Empir Software Eng
DOI 10.1007/s10664-011-9161-0

Developing a grounded theory to explain the practices of self-organizing Agile teams

Rashina Hoda · James Noble · Stuart Marshall

Editors: Carolyn Seaman, Jonathan Sillito, Rafael Prikladnicki, Tore Dybå, and Kari Rönkkö

Abstract Software Engineering researchers are constantly looking to improve the quantity and quality of their research findings through the use of an appropriate research methodology. Over the last decade, there has been a sustained increase in the number of researchers exploring the human and social aspects of Software Engineering, many of whom have used Grounded Theory. We have used Grounded Theory as a qualitative research method to study 40 Agile practitioners across 16 software organizations in New Zealand and India and explore how these Agile teams self-organize. We use our study to demonstrate the application of Grounded Theory to Software Engineering. In doing so, we present (a) a detailed description of the Grounded Theory methodology in general and its application in our research in particular; (b) discuss the major challenges we encountered while performing Grounded Theory's various activities and our strategies for overcoming these challenges; and (c) we present a sample of our data and results to illustrate the artifacts and outcomes of Grounded Theory research.

Keywords Empirical research · Software engineering · Grounded theory · Agile software development · Self-organizing

R. Hoda (✉) · J. Noble · S. Marshall
School of Engineering and Computer Science,
Victoria University of Wellington, Wellington, New Zealand
e-mail: rashina@ecs.vuw.ac.nz, rashina@gmail.com

J. Noble
e-mail: kjx@ecs.vuw.ac.nz

S. Marshall
e-mail: stuart@ecs.vuw.ac.nz

Published online: 15 April 2011

1 Introduction

Software engineering researchers are constantly looking to improve the quantity and quality of their research findings through the use of an appropriate research methodology. Over the last decade, there has been a sustained increase in the number of researchers exploring the various aspects of software engineering (Carver 2004; Crabtree et al. 2009; Coleman and O'Connor 2007; Hoda et al. 2010a, c, d; Martin et al. 2009; Whitworth and Biddle 2007). In particular, 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; Dybå and Dingsoyr 2008). Many of these have used Grounded Theory as a research method.

1.1 Research on Agile Software Development

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 (Dybå and 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 (Dybå and 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 2002) 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 as decision making authority is brought to the level of operational problems, which increases the speed and accuracy of problem solving (Moe and 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. For example, Moe et al. (2008) note that Scrum emphasizes self-organizing teams but does not provide clear guidelines on how they should be implemented. Our research focuses on how Agile teams self-organize in practice.

1.2 Grounded Theory in Software Engineering

Researchers have explored various aspects of software engineering using Grounded Theory as a research method. For example, Coleman and O'Connor (2007) conducted a Grounded Theory research on the use of software process improvement (SPI) models in the Irish software industry. They collected data in phases through interviews with senior managers within 21 software organizations. The main findings of the study suggest that the cost of process is a major motivation behind software companies not using "best practice" SPI models and that companies tailor standard software processes to their own contexts. The greatest cost of process perceived by participating organizations was the cost of documentation. In an effort to reduce documentation-related costs, organizations substituted documentation with verbal communication.

Dagenais et al. (2010) used Grounded Theory to study the prominent features of project landscapes, the orientation obstacles faced by new team members, and the orientation aids that can assist new comers move into a new project landscape. The study was based on qualitative data collected from 18 participants located in 8 countries, most of whom were developers with less than an year of experience. The study concluded that newcomers often face unfamiliar and unfriendly project landscapes while helpful landscape inhabitants greatly help newcomers settle in. The presence of a human guide or mentor, as compared to guidebooks, was found to be immensely useful.

Grounded Theory is being increasingly used to study the human and social aspects of Agile software development teams. The social nature of Agile teams was explored through a Grounded Theory research study by Whitworth and Biddle (2007). The findings highlight the importance of social and interaction-focused practices such as daily meetings, and the use of information radiators in establishing social answerability and awareness. The results emphasize the importance of self-organizing abilities of Agile teams, while highlighting the lack of research on the topic. This study calls for more studies to be conducted on social and cultural issues on Agile

teams, specially with regards to "self-regulatory" work structures (Whitworth and Biddle 2007).

