Information Systems Research
Vol. 20, No. 3, September 2009, pp. 355–376
ISSN 1047-7047 | EISSN 1526-5536 | 09 | 2003 | 0355

DOI 10.1287/isre.1090.0237
© 2009 INFORMS

Coevolving Systems and the Organization of Agile Software Development

Richard Vidgen
School of Information Systems, Technology and Management, University of New South Wales, Sydney NSW 2052, Australia, r.vidgen@unsw.edu.au

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

Despite the popularity of agile methods in software development and increasing adoption by organizations there is debate about what agility is and how it is achieved. The debate suffers from a lack of understanding of agile concepts and how agile software development is practiced. This paper develops a framework for the organization of agile software development that identifies enablers and inhibitors of agility and the emergent capabilities of agile teams. The work is grounded in complex adaptive systems (CAS) and draws on three principles of coevolving systems: match coevolutionary change rate, maximize self-organizing, and synchronize exploitation and exploration. These principles are used to study the processes of two software development teams, one a team using eXtreme Programming (XP) and the other a team using a more traditional, waterfall-based development cycle. From the cases a framework for the organization of agile software development is developed. Time pacing, self-management with discipline and routinization of exploration are among the agile enablers found in the cases studies while event pacing, centralized management, and lack of resources allocated to exploration are found to be inhibitors to agility. Emergent capabilities of agile teams that are identified from the research include coevolution of business value, sustainable working with rhythm, sharing and team learning, and collective mindfulness.

Key words: agile software development; coevolving systems; complex adaptive systems; time-pacing; rhythm; mindfulness; innovation
History: Sandra Slaughter, Senior Editor and Associate Editor. This paper was received July 1, 2007, and was with the author $9\frac{1}{2}$ months for revisions. Published online in *Articles in Advance* August 31, 2009.

1. Introduction

Modern software development projects are enacted in increasingly turbulent business environments typified by unpredictable markets, changing customer requirements, pressures of ever shorter time-to-deliver, and rapidly advancing information technologies (Baskerville et al. 2001). The values that were held dear in conventional development methods, such as detailed upfront plans, precise prediction and rigid control strategies, are being called into question by more subtle ways “to bound, direct, nudge, or confine, but not to control” (Highsmith 2000, p. 40). Communities have formed around new “agile” development methods, such as Scrum (Schwaber and Beedle 2002) and eXtreme Programming (XP) (Beck and Andres 2004). These methods are promoted through the “Manifesto for Agile Software Development” (Agile Manifesto 2001), which specifies a set of agile values and principles. Agility itself is defined by Highsmith and Cockburn (2001) as the “the ability to create and respond to change” (p. 120). Anecdotal evidence in the forms of lessons learned and experience reports (Poole and Huisman 2001, Bedoll 2003, Rasmusson 2003, Jackson et al. 2004, Schatz and Abdelshafi 2005, Coffin 2006), as well as several scientific studies (Fitzgerald et al. 2006, Fruhling and De Vreede 2006, Sfetsos et al. 2006, Dawande et al. 2008), have argued for the appropriateness and effectiveness of various agile methods and practices. For example, in a theoretical analysis of pair programming (a common agile practice), Dawande et al. (2008) found that pair programming will likely outperform solo working when knowledge sharing between developers is efficient or when a scarce expert resource can be shared through pairing.

The application of agile methods in software development, however, has not been without scepticism and criticism. Rakitin (2001, 2005) argues that processes, documentation, user contracts, and plans are essential in software development, whereas agile values, such as interactions between people and responding to change, reflect a hacker culture that allows people to throw together code with little or no respect for engineering discipline. Rakitin's "hacker interpretations" of the Agile Manifesto puts agile methods at the opposite pole of planning and discipline and regards agile values as chaos generators. In the same vein, Stephens and Rosenberg (2003) cast doubt on both the XP practices and the philosophy behind XP and try to restore the values of documentation and upfront planning and design in software development. Some agile advocates also consider agile and plan-driven software development methods to be polar opposites (Boehm 2002). In attempting to clarify what agile means, "authorities seeking to describe agile software development methods often cast about for its opposite" (Baskerville 2006, p. 113) and in doing so place "agile" and "plan-driven" methods as polar extremes.

The ongoing debate on agile versus plan-driven methods reflects a lack of understanding of agile concepts and how agile software development is conducted in practice, which can be attributed in part to the weak theoretical grounding of agile methods (Turk et al. 2002, Conboy and Fitzgerald 2004) and further conceptual problems such as lack of clarity and lack of "theoretical glue" (Conboy 2009). These issues raise the concern that agile methods may be "reduced to a series of steps executed by rote" (Highsmith 2000, p. 14). One theory that holds promise for deepening our understanding of agile software development is complex adaptive systems (CAS) theory. Indeed, agile advocates claim that CAS is an appropriate theory for software development (Schwaber 1996, Highsmith 2000) and "the only way to make sense of the world" (Highsmith 2002, p. 48). These claims, however, are a postrationalization to justify what is already done in practice and are not based on scientific and systematic studies. Augustine et al. (2005) argue that software projects are complex adaptive systems and build a CAS-based agile project management framework tailored for XP, but their work is mainly based on their experience of rescuing a mission-critical product-development project. In contrast, Meso and Jain (2006) start by identifying seven CAS principles and then map them to agile practices (such as frequent releases, minimal planning, and continuous learning) suggested by various agile methods. Although Meso and Jain's work shows that applying CAS in the study of agile software development could yield fruitful insights of agile organizations and practices, their work remains conceptual and empirical evidence has yet to be collected to validate the links they draw between CAS and agile practices.

The aim of the current study is to develop an empirically based framework grounded in CAS theory that can be used to guide the organization of agile software development. To achieve this end we analyze the practices used in software development processes and identify enablers and inhibitors of agility and the emergent capabilities that a team needs to possess to be considered agile. In doing this we adopt a broad view of a software development process, which is not only a framework for the tasks and series of steps that are required to build the software (Pressman 1997) but also incorporates the tools used and the people building the software (Schach 1998). In contrast to a development method, which Iivari et al. (1998) define as consisting of "a well-defined sequence of elementary operations which permits the achievement of certain outcomes if executed correctly" (p. 165), in the current study a software development process is viewed as the actual way a software product is developed in a real-world context. Such a process may or may not apply a development method and may or may not follow a method faithfully. In the next section the theoretical basis of the study is constructed, and then applied in the following sections to guide the empirical investigation. The study uses an interpretive case study approach to investigate and contrast the processes of two software development teams, one espousing agile methods, the other a more traditional waterfall approach.

2. Theoretical Development

A complex adaptive system is composed of loosely interconnected autonomous parts, or agents. Agents have the ability to intervene meaningfully in the course of events (Choi et al. 2001) because they have their own local rules (schemata) which are the changeable cognitive structures used to make sense of the environment and determine what action to take. The behaviors of a complex adaptive system resulting from its loosely coupled agents following their local (and sometimes rather simple) rules can be strikingly complex. The concepts, insights, and analytical tools of CAS theory have been applied in management and organizational studies (Mittleton-Kelly 1997, Brown and Eisenhardt 1998, Anderson 1999, Haeckel 1999, Stacey 2003). Although Anderson (1999) suggests that CAS should no longer be considered a new theory in organizational studies, the use of CAS theory in the information systems (IS) domain has been more recent and represented by several special issues, e.g., Communications of the ACM (Crawford 2005), IT and People (Jacucci et al. 2006), and Journal of Information Technology (Merali and McKelvey 2006). Little of the work on CAS in the IS domain has been empirical, reflecting perhaps the difficulty of making the rather abstract ideas of CAS theory sufficiently concrete to support case study research.

There is no definitive account of CAS theory but Volberda and Lewin (2003) summarize and distil the academic and practitioner writing on complexity studies to propose three principles of coevolving, self-renewing organizations: match coevolutionary change rate, optimise self-organization, and synchronize exploitation and exploration. These principles provide the theoretical structure for the selection and encapsulation of key CAS concepts, including coevolution, the edge of chaos, interconnected autonomous agents, self-organization, and the edge of time (Brown and Eisenhardt 1998, Anderson 1999, Stacey 2003). Volberda and Lewin (2003) say these principles are "higher-order principles that must underline any theory of self-renewal and its associated enabling managerial routines and capabilities involving strategy, structures, processes, and leadership" (p. 2126). We consider that these three principles constitute an appropriate theoretical basis for a CAS-grounded study of agile software development and now review the underlying CAS concepts of each principle in turn.

2.1. Principle 1. Match Coevolutionary Change Rate

A CAS tends to alter its structures or behaviors in response to interactions with other CAS. These different systems coexist and coevolve in an ecosystem in which adaptation by one system affects the fitness of the other systems in the ecosystem, thus leading to further adaptations and reciprocal change (Kauffman 1993). Mittleton-Kelly (2003) argues coevolution is not the same as proactive or reactive response. It asks for an awareness of both changes in the environment and the possible consequences of actions, which reverberate around the ecosystem playing themselves out in unpredictable and unexpected ways. Principle 1 states that organizations need to match or exceed the coevolutionary rate of the system in which they are embedded (McKelvey 2003). Mittleton-Kelly (2003) also notes the importance of coevolutionary rate in the context of the firm, which, for example, might be to match or exceed the rate of change of new product improvements made by competitors.

