Agile Software Engineering Techniques: The Missing Link in Large Scale Lean Product Development

Scheerer, Alexander; Schmidt, Christoph T.; Heinzl, Armin; Hildenbrand, Tobias*; Voelz, Dirk*

Institute for Enterprise Systems
University Mannheim
Schloss
68131 Mannheim, Germany
{scheerer, christoph.schmidt, heinzl}@uni-mannheim.de

*SAP AG
Dietmar-Hopp-Allee 16,
69190 Walldorf, Germany
{tobias.hildenbrand, dirk.voelz}@sap.com

Abstract: Many software development companies have fundamentally changed the way they organize and run their development organizations in the course of the last decade. Lean and agile software development became more and more common. Lean focuses on continuous value generation based on a framework of principles known from manufacturing. But how do software developers actually implement these principles in their daily work? Based on insights from several software development teams at a large-scale enterprise software company in Germany, we show that agile software engineering techniques seamlessly integrate into lean product development principles. This paper shows empirical insights on how to implement these principles in a professional context and every-day work.

1. Introduction

For many years, large software firms have managed their development organization in a waterfall-like and plan-driven manner to overcome scaling and complexity issues. However, this often resulted in manifold issues such as an ever-growing complexity in the technology stack and an unnecessary level of bureaucratic overhead. Companies realized that a fundamental change was necessary to cope with these scaling problems after it became inevitable that the traditional approach was especially insufficient in large-scale organizations. Lean software development with its underlying principles, which are derived from lean manufacturing [WJ03], promised a solution for many companies. The approach focuses on value and value creation addressing the central identity of the software industry where margins come from innovation and economies of scope, instead of economies of scale [HM12].

The fundamental transition from a traditional plan-driven to a lightweight development approach poses a significant challenge for large companies whose employees have been using the former approach for years. We follow Conboy's [Co09] definitions and consider lean as a framework of principles aiming at the "contribution to perceived customer value through economy, quality, and simplicity". Lean pursues a similar set of

objectives as the concept of information systems development (ISD) agility. Agile ISD methods build on “the continual readiness […] to rapidly or inherently create change, proactively or reactively embrace change, […] while contributing to perceived customer value” [Co09].

As Scrum has become a quasi-standard agile method used by many software companies of different sizes, most large-scale lean implementations also started their transition to lean on the team level with this method. However, empirical evidence on both lean and agile methodologies is scarce [PP03], [WCC12], especially when it comes to the adoption by large software vendors. The transition towards lean and agile development remains more a "leap of faith" than a concise engineering effort since too little is known about the idiosyncrasies and pitfalls associated with it.

Are lean principles combined with Scrum really enough to meet the needs of large-scale software development companies? Despite combining lean with agile concepts on the team level by implementing Scrum as process framework, many developers perceive both lean product development and agile software development to be on a yet too general and abstract level for their daily work (see [Re09] and [HM12], for instance). Following the lean idea, its implementation should focus on software products and target those levels where value is created: the development teams, their code, and each individual developer. This is where Agile Software Engineering Techniques (ASET) come into play (cf. Figure 1). We argue in this paper that agile techniques are the missing link to effectively implement and sustain lean product development principles in large scale software companies and finally make them valuable for software developers’ daily work.

