Freie Universität Berlin

Masterarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Qualitätserlebnis statt Qualitätssicherung. Eine Mehrfachfallstudie agiler Teams

Holger Schmeisky

Matrikelnummer: 4602639

holger@schmeisky.com

Betreuer: Franz Zieris

Erstgutachter: Prof. Dr. Lutz Prechelt

Zweitgutachter: Prof. Dr. Ina Schieferdecker

Berlin, 27.03.2014

Agile Teams können hohe Qualität produzieren, ohne nachgelagerte Qualitätssicherung, aber mit einem starken Qualitätserlebnis. Ein starkes Qualitätserlebnis erlaubt es Teams durch schnelles und deutliches Feedback, Probleme schnell zu erkennen und sie zu beheben. Dieser Mechanismus ist das Ergebnis dieser explorativen Mehrfachfallstudie bei SoundCloud und Immobilienscout24. Des Weiteren stelle ich die Einflussfaktoren vor, die bestimmen, ob ein agiles Team ein starkes oder ein schwaches Qualitätserlebnis hat.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

27.03.2014

Holger Schmeisky

Inhaltsverzeichnis

1 Einführung **1**

2 Literatur **3**
2.1 Traditionelle Qualitätssicherung . . . 3
2.2 Agile Qualitätssicherung . . . 4
2.2.1 Scrum . . . 4
2.2.2 Kanban . . . 7
2.3 Automatisierte Tests . . . 7
2.4 Empirische Studien zu Qualitätssicherung . . . 8

3 Methodik **13**
3.1 Forschungsfrage . . . 13
3.2 Die Fallstudie als Forschungsmethode . . . 13
3.3 Studienentwurf . . . 14
3.3.1 Einfach- oder Mehrfachfallstudie? . . . 15
3.3.2 Unit of Analysis . . . 15
3.3.3 Logik & Interpretation . . . 15
3.3.4 Prinzipien der Datensammlung . . . 16
3.3.5 Sicherung der Validität . . . 16
3.4 Datensammlung . . . 17
3.4.1 Grober Aufbau . . . 17
3.4.2 Ethische Standards . . . 18
3.4.3 Auswahl der Fälle . . . 18
3.4.4 Fallstudienprotokoll . . . 19
3.4.5 Entwicklerbegleitung . . . 20
3.4.6 Dokumente . . . 21
3.4.7 Interviews . . . 21
3.5 Analyse . . . 23
3.5.1 Offene Fragen klären . . . 23
3.5.2 Validität der Daten . . . 24
3.5.3 Hypothesen entwickeln . . . 24

4 Die Fälle **25**
4.1 SoundCloud . . . 25
4.2 SoundCloud – Team Payment . . . 25
4.3 Immobilienscout24 . . . 26
4.4 Immobilienscout24 – Online Marketing . . . 27
4.5 Immobilienscout24 – Anbieten Profi . . . 28

5 Das Qualitätserlebnis **30**

6 Qualitätserlebnis in den Teams 35
6.1 Hohe Qualität erzeugen . . . 36
6.1.1 Interne Qualität . . . 36
6.1.2 Produktvision . . . 37
6.2 Qualität erkennen . . . 38
6.2.1 Gute, automatisierte Tests . . . 38
6.2.2 Automatisiertes Feedback . . . 41
6.2.3 Überwachung . . . 41
6.2.4 Automatisiertes Deployment . . . 41
6.2.5 Wenig Zeit bis Veröffentlichung . . . 42
6.2.6 Klare Grenzen in der Software . . . 43
6.3 Schicksal bestimmen . . . 43
6.3.1 Mitbestimmung . . . 44
6.3.2 Verantwortlichkeit für Veröffentlichung . . . 45
6.3.3 Rechenschaft . . . 45
6.3.4 Motivation . . . 46
6.3.5 Modularisierung auf Teamebene . . . 47
6.4 Effektivität QS bei Team Payment . . . 47
6.5 Firmenvorgaben . . . 48

7 Weitere Ergebnisse 50
7.1 Integrationstests . . . 50
7.2 Geringe Anzahl Kommentare . . . 50
7.3 Engineer in Test . . . 50
7.4 Separate Tester . . . 51
7.5 Motivation . . . 51
7.6 Bereitschaft . . . 52

8 Schluss und Ausblick 53

1 Einführung

Schnell neue Software veröffentlichen zu können ist in der schnelllebigen Internetwelt äußert wichtig, um auf die sich ständig ändernden Kundenbedürfnisse reagieren zu können. Für auf diesem Markt aktive Unternehmen, stellen klassische Entwicklungsprozesse mit nachgelagerten Testphasen unter Umständen ein großes Hindernis dar, da sie keine schnelle bzw. ad-hoc Veröffentlichung erlauben. Dieses Hindernis kann so groß sein, dass durch diese Verzögerung Firmen pleitegehen. Um flexibler zu sein, setzen viele Firmen auf agile Methoden. Allerdings scheint im Vergleich zu traditionellen Methoden, die Qualitätssicherung (QS) bei agilen Methoden nur wenig untersucht worden zu sein (wie im Laufe dieser Arbeit gezeigt wird). Es fehlt sowohl an etabliertem Wissen als auch an empirischen Untersuchungen, sodass Teams, die agil arbeiten wollen, kein bewährtes Vorgehen für die schnelle, aber sichere Entwicklung, vorliegt.

Agile Methoden stellen wesentliche Elemente etablierter, plan-getriebener Softwareprozesse wie dem Wasserfallmodell in Frage. Dazu gehört auch die Art der QS: In traditioneller QS sollen dedizierte Tester in einer der Entwicklung nachgelagerten Testphase die geforderte Qualität der Software gewährleisten. Dagegen müssen in agiler QS die Entwickler selbst die Qualität sicherstellen. Allein durch diesen Kontrast lassen sich viele Prinzipien der traditionellen Qualitätssicherung nur schwer anwenden, und es ist lohnenswert zu untersuchen, wie agile Methoden diese ersetzen.

In der Literatur sind bisher wenige Untersuchungen zur Qualitätssicherung (QS) in agilen Teams vorgestellt worden. In den relevanten Veröffentlichungen wird häufig betont, dass es Firmen nicht leicht fällt, Qualitätssicherung in agilen Methoden gut umzusetzen. Sicher ist jedoch, dass agile QS anders geplant und durchgeführt werden muss, als traditionelle QS. Der menschliche Faktor scheint dabei eine große Rolle zu spielen. Es existieren aber bisher weder ausgereifte Theorien noch empirische Hinweise, was zu guter oder schlechter QS im agilen Kontext führt. Insbesondere die Rolle des separaten Testers ist interessant, weil sie in traditionellen Methoden eine zentrale Rolle spielt, meist aber gar kein Teil von agilen Methoden ist. Beispielsweise berichtet eine Forschergruppe ([2]) davon, dass es zu Problemen führt, einfach diese Tester wegzulassen.

In dieser Arbeit stelle ich den Aufbau und die Ergebnisse einer explorativen Mehrfachfallstudie vor. Aufgrund der Lücken in der Forschung stellte ich die Forschungsfrage: **wie** die Qualitätssicherung in agilen Teams funktioniert, **ob** sie funktioniert und **warum** sie funktioniert. Dazu untersuchte ich in dieser Studie drei Teams der Firmen SoundCloud und Immobilienscout24.

Eines dieser Teams arbeitet nach einem Softwareprozess mit nachgelagerter Qualitätssicherung, in dem die Software in einer 1-wöchigen Testphase manuell getestet wird, bevor sie veröffentlicht wird. Zwei dieser Teams veröffentlichen ihre Software selbstverantwortlich mehrmals pro Woche. Diese beiden Teams können deutlich flexibler arbeiten und produzieren zufriedenstellende Qualität, obwohl sie gegen die Prinzipien der traditionellen QS verstoßen.

Im Gegensatz zu dem Team mit traditioneller Qualitätssicherung spüren die Entwickler die Auswirkungen der Qualität ihrer Arbeit sehr schnell, direkt und unmittelbar. Dadurch entdecken sie Probleme in der Qualität sehr schnell und reagieren darauf. Diese starke und schnelle Feedbackschleife für die Qualität der eigenen Arbeit trägt maßgeblich zum Erfolg der Qualitätssicherung in diesen Teams bei. Diesen Mechanismus und welche Faktoren ihn erzeugen, arbeitete ich aus den Daten heraus. Ich nenne diese Form der Qualitätssicherung ein **starkes Qualitätserlebnis**.

Die Arbeit gliedert sich wie folgt: Zuerst wird die relevante **Literatur** vorgestellt und offene Fragen werden herausgearbeitet. Anschließend werden die Gründe für die Durchführung einer explorative Fallstudie, diese **Methodik** und der Entwurf der Studie vorgestellt. **Das Qualitätserlebnis** ist das wichtigste Ergebnis dieser Arbeit. Um das Ergebnis zu illustrieren, werden die Ausprägungen des **Qualitätserlebnis in den Teams** vorgestellt. **Weitere Ergebnisse**, die nicht so zentral wie das Qualitätserlebnis sind, werden als Letztes vorgestellt.

2 Literatur

In der Software Engineering Literatur werden grundsätzlich zwei Ansätze zur Qualitätssicherung (QS) in der Softwareentwicklung unterschieden: Traditionelle QS und agile QS. Der größte Unterschied zwischen beiden Ansätzen ist, dass traditionelle QS Entwicklung und Testen trennt, während agile QS sie integriert. Weil bei der agilen Entwicklung die Qualitätssicherung mehr im gesamten Prozess integriert ist, werden die agilen Prozesse Scrum und Kanban vorgestellt. Anschließend werden empirische Studien zur Qualitätssicherung in Softwareteams ausgewertet.

2.1 Traditionelle Qualitätssicherung

Traditionelle Methoden der Qualitätssicherung betonen eine strikte Trennung zwischen Entwicklungsaktivitäten auf der einen und Qualitätssicherungsaktivitäten auf der anderen Seite. Sie werden in dieser Arbeit als traditionelle Methoden bezeichnet, weil sie weit verbreitet in der Softwareentwicklung sind und auf etablierten, lange bestehenden Prinzipien basieren.

Der Ausdruck **traditionelle QS** ist kein gebräuchlicher Begriff in der QS Literatur. Die Autoren, die agile Methoden untersuchen, benutzen ihn aber um die agilen QS Methoden von den bereits existierenden abzugrenzen. Zum Beispiel schreiben TALBY ET. AL. *„Agile development completely redefines quality assurance work [...] making some traditional QA responsibilities and outputs irrelevant“* ([1, S. 30]). In diesem Zitat beziehen sich die Autoren mit dem Begriff *„traditionelle QS“* auf Entwicklungsprozesse mit getrennten Entwicklungs- und Testteams, die in der von ihnen untersuchten Organisation etabliert waren. In diesen Prozessen liefern Teams *„vollständige“* Programme an die QS-Teams. KETTUNEN ET. AL. schreiben *„The software development process is based on traditional plan-driven methods [...] or agile development methods [...]“* ([2, S. 231]), und stellen dabei lineare und iterative Methoden gegenüber. ITKONEN ET AL. führen aus, *„[...] agile methods seem to lack aspects that traditionally are considered important and fundamental to successful quality assurance.“* ([3, S. 1]) und charakterisieren traditionelle QS als etabliertes Qualitätssicherungswissen.
Um zu definieren, woraus traditionelle Qualitätssicherung besteht, benutze ich die Charakterisierung traditioneller QS von ITKONEN ET AL. in [3] und das Standardwerk *„The Art of Software Testing“* ([4]) von MYERS ET AL.. Das Buch wurde zuerst 1977 und eine zweite Edition in 2011 publiziert. Google Scholar zählt ca. 3500 Referenzen auf dieses Buch.

In traditioneller QS ist Testen die wichtigste Aktivität. MYERS definiert sie als *„the process of executing a program with the intent of finding errors.“* ([4, p.11]). Um ein Programm erfolgreich zu testen, ist eine **destruktive** Haltung nötig. Entwicklern wird diese destruktive Haltung in der Regel nicht zugesprochen, da sie vermutlich demonstrieren wollen, dass ihr Programmm so funktioniert, wie es soll. MYERS empfiehlt sogar *„a*

programmer should avoid attempting to test his or her own program" ([4, S. 17]). Traditionelle QS empfiehlt daher **unabhängige** Tester. Da Testen ein *„extremely creative and intellectually challenging task"* ([4, S. 20]) ist, sollten diese unabhängigen Tester **bestimmte Fähigkeiten** besitzen, die anders sind als die Fähigkeiten eines Entwicklers. Die Ergebnisse dieser QS sollten benutzt werden, um die erreichte Qualität zu bewerten. Idealerweise geschieht dies mit **Metriken**, wie Anzahl gefundener Fehler, Fehlerarten und Testabdeckung ([3, S. 3]).
Damit das Testen unabhängig vom Entwickeln geschehen kann, ist eine geschriebene Spezifikation nötig. Daher passt traditionelle QS gut zu plangetriebenen Prozessen, die Wert auf ausführliche Spezifikationen legen. In diesen Prozessen mit aufeinanderfolgenden Phasen ist das Testen meistens eine **separate Phase** nach der Entwicklung.

2.2 Agile Qualitätssicherung

Es gibt keine einheitliche **agile Qualitätssicherung**, daher wird erst das agile Manifesto, auf dem agile Entwicklugsmethoden beruhen, vorgestellt und anschließend die zwei agilen Entwicklungsprozesse Scrum und Kanban mit ihren qualitätsichernden Elementen. Zuletzt werden empirische Studien zu agiler Qualitätssicherung ausgewertet.

Agile QS-Prozesse zeichnen sich dadurch aus, dass sie auf den Werten und Prinzipien des Agilen Manifests [5] beruhen. Aus den Prinzipien dieses Manifests, ergeben sich bereit Konsequenzen, die sich mit traditioneller QS nur schwer vereinen lassen.
Das agile Manifest Manifest stellt die Generierung von Mehrwert für den Kunden über die Einhaltung von vordefinierten Entwicklungsprozessen- und Plänen. Die höchste Priorität für agile Entwicklung ist, *„to satisfy the customer through early and continuous delivery of valuable software"* [5, S. 3]. Die Qualität des Produkts muss **früh** und **oft** beurteilt werden. Das lässt eine Test- und Integrationsphase nach der Entwicklung ungeeignet erscheinen.
Agile Entwicklung *„welcome[s] changing requirements, even late in development"* [5, S. 3], wobei Änderungen durch **Kundenfeedback** gemeint sind. In agiler Entwicklung ist die **direkte, persönlich Kommunikation** der wichtigste Kommunikationskanal, weil sie effektiver ist als geschriebene Dokumentation. *„Business people and developers work together daily throughout the project"* [5, S. 3], da angenommen wird, dass sich die Kundenanforderungen zusammen mit dem Produkt weiterentwickeln.
Aus diesen Gründen existiert in einem agilen Softwareprozess oft keine vollständige, geschriebene **Spezifikation** der Software. Das korrekte Verhalten der Software ist nur in den Köpfen der Anforderer und Entwickler vorhanden. Ohne diese geschriebene Spezifikation ist es aber unmöglich, das Testen vom Entwickeln zu trennen.

2.2.1 Scrum

Scrum ist ein populäres agiles Projektmanagementframework, dass in vielen Firmen eingesetzt wird. Das Framework selber schreibt keine konkreten Entwicklungstechni-

ken vor, enthält aber viele Elemente, die für die Qualitätssicherung relevant sind. Es basiert auf 3 Säulen:

- **Transparenz**: Alle wichtigen Aspekte des Prozesses müssen für alle Beteiligten sichtbar sein. Dafür müssen diese klar definiert sein, damit es ein gemeinsames Verständnis gibt.
- **Inspektion**: Die Anwender von Scrum beobachten den Fortschritt des Produkts permanent, um unerwünschte Abweichungen festzustellen.
- **Adaption**: Wird eine zu große Abweichung des tatsächlichen Zustands vom Geplanten festgestellt muss so schnell wie möglich der Prozess oder das Material angepasst werden.

Scrum beschreibt verschiedene Rollen im Softwareentwicklungsprozess. Ein Scrum-Team besteht aus Product Owner, Entwicklern und einem Scrum Master und ist selbstorganisierend und funktionsübergreifend. Der **Product Owner** (PO) ist dafür zuständig, dass das Team möglichst viel Wert (für die Firma) erzeugt. Dafür formuliert und priorisiert er die Anforderungen. Der **Scrum Master** coacht die Entwickler darin, ihren Prozess selber zu gestalten, z.B. indem sie Praktiken und Artefakte richtig einsetzen und beseitigt externe Behinderungen, z.B. eine langsame Build-Umgebung. Die **Entwickler** erledigen die Arbeit. Das Team hat die Macht und Verantwortung, seine Arbeit selber zu organisieren. Jeder im Team ist für das gesamte Produkt verantwortlich, nicht nur für einen Teilbereich. Es gibt keine unterschiedlichen Rollen wie dedizierte Architekten oder Tester.

Die wesentliche Elemente eines Scrum-Entwicklungsprozesses sind:

- **Sprints**: Eine Iteration fester Länge (1-4 Wochen) an deren Ende immer ein potenziell auslieferbares Produkt steht.
- **Product Backlog**: Eine priorisierte Liste aller Arbeit, die an dem Produkt nötig wird. Es ist kein starr festgelegter Plan, sondern kann flexibel an Änderungen, die sich mit der Zeit ergeben, angepasst werden. Der Product Owner fügt *„Backlog Items“* (oder Punkte) hinzu, priorisiert sie und arbeitet sie aus. Er enthält neben Features auch Fixes, technische Verbesserungen und alle anderen Arten von Änderungen am Produkt. Punkte müssen nicht sofort bearbeitbar sein, denn es wird zwischen bereiten (*„Ready“*) Punkten und nicht bereiten Punkten unterschieden.
- **Sprint Planning**: Nach jedem fertigen Sprint planen PO und Entwicklerteam die Arbeit für den nächsten Sprint. Der PO stellt seine gewünschten Ziele für diesen Sprint vor und mit welchen Product Backlog Punkten sie erledigt werden können. Das Team wählt die Punkte aus dem Product Backlog, von denen es ausgeht, dass es sie im Sprint erledigen kann, nimmt sie in den Sprint Backlog und plant, wie es sie im Sprint erledigt (durch weitere Aufteilung der Aufgaben).
- **Sprint Backlog**: Der Sprint Backlog beschreibt während des Sprints die vom Team

noch zu erledigende Arbeit, um das Sprintziel zu erreichen. Der Sprint Backlog wird vom Entwicklerteam um neue Arbeit erweitert, um erledigte Arbeit gestrichen oder um unnötig gewordene Arbeit reduziert. Der Sprint Backlog setzt eine von allen Beteiligten verstandene *„Definition of Done"*, d.h. ein Verständnis die notwendigen Schritte, um ein Element des Sprint Backlog abzuschließen.

- **Daily Standup**: Jeden Tag bespricht das gesamte Entwicklerteam in einem 15-Minuten Meeting welche Arbeit erledigt wurde und welche Arbeit noch zu erledigen ist. Dadurch koordinieren sich die Teammitglieder und der Sprint Backlog wird angepasst.
- **Sprint Review**: Nach jedem Sprint wird mit allen Teammitglieder (evtl. auch externen Interessenten) die erledigte Arbeit präsentiert und am Produkt demonstriert. Außerdem, auf das Produkt bezogene, positive und negative Eindrücke diskutiert.
- **Retrospektive**: Hier wird der Prozess im letzten Sprint ausgewertet. Es wird was positiv und was negativ aufgefallen ist, in Bezug auf Menschen, Beziehungen, Prozesse und Werkzeuge. Potenzielle Verbesserungen werden ausgearbeitet und ein Plan wird erstellt, um diese im nächsten Sprint umzusetzen.
- **Produktvision**: Als Produktvision wird eine möglichst anschauliche, prägnante Zusammenfassung des Ziels des Teams, d.h. des Produkts das sie erstellen wollen, bezeichnet. Sie hilft den Entwicklern, Entscheidungen und Priorisierungen beim Entwickeln selbstständig im Sinne der Produktvision zu treffen.

