Building a Self-organizing Software Development Team: Multiple Case Study

Henri Karhatsu

Helsinki May 29, 2010

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science

Tekijä — Författare — Author
Henri Karhatsu

Työn nimi — Arbetets titel — Title

Building a Self-organizing Software Development Team: Multiple Case Study

Oppiaine — Läroämne — Subject

Computer Science

Työn laji — Arbetets art — Level Master's Thesis	Aika — Datum — Month and year May 29, 2010	Sivumäärä — Sidoantal — Number of pages 92 pages + 6 appendix pages

Tiivistelmä — Referat — Abstract

This thesis examines how a self-organizing software development team can be built. This is done by first defining the characteristics that determine the performance of a self-organizing team, which are autonomy, team orientation, shared leadership, redundancy, learning, and communication and collaboration. Secondly, the thesis explores how different agile software development methods support self-organization. It is found that the methods provide with several practices that support different characteristics of a self-organizing team. Based on the characteristics and the practices a theoretical framework for building a self-organizing software development team is created.

The theoretical model is tested with two case studies. The primary data collection methods were participant observation and thematic interviews. The case studies were software development projects that lasted seven weeks. It is shown that even in such a short period of time a team can become self-organizing when it follows certain agile software development practices. The empirical evidence also indicates that when the practices are not followed, building self-organization in a team deteriorates.

It is shown that autonomy together with communication and collaboration are the major components for building a self-organizing team. The practices related to these are authorize team, have someone to protect team, close and open relationship with customer, work together in open workspace, share information daily, and visualize progress. The other practices are invest in learning, continuous feedback, automated testing and continuous integration, short iterations, end-of-iteration review sessions, track progress (for learning), share responsibility of work, agree on uniformity, keep tasks small (for redundancy), let team to participate in iteration planning and goal setting, guide with mission, prioritize clearly (for team orientation), manage with lead-and-collaborate principle, and cross-functional teams with wide knowledge (for shared leadership). In addition a team needs agile knowledge that can be in the team or provided for example by a coach.

ACM Computing Classification System (CCS): D.2.9 Software engineering - Management

Avainsanat — Nyckelord — Keywords
self-organizing team, self-management, empowerment, agile software development

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library, serial number C-

Muita tietoja — övriga uppgifter — Additional information

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

During the past ten years software development industry has faced a major change as different agile software development methods have been started to use instead of traditional plan-driven methods [Abr02, DyD08]. One of the characteristics of agile software development is that it relies on people and their competence rather than processes [CoH01]. For example the teams using Scrum should be self-directed and empowered [Lef07, p. 41]. This refers to the concept of self-organizing teams.

There is plenty of self-organization literature available [e.g. GuD96]. Many studies show that organizations gain several benefits for having self-organizing teams [BHS92, GuD96, UFC96, CoB97, Jan98, Jia08]. On the other hand, other studies show contradictory results [GuD96, RCL03, TaP04]. This leads to a relevant question: why certain self-organizing teams are very successful as the others fail?

The question can be approached by trying to understand better what self-organization actually means and what it requires from individual team members, management, and organizations. For example Scrum demands for self-organizing teams but there are no clear instructions what this actually means. Extreme Programming method says that one of the responsibilities of a coach is "the acquisition of toys and food" [Bec99]. But as Blotner states, this cannot be the only guideline, more management best practices are needed [Blo03]. The same goes with self-organizing teams; there is a need to understand better what makes a team self-organizing. And especially, what makes a successful self-organizing team?

As the concept of a self-organizing team has landed to software development, the industry can learn from the experiences of other industries. This has actually been done, for example the famous article of Takeuchi and Nonaka [TaN86] had an influence on Scrum software development method [Abr02, p.27]. But still there is a need for software development industry to define how successful self-organization can be achieved in its own field.

This thesis faces the challenge to understand better what self-organization requires and how it can be supported in software development. The aim is to provide with practical instructions both for management, team leaders, and individual team members.

One approach of the thesis is a short time period. Some researchers claim that transforming a work group into a self-organizing work team takes years and is in many ways an endless journey [Jia08]. This thesis takes the opposite viewpoint. First a theoretical framework is created to suggest what needs to be done in order a team to become self-organizing. Then two case studies are analyzed, of which both were projects that lasted only seven weeks. As a result it will be seen that it is possible to build self-organizing software development teams also in a very short period of time.

