Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Balancing Self-Organizing Agile Teams: A Grounded Theory

blinded for review · blinded · blinded

Received: date / Accepted: date

Abstract Self-organizing teams are one of the critical success factors on Agile projects but unfortunately little is known about the self-organizing nature of Agile teams and the challenges they face in industrial practice. Based on a Grounded Theory study of 40 Agile practitioners across 16 software development organizations in New Zealand and India, we describe how self-organizing Agile teams perform balancing acts between (a) freedom and responsibility (b) cross-functionality and specialization, and (c) continuous learning and iteration pressure, in an effort to maintain their self-organizing nature. We use our study to demonstrate the application of Grounded Theory to Software Engineering. We discuss the relationship between the balancing acts and the general principles of self-organization — requisite variety, redundancy of functions, minimum critical specification, and learning to learn — from the organizational perspective. We then discuss the balancing acts in relation to the specific conditions of self-organization — autonomy, cross-functionality, and self-transcendence — as applied in Agile software development. In doing so, we discuss the major challenges we encountered while performing Grounded Theory's various activities, and present our strategies for overcoming these challenges.

Keywords Empirical Research · Software Engineering · Grounded Theory · Agile Software Development · Self-organizing

blinded for review

1 Introduction

Software engineering researchers are now exploring the structure and behaviour of Agile software development teams (Cockburn, 2003; Moe et al., 2008; Nerur, 2005; Sharp and Robinson, 2004), partly in response to the Agile software movement's increasing popularity within industry over the past decade (Begel and Nagappan, 2007; Nerur, 2005; Dybåand Dingsoyr, 2008). However, while self-organization is a vital characteristic of Agile teams, there has been limited research on the subject and almost none across multiple projects, organizations, and cultures.

Agile software development methods emerged in the late 1990s (Larman and Basili, 2003). The term 'Agile' was adopted as the umbrella term for methods such as Scrum (Schwaber and Beedle, 2002), XP (eXtreme Programming) (Beck, 1999), Crystal (Cockburn, 2004), FDD (Feature Driven Development) (Palmer and Felsing, 2001), DSDM (Dynamic Software Development Method) (Stapleton, 1997), and Adaptive Software Development (Highsmith, 2000). Scrum and XP are considered to be the most widely adopted Agile methods in the world (Pikkarainen et al., 2008). XP focuses on developmental practices, while Scrum mainly covers project management (Dybåand Dingsoyr, 2008). The developers of some of these methods collaboratively wrote the Agile Manifesto (Highsmith and Fowler, 2001) that values "*individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, responding to change over following a plan.*" The Manifesto goes on to say "*that is, while there is value in the items on the right, we value the items on the left more.*"

Agile software development methods follow an iterative and incremental style of development where collaborative self-organizing teams dynamically adjust to changing customer requirements (Dybåand Dingsoyr, 2008; Martin, 2002). The principles behind the Agile Manifesto (Highsmith and Fowler, 2001) include fast, frequent, consistent, and continuous delivery of working software; responding to changing requirements; encouraging effective communication; and motivated and well-supported self-organizing teams.

Agile teams are self-organizing teams (Chow and Cao, 2008; Cockburn and Highsmith, 2001; Highsmith and Fowler, 2001; Martin, 2002; Schwaber, 2009; Sharp and Robinson, 2008) composed of "*individuals [that] manage their own workload, shift work among themselves based on need and best fit, and participate in team decision making.*" (Highsmith, 2004). Takeuchi and Nonaka (1986) describe self-organizing teams as exhibiting autonomy, cross-fertilization, and self-transcendence. Self-organizing teams must have common focus, mutual trust, respect, and the ability to organize repeatedly to meet new challenges (Cockburn and Highsmith, 2001). Self-organizing teams are not leaderless, uncontrolled teams (Cockburn and Highsmith, 2001; Takeuchi and Nonaka, 1986). Leadership in self-organizing teams is meant to be light-touch and adaptive (Augustine et al., 2005), providing feedback and subtle direction (Anderson et al., 2003; Chau and Maurer, 2004; Takeuchi and Nonaka, 1986). Leaders of Agile teams are responsible for setting direction, aligning people, obtaining resources, and motivating the teams (Anderson et al., 2003). Agile projects have job titles such as Scrum Masters (Schwaber and Beedle, 2001) and (XP) Coaches (Fraser, 2003) instead of traditional managers.

Self-organizing teams have been identified as one of the critical success factors of Agile projects (Chow and Cao, 2008). Self-organization can also directly influence team effectiveness (Moe and Dingsoyr, 2008) decision making authority is brought to the level of operational problems, which increases the speed and accuracy of problem solving. Moe et al. (2008) note that Scrum emphasizes self-organizing teams but does not provide clear guidelines on how they should be implemented.

The aim of this paper is to demonstrate an application of Grounded Theory to Software Engineering. First we present the methodology as applied to Grounded Theory study of 40 Agile practitioners across 16 software organizations in New Zealand and India conducted over a period of 3 years. Second, we present a portion of the findings of our study: a grounded theory of the balancing acts performed by self-organizing Agile teams. Third, we place our research findings in relation to the general principles of self-organization from an organizational perspective and to the specific conditions of self-organization as applied to Agile software development. Finally, we discuss the major challenges we encountered while performing Grounded Theory's various activities, and present our strategies for overcoming these challenges.

The rest of the paper is structured as follows: section 2 describes the application of Grounded Theory in our research. Section 3 presents the research findings, followed by a discussion of related work in section 4. Section 5 describes the limitations of the study and criteria for evaluating a grounded theory. The paper concludes in section 6.

2 Grounded Theory

Grounded Theory (GT) is the systematic generation of theory from data analyzed by a rigorous research method (Glaser, 1978, 1998). GT was developed by sociologists Glaser and Strauss (1967) as a result of their collaborative research on dying hospital patients. They published "The Discovery of Grounded Theory" (1967) which laid the foundations of GT. Glaser defines GT as:

> "*The grounded theory approach is a general methodology of analysis linked with data collection that uses a systematically applied set of methods to generate an inductive theory about a substantive area.*" (Glaser, 1992)

The aim of GT is to generate a theory as an interrelated set of hypotheses generated through constant comparison of data at increasing levels of abstraction (Glaser, 1992). In generating a theory, the GT researcher uncovers the main concern of the research participants and how they go about resolving it. The distinguishing feature of the GT method is the absence of a clear research problem or hypothesis up-front, rather the researcher tries to uncover the research problem as the main concern of the participants in the process (Allan, 2003; Parry, 1998).

Differences between the two originators of Grounded Theory led to the emergence of two versions of the Grounded Theory method: Glaser's version of GT, often referred to as classic GT and Strauss's version, called Straussian GT (Glaser, 2004). We have employed classic Grounded Theory or the Glasserian version mostly because our mentors (co-authors) were better conversant in this version and due to a larger number of resources available (Glaser, 2010).

We chose GT as our research method for several reasons. Firstly, Agile methods focus on people and interactions and GT, used as a qualitative research method, allows us to study social interactions and behaviour (Parry, 1998). Secondly, GT is most suited to areas of research which have not been explored in great detail before, and the research on self-organizing nature of Agile teams is limited. Thirdly, GT is one of the few research methods that focuses on theory generation, rather than extending or verifying existing theories. Finally, GT is being increasingly used to study Agile teams (Cockburn, 2003; Coleman and OConnor, 2007; Martin et al., 2009; Whitworth and Biddle, 2007).

We now describe our application of the GT method in detail, while critically reflecting on the major challenges of performing its various components and the strategies we found useful in overcoming these challenges.

In the following sections, we describe the main components of the GT method as applied in our research. We also present the challenges faced in applying the GT method in Software Engineering research and the strategies we found useful in overcoming them.

2.1 Area of Interest and the Research Question

In order to study effectively and uncover the main problems of the participants, the researcher is recommended to refrain from formulating a research problem or question up front (Glaser, 1978). The rationale behind such a recommendation is (a) the GT method is meant to generate theory and having a preconceived research problem can cause the research to be influenced by the existing research literature in the area; and (b) the research problem should be the problem of the participants under study and should not be preconceived or forced, rather it should be be allowed to emerge.

Although the researcher is advised against formulating a research question up front, they are required to choose a general area of interest. The plethora of subject areas within Software Engineering makes choosing one a daunting task. We picked *Agile Project Management* as our general area of interest primarily due to our exposure to and subsequent interest in Agile software development during our Masters study.

The challenge facing the Software Engineering (SE) researcher is that it can become difficult for the new researcher to not spell out a specific research question when asked by his peers or other researchers or even the participants themselves. It can appear to be a lack of preparation on part of the researcher. The strategy we found useful in this context was making sure that we explained not only our area of interest to the participants but also informed them briefly about the nature of the research method. Although it was new to most people, they appreciated our effort to search for the real problems faced by the participants than to have a clear research question that was of little relevance or interest to them. Similarly, peers and other academics demonstrated an interest in learning more about the Grounded Theory method which is rather new to the Software Engineering world.

2.2 Literature Review

Once the researcher chooses their area of interest, they are faced with the dilemma of whether to conduct extensive literature review in that area (as is common is most research methods) or to move directly to data collection and analysis. In classic GT, Glaser recommends that the researcher should start right off with regular data collecting, coding and analysis without any preconceived problem, a methods chapter or an *extensive* review of (research) literature in the *same substantive area*. Glaser insists that "*undertaking an extensive literature review before the emergence of the core category violates the basic premise of GT*" (Glaser, 2004).