Das wichtigste an diesen Elementen ist, dass ihre Ergebnisse für das Team sichtbar sind: Der Product und der Sprint Backlog sind leicht einsehbar und im Sprint Planning ist das gesamte Entwicklerteam beteiligt, nicht nur einige wenige Personen. Meist wird der Sprint Backlog als große Tafel in einem Raum visualisiert, auf der Karten für die Aufgaben kleben.

Viele Elemente von Scrum haben eine Bedeutung für die Qualität der Software: Am Ende des Sprints muss lauffähige Software vorliegen, so ist die Feedbackschleife zwischen Programmieren und Ausführen relativ kurz (nur 1 Sprint lang). Das Team ist in die Planung einbezogen (Sprint plannings mit Entwicklern, Schätzungen werden vom Team getroffen) und kennt so besser die Anforderungen. Die Planung beinhaltet nicht nur Features, sondern auch Verbesserungen der internen Qualität. Daily Scrums, Retrospektiven und transparente Aufgaben fördern die Kommunikation im Team und verhindern Einzelgängertum, das zu Fehlern führen könnte. Letztlich ist Die Definition of Done die primäre Stellschraube für Qualität, denn hier kann sehr leicht mehr Qualität oder mehr qualitätssichernde Maßnahmen gefordert werden.

2.2.2 Kanban

Kanban ist ein Managementansatz und befasst sich mit dem **Fluss** von Aufgaben durch eine Wertschöpfungskette. Es umfasst u.A. Prinzipien des Management und Methoden für Change Management. Teams, die nach Kanban arbeiten verwenden meistens nur folgende zwei Elemente aus Kanban und viele aus Scrum:

- Das **Kanban-Board**: Visualisierung des Prozesses durch Spalten und Karten. Eine Karte entspricht einer Aufgabe, eine Spalte einer Station im Prozess.
- Das **WIP-Limit**: Jede Spalte hat ein Limit an Karten die sie enthalten kann (z.B. 1 pro Entwickler). Wird ein Platz, zieht sich diese Spalte eine Karte aus der vorherigen Spalte (*„pull“* statt *„push“*)

Ein agiles Team das Kanban einsetzt, unterscheidet sich von einem Scrum-Team hauptsächlich in dem Verzicht auf feste Iterationen (Sprints).

2.3 Automatisierte Tests

Automatisierte Tests spielen in agiler Qualitätssicherung eine wichtige Rolle. Scrum und Kanban schreiben sie nicht vor, weil sie keine Entwicklungstechniken vorschreiben. Um aber die vom Manifest geforderte Flexibilität für Änderungen zu erreichen, sind automatisierte Tests ein gutes Mittel. In der Literatur werden folgende Arten von Tests unterschieden (aus [7]):

- **statische Tests**: Tools zur Codeanalyse laufen automatisch und liefern Metriken über den Code (z.B. Schachtelungstiefe)
- **Unittests**: Prüfen isoliert vom Rest des Systems einen Baustein.
- **Integrationstests**: Prüfen automatisiert das Zusammenspiel mehrerer Bausteine innerhalb eines Moduls (z.B. alle eigenen Klassen, aber ohne Datenbankanbindung).
- **Systemtests**: Prüfen die gesamte Wirkungskette, meistens über Interaktion mit der Benutzeroberfläche und in Zusammenwirkung mit anderen beteiligten Systemen auf einer produktionsnahen Umgebung.
- **explorative Systemtests**: Ein Tester (ein Teammitglied) testet ein Feature und überprüft durch spontanes Ausprobieren neuer Pfade, ob die Applikation sich auch bei ungewöhnlichem Verhalten, wie gewünscht verhält.
- **Akzeptanztests**: Mit diesen Tests überprüft der Product Owner, ob seine Anforderungen erfüllt wurden.

Mocking bezeichnet das Ersetzen von Objekten in automatisierten Tests durch andere Objekte, wodurch das Testen u.U. leichter fällt. Es gibt verschiedene Arten des Mockings:

- **Stub**: identische Schnittstelle, die vordefinierte Werte zurückgibt.
- **Mock**: Stub, der nicht nur Daten zurückgibt sondern auch bestimmte Aufrufe und Daten erwartet.
- **Fake**: Ersetzt Objekt durch einfache Implementierung.
- **Dummy**: Verwende kein echtes Objekt (z.B. *null*).

Continuous Integration (CI) bedeutet, dass jede Codeänderung dazu führt dass die Software neu gebaut und alle Tests ausgeführt werden. CI wird durch dedizierte Software, die i.d.R. auf einem **CI Server** installiert ist, sichergestellt bzw. den Entwicklern zur Verfügung gestellt. Voraussetzung für den Einsatz von CI ist ein zentrales Code-Verzeichnis, auf das der CI-Server zugreifen kann. Nach einem commit in dieses Verzeichnis wird die Software kompiliert / gebaut. Anschließend werden alle automatisierten Tests auf allen Ebenen (statisch bis Systemtests) durchgeführt. Das Ergebnis ist für das Team gut sichtbar (Benachrichtung auf dem Desktop, Email bei Fehlschlag, Ampel, ...). Die CI sollte innerhalb weniger Minuten das Ergebnis liefern, damit die Entwickler ihr Änderungen möglichst oft prüfen können.

2.4 Empirische Studien zu Qualitätssicherung

Im folgenden werden existierende empirische Studien zum Thema QS (traditioneller und agiler) dargesellt. Die Darstellung dieser Studien wird zeigen, dass es dort große Lücken gibt, insbesondere was die tatsächliche Durchführung von QS in Softwareteams betrifft. Es wird sich zeigen, dass agile QS zwar gut funktionieren kann und Vorteile gegenüber traditioneller QS bietet, agile QS aber schwer umzusetzen ist.

In *„ 'Good' Organisational Reasons for 'Bad' Software Testing"* untersuchen MATIN ET AL. ([8]) in einer ethnographischen Studie die Testpraktiken in einem Softwareunternehmen. Die Autoren betonen, dass es einen großen Unterschied zwischen der QS-Forschung und Praxis gibt, und dass es nur wenige empirische Daten zum Testen in echten Teams gibt. Ihrer Meinung nach, ist die Testforschung mehr darauf bedacht, die technischen Voraussetzungen für das Testen zu verbessern. Aus ihrer Untersuchung schließen sie aber, dass es in der Praxis wichtiger ist, wie Tests so gestaltet werden können, dass sie die Anforderungen der Organisation erfüllen und gleichzeitig den Aufwand dafür zu minimieren. Die Testforschung sollte sich also mehr der Beziehung zwischen Testen und Organisation widmen. Dies sollte insbesondere durch empirische Untersuchungen von QS-Prozessen geschehen.

AHONEN ET AL. untersuchen in *„Impacts of the organizational model on testing"*([9]), wie verschiedene Organisationsmodelle der Testgruppe in einer Firma die Art des Testens selbst stark beeinflussen. Sie beschreiben als Organisationsformen Tester im Team, Tester als separate Abteilung und Tester als Ressourcenpools, die nach Bedarf an Projekten arbeiten. Insbesondere im Ressourcenpool Modell beobachten sie wenig Vertrauen zwischen Entwicklern und Testern. Sie schließen, dass die organisatorischen Probleme

beim Testen schwieriger sind als die der Testinfrastruktur, aber nicht so schwer wie das Design der Testfälle selber.

GUO ET AL. begründen in ihrer Studie *„Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows“* ([10]), dass, ein Defekt umso weniger wahrscheinlich gelöst wird, je weiter entfernt (geographisch und organisatorisch) der Ersteller eines Defektberichts von dem, dem er zugewiesen wird. In ihrer Studie wollten sie untersuchen, ob es eine Verbindung gibt, zwischen der Wahrscheinlichkeit mit der ein Defekt gelöst wird und dem Zuweiser. Um sie zu beantworten analysierten sie die Windows Vista Defektdatenbank und führten eine Umfrage unter den Entwicklern durch. Die Defektdaten zeigten, dass ein Defekt weniger wahrscheinlich gelöst wird, wenn er von jemandem mit einem anderen Manager, von einem anderen Gebäude oder aus einem anderen Land zugewiesen wird. Diese Beziehung wurde durch die Umfrageantworten bestätigt. Die Validität der Egebnisse gründet sich auf die relative große Defektdatenbank und die unterstützende Untersuchung durch Umfragen.

MÄNTILÄ ET AL. argumentieren in ihrer Studie *„Who tested my software? Testing as an organizationally cross-cutting activity“* ([11]), dass in einer Organisation viele verschiedene Personengruppen Software testen und dass spezialisierte Tester meist nur weniger wichtige Defekte finden. Es handelt sich um eine explorative Fallstudie von 3 Softwarefirmen, die Ingenieurs- und Geschäftssoftware produzieren. Um zu untersuchen, wer die Software testet, führten sie Interview mit Mitarbeitern durch, analysierten Dokumente aus dem Entwicklungsprozess, analysierten die Defektdatenbank und ließen ihre Schlüsse von Managern überprüfen.
Die Autoren schlossen, dass viele verschiedene Gruppen, wie Verkauf, technischer Support und Benutzer, Defekte in der Software finden und melden. Die Autoren begründen das damit, dass viel Domänenwissen nötig ist, um zu bestimmen, ob die Software richtig funktioniert oder nicht. Sie unterstützen diese These damit, dass die spezialisierten Tester die Defekte mit der niedrigsten Wichtigkeit und der niedrigsten Fixrate fanden. Hierbei ist allerdings zu beachten, dass die Wichtigkeit subjektiv gewählt wird und die Tester vielleicht nur am bescheidensten waren. Die Manager der Organisatoren bestätigten allerdings dieses Phänomen und fügten hinzu, dass aus ihrer Erfahrung, spezialisierte Tester zwar viele Defekte finden können, dass die Relevanz für Endnutzer aber nicht immer hoch ist. Daraus schließen die Autoren, *„Specialized testers are more likely to find and report defects that are unlikely to occur in real use or those that have only minor effects.“* ([11, S. 165]). Zusammen mit den Ergebnisse von GUO ET AL., schlagen die Autoren sogar die Hypothese vor, dass *„lowest fix ratios of defects are found by individuals who are testing specialists with a large organizational distance“* ([11, S. 166]).
Die Ergebnisse dieser Studie sind wahrscheinlich valide, weil die Autoren ihr Fallstudiendesign sehr klar schildern. Sie stellen viele mögliche Einschränkungen der Validität explizit dar und beschreiben, wie sie ihnen von vornherein entgegen wirken. Schließlich bezeichnen MÄNTILÄ ET AL. die aktuelle Menge an Berichten über Softwaretesten in der Praxis ebenfalls als *„shallow“* ([11, S. 11]).

TALBY ET AL. führten in *„Agile software testing in a large-scale project“* ([1]) eine explora-

tive Fallstudie des ersten Extreme Programming Projekts bei der Israelischen Luftwaffe durch. Sie stellen dar, dass die Qualität und Produktivität des Projekts sich gegenüber vergleichbaren Projekten mit traditioneller QS verbesserte. Das untersuchte Team entwickelte ein Enterprise Informationssystem, das als *„critical to daily operations“* bezeichnet wurde ([1, S. 30]) und in ihm arbeitete ein dedizierter Tester. Die Forscher analysierten Projektdaten, Dokumente, beobachteten Teamreflektionsmeetings und führten Umfragen und Interviews unter den Entwicklern durch.
Die Autoren bezeichnen die organisatorischen Änderungen, um einen agilen Prozess einzuführen als *„radical“* [1, S. 30]. Sie identifizierten als Hauptproblem *„How an organization promotes developer testing [...]“* [1, S. 32]. Der wichtigste Erfolgsfaktor war in diesem Fall, dass nur automatisiert getestete Features als fertiggestellt gezählt wurden. Wenn sie funktionierten, es aber keinen Test für sie gab, wurden sie nicht als fertig angesehen. Hauptsächlich durch diese Metrik wurde erreicht, dass die Entwickler Tests für ihren gesamten Code schrieben.
Diese Vorgehensweise hatte zwei Vorteile: Zum einen wurden viele Fehler früher als in vergleichbaren Projekte gefunden. Zum anderen gestalteten die Entwickler ihren Code testbar, wodurch der Aufwand zum Testen sank. Dieser Effekt konnte ist plausibel, konnte in der Studie jedoch nicht quantitativ belegt werden.
Anfangs arbeitete der dedizierte Tester nach der Spezifikation und entdeckte nur kleinere Versagen, die oft Spezifikationsfehler, statt Fehler in der Software waren. Das Management ermutigte den Tester anschließend, paarweise mit den Entwicklern zu arbeiten und die Tests parallel zum Entwickeln zu schreiben, anstatt nach dem Entwickeln.
Das Team sammelte Metriken über ihre Arbeit, z.B. die Defektlebenszeit, die die Forscher auswerten konnten. Die Entwickler wendeten im Gesamtdurchschnitt (arithmetischer Mittelwert) nur etwas mehr als eine Stunde auf, um einen Defekt zu reparieren (Zeit die ein Entwickler daran arbeitet). Während der gesamten Beobachtung betrug die durchschnittliche Defektlebensdauer nur etwas mehr als eine Woche (Zeit von Eröffnen bis Schließen des Defekts). Das untersuchte Team konnte, auch mit steigender Softwarekomplexität, in jeder Iteration alle offenen Defekte zu lösen. Diese Zahlen waren um eine Größenordnung besser, als in vergleichbaren Projekten mit traditioneller QS in der selben Organisation.
Weil in der Studie eine Vielzahl von Untersuchungsmethoden (Interviews, Umfragen, Beobachtungen und Projektdaten) eingesetzt wurde, ist die Validität der Ergebnis als hoch einzuschätzen.

KETTUNEN ET AL. führten in *„A study on agility and testing processes in software organizations“* ([2]) eine Fallstudie der Testprozesse und -aktivitäten in 12 Organisationen, die Software entwickeln, durch. Sie fanden heraus, dass agiles Testen zwar schwierig umzusetzen ist, aber die Flexibilität bei Änderungen erhöht und verhindert, dass zu wenig Zeit und Resourcen für das Testen geplant werden.
Die Forscher benutzten den Grounded Theory Ansatz, um die Softwaretestpraktiken in den Firmen zu untersuchen. Sie wollten untersuchen, welche Auswirkungen agile Entwicklung auf Testprozesse und -aktivitäten hat. Dazu führten sie 3 Runden von Interviews in den Firmen, jeweils mit einem Designer, einem Manager und einem Tester.

Zusätzlich wurden strukturierte Umfragen unter den Mitarbeitern durchgeführt.
Die Forscher untersuchten im Gegensatz zu TALBY ET AL. nicht den gesamten agilen Prozess sondern konzentrierten sich auf die Effekte, die die Anwendungen einiger agiler Prinzipien auslösen. Acht der zwölf untersuchen Organisationen wenden weniger als zwei agile Praktiken an, z.B. nur Unit Tests ([2, Tabelle 1]).
Aus den Ergebnisse folgern sie, dass das frühe Feedback durch agiles Testen wertvoller ist, da Defekte in frühen Entwicklungsphasen effizienter gelöst werden können als in späten Phasen. Außerdem waren die Teams leichter in der Lage, das Ziel der Entwicklung anzupassen. Durch die frühe Involvierung kennen die Tester das Programm besser und können effizienter Testen. Insgesamt werden gleich viele Testressourcen benutzt, aber auf eine gleichmäßigere und vorhersagbare Art, weil das Testen nicht erst nach dem Entwickeln startet, wie in den plangetriebenen Prozessen, die beschrieben wurden. In letzteren kann es passieren, dass Software mit bekannten Defekten ausgeliefert wird, weil das Testen reduziert oder ausgelassen wird, wenn eine Deadline eingehalten werden muss. Dieses Problem wird durch agile Testen gemildert, weil das Testen früh anfängt und parallel zum Entwickeln durchgeführt wird.
Die Autoren stellten folgende Probleme von agilem Testing fest. Zunächst führt agiles Testen häufig zu fehlender Dokumentation. Des Weiteren kann es in Organisationen ohne dedizierte Tester passieren, dass Entwickler nicht testen, weil es nur eine zusätzliche Aufgabe zum Entwickeln von Features ist. Das größte Problem, dass sie identifizierten war, wie schwierig es ist, Tester erfolgreich parallel zu Entwicklern arbeiten zu lassen. Sie schließen damit, zu behaupten *„successfully incorporate testing into agile methods, the development organization is forced to think and revise their testing processes in detail“* ([2, S. 239]).

Insgesamt gibt es nicht viele empirische Untersuchungen über Qualitätssicherung (traditioneller und agiler) in echten Softwareteams. Die Untersuchungen über agile Teams beschreiben größtenteils Prozesse mit separaten Testern, aber keine über Teams, die von absichtlich ohne Tester arbeiten. In den existierenden Untersuchungen wurden außerdem zahlreiche offene Fragen aufgeworfen. Z.B. wie die Organisationsform mit der Effektivität der Qualitätssicherung zusammenhängt.
Die vorhanden Ergebnisse deuten aber, wie oben beschrieben, auf eine Vorteilhaftigkeit von agilem Testen gegenüber traditioneller QS hin. So stellen sich traditionelle Prinzipien weniger effektiv dar. Dedizierte Testteams finden Defekte von geringerer Relevanz und mit schlechterer Lösungswahrscheinlichkeit. Außerdem kann es unter Umständen zu schlechter Stimmung und wenig Vertrauen zwischen Entwicklern und Testern kommen, was nicht förderlich für die Qualität ist.
Von agilen Teams gibt es nur sehr wenige empirische Untersuchungen der QS. Oft sind die untersuchten Teams keine richtig agilen Teams und benutzen nur 1-2 Praktiken aus XP. Es scheint einige Erfolgsfaktoren zu geben, die vor allem die Geisteshaltung der Entwickler betreffen, z.B. ob ungetestete Features als fertig gezählt werden oder nicht. Einfach keine Tester zu haben kann dazu führen, dass Entwickler Tests überspringen und die Qualität sinkt. Es wird deutlich, dass es für Firmen schwierig ist, agile QS richtig umzusetzen und es nicht klar, was dabei zu einer guten Umsetzung führt.

Wird es aber richtig umgesetzt, ist agile QS effektiver als vergleichbare traditionelle QS. So wird weniger Zeit für Fehlerbehebung aufgewendet und Fehler werden schneller behoben. Außerdem ist agile QS flexibler anpassbar an sich ändernde Anforderungen. Vielleicht am wichtigsten ist, dass agile QS besser verhindert, dass zu wenig Zeit und Ressourcen für das Testen eingeplant werden, und das Testen aufgrund einer Deadline ausgelassen oder reduziert wird.

3 Methodik

In diesem Abschnitt stelle den Entwurf der explorativen Mehrfachfallstudie und die Methoden zur Datensammlung und Analyse vor.

3.1 Forschungsfrage

Die im vorherigen Kapitel durchgeführte Literaturanalyse belegt also: (1) es gibt große Lücken in der Erforschung agiler QS, (2) agile QS ist lohnenswert, aber schwierig umzusetzen, und (3) es gibt einige Erfolgsfaktoren, die von Einstellung der Entwickler abhängen. Daher lege ich der vorliegenden Arbeit die folgende Forschungsfrage zugrunde:

> Wie und unter welchen Bedingungen funktioniert agile Qualitätssicherung?
> Wenn sie gut funktioniert, warum funktioniert sie?

3.2 Die Fallstudie als Forschungsmethode

Bei Forschungsfragen dieser Art ist das Phänomen nicht klar vom Kontext trennbar. Zum Beispiel können die folgenden Faktoren jeweils Kontext oder Phänomen sein:

- Firmenkultur - funktioniert agile QS in bestimmten Firmen gut, weil sie eine bestimmte Firmenkultur haben?
- Anforderungen - funktioniert agile QS nur bei bestimmten Produkten gut, weil sie bestimmte Anforderungen haben?
- Team - funktioniert agile QS nur bei bestimmten Teams gut, weil sie eine bestimmte Zusammensetzung haben?
- Technologie - funktioniert agile QS nur bei bestimmter Technologie gut, weil die Technologie bestimmte Eigenschaften hat?