Adaptive organizations must "develop routines, capabilities, and measures which monitor and track rates of change in all aspects of their environment" (Volberda and Lewin 2003, p. 2126). The combination of structures, strategies, and processes adopted by organizations governs the pace of coevolution through regulating each organization's internal rate of change. The rate of change should be sufficient to enable organizations to evolve to the edge of chaos (Anderson 1999, McKelvey 2003), which is a region characterized by bounded instability, i.e., one that is paradoxically stable and unstable at the same time (Stacey 2003). At the edge of chaos "organizations never quite settle into a stable equilibrium but never quite fall apart, either" (Brown and Eisenhardt 1998, p. 12). The edge of chaos provides organizations "with sufficient stimulation and freedom to experiment and adapt but also with sufficient frameworks and structure to ensure they avoid complete disorderly disintegration" (McMillan 2004, p. 22), and "gives them a selective advantage: systems that are driven to (but not past) the edge of chaos out-compete systems that do not" (Anderson 1999, p. 223). The achievement of the edge of chaos is also "a requirement for the emergence of novelty" (Stacey 2003, p. 262). Brown and Eisenhardt (1998) site the edge of chaos between structure, which they define as bureaucratic organizations attempting to run using command and control mechanisms, and chaos. They contend that, to compete at the edge, organizations must understand what to structure and what not to structure, to foster communication, and to capture cross-business synergies. McKelvey (2003) suggests that, in the context of organizations, it is better to think of a "region of emergent complexity" rather than an "edge of chaos." This region lies between stasis and chaos and is defined by two critical values. If an organization falls below the first critical value because it exhibits minimal response to addressing the adaptive tensions it faces then order will prevail. If the organization over-responds to its adaptive tensions, for example, by initiating too many change programmes too quickly, then it may exceed the second critical value and chaos will ensue.

In the context of software development, user requirements embody the most significant sources of change that a team will encounter and have to continue to respond to. Principle 1 directs our attention to the mechanisms of monitoring and tracking changes to user requirements and the practices that enable the development team to match and exceed those rates of change.

2.2. Principle 2. Optimize Self-Organization

Self-organization is the ability of interconnected autonomous agents of a complex adaptive system to evolve into an organized form without external force. Agents are autonomous because they have the ability to intervene meaningfully and to determine what action to take given their perceptions of their environment. Agents are interconnected in such a way that they are responsive to the change around them but not overwhelmed by the information flowing to them through that connectivity (Mittleton-Kelly 2003). In an organizational context, self-organization is the spontaneous coming together of a group to perform a task (or for some other purpose): the group decides what to do, how and when to do it, and no one outside the group directs those activities explicitly (Mittleton-Kelly 2003). Drawing on Nonaka (1988) and Anderson (1999), Volberda and Lewin (2003) argue that self-organization requires a fundamental departure from the command and control philosophy of traditional hierarchical bureaucratic organizations. It is consistent with the often espoused idea of delegating decision making to the lowest possible level and it implies maximizing capabilities of scope at every level of organization. The roles played by individuals in an organization are therefore reshaped in the light of self-organization with emphasis placed on increased autonomy, more interactions with other individuals and environment, and greater participation, especially in the decision-making process (Ashmos et al. 2002). The meaning of leadership shifts from leading and controlling to participating and mediating (Wilkinson and Young 2003). Self-organization, however, does not mean that individuals or units can "pull in all directions at will or break all rules" (Volberda and Lewin 2003, p. 2126). Individuals and teams must still define and follow local rules (and allow these rules to evolve over time) in the course of self-organization. It is worth noting the difference between self-organization and a much advocated management practice, self-management. Self-management is premeditated or deliberately implemented by management, whereas self-organization is truly emergent. Self-organization may be achieved through implementing self-management, but as Stacey (2003) emphasizes:

> It is the very essence of self-organization that none of the agents, as individuals, nor any small group of them on their own, can directly design, or even directly shape, the evolution of the system as a whole. The impact of any agent, no matter how powerful, on the systems is indirectly through their local interaction only… No agent is setting the simple rules for others to follow and then 'allowing' them to self-organize. If they were, the system could no longer be described as a self-organizing one (p. 267).

This principle has two implications to the current study. On one hand, it focuses attention on how control and decision making are distributed in a team to promote self-organization. On the other hand, the emphasis on local autonomy poses a challenge to the ability and attitude of the developers in a project team, requiring that individuals use their autonomy to maximize their capabilities. One crucial aspect not covered by Volberda and Lewin's (2003) formulation of this principle is that self-organization needs energy to flow into and within it constantly to move to and maintain the new form (Prigogine and Stengers 1984). This energy can be in the form of information, knowledge, or other resources needed to sustain self-organized activities and therefore how a team communicates and collaborates to keep informational resources flowing within it also needs to be investigated.

2.3. Principle 3. Synchronize Exploitation and Exploration

This principle is concerned with balancing concurrent innovation and knowledge creation (exploration) with improvements in productivity, improvements in processes, and product extensions and enhancements (exploitation). Brown and Eisenhardt (1998) conceptualize the balance of exploration and exploitation as *the edge of time*, that is, "rooted in the present, yet aware of the past and future" (p. 12). Drawing on March (1991) and Levinthal and March (1993), Volberda and Lewin (2003) claim that "the long-term survival of an organization depends on its ability to engage in enough exploitation to ensure the organization's current viability and engage in enough exploration to ensure its future viability" (p. 2127). Both attributes need to be present and operate simultaneously. Organizations must avoid being mired in the past but not so overenamored with the future that they waste time and effort overplanning it. Organizations that focus on the past and exploitation become trapped but those that forget the past are always starting from new and repeating mistakes.

Because our focus is on the software development process, this principle guides our study to discover how a team continually leverages its current resources and capabilities (exploitation) while exploring new opportunities, learning about new technologies and ways of developing software, and being open to innovation in their development process.

In summary, the theoretical basis presented above provides the structure for the empirical investigation of how agile software development is practiced.

3. Research Approach

3.1. Research Method

This study adopts an interpretive research approach. It emphasizes software development processes as made and enacted by people with different values, expectations, and strategies, as a result of different frames of interpretation. These frames act as filters enabling people to perceive some things but ignore others (Melao and Pidd 2000). Case study is considered an appropriate empirical research method to investigate real-life contexts, such as software development processes, where control over the context is not required or possible (Yin 2003). A multiple-case design is used to ensure that "the events and processes in one well-described setting are not wholly idiosyncratic" (Miles and Huberman 1994, p. 172). Thus the multiple-case design allows us to apply literal and theoretical replication logics (Yin 2003) through the comparison and contrast of two cases that are analyzed using the same theoretical lens.

The two cases reported here represent two software development teams, Pongo and SysCheck. Pongo has adopted XP, one of the most popular agile methods. SysCheck uses a variation of the traditional waterfall-style method. Pongo acts as an exemplar case of agile software development, whereas SysCheck provides a contrasting case to Pongo. Both cases are used to study the factors that enable and inhibit agility and agile team capabilities rather than being viewed as opposing cases (e.g., agile and nonagile). The core case study questions (see Table 1) are derived from the theoretical framework introduced in §2 and are consequently organized by the three principles. Other relevant questions include those about organizational context, team and interviewee background, and several questions to probe interviewees' understanding of agility and agile software development.

The main data collection method used is semi-structured interviews with open-ended questions. Interviews were conducted in English and lasted

Table 1 Core Case Study Questions

Organizing principles	Core case study questions	
Principle 1: Match coevolutionary change rate	1–1	How are user requirement changes monitored and tracked?
	1–2	How is the user requirements change rate matched or exceeded?
Principle 2: Optimize self-organizing	2–1	How is management distributed?
	2–2	How are the capabilities of individuals maximized?
	2–3	How are communication and collaboration facilitated?
Principle 3: Synchronize exploitation and exploration	3	How are exploitation and exploration in software development synchronized?

Table 2 Interviews Conducted

Case	Interviewees	1st visit	2nd visit	Total interviews
Pongo	4	1 group interview 4 individual interviews	2 group interviews 3 individual interviews	10
SysCheck	3	1 individual interview	3 individual interviews	4

between 30 minutes and 2 hours and were recorded and transcribed. Interview quotes are reproduced verbatim. Most subjects have been interviewed twice within a six-month timeframe (as shown in Table 2). Documentation review and field notes were complementary data collection methods. Sources include software development documents, project management documents, and corporate websites and brochures.

Within-case analysis and cross-case comparison are two major steps of the data analysis. The level of analysis is at the team level. The specific data analysis techniques for within-case analysis are coding using the NVivo software package and a descriptive write-up for each case. In the cross-case comparison, the software development processes of Pongo and SysCheck are contrasted and compared and then agile enablers, inhibitors, and agile team capabilities are identified accordingly.

3.2. Case Sites

Pongo is a software development team in an Italian software house specializing in network security and management systems. Pongo, "Play-Doh" in English, symbolizes malleability—a quality the team feels is most necessary to support change. After failing to deliver its last project, the team embarked on a collaborative effort with an XP training laboratory and underwent intensive XP training for six months. Following training, the Pongo team successfully completed several projects using XP and believe that they reached their goals of developing software "good, fast, and cheap" and "working in an enjoyable way" (Dani et al. 2003). Therefore, Pongo is considered here to be an exemplar case of agile software development.

SysCheck (a pseudonym for the purpose of anonymity) is a software development team from a major multi national IT company. The company is considered to be a hierarchical and bureaucratic organization by the interviewees. SysCheck is required to use the waterfall model. SysCheck is aware of agile methods and has also adopted some agile-like practices to circumvent the restrictions of the waterfall processes imposed by the company. The profiles of the two case teams are summarized in Table 3.