Glaser's stance on literature review in GT has been a topic of debate (Thomas and James, 2006; Suddaby, 2006). While GT does not involve formulating a hypothesis up front based on extensive literature review, the use of literature is not prohibited in GT. Glaser (1978) strictly warns against extensive literature review in the *same area* of research during the *early stages* of the GT method. The rationale behind a minimal literature review before the emergence of the core category is in many ways the same as that behind not starting with a specific research question, namely: avoiding clouding of researcher's mind with preconceived ideas and focusing on generating theory rather than verifying existing theories (Glaser, 1978).

If there is a particularly good theory in the substantive area of research, the researcher can cover this earlier and explore emergent fits (Glaser, 1978). However, there was no well-established theory on self-organizing teams in Agile software development and so, following Glaser's advice, we kept literature review to a minimum in the beginning. We read just enough information on Agile methods to understand the basic facts and terminology in order to effectively converse with our participants during interviews. Our deeper understanding of Agile methods and in particular the self-organizing nature of Agile teams came mostly from our participants in the early stages of our research.

While extensive literature review in the same substantive area as the research is discouraged early on, reading of substantive areas *different* from that of the research is considered vital in order to increase the researcher's ability to think theoretically in general (Glaser, 1978).

Literature can be extensively reviewed and used in later stages of the GT method. The advantage of literature review in later stages of GT is that it allows the researcher to quickly spot literature that is related to the already developed concepts and categories of the emerging theory. Besides this, we found another advantage of avoiding extensive literature review up front. Being a novice in the field made the participants feel like they are informing a novice and not being interrogated by an expert. As a result they felt comfortable in expressing their honest opinions and sharing their real experiences.

Following classic GT, we conducted our literature review on self-organizing Agile teams after we had a established an emerging theory from data collection and analysis. Our order of presentation in this paper reflects our order of application of literature review such that we present our literature review of self-organizing teams in section 5 after we describe the research findings.

The challenge that face the SE researcher here include a personal urge to do extensive literature review to gain quick knowledge about the area under study. In order to curb the urge to 'study-up' extensively, we relied mostly on our participants to inform us about Agile software development. We also read popular practitioner literature (of non-theoretical nature) to gain further understanding of new concepts and terms since Glaser suggests such literature can be considered more data (Glaser, 1978).

2.3 Data Collection: Theoretical Sampling

Having read some basic concepts in the area of interest, the researcher can move on to data collection. Data collection in GT is guided by a process called *Theoretical Sampling*:

> "*Theoretical sampling is the process of data collection for generating theory whereby the analyst jointly collects, codes, and analyzes his data and decides*

what data to collect next and where to find them, in order to develop his theory as it emerges." (Glaser, 1978)

In this section, we describe how we went about recruiting participants and conducting interviews and observations. We also reflect on our experiences of theoretical sampling, highlighting the challenges we encountered and the strategies we found useful.

2.3.1 Recruiting Participants

Before commencing data collection, we applied and received the approval of the Human Ethics Committee (HEC) at our university. Next, we went about finding participants.

Finding participants can be difficult at best and extremely challenging at worst. In the absence of an umbrella organization or user group for Agile practitioners in New Zealand in the early periods of our research, we had to resort to searching and contacting Agile companies and practitioners individually with limited success. At an event organized by some Agile companies in New Zealand we got the opportunity to meet and interact with several Agile practitioners, some of whom offered to participate in our research. It was at that very event that the foundations of an umbrella Agile group, the Agile Professionals Network (APN), were laid. It took a while before the group grew and became active and so we struggled to find more participants in New Zealand in the interim.

Therefore, eager to collect data and proceed with our research, we directed our attention to other destinations. We chose to study Agile practitioners in India primarily because it was home to a well-established and flourishing software industry with increasing number of Agile adoptions. In exploring the Indian software industry resources online, we discovered the Agile Software Community of India (ASCI). We emailed the user group with a request for participation and fortunately, several practitioners came forth to help. We then travelled to New Delhi and Mumbai to interview our first few Agile practitioners in India.

Theoretical sampling is an ongoing process which helps decide what data to collect next based on the emerging theory. Our initial participants belonged to relatively new Agile teams and as such the emerging theory was mostly based around the initial challenges of becoming a self-organizing team. Using theoretical sampling, we were able to discern gaps in our emerging theory which prompted us to study more mature teams towards later stages of our research. We were also able to see the need to include participants from different aspects of software development at different stages of our research guided by the emerging theory, such as Agile coach, developer, tester, business analyst, customer, and senior management. Data collection by theoretical sampling helped us develop our emerging theory by (a) adapting questions to focus on emerging concerns (b) choosing participants that were better placed to provide information on the emerging concerns.

There were 40 participants in our study, half of whom were from 8 New Zealand organizations and half from 8 Indian organizations. The project duration varied from 2 to 12 months and the team sizes varied from 2 to 20 people on different projects. The products and services offered by the participants organizations included web-based applications, front and back-end functionality, and local and off-shored software development services. Half the participants were practicing in India and half in New Zealand. The organizational sizes varied from 10 to 300,000 employees. In order to respect their confidentiality, we refer to our participants by numbers P1 to P40. Table 1 shows participant and project details.

Table 1 Participants and Projects (P#: Participant Number, Agile Position: Agile Coach (AC), Agile Trainer (AT), Developer (Dev), Customer Rep (Cust Rep), Business Analyst (BA), Senior Management (SM); *Organizational Size: XS < 50, S < 500, M < 5000, L $< 50{,}000$, XL $> 100{,}000$ employees)

P#	Positions	Method	Org. Size*	Location	Domain	Team Size	Project (months)	Iteration (weeks)
P1-P7	Dev X 3, BA, AC, Tester, Cust Rep	Scrum	M	NZ	Health	7	9	2
P8	Cust Rep	Scrum & XP	L	NZ	Social Services	4 to 10	3 1o 12	2
P9-P15	Dev X 5, AC, SM	Scrum & XP	S	NZ	Environment	4 to 6	12	1
P16	SM	Scrum & XP	S	NZ	E-commerce	4	2	4
P17	AC	Scrum & XP	XL	NZ	Telecom & Transportation	6 to 15	12	4
P18	Cust Rep	Scrum	XS	NZ	Entertainment	6 to 8	9	4
P19	AC	Scrum & XP	S	NZ	Government Education	4 to 9	4	2
P20	AC	Scrum & XP	XS	NZ	Software Development	8	12	1
P21-P27	Dev X 4, AC, Tester, SM	Scrum & XP	S	India	Software Development & Consultancy	5	6	2
P28-P31	Dev X 4	Scrum & XP	XS	India	Software Development	4	1	1
P32	AT	Scrum & XP	XS	India	Agile Training	7	8	3
P33-P36	AC X 4	Scrum & XP	M	India	Software Development	7 to 8	3 to 6	2
P37	AC	Scrum & XP	M	India	Financial Services	8 to 11	36	2
P38	Designer	Scrum & XP	S	India	Web-based services	5	1	2
P39	AC	Scrum & XP	L	India	Telecom	8 to 15	3	4
P40	Dev	Scrum & XP	M	India	Software Development	15	12	1

2.3.2 Interviews and Observations

The dictum in GT is that "*all is data*" and as such several sources of data can be utilized. Qualitative data derived through interviews, however, remains the most popular and the one we used.

We collected data by conducting face-to-face, semi-structured interviews with Agile practitioners using open-ended questions over a period of 2.5 years. The interviews were approximately an hour long and focused on the participants experiences of working with Agile methods, in particular the challenges faced in Agile projects and the strategies used to overcome them. We also observed several Agile practices such as daily stand-up meetings (co-located and distributed), release planning, iteration planning, and demonstrations.

The interviews were first voice recorded and then transcribed. Voice recording the interviews helped us avoid losing information and also allowed us to concentrate on the conversation rather than focusing on jotting details down. The interview transcripts served as a good starting point for analysis (explained in the next section.) Data collection and analysis were iterative so that constant comparison of data helped guide future interviews and the analysis of interviews and observations fed back into the emerging results.

Face-to-face interviews provide the opportunity to not only record verbal information but also the mannerisms, actions, expressions which add to the verbal information. Conducting Semi-structured interviews instead of completely structured ones help with emergence of the real concerns of participants rather than forcing a topic that may be viewed as trivial by the participants.

The challenge of conducting face-to-face interviews and observations is that it may be expensive to travel to the participant's workplace. We applied for and received research funding to support travelling for data collection (see Acknowledgments).

Another challenge related to data collection is that sometimes it is easy to feel overwhelmed by the sheer amount of raw data and emerging codes, concepts, and categories. The researcher may feel lost and unable to decide the next move in terms of what data to collect. Theoretical sampling narrows the focus of the study by guiding and steering it until a clear research problem emerges.

2.4 Data Analysis: Substantive Coding

The researcher can begin data analysis — called *coding* in GT — as soon as they have collected some data. There are two types of codes produced as a result of data analysis or coding: Substantive Codes and Theoretical Codes. The substantive codes are "*the categories and properties of the theory which emerges from and conceptually images the substantive area being researched* (Glaser, 2005). In contrast, theoretical codes "*implicitly conceptualize how the substantive codes will relate to each other as a modeled, interrelated, multivariate set of hypothesis in accounting for resolving the main concern*" (Glaser, 2005). In this section, we describe the coding mechanisms — *Open Coding* and *Selective Coding* — that lead to substantive codes. We also describe the Core Category that marks the end of open coding and the beginning of selective coding.

2.4.1 Open Coding

Open Coding is the first step of data analysis. We used open coding to analyze the interview transcripts in detail (Georgieva and Allan, 2008). To explain open coding, we present an example of working from interview transcripts to results for the category "Balancing Acts".

