A PARADOXICAL PERSPECTIVE ON CONTRADICTIONS IN AGILE SOFTWARE DEVELOPMENT

Wang, Xiaofeng, Lero, The Irish Software Engineering Research Centre, Limerick, Ireland, xiaofeng.wang@lero.ie

Ó Conchúir, Eoin, Lero, The Irish Software Engineering Research Centre, Limerick, Ireland, eoin.oconchuir@lero.ie

Vidgen, Richard, University of Bath, Bath, BA2 7AY, UK, r.t.vidgen@bath.ac.uk

Abstract

An ongoing debate on agile methods focuses on the contradictions in software development, especially responding to change vs. following a plan, and people vs. processes. Unlike the 'either-or' perspective adopted in the existing agile literature, this paper introduces a paradoxical view on the contradictions in agile software development and uses two agile processes to illustrate it, arguing that a paradoxical perspective can help to gain a better understanding of the nature of and ways of dealing with the contradictions in agile software development. Taking a paradoxical perspective on responding to change vs. following a plan, and people vs. processes, this paper reveals that an agile process is a planning-driven process geared to responding to change, and it is a process that provides a supporting structure for people to learn and to improve their competences.

Keywords: Contradiction, Paradox, Agile methods, Planning, Process.

1 INTRODUCTION

Agile software development methods, such as eXtreme Programming (XP) (Beck 1999) and Scrum (Schwaber and Beedle 2002), are a response to the inefficiency of existing software development methods in rapidly changing environments (Highsmith 2002). They have gained new momentum through the promotion of the Agile Alliance and the publication of the Agile Manifesto which lays out a set of values and principles in support of agile methods (Agile Manifesto 2001). However, the Agile Manifesto and agile methods have evoked a vivid debate which dwells especially on two contradictions: responding to change vs. following a plan, and people vs. processes. Despite the manifesto claims that "there is value in the items on the right" (Agile Manifesto 2001), the existing agile literature mainly adopts an 'either-or' perspective, trying to emphasize the value in the items on the left at the cost of the right ones.

In this paper a paradoxical view on the two contradictions is argued for by drawing on a conceptual framework of dealing with contradictions informed by Stacey (2003) and Poole and Van de Ven (1989). To illustrate this perspective, the software development processes of two teams using XP have been described and analysed. The objective of the paper is to gain a better understanding of the nature of, and ways of dealing with, the contradictions in agile software development using the paradoxical perspective.

To this end, the paper is organized as follows. The next section reviews how contradictions are viewed and dealt with in agile literature. The different ways of dealing with contradictions in general are presented in Section 3 and then followed by a reflection of the literature introduced in the previous section to reveal a lack of, and an argument for, a paradoxical view in agile literature. The paradoxical view for understanding the contradictions in agile processes is illustrated by two cases and discussed in more detail in the following two sections. The last section reflects on the limitations of the study and potential future work.

2 CONTRADICTIONS IN AGILE SOFTWARE DEVELOPMENT

"Agile" is the name adopted by the Agile Alliance for the set of software development methods that were initially called "light-weight" methods. The Agile Alliance defined the Agile Manifesto in 2001, which states a set of values and principles behind these methods (Agile Manifesto 2001). The four agile values are specified as follows:

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

While admitting that there is value in the items on the right, advocates of agile methods give greater value to the items on the left.

The Agile Manifesto and agile methods have provoked a vivid and often controversial exchange of opinions, both vigorous criticism and equally enthusiastic support in the software community. Rakitin (2001) argues that process, documentation, user contract and plan are essential in software development, whereas agile values, such as people interaction and responding to change, reflect a hacker culture which allows people to irresponsibly throw together code with no respect for engineering discipline. In an online newsletter, Rakitin (2005) questions the Agile Manifesto: "We have been working so hard to improve the perception of software engineering and instil discipline in developers and managers. And then this…?" Even though Rakitin believes agile methods may work well under certain circumstances, he notes: "Agile methods can be effective but so can other methods". For example, Rakitin (2005) believes that where software is being developed for use solely