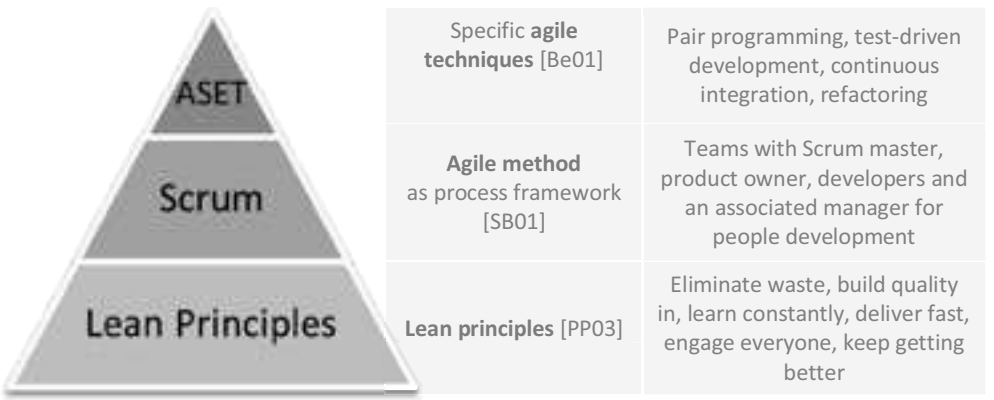

Figure 1 Overview of underlying key concepts in our case study

Previous studies already identified agile development techniques as a major success factor in agile software projects, e.g. [Be01]. Hence, we assume a positive outcome of adopting ASET in itself. Based on that, we discuss how agile techniques are adopted in combination with Scrum as process framework and Lean software development with results from an empirical study we carried out at a large enterprise software company in Germany that implements Lean. In order to guarantee the teams exposure to agile techniques we focused on pair programming (PP), test-driven development (TDD), continuous integration (CI), and refactoring (REF) [SS11] which had been introduced to the developers via an intensive team course prior to our investigation.

2. The Research Process

We conducted our study on PP, TDD, CI, and REF during summer and fall of 2011. It comprised twenty one-hour-long interviews with members from four development teams. A questionnaire survey covering the same teams complemented the interviews. All interviewees had attended a one-week training on agile development techniques followed by a three-week coaching phase. To get an in-depth understanding of the techniques and their application in practice, the research team attended the training program as well. This setup ensured that every developer could draw from at least three months of practical and solid theoretical knowledge. The teams had all attended a lean training course within the last two years and were embedded in a lean organization with a focus on continuous value generation. The interviews were transcribed and the data carefully analyzed with statistical and qualitative software packages.

All teams were co-located, had implemented Scrum as a process framework [SS11], [SB01] for at least one year and had several months of ASET experience in a productive setting after the training (cf. Table 1). We interviewed representatives of all Scrum roles (Product owner, Scrum master, and development team member) as well as the teams' line managers and supplemented our analysis with relevant documents concerning the underlying development framework and training materials.

	Team Context		
	Scrum Experience [months]	ASET Experience [months]	Team continuity [months]
Team 1	24	7	24
Team 2	12	3	24
Team 3	48	5	24
Team 4	18	8	12

Table 1 Team contexts

3. The Teams' Attitude towards ASET

The entire training sessions were solely teaching agile techniques with the intention to support software engineers in their daily development work, i.e. having a substantial effect on the development time and gains in the software quality as previously stated by different studies [DB11], [Na08], [MW03], [CW01], [Dy07], [MT04].

Our study results clearly demonstrate that most study participants shared these upfront expectations. A great majority of the respondents generally confirmed (or strongly confirmed) that they enjoyed the taught practices in the first place. If asked whether developers considered ASET beneficial, the overall agreement was even higher than the enjoyment rate (cf. Figure 2). One respondent described the trainings as "finally something tangibly helpful for my daily work after Scrum and lean". In a hypothetical own company, a great majority would adopt these techniques. Overall, no group of strict opponents could be found.

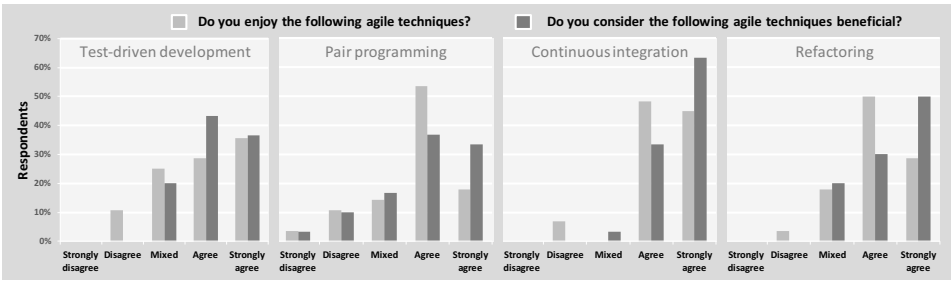

Figure 2 Respondents' enjoyment and conviction of the studied agile techniques

4. Heterogeneous Adoption Patterns

Despite the high level of agreement regarding the benefits and enjoyment of agile techniques, our analysis revealed various unexpected patterns which indicated variations in the degree of the adoption of each agile technique. In particular and contrary to our expectations, our survey results clearly indicate considerable variations among teams intensity adopting pair programming or test-driven development. Table 2 demonstrates these variations with team-based average values for the PP-related question "How much of your development time do you program in a pair?" and "What percentage of code do you write in a test-first manner?" The varying adoption intensity patterns are somewhat surprising, since all teams had participated in the same training program after which they were free to adopt the learned techniques.

