Empirical Insights into the Perceived Benefits of Agile Software Engineering Practices: A Case Study from SAP

Christoph Tobias Schmidt
University of Mannheim
L15, 1-6
68161 Mannheim, Germany
christoph.schmidt@uni-mannheim.de

Srinivasa Ganesha Venkatesha
SAP Labs India
EPIP Area, Whitefield
Bangalore 560037, India
srinivasa.gv@sap.com

Juergen Heymann
SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
juergen.heymann@sap.com

ABSTRACT

SAP AG has taught more than 4,000 developers the Agile Software Engineering methodology since 2010. As such, the company offers a unique setting to study its impact on developers' work. In this paper, we discuss how developers perceive the impact of pair programming and test automation on software quality, delivered feature scope, and various team work aspects. We draw our findings from a company-wide survey with answers from 174 developers working in 74 teams and 15 product owners from five locations worldwide. We complement our findings with insights from two in-depths case studies with two development teams. As expected, our findings confirm that the studied practices help developers develop better software. Deviating from existing preconceptions, however, the responding developers do not report a significant drop in their development speed. In addition, high adopters are more proud of their own contributions to the team, report a better team learning, and feel more motivated.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Agile Software Development—*Pair programming, Automated Testing*

General Terms

Software Engineering in Practice

Keywords

Agile Software Engineering, Pair Programming, Automated Testing, Perceived Benefits, SAP AG, Case Study

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591189

1. INTRODUCTION

During the last ten years, many companies have fundamentally changed the way how they address challenges in the software development process from a plan-driven to an agile approach [5, 16]. Agile software development embraces a new way of thinking and working. It impacts how companies collaborate with their customers, how companies coordinate and govern the development organization, and most importantly, how engineers develop software. Agile Software Engineering values a set of core principles such as iterative shipments of working software increments, early investment into software quality by all developers, close customer collaboration to receive fast feedback, and a high focus on collaboration among software developers [7]. Several consultants have proposed many different agile methods and development practices which help developers implement these agile principles into their daily work and change how they accomplish their development tasks.

According to a world-wide survey with professional software developers [16], Scrum [15] and eXtreme Programming [2] are today the most widely used agile methods among professional software developers. In 2012, more than half of the representatively surveyed projects used Scrum. Among the used agile practices, automated testing with unit tests (used by more than 70% of the developers), refactoring (used by about half of the developers), and pair programming (about 30%) were the most popular.

During the last years, there has been a growing interest to understand how Agile Software Engineering impacts the development work and the performance of software developers cf. [1, 4, 8, 11, 13]. But still today, research in this field remains limited and there is a constant call for more empirical studies with professional software developers [3] as many studies are based on single case studies or student-based experiments.

This paper addresses this research gap with a case study from SAP. The company has jumped on the bandwagon of Agile Software Engineering and trained about 4,000 of its developers during the last three years. Almost all of these developers work in Scrum teams. Hence, the company provides a unique opportunity to study the impact of Agile Software Engineering practices used in Scrum teams. We are particularly interested in the perceived impact of agile engineering practices from the developer and the product owner perspective.

We discuss the benefits of pair programming and continuous quality checks with automated software tests early on

in the software development process. Many empirical studies on Agile Software Engineering are case studies with a limited number of software development teams or take their insights from student experiments [5]. This paper, however, shows findings from more than 200 professional software engineers working in 74 development teams in an international, large-scale software company.

We structure our paper as follows. First, we give an overview of our study context. Then, we provide our survey findings which we conducted in 2012. We discuss how high and low adopters of pair programming and automated testing differ in their perception. We look at the benefits of the studied practices concerning the delivered software quality, the delivered feature scope, and various team work aspects. We complement our survey findings with insights from two teams based in Germany and India. These case studies enabled us to analyze objective archival data of the actual test coverage and the defect rates. Finally, we provide an outlook for potential follow-up studies in this field of research.

2. RESEARCH CONTEXT

2.1 SAP Case Study

SAP AG is a world leader in enterprise software and software related services. It is a large global organization with locations in more than 130 countries. From back office to boardroom, warehouse to storefront, desktop to mobile device - SAP empowers people and organizations to work together more efficiently and use business insight more effectively. SAP applications and services enable more than 248,500 customers to efficiently operate and improve their businesses.

During the last years, SAP has fundamentally changed its development methodology. Starting in 2008, many development teams were reorganized into Scrum teams of about ten developers [14]. After 2010, the Agile Software Engineering training program was introduced at large scale. The aim of the training program was to improve development efficiency and bring innovation to customers at faster rates. The team-based program teaches Agile Software Engineering practices such as test-driven development, test isolation, continuous integration, refactoring, pair programming besides a general Scrum refresher. During a one week class training, these agile practices are illustrated based on hands-on exercises. Afterwards, the teams are coached in their day-to-day work environment for three weeks while working on their normal development tasks. Despite the tremendous investment, development teams are not pressured to use the taught practices after participating in the program. Instead, they can voluntarily decide how and when to apply them.