Martin et al. (2009) used Grounded Theory to study the role of the on-site customers in extreme programming (XP) projects. The research was carried out using classic Grounded Theory via informal, semi-structured interviews with XP teams across different organizations. The research discovered that the on-site customer role, although perceived as rewarding by some customers, was also seen as over-burdening and inherently unsustainable. They concluded that the on-site customer role in XP projects needs to be played by a team of people, instead of by a single person as initially assumed in literature. This informal XP customer team consists of different roles. Of these different roles, it is the Negotiator role that is the closest to the classic customer representative role and interacts directly with the development team. They also describe certain customer practices as Customer Boot Camp and Pair Customering. These practices—when combined with the customer roles they identified—can help reduce the burden placed on the on-site customer role and the XP team.

These research studies, as well as our own experiences of applying a Grounded Theory methodology to study everyday practices of self-organizing Agile teams, leads us to believe that Grounded Theory is well suited to exploring how software practitioners collaborate and engineer software. In this paper, we focus on the application of Grounded Theory to our research.

Our contributions in this article are: (a) a detailed description of the Grounded Theory methodology in general and its application in our research in particular; (b) discussion of the major challenges we encountered while performing Grounded Theory's various activities and our strategies for overcoming these challenges; and finally, (c) a sample of our data and results to illustrate the artifacts and outcomes of Grounded Theory research.

The rest of the article is structured as follows: Section 2 describes the various components of Grounded Theory interwoven with a discussion of the major challenges we faced while performing each of the GT activities and our strategies for overcoming them. We also present sample of our data and results to illustrate the artifacts and outcomes of GT research. Section 3 describes some limitations of the study and general challenges and strategies of applying GT in Software Engineering research. The paper concludes in Section 4.

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 O'Connor 2007; Martin et al. 2009; Whitworth and Biddle 2007).

Through the course of our research, we discovered a strong synergy between our research topic (Agile software development) and our research method (Grounded Theory). There are several commonalities between the two: both advocate minimum initial planning—Agile methods advocate minimum design and planning upfront while Grounded Theory recommends minimum initial literature review; both are iterative and incremental in nature—Agile methods have set iterations in which the teams develop small chunks of working functionality towards the final product while the Grounded Theory method involves iterative rounds of data collection and analysis such as each iteration brings the researcher a step closer to the main concern of the study; both focus on the human and social aspects—Agile methods value "*people and interactions over processes and tools*" (Highsmith and Fowler 2001) while GT focuses on studying the human experience and social interactions in a given substantive area. This synergy futher made GT a suitable and convenient research method for us to explore the human aspects of Agile Software development.

Figure 1 presents an overview of the GT method. Table 3 in the Appendix provides a glossary of general GT terms (Hoda et al. 2010b).

In the following sections, we describe the main components of the GT method along with examples from the application of GT to our research. The order in which we describe the GT components follows Fig. 1. We also present the challenges faced in applying the various components of 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 effectively study and uncover the main problems of the participants, the researcher is recommended to refrain from formulating a research problem or

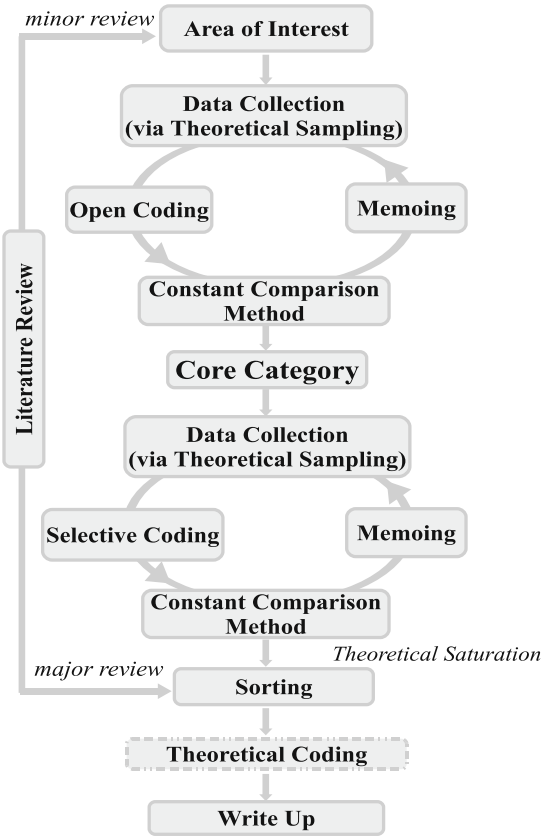

Fig. 1 Overview of the Grounded Theory method

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 (Glaser 1978).