Weil diese und andere möglicherweise relevante Faktoren im Vorhinein unbekannt sind, wären Umfragen oder Experimente schwierig durchzuführen. Die Fallstudie ist allerdings gut dafür geeignet, da bei Fallstudien Phänomen und Kontext zusammen untersucht werden und nicht vorher getrennt werden müssen.

Die wichtigsten Merkmale einer Fallstudie, im Vergleich zu anderen Forschungsmethoden sind (nach [12]):

- In einer Fallstudie ist **keine Kontrolle** auf den Untersuchungsgegenstand ausübbar.
- In einer Fallstudie werden viele **verschiedenartige, breite Daten** gesammelt: Dokumente, Interviews, Artefakte, Beobachtungen, ...

- Fallstudien benutzen nicht statistische Inferenz (wie Umfragen oder Experimente), sondern **Triangulation** um objektive Erkenntnisse zu erlangen. Triangulation bedeutet, dass ein Untersuchungsgegenstand (z.B. eine QS-Praktik in einem Team) aus mehreren Perspektiven untersucht wird: Durch Beobachtung und Interviews (**Datentriangulation**), Interviews aus verschiedenen Perspektiven auf einen Gegenstand und durch verschiedene Forscher (**Forschertriangulation**). Ein Beispiel: Wenn in einem Softwareteam ein Forscher beobachtet, dass *„XY funktioniert* so", ein Entwickler dazu erzählt *„XY soll* so *funktionieren*", der Manager betont *„wir achten darauf, dass die XY* so *machen*" und aus diesen Daten zwei Forscher unabhängig voneinander zu dem Schluss gelangen, dass XY *„so"* funktioniert, dann funktioniert XY wahrscheinlich wirklich *„so"*.

- Fallstudien muss ein **solides Design** zugrunde liegen, das expliziter beschrieben werden muss als z.B. bei Experimenten. Dort ist die Logik hinter Schlussfolgerungen klarer, weil sie einem vorgegebenen Schema folgt. Bei Fallstudien ist nicht offensichtlich, warum aus der Beobachtung von einem speziellen Phänomen allgemeine Schlussfolgerungen gezogen werden können.

- Die **Sicherung der Validität** ist bei Fallstudien besonders wichtig, weil es keine vorgegeben Kriterien, wie p-Level, für die Interpretationen der Daten gibt (mehr dazu in Abschnitt Sicherung der Validität).

Es gibt explorative, deskriptive und erklärende Fallstudien. Explorative Fallstudie sind dafür geeignet, zu einem Phänomen Theorien und Hypothesen zu entwickeln. Deskriptive Fallstudien beschreiben eine bestimme Situation. Erklärende Fallstudien können dazu benutzt werden, um Theorien mit Ursache-Wirkungsprinzipien zu validieren.
Da ich vor Beginn der Studie noch keine konkreten Hypothesen über die Erfolgsfaktoren für agile QS formulieren kann, führe ich eine **explorative** Fallstudie durch, um Hypothesen und Theorien zu identifizieren.

Fallstudien werden häufiger in den Sozialwissenschaften und der Wirtschaftsinformatik, jedoch eher selten im Software Engineering, durchgeführt. Runeson und Höhst weisen daher auf einige Unterschiede zwischen Fallstudien im Software Engineering und anderen Wissenschaften hin ([13, S. 132]):

- In anderen Feldern wird Software meist nur benutzt, während sie hier entwickelt wird

- In anderen Feldern wird meist Routinearbeit untersucht, während Softwareentwicklung komplexe Ingenieursarbeit ist, die von hochspezialisierten Arbeitern durchgeführt wird

3.3 Studienentwurf

Nach Yin ([12, S.26]) ist der Studienentwurf der **Plan**, um von den Forschungsfragen zu den Schlüssen zu gelangen. Außerdem ist es die **Logik** mit welcher aus den gesammel-

ten Daten die Fragen beantwortet werden können. Bei Fallstudien sind die wichtigsten Punkte:

- **Fragen**, die die Studie leiten
- **Units of analysis** (Definition der Gegenstände, die untersucht erden)
- Die **Logik** mit der Daten und Hypothesen verknüpft werden
- Kriterien um die Resultate zu **interpretieren** (insbesondere welche Schritte zur Sicherung der Validität unternommen wurden)

Hierbei handelt es sich um einen flexiblen Entwurf, d.h. er kann und soll während der Untersuchung den Gegebenheiten angepasst werden. Bei den Anpassungen dürfen aber nicht die ursprünglichen Ziele außer Acht gelassen werden. Wenn es zum Beispiel notwendig wird, die Forschungsfrage anzupassen, dann sollte besser eine neue Fallstudie entworfen werden.

3.3.1 Einfach- oder Mehrfachfallstudie?

Beim Entwurf einer Fallstudie stellt sich als Erstes die Frage, wie viele Fälle untersucht werden sollen. Es ist zeitlich möglich mehr als ein Team zu untersuchen. Die Untersuchung von nur einem Team wäre eingeschränkt auf den Kontext dieses einen Teams. Daher plante ich eine Mehrfachfallstudie.

3.3.2 Unit of Analysis

Der **Unit of Analysis** ist der Gegenstand, der untersucht werden soll: Das kann eine Person, eine Gruppe von Personen, eine Entscheidung, eine Kultur, eine Organisationsform und noch viele weitere sein.
Es ist wichtig, den Unit of Analysis klar zu definieren, weil er maßgeblich bestimmt welche Daten gesammelt werden. Meistens ergibt sich der Unit of Analysis bereits aus der Forschungsfrage.
Ursprünglich sollte der Unit of Analysis eine Softwareorganisation sein, weil die QS-Strategie das gesamte Unternehmen betrifft und von ihm vorgegeben wird. Nach der ersten Untersuchung änderte ich ihn zum agilen **Team**, weil es im ersten Fall keine firmeneinheitliche QS-Strategie gab, sondern jedes Team in der Firma seine eigene Strategie entwickelte.

3.3.3 Logik & Interpretation

Die Logik beschreibt im Voraus, wie die gesammelten Daten benutzt werden, um damit die Forschungsfragen zu beantworten. Es ist sinnvoll, im Vorhinein dazu Überlegungen anzustellen, um anschließend genügend Daten zu sammeln.

Ich werde die gesammelten Daten zu jedem Fall (also Team) in eine lineare Form bringen, indem ich einen Bericht schreibe, der die Qualitätssicherung dieses Teams beschreibt und analysiert. Die Struktur des Berichts wird sich aus den Daten ergeben. Anschließend werde ich anhand dieser Berichte Gemeinsamkeiten und Unterschiede zwischen den Teams herausarbeiten. Ich muss also genügend Daten sammeln, damit diese Erklärung so vollständig ist, dass nach der Lektüre keine relevanten offenen Fragen bleiben. Außerdem muss ich für die Beschreibung komplizierter und schwieriger Sachverhalte genügend verschiedenartige, bestätigende Datenquellen (z.B. Beobachtung, Interviews mit verschiedenen Leuten) sammeln, um einen Schluss einem skeptischen Leser gegenüber überzeugend zu begründen.

3.3.4 Prinzipien der Datensammlung

Laut Yin ([12, S. 114]) gibt es bei Fallstudien drei Prinzipien des Datensammelns:

- **Triangulation**: Beobachtung eines Gegenstandes aus verschiedenen Blickwinkeln. Sie werde ich hauptsächlich durch Datentriangulation erreichen (vgl. Abschnitt 3.2): Beobachtungen und Interviews, Interviews mit Personen, die verschiedenen Perspektiven haben (z.B. Entwickler und Product Owner).
- **Fallstudiendatenbank**: Aufzeichnung sämtlicher Daten in einer für Andere zugänglichen Form. Das werde ich durch Sammlung aller Notizen, Dokumente, Transkriptionen und Interviewaufzeichnungen elektronisch im SVN-Verzeichnis der Masterarbeit erreichen[1].
- **Klare Beweiskette**: Klare Nachvollziehbarkeit von fertigen Schlüssen zurück zu den Daten. Das werde ich durch Angabe der Datenquelle bei allen Aussagen in den Fallberichten erreichen. In dieser Ausarbeitung lasse ich diese Angaben weg, damit sie lesbarer ist. Sie können aber in den Fallberichten nachgeschlagen werden.

3.3.5 Sicherung der Validität

Die Sicherung der Validität ist bei Fallstudien besonders wichtig, weil es bei ihnen keine festen Kriterien zur Bewertung der Validität gibt (vgl. p-Level bei statistischen Auswertungen). Die vier Kriterien Konstruktvalidität, interne Validität, externe Validität und Verlässlichkeit werde ich durch verschiedene Maßnahmen versuchen einzuhalten.

Konstruktvalidität bedeutet, dass tatsächlich das gemessen wird, was gemessen werde soll und nicht etwas anderes. Konstruktvalidität ist bei Fallstudien schwierig herzustellen, weil leicht nur subjektive Sichtweisen wiedergegeben werden, statt objektive Daten zu sammeln. Um sie zu erreichen, wende ich folgende Methoden an:

[1] Verfügbar aus dem Netzwerk der FU Berlin: `https://svn.imp.fu-berlin.de/agse/students/schmeisky/`

- Triangulation: Hauptsächlich wird Datentriangulation angewandt.
- Validierung durch untersuchte Personen: Diskussion der Ergebnisse mit Managern und Entwicklern der Firmen.
- Klare Beweiskette von Schlussfolgerungen zurück zu den Daten: Quellverweise in den Berichten.

Interne Validität bedeutet, dass die gezogenen Schlüsse wahrscheinlich zutreffen (z.B. Schlüsse aus Aussagen über eine Vorgehensweise zu der Vorgehensweise). Dies erreiche ich nur durch Triangulation und Validierung. Dadurch ist die interne Validität nicht sehr hoch. Da ich aber eine explorative Fallstudie durchführe, und nicht den Anspruch habe, Ursache-Wirkungsbeziehungen zu belegen, ist dieser Grad an interner Validität akzeptabel.

Externe Validität bedeutet, dass Ergebnisse auf andere Kontexte verallgemeinerbar sind. Da ich mehrere Fälle untersuche, sind die Ergebnisse nicht nur auf einen Firmenkontext beschränkt und beschreiben nicht nur diesen Spezialfall. Sie sind aber auch nicht durch theoretische Replikation validiert und damit nicht ohne Weiteres auf andere Fälle verallgemeinerbar.

Verlässlichkeit bedeutet, dass ein anderer Forscher die gleichen Ergebnisse erarbeiten würde, wenn er seine Forschung mit den gleichen Zielen starten würde.Verlässlichkeit erreiche ich durch:

- die Verwendung eines Fallstudienprotokolls, das die einzelnen Untersuchungsschritte dokumentiert
- eine Fallstudiendatenbank, die erlaubt die Rohdaten unabhängig neu zu analysieren

3.4 Datensammlung

In diesem Abschnitt beschreibe ich die Methoden, die ich zur Datensammlung einsetzte. Dazu gehört der grobe Aufbau der Datensammlung, das Vorgehen bei der Auswahl der Fälle, die Verwendung des Fallstudienprotokolls und die Beschreibung der verwendeten Methoden zur Datensammlung. Die Analyse ist ebenfalls Teil dieses Abschnitts, weil Analyse und Datensammlung bei Fallstudien ineinandergreifende Prozesse sind.

3.4.1 Grober Aufbau

Grob unterteilt sich die Datensammlung für jeden Fall in eine Vorstudie und eine Hauptstudie. Vor der Untersuchung kannte ich beide Firmen nicht, daher musste ich erst den Kontext kennenlernen, um anschließend die Qualitätssicherung zu untersuchen und zu

verstehen. Um über den Kontext zu lernen, begleitete ich zuerst einen Entwickler mehrere Tage in der **Vorstudie**. In der anschließenden **Hauptstudie** führte ich Interviews mit Personen, die verschiedene Blickwinkel auf die Qualitätssicherung haben: Entwickler, Produktverantwortliche, Teamleiter, Scrum Master. Idealerweise hätte ich in jedem Team genügend Leute interviewt bis ich **Sättigung** ([13, S. 147]) erreicht hätte, d.h. dass durch ein weiteres Interview keine weiteren Erkenntnisse hinzukommen würden. Dieser Anspruch wird jedoch durch den Umfang einer Masterarbeit begrenzt.

Bei einer Fallstudie sind Datensammlung und Analyse ineinander greifende Prozesse. Stellte sich in der Analyse heraus, dass noch Fragen offen sind, sammelte ich diese und schrieb den Entwicklern E-Mails oder führte weitere Interviews, bis die offenen Fragen beantwortet waren.

3.4.2 Ethische Standards

Da Menschen und Firmen Gegenstand meiner Forschung sind, habe ich meine Untersuchung den üblichen ethischen Standards in diesem Bereich unterworfen (nach [13, p. 142]):

- **Einverständnis**: Allen Beteiligten muss klar sein, dass sie auf keine meiner Fragen antworten müssen und auf Wunsch nicht Teil der Datensammlung sind.
- **Vertraulichkeit**: Allen Beteiligten muss klar sein, dass die gesammelten Daten vertraulich behandelt werden und nicht ohne Zustimmung veröffentlicht werden (z.B. in Form von Zitaten).
- **Schriftliche Ethik-Vereinbarung**: Eine schriftliche Zusicherung der Einhaltung der Ethik-Richtlinien wurde den Firmen optional angeboten. [2]

3.4.3 Auswahl der Fälle

Die Idee für diese Masterarbeit lieferte ein Vortrag von Alexander Große, Entwicklungsleiter bei SoundCloud, auf den Berlin Expert Days [3]. Er präsentierte dort den Entwicklungsstil der Firma SoundCloud, der sich so stark von den mir bekannten Stilen unterschied, dass ich ihn gerne untersuchen wollte.
Um herauszufinden, ob diese Arbeitsweise nur bei einem Team in einer Firma funktioniert, wollte ich mehrere Teams aus verschiedenen Firmen untersuchen. Da SoundCloud eine Webplattform ist, war es sinnvoll eine weitere Webplattform zu suchen, weil es durch die gleichen technologischen Gegebenheiten genügend Berührungspunkte geben würde. Auf der plat_form Minikonferenz[4] sprach ich Bastian Buch, Teamleiter bei

[2] Die Vereinbarung mit Immobilienscout24 befindet sich unter `daten/immoscout/vereinbarung.tex`. SoundCloud wollte keine Ethikvereinbarung.

[3] `http://bed-con.org`

[4] `https://www.plat-forms.org/`

Immobilienscout24, an, der auf Anhieb Interesse an dem Thema zeigte. Sein Team zeigte auf den ersten Blick Ähnlichkeiten zu SoundCloud.
Für den dritten Fall bat ich Bastian Buch, den Kontakt zu einem Team aus der Kernapplikation von Immobilienscout24 herzustellen. Er stellte den Kontakt zum Teamleiter eines Teams der Legacyapplikation her.

3.4.4 Fallstudienprotokoll

Yin ([12, S. 80f]) beschreibt das Fallstudienprotokoll als Werkzeug um den Forscher beim Untersuchen eines Falls zu leiten.

Dementsprechend beschreibt das Fallstudienprotokoll zuerst die allgemeine Richtung der Fallstudie, um den Personen, die untersucht werden, zu erklären, worum es in der Studie geht. Es beschreibt die notwendigen Prozeduren vor Ort: eine Aufklärung darüber, dass Daten gesammelt werden und den Hinweis, dass die Teilnahme freiwillig ist. Es beinhaltet weiterhin die **Fallstudienfragen**, die an den Untersucher gerichtet sind und die Datensammlung leiten. Das Ausformulieren dieser Fragen soll helfen, den Blick auf das Wesentliche zu behalten und sich nicht in unwichtigen Details zu verlieren. Nach dem ersten Fall können diese Fragen für den nächsten Fall angepasst werden.
Das Fallstudienprotokoll beinhaltet außerdem eine **Gliederung des Fallberichts**, um in Erinnerung zu rufen, welche Daten noch gesammelt werden müssen und welche schon gesammelt wurden.
Während der Untersuchung sah sich von den Untersuchten kaum jemand das Fallstudienprotokoll an. Die Fragen waren aber sehr hilfreich um zu wissen, wozu ich noch Fragen stellen musste und was ich schon wusste.

Die Fallstudienfragen sollten ausreichend sein, um, wenn sie beantwortet sind, die Qualitätssicherung eines Teams zu beschreiben. Beim ersten Fall waren die Fallstudienfragen, noch sehr breit, eher die ganze Firma betreffend, und verengten sich bei den späteren Fällen zunehmend. Es folgen die Fragen für den ersten Fall[5]:

- *Wie sieht die Software aus?*
  (Architektur, Sprachen, Plattformen, wichtige Tools)
- *Informationen über die Firma*
  (Größe, Alter, Verhältnis Entwickler zu restlichem Personal, Atmosphäre, Lage, Anzahl Gebäude und Verteilung)
- *Wie sind die Entwickler organisiert?*
  (Teams, Projekte, Rollen, ...)
- *Wie werden agile Methoden eingesetzt?*

[5] aus dem Fallstudienprotokoll `daten/soundcloud/fallstudienprotokoll.tex`

- *Wie werden Aufgaben an Entwickler verteilt?*
- *Wie planen Entwickler ihre Zeit?*
- *Wie veröffentlichen Entwickler ihre Änderungen? Wann bekommt ein Nutzer sie zu sehen?*
- *Wie werden gefundene Defekte behandelt?*
- *Welche Stationen sind für eine einzeilige Änderung nötig?*
  Gibt es auch einen Notfallweg, für z.B. sofortige Veröffentlichung?
- *Gibt es Unterschiede zwischen dem Ist-Zustand und dem Soll-Zustand?*
- *Sonstiges (Keep an open mind)*

Zwischen SoundCloud und dem ersten Team bei Immobilienscout24 haben sich die Fragen verändert. Der Unit of Analysis war nun nicht mehr die Firma sondern das Team, daher sind die Fragen teamspezifischer und es gibt weniger Fragen über die Firma. Sie orientierten sich an der Struktur des Berichts zu SoundCloud, weil es hilfreich war, das zweite Team nach der gleichen Struktur zu erfassen. Außerdem sollte vermieden werden, so viele offenen Fragen wie bei SoundCloud zu haben. Neu hinzugekommene Fragen sind u.A. [6]:

- *Was ist für das Team das wichtigste Qualitätskriterium?*
- *Wie werden dabei [beim Entwickeln] automatisierte Tests eingesetzt?*
- *Wodurch werden Defekte entdeckt?*
- *Zeigen die Entwickler Verantwortungsgefühl?*

Beim dritten Team kristallisierte sich heraus, dass Feedback und Verantwortlichkeiten eine wichtige Rolle spielen. Entsprechend gab es neue Fragen dazu[7]:

- *Welche Aufgaben haben die Entwickler außer Programmieren?*
- *Haben die Entwickler das Gefühl, Aufgaben abschließen zu können?*
- *Überwachung der Software durch Entwickler?*

3.4.5 Entwicklerbegleitung

Das Ziel für die Entwicklerbegleitung in der Vorstudie war, genügend über den Prozess und das Produkt des Teams zu lernen, um den Kontext des Teams zu verstehen. Außer-

[6] aus dem Fallstudienprotokoll `daten/immobilienscout_online_marketing/fallstudienprotokoll.tex`

[7] aus dem Fallstudienprotokoll `daten/immobilienscout_anbieten_profi/fallstudienprotokoll.tex`

dem sollten Phänomene beobachtet werden, um diese Beobachtung als Triangulation einzusetzen. Dazu begleitete ich einen Entwickler jeweils 3 Tage á 3-4 Stunden.