	Adoption intensity					
	PP		TDD		REF	CI
	Interview result	*Average development time used for PP*	*Interview result*	*Average of new code written in a test-driven mode*	*Average development time used for REF*	*Average time between code checkins [days]*
Team 1	Low	18%	Low	23%	18%	<2
Team 2	Low	36%	Varying	58%	26%	<1
Team 3	High	60%	High	71%	36%	<1
Team 4	High	76%	High	62%	36%	<2

Table 2 Teams' intensity of adopting PP, TDD, REF, and CI

In-depth insights from our interviews corroborate our survey findings on the heterogeneous adoption patterns among different teams (cf. qualitative indications In Table 2). Moreover, our interview results reveal formerly unknown team-specific factors which help to explain and understand the varying adoption patterns. These factors accentuate respective advantages and disadvantages of the individual techniques and uncover additional contingency factors which hinder or diminish the expected adoption. These will be discussed in the following sections.

4.1. Variations in Pair Programming Adoption Intensity

While programming in a pair, two developers of a team work at one computer, sharing a single keyboard and a mouse for their development work. The partners frequently alternate their active driver and passive navigator roles. Two of the four studied teams applied pair programming to a high degree of their development time with a partner (high adoption intensity), while the remaining teams used the technique occasionally for specific aspects of their work only (low adoption intensity).

According to this pattern, Team 1 (cf. Table 2) is categorized as a team with low adoption intensity. Team members paired if "the work gets more complex or if we see it as helpful to share knowledge". The technique was described as beneficial for security relevant tasks to mitigate the risk of defective code modifications. The pronounced use of a code review system explains the low intensity. Every code change was reviewed in order to achieve a 100% review level. The team regarded code reviewing as superior to pair programming, as knowledge and ideas discussed in a PP session remained mostly within that pair, but not documented. However, pair programming was considered a complementary technique rather than a substitute as the low adoption intensity might suggest. Developers acknowledged their partners' fast feedback and the possibility to jointly solve a coding task, which reportedly increased the team cohesion. The team mentioned several impediments for coding in pairs: potential programming partners from the open-source community worked at different locations and time-zones. Secondly, the team considered the required synchronization within team-internal pairs as a burden for its productivity. This was perceived as particularly unpleasant, since some team members preferred flexible work hours.

For some developers of Team 2, PP was the default mode, but most members only used it parsimoniously (see Table 2). The team attributed the creation of a broader knowledge base among individuals to pair programming which led to higher project flexibility. In particular, the reduction of the "vacation problem" was regarded as valuable, i.e. the team was still productive if individuals were absent. Despite these respected benefits, reported disadvantages out-weighed the advantages for the team in particular situations explaining the variation in this adoption pattern. One developer said he felt stifled in his work flow leading to more stress for him than working alone. Unequivocally, the team shared the opinion to have delivered less features per Scrum sprint.

For Team 3, pair programming was the default work mode. In the planning session, the team decided which tasks to develop in pairs, with coding of new functionality always being developed by two programmers. Still a few disadvantages of pair programming were mentioned as it "can cause conflicts in the team; but the team has to learn how to deal with that". One developer reported to sometimes be "overambitious and to forget to take breaks" which can be more stressful than working alone, since you are busy all the time. Generally, advantages clearly out-weighed the disadvantages for this team. Members emphasized a stimulated discussion culture leading to common quality awareness, and the immediate avoidance of trivial errors as the main benefits.

Team 4's intensity of paired programming was the highest. Almost every coding task was developed by a pair and even non-coding tasks, such as configurations of productive systems, were realized in teams of two: "we try to develop everything possible with a partner". Developers mentioned that they have more fun at work and a better team spirit. The technique was appreciated because of its good fit into the adopted Scrum framework due to its positive effects on teamwork and a common code responsibility. Finally, pair programming reportedly fostered the intra-team discussion culture.