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.

Subsequent interactions with peer researchers led us to the realization that some researchers are forced to research a particular topic because of the terms of their scholarship or funding. Grounded Theory can still be applied in such situations but admittedly, the researcher will need to cultivate a taste and interest in their prescribed area of research over time, which may not come naturally.

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 Minor 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). We found it useful to read some articles describing research conducted using GT in the area of Nursing.

Literature can be extensively reviewed and used in later stages of the GT method (explained under "Major Literature Review" in Section 2.9). 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 after we describe the research findings.

The challenge faced by 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

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. 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.

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

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

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 2010), 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 2010). 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.

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 and observations, 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 3 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. Although Glaser recommends otherwise, we found that 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).

2.4 Data Analysis

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 the following sections , we describe the coding mechanisms—*Open Coding* and *Selective Coding*—that lead to substantive codes and *Theoretial Coding* that leads to theoretical codes.

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 (Glaser 1978, 1998). To explain open coding, we present an example of working from interview transcripts to results for the category "*Balancing Acts*"—which describe the practices of self-organizing Agile teams (Hoda et al. 2010a).

We began by collating key or main 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

2.4.2 Constant Comparison Method

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 2a shows the emergence of the category "Balancing Acts" from underlying concepts and Fig. 2b 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 theoretical 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

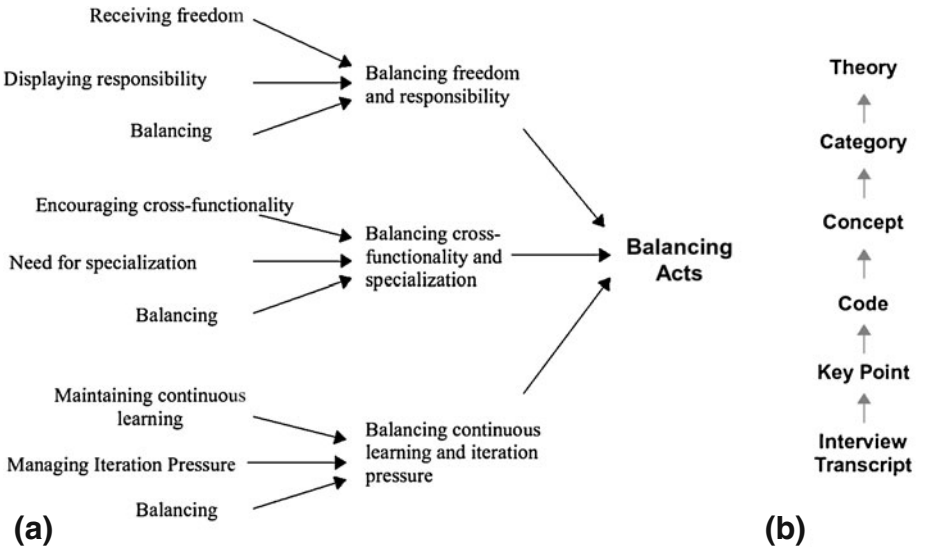

Fig. 2 (a) Emergence of Category "Balancing Acts" from underlying concept, (b) Levels of Data Abstraction in GT

are well-trained in analytical thinking and abstraction. The ability to repeatedly raise concepts in higher levels of abstraction is something we are familiar with. We found this ability extremely relevant when employing GT's 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?" Answering these questions allowed us to continue coding effectively without feeling overwhelmed by the data.

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.

2.4.3 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 publications (Hoda et al. 2010a, c, d, e).

2.4.4 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

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 described 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. We created separate files for memos on different topics and saved the files using the topic name for easy recall. This also made it easier to do Sorting (explained in the next section.)

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 above memo is the concept 'cross-functionality'.

2.6 Sorting

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.

We took printouts of all our memos and sorted them by their topics so that related topics were ordered one after the other. Then we proceeded to generate an outline of the theory using these topic names in the same order. This outline later formed the outline of our thesis.

