detection*. Bei der statischen Bug Pattern Detection kann mithilfe verschiedener Regeln nach Defekten im Code gesucht werden. Die Regeln haben das Ziel spezielle Code-Strukturen zu entdecken. Da eine statische Analyse durchgeführt wird, verwendet sie den Quellcode, kompilierte Dateien und Metadaten (z.B. Kommentare und Debug-Punkte). Dadurch können unter anderem Bugs, duplizierter Code oder auch schlechter Codestyle gefunden werden. Die meisten dieser Probleme stellen einzeln kein kritisches Problem dar. Erst deren Summe kann Probleme hervorrufen. Die entdeckten Probleme werden nach Kategorien unterteilt. Diese unterscheiden sich häufig

je nach Anwendung. Meistens werden sie nach dem Bereich sortiert, den die Probleme beeinflussen, wie z.B. Sicherheitsprobleme oder Performance Einbußen. [46]
Viele der vorhandenen Tools zum Architektur-Reenigineering haben solch eine Prüfung integriert. *PMD*[3] ist ein bekanntes Beispiel für ein Tool, welches Sourcecode Analysen durchführt. Es ist bereits in vielen IDEs und Build Infrastrukturen integriert und wird deshalb häufig in der Praxis zur Analyse der Komplexität eingesetzt [15]. Bei der Bug Pattern Detection kann viel projektspezifisches Feintuning vorgenommen werden. Gleischer u.a. [46] konnten durch praktische Untersuchungen feststellen, dass dies unter Umständen sehr aufwendig werden kann, weil dazu vorerst die notwendigen Qualitätsfaktoren identifiziert werden müssen. Dazu gehört z.B. ein Toleranzwert für Fehler. Störend ist auch, dass oft Probleme gefunden werden, die eigentlich keine sind (false positives). Dies kann unter anderem an einer simplen, falschen Erkennung liegen oder daran, dass die gefundenen Probleme nicht relevant für das Projekt sind. Durch Feineinstellungen können diese Probleme allerdings verringert werden. In ihrer praktischen Studie konnten sie feststellen, dass diese Analysen sehr hilfreich sind, um Probleme zu finden. Die von Gleischer u.a. untersuchten Firmen, haben aufgrund dessen beschlossen, diese Tools einzuführen, wenn dies nicht bereits geschehen war. [46]
Dieser Ansatz kann erfolgreich zu einer Steigerung der Codequalität führen. Bei einem lang laufendem Projekt ist es sinnvoll diese Prüfung auch nachträglich einzuführen, um eine einheitliche Qualität des gesamten Quellcodes sicherzustellen. Aber auch bei kleineren Projekten kann ein Einsatz durchaus Sinn machen. Dieser Ansatz macht allerdings in keiner Weise die entwickelte Architektur wieder sichtbar und kann dadurch nicht verwendet werden, um die Weiterentwicklung direkt zu unterstützen oder um konkrete Bugs zu finden. Er kann aber gute Hinweise auf potentielle Problemstellen liefern.

3.4.2 Conformance Prüfung

Als *Architecture Conformance* Analysen wird der Vergleich der Soll- und Ist-Architektur bezeichnet. Es handelt sich entsprechend um eine Feststellung der Softwareerosion [29]. Der Vergleich ist hilfreich, weil Artefakte, wie die Dokumentation und der Code selbst, einzeln gepflegt werden. Dadurch können sich diese schnell unterscheiden und in unterschiedliche Richtungen entwickeln [46, 61]. Passos u.a. [66] haben drei verschiedene Analyse-Techniken identifiziert. Eine Bewertung der Ist, bzw. Soll-Architektur findet dadurch nicht statt. In den meisten Fällen werden diese Analysen als statisches Verfahren durchgeführt.

[3] http://pmd.github.io/, Abgerufen: 31.08.2016

Dependency Structure Matrices (DSM) DSM stellt eine einfache Matrix zur Präsentation von Abhängigkeiten dar. Für jede Klasse existiert eine Spalte und eine Zeile. In den zugehörigen Feldern werden die Anzahl der Abhängigkeiten, also z.B. direkte Methodenaufrufe der Klasse eingetragen. Dadurch, dass es immer zwei Felder (A -> B und B -> A) pro Kombination gibt, können die Verbindungen gerichtet festgehalten werden. Durch Formulierung von Regeln kann geprüft werden, ob nicht erlaubte Abhängigkeiten oder circuläre Abhängigkeiten existieren, die die Architektur verletzen könnten. Abbildung 3.1 zeigt eine beispielhafte Matrix. Viele Tools bieten die Möglichkeit eine solche DSM zu integrieren. Einige der weiter verbreiterten IDEs haben diese Möglichkeit bereits integriert oder können mithilfe eines Plugins erweitert werden.

			.io			.jetty								
		servlet.jetty	bio	nio	io	deployer	webapp	security	servlet	bio	nio	handler	jetty	xml	resource
	servlet.jetty														
.io	bio							1		1					
	nio								1		4		1		
	io		2	8				1	3	3	4	4	67		
.jetty	deployer						1								
	webapp					1		1				1	1		
	security						10		3				4		
	servlet						13	1				2	4		
	bio							1					2		
	nio								1						
	handler					4	6	3	13				7		
	jetty	2		2		4	7	28	36	7	12	50			
	xml					1	3								
	resource					3	7	4	2			2	4	1	

Abbildung 3.1: Dependency Structure Matrix [33]

Source Code Query Languages (SCQL) SCQL bietet mehr als die Darstellung von Abhängigkeiten. Es kann prüfen ob Code Konventionen eingehalten werden und automatisiert Bugs und potentielle Stellen für Refaktorisierungen finden. Diese Technik wird häufig zur Überprüfung von Architektur-Constraints eingesetzt. Eine, eventuell aufwendige,

Modellierung der Constraints ist zuvor erforderlich. Eine SCQL ist Abhängig von der Programmiersprache und muss für diese entwickelt werden. Für Java existiert z.B. *Java Tools Language* [24] oder *Browse-By-Query*[4]. Die einzelnen SCQLs sind sehr unterschiedlich umgesetzt. Zum Teil arbeiten diese mithilfe von Queries, die für den abstrakten Syntax Baum (AST) gebaut werden. Andere arbeiten mit SQL ähnlichen Statements. In manchen Tools ist es zusätzlich möglich logische Statements zu integrieren, um komplexere Abfragen aufzubauen. [29, 78]

Reflexion Models (RM) Im Unterschied zu den anderen Methoden muss bei RM vor dem Einsatz eine sehr detaillierte Architektur modelliert werden. Diese beinhaltet die Komponenten und deren Interaktionen. Zusätzlich muss ein Mapping des Modells auf den Code definiert werden. Die Implementierung kann anschließend automatisch auf identische Strukturen (*Convergence*), zusätzliche Strukturen (*Divergence*) und fehlende Strukturen (*Absence*) geprüft werden. Diese Technik hat den Vorteil, dass sie gut mit der Größe des Systems skaliert. Sie ist sowohl für kleine, wie auch große Projekte anwendbar. Das vollständige Modell muss nicht von Anfang an vorhanden sein. Es kann schrittweise verfeinert werden. [61, 29]

In [61] ist ein Beispielvorgehen, bei dem ein Entwickler versucht mithilfe dieses Ansatzes eine unbekannte Architektur kennenzulernen, beschrieben. Das Vorgehen ist in Abbildung 3.2 (a) dargestellt. Dazu entwirft er zu Anfang ein grobes Model, von dem er glaubt wie die Software arbeitet. Im zweiten Schritt wurden Strukturinformationen aus dem Code mithilfe eines Tools ermittelt. Im dritten Schritt hat er ein Mapping zwischen den extrahierten Strukturen und dem gegebenen High-Level Model erzeugt. Das High-Level Model kann z.B. aus der gegebenen Ordner- und Dateistruktur stammen. Im vierten Schritt wird das Reflexionsmodell generiert. Dazu dient das zu Anfang entworfene Model, das Mapping und das extrahierte Sourcecode Modell. Abbildung 3.2 (b) zeigt das generierte Reflexion Modell. Hier ist zu sehen, wie die drei unterschiedlichen Strukturvorkommen dargestellt werden können. Gleischer u.a. [46] haben in ihrer Untersuchung die Anwendung *ConQAT*[5] verwendet.

Diese Vorgehen haben häufig das Problem, dass kein Wissen existiert, wie die aktuelle Architektur der Anwendung aussieht und wie sie optimalerweise aussehen sollte. Bei der Untersuchung von Passos [66] existierte in den meisten Fällen von vorn herein keine Dokumentationen. In einem Fall war diese z.B. unbekannt, weil auf Basis eines vorhandenen Systems einer anderen Firma weiterentwickelt wurde. Aus Kostengründen wurde deshalb ebenfalls darauf verzichtet

[4] http://browsebyquery.sourceforge.net/, Abgerufen: 06.08.2016
[5] https://www.cqse.eu/en/products/conqat/overview/, Abgerufen: 19.11.2016

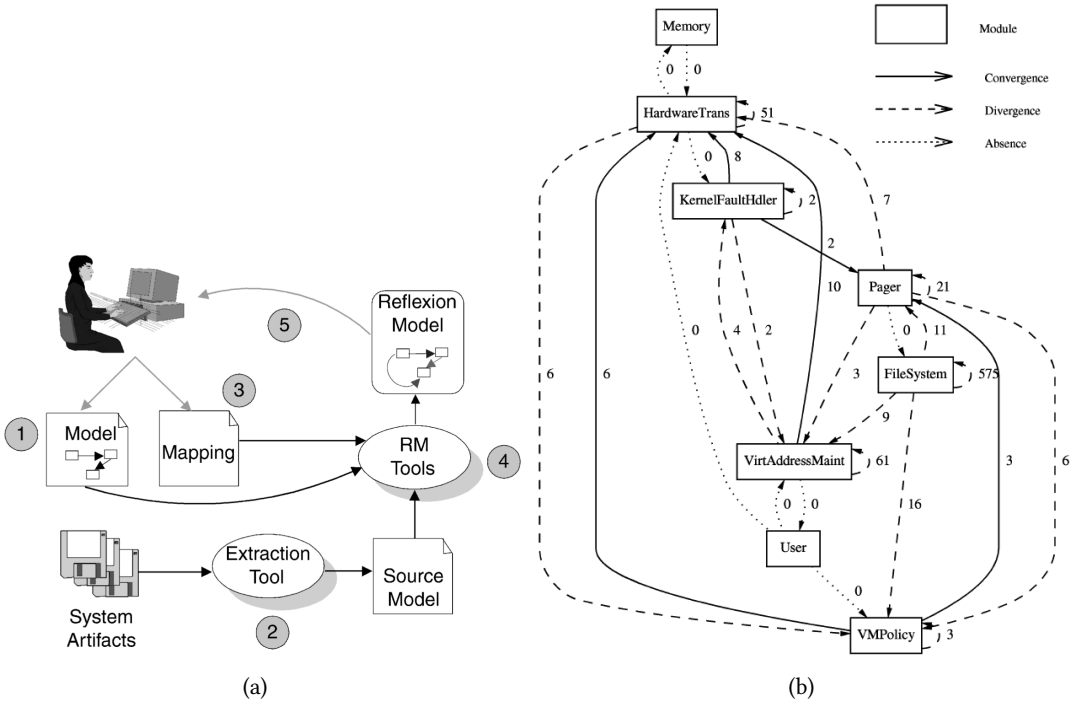


Abbildung 3.2: Generierung eines Reflexion Model [61]

dies nachzuholen. Das Unternehmen begründete dies damit, dass der Aufwand nicht genügend Vorteile bringe und das notwendige Aktualisieren sehr aufwendig sei. Zusätzlich ist die Übertragung des Modells, in das für das Tool lesbare Format, sehr aufwendig und komplex. Aufgrund der Komplexität können in die Modelle schnell Fehler eingebaut werden. Dies ist gerade bei der RM Methode von Bedeutung.

Statt einer Modellierung der exakten Anwendung, müssen bei SCQL einzelne genaue Regeln definiert werden. Sie werden in einer Art SQL-Sprache beschrieben und greifen auf konkrete Implementierungsdetails zu. Dies bewirkt, dass diese Regeln teilweise nicht allgemein projektübergreifend verwendet werden können. Allgemeine Regeln, wie z.B. die Suche nach leeren Exception-Handler, können wiederverwendet werden und sind Teil der Bug Pattern Detection. Aus Gründen der Einfachheit wird deshalb in vielen Projekten ausschließlich auf die universellen Regeln gesetzt. Silva und Perera [29] konnten in ihren Untersuchungen feststellen, dass die aktuell vorhandenen Lösungen noch nicht zufriedenstellend sind und nur wenige Unternehmen Werkzeuge einsetzen, um eine Conformence Prüfung durchzuführen.

Allgemein wurde festgestellt, dass der Einrichtungsaufwand relativ gering ist und im Schnitt circa eine Stunde betrug. Die Ergebnisse lieferten präzise Stellen, die direkt im Sourcecode gefunden und dadurch korrigiert werden konnten. [66]
Alle Probleme konnten allerdings nicht gefunden werden. Es konnten aber gute Hinweise auf Stellen geliefert werden, an denen durch eine Refaktorisierung nachgebessert werden muss. Dynamische Informationen können mithilfe dieser Vorgehen nicht geprüft werden. Dies ist z.B. bei der Verwendung von Reflexion erforderlich, wie es unter anderem *Java* oder *PHP* unterstützten. Bei Reflexion wird der auszuführende Code während der Laufzeit bestimmt.

3.4.3 Soll-Architektur Ermittlung

Eine Architektur Ermittlung (engl. architecture discovery) ist erforderlich, wenn keine Soll-Architektur verfügbar ist. Eine Conformance Prüfung ist deshalb nicht möglich. Lilienthal nennt das Ermitteln der Soll-Architektur aus einer vorhandenen Software „*Software-Archäologie*“ [54]
Dieser Vorgang muss händisch durchgeführt werden. Im Folgenden wird ein mögliches Vorgehen beschrieben.
Die Architektur muss dazu im ersten Schritt aus den Anforderungen und Use Cases entwickelt werden. Im nächsten Schritt werden Diagramme (z.B. Klassendiagramme) aus dem vorhandenem Sourcecode generiert. Die generierten Diagramme werden anschließend, bis ein zufriedenstellendes Modell entstanden ist, schrittweise zusammengefasst und strukturiert. Die Generierung der Klassendiagramme kann durch Standard Programme geschehen. Viele IDEs haben diese Funktion bereits integriert. Im dritten Schritt werden das zu Anfang entworfene und das generierte Modell miteinander kombiniert. Schlussendlich soll eine Soll-Architektur mit den am besten passenden Architekturstilen vorhanden sein. [72]

3.4.4 Architekturmonitoring

Wie festgestellt wurde, ist für eine erfolgreiche Weiterentwicklung ein guter Wissensstand über die Anwendung notwendig. Ein dauerhaftes Monitoring, welches den aktuellen Stand darstellt und bei Problemen alarmiert, könnte dies bewerkstelligen.
Durch aktuelle Kenntnisse des Ist-Zustandes ist dies realisierbar. Um diesen zu erhalten, können die zuvor vorgestellten Analysen eingesetzt werden. Ein Monitoring kann ein besseres Bewusstsein aller Teammitglieder für die Architektur bewirken und erhöht dadurch deren Verständnis für den aktuellen Stand der Anwendung. Problematische Stellen, z.B. aufgrund von nicht erlaubten Abhängigkeiten oder Zugriffen an Schnittstellen vorbei, können automatisiert

und frühzeitig erkannt werden. Um Abweichungen direkt erkennen zu können, kann der geplante Soll- mit dem Ist-Zustand verglichen werden. Dies ermöglicht die frühe Erkennung einer Architekturerosion und deren Auflösung. Dadurch kann ein Scheitern des Projektes und durch Verzögerungen entstehende Mehrkosten verhindert werden. Diese Analysen müssen nur durchgeführt werden, wenn etwas an dem Quellcode verändert wurde.
Durch die Beobachtung von weiteren Metriken zur Laufzeit, können weitere Defekte oder Hinweise auf notwendige Änderungen erkannt werden. Es kann z.B. beobachtet werden, wie lange einzelne Anfragen dauern. Wenn sich diese Laufzeiten auf Dauer verlängern, kann dies ein Hinweis auf mögliche Probleme darstellen. Die Laufzeitänderungen können unter anderem durch ein nicht einkalkuliertes Nutzeraufkommen beeinflusst werden. Ein Beobachten der Anzahl der Benutzer, die gleichzeitig aktiv sind, kann dazu führen, dass eine steigende Tendenz erkannt wird, die bei der Entwicklung nicht einkalkuliert wurde. Neben zahlreichen individuellen Beobachtungspunkten können jegliche Hardwareauslastungen (z.B. Prozessorlast, Speicherauslastung, usw.) überwacht werden. Die Auswahl der zu überwachenden Eigenschaften muss individuell, je nach Anwendung und Relevanz entschieden werden.

Für ein Monitoring ist eine übersichtliche Darstellung sehr wichtig. Nur so können Informationen schnell erfasst werden. Informations- und Fehlermeldungen können bei Verletzungen angezeigt werden. Die Visualisierung von Daten ist ein komplexes Problemfeld. Da diesem Thema vollständig eigene Arbeiten gewidmet werden (z.B. [47, 65]), wird dies hier nicht weiter betrachtet.
Ein Monitoring kann mithilfe der zuvor vorgestellten Analysemöglichkeiten durchgeführt werden. Das Tool *Sonarqube*[6] bietet z.B. eine Basis zur Darstellung und Sammlung von Daten dazu an. Die Anwendung bietet eine Weboberfläche zur Visualisierung der Daten und lässt sich beliebig durch Plugins erweitern.

3.5 Architekturanalysen in der Praxis

Die Untersuchung von Passos in [66] haben gezeigt, dass eine automatische Conformance Prüfung in wenigen Firmen stattfindet (z.B. [35]). Häufig wird der Nutzen der Prüfungen unterschätzt. Tools, wie zum Beispiel zur Bug Pattern Detection, Code Style Prüfung oder ähnlichem werden häufiger eingesetzt. Bei deren Einsatz wird meistens auf eine allgemeingültige Konfiguration zurückgegriffen, ohne diese projektspezifisch anzupassen.

[6] http://www.sonarqube.org/, Abgerufen: 19.11.2016

Lilienthal [54] wählt einen anderes Ansatz. Sie geht in fremde Firmen und Projekte und führt dort Architekturanalysen durch. Bei den Analysen handelt es sich um Conformance Prüfungen mithilfe von Reflexion Modellen. Fallstudien haben gezeigt, dass dieser Ansatz am klarsten strukturiert ist und deshalb sehr gut für eine Conformance Prüfung geeignet ist [29]. Technische Schulden und Gründe für eine stockende Entwicklung sollen erkannt werden können. Architekturverbesserungen werden im Rahmen der Analysen ausgearbeitet. Als Analysetool verwendet sie häufig den *Sonargraph*[7]. Dieser extrahiert z.B. Strukturinformationen aus dem Sourcecode und stellt diese als Baum, inklusive deren Beziehungen, dar. Nach dem Extrahierungsschritt wird die Soll-Architektur auf die gefundenen Strukturen abgebildet. Wenn keine Soll-Architektur existiert, muss diese während der Analyse erarbeitet werden. Dies geschieht direkt in der Analysesoftware, durch die Definition von Schichten, unabhängigen Subsystemen[8] und uneingeschränkten Subsystemen[9]. Packages und Klassen werden dabei den Konstrukten zugeordnet. Für jedes Konstrukt können zusätzlich Schnittstellen definiert werden. Nur diese erlauben den Zugriff von außen. Durch diese Zuweisung werden Verletzungen der Schichten und Zugriffe erkannt und dargestellt. Die gefundenen Verletzungen müssen mit den Entwicklern durchgegangen werden, um die Gründe zu erfahren und Lösungen zum Auflösen dieser zu erarbeiten. Die verwendete Sonargraph Anwendung ermöglicht es Klassen in andere Packages zu verschieben, um dadurch die Auswirkungen zu simulieren. Die gesamte Analyse geschieht in Zusammenarbeit mit den Entwicklern. Einige Entwickler sehen in diesem Rahmen häufig zum ersten Mal die Gesamtstruktur der Anwendung. Die Analyse mit den Entwicklern ist wichtig, da diese dadurch einen Überblick bekommen und zu einer erfolgreicheren Lösung verhelfen können.
Der Vorteil, wenn dieses Verfahren durch eine nicht am Projekt beteiligte Person durchgeführt wird, ist dass eventuell mehr Nachfragen zu Bereichen kommen, die andere bereits für selbstverständlich erachtetet haben. Außerdem werden schneller unangenehme Stellen angesprochen, als bei Personen die diese eventuell selbst mit verursacht haben. Externe Personen können außerdem bewirken, dass neues Wissen und andere Sichtweisen mit eingebracht werden. Wenn diese Rolle durch eine, nicht dem Unternehmen angehörige Person durchgeführt wird, sind die Kosten allerdings höher. Die Durchführung sollte bei großen Projekten mehrfach im Projektverlauf stattfinden. Dadurch wird es möglich einen Verlauf zu erkennen, um zu sehen ob die neuen Maßnahmen einen Erfolg liefern.

[7] https://www.hello2morrow.com/products/sonargraph, Abgerufen: 19.11.2016
[8] sollten sich z.B. gegenseitig nicht kennen
[9] können durch andere Subsysteme verwendet werden

3.6 Fazit

Dieses Kapitel hat gezeigt, dass es durchaus möglich ist nur aus Quellcode eine Architektur zu reengineeren. Allerdings müssen viele Einschränkungen in Kauf genommen werden. Das gewünschte Ziel, eine vollständige Architekturdokumentation automatisch generieren zu können, kann bisher nicht erreicht werden. Die Einarbeitung in eine unbekannte Anwendung ist somit nicht gut unterstützt. Andere hilfreiche Informationen können aber gewonnen werden. Es konnte bei den Analysetechniken festgestellt werden, dass statische Analysen sehr effizient sind und sich deshalb auch ein Einsatz in großen Projekten lohnen kann. Es können automatisiert kritische Fehler in Anwendungen gefunden werden [46]. Dynamische Analysen sind sehr komplex. Zu diesem Thema fehlen aktuelle Arbeiten, zusätzlich gibt es nur wenige Werkzeuge, die dies unterstützen. Ähnlich ist dies bei der Erkennung von Entwurfsmustern, dies wird ebenfalls nicht praktisch eingesetzt.
Im anschließendem Absatz wurden Analysemöglichkeiten vorgestellt, die ebenfalls in der Praxis eingesetzt werden können. Allgemein konnte zum Einsatz der Tools festgestellt werden, dass diese sehr viel effizienter eingesetzt werden können, wenn die Analyseregeln individuell an das jeweilige Projekt angepasst werden. Häufig sind nur die Standard Regeln im Einsatz. Aber auch diese können bereits zu einer besseren Codequalität beitragen. Die Conformance Prüfung eignet sich gut zum Auffinden von Verletzungen, wie nicht erlaubte Strukturen, Abhängigkeiten oder Zugriffen. Für diese Erkennung müssen allerdings Modelle für die Architektur entwickelt werden. Die Definition von Regeln und das Handling der Tools ist häufig komplex und ohne Übung fehleranfällig. Da dies von Tool zu Tool unterschiedlich ist, sollte vor dem Einsatz geprüft werden, welches für längere Zeit hilfreich ist. Es macht Sinn, dass Entwickler sich ausführlicher in dieses Tool einarbeiten. Dies bedeutet zwar, dass anfänglich mehr Zeit investiert werden muss, bringt in Zukunft allerdings Vorteile, da es anschließend schneller in neuen Projekten eingesetzt werden kann. Durch den Einsatz von individuellen Regeln, kann die Anwendung besser im Bereich der *„Guten Architektur“*, wie zu Beginn in Abbildung 2.2 dargestellt, gehalten werden. Die späten Refaktorisierungsmaßnahmen werden verringert, weil Probleme früher gefunden und direkt behoben werden können. In den meisten Fällen ist eine Kombination von unterschiedlichen Tools sinnvoll, da die einzelnen meist nur einen Teil der Probleme angehen [35]. Um den Vorgang möglichst komfortabel und gleichzeitig verpflichtend zu gestalten, sollten die Analysen in einem Continuous Integration System integriert werden. Eine Integration der Tools in den Entwicklungsprozess verpflichtet die Entwickler sich an spezielle Coderichtlinien zu halten. Diese Richtlinien können z.B. durch die *Definition of Done* festgelegt werden. Die Integration kann z.B. dadurch realisiert sein, dass Branches im Git

Repository nur in den Hauptentwicklungsstrang gemerged werden können, wenn alle Prüfungen erfolgreich durchgelaufen sind. Die durchgehende Analyse der Anwendung kann als Monitoring realisiert werden. Dadurch soll es allen Entwicklern möglich sein den aktuellen Stand im Blick zu behalten. Außerdem können sie direkt erkennen, wie die Architektur durch ihre Änderung beeinflusst wird. Für ein Monitoring ist eine gute und übersichtliche Visualisierung sehr wichtig. Bei Projekten mit langer Laufzeit macht es durchaus Sinn, diese Prüfungen auch im Nachhinein noch einzuführen. Dies führt, in den meisten Fällen vorerst zu einer aufwendigen Refaktorisierungsphase, welche die Anwendung auf einen einheitlichen Qualitätsstandard hebt. Wenn keine Soll-Architektur festgehalten ist, sollte diese in diesem Schritt entwickelt und dokumentiert werden. Es ist nicht sinnvoll, wenn diese alleine in den Köpfen der Entwickler vorhanden ist.

Die Analyse mit Sonargraph macht bei großen Anwendungen ebenfalls Sinn. Eventuell ist es gut, wenn dies regelmäßig in Projekten als Architektur-Review durchgeführt wird. Dieses Review könnte durch einen nicht an der Entwicklung beteiligten Spezialisten des Unternehmens geleitet werden. Refaktorisierungsschritte können dadurch geplant werden. Der Vorteil bei einem Spezialisten ist, dass dieser die gesamte Anwendung unvoreingenommen betrachten und Anregungen für Änderungen, die er in anderen Projekten gesehen hat, einbringen kann. Außerdem muss nur dieser mit den Analysetools im Detail zurechtkommen. Ein weiterer, für viele Unternehmen sehr wichtiger Nebeneffekt ist, das die Kosten niedriger sind, weil weniger Lizenzen benötigt werden.

Über den Reengineering Ansatz gibt es zahlreiche Forschungen, bei denen viele nicht zu einem vollständig zufriedenstellendem Ergebnis kommen. Die meisten Untersuchungen wurden in reinen Testszenarien mit extra dafür entwickelten Werkzeugen durchgeführt. Weitere Untersuchungen zum Einsatz in der Praxis sind erforderlich. De Silva u.a. stellen in [72] einen Überblick über Möglichkeiten, Tools und Technologien zum Finden, Minimieren und Kontrollieren einer Architekturerosion vor.

Zusammenfassend kann gesagt werden, dass der Reengineeringansatz nur beschränkt für das Ziel, herauszufinden welche Architektur aktuell ist, verwendet werden kann. Viele der Ansätze können allgemeine Fehlerstellen finden, haben aber nicht das primäre Ziel einen Überblick geben, speziell gesuchte Stellen finden oder sogar eine Einarbeitung ermöglichen zu können. Bei einer Weiterentwicklung kann der Ansatz in einzelnen Phasen hilfreich sein, wie zuvor in [41] festgestellt wurde. Es kann z.B. unterstützen, wenn eine Stelle gefunden wurde, an der Änderungen notwendig sind. Für diese Stelle kann ein Baum aufgebaut werden, um

herauszufinden welche Codestellen diese aufrufen und welche wiederum von ihr aufgerufen werden. Dies ermöglicht eine bessere Kontrolle und Verhinderung von Seiteneffekten.
Eine Diskussion auf der Ebene von Mustern und Strukturen ist nicht möglich. Der alleinige Ansatz des Reengineering, um ein Architektur-Redesign durchzuführen, ist nicht sehr erfolgsversprechend. Bei Conformance Prüfungen ist es erforderlich, dass eine Soll-Architektur existiert. Das Entwerfen dieser ist schwer und bewirkt keine Prüfung dieser Architektur. Die Soll-Architektur kann ebenfalls nicht optimal sein. Aus diesem Grund darf das Ziel nicht starr festgelegt sein und sollte regelmäßig überprüft und angepasst werden.
Ein weiterer wichtiger Punkt ist, dass mit diesem Ansatz nicht herausgefunden werden kann, aus welchem Grund bestimmte Lösungen entwickelt wurden. Dies zeigt, dass bereits vor, bzw. auch während der Entwicklung eine ausführliche Debatte über die Architektur notwendig ist. Die getroffenen Entscheidungen müssen direkt begründet dokumentiert werden. Nur so sind diese im Anschluss nachvollziehbar. Dies macht deutlich, dass auf eine hohe Qualität der Softwarearchitektur von Beginn an der Entwicklung geachtet werden muss. Eine Kombination aus Reengineering und regelmäßigen Debatten kann die Qualität nochmals steigern.
Im nächsten Kapitel soll aufgrund der gewonnen Erkenntnisse die Architekturentwicklung von einem anderen Standpunkt aus betrachtet werden. Es werden Möglichkeiten untersucht, um eine Architektur begleitend mit der Anwendung zu entwickeln.

4 Top-Down: Integration von Architekturdebatten in agilen Prozessen

Dieses Kapitel beschreibt Möglichkeiten, um Architekturdebatten in den Entwicklungsprozess zu integrieren. Es wird untersucht, wie dies in einem agilen Prozess, wie Scrum, integriert werden kann. Die Schwierigkeit besteht häufig darin die Agilität nicht einzuschränken. Vorhandene Arbeiten und Ideen werden vorgestellt und diskutiert. Einige der Inhalte wurden im Buch *Agile Software Architecture* von Babar u.a. [9] veröffentlicht. Da die Inhalte von anderen Personen stammen, werden diese direkt referenziert.

Zu Anfang werden die Ziele einer vollständigen Integration in den Entwicklungsprozess vorgestellt. In einer Debatte müssen zwingend Entscheidungen gefällt werden. Bei Entscheidungen spielt das *Wer?*, *Wie?* und *Wann?* eine wichtige Rolle. Für diese Fragen soll in diesem Kapitel versucht werden Antworten zu finden.
Unterschiedliche Arten zur Entscheidungsfindung werden hierzu vorgestellt. Anschließend wird darauf eingegangen wie die agilen Prinzipien auf die Architekturentwicklung abgebildet werden können. In den folgenden Schritten werden Möglichkeiten erläutert, um die Anforderungen an eine Architektur zu ermitteln, eine leichtgewichtige Dokumentation zu erzeugen und die Rolle eines Softwarearchitekten zu integrieren. Zum Ende dieses Kapitel werden vorhandene Ideen, zu Vorgehen mit einer integrierten Debatte, vorgestellt und ein Fazit gezogen.

4.1 Ziele

Ziel einer direkten Integration ist von vornherein eine zufriedenstellende Architektur zu entwickeln. Die technischen Schulden sollen hierzu durchgehend in einem akzeptablem Rahmen gehalten werden. Der Prozess soll in den Scrum Prozess integriert werden, ohne das dessen Agilität eingeschränkt wird. Zu diesem Zweck muss die Architektur schrittweise entwickelt werden. Ein umfangreiches Vorabdesign, wie in traditionellen Methoden, ist dabei kontrapro-

duktiv. Durch die schrittweise Entwicklung und dem Einhalten der agilen Prinzipien wird die Architektur flexibler gegenüber Änderungen [45]. Neben der Erweiterung der Architektur muss ein Vorgang integriert werden der den aktuellen Zustand überprüft und wenn notwendig erforderliche Anpassungsprozesse auslöst. Dies kann z.B. durch manuelle Reviews oder durch die im vorherigen Kapitel vorgestellten Tools geschehen. Eine spezielle Rolle in Form eines Architekten, die den Überblick über die Gesamtarchitektur hat und die Entwicklung dieser leitet, kann sehr hilfreich sein. Als Nebenprodukt der Entwicklung sollte eine Dokumentation entstehen. Diese sollte sich dauerhaft mit der Anwendung weiterentwickeln. Die Dokumentation sollte in erster Linie für die Entwickler sein und z.B. zur Einarbeitung neuer Personen dienen können.

4.2 Entscheidungsfindung

Um eine Architektur zu entwickeln müssen Entscheidungen getroffen werden. In der Literatur existieren z.B. zahlreiche Architekturmuster. Welche allerdings für das aktuelle Projekt relevant sind muss individuell entschieden werden.

Eine Form der Entscheidungsfindung ist die *autokratische* Entscheidungsfindung. Hier werden sämtliche Entscheidungen allein von einer Führungsperson getroffen [34]. Diese Form ist nicht sehr modern und widerspricht den agilen Prinzipien. Die autokratische Entscheidungsfindung sollte aus diesem Grund zwingend vermieden werden.
Ein weiterer Ansatz ist die *demokratische* Entscheidungsfindung. Bei dieser wird über mehrere Möglichkeiten abgestimmt. Es wird eine Mehrheitsentscheidung getroffen. Die Möglichkeit mit den meisten Stimmen gewinnt. Bei dieser Lösung müssen nicht alle mit der getroffenen Entscheidung einverstanden sein. [64]
Anders ist dies bei einem *Konsens*. Bei dieser Art müssen sich alle über eine gemeinsame Lösung einigen, bzw. ihr nicht aktiv widersprechen [38, 64]. Diese Lösung erfordert eventuell, dass andere Personen überzeugt werden müssen. Die Vorschläge müssen deshalb begründet und erklärt werden können. Bei dieser Art wird diskutiert wer dafür ist. Bei Unstimmigkeiten kann diese Form zu ausführlichen Diskussionen führen.
Beim *Konsent* wird stattdessen explizit nach Einwänden gegen eine Lösung gefragt. Durch diesen Ansatz wird versucht eine bessere Lösung zu erarbeiten. Für die erhobenen Einwände müssen Lösungen gefunden werden. Je nach Vorgang kann entschieden werden ob es unterschiedliche Arten von Einwänden geben kann. Es kann z.B. nach Einwänden, die die Entscheidung blockieren (*Veto*) und nicht blockierenden Einwänden unterschieden werden. Dies

macht z.B. Sinn, wenn jemand einen Einwand hat, die vorgeschlagene Lösung aber trotzdem besser als der aktuelle Zustand ist. [64]
Der *konsultative* Einzelentscheid ist eine weitere Möglichkeit. Bei diesem Vorgehen wird eine Person mit einer Entscheidungsfindung beauftragt. Sie kann sowohl ein Teammitglied, wie auch eine fremde Person sein. Vor der Entscheidungsfällung muss sie die Meinungen der einzelnen Teammitglieder erfassen. Die letztendliche Entscheidung trifft sie aber alleine und für alle Teammitglieder verbindlich. [64]

Bei der agilen Entwicklung ist es wichtig sämtliche Entscheidungen im Team zu treffen. Aus diesem Grund ist besonders die Konsens und die Konsent Entscheidungsfindung relevant. Die Konsent Entscheidungsfindung hat den Vorteil, dass die Entscheidung durch das Auflösen von Einwänden weiter verbessert werden kann. Dies muss solange durchgeführt werden bis alle Einwände beseitigt wurden. Beim Konsens können Entscheidungen durchgesetzt werden obwohl es ungelöste Einwände gibt.

4.3 Architekturmuster

Architekturmuster können das Verständnis einer Anwendung vereinfachen. Sie stellen bekannte, sich bewährte Lösungen für Architekturen zur Verfügung. Dies ist ähnlich wie Designmuster zu verstehen. Muster können unter Umständen allerdings zu größeren und overengineerten Lösungen führen [51]. Dies kann durch zusätzliche Klassen oder ähnlichem geschehen.
Vorteil von Mustern ist, dass diese eine gemeinsame Sprache für die Entwickler definieren. Dies fängt bei einfachen Bezeichnungen und Namensgebungen, wie z.B. *Service* an und kann bis zu kompletten Kontrollstrukturen die einen Informationsfluss darstellen gehen. Die Zusammenarbeit kann dadurch effektiver werden.
Es ist möglich eine Beschreibung der Mustern und deren Zweck nachzulesen. Durch die Beschreibung kann eine Intention erkannt werden. Wenn beim Kennenlernen einer neuen Software bekannte Muster gefunden werden ist dessen Aufbau schneller nachvollziehbar.
Bei der Planung einer Architektur kann direkt überlegt werden wie etwas realisiert wird. Eine Vereinbarung unter den Entwicklern auf Basis von Architekturmuster wird möglich.

4.4 Agile Architektur-Prinzipien

Wenn ein agiler Prozess wie Scrum angepasst wird ist es wichtig, dass der Prozess nach der Anpassung weiterhin agil ist. Um dies zu erreichen muss darauf geachtet werden die agilen

Prinzipien nicht zu verletzen.
Es existieren verschiedene Ansätze, um agile Vorgänge und die Architekturentwicklung zu vereinen. Dies ist ohne Probleme möglich, da alle agilen Ansätze und die meisten Vorgehen nur Vorschläge zu einer Arbeitsweise machen. Viele sind außerdem nicht auf einen speziellen Tätigkeitsbereich, wie die Softwareentwicklung, begrenzt (z.B. Kanban). Sie sind als Framework gedacht und dürfen dadurch nach Belieben angepasst werden. Dies ermöglicht zusätzliche Arbeitsschritte und Rollen hinzuzufügen. [1]

Woods [85] hat gezeigt, dass einzelne Aussagen des Manifests ebenfalls auf die Architekturentwicklung bezogen werden können. Im Folgenden werden von ihm vorgestellten Praktiken aufgezählt. Teilweise wurden die Punkte durch weitere Informationen und Quellen ergänzt.

Änderungen zulassen:

Inkrementelle Auslieferung Nur aktuell wichtige Entscheidungen sollen getroffen werden. Es darf nur eine Architektur modelliert werden, die gerade ausreichend für die nächsten Schritte ist. Dies ermöglicht gleichzeitig eine regelmäßige Überprüfung ob die Architektur der Praxis gerecht wird [82, 83]. Sobald ein individuelles Design benötigt wird, soll dieses in der konkreten Iteration entworfen und anschließend direkt umgesetzt werden. Die einfache Lösung kann, wenn erforderlich, durch Refaktorisierungen verfeinert werden. [60]

Eindeutige Architekturprinzipen Die Prinzipien sollen alle Entwickler befähigen gute Architekturentscheidungen treffen und nachvollziehen zu können. Bewährte Design Praktiken sollten verwendet werden, da sich diese besser verständlich sind.[82].

Entscheidungen und Gründe erfassen Für ein besseres Verständnis sollen alle Entscheidungen, inklusive deren Begründungen erfasst werden.

Komponenten klar definieren Für jede Komponente müssen die Aufgaben, Abhängigkeiten und Schnittstellen definiert werden. Es ist wichtig, dass alle architektonischen Komponenten und Beziehungen direkt ersichtlich sind. Zu klar definierten Komponenten gehört ebenfalls eine deutliche Separierung der Zuständigkeiten (*Seperation of Concerns*). [73].

Menschen über Tools und Prozesse:

Informationen über einfache Tools teilen Informationen sollen für alle Stakeholder einfach verständlich und zugreifbar zur Verfügung gestellt werden. Beim Entwerfen kann

z.B. ein einfaches Whiteboard oder Papier genutzt werden [5]. Zur Dokumentation sind Fotos ausreichend. Dies ermöglicht alle direkt mitzuarbeiten ohne Probleme mit den Werkzeugen zu haben.

Für jedes auslieferbare Artefakt Kunden definieren Damit die Relevanz von Dokumenten und anderen Artefakten sichergestellt ist soll vor Erstellung ein Kunde definiert werden. Überflüssige Arbeit soll so verhindert werden.

Software über Dokumentation:

Nur ausreichend gute Artefakte Die Artefakte (Dokumentation und Modelle) sollten minimal sein, da sie kein Teil des auszuliefernden Systems sind. Dies entspricht dem *KISS - Keep it Simple and Stupid* Prinzip.

Nur lauffähiges ausliefern Jede Idee, jeder Sourcecode und jedes Designprinzip, dass ausgeliefert wird muss z.B. durch Tests oder Prototypen vor der Auslieferung geprüft werden.

Lösungen die mehrere Bereiche des Systems betreffen definieren Designentscheidungen die z.B. Qualitätseigenschaften für weite Bereiche der Anwendung betreffen müssen deutlich definiert werden.

Zusammenarbeit über Verträge:

Im Team arbeiten Architekten und Entwickler müssen in einem Team arbeiten. Eine enge Zusammenarbeit ist für beide Seiten sehr hilfreich.

Designarbeit auf Stakeholderinteressen konzentrieren Bei der Entwicklung einer Architektur, soll sich auf die Interessen der Stakeholder konzentriert werden. Nicht relevante Eigenschaften sollen bewusst vernachlässigt werden. Es sollen keine Modellierungen von allgemein nicht relevanten und für den nächsten Schritt irrelevante Optionen durchgeführt werden. Entscheidungen die für den aktuellen Sprint nicht wichtig sind sollen verzögert werden. [82] Der Fokus soll auf das Lösen eines Problems liegen und darauf eine möglichst erweiterbare und wiederverwendbare Lösung zu entwickeln. [60]

Auf Architekturinteressen konzentrieren Bei der Arbeit mit Architekten soll sich auf die Architekturentscheidungen wie die globalen Designentscheidungen konzentriert werden.

Es ist wichtig die Prinzipien bestmöglich zu unterstützen. Dadurch kann sichergestellt werden, dass die Agilität nicht eingeschränkt wird. Bei einer nicht Beachtung kann es z.B. auf ein großes Vorabdesign hinauslaufen. Dies würde den traditionellen Modellen, wie dem Wasserfallmodell,

sehr ähneln. Vorteile, wie die Möglichkeit schnell auf Änderungen reagieren zu können, gehen verloren.

4.5 Architekturanforderungen

Um eine Architektur entwickeln zu können, muss bekannt sein, was diese leisten muss. Was eine Architektur leisten muss, kann durch die signifikanten Anforderungen (Architecture Significant Requirements - ASR) ausgedrückt werden [49]. Diese Anforderungen können z.B. Aussagen über die Performance eine Anwendung machen. Aussagen über die notwendige Wartbarkeit, sowie Weiterentwickelbarkeit müssen im Vorhinein durch die geplante Laufzeit der Anwendung ermittelt werden. Viele der ASRs können von den nichtfunktionalen (NF) Eigenschaften stammen. Diese werden in den agilen Vorgehen allerdings häufig vernachlässigt, da das Interesse vollständig auf den funktionalen Anforderungen beruht [21]. Für NF Eigenschaften existiert keine eindeutige Definition. Es kann sich bei ihnen z.B. um technische, wie auch nichttechnische oder Architektur-Bedingung handeln. [32, 40]
Von den Features, welche die Anwendung leisten soll, kann auf die ASRs geschlossen werden. Untersuchungen anderer Arbeiten in [22] haben gezeigt, dass diese häufig in keiner Form dokumentiert oder erfasst werden. Häufig existieren diese Anforderungen mehr oder weniger bewusst in den Köpfen der Stakeholder. Bei diesem Thema wird erneut häufig die Argumentationen verwendet, dass ein explizites Niederschreiben nicht notwendig ist und die gewünschte Qualität durch Refaktorisieren erreicht werden soll. Diese Argumentation ist, wie in den vorherigen Kapiteln erwähnt eine häufige Begründung, um nicht von vornherein auf Qualität achten zu müssen. Das Problem ist, dass eine frühe Modellierung häufig mit einem Big-Up-Front-Design verglichen und deshalb vermieden wird [1]. Die meisten vorhanden Möglichkeiten ASRs zu dokumentieren sind sehr schwergewichtig und dadurch nicht besonders gut für agile Vorgehen geeignet [22].

Cleland-Huang u.a. beschreiben in [22] ein Vorgehen bei dem sie mithilfe von *Personas* Architekturanforderungen schrittweise erarbeiten. Die Personas werden in vier Schritten erzeugt:

1. Sammlung und Gruppierung von Features und Identifizierung deren Relevanz für die Architektur.
2. Erstellung der individuellen Personas.
3. Zuweisung der Features und Hinzufügen von Detailinformationen zu den Personas.

4. Personas mithilfe eines projektweiten Repository für eine einfache Kommunikation veröffentlichen.

Die meisten Projekte bestehen aus einer Sammlung von Feature Requests. Viele Stakeholder betrachten diese allein unter dem Aspekt der Funktionalitäten. Feature Requests können bereits Ansatzpunkte für die Architektur beinhalten. Durch Fragen wie: „Wie schnell?“, „Wie sicher?“ oder „Wie verfügbar?“ können hilfreiche Ansätze erhalten werden. Die Qualitätsziele der Architektur sind dadurch gut ermittelbar. Alle herausgearbeiteten Ziele, wie z.B. *Daten Vertraulichkeit*, werden auf den zuvor erarbeiteten Personas notiert. Die Ziele werden auf allen Personas notiert. Eine farblichen Markierung in Schwarz, Grau und Weiß kennzeichnet die Relevanz der Anforderung. Die Personas werden vor der Implementierung entwickelt und dienen hauptsächlich zur Kommunikation und für ein besseres Verständnis. Im Laufe der Implementierung werden sie weiterentwickelt.

Aus den herausgearbeiteten Zielen müssen in den nächsten Schritten die Architekturlösungen erarbeitet werden. Die zuvor erfassten Qualitätsziele werden dazu in *„soft goals“* umgewandelt und inklusive deren Konflikte und Einflüsse betrachtet. Hierzu wird ein Graph mit unterschiedlichen Zielen und Subzielen entworfen. Ein Beispiel dazu ist in Abbildung 4.1 dargestellt. Dort ist zu sehen, dass die Abhängigkeiten der Ziele untereinander bewertet werden. Eine Bewertung welche Ziele andere unterstützen und welche andere stattdessen behindern wird möglich. Technische Ziele, wie z.B. Plattform Portabilität und Ziele wie die Verwendung von Programmiersprachen, die die Entwickler gut beherrschen können mit einfließen.
[23] Aus den erarbeiteten Zielen muss in den nächsten Schritten eine Vision für die Architektur entworfen und eine Evaluierung gegenüber der Personas durchgeführt werden. Die modellierte Architektur muss den Ansprüchen dieser gerecht werden. [22]

NORMATIC [40] ist ein Werkzeug zur Arbeit mit den NF-Anforderungen. Es kann zum Identifizieren, Verbinden und Modellieren der NF-Anforderungen mit den funktionalen Anforderungen eingesetzt werden. Zur Visualisierung werden ähnliche Darstellungen wie in Abbildung 4.1 eingesetzt. Es soll außerdem bei der Projektplanung helfen. Bei der Entwicklung stand das Ziel die agile Entwicklung und im speziellen Scrum zu unterstützen im Vordergrund. Beim Einsatz innerhalb eines Scrum Prozesses kann es z.B. bei Schätzungen von User Stories helfen.