The training program meant a significant investment and a considerable change to the company's development culture. Hence, various research efforts have been taken to understand its impact. Among others, we conducted a survey study to better understand the perceived impact for developers and product owners. In addition, we interviewed several developers who could draw from at least six months of experience with the studied development practices at the time of our interviews. In addition, several Agile Software Engineering coaches were interviewed. These coaches had supervised and mentored many development teams during their day-to-day work.

The study had two main objectives: Learning about the program's sustained adoption and understanding the change for developers' work related to Agile Software Engineering. In this paper, we primarily present data from a survey study and discuss them with our insights from two case studies with two development teams.

2.2 Research Approach

In spring 2012, we invited about 900 developers from several SAP locations world-wide to voluntarily participate in an online survey. All invitees had participated in the same training program at least six months before that point of time. In total, more than 200 people answered the survey. The respondents included 15 product owners and 174 developers who had indicated in the questionnaire to be spending more than 20% of their working time developing software. Overall, all respondents worked in 74 development teams in five countries.

The questionnaire included three sections. In the first, we asked how intensively the developers used the taught development practices in their daily work. For instance, we asked *"how much of your development time do you program in a pair?"* (pair programming) or *"what is the test coverage of your new code?"* (test coverage). The respondents answered on a 10-point scale ranging from "about 10%" to "about 100%".

The second section of the survey covered three areas of interest that we expected to be impacted by the studied practices. First, we asked if the developers experienced a change in the quality of software they delivered. Second, we asked if their team delivered more or less features than before and, third, if they saw any impact on different team work factors. For instance, we asked *"To what extent did the delivered feature scope per sprint change after you participated in the training"*, *"To what extent did the number of reported defects change"* or *"To what extent did the common understanding (i.e. common quality understanding, common team goals, understanding of done) change"*. The respondents answered these questions on a Likert-scale ranging from "deteriorated drastically (-2), got worse (-1), did not change (0), got better (+1), to improved significantly (+2)". Finally, we asked the participating developers to rate how proud they were of their own work.

3. SURVEY FINDINGS

We first describe how intensely the responding developers used the studied development practices. Then, we present the perceived effects on software developers' work.

3.1 Adoption Intensity

Despite having participated in the same training program, developers varied widely in terms of how intensively they adopted the taught agile engineering practices. In this paper, we focus on two agile development practices: pair programming and test-driven development. Both development practices are not only the most widely used agile engineering practices [16, 2], but also the ones that most fundamentally change the way how the software is developed. Both practices endorse core agile principles [7] as, for instance, a strong focus on developer collaboration, intensive investment into high software quality early on in the development process, and the iterative work mode.

What is **pair programming**? Two developers work on a single computer with only one keyboard and one computer mouse to collaboratively develop software [17]. One developer is the so-called *driver* who has the keyboard at that time to write new code. The partner is the *navigator* who permanently monitors the driver, provides feedback, and gives suggestions how to better solve the implementation tasks. The two developers frequently switch their roles.

In the survey, we asked two questions about pair programming. Developers rated how much of their development time they programmed with a partner (PP1)[1] and the number of backlog items which they accomplished with a programming partner (PP2)[2]. The given answers correlated strongly (P1 and P2 have a Pearson correlation coefficient of 0.9 on a 0.01 significance level). Figure 1 shows the distribution of the given answers to those two questions. On average [(PP1 + PP2)/2], all respondents reported to pair program about 40% of their time/backlog items (standard deviation of 26%).

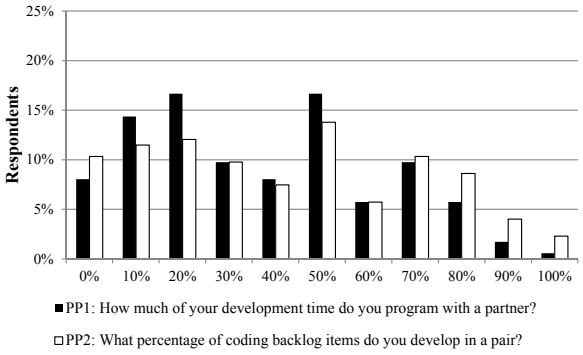

Figure 1: Pair Programming Adoption Intensity

Test-driven development is an iterative development approach that relies on the repetition of very fine-granular development cycles. Before writing the software code, developers write an automated test case that covers the to be implemented software functionality. Test cases comprise a set of specified conditions under which the software code is working as originally expected. After a test cases is implemented, only the necessary software code to pass exactly that newly added test case is added. Once the implemented functionality passes the test cases, the productive code is refactored [2], i.e. the internal code quality is improved without changing its external functionality, and the test-driven development cycle is repeated. Developers use their test cases to automatically and continuously check the quality of their software code.