1.2 Research Problem

The research question of the thesis is: *How to build a self-organizing software development team?*

The main question is divided into four sub questions:

1. How to define a self-organizing team?
2. What are the characteristics determining the performance of a self-organizing team?
3. How to build a self-organizing team?
4. How agile software development methods support building of self-organization in a team?

1.3 Scope of Research

The thesis concentrates on self-organization in a team level instead of organizational or individual levels. Individuals as members of a team are considered but single employee empowerment is not in the focus of this thesis.

Motivated employees are one advantage of self-organizing teams [GuD96, CoB97, Jan98, Jia08, MDD09]. The opposite viewpoint is that why self-organizing teams motivate employees? This aspect is however not discussed in this thesis. Similarly, as autonomy is very important for self-organization, the thesis does not examine how individuals experience it for example psychologically.

The thesis presents six characteristics of a self-organizing team (see Section 2.4). Each of them could be explored very thoroughly, for example learning is a subject of which there is available extensive amounts of literature. However, this thesis examines the characteristics rather cursory concentrating mainly on the team aspect.

Even though Section 2 discusses self-organizing teams in a generic level, the focus of the thesis is in software development. The thesis may however produce results that could be generalized to other fields with further study.

1.4 Structure of Thesis

In a high level Sections 2 and 3 provide a theoretical answer to the research question. Section 4 explains the study context and research methodology, and Section 5 describes the empirical results. Section 6 discusses the empirical results and suggests theoretical and managerial implications. Section 7 concludes the thesis.

The thesis starts with defining a self-organizing team in Section 2.1. This is an answer to the research question 1. After that the advantages and challenges of self-organizing teams are discussed in Sections 2.2 and 2.3. It will come out that not all self-organizing teams perform equally well. As an answer to the research question 2, Section 2.4 then represents six characteristics that determine the performance of a self-organizing team. Section 2.5 discusses more the building aspect, how self-organization can be built in a team. Section 2.6 provides with an answer to the research question 3 by presenting an initial theoretical framework.

The software development approach is taken into account in Section 3. First in Section 3.1 totally eight agile software development methods are presented briefly. Section 3.2 discusses how these agile methods support different characteristics of a self-organizing team. As a result of this a cross tabulation is made containing the eight agile methods and the six self-organizing team characteristics. It is also an answer to the research question 4. Section 3.3 presents the final theoretical framework, which is a theory based answer to the research question.

Section 4 describes the study context, the research methodology, and the data collection methods. Section 5 presents the empirical research results. First in Section 5.1 the project contexts are described in order to better understand for example what the work in the studied projects was like. Then in Section 5.2 it is explored how each of the six characteristics of a self-organizing team appeared in the case studies. The purpose is to see whether the teams behaved liked the theoretical framework suggests and what kind of consequences there were. Section 5.3 summarizes how the different practices of the theoretical framework were used in the two projects, and Section 5.4 presents the primary empirical conclusions.

Section 6 discusses the primary empirical conclusions and presents the theoretical and managerial implications. Finally, Section 7 concludes the thesis, provides with an answer to the research question, examines the limitations of the thesis, and suggests what further study is needed.

2 Self-organizing Teams

The history of self-organizing teams in literature goes back to the 1950s when Trist et al. examined self-regulated coal miners [TaP04]. Since then the management literature has discussed a lot this subject using different terms like empowered teams, autonomous work groups, semi-autonomous work groups, self-managing teams, self-determining teams, self-designing teams, crews, cross-functional teams, quality circles, project teams, task forces, emergency response teams, and committees [GuD96].

All of the terms are somehow related to group work with some employee participation. For example quality circle refers to a group of people who try to find quality improvements [KLJ89]. Task forces are another kind of group for solving problems in some relatively well-bounded mandate [GuD96]. The main difference between self-organizing teams and the others is the degree of empowerment and authority that the team has in order to make decisions: in self-organizing teams it is bigger than in the others [Jia08]. Instead of having someone saying what to do, the group decides what is best for it. This transformation in management style is often referred as moving from command-and-control management into leadership-and-collaboration [CoH01, NeB07].