Table 3 Profiles of the Two Cases

	Pongo	SysCheck
Team composition	3 developers (1 assumes the role of XP coach), 1 project manager	4 developers, 1 project manager
Location	Collocated in an open office space	Collocated in a semi open office space
Development method	XP	Waterfall
Years of use of the method	More than 5	More than 5
Type of software developed	Applications for external customers	Applications for internal use
Company background	Small software house, specializing in network security and management systems	A major IT company providing both IT products and services

4. Case Analysis

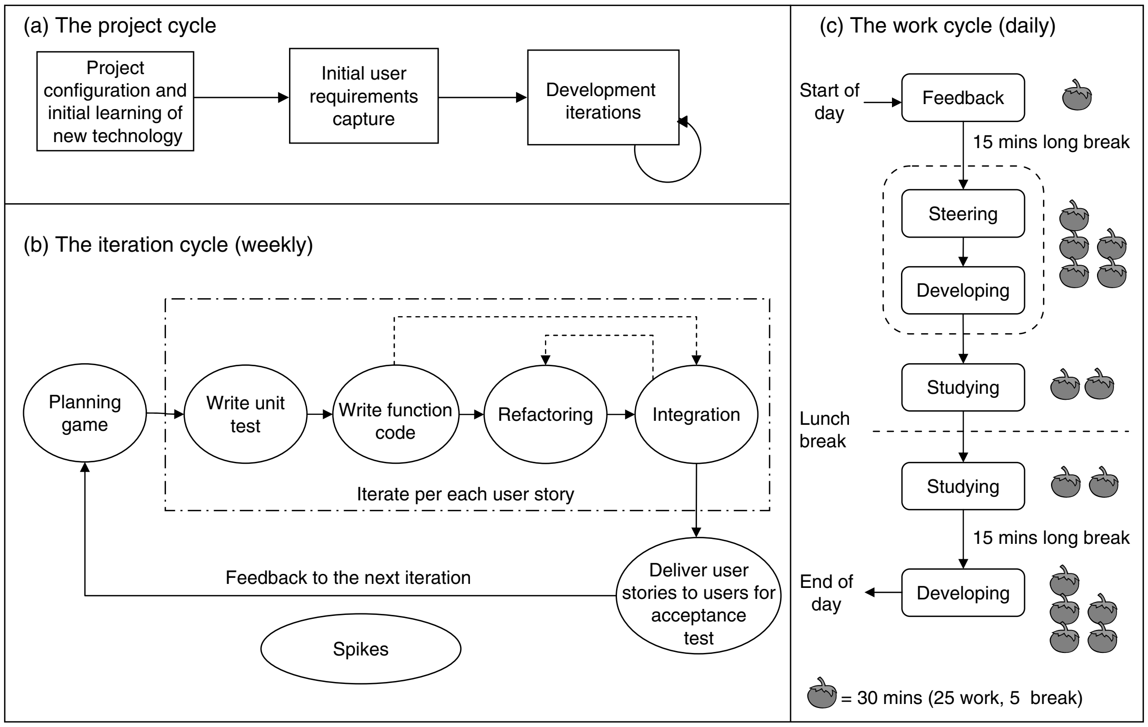
4.1. Pongo

1–1: *How are user requirement changes monitored and tracked*? The life cycle for Pongo consists of project configuration, initial requirements capture, and then a number of development iterations (Figure 1(a)). User requirements are gathered throughout the whole project life span with user requirement changes captured constantly in the planning game that initiates each iteration (Figure 1(b)). User requirements are structured as user stories, which are estimatable and testable statements of requirements together with acceptance tests to specify what constitutes a complete and acceptable piece of software for each user story.

The team delivers working software in an incremental way each week for acceptance test (Figure 1(b)), as one developer comments:

> Every week we have a delivery. It's better to discover that you have not understood some user requirements after a week than after one month. (Developer A/Pongo)

Figure 1 Development Life Cycle of the Pongo Team


(a) The project cycle
Project configuration and initial learning of new technology
Initial user requirements capture
Development iterations
(b) The iteration cycle (weekly)
Planning game
Write unit test
Write function code
Refactoring
Integration
Iterate per each user story
Deliver user stories to users for acceptance test
Feedback to the next iteration
Spikes
(c) The work cycle (daily)
Start of day
Feedback
15 mins long break
Steering
Developing
Studying
Lunch break
Studying
15 mins long break
End of day
Developing
= 30 mins (25 work, 5 break)

Note. Adapted from Vidgen and Wang 2006. A flow following the dotted arrow may happen, but less often.

The rapid turnaround of user stories gives customers quick feedback on the requirements they have requested, allows them to monitor the progress of the developers, and gives them the opportunity to learn to use the software through the execution of acceptance tests.

The team realizes the importance of separating business complexity from technical complexity, and guides customers to write user stories and acceptance tests in such a way that they identify precise business scenarios from a business perspective. User stories and acceptance tests are seen as the interface between the customers and the team; the customers address business complexity whereas technical complexity is internal to the team and not intermingled with user stories and acceptance tests.

1–2: *How is the user requirements change rate matched or exceeded*? Pongo uses one-week iterations. No change should be 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, the team does change iteration length to account for different projects, different stages of a project, different frequency with which they communicate with customers, and different sets of user stories being implemented:

> Every time we have to ask ourselves, what is the best [*iteration length*] for this particular set of stories, every time, because the risk is that if you don't recognize that specific set of user stories is suitable for a period of time ... we prefer [to fix iterations] but it's not always suitable, not always. (Coach/Pongo)

Similar to the one-week iteration that paces the project, a working day is paced according to pomodoro time (Figure 1(c)). Pomodoro, "tomato" in Italian, comes from a tomato-shaped kitchen timer the team used when they were trained at the XP training laboratory. The timer is set for 25 minutes of work followed by a five-minute break when the team members can check emails, take coffee or have a chat. The team also reports that it is not easy to maintain the pace set by iterations and pomodori. Holding the pace and keeping focus require continual effort from the team members.

The pomodoro is the basic unit of planning. All the team members participate with the customers in the weekly planning game in which the developers estimate the effort needed to implement the user stories selected by the customers. Although a close interaction with customers helps when capturing user stories the team finds that sometimes they become trapped into long discussions with customers during the planning game with the result that no user stories are generated by the end of the game:

> For example, if we have a very long planning game, if it's difficult to communicate with customer, we are of course not agile at this moment. (Developer A/Pongo)

The team recognizes that when this kind of ineffective discussion happens they need the courage to stop interaction with customers.

Estimates of user stories are expressed in the unit of the pomodoro and the team keeps the average estimate of a user story to around 25 to 30 pomodori, which is considered an appropriate size:

> Typically when you have user stories big, it's not easy to have feedback from your activity... it's also very hard to imagine if your estimate is realistic or not. (Coach/Pongo)

The principle of writing user stories is "as small as possible." A written user story should fit to a quarter A4-sized story card. If a story card cannot contain a user story, it is a sign that the story is too big and needs to be broken into smaller ones. The capacity of the team to implement user stories in an iteration is called "la cassetta dei pomodori" (the basket of tomatoes). In the planning game, this capacity is compared with the sum of the estimates of user stories. This gives the customer and the team an idea of which and how many user stories can be implemented in that iteration. If all the stories cannot be implemented in the iteration then the customer prioritizes the stories and chooses the stories that are to be delivered at the end of the iteration using the rule "greatest value to the customers first." The "basket of tomatoes" should always match the estimates of the user stories chosen for that iteration so that the team can work at a comfortable pace. Because the team plans for one week only and user stories are generally small in size, the team tends to get accurate estimates for user stories. Constant prioritization of tasks helps them make a quick decision on what to drop as and when circumstances change and to adjust plans accordingly. This typically takes place in the daily steering sessions (Figure 1(c)). In the daily steering session the team quickly plans what to do that day by picking up tasks from the storyboard. They also raise issues such as technical obstacles and ask others for help.

The team considers planning a natural step following frequent external and internal feedbacks:

> Always plan, this is the core. (Project Manager/Pongo)

The team also realizes that planning is a learned ability which is improved and refined through the development process with the ability of a team to understand systems, situations, and problems.

By planning in such a way and pacing development with iterations and pomodori, Pongo has discovered rhythm, an emergent state of working, a state difficult to reach but easy to lose:

> It's a special condition, it's a magic condition, like in a sports team. There could be some moment, some situation in which every component lives in another layer, all the things are much easier. (Project Manager/Pongo)

The Pongo coach describes a similar experience of rhythm:

> When you can maintain a rhythm you have no anxiety. You have no worry about something particularly, so you are not stressed. And playing an instrument is the same. I play the guitar so I know what I am talking about. When you reach the right rhythm, you can feel it. It is a special condition where work is ideal, but it is a special condition you cannot reach every day... Rhythm is something that is very close to life. You are working without any anxiety, something that life does. (Coach/Pongo)

2–1: *How is management distributed*? For Pongo, management is ultimately an internal process of the team. Every team member is involved in management and assumes responsibilities to make decisions for the team, even though roles like project manager and XP coach exist in the team:

> Project manager is kind of activity that is absorbed by all eventually, all the team has to participate in managing the project in all the aspects of the project. (Coach/Pongo)

However, when facilitating team self-management it is possible that the project manager becomes externalized from the team:

> I don't live inside [*the team*] now. I can give them my feedback, but it isn't the same feedback ... The risk is it becomes idiosyncratic. (Project Manager/Pongo)

One important mechanism the team uses to implement self-management is constant observation, to be attentive of what happens to other team members and in the environment. To effectively observe others, one needs to be able to self-observe, as one developer comments:

> Feedback comes from continuous watching the activities of the project ... Before observing others, you must be able to observe yourself ... but not in the sense of control, you have to look around, and watch what the others are doing, if they are focusing on the things they are doing ... It's good that every team member can do if something goes wrong. A team is like a person, every part, member of the team must check if the other part works good, like the body, if one part doesn't work, the other has to help. (Developer B/Pongo)

2–2: *How are the capabilities of individuals maximized*? Pongo team members are involved in all development activities of a project, and all have to deal with customers, analyse user requirements, and write code. There are no dedicated traditional roles, such as system analyst, designer, programmer, or tester, in the Pongo team. Each team member is able to assume all the roles, because comprehensive competences are required to work with user stories, which are self-contained and encapsulate different development activities including analysis, design, and coding.