Extensive test-driven development leads to a high, semantic **test coverage** of the production code with automated test cases. Regarding test automation, we included two questions in the survey: the general test coverage of new code (TC1)[3] and the share of newly added or modified software functionality that is covered by automated tests (TC2)[4]. Both variables correlated (Pearson correlation coefficient of 0.5 on a 0.01 significance level). We also averaged the answers for TC1 and TC2. The averaged values show an average test coverage of 61% and a standard deviation of 25% (see Figure 2).

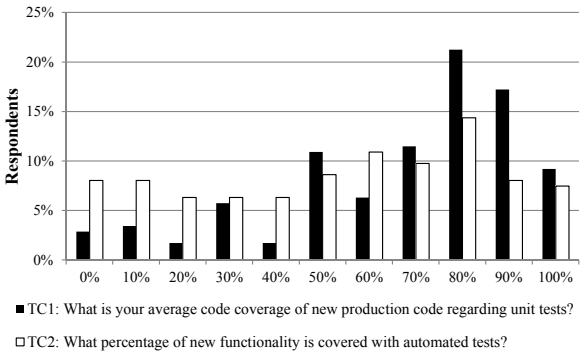

Figure 2: Reported Test Coverage

In a next step, we clustered the respondents according to their intensity of using pair programming and their test coverage. For that purpose we ran a two-step cluster analysis with IBM SPSS Statistics for the variables PP1 and PP2, and a second cluster analysis for TC1 and TC2. The algorithm ended with three clusters for pair programming. As we were interested in the differences between high and low adopters only, we dropped the partial adopters to study the extrema in our population. High adopters reported to pair program about three quarter of their development time and new functionality (75%). High adopters of the second practice, reported a test coverage of 76%, on average. Low adopters of the pair programming variables only paired about 15% of their time. Developers with a low test coverage had, on average, 32% of their new code or software functionality covered with automates test cases.

Table 1: High and Low Adopters

	Pair Programming (PP1+PP2)/2	**Test Coverage** (TC1+TC2)/2
All	n = 166	n= 159
- Mean	40%	61%
- Standard Dev.	0.26	0.25
High adopters	n = 40	n = 105
- Mean	75%	75%
- Standard Dev.	0.09	0.13
Low adopters	n = 76	n = 54
- Mean	15%	32%
- Standard Dev.	0.10	0.15

Based on these clusters, we analyzed differences in the perceived benefits of these agile practices as described in the following section.

[1]PP1: How much of your development time do you program in a pair?
[2]PP2: What percentage of coding backlog items do you develop in a pair?
[3]TC1: What is your average code coverage of new production code regarding unit tests?
[4]TC2: What percentage of new functionality is covered with automated tests?

3.2 Perceived Benefits

The Agile Engineering approach promises several benefits. Due to the continuous focus on software quality right from the beginning of the development process, developers are expected to produce better software while delivering the same feature scope. We further expected that the required additional effort to guarantee high software quality early on in the process is outweighed by the saved time that developers would have to spent on fixing bugs later in the process. Besides, agile development teams are expected to show better team work than non-agile teams [2, 15] due to a higher transparency and a continuous emphasis on collaboration among the team members. To evaluate if these promises actually hold true, we included aspects of all three categories (quality, scope, team work) in the second part of our survey.

Unfortunately, our research context did not allow us to repeat the survey before and after the developers had participated in the training program. Without a pre-training reference point, we were not in a situation to analyze developers' view on the studied practices over time. For that reasons, we asked developers to rate how the quality of their software, the feature scope they delivered, and various teamwork factors *changed* comparing the situation before and six months after the training[5]. Developers rated the perceived change on a scale ranging from "deteriorated drastically (-2), got worse (-1), did not change (0), got better (1), to improved significantly (+2)". We expected the high adopters of the studied practices to answer significantly higher to the three categories of potential benefits than the low adopters.

3.2.1 Quality

First and foremost, Agile Software Engineering practices aim at delivering better software quality. The studied development practices help developers focus on high software quality early on in the development process. For that reasons, agile development teams mostly do not have dedicated testers who ensure a high level of quality after the software has been finalized by the developers.

Both pair programming and automated tests let developers continuously check the quality of new or modified code. Through pair programming, two developers permanently discuss how to best solve their development tasks. They review their work results to prevent and fix software bugs before they might cause any problem. Due to the test-driven development approach, a comprehensive test suite of automated test cases helps developers find errors early on in the development process and safeguard already built functionality. Consequently, we expected high adopters of each development practice to report a significantly higher level of improvement than the low adopters.