Die Einführung beim Entwickler setzte die Akzente für die Begleitung und für die Phänomene, die mir dabei erklärt wurden, daher ist sie relevant für die Methodik. Die Einführung beim Entwickler sah folgendermaßen aus:

- *Ich möchte die Qualitätssicherung des Teams untersuchen.*
- *Ich möchte wissen: Wie arbeitet dieses Team tatsächlich? Nicht nur wie es sein soll, sondern wie es tatsächlich arbeitet.*
- *Dabei können auch Sachen wichtig sein, die nicht zur klassischen Qualitätssicherung zählen, sondern eher zum Entwicklungsprozess.*

Die Notizen bei der Begleitung unterteilen sich hauptsächlich in folgende zwei Arten. Welcher Art eine Notiz ist, ist im zugehörigen Protokoll vermerkt.

- **Erklärung**: eine Person erklärt mir etwas mündlich und ich notiere diese Erklärung so wie ich sie verstehe.
- **Beobachtung**: Ich beobachte, wie etwas passiert und notiere es.

3.4.6 Dokumente

Bei der Entwicklerbegleitung bietet es sich an, Dokumente über das Team und seine Arbeit zu sammeln. Allerdings bedeutet ein Dokument an sich nichts, ohne zu wissen, wie es verwendet wird. Daher spielen Dokumenten im Vergleich zu den Interviews eine untergeordnete Rolle in der Durchführung meiner Fallstudie.

3.4.7 Interviews

Die Interviews sind die Hauptdatenquelle der vorliegenden Fallstudie. Ich führte sie in jedem Team mit Personen, die verschiedene Rollen im Team repräsentieren.

Nach Runeson et al. ([13, S. 146]) ist das Interview ein Dialog zwischen Forscher und Subjekte(n). Der Vorteil von Interviews liegt darin, dass in ihnen sehr kompakt und intensiv ein bestimmtes Thema erfragt werden kann. Das Interview wird durch Fragen vom Forscher geleitet, welche aus der Forschungsfrage formuliert werden.
Die Fragen können offen (viele, breite Antwortmöglichkeiten) oder geschlossen sein (wenige Antwortmöglichkeiten). Es gibt unstrukturierte Interviews (Forscher sagt nur allgemeine Richtung und sein Interesse an), voll-strukturierte Interviews (alle Fragen sind vorformuliert, wie ein Fragebogen) und semi-strukturierte Interviews (es werden Fragen geplant, das Interview wird aber nicht auf die Fragen beschränkt und es müssen nicht alle gestellt werden).
Ich führte semi-strukturierte Interviews mit mehr offenen als geschlossenen Fragen,

weil ich erwartete, dass sich im Verlauf des Interviews interessante Fragen ergeben würden und ich diesen nachgehen wollte. Zu den Interviews arbeitete ich einen Interviewleitfaden, basierend auf den Fallstudienfragen aus. Er ist Teil des Fallstudienprotokolls.

Ein Interview besteht aus mehreren Phasen, die ich mit folgenden Themen füllte:

- Vorher: Smalltalk, reden über belanglose Dinge um Vertrautheit herzustellen
- **Einführungsphase**: Ziele von Interview und Studie vorstellen, erklären wie ich die Daten verwende, kurz erzählen was ich bereits weiß, damit das Subjekt es nicht wiederholen muss, Vertraulichkeit versichern
- **Leichte Einführungsfragen**: Hintergrund und Erfahrung des Subjekts, Rolle in der Firma
- **Hauptfragen**: Fragen aus dem Interviewleitfaden fragen
- **Zusammenfassung**: Hauptpunkte zusammenfassen, um Feedback zu bekommen, Missverständnissen vorzubeugen und um dem Subjekt die Zeit zu geben zu überlegen, ob etwas vergessen wurde

Hier ein Auszug aus einem Interviewleitfaden für einen Entwickler aus dem dritten Fall[8]:

- *Standardeinführung*
- *Mehr Frontend- oder Backendaufgaben?*
- *Was für unterschiedliche Arten von Aufgaben hast du?*
- *Wie sehr bist du involviert in Planung / Abnahme / Qualitätssicherung?*
- *Wie erledigst du Arbeit? Irenes Rolle und ob sie Bottleneck ist?*

...

- *Was ist für dich die wichtigste Qualitätsfacette eurer Anwendung?*
- *Wie viel arbeitet ihr mit Geschäftsmetriken als Maß für eure Qualität?*
- *Welche Elemente, Praktiken, Schritte, Technologien dienen zur Qualitätssicherung in seinem Team?*
- *(nach fragen: Tests, Manuelle Tests, Sonar, Produktvision, Verantwortung, Clean Code, Methoden klein?)*
- *Welche Rolle spielen dabei automatisierte Tests?*
- *Was ist für dich ein guter Test?*

[8]aus dem Fallstudienprotokoll `daten/immobilienscout_anbieten_profi/fallstudienprotokoll.tex`

- *Was ist am Wichtigsten für eure QS-Strategie (und am 2. Wichtigsten ...)*

...

- *Wie zufrieden seid ihr mit der Qualität, die ihr produziert?*
- *Was für Probleme gab es schon?*
- *Wie viel Downtime hast du, wo du auf andere warten musst?*

...

- *Sonst noch Fragen? (Keep an open mind)*
- *Zusammenfassung, korrigieren lassen*

3.5 Analyse

Um die Daten zu analysieren, verwendete ich Praktiken aus der Grounded Theory (aber keinen Grounded Theory Ansatz). Zuerst transkribierte ich die interessanten Teile der Interviews und schrieb die Notizen aus der Begleitung ins Reine. Anschließend kodierte ich diese Daten, indem ich interessante Stellen und Zitate im Text mit Codes (z.B. „*#Verantwortung*“) versah. Um Ordnung in diese Fülle von Daten zu bringen, brachte ich sie in eine narrative Form, sodass ein Bericht[9], der den Entwicklungsprozess eines Teams und seine Qualitätssicherung beschreibt, entstand. In ihm verwies ich mit Zeilen- oder Zeitangaben auf entsprechende Stellen in der Quelldatei. Aus diesen Berichten arbeitete ich Konzepte heraus und leitete die Hypothesen ab. Von diesen Erkenntnissen ließ ich mich in der weiteren Untersuchung leiten, indem ich sie für neue Fragen und zum theoretischen Sampling verwendete.

3.5.1 Offene Fragen klären

Nach der ersten Datensammlung in den Teams und der Analyse durch das Schreiben des Berichts ergaben sich für jedes untersuchte Team, neue, offene Fragen. Um sie zu beantworten, führte ich weitere Interviews mit Teammitgliedern. Beispielhaft für solche Fragen sind [10]:

- *Inwiefern benutzt das Team statische Codeanalyse?*
- *Inwiefern benutzt das Team Technical Debt als Metapher?*

---

[9] Die Berichte zu den Fällen finden sich unter
`daten/soundcloud/bericht_soundcloud.tex`
`daten/immobilienscout_online_marketing/bericht_immoscout.tex`
`daten/immobilienscout_anbieten_profi/bericht_is24_anbieten.tex`

[10] aus dem Fallstudienprotokoll
`daten/immobilienscout_online_marketing/fallstudienprotokoll.tex`

- *Was ist für das Team ein guter Test?*
- *Wer bekommt auf den Deckel, wenn die Qualität zu schlecht ist?*
- *Wie ist die Fehlerkultur?*

3.5.2 Validität der Daten

In den ersten beiden Fällen funktionierte die Datentriangulation gut. Ich beobachtete mehrere Phänomene, die mir genau so in Interviews geschildert wurden und die Aussagen verschiedener Interviewpartner mit verschiedenen Rollen bestätigten sich gegenseitig. Beispielsweise beschrieben POs und Entwickler ihre Arbeitsweise beide als kooperativ und als einen Dialog. In einem anderen Fall wurde wurde mir erzählt, dass fast jeder Code einem Review unterzogen wird und ich konnte es genau so beobachten. Beim dritten Fall war es schwieriger, sicherzugehen ob Beobachtungen korrekt sind, weil das Team größer war und die Mitglieder einen Arbeitsalltag mit viel mehr Unterbrechungen hatten. Allerdings bestätigten sich auch hier die Aussagen unterschiedlicher Teammitglieder gegenseitig. Nach der Analyse diskutierte ich meine Ergebnisse mit dem Entwickler und konnte so Widersprüche aus dem Weg räumen.

3.5.3 Hypothesen entwickeln

Um die Hypothesen zu identifizieren, schrieb ich zu mehreren Zeitpunkten der Untersuchung meine aktuellen Hypothesen auf und sammelte neue Daten, um sie zu überprüfen. Dazu wählte ich insbesondere nach dem zweiten Fall einen dritten Fall aus, der neue Einsichten versprach.

Nach dem ersten Fall nahm ich die Begründung für eine effektive Qualitätssicherung, die mir der Entwickler genannt hatte, und versuchte, sie im zweiten Fall zu überprüfen. Im zweiten Fall zeigte sich allerdings ein anderes Bild und dies war nicht möglich. Zum Beispiel waren im ersten Fall die Entwickler auf Bereitschaft, im zweiten aber nicht. Stattdessen versuchte ich, die Phänomene mit einem Modell für Motivation zu erklären und wählte einen dritten Fall, um diese Theorie zu überprüfen. Dort stellte sich heraus, dass Motivation nur ein Faktor unter mehreren ist. Daher entwickelte ich aus den Daten aller drei Fälle ein Modell, dass die gemeinsamen Zusammenhänge und relevanten Faktoren erklärt. Dieses stellt ich Managern beider Firmen vor und diskutierte es, um Widersprüche und fehlende Elemente zu identifizieren. Dadurch kamen Elemente hinzu, jedoch ohne den Kern des Models zu verändern. Diese finale Hypothese wird im Abschnitt Das Qualitätserlebnis dargelegt.

4 Die Fälle

In diesem Abschnitt stelle ich die drei untersuchten Teams, Payment, Online Marketing und Anbieten Profi, und die zwei Firmen, SoundCloud und Immobilienscout24, in denen sie arbeiten, vor.

4.1 SoundCloud

Das Produkt von SoundCloud (SC) ist die Seite `soundcloud.com`, wo Künstler ihre selbsterstellte Musik hochladen, mit anderen teilen, und Feedback einholen können. Auf SoundCloud können 2h Musik kostenlos hochgeladen werden und gegen Bezahlung mehr. `soundcloud.com` hat ca. 10 Millionen Nutzer und ist unter den 200 meistbesuchten Websites der Welt.

Die Webplattform begann als kleine Rails-Applikation (das **Mothership**), die über die Zeit in Funktionalität und Umfang wuchs. Da die Größe der Applikation zunehmend zu Problemen führte, wurde Anfang 2012 beschlossen, die große Rails-App in verteilte Dienste aufzuteilen. Seitdem wird schrittweise Funktionalität aus dem Mothership ausgebaut und in verteilte Dienste verlagert. Neue Funktionalität wird meist in den neuen Diensten entwickelt. Die Dienste sind mit vielen verschiedene Technologien umgesetzt, z.B. werden Go, Haskell und Ruby verwendet.

Für SoundCloud arbeiten ca. 80 Entwickler. Die gesamte Softwareentwicklungsbelegschaft besteht nur aus Entwicklern und es gibt keine Sonderrollen wie Tester, Architekt oder Softwareentwickler. Die Entwickler sind in Teams organisiert, die grob gesagt einen Verantwortungsbereich (z.B. Suche oder Bezahlung) betreuen. Ein Team hat eine Tischgruppe an der alle Mitglieder sitzen. Generell bevorzugt SoundCloud vertikale Teams, die komplett für einen bestimmten Teil der Seite verantwortlich sind. Entwickler dürfen ca. 20% ihrer Arbeitszeit für Aufgaben verwenden, die nicht ihrer Hauptaufgabe entsprechen, aber sinnvoll für die Firma ist, z.B. Arbeit an einem Tool oder einem anderen Projekt .

4.2 SoundCloud – Team Payment

Team Payment (PM) von SoundCloud ist verantwortlich für alle Aufgaben, die die Bezahlung durch Benutzer betreffen:

- User Lifecycle (von Probeabo, über Abonnement bestellen, zu Erneuerung, bis Kündigung) inkl. Emailbenachrichtigungen
- Betrugserkennung
- Reporting an Finanzabteilung

Die Software von Team Payment ist hauptsächlich der Dienst **Buckster**, der eine API bietet für alle die Bezahlung betreffenden Funktionen, z.B. das Bezahlen oder Einlösen von Gutscheinen. Diese Schnittstelle wird vom Frontend aufgerufen, Buckster kommuniziert mit Bezahldienstleistern und wandelt erfolgreiche Bezahlungen in Abonnements um. Zum Zeitpunkt der Untersuchung (August 2013) befanden sich diese Funktionen teilweise noch im Mothership und wurden Schritt für Schritt nach Buckster migriert. Zusätzlich betreut das Team einen Emaildienst, der hauptsächlich von ihnen selbst, aber zum Teil auch von anderen Teams benutzt wird.

Team Payment besteht aus zwei Entwicklern (seit 2010 Jahren bei SC und seit Mai 2013 bei SC) und einem Product Owner (seit Anfang 2012 Jahren bei SC). Zwei Mitglieder verließen kurz vor meiner Beobachtung das Team. Ich habe den Dienstälteren der beiden Entwickler begleitet und interviewt.

4.3 Immobilienscout24

„ImmobilienScout24 ist der größte deutsche Internet-Marktplatz für Immobilien. Mit über 10 Millionen Besuchern (Unique Visitors; laut comScore Media Metrix) pro Monat ist die Website auch das mit Abstand meistbesuchte Immobilienportal im deutschsprachigen Internet.
Das Unternehmen sitzt in Berlin und beschäftigt über 600 Mitarbeiter. Seit über 10 Jahren ist ImmobilienScout24 erfolgreich im Internet tätig.“
(von `www.immobilienscout24.de`) Immobilienscout24 betreibt rund um das Kernprodukt, Immobilien anbieten und suchen, viele weitere Dienstleistungen und andere Suchplattformen. Die Plattform wird von vielen gewerblichen Anbietern genutzt.

IS24 setzt auf hauptsächlich auf JVM-basierte Technologien. Die meiste Software ist in Java geschrieben, stellenweise wird auch Scala und Groovy verwendet. Das Immobilienportal war eine sehr große, monolithische Java-Anwendung, die **Kernapplikation**. Aufgrund diverser Probleme mit dieser Struktur, wird sie in verteilte Dienste modularisiert . Die Modularisierung ist bereits fortgeschritten, aber nicht abgeschlossen. An der Kernapplikation arbeiten noch mehrere Teams. IS24 hat seine Infrastruktur ingesourct, betreibt also selber die Server und entwickelt Tools, z.B. zum automatisierten Erstellen von VMs und Deployment, um sie als private Cloud zu betreiben, um flexibler zu sein.

Bei Immobilienscout24 arbeiten ca. 100 Entwickler. Immobilienscout24 war mal „klassisch“ organisiert mit separaten POs, Entwicklern, Architekten, Testern und Operatoren . Jetzt integrieren sich die Teams mehr und sollen funktionsübergreifende Kompetenzen, wie PO, Designer, Entwickler und Tester in einem Team vereinen. Das ist allerdings unterschiedlich weit umgesetzt. Die Firma favorisiert agile Prozesse und die meisten Teams arbeiten nach einem auf Scrum basierenden Prozess. Es gibt aber keine konkreten Vorgaben, wie ein Prozess aussehen muss.

4.4 Immobilienscout24 – Online Marketing

Team **Online Marketing** (OM) ist das Entwicklungsteam der Service Line Online Marketing. Ich untersuchte es im September 2013. Es besteht aus 2 Entwicklerteams mit jeweils 4 Entwicklern, einem *Technical Lead (TL)* / *Scrum Master* und einem *Product Owner (PO)* für beide. Die beiden Teams arbeiten unabhängig voneinander im selben Bereich (Online Marketing), daher werden hier zuerst die Aufgaben für beide vorgestellt, die spätere Untersuchung betrifft aber nur eines der Teams.
Das Entwicklungsteam entwickelt verschiedene Lösungen für die Anforderungen des Onlinemarketings der Firma, umgesetzt mit jeweils einem Dienst:

- Suchmaschinenoptimierung mithilfe von generierten **Ergebnislisten**. Auf den Hauptseiten `immobilienscout24.de` und `.at` ist eine Eingabe in ein Suchfeld nötig, um zu den Exposés der Immobilien zu gelagen. Daher werden extra Seiten mit vorgefertigten Suchen für Suchmaschinencrawler erzeugt, die optimiert sind für hohe Relevanz
- Marktplatzintegration mit Partnerplattformen wie `meinestadt.de` oder `morgenpost.de`
- Landingpages für Suchmaschinenkampagnen (z.B. AdWords)
- Klickmessungen der eingehenden Verweise und Reporting an die Finanzabteilung
- Marketingmanagementtools zur Analyse der Marketingmaßnahmen
- interaktive HTML5-Werbemittel
- APIs für den Export von Partnerobjektdaten
- angeschlossene Vermarktungsplattformen

Der größte Dienst ist der **Integrierte Katalog** (InKa) für die Ergebnislisten auf `is24.de`, die nächstgrößeren der InKa für `is24.at` und der Server für die Klickmessungen. Die anderen Produkte sind kleiner. Die meisten Produkte sind sehr technisch und nicht oder eher weniger für Endbenutzer konzipiert. Daher sind die Geschäftsanforderungen selbst oft technisch: Z.B. Headerformate, Responses, Positionierung im Quelltext, Orte von denen Elemente geladen werden.

Der InKa (für .at und .de) ist in Java geschrieben und eine eigenständige *Spring MVC 3* Anwendung. Die anderen Dienste sind ebenfalls eigenständig und sind größtenteils in Java geschrieben. Einige kleinere Dienste sind in PHP, weil früher PHP benutzt wurde.

Früher war das Team Online Marketing ein großes Team aus 7 Entwicklern, einem Testingenieur, einem Product Owner und einem Teamleiter. Im April 2013 wurde das Team in 2 Unterteams aufgeteilt: Team A mit Entwicklern, die mit ihren bisherigen Aufgaben ausgefüllt sind und Team B mit Entwicklern, die gerne mehr Aufgaben übernehmen wollen. Ich hab nur das Team B untersucht.

Das Team B besteht aus 4 Entwicklern. Der PO arbeitete 10 Jahre als Softwareentwickler gearbeitet, und hat daher relativ viel technisches Verständnis. Das ganze Team (Entwickler, PO und Teamleiter) sitzt an einer Tischgruppe in einem etwas abgeteiltem Bereich. Im Teamraum ist ein großes Whiteboard mit Klebezetteln, ein großer Fernseher mit Produktstatistiken, mehrere kleine Whiteboards und Flipcharts.

4.5 Immobilienscout24 – Anbieten Profi

Das Team **Anbieten Profi** (AP) ist zuständig für Insertionen auf `immobilienscout24.de`. Anbieter, die auf `immobilienscout24.de` Immobilien inserieren wollen, benutzen den **ScoutManager**, einen geschlossenen Bereich in dem Anzeigen erstellt und verwaltet werden. Den Scoutmanager betreuen mehrere Teams und jedes ist in etwa für einen Bereich zuständig, aber es gibt keine klare Abgrenzung. Team Anbieten Profi betreut im ScoutManager hauptsächlich den Insertionsbereich, mit dem Anbieter Inserate erstellen und gestalten, die Verwaltung bestehender Inserate und die Verlinkung anderer Angebote der Firma, wie Bausparfinanzierung. Dabei sind ihre Aufgaben mehr im **Frontend** als im Backend angesiedelt. Die Seiten werden von einem Designer designt, die Entwickler passen die Templates an und setzen die Programmlogik um.