Task self-assignment is an effective way to improve the competences of each team member. During the steering session, the team members sign up and take ownership of tasks they would like to implement in that day. 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 so good at, and then pair up with a more experienced member to acquire new skills. Through pair programming, learning happens naturally and in a mutual way. That is also the reason why the team always ensures that team members work in pairs after a spike task (a task to explore an unknown technical issue) or a study session (Figure 1(c))—what is learned by a single developer can then be shared by working with another developer as a pair.

2–3: *How are communication and collaboration facilitated*? The team works in an open working space, which facilitates everyone's involvement in communication. Regular weekly and daily meetings (planning game, steering, and feedback sessions) also facilitate and structure team communication.

Pair programming is the main collaboration mechanism in Pongo. During development time, the developers always work in pairs on tasks. They physically sit together and share one desktop, one using the keyboard, as "driver," the other using the mouse, as "navigator." Generally team members are self-pairing. Pair rotation happens frequently, sometimes to the extent of per pomodoro. Between the two paired developers, the owner of the task generally stays, and the other goes to pair with a different developer. When working in pairs, however, it is not always easy for the pair to recognize when to stop talking and start writing code:

> When the pair starts long discussion, you don't communicate really ... When you start this type of discussion which is negative, you could spend 20 minutes on discussion without writing anything. (Coach/Pongo)

Effective communication and collaboration lead to constant sharing and team learning. Pongo considers sharing an important aspect of team working. What is shared among the team members is not only the technical knowledge related to different areas of a project (which helps to distribute competences among the team) but it is also the knowledge about who knows what, which helps the team members self-organize to implement tasks and facilitate learning:

> As a team, you have to face every moment, in any case, without barriers. (Developer B/Pongo)

For Pongo, learning means doing things differently, as the project manager explains:

> What is the effect of learning? That you change something ... If I learn, I don't do the same thing in the same mode. If I learn something, I modify, I change my behaviour. If I see that the behaviour is the same, I think there is no more learning there in that cycle. (Project Manager/Pongo)

3: *How are exploitation and exploration in software development synchronized*? A working day for Pongo starts with a feedback session in which the team reflects on the previous day. The feedback is focused on the development process, not technical issues. The team members also talk about the feelings they had, anxieties felt, what the team achieved, or whether something went wrong during the previous day. 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 help keep them motivated. Sometimes the team also challenges the practices that work well, as the coach explains:

> It's important to perturb the system from time to time to see if it can survive. (Coach/Pongo)

The regular use and review of the practices leads to the internalization of the process in the team's day-to-day life, becoming a part of the mental model of the team:

> We practice [*XP*] everyday, and we have to deal with these practices all the time... It's what we do everyday, it's real, it's something concrete. (Developer B/Pongo)

Although the daily feedback session helps the team exploit and retain what the team is doing well, studying new things helps them understand what they might possibly do. Four pomodori (two hours) are reserved for studying everyday, generally two before lunch and two afterward. The team members can freely choose the content they would like to explore in the daily study time out of their personal interests. New ideas that emerge from study can then be tried out in the time allocated for development. Of the two hours, one is dedicated to exploring issues not directly relevant to the current project, which can particularly benefit the team with regard to creativity:

> We tried to experience different ways to approach the study, and we found that to split study time considering one part dedicated to project related issues, and the other dedicated to issues one wanted to explore for some reason, was better than always study the project related issues, because you could find some results that seem not useful in the future for the project, but sometimes, magically, it works, that happens. (Coach/Pongo)

Allocated study time allows the team members to learn new things within working hours and helps to keep them motivated. Pongo believe that the productivity of the team will drop if there is no time for personal interests to be pursued. Further, the presence of study time in daily work serves as a break from intensive development activities and thus helps the team to work at a sustainable pace:

> When sometimes we skip study time, we have to develop all the time so we have to do the same activity along the day, our efficiency is lower. It's very important to switch between activities of different kind...When we study, and to pay attention to other issues, when we begin, when we start again the development, we can start with more sources, more imagination. (Coach/Pongo)

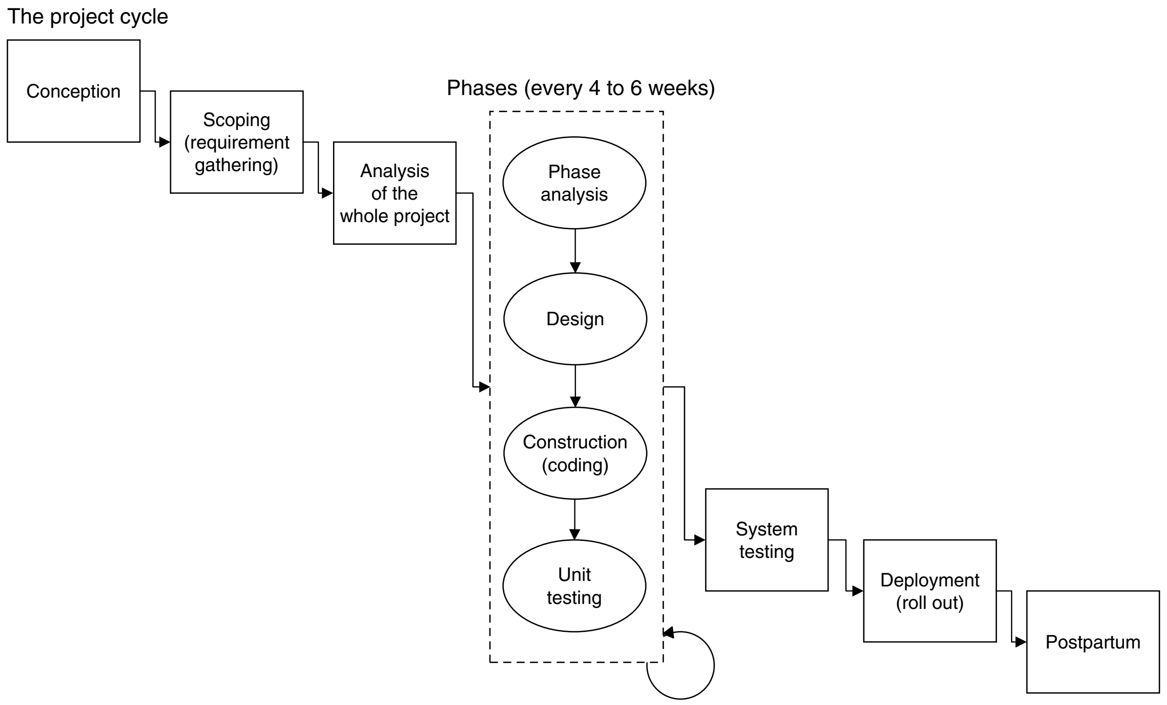
4.2. SysCheck

1–1: *How are user requirement changes monitored and tracked*? SysCheck follows a waterfall style lifecycle that is roughly divided into sequential stages including requirements gathering, system analysis, design, construction, and testing (Figure 2). The business requirements are generally dictated by the business units of the company, who decide which projects to initiate. The business requirements are then broken down into product requirements and elaborated with technical details by the project manager with the involvement of the team members. The product requirements must be signed off by senior management before the work schedules can be made.

Because the system the team develops is a command-line package with a limited user interface, the team does not believe there is a necessity for close interaction with customers and consequently customers are only involved in the early scoping phase (Figure 2) and in the testing phase toward the end of a project.

1–2: *How is the user requirements change rate matched or exceeded*? Generally the team delivers a new major version of the product at the end of a 9 to 12 month period. Though the system the team is working on has been around for so long that the requirements are considered stable and well-defined, there are often changes coming from the business that impact on the project, such as organizational changes, project cut, or scope changes, which may require the team to deliver something quickly and bring the project to a close. As a response to this uncertainty, rather than have one big construction (coding) phase, the team breaks the

Figure 2 Development Life Cycle of the SysCheck Team



Note. Adapted from Wang and Vidgen 2007.

development work into multiple phases of four to six weeks each. In each phase, a mini-waterfall cycle is followed, including analysis, design, coding, and unit testing (Figure 2). The most crucial blocks of work are put in early phases and packaged in a self-contained way such that, though the team does not actually deliver anything to the customer at the end of a phase, they could if they have to, as one developer describes:

> Each phase will basically stand by itself, and in our case we know that we have time to finish it and we can finish it, so if we start a phase we will finish it...we always finish a phase. (Developer A/SysCheck)

This phased approach is seen as a way to protect the team from unexpected change:

> It's kind of like defensive planning, we plan in such a way that if we have to go out of the door we could go out of the door. (Project Manager/SysCheck)

If any significant change needs to be made to the signed-up requirements then a change request must be completed and signed off and a change control mechanism is triggered. Once the request is authorized the team can implement the change.

The team has no shared routines at the daily level:

> [*Do you have any daily routine to organize your work*?] No, not really, just work, work, eat, work again, and that's it. (Developer A/SysCheck)

Project planning is an important activity at the start of a project and is time consuming. The plan covers the entire lifespan of the project. The project manager breaks down the product requirements into development tasks, consults the team members for task assignment, and works out a work schedule for the whole project. The work schedule as a result is considered the most important document by the project manager and it is meant to be followed:

> It [*the work schedule*] is done up in front, we're trying to get schedule as accurate as possible to start... I would be very aware in my head of the schedule and what's going to come up next week and what we should finish and what we should start, that kind of thing... The schedule is kind of like, if we were driving, for me to be the map, and if we start to go out of the map, we have to figure out how to get back on to the map...the schedule would be what I view as a kind of directive, so where we are going. (Project Manager/SysCheck)

the project manager tries to make the work schedule as accurate as possible. He manages to control the size of a task with a subtask generally taking several days while a predictable piece of work may have an estimate of up to three weeks.

