2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Quality Experience: A Grounded Theory of Successful Agile Projects Without Dedicated Testers

Lutz Prechelt
Freie Universität Berlin
14195 Berlin, Germany
prechelt@inf.fu-berlin.de

Holger Schmeisky
Freie Universität Berlin
14195 Berlin, Germany
holger.schmeisky@inf.fu-berlin.de

Franz Zieris
Freie Universität Berlin
14195 Berlin, Germany
zieris@inf.fu-berlin.de

ABSTRACT

Context: While successful conventional software development regularly employs separate testing staff, there are successful agile teams with as well as without separate testers. *Question:* How does successful agile development work without separate testers? What are advantages and disadvantages? *Method:* A case study, based on Grounded Theory evaluation of interviews and direct observation of three agile teams; one having separate testers, two without. All teams perform long-term development of parts of e-business web portals. *Results:* Teams without testers use a quality experience work mode centered around a tight field-use feedback loop, driven by a feeling of responsibility, supported by test automation, resulting in frequent deployments. *Conclusion:* In the given domain, hand-overs to separate testers appear to hamper the feedback loop more than they contribute to quality, so working without testers is preferred. However, Quality Experience is achievable only with modular architectures and in suitable domains.

CCS Concepts

•**General and reference** → **Empirical studies;** •**Software and its engineering** → **Agile software development;** *Software testing and debugging;*

Keywords

Agile Development, Software Quality Assurance, Industrial Case Study, Grounded Theory Methodology, Testing

1. INTRODUCTION

1.1 Conventional: Testing is done by testers

In conventional, document-driven software development processes, analytic quality assurance (QA) consumes a large fraction of the overall effort. Testing, its most prevalent activity, is taught in dozens of books and investigated in a large body of scientific literature [5, Chapter 4], [11, 13]. Software organizations devote whole departments to testing, consisting of people who work exclusively in a tester role, perhaps even discriminating different subroles [12].

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884789

1.2 Observation: Agile testing is not always done by testers

In contrast, in several companies we work with we noticed a curious diversity: Some of the teams had members or partner teams who exclusively were considered testers, whereas other teams had not even a separate tester *role*. All of these teams considered themselves to be agile teams (and we thought so, too). All of these teams appeared to be (and consider themselves) generally successful in their development efforts. All of these teams worked in the same domain: e-business web sites. This piqued our interest: Should not the difference between teams with and teams without separate testers be fundamental and visible in their success?

1.3 Agile: Should testing be done by testers?

Agile development methods, with their strong emphasis on personal communication, shun departments and favor multidisciplinary teams instead ("Build projects around motivated individuals." [1]). They do recognize testing as a *function*, but have no uniform attitude towards an explicit tester *role*, let alone to the need for separate testing *personnel*: Extreme Programming (XP) defines a *Tester* role [3, Chapter 10], although it is not necessarily filled by separate people [3, p.50]. In contrast, the Scrum Guide states that the Development Team (as opposed to the overall Scrum Team) must be cross-functional and self-organizing [21, p.5] and insists that all its members absolutely must have the same job title of *Developer* [21, p.6]. There must be no sub-teams "regardless of particular domains that need to be addressed like testing or business analysis" [21, p.6]. In Kanban, one or several forms of testing become natural stages in the pipeline [4, Figure 8]. Kanban thinking starts from whatever organization and roles are currently used and is therefore agnostic on the separate-testers issue [2, Chapter 5]. Summing up, there is no unanimous preference for or against separate testers in these sources.

1.4 Research questions and contribution

We ask two research questions:

RQ1: How is quality assured in agile teams that do not

employ separate testers?
RQ2: What are advantages and disadvantages of not employing separate testers?

Our main research contribution is **the notion of Quality Experience**: A concept that explains how and when teams are able to assure quality well without separate testers. This is the first time we know of that the circumstances, nature, and effects of agile quality assurance without separate testers have been contrasted to quality assurance with separate testers concretely and with empirical evidence.

Our results suggest that a Quality Experience mode of quality assurance has strong positive effects on business flexibility and developer motivation. The mode can only be reached under certain circumstances. If it *can* be reached, hand-overs to separate testers would get in the way. If it *cannot* be reached, working with separate testers is presumably preferable for the well-known conventional reasons.

1.5 Structure of the article

Our study is a case study of three agile software development teams with and without separate testers at two different companies. We will now describe our research methodology (Sections 2.1, 2.3, 2.4), notation (Section 2.5), the software domain (Section 2.2), and the teams (Sections 2.6 to 2.8), then report our various findings (Sections 3.1 to 3.8). We discuss limitations (Section 4) and related work (Section 5) before we present our conclusions (Section 6).

2. CONTEXT AND METHOD

2.1 Method type

Our study applies case study methodology [27]. In the terminology of Yin, ours is an exploratory, holistic, 3-case study. Each case is one agile software team, selected by convenience sampling. Two of the teams (OnM, OffProf) are from the same company[1], the third (Pay) is from another. Triangulation occurred mostly across the several team members of each case. We largely follow the case study reporting guidelines of Runeson and Höst [19], but as our article focuses on reporting a Grounded Theory, not a case study, we apply them only loosely. We assume analytic generalization to comparable cases, but of course no statistical generalization [27].

2.2 Context: Domain

All three teams work in the same domain: The in-house development of a single, large, mass-user web portal at a company that makes money from direct payments of some users of that portal. This domain has a number of properties that are relevant for the results of our study and that all three teams have in common, for instance:
(1) There are millions of users.
(2) There are several large subgroups of different user types; in particular paying users (customers) and non-paying users.
(3) Any individual customer accounts for only a small part of the revenue stream and that part is usually further divided into small pieces so that failures of individual transactions are not overly expensive.
(4) The portal and its development are complex in multiple respects such as functionality, scalability, access channels, payment channels, etc.
(5) Defects in individual services within the portal are critical for the business but not quickly catastrophic.
(6) New versions of parts of the software can in principle be deployed immediately. The rollouts can be complete (to all users) or partial (e.g. as A/B testing).

[1]The same-company similarity between OnM and OffProf is hardly relevant for our purpose here.

2.3 Method: Data collection

The data collection for our study was performed by Holger Schmeisky, initially by passive observation, informal conversation, and obtaining documents (round 1), then by formal, semi-structured interviews (round 2), and finally by follow-up steps using emails for clarification and a few additional interviews (round 3).

In **round 1**, for each team, he shadowed one developer for several days, performing direct observation and listening to spontaneous explanations, taking notes along the way, to obtain an overview of the domain, developer activites, and team workflows. He also obtained a number of documents from the development process that were used in intermediate steps of the analysis, but not in this article.