We began by collating key points from each interview transcript. Then we assigned a code — a phrase that summaries the key point in 2 or 3 words — to each key point (Georgieva and Allan, 2008):

Interview quotation: "*that is the freedom that Agile gives you. With freedom comes, of course, some responsibility and some answer-ability...we are given laptops - you can install anything on it, you can change the Operating System: you go to Linux, you go to Mac, whatever you want and that is the freedom. And also the responsibility is that you don't install some pirated software on it!*" — P22, Developer, India

Key Point: "Teams balancing freedom with responsibility"

Codes: Receiving freedom, Displaying responsibility, Balancing

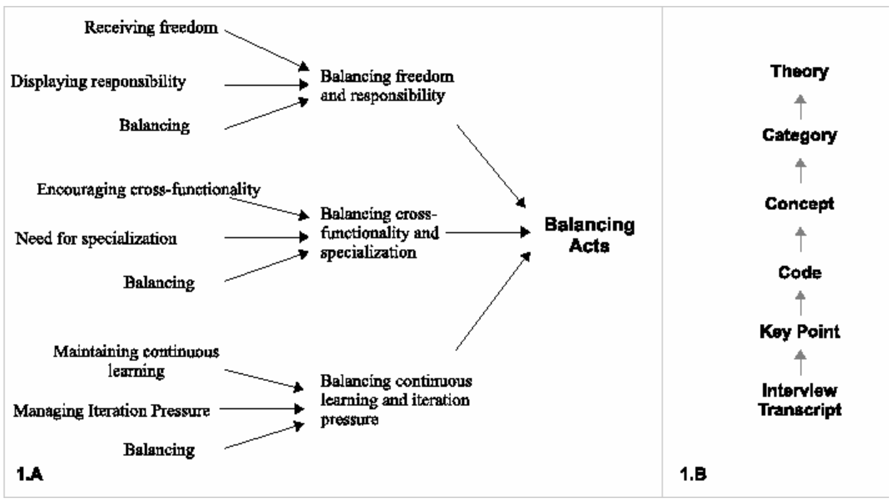
Receiving freedom
Displaying responsibility
Balancing
Balancing freedom and responsibility
Encouraging cross-functionality
Need for specialization
Balancing
Balancing cross-functionality and specialization
Maintaining continuous learning
Managing Iteration Pressure
Balancing
Balancing continuous learning and iteration pressure
Balancing Acts
1.A
Theory
Category
Concept
Code
Key Point
Interview Transcript
1.B

Fig. 1 .A Emergence of Category "Balancing Acts" from underlying concept 1.B Levels of Data Abstraction in GT

The codes arising out of each interview were constantly compared against the codes from the same interview, and those from other interviews and observations. This is GT's Constant Comparison Method (Glaser and Strauss, 1967; Glaser, 1992) which was used again to group these codes to produce a higher level of abstraction, called concepts in GT.

Concept: Balancing freedom and responsibility

Other concepts that emerged in a similar fashion include: "balancing cross-functionality and specialization" and "balancing continuous learning and iteration pressure". The constant comparison method was repeated on these concepts to produce another level of abstraction called a Category. As a result of this analysis, these concepts gave rise to the category Balancing Acts. Figure 1.A shows the emergence of the category "Balancing Acts" from underlying concepts and 1.B depicts the levels of data abstraction in GT.

Category: Balancing Acts

We analyzed the observations and compared them to the concepts derived from the interviews. We found our observations did not contradict but rather supported the data provided in interviews, thereby strengthening the interview data.

Line by line data analysis is more effective and useful than word-by-word analysis which can be tedious and potentially misguiding (Allan, 2003). The use of key points made it easy to focus while conducting coding (Allan, 2003). As per the rules of open coding, we did our own analysis of data. Furthermore, in order to preserve consistency in the application of the method, all the data was personally collected and analyzed by the same individual, the primary researcher.

The challenge for the SE researcher in applying open coding is that often with little sociological and theoretical training, deriving codes, concepts, and categories in the above manner can be difficult. SE researchers may not be used to thinking in social terms. We tried to overcome this problem by thinking of the constant comparison method as a model for data abstraction and normalization. Once we were able to view the process as a Software Engineering method of abstraction, it became easier to apply it. One of the advantages of an SE researcher using GT is that we are well-trained in analytical thinking and abstraction. The ability to repeatedly raise codes in higher levels of abstraction is something we are familiar with. We found this ability extremely relevant when employing the constant comparative method.

The SE researcher can also become overwhelmed as one set of data (transcripts) seems to get converted to another set of data (codes) repeatedly. Growing number of interviews means increasing amounts of codes which can be further confusing. The strategy we found useful when conducting open coding was asking few questions of the data as suggested by Glaser (1998): "what is this data a study of?", "what category does this incident indicate?", "what is actually happening in the data?", "what is the main concern being faced by the participants?" and "what accounts for the continual resolving of this concern?"

Finally, some GT researcher use software research tools such as NVivo (NVivo, 2010) to conduct their analysis. We tried using this software but found that its structural framework limited the ways in which we could organize and interact with the data. Arguably other researchers may find it useful due to the same reason (Parry, 1998). Initially, we preferred the ease of using spreadsheets since it gave us greater freedom in organizing the data. As the data grew in volume, the spreadsheets got difficult to maintain. Finally, we settled for using print-outs of the interview transcripts and physically coded along the margins using pen on paper.

2.4.2 Core Category

The end of open coding is marked by the emergence of a Core category (Glaser, 1992). The core is the category that "*accounts for a large portion of the variation in a pattern of behaviour*" (Glaser, 1978) and is considered the "main theme" or "main concern or problem" for the participants (Glaser, 1978).

There are several criteria for choosing a category as the core. Some of these criteria are: it must be central and related to several other categories and their properties; it must reoccur frequently in the data; it takes the longest to saturate; it relates meaningfully and easily with other categories. (Glaser, 1978). We found the category that passed all the criteria for core was "becoming a self-organizing team".

The core category captures the main concern of the participants which becomes the research problem. The challenge for the SE researcher, however, is that discovering the core can be time consuming and tedious. In absence of a core category, the SE researcher can easily feel confused and lost. Trusting the core to emerge is perhaps the most demanding of the whole GT process. The solution is to continue patiently and rigorously with constant comparisons and writing of theoretical memos (explained later in section 2.5) and as Glaser reassures enumerable times, "*it just has to emerge*" (Glaser, 1978). The moment of eureka experienced when discovering the core is truly worth the patience and toil.

Another challenge is the difficulty in discerning the core from near-core categories. In our case, "lack of customer involvement" was one of the most common concern of the participants and looked promising to be the core. The solution to expose 'red-herrings' (a near-core appearing to be the core category) is to return to the list of criteria governing the core category (described above). In checking the category "lack of customer involvement" against the core-criteria list, it did not meet all the criteria, in particular it didn't account for most variations in data. In fact, it became apparent that "lack of customer involvement" was one of the challenges faced in the process of becoming a self-organizing Agile team.

Usually, it is not possible to summarize the entire GT research findings in a single paper and so we found it useful to focus on different aspects of the results in dedicated papers. In this paper, we present the balancing acts, which are an integral part of becoming a self-organizing Agile team.

2.4.3 Selective Coding

Once the core category is established, the researcher ceases open coding and moves into Selective Coding, a process which involves selectively coding for the core variable by delimiting the coding to "*only those variables that relate to the core variable in sufficiently significant ways as to produce a parsimonious theory*" (Glaser, 1978, 2004). The core category guides further data collection, analysis, and theoretical sampling (Glaser, 1978).

We found selective coding to be much easier as compared to open coding because of three reasons: (a) by the time we reached the selective coding stage, we were already familiar with the constant comparison method (b) we were far more confident in our application of GT in general as compared to when we had started (c) it was much easier to selectively code for only those categories that related to the core rather than continue coding for all categories.

When further data collection and analysis on a particular category leads to a point of diminishing results, the category is said to have reached Theoretical Saturation (Glaser, 1992). The researcher can stop collecting data and coding for that category.

2.5 Memoing

2.5.1 Generating Memos

Memoing is the ongoing process of writing theoretical memos throughout the GT process. Memos are "*theoretical notes about the data and the conceptual connections between categories written down as they strike the researcher*" (Glaser, 1978). Memoing is considered a "core stage" or "the bedrock" of theory generation (Glaser, 1978).

Memos tend to be free-flowing ideas of the researcher about the codes and their relationships and they should not be mixed with data. We wrote memos as and when we had ideas about the emerging codes and their relationships. As recommended by Glaser (1978), we would often interrupt coding and other activities to capture our ideas in a memo as they came to us. An example memo on "cross-functionality" is depicted below:

> “ cross-functionality may not only imply the teams' ability to help with or perform each other's tasks, but also refers to their mere understanding of each other's tasks and perspective. If the developer is able to understand the testers work (aim, goal, what they are looking for) then they can help not by performing the testing, but doing their job (development) while keeping the tester's perspective in mind - so they

> would handle certain problems before passing the code to the tester. This makes the tester's job easier simply because the developer understood the (testers) perspective better (example: JC-developer helping JB-tester). Despite cross-functionality in the team, there is always room for specialists due to demands of specific technology or expertise (ZL-CA). The ideal situation would be lite and unobtrusive cross-functionality with room for specialization as required - a balance ”

We found that memoing was a low-tech and powerful way to allow us to pour-out all the ideas and thoughts in our mind about a certain code, concept, or category. With further data collection and analysis, we were able to modify the memos to reflect new ideas. It allowed us to see relationship between different concepts and later, between different categories as we noted down the similarities or differences between each or how one effected the other. We found it most useful to record memos electronically on the computer because it allowed us to store, search, retrieve, and edit memos with greater ease than using pen and paper.