Dieses Vorgehen ist klar strukturiert und zeigt eine Möglichkeit, um einen Überblick über verschiedene Anforderungen an die Anwendung zu erhalten. Neue Anforderungen und Änderungen können zu jeder Zeit hinzugefügt werden. Konfliktstellen können dadurch erkannt werden. Dies ermöglicht eine iterative Entwicklung und ist somit gut für einen agilen Prozess geeignet. Es muss allerdings darauf geachtet werden, dass durch die Sammlung und Analyse

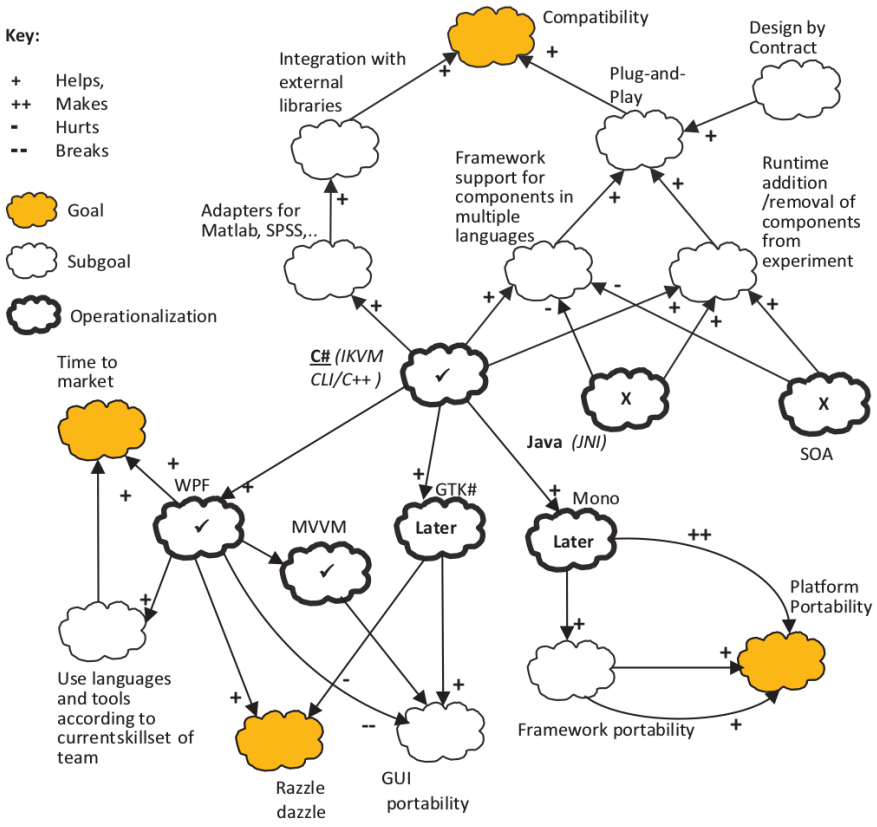


Abbildung 4.1: SoftGoal Abhängigkeitsgraph [22]

der Anforderungen keine Vorabplanung durchgeführt wird. Das Ermitteln der Anforderungen ist ein wichtiger Teil der Anforderungsanalyse (engl.: requirements engineering). Dies ist kein Teil dieser Arbeit. Im weiteren wird davon ausgegangen, dass die gewünschten Anforderungen bereits erfasst sind. Das Vorgehen zeigt stattdessen, wie verschiedene Anforderungen miteinander in Verbindung stehen und das es notwendig ist diese gegeneinander abzuwägen.

4.6 Die Rolle eines Softwarearchitekten

Eine Person die den gesamten Vorgang leitet ist bisher nicht vorgesehen. Dies kann z.B. durch eine Art von Architektenrolle geschehen. Wie bereits festgestellt ist es möglich eine an die agilen Werte angepasste Rolle in Projekten mit Scrum einzusetzen (siehe Kapitel 2.4.1). In vielen Arbeiten wird der Begriff des Architekten verwendet. Dessen Aufgaben sind meist allerdings

nicht klar beschrieben [80]. Dieser Abschnitt soll beschreiben, was ein Softwarearchitekt ist und wie dieser in einem agilen Prozess eingesetzt werden kann.

In traditionellen Methoden ist der Architekt kein Teil des Teams. Er hat vor Beginn der Implementierung eine Architektur entwickelt und diese den Entwicklern zur Realisierung vorgelegt. Seine Beteiligung am Projekt war damit meist größtenteils abgeschlossen. Die Entscheidungen wurden nicht gemeinsam mit den Entwicklern getroffen. In den meisten Fällen hatten die Entwickler kein Mitspracherecht. Dies hatte zur Folge, dass viele der geplanten Architekturen nur grob beachtet wurden. Viele Teile der Anwendung wurden deshalb durch die Entwickler selbst, ohne oder mit nur wenig Absprachen mit dem eigentlichen Architekten umgesetzt. Die geplante Architektur konnte meistens, wegen auftretenden Problem während der Entwicklung, nicht exakt wie geplant realisiert werden. [80]
Um diesen Missstand zu beheben muss der Architekt ein Teil des Teams sein und das Projekt während der gesamten Laufzeit betreuen. Während dieser Zeit dient er dem Team eher als Mentor und versucht dadurch mit Ratschlägen und Tipps zu unterstützen [6, 57]. Neben dem Mentoring sollte er sein Wissen durch Schulungen und Kurzvorträge an andere weitergeben [6]. Ambler [6] bezeichnet den Architekten als Architektur-Besitzer (*Architecture-Owner*). Dadurch ähnelt der Name mehr dem Product Owner. Der Product Owner ist verantwortlich für die Anforderungen des Teams. Der Architecture-Owner ist verantwortlich für die Architektur des Teams. Damit die Rolle den Grundlagen von Scrum entspricht darf er keine besonderen Rechte, im Vergleich zu den anderen Teammitgliedern, haben. Hierarchisch ist er auf der gleichen Ebene wie die Entwickler angesiedelt. Alle Entscheidungen werden gemeinsam im Team beschlossen. Dies wurde ebenfalls in dem Abschnitt über Entscheidungsfindung (siehe Kapitel 4.2) als gute Lösung festgestellt. Die Gefahr, dass nur der Architekt das gesamte Wissen besitzt und somit eine Gefahr bei einem Ausfall darstellt wird verringert [57, 80]. Außerdem werden Situationen wie in [80] verhindert. Hier haben sich die Entwickler übergangen gefühlten, weil nicht nach deren Meinung gefragt wurde. Einzig die Empfehlung von Ambler gibt ihm, wenn kein Konsens gefunden werden kann, das erweiterte Recht eine Entscheidung zu treffen [6]. Dadurch kann ein blockierender Konflikt aufgehoben werden. Diese Entscheidungsgewalt steht eigentlich im Konflikt mit dem Scrum Guide.
Seine Hauptaufgabe sollte es sein den Überblick über die Architektur zu behalten, um dessen Entstehung und Weiterentwicklung gezielter leiten zu können [4, 8]. Er übernimmt keine Aufgaben des Product Owners oder des Scrum-Masters. Er kann allerdings, wie der Product Owner, eine Priorisierung der Aufgaben vornehmen. Diese richtet sich nach den Kosten, den Risiken und den technischen Abhängigkeiten [57]. Der Product Owner priorisiert in erster Linie nach der Wichtigkeit der zu entwickelnden Funktionalität. Eine Aufgabe von

ihm ist es außerdem die initiale Architekturvision zu entwerfen und regelmäßige Reviews der Architektur durchzuführen. Dies führt er nicht alleine durch, sondern mit den andern Entwicklern zusammen. Die dadurch gefundenen technischen Schulden muss er managen [4]. Ein guter Architekt sollte ebenfalls ein sehr erfahrener Entwickler sein, da die Problemlösung oft ein breites Wissen über das Problemfeld und der Technik erfordert [57]. Um das Problemfeld möglichst gut zu verstehen sollten Architekten direkten Kontakt zu den Stakeholdern haben und in den gemeinsamen Meetings ebenfalls mit anwesend sein [57]. Er dokumentiert die Architektur und kommuniziert diese den Stakeholdern gegebenenfalls [4].

4.7 Architekturdokumentation

Die durch die zuvor vorgestellten Verfahren getroffene Entscheidungen müssen für eine spätere Nachverfolgbarkeit notiert werden. Dies muss auf eine einheitlich strukturierte Art und Weise geschehen. Die Informationen müssen bei Bedarf schnell gefunden und erfasst werden können. Eine zusätzliche Dokumentation der Anwendung ist wichtig. Es ist unmöglich das erforderliche Wissen, um eine Anwendung langfristig warten zu können, alleine aus dem Quellcode zu erhalten [37]. Bei einer Dokumentation muss nach den Adressaten unterschieden werden. Eine Dokumentation zur Installation und Inbetriebnahme einer Anwendung ist für den Kunden interessant und häufig ein Teil des Vertrages. Eine Beschreibung der Architektur und wie sie sich diese entwickelt hat ist für den Endkunden in den meisten Fällen stattdessen uninteressant. Häufig ist es wichtig zu wissen, warum die Architektur sich entsprechend entwickelt hat. Zu diesem Zweck können Begründungen für Entscheidungen notiert werden. Eine Begründung kann z.B. eine Anforderung des Kunden sein oder eine technische Gegebenheit. Wenn dies nicht geschieht ist es zu einem späteren Zeitpunkt schwer alle Entscheidungen nachvollziehen zu können. In einer strukturierten und übersichtlichen Dokumentation wird es möglich Anhaltspunkte für Änderungen zu finden, weil z.B. einzelne Begründungen nicht mehr zutreffen. Diese Art der Dokumentation dient in erster Linie den Entwicklern und weiteren Stakeholdern wie z.B. Personen die ein Review durchführen. Wenn die wesentlichen Informationen vorhanden sind kann sie als zentraler Bestandteil zur Kommunikation dienen [37]. Bei einer solchen Dokumentation ist es sehr wichtig, dass Beweggründe für Entscheidungen notiert werden, um diese später nachvollziehen zu können. Wenn die Beweggründe nicht mehr mit dem Ziel der Anwendung übereinstimmen kann dies ein Anzeichen dafür sein, dass eine Änderung notwendig ist.
Das Problem bei einer Architekturdokumentation ist häufig, dass diese sehr mühselig zu pflegen ist. In vielen Fällen ist ebenfalls nicht klar, wie eine Dokumentation im Idealfall aussehen sollte.

In traditionellen Vorgehensmodellen ist eine solche Dokumentation meist mehrere hunderte Seiten lang [48]. Dadurch ist es besonders schwer diese zu überprüfen, dauerhaft aktuell zu halten und in kurzer Zeit die erforderlichen Informationen zu finden.
Aus diesem Grund muss eine Möglichkeit gefunden werden, um eine leichtgewichtige Dokumentation zu erzeugen. Eine leichtgewichtige Dokumentation sollte die Anforderung erfüllen in ihrer vollständigen Länge übersichtlich zu sein. Dies soll es außerdem ermöglichen sie in einer akzeptablen Zeit zu aktualisieren. Die Erstellung soll keinen exorbitanten Mehraufwand bedeuten. Die Dokumentation soll sowohl zuvor nicht an dem Projekt beteiligten Personen, wie auch Personen die das Projekt bereits kennen ermöglichen einen Überblick über den aktuellen Stand zu erhalten. Gründe und mögliche Alternativen für getroffene Entscheidungen sollten ein fester Bestandteil sein. Um diese Anforderungen zu erfüllen muss eine Vorlage für die Dokumentation im Projekt gewählt werden. Dazu werden ins folgendem Abschnitt zwei Ansätze vorgestellt.

4.7.1 Dokumentations Frameworks

Um eine Architekturdokumentation übersichtlich zu halten sollten feste Strukturen in dieser vorhanden sein. Durch die Struktur werden die Inhalte vorgegeben. In diesem Abschnitt werden zwei Vorlagen für eine Dokumentation vorgestellt und bewertet.

Framework A Hadar u.a. haben in [48] eine Spezifikation für eine kurze Dokumentation entwickelt. Dieses Dokument soll von den Architekten pro Release gepflegt werden.
Für eine möglichst klare und einheitliche Struktur ist die Dokumentation in fünf Abschnitte unterteilt. Zu Anfang wird eine Übersicht über das Produkt gegeben. Dazu werden die Ziele, Businesswerte und potenzielle Kunden in 100 Wörtern beschrieben. Im zweiten Abschnitt wird das Produktziel für den kommenden Release beschrieben. Damit die Beschreibung nur die wesentlichen Informationen enthält ist die Länge auf maximal 100 Wörter begrenzt. Der dritte Abschnitt beinhaltet eine Architekturübersicht über das aktuelle System. Geplante Änderungen für den nächsten Release werden zusätzlich beschrieben. Dies bedeutet, dass die aktuellen und zukünftigen Hauptkomponenten mit den geplanten Änderungen beschrieben werden. Dazu zählen ebenfalls Komponenten die neu hinzukommen. Dieser Abschnitt ist der Hauptbestandteil des Dokuments und ist auf 1000 Wörter begrenzt. Der vierte Teil beschreibt die grundlegenden Anforderungen. Statt in Textform werden diese tabellarisch aufgezählt. Zu jeder Anforderung wird notiert wie die nichtfunktionalen Anforderungen die Architektur beeinflussen. Der fünfte und letzte Abschnitt beinhaltet ein tabellarisches Acronym- und

Definitionsverzeichnis. Hier werden Abkürzungen erklärt. In jedem Abschnitt ist es erlaubt auf weitere relevante Dokumentation zu verweisen. Diese können sowohl von dem Produkt selber, wie auch von ähnlichen Produkten sein.
Um ihren Ansatz zu unterstützen haben Hadar u.a. ein Tool zur Erstellung der Dokumentation entwickelt. Dieses Tool beinhaltet ein Modellierungswerkzeug. Dadurch können alle relevanten Informationen entweder durch ein Formular oder direkt in dem Modell eingetragen werden.

Dieser Vorschlage bietet durch die klare Strukturierung eine Möglichkeit relevante Informationen möglichst präzise aufzuschreiben. Die Begrenzung auf eine maximale Wortanzahl verpflichtet die Personen die schriftlichen Punkte übersichtlich und präzise zu halten. Durch die Möglichkeit auf weitere Dokumentation zu verweisen können Details zu speziellen Punkten ausgelagert werden. Das entstehende Dokument bietet dadurch einen Gesamtüberblick der Architektur und der Ziele für den nächsten Release. Bei Bedarf kann in zusätzliche, verlinkte Dokumente für weiterführende Informationen und Details gesehen werden.

Framework B Der zweite Ansatz, um eine Dokumentation durchzuführen, stammt von Van Heesch u.a [79]. Sie sagen, dass es unmöglich ist eine Architekturdokumentation zu erzeugen die für alle Stakeholder geeignet ist. Jede Gruppe von ihnen benötigt einen unterschiedlichen *Viewpoint*. Die Dokumentation einer Architkektur ist laut ihnen eine Sammlung der getroffenen Architektur-Entscheidungen. In ihrem Ansatz setzten sie vier unterschiedliche Viewpoints ein. Diese sollen einen Großteil der Stakeholder-Interessen befriedigen. Dieses Framework beinhaltet die folgende Viewpoints und deren beschriebenen Umsetzungen.

Entscheidungsdetails Um die Details einer Entscheidung zu beschreiben wurde ein Template mit fest definierten Punkten entwickelt. Folgende nur genannte Einträge sollen ausgefüllt werden: Namen, Status, Entscheidungsgruppe, Problem, Entscheidung/Lösung, Alternativen, verwandte Entscheidungen, Beziehungen zu Systemeigenschaften (z.B. funktionale und nichtfunktionale Anforderungen) und Historie. Eine genaue Beschreibung der einzelnen Punkte ist aus der original Arbeit zu entnehmen.

Entscheidungs Beziehungen In diesem Viewpoint werden mithilfe eines grafischen Netzwerkes Beziehungen zu anderen Entscheidungen dargestellt. Dies können z.B. Auslöser, Abhängigkeiten oder auch Ersetzungen alter Entscheidungen sein. Ein Beispiel für solch einen Graphen ist in Abbildung 4.2 zu sehen.

Entscheidungs Chronologie Die Entwicklung der Entscheidung wird in diesem Punkt chronologisch beschrieben. Architektur Iterationen mit einem möglichem Datum sind sicht-

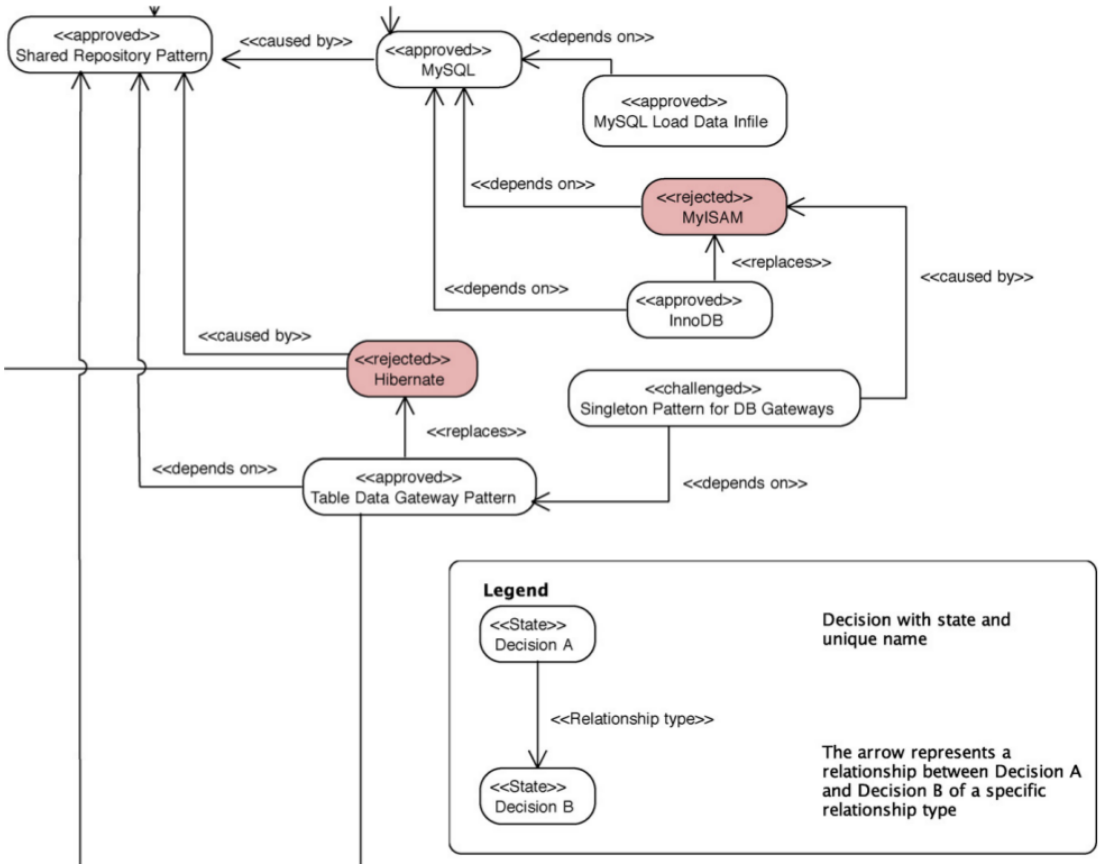


Abbildung 4.2: Entscheidungsbeziehungen als Graph [79]

bar. Dadurch kann nachvollzogen werden wann welche Entscheidungen verworfen und welche getroffen wurden.

Entscheidungs Stakeholder Beteiligung Dies beschreibt die Verantwortlichkeiten der einzelnen Stakeholder beim Entscheidungsprozess. Hier wird unter anderem beschrieben wer welche Informationen besitzt. Dies ist wichtig, da es nicht immer möglich ist alles vollständig und verständlich aufzuschreiben. Für die Darstellung kann z.B. eine ähnliche Darstellung wie in einem Use-Case Diagramm gewählt werden. In diesem sind unterschiedliche Personen, inkl. deren Rolle, eingezeichnet. Die Personen können auf verschiedene Arten mit den Entscheidungsmöglichkeiten interagieren (z.B. Empfehlen, Zurückweisen, Validieren, Bestätigen).

Dieser Ansatz hat den Vorteile, dass der gesamte Entstehungsprozess notiert wird. Dadurch wird es möglich zu erkennen wann und wer zu dieser Entscheidung beigetragen hat. Außerdem ist erkennbar welche Alternativen in Betracht gezogen wurden und entweder aus bestimmten Gründen zurückgewiesen oder einfach nicht ausgewählt wurden. Dies kann vorkommen wenn eine ebenfalls akzeptable Lösung als besser empfunden wurde. Zu einem späteren Zeitpunkt kann so geprüft werden ob diese Entscheidung noch aktuell ist. Bei einer notwendigen Änderung ist direkt ersichtlich welche weiteren Entscheidungen betroffen sind und eventuell ebenfalls angepasst werden müssen. Die Durchführung dieser Dokumentation kann durch die unterschiedlichen Grafiken schnell aufwendig werden. Es muss ein Prozess etabliert werden der annähernd zeitgleich beim Treffen der Entscheidungen die Informationen sammelt. Eine anschließende Dokumentation wird in der Regel zu ungenau.

4.8 Vorgehen zur Architekturentwicklung in Scrum

Praktische Untersuchungen in [80] konnten zeigen, dass die Qualität und die Entwicklungsgeschwindigkeit deutlich gehoben werden kann wenn das *Wer?*, *Wie?* und *Wann?* für die Architekturdebatten klar geregelt ist. Außerdem empfehlen unterschiedliche Arbeiten in [68] eine Analyse der Produktvision und der Anforderungen. Die Analyse sollte gerade bei größeren Projekten auf einer abstrakteren Ebene als den üblichen User Stories stattfinden. Dieser Abschnitt beschäftigt sich mit dem Wann und Wie. Ein Ansatz von Pérez u.a. [68] versucht diese Fragen zu beantworten. Sie haben ein Vorgehen entworfen, um in Scrum eine Architekturentwicklung zu integrieren. In den folgenden Abschnitten werden die Bestandteile des Vorgehens vorgestellt. Anschließend wird die Integration in das Scrum Framework präsentiert. Die einzelnen Bestandteile sind nicht abhängig von Scrum und können deshalb auch in anderen Verfahren zum Einsatz kommen.

4.8.1 Ermittlung der Features

Um herauszufinden was die Anwendung leisten muss werden Features definiert. Dies geschieht durch ein Brainstorming mit allen an der Entwicklung beteiligten Personen. Die Features enthalten eine kurze Beschreibung und eine eindeutige Bezeichnung. Sie werden unsortiert aufbewahrt. Pérez u.a. nennen den Aufbewahrungsort *Feature Pool*. Die Features sind meist rein funktional (z.B. Zugriff auf Daten XY muss möglich sein). Durch die Zusammenarbeit mit dem Kunden müssen dazugehörige nichtfunktionale Anforderungen ermittelt werden (z.B.

Performance des Zugriffes). Diese können auf gleiche Weise, wie Features verwendet und aufgeschrieben werden.

Anschließend werden die Features auf Verbindungen untersucht. Dabei wird hauptsächlich darauf geachtet ob diese Verfeinerungen oder Abhängigkeiten voneinander sind. Die Beziehungen können in Form eines Baums dargestellt werden (*Feature Baum*). Die Wurzel stellt die Anwendung selbst dar. Bei einer Planung mit mehreren Releases werden die Features gruppiert. Dies geschieht mithilfe des *Olympischen Feature Pools*. Es werden *Schwimmbahnen* eingezogen und die Features nach Priorität und Realisierungsreihenfolge einsortiert. Der zuvor erzeugte Baum kann die Gruppierung unterstützen, weil er vorhandene Abhängigkeiten darstellt. Die Abhängigkeiten müssen der Reihe nach Implementiert werden. Die Gruppierung führt eine sichtbare Priorisierung ein.

Durch die unterschiedlichen Darstellungen wurde aufgenommen was entwickelt werden soll (Feature Pool), wann dies entwickelt werden soll (Olympischer Feature Pool) und wie dies entwickelt werden soll (Feature Baum, aufgrund der Abhängigkeiten).

Für die Entwicklung innerhalb des Scrum Zyklus werden die Features anschließend in User Stories oder Epics umgewandelt. Diese befüllen das Product Backlog. Epics müssen vor der Entwicklung in User Stories umgewandelt werden. Die Umwandlung und die Bedingungen für User Stories werden hier nicht beschrieben.

4.8.2 Entwicklung der Softwarearchitektur

Eines der agilen Prinzipien sagt aus, dass zu jeder Zeit Änderungen willkommen sein sollen. Die gesamte Entwicklung ist darauf ausgerichtet schnell auf Änderungen, z.B. durch neue Anforderungen, reagieren zu können. Dies ist wichtig, weil dadurch das Ziel am Anfang der Entwicklung nicht vollständig verstanden und definiert sein muss. Eine Architektur ist in den meisten Fällen allerdings starr und schwerfällig gegenüber Anpassungen die große Teile der Anwendung betreffen. Trotz der Schwierigkeit ist es wichtig, dass die Architektur bei jeder Iteration an die aktuellen Bedürfnisse angepasst werden kann. Im Folgenden wird erläutert, wie eine Architektur mit Beachtung der agilen Prinzipien entwickelt werden kann.

Plastic Partial Components

Die *Plastic Partial Components* (PPC, formbare unvollständige Komponenten) bieten die Möglichkeit eine flexible Architektur zu entwickeln. Dieses Konzept wurde in unterschiedlichen

Arbeiten eingeführt [45, 67, 68]. Eine PPC ist eine spezialisierte Komponente, welche alle Eigenschaften und das Verhalten anderer Komponente erbt. Ausschließlich Basis-Funktionalitäten, welche nicht ererbt werden können, werden innerhalb der PPC realisiert. PPCs können durch *Variants* angepasst werden. Variants stellen die Realisierung von Features, die nicht relevant genug sind um eine Hauptkomponente der Anwendung zu sein, dar. Sie fügen der Anwendung eine zusätzliche Funktionalität hinzu. Es ist denkbar, dass diese im Laufe der Zeit entfernt oder an weiteren Stellen benötigt werden. Kenntnisse wie sie gebunden sind besitzen sie nicht. Dadurch wird es möglich ähnliche Komponenten wiederzuverwenden indem nur einzelne Bestandteile ausgetauscht werden. Die Verbindung von PPCs und Variants geschieht durch *Variability Points*. Sie beinhalten die Definition der Verbindung und dessen Verwendung.

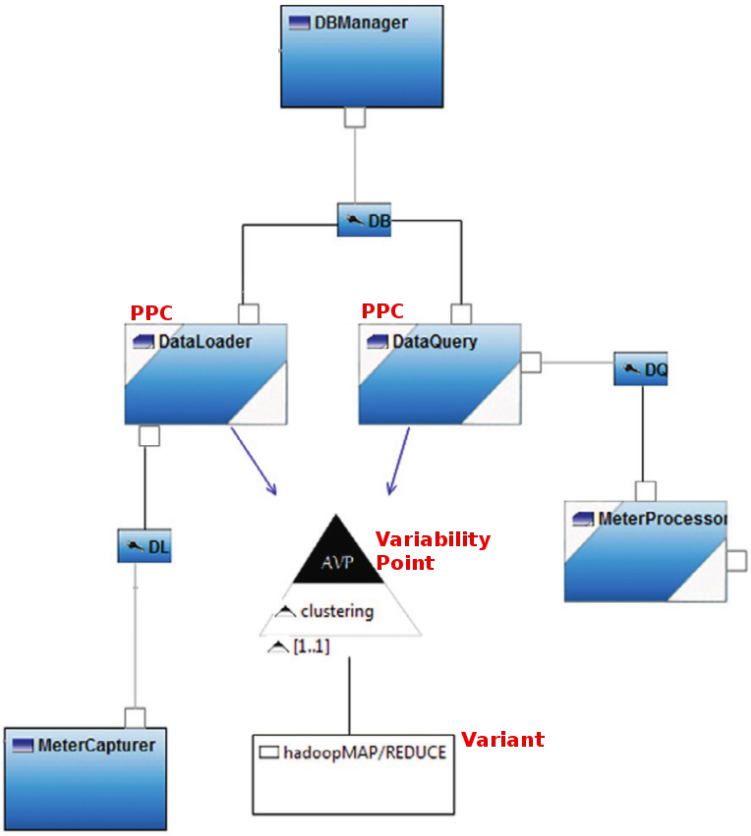


Abbildung 4.3: Plastic Partial Components [68]

In Abbildung 4.3 ist ein Ausschnitt eines Diagramms zu sehen, bei dem die zuvor beschriebenen Bestandteile vorhanden sind. *DataLoader* und *DataQuery* sind hier PPCs. Beide besitzen den Variability Point für *clustering*, welcher durch die *hadoopMap/Reduce Variant* realisiert wird.

PPCs sind in erster Linie für die Entwicklung von mehreren nicht identischen aber sehr ähnlichen Systemen gedacht. Bei diesen können einzelne Teile deaktiviert oder ausgetauscht werden. Diese Art der Entwicklung wird auch *Product Line Engineering* genannt. Eine schnelle Möglichkeit, um auf Änderungswünsche reagieren zu können, ist damit nicht gewährleistet. Diese Variabilität wird häufig bei Service-basierten Anwendungen eingesetzt. Diese können weitestgehend beliebig zusammengesetzt werden. [45]
Dieser Ansatz erfordert allerdings eine gewisse Vorausplanung, um mögliche Variability Points zu identifizieren. Um diese Up-Front Planung zu umgehen muss diese Lösung mit einem agilen Vorgehen vereint werden. Dies erhöht die Flexibilität der Anwendung. Auf Änderungswünsche kann somit besser reagiert werden.

Aus dem Ansatz der PPCs wurde das Konzept der *Working Architecture* entwickelt. Dieses hat das Ziel den Prozess agil zu gestalten. Jede Komponente der Architektur wird als PPC realisiert. Die PPCs werden nach Kundenwunsch iterativ erweitert und verändert. Aufwendige Refaktorisierung sollen durch regelmäßige kleine Anpassungen verhindert werden. Bei der Entwicklung wird initial eine Basisarchitektur entworfen. Die Entwicklung findet anhand der bekannten User Stories statt. Pro Iteration (in Scrum Sprint) werden User Stories ausgewählt. Nach der Auswahl werden die PPCs, Variants und Variability Points identifiziert und anschließend realisiert. Die variable Architektur wird schrittweise erweitert. [56]

4.8.3 Integration in Scrum

Scrum ist ein Framework und darf nach belieben an die Bedürfnisse angepasst werden. Im Folgenden wird eine Anpassung von Pérez u.a. [68] vorgestellt. Diese integriert die zuvor vorgestellten Themen in das Vorgehen. Das vollständige Vorgehen ist in Abbildung 4.4 zu sehen. Bei dieser Variante von Scrum wird wie üblich mit dem Product Backlog gearbeitet. Dieses wird durch die zuvor beschriebene Methode zur Ermittlung der Features gefüllt. Hier kommen der vorgestellte Feature Pool, Feature Tree und Feature Olympic Pool zum Einsatz (siehe Kapitel 4.8.1). Die entwickelten Features werden in Stories umgewandelt und in das Product Backlog übertragen. Wie bei Scrum üblich beinhalten diese die Akzeptanzkriterien aber keine technischen Details.
Vor dem Beginn eines Sprint-Zyklus wird das Backlog-Grooming (bzw. Refinement zum Priorisieren, Teilen, Schätzen u.s.w. der Stories) durchgeführt. Dies ist keine Erneuerung und ist bereits in Scrum vorhanden. Die Stories werden zu diesem Zeitpunkt genauer betrachtet. Da tiefer in das Backlog gesehen wird bewirkt dies, dass mögliche zukünftige technische Herausforderungen erkannt werden können. Neu ist der Punkt des *Sprint Agile Architecting*. Dieser

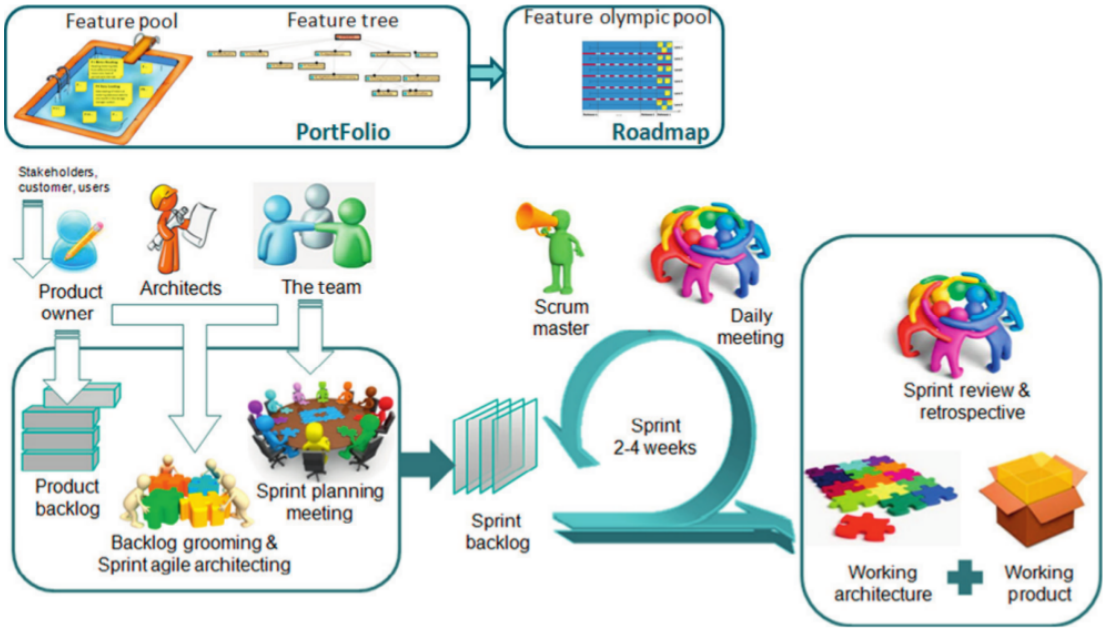


Abbildung 4.4: Integration einer Architekturentwicklung in Scrum [68]

findet im Meeting zum Backlog Grooming statt. Für die ausgewählten Stories werden die PPCs, Variants und Variability Points identifiziert. Hierdurch wird festgestellt ob die Working Architecture aktualisiert werden muss. Beide Teile dieses Meetings finden zeitgleich statt. Eine weitere Aufteilung in Abschnitte ist nicht vorgesehen. Dies ist sinnvoll, da unter anderem für ein zuverlässiges Schätzen der Aufwände erforderlich ist zu wissen, wie diese technisch realisiert werden sollen. Es fällt auf, dass die Personenrolle des Architekten hinzugekommen ist. Mehrere Architekten können zeitgleich Teil des Teams sein. Die Rollen des Architekten wurde zuvor in 4.6 vorgestellt. Der anschließende Schritt ist das Sprint Planning Meeting. Dies stammt ebenfalls aus dem originalen Scrum und dient zur Auswahl der Stories für den nächsten Sprint. Am Ende des Sprints steht die *Working Architecture* und das *Working Produkt* zur Auslieferung an den Kunden bereit. Das Daily Meeting, das Sprint-Review und die Retrospektive werden wie zuvor übernommen.

4.9 Model-Driven Engineering

Ein Ansatz um eine Software gesteuert durch eine Architektur und deren Modelle zu entwickeln ist das *Model-Driven Engineering* (MDE).

Model-Driven Engineering wird häufig auch als *Model-Driven Development* (MDD) oder *Model-Driven Architecture* (MDA) bezeichnet. Bei der Entwicklung von Architekturen wird häufig mit einfachen Modellen begonnen. Grobe Strukturen und Kommunikationswege werden visuell dargestellt. Modelle haben den Vorteil, dass diese für den Menschen einfacher verständlich sind. Anhand von Quellcode ist es schwer Strukturen in kurzer Zeit zu erkennen. Die Modelle werden im Laufe der Entwicklung dauerhaft erweitert, verfeinert und angepasst.

Die Idee beim Model-Driven Development ist aus den bereits vorhandenen Modellen mithilfe von Werkzeugen Quellcode zu generieren. Eine automatische Generierung soll helfen die Entwicklungsgeschwindigkeit zu erhöhen und die Komplexität zu verringern. Ein großer Teil der gewonnene Zeit die nicht für die Implementierung benötigt wird, muss stattdessen in die Modellierung investiert werden [50]. Die Komplexität soll geringer werden, weil die Anwendung über ein einfach bedienbares Werkzeug, statt einer manuellen Implementierung erzeugt werden kann. Die Modelle müssen zu diesem Zweck detailliert nach den Richtlinien der Tools entworfen werden. Nur in diesen Fällen ist eine sinnvolle und erfolgreiche Generierung möglich. Dies erfordert die klare Einhaltung von Regeln. Bei Modellen die für Menschen gemacht sind ist es einfacher und hilfreicher nicht streng nach Vorschrift der Modellierungsart, bzw. Sprache zu arbeiten. Es ist ausreichend wenn die Ergebnisse verständlich sind. Zusätzlich entsteht eine starke Abhängigkeit gegenüber der Tools.
Das Problem bei diesem Ansatz ist, dass eine Generierung nie vollständig möglich ist. Nachbesserungen sind erforderlich, weil nicht jedes Problem automatisch gelöst werden kann. Dies kann sein, weil Informationen zu dem Zeitpunkt der Generierung fehlten, widersprüchlich waren, sich häufig änderten oder die Werkzeuge entsprechende Konstrukte nicht beherrschen. [44] Aus diesen Gründen ist manuelle Nacharbeit erforderlich. Diese Nacharbeit umfasst die Implementierung von einzelnen Details und Anpassungen im generierten Code. Das Problem bei der Anpassung im generierten Code ist, dass dieser schwer lesbar ist. Da dieser nicht von einem Menschen geschrieben ist werden keine aussagekräftigen Benennungen oder für den Menschen schwer lesbare Strukturen verwendet. Eine einfache Anpassung der Modelle wird dadurch schwer, bis unmöglich.
Bei einer Aktualisierung des Modells ist es den Tools meist nicht möglich die händischen Anpassung zu übernehmen. Es ist deshalb erneut erforderlich die Anpassungen im Code durchzuführen.
Die Entwicklung mithilfe von MDE ist sehr viel abstrakter als die manuelle Realisierung. Eine höhere Abstraktion führt aber nicht automatisch zu einer besseren Software geben Hutchinson u.a. [50] zu bedenken.

Bei MDD ist noch viel Arbeit notwendig. Statt wie gewünscht die Komplexität zu verringern, wird sie teilweise sogar erhöht [44]. Ohne die Generierung von Modellen wird die Komplexität der Architektur allerdings ebenfalls erhöht, da die Entstehung einer Accidental Architektur gefördert wird. Hutchinson u.a. [50] haben in ihrer Studie herausgefunden, welche Modellierungsarten häufig verwendet und wofür Modelle außerdem eingesetzt werden. Weitere Studien dieser Art und über den kokreten Einsatz eines solchen Verfahrens sind erforderlich.

4.9.1 Agile Model Driven Development

Im zuvor vorgestellten MDE ist der agile Aspekt vollkommen ausgelassen. Ambler stellt in [5] sein Konzept des *Agile Model Driven Developments* (AMDD) vor. In Abbildung 4.5 ist das Vorgehen abgebildet. In diesem Ansatz werden vor der konkreten Realisierung Modelle

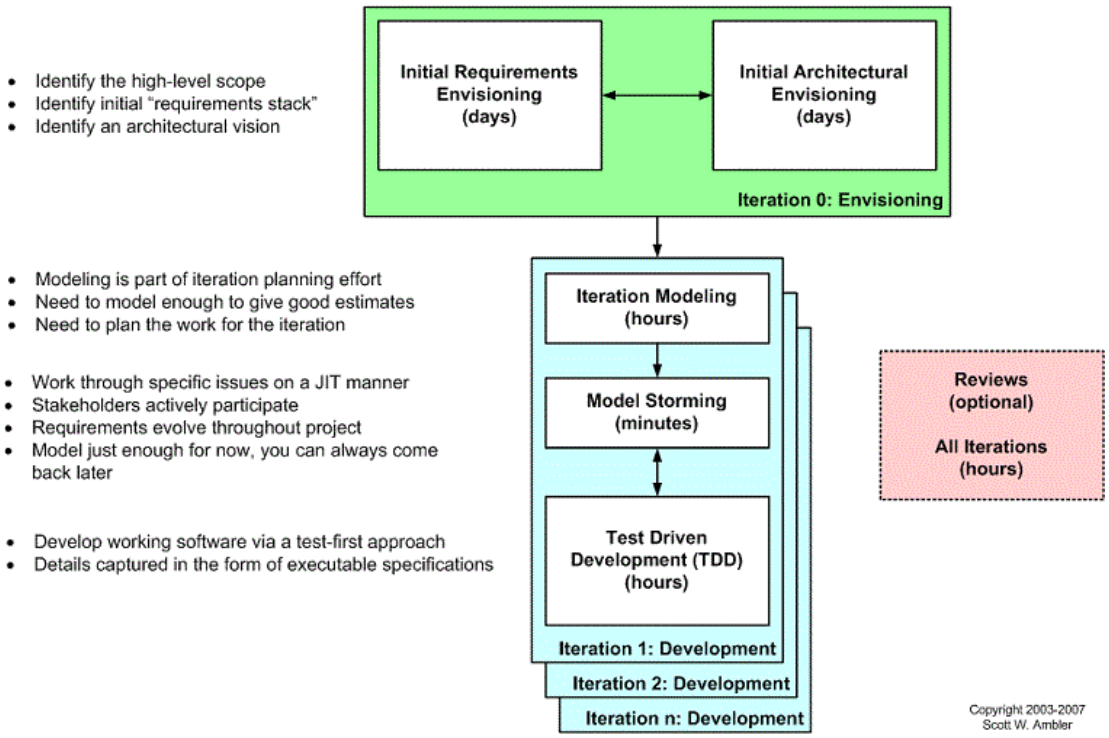


Abbildung 4.5: Agile Model Driven Development [5]

entworfen. Sie sollen zeigen wie etwas umgesetzt wird. Der Unterschied zum ursprünglichen MDD ist, dass nur Modelle entworfen werden die für die nächsten Schritte gerade ausreichend

sind. Aus den bei MDD genannten Gründen wird auf die Generierung von Quellcode verzichtet. Die Modelle werden vollständig manuell implementiert.

Die Entwicklung startet mit der Iteration 0. Dieser Abschnitt ist kein Teil der iterativen Entwicklung und wird nur einmalig durchlaufen. In dieser Phase werden die High-Level Anforderungen identifiziert. Dazu wird allgemein analysiert wie die Benutzer mit dem System arbeiten sollen. Ein allgemeines Verständnis über das Gesamtziel soll geschaffen werden. Eine Basisarchitektur mit Berücksichtigung von Gegebenheiten, wie die Infrastruktur, wird entwickelt. Dazu können z.B. die allgemein notwendigen Komponenten identifiziert werden. Die Modellierung soll möglichst einfach geschehen. Dazu wird nur das modelliert was aktuell wichtig ist. Dies entspricht dem Prinzip der maximalen Einfachheit aus dem agilen Manifesto [43]. Eventuelle zukünftige Anforderungen werden nicht berücksichtigt. Technisch soll die Modellierung möglichst einfach, z.B. mit einem Whiteboard, ablaufen. Die Zeit die in diese Phase investiert werden muss ist sehr von der geplanten Gesamtdauer des Projekts abhängig. Sie kann entsprechend zwischen ein paar Stunden und mehreren Wochen liegen.

Nach der initialen Phase startet die iterative Entwicklung. Erst in dieser Phase wird Quellcode produziert. Zu Beginn jeder Iteration findet ein Modellierungsprozess statt (*Iteration Modelling*). In diesem Abschnitt wird geplant was in der aktuellen Iteration umgesetzt werden soll. Außerdem werden Aufwandsschätzungen durchgeführt. Für möglichst genaue Schätzungen wird darüber diskutiert wie die einzelnen Aufgaben implementiert werden sollen. Um ein konkretes Problem zu lösen findet direkt vor der Umsetzung nochmals ein spontanes *Model Storming* statt. An diesem nimmt eine Auswahl von relevanten Entwicklern teil. Es soll nicht allzu viel Zeit investiert werden. Häufig wird dies zu zweit, wie eine von Art Pair Programming durchgeführt. Die eigentliche Entwicklung geschieht mit dem testgetriebenen Ansatz (*Test Driven Development* - TDD). Vor der Implementierung der Lösung werden Tests, die das Ziel beschreiben, umgesetzt. Erst anschließend findet die Realisierung statt. Die Lösung wird schrittweise durch refaktorisieren verbessert. Erst wenn alle Test korrekt durchlaufen ist die Aufgabe abgeschlossen.
Optionale Reviews, z.B. durch Codereviews, sind möglich. Hilfreich können diese eventuell bei komplexen Problemen sein.

Dieses Vorgehen beschreibt zu welchen Zeitpunkten modelliert werden kann. Es beschreibt allerdings nicht auf welcher Basis Entscheidungen getroffen werden. Theoretisch könnten diese Ideen ebenfalls mit dem Scrum Vorgehen, ähnlich wie in Kapitel 4.8, verbunden werden. In diesem Fall würde auf der Basis von Stories und Tasks gearbeitet werden. Das Iteration Modelling könnte ein Teil des Sprint Plannings werden.