The **round-2 and round-3** interviews were audio-recorded and transcribed. They were semi-structured, but did not all follow the same structure. Rather, he modified the questions as he went along and obtained more insight. The first interviews were broad, asking about the company (structure, business, culture), software (architecture, technology, tools), team organization (size, projects, roles), manner of use of agile methods, nature of tasks and task assignment, work planning, publication/release rules (including emergency rules), defect repair process and culture, deviations between intended process and actual process.
Later interviews reduced company-level questions and added delivery-related ones e.g. regarding what the team considered to be the primary quality attribute(s), the use of automated testing, modes of defect detection, the sense of responsibility exhibited by the developers, and developer responsibilities beyond writing software (such as service monitoring).

For each team, Holger interviewed one or more representatives of each relevant role, in particular (where applicable) developer (marked **D** in our data references), product owner (**PO**), Scrum master/coach (**SM**), tester (**QA**), team lead[2] (**L**), and higher-level manager (**M**). After round 2 and initial data analysis, we presented our findings to all three teams for validation.

In **round 3**, gaps and clarifications that came up during the subsequent finer analysis were handled by email or follow-up interviews with the same interviewee; those often covered rather specific issues. After round 3 and the complete data analysis, one member of team Pay validated our complete writeup. We have since presented the results twice (to about 50 practitioners from other companies, mostly from the web portal domain); the only objections we received (in the form of "yes, but…" questions) were from people working in different domains with additional constraints.

2.4 Method: Data analysis

Our data analysis used GTM, Grounded Theory Methodology [23], as follows: Theoretical Sensitivity [23, Section

[2]primarily a communication and coordination role

II.6] was initially sharpened by data collection round 1. Based on the data from round 2, Holger Schmeisky started with Open Coding [23, II.5]. Lutz Prechelt later joined him in this work in Pair Coding manner [20].
The Constant Comparison [23, II.1] turned out to be relatively easy, as the various concepts had not much tendency to overlap. The nodes of Figure 1 represent the respective resulting concepts relevant for the outcomes of our study.

The edges of Figure 1 represent the relevant results of our Axial Coding [23, II.7], also partially done in pair coding mode. Source nodes represent strategies or context conditions, target nodes represent consequences. Franz Zieris checked the consistency of the Open and Axial Codes.

Selective Coding [23, II.8] chose Quality Experience as the core category and the boldfaced nodes in Figure 1 represent its immediate elements. Data collection round 3 served to fulfill the requirements of Theoretical Sampling [23, II.11], although we may not have achieved Theoretical Saturation [23, II.11].

We stopped collecting data when we had understood all important relationships between the factors we had uncovered as shown in Figure 1. Further data collection might have extended our theory (by uncovering further factors), but all elements of the current one would remain valid.

2.5 Notation

When we refer to data from round 1, we will mark its type as **O** for passive observation data or **N** for notes from informal conversation. A complete data reference will mention team name, person role and number (if applicable), data file, and, beginning line number (if the file is long), for instance (TeamX.D2/O3-147).

For rounds 2 and 3, interviews are marked as **I**, and the data references mention time (in minutes:seconds) rather than line number. For instance (TeamX.PO/I1-13:22) refers to a statement made after 13 minutes and 22 seconds in interview 1 at team TeamX, which was an interview with the Product Owner. Most interviews were in German and the quotations are hence translated and may use diction uncommon in English. Emails will be marked as **E** and again use line numbers, as in (TeamX.QA2/E2-5). The companies did not give permission to publish the raw data. To scrutinize it, you will have to visit our research group.

2.6 Context: Team Pay (SoundCloud)

SoundCloud is a music sharing service: artists can present themselves and upload their own music (3 hours for free, more against payment); other users can browse this music, listen to it, comment on it, and share it on social networks. SoundCloud has about 10 million users and is usually among the world's top 200 web sites[3]. It supports web browsers, iOS and Android apps, and community-built apps based on the SoundCloud API.

The SoundCloud architecture started as a single Rails application (now called *mothership*) but since 2011 has gradually been split into separate services. New services are realized in a variety of technologies. Any SoundCloud development team consists of developers only (there are no architects, testers, etc.) and most are vertical, i.e., in charge of one functional area completely. SoundCloud has around 80 developers overall.

[3] http://www.alexa.com/siteinfo/soundcloud.com

Team Pay is responsible for the *Buckster* service that contains all payments-related functionality such as user subscriptions, fraud detection, and reporting. A few parts of Buckster were still in the mothership during our study. Team Pay consisted of two developers (previously three) and one product owner[4] and used a lightweight Kanban process, without fixed iterations. The product owner assesses the quality as follows: *"As we have relatively few severe bugs, I believe that we are doing a good job."* (Pay.PO/I5-15:10).

2.7 Context: Team OnM (IS24)

ImmobilienScout24 (or IS24 for short) is by far the largest real-estate web portal in the German-language-area, with about 1.5 million offerings and about 10 million monthly visitors. Its core are real estate offerings (sale or rent; houses, apartments, commercial property) but the portal also brokers financing, insurance, and many other services.

IS24 has about 180 software developers organized in about two dozen more-or-less cross-functional teams. Each team uses some (typically Scrum-ish) flavor of agile process. The portal software was originally a large, monolithic Java EE application, built by a department-structured organization. Much of the software has since been split into separately deployed services, the remainder is called the *core application*.

Team OnM (full name "Online Marketing") develops a range of services with little end-user visibility such as search-engine optimization, marketplace integration with partner portals, landing pages for AdWord campaigns, visitor and campaign analytics and reporting, and data export APIs.

The team with formerly seven developers had recently been split into a sub-team oriented towards routine tasks and another for new tasks; we talk only about the latter here. It consists of four developers, a technically knowledgeable product owner, and a technical lead and uses a Kanban process without fixed iterations. When we asked the product owner how happy they were with the quality of their work, the answer was *"Extremely happy."* (OnM.PO/I5-12:52). He is particularly happy with their flexibility: *"We are more effective and more flexible [than we used to be].*[5] *[We can handle very small] cycle times for requirements."* (OnM.PO/I5-21:25).

2.8 Context: Team OffProf (IS24)

OffProf (full name "Offerings Professional") is another team within IS24, but with rather different character. OffProf develops some functional areas of the ScoutManager, a closed-group part of the portal used for creating advertisements by private individuals and real-estate agents. The ScoutManager is part of the core application, which is still very large, with over 10,000 source files that form a single Spring MVC application and several teams working on it.