Software quality can generally be categorized into external and internal software quality [10]. In our research context, software users can issue tickets to report defects or bugs in the software that need to be fixed. Hence, we consider the *number of reported defects per sprint (Q2)*[6] as a reliable indicator for the external software quality. Developers are aware of these tickets as those incidents are directly routed to the development teams to be solved. Hence, we asked the developers to indicate the number of reported defects per sprint.

While the external quality of a software determines the fulfillment of software users' requirements, the internal quality determines the ability to move forward on the development project. The internal software quality is considered a necessary prerequisite for good external quality. It comprises various aspects as maintainability, readability, or testability of the software code [10].

For the internal software quality, we asked developers to report the perceived changes in the *API quality (Q3)*[7] of their software, their *code modularity (Q4)*[8], and their *code understandability (Q5)*[9]. We averaged the provided answers of all developers for Q3, Q4, and Q5 to come up with a single indicators (Qint)[10] (see table 2). Finally, we asked the developers to rate the overall *change in the software quality in general* after they had participated in the training program (Q1)[11].

Table 2 reports the answers for the high and low adopters of both agile development practices. Both cohorts (low and high adopters) report a perceived improvement of all software quality aspects. Moreover, the high adopters report a higher level of improvement regarding all studied software quality aspects compared with the low adopters. However, not all differences are statistically significant. For both agile development practices, we tested the null hypotheses that the distributions of high and low adopters are identical for the different variables. Due to the ordinal scale and no normality in the data of the provided answers, we ran a Mann-Whitney-U test to check our hypotheses on a significance level of 0.05. For the perceived change in quality, only two hypothesis could be falsified: high adopters of pair programming see a significantly higher level of improvement of the software quality in general (Q1) and high adopters of pair programming see a significantly higher level of improvement regarding the internal software quality metrics (Qint).

3.2.2 Delivered Scope

On the one hand, writing test cases can be very time-consuming. This might lead to a reduced velocity of delivering new software features as developers have less time writing new feature code. On the other hand, test cases help developers mitigate errors and consequently minimize rework. Hence, Agile Software Engineering practices are often considered to lead to improved software quality at the expense of a reduced development scope in a given period of time. Consequently, we were interested in the effects on the development velocity. In our survey, we asked two questions covering the change in the delivered scope. First, developers reported *how much the delivered feature scope changed (S1)*[12] and the *change in the amount of developed software code (S2)*[13].

Overall, all answers to these questions demonstrate no notable reduction in the delivered scope after the training. On a scale ranging from -2 (deteriorated significantly) to +2 (improved significantly), the average answers of all develop-

[5]To what extent did {Q1, Q2, Qint, S1, S2, TW1, TW2, TW3} change since you participated in the training program?

[6]Q2: Change in number of reported defects per sprint

[7]Q3: Change in the API quality

[8]Q4: Change in the code modularity

[9]Q5: Change in the code understandability

[10]Qint: (Q3 + Q4 + Q5) / 3

[11]Q1: Change in software quality in general

[12]S1: Change in delivered feature scope per sprint

[13]S2: Change in delivered software code per sprint

Table 2: Differences in the Perceived Benefits Comparing High and Low Adopters

	Pair programming					Test coverage				
	High adopters		Low adopters			High adopters		Low adopters		
Adoption intensity	Mean .75	SD .09	Mean .15	SD .10	Sig.*	Mean .75	SD .13	Mean .32	SD .15	Sig.*
Change in software quality (✢)										
- Q1: Quality in general	**1.03**	.49	**.69**	.66	.005	**.93**	.60	**.79**	.69	-
- Q2: Reported defects	**.62**	.64	**.39**	.68	-	**.60**	.68	**.42**	.66	-
- Qint: Internal Quality	**.79**	.41	**.55**	.57	.010	**.72**	.48	**.60**	.53	-
Change in delivered scope (✢)										
- S1: Delivered features	**.34**	.78	**-.01**	.58	.007	**.13**	.74	**.13**	.59	-
- S2: Delivered software code	**.11**	.70	**-.08**	.61	-	**.01**	.72	**.04**	.59	-
Change in team work (✢)										
- TW1: Motivation	**.85**	.54	**.36**	.75	.000			- - -		
- TW2: Common understanding	**1.15**	.59	**.64**	.70	.000			- - -		
- TW3: Team learning	**1.15**	.63	**.61**	.62	.000			- - -		
Work pride (✳)										
- P1: Own contribution	**1.23**	.58	**.92**	.79	.039	**1.24**	.55	**.85**	.85	.003
- P2: Team's contribution	**1.05**	.57	**.70**	.89	.026	**.96**	.74	**.76**	.81	-

(✢) Scale 1: (-2) deteriorated drastically, (-1) got worse, (0) did not change, (1), got better, (2) improved significantly
(✳) Scale 2: (-2) strongly disagree, (-1) disagree, (0) neutral, (1) agree, (2) strongly agree
** Mann-Whitney U Test (significance level of 0.05) testing the difference between high and low adopters*

ers are 0.15 (for $S1_{all}$) and 0.04 (for $S2_{all}$). Consequently, developers do not report a change of the delivered speed or even a deteriorating development velocity. Contrary to our expectations, the answers of high adopters of pair programming even indicate a small improvement of the development speed compared to before the training. Comparing high and low adopters, only the answers of the high adopters of pair programming are significantly different from the low adopters.