4.10 Fazit

In diesem Kapitel wurden Möglichkeiten zur agilen Architekturentwicklung aus der Literatur diskutiert. Zu Anfang wurden unterschiedliche Arten der Entscheidungsfindung vorgestellt. Bei der agilen Entwicklung ist es wichtig, dass alle Teammitglieder die Entscheidungen verstehen und unterstützen. Wenn dies nicht der Fall ist entstehen schnell Konflikte wodurch die Entwicklung ins Stocken geraten kann.
Um die Vorteile, die eine agile Entwicklung bewirkt, nicht zu gefährden müssen gewissen Prinzipien bei den Architekturdebatten berücksichtigt werden. Dazu wurden Vorschläge um die agilen Prinzipien auf die Architekturdebatte abzubilden analysiert.
Im nächsten Schritt wurde untersucht, woher Anforderungen für eine Architektur stammen können. Hierbei kann es sich z.B. um die NF-Anforderungen handeln. Zur Unterstützung dieses Ansatzes wurden Personas verwendet welche bereits zuvor im agilen Kontext bekannt waren. Dieser Ansatz unterstützt sowohl die Identifikation, wie auch die Dokumentation der NF-Anforderungen. Da sich Anforderungen schnell überschneiden und sich widersprechen können, wurde eine Möglichkeit vorgestellt, um die Anforderungen gegeneinander abzuwägen.
Anschließend wurde die traditionelle Rolle des Softwarearchitekten auf den Einsatz in einem agilen Vorgehen hin untersucht. Die ursprüngliche Rolle hat nicht mehr sehr viel gemeinsam mit der Rolle in einem agilen Vorgehen. Die ursprüngliche Trennung vom Team widerspricht den agilen Werten. Der moderne Architekt entwickelt die Architektur nicht mehr selbst. Stattdessen unterstützt er das gesamte Team bei dieser Aufgabe.
Im nächsten Schritt wurde vorgestellt, wie eine Architektur dokumentiert werden kann. Für eine Dokumentation ist wichtig, dass diese in einem einheitlichem Format erstellt wird. Dadurch wird es einfacher sich in dieser zurecht zu finden. Außerdem ist wichtig, dass dessen Länge begrenzt wird. Ansonsten wird diese sehr schnell unübersichtlich.
Zum Ende des Kapitels wurden zwei Vorgehen präsentiert, welche einen konkreten Entwicklungsprozess beschreiben. Das erste Vorgehen hat das Ziel die Entwicklung in den allgemein bekannten Scrum Prozessen zu integrieren. Durch dessen Verbreitung ist eine Einführung dieses Prozesses in vielen Unternehmen einfacher durchführbar. Hier wird an unterschiedlichen Schritten des Prozesses angesetzt. Hierzu gehört die Ermittlung der Anforderungen und eine anschließende Möglichkeit eine variable Architektur zu bauen. Die einzelnen Schritte können auch unabhängig von Scrum eingesetzt werden und dadurch ähnliche Vorgehen unterstützen.
Der zweite Ansatz war ein Modell-getriebener Ansatz. Dies heißt, dass die Entwicklung anhand von zuvor modellierten Lösungen durchgeführt wird. Die größte Herausforderung hierbei ist, dass dieser Ansatz agil bleibt und deshalb immer nur schrittweise modelliert werden darf.

Die in diesem Kapitel angesprochenen Ansätze ermöglichen eine Integration von Architekturdebatten in eine agile Entwicklung. Teilweise sind die vorgestellten Prozess in einzelnen Projekten testweise eingesetzt worden. In dieser Arbeit wurden nur die Ansätze vorgestellt und nicht deren praktische Evaluationen. Der Scrum Ansatz wurde z.B. in einem Projekt im Rahmen der Forschung eingesetzt. Hierbei wurde festgestellt, dass zum Einsatz Schulungen und harte Disziplin notwendig sind. Ansonsten wurde schnell wieder nach dem originalem Scrum gearbeitet. Erst nach einiger Zeit, als die Anwendung komplexer wurde, konnten Vorteile erkannt werden. Die Architekten konnten helfen die Komplexität durch Priorisieren von Aufgaben, Refaktorisierungen und Restrukturierungen der Architektur, zu managen. [68]
Allgemein wurde festgestellt, dass Tools zur Unterstützung immer möglichst einfach gestaltet sein sollten [5]. Dies geht sogar so weit, dass statt einer digitalen Anwendung zum Modellieren, ein Whiteboard verwendet werden soll [83]. Es ist ausreichend wenn die Ergebnisse als Foto festgehalten werden.
Die hier vorgestellten Thematiken wurden durch theoretische Ansätze entwickelt. Zum Großteil gab es einzelne Testprojekte, um den Vorgang zu evaluieren. Ein Einsatz in der Masse ist allerdings unbekannt. Ein allgemeines, starres Vorgehen ist in den meisten Fällen nicht einsetzbar. Durch die Unterschiede in den Firmen, Organisationsformen und Projekteigenschaften ist es erforderlich individuelle Anpassungen vorzunehmen.

Im nächsten Schritt soll anders vorgegangen werden. Es werden Untersuchungen in unterschiedlichen Unternehmen durchgeführt. Diese haben das Ziel herauszufinden wie deren Entwicklung organisiert ist. Durch die Erfassung der realen Entwicklungsprozesse soll untersucht werden, welche der theoretischen Vorschläge in der Praxis eingesetzt werden. Außerdem wird es möglich aus den vorhandenen Vorgehen einen neue Vorschlag zu entwickeln, wie eine agile Entwicklung mit Architekturdebatten stattfinden kann.

5 Interviews zur Untersuchung der Architektur-Diskussionen

In diesem Kapitel werden Untersuchungen in der Praxis durchgeführt. Bei den Untersuchungen soll festgestellt werden, was unternommen wird, um die Architekturerosion zu managen. Es soll herausgefunden werden, wie Firmen mit der Thematik der Architekturentwicklung umgehen und was getan wird, um diese möglichst optimal zu entwickeln.

Um dies zu untersuchen wurden unterschiedliche Firmen und Projekte als Fallstudien untersucht. Ziel war es Erfahrungen zu sammeln, wie eine Architekturdebatte in realen Projekten aussehen kann. Die praktisch gesammelten Aspekte sollen dazu dienen einen Vorschlag zu entwickeln, wie eine aktive Architekturdebatte in den agilen Scrum Prozess integriert werden kann. Es wird außerdem untersucht, welche der in den vorherigen Kapiteln vorgestellten Konzepte in der Praxis eingesetzt werden. Wenn keine Debatten durchgeführt werden, soll festgestellt werden, wie die Entwicklung ohne diese stattfindet und welche Gründe dafür existieren.

Zu Anfang dieses Kapitels wird das allgemeine Vorgehen erläutert. Dazu gehören unter anderem die Kriterien zur Auswahl der Firmen. Zur Informationserfassung dienen Experteninterviews. Um diese zu unterstützen wird ein Interviewleitfaden entwickelt. Dieser wird vorgestellt. Besonderheiten der einzelnen Interviews werden in Einzelauswertungen herausgearbeitet. Darauf folgen Gesamtauswertungen zu speziellen Themenschwerpunkten. Ein weiterer Bestandteil ist das Aufstellen einer Theorie, wie die Beobachtungen in ein Scrum Vorgehen eingebaut werden können. Zuletzt wird das Kapitel zusammengefasst und ein allgemeines Fazit gezogen.

5.1 Vorgehen und Methodik

Dieser Abschnitt beschreibt das Vorgehen für die Durchführung der Interviews.

Ursprünglich war geplant die Erfassung der Praxis in zwei Schritten durchzuführen. Im ersten Schritt sollten Beobachtungen, ohne in den Vorgang einzugreifen, durchgeführt werden. Zu

diesem Zweck sollten ein bis zwei Meetings als nicht teilnehmender Beobachter in verschiedene Firmen besucht werden. Diese Meetings sollten, sofern vorhanden, für Architekturdebatten gedacht sein. Es sollten Besprechungen sein die Teil des normalen Prozesses sind und nicht extra wegen der Untersuchung angesetzt werden. Ziel war es festzustellen, wann und wie technische Architekturdebatten durchgeführt werden. Durch mehrere Beobachtungen des gleichen Projektes in der gleichen Firma wird es möglich einen Verlauf zu erkennen. Das zweite Ziel war die Identifikation von potentiellen Interviewpartnern.
Es stellte sich allerdings als problematisch heraus Termine in den Firmen zu bekommen. Die angefragten Firmen waren offen für Beobachtungen, wussten aber nicht wann ein Meeting dieser Art stattfinden wird. Durch den ungenauen Zeitrahmen wurde eine Planung schwer. Zusätzlich entstand schnell die Meinung, dass es schwer sein wird in kurzer Zeit viele hilfreiche Informationen erfassen zu können. Als weitere Gefahr wurde die Beeinflussung des Ablaufes durch eine fremde Person im Meeting gesehen. Aufgrund dieser Schwierigkeiten wurde beschlossen die Erfassung auf Experteninterviews zu konzentrieren. Die Interviewpartner wurden deshalb durch die Kontaktpersonen vorgeschlagen. Die Interviews sollten Informationen, Meinungen und Ansätze liefern. Der Erfahrungswert des Interviewpartners spielte deshalb eine wichtige Rolle.

Die Untersuchungen sollten anhand von realen Projekten in der Industrie durchgeführt werden. Theoretische Untersuchungen oder Untersuchungen in Projekten, die nur dem Forschungszweck dienen, können die Realität nicht vollständig abbilden. Ein reines Forschungsprojekt könnte zum Beispiel mit Studierenden durchgeführt werden. Bei dieser Untersuchung müsste zuvor ein konkretes Vorgehen entwickelt werden, welches evaluiert werden soll. Die Studierenden haben allerdings nicht die gleiche Erfahrung, wie Personen die längere Zeit in der Praxis arbeiten. Ebenfalls könnte das exemplarische Projekt nur über einen stark begrenzten Zeitraum, dessen Länge für ein reales Projekt nicht realistisch ist, gehen.

Die Interviews dienen nicht nur der reinen Informationsbeschaffung. Ihr Hauptzweck ist stattdessen das Erfassen von firmenspezifischem Prozesswissen [3]. Die Untersuchungen stellen Fallstudien dar. Sie stellen eine qualitative, statt einer quantitativen Forschung dar.

5.1.1 Auswahl der Firmen und Projekte

Dieser Abschnitt beschreibt die Auswahl und die erste Kontaktaufnahmen mit den Unternehmen.

Ziel war es zwischen fünf und acht Unternehmungen zu besuchen. Es sollten mindestens fünf Unternehmen sein, um eine breitere Informationsbasis zu erhalten und maximal acht, um den zeitlichen Aufwand in einem akzeptablen Rahmen zu halten.
Bei der Auswahl wurde auf ein unterschiedliches Geschäftsfeld geachtet. Dazu zählen Firmen die im Projektgeschäften arbeiten. Dies heißt, diese Firmen haben unterschiedliche Projekte mit verschiedenen Kunden. Bei diesen Projekten ist meist ein festes Ende vordefiniert. Andererseits wurden Unternehmen mit Produktgeschäft ausgewählt. Diese haben ein oder mehrere eigene Produkte. Sie Entwickeln, Pflegen und Vermarkten diese. In der dritten Kategorie wird nur für interne Kunden entwickeln. Das entwickelte Produkt wird nicht vermarktet, sondern nur firmenintern eingesetzt. Alle Entscheidungen die bei den letzten beiden Kategorien getroffen werden sind mit großer Wahrscheinlichkeit zu einem späteren Zeitpunkt erneut relevant. Es kann passieren, dass die Entwickler sich nochmals mit ihnen beschäftigen müssen.

Bei der Auswahl der Firmen haben ein paar Faktoren eine wichtige Rolle gespielt. Zum Einen sollte innerhalb des Unternehmens eine Personen als Einstiegspunkt bekannt sein. Diese sollte für die erste Kontaktaufnahme dienen. Deshalb musste nicht über eine allgemeine Kontaktmöglichkeit das Gespräch aufgenommen werden. Hierbei hätte es sich z.B. um ein Kontaktformular auf der Webseite handeln können. Die Wahrscheinlichkeit, dass diese Nachricht nicht beachtet würde, wurde als sehr hoch betrachtet. Die bekannte Person sollte stattdessen als Einstiegspunkt dienen, um direkt konkrete Ansprechpartner zu vermitteln und im Notfall nochmals nachhaken zu können. Dieser Verdacht kann dadurch bestärkt werden, dass es bereits in manchen Fällen über die direkte Kontaktaufnahme schwierig war Antworten zu erhalten. Vor dem Beginn der Untersuchungen wurde mit drei der acht Unternehmen ein Kennlerngespräch zur Absprache der Rahmenbedingungen durchgeführt. Zu dieser Zeit waren die Beobachtungen noch ein Teil des Plans.
Als Interviewpartner werden Einzelpersonen, als Vertreter für das gesamte Team, ausgewählt. Diese sollten die Architekten oder zumindest in irgendeiner Weise verantwortlich für die Architektur sein. Alternativ sollte eine Person gewählt werden, die durch ihre Teilnahme am Prozess detailliert über diesen berichten kann.
Die ausgewählten Unternehmen werden gemeinsam mit den Interviewberichten im Anhang genauer vorgestellt.

5.1.2 Experteninterviews

Die Interviews dienen zur Erfassung von Prozesswissen. Es soll herausgefunden werden, wie der Prozess der Architekturentwicklung abläuft. Die vorhandenen Handlungsabläufe und Inter-

aktionen zwischen verschiedenen Personengruppen sind dabei ebenfalls relevant. Es handelt sich hierbei dementsprechend nicht um Fachwissen, welches Objektiv und relativ einfach nachzulesen ist. Dieses Wissen kann nur durch eine Teilnahme an dem Prozess, Beobachtungen oder Befragungen erlangt werden. Neben dem Prozesswissen wird zusätzlich Deutungswissen ermittelt. Dies bedeutet, es wird nach subjektiven Meinungen und Bewertungen der Personen gefragt. Das Deutungswissen soll Vor- und Nachteile in den eingesetzten Vorgehen ermitteln und als Meinung zu verschiedenen vorhandenen Techniken und Methoden dienen. [3]

Zur Vorbereitung und zur Unterstützung der Interviews wurde ein Leitfaden mit Themen und Fragestellungen entwickelt. Die Fragen sollen während des Interviews beantwortet werden. Sie werden nicht direkt gestellt, sondern sollen in den Verlauf des Gesprächs integriert werden. [3]
Der Leitfaden wird anschließend in Kapitel 5.2.1 vorgestellt. Die Durchführung soll wenn möglich in den jeweiligen Firmen der Personen stattfinden. Damit der Interviewer sich besser auf das Gespräch konzentrieren kann werden wenige Notizen gemacht. Stattdessen soll eine Audioaufzeichnung stattfinden. Diese dient im Nachhinein als Dokumentation und zur Formulierung der Ergebnisse. Zum Schutz der Privatsphäre werden die Aufnahmen nicht veröffentlicht oder weitergegeben.

5.1.3 Nachbereitung und Auswertung

Dieser Abschnitt beschreibt die Nachbereitung der Interviews. Das Vorgehen für eine Auswertung und der darauf folgenden Fazitbildung wird beschrieben. Zum Schutz der Privatsphäre der Personen und um keine kritischen Informationen über die Firmen preiszugeben werden alle Informationen anonymisiert beschrieben und ausgewertet.

Zur Nachbereitung der Interviews werden die Ergebnisse verschriftlicht. Aus diesem Grund werden mithilfe der Audioaufnahmen Texte verfasst, die die Inhalte wiedergeben. Die Darstellung der Informationen spiegelt nicht zwingend den korrekten zeitlichen Ablauf wieder. Stattdessen wird darauf geachtet inhaltlich zusammengehörige Informationen zusammen vorzustellen. Die Informationen können im Interview zu verschiedenen Zeitpunkten angesprochen worden sein. Die Informationsberichte geben nicht den direkten Wortlaut der Interviewpartner wieder, sondern werden für ein besseres Verständnis umformuliert. Die Verschriftlichung der Inhalte ist in den Anhängen B bis I zu finden. Die dort dargestellten Informationen beruhen auf den persönlichen Meinungen und Einschätzungen der Interviewpartner.

Die Interviews werden in zweit Schritten ausgewertet. Im ersten Schritt wird eine Einzelauswertung durchgeführt. In dieser wird kurz vorgestellt, was für die Architekturentwicklung unternommen wird. Hier wird besonders darauf geachtet, welche der in den vorherigen Kapiteln beschriebenen Praktiken eingesetzt werden. Im zweiten Schritt wird eine Analyse, angelehnt an die qualitative Inhaltsanalyse von Mayring [59], durchgeführt. Die Informationen aus den Interviews werden nicht als Fakten betrachtet. In anderen Teams und Unternehmen können diese vollständig anders aussehen. Stattdessen werden sie als Deutungswissen interpretiert. [3] Für die Analyse wurde ein Kodierleitfaden entwickelt. Dieser befindet sich im Anhang J. Zu den Punkten aus dem Kodierleitfaden werden Antworten in den Interviews gesucht. Die Zitate dienen als Ankerbeispiele und werden bei der Auswertung eingesetzt. Aus den Zitaten wurden Füllwörter entfernt und wenn notwendig der Satzbau korrigiert.

Der folgenden Schritt, sieht eine Theorieentwicklung als Fazit der gewonnen Erkenntnisse vor. Zu diesem Zweck wird ein Vorschlag entwickelt, der eine aktive Architekturentwicklung in das Scrum Framework einbettet. Dazu werden Ideen aus den vorherigen Kapiteln mit Erkenntnissen aus der Praxis kombiniert.

5.2 Themenauswahl

Dieser Abschnitt beschreibt den Prozess und das Ergebnis der Leitfadenerstellung.
Um die Interviews zu strukturieren wurden im Vorhinein Themen, die mit der Architekturentwicklung zusammen hängen können, erarbeitet. Diese wurden in einer Mindmap gesammelt. Die Mindmap ist in Anhang K dargestellt. Im Rahmen der Untersuchungen und Recherchen konnten acht relevante Gruppierungen identifiziert werden. Innerhalb der Interviews werden nicht alle Themen vollständig behandelt. Die Kategorie der Anforderungen wurde vollständig ausgelassen. Von den weiteren Themenbereichen wurden ebenfalls nicht alle Unterthemen verwendet. Die Bereiche wurden während des Interviews teilweise miteinander kombiniert. Die genaue Auswahl der Themen ist im Interviewleitfaden zu erkennen.

5.2.1 Leitfaden

Dieser Abschnitt stellt den Leitfaden vor. Dieser dient zur Unterstützung während der Interviews. Die Fragen werden dem Interviewpartner nicht direkt gestellt, sondern sollen im Rahmen des Gespräches beantwortet werden. Sie dienen hauptsächlich dazu eine Struktur zu integrieren und einen flüssigen Ablauf des Gesprächs zu ermöglichen. Der Leitfaden muss

nicht zwingend in der vorgegebenen Reihenfolge abgearbeitet werden. Die Reihenfolge ergibt sich aus dem aktivem Gespräch. Damit die Antworten spontan kommen und nicht durch vorherige Recherchen entstehen, wird der Leitfaden den Interviewpartnern nicht ausgehändigt. Im Folgendem werden die Punkte des Leitfadens kurz erläutert. Der konkrete Leitfaden ist in Anhang A zu finden.

Der Leitfaden wurde in sieben Bereiche unterteilt:

1. **Unternehmen und Organisation** Durch diesen Abschnitt soll der Projektkontext, also das was und wie herausgefunden werden. Dazu gehören die beteiligten Personen, inklusive deren Rollen und eine Beschreibung des Vorgehens.

2. **Debatte** Dieses Thema hat das Ziel herauszufinden in welchen Momenten und Situationen über die Architektur geredet wird. Wenn eine Diskussion stattfindet soll herausgefunden werden was als Diskussionsgrundlage dient.

3. **Kommunikation und Modellierung** Eine Debatte kann sehr unterschiedlich stattfinden. Hier soll herausgefunden werden, wie die eigentliche Diskussion stattfindet. Dazu gehört z.B. eine Modellierung, wie auch die beteiligten Personen. Ein weiterer untersuchter Punkt ist, wie und ob ein Informations- und Wissensaustausch über verschiedene Teams hinweg abläuft.

4. **Architekt** Aus traditionellen Vorgehen ist die Rolle des Architekten bekannt. Diese Rolle ist in der Literatur in angepasster Weise ebenfalls häufig zu finden. Wenn diese Rolle in dem besuchtem Team existiert soll untersucht werden wie diese in das Team integriert ist und welche Aufgaben sie hat. In diesem Abschnitt soll ermittelt werden, ob es weitere Rollen gibt die in anderen Unternehmen und Teams nicht vorhanden sind. Diese können z.B. aus den früher eingesetzten Entwicklungsvorgehen stammen.

5. **Review der Architektur** Die Überprüfung der Architektur kann vor vielen potentiellen Fehlern schützen. Dieser Abschnitt soll untersuchen, wie z.B. mit alten Entscheidungen umgegangen wird, wenn sich die Anforderungen ändern. Es soll herausgefunden werden, ob diese nochmal überprüft werden. Der mögliche Einsatz von halb-, bzw. automatischen Analysewerkzeugen soll untersucht werden. Die Analysen sollen das Ziel haben dauerhaft eine hohe Codequalität sicherzustellen und mögliche Probleme vor dem Eintreten zu finden.

6. **Dokumentation** Die Dokumentation ist laut der Literatur in vielen Projekten problematisch. Aus diesem Grund wird erfragt, ob das Team hierfür eine Lösung hat und welche

Informationen festgehalten werden. Außerdem soll in diesem Abschnitt geklärt werden, wie sich neue Teammitglieder in die Anwendung einarbeiten können.

7. **Beurteilung des Vorgehens** Dieser Abschnitt beschäftigt sich mit der Weiterentwicklung und Beurteilung des Vorgehens. Hier soll der Interviewpartner über Aktivitäten zur Optimierung des aktuellen Vorgehens berichten. Außerdem soll er ihm bekannte Defizite erläutern. Anschließend soll er beschreiben wie für ihn ein optimales Vorgehen aussehen kann. Zu diesem Zweck kann er beschreiben, welche Änderungen er an der aktuellen Situation vornehmen würde. Bei diesem Punkt wird viel Wert auf die persönliche Meinung des Interviewpartners gelegt.

5.3 Einzelauswertung

Dieser Abschnitt behandelt die einzelnen Interviews. Die genauen Inhalte der Interviews sind im Anhang zu finden. Hier wird eine übersichtliche Einzelauswertung durchgeführt. Alle Informationen die nicht in direktem Bezug zum Thema dieser Arbeit stehen werden ausgelassen. Ähnliche Aussagen werden verkürzt und zusammengefasst. Eine genaue Beschreibung von unternehmensspezifischen Rollen und Hintergrundwissen ist den Interviewberichten zu entnehmen.

5.3.1 Interview A

Das Team aus Interview A arbeitet nach Scrum. Zur Diskussion werden hauptsächlich die in Scrum vorgesehenen Meetings eingesetzt. Hierbei ist besonders das Sprint Planning wichtig. Die User Stories werden hier in Tasks umgewandelt. Eine Besonderheit ist das optional stattfindende *Tec-Grooming* Meeting. Hier werden im allgemeinen Grooming aufgekommene technische Fragen diskutiert. Dies wird meistens spontan nach dem Grooming, nur mit den Entwicklern, abgehalten. Eine weitere Besonderheit im Prozess sind die zweistufigen Code Reviews. Die abgearbeiteten Stories werden im ersten Schritt durch ein Teammitglied und im zweiten Schritt durch eine Person aus einem anderem Team begutachtet. Dies ist in diesem Unternehmen hilfreich, weil alle Teams auf einer gemeinsamen Code-Basis arbeiten. Eine weitere Besonderheit zum normalen Scrum gibt es beim Daily Meeting. Dieses findet an zwei Tagen der Woche mit allen iOS Entwicklern statt. Durch die große Anzahl an Personen ist es möglich, dass dies sehr ineffizient wird.
Um Feedback zu erhalten wurden unterschiedliche Vorgänge eingeführt. Hierzu gibt es *Design-*

Approaches und *Proposals*. Ein Design-Aproach wird vom Entwickler schriftlich erstellt, wenn er den Wunsch nach Feedback hat. Teamübergreifend können sich alle Entwickler hierzu äußern. Proposals werden erstellt, wenn größere Änderungen notwendig sind. Diese können häufig mehrere Teams betreffen. Nach der Vorstellung des Proposals haben alle eine Woche Zeit sich hierzu zu äußern. Sofern keine Gegenstimmen kommen wird der Vorschlag umgesetzt. Wenn Features nur durch einen Workaround umgesetzt werden können wird für diese ein neues *Technical-Debt Ticket* angelegt. Dadurch kann der Workaround zu einem anderem Zeitpunkt aufgelöst werden und gerät nicht in Vergessenheit.
Ein allgemeiner teamübergreifender Wissensaustausch wird durch Vorträge und Besprechungen im wöchentlich stattfindendem *Community-Meeting* gefördert. Hier werden z.B. die Approaches und Proposals vorgestellt.
Eine Person, die die Entwickler im Rahmen einer Architektenrolle leitet, gib es in diesem Team nicht. Es kann aber vorkommen, dass Lösungswege durch andere Teams vorgegeben, bzw. durch deren Vorarbeiten eingeschränkt werden.
Die Dokumention besteht hauptsächlich aus den schriftlichen Proposals und Approaches. Eine weitere Dokumentation von Architekturenscheidungen ist nicht vorhanden.

Während des Interviews wurde deutlich, dass der Interviewpartner die Architekturentwicklung genauer strukturieren möchte. Allerdings wurde es als große Herausforderung gesehen Prozessänderungen vorzunehmen. Das Management muss hierbei zustimmen. Ebenfalls müssen alle Teammitglieder von dem Vorgehen überzeugt werden. Dies kann besonders zu Anfang zu vielen Meetings führen. Die Entwicklung verlangsamt sich im ersten Moment. Durch die Einführung von verschiedenen Möglichkeiten um Feedback zu erhalten und Änderungen vorzunehmen wird deutlich, dass ihnen eine hohe Codequalität für eine langlebige Anwendung wichtig ist. Die Approaches und Proposals existieren noch nicht besonders lange. Mit der Zeit muss sich zeigen, wie diese angenommen werden. Vorstellbar ist, dass die verschiedenen Ansätze zu einer Strategie zusammengefasst werden, da diese sich in vielen Punkten ähneln.

5.3.2 Interview B

Das Unternehmen B hat vor circa neun Monaten mit dem Wechsel des CTOs auf die agile Entwicklung umgestellt. Die aktuelle Architektur ist relativ ungeplant entwickelt worden. Der Interviewpartner sagte, dass es keine Schichten oder ähnliches gibt und die Architektur so weiterentwickelt wurde wie es in dem Moment am besten gepasst hat.
Die interviewte Person hatte keine genauen Kenntnisse von den Abläufen innerhalb der Teams. Für einen teamübergreifenden Austausch steht ein Termin zur Verfügung. In diesem

wird die gesamte Planung der nächsten Woche, inklusive der Ankündigung von personellen Veränderungen besprochen. Es ist fraglich wie intensiv hier über Architekturthemen geredet wird. Vielversprechender wirkt der Ansatz des geplanten *Offsite-Meetings*. Im Unterschied zu dem anderen Terminen nehmen hier auch Personen aus dem Produktmanagement teil. Hier sollen die zukünftigen Ziele und der aktuelle Stand des Unternehmens besprochen werden. Dabei kann es sich auch um die Softwarearchitektur handeln. Derzeit ist dieses Meeting als einmaliger Termin angesetzt. Der Interviewpartner würde sich aber wünschen, dass dies regelmäßig stattfindet. Dadurch könnten regelmäßig notwendige Änderungen besprochen werden. Außerdem verbessert dies das allgemeine Verständnis für die Architektur. Einen weitere fester Termin, bei dem sich die Produktentwicklung mit der Softwareentwicklung abstimmt wurde nicht festgestellt.

Abstimmungen zwischen einzelnen Entwicklerteams geschehen individuell. Jedes Team handhabt dies unterschiedlich. Obwohl Werkzeuge zur statischen Codeanalyse eingesetzt werden, existiert keine Mindestbewertung für neuen Quellcode. Dadurch können sich schnell kleinere technische Schulden ansammeln. Dies führt wiederum zu einer Architekturerosion.
Der alte CTO hat manuell ein Schaubild mit den Komponenten und Prozessen gepflegt. Durch den Führungswechsel wurde dies seit circa einem Jahr nicht aktualisiert.

Der Interviewpartner ist skeptisch gegenüber Tests eingestellt. Er sagt, dass Tests wichtig sind. Schränkt aber ein, dass nur für besonders wichtige Bereiche Tests entwickelt werden sollten. Dies richte sich hauptsächlich nach den entstehenden Kosten bei einem Ausfall.
Als großes Problem konnte festgestellt werden, dass das Produktmanagement zu wenig über die Architektur weiß. Dies führt zu vielen Nachfragen und im schlimmsten Fall zur Entwicklung von sich gegenseitig ausschließenden Funktionalitäten.
Der Gesprächspartner hatte zusätzlich das Gefühl, dass seit der Umstellung viele Personen die Verantwortungen auf andere schieben. Seitdem würde eher gegeneinander, statt miteinander gearbeitet.

Die Anwendung des Unternehmen B wirkt sehr historisch gewachsen. Es scheint keine direkt erkennbaren Strukturen innerhalb der Anwendung zu geben. Ein teamübergreifender Austausch fand bisher nur sehr eingeschränkt in einem Meeting mit vielen weiteren Themen statt. Zu den Abläufen innerhalb der Teams konnte nicht viel berichtet werden. Das angesetzte Offsite-Meeting zeigt aber, dass die Problematik bekannt ist und vielleicht bei einer regelmäßiger Durchführung verringert werden kann. Zum Problem könnte hierbei die Anzahl der Teilnehmer werden. Wenn sowohl alle Personen der Produktentwicklung und alle Personen der Softwareentwicklung teilnehmen wird dies ein sehr großes Meeting. Außerdem kann es

zu Problemen führen, dass die Produktentwicklung mehr über die Architektur wissen soll. Die Produktentwicklung ist für die fachliche Seite zuständig und formuliert in erster Linie welche Funktionalitäten vorhanden sein sollen. In den meisten Fällen kennen diese sich mit der technischen Seite nicht aus. Technische Entscheidungen sollten nicht von der fachlichen Seite getroffen werden.

5.3.3 Interview C

Die Anwendung von Unternehmen C ist durch *Self-Contained Systems* so realisiert, dass die Teams möglichst unabhängig voneinander arbeiten können. Das Vorgehen innerhalb der Teams wird mit einer individuell angepassten Version von Kanban organisiert. Einmal die Woche oder wenn das Board leer läuft werden neuen Stories vorgestellt. Erst direkt vor der Realisierung teilen zwei Personen die Story in Tasks auf. Dadurch wird die Umsetzung besprochen. Nach der Umsetzung findet ein Code Review durch eine weitere Person statt. Dies bewirkt, dass sich zu dem Zeitpunkt mindestens drei Leute mit der Story beschäftigt haben.

In jedem Team ist ein Architekt integriert. Dieser entwickelt selbst mit. Seine Rolle ist ähnlich wie die des Architekten im AMDD definiert (siehe Kapitel 4.9.1). Obwohl er das Recht hätte Vorgaben zu machen, setzt er dies nicht ein.

Eine Austauschmöglichkeit für Architekten ist durch das *Jour Fixe* regelmäßig möglich. Dies stärkt die Wissensverbreitung unter den Architekten und ermöglicht es einen Überblick über die Gesamtarchitektur zu erhalten. Außerdem schafft dies eine Austauschmöglichkeit zu Problemen.
Das *Code-Camp* als gesonderter Zeitraum eignet sich gut um über im Code aufgefallene Dinge zu reden. Hier können gezielt Unstimmigkeiten in der technischen Realisierung angesprochen werden. Gleichzeitig kann sich dadurch ein ähnliches Verständnis für Qualität bei den Teilnehmer entwickeln. Sehr positiv ist, dass während der Entwicklung durchgehend 20% der Zeit für Ausbesserungen und Refaktorisierungen eingeplant ist. In dieser Zeit können z.B. die im Code-Camp aufgefallenen Dinge gelöst werden.
Die Entwicklung der Architektur wird nicht dokumentiert. Es existieren nur Protokolle zu den Jour Fixe Treffen.
Um die Architektur zu überprüfen wurden Versuche, z.B. mit Sonargraph und Moose, durchgeführt. Sonargraph wurde wegen zu starren Regeln verworfen. Moose wurde zum Zeitpunkt des Interviews evaluiert. Dies zeigt, dass das Unternehmen bereit ist zu experimentieren um die Qualität zu steigern.
Ein Nachteil, der selbst erkannt wurde, ist die fehlende Dokumentation. Das Team würde dies

gerne bei einem neuen Projekt anders machen. Bei der Dokumentation ist allerdings immer schwer festzustellen, was lohnenswert ist und wie dies aktuell gehalten wird.

Das Unternehmen C hat einen sehr strukturieren Vorgang mit mehreren Teams. Dieser Vorgang ist für Außenstehende teilweise schwer auf Anhieb zu verstehen, weil viele eigene Bezeichnungen für Rollen und Meetings existieren. Grundlegend werden vorhandene Meetings für die Architekturdebatten verwendet. Probleme können z.B. im Daily Meeting angesprochen werden. Hier wird vor allem die spontane, direkte Kommunikation eingesetzt. Der kontinuierlich eingeplante Zeitraum für Verbesserungen eignet sich sehr gut, um die technische Schulden zu managen.

5.3.4 Interview D

Das Unternehmen D befindet sich noch in der aktiven Umstellungsphase. In einigen Teams wird noch nach einer traditionellen Methode gearbeitet. Dort wo bereits agil Entwickelt wird, wird stark auf einen korrekten Scrum Ablauf mit wenigen Anpassungen geachtet.
Das Vorgehen in den agilen Teams ist sehr strukturiert. Diese arbeiten vollständig nach Scrum, ohne besondere Anpassungen oder Ergänzungen. Es ist für die einzelnen Teams möglich Anpassungen und zusätzliche Diskussionsmöglichkeiten einzuführen. Für Architekturdebatten und zum Ansprechen von Problemen werden die vorgesehenen Meetings verwendet. Wenn beim Sprint Planning größere Differenzen beim Schätzen auftreten, wird konkreter über die Umsetzung gesprochen. Beim Auftreten von Besonderheiten während der Entwicklung, werden diese in einem zusätzlichem Termin vor dem Review vorgestellt. Dies beinhaltet die damit verbundenen Änderungen an der Architektur. Wenn unvorhergesehen große Änderungen notwendig werden, müssen diese dem PO vor der Umsetzung vorgelegt werden. Bei einer Gefährdung des Sprint Ziels wird dieses entsprechend angepasst. Größere Änderungen werden in Tickets gefasst, welche anschließend mit eingeplant werden können. Die größten Diskussionszeitpunkte sind vor der Entwicklung im Planning und anschließend beim Code Review des Pull Requests.

Ein Austausch zwischen den Teams findet individuell, z.B. zu Programmierschnittstellen, statt. In Zukunft soll dies aber genauer geregelt werden. Weitere Austauschmöglichkeiten zwischen den Teams sind nicht vorhanden. Für die Zukunft sind auch hierzu Gremien für die unterschiedlichen Rollen aller Teams geplant.

Die Architekten sind direkt in das Team integrieren und entwickeln selbst mit. Da die Teams zum Teil sehr groß sind (bis 20 Personen) kann es vorkommen, dass ein Team mehrere Ar-

chitekten hat. Bei den großen Teams kann es sich allerdings auch um traditionell agierende Teams handeln. Durch seine Rolle dient er als Ansprechpartner. Außerdem ist er ein Mentor und besucht die firmenweiten Architektur-Diskussionen.

Der Sonargraph Architekt (siehe Kapitel 3.5) wurde einmalig testweise eingesetzt. Dies ist nicht besonders gut angekommen. Die Motivation eine Erosion zu beheben war nicht besonders groß. Kleine Änderungen wurden übernommen, größere Aufwände aber nicht als relevant genug erachtet.

Besonders hervorzuheben ist die ausführliche Dokumentation. Es werden Architektur-Blueprints und große Entscheidungen, inklusive der Rahmenbedingungen, Begründungen und Alternativen festgehalten. Dieses ausführliche Vorgehen ist ungewöhnlich und kann eventuell auf die langjährige Erfahrung mit den traditionellen Methoden zurückgeführt werden. Im Unterschied zur aktuellen Situation wurde dort die Dokumentationen nicht durch die Entwickler gepflegt. Zu diesem Zweck wurden spezielle Personen eingestellt.

Die größte Schwierigkeit bei der Entwicklung ist die Kommunikation zwischen den Teams. Die unterschiedlichen Planungsrhythmen der Teams erschweren dies enorm. Abhängigkeiten zu anderen Teams sind dadurch schwer innerhalb eines Sprints aufzulösen. Eine große vorausschauende Planung ist ebenfalls nicht möglich, da dies der agilen Entwicklung widerspricht.

5.3.5 Interview E

Der Interviewpartner von Unternehmen E konnte von Erfahrungen aus unterschiedlichen Projekten in verschiedenen Unternehmen berichten. Zur Auswertung wird das zweite ausführlicher besprochene Projekt verwendet. Dieses Projekt war beim Interview bereits abgeschlossen. In dem Projekt wurde Scrum mit der Besonderheit, dass der Projektleiter zu jeder Zeit die Entscheidungsgewalt haben konnte, eingesetzt. Dieses Recht hat er allerdings nicht angewendet. Dies ähnelt dem Architekten im Kapitel 4.9.1. Dieser hatte ebenfalls die letzte Entscheidungsgewalt. Weitere Gemeinsamkeiten sind zwischen ihnen nicht vorhanden.

Eine Architektur-Diskussion fand hauptsächlich im zweiten Teil des Sprint Plannings, während dem Aufteilen in Tasks, statt. Möglichkeiten zur Realisierung werden direkt besprochen. Es wird besonders drauf geachtet, dass keine individuellen Lösungen geschaffen werden. Diese sollen wenn möglich in anderen Stories wiederverwendet werden können. Probleme während der Entwicklung werden meistens direkt in Partnerarbeit gelöst oder im Daily Meeting angesprochen. Zur Lösung wurde außerdem manchmal im Anschluss an das Daily Meeting ein zusätzliches Treffen organisiert. Die zusätzliche Diskussion fand nur mit den interessierten

Personen statt. Dadurch sollten die Meetings effizienter gehalten werden. Eine allgemeine Austauschmöglichkeit für Vorträge und allgemeine Themen war einmal pro Woche vorgesehen. Dieser Termin ist allerdings häufig ausgefallen, da keine Themen vorbereitet wurden.

Eine Architektenrolle gab es in diesem Team nicht. Für einzelne Teilbereiche der Anwendung haben sich mit der Zeit allerdings Spezialisten gebildet. Die Entwicklung sollte in erster Linie durch das Team selbst stattfinden. Der Interviewpartner sagte aber, dass es zu Beginn eines Projekts hilfreich sein kann einen Architekten zu integrieren. Dieser sollte helfen die Grundsteine zu legen und das Team anleiten die Weiterentwicklung selbstständig durchzuführen. In keinem Fall sollte dieser Entscheidungen ohne das Team fällen.

Auffällig war, dass keinerlei Dokumentation geführt wurde. Eine Überwachung der Codequalität fand mithilfe von Sonarqube statt. Zu einem Zeitpunkt der Entwicklung, als fachlich nicht viel zu tun war, wurde 30% der Zeit zur Verfügung gestellt, um Sonarqube Tickets zu bearbeiten.

5.3.6 Interview F

Die Entwicklung in diesem Unternehmen wurde vor einem Jahr und drei Monaten mit Kanban begonnen. Zur Umsetzung von Stories setzen sich meist zwei Personen zusammen und überlegen sich eine Aufteilung in Tasks und eine Lösungsstrategie. Hier findet die hauptsächliche Diskussion über die Architektur statt. Für jede Story übernimmt eine Person die Verantwortung und dient deshalb als Ansprechpartner zum aktuellen Stand. Bei einer komplexen Story schreibt er diese nochmals aus seiner Sicht auf. Dies bewirkt automatisch eine Kontrolle, ob die Aufgabenstellung richtig verstanden wurde. Fraglich bleibt allerdings, wie unterschiedlich beide Versionen der Story sind oder ob der Großteil übernommen wurde.

Der Ablauf der Entwicklung und das Ergebnis wird zwei Wochen nachdem die Story im Produktivsystem läuft besprochen. Da hier das Projekt Management, das Business Development, wie auch mindestens ein beteiligter Entwickler teilnimmt wird versucht die Zusammenarbeit zu verbessern. Hier wird hervorgehoben, was gut und was nicht gut gelaufen ist. Dabei kann es sich z.B. um fehlende fachliche Informationen handeln. Über die technische Umsetzung wird allerdings seltener geredet.

Probleme werden beim nächsten Daily Meeting angesprochen oder versucht direkt durch Pair Programming zu lösen. Wenn erforderlich machen die Personen, für die dieses Thema relevant ist, einen individuellen Termin für eine weitere Diskussion aus. Eine Folge davon können technische Tickets für Umbaumaßnahmen sein. Wenn ein Mitarbeiter Feedback zu

einem Thema haben möchte, kann er dieses einmal pro Woche beim *Table-Talk* vorstellen. Bei diesem Treffen können auch weitere Vorträge zu Themen gehalten werden.

Zur Dokumentation wird eine *Wiki-System* eingesetzt. Besonders ist in diesem Unternehmen, dass ein aktuelles Diagramm aller Microservices und deren interner Aufbau manuell gepflegt wird. Dies wird meist als Grundlage für Diskussionen eingesetzt. Durch das Wachsen der Anwendung wird das Diagramm immer komplexer und größer. Es wird interessant, ob es mit der Zeit weiterhin pflegbar bleibt. Aktuell ist dies möglich, weil das Unternehmen und die Anwendung noch relativ jung ist.

Ein Monitoring auf Codebasis findet mit einem ähnlichem Tool wie Sonarqube statt. Neue Möglichkeiten werden regelmäßig von Entwicklern ausprobiert und bei Erfolg den anderen präsentiert.

Ein identifiziertes Problem ist die zunehmende Teamgröße. Irgendwann müssen die Entwickler in mehrere Teams aufgeteilt werden, damit die Meetings effizient bleiben. Außerdem würden sie gerne mehr Zeit haben, um sich in Themen einzuarbeiten die zu dem Zeitpunkt nicht direkt firmenrelevant sind.

Dieses Unternehmen nutzt hauptsächliche vorhandene Umstände und Termine, um über die Architektur zu reden. Die meisten Probleme werden spontan und individuell durch kleine Gruppen beschlossen und gelöst.

5.3.7 Interview G

Bei Unternehmen G wird zwischen der Mikro- und der Makro-Architektur unterschieden. Die Mikro-Architektur beschreibt den Aufbau einer Komponenten. Die Makro-Architektur beschreibt den Aufbau und die Kommunikation der Komponenten.
Durch den Beschluss die Anwendung neu zu entwickeln konnten technisch viele Eigenschaften beachtet werden die in der alten Version nicht beachtet wurden. Dies beinhaltet eine Aufteilung der Anwendung mit möglichst wenig Schnittstellen untereinander.

Die Teams selbst können frei entscheiden wie sie über die Architektur diskutieren wollen. Eine erste Lösungsmöglichkeit wird direkt vor der Umsetzung besprochen. Sobald sich jemand entscheidet eine Story umzusetzen wird direkt davor über diese geredet. Dies kann über fachliche Unklarheiten oder die technische Umsetzung sein. Ein wichtiger Punkt hierbei ist, dass die gesamte Entwicklung per Pair Programming durchgeführt wird. Dadurch wird während der Entwicklung durchgehend über die Lösung diskutiert. Viele logische Fehler können bereits beim laut Aussprechen des Vorgehens gefunden werden. Zusätzlich wissen

immer mindestens zwei Personen über die Lösung Bescheid. Neben dem Pair Programming wird nach dem *Test-First* Prinzip gearbeitet. Dieses Unternehmen arbeitet allerdings ohne Code Reviews.

Für die teamübergreifende Diskussion wird viel unternommen. Es findet z.B. alle zwei Wochen eine Entwickler Convention mit Vorträgen statt. Zusätzlich werden unregelmäßig weitere Events, wie eine interne Konferenz zu einem speziellem Thema, organisiert. Diese Events fördern den Austausch der Entwickler untereinander. So kommen z.B. Personen des Unternehmens in Kontakt, die unter anderen Umständen nicht miteinander geredet hätten. Absprachen, z.B. zu benötigten Schnittstellen, werden individuell vorgenommen.

In diesem Unternehmen ist pro Team ein Architekt integriert. Er ist explizit für die nichtfunktionalen Anforderungen zuständig. Wie in den meisten anderen Unternehmen hat dieser ebenfalls die Möglichkeit Widerspruch gegen Entscheidungen einzulegen. Dies wird aber praktisch nicht getan. Ein Austausch der Architekten findet wöchentlich statt, um über die überliegende Architektur zu reden.

Tools zur Analyse werden nur eingesetzt, wenn Fehler vorhanden sind. Nachdem diese gelöst sind, werden sie wieder abgeschafft. Dieses Vorgehen hilft nur Fehler zu finden nachdem diese Aufgetreten sind. Fehler, die nicht durch Tests entdeckt wurden, können so allerdings unbemerkt im Produktivsystem landen.

Besonders interessant ist der Versuch die Systemlandschaft automatisch durch einzelne Statusberichte zu visualisieren. Dies würde einen Blick auf die aktuelle Anwendung ermöglichen der nicht mühselig manuell gepflegt werden muss. Diese Ansicht kann dazu verwendet werden, um allgemein herauszufinden, welche Systeme existieren und wovon diese abhängig sind.

Der Interviewpartner dieses Unternehmens ist der Meinung, dass Kanban besser als Scrum für die Entwicklung ist. Dies begründet er dadurch, dass viele Unternehmen Scrum als festes Regelwerk mit genauen Vorgaben interpretieren. Dadurch wird die Entwicklung wieder sehr starr und unflexibel. Scrum kann seiner Meinung nach aber gut helfen, um mit der agilen Entwicklung zu beginnen.

Dieses Unternehmen unternimmt viel um die agile Entwicklung und den Austausch der Personen untereinander zu fördern. Die Entscheidung, zu Beginn der Entwicklung, die Anwendung so zu unterteilen, dass die Teams möglichst unabhängig arbeiten können, war sehr wichtig. Dadurch wurde die Komplexität einer allgemeinen großen Architektur aufgeteilt. Zum einen gibt es die große überliegende Architektur, die die Verbindung der einzelnen Systeme dar-

stellt und die vielen kleineren Architekturen der einzelnen Systeme, für die die Teams selbst verantwortlich sind.

5.3.8 Interview H

Unternehmen H arbeitet in dem besuchten Team nach Scrum. Besonders ist, dass die vollständige Entwicklung per Pair Programming mit wechselnden Partnern durchgeführt wird. Dadurch findet eine direkte Wissensverbreitung statt. Eine weitere Besonderheit bei deren Vorgehen ist, dass zwei Standups an einem Tag gehalten werden. Dies kann unter Umständen viel Zeit beanspruchen, weil Einzelpersonen zweimal die gleichen Dinge erzählen. Hilfreich ist dies nur wenn die einzelnen Aufgaben sehr klein geschnitten sind.

Die erste Diskussion über eine Story beginnt im Sprint Planning Meeting. Das zweite mal wird ausführlicher direkt vor der Entwicklung darüber geredet. Hierzu setzten sich bei einer komplexen Story mehrere Teammitglieder zusammen. Bei nicht besonders komplexen Stories wird dies nur durch das umsetzende Team besprochen. Die Ergebnisse der Umsetzung werden nochmals von einer dritten Person gereviewt. Diese Arbeitsweise bewirkt, dass bereits während der Entwicklung viele Fehler verhindert werden können. Außerdem ist das gesamte Wissen über die Umsetzung einer Story drei unterschiedlichen Personen bekannt.