The core application has a weekly release cycle as follows: All core-related teams provide a development snapshot every Tuesday at 10am, which is then tested by the separate QA department, handed over to the operations department, and deployed on Wednesday the next week (day 8 after submission); defect fixes can be inserted into an ongoing QA week, but still normally need to wait for a deployment Wednesday.

[4] This team is smaller than the other two, but we encountered no team-size-related effects during our analysis.

[5] Square brackets indicate parts added or modified by the authors to enhance clarity of quotes that otherwise lack context.

Table 1: Overview of the investigated teams

Label	Full name	Company	Team members	Tasks	Deployment
Pay	Payment	SoundCloud	2 Developers, 1 Product Owner	Back-end service for all payment-related functionality	Continuous
OnM	Online Marketing	ImmobilienScout24	4 Developers, 1 Product Owner, 1 Technical lead	Several services concerning Online Marketing and Analysis	Continuous
OffProf	Offerings Professional	ImmobilienScout24	6 Developers, 1 Tester, 1 Product Owner, 1 Scrum Master, 1 Team lead	Front-end functionality and services for creating real-estate advertisements in closed-group portal	Weekly, with an 8-day delay; separate testers and operators

Team OffProf's tasks are mostly frontend-related. OffProf consists of six developers, one tester, a non-technical product owner, a Scrum master, and a team lead. The tester and product owner are the only female participants in our study. When we asked the product owner how good she considered the quality of what they produce, the answer was *"Given that we're in the legacy app: Really not bad."* (OffProf.PO/I1-20:33), a fully positive (in German manner of speaking), if not enthusiastic answer.
Refer to Table 1 for a quick reminder of these team descriptions.

3. RESULTS

3.1 Quality Experience

The notion of Quality Experience is at the center of our results, so we start our discussion from it. The term was first used by OffProf's Scrum master at one point: *"I believe when you first shove it into some pipeline and know it will come back at some point if there are issues, then you have a completely different quality experience."* (OffProf.SM/I4-29:54). In this statement, which compares OffProf's situation to that of a continuous delivery team, "quality experience" is a fuzzy notion and the Scrum master did not elaborate. Yet during our analysis, we found that this was a very appropriate term. We eventually refined the notion into the following form, in harmony with the teams' views as well as the two respective process realities:

Quality Experience is a mode of quality assurance and deployment in which the team
(1) *feels fully responsible* for the quality of their software;
(2) receives *feedback* about this quality, in particular the quality of their recent changes, that is (2a) *quick* (available early), (2b) *direct* (not be intermediated), and (2c) *realistic* (coming from non-artificial settings); and
(3) *rapidly repairs deficiencies* when they occur.
These five *elements* of Quality Experience will be core concepts in our discussion and the presentation of evidence.

Quality in this context is not just the absence of defects. For all three teams, quality is a holistic attribute that covers most aspects of business value, from functional defects and gaps over all kinds of attractiveness issues to operation problems regarding deployability, scalability, monitorability, and so on.

Of our three teams, Pay has clearly the strongest Quality Experience, OnM a little less strong, and OffProf has a comparatively weak one. (Mnemonic: The team with the longest name has the weakest Quality Experience.) Our discussion will be arranged around the above-mentioned elements of Quality Experience plus a number of influencing factors and mostly talks about *strengthening influences*: How does one factor increase the intensity or quality of another?

Unfortunately, most factors are related to most other factors *somehow*. We will constrain our discussion to those of the factors and relationships that contribute the most to understanding Quality Experience[6]. The remainder is shown in Figure 1.

In order to avoid forward references, the discussion will roughly work its way from the top of the figure (Modular Architecture) to the bottom (Frequent Deployments). We group nodes and edges into subsections such as to simplify understanding and for clarity often mention the individual nodes (e.g. Modular Architecture) or edges (e.g. Modular Architecture ⟶ Empowered to Deploy) being discussed.

We will rely on data from teams Pay and OnM to describe how a development situation with strong Quality Experience comes to be and what it looks like; and on data from team OffProf to describe what keeps strong Quality Experience from arising and what such a situation looks like. The figure pertains to the case of strong Quality Experience only and this is also the main perspective of the text; the alternative case at OffProf is described only for contrast.

3.2 Architectural precondition

Modular Architecture (see Figure 1)[7]
The fundamental precondition for an arrangement that enables a strong Quality Experience appears to be a software architecture that sufficiently decouples the work of one team from that of another (here: separate web services communicating via HTTP). Without such Modular Architecture, much additional beyond-team coordination effort is required (OffProf.SM/I4-42:17), it is not always clear which team owns a particular piece of code (OffProf.SM/I4-33:25), and defect introduction frequency is higher.

Of our teams, OnM and Pay have high general quality, with zero known open defects (OnM.PO/I5-12:52),

[6]We do not expect them to be an appropriate set of factors for other purposes. Also, expect many identifiable *intermediate* factors and additional relationships to be missing in our discussion.
[7]This section talks about the Modular Architecture node. Each section will list the relevant nodes at the beginning. The node names should be considered identifiers, not explanations.