within the company, the risk of failure is low, the project team is highly motivated and led by an excellent leader, or long term maintenance of software is not a concern, both agile and traditional methods can be effective. He also argues for the necessity of documentation: "With no design documentation or requirements to refer to, you're left with only the code. I contend that even incomplete or outdated design documentation is better than no documentation. How many times have you seen cases where maintenance is so difficult because of a lack of design documentation that we decide it would be easier to start over than to figure out how the existing code works?" In the same vein, Stephens and Rosenberg (2003) dismiss the existence of an agile method and claim that it is something that developers can only aspire to, and is only determined by hindsight. They attempt to restore the values of documentation, and upfront planning and design in software development. Glass (2001) argues that more emphasis must be put on the maintenance documentation, especially in highly evolutionary development, where emphasizing working software over documentation would counteract iterative and evolutionary development. Fruhling and De Vreede (2006) consider agile methods to be unstructured, unpredictable, and neglecting planning. Kalermo and Rissanen (2002) argue that agility is derived to a large extent from individuals and their competences and tacit knowledge, which sets high expectations for the developers and also for the managers, and the applicability of agile methods can be limited if relying on such a foundation. The criticism of agile methods has put people, working software and responding to change on the opposite ends of process, structure and plan.

Agile advocates claim that agile methods reflect a different assumption about the environment of software development than traditional methods, that is, change will always happen during the project life span, and "because we cannot eliminate these changes, driving down the cost of responding to them is the only viable strategy" (Highsmith and Cockburn 2001, p. 120). They also claim that what is new about agile methods is not the practices they use, but their recognition of people as the primary drivers of project success, coupled with an intense focus on effectiveness and maneuverability. Highsmith and Cockburn (2001), however, do not deny the values of documentation, process, tools and plans. They also stress the importance of retaining quality, and clarify that the reason agile methods are sometimes confused with ad hoc or "cowboy" coding is because "the design is done on an ongoing basis, in smaller chunks, as opposed to all at once and up front" (p. 120). In response to the opinion that agile methods lack discipline, in the article "Agility through discipline" (Beck and Boehm 2003), Beck claims that "agility or discipline" looks like false dichotomies, and agility is only possible through greater discipline on the part of everyone involved. He suggests that the only way to achieve the results we seek is to view the world in "both-and" rather than "either-or" terms. By describing what he means by discipline in the context of agile software development, he advocates that XP practices "all contribute to 'perfecting' mental faculties" of the team (p. 44), the team members know what to expect of each other, so the team's behavior is orderly, and XP teams are conscious of their collective rules for planning, development, integration, and deployment. "Given a broader view of discipline, Extreme Programming is far more disciplined than most processes, providing a clearer collective picture of what activities are expected and more opportunities for learning. Indeed, without conforming to these positive senses of 'discipline', the social contract of Extreme Programming would rapidly disintegrate" (Beck and Boehm 2003, p. 44).

Boehm (2002) proposes a balanced view on both agile and plan-driven approaches, and suggests reconciling the two, believing that synthesizing them can provide developers with a comprehensive spectrum of tools and options. Along this line of thinking, Boehm and Turner (2003) examine the agile versus plan-driven debate and provide recommendations for how and when to mix the two approaches. They highlight both the home grounds and risks associated with each approach, and describe how they would choose which approach to use and under what circumstances they would mix them. Meanwhile, the possibilities of reconciling XP with CMMI (Capability Maturity Model Integration) have also been explored in Paulk (2001), Turner (2002) and Lycett et al. (2003). These studies offer their own resolutions of the agile versus engineering debate, and claim that agile methods are not against engineering practice. If used thoughtfully, they provide a clear mandate for making engineering practice lean and well focused. Vinekar et al. (2006) provide some evidence of agile and waterfall

methods coexisting in organizations. They find emerging evidence indicating that most organizations are attempting to utilize both. Based on this observation, they suggest adopting an organizational form that may enable this duality. Drawing on the extensive literature in organizational theory and management, they advocate ambidexterity as a viable solution. Through an ambidextrous organizational structure, systems development organizations can reap the benefits of both agile and traditional systems development.

In summary, the debate on agile methods represents three interpretations of the Agile Manifesto: firstly, as a set of contradictory or mutually exclusive values; secondly, as a 'both-and' co-presence rather than an 'either-or'; and thirdly, as a spectrum that combines and compromises, to various extents, the different values. These three views reflect different ways of understanding and dealing with the agile contradictions by the software community. No consensus understanding has been achieved. Based on this observation, it is argued that a deeper understanding of how contradiction can be dealt with in general may lead to a better understanding and therefore new ways of dealing with contradictions in agile software development.