The challenge involved in sorting the memos is that while it is much easier to group together related memos, the ordering of the memos may not be immediately obvious. It takes some shuffling around of memos and thinking out the relationships between the different memo topics to find an order that makes most sense. We found it useful to draw out the relationships between the different categories with pen on paper. Once the relationships were established in a diagram (using lines to connect categories), it was easier to spot how the memos (covering different categories and concepts) were related.

2.7 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). 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.

Following Glaser's 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). 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.

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.

2.8 Write-up

The final step in GT is writing up the theory, which follows the theoretical outline generated as a result of sorting and theoretical coding. Our theory revolves around self-organizing Agile teams: the *roles* that make Agile teams self-organizing (Hoda et al. 2010c), the *practices* that help Agile team achieve and sustain self-organization (Hoda et al. 2010a); the *factors* that influence self-organizing Agile teams (Hoda et al. 2009, 2010e, d). In this section, we present a small sample of our write-up of results describing the practices of self-organizing Agile teams—*the balancing acts*—to further demonstrate the artifacts and outcomes of Grounded Theory research.

The term Balancing Acts emerged as a result of our data analysis (as described in Sections 2.4.1 and 2.4.2) to describe practices of self-organizing teams that balance between different (and often contrasting) concepts, such as between freedom and responsibility (Hoda et al. 2010a). In each of the balancing acts, we describe how

Agile teams were able to achieve a balance. We have selected quotations drawn from our interviews that serve to highlight the results and that are spread across most participants. Figure 3 presents an overview of the practices of self-organizing Agile teams (the balancing acts) and how they related with previous literature on the general principles of self-organization (GP) and the specific conditions of self-organization (SP). Table 2 represents the relationships between our research findings and the principles and conditions of self-organization (Morgan 1986; Takeuchi and Nonaka 1986).

Balancing Freedom and Responsibility 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

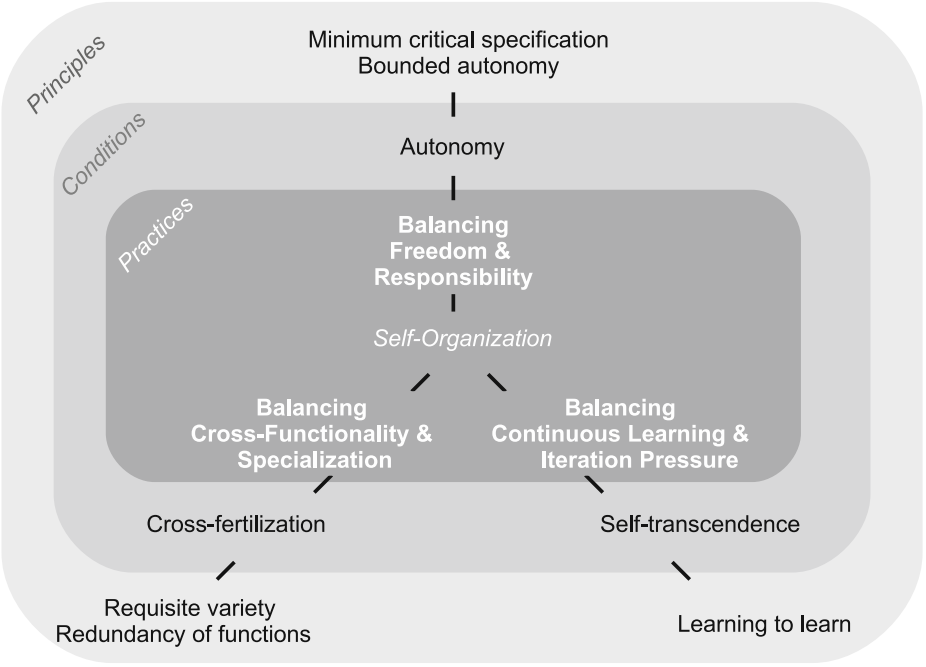

Fig. 3 Practices (Balancing Acts) of Self-Organizing Agile Teams derived from our GT research (Hoda et al. 2010a); General Principles (Morgan 1986) and Specific Conditions (Takeuchi and Nonaka 1986) of Self-Organization