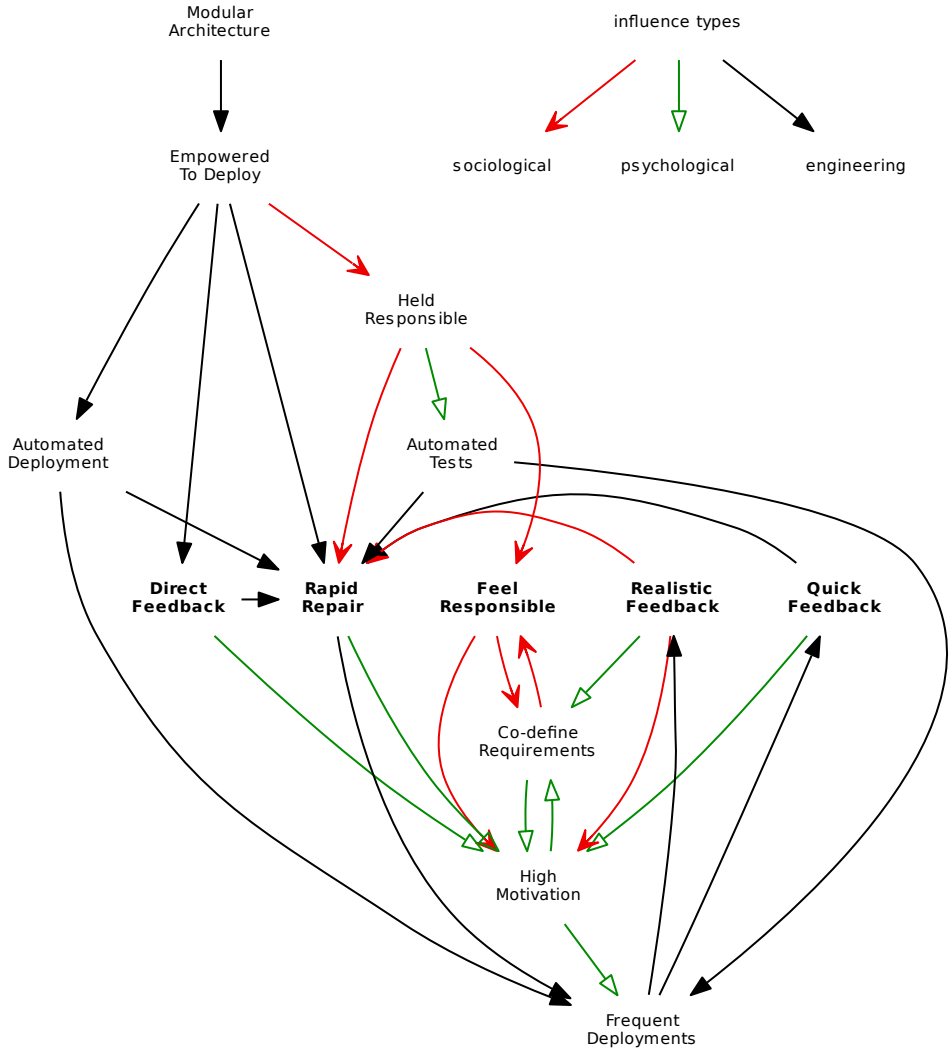

Figure 1: The major strengthening influences between some major factors in the causes-and-effects network around Quality Experience. The boldfaced nodes in the center are the elements of Quality Experience. The labeling of edges as sociological, psychological, or engineering influences exists to simplify interpretation only; we make no claims in this regard. (The diagram is not a process diagram and the edges indicate neither data flow nor control flow relationships.)

(Pay.PO/I5-15:10). OffProf is suffering from legacy code: most of their code is in good shape, but some parts are rather old and exhibit design decay. They are reasonably covered by unit tests, but it is difficult to make changes to the parts because they are hard to understand (OffProf.D/N11-34).

The lack of a sufficiently modular architecture is why team OffProf does not get to high Quality Experience: The dependencies across all teams working on the large core application are so strong that a huge test suite is required for the continuous integration[8], a within-team tester is deemed necessary who performs thorough functional and exploratory testing of each substantial change (OffProf.QA1/I2-5:10), and the company does not dare deploying without the centralized QA step: *"It is often very hard for the [core application] teams to work cleanly enough to prevent fixes during the QA phase. This is often not the team's fault, but the system's. [...] There are too many dependencies, [...]."* (OffProf.QA2/I5-19:20).

Obviously, teams Pay and OnM are in a more comfortable situation. But what makes them abandon the idea of separate testers?

3.3 Conscious empowerment decision

Empowered To Deploy

In our web portal domain, if the team is as competent as ours are and the Modular Architecture precondition is fulfilled, it appears that a company can essentially *decide* to make the team obtain a strong Quality Experience by handing over complete control over deployment (and monitoring) of their part of the portal to the development team.

In our Pay and OnM cases, this hand-over was indeed a conscious decision. The higher-level manager overseeing Pay

[8]it takes 20 to 40 minutes to report green (OffProf.D/E3-34)

puts it like this: *"[We want] the team to own, end-to-end, the stuff they produce so there are no external dependencies, no external QA teams or anything that says you can release or you cannot."* (Pay.M/I6-14:55).[9] And the team lead of OnM has clear expectations of his developers: *"[The notion of quality concerns] the* whole *thing being implemented. End-to-end, [from planning and development] to operation, including monitoring and so on."* (OnM.L/I4-31:03).

Modular Architecture ⟶ Empowered To Deploy

This development style is not possible for the legacy application at either company: Both legacy applications, SoundCloud's mothership and IS24's core application, require a large amount of experience to handle deployment properly, making it a task for specialized experts only: *"But there again the system throws a monkey wrench in the works, because it is not just push-a-button-and-be-live; rather, there are a lot of manual steps in the production department."* (OffProf.QA2/I5-53:35), (Pay.D2/E10-22).

3.4 The role of responsibility

Held Responsible

Feel Responsible

From the empowerment it is only a small step to a much-widened sense of responsibility. For instance, most software developers might subscribe to the following statement: *"Exactly. That is the basic idea [of the company culture]: every developer is responsible for the code he develops."* (Pay.D2/I1-47:01). But in our case, the developer making the statement had in mind a wide meaning of "responsible"; developers are not merely *held* responsible by the company, they also inwardly *feel* fully responsible: *"The most important thing is [...]: Every developer is responsible for what he is doing. And at no time he thinks he can just write the code and rely on somebody else to fix it."* (Pay.D2/I1-57:35). Another said *"[We deploy,] therefore we are responsible for our software to be fit."* (OnM.D2/I1-43:45), and the product owner summarizes *"In the end, we as a team are responsible for what we release."* (OnM.PO/I5-4:52).

Team OnM had had a setting with a separate QA person previously. D2 characterized it like this: *"By this, we handed over the responsibility. When we had that mode active, we said [the tester] has to find this [...]. He needs to have something, too, needs to be kept busy. We handed off responsibility."* (OnM.D2/I7-22:00). But once they switched, they felt differently: *"Working [without a separate Tester] changed things. You have nobody to blame, because* we *did it. That way, you take personal any error happening in the field. That way, you bear more responsibility. Before, you could duck away."* (OnM.D2/I8-57:45).

Held Responsible ⟶ Feel Responsible

In most of the statements we heard, these two aspects are almost inseparable: As a result of the empowerment to deploy, the company of course *assigns* the developers a more comprehensive quality responsibility and the developers also *perceive* their responsibility to have widened: *"[The team] are responsible for ensuring [everything works] and have the means to do so."* (Pay.D1/I6-15:08).

The sense of responsibility has remarkable consequences. For instance the Pay team self-organized into a round-robin on-call duty for 24h monitoring of their payment service (Pay.D2/I1-29:40), intervening in case of severe problems even in the middle of the night (Pay.D2/I4-68:35). And the team does not consider this extraordinary at all: *"In most cases it is not about getting woken up, but the implication that people are not able to upgrade [their SoundCloud subscription]. That is what we want to prevent. [...] I demand of myself that the system is running."* (Pay.D2/I4-69:06).