The challenge for SE researchers in this component of GT is that they may not be able to express their ideas well enough in writing due to potential lack in such training. We found our natural inclination towards literature an advantage because we were already used to writing articles, poems, and stories which are all forms of articulating ideas into words. A SE researcher with little knowledge or inclination towards writing, on the other hand, can think of memoing as 'thinking aloud'. We should not be bothered about format, structure, spelling (note spelling of 'lite' in above example), or style; rather we should focus on getting our ideas down.

Another related challenge is that memoing can easily become an exercise in tracing where the codes originated (Allan, 2003). One way we managed to avoid this was by avoiding to write about the participants and instead focused on the codes and concepts. In our example above, we do refer to some participant identifiers but only as a reminder of their context. The main focus of the memo is the concept 'cross-functionality'.

2.5.2 Sorting Memos

Once the researcher has nearly finished data collection and coding is almost saturated, they can begin conceptually sorting the theoretical memos. Sorting of the memos forms a theoretical outline. Sorting is an "essential step" and "can't be missed" (Glaser, 1978). The advantage of sorting is that it "*puts the fractured data back together*" (Glaser, 1978). The researcher should bear in mind that they need to sort ideas not data. They should avoid sorting memos in chronological order. The sorting should be done on a conceptual level resulting in an outline of the theory in terms of how the different categories relate to the core-category.

2.6 Theoretical Coding

In this section, we describe the final step of coding, also known as Theoretical Coding that yields theoretical codes. Theoretical coding is defined as "*the property of coding and Constant comparative analysis that yields the conceptual relationship between categories and their properties as they emerge.*" (Glaser, 1992). Although theoretical codes are not strictly necessary, but "*a GT is best when they are used.*" (Glaser, 2005).

Theoretical coding involves conceptualizing how the categories (and their properties) relate to each other as a hypotheses to be integrated into a theory (Glaser, 1978). Following Glasers recommendation, we employed theoretical coding at the later stages of analysis (Glaser, 1992), rather than being enforced as a coding paradigm from the beginning as advocated by Strauss and Corbin (1990).

The challenge for the SE researcher is that generally they are not used to thinking theoretically and the theoretical framework underlying the substantive data may not be become visible. In order to overcome this problem we reminded ourselves that theoretical codes are based on sorting memos and not data (Glaser, 2005). By staying open to the emerging concepts and categories and reading about different theoretical codes, we became sensitive to seeing theoretical codes.

Glaser lists several common structures of theories known as theoretical coding families (Glaser, 1992, 2005). Some for these include: The Six C's (causes, contexts, contingencies, consequences, covariances, and conditions); Process (stages, phases, passages etc); Degree family (limit, range, intensity, etc); Dimension family (dimensions, elements, divisions, etc); Type family (type, form, kids, styles, classes, genre) and lots more.

By comparing our data with the theoretical coding families, it emerged that the coding family best 'fit' to describe our findings about the practices of self-organizing Agile teams was '*Balancing*' (Glaser, 2005). Glaser describes Balancing as "*handling many variables at once in order to start an action, keep an action going or achieve a resolution. One gets an equilibrium between all the variables*" (Glaser, 1992). In our research, by balancing — freedom and responsibility; cross-functionality and specialization; and continuous learning and iteration pressure — Agile teams achieved an equilibrium which was: self-organization.

Table 2 Overview of Balancing Acts

Balancing Acts Overview	
Balancing freedom and responsibility	When committing to a team goal During self-assignment
Balancing cross-functionality and specialization	Across functional roles Across technical areas of expertise
Balancing continuous learning and iteration pressure	Through pair-in-need Through retrospectives

3 Findings: Balancing Self-Organizing Agile Teams

In the following sections we present our research findings — the balancing acts (authors' previous work blinded for review). The term Balancing Acts emerged as a result of our data analysis (as described in section 3.4.1) to describe the efforts made by self-organizing teams to balance between different (and often contrasting) concepts, such as between freedom and responsibility. In each of the balancing acts, we describe how Agile teams were able to achieve a balance and the consequence of imbalance. We have selected quotations drawn from our interviews that serve to highlight the results and that are spread across most participants. Due to space reasons we cannot describe all the underlying key points, codes, and concepts from our interviews and observation that further ground the discussion. An overview of the balancing acts is provided in Table 2.

3.1 Balancing Freedom and Responsibility

Agile teams need senior management support, in terms of freedom, to manage their own affairs. Agile teams facing a management that was still dictating terms are unlikely to self-organize:

> "*[If] they are forced to commit to a goal that they didn't believe in - because of management pressure...if you dont give that freedom...if you have micro-management, how can you expect people to be self-organizing? How can they take ownership of what they commit to?...[if] you have somebody from management who sits over it, who dictates it, that takes out the self-organizing nature.*" — P25, Agile Coach, India

Agile teams were provided freedom by their senior management to organize themselves, giving them a concrete sense of empowerment because of their ability to self-assign, self-commit, self-manage, self-evaluate, and self-improve. Participants found that the "*team is making a lot more decisions*" (P8) and "*every person is contributing to the decision-making*" (P20).

> "*If the team is really at the peak of self-organization - the developers are also empowered, everybody is empowered- they can make decisions. If you don't have the scrum master - he's on vacation or something - then if that's not the case you'd expect everything to stop, right? but it doesn't stop - it goes on.*" — P25, Agile Coach, India

When Agile teams are provided freedom by the management to organize and manage their own affairs, it fosters "*self-monitoring, self management, higher levels of commitments and responsibility*" (P34). Participants mentioned that team members were "*putting their hand up to do stuff*", they "*get better [at] organization*", and at the same time there was "*a lot more ownership*" and "*sense of responsibility and accountability*" (P20, P32). Agile teams are aware of their responsibility to adhere to Agile practices, responsibility towards other each other, and responsibility to achieve team goals. There were many project-related activities that required self-organizing teams to perform the balancing act between freedom and responsibility. We describe two most common ones below:

When Committing to a Team Goal

Agile methods require customer representatives to provide project requirements in the form of user stories. These user stories are broken down into developmental level tasks by the teams (Beck, 1999; Schwaber and Beedle, 2002). While Agile methods grant customers the ability to prioritize user stories every iteration, the decision of how many stories will be attempted in an iteration (developmental pace or team velocity) is ideally the teams decision, based on their capability:

> "*We have stories which we estimate complexity of and we say 'well, we can fit this much complexity into next two weeks'*" — P8, Agile Coach, NZ

We found that Indian teams enjoyed the same freedom to plan their iterations and commit to a team goal as their NZ counterparts:

> "*We are participating in all the sprint planning activities and we have a clear say in that okay we'll be able to do this particular stuff in this particular sprint or we have some extra load on us or not.*" — P26, Tester,India

We found that teams in both cultures enjoyed the freedom to set the team goal and at the same time realized their responsibilities to ensure they achieved the set iteration goal as a team through a collaborative effort:

> "*The sprint is a commitment of the team...it's a team effort as opposed to an individual effort.*" — P2, Developer, NZ

> "*We are given responsibility and were given complete freedom....At the end of the day [management] wants the tasks to be done but [they] want that we do it our way...thats [how] we are self-organizing.*" — P28, Developer, India

During Self-Assignment

Committing to a team goal every iteration is the group decision. Self-assignment of tasks within the committed user stories, on the other hand, is an individual decision. Most participants appreciated the freedom they had to be able to pull the tasks from the story board and self-assign them (P9, P11, P21, P24, P25, P28, P30, P32).

> "*[In]Agile teams its all about pull instead of push so...you will define tasks yourself and as soon as you are done with the current task, you pick up a new one. Thats how it works.*" — P21, Developer, India

Teams have the freedom to pick tasks but they also bear the responsibility to pick tasks based on business value first, as recommended by Agile methods:

> "*So focus is on delivering business value as soon as possible - as a result of that you take items which are most required from point of view of business.*" — P24, Developer, India

In situations were several tasks are of the same business value or priority, individuals realize their responsibilities towards other team members and avoid picking tasks based on ease of implementation. The story board - a physical paper chart posted on the wall; containing all stories, tasks, their states, and names of individuals working on them - provides high visibility and transparency about task assignments (Sharp and Robinson, 2008) such that "when someone looks to the board they can see who is working on which task" (P9). The visibility and transparency provided by the story board reinforces the need to pick tasks responsibly:

> "*Youre assigning to yourself but youre part of this team of people so you know that people aren't stupid...we joke about choosing a particular thing and we laugh about them being easy or not.*" — P11, Developer, New Zealand

"*Individuals sign up for easy stories [is] visible, [there is more] sense of responsibility*" — P32, Agile Coach, India

Individuals try to avoid potential conflicts around task dependencies. For instance, members in some teams unofficially announced the task as they picked it from the board such that any potential conflict could be easily raised by others and mutually resolved. Such actions displayed responsibility towards team members.

Using the freedom available in an Agile environment with responsibility requires "*people to be proactive and do things for themselves*" (P13) and "*assign[ing] to themselves needs maturity*" (P33). Relatively inexperienced Agile teams had issues with accepting autonomy and kept looking up to their seniors and Agile coaches for guidance and decision making. More mature Agile teams (fluent in use of Agile practices, for usually more than a year), however, valued their freedom and were able to effectively balance the freedom they had been provided by senior management with the responsibility.