3 CONCEPTUALIZATION OF CONTRADICTIONS

3.1 Different ways of dealing with contradictions

The definition of contradiction adopted in this study is "a state of opposition in things compared" (Oxford dictionary). For example, a 'round square' is a contradiction in terms. Stacey (2003) provides a number of different ways in which contradictions encountered in people's thinking can be dealt with. A contradiction in terms of two contradicting themes A and B, can be posed in terms of 'either-or' when viewed as a dichotomy or dilemma, while perceiving it as a duality or paradox embodying a 'both-and' approach. Poole and Van de Ven (1989) elaborate on the 'both-and' approach drawing from an extensive collection of literature of sociology and psychology, and add a 'synthesis' view on contradictions. Table 1 is a summary of different ways of dealing with a contradiction based on Stacey (2003) and Poole and Van de Ven (1989).

Approaches to deal with contradicting themes A and B		Stacey 2003	Poole and Van de Ven 1989
Either-or	Dichotomy	Polarised opposition of A and B which requires an 'either-or' choice.	-----
	Dilemma	An 'either-or' choice between two equally unattractive alternatives.	
Synthesis		-----	o A new perspective is sought in order to eliminate the opposition between A and B.
Both-and	Dualism	Keeping both A and B but locating them in different spaces or times.	o A and B are situated at two different levels or locations in the social world, and as such they are spatially separated by clarifying the levels of analysis; or o A and B are separated temporally in the same location, which takes the dimension of time into account.
	Paradox	A state in which A and B are simultaneously present; neither can be resolved or eliminated.	o A and B are kept separate, and their contrasts appreciated, from which the paradox is used constructively.

Table 1. Three ways of dealing with contradictions.

Using Table 1 as a lens to re-examine the debate described in Section 2, it becomes clear that the criticism of agile methods reveal an 'either-or' view of the contradictions introduced in the Agile Manifesto. Agile values cannot co-exist with the values of traditional software development (e.g., Rakitin 2001, 2005, Stephens and Rosenberg 2003). Meanwhile, the voices of agile proponents are not unanimous. Although a 'both-and' view is advocated by Beck (Beck and Boehm 2003), many agile advocates also consider the agile and plan-driven software development methods polar opposites (Boehm 2002, Baskerville 2006). The reconciliation approach reflects a synthesis or even dualist view on the contradictions in agile software development. The paradoxical view, however, has not been adopted in agile literature.

3.2 A paradoxical view on agile software development

A paradoxical view - the notion that paradoxes can never be resolved, only lived with – leads to a view of organizational dynamics couched in terms of continuing tension-generating behaviour patterns. Fiol (2002) argues that capitalizing on a paradox means utilizing the inherent tensions to one's advantage rather than ignoring or resolving them. The benefit of a paradoxical view is that it allows one to discover different assumptions, shift perspectives, pose problems in fundamentally different ways, focus on different research questions, and come up with answers that stretch the bounds of current thinking.

An example of constructively using paradox can be found in Streatfield (2001) and Vidgen et al. (2004). Streatfield (2001) presents a view of management as a contradiction of 'in control' and "not in control" at the same time. "In control" means selecting, designing, planning a course of action, correcting deviation, working in a stable environment with regular patterns, conformity, and consensus forming, and "not in control" means seeing action as evoked, provoked, emerging, amplifying deviation, and an unstable and unpredictable environment with diversity and conflict. Vidgen et al. (2004) argue that, rather than avoid conflict and tension and accept one or other of these poles, managers must work with the paradox of control. They should accept and even embrace the paradox of control, i.e., they are simultaneously 'in control' and 'not in control' and need the courage to live with the resulting anxiety.

This study applies a paradoxical view on two contradictions in the Agile Manifesto: responding to change vs. following a plan, and individuals and interactions vs. processes and tools which is narrowed down to people vs. processes due to the social and organizational focus of the study. According to Table 1, the paradoxical view of the two contradictions is framed as follows:

- Responding to change vs. following a plan: responding to change and following a plan are coexistent simultaneously in the same development process
- People vs. processes: both people and process are important components and should be taken care of simultaneously in the same development process

In the next section two cases are used to illustrate the paradoxical view on these two contradictions in agile software development. It is worth emphasizing that, in terms of the two contradictions, both cases are considered exemplars of dealing with them constructively using a paradoxical view, even though the specific practices that are use differ in the two cases.

4 A PARADOXICAL VIEW ON TWO AGILE PROCESSES

This study is exploratory in nature, with the intention of investigating contemporary phenomenon in a real-life context. An interpretive multiple-case study is considered a suitable research approach (Yin 2003). Each case is a software development team who have adopted a set of agile practices from XP and have used them for some time (names used are pseudonyms). Data regarding the agile process of each team is collected through semi-structured interviews using open-ended questions with the team

members, including both project managers and developers. The data analysis process follows a sequence of preliminary coding, within-case analysis and cross-case comparison.