Das Team besteht aus 10 Mitarbeitern: 6 Entwicklern, einer Testingenieurin, einem (Teilzeit-)-Scrum Master, einer PO, und einem Teamleiter. Es arbeitet erst seit einem Jahr in dieser Zusammensetzung. Die PO hat keinen technischen Hintergrund.

Der ScoutManager ist Teil der **Kernapplikation** von Immobilienscout24, einer großen monolithischen Spring 3 Applikation mit mehr als 10.000 Quellcodedateien und über 18.000 Tests. Sie benötigt diverse andere Dienste um zu funktionieren: Datenbankserver, Nutzerdatenserver, Videoserver, ... Die Kernapplikation hat einen **1-wöchigen Releasezyklus**: Dienstags um 10 Uhr wird der Testabteilung ein Snapshot zur Verfügung gestellt, der im Normalfall 8 Tage später veröffentlicht wird.

Diesen Fall wählte ich mithilfe von Theoretical Sampling aus. Nach der Datensammlung bei Team OM war genügend Zeit für einen weiteren Fall und es gab mehrere Fälle bei Immobilienscout24 zur Auswahl:

- Team Österreich: Greenfield Projekt mit ähnlicher Arbeitsweise wie Team OM und Team PM
- Team Anbieten Profi: Teil der Legacyapplikation mit nachgelagerter QS durch die Testabteilung

Ich wollte den Fall wählen, der mehr Erkenntnisse zu liefern versprach. Die zwei bisher untersuchten Teams (Payment und Online Marketing), hatten keine nachgelagerte QS. Würde ich noch ein Team ohne nachgelagerte QS untersuchen, würde ich also nur wenige neuer Erkenntnisse erlangen.
Um neue Erkenntnisse zu erlangen müsste ich ein Team untersuchen, das anders ar-

beitet, z.B. nachgelagerte QS hat. Also wählte ich das Team Anbieten Profi, um einen größeren Erkenntnisgewinn zu erlangen.

5 Das Qualitätserlebnis

Für die untersuchten Teams ist es sinnvoll, wenn Veränderungen schnell veröffentlicht werden. Dadurch lassen sich neue Ideen ausprobieren und Feedback von Kunden einholen, um diese schnell zu verbessern. Entwickler können auf Probleme von Benutzern besser reagieren, da nur eine geringe Verzögerung zwischen der Entwicklung und dem Auftreten der Probleme entsteht.
Traditionelle Qualitätssicherung ist dabei problematisch, weil die nachgelagerte Testphase die Zeit bis zur Veröffentlichung verlängert. Zwei der drei untersuchten Teams haben die Sicherheit, die traditionelle QS bietet, ersetzt durch ein intensives **Qualitätserlebnis**. Die Teams arbeiten ohne nachgelagerte Testphase, aber die Feedbackschleife für alle Aspekte der Qualität ist sehr kurz und sehr stark. Dadurch erkennen sie Abweichungen in der externen oder internen Qualität schnell, reagieren darauf und erreichen so hohe Qualität.

Ein schwaches Qualitätserlebnis haben Entwickler, wenn sie die Auswirkungen ihrer Arbeit nur diffus, verzögert und mittelbar zu spüren bekommen:

- **Diffus**: Auftretende Effekte können die Entwickler nicht auf ihre eigene Arbeit zurückführen, da es mehrere sich vermischende Ursachen gibt.
- **Verzögert**: Es dauert so lange, bis Änderungen veröffentlicht werden, dass die Entwickler bereits an etwas Anderem arbeiten.
- **Mittelbar**: Sie sind nicht diejenigen, die Abweichungen entdecken, sondern bekommen sie nur von anderen vermittelt.

Ein starkes Qualitätserlebnis haben Entwickler, wenn sie die Auswirkungen ihrer Arbeit schnell, direkt und unmittelbar zu spüren bekommen:

- **Schnell**: Es werden oft neue Versionen veröffentlicht (alle 2-3 Tage) und der Veröffentlichungsprozess ermöglicht schnelle Veröffentlichungen.
- **Direkt**: Die Effekte an der Software (z.B. auftretendes Versagen) können Entwickler eindeutig auf die eigene Arbeit zurückführen.
- **Unmittelbar**: Die Effekte ihrer Arbeit nehmen Entwickler selber wahr, z.B. weil sie selbst die Überwachungssoftware benutzen.

Mit Qualität meine ich hierbei nicht die Abwesenheit von Fehlern, sondern dass das Produkt einen Wert hat: Für die Kunden, die es benutzen, für die Firma, die damit Geld verdient und für die Entwickler, die damit flexibel neue Features entwickeln können.

Ich habe drei Säulen identifiziert, die es einem Team ermöglichen, ein starkes Qualitätserlebnis zu haben:

- **Hohe Qualität erzeugen**: Die Entwickler müssen die Fähigkeiten haben, Software von hoher interner und externer Qualität zu erzeugen.

- **Qualität erkennen**: Die Entwickler müssen die Qualität der eigenen Arbeit erkennen können.
- **Schicksal bestimmen**: Die Entwickler müssen bestimmen können, in welche Richtung sich das Produkt entwickelt und die Fähigkeiten haben, dafür zu sorgen, dass es diese Richtung nimmt.

Diese Säulen bestimmen, ob Entwickler ein starkes oder schwaches Qualitätserlebnis haben. Wenn sie stark ausgeprägt sind (hohe Fähigkeiten gute Qualität zu erzeugen, Qualität eigener Arbeit gut erkennbar und Entwickler bestimmen Schicksal des Produkts zu hohem Grade) können die Entwickler ein starkes Qualitätserlebnis haben.

Interessant ist die Frage, wie ein starkes Qualitätserlebnis erreicht wird. Für jeden der drei Säulen habe ich aus den Daten der Fälle mehrere Einflussfaktoren herausgearbeitet. Diese Einflussfaktoren sind nicht nur technsicher, sondern auch organisatorischer Natur. Folgende Einflussfaktoren habe ich beobachtet, die dazu führen, dass ein Entwicklerteam hohe Qualität erzeugen kann:

- Die Fähigkeiten hohe **interne Qualität** zu erzeugen und zu erhalten. Das können Prozesskomponenten, wie Reviews und Zeit für technische Verbesserungen, und Produktaspekte, wie gute Architektur und konsistente Codestandards, sein. Bei niedriger interner Qualität wäre ein Team zu sehr damit beschäftigt mit den Auswirkungen der niedrigen internen Qualität zu kämpfen (z.B. Architekturschwächen).
- Eine gute **Produktvision**, die deutlich in den Köpfen der Entwickler lebt. Das bedeutet, sie wissen, wie sich das Produkt entwickeln soll und sie kennen die echten Anforderungen. Die Produktvision muss die Kundenbedürfnisse gut abbilden.

Ich habe folgende Faktoren beobachtet, die es einem Team ermöglichen, die Qualität seiner Arbeit zu erkennen:

- Die Möglichkeit schnelles, **automatisiertes Feedback**, zu bekommen, um die Qualität von Änderungen schnell zu erkennen. Das können zum Beispiel Tests und statische Codeanalysetools sein, die permanent von einem **Continuous Integration**-Server ausgeführt werden.
- Damit das Feedback aussagekräftig ist, sind **gute, automatisierte Tests** nötig. Das heißt, die Tests sind:
  - *schnell*: ihre Laufzeit ist kurz.
  - *verlässlich*: ihr Ergebnis hängt nicht von äußeren Einflüssen, wie externen Diensten, ab.
  - *aussagekräftig*: Sie müssen tatsächlich Defekte entdecken können, und nicht immer ein positives Ergebnis liefern, auch wenn etwas nicht funktioniert.
- Es vergeht nur **wenig Zeit** vom Anfang einer Aufgabe bis zur Veröffentlichung der Änderungen. Dadurch sind für die Entwickler die Auswirkungen (Versagen tritt auf, Nutzerverhalten ändert sich) der eigenen Handlungen (Code schreiben)

leichter erkennbar.
Dieser Punkt ist wahrscheinlich der wichtigste. Selbst kleine Verzögerungen von nur einer Woche, können das Qualitätserlebnis bereits deutlich schwächen.

- Möglichst einfach und ohne Zusatzaufwand können Entwickler Software **automatisiert veröffentlichen** (z.B. mit einem Knopfdruck oder einem Befehl). Durch das Wegfallen manueller Schritte kann meist der Veröffentlichungsprozess beschleunigt werden.
- Die Entwickler **überwachen** die Software selber, indem sie selber Daten für Metriken sammeln, die ihnen sagen, wie die Software im Betrieb funktioniert und sie diese Metriken gut sichtbar anzeigen.
- Es gibt **klare Grenzen in der Software**. Dadurch ist klar, wo der Verantwortungsbereich des Teams (in der Software) anfängt und wo er aufhört. Wenn mehrere Teams an einer geteilten Codebasis mit vielen Abhängigkeiten untereinander arbeiten, ist das schwierig.

Ich habe diese Einflussfaktoren beobachtet, die dazu führen, dass ein Team das Schicksal seines Produkts bestimmen kann:

- **Motivierte** Entwickler. Dazu gehört auch, dass sie möglichst wenig frustriert sind. Nur motivierte Entwickler übernehmen Verantwortung für ihre Arbeit und sind besorgt um die Qualität ihrer Arbeit.
- Die **Verantwortung für die Veröffentlichung** tragen die Entwickler. Sie bestimmen, wann neue Versionen veröffentlicht werden.
- Automatisierte Veröffentlichung erleichtert es, den Entwicklern die Verantwortung für die Veröffentlichung zu übertragen. Durch die automatisierte Veröffentlichung ist der Aufwand für die Veröffentlichung selber niedrig.
- Von den Entwickler wird **Rechenschaft** für ihre Software gefordert. Sie müssen wissen, ob ihre Software funktioniert und sie müssen reagieren, wenn Versagen auftreten. Zusammen mit der Verantwortung für die Veröffentlichung und der Überwachung bedeutet das, dass sie nach einer Veröffentlichung immer sicherstellen müssen, dass die Software richtig funktioniert.
- Die Entwickler haben die Möglichkeit Produktentscheidungen aufgrund technischer Erwägungen **mit zu bestimmen**. Das bedeutet, die Meinung der Entwickler zur technischen Umsetzbarkeit hat bei Produktentscheidungen Gewicht und es werden ihnen nicht nur Anforderungen diktiert. Ist diese Mitbestimmung nicht gegeben, identifizieren sich Entwickler nicht mit ihrem Produkt.

Als maßgeblichen technischen und organisatorischen Faktor, beobachtete ich die **Modularisierung auf Teamebene**. Am wichtigsten ist, dass ein Team ein Modul technisch und organisatorisch alleine kontrolliert und seine Software nicht stark von anderen Teams abhängig ist. Die Modularisierung ist keine rein technische, sondern auch ei-

ne organisatorische Angelegenheit. Die Modularisierung auf Teamebene hat positive Auswirkungen auf fast alle Einflussfaktoren:

- Interne Qualität: bessere Modularisierung der Software.
- Produktvision: Es ist leichter für ein kleines Modul die Produktvision zu kennen.
- Gute, automatisierte Tests: Kleinere Module sind leichter zu Testen, weil an den Modulgrenzen gemockt werden kann. Da sie kleiner sind, haben sie weniger Tests und dadurch ist deren Laufzeit geringer.
- Wenig Zeit bis Veröffentlichung: Kleine Module können schneller und häufiger veröffentlicht werden.
- Verantwortlichkeit für Veröffentlichung: Es kann so modularisiert werden, dass die einzelnen Module separat veröffentlicht werden können. Dadurch ist es leichter die Verantwortung dafür dem Team zu geben.
- Klare Grenzen in der Software: Wenn ein Team ein eigenes Modul kontrolliert, ist klar, dass die Verantwortung des Teams an den Modulgrenzen anfängt und aufhört.
- Rechenschaft: Es ist leichter, für einzelne Module Rechenschaft zu fordern als für eine große Applikation.
- Mitbestimmung: Ein Team kann leichter Entscheidungen für sein eigenes Modul treffen, als für eine stark verzahnte, große Applikation.

Abbildung 1 fasst diese Einflussfaktoren noch einmal zusammen. Ich habe die Idee des Qualitätserlebnisses und die Einflussfaktoren Entwicklern und Managern von SoundCloud und Immobilienscout24 vorgestellt. Von ihnen habe ich Verbesserungsvorschläge erhalten (die in diese Beschreibung eingeflossen sind), aber keine großen Einwände gegen das Modell und seinen Implikationen für die Arbeitsweise eines Teams erhalten. Das Modell spiegelt also deren Arbeitsrealität wider.

Als nächstes werde ich beschreiben, wie das Qualitätserlebnis und die Einflussfaktoren bei den untersuchten Teams ausgeprägt sind.

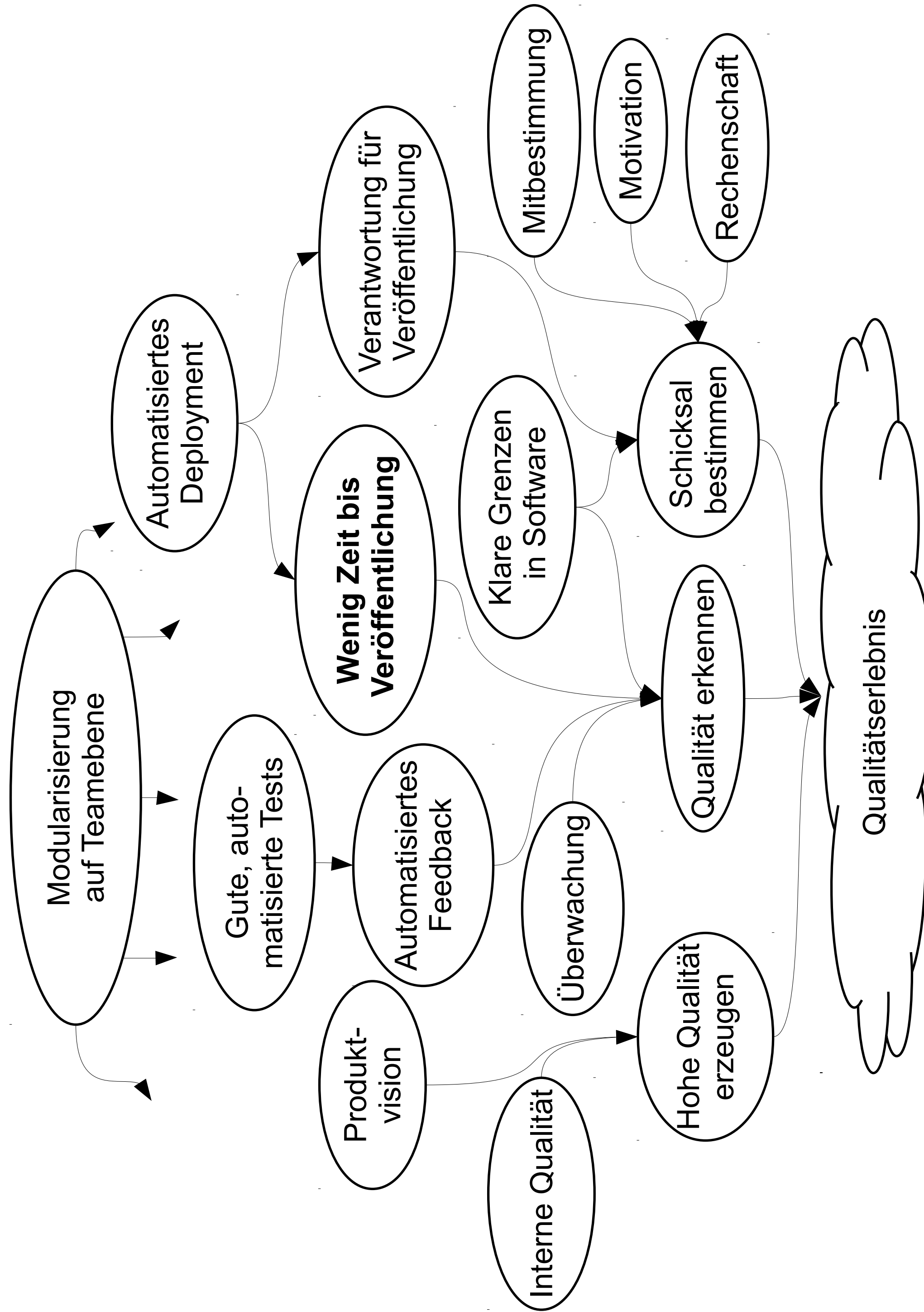

Abbildung 1: Einflussfaktoren für ein starkes oder schwaches Qualitätserlebnis. Modularisierung auf Teamebene ist ein ermöglichender Faktor, der in die meisten anderen Faktoren hineinspielt, daher sind die Pfeile nur angedeutet.

6 Qualitätserlebnis in den Teams

Im Folgenden beschreibe ich das Qualitätserlebnis in den untersuchten Teams und die Ausprägungen der Einflussfaktoren.

Team Payment (PM) hat ein sehr starkes Qualitätserlebnis. Die Entwickler bekommen sehr schnell und sehr deutlich Feedback zu ihrer Arbeit. Sie bekommen es sehr schnell, da sie mehrmals pro Woche veröffentlichen. Das Feedback ist sehr direkt, weil sie die Ursachen von Problemen direkt auf die eigene Arbeit zurückführen können. Es ist unmittelbar, weil sie ihre Software im Betrieb selber überwachen und bei Problemen selber reagieren müssen. Durch dieses starke Qualitätserlebnis entdecken sie Probleme sehr schnell und reagieren sehr schnell darauf. Dadurch stellen sie, ohne nachgelagerte Testphase, eine hohe Qualität sicher.
Das Team PM erreicht dieses starke Qualitätserlebnis, indem sie die (1) Voraussetzungen, ein Produkt mit hoher Qualität zu erzeugen schaffen, (2) die Auswirkungen der Qualität ihrer Arbeit deutlich auf ihre eigene Arbeit zurückführen und (3) das Schicksal ihres Produkts selber bestimmen können. Der wesentliche Grund, der das ermöglicht, ist, dass sie an einem eigenständigen Modul arbeiten, das sie komplett kontrollieren.

Das Team Online Marketing (OM) ist dem Team PM von SoundCloud sehr ähnlich: Sie spüren die Auswirkungen der Qualität ihrer Arbeit ebenfalls sehr schnell, weil sie mehrmals pro Woche veröffentlichen, sehr direkt, weil sie die auftretende Effekte auf ihre Arbeit zurückführen können und unmittelbar, weil sie ihre Software selber im Betrieb überwachen. Dadurch erkennen sie Probleme ebenfalls sehr schnell und reagieren schnell darauf. So stellen sie auch ohne nachgelagerte Qualitätssicherung eine hohe Qualität sicher.
Sie erreichen dieses starke Qualitätserlebnis, dadurch, dass sie ebenfalls eigenständige Module haben, (1) die Voraussetzungen, Qualität zu erzeugen schaffen, (2) die Auswirkungen der Qualität ihrer Arbeit gut erkennen können und (3) das Schicksal ihres Produkts weitgehend selber bestimmen. Auch auf tieferen Ebenen gibt es viele Ähnlichkeiten zu Team PM.

Interessant ist, dass bei Team PM die Ausgangssituation bereits so gegeben war, bei Team OM aber bis vor ein paar Jahren eine starke organisatorische Trennung zwischen Aufgaben wie Planen, Entwickeln, Qualitätssicherung und Veröffentlichung bestand. Das Team stellte allerdings Probleme mit dieser Arbeitsweise fest. Es hat nach und nach die Verantwortung für diese Aufgaben übernommen. Zum Beispiel wurde die Veröffentlichung automatisiert und an die Entwickler gegeben und die Entwickler fingen an, mehr Tests selber zu schreiben. Die Entwickler hatten den Wunsch, mehr Verantwortung zu übernehmen und haben jetzt, mit dieser Verantwortung, das Gefühl, besser arbeiten zu können.