Rather often a word empowerment is used to indicate that an employee has power to decide and act [e.g. Lep93]. Indeed, most of the research in this area has focused on single employee empowerment [Jia08]. However, when the concept of empowerment is applied to the group level, it becomes especially interesting. This is because when people together are developing something complex and technical, it can be said that a team is a basic work unit [McC03]. Empowerment as a team design construct appeared first in the literature of Total Quality Management [Jia08].

As software development is typically related to complex and technical projects, this thesis will concentrate on self-organizing teams instead of self-organizing individuals. Although as a group of people consists of individuals, the individual level will have an important role in the thesis as well.

This section defines first what a self-organizing team means. Then it describes the advantages, challenges, and characteristics related to self-organizing teams. The section is not restricted to software engineering industry only but is basically applicable for any work that is done in groups. The software engineering approach is taken into account in Section 3.

2.1 Definition of a Self-organizing Team

Before further discuss on self-organizing teams there is a need to define what a team or a group means. Guzzo and Dickson define a work group as follows [GuD96]: “Work group is made up of individuals who see themselves and who are seen by others as a social entity, who are interdependent because of the tasks they perform as members of a group, who are embedded in one or more larger social systems (e.g. community, organization), and who perform tasks that affect others (such as customers or coworkers)." This means the following. Firstly, individuals have to work together. Secondly, outsiders of the group should see the individuals as a group. Thirdly, the work group belongs to some larger context like a company. Finally, the work group does not work just for themselves but for an outsider, typically for a customer.

But is a group the same as a team? For many they are synonyms [GuD96, CoB97] but Katzenbach and Smith argue that a team is much more than a group [KaS93]. In order a group to become a team there needs to be more commitment, more mutual responsibility, and the synergy between members needs to be greater. Katzenbach and Smith actually provide with a comprehensive list of differences between a group and a team (Table 1).

Working group	Team
Strong, clearly focused leader	Shared leadership roles
Individual accountability	Individual and mutual accountability
The group's purpose is the same as the broader organizational mission	Specific team purpose that the team itself delivers
Individual work products	Collective work products
Runs efficient meetings	Encourages open-ended discussion and active problem-solving meetings
Measures its effectiveness indirectly by its influence on others	Measures performance directly by assessing collective work products
Discusses, decides, and delegates	Discusses, decides, and does real work together

Table 1: Differences of a working group and a team [KaS93].

What is noteworthy in Table 1 is the focus between an individual and a team. In groups for instance the accountability and work products are in an individual level but in teams they are in a team level. Another notable aspect is the role of the leader, which in groups is claimed to be very strong. However, in teams Katzenbach and Smith use the term shared leadership. As will be seen later, these aspects play an important role in self-organizing teams.

Katzenbach and Smith also provide with a definition for a team [KaS93]: "A team is a small number of people with complementary skills who are committed to a common purpose, set of performance goals, and approach for which they hold themselves mutually accountable." Compared to the work group definition they emphasize commitment and mutual accountability. However, this definition ignores the importance of authority for a team to make decisions and take necessary actions in their work. This is why the Guzzo's and Dickson's definition for autonomous work groups is used, as an autonomous work group being a synonym for a self-organizing team:

> "Autonomous work groups are teams of employees who typically perform highly related or interdependent jobs, who are identified and identifiable as a social unit in an organization, and who are given significant authority and responsibility for many aspects of their work, such as planning, scheduling, assigning tasks to members, and making decisions with economic consequences (usually up to a specific limited value)." [GuD96]

In this thesis the term that is mainly used is *self-organizing* instead of for example *autonomous*, *self-directed*, or *self-managed*. Autonomous is somewhat used when the concept of autonomy is described (see Section 2.4.1). Same way the word *team* is mainly used but in some cases also the word *group* appears, especially if the characteristics of a team (Table 1) are missing.

The next section describes the reasons for a traditional work group to become a self-organizing team. It lists different advantages that the literature suggests that self-organizing teams bring for organizations. The subsequent section then explores some challenges related to self-organizing teams.

2.2 Rationale for Self-organization

There are many advantages that self-organizing teams may bring to organizations in which the teams work. Cohen and Bailey categorize these into three groups: performance effectiveness in terms of quantity and quality, member attitudes, and behavioral outcomes [CoB97]. Each of them will be explored next.