The answers provided by the 174 developers was in line with the answers provided by the 15 participating product owners. These product owners reported no change of the delivered scope regarding both questions (on average, $S1_{PO}$=0.02 and $S2_{PO}$=0.00).

3.2.3 Team Work and Work Pride

Various studies emphasize the importance of effective team work for high performance software development teams c.f. [9]. Pair programming may lead to a high level of collaboration within the development team which, in turn, may positively influence its performance. In line with previous studies cf. [17, 4], we anticipated three major benefits of pair programming for software development teams. First, we expected pair programmers to have a higher team motivation (TW1)[14], which is generally considered to be an important antecedent for successful team work [12]. Second, pair programming helps developers share their ideas, build an awareness of other team members' expertise, and develop a common understanding of what good software quality is, about common team goals, and what needs to be done for development task to be considered 'done'. Therefore, we asked developers about the perceived change in the common understanding after the training (TW2)[15]. Other studies have shown that a common understanding of expertise in the team is important for successful software development teams [6]. Third, pair programmers permanently learn from each other on how to develop software. Pair programmers discuss how to solve a problem, constantly observe their programming partner, and provide each other feedback on work results. Hence, pair programming is expected to increase team learning between developers. We therefore asked the developers how much the team learning changed after the training (TW3)[16].

For TW1, TW2, and TW3, high adopters of pair programming reported a significantly stronger improvement than the low adopters: motivation (0.85 vs. 0.36), common understanding (1.15 vs. 0.64) and team learning (1.15 vs. 0.61). The answers of the participating product owners supported these findings: motivation (0.42), common understanding (1.00) and team learning (1.00)

Finally, we asked developers to rate their personal work pride on a Likert-Scale ranging from strongly disagree (-2), disagree (-1), neutral (0), agree (1), strongly agree (2): developers' pride with their personal contribution to the team work (P1)[17]and developers' pride with their team's newly developed software (P2)[18]. Overall, we found a very high level of agreement for both statements among all answering developers (overall average for P1=1.10 and P2=0.88). For pair programming, however, high and low adopters differed significantly in their answers with high adopters being more proud of their work than low adopters. Furthermore, developers with a higher test coverage reported a significantly higher agreement to P1 whereas the answers for P2 did not differ significantly. Figure 3 provides an overview of these findings.

[14]TW1: Team motivation
[15]TW2: Common understanding (common quality understanding, common team goals, understanding of 'done')
[16]TW3: Learning in the team (activities to acquire, share, refine or combine task-relevant knowledge)
[17]P1: I am proud of my own contribution to the team's work
[18]P2: I am proud of the team's newly developed software

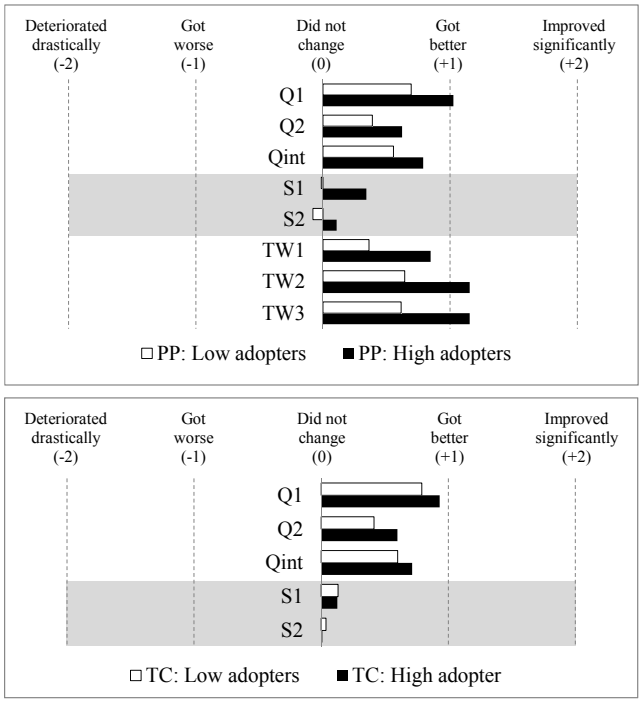

Figure 3: Reported Benefits - High and Low Adopters

3.3 Discussion of the Survey Findings

Our survey results indicate that the studied engineering practices are well established among more than half of the responding developers. High adopters report a positive impact on the software quality while delivering the same scope of features. In particular, our respondents' answers confirm our expectations that pair programming helps developers deliver better internal software quality, i.e. a better modularity and understandability of their code. Second, the potential reduction in the delivered feature scope can not be confirmed with our data. Furthermore, high adopters report a positive impact on different team work factors, i.e. high adopters of pair programming experience a positive effect for their team work. They see a higher motivation in their team, a better common understanding among team members, and an improved knowledge transfer within the development team. Finally, high adopters of both practices are more proud of their contributions.