2–1: *How is management distributed*? Project management is seen mainly as the responsibility of the project manager, as this developer comments:

> The project manager sets targets, deadlines... I suppose the project manager would adjust the task states if something comes up or if the priority of the tasks changes, then he can have a meeting with us and tell us about the changes. (Developer B/SysCheck)

The project manager, who is also the technical lead and involved in the development tasks as the other developers, nevertheless tries not to micro-manage the project. He is seen as open-minded and giving the developers a certain amount of autonomy.

2–2: *How are the capabilities of individuals maximized*? There are no specific functional roles in the team. The developers are not specialized on specific tasks and everybody has the opportunity to do different things, as the project manager claims:

> We don't kind of say, okay, you do this you do that, everybody kind of chips in. (Project Manager/SysCheck)

The project manager assigns tasks to the team members in such a way that everyone can get what they are interested in doing rather than what they are good at. The team has an open discussion on what work they are really interested in doing, then the project manager arranges the task assignment based on the collected opinions. The team members help each other in task implementation, which is self-arranged through informal communication. If someone runs into difficulties they ask for help from the other developers directly. SysCheck emphasizes the importance of people in software development, regardless of what software development processes they use:

> You need to respect the fact of human being, and you need to listen to that, if you want good communication [*and*] people working together well... You need people to feel confident and comfortable with each other... If people are unhappy, the project falls apart. (Developer A/SysCheck)

2–3: *How are communication and collaboration facilitated*? The team relies heavily on informal, person-to-person communication. Depending on where a project is, the team does have some formal meetings at different frequencies. For instance, they meet formally once every two or three weeks in the middle of a design phase. During an intensive testing phase, they meet formally every few days. However, the team members do not have positive opinions on formal team meetings:

> The meeting is here to help people to communicate on the project, but you don't have to be at the table to speak about your project, you can go from your desk and go to another person... I find it's pretty depressing to be at the table. (Developer A/SysCheck)

Though the team does not use pair programming in development, they use the idea of pairing when checking code in and out of the code base. Two team members work as a pair in this case to look over each other's shoulders.

3: *How are exploitation and exploration in software development synchronized*? There is no built-in practice targeting the improvement of the process during development to exploit what the team has already done well or to weed out what does not work well. The team holds a formal project "postpartum" meeting at the end of a project, at which the issues regarding the development process are discussed. But it is not seen as meaningful by the team members. Indeed, the team has no clear awareness of the development method they are using and neither do they care about it. It is seen as imposed by the management:

> The project life cycle is not decided by us. There is somebody from upstairs that says 'listen, this is the project life cycle that we use as a company, so you have to use it'... We don't concern as much the process, maybe the project manager, I should say he knows more about the approach, me personally I just yeah I have a task, I implement it, that's it. (Developer B/SysCheck)

Driven by a common-sense view, the team borrows and blends some agile concepts into the process. They believe it makes them work more efficiently, as the project manager comments:

> I think agile is great, and I like the whole idea the way it works and the concepts and all that. But like I said, to me they are kind of common sense concepts. If we

can use them, we should use them. (Project Manager/ SysCheck)

The team does not have a built-in practice to support exploration. The team members may explore new ideas ad hoc, but it mainly relies on the maturity level and willingness of each individual.

4.3. Summary of Findings

The findings from the cases are summarized in Table 4, reflecting how software development is enacted and organized in Pongo and in SysCheck. Although they broadly follow a waterfall lifecycle, SysCheck have a number of practices that would contribute to agility: project manager consults team members about their task preferences; project manager takes account of personal development needs when allocating work; functional roles are not separated; project is broken down into mini-phases to cope with unexpected schedule changes; and pairing to check code in and out. These practices that may contribute to greater agility are flagged with a "+" in Table 4. Pongo, on the other hand, faces a set of issues that may put the team at the risk of losing agility: striving to maintain the pace set up by fixed-length iterations and pomodori in turbulent environments; over-communication between the team and customers; over-communication between team members; and the project manager becoming externalized from the team. These potential sources of loss of agility are flagged with a "−" in Table 4.

5. Discussion: Agile Organizing Framework

In this section the case study findings are discussed and key themes identified, leading to the production of an agile organizing framework that identifies enablers and inhibitors of agility, and the emergent capabilities of an agile team.

5.1. Match Coevolutionary Change Rate

5.1.1. Evolving Business Value. Agility in software development is not only responding to change, or even proactively creating change, it is the coevolutionary capability of a team and their customer. This coevolutionary capability is expressed through the medium of user requirements, which evolve through the evolutionary process (Aldrich 1999) of variation (generating new requirements), selection (establishing the relative priority of requirements), and retention (implementing the chosen requirements). Janzen (1980) says coevolution is more than "interaction" or "mutualism"—each population must make evolutionary changes in response to a selection pressure from an associated species. Thus, although development is driven by business value it is not led blindly; the development team needs to challenge and be proactive in its communication with customers, as in the Pongo case. The user requirements reflect both developer and customer understanding of the business domain and what constitutes a potential solution. A close relationship between a team and their customers is needed such that developers understand the (changing) business environment and customers understand the (changing) capability of technology such that each can apply an informed selection pressure on the other. This suggests that the up-front specification of requirements, as in the SysCheck case will inhibit coevolutionary potential.

5.1.2. Coping with Change. For Pongo change is inevitable, routine, and expected but work is stable and fixed over short cycles. Time pacing is a fundamental building block for Pongo to cope with change and drive the engine of team and customer coevolution. Time pacing with one-week iterations that are self-contained and ring-fenced gives Pongo a steady pace, a clear focus, and freedom from distractions. Viewed as a contract between the team and the customer, short fixed-length iterations in which user stories do not change offer the team short-term certainty to focus on work and to work without anxiety. Because iterations are short, user requirements are only frozen for a short while and new changes can be accommodated quickly. They deliver software at the end of each iteration and draw frequent satisfaction from this closure. Similarly, the daily routine for Pongo, time paced by pomodori, includes short and guaranteed breaks to give respite from intensive working. The SysCheck development process is driven by events, such as the end of scoping, the end of analysis, etc., and the officially signed-off documents act as tokens for the transition from one stage to the next. Work is fixed over long cycles but subject to unpredictable and exceptional changes requested

Table 4 Summary of Findings from the Pongo and SysCheck Cases

	Pongo findings	SysCheck findings
1–1: How are user requirement changes monitored and tracked?	—Ongoing requirements gathering throughout project —Incremental delivery at the end of each iteration; customer learning how to use software through iterative acceptance tests —Separating business complexity from technical complexity —Over-communication between developers and users to detriment of user story generation (−)	—Upfront user requirement specification (dictated by the business unit); official sign-off by senior management —Formal customer involvement limited to early scoping and final test phases
1–2: How is the user requirements change rate matched or exceeded?	—One-week iterations, frozen user requirements within an iteration, but varying iteration length according to the project context —Pacing a working day by 25 minute "pomodori" with 5 minute break per pomodoro —Pace set by iterations and pomodori can be difficult to maintain (−) —Planning at iteration and daily levels —Managing granularity of story size (should fit to a quarter A4 sheet) —Matching estimates with the velocity (the basket of tomatoes) of the team —Task prioritization and reprioritization	—Following waterfall stages (e.g., scoping, analysis, construction), but using self-contained internal phases, each four to six weeks duration, to help protect the team from unplanned change (+) —Change control procedure for substantial changes to requirements —Up-front planning of the whole project —Endeavouring to follow the entire work schedule faithfully —Large size of task (several days up to several weeks)
2–1: How is management distributed?	—All team members involved in project management and assume responsibility for making decisions —Team members observe each other and self-observe —Project manager may become externalized to team (−)	—Management centralized through the project manager —Project manager consults developers about their preferred tasks (+)
2–2: How are the capabilities of individuals maximized?	—All team members are involved in all aspects of development activity —Task self-assignment based on the interests of developers and supported by daily steering session and pair programming —Pair programming after spike or study session to share learning	—Team members freely involved in all activities—no separation of functional roles (+) —Tasks are assigned by the project manager based on the interests and developmental needs of the team members (+)
2–3: How are communication and collaboration facilitated?	—Open working space —Regular weekly and daily meetings —Pair programming and pair rotation —Over-communication between developers to detriment of code production (−)	—Relying on informal communication between team members —Pairing to check code in and out of the code base (+)
3: How are exploitation and exploration in software development synchronized?	—Daily feedback session on the progress of the previous day, focusing on positive aspects as well as issues arising —Willingness to perturb the process to test its robustness and to experiment —Reserving four pomodori (two hours) for study per day: one hour for project-related study and one for nonproject-related	—Postpartum (at end of project) —Use of "common sense" when following the development process —Ad hoc and informal exploration

Notes. "−" = potential sources of loss of agility for Pongo, "+" = potential sources of agility for SysCheck.

by senior management at short notice, often leading to instability. The phased approach SysCheck uses can be seen as time-pacing of a kind but the phases are much longer than the iterations of Pongo and lack immediacy and responsiveness. For SysCheck, phases are seen more as a way to protect the team from change rather than an active desire to embrace change.