Interessant ist ebenfalls das Vorgehen des Todo-Boards. Dieses ermöglicht es schnell und unkompliziert Stellen zu notieren die ihnen aufgefallen sind. Dadurch werden diese nicht so schnell vergessen und können nach und nach abgearbeitet werden.

Das Team besitzt zwei Personen die eine Art von Architektenrolle übernehmen. Diese haben nicht offiziell die Rolle des Architekten, übernehmen aber viele dessen Aufgaben und werden im Team auch als Architekt bezeichnet. Dieses Konstrukt wirkt verwirrend, weil sie inoffiziell Architekten sind, offiziell aber nicht. Dies kann daher kommen, dass in den meisten anderen Projekten kein Architekt vorhanden ist. Diese Teams bestehen meist nur aus sehr wenigen Personen, wodurch eine weitere Rolle nicht sinnvoll ist. Die Rolle eines Architekten wird häufig durch die Projektleitung eingenommen. Dies kann allerdings einen Rollenkonflikt bewirken.
Die Architekten im besuchten Team sind dafür zuständig das Gesamtsystem im Auge zu behalten. Hierzu gehört auch, dass sie auf die nichtfunktionalen Anforderungen achten. Da die Architektur der Anwendung noch sehr lebendig ist existiert ein Gesamtbild nur in den Köpfen der Architekten. Eine Übersicht in die alle reinschauen können ist nicht vorhanden.
Da die Architekten selbst primär Entwickler sind arbeiten diese auch im Paar zusammen. Dies ist wichtig, damit sie nicht die alleinigen Wissensträger zu speziellen Codeabschnitten werden.

Eine Dokumentation wird in diesem Team nicht durchgeführt. Nur wichtige Entscheidungen werden durch die Architekten per E-Mail verteilt.

Eine Überprüfung der Architektur findet durch regelmäßige Analysen mit Sonargraph statt. Die Interviewpartnerin leitet diese Analysen regelmäßig und führt diese Art von Analysen ebenfalls in anderen Projekten anderer Firmen durch. Das genaue Besprechen der Anwendung fördert nicht nur das Verständnis des Teams für den Aufbau des aktuellen Systems, sondern fördert ebenfalls den Zusammenhalt im Team. Dies kann eventuell dazu führen, dass viele verschiedene Stile im Quellcode erkannt werden können. Diskussionen werden dadurch im Team angeregt. Viele sehen den Aufbau der Anwendung zum ersten mal. Bei regelmäßiger Durchführung wird Sichtbar, ob eine stetige Verbesserung stattfindet.

Für den Austausch im Team findet alle zwei Wochen eine Architekturbesprechung statt. Hier werden die gesammelten Todo-Notizen und weitere Änderungen an der Architektur besprochen und vorgestellt.
Da in den anderen Teams keine Architekten vorhanden sind, findet kein Unternehmensweiter Austausch für Architekten statt. Stattdessen gibt es einen regelmäßigen Austausch durch Vorträge und Workshops. Zu diesem Zweck werden Personen, die sich mit ähnlichen Themen beschäftigt haben, konkret angesprochen. Diese sollen sich zusammensetzten und gemeinsam einen Vortrag vorbereiten. Dieses Vorgehen bewirkt, dass die Personen eine Selbstreflexion zu ihrer Arbeit durchführen und sich bewusst werden, was gut und was nicht so gut war. Wenn gleichzeitig mehrere Personen zum Vortrag beitragen, können unterschiedliche Vorgehensweisen miteinander verglichen werden.

Dem Unternehmen ist eine gute Architektur wichtig. Dies fällt ebenfalls daran auf, dass jeder Mitarbeiter eine *iSAQB* Schulung erhält. Dies ist eine explizite Architekturschulung.

5.4 Gesamtauswertung

Dieser Abschnitt stellt eine Verbindung der Interviews untereinander her. Zu diesem Zweck wird untersucht, welche Aussagen in den einzelnen Interviews zu den Punkten des Kodierleitfadens aus Anhang J getroffen wurden. Die einzelnen Punkte sind meist nicht vollständig überschneidungsfrei. Alle Möglichkeiten um über Probleme zu reden stellen z.B. auch eine Diskussionsmöglichkeit dar. Wenn bei den verwendeten Zitaten Wörter hinzugefügt oder entfernt wurden, ist dies mit eckigen Klammern gekennzeichnet. Für jeden Abschnitt wird eine Tabelle erstellt, die einen direkten Vergleich der Unternehmen ermöglichen soll.

Unternehmen	*Scrum*	*Kanban*	*Projektgeschäft*	*Produktgeschäft*	*interne Entwicklung*	*Teamgröße*	*Teamanzahl*	*Interviewdauer*
A	✓			✓		9	7	63 Minuten
B	✓	✓		✓		5	5	56 Minuten
C		✓			✓	6 - 12	5	59 Minuten
D	✓				✓	3 - 20	>20	44 Minuten
E	✓		✓			6 - 7	2	77 Minuten
F		✓		✓		12	1	51 Minuten
G		✓		✓		unbekannt	unbekannt	74 Minuten
H	✓		✓			11	1	51 Minuten
Ø	5/8	4/8	2/8	4/8	2/8	5 - 20	1 - 20+	60 Minuten

Tabelle 5.1: Kontextinformationen

5.4.1 Projekt und Interview Kontext

Der Kontext der besprochenen Projekte wird in diesem Abschnitt vorgestellt. Zu diesem Zweck sind in Tabelle 5.1 allgemeine Informationen zu den Projekten und den Interviews dargestellt. Ziel dieser Übersicht ist es schnell nachschauen zu können, welcher Kontext in den einzelnen Teams vorhanden war ohne die Einzelberichte lesen zu müssen.

Die Teamzusammensetzungen und Vorgehen können sich innerhalb eines Unternehmens stark unterscheiden. Gerade zu Teams im Projektgeschäft muss erwähnt werden, dass deren Zusammenstellung durch den Abschluss eines Projekts meistens zeitlich begrenzt ist. Die Projektorganisation kann in einem Folgeprojekt vollständig unterschiedlich sein. Die hier und im Folgenden dargestellten Informationen beziehen sich deshalb immer auf ein konkretes Projekt. Außerdem unterscheidet sich in manchen Projekten das Vorgehen innerhalb der einzelnen Teams. Viele Teams haben individuelle Änderungen eingeführt, die den Interviewpartnern nicht immer vollständig bekannt waren. Bei der Auswertung wurde für die einzelnen Teams Scrum, bzw. Kanban ausgewählt, wenn deren Vorgehen auf einem der beiden basierte. Teilweise haben die Teams Anpassungen vorgenommen und deshalb eine andere Bezeichnung verwendet. Das Vorgehen entsprach im Allgemeinen aber einem der Beiden.

In der Tabelle ist auffallend, dass in Unternehmen B sowohl mit Scrum wie auch Kanban gearbeitet wurde. Da in diesem Interview keine Informationen eines konkreten Teams eingeflossen sind, wurde sich bei der Auswertung nicht zwischen einem Vorgehen entschieden. Allgemein haben circa je die Hälfte der Teams Scrum (5) oder Kanban eingesetzt (4). Andere agile Vorgehen wurden nicht verwendet. Zum Teil wurden während der Entwicklung Praktiken aus XP eingesetzt. Dies wird bei den Aktivitäten während der Entwicklung zum Management der Erosion (Kapitel 5.4.3) genauer beschrieben.
Bei der Auswahl der Unternehmen wurde darauf geachtet, dass diese in verschiedenen Geschäftsbereichen tätig sind. Die Hälfte der Unternehmen ist im Produktgeschäft (4) tätig, ein Viertel im Projektgeschäft (2) und ein Viertel führt eine reine internen Entwicklung (2) durch. Die Anzahl der Teams und die Anzahl deren Mitglieder unterscheidet sich stark. Die Größe beschreibt alle Teammitglieder. Es wurde sich für keine weitere Aufteilung entschieden, da in vielen Fällen alle Personen bei Entscheidungen miteinbezogen werden. Die Anzahl der Teams und dessen Größe wurde bei Unternehmen G nicht erfasst. Das Unternehmen ist aber unter den größeren Unternehmen zwischen A und D einzuordnen. Zu Unternehmen D muss gesagt werden, dass in der großen Teamanzahl Teams inbegriffen sind, die noch nicht agil Arbeiten. Dies kommt daher, dass sich das Unternehmen noch in einer nicht abgeschlossenen Umbauphase befindet. Das Vorgehen wurde noch nicht in jedem Team auf eine agile Entwicklung umgestellt.
Die Interviews haben zwischen 44 und 77 Minuten gedauert. Im Schnitt gingen die Interviews eine Stunde.

5.4.2 Diskussionsmöglichkeiten

Dieser Abschnitt vergleicht die unterschiedlichen Möglichkeiten für Diskussionen während der Entwicklung. Hierzu zählen die geplanten und spontanen Diskussionen im Team, wie auch der teamübergreifende Austausch.
Die Tabelle 5.2 gibt an, welche Diskussionen im Prozess vorhanden sind. Wenn ein Punkt nicht markiert oder nicht vorhanden ist, heißt dies nicht zwingend, dass dieser nicht durchgeführt wird. Es ist ebenfalls möglich, dass zu diesem Zeitpunkt nicht über Architekturthemen geredet wird.

Viele der regelmäßigen Meetings aus dem Scrum Manifesto werden bereits für eine Diskussion eingesetzt. Die Kanban Teams (C, F, G) verwenden diese ebenfalls zum großen Teil. Die Meetings und deren regelmäßigen Zyklus haben sie übernommen.
Der erste Zeitpunkt, an dem über die Architektur geredet wird, ist meist das Sprint Planning. In

Unternehmen	*Sprint Planning*	*Daily Meeting*	*spontanes Meeting*	*Teamintern*	*Teamübergreifend*
A	✓	✓	✓	- Tec-Grooming	- Community Meeting
B	✓	✓	✓		- individuelle Absprachen zwischen den Teams - Pizza-Mittwoch
C		✓		- Vorstellung der Stories - Code-Camp	- Jour Fixe
D	✓	✓	✓		- individuelle Absprachen
E	✓	✓	✓		- allgemeines Treffen
F		✓	✓	- Vorstellung der Stories	- Table-Talk - Slack
G		✓	✓	- allgemeine Teambesprechung - Story-Kickoff - Understanding	- Besprechung der Makro-Architektur - Entwickler Convention - interne Konferenzen
H	✓	✓	✓	- Vorträge und Workshops	- ausführliche Architektur-analyse
Ø	5/8	8/8	7/8		

Tabelle 5.2: Diskussionsmöglichkeiten

diesem Meeting *„werden schon viele Aspekte diskutiert, die in Richtung der Architektur gehen“* [D, 11:57min]. Die Diskussion findet hauptsächlich im zweiten Teil, beim Aufteilen in Tasks statt: *„dort muss man schon technisch diskutieren, was müssen wir jetzt eigentlich alles ändern“* [E, 06:25min]. Früher wird nicht darüber geredet, weil die Teams dann erst entschieden haben, an diesem Ticket zu arbeiten und dadurch *„nicht zu sehr im Vorhinein über Architektur reden und dann eventuell über etwas reden, was [sie] doch nie machen“* [A,14:58min]. Dort wird auch diskutiert: *„Wie machen wir es den jetzt?“* [G, 18:03min]. Bei diesem Treffen *„überlegen die Entwickler, [...] welche technischen Schritte müssen wir umsetzten“* [F, 09:04min]. Die Schritte stellen einzelne Tasks dar (Task-Breakdown). Die Kanban Teams führen kein Planning-Meeting durch. Sie stellen die neuen Stories in regelmäßigen Kickoff-Meetings vor. Die Aufteilung in konkrete Arbeitsschritte wird in Unternehmen C und F in einem spontanen Meeting, direkt vor Beginn der Entwicklung vorgenommen.

Unternehmen A hat zur Lösung von technischen Unklarheiten ein zusätzliches *Tec-Grooming* eingeführt: *„Da müssen wir uns nochmal überlegen, wie wir das eigentlich machen wollen bevor wir das schätzen können und setzten uns dann aber relativ spontan zusammen.“* [A, 16,42min]. Das zu diesem Zeitpunkt gehörende Schätzen (*Estimation*) der Stories wurde in Unternehmen G in *Understanding* umbenannt. Dies wurde getan, weil es ihnen beim Schätzen nicht um die Dauer der Entwicklung geht. Der Vergleich der geschätzten *Story-Points* soll stattdessen in Erfahrung bringen, ob alle Entwickler ein ähnliches Verständnis davon haben was zur Realisierung getan werden muss.

In kürzeren Zeitabständen wird das Daily Meeting (bzw. Standup) um über Architekturthemen zu reden verwendet. In manchen Projekten wird dies zum teamübergreifenden Austausch eingesetzt (A, E). In Unternehmen A wird das Daily Meeting z.B. zwei mal in der Woche mit allen Teams der Abteilung durchgeführt. Es dient häufig als Einstiegspunkt, um Probleme anzusprechen: *„Meist wird sowas im Standup angesprochen.“* [F, 25:45min]. *„Da dies nicht einmal am Anfang des Sprints, sondern jeden Tag“* [H, 07:25min] stattfindet, eignet es sich gut um kurzfristig Dinge anzusprechen bei denen Feedback gewünscht ist. Häufig löst dies ein zusätzliches, mehr oder weniger spontanes, Meeting aus. Das Daily Meeting soll dadurch nicht unnötig in die Länge gezogen werden. Die Lösungen der Probleme werden häufig erneut im Daily Meeting kurz erklärt.

Ein paar der Unternehmen haben einen zusätzlichen Zeitraum eingeführt, um über den Zustand der aktuellen Architektur zu reden. Das Vorgehen dabei ist bei den Unternehmen sehr unterschiedlich. Dies ist eine Art von *„Retrospektive für die Architektur“* [C, 08:22min]. Ein extra Zeitraum macht Sinn, weil sich gezeigt hat, dass *„wenn man so ein dediziertes Meeting hat, dann kommen doch mehr Sachen raus“* [C, 09:39min], als bei Meetings bei denen andere Themen im Fokus stehen. Bei Unternehmen C ist dies das *Code-Camp*. Unternehmen D führt etwas ähnliches durch. Bei ihnen ist es allerdings konkreter auf einzelne Stories bezogen. Hierbei geht es vor einem Review darum, dass *„Besonderheiten der Entwicklung nochmal vorgestellt werden bevor ein Product Owner dazu kommt“* [D, 11:00min]. Unternehmen C (Jour Fixe) und G führen regelmäßige Debatten über die oberhalb liegende Gesamtarchitektur durch. An diesen nehmen nur die Architekten der Teams teil. Hier sollen Fragen wie: *„Wie würden wir das einheitlich lösen?“* [C, 11:21min] beantwortet werden. Die Ideen tragen die Architekten ihren Teams vor. Für den Zustand der einzelnen Komponenten sind die Teams vollständig selbstverantwortlich. Das Team von G hat zu diesem Zweck zweimal in der Woche eine allgemeine Teambesprechung. Unternehmen H führt eine sehr intensive Analyse der entstandenen Architektur durch. Dies wird im Abschnitt zum Management der Architekturerosion (Kapitel 5.4.3) genauer vorgestellt.

Absprachen zwischen einzelnen Teams, z.B. zu Schnittstellen, werden meist individuell gehandhabt. Unternehmen B dokumentiert umgesetzte Schnittstellen *„formal über ein Wiki“* [B, 23:25min]. Die Einträge stellen die *„Verträge zwischen den Teams“* [B, 23:28min] dar. Unternehmen D führt dies ebenfalls sehr individuell durch. In Zukunft soll ein *„Provider-Consumer Regelgespräch eingeführt [werden], wo solche Austauschmöglichkeiten stattfinden“* [D, 09:45min] können. Die Absprachen können meist nur sehr individuell durchgeführt werden, weil mit *„manchen Teams arbeitet man sehr eng und kurz getaktet“* [G, 31:56min] und *„in anderen Fällen ist das etwas mühseliger, da hängt es dann wirklich von den einzelnen Personen ab“* [G, 32:53min].

Für den allgemeinen teamübergreifenden Austausch stellen viele Unternehmen einen Zeitraum zur Verfügung. Dieser Austausch kann über Architekturthemen, wie auch allgemeine weitere Themen stattfinden. Unternehmen A hat zu diesem Zweck ein wöchentliches Community-Meeting. Inhalt dieses Meetings *„kann im Prinzip alles sein“* [A, 20:50min]. Dies umfasst das Besprechen von Problemen, wie auch das Halten von Vorträgen. In Unternehmen B wurde zu diesem Zweck früher regelmäßig der Pizza-Mittwoch durchgeführt. D möchte dies in Zukunft wieder einplanen. Derzeit bestehe das Problem, dass die *„die was vorzustellen haben, keine Zeit haben etwas vorzustellen“* [D, 17:28min]. Bei Unternehmen F wird dies als *Table-Talk* bezeichnet. Hier kann außerdem ein Mitarbeiter vorstellen, *„woran er gerade arbeitet, wo er sagt, hier brauche ich nochmal Input“* [F, 24:14min]. Dies kann er machen, um Feedback zu erhalten. Unternehmen G hat mehrere Möglichkeiten zum teamübergreifenden Austausch. Zum Einen ist dies eine Entwickler Convention, mit einer *„klassische[n] Vortragssituation“* [G, 16:27min]. Zum Andern werden unregelmäßig interne Konferenzen zu speziellen Themen mit externen Referenten durchgeführt.

Neben den festen und spontanen Meetings findet meist zusätzlich ein sehr individueller aber wichtiger Austausch statt. Dies kann z.B. über Chatsysteme (z.B. Slack in Unternehmen F), wie auch bei Flurgesprächen an der Kaffeemaschine geschehen. Außerdem kann zu allen Diskussionen gesagt werden, dass diese nur mit den dafür relevanten Personen durchgeführt werden sollten. Mit zu vielen Personen *„schweift [die Diskussion] schnell ab oder man hängt sich an Detailfragen auf“* [E, 7:54min].

Die Art, wie eine Diskussion stattfindet, ist abhängig von den Personen im Team. Eine Diskussion kann z.B. mithilfe von Entwurfsmustern geschehen. Hierdurch sprechen die Entwickler eine gemeinsame Sprache. Laut H ist die Einstiegshürde allerdings hoch, weil gute Kenntnisse über die Muster vorhanden sein müssen. Sobald diese überwunden sei, ist die Diskussion im Team einfacher. Wie unter anderem auch D berichten konnte hängt die Art der Diskussion stark vom Team ab und entwickle sich meist ohne konkrete Absprachen.

Zusammenfassung

Zusammenfassend kann gesagt werden, dass verschiedene Diskussionsräume hilfreich sein können. Vor Beginn einer Story sollte mit mindestens einer weiteren Person ein möglicher Lösungsweg erarbeitet werden. Das Daily Meeting kann frühzeitig genutzt werden um Problemen anzusprechen. Das Problem selbst sollte nicht bei diesem Treffen diskutiert werden. Dieses würde dadurch unnötig in die Länge gezogen werden. Außerdem sind viele Personen anwesend, die nicht zur Lösung beitragen können und stattdessen nur von der Arbeit abgehalten werden.

Um über den aktuellen Zustand der Architektur zu reden ist ein dediziertes Meeting sehr wertvoll. Hier können geplante Änderungen, die nicht in Zusammenhang mit einer Story stehen, besprochen werden. Außerdem kann dieses Meeting für eine Art von Retrospektive, bzw. Review für die Architektur eingesetzt werden.

Der teamübergreifende Austausch zu Schnittstellen sollte nicht genauer geregelt werden. Hierzu sind die Teams und Projekte meist zu individuell. Individuell angesetzte Absprachen könnten hier die passendere Lösung sein. Ein weiterer teamübergreifender Austausch, im Rahmen einer festen Vortragssituation, ist wichtig für eine Wissensverbreitung. Die Themen sind meist allgemein und helfen dadurch nicht zwingend bei einem konkreten Problem. Sie können aber den Einstieg in eine neue Technologie vereinfachen und neue Ideen schaffen.

5.4.3 Management der Architekturerosion während der Entwicklung

Dieser Abschnitt vergleicht die Aktivitäten während der Entwicklung zum Erhöhen der Codequalität und dem frühzeitigen Entdecken von Fehlern während der Entwicklung. Hierbei handelt es sich sowohl um Analysen mithilfe von Tools, wie auch um eingesetzte Praktiken. Konkret handelt es sich hierbei um die Punkte der Aktivitäten während der Entwicklung, der Architekturerosion und dem Problemmanagement aus dem Kodierleitfaden. Tabelle 5.3 zeigt die unterschiedlichen eingesetzten Praktiken der einzelnen Projekte.

Alle Unternehmen verwenden Tests um die Lauffähigkeit der Anwendung zu überprüfen. Der Umfang des Einsatzes unterscheidet sich zum Teil. So sagt Unternehmen B, dass sie nur das testen *„was testens wert ist“* [B, 40:57min]. Die Entscheidung, was relevant ist, hängt davon ab *„wie viel Geld [es] kostet wenn es kaputt geht“* [B, 41:03min]. Die Frage nach den entstehenden Kosten ist häufig schwer zu beantworten. Bei einem Defekt kann es sich um einen vollständigen Ausfall oder um eine fehlerhafte Berechnung mit falschen Ergebnissen handeln. Eine hohe Testabdeckung ist wichtig, *„weil spätestens wenn du agil arbeitest, hast du häufig Refaktorings und du willst natürlich nicht, dass du dir etwas kaputt machst. Und so etwas merkst du natürlich*

Unternehmen	*Tests*	*CI-Plattform*	*statische Codeanalysen*	*Conformence Checks*	*Pair Programming*	*Code Review*	*Weiteres*
A	✓	✓			✓	✓	- Zwei-Stufiges Code Review - Approaches und Proposals - Technical-Debt Tickets
B	✓	✓	✓		✓	✓	
C	✓	✓	✓	✓	✓		
D	✓		✓		✓	✓	
E	✓		✓		✓	✓	
F	✓	✓	✓		✓	✓	- Story Verantwortlicher
G	✓				✓		
H	✓			✓	✓	✓	- Todo-Board
Ø	8/8	4/8	5/8	2/8	8/8	7/8	

Tabelle 5.3: Maßnahmen während der Entwicklung

nur, wenn du alles automatisiert testen kannst.“ [E, 34:55min]. Bei einigen Unternehmen werden Features ohne Umweg direkt auf das Produktivsystem (G) übertragen. Deswegen geht bei ihnen *„kein Feature rein, was nicht getestet ist“* [F, 25:05min]. Bei Unternehmen G geht dies soweit, dass diese sogar nach dem Test-First Prinzip arbeiten. Unternehmen H stimmt ebenfalls mit dieser Einstellung überein: *„Testabdeckung ist eines unserer wichtigsten Sicherheitsnetze die wir haben“* [H, 31:51min]. Dies ist wichtig, weil *„ein System wird viel leichter refaktorierbar. Die Leute haben viel mehr Mut zum refaktorieren.“* [H, 32:31min]. Die Entwickler sollen sicher sein, dass sich durch ihre Änderungen die Funktionalität der Anwendung nicht geändert hat. Der Interviewpartner aus Unternehmen B ist etwas kritischer. Er habe *„im Arbeitsalltag noch keine Projekte gesehen, wo der Code so gut strukturiert war, dass [er] etwas refaktoriere und den Test als Beweis hernehmen kann, dass alles funktioniert“* [B, 41:55min]. Hiermit ist gemeint, dass bei vielen Änderungen die Tests so nah am Code sind und deshalb ebenfalls angepasst werden müssen. Dadurch können diese nicht immer eine vollständige Sicherheit darstellen. Auch wenn die Tests angepasst werden müssen ist es aber möglich unerwünscht Seiteneffekte durch andere Tests festzustellen.

Viele Unternehmen führen die Tests und weitere Überprüfungen automatisch mithilfe von *Continuous Integration Plattformen* (CI, z.B. Jenkins[1]) durch. Diese Tools werden häufig zur Berechnung weiterer Metriken eingesetzt. In den meisten Fällen sind Richtlinien vorhanden die bei einer Unterschreitung verhindern, dass eine Story abgeschlossen werden kann. In Unternehmen B werden diese bewusst nicht eingesetzt. Der Interviewpartner ist der Meinung, *„dass die Entwicklungsgeschwindigkeit und die Mentalität der Entwickler“* [B, 36:21] dadurch negativ beeinflusst wird. Die Entwickler würden die Tools irgendwann so gut kennen, dass nur für diese entwickelt würde. Das Ziel guten Code zu erzeugen würde in den Hintergrund geraten. Das Unternehmen setzt solche eine Prüfung zwar ein, hat aber keine Richtlinien für Minimalbewertungen festgelegt. Da der Einsatz einer CI-Plattform sehr unterschiedliche Prüfungen und Richtlinien einführen kann, ist es sehr schwierig diese Aussagen direkt zu vergleichen. Der Interviewpartner aus Unternehmen G empfindet eine *Checkstyle-Prüfung* z.B. als sinnlos, eine Erkennung von potentiellen Bugs teilweise aber hilfreich. Eine CI-Plattform kann statische Codeanalysen durchführen. In einigen Fällen werden hierzu zusätzliche Werkzeuge verwendet. H nutzt so etwas sporadisch in andern Projekten. Die Interviewpartnerin würde sich dies häufiger wünschen. Trotzdem ist sie der Meinung: *„Wenn man die Leute auf Qualität trainiert, dann sehen die auch was komisch ist.“* [H, 30:47min]. Dadurch sind sie *„mit ihren Augen die besten Bug-Detektoren“* [H, 30:44min]. Unternehmen B setzt etwas ein, dass *„über diverse Metriken mitteilen [kann], ob ein Stück Code überarbeitungsbedürftig ist“* [B, 35:02min]. Zum Teil werden die Ergebnisse nicht als relevant genug betrachtet, weil gesagt wird, dass es Bugs gibt, *„wo man weiß, den gibt es seit drei Jahren, [es] stört keinen wenn der noch weitere drei Jahre existiert“* [B, 41:08min]. Zur statischen Analyse haben einige Teams selbstständig Sonarqube eingeführt. Bei Unternehmen E wird dies zur regelmäßigen Prüfung des aktuellen Stands genutzt. Als während der Entwicklung von der fachlichen Seite weniger zu tun war, konnte *„30% [der] Zeit im Sprint auf das Abarbeiten von Sonar-Tickets“* [E, 31,23min] konzentriert werden. Unternehmen C plant für diese Art von Refaktorisierungen durchgehend 20% der Zeit ein. Die anderen Unternehmen haben hierzu keine Regelungen. Wenn größere Änderungen notwendig sind, werden diese aber in der Regel von einem Teil des Teams neben der Entwicklung von Features durchgeführt. Conformance Checks werden nur selten eingesetzt. Unternehmen C hat Versuche hierzu durchgeführt, war aber mit den starren Regeln nicht zufrieden. Der Interviewpartner testete zum Zeitpunkt des Interviews ein weiteres Werkzeug (Moose). Unternehmen H führt mit Sonargraph regelmäßig ausführliche Architekturanalysen durch. Die Interviewpartnerin nennt dies *„Mob-Architecting“* [H, 35:52min]. Hier nimmt das gesamte Team teil. Sie sagt: *„Das ist so ein Boost im Teamverständnis und im Architekturverständnis, dass kriegt man kaum anders*

[1] https://jenkins.io/, Abgerufen: 2.11.2016

hin" [H, 36:08]. Der Effekt beschränkt sich nicht nur auf die Architektur, sondern hilft auch beim Zusammenhalt des Teams. Die Analysen führt sie ebenfalls in andern Teams und in anderen Unternehmen durch. Unternehmen D sieht diese Art von Analyse eher kritisch und bezeichnet es als ein *„Sterben in Schönheit nach irgendwelchen Prinzipien"* [D, 26:13min]. Die Reaktionen nach einem Versuch in diesem Unternehmen waren *„totaler Widerstand oder es war nett, dass wir drüber geredet haben. Ja ein paar Punkte nehmen wir mit."* [D, 25:38min]. Ein festgestelltes Problem dabei ist, dass nicht sicher gegangen werden kann, dass die verglichene Ist-Architektur noch das richtige Ziel darstellt. Diese Art von Analyse bewirke laut H oftmals, dass viele Leute das erste Mal den wirklichen Aufbau der Anwendung zu sehen bekommen. Die anderen Unternehmen konnten von keinen Versuchen in dieser Richtung berichten.
Unternehmen G setzt die meisten Werkzeuge nur temporär ein: *„Wir setzten Tools ein wenn wir ein Problem haben und wenn das Problem weg ist, setzten wir auch das Tool wieder ab."* [G, 44:35min]. Der Interviewpartner sagt zu solchen Prüfungen: *„Das ist nichts was uns vor irgendeinem Schaden [...] retten würde."* [G, 44.25min]. Dies passt zu der Aussage, dass dem Kunden beigebracht werden muss, *„dass es normal ist, dass man immer mal ein bisschen reparieren muss"* [H, 31:29min]. Bei Unternehmen G wird dies zusätzlich damit begründet, dass jedes Werkzeug den Buildprozess verlängern würde.

Die zuvor angesprochenen Vorgehen behandeln Analysen die meist unabhängig oder nach Beendigung einer Story durchgeführt werden. Im Folgenden wird untersucht, was zum Zeitpunkt der Entwicklung eingesetzt wird.
Alle besuchten Unternehmen setzten Pair Programming ein. Dies wird als Mittel der Wahl zur Einarbeitung von neuen Mitarbeitern angesehen. Dadurch wird ihnen schrittweise ein Einblick in das System gegeben. Bei allen wird Pair Programming auch *„bei etwas schwierigeren Situationen benutzt"* [B, 19:29min]. Hier werden Kollegen zur Lösung eines konkreten Problems um Hilfe gebeten. Bei Projekten mit mehreren Teams setzt Unternehmen B dies auch zwischen den Teams ein. Wenn z.B. eine Komponente des anderen Teams verwendet, bzw. angepasst werden muss. Die Unternehmen G und H setzen *„auf täglicher Ebene Pair Programming"* [G, 16:52min] ein. Selbst bei der Entwicklung durch einen Architekten wird keine Ausnahme gemacht, *„weil sonst ist er dort Know-How Träger und das alleine nur er"* [H, 22:47min]. Da aber dieses Vorgehen gerade *„das Wissen verbreitern und andere Meinungen mit rein [zu] holen"* [G, 22:14] soll, ist dies nicht gewünscht. Wenn die Entwicklung einer Story länger als drei Tage dauert, wird in beiden Unternehmen eine Person ausgewechselt, um neues Wissen und einen anderen Blickwinkel in das Team zu bekommen.

Nach der Entwicklung setzten bis auf Unternehmen G und C, alle Unternehmen Code Reviews ein. Bei Unternehmen C kann dies durchaus innerhalb anderer Teams trotzdem geschehen.

Bei G ist dies nicht der Fall, weil jeder Push automatisch ein Deployment auf das Produktivsystem verursacht. Wenn Reviews eingesetzt werden, sind diese technisch häufig so in den Entwicklungsprozess integriert, dass eine Story nur dann abgeschlossen werden kann, *„wenn im Feld Pair-Reviewer jemand drinnen steht“* [E, 36:51min]. Code Reviews bewirken, dass *„im Vier Augen Prinzip, [noch einmal erklärt wird,] was passiert ist“* [E, 36:34min]. Hierdurch wird das Wissen zur Umsetzung nochmals auf eine weitere Person übertragen und Fehler können entdeckt werden. Unternehmen A führt eine besondere Form durch. Ihr Code Review findet in zwei Stufen statt. Im ersten Schritt wird ein Review durch eine Person des gleichen Teams durchgeführt. Anschließend wird ein zweites Review durch eine zufällige Person eines anderen Teams durchgeführt. Dies ist in diesem Unternehmen möglich und auch sinnvoll, da alle auf der gleichen Codebasis arbeiten. Die Gefahr, dass komplexe Code Reviews nur Oberflächlich bearbeitet werden ist allerdings groß. *„Reviews haben wir nur die klassischen. 20 Zeilen Codeänderungen und 100 gefundene Fehler, 2000 Codeänderungen und alles gut“* [B, 18:55min]. Dieses Problem kann eventuell verringert werden, wenn die Code Reviews zusammen mit den Entwicklern der Story durchgeführt werden. Diese erklären dabei Schrittweise ihre Lösung. In Unternehmen B und H werden nur *„bei irgendwelchen kritischen Sachen“* [H, 09:09min] Code Reviews durchgeführt. Dies bewirkt, dass nicht der vollständige Code durch andere Personen begutachtet wird. Hier hängt es davon ab, was der Entwickler als kritisch ansieht. Dadurch können sich schnell Fehler einschleichen. Unternehmen H würde dies gerne verbessern.
Bei Code Reviews ist es *„allerdings immer relativ schwierig über Architektur zu reden“* [A, 26:04min]. Die Arbeit der Umsetzung ist zu diesem Zeitpunkt bereits geschehen. Änderungen in der Architektur würden bewirken, dass viel der investierten Arbeit weggeworfen werden müsste. Aus diesem Grund sollte gerade bei Problemen frühzeitig, noch während der Entwicklung, um Feedback gebeten werden. Pair Programming oder ein vorheriges Besprechen der Lösungsstrategie könnte dieses Problem verringern.

Unternehmen F hat eingeführt, dass für jede Story ein Entwickler verantwortlich ist. Dieser muss nicht zwingend die Entwicklung durchführen. Er dient hauptsächlich als Ansprechpartner bei Fragen zu der Story.

Eine Besonderheit im Vorgehen von Unternehmen A sind die *Approaches* und *Proposals*. Diese werden benutzt, um Feedback zu Ideen oder um Hilfe bei Problemen zu erhalten. Bei Problemen kann es vorkommen, dass sie *„ein Workaround machen“* [A, 27:25min]. Hierbei sei wichtig, dass zuvor feststellt wird ob dies etwas ist, dass länger in der Anwendung bleiben soll oder ob es nur ein zeitlich begrenzter Test ist. Je nachdem welchen Bereich die Änderung betrifft würde dies zuvor über den Chat abgesprochen werden. Zur Erinnerung und zur Lösung der Workarounds werden *Technical-Debt* Tickets angelegt. Dies ist ähnlich wie das *Todo-Board* von

Unternehmen H. In Unternehmen B kann bei Problemen der Lead Developer um Unterstützung gebeten werden. *„Wenn ich helfen kann, mache ich Hilfe zur Selbshilfe“* [A, 16:15min]. Der Lerneffekt sei hierdurch viel höher. Häufig werden aber auch andere Kollegen direkt angesprochen. Bei Unternehmen C und H dienen die Architekten als Ansprechpartner. *„Dann sprechen sie schon meistens häufiger als erstes mit mir darüber und dann reden wir darüber, wie wir es anders machen wollen“* [C, 12:43min]. Das Problem und eine Lösung wird den anderen Teammitgliedern im Daily Meeting vorgestellt. Kleinere Änderungen werden direkt im Rahmen der Story realisiert. Für größere Änderungen werden, wie in den anderen Unternehmen, technische Stories erstellt. Nachdem bei H die Architekten angesprochen wurden, wird das Problem in einer größeren Runde mit den Entwicklern diskutiert. Die Unternehmen D, E und F nutzen hauptsächlich die Daily Meetings zum Ansprechen von Problemen. Wenn dies im schlimmsten Fall das Sprint-Ziel gefährdet wird dies bei Unternehmen D angepasst. Bei Unternehmen F wird in vielen Fällen im Anschluss des Daily Meetings ein spontanes Treffen durchgeführt. *„Dann ziehen wir es raus und nach dem Standup setzten sich die Leute, die das interessiert und der der daran arbeitet [zusammen].“* [E, 25:52min]. Dies hat den Sinn, dass das Daily Meeting nicht in die Länge gezogen wird. Außerdem nehmen hier viele Personen Teil die zur Lösung nicht beitragen können und deshalb nur von der Arbeit abgehalten werden.

Zusammenfassung

Tests sind hilfreich um die Lauffähigkeit einer Anwendung auf Dauer sicherzustellen. Auch bei der Architekturentwicklung können sie helfen, um eine Sicherheit bei Refaktorisierungen zu gewährleisten. Eine Entwicklung nach dem Test-First Prinzip ist in den meisten Fällen praktisch nicht realistisch. Hierzu ist eine sehr genaue Kenntnis der Fachlichkeit und deswegen eine sehr enge Zusammenarbeit mit dem Kunden notwendig.
Weitere Prüfungen mithilfe von statischen Analysen können helfen automatisiert problematische Codestellen zu finden. Da diese Tools hauptsächlich nur einen einmaligen Einrichtungsaufwand benötigen, ist es sinnvoll sie von Anfang an im Projekt zu integrieren und gemeinsam mit den Tests regelmäßig auszuführen. Eine Conformance Prüfung im Rahmen einer Architektur-Retrospektive kann das Verständnis für die Architektur stark erhöhen. In der Praxis wird dies sehr selten durchgeführt. Solch eine Analyse kann sehr komplex und zeitaufwendig sein und ist deshalb auch nicht unbedingt in jedem Projekt sinnvoll einsetzbar. Bei langlebigen Anwendungen können solche Analysen allerdings eine wichtige Rolle für den Erfolg des Projekts darstellen.
Wenn während der Entwicklung im Paar gearbeitet wird, werden Lösungswege automatisch

miteinander diskutiert. Die Partner können gleichzeitig voneinander lernen. Einig sind sich alle, dass dies mindestens bei Problemen und zum Einarbeiten neuer Kollegen eingesetzt werden sollte.
Code Reviews sind empfehlenswert, um kleine Probleme zu finden. Architekturänderungen sind aber schwer zu diskutieren, da die Entwicklung zu diesem Zeitpunkt bereits abgeschlossen ist. Um Probleme nach der Entwicklung vorzubeugen, kann es helfen vor Beginn der Implementierung einen groben Lösungsansatz zu besprechen.

Wenn Teams Techniken einsetzten wollen, mit dem Ziel die Codequalität zu erhöhen, sollte ihnen dies ermöglicht werden. Auch bei Zeitmangel sollte auf Techniken wie Reviews nicht verzichtet werden. Ein vielfaches der gesparte Zeit wird anschließend häufig benötigt, um übersehene Probleme zu korrigieren.

5.4.4 Softwarearchitekt

In diesem Abschnitt wird die Rolle des Softwarearchitekten untersucht. Hierzu wird verglichen, wie diese Rolle in das Team integriert ist und welche Aufgaben von ihr übernommen werden. Eine Übersicht ist in Tabelle 5.4 zu finden.

In 50% der besuchten Teams war eine Architektenrolle vorhanden. Unternehmen H hat keine direkte Architektenrolle. Diese Aufgabe wird von den Entwicklern mit der meisten Erfahrung übernommen. In Unternehmen B hat zu einem früheren Zeitpunkt der interviewte Lead Developer eine ähnliche Rolle gehabt. Derzeit konzentriert sich seine Aufgabe auf DevOps Tätigkeiten. Für die Zukunft würde er sich wünschen seine Tätigkeit wieder mehr auf diese Aufgaben konzentrieren zu können, um sich *„nur noch Lösungen für die Gesamtarchitektur zu überlege[n]“* [B, 24:23min].
Die Architekten in den Unternehmen entwickeln selbst mit. *„Wenn der Architekt nicht mehr selbst codet, dann ist er kein Architekt mehr.“* [D, 19:02min]. Dies ist wichtig damit er selbst wisse ob seine Konzepte gut sind. Laut Interviewpartner G solle er optimalerweise mindestens 50% der Zeit mit entwickeln. Die zur Verfügung stehende Entwicklungszeit ist meist aber sehr begrenzt. *"Das ist aber schwierig, weil dein Kalender einfach fragmentiert durch Meetings ist.“* [G, 34:43min]. Für ein erfolgreiches Pair Programming, wie bei Unternehmen G ist aber ein längerer Zeitraum am Stück notwendig. In den Unternehmen mit Architekt (C, D, G, H) stimmen alle mit folgender Aussage überein: *„Grundsätzlich ist es so, dass bei uns die Architektur vom Team gemacht wird.“* [C, 11:46min]. Der Architekt könne allerdings Vorgaben machen. *„Aber praktisch kommt das auch nicht vor".'* [C, 11:46min]. Einen Nachteil von Vorgaben beschreibt G wie folgt: *„Dann hätte ich einen schlechten Stand und das Ergebnis wäre auf jeden*

Unternehmen	nicht vorhanden	Entwickeln	Coaching / Mentoring	Ansprechpartner	Überblick behalten	hat Entscheidungsgewalt	nutzt Entscheidungsgewalt
A	✓						
B	✓						
C		✓	✓	✓		✓	
D		✓	✓	✓			
E	✓						
F	✓						
G		✓	✓	✓		✓	
H		✓	✓	✓	✓		
Ø	4/8	4/8	4/8	4/8	1/8	2/8	0/8

Tabelle 5.4: Aufgaben des Architekten

Fall schlechter.“ [G, 36:57min]. Die Teammitglieder müssen von den Ideen des Architekten erst überzeugt werden: *„Wenn ich irgendwas haben möchte muss ich Argumentationsarbeit leisten.“* [C, 12:21min].

Die Aufgabe des Architekten ist *„dafür zu Sorgen, dass die Qualität des Systems erhalten bleibt“* [H, 19:42min]. Dies wird unter anderem dadurch gemacht, dass er dafür sorgt, dass die *„nicht-funktionale Anforderungen eingehalten werden“* [G, 12:33min]. Die Hauptaufgabe des Architekten ist in allen Unternehmen das Coachen, bzw. Mentoring der Teammitglieder. Außerdem dient er als allgemeiner Ansprechpartner bei technischen Fragen. Die Fragen können sowohl vom Team selbst, wie auch von Außenstehenden kommen. *„Er muss dabei die Sachen nicht mal selbst beantworten, sondern kann dass dann halt auch an andere im Team weitergeben.“* [D, 19:38min]. Dies ist erforderlich, weil es für einige Bereiche der Anwendung immer Spezialisten geben wird. E konnte von einem Projekt berichten bei dem es zwar keine offiziellen Architekten gab, stattdessen aber für alle Teilbereiche Spezialisten. Diese haben sich mit der Zeit automatisch gebildet und dadurch ähnliche Aufgaben, wie die eines Architekten, übernommen. Auch wenn der Architekt nicht alle Details kennen muss, sollte er die Gesamtarchitektur im Auge behalten. Bei Unternehmen wie G ist er dafür mitverantwortlich, dass die überliegende

Gesamtarchitektur, die die einzelnen Komponenten verbindet, weiterentwickelt wird.
Der Architekt ist häufig als technische Vertretung für die Teams in Meetings anwesend. *„Ich bezeichne mich immer als Meeting-Firewall fürs Team."* [G, 35:55min] Dies sind häufig Meetings, die lange vor der Entwicklung ohne Softwareentwickler stattfinden. Der Interviewpartner E sagt, dass ein Architekt zu Beginn eines Projektes unterstützen kann: *„der hilft somit die Grundsteine zu legen."* [E, 24:18min]. Die Entwicklung solle mit der Zeit vollständig vom Team übernommen werden: *„Aber dessen Aufgabe ist es eigentlich auch das Team dahin zu führen, dass die das nachher selbstständig weiterentwickeln."* [E, 24:22min]. Das Ziel eines guten Architekten solle es also sein, sich mit der Zeit selbst überflüssig zu machen. Dies kann bei der konkreten Entwicklung der Architektur möglich sein. Der Architekt kann aber dafür sorgen, dass der Prozess überhaupt stattfindet und gleichzeitig eine Stellvertreterrolle für die technische Seite des Teams übernehmen.

Zusammenfassung

Die Rolle des Architekten hat nichts mehr mit dessen früheren Aufgaben zu tun. In den hier untersuchten Unternehmen ist dieser tief ins Team integriert und entwickelt selbst, soviel wie zeitlich möglich, mit. Das Recht Vorgaben zu machen wird nicht benötigt. Dies wurde auch wenn er das Recht hatte bewusst nicht eingesetzt. Es konnte häufig nachgewiesen werden, dass Entscheidungen ohne Teamabsprachen kritische Folgen haben können. Stattdessen sollte er das Team anleiten und coachen. Dies bedeutet, dass er z.B. durch eine Moderation und gezieltem Nachfragen die technische Entwicklung der Anwendung vorantreibt. Mit fortschreiten des Projektes kann das Team die Aufgabe übernehmen die Architektur selbstständig weiterzuentwickeln. Seine Aufgabe ist außerdem als technische Vertretung für die Entwickler nach außen zu agieren. Dadurch kann er früh an Meetings mit dem Kunden teilnehmen und bereits bei der Formulierung von Features unterstützen. Durch das frühe Mitspracherecht kann er bewirken, dass technisch sehr aufwendige Wünsche, die keinen besonderen Mehrwert liefern, früher verworfen werden.

5.4.5 Dokumentation

In diesem Abschnitt wird die Dokumentation in den Unternehmen verglichen. Hierbei geht es sowohl um eine direkte Dokumentation der Architektur, wie auch um das Festhalten von Entscheidungen. Tabelle 5.5 stellt die unterschiedlichen Dokumentationsarten der Unternehmen gebündelt dar.

Unternehmen	*Inline*	*Installation*	*Architekturmodelle*	*Entscheidungen*	*Weiteres*
A	✓	✓			Technical-Debt Tickets Approaches und Proposals
B	✓	✓			Fachliches Konzept
C	✓	✓			Jour Fixe Protokoll
D	✓	✓	✓	✓	Tests
E	✓	✓			Kommentare an Stories
F	✓	✓	✓		Slack-Archiv
G	✓	✓	✓		Aufbau einer automatischen Modellerzeugung
H	✓	✓		✓	Todo-Board
Ø	8/8	8/8	3/8	2/8	

Tabelle 5.5: Dokumentation

Grundsätzlich erstellen allen Teams eine inline Dokumentation. Hierbei werden meist einzelne Methoden, bzw. Zeilen beschrieben. Ein besseres Verständnis für die Architektur ermöglicht dies aber nicht. Sie unterstützen aber bei der Weiterentwicklung und der damit verbundenen Verwendung vorhandener Methoden. Interviewpartner E konnte von Beschreibungen berichten, die nicht mehr zur derzeitigen Funktionsweise gepasst haben. Wenn diese Fehlinformationen liefern ist dies sehr problematisch. Eine andere Art von Dokumentation, die alle besuchten Unternehmen durchgeführt haben, ist eine Installationsanleitung. Diese dient zum Aufsetzten des Systems und hilft deshalb ebenfalls nicht besonders gut um zu verstehen wie die Anwendung aufgebaut ist.
Eine richtige Architekturdokumentation wird selten durchgeführt. Interviewpartner E konnte von verschiedenen Projekten berichten: *„Meine Erfahrung ist, dass ist immer alles sehr schlecht.“* [E, 38:23min]. Der Interviewpartner A kann dies ebenfalls von seinem Projekt berichten: *„Es ist aber nicht so, [...] dass wir eine Architekturdokumentation haben.“* [A, 48:14min]. G sagt, dass sie durch Pair Programming keine ausführliche Dokumentation benötigen: *„Darüber verbreitet sich Wissen und deswegen brauchen wir auch keine Dokumentation über die interne Mikro-Architektur.“* [G, 17:08min]. Dies stimmt zu dem aktuellen Zeitpunkt, *„aber wenn das jetzt Code ist, wo jemand das Projekt verlassen hat, dass versteht kein Mensch mehr“* [E, 42:04min].