We should not ignore that there are certain limitations to our research approach. Our data is based on developers' subjective perceptions asked about six months after the trainings. For instance, a high test coverage might show its impact on the reported defects only after a longer time period due to longer release cycles. Second, developers answers are based on subjective self-assessment which might be biased towards individuals' expectations. Third, the survey findings do now allow us to study team specific context factors to understand, for instance, why some developers still remain reluctant to use the methods or why the studied practices might be less effective in certain work contexts.

To overcome these limitations, we conducted several team-based case studies at two SAP locations. These case studies included interviews with developers and project leaders. We also analyzed the teams' actual test coverage as well as the reported defect rates by comparing two releases before and after the training. As we are interested in the impact of pair programming and a high code coverage with automated tests, we present findings from two high adopter teams in the following section. We conclude our paper with a discussion of the survey and the case study.

4. INSIGHTS FROM TWO HIGH ADOPTER TEAMS

We present our case study findings from one development team in Germany and second team from India. All teams had participated in the same training program more than half a year before we started the first interviews. We selected these high adopter teams as they were working on different types of software using either Java or ABAP as their programming language. Both teams worked in a Scrum mode and were stable over a longer period of time. We introduce each team and provide relevant context information. Then, we present findings from the interviews with different team members as well as from our archival data analysis.

4.1 Team A

Team A is located in Germany and develops software with Java technology. The team's software helps customers model and automatically execute business processes. The software is highly complex in nature as it involves various actors and events to make the development and deployment of business processes as flexible as possible.

Already before the training, the co-located team was well experienced in working with Scrum. The team had a product owner, a scrum master, several developers, and team-internal quality assurance engineers. As a whole, the team could draw on many years of experience in professional software development with a mix of very experienced developers and only a few newcomers. Before the training, the team had faced many quality issues at the end of their development cycles of previous releases causing a lot of stress and bug fix efforts.

The team had participated in the training on Agile Software Engineering about eight months before our interviews after which they radically changed the way in which they accomplished their implementation tasks. In particular, the team decided to extensively use pair programming to implement the technical backlog items and to focus heavily on test-driven software development. According to the survey results, developers reported a test coverage of more than 80% and claimed to develop about 55% of their time with a partner. One developer explained "we now have an internal test library for writing tests for our product. This API is easy to use, extensible and shows fast results. No wonder that writing tests along with the code has become the habit".

As a result of the new work mode, the team reported positive changes on many dimensions affecting both internal and external quality aspects. One developer explained that "pair programming will reduce the number of bugs at the early stage as we are constantly reviewing the codes as we write them". As an example of how their external quality significantly improved, we may take a look at Figure 4. It shows the reported customer defects over two releases (x, x+1) as against the time lines in months after release to

customer phase. Release x was developed before the training and release x+1 was developed shortly afterward.

When interpreting reported defects, the following fact has to be taken into consideration: With an increasing number of active customers using the software, the number of defects is also expected to increases as more functionality is actively used leading to a higher defect rate over time. Considering this well-known effect and to correctly interpret the data, we normalized the reported number of defects by the number of active software installations at customers' sides. Moreover, we disclose the gathered data points only relative to the number of defects reported in the first months of release x.

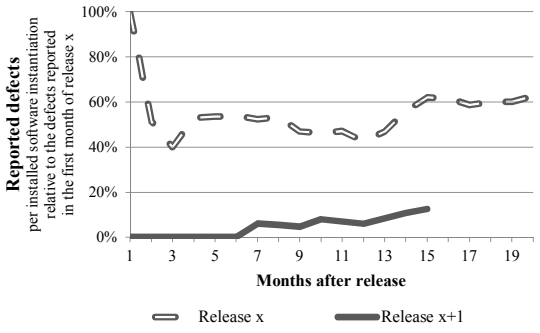

Figure 4: Team A: Reported Defects of the Release Before and After the Training

Our findings clearly indicate that the number of defects significantly dropped to a level close to zero from release x. In general, the team's rework time significantly decreased compared to the release before the training. The Scrum Master of the team nicely described this finding: "After the training, the total development velocity has improved and rework decreased significantly". The manager of the team also mentioned the positive effect in our interview stating that the "developers are more proud, more motivated and less stressed". Finally, the vice president of the team's product area summarized these effects when we asked him about the main benefits of the new development approach:

> *As a team you gain reliability. Why? Due to the high test coverage, developers who change code know that the old features still work and the new features also work. Second, I see a positive effect on the quality and speed of development [...] and quality and speed again motivate the teams [...] they feel that the can produce and that they can do more than before. (Team A's Vice President)*

Overall, Team A welcomed the studied Agile Software Engineering practices and acknowledged the positive impact on the daily work. First, the high test coverage helped the team not only to deliver better software quality but also to reduce the work stress during the development sprints and before the end of the releases. Due to pair programming, developers appreciated the improved knowledge transfer in the team and the continuous partner who reviewed the code. Both practices helped the developers to "sleep well at night" knowing the software they build is safeguarded by automated tests while being away from their desk. Finally, our analysis of the reported defects demonstrates the positive impact for the company as every defect represents an unsatisfied customer and related costs for fixing the defect.

4.2 Team B

Team B works on delivering a state of the art development environment to enhance developers' productivity. The development environment comprises various development tools, re-usable software components, and code libraries to enable application developers to build applications faster and more productively.

Team B is located in India and is a little less experienced than Team A with young developers who are mostly new to the used technology. The team consisted of a product owner, a scrum master and 8 developers with 2 quality engineers as part of their scrum team. Already before the training, the team was well versed with Scrum and was open to embrace Agile Engineering practices. They worked on a release y without Agile Software Engineering practices and took the training just before the release y+1.

To better understand the impacts of the changed development approach, the team tracked two metrics: number of bugs found after the development phase and the number of features delivered per release. After the training and coaching phase, the team consistently implemented the practices as taught in the training, especially test-driven development and pair programming. The team reaped the benefits fully in release y+1. As can be seen from the Figure 5, the defects found after the development phase reduced significantly in release y+1 (drop of more than 60%) compared to the previous release. Moreover, there was an increase (more than 50%) in the number of delivered features in the second release. The features were comparatively equal in terms of size and complexity.

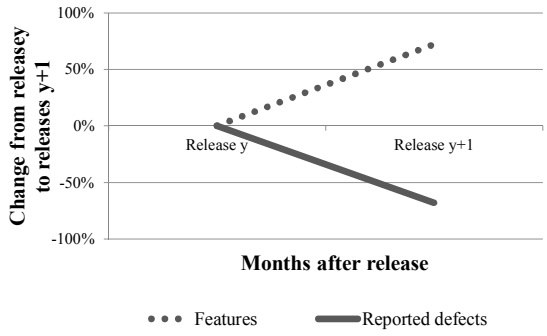

Figure 5: Team B: Change of Delivered Features and Reported Defects Before and After the Training

The team was also looking for a development practice which sustainably bridges the knowledge gap on the code level among the team members. They realized that pair programming helps them to exactly achieve that goal as one young developer explained "we can learn things faster by pairing with experts during pair programming". As they started adopting pair programming more rigorously, they found out that it is not only a practice which helps them to bridge the knowledge gap, but also a way to write more testable code. While pair programming with two developers who constantly switch their roles from navigator to driver and vice versa, they found it easy to challenge each other on the code quality. From their experience, they found that

test-driven development and pair programming have a very close relationship as development practices. When we interviewed this team, one of the developers explained "while pairing, it is easy for me to remind my partner to focus on the testable code. Test-driven development becomes a natural practice. For example, I learned a lot about many refactoring practices in Eclipse while pairing".

It is a normal tendency to ask many productivity related questions when teams adopt pair programming when they are expected to deliver a set of features in a fixed time frame. Why do two developers have to work on a feature when the same feature can be done quite independently? This team also confronted us with these same questions. Nevertheless, they sustained it for a longer time. As can be seen from the Figure 5, this payed off. This team delivered more features with better quality as the defects reported in the subsequent release have become less. We interviewed a trainer about these issues. He had coached Team B and told us about his experiences: "When we combine collaborative development practices like pair programming along with test-driven development, the team benefits a lot and delivers high quality products without compromising on the speed".

Overall, Team B adopted the studied Agile Software Engineering practices to a high degree and was quite satisfied with its promises. The team reduced the defects within the development cycle drastically by writing a lot of testable code using test-driven development. They could enhance the knowledge base and the confidence level of the entire team by constantly doing pair programming. The number of features delivered to the customers have seen an upward trend without affecting the quality with the same release timelines. Finally, our analysis of their defects and the number of features delivered in similar release cycles demonstrates the positive impact on the team of their quality and agility and finally on to the customers.

4.3 Discussion of the Interview Findings

So, what does the above two case studies convey to us as far as the survey findings are related? The teams who have adopted the Agile Software Engineering practices have reported many benefits of its usage in their own context. Though the contexts (location, experience and technology) in both teams were different, they report similar benefits in terms of higher productivity, quality and team work. The teams in which pair programming has become a standard practice report common code ownership where most of their developers are capable of working on other developers code base with ease. Pair programming has also resulted in bridging the knowledge gap levels within the team. The test-driven development has resulted in higher test automation both at unit level and at system integration level offering more confidence to the developers about the quality of their software. Overall, the higher adopter teams perception about the practical use of Agile Software Engineering practices has been very positive.