Consequence of Imbalance

The importance of balancing freedom and responsibility was most apparent when a team was unable to use their freedom in a responsible manner forcing senior management to intervene. For example, the general manager of an Agile organization in India shared an experience where they had to interfere with a self-organizing team which was unable to balance between freedom and responsibility. The team had a couple of senior developers who were extremely proficient at their tasks but were misusing the freedom provided by the senior management to dictate and override the rest of the team. Their influence had become so strong that it led to a clear unofficial divide in the team between those that sided with them in every decision fearing fall-back and the few that still tried to be democratic. These members had clearly lost their sense of responsibility towards other team members by not including them in decision making. The Agile Coach, acting as a (blinded for review), sought senior management interference and removed those senior developers from the team. It took the rest of the team some time to return to their previous self-organizing nature. The consequence of imbalance between freedom and responsibility is senior management intervention, which may lead to restrictions on the teams self-organization.

3.2 Balancing Cross-Functionality & Specialization

Cross-functionality is the ability of team members to look beyond their area of specialization by taking an interest in activities outside their specialization. Cross-functionality allows team members to gain a more rounded vision of the project through understanding it from multiple perspectives. We found that while Agile teams generally promote cross functionality, they cannot completely dispose off specialization, as we describe later in this section. Therefore, most teams tried to balance between cross-functionality and specialization across (a) different functional roles, such as between developers and testers, and across (b) different technical areas of expertise, such as between database management and graphical user interface design. We describe these in the following subsections.

Across Functional Roles

Cross-functionality across functional roles was common among the participants and was considered to be one of the key characteristics of Agile teams:

> "*The whole thing with Agile is getting people to be more cross-disciplinary, to take an interest in somebody elses perspective, to stop this artificial division between developers and analysts and testers.*" — P17, Agile Coach, NZ

Team members in different functional roles interact and collaborate with each other in order to gain better understanding of each others functional perspectives in the larger scheme of the project. In particular, we found developers and testers were not pitched against each other, rather they interacted frequently in the interest of the project, leading to cross-functionality:

> "*If I think Im writing something that is a bit tricky then I pull the tester over to sit with him and say…this is how its looking, because they tend to have a different view on things and sometimes as a developer you forget the other view and you need to step back and get that input. So I quite like to…get them involved.*" — P13, Developer, NZ

Understanding each others perspectives meant team members could help with some parts of other functional areas within the limitations of their cross-functional abilities and after they had tended to their own specialized tasks:

> "*You choose anything that you wanted, generally testers would stick to testing first , BAs would stick to requirements first, and developers stick to development first…[but] as we progressed obviously a lot of the BA work dies down so I'll say…'can I help with development?' And someone will say 'well this bit's quite easy'…so I'll go in and just assign [it to] myself.*" — P4, Business Analyst, NZ

Across Technical Areas of Expertise

Cross-functionality across different areas of expertise (within the same functional role) allowed team members to become familiar with most technical aspects of the project so they could easily manage any area, which is consistent with XPs collective code ownership principle (Beck, 1999).

> "*So we encourage people not to get boxed into 'I only do database access stuff!'…One of our keys is that we want everyone to know as much of the code base as possible, so that if someone leaves or can't work on another problem because they're busy, someone else should be able to come in and at least feel a little bit familiar with whats going on*" - P13, Developer, NZ

Flexibility to work in multiple technical areas was welcomed by developers because it helped them maintain interest in their work. Developers in India were equally excited by the ability to pick tasks across different technical areas:

> "*From the sprint backlog you want to pick the XML parser task or you want to pick the GUI design task that is entirely up to you and that is the freedom that Agile gives you.*" — P22, Developer, India

Most Agile teams we studied were highly cohesive and cooperative, helping each other learn new skills across different technical areas:

> "*We just didn't do things based on technical skills...people would just grab whatever and if they couldn't do it themselves, they get help. And that worked well.*"
> — P11, Developer, NZ

Consequence of Imbalance

The example below serves to highlight the need for balancing between specialization and cross-functionality when working in a self-organizing team. A business analyst in a NZ team secretly misused their cross-functional programming skills and had started causing damage to the project code base. The BAs unofficial involvement in programming had caused the team several hours of rework. The Agile Coach took on a (blinded for review) role - securing senior management support to remove the business analyst. The team performance rose dramatically afterwards, as confirmed by their customer representative:

> "*[When] we got our scrum coach in...that BA was moved to another project and their contract was not renewed...Once we had [the Coachs] involvement the work got back on track - wed gone four months down the wrong road and [the team] were able to get us back to where we should be in I think about six weeks.*"
> — P7, Customer Representative, NZ

A teams failure to balance between cross-functionality and specialization, as in this example, can invite senior management intervention. Frequent senior management interference can pose a threat to the teams self-organizing nature.

3.3 Balancing Continuous Learning & Iteration Pressure

Self-organizing Agile teams recognize the need to indulge in continuous improvement powered by constant self-evaluation and continuous learning:

> "*I think we just need to keep going and we need to keep improving. I think the minute you think youre there, youre not. Because you can always do better, you can always learn from what went well, what didn't go well and tweak things slightly.*" — P20, Senior Developer, NZ

> "*I think that sort of fits in well with the whole idea of Agile, where youre constantly going is this working for us as a team? or for me as an individual?*'"
> — P13, Developer, NZ

Along with the need for continuous learning and improvement, Agile teams are very much aware of the pressures of delivering their iteration goals. Agile teams face iteration pressure - the pressure to deliver to a committed team goal every iteration. Iteration pressure, in itself, is not detrimental to the team, in fact some amount of iteration pressure is necessary to motivate teams to deliver their goals. Short iteration lengths or an extremely high and unsustainable development velocity, on the other hand, can cause excessive iteration pressure. For instance, a developer found one week iterations to be very demanding:

> "*Im always feeling the need to rush, rush, rush!...after one week [iteration], we want to remove all these stickies [tasks] from the wall. So its always pressure...if you have [longer] development time, then I can adjust my work like if we spent a little bit longer than we expected, I can catch up next week.*" — P10, Developer, NZ

Creating and maintaining a continuous learning environment requires teams to set some explicit time aside for learning each iteration. Iteration pressure, on the other hand, implies they may not have any extra time to spare:

> "*There is a lot of fluidity between roles in an Agile team... You need to actually allow time for other team members to learn what you do and for you to learn what they do. Often we tend to fill up our sprints with so much that a good teaching environment isn't necessarily there...they can see what youre doing but you need to be able to take the time to explain in really good detail.*" — P6, Tester, NZ

Continuous learning involves different types of learning — learning Agile practices, learning new or complex technical skills, learning cross-functional skills, and learning from the teams own experiences ? all of which fuel self-improvement. We now describe how Agile teams attempt to achieve the difficult task of balancing continuous learning of different types alongside the pressure to deliver the team goal every iteration.

Through Pair-in-Need

Pair programming, a particular form of collaboration, is a standard XP practice where developers work in pairs on every task (Beck, 1999). Our participants, practicing combinations of Scrum and XP, used a practice we call Pair-in-Need?where pairing was done on a need basis to "distribute knowledge" (P21) and complete complex tasks.

> "*The way we do it is that if things are unpredictable we always take up user stories as a pair. There are written tasks for which we dont really need to sit together we can part, but if something requires - this is complex, this is design-intensive - we sit together and pair it.*" — P25, Agile Coach, India

Collaboration through Pair-in-Need becomes an important source of learning. Agile Coaches in relatively new teams and senior team members in mature teams often take on a (blinded for review) role to help new comers learn the basic Agile practices and catch up to the teams velocity.

> "*When you join a new organization and that too from a traditional to a new Agile methodology you have to have some space for yourself [to catch up to team velocity]. But [my team] held my finger and they didn't ask me to just walk - they let me run with them! And that was the best thing that I have seen.*" — P26, Tester, NZ

In order to balance continuous learning and iteration pressure, helping team members through collaboration should be considered acceptable by the team as a task that promotes both learning and delivering iteration goal (P11).

Pair-in-Need worked well for these teams because it allowed them to both learn how to tackle new and complex tasks with the help of a peer and at the same time move closer to their set iteration goal.

Through Retrospectives

Retrospectives are often used an an effective tool to evaluate the learning by the team over an iteration and suggest concrete steps for improvement.

> "*With every retrospective we certainly came up with ideas to improve our process, and I think with all those retrospective sessions under our belt, with all the experience sizing, planning, everything combined, it really made us evolve as a team. I'd certainly say our team dynamics expanded well beyond what we thought they would. At the moment were exceptional, were just a little family that works together.*" — P4, BA, NZ

Retrospectives are a powerful mechanism for the team to engage in self-evaluation and self-correction:

> "*The key here that makes it all work is this practice of retrospectives. Because that essentially says you say stop, how are we doing guys? What are the good things that were doing, what are the not so clever things that were doing, how do we stop the not so clever things, how do we start better things? Because then with this practice and with the continuous kneading out the things that dont quite work and focusing on the things that work, you grow that eco-system, you develop it, and youre bound to be successful.*" — P17, Agile Coach, NZ

Retrospectives can be used to assess whether the iteration pressure is unbearable for the team and suggest ways to overcome it. During an interview, a Tester revealed that they were facing iteration pressure because "testing was always pinched at the end" and resolved to take the matter up in a next retrospective because "thats what [retrospectives] are for" (P6). Participants found retrospectives to be "a key ingredient in Agile methodology" (P26) which allowed them to evaluate team practices, including team velocity, and correcting them as needed.

Consequence of Imbalance

A team in New Zealand faced excessive iteration pressure when their only tester on the team left unexpectedly. The team realized the need to automate their testing efforts. The Agile Coach helped the team not to succumb to iteration pressure and the team took the time to simply work on the needed improvement in testing. The improvement involved the team learning new tools and techniques and implementing their own automating testing framework.