The findings of the study show that time pacing, or temporal pacing, is a way of combining flexibility and control in turbulent environments. Gersick (1994) suggests that temporal pacing is a prominent mechanism for keeping organizations adaptive in the face of uncertainty. Brown and Eisenhardt (1998) define time pacing as an internal metronome that drives organizations according to the calendar, e.g., "creating a new product every 9 months, generating 20% of annual sales from new services" (p. 167). This is opposed to event pacing, in which change happens as a response to events in the environment such as competitor moves or the discovery of a new technology. Time pacing in software development means that the development cycles are triggered by the elapse of time allowing a team to change frequently but stopping them from changing too often or too quickly, thereby providing stability for development activities. Time pacing based on short timeframes, therefore, reduces the risk and cost of responding to change and keeps a team focused on work. The findings of this study suggest that several aspects need to be considered when setting the pace. First of all, the team should consider the coevolution rate with customers, which will vary from team to team and project to project. Each team needs to find its own pace in each specific context. Secondly, a team needs to understand what pace can be sustained over time. A suitable pace strikes a balance such that the iteration cycle is long enough to get some meaningful work done but short enough not to lose momentum and responsiveness to change. Once a pace is set it is important to stick to it as regular pacing brings stability to a team and small, frequent closures at the end of each boxed time period help keep team members satisfied and motivated.

Time pacing provides a basis for accurate planning. Pongo plans in detail for the short term through weekly iterations and replans as needed at the start of each day. Although the team recognizes that planning is an uncertain process, frequent planning for short time periods in fine-grain detail helps them improve their planning capability leading to accurate estimates of what can be delivered in an iteration. Planning in SysCheck is done at the beginning of a project and covers the whole lifespan of the project. The project manager develops the work schedules that the developers are to follow and for SysCheck "the plan is the plan"—it should be followed faithfully with change being dealt with on an exception basis. The change control mechanism that SysCheck is mandated to use is peripheral to its development process rather than a core component.

The Pongo case suggests 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. 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. Both high-level, sketchy, long-term plans and detailed, accurate, short-term plans are necessary in an agile process. Time pacing provides a team with a given amount of certainty which makes frequent and accurate short-term planning possible and meaningful. To achieve accurate estimates in planning, a team should be able to break down their work at a fine granularity level, which enables a bottom-up planning process that leads to reliable plans for the short term. This is in contrast to the traditional top-down planning at the project level, which generally results in coarse-grained tasks even for the short term. Regardless, a team needs constantly to adjust the plans according to what happens in its environment. Baskerville (2006) uses the term "artful planning": 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" (p. 115). Agile planning can provide software development teams with an ability to work with stability while embracing uncertainty, providing developers with a sense of security, and control over their work. However, uncertainty is inevitable and is also a source of novelty in an agile process where ongoing change arises from close relationships with customers and evolving user requirements. A truly agile process is a delicate balance of stability and uncertainty which enables a software development team to work adaptively at a fast yet sustainable pace.

Working with a sustainable pace and leveraging both stability and uncertainty, a working rhythm can emerge. This is a special condition where work is ideal, the team is not anxious or stressed about the work, and there is a synergy of the team that brings the team members to a level transcending the individual. It provides a relief against anxiety and a guard against overworking. Rhythm is different from the mechanical metronome or heartbeat metaphor by which organizations synchronize their clock with the marketplace and their environment. Instead, rhythm is a subtler state related to the flow and feel of work that can emerge from a time-paced agile process. It is difficult for a team to reach such a state but easy to lose it.

5.2. Optimize Self-Organizing

5.2.1. Self-Management and Team-Discipline. The developers of Pongo have a large degree of autonomy and carry out their activities in a self-managed manner. Total team involvement in all development activities with no separation of functional roles gives everyone a chance to develop different competences and for these competences to be distributed among the developers, leading to an autonomous team that can work on any aspect of the development. Pongo endorses self-management but this does not mean that there are no rules or that the rules can be broken; team members are autonomous but disciplined, which is achieved through the simultaneous presence of peer-discipline and self-discipline. Observation and self-observation keep the team members aware of what state the team is in and stimulate self-responsibility, which is necessary for resolving management into an internal process. The project manager of Pongo is more like a peer in terms of team interactions and all team members are encouraged to interact directly with other team members. For SysCheck, project management is something external to the developers and is primarily the responsibility of the project manager. Although the team members have a certain degree of autonomy, it is not an inherent characteristic of the process and depends greatly on the open-mindedness and experience of the individual project manager. The project manager is the hub of the formal communication and collaboration of the team, even though the team members interact with each other directly in an informal manner. Sharing thus depends on informal relationships between team members and the communication skills of the project manager, who is in a sense external to and separate from the project team.

Agility is, therefore, closely linked to the ability of a team to be autonomous and self-managing. Team self-management not only needs team autonomy, but also requires team discipline, which is self-generated and from peers, not simply imposed by managers. There is no lack of discipline in a truly agile team, yet the team can work with ease and satisfaction, where agility is "intimately related to the relaxed, competent atmosphere that pervaded the developer group" (Sharp and Robinson 2004, p. 373). This is in contrast to *mindlessness*, which Butler and Gray (2006) see as the mechanical use of "cognitively and emotionally rigid, rule-based behaviours" (p. 215), whereas *mindfulness* involves openness to novelty, alertness to distinction, sensitivity to different contexts, awareness of multiple perspectives, and an orientation in the present. Drawing on an analogy with individual and collective learning, Butler and Gray go on to define collective mindfulness, providing examples of organizational entities such as hospitals providing life-and-death services and aircraft carriers coordinating resources in hostile environments. Collective mindfulness is more than the sum of individual mindfulness and "ultimately relates as much to the distribution of decision-making rights (i.e., power) as it does to the capabilities of any particular individual." (p. 216). In a self-managed team, a manager's role is more of a facilitator, creating an environment that fosters the emergence of self-organization. In such an environment, managers need to take a subtler approach than command and control, and should nudge, remind, and reinforce agile behaviors through communication with team members. They must work with the paradox of control—they are simultaneously in control and not in control (Streatfield 2001).

5.2.2. Supportive Structures for Communication and Collaboration. Interactions among team members, in the form of communication and collaboration, are an indispensable component of a self-managing team. In Pongo, spontaneous interactions happen all the time in the open work space. They are fostered by interconnected practices, such as task self-assignment facilitated by daily steering meetings, pair programming and pair rotation, which help to create the supporting structures that in turn create a favorable environment for interactions to happen. Learning emerges from the interactions of the team members and no one is left to their own devices in the learning process. In SysCheck team interactions happen mainly informally, on a one-to-one basis, and knowledge sharing depends largely on people's willingness to share and their personal interests. There are no evident structures to support team interaction and the learning process. Note that these supportive 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, rather they provide an enabling context to sustain team self-organized activities and facilitate sharing and learning among team members. Multiskilling of the developers (no separation of functional roles), as reported in both cases, also supports interactions among team members.

Through effective communication and collaboration, a team shares not only knowledge about the project but also their understanding of the working context. Context sharing is a precondition for a team to provide effective feedback, interpret that feedback in a sensible way, and take appropriate action. Further, team learning emerges as a result of close interactions among team members. Team learning is a prerequisite for organizational evolution and coevolution (Mittleton-Kelly 2003). It is different from individual learning, though closely related and dependent on it. Team learning is a collective result whereby a team as a whole acquires new knowledge and competences as a result of individual learning being shared among team members.

5.3. Synchronize Exploitation and Exploration

5.3.1. Process Adaptation and Improvement. SysCheck emphasizes the importance of "common sense," which they believe is needed in addition to simply following a mandated method. However, Pongo goes a step further: as well as having time for the formalized process review to let people's common sense speak, there is an opportunity for the team to reflect on and critized "common sense" practices that would otherwise be taken for granted.

The findings of the study suggest that a process needs continuous adjustment and adaptation to avoid rigidity and deterioration. First, the agile team must be able to *adapt* the process to the development context taking account of factors such as the type of application and the customer. Where a one-week iteration might work for one customer, for another customer (or for the same customer with a different business application) a different iteration length may be appropriate. Regularly reviewing process allows a team to take gradual steps to change and improve the process rather than leaving it to a stage where no effective action can be taken. It is not merely a passive responding to change—it is an active seeking of opportunities for change. Second, regularly reviewing process makes the practices meaningful to developers, infused into a team's life, and internalized as a part of the mental model that guides the behavior of team members. Zmud and Apple (1992) suggest that routinization and infusion are important aspects of capturing and retaining innovative organizational behavior. Teams using agile practices as a routine part of their development work may be in a good position to discover innovative ways of using them, and thus have the potential to *improve* the development process.

5.3.2. Routinizing Exploration. Development of the product through the iterative delivery of user stories should be balanced with routinized exploration, in which team members can research new ideas and new areas. Pongo routinizes exploration through the allocation of study time whereas in SysCheck no resources are allocated formally for exploratory activities. In SysCheck time needs to be accounted for against project plan activities and the project manager is constrained by the need to report full utilization of developers. In practice, the developers are given leeway to explore areas they are interested in as and when they can.

The Pongo case study suggests that exploration can facilitate innovation, which the UK Department of Trade and Industry defines simply but succinctly as the "successful exploitation of new ideas" (DTI 2003, p. 18). The "vast majority of authors" dichotomize innovations as "incremental/minor" or "radical/major" (Freeman 1994, p. 474), although these classifications can also be thought as defining a continuum of innovation types (Abernathy and Clark 1985). Regardless, the "magic" of exploration is not guaranteed. Exploration needs to be organized and slack resource is needed to nourish the emergence of new ideas. Organized exploration explicitly acknowledges and encourages team members' desire to learn but at the same time it decouples learning from development activities so that team members can focus and separate exploitation from exploration.