Die Dokumentation sollte aber das Ziel haben auf lange Zeit verständlich zu sein.
In Unternehmen B wurde lange Zeit ein Architekturdiagramm manuell vom ursprünglichem CTO gepflegt. Inzwischen *„hinkt [dieses] so ungefähr ein Jahr hinterher“* [B, 50:11min]. Mit dem Wechsel des CTOs wurde das Diagramm nicht weiter gepflegt. *„Prinzipiell existiert dies eigentlich nur im Kopf weniger Leute.“* [B, 49:41min]. Dies sei *„sehr problematisch, aus Sicht der Fachabteilungen. Da fehlt es definitiv an Verständnis der Gesamtarchitektur.“* [B, 50:43min]. Dies hatte einmal zur Folge, dass zwei Funktionalitäten gebaut wurde die sich gegenseitig ausgeschlossen haben. Es ist fraglich, ob die fachliche Produktentwicklung wirklich exakte Kenntnisse über die technische Architektur haben muss. Zum einen ist die Wahrscheinlichkeit, dass das notwendige technische Wissen vorhanden ist sehr gering. Zum anderen entsteht die Gefahr, dass dem Team von außerhalb technische Details vorgegeben werden. Bei D werden teilweise Architektur-Blueprints zur Dokumentation von Entscheidungen erstellt. Unternehmen F pflegt ein vollständiges Diagramm der Architektur. Vor der Entwicklung malen die Entwickler meist händisch ihre Änderungen in das Diagramm. Anschließend werden diese in die digitale Version übertragen *„Und wenn es eingebaut ist malen wir es nochmal schön auf.“* [F, 21:11min]. Sie achten dabei sehr darauf, dass dies aktuell gehalten wird. Unternehmen G hat ebenfalls ein Modell der Makro-Architektur. Das Problem hierbei ist: *„Das ist ein großes Netz, da sind viele Systeme drauf und viele Pfeile und so, was soll ich da jetzt raus lesen.“* [G, 48:39min]. Durch die Größe ist das Diagramm sehr unübersichtlich und deshalb nicht besonders hilfreich. Außerdem *„ist das auch noch manuell gepflegt, das entspricht sowieso nicht der Wahrheit.“* [G, 26:44min]. Dadurch würden zum Teil vollständige Komponenten fehlen. Da alle Services bereits derzeit einen maschinenlesbaren Statusbericht zur Verfügung stellen, soll dieser mit hilfreichen Informationen (z.B. betreuendes Team, Abhängigkeiten, ...) angereichert werden. Hieraus soll automatisch ein Modell abgebildet werden.

Unternehmen H sagt, dass es viel wichtiger ist *„die Entscheidungen und nicht die Bilder zu dokumentieren“* [H, 40:17min]. Dies ist wichtig, weil ein Diagramm nicht mehr aussage weshalb sich etwas so entwickelt hat. *„Das Optimum [...] ist, dass du zumindest zu jedem Checkin eine Referenz auf die Story eingibst.’* [E, 39:03min]. Durch diese einfache Möglichkeit können bereits einzelne Entwicklungsabschnitte mit Stories in Verbindung gebracht werden. An der Story kann eventuell eine Diskussion stattgefunden haben. Diese kann aber auch helfen, um den passenden Ansprechpartner zu finden, sofern dieser noch im Unternehmen ist. Interviewpartner D sagt: *„Tests sind z.B. auch wunderbare Dokumentationen.“* [D, 34:27min]. Die Tests können Informationen dazu liefern, wie etwas verwendet wird und was das erwartete Ergebnis ist. Gesprächspartnerin H kannte dieses Argument, war aber skeptisch weil sie praktisch bisher nicht feststellen konnte, dass Tests zum Verständnis von Software eingesetzt

wurden. D erzählte: *„Seit dem wir viel mit agilen Vorgehen unterwegs sind, dokumentiere ich mehr als vorher.“* [D, 29:08min]. Dies komme daher, dass Leute die Wert auf Qualität legen diese verwenden. Interviewpartner A sagt, dass bei ihnen die Proposals und Approaches als Entscheidungsdokumentation dienen könnten. Hierzu müsse aber an den dazugehörigen Einträgen eine Diskussion stattgefunden haben. Bei einer persönlichen Diskussion fehlt dies. Es ist wichtig die Gründe bei den Entscheidungsdiskussion zu notieren. Dadurch kann zu einem späterem Zeitpunkt evaluiert werden, ob dies noch aktuell ist: *„Hat sich das geändert? Können wir jetzt mit ruhigem Gewissen eine andere Entscheidung treffen.“* [D, 31:11min]. E sagt, dass diese Art der Dokumentation meist nur zu Anfang eines Projektes stattfindet, *„da sind noch alle motiviert, dann schreiben sie auch viel auf. Dann zwei Jahre später, guckst du rein und es hat sich nicht verändert“* [E, 40:56min]. Tragisch sei dies aber erst, wenn die Leute die dies umgesetzt haben nicht mehr im Team sind.
Diese Art von Dokumentation eignet sich nicht zum Einarbeiten neuer Mitarbeiter. *„Das ist eine Dokumentation für Interne“* [D, 31:11min]. Gedacht ist sie stattdessen, um zu einem späterem Zeitpunkt Entscheidungen nachvollziehen zu können. In Unternehmen H werden gemeinsam getroffene Entscheidungen durch die Architekten per E-Mail verteilt. Die Entscheidungen stellen keine Richtlinien dar. *„Es ist was flüchtiges, also es war in dem Moment wichtig, um dann festzustellen, dass man so weiter macht. Aber dann verfliegt es auch schon wieder.“* [H, 28:03min]. Dies bedeutet, dass es wichtig ist nicht auf früher festgelegte Entscheidungen zu beharren, sondern diese unter Umständen anzupassen oder vollständig zu verwerfen. Entscheidungen werden nur getroffen, um zu dem aktuellen Zeitpunkt in eine spezielle Richtung zu entwickeln. Die Meinung von G stimmt hierzu überein: *„Was nutzt es, wenn ich dann in die TD [(Technical Designer)] Runde reingehe und 12 Leute sagen mir, aber gucke mal, da haben wir es ja entschieden. Ja, aber die Entscheidung war falsch.“* [G, 52:38min].
Ob eine Entscheidung oder das technische Konzept einer Realisierung dokumentiert wird, hängt größtenteils von dessen Umfang ab: *„es muss so ein gewissen Maß an Größe haben.“* [B, 47,45min]. Auch bei D hängt das Ausmaß der Dokumentation sehr von der Größe, bzw. Komplexität der Entscheidung ab. Die Ergebnisse werden häufig in einem Wiki festgehalten. Wie auch bei G: *„Es gibt ein Wiki. Ich nenne es das Datengrab.“* [G, 51:28min]. In einigen Fällen werden hier Informationen hinterlegt. *„Ich kenne nur niemanden der da rein guckt.“* [G, 51:58min]. Das Problem, dass eine Dokumentation ohne Sinn erzeugt wird, ist groß. Erst wenn Personen sie aktiv nutzen bleibt sie auf einem aktuellem Stand. *„Wenn sie irgendwann nicht mehr genutzt wird, dann veraltet sie aber auch mit Recht.“* [D, 30:20min].

Die Technical-Debt Tickets von Unternehmen A sind ähnlich wie das Todo-Board von H. Hier werden gefundene technische Schulden für eine spätere Beseitigung notiert. Dadurch

soll verhindert werden, dass diese in Vergessenheit geraten. Bei dieser Art von Ticket wird häufig eine Begründung für die Änderung notiert. Wenn diese, wie im Vorschlag von E, in der Commit-Nachricht verlinkt wird, kann die Begründung später nachgelesen werden. Bei Unternehmen B existiert vom Business Development ein fachliches Konzept welches direkt auf die Architektur einwirkt. Dieses kann ebenfalls als indirekte Dokumentation der Architektur betrachtet werden. Eine Aussage, wie gut eine Abbildung vom fachlichem Konzept auf die tatsächliche Implementierung stattfinden kann, kann hier nicht getroffen werden.
Das Treffen der Architekten wird in C mithilfe eines Jour Fixe Protokolls festgehalten. Hier wird über die gesamte überliegende Makro-Architektur geredet. Entscheidungen werden für alle zugänglich notiert und veröffentlicht.
Bei F wurde erzählt, dass viele der Diskussionen über das Chat System Slack stattfinden. Die Archive der alten Diskussionen können zum späteren Nachvollziehen genutzt werden. Die Gefahr bei solch einer Dokumentation ist, dass diese sehr unstrukturiert ist. Außerdem kann es vorkommen, dass sich auf Gespräche z.B. aus einem Meeting bezogen wird. Diese sind zu einem späteren Zeitpunkt nicht mehr nachvollziehbar.

Zusammenfassung

Eine Inline- und Installations-Dokumentation ist wichtig und hilft bei der Weiterentwicklung und dem Aufsetzen der Anwendung. Eine Übersicht über die Anwendung liefert dies aber nicht. Allgemein konnte kein Unternehmen sagen, dass ihre Dokumentation dazu genutzt werden kann, um sich in die Anwendung einzuarbeiten. Die Dokumentationen ist für Personen gedacht, die die Anwendung bereits kennen.
Allgemein ist schwer abzuschätzen, welches Maß an Dokumentation notwendig ist. Hier unterscheiden sich die Firmen teilweise sehr. Die meisten führen keine oder nur eine geringe Dokumentation der Architektur durch. Ein manuelles gepflegtes Diagramm, welches von manchen Unternehmen gepflegt wird, wird auf Dauer wahrscheinlich zu komplex und dadurch nicht mehr pflegbar. Häufig kann eine Art von Verlauf aus Chat-Protokollen, Meeting-Protokollen oder E-Mails herausgelesen werden. Wenn währenddessen zusätzliche mündliche Absprachen stattfanden gehen diese verloren. Sehr nützlich ist so etwas wie ein Todo-Board, um technische Schulden zu Dokumentieren. So gehen diese nicht verloren und können gezielt abgearbeitet werden.
Wichtig bei allen Dokumentationen ist, dass die Gründe für eine Entscheidung notiert werden. Das reine Erzeugen von Diagrammen macht keinen Sinn, wenn die Begründung für die Art der Entwicklung nicht erfasst wird.

5.5 Ableitungen zum Kontext der Unternehmen

Dieser Abschnitt soll untersuchen ob Zusammenhänge zwischen den einzelnen Faktoren erkannt werden können. Ein eindeutiges Ableiten von Rückschlüssen ist schwer, weil die Unternehmen sehr unterschiedlich vorgehen.

Durch die Arbeit im Projektgeschäft (E, F) kann es vorkommen, dass die Zusammenstellung der Teams variiert. Da die Teams ihr Vorgehen in den meisten Fällen selbst auswählen und anpassen kann die Vermutung entstehen, dass deren Vorgehen nicht so gut strukturiert ist, wie bei Teams die sehr lange existieren. Dieser Verdacht konnte allerdings nicht bestätigt werden. Eventuell ist sogar gegenteiliges der Fall, weil diese Teams häufiger die Chance haben von Vorne zu beginnen, um somit andere Wege auszuprobieren. Da bei den Fallstudien nur zwei Unternehmen im Projektgeschäft tätig waren, ist es schwer eine verlässliche Ableitung zu tätigen.

In drei der acht Unternehmen wurden händisch Diagramme der Gesamtarchitektur erstellt (B, F, G). Bei den Interviews konnte allerdings festgestellt werden, dass die nur noch in einem Unternehmen auf dem aktuellen Stand war (F). Bei den anderen Unternehmen wurde die Pflege entweder vollständig beendet (B) oder versucht durch einen anderen Weg automatisch aktuell zu halten (G). Da das Unternehmen mit dem aktuellem Diagramm (F) noch eine relativ übersichtliche Architektur besaß, war eine Pflege dort möglich. Es besteht aber der Verdacht, dass diese Pflege ebenfalls vernachlässigt wird sobald die gesamte Anwendung an Komplexität gewinnt.

In Ansätzen können Hinweise gefunden werden, dass die größeren Unternehmen (A, C, D, G) versuchen starke Strukturen zu integrieren. Diese Strukturen beschränken sich auf Belange oberhalb der teaminternen Organisation. Bei diesen Unternehmen existieren häufig feste Gremien zum Austausch oder sollen bald eingeführt werden. Teamintern bestimmen die Teams ihr Vorgehen selbst. Dies kann darauf zurückzuführen sein, dass in den meisten Interviews festgestellt wurde, dass die Kommunikation über Teamgrenzen hinweg am meisten Probleme bereitet.

Weiter Rückschlüsse sind schwer zu fällen. Es konnte keine auffällige Verbindung, z.B. zwischen der Größe des Teams und der Art der internen Absprache, gefunden werden. Weitere Aussagen würden auf reinen Vermutungen basieren.

5.6 Abbildung in das Scrum Framework

In diesem Abschnitt werden die gewonnen Erkenntnisse aus den Interviews und den vorherigen Kapiteln vereint und in eine Idee für einen Scrum Prozess integriert. Dieser Prozess stellt eine analytische Erweiterung des Prozesses auf Basis der gewonnenen Erkenntnisse dar. Der Prozess stellt keine allgemeingültige Lösung dar. Jedes Projekt und dessen Kontext ist individuell und benötigt deshalb eine Anpassung des Vorgehens. Der im Folgenden entwickelte Prozess stellt nur ein stark vereinfachtes Beispiel dar und ist nicht praktisch evaluiert.

Der im Anschluss beschriebene Prozess ist eine Anpassung von Scrum. Das gesamte Vorgehen ist in Abbildung 5.1 dargestellt. Die Grafik ist eine angepasste Version der Abbildung aus Kapitel 2.4.1 von [16]. Grundsätzlich ist das Vorgehen gleich geblieben. Die Features werden im

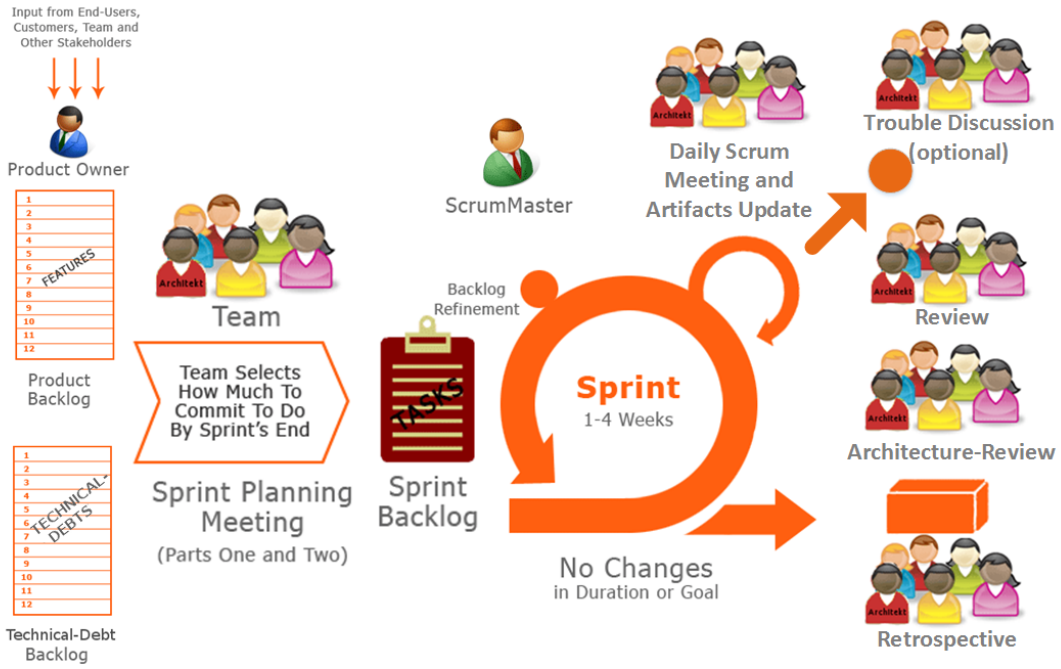


Abbildung 5.1: Vorgehen mit Debatten, Abbildung angelehnt an [16]

Product Backlog gesammelt und beim Sprint Planning vom Team ausgewählt. Anschließend findet der Sprint inklusive der Daily Meetings, dem Review und der Retrospektive statt. Diese Bestandteile bleiben unverändert.
Neu ist, dass eine Person die Zusatzrolle des Architekten hat. Diese Person ist in erster Linie ein Entwickler und hat keine besonderen Rechte den anderen gegenüber. Seine Zusatzfunktion

soll als technische Vertretung des Teams stattfinden. Dadurch kann er die technische Seite des Teams, z.B. in Meetings mit dem Kunden oder für Absprachen mit anderen Teams, vertreten. Seine zweite Aufgabe ist das Coachen und Leiten der Architekturentwicklung. Leiten bedeutet, dass er Diskussionsthemen vorbereitet und darauf achtet, dass sich die Architektur wie der Rest der Anwendung weiterentwickelt. Er hat keinerlei Bestimmungsgewalt. Alle Ideen die er hat sind nur Vorschläge, von denen er das Team überzeugen muss. In der Abbildung ist der Architekt zwischen den Entwicklern dargestellt (rot mit Aufschrift Architekt). Dies soll verdeutlichen, dass er hauptsächlich den Entwicklern zugeordnet ist.
Eine weitere Erneuerung ist die Problemdiskussion (*Trouble-Discussion*). Diese soll stattfinden, wenn im Daily Meeting ein Problem angesprochen wird über welches nochmal geredet werden muss. Dieses Treffen ist optional und sollte spontan bei Problemen eingesetzt werden. Es müssen nicht alle Teammitglieder teilnehmen. Es reicht, wenn der Architekt, die Personen mit dem Problem und die die zur Lösung beitragen können teilnehmen. Durch eine kleinere Gruppe wird das Meeting effizienter. Außerdem werden weniger Personen von der eigentlichen Arbeit abgelenkt. Das Ergebnis der Besprechung kann anschließend z.B. per E-Mail verteilt oder im nächstem Architektur-Review vorgestellt werden.
Nach Beendigung eines Sprints findet ein zusätzliches Architektur-Review statt. Im Gegensatz zum normalen Review nimmt hier nur das Entwicklerteam teil. Hier soll der aktuelle technische Zustand des Systems untersucht werden. Dazu zählen in erster Linie Punkte, die den Entwicklern während der täglichen Arbeit aufgefallen sind. Diese werden als Stories formuliert und in das Technical-Debt Backlog übertragen. Eine zweite Analyse, die regelmäßig durchgeführt werden kann, ist eine Analyse mit Sonargraph wie in Kapitel 3.5 vorgestellt wurde. Durch eine regelmäßige Durchführung kann die Entwicklung im Vergleich zum vorherigen Sprint verglichen werden. Dies ermöglicht festzustellen ob sich die Gesamtarchitektur verbessert oder verschlechtert hat. Notwendige Änderungen werden als Technical-Debt Story notiert.
Das Technical-Debt Backlog wird gemeinsam mit dem Product Backlog im Refinement verfeinert. Eine Trennung findet nicht statt. Beide Backlogs dienen gleichermaßen als Input im Sprint Planning.
Eine zusätzliche Architektur-Retrospektive, bei der über die Art der Architekturentwicklung geredet wird, ist nicht vorgesehen. Dies hat den Grund, dass bereits in der normalen Retrospektive über die Art der Arbeit geredet wird. Die Architekturentwicklung sollte auch ein Teil dieser sein. Die Arbeitsweise der Architekturentwicklung ist nicht zwingend von der allgemeinen Arbeitsweise trennbar. Außerdem wurde versucht die neuen Meetings auf ein Minimum zu begrenzen.
Es muss festgelegt werden, wie viel Zeit des Sprints für Features und wie viel für die Abar-

beitung der technischen Tickets zur Verfügung gestellt wird. Das Team muss dies mit dem Architekten und dem Produkt-Owner abmachen. Empfehlenswert ist hier ein fester Zeitraum, der nur gering variiert wird. Die Technical-Debt Stories werden während der Entwicklung wie normale Stories betrachtet und abgearbeitet. Im Review werden die Technical-Debt Stories nicht vorgestellt. Diese können im Architektur-Review durchgegangen werden.

Weitere Erkenntnisse, wie z.B. zum Austausch zwischen den Teams oder konkreten Absprachen, z.B. zu Schnittstellen, werden nicht in das Verfahren integriert. Dieses Verfahren stellt den Ablauf für ein einzelnes Team dar. Ein allgemeiner Austausch hat in den meisten Fällen keine direkte Verbindung zu dem Projekt und sollte deswegen nebenher stattfinden. Absprachen in einem Projekt sind meist sehr individuell. Das Festlegen eines genauen Rahmens hierzu würde die Agilität vermutlich einschränken. Aus diesem Grund sollten diese Absprachen individuell durchgeführt werden.
Die Thematik der Dokumentation ist nur im Rahmen des Technical-Debt Backlogs behandelt worden. Dies stellt noch keine Architekturdokumentation dar. Der Umfang und die Art der Erstellung der Dokumentation hat keinen direkten Einfluss auf das Vorgehen und sollte innerhalb der Stories mit beachtet werden.
Der Einsatz von Tools und Praktiken wurde in dem Vorgehen nicht berücksichtigt. Der Einsatz von Tools ist in den meisten Fällen sinnvoll, sollte aber vom Team entschieden werden.

5.7 Fazit

Zu Anfang dieses Kapitels wurde die Herangehensweise an die Interviews zur Informationsbeschaffung vorgestellt. Dazu zählt unter anderem die Auswahl der Firmen. Wie gewünscht war es möglich Firmen mit unterschiedlichen Organisationstypen zu interviewen. Hierzu konnten für jeden Bereich (Projektgeschäft, Produktgeschäft, interne Entwicklung) unterschiedliche Interviewpartner gefunden werden. Die Inhalte der Interviews wurden anhand einer Sammlung von Themen ausgewählt. Aus diesen wurde ein Interviewleitfaden zusammengestellt. Der Leitfaden hat den Interviews eine Struktur gegeben. Er wurde aber nicht als fester Fragebogen eingesetzt. Dadurch konnte sich besser ein Gespräch entwickeln.

Die durchgeführten Interviews wurden im ersten Schritt einzeln ausgewertet. Zu diesem Zweck wurden hauptsächlich Besonderheiten hervorgehoben. Anschließend fand die Gesamtauswertung nach unterschiedlichen Oberkategorien statt. Hierzu wurde ein Kodierleitfaden, ähnlich zu einer qualitativen Inhaltsanalyse, erstellt. Die Punkte des Kodierleitfadens wurden in Abschnitten mit Oberkategorien analysiert. Hierzu wurden passende Zitate aus den Interviews

ausgewählt. Im ersten Schritt wurden allgemeine Diskussionsmöglichkeiten im Prozess verglichen. Dabei konnte festgestellt werden, dass viele der bereits aus Scrum bekannten Meetings eingesetzt werden. Als Ergänzung, fanden zum Teil zusätzliche Diskussionen statt. Diese waren in den Unternehmen sehr unterschiedlich aufgebaut. Das alle Unternehmen versuchen diesem Problem entgegenzuwirken macht deutlich, dass ihnen das Problem bewusst ist. Die starken Unterschiede in den Ansätzen zeigen aber auch, dass es noch kein richtiges Vorgehen gibt. Die Teams versuchen das Problem aus der Not zu lösen.
Im zweiten Schritt wurden die Aktionen während der Entwicklung verglichen. Diese haben das Ziel die Erosion gering zu halten. Hier waren sich die Interviewpartner größtenteils einig was z.B. Pair Programming und Code Reviews angeht. Auch wenn diese Maßnahmen in sehr unterschiedlichem Ausmaß eingesetzt wurden (z.B. dauerhaftes Pair Programming oder nur bei Problemen). Tests wurden von allen als sehr wichtiges Mittel angesehen, um eine gewisse Sicherheit beim Refaktorisieren zu gewährleisten. Es konnte aber auch festgestellt werden, dass dies schwieriger wird, wenn die Tests ebenfalls angepasst werden müssen. Eine ausführliche Prüfung der gesamten Anwendung fand nur in einem Unternehmen regelmäßig statt. In den anderen Unternehmen wurde nie die gesamte Anwendung auf eine vorhandene Erosion hin untersucht.
Die Meinungen zur Architektenrolle ähnelte sich teilweise. In der Hälfte der Teams war ein Architekt vorhanden. Alle waren sich dabei einig, dass der Architekt in erster Linie ein Entwickler sein sollte und deshalb auch keine Vorgaben oder ähnliches machen darf. Er kann eine technische Vertretung für das Team darstellen und darauf achten, dass eine Weiterentwicklung der Architektur stattfindet.
Der letzte verglichene Aspekt handelte von der Dokumentation. In der Mehrzahl der Teams wurde diese vernachlässigt. Teilweise wurden manuell Diagramme gepflegt. Bei diesen ist fraglich, wie effizient diese auf Dauer weitergepflegt werden können. Entscheidungen wurde ebenfalls selten notiert. Ein allgemeines Vorgehen zur Dokumentation konnte nicht identifiziert werden. Wenn eine Dokumentation erstellt wird ist es sehr wichtig die Gründe für eine Entscheidung zu notieren. Nur so kann dies zu einem anderem Zeitpunkt nachvollzogen werden.

Nach der Gesamtauswertung wurde eine Idee für ein neues Vorgehen vorgestellt. In diesem Vorgehen sind Informationen aus den Interviews eingeflossen. Die Aktivitäten während der Entwicklung, wie auch die Dokumentation wurde zum großen Teil außen vor gelassen.
Das Vorgehen führt einen neuen technischen Aspekt ein. Zuvor wurde rein auf Feature Basis gearbeitet. Dies erhöht die Komplexität des Vorgehens, weil ein zusätzliches Backlog eingeführt wurde. Eventuell könnte es helfen beide Backlogs zu einem zu vereinen. Spätestens im

Sprint-Backlog werden die Stories gleich behandelt. Durch die Trennung wird allerdings eine getrennte Priorisierung möglich. Die Priorisierung findet außerdem durch unterschiedliche Personen statt. Ansonsten ist vorstellbar, dass neue Funktionalitäten schnell eine höhere Priorität bekommen als eine Refaktorisierung. Durch die Meetings zum Review der Architektur und der Problemdiskussion wird zusätzliche Zeit benötigen. Problemdiskussionen finden allerdings bereits jetzt häufig statt, sind aber nicht in dem Vorgehen eingeplant. Außerdem ist vorstellbar, dass im gesamten Zeit eingespart wird. Es wird möglich notwendige Änderungen frühzeitig zu erkennen. Die Zeit müsste ansonsten in die Lösung der Probleme beim Auftreten investiert werden. Dies wird von der Feststellung bestätigt, dass eine hohe Erosion die Weiterentwicklung stark verlängern kann.
Das Vorgehen selbst wurde rein theoretisch entworfen. Eine praktische Evaluation fand nicht statt. Aus diesem Grund stellt es nur einen Vorschlag dar. Ob dieser Vorschlag praktisch einsetzbar ist muss ausführlich untersucht werden.

Die gesamte Auswertung konnte zeigen, dass die Unternehmen sich des Problems bewusst sind. Die unterschiedlichen Ansätze zeigen aber auch, dass dies eher ein Handeln aus der Not ist. Die Unternehmen haben erkannt, dass es für eine langlebige Software von starker Bedeutung ist eine gute Architektur zu entwickeln. Wie dies auf Dauer sichergestellt werden kann, ist aber ungewiss. Um dies zu lösen werden unterschiedliche Ansätze verfolgt. Diese sollen das eingesetzte Vorgehen ergänzen. Es wurde auch festgestellt, dass die Architektur kein starres Konstrukt darstellt, sondern wie die Anwendung stetig weiterentwickelt werden muss. Das Team selbst stellt eine wichtige Komponente bei der Architekturentwicklung dar. Eine Strukturierung der Anwendung, die die Entwicklung durch mehrere Teams unterstützt, kann helfen. Die einzelnen Teams müssen dabei möglichst unabhängig arbeiten können. Gerade Absprachen über Teamgrenzen hinweg sind schwierig und können starke Probleme verursachen. Einige sehr interessante Informationen aus den Interviews sind nicht mit in die Auswertung eingeflossen. Diese sind nur in den Interviewberichten im Anhang vorhanden. Diese Informationen standen meist nicht direkt mit der Architektur in Verbindung, können aber trotzdem wichtige Punkte für den Erfolg einer Entwicklung darstellen. Durch die Begrenzung auf einen Themenschwerpunkt wurde entschieden diese Aspekte hier nicht weiter zu beachten.

6 Schluss

In diesem Kapitel wird die Arbeit zusammengefasst und ein allgemeines Fazit gezogen. Zusätzlich wird ein Ausblick zur Fortführung des Themas gegeben. Dieser drückt aus, in welchen Bereichen weitere Untersuchungen notwendig sind und welche Bereiche in dieser Arbeit nicht betrachtet wurden.

6.1 Zusammenfassung

Das Ziel dieser Arbeit war zu untersuchen, wie eine Architekturentwicklung im Rahmen eines agilen Entwicklungsprozesses stattfinden kann. Durch Architekturdebatten soll es möglich werden, langlebige Softwareanwendungen zu entwickeln. Für eine langlebige Anwendung ist es wichtig, dass nichtfunktionale Anforderungen, wie Erweiterbarkeit und Wartbarkeit, unterstützt werden. Dieses Ziel kann nur durch eine geringe Erosion der Anwendung mit wenigen technischen Schulden erreicht werden. Die Debatten sollen die Schulden einerseits verhindern und andererseits aufdecken. Dies ermöglicht es die Erosion zu kontrollieren.

Zu Beginn wurde im Kapitel 2 untersucht, was eine Softwarearchitektur ist und welche Auswirkungen es haben kann, wenn sich diese zufällig entwickelt. Es konnte festgestellt werden, dass sich sowohl bei der zufälligen wie auch der gerichteten Entwicklung technische Schulden ansammeln. Aus diesem Grund muss die Architektur stetig neben der Anwendung weiterentwickelt und angepasst werden.
Bei der Betrachtung verschiedener agiler Vorgehen wurde festgestellt, dass diese keine besonderen Empfehlungen für eine Architekturentwicklung machen. Dies ist allerdings auch nicht verwunderlich, weil die Vorgehen auf eine Feature-basierte Entwicklung setzten und meist keinen oder nur einen sehr geringen technischen Anteil besitzen.

Die Problematik wurde von unterschiedlichen Standpunkten aus untersucht. Zu Anfang wurde im Kapitel 3 betrachtet, wie die Architektur einer vorhandenen Anwendung wieder herausgearbeitet werden kann. Zu diesem Zweck wurden unterschiedliche Szenarien konstruiert, die Gegebenheiten bei der Weiterentwicklung einer veralteten Anwendung repräsentieren

können. Das ursprüngliche Ziel, eine Art von Architekturdokumentation generieren zu können, wurde schnell verworfen. Die Dokumentation sollte die Einarbeitung neuer Personen ermöglichen. Das automatische Extrahieren von Strukturen und Mustern hat sich als sehr komplex und fehleranfällig herausgestellt. Bei den generierten Artefakten fehlten außerdem Begründungen für die Art der Realisierung. Intentionen können nur durch bekannte Muster oder einer manuellen Beschreibung erkannt werden.
Ein weiterer untersuchter Punkt war das automatische Finden von möglichen Problemen. Die ursprüngliche Idee war das gezielte Finden von bekannten Fehlern. Dies ist mit den verfügbaren Tools allerdings nur eingeschränkt möglich. Erfolgreich können diese Werkzeuge allgemeine Problemstellen finden und potentielle Bugs aufdecken. Der Einsatz solcher Werkzeuge sollte nicht erst nach der Entwicklung stattfinden, sondern ein dauerhafter Bestandteil des Entwicklungsprozesses sein.

Im Kapitel 4 wurde das Problem von der anderen Seite betrachtet. Hier wurde untersucht, welche Ansätze in der Literatur vorhanden sind, um eine gezielte Architektur neben dem üblichem Entwicklungsprozess entstehen zu lassen. Hierbei wurde sehr viel Wert darauf gelegt, dass die Anpassungen die Vorteile der agilen Entwicklung nicht beeinträchtigen. Um eine Architektur gezielt entwickeln zu können müssen die Anforderungen an diese bekannt sein. Zu diesem Zweck wurde ein Vorgehen vorgestellt bei dem die Anforderungen gesammelt und mithilfe eines Graphen gegeneinander abgewägt werden können. Ein Abwägen der Anforderungen ist notwendig, da diese sich teilweise gegenseitig ausschließen oder negativ beeinflussen können.
Im nächsten Schritt wurde untersucht, wie die Rolle eines Softwarearchitekten bei der Entwicklung unterstützen kann. Es konnte festgestellt werden, dass die Rolle nicht mehr viel mit ihrer ursprünglichen Bedeutung gemeinsam hat. Statt eigenständig eine Architektur zu planen, ist es im agilen Kontext wichtig, dass diese schrittweise durch das gesamte Team erstellt wird.
Die Problematik der Dokumentation konnte nicht gelöst werden. Deswegen wurden zwei unterschiedliche Vorschläge zur Dokumentation vorgestellt. Diese hatten das Ziel, eine kurze und informative Dokumentation zu erstellen, welche einfach aktuell gehalten werden kann. Um das Ziel zu erreichen, hatten diese eine klare vordefinierte Struktur und eine begrenzte Länge der einzelnen Abschnitte.
Im Anschluss wurde ein konkretes Vorgehen vorgestellt, welches die Ermittlung der Funktionalitäten und ein Konzept für die Entwicklung einer flexiblen Architektur beinhaltet. Das gesamte Vorgehen wurden in den Scrum Prozess integriert.
Zusätzlich wurde ein weiterer Ansatz zur iterativen modellgetriebenen Entwicklung vorgestellt. Der Quellcode sollte dabei nicht wie bei Model-Driven Development üblich generiert werden.

Die Implementierung sollte von Hand geschehen, weil eine Generierung meist nicht vollständig möglich ist.

Das Kapitel 5 stellt den größten Teil dieser Arbeit dar. In diesem Kapitel wurde untersucht, wie in der Praxis gearbeitet wird. Zu diesem Zweck wurden acht Experteninterviews mit unterschiedlichen Personen aus unterschiedlichen Firmen durchgeführt. Zur Durchführung der Interviews wurden Themen gesammelt, welche in einem Interviewleitfaden zusammengefasst wurden.
Die Experteninterviews wurden anschließend zunächst einzeln ausgewertet indem Besonderheiten hervorgehoben wurden. Außerdem wurde betrachtet, welche der in den vorherigen Kapiteln vorgestellten theoretischen Aspekte in der Praxis eingesetzt werden.
Im nächsten Schritt wurden die Themen in den Interviews miteinander verglichen. Zu diesem Zweck ist der Projektkontext, die aktuellen Diskussionsmöglichkeiten, die Aktivitäten während der Entwicklung, die Rolle des Architekten und die Erstellung der Dokumentation in den einzelnen Teams miteinander verglichen worden.
Anschließend wurde ein Vorschlag für ein mögliches Vorgehen auf Basis von Scrum mit integrierten Architekturdebatten vorgestellt. In dieses Vorgehen ist ein Teil der gewonnen Erkenntnisse aus den Interviews, wie auch aus den vorherigen Kapiteln, eingeflossen. Im Vorgehen sind die Aktivitäten während der Entwicklung, wie auch die Dokumentation, bewusst ausgelassen. Dies wurde entschieden, weil dies von den Teams selbst und abhängig vom Projektkontext entschieden werden sollte.

6.2 Fazit und Ausblick

Ein erfolgreicher Softwarelebenszyklus hängt von vielen verschiedenen Faktoren ab. Die Faktoren können vom Team, dem Unternehmen und der Art des Projekts beeinflusst werden. Ein allgemeingültiges Vorgehen, welches immer anzuwenden ist, ist deshalb nicht vorhanden. In der Theorie existieren Ansätze, die sich auf eine Architekturentwicklung innerhalb der agilen Entwicklung konzentrieren. Es ist noch offen, ob diese praxistauglich sind.

Im Laufe der Zeit wurden verschiedene Ansätze verfolgt, um die Softwarekrise zu bewältigen. Der agile Ansatz verspricht viele der Probleme aus den traditionellen Modellen zu lösen. Dabei wird die Langlebigkeit von Softwareanwendungen meist nicht betrachtet. Die Folgen für die Zukunft sind derzeit noch schwer abschätzbar.
Wie Architekturdebatten in die Entwicklung integriert werden können, ist noch nicht klar. Die Untersuchungen in der Praxis haben aber gezeigt, dass diese teilweise bereits eingeführt

werden. Ein weiterer Dialog zwischen der Theorie und der Praxis ist notwendig, um durch die praktischen Erfahrungen Ansätze für neue Theorien zu sammeln. Weitere Forschungsarbeiten sind hierzu erforderlich.
Diese Arbeit stellt keine universelle Lösung dar. Stattdessen ist sie eher eine kritische Betrachtung der agilen Entwicklungsvorgehen. Sie kann Hinweise liefern, was zur Architekturentwicklung gemacht werden kann. Außerdem macht sie deutlich, dass weitere Forschungsarbeit erforderlich ist. Eine Aufarbeitung und Erweiterung der Theorien zur agilen Entwicklung ist notwendig.

Literaturverzeichnis

[1] Abrahamsson, Pekka ; Babar, M.A. ; Kruchten, Philippe: Agility and Architecture: Can They Coexist? In: *IEEE Software* 27 (2010), mar, Nr. 2, S. 16–22. – URL http://ieeexplore.ieee.org/document/5420791/?arnumber=5420791. – ISBN 0740-7459 VO - 27

[2] Aldrich, Jonathan: Using Types to Enforce Architectural Structure. In: *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, IEEE, feb 2008, S. 211–220. – URL http://ieeexplore.ieee.org/document/4459159/?arnumber=4459159. – ISBN 978-0-7695-3092-5

[3] Alexander, Bogner ; Beate, Littig ; Wolfgang, Menz: *Interviews mit Experten*. Springer, 2014. – URL http://link.springer.com/book/10.1007/978-3-531-19416-5. – ISBN 978-3-531-19415-8

[4] Ali Babar, Muhammad: *Making Software Architecture and Agile Approaches Work Together: Foundations and Approaches*. Elsevier Inc., 2013. – 1–22 S. – URL http://dx.doi.org/10.1016/B978-0-12-407772-0.00001-0. – ISBN 9780124077720

[5] Ambler, Scott W.: Agile Model Driven Development (AMDD). In: *Xootic Magazine* (2007), S. 13–21. – URL http://www.agilemodeling.com/essays/amdd.htm. ISBN 9780511584077

[6] Ambler, Scott W.: *The Architecture Owner Role: How Architects Fit in on Agile Teams*. 2012. – URL http://www.agilemodeling.com/essays/architectureOwner.htm. – Zugriffsdatum: 2016-08-09

[7] Arcelli Fontana, Francesca ; Zanoni, Marco: A tool for design pattern detection and software architecture reconstruction. In: *Information Sciences* 181 (2011), apr, Nr. 7, S. 1306–1324. – URL http://linkinghub.elsevier.com/retrieve/pii/S0020025510005955. – ISBN 00200255 (ISSN)

[8] Babar, M a.: An exploratory study of architectural practices and challenges in using agile software development approaches. In: *2009 Joint Working IEEEIFIP Conference on Software Architecture European Conference on Software Architecture* (2009), S. 81–90. – URL http://ieeexplore.ieee.org/document/5290794/. ISBN 9781424449859

[9] Babar, Muhammad A. ; Brown, Alan W. ; Mistrik, Ivan: *Agile Software Architecture*. Elsevier, 2014. – 315–333 S. – URL http://www.sciencedirect.com/science/article/pii/B9780124077720000125. – ISBN 9780124077720

[10] Bandi, Ajay ; Williams, Byron J. ; Allen, Edward B.: Empirical evidence of code decay: A systematic mapping study. In: *2013 20th Working Conference on Reverse Engineering (WCRE)*, IEEE, oct 2013, S. 341–350. – URL http://ieeexplore.ieee.org/document/6671309/?arnumber=6671309. – ISBN 978-1-4799-2931-3

[11] Bertolino, Antonia ; Inverardi, Paola ; Muccini, Henry: Software architecture-based analysis and testing: a look into achievements and future challenges. In: *Computing* 95 (2013), aug, Nr. 8, S. 633–648. – URL http://link.springer.com/10.1007/s00607-013-0338-9. – ISSN 0010-485X

[12] Booch, Grady: The Accidental Architecture. In: *IEEE Software* 23 (2006), may, Nr. 3, S. 9–11. – URL http://ieeexplore.ieee.org/document/1628932/?arnumber=1628932. – ISBN 07407459

[13] Bosch, Jan: Architecture in the Age of Compositionality. In: Intergovernmental Panel on Climate Change (Hrsg.): *Climate Change 2013 - The Physical Science Basis* Bd. 53. Cambridge : Cambridge University Press, 2010, S. 1–4. – URL http://link.springer.com/10.1007/978-3-642-15114-9_1. – ISBN 9788578110796

[14] Brown, Nanette ; Ozkaya, Ipek ; Sangwan, Raghvinder ; Seaman, Carolyn ; Sullivan, Kevin ; Zazworka, Nico ; Cai, Yuanfang ; Guo, Yuepu ; Kazman, Rick ; Kim, Miryung ; Kruchten, Philippe ; Lim, Erin ; MacCormack, Alan ; Nord, Robert: Managing technical debt in software-reliant systems. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*. New York, New York, USA : ACM Press, 2010, S. 47. – URL http://portal.acm.org/citation.cfm?doid=1882362.1882373. – ISBN 9781450304276

[15] Buchgeher, Georg ; Weinreich, Rainer: *Chapter 7 - Continuous Software Architecture Analysis*. Elsevier Inc., 2014. – 161–188 S. – URL http://www.sciencedirect.

com/science/article/pii/B978012407772000006X. – ISBN 978-0-12-407772-0

[16] Buddha, Agile: *Agile/Scrum Foundation Training and Coaching Workshop*. 2012. – URL http://www.agilebuddha.com/wp-content/uploads/2012/04/scrum-methodology.gif. – Zugriffsdatum: 2016-05-17

[17] Buschmann, Frank: Gardening Your Architecture, Part 1: Refactoring. In: *IEEE Software* 28 (2011), jul, Nr. 4, S. 92–94. – URL http://ieeexplore.ieee.org/document/5929528/?arnumber=5929528. – ISBN 0740-7459

[18] Cataldo, Marcelo ; Herbsleb, James D. ; Carley, Kathleen M.: Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In: *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '08* (2008), S. 2 – 11. ISBN 9781595939715

[19] Charette, R.N.: Why software fails [software failure. In: *IEEE Spectrum* 42 (2005), sep, Nr. 9, S. 42–49. – URL http://ieeexplore.ieee.org/document/1502528/?arnumber=1502528. – ISSN 0018-9235

[20] Chen, Lianping ; Babar, Muhammad A.: Towards an Evidence-Based Understanding of Emergence of Architecture through Continuous Refactoring in Agile Software Development. In: *2014 IEEE/IFIP Conference on Software Architecture*, IEEE, apr 2014, S. 195–204. – URL http://ieeexplore.ieee.org/document/6827119/?arnumber=6827119. – ISBN 978-1-4799-3412-6

[21] Chung, Lawrence ; do Prado Leite, Julio Cesar S.: On Non-Functional Requirements in Software Engineering. In: *Conceptual modeling: Foundations and ...*. URL http://link.springer.com/chapter/10.1007%2F978-3-642-02463-4_19?LI=true, 2009, S. 363–379. – ISBN 9783642024627

[22] Cleland-Huang, Jane ; Czauderna, Adam ; Mirakhorli, Mehdi: Driving Architectural Design and Preservation from a Persona Perspective in Agile Projects. In: *Agile Software Architecture*. Elsevier, 2014, S. 83–111. – URL http://www.sciencedirect.com/science/article/pii/B9780124077720000289. – ISBN 9780124077720

[23] Cleland-Huang, Jane ; Mirakhorli, Mehdi ; Czauderna, Adam ; Wieloch, Mateusz: Decision-Centric Traceability of architectural concerns. In: *2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, IEEE, may 2013, S. 5–11. – URL http://ieeexplore.ieee.org/document/6620147/?arnumber=6620147. – ISBN 978-1-4799-0495-2

[24] Cohen, Tal ; Gil, Joseph (. ; Maman, Itay: JTL: the Java tools language. In: *ACM SIGPLAN Notices* 41 (2006), oct, Nr. 10, S. 89. – URL http://portal.acm.org/citation.cfm?doid=1167515.1167481. – ISSN 03621340

[25] Conway, M E.: How do committees invent. In: *Datamation* 14 (1968), Nr. 4, S. 28–31

[26] Coplien, James O. ; Reenskaug, Trygve: *The DCI Paradigm: Taking Object Orientation into the Architecture World*. Elsevier Inc., 2013. – 25–62 S. – URL http://dx.doi.org/10.1016/B978-0-12-407772-0.00002-2. – ISBN 9780124077720

[27] Cunningham, Ward: The WyCash portfolio management system. In: *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum) - OOPSLA '92*. New York, New York, USA : ACM Press, 1992, S. 29–30. – URL http://portal.acm.org/citation.cfm?doid=157709.157715. – ISBN 0897916107

[28] Dalgarno, Mark: When Good Architecture Goes Bad. In: *Methods & Tools - Spring 2009* (2009). – URL http://www.methodsandtools.com/archive/archive.php?id=85

[29] De Silva, Mahesh ; Perera, Indika: Preventing software architecture erosion through static architecture conformance checking. In: *2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS)*, IEEE, dec 2015, S. 43–48. – URL http://ieeexplore.ieee.org/document/7398983/?arnumber=7398983. – ISBN 978-1-5090-1741-6

[30] Detten, Markus von ; Becker, Steffen: Combining clustering and pattern detection for the reengineering of component-based software systems. In: *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS - QoSA-ISARCS '11*. New York, New York, USA : ACM Press, 2011, S. 23. – URL http://dl.acm.org/citation.cfm?doid=2000259.2000265. – ISBN 9781450307246

[31] Di Stefano, Marco: *Code Refactoring vs Architecture Refactoring | Refactoring Ideas*. 2013. – URL http://www.refactoringideas.com/code-refactoring-versus-architecture-refactoring/

[32] Domah, D ; Mitropoulos, F J.: The NERV methodology: A lightweight process for addressing non-functional requirements in agile software development. In: *SoutheastCon 2015* (2015), S. 1–7. – ISBN 9781467373005