> "*We've just basically reduced our velocity and taken the time to do those things because we knew they were important. We made a call that we were going to not going to wimp out, and go back to the manual testing... make it automated...the new tester had more coding skills and therefore we've taken automation a lot further. ...we seem to be the only team I can find in New Zealand doing one hundred percent automation.*" — P14, Agile Coach, NZ

A balance between continuous learning and iteration pressure is necessary to allow Agile teams to keep improving and transcending beyond their current abilities.

4 Related Work

Following Grounded Theory, we critically reviewed the literature on self-organizing Agile teams once the findings seemed sufficiently grounded and developed. The purpose of literature review after analysis is to (a) protect the findings from preconceived notions and (b) to relate the research findings to the literature through integration of ideas (Glaser, 1978). We first discuss how the balancing acts are related to the general principles of self-organization from an organizational perspective (Morgan, 1986). Then we discuss how the balancing acts relate to the fundamental conditions of self-organization as applied in Agile software development in particular (Takeuchi and Nonaka, 1986). Finally, we discuss other related works in light of our research findings. Table 3 presents an overview of the relationships between the balancing acts and the principles of self-organization (discussed in section 5.1) and the conditions of self-organization in Agile software development (discussed in section 5.2)

Table 3 Balancing Acts and the Principles and conditions of Self-organization

Balancing Self-Organization	**Principles of Self-Organization** (Morgan, 1986)	**Conditions of Self-Organization** (Takeuchi et al.,1986)
Balancing freedom and responsibility	Minimum critical specification Bounded autonomy	Autonomy
Balancing cross-functionality and specialization	Requisite variety Redundancy of functions	Cross-fertilization
Balancing continuous learning and iteration pressure	Learning to learn	Self-transcendence

4.1 Balancing Acts and Principles of Self-Organization

Morgan (1986) defined four principles of self-organization as: minimum critical specification, requisite variety (Ashby, 1956), redundancy of functions, and learning to learn. Several researchers have studied and used some or all these principles to explain their findings or further their research (Hut and Molleman, 1998; Molleman, 1998; Nonaka, 1994; Nerur and Balijepally, 2007; Moe et al., 2008). We discuss how our research findings relate to these principles while placing them in context of other studies on self-organization that have used these principles.

Minimum Critical Specification

Minimum critical specification refers to the senior management defining only the critical factors that are needed to direct the team and placing as few restrictions on the team as possible (Morgan, 1986). (Morgan, 1986) also emphasizes the need for self-organizing teams to work in an environment of "bounded" or "responsible autonomy". Hut and Molleman (1998) note that the role of management is extremely important in providing autonomy to the team and for team empowerment. In our research, we found that the freedom provided by senior management was extremely important for Agile teams to self-organize. Hut and Molleman (1998) suggest that while interventions by senior management can "dramatically undermine empowerment", such interventions "may sometimes be inevitable". As such, they propose *boundary management* in order to find the "right balance" (Hut and Molleman, 1998). In our research, we found that senior management was forced to intervene at times when the teams crossed their boundaries of freedom, in an effort to restore the balance. Similarly, Molleman (1998) discusses the need for 'balance of power' which we describe as a balancing act between freedom provided by senior management and responsibility displayed by the team in return.

Requisite Variety and Redundancy of Functions

Morgan (1986) defines requisite variety (Ashby, 1956) as the need for any control system to match the complexity and diversity of the environment being controlled. In other words, the team should "*embody critical dimensions of the environment with which they have to deal so that they can self-organize to cope with the demands they are likely to face.*"

Nerur and Balijepally (2007) relate this principle to Agile software development by comparing variety among team members to interchangeable roles or cross-functionality. Requisite variety implies that changes in the environment of the organization are best handled by self-organizing teams. In other words, if the amount of variety or fluctuations in the environment is low, self-organizing teams — composed of members possessing variety of skills — are not required. Self-organizing teams are effective when there are changes in the organizational environment. It is not surprising that self-organizing teams are seen as improving the flexibility of an organization in terms of its ability to respond to change and as influential in improving the quality of the employee's working life (Hut and Molleman, 1998; Molleman, 1998). Both these aspects of self-organizing teams are well-suited to Agile methods which focus on responding to change and on the people that enable it (Beck, 1999; Schwaber and Beedle, 2002; Highsmith and Fowler, 2001). In our research, we found that the teams were facing dynamic environments, in terms of changing customer requirements and technologies, and were composed of individuals possessing variety of skills to respond to these changes, thus fulfilling requisite variety (Ashby, 1956).

The principles of requisite variety and redundancy of functions are closely related. The principle of requisite variety suggests that redundancy (variety) should be in-built where it is "directly" needed. Redundancy of functions, refers to the multi-functionality of workers where workers are able to perform a wide variety of team tasks through cross-training (Hut and Molleman, 1998). Nonaka (1994) refers to this principle as cross-functionality in a self-organizing team. In our research, we found that teams promoted cross-functionality across technical areas of expertise as well as across functional roles. Molleman (1998) argues that multi-functionality (achieved by cross-training) or cross-functionality is related to improved team performance. However, he also points out that limitations to cross-functionality, such as expense of cross-training, imply a need for finding an 'optimal level' of cross-functionality for the team. In our research, we find that while teams promote cross-functionality, they also acknowledge that some amount of specialization persists. Finding the ‘optimal level’, therefore, is a balancing act between cross-functionality and specialization that our participants performed.

Learning to Learn

Learning to learn refers to the team’s ability to reanalyze problems, reappraise the best work method, and reconsider the required output as necessary (Hut and Molleman, 1998). Nerur and Balijepally (2007) suggest self-organizing Agile teams are able to iteratively solve problems using ‘learning to learn’ via double-loop learning (Morgan, 1986). The specific Agile practices that facilitate ‘learning to learn’ include reflection workshops, stand-up meetings, pair programming, etc (Nerur and Balijepally, 2007). In our research, we found a couple of these mechanisms of double-loop learning — retrospectives and pair-in-need — particularly enabled teams to balance between continuous learning and iteration pressure.

4.2 Balancing Acts and Specific Conditions of Self-Organization in Agile

We have discussed how the balancing acts are related to the general principles of self-organization from an organizational perspective. We now discuss the relationship between the balancing acts and the fundamental conditions of self-organization as applied to Agile software development (Takeuchi and Nonaka, 1986) in particular.

One of the earliest papers to mention and describe self-organizing teams was “The New New Product Development Game” by Takeuchi and Nonaka (1986), where they define a group to possess self-organizing capability when it exhibits three conditions: autonomy, cross-fertilization, and self-transcendence. After a careful study of the three conditions of self-organizing teams, we found relationships between those conditions and the balancing acts found as a result of our study. We discovered that each of the balancing acts were performed in order to uphold each of the three fundamental conditions of self-organizing teams, namely: balancing freedom and responsibility in order to uphold the condition of autonomy, balancing cross-functionality and specialization in order to uphold the condition of cross-fertilization, and balancing continuous learning and iteration pressure in order to uphold self-transcendence. In unison, the balancing acts were performed by the teams in an effort to uphold their self-organizing nature. We discuss each of these relationships below.

Autonomy

According to Takeuchi and Nonaka (1986), a team possesses autonomy when (a) they are provided freedom by their senior management to manage and assume responsibility of their own tasks and (b) when there is minimum interference from senior management in the teams day to day activities (Takeuchi and Nonaka, 1986). We found that participants were provided freedom by senior management to manage their own tasks which fulfills the first criteria of autonomy. In order to ensure there was minimum interference from senior management — the second criteria of autonomy — the teams assumed responsibility in using that freedom. Thus by balancing between freedom and responsibility they ensured that they were able to not only achieve but also sustain autonomy.

Cross-Fertilization

Takeuchi and Nonaka (1986) found that a team possesses cross-fertilization when (a) it is composed of individual members with varying specializations, thought processes, and behaviour patterns and (b) these individuals interact amongst themselves leading to better understanding of each others perspectives (Takeuchi and Nonaka, 1986). In our study, we found that teams consisted of individual members with varying specializations — developers, testers, business analysts — which fulfills the first criteria for cross-fertilization. In order to ensure that these individuals benefited from understanding each others perspectives — the second criteria of cross-fertilization — the teams frequently interacted across different functional roles and attempted tasks across different technical areas. Teams found it impossible to completely avoid specialization but tried to be as cross-functional as possible. The teams ability to balance specialization and cross-functionality meant they could achieve and sustain cross-fertilization.

Self-Transcendence

According to Takeuchi and Nonaka (1986), a team possesses self-transcendence when (a) they establish their own goals and (b) keep on evaluating themselves such that they are able to devise newer and better ways of achieving those goals. In our study, we found that teams were able to establish their own goals in terms of deciding how much to commit to in an iteration, thus fulfilling the first criteria of self-transcendence. Teams not only established their own goals but also assumed full responsibility to achieve those goals causing pressure to deliver. While some iteration pressure motivated teams to achieve their goals, excessive pressure resulted in a neglect of learning and improvement. In order to balance between iteration pressure and the need for continuous learning, the teams practiced Pair-in-need to both complete tasks and learn from each other in the process. The other technique was to engage in retrospective meetings to self-evaluate and suggest ways of improvement. Teams used retrospectives to find a balance between the amount of time they devoted to finishing tasks versus the time they would spend specifically on learning new and better ways of working — thus fulfilling the second criteria of self-transcendence. By balancing between iteration pressure and the need for continuous learning, teams were able to achieve self-transcendence.