Im Gegensatz zu den beiden vorherigen Teams, hat Team Anbieten Profi (AP) ein schwaches Qualitätserlebnis: Auswirkungen ihrer Arbeit bekommen sie nur verzögert, diffus und mittelbar mit. Verzögert, weil es einen 8-tägigen Releaseprozess gibt, auf den sie

keinen Einfluss haben. Diffus, weil ein Effekt an der Software viele verschiedene Ursachen haben, weil mehrere Teams an einer Codebasis arbeiten. Mittelbar, weil sie selbst nur wenig darüber wissen, wie die eigene Software im Betrieb funktioniert, weil dafür ein anderes Team zuständig ist. Allerdings biete die nachgelagerte Testphase eine hohe Sicherheit.
Die Voraussetzungen für ein starkes Qualitätserlebnis sind beim Team AP vor allem durch die lange Verzögerung bis zur Veröffentlichung nicht gegeben.

6.1 Hohe Qualität erzeugen

Die Teams PM und OM haben die Fähigkeit Software von hoher interner und externer Qualität zu erzeugen. Im Team AP sind diese Voraussetzungen zum Teil nicht gegeben, weil es Teile der Software gibt, die nicht mehr dem technischen Stand entsprechen. Die Teams erzeugen hohe interne Qualität durch klare Codestandards, Reviews und Zeit für technische Verbesserungen. Die Produktvision ist allen Entwicklern sehr deutlich, da sie viel mit dem PO kommunizieren und früh einbezogen werden.

6.1.1 Interne Qualität

Die Teams verwenden verschiedene Maßnahmen, interne Qualität zu erzeugen und zu erhalten. Dazu gehören klare Vorstellungen, wie guter Code geschrieben wird und regelmäßige Investitionen in Form von Zeit in technische Verbesserungen.

In allen Teams ist der Code aufgeräumt und nach klaren, konsistenten Standards geschrieben. Die Teams OM und AP arbeiten nach den Regeln von Clean Code und Pair Programmen regelmäßig bei Features. Im Team OM konnte ich dabei ein **„aggressives“ Korrekturverhalten** beobachten: Sieht ein Entwickler im Code etwas, was seiner Meinung nach nicht den Regeln entspricht, spricht er den Schreiber darauf an und diskutiert das Problem. Bei den anderes Teams konnte ich das nicht explizit beobachten, es könnte aber vorhanden sein. Im Team PM werden Reviews für sämtliche Änderungen durchgeführt.
Team PM hat klare Codestandards, die viele Architekturregeln formulieren. Team OM und Team AP führen **Sonarqube**[11]-Tests für umfangreiche **statische Codeanalysen** durch. Beide Teams investierten Zeit darin, Regelverletzungen aus ihrem Code zu entfernen. Team AP führt nach Änderungen, die Sicherheitslücken enthalten können ein Sicherheitsreview mit Fokus auf häufige Sicherheitsprobleme, wie richtiges Escapen, Exploitmöglichkeiten und Performanceprobleme durch.

Team PM und Team OM haben in ihrer Software keine ungewarteten Teile. Team OM hatte 2011 viele strukturelle Schwächen im Code und viele ungelöste Defekte. Durch schrittweise Verbesserungen der Schwächen mit Unterstützung des OP, schaffte das

[11]Sonarqube ist eine Art CI für statische Codeanalyse, der bei jedem Commit viele verschiedene Tools zur statischen Codeanalyse ausführt und die Ergebnisse in einer leicht verständlichen Übersicht darstellt.

Team eine stabile, gut gewartete Codebasis. So sind von früher 300 offenen Defekten alle geschlossen. Neue Defekte werden meistens innerhalb eines Tages repariert. Team PM hat ebenfalls keine länger offenen Defekte und neue Defekte werden innerhalb eines Tages repariert. Die Teams erreichten dieses hohe Niveau, indem sie, in Zusammenarbeit mit dem PO, regelmäßig Zeit für technische Verbesserungen verwendeten.

Der Code der Kernapplikation, für den Team AP zuständig ist, ist gut gepflegt und größtenteils gut wartbar. Es gibt aber Teile, deren Technologie nicht mehr aktuell ist und in die keiner der Entwickler mehr eingearbeitet ist. Laut PO behindern diese ungewarteten Teile oft die Entwicklung von Features, da sie sich nur schwierig ändern lassen. Während eines Sprints sind zwei Entwickler nur mit dem Beheben von Defekten beschäftigt.
Bis Sommer 2013 hatte das Team keine separate Zeit, um im Rahmen des Sprints diese strukturellen Schwächen zu beseitigen. Seitdem wendeten sie 20% eines Sprints für die Beseitigung dieser Schwächen auf. Dadurch konnten bereits viele strukturelle Schwächen, in Form von veralteter, unwartbarer Software repariert werden. Diese Zeit reichte aber nicht, um größere, veraltete Softwareteile abzulösen. Dem Team sind diese Probleme bewusst und drei separate Entwicklern arbeiten seit Januar 2014 daran, diese veralteten Teile abzulösen.

6.1.2 Produktvision

In allen Teams lebt die Produktvision in den Köpfen der Entwickler. Die Teams erreichen das, indem die Entwickler früh und stark in die Planung involviert sind und oft Rücksprache mit den POs halten.

Am Wichtigsten ist, dass in allen Teams die POs mit den Entwicklern in einem Raum sitzen und sie viel mit ihnen kommunizieren. Die POs begrüßen diese vielen Rückfragen der Entwickler, da sie so Anforderungsfehler vermeiden. Der PO im Team OM führte z.B. an, dass Abnahmen fast immer erfolgreich sind, die Entwickler seine Anforderungen also meist sehr gut verstehen. Im Team PM ist die Produktvision so gut, dass keine explizite Akzeptanzkriterien benutzen, sondern nur grobe ToDo-Punkte.
Damit die Entwickler die langfristige Entwicklung des Produkts kennen, stellen die POs den Entwicklern früh neue Anforderungen vor und diskutieren sie mit ihnen. Bei Team AP gibt es dafür ein eigenes Meeting und bei Team OM sind langfristig geplante Features am Planungsboard sichtbar. Im Team PM planen Entwickler und PO zusammen, welche Features im nächsten Quartal umgesetzt werden können.

Der PO vom Team schätzt diese hohe Kenntnis der Produktvision durch die Entwickler. PO Team PM: *„Ich finde das super wichtig und super gut, dass die Entwickler [...] dass die verstehen. Die haben den Scope. Das ist eine superwichtige Sache um als Team zu funktionieren, dass alle wissen worum es geht. "*.

Inwiefern die Produktvision die Kundenbedürfnisse erfüllt, konnte ich nicht beurteilen.

6.2 Qualität erkennen

Die Entwickler der Teams OM und PM können die Qualität ihrer Arbeit gut erkennen. Sie überwachen die Software im Betrieb, sie bekommen schnelles, automatisiertes Feedback der CI, es vergeht wenig Zeit zwischen Veröffentlichungen und durch die Modulgrenzen gibt es klare Grenzen.

Für Team AP ist es schwieriger, die Auswirkungen der Qualität der eigenen Arbeit zu erkennen. Das automatisierte Feedback ist langsamer, die Veröffentlichung von Änderungen erfolgt erst nach mindestens einer Woche und die auftretenden Effekte (z.B. Versagen) können viele sich vermischende Ursachen haben, weil mehrere Teams an der gleichen Codebasis arbeiten.

6.2.1 Gute, automatisierte Tests

Alle Teams haben die gleichen Ziele für ihre Tests: Sie sollen die **Korrektheit** überprüfen, **schnelles Feedback** liefern und **verlässliche** Aussagen über Änderungen, unabhängig von externen Einflüssen, geben.

In allen Teams sind die Tests das wichtigste Mittel, um nach Änderungen zu prüfen, ob die Software korrekt funktioniert. Die Tests der Teams OM und PM liefern dieses Feedback schnell und sind verlässlich.
Im Gegensatz zu den anderen beiden Teams testen die Tests von Team AP nicht nur ein einzelnes Modul, sondern die gesamte Kernapplikation. Dabei testen sie nicht nur die einzelnen Komponenten für sich, sondern integrieren sie. Dabei wird auch die Integration mit Modulen getestet, die nicht mehr Teil der Kernapplikation sind. Außerdem gibt es in der Kernapplikation viele Abhängigkeiten im Code und selbst kleine Änderungen können an entfernten Stellen Auswirkungen haben. Daher hat das Team AP deutlich mehr Tests, die langsamer auszuführen sind, als die anderen beiden Teams.

Team Payment strukturiert die Tests wie eine **Pyramide**: Viele Unittests testen kleinteilig, Integrationstests umfassend das Modul und wenige Webtests testen mit externen Abhängigkeiten. Dadurch sind die Unit- und Integrationstests schnell und verlässlich, die Webtests entdecken Probleme mit anderen Komponenten und die Integrationstests können andere Teams als API-Dokumentation verwenden.
Die ca. 800 **Unittests** werden beim Entwickeln geschrieben. Für jede public Methode einer Klasse muss ein Unittest existieren. Unittests benutzen richtige Objekte aus der Datenbank (mit Testdaten), aber externe Dienste werden gemockt. Die Unittests werden bei jedem Speichern ausgeführt und liefern sofort (d.h. beinahe instantan) Feedback.
Die ca. 200 **Integrationstests** integrieren die Klassen aus Buckster und mocken externe Module weg. Die Integrationstests werden ebenfalls bei jedem Speichern ausgeführt und liefern schnell (d.h. in weniger als 10 Sekunden) Feedback.
Letzte Instanz sind ca. 20 **Selenium Browsertests**, die in einem separatem Verzeichnis liegen und manuell angestoßen werden müssen. Sie testen die Integration mit ande-

ren Diensten. Diese Tests testen die **Arbeitsabläufe** auf der Homepage, und ob sie so funktionieren wie sie sollen. Es wird nicht zu jeder Story ein End-To-End-Test geschrieben, sondern eher einer zu einem Feature (mit positivem und negativem Fall). Die Tests auszuführen dauert **ca. 10 Minuten** und das Team führt sie normalerweise nur vor der Veröffentlichung aus. Es wird nicht veröffentlicht, ohne dass diese Tests ein positives Ergebnis liefern.
Die Entwickler investierten am Anfang der Migration viel Zeit darin, eine gute Teststrategie zu entwerfen und sind nach mehreren Iterationen bei dieser angelangt. Die Strategie ist seit April 2013 stabil und wird nicht mehr wesentlich geändert. Tests zu schreiben gehört fest zum Entwickeln eines Features dazu und steht zeitlich nicht in Konkurrenz dazu.

Team Online Marketing benutzt ebenfalls drei Ebenen, um die Tests zu strukturieren, allerdings mit fließenden Grenzen. Auf den niedrigeren Ebenen gibt es mehr Tests, auf den höheren Ebenen eher weniger:

- **Unittests**, die fast alles außer die zu untersuchende Klasse mocken
- **Integrationstests**, die externe Dienste benutzen können (aber auch mocken können), aber nicht über einen Browser gehen
- **Webtests**, die mit Selenium über einen Browser auf die Seite zugreifen

Außer den Webtests, werden die Tests beim Entwickeln bei jedem Speichern ausgeführt und sind sehr schnell (insgesamt < 2 Min., durch Integration mit IDE werden für eine Änderung alle relevanten Tests fast instantan ausgeführt). Die Webtests auszuführen dauert ca. 3 Minuten. Es gibt keine geschriebenen Regeln, wie genau Tests aussehen müssen, aber es gibt ein paar Punkte auf die alle Entwickler achten:

- Es muss einen Unittest zu einer Funktionalität (z.B. zu einer Auflistung, einer Aktion oder einer Abfrage)
- Unittests müssen so strukturieren sein, dass sie leicht erfassbar sind – d.h., dass sichtbar ist, was getestet wird und wie. Das wird vor allem durch Hilfsfunktionen zur übersichtlicheren Strukturierung erreicht
- Sie sollen so schnell wie möglich sein (Laufzeit)

Lokal und auf dem Entwicklungsserver werden externe Abhängigkeiten **weggemockt**, damit die Tests schnell ausgeführt werden können und ihr Ergebnis nicht von der Verfügbarkeit externer Dienste abhängig ist. Das Team hatte lange Zeit Probleme mit **flackernden** Tests, d.h. Tests die erfolgreich oder nicht erfolgreich sind, abhängig davon, ob externe Abhängigkeiten funktionierten oder nicht.
Die Tests werden mit **Sonarqube** mit analysiert und Verletzungen der Stilregeln werden auf Teammonitor angezeigt:

- Die **Code Coverage** durch Unittests muss $> 85\%$ sein
- Es darf keine ignorierten Tests geben

Die Tests waren bis vor ca. 2 Jahren in schlechten Zustand: Der Code war schwer zu warten, sie waren langsam und unverlässlich (flackerten). Das Team hat viel Zeit investiert, den Testcode zu pflegen, aufzuräumen und in guten Zustand zu bringen. Außerdem hat es viel Zeit investiert, die Laufzeit der Tests zu reduzieren, indem es die Anzahl der Webtests reduzierte und es ermöglichte, sie parallel auszuführen. Außerdem investierte es viel Zeit darin, die Verlässlichkeit der Tests zu erhöhen, indem es mehr Mocking verwendete. Tests zu schreiben gehört fest zum Entwickeln und steht zeitlich nicht in Konkurrenz dazu.

Team Anbieten Profi muss, zum Überprüfen der Korrektheit, die Tests für die gesamte Kernapplikation ausführen, weil es viele Abhängigkeiten innerhalb der Applikation gibt. Die Tests der Kernapplikation geben gut Auskunft über die Korrektheit der Software, allerdings sind sie langsam.
Die Tests sind strukturiert wie eine **Pyramide** mit vier Ebenen: Unit-, Integrations-, System- und Webtests mit jeweils weniger Mocking auf höheren Ebenen und längerer Laufzeit

- **Unittests**: ca. 14.500. Testen nur die Komponente unter Test, der Rest wird weggemockt. Laufzeit ca. 5 Minuten.
- **Integrationstests**: Testen die Integration mit externen Diensten wie Datenbanken, aber ohne Webcontainer. Laufzeit ca. 10-20 Minuten.
- **Systemtests**: Testen mit dem kompletten Webcontainer, aber ohne Webbrowser, Laufzeit ca. 2-5 Minuten.
- **Webtests**: ca. 2000. Testen durch die Weboberfläche auf dem Previewsystem. Von diesen 2000 sind ca. 200 “Smoke Tests” (d.h. die wichtigsten), die mit jedem CI-Lauf ausgeführt werden. Laufzeit (der Smoke Tests) ca. 7-11 Minuten.

Die Tests sind, ähnlich wie die von Team OM, so strukturiert dass sie leicht erfassbar sind. Ein Testlauf dauert ca. 20-40 Minuten. Dies ist bereits eine große Verbesserung zu früher, aber die Entwickler wollen die Laufzeit weiter reduzieren. Zum Beispiel wählten sie die Smoke Tests aus und versuchen die Infrastruktur zu optimieren.
Hierbei ist zu bedenken, dass die Tests, im Gegensatz zu denen der anderen Teams, mehrere Komponenten miteinander integrieren. Laut Aussage der Entwickler ist es durch die Struktur der Kernapplikation mit vielen Abhängigkeiten ineinander nötig, alle diese Tests nach Änderungen auszuführen. Wäre dies nicht der Fall, würden viele Fehler unentdeckt bleiben. Außerdem arbeiten, im Gegensatz zu den anderen Fällen, mehrere Teams an dieser Codebasis unabhängig voneinander. Dadurch ist der Koordinationsaufwand um die Tests zu schreiben höher.
Die Webtests schlagen von Zeit zu Zeit aufgrund von Infrastrukturproblemen fehl, und nicht wegen Fehlern im Code. Dadurch sind sie nicht ganz verlässlich. Allerdings ist das Problem laut Aussage der Entwickler nicht so groß, da sie trotzdem gut einschätzen können, ob eine Änderung Fehler verursacht hat.
Insgesamt ist die Testabdeckung gut: Jeder neu geschriebene Code ist durch Tests abgedeckt, viel von dem alten Code ist getestet.

Neben den automatisierten Tests, testet im Team AP die Testingenieurin jede Story ausführlich **manuell und explorativ**, nachdem die Entwickler fertig sind.

6.2.2 Automatisiertes Feedback

Bei Team OM und Team PM wird nach jedem Commit in wenigen Minuten die Software gebaut, alle Tests ausgeführt und den Entwicklern das Ergebnis angezeigt. Diese Ergebnisse sind dank der guten Tests verlässlich und aussagekräftig. Bei Team OM gehören dazu auch die Ergebnisse der statischen Codeanalyse. Bei Problemen (z.B. roten Tests) hat der Verursacher (mit dem letzten Commit) ein wenig Zeit, das Problem zu beheben, bis seine Kollegen ihn darauf hinweisen.

Bei Team AP dauert ein Lauf der CI ca. 20-40 Minuten. Für die Entwickler kommt dadurch das Feedback über Änderungen erst sehr spät: Entwickler Team AP: *„Die Tests brauchen sehr lange. Das läuft und läuft und erst nach 45 Minuten kriegst du mit: Mist ist fehlgeschlagen. [...] Die ganze Zeit sieht es gut aus und nach 45 Minuten tritt 1 blödes Problem auf. Das Feedback ist dann* ***zu spät****. Man versucht ja eigentlich herauszufinden, dass ein Change nichts kaputt macht."* . Es gibt noch einige kleinere Probleme, Ursachen von fehlgeschlagenen Tests zu finden. Das größte Problem ist allerdings, dass das Feedback über Codeänderungen erst so spät kommt. Den Teams der Kernapplikation ist dieses Problem bewusst, und sie planen verschiedene Maßnahmen, um die Laufzeit der CI zu reduzieren.

6.2.3 Überwachung

Die Teams OM und PM sammeln Metriken der Software im Betrieb, wie Art von HTTP-Responses, Veröffentlichungen, Anzahl Verkäufe, Traffic und Kontakte. Diese Daten zeigen sie auf einem großen Monitor im Teamraum an. Dadurch können sie Probleme schnell erkennen (z.B. sinkende Verkaufszahlen nach einer Veröffentlichung) und darauf reagieren.

Das Team AP hat keine Überwachung ihres Teils der Software im Betrieb. Der Scrum Master begründet dies damit, dass das Team kein Bedürfnis nach dieser Überwachung hat, weil es ohnehin zu lange dauert bis eine Änderung veröffentlicht wird, um sinnvoll auf Metriken zu reagieren.

6.2.4 Automatisiertes Deployment

SoundCloud hat ein Tool namens **Bazooka** entwickelt, dass es Entwicklern ermöglicht sehr einfach produktiv zu deployen. Bazooka baut automatisch aus einem speziellen git-repository anhand einer Makefile ein deploybares Artefakt. Mit wenigen Befehlen kann dieses auf beliebig viele Server deployt und skaliert werden.

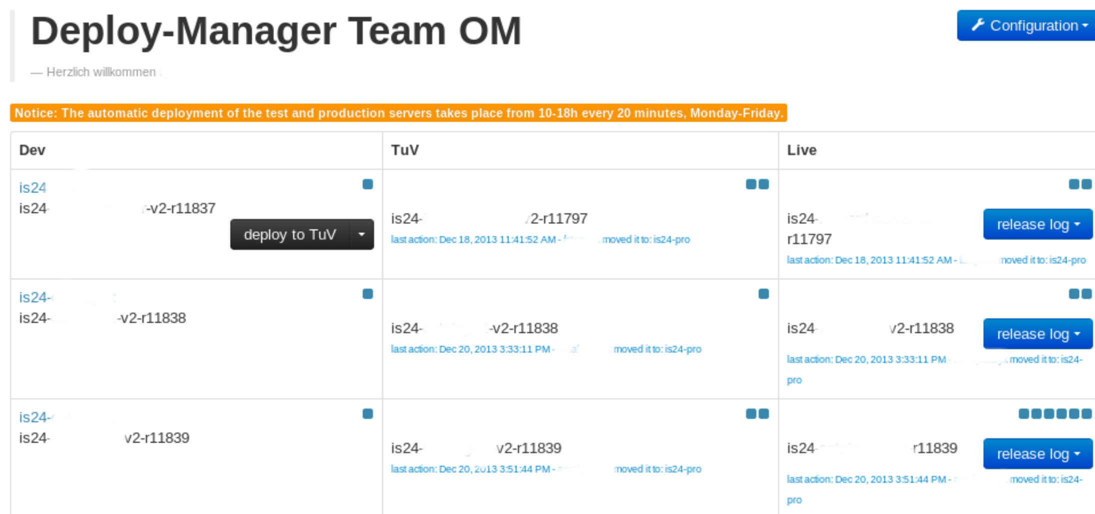

Abbildung 2: Der DeployManager. Für eine Modul gibt es eine Zeile und es kann per Klick auf *„deploy ...“* auf das Test und auf das Livesystem deployt werden. Unter dem Ausklappknopf neben *„deploy“* wird angezeigt, welche Commits (mit Commitnachrichten) in dieser Version neu sind.