Table 2 Practices of self-organizing Agile teams (Hoda et al. 2010a) and the principles (Morgan 1986) and conditions of self-organization (Takeuchi and Nonaka 1986)

Principles of self-organization (Morgan 1986)	Conditions of self-organization (Takeuchi and Nonaka 1986)	Practices of self-organizing Agile teams (Hoda et al. 2010a, b, c, d, e)
Minimum critical specification & Bounded autonomy	Autonomy	Balancing freedom & responsibility
Requisite variety & redundancy of functions	Cross-fertilization	Balancing cross-functionality & specialization
Learning to learn	Self-transcendence	Balancing continuous learning & iteration pressure

In contrast, 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 don't 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

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). 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 we're 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. [They] have satisfaction that [we] did it in the best possible way... and if there's certain thing missing then we can just ask our friends and our colleagues whether they know a better way to do this...that's [how] we are self-organizing.*" — P28, Developer, India

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.

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. 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

We found that while Agile teams generally promote cross functionality, they cannot completely dispose off specialization. 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.

> "*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

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.*" — P21, 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

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 you're there, you're 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 you're 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:

"*I'm always feeling the need to rush, rush, rush!...after one week [iteration], we want to remove all these stickies [tasks] from the wall. So itŠs 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 you're 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 team's own experiences—all of which fuel self-improvement.

2.9 Major Literature Review

Once the findings seemed sufficiently grounded and developed, we then reviewed the literature on self-organizing Agile teams. The purpose of major 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 practices of self-organizing Agile teams (the balancing acts) are related to the general principles of self-organization from an organizational perspective (Morgan 1986). Then we discuss how the practices relate to the fundamental conditions of self-organization as applied in Agile software development (Takeuchi and Nonaka 1986).

Balancing Acts and General 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 (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 the balancing act between freedom provided by senior management and responsibility displayed by the team in return.

Requisite Variety and Redundancy of Functions Requisite variety is derived from the "law of requisite variety" (Ashby 1956) that claims variety can be handled by variety. Applying this to organizational theory, Morgan (1986) suggests 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 is 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 envrionment. It is not surprising then 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 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).

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 if 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, standup meetings, pair programming, etc (Nerur and Balijepally 2007). In our research, we found a couple of these mechanisms of double-loop learning—retrospectives (a Scrum practice (Schwaber and Beedle 2002)) and pair-in-need (XP's pair-programming (Beck 1999) on a need-basis)—particularly enabled teams to balance between continuous learning and iteration pressure.

Balancing Acts and Specific Conditions of Self-Organization

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).

We found that the subject of self-organizing teams has been largely ignored in research literature on Agile methods. 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.

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 other's 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, by balancing between iteration pressure and the need for continuous learning, teams were able to achieve self-transcendence.

3 Discussion: Evaluating Grounded Theory

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

3.1 Limitations

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.

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.

3.2 Challenges and Strategies

Software Engineering researchers applying GT often encounter criticism regarding the generalizability of their findings. The findings of a GT research are not claimed

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 ability 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 make falsifiable predictions in other substantive areas such as teams in sales and marketing.

Another common criticism faced by SE researchers applying GT is that GT lacks a specific validation phase unlike most other 'scientific methods'. Glaser clearly states that the focus of GT is the *generation* of theory and validation may be undertaken by other researchers using different methods. We, however, found that several activities during a GT research can help evaluate and validate the emerging results of a GT study. One way to evaluate an emerging theory is to see whether the theory also explains and fits the experiences of different practitioners who were not involved in theory generation. For example, we presented our emerging results to several practitioner groups in India and New Zealand. Our confidence in the validity of our theory was helped by these practitioner groups recognising their own experiences in theory generated from others' experiences. 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.

Another source of validation of the emerging theory is to explore how well the theory fits with previous literature on the subject. Since the major literature review in the *same* subtantive area of research is conducted only after the main concepts and categories are established, it becomes an important source of validating the emerging theory. For example, once we had established the three balancing acts as the practices of Agile teams that particularly enable self-organization, we conducted an extensive literature review on self-organization in Software Engineering. We found that previous literature in organizational theory had defined the general principles of self-organization (Morgan 1986). More recent literature in Agile software development described the three conditions of self-organization (Takeuchi and Nonaka 1986). Both these principles and conditions of self-organization fit perfectly with our practices of self-organizing teams. As depicted in Fig. 3, our findings about the practices re-enforces the principles and conditions found in literature as well as builds on them. In particular, while the principles and conditions are abstract, the practices that emerged through our GT research are of a more concrete nature. These practices spell out *how* Agile teams go about fulfilling the principles and conditions of self-organization on an everyday basis.