Held Responsible ⟶ Automated Tests

This also leads to extra effort and care for ensuring the effectiveness of the pre-deployment quality assurance, in particular the automated tests: *"I know that when there is an error I might be woken up at 2 in the morning. That is enough motivation to say: I will [write] twice and thrice [as many tests] and make sure I will not be woken up."* (Pay.D2/I1-84:58).

Co-define Requirements

The developers substantially participated in the requirements definition, apparently based on (1) their increased understanding of the domain as produced by more realistic feedback from the field (as discussed below), (2) their generally high motivation (also discussed below), and (3) their feeling of responsibility. Their participation reinforced the motivation and the feeling of responsibility. Due to our research question, we did not specifically investigate requirements definition processes, so the evidence for these relationships is spread out too far over our data to summarize it neatly here with quotes, but for example the OnM product owner told us he and the team now *"partially work on the requirements together"* (OnM.PO/I5-9:09). In team Pay, not only are developers *"quite heavily involved in the planning of the product"* (Pay.PO/I5-11:44): If the PO asks for changes that the developers do not immediately agree with, he has to justify the decision: *"They always challenge me: Do you really need that? Did you look at the numbers?"* (Pay.PO/I5-10:09). Both teams consider this helpful (Pay.PO/I6-35:45), (Pay.D2/I1-43:10), (OnM.PO/I5-12:35), (OnM.L/I4-31:03).

3.5 The role of feedback

All agile methods emphasize the motivational value and practical importance of feedback for iterative software development. In our setting, the feedback from monitoring field use of the freshly deployed modified software plays the same roles for creating Quality Experience. It primarily consists of observing web analytics information such as type and number of HTTP responses, number of sales in total, incoming traffic through web banners, etc. During our analysis, we found three properties of field-usage feedback that are relevant and beneficial: The feedback should be available early (*quick*), not be intermediated (*direct*), and preferably come from non-artificial settings (*realistic*).

Quick Feedback

The most obvious of these three properties is avoiding delay: The earlier a developer can receive feedback on code written today, the smaller he or she can make the iterations and the faster the learning (and hence the improvement of the software) can progress. It is an implicit goal in the business value thinking of the Pay product owner: *"So we sat down last week and analyzed what we need to start and what we can do later and prioritize that. We ended up with very little, so that we can already launch next week."* (Pay.PO/I5-9:24). His higher-level manager even considers it

[9] This was a group interview with Pay's D1, D2, PO, and M.

the top priority: *"Fast feedback is the most important thing."* (Pay.M/I6-10:52).

Conversely, the *lack* of Quick Feedback (and Rapid Repair, see below) is a top source of frustration for the OffProf team: *"It is so frustrating, we would be able to do so many cool things if we had newer, modularized code. We cannot be courageous. We have to consider ten times: 'Should I put this live?', because everything has to fit. Because we always have to wait at least a week to react to an error."* (OffProf.PO/I1-23:17).

Direct Feedback

The second property is related to the interpretability, actionability, and motivational value of the feedback: If developers make observations themselves, they understand their meaning well, can act right away, and can easily develop a sense of urgency. Team OnM has direct access to feedback and states: *"It's no use if somebody only tells us where something is wrong. We want to see it ourselves."* (OnM.D2/N5-52).

If, in contrast, they receive the observation indirectly from third parties (such as testers or an operations department), many misunderstandings become possible, clarifications will often be needed, processes become more complicated, and the sense of urgency will be diminished. OffProf has this problem with defect reports from the QA team: *"[When a bug comes in] there is the question, whether it is regular or critical, I have to talk to the PO, they will enter it [in our JIRA], we have to agree in the daily [stand-up meeting] who takes care of this bug. Then we have to talk to the QA, how she found it, because we did not find it, [our internal QA] did not find it, but they found it."* (OffProf.D/I3-92:14).

Empowered To Deploy ⟶ Direct Feedback

In principle, any web portal team could be provided with (and benefit from) direct feedback from live use. In practice, however, if the feedback comes with enough delay, the perceived value of this feedback is much lower and this appears to be the reason why Pay and OnM make much more use of direct feedback. OffProf also has (limited) access to direct feedback, but it hardly motivates the team (OffProf.D/I3-22:22) because it originates from changes they made more than 8 days ago and it mixes effects of changes in the whole legacy application, not just OffProf's component. Pay and OnM, in contrast, have invested in feedback automation and had a large monitor in their team space that showed real-time data from service operation (Pay/O4-27), (OnM.D2/I7-5:12). Team Pay has even built a monitoring service that will alert them by SMS during nighttimes and weekends in case of various types of serious problems, not just server outages (Pay.D2/I1-29:05).

Realistic Feedback

The third property reflects that internal testing cannot compete with field use in terms of uncovering imperfections: *"When [changes] are finished: Get 'em out! Because only in the wild we can see the errors that may occur."* (OnM.D2/I1-69:28). This quote also nicely shows the desire for quick feedback and the sense of urgency provided by direct feedback.

3.6 Rapid repairs of field defects

A second issue for the Empowerment To Deploy, besides the architectural precondition, is staff capacity, because without help from separate testers and an operations department, the available manpower is smaller. This is solved by automation.

Automated Deployment

The smaller part of the automation contribution concerns the deployment. As a response to their wish[10] to deploy by themselves, teams OnM and Pay both fully automated the deployment process. OnM developed a graphical application and the product owner now deploys with a mouse click (OnM.D2/N11). For Pay, a developer deploys via `rake deploy production` from the command line (Pay.D2/I4-7:40). This reduces the wall-clock time required for a deployment after testing, roughly one day at team OffProf (OffProf.D/I3-103:49), down to a minute.

Automated Tests

The larger contribution comes from almost fully automated testing. Deploying frequently clearly brings forward the need for automated tests that check nearly all of what is necessary to assure correctness. We have already seen the serious attitude of the Pay team in this regard (Pay.D2/I1-84:58). Their product owner, who comes from a background of extensive manual testing, considers automated testing a key ingredient for being able to build the service without separate testers: *"Payment is a sensitive topic and damage can be high [...]. We initially worried whether we can handle it with just a small team. In the end we covered it well with these automated tests and it works fine."* (Pay.PO/I5-23:55). Pay initially spent a lot of time on creating a suitably balanced testing pyramid, adding new types of tests and removing those that proved ineffective (Pay.D2/I1-76:53). Team OnM similarly invested a lot of time into their test code, mainly to make it run faster and easier to maintain (OnM.L/I3-21:20).