4.1 Background to the cases

Floresoft is a software development team in a small software house which is specialized in network security and management systems development. The team has more than five years of experience of XP and successfully completed several projects using it. The team has four members: the project manager, who is also at the management level of the company and has many years of experience in system analysis but limited experience in programming; the XP coach, who is also a developer; and two developers. All projects are application development for specific customers.

LicenseMan is a software development team in a major IT company providing both IT projects and services. The team has more than two years of experience using XP. It is composed of one project manager, six developers, one test manager and one onsite customer, collocated in the same lab area. The project manager also plays the role of XP coach. Among the six developers there is one team lead who concentrates on the development process and is responsible for the direction of resources, and one tech lead who is knowledgeable on the entire system and responsible for design and technical mentoring of the team members. The test manager works closely with people from the business side, responsible for managing the test cycle for the system, including development of schedules and test plans and coordination of test execution and implementation. The onsite customer represents the business unit that is financing the project, responsible for definition and prioritisation of requirements.

4.2 Responding to change vs. following a plan

Floresoft uses an iterative process model. An iteration is as short as a week. The team has experimented with longer iterations of one month and shorter ones of two or three days, and came to the one-week length. They feel it is a good pace for them and try to stick to the fixed length whenever possible. An iteration is viewed as a contract between the team and their customers. Generally no change is introduced into the user stories under development during an iteration. The customers can check the progress of the development anytime they want, or clarify the understanding of the user stories, but can only change the stories when an iteration is completed. Although preferring one-week iterations, instead of sticking to the length rigidly, the team does change iteration length according to different projects, different stages of a project, different frequency of communication with the customers, and different sets of user stories they have to implement. The team considers that user requirements gathering is an evolving process and the understanding of user requirements is an ability to be learnt. They do not assume that they can gather and understand all requirements at the beginning of a project - user requirements are constantly captured throughout the life span of a project. The team delivers software in an incremental way each week and the delivered software is tested by the customers. In this way, the team can have early and frequent feedback from the customers.

Meantime, the team plans frequently, and sees planning as a natural step following frequent external and internal feedbacks. But the team plans in detail only for a short period. They plan weekly for an iteration; and they plan daily for a working day. While planning, the team uses half an hour as the unit of measure for the estimates of the work. The average estimate of a piece of work is 12 to 15 hours. The capacity of the team to work in an iteration or a day is compared with the sum of the estimates of the work in that iteration or day. This gives the customers and the team an idea of which and how much work can be implemented, and the customers prioritise the work according to the "greatest value to the customers first" principle. Although always planning, the team has a practical attitude towards plans. They are prepared to change them whenever needed through prioritisation of tasks, which helps them make a quick decision on what to drop when circumstances arise and adjust plans accordingly.

In the case of LicenseMan, the team uses two-week iterations, and generally does not change the iteration length. In principle, user stories are not allowed to change during an iteration, but the team

does adjust them according to the suggestions of the onsite customer. The team requested, as suggested by XP, an internal customer from the business unit to join the team and sit together with the developers. The onsite customer is involved completely in every step of the development. Since she and the team are collocated, there is constant communication and interaction between them. The onsite customer gets the opportunity and flexibility every two weeks to steer the release. She not only gives feedback about the development to the team, but also gets feedback from the team about user requirements and the system on a daily basis.

Meanwhile, planning happens at three levels in the process of LicenseMan: release, iteration and daily. Release planning provides a high-level outline of the project. Instead, iteration planning goes very deep and in detail for every two weeks. Daily planning happens at the stand-up meeting of the day. While planning, LicenseMan uses "a perfect engineering day", i.e., eight hours devoted to the problem at hand with no interruptions, to estimate efforts required for implementing a piece of work, which can be as small as 1/4, but no bigger than four, perfect engineering days. The team generally builds in a given slack time in estimates to allow for unknowns so that the team is not under the pressure to get story done as quickly as possible, rather, they can work to get quality code. They realize that estimation is always "*gonna be a little bit guess work*" (Team lead), though the ability to estimate properly would be improved over time.

Table 2 summarizes the practices regarding responding to change and following a plan in the two cases.

	Floresoft	LicenseMan
Practices regarding responding to change and following a plan	• Short one-week iterations, no requirement change during an iteration; Ongoing requirements gathering; Regular, frequent and incremental delivery • Planning at iteration and daily levels, detailed plans for short terms; Adjusting plans constantly through task prioritisation	• Short two-week iterations; Collocation of the customer and the development team and close interaction between them • Planning at release, iteration and daily levels, detailed planning for short terms

Table 2. Practices regarding responding to change and following a plan in the two cases.