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 researcher can encounter any of these. The SE researcher may not be well trained in the art of interviewing for

research and as such may struggle to illicit useful 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 and to validate the interpretation of the interview data, is to supplement it with observations of workplaces and activities (Parry 1998). The data derived from observations did not contradict, but rather supported our 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 primary researcher, namely the first author.

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).

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. This articulation of how practices are applied in real projects supports further academic discourse by identifying useful avenues of further research.

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.

4 Conclusion

From our experience of applying Grounded Theory as a qualitative research method, we argue that Grounded Theory is a useful research methodology to explore the human and social apsects of Software Engineering. Grounded Theory enables the researcher to effectively capture the experiences of how people engineer software. Grounded Theory scores over other research methods due to its unique focus on uncovering the real concerns of the participants rather than forcing a preconceived research problem. It is specially suited to study the human and social aspects of Agile software development methods due to the synergy between GT and Agile methods such as their lack of extensive planning upfront, their iterative and incremental nature, and their focus on the people and their interactions.

We are confident that our experiences will help other Software Engineering researchers in successfully applying Grounded Theory as a qualitative research method. In particular, there are three main contributions of this paper: Firstly, we have presented evidence that Grounded Theory can be applied to the area of Software Engineering by describing the methodology in the context of our GT study of 40 Agile practitioners across 16 software organizations in New Zealand and India. Secondly, we identified and discussed the major challenges we encountered while applying Ground Theory, along with the strategies that we employed to overcome these challenges. As is evident from the various challenges we faced as Software Engineering researchers attempting to apply Grounded Theory, it is not a trivial task. We did, however, find several strategies that were useful in eliminating most of these challenges.

Finally, we presented a sample of our data and results to demonstrate the artifacts and outcomes of Grounded Theory research. We described 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. 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 selftranscendence, 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.

Future studies could both extend our study on self-organizing teams outside Software Engineering, and identify and incorporate new challenges and strategies from other companies and practitioners within Software Engineering. We are confident that our research will help others generate theory grounded in other contexts within Software Engineering.

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

Appendix

Table 3 A glossary of grounded theory terms

Term	Description
Theoretical Sampling	A process which allows the researcher to collect, code, and analyze the data and then decide what data to collect next.
Open Coding	The first step of data analysis; starts by collating main points from raw data which are then assigned a code.
Code	A phrase that summaries the main points in 2 or 3 words.
Constant Comparison	A process where codes arising out of each interview are constantly compared against the codes from the same interview, and those from other interviews and observations to find common patterns in data.
Concepts	A higher level of abstraction produced as a result of applying constant comparison on codes.
Category	The next level of abstraction produced as a result of applying constant comparison on concepts.
Memoing	The ongoing process of writing theoretical notes throughout the GT process. Memos capture the conceptual links between categories as the researcher notes down their reflections on different categories.
Core Category	Several categories emerge as a result of data analysis and the one that is able to account for most variations in the data and relates meaningfully and easily with other categories is called the Core Category (Glaser 1978).
Selective Coding	A coding procedure where the researcher codes selectively for the core category and only those categories that are closely related to the core. Once the core category is established, the researcher ceases open coding and uses selective coding.
Theoretical Saturation	When further data collection and analysis on a particular category leads to a point of diminishing results—no new insight into the category is generated—the category is said to have reached *theoretical saturation* (Glaser 1992). The researcher can then stop collecting data and coding for that category.
Extensive Literature Review	As the theory starts to emerge, the researcher can conduct extensive literature review to see how the literature in the field relates to their emerging theory.
Sorting	The process of conceptually arranging the theoretical memos once the researcher has nearly finished data collection and coding is almost saturated. Sorting of the memos forms a theoretical outline (Glaser 1978) and should be done on a conceptual level resulting in an outline of the theory describing how the different categories relate to the core-category.
Theoretical Coding	A coding procedure where the researcher uses one of the the several theoretical frameworks (theoretical coding families) to describe how the emergent categories relate to each other as a hypotheses to be integrated into a theory (Glaser 2005).
Write-up	The final step in GT is writing up the theory, which follows the theoretical outline generated as a result of sorting and theoretical coding.