5.4. Agile Organizing Framework

The discussion is summarized in Table 5 in which the enablers and inhibitors of agility, and the emergent capabilities of agile teams are presented. The enablers, when deployed properly, should help teams find—and remain in—the region of emergent complexity ("edge of chaos"). The inhibitors will make it difficult to achieve the region of emergent complexity, whether it be through contributing to stasis (e.g., over-communication between developers) or to chaos (e.g., over-responding to unplanned disturbances). The agile inhibitors suggest that traditional development methods, far from having too much structure, often lack structure in key areas, leading to the use of local organizing practices (see Table 4). Where agile teams embrace change and uncertainty by constant planning to achieve stability, traditional teams, by contrast, are plan-driven, and see unforeseen events as disturbances to be managed on an exceptional basis. In a volatile environment it seems reasonable to conjecture that traditional projects may display chaotic behavior, rather than order, as they seek to cope with unexpected and unwelcomed events. But being agile is not easy. Firstly, Pongo find the continual team and individual effort can be difficult to maintain. As Brown and Eisenhardt (1998) point out, time-pacing is relentless and if a business is not setting its own pace then it will be driven by the actions of its competitors resulting in greater uncertainty and the increasing likelihood of "death-marches" to catch up (p. 188). Secondly, although effective communication with customers and between team members is essential to being agile, over-long communication and extensive discussion can lead to stasis and a consequent deterioration in the coevolutionary process. Third, the project manager has an on-going struggle to not be externalized from the project team; the project manager needs to continue to live inside the team at the same time as fulfilling the role as a facilitator of a self-organizing team. Consequently, the effectiveness of agile methods use, as Maruping et al. (2009) argue, is contingent. We need to examine closely how practices are implemented rather than relying on the simple classifications such as agile versus nonagile (traditional) practices (Conboy 2009).

By drawing on a theoretical framework that is grounded in CAS, the agile capabilities reported in Table 5 go beyond the advocational literature found in the agile field (Baskerville and Pries-Heje 2004) and point to new and promising directions for future investigation. In this study, we have enriched the three coevolving systems principles of Volberda and Lewin (2003) by explicitly establishing a link between the three principles and emergent agile capabilities, namely: coevolution for business value, sustainable working with rhythm, collective mindfulness, sharing of learning, process adaptation and improvement, and product innovation. Although many of the agile enablers are known from the literature, the research identifies a particular and under-developed theme in agile software development—metronomic time-pacing versus emergent rhythm—that has hitherto been given little attention. Given that the three principles identified by Volberda and Lewin are a coherent and mutually self-reinforcing set of ideas, it seems reasonable to expect that all of the six capabilities will need to be present in some mix for a team to be truly agile. For example, the coevolution of business value will likely require the development team to be innovative while process adaptation and improvement will likely require collective mindfulness. The resulting theoretical framework and its

Table 5 Agile Organizing Framework—Enablers, Inhibitors, and Emergent Capabilities

Coevolving, self-renewing principles	Agile team capabilities	Agile enablers	Agile inhibitors
Principle 1: Match coevolutionary change rate	Coevolution of IT team and customer to create business value	Driven by evolving business value: • Continuous gathering of requirements • Frequent, iterative delivery of business value • Close, effective customer interaction	• Management dictating and signing off requirements • Requirements identified up-front of the project • Weak IT/business relationship • Over-communication between team and customer
	Sustainable working with rhythm	Change is embedded in and core to development: • Time-pacing through short, fixed-length iterations (e.g., one week) • Regular and frequent breaks and closure • Planning using small units of time (e.g., 30 minutes) • Multilevel planning and replanning (daily, iteration, release) • Small granularity of requirements	• Event-pacing by planned events (e.g., end of scoping, end of analysis) and unplanned disturbances (e.g., major change to user requirements mid-project) • Elaborate change control procedures peripheral to the development process • Unsustainable time-pacing • Up-front planning for the whole project and following the plan rigidly • Large granularity of requirements, deliverables, plans
Principle 2: Optimize self-organizing	Collective mindfulness	Self-management and team-discipline: • Shared responsibility for project management • Team discipline through peer and self-observation	• Centralized project management which is external to the team members • Project manager becomes a bottleneck • Project manager externalized from team
	Sharing and team learning	Supportive structures for communication and collaboration visible to the team: • Formed by interconnected practices (e.g., learning-oriented task self-assignment supported by daily meeting, pair programming, and pair rotation) • Fostered by open working spaces • Multiskilling (e.g., no separation of functional roles)	• Over-reliance on informal communication and collaboration • Tasks allocated centrally by project manager with little consultation of team • Isolated communication and collaboration depending on the willingness and attitudes of individual developers • Over-communication between team members
Principle 3: Synchronize exploitation and exploration	Process adaptation and improvement	Reviewing and improving process regularly: • Adapt process to development context (e.g., different iteration lengths for different projects) • Remove redundant activities • Test the process by challenging effective practices	• Development process not internalized by team members • Process is imposed by management and perceived as external to the team • Over-reliance on "common sense"
	Product innovation	Routinizing exploration: • Formalize study as a part of the development process • Allocate study time for both project and nonproject investigations	• Resource not specifically allocated to exploration • Focus on timesheets and billable project time • Exploration is not shared by the team

application to this empirical study on software development processes has demonstrated a concrete and feasible way to apply CAS-grounded theory in IS research, the importance of which has been emphasized by Anderson (1999), Jacucci et al. (2006), and Merali and McKelvey (2006).

The detailed analysis should be useful to practitioners through the identification of development practices and agile enablers and inhibitors. These practices should not be seen as a pick-and-mix menu from which managers can select only those they feel comfortable with; our research suggests that the agile capabilities are interdependent and are built through interdependent practices (and the inhibitors serve as a useful reminder of where agility might be lost). For example, the relentlessness of time-paced iterative delivery to the customer is mitigated by developers having daily study time. Of further value to practitioners is the identification of agile capabilities, which will help managers assess the extent to which a team is truly agile (and is therefore in the emergent region of complexity) through a focus on project outcomes and collaborative achievement (Maruping et al. 2009). Because agile capabilities are emergent properties of agile teams they could be used as the basis of an agility assessment method that treats the development team as a black box thus allowing managers to assess the agile capability of a team rather than its adoption and use of agile methods.

The research reported here, of course, has limitations. It relies to a large extent on observation and self-reporting of agility. Further work is therefore needed in the assessment of agility to allow informed judgements to be made concerning the effectiveness, or otherwise, of agile practices. For the organization as a whole to be agile, i.e., not just the software development team, then the customer must also have the space to evolve their business practices such that the business and technology can indeed be said to *coevolve*. The coevolution of business and IT suggests that our work on agile development may help us understand the larger business/IT alignment issue (Luftman 2005). However, to study coevolution in this way will require research designs that take greater account of the relationships between developers and business users and other stakeholders. The framework needs to be refined and tested with further cases, especially more work is needed to further validate the three organizing principles proposed by Volberda and Lewin (2003). From a research methodology perspective, although case study allows the capture of detail and the analysis of many variables, the method is criticized for difficulties of generalization and is subject to questions of external validity because of the unique characteristics of each case (Kitchenham et al. 2002). Although this is a valid criticism, we follow Walsham (1995), who argues that when using case study, researchers are not necessarily looking for generalization from a sample to a population, but rather they are looking for the plausibility and logical reasoning through developing concepts and theory, drawing specific implications, and contributing rich insight.

6. Summary

In this paper we have built an organizing framework for agile software development that gives a deeper understanding of agile practices, identifies enablers and inhibitors of agility, and shows what capabilities an agile team would be expected to possess. The framework is rooted in the empirical evidence drawn from the two software teams, one espousing a traditional waterfall model (SysCheck) and the other an agile method (Pongo). Volberda and Lewin's (2003) three principles of coevolving systems—match coevolutionary change rates, optimize self-organization, and synchronize concurrent exploration and exploitation—are used to inform the empirical investigation in the agile domain. Agile teams can be recognized by their ability to: work with customers to coevolve business value, work sustainably with rhythm, be collectively mindful, create team learning, adapt and improve the development process, and to create product innovations. A further implication of the work is that traditional projects may often lack appropriate structures which they then compensate for through informal mechanisms. Rather than being bastions of order in an uncertain world, traditional teams may indeed become chaotic should their plan-driven organization be overwhelmed by events. Future work will involve conducting more case studies of software development teams and developing methods for assessing the agility of teams based around the agile capabilities identified in the agile organizing framework.

References

Abemathy, W. J., K. B. Clark. 1985. Innovation: Mapping the winds of creative destruction. *Res. Policy* **14** 3–22.

Agile Manifesto. 2001. Accessed May 10, 2007, http://www.agilemanifesto.org/.

Aldrich, H. 1999. *Organizations Evolving*. Sage, London.

Anderson, P. 1999. Complexity theory and organization science. *Organ. Sci.* **10**(3) 216–232.

Ashmos, D. P., D. Duchon, R. R. McDaniel, Jr., J. W. Huonker. 2002. What a mess! Participation as a simple managerial rule to "complexify" organizations. *J. Management Stud.* **39**(2) 189–196.

Augustine, S., B. Payne, F. Sencindiver, S. Woodcock. 2005. Agile project management: Steering from the edges. *Comm. ACM* **48**(12) 85–89.

Baskerville, R., J. Pries-Heje. 2004. Short cycle time systems development. *Inform. Systems J.* **14**(3) 237–264.

Baskerville, R., J. Levine, J. Pries-Heje, B. Ramesh, S. Slaughter. 2001. How internet software companies negotiate quality. *IEEE Comput.* **34**(5) 51–57.