As shown in the two cases, the processes of Floresoft and LicenseMan are paced by fixed-length iterations, one week and two weeks respectively. The work within an iteration is generally self-contained. Both teams prefer fixed-length iterations which brings a short-term certainty to the team, as the following comment describes:

> *"You have a regular heartbeat of the project, such as every 2 weeks, every 2 weeks, every 2 weeks." (Project manager/LicenseMan)*

The iterative model that the teams adopt to enable them to respond to change enables the teams to think through what they should do for a short time period and thus provides a good basis for planning. Both teams can plan in a fairly accurate manner for short terms. Meanwhile, frequent planning allows the teams to constantly respond and react to the uncertainty inherent in the project, as the following comment suggests:

> *"The team have to know when necessary to spend one minute day to day to plan, five minutes to change direction." (Project manager/Floresoft)*

Frequent planning is a response to the evolving user requirements and other uncertainties the teams using the agile processes have to confront all the time. They do follow the plan they come up with, but meanwhile frequently adjust their plans through task prioritisation. A rationale behind it is that, as Floresoft realizes, planning is a learnt ability, which is improved in the development process with the ability of a team to understand the system, situations, problems, etc. With iterations and frequent planning, both teams are able to obtain fairly accurate estimates of their work.

4.3 People vs. processes

The Floresoft team realizes that management is eventually an internal process of the team, because only the team members who work together and share the same working context can provide effective feedbacks, interpret them in a sensible way, and take suitable action subsequently. The control of the process should be in the hands of the team, not someone living outside the team. The same reasoning applies to the role of the XP coach. Even though one team member assumes the role of the coach, who ensures the smooth progress of the development process, both he and the other team members believe that everyone in the team is a coach. To implement self-management, the team members are constantly attentive of what happens to the other members and in the environment. An open work space design the team adopts makes this observation possible. To effectively observe each other, however, the team members realize that they need to be able to self-observe firstly to stimulate self-responsibility which is necessary for devolving management into the internal process of the team.

Floresoft team members are involved in all development activities of a project, and all have to deal with customers, analyse user requirements and write code, together. There are no traditional roles such as system analyst, designer, programmer or tester in Floresoft. Meanwhile, the team members sign up tasks they would like to implement during the day and take ownership of them. Neither the project manager nor the coach assigns tasks. Generally the developers choose tasks they feel confident in completing, but they also pick up tasks that they are not good at, and then look for help at the daily planning time and work with a more skilled member through pair programming, in order to acquire new skills.

Meanwhile, Floresoft also put focus on improving their development process. The team does a retrospective on the process in the feedback session that starts a working day, to adjust it according to the up-to-dated context and the emergent issues, and thus to keep the process flexible and responsive. One thing the team realizes is the importance of feedback on the positive aspects of the previous day, which can provide them with satisfaction and keep them motivated.

In the case of LicenseMan, the team self-manages to a certain degree. The project management load is distributed, such as making sure that documents are signed up, or the working schedule is balanced among team members. Typically a lot of initial technical decisions are made at the group level. Two or three developers go and explore issues and come up with solutions. The team members decide when to do pair programming and when not to do it. And when they do pair, they pair with each other in a self-managed way, without the meditation of the project manager or the leads. They also self-assign tasks by picking up the tasks they would like to work on and take ownership of them. No task assignment is from the project manager or the team lead. It is recognized by the team, nevertheless, the necessity of someone taking care of the process when things do not go well. For example, the team lead sometimes asks around whether anyone wants to do a "boring" user story, or the tech lead needs to facilitate the pairing if no pair programming is going on for quite a while. But the management is more done through “nudging” at a day-to-day level.

When self-assigning tasks, the developers generally tend to pick up something they are interested in doing, or something new and challenge to do. If a task is challenge enough and the developer feels that he is not able to implement it alone, he can raise this issue at the daily stand-up meeting and ask for help from the rest of the team, then pairs with another developer who has the relevant knowledge and competence. The team recognizes the communication function of formal meetings such as daily stand-up meetings. Since the team does not do pair programming and pair rotation on a regular basis, to compensate for this fact, the stand-up meeting generally lasts longer to allow the entire team to be updated. The team also holds regular retrospective meetings to review their development process.

Table 3 summarizes the practices regarding people and processes in the two cases.