Note that OffProf invested a lot of effort into automated testing as well; for example 30 complex web tests are executed with every continuous integration run, and 350 additional ones are run every 2 hours (OffProf.D/N2-30). But the team has a hard time making these tests adequate: *"[There] is often discussion between [the QA] and the others about what makes a good test. [But] as long as they cannot experience what it is like to go live immediately, [this discussion is theoretical]."* (OffProf.SM/I4-32:25).

Rapid Repair

Automated testing and deployment, together with the availability of quick, direct, realistic feedback lead to the most strongly dependent node in our diagram, Rapid Repair, by which we mean both a capability and a practice. It means that whenever a defect slips through to deployment, it will typically be online only for at most a few hours before it is corrected: *"The number of fixes taking more than one day is vanishingly low. Normally directly fixed within one day and live, too."* (OnM.L/I4-23:25). The directness of the feedback is an important contributor to this speed (OnM.D2/I7-8:30). At Pay, the repair speed is fast enough to give their PO a sense of reassurance even on catastrophic failures: *"In our last issue, nothing [of one specific functionality] worked for 2 hours. But we saw that very quickly and could fix it really fast. [...] I tend to be nervous about [an outage like this],*

[10] We have not investigated much the history of how the Quality Experience situation was reached or created, but our impression is that both organizations moved harmoniously, without forcing anything on anyone.

[. . .], but so far we were always very fast fixing them and I get a lot more relaxed lately." (Pay.PO/I5-16:35).

The capability of rapid repairs also contributes to high motivation (our next topic), as we can see when we compare the above statement to the depressed tone at OffProf when they will comment on *their* repair interval in the coming section: *"And then it takes another 2 weeks. . . "* (OffProf.D/I3-106:00).

3.7 Motivation effects

High Motivation

Both the feeling of responsibility and the availability of good feedback contributed to the high level of motivation we found in the Pay and OnM teams. This motivation is visible in many of the statements from those teams including the *"That is enough motivation to say: I will [write] twice and thrice [as many tests]"* (Pay.D2/I1-84:58) and *"They always challenge me"* (Pay.PO/I5-10:09) from Section 3.4, or *"Get 'em out"* (OnM.D2/I1-69:28) from Section 3.5.

It is most impressively seen when contrasting it with the frustrated and leaden feeling conveyed by the OffProf developers in face of their two-week waiting time for a redeployment in case of a mistake: *"Especially bugs [in the live system] are very annoying. [. . .] Sometimes I think 'What will our customers be thinking?'. But this is probably a developer thing, to be so angry with oneself for overlooking something. [. . .] And then it takes another 2 weeks. . . "* (OffProf.D/I3-106:00).

Contrast this with the lighthearted feeling exhibited by OnM.D2's summary of what successful iterative development is like: *"We saw that we could achieve more. That we were iterating faster and got our product out faster, creating some leeway. We attempt to create free space for ourselves, so we can develop new ideas and test them and less time is occupied by routine stuff."* (OnM.D2/I1-59:19). The notion of "routine stuff" refers in particular to lack of automation.

3.8 The consequence: frequent deployment

Frequent Deployments

Driven by the teams' high motivation and the testing and deployment automation, teams Pay and OnM have moved to a finer-grained iteration style (OnM.D2/N12) and now deploy new versions of their service(s) several times a week (Pay.D2/N15-22), (OnM.D2/E12). This has three important consequences.

(1) It makes the risk from not having separate testers bearable, because failures are short-lived: *"The incremental rollout is very important. Not just from the planning perspective, but also from the code perspective, that you can just roll back very quickly."* (Pay.D2/I1-40:00), also reported by OnM (OnM.D1/N7-47). Also, debugging is simple when changes are small (OnM.D2/I1-70:12), (Pay.D2/N15-23).

(2) It provides feedback more quickly and therefore makes realistic feedback more easily available: *"We always get everything out immediately, to get feedback."* (OnM.D2/I1-70:08).

(3) It allows better strategies for achieving larger goals. For instance it allowed team Pay to perform a risky migration piecemeal, thus lowering risk immensely, without becoming too slow (Pay.D2/I1-36:50).

Contrast this with how the OffProf team would like to work, but cannot: *"Now we have 1 week releases, it used to be 2 weeks. This is very frustrating when you know the other world, where things go faster; especially when A/B-testing. [With several iterations,] that quickly adds up to a month or two. The result is neither fast nor agile in our view."* (OffProf.PO/I1-21:20). To cope with the slowness, OffProf started to run multiple A/B tests concurrently, with multiple complex (yet provisional) changes to the database schema, and struggled with the resulting complexity (OffProf.SM/I4-23:35).

This work style is *"[. . .] neither fast nor agile"*, indeed. Agile development should allow making and validating many quick small steps in order to keep investments into wrong ideas small and keep progress on good ideas fast and on track. And by enabling Frequent Deployments, this is exactly what the Quality Experience mode of quality assurance provides.

4. LIMITATIONS

We see three main limitations of our case study: First, many of the constraints and behaviors uncovered in our study are tightly bound to the domain as characterized in Section 2.2, and we make no claim with respect to the generalizability of our results outside of this domain. Second, even within the domain, the set of teams is too small to claim generalizability on a quantitative level and so we make no quantitative claims (statistical generalization) either. Third, our characterization of the conditions under which the Quality Experience mode of quality assurance can be reached is only coarse, in particular with respect to issues of company culture, which was so unproblematic at SoundCloud and IS24 that the topic never arose.

Other than that, however, we expect no problems with the validity or generalizability of our results: We are convinced that our interviewees were open and honest, and that our interpretation of their statements is justified in both a historical sense (for the past, in the given contexts[11]) and an engineering sense (for the future, in new contexts).

We have investigated the Co-define Requirements topic only superficially. It deserves a study of its own.

5. RELATED WORK

5.1 Agile testing with or without separate testers

Much research into quality assurance in agile development methods appears to take it for granted that agile teams work with separate testers; for instance [6, 14, 24, 26]. Regarding our research question whether agile testing should better be done with or without separate testers, we found only two studies that contribute substantially.

Guo et al. [8] analyzed the Windows Vista defect database (not a fully agile context) and found that defects become less likely to be fixed as the distance (whether geographical or organizational) grows between assigner and assignee. The highest fix ratios are found for bugs developers assign to themselves, which is just what will happen most of the time if there are no separate testers.