Baskerville, R. L. 2006. Artful planning. *Eur. J. Inform. Systems* **15**(2) 113–115.

Beck, K., C. Andres. 2004. *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison Wesley, Boston.

Bedoll, R. 2003. A tail of two projects: How "agile" methods succeeded after "traditional" methods had failed in a critical system-development project. *Extreme Programming and Agile Methods—XP/Agile Universe 2003*, LCNS, Vol. 2753. Springer-Verlag, Berlin, 25–34.

Boehm, B. 2002. Get ready for agile methods, with care. *IEEE Comput.* **35**(1) 64–69.

Brown, S., K. Eisenhardt. 1998. *Competing on the Edge: Strategy as Structured Chaos*. Harvard Business School Press, Boston.

Butler, B. S., P. H. Gray. 2006. Reliability, mindfulness, and information systems. *MIS Quart.* **30**(2) 211–224.

Choi, T. Y., K. J. Dooley, M. Rungtusanatham. 2001. Supply networks and complex adaptive systems: Control versus emergence. *J. Oper. Management* **19**(3) 351–366.

Coffin, R. 2006. A tale of two projects. *Agile 2006, Proc.* IEEE, New York, 155–161.

Conboy, K. 2009. Agility from first principles: Reconstructing the concept of agility in information systems development. *Inform. Systems Res.* **20**(3) 329–354.

Conboy, K., B. Fitzgerald. 2004. Toward a conceptual framework of agile methods. *Proc. Extreme Program. Agile Methods—XP/Agile Universe 2004, Calgary, Alberta, Canada*.

Dani, M., A. Gualfetti, L. Mengoni, F. Cirilo. 2003. How we became the Pongo team. *XP2003 Conf., Genova, Italy*.

Dawande, M., M. Johar, S. Kumar, V. S. Mookerjee. 2008. A comparison of pair versus solo programming under different objectives: An analytical approach. *Inform. Systems Res.* **19**(1) 71–92.

DTI. 2003. Innovation report—Competing in the global economy: The innovation challenge. Report, Department for Trade and Industry, London.

Eisenhardt, K., D. Galunic. 2000. Coevolving: At last, a way to make synergies work. *Harvard Bus. Rev.* **78**(1) 91–101.

Fitzgerald, B., G. Hartnett, K. Conboy. 2006. Customising agile methods to software practices at Intel Shannon. *Eur. J. Inform. Systems* **15**(2) 200–213.

Freeman, C. 1994. Critical survey: The economics of technical change. *Cambridge J. Econom.* **18** 463–514.

Fruhling, A., G. J. De Vreede. 2006. Field experiences with extreme programming: Developing an emergency response system. *J. Management Inform. Systems* **22**(4) 39–68.

Gersick, C. 1994. Pacing strategic change: The case of a new venture. *Acad. Management J.* **37**(1) 9–45.

Haeckel, S. 1999. *Adaptive Enterprise: Creating and Leading Sense-and-Respond Organizations*. Harvard Business School Press, Boston.

Highsmith, J. 2000. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing, New York.

Highsmith, J. 2002. *Agile Software Development Ecosystems*. Addison-Wesley, Boston.

Highsmith, J., A. Cockburn. 2001. Agile software development: The business of innovation. *Computer* **34**(9) 120–122.

Iivari, J., R. Hirschheim, H. K. Klein. 1998. A paradigmatic analysis contrasting information systems development approaches and methodologies. *Inform. Systems Res.* **9**(2) 164–193.

Jackson, A., S. L. Tsang, A. Gray, C. Driver, S. Clarke. 2004. Behind the rules: XP experiences. *Proc. Agile Development Conf.* IEEE Computer Soc, Los Alamitos, CA, 87–94.

Jacucci, E., O. Hanseth, K. Lyytinen. 2006. Introduction: Taking complexity seriously in IS research. *Inform. Tech. People* **19**(1) 5–11.

Janzen, D. 1980. When is it coevolution? *Evolution* **34**(3) 611–612.

Kauffman, S. 1993. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, New York.

Kitchenham, B., S. L. Pfleeger, L. Pickard, P. Jones, D. C. Hoaglin, K. E. Emam, J. Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Software Engrg.* **28**(8) 721–734.

Levinthal, D. A., J. G. March. 1993. The myopia of learning. *Strategic Management J.* **14** 95–112.

Luftman, J. 2005. Key issues for IT executives 2004. *MIS Quart. Executive* **4**(2) 269–285.

March, J. G. 1991. Exploration and exploitation in organizational learning. *Organ. Sci.* **2**(1) 71–87.

Maruping, L. M., V. Venkatesh, R. Agarwal. 2009. A control theory perspective on agile methodology use and changing user requirements. *Inform. Systems Res.* **20**(3) 377-399.

McKelvey, B. 2003. MicroStrategy and MacroLeadership: New science meets distributed intelligence. A. Y. Lewin, H. W. Volberda, eds. *The Coevolution Advantage: Mobilizing the Self-Renewing Organization*. M. E. Sharpe, Armonk, NY.

McMillan, E. 2004. *Complexity, Organizations, and Change*. Routledge, Taylor and Francis Group, London.

Melao, N., M. Pidd. 2000. A conceptual framework for understanding business processes and business process modeling. *Inform. Systems J.* **10**(2) 105–129.

Merali, Y., W. McKelvey. 2006. Using complexity science to effect a paradigm shift in information systems for the 21st century. *J. Inform. Tech. Special Issue Complexity Inform. Systems* **21**(4) 211–215.

Meso, P., R. Jain. 2006. Agile software development: Adaptive systems principles and best practices. *Inform. Systems Management* **23**(3) 19–30.

Miles, M. B., A. M. Huberman. 1994. *Qualitative Data Analysis: An Expanded Sourcebook*. Sage, Thousand Oaks, CA.

Mittleton-Kelly, E. 1997. Organisations as co-evolving complex adaptive systems. *British Acad. Management Conf., BPRC* (*Business Processes Resource Center*), Paper Series, No. 5.

Mittleton-Kelly, E. 2003. Ten principles of complexity and enabling infrastructures. *Complex Systems and Evolutionary Perspectives of Organisations: The Application of Complexity Theory to Organisations*. Pergamon, Oxford, UK.

Nonaka, I. 1988. Creating organizational order out of chaos: Self-renewal in Japanese firms. *California Management Rev.* **30**(3) 57–73.

Poole, C., J. Huisman. 2001. Using extreme programming in a maintenance environment. *IEEE Software* **18**(6) 42–50.

Pressman, R. S. 1997. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York.

Prigogine, I., I. Stengers. 1984. *Order Out of Chaos: Man's New Dialog with Nature*. Flamingo, London.

Rakitin, S. 2001. Manifesto elicits cynicism. *IEEE Comput.* **34**(12) 4.

Rakitin, S. 2005. Agile methods—Beyond the hype. *Food for Thought Newsletter from Software Quality Consulting* **2**(7). Accessed April 5, 2007, http://www.swqual.com/newsletter/vol2/no7/vol2no7.html.

Rasmusson, J. 2003. Introducing XP into Greenfield projects: Lessons learned. *IEEE Software* **20**(3) 21–28.

Schach, S. R. 1998. *Software Engineering with JAVA*. McGraw-Hill, New York.

Schatz, B., I. Abdelshafi. 2005. Primavera gets agile: A successful transition to agile development. *IEEE Software* **22**(3) 36–41.

Schwaber, K. 1996. Controlled chaos: Living on the edge. *Amer. Programmer* **9**(5) 10–16.

Schwaber, K., A. Beedle. 2002. *Agile Software Development with SCRUM*. Prentice-Hall, Upper Saddle River, NJ.

Sfetsos, P., L. Angelis, I. Stamelos. 2006. Investigating the eXtreme programming system—An empirical study. *Empirical Software Engrg.* **11**(2) 269–301.

Sharp, H., H. Robinson. 2004. An ethnographic study of XP practice. *Empirical Software Engrg.* **9**(4) 353–375.

Stacey, R. D. 2003. *Strategic Management and Organisational Dynamics: The Challenge of Complexity*, 4th ed. Financial Times, Prentice Hall, Harlow, UK.

Stephens, M., D. Rosenberg. 2003. *Extreme Programming Refactored: The Case Against XP*. Apress, New York.

Streatfield, P. 2001. *The Paradox of Control in Organizations*. Routledge, London.

Turk, D., R. France, R. Bernhard. 2002. Limitations of agile software processes. *Proc. 3rd Internat. Conf. eXtreme Programming Agile Processes Software Engineering*. Alghero, Sardinia, Italy.

Vidgen, R., X. Wang. 2006. Organizing for agility: A complex adaptive systems perspective on agile software development process. *Proc. 14th Eur. Conf. Inform. Systems, Göteborg, Sweden*.

Volberda, H. W., A. Y. Lewin. 2003. Guest editors' introduction co-evolutionary dynamics within and between firms: From evolution to co-evolution. *J. Management Stud.* **40**(8) 2111–2136.

Walsham, G. 1995. Interpretive case studies in IS research: Nature and method. *Eur. J. Inform. Systems* **4**(2) 74–81.

Wang, X., R. Vidgen. 2007. Chaos and order in agile software development: A comparison of two software development teams in a major IT company. *Proc. 15th Eur. Conf. Inform. Systems*. St. Gallen, Switzerland.

Wilkinson, I., L. Young. 2003. A view from the edge. *Marketing Theory* **3**(1) 179–181.

Yin, R. K. 2003. *Case Study Research: Design and Methods*. Sage, Thousand Oaks, CA.

Zmud, R. W., L. E. Apple. 1992. Measuring technology incorporation/infusion. *J. Product Innovation Management* **9**(2) 148–155.