4.3 Other Related Work

Schwaber (2010), the co-creator of Scrum, describes that Agile processes "employ self-organizing teams" which are cross-functional, not limited by their "job titles, training and experience" rather the team "self-organizes based on its strengths and weaknesses to do the work at hand." Schwaber (2010) suggests individuals on the team need to co-ordinate their individual self-organization with the rest of the team via daily synchronization meetings like the daily Scrums. Schwaber (2010) argues that Scrum facilitates self-organizing by providing the team authority and expecting responsibility in return, which is directly aligned with our finding that balancing the freedom (and authority) available in an Agile team with the responsibility that is expected of the team in return is important in achieving and sustaining self-organization.

The other co-creator of Scrum, Sutherland (2010) describes how "the first Scrum team was created directly" from Takeuchi and Nonaka (1986)'s paper which explains that self-organizing teams consist of "members with diverse backgrounds" are "given a free hand" from the top management. In our study, we have identified senior management support as an important environmental factor that effects the success of self-organizing teams.

Larsen (2010) defines a self-organizing Agile team as a "*group of peers using one or more Agile methods that share a goal and accomplish the goal through collaboration.*" The team approach problem-solving collaboratively and strive for continuous improvement. Our study identified that self-organizing Agile teams perform a collaborative practice of Pair-in-Need as a problem-solving strategy. Their effort to balance iteration-pressure and continuous learning ensures continued self-transcendence through continuous improvement.

The importance of self-organizing teams in Agile software development and the need for self-assignment, collective responsibility, cross-functionality, and continuous learning in such teams has been mentioned (Berteig, 2010; Elssamadisy, 2008). The importance of retrospectives in encouraging continuous learning has also been widely acknowledged (Derby and Larsen, 2006; Elssamadisy, 2008). Balancing self-organization ensures less management intervention (Anderson et al., 2003; Takeuchi and Nonaka, 1986).

Self-organization has also been discussed from the complexity science/theory perspective (Takeuchi and Nonaka, 1986; Augustine et al., 2005). Comparing to the evolutionary theory context, Takeuchi and Nonaka (1986), explain the term "self-organization" to refer to "*a group capable of creating its own dynamic orderliness*". Augustine et al. (2005) compare Agile projects to Complex Adaptive Systems and suggest that the complex interactions among members leads to self-organization and emergent order. Other proponents of this view insist that senior management and managers , while relinquishing control, must provide an environment that is conducive for self-organization to emerge (Lewin, 2000), supporting our findings. In particular, their advice on senior management maintaining some minimum amount of control on the teams supports our findings which clearly show the negative effects of providing freedom without boundaries (section 3.1)

While practitioner-based literature on self-organizing Agile teams abound, research literature on the subject is relatively limited. In a single case study, Moe et al. (2008) studied barriers to self-organization by focusing on one aspect of self-organization — autonomy. They found that the management did not provide an environment conducive to self-organization that led to reduced external autonomy. Their findings reflect our assertion that senior management support, in terms of providing freedom, is important for teams to self-organize. They claim that high individual autonomy proved to be a barrier to self-organization as members preferred individual goals over team goals. In contrast, our cross-cultural study found that the New Zealand teams individualistic culture (Aston et al., 2008) did not negatively affect collaboration and co-ordination on these teams. The teams inability to balance freedom and whole team responsibility, however, can be an important barrier to self-organization.

The term self-organizing (or self-organized) can be confused with self-managing (or self-managed). Anderson and McMillan (2003) distinguish between self-managed and self-organized teams as: "*self-managed teams have at least one individual whose primary role is organizational whereas self-organized teams have no designated leader*". They point out that self-managed teams are part of formal organizational structure and are not spontaneously formed. Using this definition, the functional roles defined on Agile teams (such as developer, tester, Agile coach etc) fall under the self-managed team definition since they are formal and non-spontaneous and the leadership attributes are ascribed to the Agile coach. (Moe et al., 2009a,b) also refer to Agile teams as self-managed. In their research paper, (Moe et al., 2009b) draw on a case study of a Scrum pilot project and refer to the Scrum team as self-managing.

According to Anderson and McMillan (2003), self-organized teams are informal, temporary, and formed spontaneously around issues. The self-organizational roles on Agile teams (self-organizational roles blinded for review) fall under the self-organized team definition since they are informal, temporary, and formed spontaneously around issues (authors' previous work blinded for review). Furthermore, as discussed in our findings section, while these teams decide their goals and self-select tasks, the boundaries around these teams are influenced by senior management. Deciding their goals and self-assigning tasks are both principles of self-organizing teams as opposed to self-managed teams (Anderson and McMillan, 2003). Thus, we conclude that the functional roles (developer, tester, Agile coach, etc) make Agile software development teams self-managing. The self-organizational roles (blinded for review), however, make Agile software development teams self-organizing.

5 Discussion: Evaluating Grounded Theory

In this section we discuss the limitations of our study, and our experience with GT more generally.

The inherent limitation of a Grounded Theory study is that the resulting theory can only be said to explain the specific contexts explored in the study. Since the codes, concepts, and category emerged directly from the data, which in turn was collected directly from real world, the results are grounded in the context of the data Adolph et al. (2008). Those contexts were dictated by our choice of research destinations, which in turn were in some ways limited by our access to them.

We do not claim the results to be universally applicable: rather, they accurately characterize the context studied Adolph et al. (2008). Generalizability in GT is achieved not through claims of universality of the theory, rather through the ablity of the generated theory to be modifiable to fit, work, and be relevant in new and different contexts (Glaser, 1992). For example, our substantive theory about balancing self-organization in Agile software development teams can be modified to fit into other substantive areas such as teams in sales and marketing.

As with any empirical Software Engineering, the very high number of variables that affect a real Software Engineering project may it difficult to conclusively identify the impact that any one factor has on the success or failure of the project. The positive influence of these balancing acts in achieving and maintaining self-organization, however, was clearly evident.

One way to evaluate an emerging theory is to present it to experts in the field to gain their feedback. When the experts in the field find the research findings useful, it becomes an important source of verifying the fit, work and relevance of the theory (Glaser, 1978). We presented our emerging results to several practitioner groups in India and New Zealand. Our findings were met with enthusiasm and appreciation. We also submitted our findings as papers to several conferences and received valuable feedback from the reviewers. Receiving comments such as 'rings true' and 'well applied' from the expert reviewers and the Agile practitioners made us confident of our emerging theory. Frequent discussions with the research supervisors about emerging codes, concepts, and categories, as well as frequent presentations to — and feedback from — the Agile practitioner communities in NZ and India, helped validate the emerging results.

Data derived from interviews is known to be prone to bias Parry (1998). There are four types of data that can be presented to the researcher: (a) Baseline data, the best description a participant can offer (b) Properline data, what the participant thinks it is proper to tell the researcher (c) Interpreted, what is told by a trained professional who wants to make sure that others see the data his professional way (d) Vagueing it out, the vague information provided by a participant that is not bothered to provide information to the researcher (Glaser, 1978). The ideal data to collect is Baseline, however the researcher can occasionally encounter other types of data. The SE researcher may not be well trained in the art of interviewing for research and as such may struggle to illicit baseline data from the participants. We found that it takes time to build the ability to discern the type of data being provided during an interview and skill to be able to ask questions that can counter-check the data provided.

Another effective way to ensure authenticity of the data collected through interviews is to supplement it with observations of workplaces and activities Parry (1998). The data derived from observations did not contradict, but rather supported the interview data, thereby strengthening it. We gathered a rounded perspective of the issues by interviewing practitioners representing other aspects of software development such as customer representative and senior management besides focusing on the development team (developer, tester, Agile coach, business analyst). In order to minimize any loss or misinterpretation, all data was personally collected and analyzed by the same researcher, namely the first author. Finally, frequent discussions with the research supervisors about emerging codes, concepts, and categories, as well as frequent presentations to, and feedback from, the Agile practitioner communities in NZ and India, helped validate the emerging results.

While the above measures help ensure the authenticity of the data collected, another bias exists in the form of personal biases of the researcher (LaRossa, 2005). As human beings, we all have inherent biases about just about everything. In order to guard against our personal bias, the most effective strategy was to make ourselves explicitly aware of our own potential biases about different situations and experiences (Glaser, 1978; Parry, 1998). (Description of personal experience of authors blinded for review.)

As GT researchers, we "*tell it like it is*" (Glaser, 1978) and the experts in the field will almost always "know it like it is" and so the novelty of results derived from a GT can sometimes be questioned. A GT study aims to generate a theory grounded in the carefully selected data. If the theory accurately reflects the underlying conditions, then participants or experts should indeed find that the theory aligns with their experience: thus, it may not appear novel to them. The key contribution of a GT study, carried out correctly, is that the theory integrates a range of the perspectives and contexts, rather than relying on the experience or intuition (or prejudices or assumptions) of a select few.

Some researchers feel that it is nearly impossible to let the research question emerge in the process of conducting GT (Suddaby, 2006). Avoiding extensive literature review up-front and trusting the emergence of core concern make such skeptics nervous. Our own experience of using GT as a research method in a SE area with no previous theoretical training to begin with is a demonstration of an application of GT. Emergence can happen as long as the fundamental tenants of the methods are adhered to and the researchers is able to use theoretical sampling effectively to continuously narrow the focus of the study to a single most relevant topic or concern. However, our application of Grounded Theory to SE research was not smooth-sailing, as is evident from the various challenges we faced (and have described) in each of the GT steps. However, the strategies we found useful in overcoming these challenges makes us confident that we will employ GT again were we to undertake a similar study in the future. We are hopeful our description of the challenges we faced and the strategies we found useful will help other SE researchers attempting to use GT.