[33] Dörnenburg, Erik: *Dependency Structure Matrix*. 2010. – URL https://erik.doernenburg.com/assets/attachments/2010/04/DSM2.png. – Zugriffsdatum: 2016-08-06

[34] Dr. Fleig, Jürgen: *Mitarbeiter an Entscheidungen beteiligen?* 2015. – URL http://www.business-wissen.de/artikel/entscheidungsfindung-mitarbeiter-an-entscheidungen-beteiligen/. – Zugriffsdatum: 2016-08-22

[35] Dragomir, Ana ; Harun, M. F. ; Lichter, Horst: On bridging the gap between practice and vision for software architecture reconstruction and evolution. In: *Proceedings of the First International Conference on Dependable and Secure Cloud Computing Architecture - DASCCA '14*. New York, New York, USA : ACM Press, 2014, S. 1–4. – URL http://dl.acm.org/citation.cfm?id=2578235http://dl.acm.org/citation.cfm?doid=2578128.2578235. – ISBN 9781450325233

[36] Ducasse, Stephane ; Pollet, Damien: Software architecture reconstruction: A process-oriented taxonomy. In: *IEEE Transactions on Software Engineering* 35 (2009), Nr. 4, S. 573–591. – ISBN 0098-5589 VO - 35

[37] Eloranta, Veli-Pekka ; Koskimies, Kai: *Lightweight Architecture Knowledge Management for Agile Software Development*. Elsevier Inc., 2014. – 189–213 S. – URL http://www.sciencedirect.com/science/article/pii/B9780124077720000071. – ISBN 978-0-12-407772-0

[38] Europäisches Parlament: *Konsens, Konsensverfahren, Konsensprinzip, Entscheidung im Konsens*. 2009. – URL http://www.europarl.europa.eu/brussels/website/media/Definitionen/Pdf/Konsens.pdf. – Zugriffsdatum: 2016-08-22

[39] Falessi, Davide ; Cantone, Giovanni ; Sarcia, Salvatore A. ; Calavaro, Giuseppe ; Subiaco, Paolo ; D'Amore, Cristiana: Peaceful coexistence: agile developer perspectives on software architec- ture. (2010), Nr. April 2010, S. 2010

[40] Farid, Weam M. ; Mitropoulos, Frank J.: NORMATIC: A visual tool for modeling Non-Functional Requirements in agile processes. In: *2012 Proceedings of IEEE Southeastcon*, IEEE, mar 2012, S. 1–8. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6196989. – ISBN 978-1-4673-1375-9

[41] Fausten, Leon: Softwarearchitektur und Agile Entwicklung - Experimentaufbau zur toolgestutzten Architekturrekonstruktion . (2016). – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2016-proj/fausten.pdf

[42] Fowler, Martin: *The New Methodology*. 2005. – URL http://www.martinfowler.com/articles/newMethodology.html. – Zugriffsdatum: 2016-06-18

[43] Fowler, Martin ; Highsmith, Jim: The agile manifesto. In: *Software Development* 9 (2001), S. 28–35. – URL http://www.pmp-projects.org/Agile-Manifesto.pdf. – ISBN 1070-8588

[44] France, Robert ; Rumpe, Bernhard: Model-driven Development of Complex Software: A Research Roadmap. In: *Future of Software Engineering (FOSE)* (2007), S. 37–54. – URL http://dx.doi.org/10.1109/FOSE.2007.14. – ISBN 0769528295

[45] Galster, Matthias ; Avgeriou, Paris: Supporting Variability Through Agility to Achieve Adaptable Architectures. In: *Agile Software Architecture*. Elsevier, 2014, S. 139–159. – URL http://www.sciencedirect.com/science/article/pii/B9780124077720000058. – ISBN 9780124077720

[46] Gleirscher, Mario ; Golubitskiy, Dmitriy ; Irlbeck, Maximilian ; Wagner, Stefan: On the Benefit of Automated Static Analysis for Small and Medium-Sized Software Enterprises. In: *Lecture Notes in Business Information Processing* Bd. 94 LNBIP. URL http://link.springer.com/10.1007/978-3-642-27213-4_3, 2012, S. 14–38. – ISBN 9783642272127

[47] Grundy, J. ; Hosking, J.: High-level static and dynamic visualisation of software architectures. In: *Proceeding 2000 IEEE International Symposium on Visual Languages*, IEEE Comput. Soc, 2000, S. 5–12. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=874344. – ISBN 0-7695-0840-5

[48] Hadar, Irit ; Sherman, Sofia ; Hadar, Ethan ; Harrison, John J.: Less is more: Architecture documentation for agile development. In: *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, IEEE, may 2013, S. 121–124. – URL http://ieeexplore.ieee.org/document/6614746/?arnumber=6614746. – ISBN 978-1-4673-6290-0

[49] Hofmeister, Christine ; Kruchten, Philippe ; Nord, Robert L. ; Obbink, Henk ; Ran, Alexander ; America, Pierre: A general model of software architecture design derived from five industrial approaches. In: *Journal of Systems and Software* 80 (2007), jan, Nr. 1, S. 106–126. – URL http://www.sciencedirect.com/science/article/pii/S0164121206001634. – ISSN 01641212

[50] Hutchinson, John ; Whittle, Jon ; Rouncefield, Mark: Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. In: *Science of Computer Programming* 89 (2014), Nr. PART B, S. 144–161. – URL http://dx.doi.org/10.1016/j.scico.2013.03.017. – ISBN 0167-6423

[51] Ihme, Tuomas ; Abrahamsson, Pekka: Agile architecting: The use of architectural patterns in mobile java applications. In: *International Journal of Agile Manufacturing* Bd. 8, 2005, S. 97–112. – ISSN 15362639

[52] Jansen, A. ; Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, IEEE, 2005, S. 109–120. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1620096. – ISBN 0-7695-2548-2

[53] Kruchten, Philippe: Software architecture and agile software development: a clash of two cultures? In: *2010 ACM/IEEE 32nd International Conference on Software Engineering* 2 (2010), S. 497–498. – ISBN 978-1-60558-719-6

[54] Lilienthal, Carola: *Langlebige Software-Architekturen*. dpunkt.verlag, 2016. – ISBN 978-3-86490-292-5

[55] Lilienthal, Carola: *Softwarearchitektur*. 2016. – URL https://www.wps.de/themen/softwarearchitektur/. – Zugriffsdatum: 2016-05-08

[56] Madison, James: Agile Architecture Interactions. In: *IEEE Software* 27 (2010), mar, Nr. 2, S. 41–48. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5420794. – ISBN 0740-7459 VO - PP

[57] Marić, Mirjana ; Tumbas, Pere: The Role of the Software Architect in Agile Development Processes. In: *International Journal of Strategic Management and Decision Support System in Strategic Management* 21 (2016), Nr. 1

[58] Martin, Robert C.: Design Principles and Design Patterns. (2000). – URL http://www.cvc.uab.es/shared/teach/a21291/temes/object_oriented_design/materials_adicionals/principles_and_patterns.pdf

[59] Mayring, Philipp: Qualitative Inhaltsanalyse. In: *Forum Qualitative Sozialforschung* 1 (2000), Nr. 2, S. Art. 20. – URL http://www.qualitative-research.net/fqs/. – ISBN 9783531920528

[60] Meyer, Bertrand: *Agile!: The good, the hype and the ugly.* 2014. – 1–170 S. – ISBN 9783319051550

[61] Murphy, Gail C. ; Notkin, David ; Sullivan, Kevin J.: Software reflexion models: Bridging the gap between design and implementation. In: *IEEE Transactions on Software Engineering* 27 (2001), Nr. 4, S. 364–380. – ISBN 0897917162

[62] Naur, P. ; Randell, B. ; Committee, Nato S.: Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. In: *NATO Software Engineering Conference* (1968), Nr. October 1968, S. 231. – URL http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html. – ISBN 0073655783

[63] Nord, Robert L. ; Ozkaya, Ipek ; Kruchten, Philippe: Agile in Distress: Architecture to the Rescue. In: *Agile Methods. Large-Scale Development, ...*. URL http://link.springer.com/chapter/10.1007%2F978-3-319-14358-3_5, 2014, S. 43–57. – ISBN 978-3-319-14357-6

[64] Oestereich, Bernd: *Verbunden im Konsent: die Prinzipien der soziokratischen Kreisorganisation.* 2015. – URL http://next-u.de/2015/konsent-prinzipien-der-soziokratischen-kreisorganisation/. – Zugriffsdatum: 2016-08-22

[65] Paredes Julia, Anslow Craig, Maurer F.: Information Visualization for Agile Software Development Teams, URL http://conferences.computer.org/vissoft/2014/papers/6150a157.pdf, 2014

[66] Passos, Leonardo ; Terra, Ricardo ; Valente, Marco T. ; Diniz, Renato ; das Chagas Mendonca, Nabor: Static Architecture-Conformance Checking: An Illustrative Overview. In: *IEEE Software* 27 (2010), sep, Nr. 5, S. 82–89. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5204070. – ISSN 0740-7459

[67] Perez, Jennifer ; Diaz, Jessica ; Costa-Soria, Cristobal ; Garbajosa, Juan: Plastic Partial Components: A solution to support variability in architectural components. In: *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, IEEE, sep 2009, S. 221–230. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5290808. – ISBN 978-1-4244-4984-2

[68] Pérez, Jennifer ; Díaz, Jessica ; Garbajosa, Juan ; Yagüe, Agustín: Bridging User Stories and Software Architecture. In: *Agile Software Architecture*. Elsevier, 2014, S. 215–241. – URL http://www.sciencedirect.com/science/article/pii/B9780124077720000083. – ISBN 9780124077720

[69] Portillo-Rodríguez, Javier ; Vizcaíno, Aurora ; Piattini, Mario ; Beecham, Sarah: Using agents to manage Socio-Technical Congruence in a Global Software Engineering project. In: *Information Sciences* 264 (2014), S. 230–259. – ISSN 00200255

[70] Rost, Dominik ; Naab, Matthias ; Lima, Crescencio ; Von Flach Garcia Chavez, Christina: Software architecture documentation for developers: A survey. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 7957 LNCS, 2013, S. 72–88. – ISBN 9783642390302

[71] Schwaber, Ken ; Sutherland, Jeff: The Scrum Guide. In: *Scrum.Org and ScrumInc* (2013), Nr. July, S. 17. – URL http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf. – ISBN 9788578110796

[72] Silva, Lakshitha de ; Balasubramaniam, Dharini: Controlling software architecture erosion: A survey. In: *Journal of Systems and Software* 85 (2012), jan, Nr. 1, S. 132–151. – URL http://www.sciencedirect.com/science/article/pii/S0164121211002044. – ISBN 0164-1212

[73] Stal, Michael: Refactoring Software Architectures. In: *Agile Software Architecture*. Elsevier, 2014, S. 63–82. – URL http://www.sciencedirect.com/science/article/pii/B9780124077720000034. – ISBN 9780124077720

[74] Standish Group: CHAOS Report. URL https://www.projectsmart.co.uk/white-papers/chaos-report.pdf, 2014. – Forschungsbericht

[75] Terra, Ricardo ; Valente, Marco T. ; Czarnecki, Krzysztof ; Bigonha, Roberto S.: Recommending Refactorings to Reverse Software Architecture Erosion. In: *2012 16th European Conference on Software Maintenance and Reengineering*, IEEE, mar 2012, S. 335–340. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6178900. – ISBN 978-0-7695-4666-7

[76] Tom, Edith ; Aurum, Aybüke ; Vidgen, Richard: An exploration of technical debt. In: *Journal of Systems and Software* 86 (2013), jun, Nr. 6, S. 1498–1516. – URL http://www.sciencedirect.com/science/article/pii/S0164121213000022. – ISBN 0164-1212

[77] Tomayko, James E. ; Nord, Robert L. ; Wojcik, Rob: Integrating Software- Architecture-Centric Methods into Extreme Programming (XP). In: *Architecture* 81 (2004), Nr. September, S. 727–746. – URL http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tn036.pdf

[78] Urma, Raoul-Gabriel ; Mycroft, Alan: Programming language evolution via source code query languages. In: *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools - PLATEAU '12*. New York, New York, USA : ACM Press, 2012, S. 35. – URL http://dl.acm.org/citation.cfm?doid=2414721.2414728. – ISBN 9781450316316

[79] Van Heesch, U. ; Avgeriou, P. ; Hilliard, R.: A documentation framework for architecture decisions. In: *Journal of Systems and Software* 85 (2012), S. 795–820. – URL http://dl.acm.org/citation.cfm?id=2148467. – ISBN 0164-1212

[80] Ven, Jan S. van der ; Bosch, Jan: *Architecture Decisions: Who, How, and When?* Elsevier Inc., 2013. – 113–136 S. – URL http://dx.doi.org/10.1016/B978-0-12-407772-0.00004-6. – ISBN 9780124077720

[81] Version One: 10th State Of Agile Report. URL http://jicru.oxfordjournals.org/cgi/doi/10.1093/jicru/ndl025, dec 2016. – Forschungsbericht. – ISBN 9789279304095

[82] Waterman, Michael ; Noble, James ; Allan, George: How Much Up-Front? A Grounded theory of Agile Architecture. In: *2015 IEEE/ACM 37th IEEE In-*

ternational Conference on Software Engineering Bd. 1, IEEE, may 2015, S. 347–357. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7194587. – ISBN 978-1-4799-1934-5

[83] Weitzel, Balthasar ; Rost, Dominik ; Scheffe, Mathias: Sustaining Agility through Architecture: Experiences from a Joint Research and Development Laboratory. In: *2014 IEEE/IFIP Conference on Software Architecture*, IEEE, apr 2014, S. 53–56. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6827100. – ISBN 978-1-4799-3412-6

[84] Wells, Don: *XP - System Metaphor*. 1999. – URL http://www.extremeprogramming.org/rules/metaphor.html

[85] Woods, Eoin: Aligning Architecture Work with Agile Teams. In: *IEEE Software* 32 (2015), sep, Nr. 5, S. 24–26. – URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7217769. – ISSN 0740-7459

Interviewleitfaden zu Agilen Architekturdebatten

Rahmeninformationen

Datum: _____

Zeitraum _____
Von ... Bis

Firma: _____

Person: _____
Name und Rolle

Kontaktmöglichkeit: _____

Ort: _____

Audioaufnahme: _____

Ablauf

1. Vorstellung Teilnehmer
2. Vorstellung Thematik
3. Unternehmen und Organisation
4. Debatte
5. Architekt
6. Kommunikation und Modellierung
7. Review der Architektur
8. Dokumentation
9. Beurteilung des Vorgehens

Unternehmen und Organisation

1. Was wird, für wen entwickelt?

2. Wie viele Personen sind an der Entwicklung beteiligt? (inkl. Rolle/Aufgabe)

3. Wie sieht die Organisation der Entwicklung/Teams aus?

4. Wie findet die Entwicklung statt? (Basisarchitektur, iterative Weiterentwicklung)

Debatte

5. Zu welchen Momenten wird über die Architektur geredet? (Wann, Konzeptionspunkte, Wo, Wer)

6. Findet diese Debatte regelmäßig statt oder gibt es einen expliziten Auslöser? (Wie häufig?)

7. Auf welcher Basis werden Entscheidungen getroffen? (z.B. Tickets, aus dem Planning?)

Kommunikation und Modellierung

8. Auf welcher Ebene wird diskutiert? (abstrakte Muster, genauere Pattern, Codebasis)

9. Wie werden Lösungen in der Debatte erarbeitet? (gemeinsames Modellieren, Vorgaben, vorher ausgearbeitete Vorschläge)

10. Wie wird entwickelt, wenn keine Debatte vorhanden ist oder Unklarheiten existieren (z.B. bei unregelmäßigen Meetings)?

11. Wie wird bei Problemen mit der Architektur vorgegangen? (z.B. zwingende Verletzungen, nicht mehr Anforderungsgerecht, Krisenmanagement)

12. Existiert ein Projekt/Teamübergreifender Austausch?

13. Existiert eine Person, die das Meeting leitet?

Architekt

14. Wenn die Rolle nicht existiert, wer entwickelt die Architektur? (alle wird schnell zu keiner, Entscheidungsfindung)

15. Was für Aufgaben hat der Architekt?

16. Existieren mehrere Architekten, tauschen diese sich aus?

17. Wie sind diese in das Team integriert? (Rechte, Weisungsbefugnis)

18. Hat der Architekt weitere Aufgaben? (Entwickler, andere Teams)

19. Gibt es weitere Rollen, die eigentlich nicht aus Scrum stammen, aber aus Tradition übernommen wurden.

Review der Architektur

20. Werden getroffene Entscheidung nochmals überprüft? (Anforderungsänderungen o.ä.)

21. Wird der Ist-Stand der Architektur monitored? (Wann, Wie oft, Wer)

22. Wird eine (semi-)automatisierte Analyse durchgeführt?

Dokumentation

23. Was wird festgehalten? (Lösung, Begründung, Wie und Wo)

24. Wie und wo werden diese Informationen festgehalten?

25. Existiert eine allgemeine Übersicht über die gesamte Architektur? Wie wird diese aktuell gehalten?

26. Wie kann sich ein neues Teammitglied am besten in das Projekt einarbeiten?

Beurteilung des Vorgehens

27. Wird das aktuelle Vorgehen weiterentwickelt? (ständige Optimierung oder machen wir schon immer so, Wie sieht das Ziel aus?)

28. Wie häufig werden Ausnahmen im Vorgehen gemacht? (z.B. Ausfall von Debatten, Bewusste Verletzung von Richtlinien)

29. Sehen Sie Schwachstellen im aktuellen Verfahren, wo treten trotzdem noch Probleme auf?

30. Was würden Sie gerne im aktuellen Verfahren ändern?

Unternehmen A

Interviewort: Im Unternehmen

Interviewlänge: 63 Minuten

Entwicklungsart: Produktentwicklung für Endkunden und Unternehmen als Webanwendung

Entwicklungsvorgehen: Kanban/Scrum

Teams: 7 Teams mit insgesamt circa 20 Entwicklern

Unternehmensbeschreibung

Das Unternehmen A entwickelt ein Endprodukt, direkt für den privaten, wie auch professionellen Kunden. Im ersten Quartal 2016 haben insgesamt circa 10,6 Millionen Nutzer das Produkt verwendet. Davon haben 904 000 Nutzer für den Dienst bezahlt[1]. Für das Produkt werden neben Mobile-Apps für IOS, Android und Windows Phone eine Webseite, einige APIs und ein separates Backend entwickelt. Die Entwicklung innerhalb des Unternehmens ist abhängig von den Teams und deren Tätigkeitsbereich.

Als Interviewpartner diente ein Entwickler für einen speziellen Bereich der iOS App. Die Person ist seit einem Jahr und einem Monat bei dem Unternehmen tätig. Das Unternehmen hat sehr viele unterschiedliche Bereiche, welche ebenfalls unterschiedlich bei der Entwicklung vorgehen. Aus diesem Grund beschreibt das Interview die Arbeitsweise des konkreten Teams.

Interviewbericht

Das Team des Interviewpartners besteht aus zwei iOS Entwicklern, zwei Android Entwickler, zwei Designern, einem Tester, einem Project Owner und einem Scrum Master. Von den Designer ist eine Person für das visuelle und eine für die User Experience zuständig (z.B. Menüführung). In dem besuchtem Team sind keine Backendentwickler vorhanden. Wenn bei der Entwicklung Backendentwickler benötigt werden, kommen diese aus einem anderen Team dazu. Dies kann zu einem häufigen Wechsel im Team führen. Bei anderen Teams kann dies vollständig unterschiedlich ablaufen. Bei diesen sind meist ebenfalls Backendentwickler ein Teil des Teams. Insgesamt existieren 7 iOS Teams mit circa 20 Entwicklern. Bei der aktuellen Entwicklung sind die iOS und die Android Seite mit unterschiedlichen Aufgaben beschäftigt. Aus diesem Grund werden zwei unterschiedliche Entwicklungsvorgehen eingesetzt. Das iOS Team arbeitet nach Scrum, das Android Team nach Kanban. Da der Interviewpartner aus dem iOS Scrum Team stammt, konzentriert sich die Beschreibung des Vorgehens hauptsächlich auf diesen Bereich. Zum Teil wird der Android Bereich allerdings ebenfalls mit einbezogen.

Die Entwicklung wird vollständig auf der Basis von Stories durchgeführt. Nachdem die Stories aufgeschrieben wurden, trifft sich das gesamte Team zum Grooming. Hier geht es darum,

[1] http://de.statista.com/, Abgerufen: 20.08.2016

eventuell vorhandene Epics in Stories umzuwandeln, Beschreibungen der User Stories zu verbessern und Schätzungen durchzuführen. Durch das Grooming kann es nochmals zu Nachforschungen zu einzelnen Themen kommen. Nach dem Grooming folgt, am Anfang jedes Sprints, das Sprint Planning. Dort werden anhand der aktuellen Velocity die Stories für den nächsten Sprint ausgewählt. Hierbei findet ein Herunterbrechen der Stories in Tasks statt. Der anschließende Sprint dauert zwei Wochen. Bei Abschluss einer Story wird ein Pull Request gestellt. Dieser wird in zwei Schritten gereviewt. Im ersten Schritt wird das Review von einem anderen Teammitglied durchgeführt. Wenn dieses sagt, dass alles in Ordnung sei, wird ein zweites Review von einer Person aus einem anderen Team durchgeführt. Die zweite Person wird zufällig ausgewählt. Zu diesem Zweck existiert ein zusätzliches Tool, welches die Person zufällig bestimmt. Dies hat den Zweck, den Wissenstransfer zwischen den Teams zu unterstützen und ein Gefühl dafür zu entwickeln, woran die anderen Teams arbeiten. Bei der Einführung solcher Vorgehen sind allerdings immer Diskussionen mit dem Management notwendig. Ew wird zusätzliche Zeit benötigt, die nicht in die direkte Entwicklung von Funktionalitäten einfließt. Wenn beide Reviews erfolgreich sind, führt der Tester eine Qualitätssicherung aus und prüft, ob alles zufriedenstellend entwickelt wurde. Erst wenn dieser Schritt erfolgreich abgeschlossen ist, gilt die Story als gelöst. Während eines Reviews ist es allerdings schwer über die Architektur zu reden. Diese Art von Diskussion betrifft meist einen Großteil des neu entwickelten Codes. Vorschläge würden meist einen Großteil der Story betreffen.
Ein Austausch der Entwickler findet im täglich stattfindendem Daily-Scrum Meeting statt. Zweimal die Woche wird dieses Meeting mit allen iOS Teams gemeinsam durchgeführt.
Wenn keine Debatte zur Entwicklung stattfindet wird sich häufig an anderem, bereits vorhandenem Code orientiert. Hier besteht das Problem, dass dieser Code veraltet ist und die ursprünglichen Entwickler dies zum aktuellen Zeitpunkt anders machen würde. Dadurch wird schnell ein schlechter Stil erzeugt.

Zur Entwicklung werden zum Teil vorgegebene Designpattern verwendet. Dies kann z.B. in Form von verschiedenen vorgefertigten UI-Komponenten sein, welche über Factory-Methoden eingebunden und erzeugt werden müssen. Andere Stellen, wie z.B. das Menü, sind fest vorgegeben. Dieses wurde vor langer Zeit entwickelt und steht dadurch fest. Die feste Struktur kann bei der täglichen Entwicklung Probleme verursachen. Ein erläutertes Beispiel ist die Berechnung der Anzahl von Benachrichtigungen. Da Benachrichtigungen von verschiedenen Stellen kommen können, ist die Berechnung auf viele unterschiedliche Komponenten verteilt. Diese sollten eigentlich unabhängig voneinander sein. Außerdem wird die Berechnung im Client durchgeführt. Der Wunsch ist aber eine Berechnung auf dem Server, weil dies ebenfalls für die Webseite und die anderen Apps benötigt wird. Dadurch kann sichergestellt werden, dass die Berechnung konsistent ist.

Zum teamübergreifenden Austausch findet einmal die Woche ein *Community Meeting* statt. Dieses ist auf eine Stunde begrenzt. An dem Meeting nehmen alle iOS Entwickler Teil. Es werden die im Anschluss vorgestellten *Approaches* und *Proposals* vorgestellt. Außerdem können allgemeine Themen, wie z.B. ein Update auf eine andere iOS Version und die Beendigung des Supports für eine alte Version, besprochen werden. Wenn genügend Zeit vorhanden ist, können zusätzlich Vorträge, die alle Entwickler betreffen, gehalten werden. Hierbei kann es sich um Themen handeln die viele Komponenten der Anwendung betreffen. Eine Umstrukturierung der Basiskomponenten ist ein Beispiel dafür. Diese Thematik betrifft alle Entwickler.

Um Probleme und Ideen zu besprechen, können gibt es unterschiedliche Wege. Diese sind freiwillig. Ein explizites Vorgehen ist nicht vorhanden.
Kleine Probleme werden, wenn möglich direkt im Rahmen der Story erledigt. Zusätzlich können jederzeit anderer Entwickler, des eigenen Teams oder aus einem anderem Team, um Rat gefragt werden.
Bei mittelgroßen Problemen, die nur einen kleinen Teil der Anwendung betreffen, kann ein

Design Approach (Lösungsvorschlag) erstellt werden. Dieser wird nur erzeugt, wenn die betroffene Person einen Wunsch nach Feedback hat. Hierzu wird in dem SCM-System (Github) ein Issue angelegt. Inhalt dieses Issues ist das betroffene Ticket und die Umsetzungsidee. Im Rahmen dieses Issues können Meinungen von anderen Entwicklern gesammelt werden. Dies ist ein teamübergreifender Prozess. Alle iOS Teams können Kommentare zu dem Issue schreiben. Es sind alle Teams beteiligt, weil auf einer gemeinsamen Codebasis gearbeitet wird. Der Austausch passiert hier hauptsächlich schriftlich. Bei Interesse einzelner Entwickler, kann es aber zu einem Treffen kommen, welches eine Diskussion vereinfacht. Die Ergebnisse des Treffen werden nicht schriftlich festgehalten. Wenn bei den Diskussionen (schriftlich und mündlich) festgestellt wird, dass allgemeiner Diskussionsbedarf besteht, kann es zu einem spontanen Treffen im Team kommen. Dort wird die Thematik nochmals erläutert und alternative Lösungsvorschläge werden gesammelt.
Bei einer größeren Änderungsnotwendigkeit kann ein *Proposal* (Vorschlag) erstellt werden. Dies ist in erster Linie für Prozessänderungen gedacht, kann aber auch für Architekturdebatten verwendet werden. Wenn ein Entwickler ein Problem hat, kann er einen Änderungsvorschlag als schriftliches Proposal einreichen. Dieses Proposal wird im nächsten Community Meeting vorgestellt. Anschließend gibt es eine *Review-Week*. Dort haben alle eine Woche Zeit Kommentare zu schreiben, Kritik zu äußern und Fragen zu stellen. Der Vorschlag wird akzeptiert, wenn es keine Gegenstimmen gibt. Dieses Vorgehen wurde erst vor drei Monaten eingeführt und ist dadurch noch relativ neu. Es muss sich erst vollständig etablieren. Wenn es für andere Teams erforderlich ist Änderungen wegen dem Proposal durchzuführen, wird ein Roll-Out Plan mitgeliefert. Dies ermöglicht die anderen Teams die Änderungen entsprechend einzuplanen und durchzuführen.

Wenn kein Lösungsvorschlag, z.B. durch vorhandene, ähnliche Probleme oder den Diskussionen existiert, haben die Entwickler freie Hand zu entscheiden. Dazu können sie sich im Team zusammensetzen um darüber zu sprechen. Dies kann z.B. zwischen dem Sprint Planning und dem Task-Breakdown stattfinden. Dort kann besprochen werden, was gemacht werden muss und wo welche Kommunikation notwendig ist. Es kann vorkommen, dass diese Diskussion bereits beim Grooming stattfindet, um die Stories besser schätzen zu können. Statt im normalen Grooming, mit allen Teammitgliedern wird meist ein zusätzliches *Tec-Grooming* angesetzt. Hier nehmen nur noch die Entwickler teil. Dieses Treffen wird spontan angesetzt und findet im Anschluss des normalen Grooming statt. Da dieses Team nur aus wenigen Entwicklern besteht ist dies ohne Probleme machbar.

Wenn Diskussionen stattfinden wird dies meistens mithilfe von Codebeispielen durchgeführt. Eine Diskussion über Pattern findet nur bei direkten Gesprächen statt. Zur Diskussion im Team stattfindet werden formlose UML-Diagramme eingesetzt. Dies heißt, dass nicht auf eine korrekte Syntax geachtet wird, sondern in erster Linie auf Verständlichkeit.

Bei der Feststellung von Problemen gibt es keinen festen Ansprechpartner oder Vorgehen. Um eine akzeptable Lösung zu erzeugen wird zunächst geprüft, ob das was gerade gebaut wird langfristig in der Anwendung bleibt oder ob dies nur ein Test ist, der nach einem speziellen Zeitraum wieder entfernt wird. Wenn es nur kurzfristig in der Anwendung verbleibt, wird meist ein Workaround verwendet. Die anderen Strukturen müssen nicht oder nur leicht angepasst werden. Wenn dies länger bleibt wird untersucht, wie die Kernstruktur geändert werden muss um das Problem zu lösen. Wenn sich die Änderungen im Rahmen halten wird dies im Zeitraum der aktuellen Story erledigt. Wenn die Änderung zu groß ist, wird ein *Technical-Debt Ticket* erstellt und die aktuelle Story mit einem Workaround gelöst. Im dem Technical-Debt Ticket, wird auf die auslösende Story verwiesen und der Workaround mit einer Verbesserungsidee beschrieben. Diese Tickets können ebenfalls für andere Teams erstellt werden, wenn dies eher deren Aufgabenbereich betrifft.

Die unterschiedlichen Vorgehen um Lösungen zu finden sind sehr individuell und müssen intuitiv

nach Gefühl gewählt werden. Keines von ihnen ist verpflichtend.

In dem besuchten Team existiert keine Architektenrolle. Dessen Aufgaben werden zum Teil von dem Plattform- und Framework-Team übernommen. Von ihnen wird die Basis für die Anwendung entwickelt. Diese leiten aber nicht die Architekturentwicklung, sondern machen stattdessen eher Vorgaben wie etwas entwickelt werden muss. Zusätzlich gibt es die Rolle des Lead Developers, der die Aufgaben übernehmen kann. Dessen Aufgabenschwerpunkt liegt allerdings auf der Kommunikation zwischen den Entwicklungs-, der Design- und der Produkt-Abteilungen. Diese Aufgabe ist weniger technisch.

Der aktuelle Stand der Anwendung wird durch ein Design Dokument, bzw. Katalog beschrieben. Hier sind in erster Linie fachliche Funktionalitäten zu erkennen, aber nicht die technische Umsetzung.
Ein Monitoring findet im Rahmen von automatisierten Jenkins Builds statt. Bei diesen werden z.B. Tests automatisiert. Für die Tests wurde eine Code Coverage eingeführt. Ein direktes Architekturmonitoring findet nicht statt. Für Probleme werden stattdessen Fehlerlogs verwendet. Im Team existiert durchaus der Wunsch so etwas einzuführen. Dies ist aber schwierig, weil es niemanden gibt der sich um so etwas kümmern kann. Ein weiteres Problem ist, dass das Management überzeugt werden muss.
Die Approaches und Proposals können als Dokumentation dienen. Vor Einführung dieser existierten diese Informationen nur in den Köpfen der Entwickler. Zusätzlich existiert eine inline Dokumentation innerhalb des Quellcodes. Im oben genannten Beispiel zur Berechnung der Benachrichtigungen existiert keine Möglichkeit ohne durchsuchen des Codes herauszufinden, welche Stellen betroffen sind. Eine schriftliche Dokumentation hängt alleine von der Motivation des Entwicklers ab. Die Einarbeitung neuer Mitarbeiter ist nur möglich, indem diese sich durch den Code arbeiten und versuchen Tickets zu lösen. Dies wird häufig mithilfe von Pair Programming durchgeführt. Das anschließende Review kann weitere Verbesserungsvorschläge liefern.

Unternehmen B

Interviewort: Im Unternehmen

Interviewlänge: 56 Minuten

Entwicklungsart: Produktentwicklung für den privaten Endanwender zur Vermittlung von Verträgen kommerzieller Kunden als Webanwendung.

Entwicklungsvorgehen: Scrum/Kanban

Teams: 5 Teams mit circa je 5 Mitgliedern

Unternehmensbeschreibung

Das Unternehmen B entwickelt eine Webplattform für Angebotsvergleiche. Es gilt als Startup und hat in einer kürzlich stattgefundenen Finanzierungsrunde eine Summe von mehr als 33 Millionen Euro erhalten[1]. Die agile Entwicklung befindet sich derzeit noch im Aufbau. Das endgültige Ziel wurde noch nicht vollständig erreicht. In einem vorab stattgefundenen Gespräch wurde über das Ziel der Entwicklungsumstellung gesprochen. Dies ist in Abbildung 1 dargestellt. Ziel ist es vollständig eigenständige Teams zu haben. Diese entwickeln spezielle Features und betreuen diese bis zu deren Lebensende. Die Eigenständigkeit der Teams soll soweit gehen, dass diese theoretisch in einzelne GmbH's umgewandelt werden könnten. Das bedeutet für alle Teammitglieder, dass sie vollständig verantwortlich sind und bei Problemen selbstständig Lösungen erarbeiten müssen. Dies befähigt sie flexiblere Entscheidungen zu treffen. Dadurch wird die Entwicklung allgemein flexibler und unabhängiger.

Weil die Teams ein gemeinsames Produkt entwickeln muss abgestimmt werden, wie die Kommunikation zwischen den entwickelten Features stattfinden soll. Dies soll in regelmäßigen Treffen geschehen. An diesem Treffen soll jeweils eine Person jedes Teams teilnehmen. Diese Person sollte für die Architektur verantwortlich sein. Bei dem Treffen wird die allgemeine, globale Architektur festgelegt. Außerdem werden die Kommunikationsschnittstellen abgesprochen. Die Schnittstellen sind die einzigen Vorgaben, die die Teams einhalten müssen.

Das Vorhaben ist noch nicht vollständig umgesetzt. Die Teams existieren bereits, arbeiten aber noch nicht vollständig eigenständig.

Vor circa einem Jahr wurde ein neuer CTO eingestellt. Dieser hat die Entwicklung auf eine agile Entwicklung umgestellt. Der Interviewpartner ist seit circa 3 Jahren im Unternehmen beschäftigt. Er ist der Lead Developer des Unternehmens und deswegen keinem Team fest zugeordnet. Über die Abläufe innerhalb der Teams hatte er keine genauen Kenntnisse.

Interviewbericht

Das System bestand früher aus zwei großen Monolithen. Ein Monolith wurde zu einem Makro-System umgebaut. Vor zwei Jahren wurde begonnen den zweiten in kleinere Subsysteme zu

[1] http://www.gruenderszene.de, Abgerufen: 03.10.2016

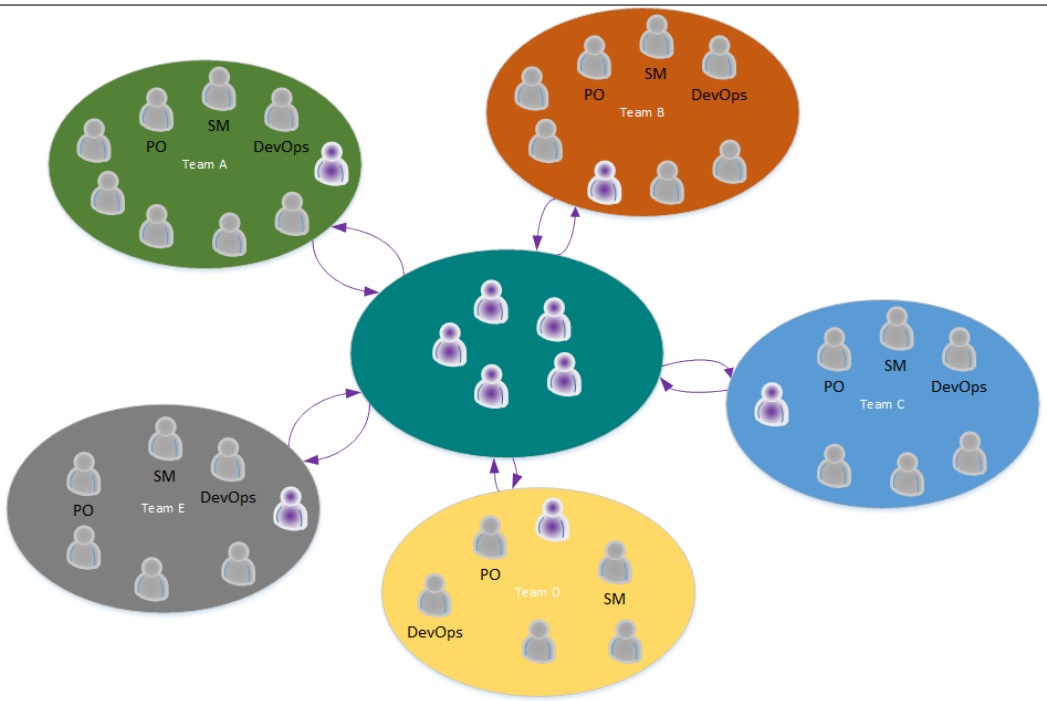


Figure 1: Agile Teams im Unternehmen B

unterteilen.

Nach der Umstellung des Entwicklungsvorgehen, vor circa neun Monaten, hat sich der Großteil der Teams für eine Entwicklung nach Scrum entschieden. Zwei der Teams habe sich stattdessen für Kanban entschieden. Vor der Umstellung existierten fünf Teams, mit je fünf Mitgliedern. Zusätzlich gab es eine Gruppe von fünf teamlosen Personen. Diese waren der CTO, der Lead Developer und ein paar Springer. In jedem Team war ein Teamleiter. Die PO-Rolle wurde von ihnen übernommen. Bei dem alten Vorgehen gab es klare Hierarchien. Ganz oben war der CTO, anschließend die Teamleiter und unterhalb die restlichen Mitarbeiter angesiedelt. Bei der Umstellung wurde die Teamleiterrolle abgeschafft. Die Personalverantwortung liegt nur noch beim CTO. Die restliche Arbeitsweise ähnelt sehr stark der vorherigen. Die Teams arbeiten meist unabhängig in ihren eigenen Projekten. Eine Änderung ist die Einführung der Scrum Prozesse, wie das Sprint Planning, die Daily Meetings, die Review-Meetings und die Retrospektiven. Zusätzlich wurden zwei agile Manager zum Coachen der agilen Entwicklung eingestellt. Ein neuer Flow-Manager kümmert sich darum, dass die anderen Arbeiten können. Er soll sich um die Impediments kümmern und die Kommunikation und das Miteinander aller Mitarbeiter über Teamgrenzen hinaus verbessern. Der Lead Developer ist in diesem Zusammenhang für alle technischen Belange zuständig.

Einen gesonderten Zeitpunkt um teamübergreifend eine Architektur-Diskussionen zu führen existiert nicht. Dies kann in dem wöchentlichem Treffen getan werden. Früher war dies die *Teamleiterroutine*. Hier haben nur die Teamleiter, der CTO und der Lead Developer teilgenommen. Inzwischen kann jeder an dem Meeting teilnehmen. Im Normalfall sind einzelne Vertreter des Teams anwesend. Hier wird besprochen, was in der kommenden Woche anliegt und was in der vergangenen Woche unerwartetes geschehen ist. Dieser Zeitpunkt ist für den Lead Developer wichtig, um herauszufinden was in den einzelnen Teams abläuft. An den gesonderten

Teammeetings nimmt er nicht teil. Hier kann er erfahren, wo Probleme sind und wo eventuell weitere Arbeit durch ihn notwendig ist. In dem Meeting wird hauptsächlich auf einer fachlichen Ebene diskutiert. Dies kann z.B. eine Diskussion über einen Ausfalls sein und wie dies in Zukunft verhindert werden kann. Die Diskussionsebene ist sehr abstrakt gehalten, damit sich nicht in Details verloren wird. Hier wird ebenfalls besprochen, wer in der nächsten Woche im Urlaub ist und welche neuen Mitarbeiter anfangen. Neue Funktionalitäten, die mehre Teams betreffen ,werden vorgestellt damit diese sich darauf einstellen können. Zu dem Treffen wird ein Dokument verfasst, welches für alle Mitarbeiter zugänglich gemacht wird.
Für die fachliche Seite gibt es ein eigenes Treffen, die *PO-Routine*. Eine Abstimmung beider Abteilungen geschieht nur durch individuelle Absprachen einzelner Kollegen.
Wie genau Absprachen in den Teams stattfinden, ist dem Interviewpartner unbekannt. Die meisten Teams machen allerdings jeden morgen ein Daily Meeting. Dort wird besprochen woran sie gerade arbeiten. Ob weitere Abstimmungen, neben den bereits in Scrum vorhanden, stattfinden ist unbekannt.
Circa drei Wochen nach dem Interview ist erstmalig ein *Offsite-Meeting* geplant. Hier nimmt der gesamte IT-Bereich und das gesamte Produktmanagement teil. In diesem Meeting sollen Visionen, Ziele und der Status Quo besprochen werden. Dies betrifft auch die Architektur. Es soll über den aktuellen Stand und die Weiterentwicklung geredet werden. Der Wunsch des Interviewpartners ist, solch ein Meeting regelmäßig Quartalsweise zu machen um mehr Absprachen untereinander treffen zu können.
Früher gab es alle zwei bis drei Wochen einen *Pizza-Mittwoch*. Hier konnten Kollegen zu einem Thema Vorträge halten. Durch solch einen Vortrag über Maschinelles-Lernen ist z.B. eine Gruppe entstanden, die sich alle drei Wochen trifft um über den Fortschritt zu sprechen. Der Pizza-Mittwoch fand im letzten Jahr allerdings sehr selten statt.

Wenn vor der Umstellung des Entwicklungsvorgehen Probleme aufgetreten sind und diese zwingend direkt im Laufe der aktuellen Entwicklung gelöst werden mussten, wurde dies im Rahmen der aktuellen Aufgabe erledigt. Wenn festgestellt wurde, dass jemand in die falsche Richtung entwickelt hat gab es zwei Möglichkeiten zu reagieren. Entweder wurde die Entwicklung gestoppt und nochmals anders angegangen oder vollständig für den nächsten Sprint zurückgestellt. Das aktuelle Vorgehen wird ähnlich hierzu sein, ist dem Interviewpartner aber nicht genau bekannt.

Nach der Entwicklung werden Pull Request gestellt und durch einen weiteren Entwickler gereviewt. Das Problem bei Code Reviews ist, dass bei wenigen Änderungen viel genauer geschaut wird, als bei vielen Änderungen. Dadurch können Probleme schnell übersehen werden. Bei Problemen während der Entwicklung kann es vorkommen, dass die Entwickler sich zum Pair Programming zusammensetzen.

Die Teams selbst sind weitestgehend unabhängig voneinander. Bei manchen Aufgaben ist allerdings trotzdem eine Abstimmung notwendig. Die Teams können die bereits vorhandenen und im Wiki-System dokumentierten Schnittstellen der anderen Teams verwenden. Wenn erforderliche Schnittstellen nicht vorhanden sind müssen Absprachen getroffen werden. Hierzu wird sich mit den entsprechenden Entwicklern zusammengesetzt.

Die explizite Rolle eines Architekten ist in dem Unternehmen nicht vorhanden. Der interviewte Lead Developer würde sich aber wünschen mehr als Berater für die Teams eingesetzt zu werden und nur noch Empfehlungen für die Teams auszusprechen. Derzeit ist dies nicht so. Aktuell übernimmt er viele DevOps Tätigkeiten und entwickelt ebenfalls mit. Allerdings können andere Kollegen bei Problemen auf ihn zukommen. Wenn dies der Fall ist, versucht er die Lösung in Zusammenarbeit mit den anderen Entwicklern zu erarbeiten. Früher hat er eigenständig Lösungen entwickelt. Dabei hat er festgestellt, dass dies nicht besonders hilfreich ist, weil der Lerneffekt für die anderen Entwickler sehr gering wird.
Bei der Einführung neuer Themen muss er die anderen Personen von seinem vorgeschlagenem Vorgehen überzeugen.

In Zukunft würde sich der Lead Developer lieber mit Lösungen für die gesamte Architektur beschäftigen. Dies ist möglich, weil er in den meisten Bereichen bereits mitgearbeitet hat und dadurch einen Überblick über die Komponenten und deren Funktionalitäten hat. Um sich mehr auf diese Aufgaben konzentrieren zu können, würde er gerne eine weitere Person für die DevOps Aufgaben einstellen.

Eine Dokumentation wird in einem Wiki erstellt. Einen festen Rahmen, was bei der Umsetzung dokumentiert wird, ist nicht vorhanden. Wenn etwas nicht ausreichend dokumentiert ist, ist ein Reverse-Engineering der Entscheidungen notwendig. Dies ist bereits vorgekommen.
Im Wiki wird z.B. vom Produktmanagement das fachliche Konzept festgehalten. Technische Konzepte werden nur nach Bedarf aufgeschrieben. Dies wird hauptsächlich bei größeren und komplexeren Konzepten gemacht, welche nicht direkt aus dem Code verständlich sind. Entscheidungen werden nur festgehalten, wenn diese wichtig sind, um das Konzept zu verstehen. Hierzu werden die Gründe und vorhandene Alternativen notiert.
Es existiert ein Schaubild mit den wichtigen Komponenten und einigen Prozessen als allgemeine Übersicht. Das Schaubild wurde seit circa einem Jahr nicht mehr aktualisiert. Der ursprüngliche CTO hat dies damals manuell gepflegt. Derzeit existiert eine Übersicht nur im Kopf der Entwickler.

Der aktuelle Stand wird mit *Travis CI*[2] und für statische Codeanalysen mit *Scrutinizer CI*[3] geprüft. Potentielle Bugs und problematische Codestellen können gefunden werden. Die Analysen bewerten den Code nach amerikanischen Schulnoten. Zum Teil nutzen die Entwickler diese Analysen nach dem Pull Requests um die Codequalität zu verbessern. Nicht alle Entwickler setzen dies ein. Richtlinie zu einer Mindestbewertung sind nicht vorhanden. Diese würden die Stimmung der Mitarbeiter verschlechtern, weil sie primär für das Tool schönen Code schreiben würden. Außerdem würde dies die Entwicklungsgeschwindigkeit verringern, weil unter Umständen eine Funktionalität nicht abgeschlossen werden kann, da der Buildserver diese nicht freigibt.
Eine Prüfung der gesamten Architektur, z.B. durch Conformance Checks wird nicht durchgeführt. Dies ist unter anderem darauf zurückzuführen, dass die Architektur nicht genau geplant ist. Es existieren keine Schichten oder ähnliches. Die Architektur ist so gewachsen, wie es zu dem Zeitpunkt am besten gepasst hat. Dadurch kommt es z.B. vor, dass an manchen Stellen viele unterschiedliche Konzepte zu finden sind. Diese wurden in einem kurzen Zeitabschnitt häufig geändert. Der alte Code wurde aber nicht auf das neue Konzept umgestellt. Um dies zu verbessern müssten alte Systeme abgeschaltet werden, dies wird derzeit nicht getan.