Mäntilä et al. [16] analyzed, in three business and engineering software producing companies, which employee groups find bugs and how likely these are fixed. They found

[11] As mentioned in Section 2.3, this was confirmed by the teams after round 2 of our data collection. It was also confirmed for this whole article by Pay.D2.

that in addition to the specialized testers, many groups find defects, e.g. sales, technical support, managers, and external end-users. The defects found by specialized testers tended to be "unlikely to occur in real use" or had "only minor effects" [16, p.165]. We do not think this strongly applies to OffProf, but nevertheless such a tendency suggests that Realistic Feedback may be an efficient approach if the associated risk is acceptable.

Olsson et al. [17] conducted a case study of 5 software companies with different innovation cycles. They propose a "stairway to heaven" model, that starts with traditional long-cycle development (level "A") and culminates in a continuous, fast-paced innovation experiment for product development (level "E"). In this model, our teams Pay and OnM are roughly on level "D", where code changes are released in a continuous delivery fashion. The study identifies several key factors for reaching higher levels, that appear in our model as well [17, Table 1]: Modular architecture; Comprehensive test automation; Deployment automation; Early, realistic feedback from end-users. However, the focus is more on organizational benefits and barriers for product development, not on quality assurance and testers. The article (vaguely) mentions that even on level "D" there might be cultural barriers against releasing inexhaustively-tested software to customers which would mean Quality Experience is incompletely established, but the domain in this case is different (SaaS products in finance/accounting).

Google and Facebook, two software companies that are arguably successful in their software development, also found that going without separate testers improved their software development. Google, when they still had separate testers, recognized that testing turned into a bottleneck and time-to-deployment became too long [25]. They replaced specialized manual testers with software engineers that would *support* the actual developers of the respective functionality when writing automated tests. *Devising* the tests, however, is the developers' responsibility (Held Responsible), not the testers'. As a result, the developers took more responsibility for quality overall (Feel Responsible).

At Facebook [7], whose business more closely resembles that of our companies, we find many of the phenomena encountered in our study: Developers, rather than separate testers, are put into the QA role, so that they can be Held Responsible more completely and will therefore also Feel Responsible. Facebook perceives this to be successful: "Personal responsibility by the engineers who wrote the code can replace quality assurances obtained by a separate testing organization." [7, p.12]. Facebook strives for Frequent Deployments to reduce risk. Deployment is basically weekly, not as frequent as with Pay and OnM, and, due to Facebook's size, without making the developers Empowered to Deploy; instead, there is internal beta testing by employees. Developers must be available for corrections when their new code is rolled out. Functionality is often rolled out to only small fractions of all users first to obtain Quick Feedback and Realistic Feedback: "Testing on real users at scale is possible, and provides the most precise and immediate feedback" [7, p.12].

Results from psychology also suggest that working without separate testers might enable software developers to work better. Shanteau [22] states that experts in a domain are able to build up expertise and make good decisions when: (1) Feedback is available (corresponding to Direct Feedback in our study); (2) Tasks are repetitive, providing frequent opportunity for learning (Frequent Deployments); (3) Some errors are expected and acceptable (Rapid Repair); (4) A problem is decomposable (Modular Architecture).

5.2 On phenomena similar to Quality Experience

None of the relationships shown in Figure 1 are flabbergastingly new, quite on the contrary: Each is known to the agile community at least in principle. For instance, in the terminology of the four-level scale of Agile Fluency [15], our teams Pay and OnM are on their way from a two-star team ("Deliver on the market's cadence") to a three-star team ("Optimize value").

Our contribution is collecting all of the relationships in a single model and providing evidence: We tell the story of two teams that exemplify the relationships shown and a third to corroborate that they indeed hinge on the influencing factors as we claim.

Many scientific and practitioner sources talk about a few of the relationships. One that talks about most of them is the Poppendiecks' book *"Lean Software Development: An Agile Toolkit"* [18], which describes 22 *thinking tools* for designers of agile development processes, grouped under 7 *lean principles*. The similarities are best explained by mapping our concepts to similar ones in the book:

- Several concepts map to various topics under thinking tool 1 "Seeing Waste" [18, pp.4-8]: Automated Deployment, Automated Tests, and Frequent Deployments to "Waiting"; Direct Feedback to "Motion"; Rapid Repair to "Defects".
- Direct Feedback, Quick Feedback, and Realistic Feedback pertain to thinking tool 3 "Feedback" [18, pp.22-27], from principle 2 "Amplify Learning", but are much more refined.
- Empowered To Deploy maps to principle 5 "Empower the Team", thinking tool 13 "Self-Determination" [18, pp.99-103].
- High Motivation also comes under principle 5, in thinking tool 14 "Motivation" [18, pp.108-109], specifically as the motivation building blocks "Competence" (via the availability of feedback), "Progress" (via the deployment-feedback cycle plus Rapid Repair), and "Belonging" (via Co-define Requirements, which again is a part of tool 13 "Self-Determination").
- Automated Tests maps to principle 6 "Build Integrity In", thinking tool 20 "Testing" [18, pp.145-149].
- Frequent Deployments, as the key outcome of Quality Experience, maps to principle 4 "Deliver as Fast as Possible" [18, pp.69-92] and is also a core topic of at least three currently-popular streams of lean thinking: Kanban [2], Continuous Delivery [9], and DevOps [10].

A few of our concepts do *not* occur prominently in the book: Neither Modular Architecture nor responsibility are explicit topics; the terms modularity or responsibility do not even appear in the index.

Conversely, some other Lean concepts can be glimpsed in our story without having a separate node in the diagram. For instance the complaint ending in *"[It] quickly adds up*

to a month or two" (OffProf.PO/I1-21:20) from Section 3.8 amounts (in its original form) to a "Value Stream Map", Lean thinking tool 2 [18, pp.9-13], and it echoes the problem of cycle time discussed in thinking tool 11 "Queueing Theory" [18, pp.77-83].

6. CONCLUSIONS

So what are the answers to our two initial research questions? Regarding **RQ1** (How is quality assured in agile teams without separate testers?), our answer could be phrased as "By creating a strong Quality Experience as defined in Section 3.1". In this work mode, the developers manage to fulfill the responsibilities of the conventional tester role by identifying aspects that are to be covered by efficient automated testing and others that are evaluated implicitly by the end-users. The risk of the latter is minimized by the developers being able to react independently and quickly in case of problems, such that the benefits of having realistic and direct end-user feedback prevail. Figure 1 summarizes the most important factors and relationships of and around Quality Experience.