6 Conclusion

There are four main contributions of this paper:

First, we have demonstrated an application of Grounded Theory to Software Engineering by describing the methodology and our GT study of 40 Agile practitioners across 16 software organizations in New Zealand and India.

Second, we described a portion of our research findings: a grounded theory of the balancing acts performed by self-organizing Agile teams between (a) freedom provided by senior management and responsibility expected from them in return; (b) specialization and cross-functionality across different functional roles and areas of technical expertise; and (c) continuous learning and iteration pressure, in an effort to maintain their self-organizing nature.

Third, we placed our research findings in relation to the general principles of self-organization from an organizational perspective. In particular, we found that the principles of minimum critical specification and boundary management was achieved through the balancing act between freedom and responsibility; the principles of requisite variety and redundancy of functions were achieved through the balancing act between cross-functionality and specialization; and the principle of 'learning to learn' was facilitated by the balancing act between continuous learning and iteration pressure. We also discussed the relationship between the balancing acts and the specific conditions of self-organization as applied in Agile software development. In particular, we found that the three balancing acts were performed by Agile teams in an effort to uphold the fundamental conditions of self-organization — autonomy, cross-fertilization, and self-transcendence, respectively. These three balancing acts were not easy to perform but, when done well, ensured the teams were able to sustain their self-organizing nature.

Finally, we discussed the major challenges we encountered while performing Grounded Theory's various activities, and present our strategies for overcoming these challenges.

We are confident that our research will help Agile teams and their management better comprehend ways to achieve and sustain the fundamental conditions and characteristics of self-organizing teams through these balancing acts. Future studies could extend our research findings to explore self-organizing teams in other fields outside Software Engineering.

Acknowledgements We thank all the participants of our study. This research is generously supported by an Agile Alliance academic grant and a (blinded for review) PhD scholarship. Thanks to Dr. (blinded for review) for his help.

References

S Adolph, W Hall, and P Kruchten. A methodological leg to stand on: lessons learned using grounded theory to study software development. In *CASCON '08*, pages 166–178, New York, 2008. ACM.

G.W Allan. A critique of using grounded theory as a research method. *EJBRM*, 2(1), 2003.

Anderson and McMillan. Of ants and men: self-organized teams in human and insect organizations. *Emergence: Complexity Organization*, 5(2):29–41, 2003.

L Anderson, G.B. Alleman, K Beck, J Blotner, W Cunningham, M Poppendieck, and R Wirfs-Brock. Agile management - an oxymoron?: who needs managers anyway? In *OOPSLA '03*, pages 275–277, New York, 2003. ACM. URL `http://doi.acm.org/10.1145/949344.949410`.

R Ashby. *An introduction to cybernetics*. Chapman and Hall, London, 1956.

J Aston, L Laroche, and G Meszaros. Cowboys and indians: Impacts of cultural diversity on agile teams. In *Agile'08*, pages 423–428, Washington, 2008. IEEE.

S Augustine, B Payne, F Sencindiver, and S Woodcock. Agile project management: steering from the edges. *Commun. ACM*, 48(12):85–89, 2005. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1101779.1101781.

K Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, first edition, 1999.

A Begel and N Nagappan. Usage and perceptions of agile software development in an industrial context: An exploratory study. In *ESEM 07*, pages 255–264, Washington, 2007. IEEE.

M Berteig. Team self-organization, 2010.

T Chau and F Maurer. Knowledge sharing in agile software teams, 2004.

T Chow and D Cao. A survey study of critical success factors in agile software projects. *J. Syst. Softw.*, 81(6):961–971, 2008.

A Cockburn. *People and Methodologies in Software Development*. PhD thesis, University of Oslo, Norway, 2003.

A Cockburn. *Crystal clear: a human-powered methodology for small teams*. Addison-Wesley Professional, 2004.

A Cockburn and J Highsmith. Agile software development: The people factor. *Computer*, 34(11):131–133, 2001.

G Coleman and R OConnor. Using grounded theory to understand software process improvement: A study of irish software product companies. *Inf. Softw. Technol.*, 49 (6):654–667, 2007.

E Derby and D Larsen. *Agile Retrospectives: Making Good Teams Great*. Raleigh: Pragmatic Bookshelf, 2006.

T Dybåand T Dingsoyr. Empirical studies of agile software development: A systematic review. *Inf. Softw. Technol.*, 50(9-10):833–859, 2008.

A Elssamadisy. *Agile Adoption Patterns: A Roadmap to Organizational Success*. Addison-Weasley Professional, 2008.

S et al. Fraser. Xtreme programming and agile coaching. In *OOPSLA Comp. 03*, page 265267, New York, 2003. ACM.

S Georgieva and G Allan. Best practices in project management through a grounded theory lens. *Electronic Journal of Business Research Methods*, 2008.

B Glaser. *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory*. Sociology Press, Mill Valley, CA, 1978.

B Glaser. *Basics of Grounded Theory Analysis: Emergence vs Forcing.* Sociology Press, Mill Valley, CA, 1992.

B Glaser. *Doing Grounded Theory: Issues and Discussions.* Sociology Press, Mill Valley, CA, 1998.

B Glaser. Remodeling grounded theory. 5(2), 2004.

B Glaser. *The Grounded Theory Perspective III: Theoretical Coding.* Sociology Press, Mill Valley, CA, 2005.

B Glaser. Grounded theory institute: Methodology of barney g. glaser, 2010. URL `http://groundedtheory.org/,AccessedonApril2,2010`.

B Glaser and A. L. Strauss. *The Discovery of Grounded Theory.* Aldine, Chicago, 1967.

J Highsmith. *Adaptive software development: a collaborative approach to managing complex systems.* Dorset House Publishing, New York, 2000.

J Highsmith. *Agile Project Management: Creating Innovative Products.* Addison-Weasley, USA, 2004.

J Highsmith and M Fowler. The agile manifesto. *Software Development Magazine*, 9 (8):29–30, 2001.

J Hut and E Molleman. Empowerment and team development. *Team Performance Management*, 4(2):53–66, 1998.

C Larman and V. R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, 2003.

R LaRossa. Grounded theory methods and qualitative family research. *Journal of Marriage and Family*, 67:837–857, 2005.

A Martin, R Biddle, and J Noble. The xp customer role: A grounded theory. In *AGILE2009*, Chicago, 2009. IEEE Computer Society.

R Martin. *Agile Software Development: principles, patterns, and practices.* Pearson Education, NJ, 2002.

N. B Moe and T Dingsoyr. Scrum and team effectiveness: Theory and practice. In *XP*, pages 11–20, Limerick, 2008. Springer.

N. B Moe, T Dingsoyr, and T Dybå. Understanding self-organizing teams in agile software development. In *ASWEC 08*, pages 76–85, Washington, 2008. IEEE.

N. B Moe, T Dingsoyr, and T Dybå. Overcoming barriers to self-management in software teams. *IEEE Software*, 2009a.

N. B Moe, T Dingsoyr, and T Dybå. A teamwork model for understanding an agile team: A case study of a scrum project. *Information and Software Technology*, 2009b.

E Molleman. Variety and the requisite of self-organization. *International Journal of Organizational Analysis*, 6(2):109–131, 1998.

G Morgan. *Images of organization.* Sage Publications, Beverly Hills, 1986.

S et al Nerur. Challenges of migrating to agile methodologies. *Commun. ACM*, 48(5): 72–78, 2005.

Sridhar Nerur and VenuGopal Balijepally. Theoretical reflections on agile development methodologies. *Commun. ACM*, 50(3):79–83, 2007.

I Nonaka. A dynamic theory of organizational knowledge creation. *Organization Science*, 5(1):14–37, 1994.

NVivo. Research software tool, 2010. URL `http://www.qsrinternational.com/products_nvivo.aspx,lastaccessed10April2010`.

S.R Palmer and M Felsing. *A Practical Guide to Feature- Driven Development.* Pearson Education, 2001.

K.W Parry. Grounded theory and social process: A new direction for leadership research. *Leadership Quaterly*, 9(1):85–105, 1998.

M. Pikkarainen, J. Haikara, O. Salo, P. Abrahamsson, and J. Still. The impact of agile practices on communication in software development. *Empirical Softw. Engg.*, 13(3): 303–337, 2008.

K Schwaber. Scrum guide, 2009.

K Schwaber. Agile processes and self-organization, 2010. URL `http://www.controlchaos.com/download/Self%20Organization.pdf, lastaccessed31March2010`.

K Schwaber and M Beedle. *Agile Software Development with SCRUM*. Prentice-Hall, 2002.

H Sharp and H Robinson. An ethnographic study of xp practice. *Empirical Softw. Engg.*, 9(4):353–375, 2004.

H Sharp and H Robinson. Collaboration and co-ordination in mature extreme programming teams. *Int. J. Hum.-Comput. Stud.*, 66(7):506–518, 2008.

J Stapleton. *Dynamic Systems Development Method*. Addison Wesley, 1997.

A Strauss and J Corbin. *Basics of qualitative research: grounded theory procedures and techniques*. Sage Publications, Newbury Park, CA, 1990.

R Suddaby. From the editors: What grounded theory is not. *Academy of Management Journal*, 49(4):633–642, 2006.

J Sutherland. Roots of scrum: Takeuchi and self-organizing teams, 2010. URL `http://jeffsutherland.com.Accessed31March2010`.

H Takeuchi and I Nonaka. The new new product development game. *Hardvard Business Review*, 64(1):137–146, 1986.

G Thomas and D James. Reinventing grounded theory: some questions about theory, ground and discovery. *British Educational Research Journal*, 32(6):767–795, 2006.

E Whitworth and R Biddle. The social nature of agile teams. In *Agile2007*, USA, 2007. IEEE Computer Society.