In summary, several impeding factors were discussed which helped to explain low adoption intensity rates. Pair programming is only possible if teams are co-located, if teams share work tasks similar in terms of content, if developers are willing to synchronize their work hours, and for pairing partners who bring the potential and willingness to reach consensus.

Coding with a partner entails three major benefits: a broader common knowledge base leading to increased team flexibility, prevention of trivial errors from the very beginning of coding due to the immediate feedback and increased quality awareness, and finally a closer collaboration among team members fostering team spirit. On the other hand, pair programming was reported to potentially lead to inter-personal conflicts, to be more stressful than working alone, and to interrupt a free flow of thoughts. Some developers reported to perceive a lower development speed of the team due to pairing, e.g. measured as delivered features per Scrum sprint. In general it can be said that through pair programming, the aspect of continuous value generation can be brought to the teams and their daily work.

4.2. Variations in Test-driven Development Adoption Intensity

Test-driven development implies alternate writing of production code and corresponding tests to cover its functionality. In the strict sense, tests are written before the code exists. Consequently, tests fail until the software functionality is implemented consistently. Hence, productive software and its test framework evolve simultaneously.

Among the two low adopting teams of TDD, Team 2's usage can be described as varying due to different subgroups resulting from the team's three technology stacks. In both teams, developers mentioned that "we do not apply TDD consistently; perhaps we use it more in the lower levels of the software stack, parts which have a library character". The TDD process felt unnatural, as stated by one interviewee, "these micro-increments do not come naturally for the seasoned developer. It feels very strange". In general, the consensus on the decisive factor was the focus on tests which resulted in high code coverage.

The benefit of TDD was recognizable "when implementing logic in code which is not straight forward". As Team 1 worked on many user interface development tasks, they described the technique as "not making any sense" since the effort was felt to be disproportionately higher compared to writing a test after the code had stabilized. "Test code can get very complex" as one developer mentioned, "we have had test code which was far more complex than the actual productive code". Nevertheless, in areas where the team decided to use TDD, the increased modularity in the design of the code was evident. Furthermore, the technique increased developers' focus on their current task at hand, as the test suite gave developers more self-confidence and "trust in my own work that lets me sleep well at night". TDD was also viewed as a possible cause of slower development speeds which needed to be factored in at the planning stage. Some programmers felt that "it disturbs the flow of thinking; it is too much back and forth". For others the increasingly small increments between test and productive code writing "are annoying" especially "if I know exactly what needs to be done, then I have to force myself to use TDD".

More senior developers noted the painful transition to TDD, as this programming style required a major change in the thinking process. One impediment in the beginning was the high feature pressure from outside the team which "increased stress on the team members", so that they "couldn't use TDD in a reasonable manner". The technique had been impaired by the large legacy code base the team had to deal with, "it basically has no code coverage and was not written to be easily testable".

In the group of high adopters, TDD was used as "often as possible" with development tasks, although many items were non-coding activities. However, TDD was often not used in the strict sense of minimal steps between writing a test and implementing the productive code, but in a test first fashion. Essentially the steps were enlarged so that fewer context switches between testing and coding were necessary. This can be illustrated by one respondent's answer, "I don't develop tests line by line; that is too strict for me".

A strong advantage of TDD was the confidence brought about by the test-suite. Through this "tightly-knit security net, you can program in a very relaxed manner. One has to keep less open ends in mind". Many interviewees had a high conviction towards agile techniques and their quality of work, with several developers mentioning that "features take more time to implement, but we want to deliver it in high quality". Developers stated that they "can only implement new functionality in existing code, if it is somewhat structured", but this structuredness can only be acquired if they are given "enough time". Moreover, TDD was also regarded as introducing structure into the idea creation process and it was reportedly motivating to develop against the red light of a non-passing test.

One major advantage of test-driven development was recognized in the large test framework produced. This high code coverage let developers "reach easily into existing code and completely refactor certain areas while still being sure that the functionality is implemented correctly". The long term maintainability of the software code was a critical aspect mentioned, "I'm not afraid to touch three year old code, make a change and let the test-suite run to tell me if everything is still working as before".