References

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

Allan GW (2003) A critique of using grounded theory as a research method. EJBRM 2(1):1–9

Anderson L, Alleman GB, Beck K, Blotner J, Cunningham W, Poppendieck M, Wirfs-Brock R (2003) Agile management - an oxymoron?: who needs managers anyway? In: OOPSLA '03, New York, 2003. ACM, pp 275–277. doi:10.1145/949344.949410

APN (2010) Agile professionals network. http://www.agileprofessionals.net/. Last accessed 20 September 2010

ASCI (2010) Agile software community of India. http://www.agileindia.org/. Last accessed 20 September 2010

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

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

Beck K (1999) Extreme programming explained: embrace change, 1st edn. Addison-Wesley Professional

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

Carver J (2004) The impact of background and experience on software inspections. Empirical Softw Engg 9(3):259–262

Chau T, Maurer F (2004) Knowledge sharing in agile software teams. Lect Notes Comput Sci 3075:173–183

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

Cockburn A (2003) People and methodologies in software development. PhD thesis, University of Oslo, Norway

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

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

Coleman G, O'Connor R (2007) Using grounded theory to understand software process improvement: a study of Irish software product companies. Inf Softw Technol 49(6):654–667

Crabtree CA, Seaman CB, Norcio AF (2009) Exploring language in software process elicitation: A grounded theory approach. In: ESEM '09: proceedings of the 2009 3rd international symposium on empirical software engineering and measurement. IEEE Computer Society, Washington, DC, USA, pp 324–335

Dagenais B, Ossher H, Bellamy RKE, Robillard MP, de Vries JP (2010) Moving into a new software project landscape. In: ICSE '10: proceedings of the 32nd ACM/IEEE international conference on software engineering, ACM, pp 275–284

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

Fraser S (2003) Xtreme programming and agile coaching. In: OOPSLA Comp 03, ACM, New York, pp 265–267

Georgieva S, Allan G (2008) Best practices in project management through a grounded theory lens. EJBRM 6(1):43–52. http://www.ejbrm.com/issue-current.htm

Glaser B (1978) Theoretical sensitivity: advances in the methodology of grounded theory. Sociology Press, Mill Valley, CA

Glaser B (1992) Basics of grounded theory analysis: emergence vs forcing. Sociology Press, Mill Valley, CA

Glaser B (1998) Doing grounded theory: issues and discussions. Sociology Press, Mill Valley, CA

Glaser B (2004) Remodeling grounded theory. FQS 5(2):1–17

Glaser B (2005) The grounded theory perspective III: theoretical coding. Sociology Press, Mill Valley, CA

Glaser B (2010) Grounded theory institute: methodology of Barney G Glaser, 2010. URL http://groundedtheory.org/, accessed on April 2