Team OM hat den **Deploymanager** (siehe Abbildung 2) entwickelt, ein Tool um Anwendungen per Knopfdruck auf einem Testsystem und live zu deployen.

Das Team AP kann nicht automatisiertes depoyen. Deployt wird manuell von Ops und die Entwickler haben keine Möglichkeit das zu beeinflussen.

6.2.5 Wenig Zeit bis Veröffentlichung

Keines der Teams arbeitet nach Continuous Deployment, der sofortigen Veröffentlichung von Änderungen. Team PM veröffentlicht aber ca. alle 2-3 Tage eine neue Version ihrer Software und in Team OM wird meistens höchstens einen Tag nachdem ein Entwickler seine Arbeit abgeschlossen hat ist veröffentlicht. Einer der Hauptgründe, warum Team OM und Team PM keinen Scrum-Prozess verwenden, ist, dass der feste Sprintzyklus zu einschränkend für die Veröffentlichungen wäre.

Beide Teams planen Features so, dass sie häufig veröffentlichen können und wenig Zeit bis zu einer Veröffentlichung vergeht. Team PM migrierte aus diesem Grund Schritt für Schritt einzelne Bezahlvorgänge aus dem Altsystem. Im Team OM ist es ähnlich: Um eine Legacyapp (des Teams OM) zu migrieren, in der es mehrere Regionen und Haustypen gibt, wurden nach und nach einzelne Regionen und Haustypen migriert. Als alle Anfragen von der neuen Applikation bedient wurden, wurde die alte Applikation abgeschaltet.

Bei Team AP dauert es mindestens 8 Tage, bis eine Änderung veröffentlicht wird. Dienstag morgens wird die aktuelle Version aus dem Quellcodeverzeichnis auf ein Testsystem gespielt und die Testabteilung testet diese bis zum nächsten Dienstag. Wenn ein Entwickler Dienstag Nachmittag die Entwicklung an einem Feature abschließt, dauert es also 15 Tage bis die Veränderung veröffentlicht wird. Laut den Entwicklern ist diese Verzögerung so lang, dass, wenn eine Änderung veröffentlicht wird und Defekte auftreten, sie diese Arbeit nicht mehr im Kopf haben.

6.2.6 Klare Grenzen in der Software

Team OM und Team PM arbeiten an klar abgegrenzten Module, die sie komplett kontrollieren. Team OM hatte immer einen Aufgabenbereich, den es separat vom Rest der Firma bearbeitete, daher ist auch die Software unabhängig vom Rest der Firma.
Im Team PM gibt es klare Grenzen durch die Modulgrenzen: Das Team trägt die Verantwortung für das Modul Buckster. Das Modul und damit auch der Verantwortungsbereich des Teams umfasst alle die Bezahlung betreffenden Aufgaben. Im Mothership waren diese Funktionen über viele Codestellen verteilt. Dadurch kam es oft zu Problemen, weil andere Entwickler etwas änderten und sich nicht bewusst waren, dass es Auswirkungen auf die Bezahlfunktionen hat. Daher kam überhaupt die Motivation diese Funktionen in ein separates Modul zu migrieren.

In der Kernapplikation gibt es keine klaren Grenzen. Das Team AP ist für den ScoutManager verantwortlich, aber andere Teams arbeiten ebenfalls daran. Diese Vermischung ist deutlich erkennbar an der Zeit, die die Teammitglieder damit verbringen, sich mit anderen Teams zu koordinieren. Werden Defekte gefunden (ob durch einen manuellen Test oder einen automatisierten Tests), kann es mühsam sein, die Ursache (oder den Verursacher) zu finden, da es viele technische und organisatorische Abhängigkeiten gibt.

6.3 Schicksal bestimmen

Team OM und Team PM können das Schicksal ihrer Produkte weitgehend selber bestimmen, Team AP dagegen ist von vielen externen Faktoren abhängig. Team OM und Team PM können die Entwicklung ihrer Module selbst bestimmen, sie bestimmen, wann neue Versionen veröffentlicht werden und sie sind verantwortlich dafür, dass sie im Betrieb funktionieren. Im Team AP können die Entwickler, aufgrund der Struktur Kernapplikation, nur bis zu einem gewissen Grade die Entwicklung mitbestimmen. Sie können nicht selbst veröffentlichen und sind nicht für den Betrieb zuständig.

6.3.1 Mitbestimmung

In allen Teams werden die Entwickler früh mit in die Planung einbezogen und können die fachliche Planung aufgrund technischer Erwägungen beeinflussen. Dadurch identifizieren sich die Entwickler mit ihren Produkten. Im Team AP gibt es dabei Einschränkungen, weil ihre Software in eine größere Applikation eingebunden ist, und sie daher nicht frei entscheidungen treffen können.

Die konkreten Planungen der nächsten Arbeiten sind in allen Teams ein Dialog zwischen PO und Entwicklern. Hierbei werden fachliche Anforderungen so angepasst, dass sie ohne ungerechtfertigt hohen Aufwand umgesetzt werden können. Dabei wissen die POs das technische Wissen der Entwickler sehr zu schätzen und lassen sich von ihnen in Produktentscheidungen beeinflussen. Sie begründen das in allen drei Fällen damit, dass ihre Produkte technische Produkte sind (auch wenn sie sehr verschieden sind) und sie die technischen Aspekte nicht außen vor lassen können. Der PO von Team PM beobachtet, dass die Entwickler sich durch diese Involvierung stärker an das Produkt binden. PO Team PM: *„Und dadurch sind die recht stark involviert darin, wie das Produkt nachher wird. Was auch OK ist. Ich glaube dadurch bindet man sich auch stärker ans Produkt und ist nicht nur Abarbeiter von Befehlen.“*.

Beispielsweise migrierte das Team PM erst die Geschenkgutscheine, die wenig Traffic verursachen, aus dem Mothership. Aus geschäftlicher Sicht wäre es sinnvoller gewesen, gleich mit der Abonnement-Erneuerung zu beginnen, weil das neue System einen neuen Abrechnungspartner benutzte. Dieser Partner hätte vorzugsweise von Anfang an viele Anfragen abgewickelt. Allerdings überzeugten die Entwickler den PO, zuerst mit den Geschenkgutscheinen anzufangen, weil sie so die neuen Funktionalitäten erst mit wenigen Nutzern testen können und auftretende Probleme leichter zu lösen sind.
Der Entwickler vom Team PM bestätigte, dass er durch diese Einflussmöglichkeiten sich stärker mit seiner Software identifiziert.
SoundCloud strukturierte die gesamte Software und Infrastruktur so, dass ein Team genügend technische und organisatorische Fähigkeiten hat, um derartige Entscheidungen auf Teamebene zu ermöglichen.

Im Team OM bezeichnet der PO seine Arbeit mit den Entwicklern als **kooperativ**. Er formuliert zwar die Anforderungen vor, diskutiert sie aber mit den Entwicklern. Die Entwickler sollen mit ihrem technischen Wissen beeinflussen, wie Features umgesetzt werden (weil viele Anforderungen des Teams technischer Natur sind, z.B. Headerformate). Weil sich fachliche und technische Planung immer gegenseitig beeinflussen, werden sie zusammen durchgeführt.
Die Entwickler entwerfen von Zeit zu Zeit Features. Beispielsweise erarbeiteten Entwickler, zusammen mit fachlichen Anforderern, in einem Kreativworkshop ein Produkt, mit teilweise technischen Anforderungen (z.B. war ein Feature des Produkts Testbarkeit):
Forscher: *„Inwiefern beeinflussen die Entwickler die fachliche Richtung?“*
PO Team OM: *„Das gibts auch auf jeden Fall. Wir haben das jetzt zum ersten Mal probiert,*

dass wir für ein komplettes Projekt [...], das einen sehr starken technischen Einfluss hat, gesagt haben da definieren wir die Produkte. Da ging es dann um Testing, wie kann ich bestimmte Funktionalitäten testen. [...] Wir haben das dann in einem Kreativworkshop erarbeitet und dann umgesetzt. Da waren die Entwickler, PO und 2 Onlinemarketingmanager mit dabei. Und das hat hervorragend funktioniert."

Der PO des Teams AP ist es sehr wichtig, die Entwickler früh bei der Planung einzubeziehen, weil sie dadurch die fachlichen Anforderungen besser mit den technischen Gegebenheiten vereinbaren kann. Allerdings ist es für das Team AP bei vielen technischen und fachlichen Aspekten schwierig, selber Entscheidungen zu treffen. Die Entwickler müssen sich dafür, wegen der monolithischen Softwarestruktur, in hohem Grade mit anderen Teams abstimmen. Z.B. können sie keine Änderungen an der Datenbank vornehmen und müssen sich in vielen Aspekten an die Vorgaben der Kernapplikation halten.

6.3.2 Verantwortlichkeit für Veröffentlichung

Die Teams OM und PM bestimmen selber, wann neue Versionen veröffentlicht werden. Im Team PM deployen die Entwickler, nach der Abnahme durch den PO eine Version mit einem einzigen Befehl. Im Team OM deployen die Entwickler technische Änderungen ohne sichtbare Änderungen selber live. Änderungen mit sichtbaren Unterschieden deployen sie auf auf das Testsystem, von dort deployt der PO sie nach erfolgreicher Abnahme live. Dies wird ermöglicht durch den DeployManager (vgl. Abildung 2).

Das Team AP hat nur sehr eingeschränkte Möglichkeiten, selbst zu bestimmen, wann neue Versionen veröffentlicht werden. Sie sind an den Releasezyklus der Kernapplikation gebunden. Nur in Ausnahmefällen können Reparaturen für kritische Defekte am Releaseprozess vorbei veröffentlicht werden.

6.3.3 Rechenschaft

In den Teams OM und PM sind die Entwickler Rechenschaft darüber schuldig, dass ihre Software funktioniert. Im Team AP ist dafür ein anderes Team zuständig.

Im Team PM sind die Entwickler von der Firma aus dafür verantwortlich, dass die Software funktioniert. Um das umzusetzen, hat das Team PM Tests in ihre Überwachungssoftware eingebaut, die, falls sie fehlschlagen die Benachrichtigung eines Entwickler auslösen. Wird außerhalb der Bürozeiten Alarm geschlagen, ist immer einer der Entwickler auf **Bereitschaft**. Dieser muss abwägen, ob er den Fehler sofort fixt oder bis zum nächsten Arbeitstag wartet. Der Entwickler vom Team PM schätzte, dass ca. 5 Fehler pro Woche außerhalb der Arbeitszeiten gemeldet werden, von denen die meisten aber kein Eingreifen erfordern.

Zusätzlich zu den Alarmen arbeitet das Team eng mit dem Community Support zusammen, um über Auffälligkeiten informiert zu werden. Der Teamleiter des Community Teams nimmt am morgendlichen Standup teil. Dadurch will das Team Fehler nach Veröffentlichungen eher identifizieren.
Um Fehlern vorzubeugen führte das Team die **5-Uhr-Regel** ein: Später als 2 Stunden vor Feierabend (also nach 17 Uhr) deployen sie nicht mehr. Die meisten Fehler innerhalb der ersten 2 Stunden nach Deployment auftreten und vor 19 Uhr (regulärer Feierabend) können sie selbst und die Kollegen noch reagieren.

Im Team OM ist es noch nie vorgekommen, dass die Software außerhalb der Bürozeiten repariert werden musste, daher gibt es auch keine Alarme. Sie sind aber für das Funktionieren der Software verantwortlich. Wenn z.B. eine neu veröffentlichte Version nicht funktioniert, müssen sie das Problem lösen.
Um solchen Problemen vorzubeugen deployen sie neue Versionen nur bis ca. 17:00 Uhr, weil es danach schwieriger wird, die passenden Ansprechpartner zu Problemen zu finden. Komplizierte Deployments, die mit anderen Systemen zusammenhängen versuchen sie eher am Anfang der Woche durchzuführen, um möglichst lange alle Ansprechpartner zur Verfügung zu haben.
Manchmal testen die Entwickler bei den manuellen Sichttests laut Aussage des Teamleiters nicht sorgfältig genug, z.B. in verschiedenen Browsern auf Anzeigefehler. Dadurch bekommen Nutzer von Zeit zu Zeit Fehler zu sehen. Er begründet dies mit zum Teil fehlender End-to-End Verantwortung. Es ist plausibel, dass der Grund für die fehlende Sorgfalt zu wenig Rechenschaft für Defekte ist.

Das Team AP ist nicht für den Betrieb der Kernapplikation verantwortlich, sondern ein separates Team, das **Application Management** (AM), betreut sie. Team AP schreibt für das AM ein **Handbuch**, das beschreibt, wie ihr Teil der Applikation funktioniert und wie er betrieben wird (z.B. welche Dienste wie gestartet werden dürfen). Für die Überwachung der Kernapplikation im Betrieb ist hauptsächlich das AM verantwortlich und greift bei Problemen ein. Die Entwickler könnten auch auf deren Messungen zugreifen, tun es aber selten. Diese Metriken sind für die gesamte Kernapplikation sind und damit für Team AP nicht sehr aussagekräftig für ihren Teil der Software.

6.3.4 Motivation

Das Team PM ist sehr hoch motiviert, ein gutes Produkt zu entwickeln und es zu betreiben. Da ist es nicht mal schlimm dafür Nachts geweckt zu werden:

Forscher: *„Wie schlimm findet ihr das, wenn ihr nachts geweckt werdet?“*
Entwickler Team PM: *„Meistens ist es gar nicht das Wecken, sondern die Implikationen, dass die Leute gar nicht upgraden können. Das wollen wir ja verhindern. [...] Ich hab ja den Anspruch, dass das System läuft.“* Außerdem konnte ich keine Frustrationspunkte erkennen.

Das Team OM ist laut Teamleiter ebenfalls motiviert. Die Entwickler empfinden die

Arbeitsweise mit umfassenden Verantwortlichkeiten als sehr angenehm und motivierend. Sie ermöglicht es ihnen Aufgaben ohne Unterbrechungen durch externe Einflüsse abzuschließen.

Die Entwickler des Team AP sind ebenfalls sehr motiviert ein gutes Produkt zu entwickeln, aber es gibt viel Frustration. Für die Entwickler ist es sehr frustrierend, dass sie nur langsames Feedback durch die Tests bekommen. Es behindert sie massiv im Entwickeln, wenn sie durch die lange Laufzeit der CI-Systeme nicht wissen, ob ihre Änderungen Tests fehlschlagen lassen.
Entwickler müssen sich oft mit anderen Teams koordinieren und müssen viel warten um Aufgaben abschließen zu können. Bei der Begleitung äußerte sich das dadurch, dass der Entwickler viel Zeit damit verbrachte, sich mit Entwicklern aus anderen Teams abzustimmen. Beispielsweise funktioniert an einem Tag sein lokaler Build nicht, wegen eines Commits aus einem anderen Team. Er musste erst den Committer suchen und konsultieren um das Problem zu lösen.

6.3.5 Modularisierung auf Teamebene

Die Teams OM und PM haben eigenständige Module, die sie komplett kontrollieren. Team Payment hat ein Modul (Buckster), über das es die komplette Kontrolle hat. Der Aufgabenbereich Online Marketing war bei IS24 immer ein separater Bereich, der durch spezielle Software gelöst wurde. Daher hat das Team Module, die unabhängig vom Rest der Software ist.

Die Software von Team AP ist eine große, monolithische Applikation. Daher gibt es keine Modularisierung auf Teamebene. Die Anwendung des Teams ist stark in die Kernapplikation integriert, an der mehrere Teams arbeiten.
Team AP sind diese Probleme bewusst. Seit Januar 2014 arbeiten drei Entwicklern daran, ihren Verantwortungsbereich aus der Kernapplikation zu lösen und in ein eigenständiges Modul zu migrieren.

Diese Modularisierung wird von den Firmen selbst vorangetrieben, weil die großen Applikationen, bei SoundCloud, wie bei Immobilienscout24 viele Probleme bereiten. An den Teams OM und PM kann man erkennen, dass die komplette Kontrolle eigenständiger Module ein starkes Qualitätserlebnis wesentlich begünstigt. Sie ermöglicht es u.A. den Entwicklern die Verantwortung für die Veröffentlichung und Überwachung zu übertragen und hat positive Auswirkungen auf das automatisierte Feedback und die Fähigkeit zur Mitbestimmung.

6.4 Effektivität QS bei Team Payment

Es ist sehr schwierig, objektiv die produzierte Qualität zu bewerten. Daher fragte ich jeweils denjenigen, der die Maßstäbe des Teams für externe Qualität festlegt: Den oder die PO.

Im Team PM ist er sehr zufrieden mit der Qualität, weil nur wenige Fehler passieren und diese sehr schnell behoben werden können. Es gab auch schon einmal einen größeren Ausfall, daraufhin wurden aber Maßnahmen (Monitoring) ergriffen, damit sich das nicht wiederholen kann.
Die interne Qualität ist wahrscheinlich ebenfalls hoch, weil das Team keine offene Bugs hat, selten Fehler passieren und wenn sie passieren, sie nur sehr wenige Benutzer betreffen.

Der PO des Teams OM ist ebenfalls sehr zufrieden: Forscher: *„Wie zufrieden bist du insgesamt mit der Qualität, die ihr produziert?“*
PO Team OM*: „Extrem zufrieden.“*
PO Team OM*: „Jetzt sind wir an einem guten Punkt, wo wir sehr effizient ist.“*

Die interne Qualität ist wahrscheinlich auch sehr gut. Das Team hat viel Zeit in technische Verbesserungen gesteckt und hat von früher 300 offenen Defekten alle repariert. Neu gefundene Defekte behebt es fast immer innerhalb eines Tages (d.h. dass der Fix live ist).

Im Team AP behindert die langsame Qualitätssicherung die Produktentwicklung massiv. Die PO sagt, sie ist nicht mutig, neue Sachen am Kunden auszuprobieren, weil es immer sehr lange dauert, Feedback zu bekommen. Sie würde gerne mehr AB-Tests durchführen, um die Akzeptanz von Features an Kunden auszuprobieren, aber das dauert momentan zu lang. Nachdem zwei Versionen (A und B) fertig gestellt sind dauert es 1-2 bis sie live sind, anschließend dauert es 2-3 Wochen um Daten zu sammeln. Dann dauert es 1-2 Wochen, bis die fertige Änderung live ist. Bei komplexeren AB-Tests, die mehrere Iterationen haben, kann sich dieser Prozess noch länger ziehen. Der Scrum-Master hat daher das Gefühl, dass das Team weniger Features am Markt ausprobiert und eher intuitiv entwickelt, weil sie durch das verzögerte Feedback kein Gefühl für die Kunden entwickeln können.
Die Entwickler sind oft von der großen Verzögerung im Zusammenhang mit Versagen frustiert. Zum einen werden Versagen von Benutzern und Testern erst, wenn die Entwickler bereits an etwas anderem arbeiten. Zum anderen dauert es zwei Wochen, nachdem ein Bugfix geschrieben wurde, bis dieser den Benutzern zugute kommt.