Performance effectiveness means for example more effectiveness and productiveness, smaller response time, better quality and customer satisfaction, and more innovations [CoB97]. This is widely supported in the literature. For example one study reported major improvements in different key ratios like quality, amount of rework, inventory level, and on time deliveries [UFC96]. Another research has shown increased throughtput yield and cycle time [BHS92]. As well the customer satisfaction has been mentioned in many articles [BHS92, Win94, Jan98, Jia08]. In general many research reports indicate that self-organizing teams are more effective than the comparison groups [GuD96].

What makes self-organizing teams then effective? The main reason is that such teams can react into problems quickly since the decision making is close to the problem [TaP04, MDD09]. Instead of waiting for a manager's acception, the team has an authority to make the necessary actions itself. This refers to the concept of team autonomy, which will be discussed later in more detail. Another aspect is that when the collaboration inside of a team is good, the team can address problems faster and make quick decisions [RCL03].

Leppit puts it in an interesting way by asking two questions [Lep93]: How much power you have if a hundred of unempowered employees work for you? And how much power you have if those one hundred employees are empowered and motivated?

The second category, member attitudes, means for instance increased job satisfaction, bigger commitment to the organization, and trust to the management [CoB97]. Also these are suggested in many other researches as well [GuD96, Jan98, Jia08, MDD09]. It can be said that self-organizing teams stimulate participation and commitment, which make the employees to care more about their work [Fen98, MDD08].

The third category, behavioral outcomes, contains the level of absenteeism, turnover, and safety [CoB97]. These refer to the change in how people behave in their organizations when getting empowered. The similar outcomes has been seen also in other studies [GuD96]. One aspect to absenteeism is that even motivated employees are sick sometimes but for self-organizing teams the absence of a member is not that crucial [MDD09]. The reason for this is that such teams have better redundancy so other team members are able to do the work of the missing person.

There is interaction between the different benefits of self-organization. If empowerment makes employees more satisfied with their jobs (member attitudes), then it is easy to understand that the productiveness and quality of work (performance effectiveness) increase as well as turnover and absenteeism decrease (behavioral outcomes). Unfortunately, as literature shows, all the good sides of self-organizing teams are not evident. The following section concentrates on the challenges that may relate to empowerment and self-organization.

2.3 Challenges Related to Self-organizated Teams

Although the literature lists different benefits related to self-organizing teams, there are also lot of conflicting research results available [GuD96, TaP04]. Some research results indicate that there is no connection between empowerment and success or that the project performance is not increased [RCL03].

Takeuchi and Nonaka identify some limitations where the concept of self-organization may not be applicable [TaN86]. These include huge projects and such projects where one genious has major influence. They also notify that self-organization requires a lot of effort from all the team members.

It might also be possible that empowered teams and decentralized decision-making work better when the project is highly innovative or when the market conditions are uncertain [RCL03]. One reason for this may be that uncertain and changing situations require new ways to think and there cannot be too much predefined guidelines for the team. On the other hand, in routine projects the management can simply tell what the team should do.

One possible challenge is the organizational context of self-organizing teams [MDD09]. For example the reward system, leadership, training, available resource, and the structure of the organization influence how teams can perform [CoB97]. Fenton-O'Creevy suggests that the resistance of middle managers can negatively affect to employee involvement [Fen98]. He however states that the resistance is more like a symptom of the problems in an organization's ability of increasing team empowerment. For example if the top level management, performance measures, and reward systems support employee involvement, then the resistance gets smaller.

Similarly the organizational formalization and whether the organization is centralized or decentralized can also have an effect to the performance of autonomous teams [TaP04]. The first means how rigorous the rules and processes are in the organization. The second assesses the level of authority and power the teams and individuals have in making decisions and getting information. If the teams do not have a clear overall picture and they do not have the authority to solve the problems, they cannot be very effective.

The viewpoints mentioned above indicate that there are certain situations when self-organization does not really realize; just calling teams self-organizing does not bring success to the organization. Instead, the organization context must be proper. Also the teams need to know how to work efficiently when they have more power [MDD09].

As a conclusion, the contradictory in research results leads to a situation that there is clearly a need to understand better why in some cases self-organizing teams perform so well and in some cases it is the opposite. As Fenton-O'Creevy states [Fen98]: "The most significant question to answer is no longer 'What are the benefits of employee involvement?', rather it is 'What makes the difference between effective employee involvement programmes and those that fail to achieve their objectives?"' The following section examines in more detail the characteristics of self-organizing teams in order to better understand their structure and behaviour.