As shown in the two cases, the developers of both teams have a large degree of autonomy and carry out their activities in a self-managed manner. Both teams emphasize the importance of people in

software development. They rely on the technical expertise of the developers which plays a significant role in effectively implementing most agile practices. The developers who have proper expertise, as Floresoft suggests, also have a sensibility of their limits and can recognize the moments when they need help from others. Relying on people also means motivating people. The developers are working much better if they are working on things they are interested in doing and if they can learn new things from doing them, as agreed by both cases. Both teams involve the developers in all activities. It helps them evolve their competences and gives them a sense of satisfaction with their work, which in turn leads to an autonomous team which can work on any aspects of the development.

	Floresoft	LicenseMan
Practices regarding people and processes	• Everyone assumes the responsibility of project manager and coach; Self-management through peer-observing and self-observing, facilitated by open work space • Total team involvement; no separation of functional roles • Task self-assignment supported by daily steering and pair programming • Daily feedback session on the process of the previous day	• Team self-manage to a certain degree, optimized by micro-level nudging from the project manager and the leads • Task self-assignment facilitated by daily stand-up meeting and pair programming • Formal meetings, as daily stand-up meetings; compensate insufficient pair programming and rotation with prolonged stand-up meetings • Iteration retrospectives

Table 3. Practices regarding people and processes.

The two teams using the agile processes endorse self-management, but the team members do not behave at will or break all the rules. They are autonomous, but equally are disciplined, which is demonstrated as a combination of peer-discipline and self-discipline simultaneously. Meanwhile, although both teams emphasize the importance of people and interactions among team members, the teams do not rely solely on the willingness and good nature of the developers to make interactions happen. The processes provide the developers with a supporting structure through the interconnection and reinforcement of the practices. This structure can be perceived by and affect each team member. A developer of LicenseMan describes her experience:

> *"At the start it was a bit of daunting, what I would think 'I have to pick up a story that I can do'... but then you get a bit confidence, and you realize there's a lot of support there... You can pair with somebody else, and that at ten o'clock you can say 'I don't know what I'm doing, I need help on this'. You get more confidence and you're not afraid of taking something you know nothing about, 'cause you want to kind of develop, you want to try something different, 'cause you know that the kind of support structure is there first." (Developer/LicenseMan)*

The supporting structure creates a comfortable and sharing environment where the team members can communicate, collaborate and learn constantly and effectively. Learning emerges from the interactions of the team members. No one is left alone on their own device in learning. Meanwhile, both teams review their processes regularly in order to keep them flexible and applicable to the specific contexts in which the teams and projects are embedded. Process reviewing reminds the teams of doing the right things and doing things right. It is seen as a crucial component of agility by the teams.

5 DISCUSSION

5.1 Planning-driven agile process

The findings of this study support the claims that, quite the contrary to the criticism, there is a fair amount of planning in agile methods (McBreen 2000, Dall'Agnol et al. 2004). Dall'Agnol et al. (2004) believe that agile methods actually highlight the importance of planning and organization in projects.

Hansson et al. (2006) suggest that, however, plans must be flexible, allowing response to changes in business and in technology. Building a detailed plan for the next few weeks is useful; having rough plans for the next few months and only vague ideas for the future is a good strategy. The plan should therefore be renegotiated and reprioritized after each time-slot, to keep it up-to-date. Similarly, Fruhling and De Vreede (2006) argue that the course of a software project cannot be planned very far into the future because the business environment changes, customers are likely to adjust the system once they have seen an example, and estimating how long it will take to do requirements is only an estimate. Therefore, when we build plans, we need to make sure that our plans are flexible and ready to adapt to changes in the business and technology. Estimates can be used for a plan, but estimates change. Baskerville (2006) terms planning in agile processes as "artful planning", which means planning for creativity and innovation, planning for serendipity, and planning not-to-plan. Artful planning is a "paradox of planning and not planning that unfolds as a practice required by settings in which large degrees of uncertainty and ambiguity are inevitable" (Baskerville 2006, p. 115).

This study goes a step further, however, by arguing that planning is of central and fundamental importance in an agile process, but takes a different form than in traditional plan-driven approaches. An agile process is not plan-driven, but planning-driven. It is about how to plan, and it is about constantly planning. Planning in agile processes is frequent and fluid. Frequent and regular planning enables a team to work with certainty, even though only for a short term; Fluid planning, constantly adjusting plans, allows a team to respond to change quickly. A paradoxical view on responding to change vs. following a plan suggests that responding to change and following a plan are coexistent simultaneously in the same development process. Frequent planning is a natural consequence of frequent feedback loops in an agile process due to close relationships between a team and their customers as well as among team members. The team needs to react to feedback, make decisions, and subsequently adjust directions they are going towards. The value of feedback would be lost if no appropriate planning follows. However, planning often and planning accurately for a short time, even though they are necessary, are not sufficient to quickly respond to change. A pragmatic view on plans is desirable to keep a team aware of the constant changes around them and avoiding the danger of ignoring potential problems posed by long-term uncertainty. A team needs to have a practical attitude towards plans and constantly adjust them according to what happens and adjust the direction the team is heading towards, that is, to constantly manage the tension between following a plan and responding to change.