4.3. Homogeneous Adoption Pattern regarding Continuous Integration and Refactoring

All studied teams continuously integrated new or modified code into their existing code base, as proposed by the literature. CI was described as not helpful on its own, but particularly beneficial in combination with a high test coverage of the code or TDD. One developer said that "continuous integration helps you to permanently get feedback on your code due to the automatic checks against the existing tests every time you integrate [...] we know after each check-in, if the product is shippable, thus we are theoretically ready to release at any time". Another developer expressed his positive attitude as follows: "from continuous integration, I do not see any disadvantages at all". Still another illustrated that "developing software without CI is like flying blind without instruments [...] you just do not know what you are doing, where you are and how much time it will cost to repair something [...] there is no feedback and no transparency". These quotes perfectly represent the teams' unrestrained positive attitude towards the technique.

Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [FB99]. Our survey results reveal that more than 80% of the respondents spent between 20 and 40% of their time refactoring existing code. Nevertheless, members of all teams finished up to 29% of their coding backlog items without refactoring. Team 1, which used the code review tool systematically, reported always paying "attention to make the code design right from the beginning" since messy code would not be accepted in the review anyways.

5. Combining ASET in a Lean and Scrum context to its full benefits

Developers who consider lean principles and Scrum as too abstract find agile techniques to be well suited to transfer and apply the lean concepts into their daily work. Hence, we consider agile techniques as the key to successfully develop software in such a context since they are anything but abstract. We found that ASET were the "missing link" to bring lean to the value stream (cp. also [CC08]).

Building upon a firm base of lean principles, the combination of the studied agile techniques revealed interesting interdependencies between the techniques applied. For instance, developers recognized their PP partner as their personified "bad conscience". Consequently, they felt encouraged to integrate their software more often, to refactor more intensively, and to adhere to the test-driven work mode, even though it was described as time-consuming and costly in the short-term, but admittedly beneficial in the long-run. In particular, combining TDD and CI was considered more advantageous since test-driven development naturally leads to an extensive test-suite making continuous code integrations more attractive.

Our interview results clearly indicate that these agile techniques breathe life into many abstract lean principles and Scrum basics. One developer mentioned that "PP fits perfectly to Scrum and its focus on team empowerment". As discussed before, continuous integration in combination with an extensive test suite enables developers to release their software at any time. Moreover, it increases transparency and allows all team members to be up to date on the current status of their software product. Striking similarities can be seen when compared to [PP03], as both aspects are directly related to the lean principle of "optimizing the whole". Furthermore, developers recognized lean's focus on the "elimination of waste" by adopting continuous refactoring in their daily work. Due to the paired work mode, developers stated to "build quality in" by avoiding trivial errors right from the beginning and to raise a common quality awareness in the team. Moreover, pairing invites developers to "learn constantly" while "engaging everyone" in the team.

Several benefits from adopting agile techniques help development teams to cope with issues prominently found in large scale software development organizations. In such environments, many people work simultaneously on a common code base which can accumulate over several years or decades. Hence, developers face an ever growing technology complexity. The combined adoption of TDD, CI and REF was regarded as one way to tackle this problem; developers attributed a higher modularization of their code to TDD, continuous refactoring helped reinforce coding conventions and thus structuredness and legibility. Consequently, developers can more easily modify the existing code base even if it was previously written by somebody else.

Large scale software development often leads to a certain customer disconnect because of complex organizational setups. TDD and CI introduce fast feedback cycles through a continuous alignment with customer requirements ideally embodied in the test framework. Thus, it fosters a close customer focus which is the prerequisite for delivering value despite an increasing systems complexity.

We conclude that the introduction of agile techniques into a large-scale lean software organization helps developers translate this new mindset into their daily work. For developers, lean and agile is no longer an abstract initiative from management, but a tangible asset which clearly changes the work of every software developer.

6. Recommendations for Large-Scale Lean Implementations

As previous studies have shown, PP, TDD, CI, and REF help developers deliver better software quality. But, there is no free lunch: the improved software quality is clearly traded for a prolonged short-term development time. However, we recommend not to one-sidedly assess gains in software quality in proportion to longer development time, but to equally consider composite effects brought about by applying agile techniques in lean and Scrum contexts. Benefits, such as developers' closer intra-team collaboration, increased common quality awareness or their intensified knowledge transfer, are particularly beneficial in long-term oriented, quality-conscious software development organizations rather than in feature quantity focused ones. For the future, we recommend to balance these trade-offs more consciously according to the requirements of each individual project.