The design justification, if you will, of a style of software development that enables Quality Experience can be summarized as follows:
(1) Hand-overs slow down the improvement cycle, so let us get rid of them and empower the development team to test and deploy all by themselves.
(2) An unwillingness to perform routine manual tasks will drive the team towards high degrees of test automation and deployment automation. This speeds up the iteration cycle further.
(3) With automated testing and deployment, small changes can be deployed very quickly, so we can risk to simply leave the arbitration of some aspects to the end-users instead of attempting to evaluate them in-house.
(4) The result is a strong feeling of responsibility as well as high motivation at the developers, a rapid stream of deployments, and minimal damage in case of field-failures (and hence modest risk); a highly desirable development situation.

The teams themselves emphasized point 3 as the key reason why it is appropriate for them to work without separate testers (OnM.D2/I9-17:00), (Pay.D2/I1-1:23:22).

So can you just kick out your testers and all will be well? From a management point of view, the following constraints need to be obeyed to realize Quality Experience:
(1) Teams must be assigned a holistic responsibility for a preferably vertical part of the web portal (service(s) or functional area(s)). Management may perceive some power loss.
(2) To accept this responsibility, the team must have sufficient freedom to decide and control all relevant aspects of their work, from participating in the definition of requirements down to operational monitoring. In particular, technology, architecture, and operation setup must allow for rapid and frequent deployment of new software versions. If this is not already the case, management must be willing to invest.
(3) The developers need to have sufficient common ground with the users of their software in order to interpret the feedback correctly.
(4) Occasional, short-lived field defects must be perceived as acceptable. Management must not be timid.

Regarding **RQ2** (advantages/disadvantages), the only *disadvantage* of a Quality Experience work mode appears to be that integration testing beyond the team level becomes harder (Pay.D1/I3-69:43). Quality Experience and testers can probably be combined if the process manages to avoid frictions from hand-overs. A *limitation* is that the above constraints cannot always be fulfilled. If development is too closely coupled with that of other teams or if deployment takes too long, a strong Quality Experience will not occur and it may be difficult to change this. Company or team culture might also get in the way.

If, on the other hand, the constraints can be fulfilled, there appear to be only *advantages*: teams are highly motivated and focused, development effort goes down (less coordination overhead, higher degree of automation), and presumably even deficiency-related losses go down, because those deficiencies are repaired much more quickly in the field than before. This is why we consider Quality Experience a desirable work mode and why we consider distilling its ingredients and relationships as shown in Figure 1 a worthwhile research contribution. Note, however, that our claim is strictly bound by the limitations described in Section 4, in particular the web portal domain.

If any further evidence of desirability is needed, here is some. Despite the high effort that will be required, IS24 and team OffProf have been working for some time now to refactor their part of the core application into the Modular Architecture that will allow the team to step over into Quality Experience as well: *"[A while ago,] we formed a team of three developers that work exclusively on modularizing the [ad-creation] workflows. Based on these learnings, we will attack the next areas then."* (OffProf.D/E3-42).

Acknowledgment

We thank SoundCloud and ImmobilienScout24 for allowing us in and thank all members of the three teams for talking to us.

7. REFERENCES

[1] Manifesto for Agile Software Development. http://www.agilemanifesto.org, 2001.
[2] D. J. Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010.
[3] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change, Second Edition*. Addison-Wesley Professional, 2004.
[4] J. Boeg. *Priming Kanban*. InfoQ/Trifork, 2nd edition, 2012.
[5] P. Bourque and R. E. Fairley, editors. *Guide to the Software Engineering Body of Knowledge (SWEBoK V3.0)*. IEEE Computer Society, 2014.
[6] L. Crispin and J. Gregory. *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.
[7] D. Feitelson, E. Frachtenberg, and K. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.
[8] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. In *Proc. 32nd ACM/IEEE Int'l. Conf. on Software*

Engineering, volume 1 of *ISCE '10*, pages 495–504, New York, NY, USA, 2010. ACM.

[9] J. Humble and D. Farley. *Continuous Delivery*. Addison-Wesley, 2011.

[10] M. Hüttermann. *DevOps for Developers*. Apress, 2012.

[11] IEEE Computer Society. *7th IEEE Int'l. Conf. on Software Testing, Verification and Validation (ICST)*, 2014.

[12] Int'l. Software Testing Qualifications Board (ISTQB). Advanced level in a nutshell. Available online: http://www.istqb.org/certification-path-root/advanced-level/advanced-level-in-a-nutshell.html, 2014.

[13] *J. of Software Testing, Verification and Reliability*. Wiley, 2015.

[14] V. Kettunen, J. Kasurinen, O. Taipale, and K. Smolander. A study on agility and testing processes in software organizations. In *Proc. 19th Int'l. Symposium on Software Testing and Analysis*, ISSTA '10, pages 231–240, New York, NY, USA, 2010. ACM.

[15] D. Larsen and J. Shore. Your Path through Agile Fluency. Available online: http://martinfowler.com/articles/agileFluency.html, 2012.

[16] M. V. Mäntylä, J. Itkonen, and J. Iivonen. Who tested my software? Testing as an organizationally cross-cutting activity. *Software Quality Journal*, 20(1):145–172, 2012.

[17] H. H. Olsson, J. Bosch, and H. Alahyari. Towards R&D as innovation experiment systems: A framework for moving beyond agile software development. In *IASTED Multiconferences - Proc. of the IASTED Int'l. Conf. on Software Engineering*, SE 2013, pages 798–805. ACTA Press, 2013.

[18] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley, 2003.

[19] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[20] S. Salinger, L. Plonka, and L. Prechelt. A coding scheme development methodology using grounded theory for qualitative analysis of pair programming. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, 4(1):9–25, 2008.

[21] K. Schwaber and J. Sutherland. The Scrum guide. Technical report, scrum.org, July 2013.

[22] J. Shanteau. Competence in experts: The role of task characteristics. *Organizational behavior and human decision processes*, 53(2):252–266, 1992.

[23] A. L. Strauss and J. M. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. SAGE, 1990.

[24] D. Talby, O. Hazzan, Y. Dubinsky, and A. Keren. Agile software testing in a large-scale project. *IEEE Software*, 23(4):30–37, 2006.

[25] J. Whittaker, J. Arbon, and J. Carollo. *How Google Tests Software*. Addison-Wesley, 2012.

[26] L. Williams, E. M. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. In *Proc. 14th Int'l. Symposium on Software Reliability Engineering*, ISSRE '03, pages 34–45, Washington, DC, USA, 2003. IEEE Computer Society.

[27] R. K. Yin. *Case Study Research: Design and Methods*. Sage, 2003.