2.4 Characteristics Determining Performance of Self-organizing Teams

As shown in Section 2.2, self-organizing teams can be more productive and motivating than traditional work groups. On the other hand, in the previous section it was mentioned that the effectiveness is not always that obvious. As an answer to the research question 2, this section identifies the characteristics that first of all make teams self-organizing but also make such teams perform well.

Moe et al. [MDR09] have proposed a framework for measuring self-organizing teams. They suggest five elements that need to be in place in order a team to be successfully self-organizing. Those are autonomy, team orientation, shared leadership, redundancy, and learning. The following subsections explore each of them. In addition, a sixth element, communication and collaboration, is discussed at the end of the section.

2.4.1 Autonomy

It is said that team autonomy is a significant factor for team effectiveness [Lan00]. There is also evidence from psychology and organization theory that individual autonomy is a critical motivator for employees [Lan00, GaD05]. As a matter of fact, self-organization does not make sense without some kind of autonomy. If an individual or a team should manage itself but there is no autonomy, then how could one call it as self-organization? Already the definition of a self-organizing team in Section 2.1 mentioned about "significant authority and responsibility for many aspects of their work" [GuD96]. Basically this is what autonomy is all about.

Janz [Jan98] studied two teams that both were called as self-directed work teams. However, there were major differences how these teams performed. It seems that one of the reasons was related to the team autonomy. In the team that was not performing so well, the major decisions and goals came outside of the team. In the better performing team the team itself defined the goals, budget, and schedules and also made the important decisions. This clearly refers to the importance of team autonomy.

It is important to notice is that a team must have a real possibility to influence on relevant matters; otherwise self-organization is more symbolic than real [TaP04]. This might be one explanatory factor for why some self-organizing teams perform much better than the others. It is not enough to just start calling a team self-organizing; the organization around it must change as well.

Takeuchi and Nonaka suggest that even thought a self-organizing team spends a lot of time on its own, it should not be completely out of control [TaN86]. They talk about subtle control that the upper management should have over the team. The management should define challenging goals but give the team substantial freedom to find the best way to implement them. On the other hand, there should be checkpoints for the management to prevent instability and chaos. Anyway, the main idea is to avoid rigid control that would diminish creativity and spontaneity.

There are different kinds of autonomy. It can be goal-defining, structural, resource related, or social [GSK05]. Another division is that autonomy can be related to people, planning, process, or product [JCN97, Jan98]. Janz et al. suggest that the type of autonomy is as important as the autonomy itself [JCN97]. However, the most relevant aspect in different types of autonomy is to discuss autonomy levels.

There are three levels of autonomy: external, internal, and individual [HoP06, MDD08]. The external refers to the degree that the people outside of the team influence to the team's decisions [HoP06]. It is noteworthy that this kind of influence can be positive as well; the team can get valuable feedback for example from the management [TaN86, HoP06].

Internal autonomy defines how the work is organized inside of a team [HoP06]. It might be that a team has substantial power to make decisions but some individuals inside of the team have none [Bar09]. Hoegl and Parboteea suggest that great care should be taken to make sure that there really is internal autonomy instead of for example team leader autonomy [HoP06]. The third autonomy level is individual, which tells how much an individual has freedom to decide about his or her own work processes [Lan00].

Figure 1 illustrates the level of autonomy in two teams. The first team (a) has no autonomy of any kind: the upper manager has a tight control, internal autonomy is replaced with team manager autonomy, and the team members have no individual autonomy. On the other hand, the second team (b) is an autonomous team in all levels: the upper manager gives external autonomy remaining a subtle control, the team has internal autonomy instead of for example team leader autonomy, and the team members have individual autonomy as well.

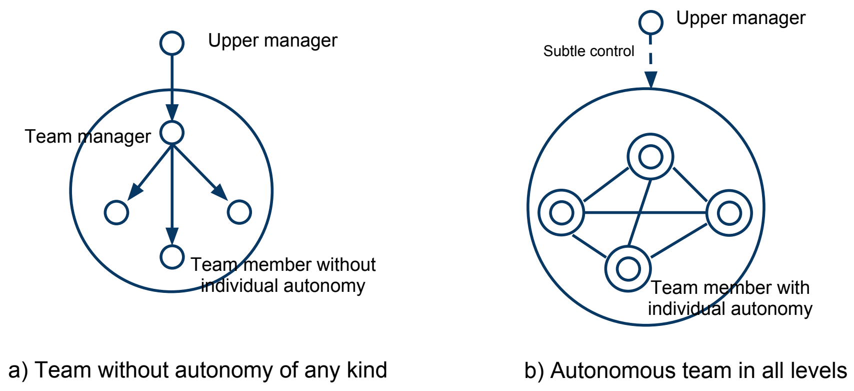
Upper manager
Team manager
Team member without individual autonomy
a) Team without autonomy of any kind
Upper manager
Subtle control
Team member with individual autonomy
b) Autonomous team in all levels