5. CONCLUSION AND OUTLOOK

Agile Software Engineering is becoming more and more popular [16]. With this SAP case study, we intend to contribute to the existing knowledge base about its effectiveness with insights from more than 200 software development professionals working in more than 70 teams. The surveyed and interviewed developers clearly indicate that the studied Agile Software Engineering practices have a positive effect on quality and efficiency of the development teams. Still, one training week with a three week coaching alone cannot create a sustainable change in development culture and there are many activities on the way to maintain the momentum. It should be noted that this program was the first 'quality oriented topic' that was truly accepted and promoted by developers.

The changed development approach should first and foremost show its benefits in developers' daily work. Hence, developers' perception of the studied engineering practices and the actual adoption rate are essential to assess its impact. One surprising result of our findings was that you can get much better quality without spending more effort. We see that the answer to bad quality is not 'work more', but 'work differently'. Also, the strength of the improvement was sometimes surprising, as documented in the case study of Team A. Another surprise was that even pair programming seems to not result in a net loss of feature output for the benefit of better quality. For test-driven development, the saved effort in debugging and defect analysis can be intuitively understood to compensate for the small additional effort of writing unit tests. For pair programming, however, the efficiency gain was not expected. Finally, our case study insights give a quantitative glimpse of strength of the impact. For the studied team, the studied development practices led the defect rate drop by a factor of more than five (see Figure 4).

Another important factor of success is the focus on engineering skills and not 'quality' per se. The focus on internal code quality as opposed to just delivering features faster is a key element to software development sustainability in the long run. In particular, pair programming can be seen as a key ingredient to enable deep changes of work practices. To put it simple, we see that "pair programming develops code *and* people"; both are necessary for successful software development. We also see several subtle positive effects on team aspects such as improved knowledge transfer, team motivation and better common understanding of the product and code.

With this study, we only see a start of a series of research on agile software development in large-scale software development. We are currently running additional studies focusing on (1) an in-depth look at team performance of Scrum teams and their driving factors and (2) inter-team coordination aspects of agile development teams.

6. REFERENCES

[1] E. Arisholm, H. Gallis, T. Dybå, and D. I. Sjoberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2):65–86, 2007.

[2] K. Beck. *extreme programming eXplained : embrace change*. Addison-Wesley, Reading, MA, 2000.

[3] T. Dingsøyr, S. Nerur, V. Balijepally, and N. B. Moe. A decade of agile methodologies: Towards explaining agile software development. *The Journal of Systems & Software*, 85(6):1213–1221, 2012.

[4] T. Dybå, E. Arisholm, D. I. Sjoberg, J. E. Hannay, and F. Shull. Are two heads better than one? on the effectiveness of pair programming. *Software, IEEE*, 24(6):12–15, 2007.

[5] T. Dybå and T. Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10):833 – 859, 2008.

[6] S. Faraj and L. Sproull. Coordinating expertise in software development teams. *Management Science*, 46(12):1554–1568, 2000.

[7] M. Fowler and J. Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[8] J. E. Hannay, E. Arisholm, H. Engvik, and D. I. Sjoberg. Effects of personality on pair programming. *Software Engineering, IEEE Transactions on*, 36(1):61–80, 2010.

[9] M. Hoegl and H. G. Gemuenden. Teamwork quality and the success of innovative projects: A theoretical concept and empirical evidence. *Organization Science*, 12(4):435–449, 2001.

[10] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[11] G. Lee and W. Xia. Toward agile: An integrated analysis of quantitative and qualitative field data on software development agility. *MIS Quarterly*, 34(1):87, 2010.

[12] J. Mathieu, M. Maynard, T. Rapp, and L. Gilson. Team effectiveness 1997-2007: A review of recent advancements and a glimpse into the future. *Journal of Management*, 34(3):410–476, 2008.

[13] S. Sarker and S. Sarker. Exploring agility in distributed information systems development teams: An interpretive study in an offshoring context. *Information Systems Research*, 20(3):440–461, 2009.

[14] J. Schnitter and O. Mackert. *Large-Scale Agile Software Development at SAP AG*, volume 230 of *Communications in Computer and Information Science*, chapter 15, pages 209–220. Springer Berlin Heidelberg, 2011.

[15] K. Schwaber and M. Beedle. *Agile software development with Scrum*. Series in Agile Software Development. Prentice Hall, Upper Saddle River, NJ, 2002.

[16] VersionOne. 7th Annual State of Agile Development Survey, 2012.

[17] L. A. Williams and R. R. Kessler. All I really need to know about pair programming I learned in kindergarten. *Communincations of the ACM*, 43(5):108–114, 2000.