While introducing agile techniques in development teams, you need to take certain contingency factors into account. Developers' personal convictions of the studied techniques predicted their actual adoption the best. Do not expect developers to naturally see long-term benefits, since short-term efforts and change issues can initially seem insurmountable.

If teams can deliberately decide whether and how to adopt agile techniques, as managers among our interviewees recommend, then management should focus on encouraging developers to jointly aspire towards organizational long-term goals, thereby motivating ASET. Training sessions alone are not self-sufficient. Organizations which want to implement lean including ASET need to take the substantial effort for designing technology-specific trainings as well as hands-on coaching into account.

We conclude that sustainable competitive advantage of modern software companies does indeed not only depend on the ability to continuously innovate new products, but also requires the capability to establish organizations and processes to develop, extend and maintain these products at competitive quality, cost and time. Combining lean, Scrum, and ASET helps software companies accomplish exactly that goal.

We are convinced that lean software development in combination with agile techniques offers a major step into the future of software engineering as it enables software companies to adapt quickly to changing environments while focusing on value creation and quality. Future studies should focus around the question of what makes software development teams successful in an agile and lean environment. Furthermore, the investigation of driving factors behind software development process flow across several development teams and organizational levels seem particularly promising. Current trends in the software industry also revolve around the question on how to combine innovative

product design and lean development, i.e. how inspiration and innovative ideas for new software products and features happen in an lean and agile development environment (see [HM12] for a first case study).

Bibliography

[Be01] Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 2001.

[CC08] Chow, T.; Cao, D.-B.: A survey study of critical success factors in agile software projects. In: Journal of Systems and Software, Vol. 81, Issue 6, pp. 961-971

[Co09] Conboy, K.: Agility from First Principles: Reconstructing the Concept of Agility in Information Systems Development. In Information Systems Research, 2009, 20; pp. 329–354.

[CW01] Cockburn, A.; Williams, L.: The costs and benefits of pair programming. In Extreme programming examined, 2001; pp. 223–248.

[DB11] Dogša, T.; Batič, D.: The effectiveness of test-driven development: an industrial case study. In Software Quality Journal, 2011; pp. 1–19.

[Dy07] Dybå, T. et al.: Are Two Heads Better than One? On the Effectiveness of Pair Programming. In Software, IEEE, 2007, 24; pp. 12–15.

[FB99] Fowler, M.; Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[HM12] Hildenbrand, T.; Meyer, J.: Intertwining Lean and Design Thinking – Software Product Development from Empathy to Shipment. In: Software for People – Fundamentals, Trends and Best Practices, 2012, pp. 217-237, Springer, Heidelberg

[MT04] Mens, T.; Tourwé, T.: A survey of software refactoring. In Software Engineering, IEEE Transactions on, 2004, 30; pp. 126–139.

[MW03] Maximilien, E.M.; Williams, L. Eds.: Assessing test-driven development at IBM, 2003.

[Na08] Nagappan, N. et al.: Realizing quality improvement through test driven development: results and experiences of four industrial teams. In Empirical Software Engineering, 2008, 13; pp. 289–302.

[PP03] Poppendieck, M.; Poppendieck, T.: Lean software development: an agile toolkit. Addison-Wesley Professional, 2003.

[Re09] Reinertsen, D.: The Principles of Product Development Flow - Second Generation Lean Product Development, 2009, Celeritas Publishing.

[SB01] Schwaber, K.; Beedle, M.: Agile Software Development with Scrum. Prentice Hall, 2001.

[SS11] Schwaber, K.; Sutherland, J.: The Scrum Guide, 2011.

[WCC12] Wang, X.; Conboy, K.; Cawley, O.: “Leagile” software development. An experience report analysis of the application of lean approaches in agile software development. In The Journal of Systems & Software, 2012, 85; pp. 1287–1299.

[WJ03] Womack, J. P.; Jones, D. T.: Lean thinking: banish waste and create wealth in your corporation. Simon and Schuster, 2003.