Um die Anwendung zu prüfen, werden funktionale, Unit- und End-to-End Tests entwickelt. Es wird nach den Kosten, die bei einem Ausfall entstehen, entschieden was getestet werden muss. Zum Teil sind seit langer Zeit bekannte Bugs vorhanden, die nicht ausgebessert werden, weil sie nicht in großem Ausmaß stören. Der Interviewpartner kennt kein so gut strukturiertes Projekt, bei dem nur durch Tests festgestellt werden kann, ob eine Anwendung nach Änderungen weiterhin einwandfrei funktioniert. Die einzelnen Komponenten sind meist so miteinander verzahnt, dass die Tests bei Änderungen ebenfalls angepasst werden müssen. Dies wird besser, je weiter die Tests vom eigentlichen Quellcode entfernt sind. Aus diesem Grund empfiehlt er nur so viel Energie in Tests zu stecken, wie gerade notwendig ist.

Er sieht es derzeit als Problem, dass die Entwickler keine Person über sich haben, die für sie verantwortlich ist. Es fühlt sich häufig niemand verantwortlich, weil keine konkrete Ansagen, wie etwas umgesetzt werden soll, gemacht werden. Für ihn wirkt es so, dass derzeit gegeneinander, statt miteinander gearbeitet wird. Beim Auftreten von Fehler ist häufig eine Beweisführung notwendig um die Ursache zu belegen. Dies dient nicht dazu einen Schuldigen zu finden, sondern

[2] https://travis-ci.org/, Abgerufen am 17.10.2016
[3] https://scrutinizer-ci.com/, Abgerufen am 17.10.2016

um herauszufinden wo das Problem gelöst werden muss. Er hofft, dass sich diese Einstellung mit der Zeit wieder legt.
Ein weiteres Problem ist, dass die Fachabteilungen für die Produktentwicklung und das Produktmanagement nicht wissen, wie die Architektur aussieht. Dadurch ist es bereits vorgekommen, dass zwei Teams unterschiedliche Funktionalitäten entwickelt haben, die sich gegenseitig ausgeschlossen haben. Dies hätte verhindert werden können, wenn in den entsprechenden Abteilungen der Stand bekannt gewesen wäre. Die Produktentwicklung sollte ein größeres Verständnis für das Gesamtsystem erlangen und nicht nur einen kleinen Bereich betrachten. Dadurch könnten viele Fragen an die Entwicklungsabteilung überflüssig werden. Einige Fragen könnten bereits von den Kollegen aus dem gleichem Büro beantwortetet werden.

Unternehmen C

Interviewort: Im Unternehmen

Interviewlänge: 69 Minuten

Entwicklungsart: Produktentwicklung für den internen Einsatz

Entwicklungsvorgehen: Kanban

Teams: 5 Teams mit 6 - 12 Mitgliedern

Unternehmensbeschreibung

Das Unternehmen C ist in der Logistikbranche tätig und hat in über 100 Ländern Niederlassungen. Externe Kunden existieren nicht. Die Entwicklung findet vollständig für interne Kunden statt.
Der Interviewpartner in Unternehmen C hat die Rolle eines Architekten. Er ist seit einem Jahr und einem Monat in dem Unternehmen tätig. Vor dem Interview fand ein Kennlerngespräch mit einem anderen Architekten statt. Bei diesem Treffen wurde bereits über deren Entwicklung geredet.

Interviewbericht

Die Entwicklung ist in verschiedene Teams nach Fachlichkeit getrennt. Die einzelnen Anwendungen werden dabei als *Self-Contained Systems* entwickelt. Dies bewirkt, dass die unterschiedlichen Anwendungen und Teams sehr unabhängig voneinander sind. Die Teams entwickeln die vollständige Anwendung, angefangen bei der Datenhaltung, bis hin zur GUI. Es existieren fünf Teams mit je zwischen 6 bis 12 Mitgliedern. Eine aufwendige Koordination der verschiedenen Teams ist weitestgehend irrelevant, da zwischen den einzelnen Anwendungen nicht viel Kommunikation notwendig ist. Falls eine Kommunikation der Anwendungen notwendig ist wird dies mithilfe von *Message-Queues* durchgeführt.

Als Basis zur Entwicklung dienen Stories. Die Entwickler nehmen sich zum arbeiten Stories aus dem Backlog. Eine Vorstellung neuer Stories findet einmal in der Woche oder wenn das Backlog leer läuft statt. Stories werden sowohl vom *Stream Owner*, *Analyst*, wie auch vom Architekten erstellt. Die Aufgaben des Stream Owners sind ähnlich zu denen des Product Owners. Die Stellung im Team ähnelt sich ebenfalls. Der Stream Owner hat allerdings ein paar weitere Aufgaben. Er organisiert und zu leitet z.B. die Reviews. Außerdem hat er viele Schnittstellen mit Entwicklertätigkeiten die der Product Owner nicht hat. Er testet ob die Akzeptanzkriterien der Stories erfüllt werden. Der Analyst stellt die Schnittstelle zur fachlichen Seite dar. Aus den Anforderungen der Fachabteilungen leitet er Stories ab.
Das Entwicklungsvorgehen ist eine Art von Kanban. Die Stories werden von oben nach unten umgesetzt. Schätzungen der Stories werden nicht durchgeführt. Direkt vor der Entwicklung wird ein *Task-Breakdown* gemacht. Hier werden die Stories durch Teamarbeit in einzelne Aufgaben unterteilt. Dadurch soll ein einheitliches Verständnis sichergestellt und Aspekte beachtet wer-

den, die Einzelperson eventuell übersehen hätten. Die einzelnen Aufgaben werden dadurch viel detaillierter betrachtet. Der Task-Breakdown wird im Normalfall in zweier Teams durchgeführt. Außerdem besprechen die Entwickler, wie die Story konkret realisiert werden kann. Dies betrifft die Architektur.

Bevor eine fertig implementierte Story in den Hauptentwicklungsstrang übernommen werden führt ein weiterer Entwickler ein Code Review durch. Dabei können erneut Probleme auffallen. Eine weitere Prüfung findet durch PMD-Prüfungen mithilfe automatisierter Jenkins-Builds statt.

Zur Diskussion werden keine Entwurfsmuster oder ähnliches verwendet. Diskussionen finden am Code statt. Beim Einsatz von Pattern besteht das Problem, dass deren Lösung häufig unnötigen Aufwand benötigt und einen *Overhead* erzeugt. Um wiedererkennbare Strukturen zu schaffen existieren Vorgaben, z.B. zur Benennung von unterschiedlichen Klassen und einer eindeutigen Package Struktur.

Wenn während der Entwicklung Probleme auftreten, wird in den meisten Fällen zuerst der Architekt gefragt. Anschließend wird das Problem am Ende des *Standup-Meetings* angesprochen. Daraufhin wird entschlossen, ob dies im Rahmen der aktuellen Story entwickelt wird oder ob der Architekt eine neue Story erstellt. Mögliche Lösungsvorschläge werden diskutiert. Die Lösungsvorschläge können sowohl vom Architekten, wie auch von jedem anderen Entwickler stammen. Bei komplexeren Problemen, die auch andere betreffen können, kann das Problem in einer größeren Runde besprochen werden. Bei der größeren Runde kann es sich z.B. um das *Jour Fixe* handeln. Beim Jour Fixe treffen sich die Architekten jedes Teams. Sie sprechen über aktuell relevante Themen und Probleme. Lösungen werden gemeinsam erarbeitet. Das Jour Fixe findet wöchentlich statt und dient als Hauptzeitpunkt zum Austausch zwischen den Architekten. Zu einem früherem Zeitpunkt wurden die Architekturthemen in einer Runde mit allen Teammitgliedern besprochen. Für viele was dies allerdings nicht relevant. Aus diesem Grund wurde die Besprechung in ein extra Treffen ausgelagert. Ein weiterer Austausch findet alle zwei Wochen im *Demo-Meeting* statt. Dies ist ähnlich wie ein Review zu verstehen. Allerdings sind die Stories meist bereits, durch den Einsatz von Continious Deployment, in der Produktionsumgebung am Laufen. Zu diesem Zeitpunkt werden die realisierten Ergebnisse den anderen Teammitgliedern präsentiert. In einer zusätzlichen Retrospektive wird über die Arbeitsweise gesprochen. Dadurch soll der gesamte Prozess verbessert werden. Ein alle zwei Wochen durchgeführtes *Code-Camp* dient zum Besprechen von Änderungen an der Architektur. Bei diesem Treffen können Personen ansprechen, welche Stellen sie derzeit stören. Diese Probleme können auch im Standup-Meeting angesprochen werden. Es konnte aber festgestellt werden, dass in einem zusätzlichem Meeting, mit einem fest eingeplantem Zeitraum, mehr Dinge gefunden und angesprochen werden.

In jedem Team ist ein Architekt integriert. Der Architekt ist gleichzeitig als Entwickler tätig. Durch weitere Verpflichtungen, wie den Meetings, entwickelt er weniger als die anderen Entwickler. Der interviewte Architekt sagte, dass er sich primär, von den Rechten und Pflichten, als Entwickler statt als Architekt sieht. Theoretisch hat er zwar die Möglichkeit eine Entscheidung vorzugeben, praktisch käme dies aber nie vor. Die gesamte Architektur soll vom Team selbst entwickelt werden. Wenn er Vorschläge macht, muss er die anderen Entwickler davon in einer Diskussion überzeugen. Die Aufgaben des Architekten sind in erster Linien zu unterstützen, zu coachen und zu beraten.

Eine explizite Dokumentation der Architektur wird nicht durchgeführt. Dadurch ist es zu manchen Zeitpunkten schwer einzelne Entscheidungen nachvollziehen zu können. Es existieren nur zu den einzelnen Jour Fixe Terminen Notizen. Ansonsten wird ein *Readme* erstellt, welches erklärt wie die Anwendung aufgesetzt wird.

Eine Überprüfung der Architektur wird in den bereits erwähnte Meetings durchgeführt. Außerdem wurde in einem Versuch der Sotograph, bzw. Sonargraph eingesetzt. Hierbei wurde allerd-

ings festgestellt, dass die Regeln teilweise zu starr sind und deswegen bewusste Verletzungen als Fehler erkannt wurden. Stattdessen wird derzeit das Werkzeug Moose[1] untersucht. Dieses kann als Query Language für den Quellcode dienen. Dadurch können Regeln variabler definiert werden. Außerdem kann aus den Ergebnissen ebenfalls eine individuelle Visualisierung erzeugt werden.
Um architektonische Aufgaben zu lösen und z.B. eine Refaktorisierung durchzuführen ist circa 20% der Zeit fest eingeplant. Bei größeren Änderungen muss dies mit dem Kunden abgesprochen werden. Da es sich um eine interne Entwicklung handelt ist dies in den meisten Fällen kein Problem.

Als eine der Schwächen, die in zukünftigen Projekten behoben werden soll, wurde die fehlende Dokumentation genannt. Bei dieser muss abgewägt werden, welche Informationen dokumentiert werden sollten und welche nicht. Das richtige Maß dazu zu finden ist meist relativ komplex.
Es existiert der Wunsch die aktuellen Teams weiter aufzuteilen und dadurch zu verkleinern. Die aktuelle Trennung nach den fachlichen Bereichen soll in kleinere Bereiche unterteilt werden. Dadurch müssen sich die Teams eines Fachbereiches sehr wahrscheinlich untereinander koordinieren. Ein weiteres Problem ist die ansteigende Personenanzahl der Teilnehmer im Jour-Fix. Aus jedem Team existiert ein Teilnehmer. Da dieses Meeting nicht gut mit der Größe skaliert, muss eine Alternative gefunden werden. Zu dieser Thematik stellt sich ebenfalls die Frage, ob die Rolle des Architekten weiterhin sinnvoll ist oder diese Aufgabe vollständig in das Entwicklerteam integriert werden sollte. Dies macht umso mehr Sinn, je kleiner das eigentliche Team wird.

[1] http://www.moosetechnology.org/, Abgerufen am 24.08.2016

Unternehmen D

Interviewort: Im Unternehmen

Interviewlänge: 44 Minuten

Entwicklungsart: Produktentwicklung für den internen Einsatz

Entwicklungsvorgehen: Scrum/Klassisch

Teams: mehr als 20 Teams mit 3 - 20 Mitgliedern

Unternehmensbeschreibung

Unternehmen D ist in der Versicherungsbranche tätig. Die gesamte Entwicklung geschieht mit über 20 Teams. Früher hat das Unternehmen vollständig nach traditionellen Methoden gearbeitet. Derzeit arbeitet noch nicht jedes Team agil. Das Unternehmen befindet sich in einer Umbauphase. Ziel ist, dass langfristig jedes Team agil arbeitet. Allgemein wird der Scrum of Scrum Ansatz verfolgt. Früher wurde das V-Model und ein ähnliches Vorgehen, wie das Wasserfallmodel für die Entwicklung verwendet. Die traditionell agierenden Teams arbeiten weiterhin nach diesem Vorgehen. Der Bericht handelt nur von den agilen Teams.
Der Interviewpartner ist seit über 14 Jahren in dem Unternehmen tätig.

Interviewbericht

Zum aktuellen Zeitpunkt entwickelt das Unternehmen ein Produkt für den internen Einsatz. Andere Fachabteilungen stellen die Kunden dar. In Zukunft ist es möglich, dass die Software auch an andere Unternehmen vertrieben wird.

Innerhalb der Teams wird viel über die Architektur diskutiert. Die Art und Weise organisieren die Teams vollständig eigenverantwortlich. Beim Sprint Planning wird *Planning-Poker* eingesetzt. Wenn große Differenzen bei den Schätzungen existieren wird über die Umsetzung diskutiert. Unklarheiten werden dadurch besprochen. Falls Änderungen an der Architektur vorgenommen werden müssen, die eventuell noch geklärt werden müssen, fällt die Schätzung höher aus. Im Sprint Review wird das Ergebnis vorgestellt. Besonderheiten bei der Entwicklung werden vor dem Review in einem zusätzlichem Meeting vorgestellt. Der PO nimmt an diesem nicht teil. Änderungen an der Architektur werden hier besprochen. Das Treffen findet unregelmäßig und nur zu speziellen Themen statt. Wenn bei einer Retrospektive Probleme mit großem Änderungsaufwand identifiziert werden, werden diese dem PO vorgelegt. Wenn notwendig wird ein neues Ticket angelegt. Im Normalfall ist die Änderung bereits in die Schätzung des Tickets mit eingeflossen. Wenn gewünschte Prozessänderungen erkannt werden, müssen diese geprüft werden. Zu diesem Zweck wird analysiert, was realistisch lösbar ist. Die Änderung wird experimentell in einem einzelnem Team eingeführt. Wenn Änderungswünsche zur Organisation um das Team herum entstehen, muss dies weiter diskutiert werden.

Probleme die während der Entwicklung auftreten können im Daily-Scrum Meeting angesprochen werden. Wenn diese das Sprint Ziel gefährden, wird dies dem SM und dem PO mitgeteilt. Das

Ziel wird als Folge entsprechend angepasst. Um von vornherein Fehler zu minimieren, wird die Entwicklung mithilfe von Feature-Branches und Pull Request durchgeführt. Die Pull Requests werden vor dem Mergen nochmals gereviewt.
Wenn im Verlauf der Entwicklung festgestellt wird, dass größere Umbauten notwendig sind, kann dies zu technische Tickets führen. Bei der traditionellen Entwicklung kam es durchaus vor, dass ein rein technisches Release gemacht wurde. In diesem Fall wurde z.B. für ein halbes Jahr nur an der Technik gearbeitet und keine neuen Funktionalitäten umgesetzt. Wenn dies bei der agilen Entwicklung auftritt arbeiten z.B. zwei Entwickler an drei bis vier Tagen daran. Der Rest des Teams entwickelt weiterhin Funktionalitäten. Ohne eine Architektur-Diskussion im Team entsteht schnell eine Architekturerosion und *Code-Ownership*. Einzelne Entwickler zeigen ungern ihren Code und ihre Realisierung und sind nicht besonders offen für Anregungen.

Die Art und Weise, wie in den Teams diskutiert wird ist individuell und hängt davon ab, worauf diese sich einigen, bzw. wie sich dies entwickelt. Über konkrete Entwurfsmuster wird wenig geredet. Viele der Strukturen sind bereits durch Frameworks und Technologien vorgegeben. Das Diskussionsvokabular entwickelt sich hauptsächlich durch die Diskussion am Anfang der Umsetzung und den am Ende stattfindenden Pull Requests. Hier kann festgestellt werden, dass verschiedene Begriffe unter Umständen unterschiedlich verstanden wurden. Um ein einheitliches Verständnis zu schaffen, muss eventuell nochmals im Team über das Problem geredet werden.

Die unterschiedlichen Teams entwickeln eigenständige Anwendungen. Zwischen diesen Anwendungen existieren allerdings Schnittstellen. Diese müssen im Team identifiziert werden. Eine Lösung wird derzeit individuell zwischen den Teams ausgehandelt. Im Laufe der Umstellung der Entwicklung soll hierfür ein Provider-Consumer Gespräch eingeführt werden. In diesem expliziten Planungszeitraum sollen Kapazitäten für Anpassungen mit eingeplant werden. Ein weiterer Austausch zwischen den Teams ist zum Zeitpunkt des Interviews nicht existent. Zu einem früheren Zeitpunkt gab es Bemühungen Vorträge zu interessanten Themen durchzuführen. Dies wurde eingestellt, weil die Personen die relevante Themen vorstellen konnten, keine Zeit dazu hatten. Im Laufe des Umbaus soll die notwendige Zeit hierfür mit eingeplant werden. Dazu sollen Gremien gebildet werden, in denen sich die unterschiedlichen Rollen der Teams treffen können. Alle Teams können Personen zu diesen Treffen schicken. Der Wissensaustausch hängt sehr von der Unternehmenskultur ab. Es muss zusätzliche Zeit investiert werden, in der kein Code produziert wird.
Eine Diskussion der Stories und Epics findet häufig anhand der Abnahmekriterien (AK) statt. Hier wird darüber geredet, wie diese erreicht werden können. Eine zusätzliche Zeitspanne wird als AK hinzugefügt, wenn die Aufgabe einen Forschungsabschnitt enthält. Dies soll verhindern, dass zu lange darüber nachgedacht wird. Der Zeitpunkt legt fest, wann eine Entscheidung spätestens getroffen sein muss.

Die Architekten sind fest in das Team integriert. Der Interviewer sagt, dass ein Architekt der nicht programmiert kein richtiger Architekt ist. Dieser muss seine eigenen Entscheidungen ebenfalls umsetzten. Nur so kann er sehen, ob diese realistisch sind. Der Architekt ist praktisch wie ein Entwickler. Nach außen hat er die Spezialrolle und dient dadurch als Ansprechpartner. Anfragen muss er nicht selbst beantworten. Es reicht aus, wenn er auf entsprechende Teammitglieder verweisen. Sein Aufgabenschwerpunkt liegt im Mentoring. Zusätzlich ist er an den unternehmensweiten Architektur-Diskussionen beteiligt. In diesen werden z.B. Rahmenbedingungen oder Protokolle für Schnittstellen besprochen. Für die Diskussionen existieren noch keine Rahmenbedingungen. Diese sollen in Zukunft aufgebaut werden. Innerhalb eines Teams können mehrere Personen eine Architektenrolle haben. Der Architekt kann zusätzlich weitere Rollen haben. Es ist aber ungewöhnlich, dass eine Kombination mit dem PO existiert. Dies kommt nur in Ausnahmefällen bei rein technischen Projekten für andere Techniker vor. Neben den in Scrum üblichen Rollen wie PO und SM existieren weitere technische Rollen. Der Build-Manager, welcher in Zukunft zum Dev-Ops-Engineer wird, kümmert sich um das Bauen

der Anwendungen und um die Testumgebung. Zusätzlich gib es einen fachlichen und einen technischen Testkoordinator, sowie einen Wiki/Socialmedia Koordinator.

Um Probleme in der Anwendung festzustellen findet ein Monitoring über Tests statt. Außerdem wird in manchen Teams Sonarqube eingesetzt. Zusätzlich wird das Verhalten zur Laufzeit beobachtet (Speicherauslastung, CPU Auslastung, ...). Im Rahmen eines Versuchs wurde der Sonargraph Architekt einmalig ausprobiert. Dieser Versuch wurde entweder durch einen totalen Widerstand oder durch eine relative Gleichgültigkeit aufgenommen. Die Personen haben einzelne Punkte mitgenommen, die restlichen aber nicht als besonders relevant empfunden. Bei dieser Art von Analyse wird eine Lücke zwischen dem Soll- und Ist-Zustand aufgedeckt. Wenn diese klein ist wird sie geschlossen. Bei einer großen Lücke ist die Motivation nicht besonders stark diese zu schließen. Die gefunden Problem wurden nicht als relevant genug betrachtet und nur als eine Verschönerung nach speziellen Prinzipien beurteilt. Ein weiteres Problem, welches bei dem Versuch festgestellt wurde, ist dass durch das Schließen der Lücke nicht zwingend zum Ziel der Entwicklung hingearbeitet wird. Eventuell ist die verglichene Soll-Architektur nicht mehr das zu gewünschte Ziel. Allgemein werden keine Prüfungen im voraus durchgeführt. Es wird erst bei festgestellten Problemen reagiert.

Bevor agil Entwickelt wurde, wurde eine große Dokumentation erstellt. Diese wurde zum Großteil von Personen geschrieben, die nur zu diesem Zweck eingestellt wurden. Dies hat die Entwickler von der Dokumentationspflicht entbunden. Sie wurde nur erstellt, weil dies vorgeschrieben war. Wenn Prüfungen durchgeführt wurden, wurde auf deren Existenz und auf Formalien geachtet. Dadurch hatte der Inhalt allerdings wenig mit der Realität zu tun. Seit dem agil Entwickelt wird dokumentieren die Entwickler selbst mehr als zuvor. Dies ist der Fall, weil hier Personen vorhanden sind, die diese Dokumentation wirklich verwenden und deshalb Wert auf Qualität legen. Wenn das Erzeugen dieser Sinn macht, wird sie auch erzeugt und hinterfragt. Bei keiner Nutzung darf die Dokumentation zurecht veralten. Dokumentiert werden hauptsächlich Architektur-Blueprints. Große Entscheidungen, die zu großen Änderungen geführt haben, werden niedergeschrieben. Hierzu werden die Begründungen, die Rahmenbedingungen zum Zeitpunkt der Entscheidung und die verworfenen Alternativen notiert. Dies soll helfen die Entscheidung später nachvollziehen zu können und ein reines Gewissen geben, wenn zu einem späteren Zeitpunkt eine andere Entscheidung getroffen wird. Die Ergebnisse können auch in den Reviews vorgestellt werden.
Für einen Einarbeitung in die Anwendung sind diese Dokumente nicht gedacht. Zur Einarbeitung existiert eine separate Anleitung zum Aufsetzten und zu den Basisarchitekturkonzepten. Eine vollständige allumfassende Dokumentation ist zu groß und aufwendig zu pflegen.
Zur dokumentation der Funktionsweise sind die Tests ausreichend. Die Unit-Tests stellen Beispiele für die Verwendung dar und geben an was passieren soll. Die fachlichen Tests machen das gleiche auf Systemebene satt auf Komponentenebene.

Während der Entwicklung werden zu manchen Zeitpunkten Ausnahmen gemacht. Die Häufigkeit stellt den Wegpunkt dar, ob sich ein Team in Richtung Agilität bewegt. Zusätzlich hängt dies davon ab, ob diese Teams agile Inselteams bleiben oder ob das gesamte Unternehmen agil wird. Seltene Ausnahmen sind in Ordnung, sollten aber nicht zur Regel werden. Wenn die Reviews oder Retrospektiven ausgelassen werden, geht der Inspect Anteil verloren. Folgendermaßen ist keine Adaption mehr möglich. Ein Zeitpunkt für vermehrte Ausnahmen ist häufig die Urlaubsphase.

Ein Problem bei der Entwicklung mithilfe von Scrum of Scrums ist die Kommunikation zwischen den Teams und deren Abhängigkeiten. Dies soll mit der bereits erwähnten Schnittstellendiskussion verbessert werden. Der Austausch über die Teams hinweg und über deren Schnittpunkte ist schwer. Jedes Team besitzt einen eigenen Planungsrhythmus. Die Teams benötigen Stories, welche innerhalb eines Sprints abschließbar sind. Bei Abhängigkeiten zu anderen Teams, tritt die Gefahr, dass eine Einigkeit nicht schnell gefunden werden kann, in circa 95% der Fälle ein.

Die Tickets können somit unter Umständen mehrere Sprints lang unerledigt im Sprint-Backlog bleiben. Die Motivation diese zu lösen sinkt. Die Personen der Teams kommen eventuell gut miteinander aus, können aber durch den organisatorischen Rahmen nicht zusammen arbeiten. Um diese Abhängigkeiten zu managen ist ein extremer Planungsaufwand erforderlich. Dies würde eine Planungssicherheit suggerieren. Da die Software aber ebenfalls altert und die Umwelt sich ändert sind Anpassungen notwendig. Beim Ignorieren der Umwelt wird die Anwendung nicht lange lauffähig bleiben, da die Technik nicht mehr zu ihr passt. Durch solch einen Versuch würde sich das Unternehmen vom agilen Ansatz entfernen. Um dies zu vermeiden wird versucht die Abhängigkeiten zu eliminieren obwohl diese vorhanden sind. Um das Probleme zu lösen wird darüber nachgedacht Paradigmen wie, dass es nur eine Wahrheit der Daten gibt, aufzugeben. Dies führt zu Redundanzen, wodurch unter Umständen verschiedene Versionen der Daten vorhanden sein können. Durch diesen Ansatz können Microservices eingesetzt werden, die nahezu isoliert arbeiten können.
Ein weiterer Punkt, der große Probleme bei der Entwicklung verursachen kann, ist die soziale Komponente der Teams. Dies wird durch die Socio-Technical Congruence beschrieben. Wenn zwei Personen nicht besonders gut miteinander auskommen, spiegelt sich dies auch in der Kopplung deren Komponenten wieder. Im schlimmsten Fall beinhalten diese viel redundanten Code.

Der Interviewte ist der allgemeinen Meinung, dass die agile Entwicklung die Entwicklung einer Architektur stärker fördert als die traditionelle Entwicklung. In der traditionellen Entwicklung legt der Projektleiter den Umfang und den Zeitraum der Entwicklung fest. Wenn dies festgelegt ist leidet die Qualität und somit die Architektur darunter. In der agilen Entwicklung haben die Entwickler eine priorisierte Liste an Themen, deren geschätzten Umfang und ein festgelegtes Zeitfenster. Die Entwickler entscheiden selbst, wie viel des Umfangs sie mit in das Zeitfenster nehmen. Erst so wird es möglich Zeit für Qualität durch Tests und weitere Maßnahmen einzuplanen.

Unternehmen E

Interviewort: In der HAW

Interviewlänge: 77 Minuten

Entwicklungsart: Projektgeschäft

Entwicklungsvorgehen: Scrum

Teams: 2 Teams mit 6 - 7 Mitglieder

Unternehmensbeschreibung

Das Unternehmen E ist eine IT-Beratungsfirma. Sie hat sich unter anderem auf die agile und die Architekturentwicklung spezialisiert. Dabei unterstützen sie sowohl bei der Organisation, z.B. als Scrum Master, wie auch als Entwickler. In der Tätigkeit als Berater war der Interviewpartner in unterschiedlichen Firmen tätig und kann dadurch von verschiedenen Erfahrungen berichten. Für das Interview diente hauptsächlich das letzte längere Projekt. In der aktuellen Firma ist die Person seit 5 Jahren tätig.

Interviewbericht

Am Anfang des Interviews wurde von einem negativ Beispiel berichtet. Die Architekturentwicklung lief in eine falsche Richtung. Die Beratungsfirma wurde beauftragt, weil es zu ständigen Verzögerungen während der Entwicklung kam. Das Problem in der Firma war, dass kein richtiges Architekturdenken vorhanden war. Es gab eine vollständig von der Entwicklung losgelöste Architekturabteilung. Diese hat ohne Beachtung der fachlichen Anforderungen die Architektur entwickelt. Die Entwicklungsabteilung war gezwungen diese Architektur umzusetzen. Wenn ein Problem durch die Standard Architektur nicht gelöst werden konnte, mussten sie sich an die Architekturabteilung wenden. Selbstständig durften keine Lösungen erarbeitet werden. Häufig hat eine passende Rückmeldung der Architekturabteilung vier bis fünf Wochen gedauert. In diesem Unternehmen wurde von Anfang an versucht das ideale System zu entwickeln und für alle Probleme gewappnet zu sein. Dadurch wurde sich aber die gesamte Zukunft verbaut, weil die Entwicklung sehr unflexibel wurde.

Das zweite berichtete Projekt wurde mit zwei Scrum Teams entwickelt. Der Interviewpartner war circa drei Jahre an diesem beteiligt. Die Software wurde für das Unternehmen selbst entwickelt. Die Teams bestanden aus sechs und sieben Entwicklern. Das Projekt war sehr groß. Der Quellcode bestand aus über 140 *Eclipse-Projekten*.
Die Entwicklung wurde mit einer Art von Scrum durchgeführt. Die Besonderheit bei deren Verfahren lag darin, dass der Projektleiter überall das Recht haben wollte, die letzte Entscheidungsgewalt zu haben. Dieses Recht hat er allerdings nie angewendet. Da dies nicht Scrum entspricht, wurde sich auf den Namen *Smart* geeinigt. In den restlichen Punkten lief die Entwicklung nach Scrum ab. Bei jeder umgesetzten Story wurde zusätzlich ein Pair-Review durchgeführt. Dies konnte einige Probleme vor deren Auftreten verhindern.

Die Entwicklung der Architektur lief anhand der User Stories ab. Im zweiten Teil des Sprint Plannings, beim Aufteilen in Tasks wurde geprüft, ob die für den Sprint ausgewählten Stories in das aktuelle System eingebaut werden können oder ob größere Umbauten notwendig sind. Dabei wurde besonders darauf geachtet, ob die Umbauten auch für andere Stories aus dem Sprint relevant sind. Es sollten keine *„Insellösungen“* geschaffen werden. Stattdessen wurde Wert auf allgemeingültige Lösungen gesetzt, die nicht an vielen verschiedenen Stellen neu realisiert werden müssen. Die Umbauten wurden anschließend in den meisten Fällen im Rahmen der Story durchgeführt. Wenn es trotz der vorherigen Überlegungen zu Problemen kommt, fallen diese während der Entwicklung auf und können im Daily angesprochen werden. Daraufhin konnte sich nochmals im Team zusammengesetzt werden, um eine Lösung zu besprechen. Häufig wurde das Problem spontan in Partnerarbeit besprochen und die Lösung im Daily vorgestellt. Sofern bei der Vorstellung keine Einwände kamen wurde dies realisiert. Wenn Einwände kamen wurde dies mit den interessierten Personen im Anschluss detaillierter besprochen. An dem Daily haben beide Teams teilgenommen. Da dadurch circa 20 Personen anwesend waren, wurde sehr auf die zeitliche Begrenzung geachtet. Die eigentlichen Diskussionen fanden mit wenigen Personen statt. Ansonsten wurden diese schnell ineffizient. Gründe dafür sind, dass wenn mehr Leute teilnehmen sich schneller an Detailfragen aufgehangen und von der Thematik abgeschweift wird. Außerdem ist für viele Personen die Thematik meist irrelevant.

Dieses Vorgehen hat keine expliziten Architekturmeetings, zum Modellieren oder ähnlichem integriert. Es wurde stattdessen auf den kontinuierlichen Prozess anhand der Stories gesetzt. Ein Informationsaustausch der Entwickler fand stattdessen hauptsächlich in den vorhandenen Treffen statt. Eine weitere Möglichkeit war, dass Entwickler sich direkt an jemand anderes gewandt haben wenn sie wussten, dass sich dieser in dem Bereich gut auskennt. Dabei konnte es sich um Personen aus dem eigenen, wie auch einem anderem Team handeln. Einen weiteren besonderen Termin zum Austausch gab es einmal wöchentlich. Die Teilnahme hier war freiwillig. Dort konnten die Entwickler für sie relevante Themen ansprechen. Häufig gab es keine Themen. Bei den in Scrum üblichen Reviews wurde nur über die fachliche Arbeit geredet. Die Architektur kam nicht zu Sprache. Ein Grund hierfür ist, dass Personen aus den Fachabteilungen anwesend waren, die die Abnahme durchgeführt haben. Ein Review der Architektur fand somit nicht statt.

Bei der Architekturentwicklung gab es keine Person, die ein allgemeines Wissen über die gesamte Anwendung und alle Teilprojekte hatte. Die Rolle eines Architekten war somit nicht vorhanden. Stattdessen gab es einzelne Entwickler die Spezialisten für Teilbereiche waren. Dieser Status hat sich hauptsächlich daraus entwickelt, dass diese die längste Zeit im Projekt waren. Meetings zu Problemlösungen wurden häufig entweder von der Person geleitet, die die größte Erfahrung hatte oder von derjenigen, die das Problem festgestellt hatte. Eine feste Regelung gab es dazu nicht.
Im Projekt waren zusätzlich ursprünglich Entwickler vorhanden. Diese kannten sich gut mit der Fachlichkeit aus. Deshalb kannten sie auch die passenden Ansprechpartner in den Fachabteilungen. Zusätzlich gab es Business Analysten mit einer ähnlichen Aufgabe. Sie dienten als Schnittstelle zwischen den Entwicklern und den Fachabteilungen. Sie gehörten nicht direkt zum Team. Sie schrieben die Stories und dachten sich die Abnahmekriterien aus. Sie waren Ansprechpartner bei Fragen und fehlenden Informationen.

Um den Prozess zu verbessern wurden Retrospektiven durchgeführt. Besonders kurz vor einem anstehenden Release häuften sich die Probleme. Die Retrospektiven konnten diese zum Großteil lösen. Probleme, die von außerhalb des Teams kamen, wurden an die Projektleitung weitergegeben. In den meisten Fällen handelte es sich um Probleme zum Vorgehen. Technologische Probleme hätten hier ebenfalls angesprochen werden können.

Eine Dokumentation wurde nicht erstellt. Allgemein konnte der Interviewpartner berichten, dass diese in den wenigsten Projekten erstellt wird. Er würde sich wünschen, dass eine Doku-

mentation die wichtigsten Entscheidungen beinhaltet. Die Entscheidungen, sollten inklusive der Begründung für die Lösung und Begründungen, weshalb die Alternativen nicht gewählt wurden notiert werden. Aus seiner Erfahrung beschränkt sich die Dokumentation in den meisten Fällen auf eine inline Dokumention und einer Installationsanleitung. Aber auch hier konnte er von Fällen berichten, bei denen diese nicht mit dem Quellcode übereinstimmte. Außerdem wurden Infos häufig in der Story als Kommentar hinzugefügt. Dies aber auch nur, wenn eine Diskussion im Ticket stattgefunden hat. Ein anschließendes Aufschreiben geschieht selten. Um dies im Nachhinein wiederzufinden, mus beim Einchecken des Codes mindestens eine Referenz auf die Ticketnummer gesetzt werden. Das Problem bei einer Dokumentation ist, dass dessen Pflege schwer ist. Zu Anfang eines Projektes ist die Motivation häufig groß. Deshalb wird begonnen eine Dokumentation zu erstellen. Diese wird im Verlauf aber meist nicht mehr aktualisiert.
Eine allgemeine Übersicht ist dadurch nicht vorhanden und kann nur durch manuelles durchgehen des Codes erhalten werden. Dies ist nicht tragisch wenn die Implementierungspersonen noch im Team sind. In vielen Fällen erinnern sich diese noch an die Umsetzung. Wenn die Person nicht mehr im Unternehmen sind wird es häufig schwieriger. Fehler können unter Umständen nicht mehr nachvollzogen werden. In diesem Fall muss der Code Zeile für Zeile, eventuell mit einem Debugger, durchgegangen werden.
Eine Einarbeitung in die Anwendung war auch in diesem Projekt nur auf diese Art möglich. Dazu hat die neue Person meist versucht erste Stories zu entwickeln. Dadurch soll sie nach und nach einen Überblick bekommen. Zur Unterstützung wurde dies häufig als Pair Programming durchgeführt.
Wenn über die Architektur geredet wurde, wurden häufig Pattern verwendet. Dadurch konnten eindeutige Umsetzungsvorschläge gemacht werden. Es kann vorkommen, dass mehr Klassen oder ähnliches benötigen werden. Die entstandenen Strukturen sind aber besser. Um dies erfolgreich einzusetzen muss die Erfahrung im Team, zu unterschiedlichen Pattern, vorhanden sein. Ein weiterer Vorteil dabei ist, dass es einfacher ist, sich im Code zurechtzufinden, weil eindeutige Namensgebungen und anderweitig bekannte Strukturen erkennbar werden.

Es konnte festgestellt werden, dass bei Ausnahmen z.B. beim Pair-Review einige Fehler übersehen wurden. Diese Ausnahme kam z.B. vor, wenn unter Zeitdruck gearbeitet werden musste. Bei Zeitdruck wurde beschlossen mehr Stories in den Sprint zu nehmen und dafür die Reviews auszulassen. Dadurch ist allerdings die Qualität gesunken. Durch die anschließende Notwendigkeit die Probleme zu reparieren, wurde im Gesamten meist mehr Zeit benötigt. Aus seiner Erfahrung konnte der Interviewpartner sagen, dass wenn ein Team Pair-Reviews, Pair Programming oder ähnliches machen möchte nicht daran gehindert werden sollte.
Um die Qualität des Codes zu überwachen wurde Sonarqube eingesetzt. Dadurch wurden z.B. Stellen mit einer zu hohen Komplexität gefunden, welche eine Umstrukturierung erforderten. Außerdem konnten die Diagramme auch von nicht Technikern verstanden werden und dadurch zur Kommunikation und Begründung für manche Änderungen eingesetzt werden. Als während der Entwicklung fachlich weniger zu tun war, durften die Entwickler circa 30% ihrer Zeit verwenden um Sonarqube-Tickets abzuarbeiten. Diese Tickets waren in der Regel rein technische Tickets.

Der Interviewpartner konnte ebenfalls von allgemeinen Problemen, die häufig auftreten berichten. Dazu zählt, dass es immer schwer ist das Management, bzw. den Kunden von Arbeiten zu überzeugen, die keine neuen Funktionalitäten liefern aber trotzdem Zeit in Anspruch nehmen. Dies ist besonders schwer, wenn die Personen von der Technik wenig Ahnung haben. Bei dieser Problematik kann es hilfreich sein, ähnliche Beispiele aus einem anderen Themenbereich zu finden. Als Beispiel nannte er, dass bei manchen Autos um eine Glühbirne zu wechseln der gesamte Scheinwerfer ausgebaut werden muss. Eine weiterer Ansatz ist die Leute davon zu überzeugen, dass nach dem Umbau die Implementierung der eigentlichen Fachlichkeit mit weniger Aufwand verbunden ist.
Außerdem entstehen weitere Probleme, wenn die Stories von Personen geschrieben werden die

von der technischen Seite keinerlei Ahnung haben oder die Stories im allgemeinen zu groß sind und nicht verkleinert oder geteilt werden können. Das Schätzen von zu großen Stories ist umso schwerer und problematisch wenn diese Zeiten, wie den halben Sprint benötigen würden.
Ein anderes großes Problem ist, wenn im Team jeder für sich entwickelt und keine Absprachen gemacht werden. In diesen Teams werden häufig keine Retrospektiven gemacht. Dadurch werden Probleme nicht besprochen weil vermutet wird, dass diese nicht deren Probleme sind sondern an der schlechten Organisation liegen.
Das Auswählen von Stories, an der Priorisierung vorbei, sollte ebenfalls vermieden werden. Entwickler sollten die Stories immer von oben herab bearbeiten, auch wenn ihnen diese nicht vollständig verständlich sind oder keinen Spaß machen. Dies kann problematisch sein, weil eventuell andere Stories von diesen abhängig sind. Wenn die Person Probleme bei der Story hat, sollte sie stattdessen eine weitere Person um Hilfe bitten oder die Story durch Pair Programming lösen.
Gerade bei neuen Entwicklern oder bei Stories, die Probleme bereiten, ist es wichtig frühzeitig um Feedback zu bitten. Zu diesem Zweck muss die Story nicht vollständig entwickelt sein. Dies ist wichtig, weil späte Strukturänderungen meist sehr schwierig und unbeliebt sind. Dies spricht für den Punkt, dass falsche Architekturentscheidungen, zu einem späterem Zeitpunkt meist schwer zu beheben sind. Um Feedback zu erhalten muss keine feste Rahmenbedingung vorhanden sein, dies sollte je nach Ausmaß unter den Entwicklern spontan stattfinden.

Der Interviewpartner hat den Wunsch in Projekten regelmäßig alte Softwarekomponenten zu überprüfen. Dadurch kann festgestellt werden ob diese noch mit den aktuellen Anforderungen übereinstimmen. Ansonsten werden viele Probleme erst erkannt wenn diese akut werden. Eventuell muss zu diesem Zeitpunkt zwischen einer aufwendigen Refaktorisierung oder einem Neubau entschieden werden. Gerade bei alten Systemen scheuen viele allerdings davor zurück diese abzuschalten. Oft ist unklar wie diese funktionieren. Die ursprünglichen Entwickler der Systeme sind meistens nicht mehr auffindbar. Dies erfordert aufwendige Analysen.

Die Rolle eines Architekten bewertet er als nicht so wichtig. In erster Linie sollte das Team selber vollständig verantwortlich dafür sein. In manchen Projekten kann es allerdings helfen wenn anfänglich eine Person im Team ist die diese Rolle übernimmt. Diese soll helfen den Grundstein für eine gute Architektur zu legen und das Team dazu leiten, diese nach und nach alleine weiterzuentwickeln und zu pflegen. Diese Aufgabe übernimmt oft jemand, der am meisten Wissen in dem Bereich besitzt. Seine Aufgabe sollte es aber sein die anderen Entwickler auf sein Level zu bringen. Er sollte sich selber überflüssig machen. In jedem Fall sollten die Entscheidungen im Team getroffen werden. Teamentscheidungen sind immer von höherer Qualität, als Einzelentscheidungen. Dies ist wichtig, da die Entwickler diese anschließend Umsetzen müssen. Um dies erfolgreich zu machen ist es notwendig, dass diese das Problem und die Lösung richtig verstanden haben. Bei dieser Thematik ist auch wichtig, dass neue Leute zu Wort kommen, da diese neue Erkenntnisse einbringen können und Fragen zu Bereichen stellen die für lange Teammitglieder selbstverständlich sind.
Die Entwicklung selbst sollte, wie bisher auch, Story-basiert ablaufen. Dabei sollte besonders auf eine gute Testabdeckung geachtet werden. Dies kann die Refaktorisierung erheblich vereinfachen. Bei der Entwicklung der Architektur muss darauf geachtet werden, dem Versuch das perfekte System zu entwickeln zu widerstehen. Dies machen die Personen eher für sich selbst, statt für den Kunden. Diese Gefahr kann auftreten, wenn extra Debatten dazu eingeführt werden. Diese Debatten dürfen dann nur für die aktuellen Stories durchgeführt werden.
Wenn sich Personen zusammensetzten um Lösungen zu finden, sollte auf einfache Techniken gesetzt werden. So sollte z.B. ein einfaches Whiteboard, statt einer komplexen Software eingesetzt werden.

Unternehmen F

Interviewort: im Unternehmen

Interviewlänge: 51 Minuten

Entwicklungsart: Produktgeschäft

Entwicklungsvorgehen: Kanban

Teams: ein Team mit 10 Leuten

Unternehmensbeschreibung

Dieses Unternehmen entwickelt eine webbasierte Jobsuchmaschine. Die zu durchsuchenden Daten werden durch externe Quellen anderer Unternehmen abgefragt. Die Daten werden aggregiert, inhaltlich geprüft und teilweise durch Zusatzinformationen ergänzt. Die Anwendung ist mithilfe von Microservices realisiert. Dies ermöglicht eine gute Aufteilung der Arbeit.

Das Unternehmen ist noch relativ jung. Die Entwicklung ist vor einem Jahr und drei Monaten gestartet. Derzeit arbeiten 10 Entwickler in dem Unternehmen. Diese sind in zwei Gruppen aufgeteilt, dessen Personen aber häufig wechseln. Der Wechsel geschieht aufgabenbezogen.

Interviewbericht

Zu Beginn der Enrwicklung hat das Business Development (BD) des Unternehmens hat den Wunsch nach einer speziellen Funktionalität. Anschließend besprechen die leitenden Personen des Entwicklungsbereich mit dem BD den Umfang der Entwicklung. Komplexe Sachen werden dabei häufig gestrichen.Diese waren eventuell falsch verstanden oder der Aufwand stand in keinem Verhältnis zum Nutzen. Wenn der Aufwand klar ist wird ein Ticket erstellt und anschließend priorisiert.
Sofern es sich um ein größeres Ticket handelt findet vor der Entwicklung nochmals eine Planung statt. Bei der Planung wird das Ticket durch den Produktmanager vorgestellt. Die Entwickler, die dies umsetzen wollen (meist 1 - 2 Personen), überlegen sich eine Lösungsstrategie. Bei sehr großen Tickets (Epic) überlegen sich die Entwickler, welche technischen Schritte umgesetzt werden müssen. Eine Person übernimmt für dieses Ticket und der Entwicklung einer Lösungsstrategie die Verantwortung. In den meisten Fällen wird die Lösung den anderen Kollegen nochmal vorgestellt. Anschließend wird das Ticket aus dessen Sicht erneut aufgeschrieben. Hierzu können Fragen gestellt werden. Diese betreffen im Normalfall nur kleine Unklarheiten, da die meisten Dinge bereits im Vorfeld abgesprochen wurden.
Wenn keine Planung stattfindet wird die Qualität anschließend durch die Reviews sichergestellt. Außerdem wird meist während der Realisierung spontan darüber gesprochen.