6.5 Firmenvorgaben

Bei SoundCloud herrschte von Anfang an eine Firmenkultur, die ein starkes Qualitätserlebnis begünstigt. Dort sind Entwickler für das Funktionieren der Anwendung verantwortlich, müssen diese mit Monitoring-Systeme überwachen und müssen außerhalb der Bürozeiten auf Bereitschaft sein. Um im Fehlerfall reagieren zu können, sind Entwickler für den Buildprozess verantwortlich und müssen ihre Software selber deployen. Dafür hat SoundCloud ein Tool für automatisiertes Deployment entwickelt. SoundCloud versucht immer vertikale Teams zu bilden, die genügend Verantwortlichkeiten vereinen um einen Verantwortungsbereich komplett zu kontrollieren.

Bei Immobilienscout24 gibt es ebenfalls einen Trend zu mehr Verantwortlichkeiten für Entwickler, allerdings ist er nicht so ausgeprägt. Für das Team OM waren im Januar 2012 die Planung, Entwicklung, Qualitätssicherung und der Betrieb deutlich getrennt. Der PO saß in einem anderen Raum, die Qualitätssicherung wurde von einer anderen Abteilung durchgeführt und für Deployments mussten sie die Ops-Abteilung anrufen. Jetzt übernehmen die Entwickler mehr dieser Aufgaben und sind stärker involviert in die gesamte Wirkungskette ihres Produkts. Die Entwickler mögen diese Arbeitsweise und wollten von sich aus mehr Verantwortung übernehmen.
Das Team AP arbeitet ebenfalls an mehr Autonomie durch Modularisierung, ist allerdings noch nicht sehr weit fortgeschritten.

7 Weitere Ergebnisse

Hier werden noch weitere Ergebnisse vorgestellt, die allerdings nicht so zentral sind, wie das Qualitätserlebnis und daher nicht so ausführlich behandelt werden.

7.1 Integrationstests

In den Firmen zeigt sich ein großer Nachteil der Modularisierung: Während in monolithischen Applikationen das Zusammenspiel von Komponenten sehr leicht getestet werden kann, erweist es sich als sehr schwierig das Zusammenspiel vieler unabhängig entwickelter Module zu testen. SoundCloud und Immobilienscout24 beobachten hierin Nachteile der Modularisierung und arbeiten an Lösungen für diese Probleme. Hierbei glänzt die Testabteilung von Immobilienscout24, da sie sehr ausführlich die Integration der Plattform testen kann.
Eine Möglichkeit, die erwogen wird, ist, die Integration erst live geschehen zu lassen, und sehr schnell von Nutzern entdeckte Probleme zu lösen. Eine andere ist, jedes Modul seine Integration mit anderen Modul selbst testen zu lassen. Welche Strategie für welche Kontexte geeignet ist, ist allerdings offen.

7.2 Geringe Anzahl Kommentare

Alle Teams benutzen Kommentare nur sehr sparsam, um ihren Code zu dokumentieren. Stattdessen verwenden sie sprechende Namen und strukturieren den Code so, dass er dadurch lesbar ist (Clean Code). Ein häufiger Einwand gegen diesen Stil ist, dass er die Einarbeitung erschweren soll. Der Entwickler aus Team OM hat sich selbst im März 2012 in die Software von Team OM eingearbeitet und es fiel ihm durch die gut lesbare Struktur des Codes sehr leicht.
Die Software vom Team PM wird von einigen anderen Teams integriert. Das Team PM hatte eine Zeitlang ein separates Wiki, in dem sie ihre API in natürlicher Sprache beschrieben. Dieses Wiki konnte aber nicht immer mit dem Code zu synchronisiert werden und war teilweise nicht aktuell. Stattdessen geben sie jetzt den anderen Teams ihre Integrationstests und haben das Wiki ganz entfernt. Die Integrationstests rufen die API-Funktionen des Dienstes des Teams auf und fungieren daher als API-Dokumentation. Die anderen Teams kommen sehr gut zurecht mit dieser Art der Dokumentation und das Team PM hat kein Problem sie immer auf dem neuesten Stand zu halten.

7.3 Engineer in Test

Bei SoundCloud gibt es Teams, die Probleme haben gute, automatisierte Tests aufzusetzen. Insbesondere bei den Anwendungen für mobile Endgeräte ist das sehr schwierig

und die Entwickler testeten die Anwendung manuell vor einer Veröffentlichung. Allerdings hatten sie oft massive Qualitätsprobleme. Statt ihnen, wie der traditionelle Ansatz wäre, manuelle Tester zur Verfügung zu stellen, hat SoundCloud einen **Engineer in Test** eingestellt, der dem Team hilft, gute, automatisierte Tests zu schreiben. Dafür schreibt er Frameworks, setzt die Testinfrastruktur auf und zeigt den Entwicklern, wie sie ihren Code testen können. Das Ziel dieser Maßnahme ist, dass die Entwickler weiterhin verantwortlich für die Qualität sind und nicht die Verantwortung abgeben.

Team OM hatte ähnliche Probleme: Die Tests waren nicht zufriedenstellend, weil sie langsam und nicht verlässlich waren. Das Team hatte nicht das Know-How um diese Probleme zu lösen. Eigentlich benötigte das Team einen Guru für automatisierte Tests, der ihnen zeigt wie sie ihre Tests strukturieren müssen und wo was getestet werden muss. Diese Rolle wäre vergleichbar mit dem Engineer in Test bei SoundCloud. Die Firma stellte allerdings keinen zur Verfügung, also haben sich die Entwickler dieses Wissen selber angeeignet.

7.4 Separate Tester

Das Team Anbieten Profi hat eine Testingenieurin, die hauptsächlich für das Schreiben der Webtests und die manuellen, explorativen Tests zuständig ist. Während das Team die Arbeit mit der firmenweiten Testabteilung als eher mühsam bezeichnete, weil sie mit hohem Koordinationsaufwand verbunden ist, klappte die Zusammenarbeit mit der Testingenieurin sehr gut. Das begründet sich in der guten Einbettung ins Team, die den Koordinationsaufwand mit ihr verringerte: Sie war in der Planung involviert und plante zusammen mit den Entwicklern, welche Aspekte auf welcher Ebene getestet werden. Ihr Feedback zu Defekten gab sie direkt und mündlich. Diese Art der Zusammenarbeit mit einem separaten Tester hat wahrscheinlich keine negativen Auswirkungen auf das Qualitätserlebnis.

7.5 Motivation

Die hohe Motivation in den Teams OM und PM lässt sich anhand der Ergebnisse einer systematischen Literaturauswertung über Motivation bei Softwareentwicklern ([14]) erklären.
Die Autoren fanden heraus, dass es Software Entwickler vor allem motiviert Software zu entwickeln (*„Yet 'the job itself' continues to be the principal motivator.“* [14, p.24]). Allerdings gibt es nur wenige Untersuchungen, was genau es ist, dass so motivierend am Entwickeln ist. Die gesammelten Studien konzentrieren sich sehr wenig auf die Spezifika der Softwareentwicklung.
Der wichtigste Faktor daneben ist das Bedürfnis sich mit seiner Aufgabe identifizieren zu können: (1) Klare Ziele zu haben, (2) ein persönliches Interesse am Produkt, (3) Verständnis des Ziels der Aufgabe, (4) wie es in das Gesamtprodukt einfließt und die

(5) Möglichkeit an einem identifizierbaren, zusammenhängenden Stück *„guter Arbeit“* (engl. *quality work*) zu arbeiten. Diese Möglichkeiten sind durch die Prozesse der Teams OM und PM gegeben. Insbesondere Punkt 5, ein zusammenhängendes Stück „gute Arbeit“, ist im Team AP nicht so deutlich gegeben, da Entwickler oft zu eigentlich abgeschlossenen Aufgaben zurückkehren müssen, z.B. weil die Testabteilung Fehler findet.

7.6 Bereitschaft

Die Entwickler bei SoundCloud sind außerhalb der Bürozeiten auf Bereitschaft, um bei Fehlern in der Anwendung zu reagieren. Damit sind nicht Probleme in der Infrastruktur gemeint, sondern Anwendungsfehler, wie zu viele geworfene Ausnahmen oder zu lange laufende Datenbankabfragen. Dieser Bereitschaftsdienst hat sich zum einen etabliert, weil das Infrastrukturteam bei vielen Fehlern nicht sinnvoll intervenieren kann, weil es die Anwendung nicht kennt. Zum anderen hofft SoundCloud, dass die Entwickler dadurch besonders motiviert sind, Fehler zu vermeiden.

Es ist sehr plausibel, dass das Wissen, dass man wegen Fehlern in der Software unter Umständen Nachts geweckt werden kann, Auswirkungen auf das Verhalten der Entwickler hat. Ich konnte allerdings nicht zeigen, wie sich das konkret auswirkt. Nach Meinung des Entwicklers selber, sind die Auswirkungen sehr positiv auf die Sorgfalt mit der er arbeitet:

„Und ich weiß auch, dass unter Umständen wenn es einen Fehler gibt, werde ich Nachts um 2 Uhr geweckt. Das ist auch schon genug Motivation zu sagen: Dann teste ich lieber doppelt-dreifach und stelle sicher, dass ich nicht Nachts geweckt werde.“

Der Bereitschaftsdienst schreckt ihn nicht ab. Im Gegenteil, wäre es einer Meinung nach schlechter, wenn die Software eine Nacht oder ein Wochenende lang nicht funktionieren würde.

8 Schluss und Ausblick

Ziel der vorliegenden Arbeit war es, die Qualitätssicherung in agilen Teams zu untersuchen. Dieses Ziel wurde gewählt, weil die Forschung, wie im entsprechenden Kapitel gezeigt wurde, zu diesem Thema eine Lücke aufweist. Um dieses Ziel zu erreichen, habe ich eine explorative Mehrfachfallstudie in mehreren agilen Teams durchgeführt. Diese Methode habe ich erfolgreich angewendet und als wesentliches Ergebnis, als Gegensatz zu traditioneller Qualitätssicherung, das Qualitätserlebnis herausgearbeitet.

Ein weiteres wichtiges Ergebnis sind die umfangreichen Daten zu den Teams. Sie stehen im Netzwerk der Freien Universität zur Verfügung und können von anderen Forschern für weitere Untersuchungen verwendet werden. Die weiteren Ergebnisse, wie die Probleme mit Integrationstests, die Verwendung nur sehr weniger Kommentare, die Auswirkungen des Bereitschaftsdienstes, die Rolle der separaten Tester und des Engineer in Test können Ausgangspunkt für weitere Untersuchungen sein.

Zwei der drei untersuchten Teams arbeiten ohne nachgelagerte Qualitätssicherung. Sie entwickeln schnell, aber sicher, weil sie ein starkes Qualitätserlebnis haben. Die Feedbackschleife für die Qualität ihrer eigenen Arbeit ist sehr kurz und sehr stark. Dadurch entdecken und korrigieren sie Abweichungen in der Qualität sehr schnell.

Für Forschung und Praxis sind die Gründe für dieses starke Qualitätserlebnis von hoher Relevanz. Um diese Gründe zu identifizieren, habe ich die Unterschiede und Gemeinsamkeiten der Fälle verglichen und folgende Säulen für die Stärke des Qualitätserlebnis herausgearbeitet: Die Fähigkeit, (1) hohe Qualität zu erzeugen, (2) die Qualität der eigenen Arbeit zu erkennen und (3) das Schicksal des Produkts zu bestimmen. Die Fähigkeit hohe Qualität zu erzeugen hängt von einer guten, deutlichen Produktivision und der Fähigkeit hohe interne Qualität zu erzeugen, ab. Die Fähigkeit, die Qualität der eigenen Arbeit zu erkennen hängt von der Zeit bis zur Veröffentlichung von Änderungen, automatisiertem Feedback, Überwachung der Software im Betrieb und klaren Grenzen in der Software ab. Die Möglichkeit, das Schicksal des Produkts zu bestimmen hängt von der Motivation der Entwickler, ihrer Möglichkeit die Entwicklung des Produkts mit zu bestimmen, ob von ihnen Rechenschaft gefordert wird, ob sie die Verantwortung für die Veröffentlichung tragen und wiederum von klaren Grenzen in der Software ab.
Interessanterweise ist der zugrundeliegende Faktor, der fast alle anderen stark beeinflusst, die Modularisierung auf Teamebene. Wenn ein Team ein Modul alleine kontrollieren kann, können die Faktoren für ein starkes Qualitätserlebnis besser umgesetzt werden.

Um das Konzept des Qualitätserlebnisses und das Modell der Einflussfaktoren zu validieren, habe ich es Entwicklern und Managern der Firmen SoundCloud und Immobilienscout24 vorgestellt. Von diesen wurden es und seine Implikationen weitgehend bestätigt. In beiden Firmen gibt es seit einiger Zeit Bestrebungen, die großen Applikationen durch Module auf Teamebene zu ersetzen. Die Gründe dafür lassen sich als

schwaches Qualitätserlebnis zusammenfassen.

Damit ergibt sich folgende Antwort auf die eingangs formulierte Forschungsfrage: Agile Qualitätssicherung funktioniert in den beobachteten Teams über eine kurze und starke Feedbackschleife, hier Qualitätserlebnis genannt. Ist diese Feedbackschleife gegeben, kann sie gut funktionieren, weil durch sie Abweichungen in der Qualität schnell entdeckt und korrigiert werden können.

Diese Ergebnisse unterliegen einigen Einschränkungen und sind nicht ohne Weiteres auf andere Teams übertragbar. Wünschenswert wäre, das Modell mit mehr Daten aus mehr Teams begründen zu können. Das Team der Legacyapplikation von SoundCloud wäre hierfür geeignet gewesen, aber leider waren die zeitlichen Grenzen der Masterarbeit erreicht. Die Validität wäre höher, wenn ein zweiter Forscher die gleichen Daten gesammelt und unabhängig analysiert hätte. Allerdings wurden die Ergebnisse von den beobachteten Personen selbst bestätigt. Sie decken sich also mit deren Wahrnehmung.

Die Ergebnisse können von agilen Teams als praktische Hilfe verwendet werden, um Probleme mit der Qualitätssicherung zu diagnostizieren und zu lösen. Die Einflussfaktoren für das Qualitätserlebnis können dabei als Grundlage verwendet werden. SoundCloud und Immobilienscout24 sind die Probleme mit großen Anwendungen bereits bekannt, und beide Firmen haben schon vor einiger Zeit die Modularisierung auf Teamebene eingeleitet. Die Ergebnisse dieser Studie können verwendet werden, um weitere Argumente zu liefern, warum noch stärker modularisiert werden sollte.

Im Vergleich zur Literatur fällt auf, dass zwei der drei Teams bewusst keine Tester einsetzen. Damit unterscheiden sie sich von den in der Literatur untersuchten Teams deutlich. In den meisten Studien wird davon ausgegangen, dass Tester zwangsläufig Teil eines Softwareprozesses sind.
Das Team, das von TALBY ET AL. ([1]) untersucht wurde, arbeitete mit einem separaten Tester, der von der Organisation zur Verfügung gestellt wurde. Allerdings ist nicht klar, welchen Beitrag dieser leistete, weil im Laufe der Untersuchung zeitweise nicht vorhanden war und nacheinander drei verschiedene Personen diese Rolle übernahmen. Es stellt sich die Frage, ob dort wie hier ein separater Tester überhaupt nötig ist.
KETTUNEN ET AL. ([2]) beschreiben, dass ohne separate Tester, Entwickler möglicherweise nicht ausreichend testen. Sie begründen das damit, dass für die Entwickler das Testen eine niedrigere Priorität hat. Diesen Effekt konnte ich nicht bestätigen. In den Teams ohne separate Tester sorgt das starke Qualitätserlebnis dafür, dass die Entwickler ihren Code immer testen. Die Autoren berichten, dass Teams ohne nachgelagerte Testphase deutlich flexibler arbeiten können. Diese Aussage kann ich bestätigen.
Die Ergebnisse von MÄNTILÄ ET AL. ([11]) und GUO ET AL. ([11, S. 166]) können ergänzt werden, durch die Entscheidung der Teams, bzw. Firmen, sich, aufgrund ihrer Erfahrungen mit separaten Testern, bewusst gegen separate Tester zu entscheiden. Der separate Tester im Team AP arbeitet dagegen besonders effektiv, weil er stark in das Team eingebettet ist.
Die Ergebnisse lassen sich in die Ergebnisse von AHONEN ET AL. ([9]) so einordnen,

dass auch hier die Organisationsform (ob Modularisierung auf Teamebene vorhanden ist, oder nicht) deutlich das Qualitätserlebnis und damit die Qualitätssicherung bestimmt.
MATIN ET AL. ([8]) schließen damit, dass in der Praxis weniger Bedarf für komplexe Testtechniken besteht. Stattdessen ist das wichtigere Problem, die Tests an die Anforderungen der Organisation anzupassen. Die Ergebnisse unterstützen diesen Schluss, weil keines der Teams explizit komplexen Testtechniken anwendet, um die Qualität zu erreichen. Stattdessen haben sie die Tests so in ihren Prozess eingebettet, dass sie ihnen ein starkes Qualitätserlebnis erlauben.

Wünschenswert wäre es, das Modell des Qualitätserlebnis durch weitere Untersuchungen zu validieren. Das ideale Vorgehen wäre, ein Team mit Qualitätsproblemen zu untersuchen und die Probleme mithilfe des Modells des Qualitätserlebnisses einzuordnen. Aufgrund des Modells könnten Vorhersagen getroffen werden, welche Auswirkungen Veränderungen, z.B. mehr Modularisierung, haben würden. Anschließend können die Auswirkungen dieser Änderungen beobachtet werden und mit den Vorhersagen verglichen werden. Auf diese Weise könnte das Modell auch auf andere Teams übertragen werden. Das Team AP, dass sich gerade aus einer großen Applikation heraus modularisiert, würde sich für diese Art der Untersuchung anbieten.

Literatur

[1] D. Talby, A. Keren, O. Hazzan, and Y. Dubinsky, "Agile software testing in a large-scale project," *IEEE Software*, vol. 23, no. 4, pp. 30–37, 2006.

[2] V. Kettunen, J. Kasurinen, O. Taipale, and K. Smolander, "A study on agility and testing processes in software organizations," in *Proceedings of the 19th international symposium on Software testing and analysis*, pp. 231–240, ACM, 2010.

[3] J. Itkonen, K. Rautiainen, and C. Lassenius, "Towards understanding quality assurance in agile software development," in *International Conference on Agility, ICAM 2005*, Citeseer, 2005.

[4] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. Wiley, 2011.

[5] M. Fowler and J. Highsmith, "The agile manifesto," *Software Development*, vol. 9, no. 8, pp. 28–35, 2001.

[6] Wikipedia, "Kanban." http://en.wikipedia.org/wiki/Kanban_10 2013.

[7] T. Linz, *Testen in Scrum-Projekten*. dpunkt, 2013.

[8] D. Martin, J. Rooksby, M. Rouncefield, and I. Sommerville, "'good' organisational reasons for 'bad' software testing: An ethnographic study of testing in a small software company," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 602–611, IEEE, 2007.

[9] J. J. Ahonen, T. Junttila, and M. Sakkinen, "Impacts of the organizational model on testing: Three industrial cases," *Empirical Software Engineering*, vol. 9, no. 4, pp. 275–296, 2004.

[10] P. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1, pp. 495–504, IEEE, 2010.

[11] M. Mäntylä, J. Itkonen, and J. Iivonen, "Who tested my software? testing as an organizationally cross-cutting activity," *Software Quality Journal*, vol. 20, no. 1, pp. 145–172, 2012.

[12] R. K. Yin, *Case study research: Design and methods*, vol. 5. SAGE Publications, Incorporated, 2008.

[13] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[14] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp, "Motivation in software engineering: A systematic literature review," *Information and Software Technology*, vol. 50, no. 9, pp. 860–878, 2008.

[15] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.