5.2 Agile process for people

Agile teams are claimed to be characterized by self-organization and intense collaboration within and across organizational boundaries, and agile methods are claimed as people-centric methods (Cockburn and Highsmith 2001). However, Cockburn and Highsmith (2001) admit that, although "people trump process" - agile teams focus on individual competency as a critical factor in project success, inadequate support can keep even good people from accomplishing the job. This study supports their claim, and demonstrates that agile processes provide people with needed support.

A paradoxical view on the people vs. processes contradiction advocates that both people and process are important components and both should be taken care of simultaneously in the same development process. Team self-management is an inherent attribute of an agile process that needs an understanding of the importance of people in software development. Agile methods are not a guarantee for success – they rely on people to make them work. Interactions among team members, in the form of communication and collaboration, are an indispensable component of a self-managing team. In an agile process, spontaneous interactions happen all the time, but team members are not left to their own devices to interact. There are supporting structures in the process, derived from the interconnections of practices, which create a favourable managerial and cultural environment for interactions to happen. Note that these structures are different from the channelled communication which Brown and Eisenhardt (1998) take as a sign of an overly structured bureaucratic organization.

The supporting structures are not rules of how to interact or imposed by an applied agile method, but emergent from the interconnected usage of practices. They are concrete structures all team members can perceive, and support and enhance the interactions among team members.

Meanwhile, to avoid rigidity and deterioration, a process needs continuous adjustment and adaptation. This is an inherent request of an agile process. Regularly reviewing process allows a team to take gradual steps to change and improve it rather than leaving it to a stage where no effective action can be taken. This is also a reflection of agile methods focusing on processes as well as on people. Process reviewing is more active than merely responding to change. It is seeking for opportunities to change. It is to break the confines of a development method a team adopts and lets people use their common sense, but at the same to reflect on and criticize on common senses that are accepted without questioning.

5.3 Limitations of the study

The two cases included in the paper are both exemplars of constructively using the paradoxical view on the contradictions in agile software development. Therefore, the case analysis was focused more on how the two teams successfully exploited the contradictions in software development to achieve desirable outcomes. However, a paradoxical view also emphasizes tension-generating behaviours and living with tensions. Problems and difficulties will arise from embracing paradox. Periods of conflict in these teams that were overcome would be particularly enlightening and useful to reveal how to use paradox constructively. This is one aspect that is underdeveloped in the current study and needs more focus in the data analysis process.

Another limitation of the study is that, although the two cases show that applying the paradoxical view on the contradictions in agile software development may bring out desirable results, what these results are is not systematically explored. The paradoxical view of agile development pinpoints tensions underlying agile processes which are considered the fuel of innovation and adaptability (Highsmith 2002, Riehle 2000). Consequently, linking the application of paradoxical view with innovation and adaptability of software development teams seems a promising avenue for more profound investigation of how to deal with contradictions in software development.

6 CONCLUSION

In this paper a paradoxical view has been introduced to re-examine the two contradictions discussed in the existing agile literature, namely, responding to change vs. following a plan, and people vs. processes. A paradoxical view – distinct from a dichotomy or even a dualist way of dealing with contradictions – emphasizes the coexistence of the competing themes in a contradiction and makes use of the tensions the contradictions generate rather than trying to eliminate them. Taking a paradoxical perspective to analyze two cases of agile processes in a real-world context, this paper argues first that an agile process is a *planning*-driven process geared to responding to change, and second that it is a process providing supporting structures for people to learn and to improve their competences. To confirm the validity of the paradoxical view on agile software development, more case studies need to be conducted, including both agile processes and traditional waterfall processes which can be compared and contrasted. Further work then can apply the paradoxical view to discover or construct new practices in agile software development that deal with other contradictions in agile software development, such as working software vs. documentation, and customer collaboration vs. contract negotiation, which are not addressed in this study. Further work can also explore explicitly each of the identified perspectives of dealing with contradictions in agile software development.

References

Agile Manifesto (2001). http://www.agilemanifesto.org/, last visit May 2007.