Die Umsetzung eines Ticket wird im Vorfeld beim Daily Meeting angekündigt. Die Person die im Vorfeld die Verantwortung für das Ticket übernommen hat, bearbeitet diese in den meisten Fällen auch. Andere Personen können die Entwicklung ebenfalls übernehmen. Die Person mit

der Verantwortung dient als Ansprechpartner zu Fragen des aktuellen Standes.
Nach der Umsetzung wird ein Pull Request erstellt, der wiederum ein Code Review durch einen anderen Entwickler auslöst. Anschließend folgt eine fachliche Abnahme durch das BD. In Ausnahmefällen, wenn es ein hauptsächlich technisches Ticket war, führt der Produktmanager die Abnahme durch. Nach der erfolgreichen Prüfung ist das Ticket *Ready for Production* und ist bei der nächsten Aktualisierung des Livesystems online. Eine Aktualisierung geschieht zwischen mehrmals täglich und einem paar mal pro Woche. Dies wird nicht automatisiert gemacht, weil für manche Änderungen die Kunden, welche die Daten zur Verfügung stellen, informiert werden müssen. Zu Beginn der Entwicklung wollte das Unternehmen mit festen Releases arbeiten. Dies wurde verworfen, weil durch die Automatisierung des Deployments das Livesystem in kurzer Zeit sehr häufig aktualisiert werden kann.

Nachdem ein Ticket zwei Wochen im Livesystem lief wird eine *Reflexion Besprechung* durchgeführt. Hier wird besprochen was bei der gesamten Entwicklung des Tickets gut gelaufen ist und was weniger gut. Der PM, das BD und mindestens ein beteiligter Entwickler nimmt daran teil. Inhaltlich geht es um das Zusammenspiel der unterschiedlichen Personenrollen und der Aufgabenstellung. Es wird z.B. besprochen, welche Informationen zusätzlich nützlich gewesen währen.
Die Teamarbeit wird im Rahmen einer Retrospektive alle drei Wochen diskutiert. Als Folge einer Retrospektive werden nicht immer Prozessänderungen beschlossen. Häufig ist es ausreichend das Bewusstsein für die Problematik zu schaffen um diese zu verringern.

Die Entwicklung wird mit Kanban durchgeführt. Die Performancemessung des Teams wird dabei allerdings ausgelassen. Eine Begrenzung von maximal zwei gleichzeitig bearbeiteten Tickets pro Person ist vorhanden. Dies hat den Grund, weil bei manchen Tickets auf Zuarbeiten von Kollegen gewartet werden muss.
Das täglich stattfindende Daily Meeting wird derzeit mit allen Entwicklern durchgeführt. Da das Unternehmen weiter wächst, muss dies in Zukunft aufgeteilt werden.
Die hauptsächliche Lösungsstrategie wird innerhalb des Task-Breakdowns besprochen. Wenn deutlich wird, dass Nachforschungsbedarf besteht wird eine extra Nachforschungsaufgabe mit einer festgelegten Maximalzeit erstellt.

Das Kanban Board enthält zwei Spalten zu speziellen Themenbereichen. Diese sind getrennt, weil diese nur von je einer, bzw. zwei Personen gesonderten bearbeitet werden. Pro Woche gibt es eine Person mit der *Unicorn* Rolle. Diese kümmert sich darum die anderen Teams von nicht mit der Entwicklung beteiligten Arbeiten freizuhalten. Dies kann z.B. das Einsortieren der Tickets sein.
Bei der Auswahl des nächsten Tickets zur Entwicklung wird im Normalfall nach der Priorisierung gegangen. Je nach Fähigkeit der Mitarbeiter gehen diese die Liste von oben durch und wählen das erste Ticket welches für sie machbar scheint.
Für Notfälle gibt es eine Fastlane in der dringende Tickets einsortiert werden. Diese dürfen alles andere Unterbrechen. Zum Zeitpunkt des Interviews wurden diese annähernd nicht mehr eingesetzt. Der Entwicklungsablauf wird durch sie gestört und die Produktivität stark einschränkt.
Wenn während der Entwicklung Probleme auftreten werden sie meistens im Standup angesprochen. Anschließend kann es zu einem spontanen Treffen zwischen den Personen kommen, für die dieses Thema relevant ist. Eine Folge kann die Erstellung von technischen Tickets sein. Eine aktive Suche nach Problemen wird nicht vorgenommen. Stattdessen wird reagiert wenn welche auftreten. Wenn die Probleme nicht besonders groß sind werden diese versucht durch Pair Programming zu lösen. In diesem Unternehmen ist es unproblematisch, wegen technischen Problemen, die Entwicklung von Features einzuschränken. Es ist möglich größere Refaktorisierungen durchzuführen.

Für einen allgemeinen Austausch findet einmal die Woche ein *Table-Talk* statt. Ein Mitarbeiter kann hier ein Thema vorstellen zu dem er Feedback haben möchte. Hier können auch neue getestete Technologien vorgestellt werden.

Eine Architekturrolle ist in diesem Unternehmen nicht vorhanden. Die Features werden versucht so klein und übersichtlich zu halten, dass eine größere Planung nicht notwendig ist. Außerdem wird bei den Microservices drauf geachtet, dass diese theoretisch in zwei bis drei Wochen vollständig neu entwickelt werden können.

Um eine hohe Qualität sicherzustellen, wird jedes Feature mithilfe von Tests überprüft. Diese werden automatisch mithilfe des Jenkins Dienstes ausgeführt. Für statische Code Analysen wird ein ähnliches Tool wie Sonarqube verwendet. Dieses prüft unter anderem auf Testabdeckung, *Magic-Numbers*, Komplexität und *Coding-Style*. Ein Conformance Check wird nicht durchgeführt. Tests in dieser Richtung gab es nicht. Die Entwickler testen öfters neue Werkzeuge. Hierfür nehmen sie sich zwischendurch zwei bis drei Stunden Zeit. Die Ergebnisse können z.B. in den Table-Talks präsentiert werden. Das Tool für die statischen Codeanalysen wurde durch solch einen Versuch eingeführt. Außerdem wurden zusätzlich weitere Tools, bzw. Skripte entwickelt welche z.B. den Stand des Produktivsystems mit dem des Stagingsystems vergleichen und bei zu großer Differenz alarmiert.
Ein weiteres Monitoring wird für jeden Service durchgeführt. Die Ergebnisse werden an einen *Slack* Server gesendet. Hier kann eingesehen werden was derzeit läuft und was nicht. Slack wird ebenfalls bei Ausfällen dazu genutzt zu alarmieren. Auf der Systemebene wird vieles mithilfe von *Google Cloud* geprüft. Hier werden vor allem auf Hardwaremetriken geachtet.

Zur Dokumentation wird ein Wiki-System eingesetzt. In diesem wird ein aktuelles Diagramm aller Microservices und deren internen Komponenten gepflegt. Es wird stark auf die Aktualität des Modells geachtet. Die Entwickler skizzieren die Änderungen in den meisten Fällen im Vorhinein per Hand. Nach der Umsetzung wird die Änderung in das Modell im Wiki-System übertragen. Bei Diskussionen wird sich häufig auf dieses Diagramm bezogen. Durch Diskussionen am Diagramm kommt es vor, dass einzelne Bereiche, Methoden, usw. verschoben werden. In dem Wiki-System werden außerdem Informationen zu dem Deploymentprozess, dem Einrichten der Entwicklungsumgebung und viele weitere festgehalten. Eine Dokumentation der Umsetzung im Wiki-System hängt sehr von dessen Komplexität ab. Ansonsten wird eine inline Dokumentation erstellt. Da große technische Entscheidungen bisher selten vorgekommen sind wurden diese nicht aufgeschrieben. Für Informationen zu Entscheidungen kann das Slack Archiv verwendet werden.
Zur Einarbeitung neuer Mitarbeiter gibt es einen *Onboarding-Prozess*. Hierzu gibt es z.B. eine vollständige Prozessbeschreibung zum Einrichten des Rechners und dem ersten Kontakt mit der Anwendung. In der Regel folgt im Anschluss eine ein bis zwei tägige Pair Programming Phase. Hier werden einfachere Aufgaben gewählt, die mit Unterstützung und ausführlichem Erläutern gelöst werden sollen. Während dieser Phase wird die neue Person regelmäßig gefragt in welchen Bereichen der Anwendung sie sich noch nicht wohl fühlt. Die nächsten Aufgaben werden versucht entsprechend dieses Bereiches zu wählen.

Ein Probleme während der Entwicklung sind die bereits genannten Fastlane Tickets. Diese kamen früher häufiger vor, weil das BD dem Kunden kurzfristig Features versprochen hatte. Dies wird inzwischen mit einer größeren Zeitspanne zur Umsetzung gemacht. Außerdem ist das Wachstum des Unternehmens für das aktuelle Vorgehen problematisch. Ein Grund hierfür sind die größer werdenden Meetings, wie auch die Organisation der Tickets. Mit zu vielen Personen ist das aktuelle Vorgehen nicht mehr effizient. Derzeit wird die Entwicklung hauptsächlich von erfahrene Senior Entwicklern durchgeführt. Laut dem Interviewpartner würde dies mit vielen Junior Entwicklern nicht funktionieren.

Allgemein würde der Interviewpartner sich wünschen, dass mehr Zeit zur Verfügung steht um nicht direkt firmenrelevante Themen zu besprechen.

Unternehmen G

Interviewort: In der HAW

Interviewlänge: 77 Minuten

Entwicklungsart: Produktentwicklung für den Endkunden als Webanwendung

Entwicklungsvorgehen: Kanban

Teams: unbekannt

Unternehmensbeschreibung

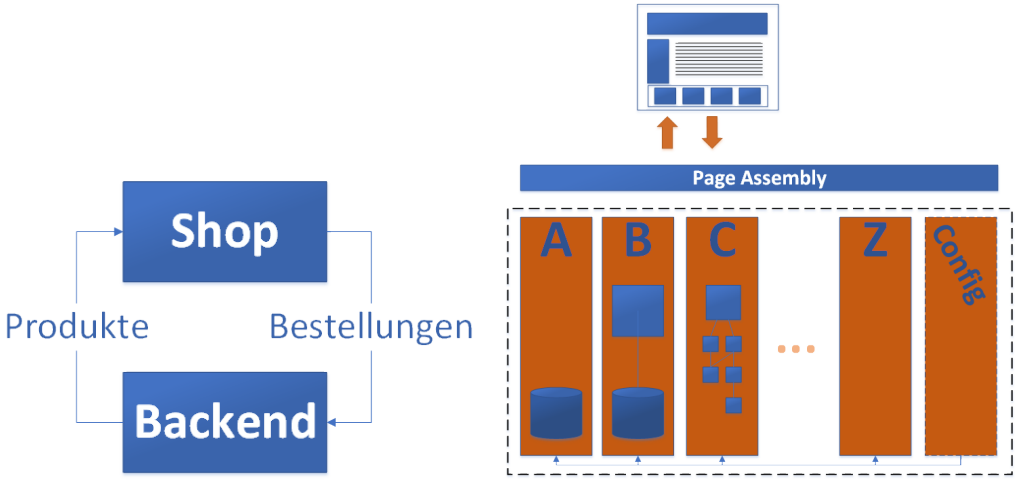
Unternehmen G entwickelt einen eigenen Webshop. Dieser wird circa eine Millionen mal pro Tag besucht und beinhaltet 2 Millionen Artikel. Pro Sekunde werden circa zwei Bestellungen getätigt[1]. Der Interviewpartner ist ein Softwarearchitekt und seit fast 7 Jahren in dem Unternehmen tätig. Er selbst hat bereits häufiger Vorträge zu Architekturthemen gehalten. Das Interview fand nicht im Unternehmen, sondern innerhalb der HAW Hamburg statt. Es hat 74 Minuten gedauert.

Interviewbericht

Unternehmen G hat neben dem eigentlichem Shopsystem noch viele weitere Anwendungen. Selbst entwickeln sie nur das Shopsystem. Dieses arbeitet mit einem Backend, welches als Blackbox-System agiert. Vereinfacht dargestellt nimmt der Shop Produkte aus dem Backend entgegen und gibt auf der anderen Seite durchgeführte Bestellungen wieder an das Backend zurück. Dieser Sachverhalt ist in Abbildung 1 (a) dargestellt. Das Blackbox System hat Schnittstellen um Produkte einzukaufen, diese mit Mediendaten anzureichern (z.B. Bilder, Videos, ...) und viele weitere. Dieses System wurde im Interview nicht genauer betrachtet.

Im Jahr 2011 wurde begonnen den Shop neu zu bauen. Alternativ hätte ein fertiger Shop eingekauft und an die eigenen Bedürfnisse angepasst werden können. Es wurde sich gegen diese Lösung entschieden, weil dadurch sehr wahrscheinlich ein System mit einer monolithischen Architektur entstanden wäre. Dieses System wäre schlecht horizontal skalierbar gewesen. Da bereits bekannt war, dass dies für die langfristigen Zielen nicht ausreichen würde, wurde sich gegen den Einkauf eines fertigen Systems entschieden.
Um eine bessere Skalierbarkeit zu erreichen entstand die Idee, dass nirgendwo vorgeschrieben sein muss, dass alles mit einer Anwendung realisiert werden muss. Von außen muss es für den Kunden nur wie eine Anwendung wirken. Aus diesem Grund ist ein System wie in Abbildung 1 (b) entwickelt worden. Die gesamte Anwendung ist in *Vertikalen* unterteilt (orange). Die Vertikalen stellen *Self-Contained-Systems* (SCS) dar. Diese sind nach fachlichen Aspekten geschnitten. Deshalb gibt es z.B. eine für Produkte, welche unter anderem Produktinformationen für

[1] laut der Webseite des Unternehmens



(a) (b)

Figure 1: Aufbau der Anwendung

die Detailseiten liefert und eine für Bestellungen, welche den Warenkorb und den Bestellprozess realisiert. Zu Beginn existierten fünf Vertikale. Mit der Zeit wurden es immer mehr. Inzwischen besteht der Kern des Shops aus über 100 Bestandteilen. Es existiert eine besondere Vertikale zur Konfiguration der anderen. Im Unterschied zu den anderen liefert diese keine Seiten aus. Der fachliche Schnitt ist in den unterschiedlichen Fachbereichen des Unternehmens wiederzufinden. Diese existierten bereits vor der Umsetzung des Systems. Früher war dies eine 1:1 Beziehung. Inzwischen ist dies eine 1:n Beziehung. Mehrere Vertikale können einem Fachbereich zugeordnet sein. Für eine Vertikale ist immer genau ein Entwicklerteam zuständig. Es kommt nicht vor, dass Teams sich eine Vertikale teilen, Teams betreuen häufig aber mehrere Vertikale. Bei der Entwicklung sind diese vollständig unabhängig. Sie besitzen keinen gemeinsamen Code mit anderen Systemen. Aus diesem Grund können sie vollständig unabhängig deployed werden. Um die Datenhaltung kümmert sich jedes System selbst. Die Kommunikation zwischen den Vertikalen ist vollständig asynchron. Wenn z.B. Produktinformationen für den Warenkorb benötigt werden wird das Produktsystem regelmäßig nach aktualisierten Informationen, wie Preisänderungen, Verfügbarkeitsänderungen, usw. gefragt. Dies ist ähnlich wie eine *Message-Queue* realisiert.
Oberhalb der Vertikalen befindet sich das *Page-Assembly* (blau). Dieses ist für das Caching und das Zusammenbauen der einzelnen Bestandteile, zu einer Seite, zuständig.

Der Aufbau der einzelnen SCS wird Mikro-Architektur genannt. Alles was übergreifend definiert wird, wird als Makro-Architektur bezeichnet. Über die Mikro-Architekturen wird nicht global debattiert. Dies ist vollständig dem zuständigen Team überlassen. Die Teams können selbstständig entscheiden, wie ihre Vertikale umgesetzt wird. Inzwischen werden viele Vertikale als Microservices realisiert. Dadurch besteht eine Vertikale wiederum aus vielen kleinen Bestandteilen.

In jedem Team existiert eine sogenannte Triade. Diese besteht aus einem *Technical Designer* (TD), einem *Business Designer* (BD) und dem *Production-Lead* (PL). Der TD kann wie die Architektenrolle verstanden werden. Er ist technisch Verantwortlich und sorgt somit dafür, dass die NF-Anforderungen eingehalten werden. Im Idealfall entwickelt dieser ebenfalls mit. Da seine anderen Aufgaben viel Organisation benötigen ist die Entwicklung aber kein Hauptbestandteil

seiner Arbeit. Der BD ist für die Fachlichkeit verantwortlich und kann wie ein Produktmanager, bzw. Product Owner, verstanden werden. Der PL ist für die Organisation verantwortlich und kümmert sich deshalb z.B. um den Ablauf der Meetings, sowie alles weitere was mit der Organisation des Teams zusammenhängt. Außerdem übernimmt er das Reporting nach außen. Zum Team gehören ansonsten die Entwickler. Den Entwicklern gehört die Architektur der Vertikalen. Der TD könnte unter Umständen allerdings ein Veto einlegen. Dies ist aber aber nie der Fall. Der Interviewpartner gehört zu den Technical Designern. Pro Team gibt es einen TD. Alle TDs treffen sich einmal die Woche um über die Makro-Architektur zu diskutieren. Hier kann außerdem z.B. vorgestellt werden, wie eine Gruppe ihre Vertikale realisiert hat.
Ein allgemeines Architekturteam existiert nicht. Dies würde auch weniger Sinn machen, da dieses die Komplexität mit über 100 Kernbestandteilen des Shops nicht beherrschen könnte.

Innerhalb der einzelnen Teams wird viel Debattiert. Da die Teams vollständig eigenverantwortlich sind werden hierzu keine allgemeinen Vorgaben gemacht. Das Team des Interviewpartners hat zu diesem Zweck zwei mal die Woche einen je einstündigen Meetingblock. Dort können z.B. neue Stories besprochen werden. Dieser Zeitraum wird vom BD genutzt um Feedback über Aufwände zu erhalten. Das Meeting kann auch für technische Debatten eingesetzt werden. Es kann z.B. über schlechte Muster im Code allgemein merkwürdige Codestellen gesprochen werden.
Für alle Teams gibt es alle zwei Wochen eine Entwickler Convention. Bei dieser können Themen unterschiedlichster Art präsentiert werden. Zum Teil werden die Vorträge von externen Personen durchgeführt. Jede Person kann frei entscheiden ob er diese besuchen möchte. Unregelmäßig finden weitere Austauschmöglichkeiten statt. Zum Zeitpunkt des Interviews fand eine interne, zweitägige Qualitätssicherungskonferenz statt. Events dieser Art werden häufiger zu verschiedenen Themen veranstaltet.
Die tägliche Entwicklung wird mit Pair Programming durchgeführt. Dadurch entsteht eine gute Wissensverbreitung und vermindert den Dokumentationsbedarf der Mikro-Architektur. Wenn ein Paar besonders lange an einer Story arbeitet wird spätestens am dritten Tag über eine Rotation geredet. Durch die Rotation soll ein Entwickler ausgetauscht werden um einen neuen Blickwinkel und ein anderes Wissen in das Paar zu führen. Ein weiterer Austausch findet beim Story-Kickoff statt. Sobald eine Story am Board hängt und ein Paar bereit ist diese zu entwickeln wird spontan, für circa 15 Minuten, über diese Story geredet. Die Diskussion kann sowohl über die Fachlichkeit, wie auch die technische Umsetzung gehen. Hier sind im Normalfall nur die ausführenden Entwickler, der BD und TD beteiligt. Das ein Paar eine Story umsetzen möchte wird bereits im Daily Meeting kommuniziert.
Bei der Entwicklung wird nur das Minimum der Fachlichkeit realisiert. Vorhersagen in die Zukunft werden nicht gemacht. Der Code wird dabei einer kontinuierlichen Refaktorisierung unterzogen. Falls beim Entwickeln größere Refaktorisierungen erforderlich sind kann es vorkommen, dass eine technische Story erstellt wird. Die Entwicklung läuft vollständig nach dem Test-First Prinzip ab. Bevor eine Story entwickelt wird, wurden durch einen weiteren Qualitätsmanager *User Aceptance Tests* (UATs) erstellt. Diese beschreiben aus fachlicher Sicht ein Wenn-Dann Szenario. Die UATs sind als Freitext formuliert und werden vom Entwickler nacheinander realisiert (Test und dessen Lösung). Das Test-First Prinzip ist wichtig, weil jedes System eine eigene *Deployment-Pipeline* hat. Diese deployed jeden Push direkt. Dadurch ist ein Push im Normalfall nach circa 15 Minuten live. Die Entwicklung von neuen Features wird mit vielen kleinen Commits direkt im Masterbranch durchgeführt. Dies bewirkt, dass ebenfalls unvollständige Stories direkt auf der Produktivumgebung laufen. Dies wird durch das Test-First Prinzip möglich. Es stellt sicher, dass alles wie erwartet funktioniert. Wenn ein unfertiges Feature in der Liveumgebung deaktiviert werden soll, muss ein *Feature-Toggle* eingebaut werden. Dies ist ein Schalter der das Feature in der Liveumgebung deaktiviert und nur in einem Vorsystem aktiviert. Zwischendurch und wenn das Feature fertig entwickelt wurde kann dies in eine QA-Umgebung aktiviert werden und z.B. dem PO gezeigt werden. Die Aktivierung für

das Livesystem ist anschließend über eine GUI möglich.
Diese Arbeitsweise sorgt dafür, dass das Team viel Verantwortung und sehr vielseitige Aufgaben hat. Die Teams agieren sehr autonom. Zentrale Strukturen sind in solch einem verteilten System nicht vorstellbar und würden immer ein *Bottleneck* darstellen, weil z.B. Freigaben durch ein Architekturboard erfolgen müssten.

Wenn Diskussionen stattfinden, wird dies meist mithilfe von formlosen Grafiken und Skizzen durchgeführt. Eingesetzt wird hier hauptsächlich ein Whiteboard oder Papier. Die Skizzen dienen nur der Kommunikation und werden nicht als Dokumentation aufbewahrt. Häufig wird auch direkt der Quellcode geschrieben und so lange überarbeitet bis er akzeptabel ist.

Da das gesamte System sehr komplex ist und viele technische Bestandteile beinhaltet existieren auch technische Stories. Dies kann z.B. durch die Aufteilung einer komplexen Story bzw. Epic entstehen. Dabei wird erst mit dem Abschluss der letzten zusammenhängenden Story ein Wert für den Kunden erzeugt. Aus diesem Grund ist es wichtig, dass diese nacheinander abgearbeitet werden und keine Stories zwischengeschoben werden. Wenn dies geschieht, wird die Entwicklung des Features nur in die Länge gezogen. Dadurch muss der Kunde länger auf eine Funktionalität warten.

Vor der Entwicklung findet eine Phase statt zu Sammlung von Anforderungen satt. Dazu finden viele Meetings, mit vielen Teilnehmer statt. An diesen Meetings nimmt der TD ebenfalls teil. Im Idealfall können mögliche Probleme, die einen Umbau der aktuellen Anwendung erfordern, so frühzeitig identifiziert werden. Hier soll nicht zu viel Zeit investiert werden. weil Die Phase würde nur unnötig in die Länge gezogen werden. Die Anforderungsphase kann bereits zwischen mehreren Monaten, bis hin zu Jahren dauern. Zu Beginn der Phase besprochene Dinge können beim Start der Realisierung bereits veraltet sein. Die eigentliche Entwicklung ist meist nach wenigen Tagen abgeschlossen. Probleme werden stattdessen beim Auftreten gelöst.

Absprachen zwischen den Teams, z.B. zu Schnittstellen, sind sehr individuell. Die meisten Teams arbeiten eng zusammen. Wenn eine Story im Detail ausgearbeitet wird und dabei Schnittstellen zu anderen Teams festgestellt werden, wird kurzerhand deren Business Designer dazu geholt. Dadurch können diese entsprechende Zeiten einplanen. In der Regel geht es hier nur darum wer was in welcher Reihenfolge erledigt. Da dieses Vorgehen sehr von den Teams abhängt, existieren manche Teams die stärken nach einem traditionellem Projektmanagement arbeiten und deswegen im Voraus genau wissen wollen in welchem Monat sie z.B. drei Tage Aufwand leisten sollen.

Der Technical Designer ist der Architekt im Team. Er dient hauptsächlich als Mentor und zum Coachen des Teams. Außerdem besucht er die Meetings während der Anforderungsphase und vertritt dort die Entwickler. Dadurch führt er viele Diskussionen mit Personen, die nicht zwingend Softwareentwickler sind. Da er an allem beteiligt ist kennt er das Gesamtsystem und hat eine Idee wie sich die Architektur entwickeln soll. Im Idealfall sollte er 50% der Zeit mitentwickeln. Durch die vielen Meetings ist es schwer längere Zeitblöcke am Stück Zeit zu haben. Für das Pair Programming ist dies zwingend erforderlich. Er ist außerdem ein Vertreter für das Team und macht deshalb Absprachen zur Weiterentwicklung des System. Theoretisch könnte er seine Aufgaben anderes definieren und vorgeben wie entwickelt werden soll. Das Ergebnis würde sich dadurch aber verschlechtern.

Ein Review alter Entscheidungen wird spätestens durchgeführt wenn diese etwas anderes behindern. Aus diesem Grund wird z.B. derzeit eine der Vertikalen neu gebaut. Diese ist langsam zu einem Monolithen gewachsen. Bei länger dauernden Umbauten muss dies mit dem Unternehmen abgesprochen werden. Wenn diese im Rahmen eines Sprints gemacht werden können wird dies ohne große Absprachen entschieden. Damit in dieser Zeit trotzdem Funktionalitäten realisiert werden, werden die Umbauten nicht am Stück durchgeführt sondern auf die Zeit neben der eigentlichen Weiterentwicklung verteilt. Das Monitoring des Gesamtsystems gehört zur Makro-

Architektur. Zu diesem Zweck wurde ein *Graphite*[2] Cluster als *SaaS* eingekauft. Jede Vertikale berichtet hierhin wichtige Metriken. Zusätzlich gibt es einen Log-Server, eine Statuspage und einen *Health-Check* pro Vertikale. Sobald z.B. ein Health-Check nicht mehr erfolgreich ist wird der *Load-Balancer* benachrichtigt, welcher als Folge diese Anwendung aushängt. Innerhalb der Teams werden im Allgemeinen keine weiteren Prüfungen integriert. Diese würden die Build-Pipeline und somit die Zeit verlängern bis ein Feature live ist. Dies wird maximal bei Bedarf durch die Werkzeuge innerhalb einer IDE gelöst. Nur wenn Probleme existieren werden entsprechende Werkzeuge eingesetzt. Nachdem diese gelöst sind werden sie wieder abgeschaltet. Die Microservices sind meistens so klein geschnitten, dass einzelne Analysen überflüssig sind. Eine Visualisierung der gesamten Systemlandschaft ist sinnvoll. Für eine allgemeine Übersicht aller Systembestandteile gibt eine eigenständige *Enterprise Architecture Management (EAM)* Abteilung. Diese pflegt manuell ein riesiges Netz aller Systeme. Dieses entspricht nicht mehr der Realität. Zum Teil fehlen vollständige Vertikale. Um dieses Problem zu beheben entstand die Idee dies automatisiert zu lösen. Jeder Microservice veröffentlicht bereits jetzt einen Statusbericht im *JSON* Format. Dieser beinhaltet z.B. Infos darüber wo er läuft, fehlgeschlagene Datenimporte und weitere Metadaten. Er könnte zusätzlich mit Informationen angereichert werden die angeben, welches Team zuständig ist, welche Beziehungen zu anderen Services existieren und praktisch alles was derzeit versucht wird manuell zu pflegen. Die einzelnen Services müssen anschließend täglich nach aktualisierten Informationen gefragt werden. Anschließend müssen die Informationen aggregiert und angezeigt werden. Dieses System ist bereits als Prototyp realisiert und in einigen Vertikalen integriert worden. Durch diese Umsetzung werden Meetings überflüssig in denen die Infos manuell ermittelt wurden. Es entsteht eine Dokumentation die auf der Wahrheit beruht und standardisiert ist. Der Realisierungsaufwand für die einzelnen Services hält sich in Grenzen, da dies für neue Services in den Erweiterungen für sämtliche Frameworks bereits integriert ist.

Wenn Microservices existieren von denen niemand mehr weiß was genau diese tun, kann unterschiedlich vorgegangen werden. Eine wichtige Informationsquelle sind die Tests und die UATs. Alternativ können diese entweder Testweise abgeschaltet werden, um anschließend zu prüfen was dadurch nicht mehr funktioniert. Außerdem kann geprüft werden, welche anderen Services auf diesen zugreifen. Eventuell sind diese noch bekannt und können dadurch Hinweise liefern.
Eine möglichst effiziente Einarbeitung findet durch Pair Programming mit einem erfahrenen Entwickler stattfinden. In der Regel hat der neue Mitarbeiter bereits am zweiten Tag den ersten Code live.
Um den Arbeitsprozess zu verbessern werden Retrospektiven durchgeführt. Allerdings ist die Optimierung des Prozesses tägliche Arbeit, die auch bei Treffen z.B. bei der Kaffeemaschine stattfindet.

Das Team hat ursprünglich mit Scrum begonnen. Scrum ist gut bei der Einführung der agilen Entwicklung. Es hilft dabei von einer projektorientierten Denkweise zu einer Produktorganisation zu kommen. Bei Scrum existiert das Problem, dass viele Vorgaben existieren. Wenn es Verbesserungsvorschläge gibt kommen schnell Einwände, weil dies nicht im Scrumbuch steht (z.B. falscher Zeitpunkt um darüber zu reden). Die Entwicklung wird in Form von Kanban durchgeführt. Eingesetzt wird der Zyklus der Aktivitäten wie die Retrospektiven oder Daily Meetings. Statt einem Deployment am Ende des Zyklus werden täglich viele Deployments durchgeführt. Eine Debatte über Story-Points findet nicht mehr statt. Story-Points werden nur verwendet um festzustellen, dass sich die Entwickler ungefähr einig sind was zu tun ist. Aus diesem Grund wurde erst vor kurzem das *Estimation* Meeting in *Understanding* umbenannt. Hier geht es nicht primär um das Schätzen der Stories, sondern um das Verstehen. Außerdem sind hierzu Verbesserungsvorschläge und Entscheidungen ob eine Story kleiner geschnitten werden muss wichtig. Diese Entscheidung wurde getroffen, weil Qualität nicht als Variable gesehen

[2] https://graphiteapp.org/, Abgerufen: 10.10.2016

wird, sondern nur der Umfang der Aufgabe. Wenn notwendig dauert das Erreichen der Qualität einen Tag länger.

Ausnahmen werden in diesem Unternehmen beim Test-First Konzept selten gemacht. Dies ist sehr wichtig, weil ansonsten ein erhöhtes Risiko entsteht, dass eine Komponente ausfällt. Eine Ausnahme wird nur gemacht, wenn die UATs und Akzeptanztests bei Beginn der Entwicklung fehlen. Etwas häufiger wird eine Ausnahme beim Pair Programming gemacht. Dies kann entweder sein, weil das Team aus einer ungeraden Anzahl von Personen besteht oder weil eine Person zu dem Zeitpunkt keine Motivation dazu besitzt. Wenn dies allerdings zu häufig auftritt befindet diese sich im falschen Team. Es handelt sich hierbei um keine 100% Regel. "'Regeln sind gut als Richtlinien, dürfen aber keine Festungsmauern werden."'

Als Schwachstelle in dem gesamten Vorgehen konnte das vorgelagerte Anforderungsmanagement identifiziert werden. Hier sind viele unterschiedliche Stakeholder mit verschiedenen Sichtweisen beteiligt. Die Verwaltung dieses Vorgehens ist sehr aufwendig. Teilweise nehmen an den Meetings bis zu 40 Personen teil. In dieser Phase existiert viel Optimierungsbedarf. Eventuell könnte dies ebenfalls mit Kanban durchgeführt werden.
Ein weiterer Punkt der genannt wurde ist, dass unklar ist was die langfristigen Ziele des Shops sind. Die Frage ob dieser den aktuellen Markt vollständig erobern, den Status Quo erhalten oder sogar große Konkurrenz versuchen soll anzugreifen muss geklärt werden.
Auf der technischen Seite existiert ebenfalls Verbesserungspotenzial. Derzeit gibt es keine *Service-Discovery*. Die unterschiedlichen Dienste müssen bekannt sein. Automatisch können Services derzeit nicht erkannt und gefunden werden. Wenn dies vorhanden wäre, kann z.B. ein Load-Balancing auf der Clientseite realisiert werden. Hilfreich ist dies auch für das ERM Team und deren automatische Analyse der Services.
Die Page Assembly sollte ebenfalls überarbeitet werden. Ursprünglich war diese nur zum Caching und zum Zusammensetzten der einzelnen Fragmente zu einer Seite gedacht. Inzwischen wurde hier immer mehr Logik implementiert. Diese sollte eher in einzelne performante und skalierbare Komponente ausgelagert werden. Der Page-Assembly nimmt langsam eine monolithischen Form an. Dadurch gibt es Niemanden mehr der sich mit den Feinheiten auskennt. Diese Komponente sollte wieder zu einer Standardkomponente werden.

Unternehmen H

Interviewort: Im Unternehmen

Interviewlänge: 51 Minuten

Entwicklungsart: Projektentwicklung

Entwicklungsvorgehen: Scrum

Teams: 11 Personen in einem Team (1 Designer, 1 PL, 2 Architeken), effektiv sind dies 8 Vollzeit Mitarbeiter

Unternehmensbeschreibung

Dieses Unternehmen entwickelt einzelnen Projekte. Hierbei kann es sich sowohl um beratende Unterstützung, wie auch um Entwicklung handeln.
Die Interviewpartnerin hat sich bereits ausführlich mit der Thematik beschäftigt. Sie hält hierzu häufig Vorträge und gibt Schulungen zur Architekturentwicklung.
Das betrachtete Projekte war für eine große Behörde aus Hamburg.

Interviewbericht

Das hier beschriebene Team arbeitet nach Scrum. Zum Teil fließen Praktiken aus Extreme Programming ein. Dies zeichnet sich dadurch aus, dass die gesamte Entwicklung durch Pair Programming stattfindet. Die Pair Programming Partner wechseln ständig. Spätestens nach der Entwicklungsdauer von drei Tagen pro Story wird eine Person ausgewechselt. Außerdem wird überprüft ob die Story weiter unterteilt werden kann. Während der Entwicklungszeit gibt es ein Standup am Morgen und eins zur Mittagszeit. Beim Sprint Planning wird das erste mal über die Umsetzung einer Story geredet. Außerdem wird vor der Entwicklung konkreter über die Umsetzung geredet. Je nach Komplexität diskutieren mehrere Teammitglieder gemeinsam oder nur das umsetzende Paar über die Umsetzung. Die Funktionalität wird im Anschluss im Review präsentiert. Code Reviews finden unregelmäßig auf Wunsch statt. Die Interviewpartnerin würde dies gerne regelmäßig einführen.
Zum Team gehört neben den Entwicklern ein Projektleiter (PL) und eine Designerin. Beide Rollen entwickeln mit. In den meisten anderen Teams ist kein Designer vorhanden. Der PL kümmert sich hauptsächlich um das Budget und ist die Kommunikationsschnittstelle zwischen dem Team und dem Kunden. Auf der Kundenseite sitzt der Project Owner (PO). Eine explizite Architektenrolle ist nicht vorhanden. Allerdings wird diese Rolle zum Teil durch erfahrene Entwickler übernommen. Diese werden häufig als Architekt bezeichnet. Mit diesen wird nach einem Sprint Planning meist über eine Lösungsstrategie geredet. Eine Ausnahme wird bei sehr simplen Stories gemacht. Bei diesen wird die Umsetzung vorher nicht besprochen.

Wenn etwas bei der Entwicklung auffällt, für die die Person in dem Moment keine Zeit hat, kann eine *Todo-Notizen* erstellt werden. Diese wird an eine bestimmte Stelle an die Wand geheftet

(Todo-Board).

Zwei Personen im Team übernehmen indirekt die Architektenrolle. Diese werden aber nicht als Architekt bezeichnet, übernehmen aber aufgrund deren Erfahrung einige der Aufgaben. Sie kümmern sich um die Abarbeitung der Todo-Tickets und besprechen diese eventuell mit der Projektleitung in Bezug auf die Verfügbare Zeit und das Budget. Die Todo-Notizen werden alle zwei Wochen in einer Architekturbesprechung vorgestellt. Hier werden außerdem technische Änderungen besprochen. Für Prozessänderungen wird die Retrospektive eingesetzt. Bei Problemen mit dem Prozess kann es zu spontanen, direkten Gesprächen kommen. Einer der Architekten behält das Gesamtsystem im Auge. Aus diesem Grund hat er z.B. bewirkt, dass nach seinem Urlaub eine Woche lang nur Refaktorisierungen durchgeführt wurden. Da dies nicht immer einfach mit dem Kunden zu vereinbaren ist wird dies nur in einem Teil der Entwicklungszeit durchgeführt. Die Aufgabe der Architekten ist es dafür zu Sorgen, dass die Qualität des Systems erhalten bleibt. Dazu muss er die nichtfunktionalen Anforderungen im Blick behalten und verstehen wie Lösungen für neue Anforderungen aussehen können. Im Normalfall entwickeln die Architekten auch im Paar. Ansonsten sind diese schnell die alleinigen Know-How Träger.
Entscheidungen die in Diskussionen getroffen wurden, werden vom Architekten per E-Mail an das Team verteilt. Dies ermöglicht eine historische Betrachtung und ermöglicht es zu verstehen, wie sich eine Architektur entwickelt hat. Ein teamübergreifender Austausch zwischen den Architekten existiert nicht, da der Großteil der anderen Teams kleiner ist und nur aus zwei bis drei Personen bestehen. In diesen Teams übernimmt der PL die Architektenrolle. Da dieser bereits viele zusätzliche Aufgaben hat, hat er nicht viel Zeit für weitere Diskussionsrunden. Durch regelmäßige Weiterbildungen und Konferenzbesuche soll dieser sich aber trotzdem weiterbilden können.

Ein teamübergreifender Austausch findet regelmäßig in Form von Vorträgen oder kleinen Workshops statt. Hier stellen Entwickler Themen vor, mit denen sie in einem Projekt viel gearbeitet haben. Durch die Aufarbeitung als Vortrag findet nochmals eine Selbstreflexion der Vortragenden statt. Damit dies stattfindet werden unterschiedliche Personen die mit dem gleichem Thema gearbeitet haben angesprochen. Diese sollen sich austauschen und das Ergebnis vorstellen.

Diskussionen im Team geschehen auf allen Ebenen. Dies kann z.B. eine Aufteilung des Systems in Elemente, Module und deren Schnittstellen sein. Bei diesen Diskussionen wird viel über Muster z.B. durch Services, Entitäten usw. geredet. Diese Diskussionsart soll in allen Projekten im Unternehmen eingesetzt werden. Auf lange Zeit ist es das Ziel eine ähnliche Architektur und Diskussionsebene in allen Projekten einzuführen. Die Diskussion mithilfe von Mustern beinhaltet eine hohe Lernhürde, sobald diese überwunden wurde ist die Diskussion einfacher.

Bei Problemen während der Entwicklung können die Architekten angesprochen werden. Diese kümmern sich anschließend darum oder sorgen dafür, dass eine Diskussion im Team stattfindet. Wenn notwendig halten diese Rücksprache mit dem PL. Für eine spätere Bearbeitung kann es vorkommen, dass technische Tickets erstellt werden.

Neue Mitarbeiter des Unternehmens erhalten eine ausführliche Einführung. Als Fortbildung zur Architekturentwicklung erhalten alle Mitarbeiter eine *iSAQB*[1] Schulung.

Ein Prozess um eine Entscheidung zu einem späteren Zeitpunkt nochmals zu evaluieren ist nicht vorhanden. Entscheidungen werden nur zu dem Zweck getroffen, um in eine Richtung weiterzuarbeiten. Diese kann jederzeit angepasst werden. Eine Überprüfung der Architektur findet regelmäßig mithilfe von Sonargraph Architect statt. Sie bezeichnet dies als *Mob-Architecting* da das gesamte Team vor einem großen Monitor sitzt und miteinander diskutiert. Dies stärkt das Team und Architekturverständnis. Da es kein Performance kritisches Projekt ist wird kein Monitoring der Hardwareressourcen durchgeführt. Statische Codeanalysen werden derzeit nicht

[1] http://www.isaqb.org, Abgerufen: 11.10.2016

eingesetzt. Die Interviewpartnerin ist der Überzeugung, dass auf Qualität trainierte Personen am besten im Fehler finden sind. Ein zusätzlicher Einsatz von Tools, wie Sonarqube kann aber zusätzlich hilfreich sein. Außerdem muss dem Kunden beigebracht werden, dass es normal ist, dass immer mal wieder etwas repariert werden muss.
Im Projekt wird sehr viel Wert auf eine hohe Testabdeckung als Sicherheitsnetz geachtet. Die Businesslogik muss eine Testabdeckung von mindestens 80% aufweisen. Dies ist für ein lang lebendes Projekt sehr wichtig. Nur so kann sichergestellt werden, dass bei Umbauten noch alles ordnungsgemäß funktioniert.

Als Dokumentation von Entscheidungen dienen die zuvor erwähnten E-Mails. Eine weitere Architekturdokumentation ist die Ausnahme. Ab und zu gibt es grobe statische oder Verteilungssichten Eine allgemeine Gesamtübersicht ist nicht vorhanden, weil die Architektur noch sehr lebendig ist. Die Architekten können den Aufbau allerdings skizzieren.

Die Einarbeitung neuer Mitarbeiter in existierende Systeme ist schwer und geschieht am besten mithilfe von Pair Programming. Außerdem kann die neue Person die alten Architekturenscheidungs-Emails durchlesen. Für die fachlichen Features existieren in vielen Fällen Präsentationen. Ursprünglich waren diese zur Vorstellung für die Anwender gedacht.

Aktuell wird überlegt die Teamgröße zu verkleinern und das Team in zwei kleinere Teams zu unterteilen. Dies wird umso dringender falls es weiter wächst. Ausnahmen werden z.B. bei internen Terminen des Scrum-Prozesses gemacht. Diese werden unter Umständen verschoben, wenn zu diesen Zeitpunkt Besprechungen mit den Kunden stattfinden sollen.

Kodierleitfaden

Kategorie	Definition	Kodierregel
geplante Diskussionen im Team	Alle Textstellen, die auf einen fest eingeplanten Zeitraum zur Diskussion im Team hinweisen.	Nur fest eingeplante, regelmäßige Zeiträume, keine spontanen Diskussionen.
spontane Diskussion im Team	Alle Textstellen, die auf spontane Diskussionen im Team hinweisen.	Diskussionen die kein regelmäßiger Bestandteil des Vorgehens sind.
Teamübergreifende Diskussionen	Alle Textstellen, die auf einen teamübergreifenden Austausch hinweisen.	Nur Informationen über den Informationsaustausch über die Teamgrenzen hinaus.
Aktivitäten während der Entwicklung	Alle Textstellen, die darauf hinweisen, was während der Entwicklung unternommen wird um eine hohe Qualität sicherzustellen.	Nur Aktivitäten, die bei oder direkt nach der Entwicklung einer einzelnen Entwicklungsaufgabe durchgeführt werden.
Problem-management	Alle Textstellen, die darauf hinweisen, wie Probleme während der Entwicklung gelöst werden.	Nur Probleme, die die Bearbeitung der aktuellen Entwicklungsaufgabe betreffen.
Architekturerosion	Alle Textstellen, die auf eine Möglichkeit hinweisen technische Schulden zu finden und anzusprechen.	Nur technische Schulden im Code. Hierbei kann es sich sowohl um Meetings und Diskussionsmöglichkeiten handeln, wie auch um eingesetzte Techniken und Tools.
Architektenrolle	Alle Textstellen, die auf eine gesonderte Architektenrolle und deren Einbettung ins Team hinweisen.	Nur Informationen auf die Existenz einer solchen Rolle und deren Integration im Team. Keine Information über dessen konkrete Aufgaben.
Architekt Aufgaben	Alle Textstellen, die auf die Aufgaben eines Softwarearchitekten hinweisen.	Nur Informationen über Personenrollen, die eine Spezialaufgabe zur Leitung der Architekturentwicklung haben.
Architektur-dokumentation	Alle Textstellen, die darauf hinweisen wie eine Architektur dokumentiert wird.	Nur Dokumentationen, die einen Überblick über die Bestandteile der Anwendung und deren Kommunikation liefern können. Keine Dokumentation von konkreten Codestellen.
Entscheidungs-dokumentation	Alle Textstellen, die auf ein Festhalten von Entscheidungen hinweisen.	Nur Entscheidungen, die relevant für die Architekturentwicklung sind und eine spätere Nachverfolgbarkeit ermöglichen.

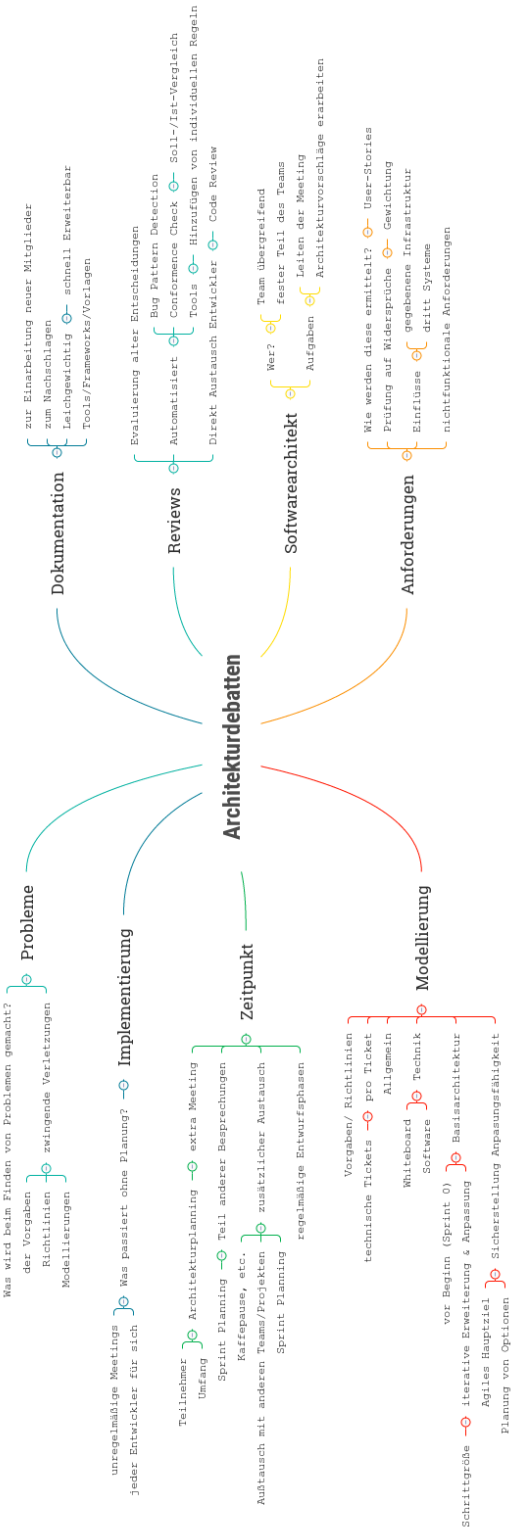


Figure 1: Themen für die Interviews

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 08.12.2016 Leon Fausten