Glaser B, Strauss AL (1967) The discovery of grounded theory. Aldine, Chicago

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

Highsmith J (2004) Agile project management: creating innovative products. Addison-Weasley, USA
Highsmith J, Fowler M (2001) The agile manifesto. Software Development Magazine 9(8):29–30
Hoda R, Noble J, Marshall S (2009) Negotiating contracts for agile projects: a practical perspective. In: XP2009, Springer, Italy, pp 186–191
Hoda R, Noble J, Marshall S (2010a) Balancing acts: walking the agile tightrope. In: Co-operative and human aspects of software engineering workshop at ICSE2010, ACM, South Africa
Hoda R, Noble J, Marshall S (2010b) Using grounded theory to study the human aspects of software engineering. In: Human aspects of software engineering (HAoSE '10). ACM, New York, NY, USA, Article 5, 2 p. doi:10.1145/1938595.1938605
Hoda R, Noble J, Marshall S (2010c) Organizing self-organizing teams. In: ICSE2010, ACM, South Africa, pp 285–294
Hoda R, Kruchten P, Noble J, Marshall S (2010d) Agility in context. In: Proceedings of the ACM international conference on object oriented programming systems languages and applications (OOPSLA '10). ACM, New York, NY, USA, pp 74–88. doi:10.1145/1869459.1869467
Hoda R, Noble J, Marshall S (2010e) Agile undercover: when customers don't collaborate. In: XP, pp 73–87
Hut J, Molleman E (1998) Empowerment and team development. Team Perform Manag 4(2):53–66
Larman C, Basili VR (2003) Iterative and incremental development: a brief history. Computer 36(6):47–56
LaRossa R (2005) Grounded theory methods and qualitative family research. J Marriage Fam 67:837–857
Martin A, Biddle R, Noble J (2009) The XP customer role: a grounded theory. In: AGILE2009, IEEE Computer Society, Chicago
Martin R (2002) Agile Software Development: principles, patterns, and practices. Pearson Education, NJ
Moe NB, Dingsoyr T (2008) Scrum and team effectiveness: Theory and practice. In: XP, Limerick, Springer, pp 11–20
Moe NB, Dingsoyr T, Dybå T (2008) Understanding self-organizing teams in agile software development. In: ASWEC '08, IEEE, Washington, pp 76–85
Molleman E (1998) Variety and the requisite of self-organization. Int J Organ Anal 6(2):109–131
Morgan G (1986) Images of organization. Sage Publications, Beverly Hills
Nerur S, Balijepally V (2007) Theoretical reflections on agile development methodologies. Commun ACM 50(3):79–83
Nerur S, et al (2005) Challenges of migrating to agile methodologies. Commun ACM 48(5):72–78
Nonaka I (1994) A dynamic theory of organizational knowledge creation. Organ Sci 5(1):14–37
NVivo (2010) Research software tool. URL: http://www.qsrinternational.com/products_nvivo.aspx. Last accessed 10 April 2010
Palmer SR, Felsing M (2001) A practical guide to feature-driven development. Pearson Education
Parry KW (1998) Grounded theory and social process: A new direction for leadership research. Leadersh Q 9(1):85–105
Pikkarainen M, Haikara J, Salo O, Abrahamsson P, Still J (2008) The impact of agile practices on communication in software development. Empirical Softw Engg 13(3):303–337
Schwaber K (2009) Scrum guide. Online document URL: www.itemis.de/binary.ashx/~download/26078/scrum-guide.pdf
Schwaber K, Beedle M (2002) Agile software development with SCRUM. Prentice-Hall
Sharp H, Robinson H (2004) An ethnographic study of XP practice. Empirical Softw Engg 9(4):353–375
Sharp H, Robinson H (2008) Collaboration and co-ordination in mature extreme programming teams. Int J Hum-Comput Stud 66(7):506–518
Stapleton J (1997) Dynamic systems development method. Addison Wesley
Strauss A, Corbin J (1990) Basics of qualitative research: grounded theory procedures and techniques. Sage Publications, Newbury Park, CA
Suddaby R (2006) From the editors: what grounded theory is not. Acad Manage J 49(4):633–642
Takeuchi H, Nonaka I (1986) The new new product development game. Hardvard Business Review 64(1):137–146
Thomas G, James D (2006) Reinventing grounded theory: some questions about theory, ground and discovery. Br Educ Res J 32(6):767–795
Whitworth E, Biddle R (2007) The social nature of agile teams. In: Agile2007, IEEE Computer Society, USA

Rashina Hoda is a researcher at the School of Engineering and Computer Science at Victoria University of Wellington, New Zealand. The topic of her doctoral research is self-organizing Agile team. Rashina has developed a keen interest in her research methodology, Grounded Theory, over the course of her doctoral research and has published several research papers describing the tenents of GT and results derived from using GT to study Software Engineering. She hold a bachelors with distinction in Computer Science from Louisiana State University, USA and is a certified Scrum Master.

James Noble is Professor of Computer Science and Software Engineering at Victoria University of Wellington, New Zealand. His research centres around software design, ranging from object-orientation, aliasing, design patterns, and Agile methodology, via usability and visualisation, to postmodernism and the semiotics of programming. James has wide experience in supervising thesis students using Grounded Theory as a research methodology.

Stuart Marshall is a lecturer in the School of Software Engineering and Computer Science at Victoria University of Wellington, New Zealand. His research interests include information and software visualisation, software user interfaces on mobile devices, and Agile software development.