Baskerville, R. L. (2006). Artful Planning. European Journal of Information Systems, 15 (2), 113-115.
Beck, K. (1999). Extreme Programming Explained. Addison Wesley, Reading, MA.
Beck, K. and B. Boehm (2003). Agility through discipline: a debate. Computer, 36 (6), 44-46.
Boehm, B. (2002). Get ready for agile methods, with care. Computer, 35 (1), 64-69.
Boehm, B. and R. Turner (2003). Using Risk to Balance Agile and Plan-Driven Methods. Computer, 36 (6), 57-66.
Brown, S. and K. Eisenhardt (1998). Competing on the Edge: Strategy as Structured Chaos. Harvard Business School Press, Boston.
Cockburn, A. and J. Highsmith (2001). Agile Software Development: The People Factor. Computer, 34 (11), 131-133.
Dall'Agnol, M., A. Sillitti and G. Succi (2004). Project Management and Agile Methodologies: a Survey. Extreme Programming and Agile Processes in Software Engineering, Proceedings. Springer-Verlag, Berlin. LNCS 3092: 223-226.
Fiol, C. M. (2002). Capitalizing on Paradox: The Role of Language in Transforming Organizational Identities. Organization Science, 13 (6), 653-666.
Fruhling, A. and G. J. De Vreede (2006). Field Experiences with Extreme Programming: Developing an Emergency Response System. Journal of Management Information Systems, 22 (4), 39-68.
Glass, R. (2001). Extreme Programming: The Good, the Bad, and the Bottom Line. IEEE Software, 18 (6), 112-111.
Hansson, C., Y. Dittrich, B. Gustafsson and S. Zarnak (2006). How Agile are Industrial Software Development Practices? Journal of Systems and Software, 79 (9), 1295-1311.
Highsmith, J. (2002). Agile Software Development Ecosystems. Addison-Wesley, Boston.
Highsmith, J. and A. Cockburn (2001). Agile Software Development: the Business of Innovation. IEEE Computer, 34 (9), 120-122.
Kalermo, J. and J. Rissanen (2002). Agile Software Development in Theory and Practice, M.Sc. Thesis on Information Systems Science. University of Jyväskylä, Jyväskylä.
Lycett, M., R. D. Macredie, C. Patel and R. J. Paul (2003). Migrating Agile Methods to Standardized Development Practice. IEEE Computer, 36 (6), 79-85.
McBreen, P. (2000). Applying the Lessons of eXtreme Programming. Technology of Object-Oriented Languages and Systems - Tools 34, Proceedings: 423-429.
Paulk, M. C. (2001). Extreme Programming from a CMM Perspective. IEEE Software, 18(6): 19-26.
Poole, M. S. and A. H. Van de Ven (1989). Using Paradox to Build Management and Organization Theories. Academy of Management Review, 14 (4), 562-578.
Rakitin, S. (2001). Manifesto Elicits Cynicism. IEEE Computer, 34(12): 4.
Rakitin, S. (2005) Agile Methods - Beyond the Hype. Food for Thought newsletter from Software Quality Consulting. July/August 2005, 2 (7). http://www.swqual.com/newsletter/vol2/no7/vol2no7.html#article, last visited 04/05/2007.
Riehle, D. (2000). A Comparison of the Value Systems of Adaptive Software Development & Extreme Programming, Proceedings of the 1st International Conference of Extreme Programming, XP2000
Schwaber, K and A. Beedle (2002). Agile Software Development with SCRUM. Prentice-Hall, Upper Saddle River, NJ.
Stacey, R.D. (2003). Strategic Management and Organisational Dynamics: The Challenge of Complexity. 4th ed., Financial Times Prentice Hall.
Stephens, M. and D. Rosenberg (2003). Extreme Programming Refactored: The Case Against XP. Apress, New York.
Streatfield, P. (2001). The Paradox of Control in Organizations. Routledge, London.
Turner, R. (2002). Agile Development: Good Process or Bad Attitude? Product Focused Software Process Improvement, Proceedings. Springer-Verlag, Berlin. LNCS 2559: 134-144.
Vidgen, R., S. Madsen and K. Kautz (2004). Mapping the Information Systems Development Process, IT Innovation Conference 2004 - IFIP 8.6, Leixlip, Co.Kildare, Ireland
Vinekar, V., C. W. Slinkman and S. Nerur (2006). Can Agile and Traditional Systems Development Approaches Coexist? An Ambidextrous View. Information Systems Management, 23 (3), 31-42.
Yin, R. K. (2003). Case Study Research: Design and Methods. Sage, Thousand Oaks, California.