Figure 1: Team autonomy model: team without autonomy (a) and autonomous team in all three levels (b).

There can be a contradiction between internal and individual autonomy [MDD08, Bar09]. At first, there is a risk that an individual has not much autonomy about his or her own work, which is bad for an individual's motivation [GaD05]. On the other hand, an individual may have too much autonomy, which may cause a threath for the team work [Lan00]. Especially if a work group consists of highly independent professionals who have no interaction among each others, the group's flexibility deteriorates [MDK09].

So even thought autonomy is a crucial factor for self-organizing teams, there are also risks associated into it. This leads to the following important area of self-organization, which is team orientation.

2.4.2 Team Orientation

Too few individual autonomy causes problems with employee motivation, too much individual autonomy is a threat for team work. So clearly there is a need to find balance between these two [Bar09]. Moe et al. use a term team orientation, which defines how well the goals of a team and individual meet [MDR09]. This is visualized in Figure 2. Another aspect in team orientation is how the team members take each other's behavior into account during group interaction [SSB05].

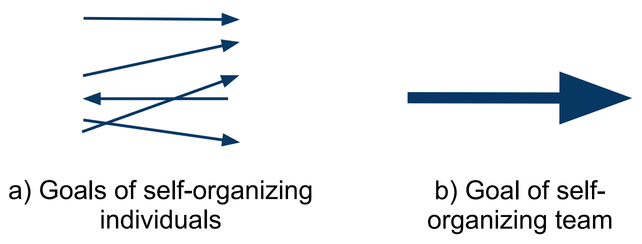
a) Goals of self-organizing individuals
b) Goal of self-organizing team

Figure 2: Separate goals of self-organizing individuals (a) and shared goal of a self-organizing team (b).

Team orientation is an important issue. When comparing two autonomous teams, Janz noticed that the better one had team-oriented goals and the worse one individual-oriented goals [Jan98]. Many researchers indeed suggest that individuals should emphasize the team goals over their own [Whi01, MoA08]. Salas et al. suggest that increased task involvement, information sharing, strategizing, and participatory goal setting are tools for increasing team orientation [SSB05].

Moe et al. [MDD09] examined a team that consisted of very independent specialists. The team had responsibility for planning their work and they decided to split the work into separate modules. Each employee was working with one module and had responsibility of planning it. Since there was no collaboration between the employees, each individual defined his own goals, which led to a bad team orientation. The separation of work led also to other problems like reduction of flexibility when one employee was sick. It also disabled the possibility of shared leadership, which is the next component of self-organizing teams to be introduced.

2.4.3 Shared Leadership

In traditional command-and-control management there is typically one person who always decides. This is problematic approach since in large and complex projects there is a need for different kinds of skills. It is impossible that one person knows everything, so why should he or she then also decide about everything [Pea04]? As concluded in Section 2.4.1, a self-organizing team has autonomy and there is no manager saying what to do. However, this does not mean that there would not be any kind of leadership in self-organizing teams [CoH01]. Its form just happens to be different from traditional.

A self-organizing team may have for example a project manager like a traditional work group might have. However, the project manager cannot be the person who decides everything. This concept is called shared leadership [KaS93]. The idea is to give leadership role to those who have the best skills and knowledge to decide about the particular issue [Pea04]. This is clearly related to the concept of internal autonomy. For each decision a self-organizing team should find the person or people who are experts in that area.

The concept of shared leadership is described in Figure 3. In the traditional command-and-control system the manager makes all the decisions. In the shared leadership model the decision makers vary. It is important to notice that also in shared leadership there can be situations when only one person is deciding (decision 3 in Figure 3b). This might be the case if one person is the only one having knowledge in some very specific area. However, as a contrast there can be cases when the whole team is deciding something (decision 4 in Figure 3b).

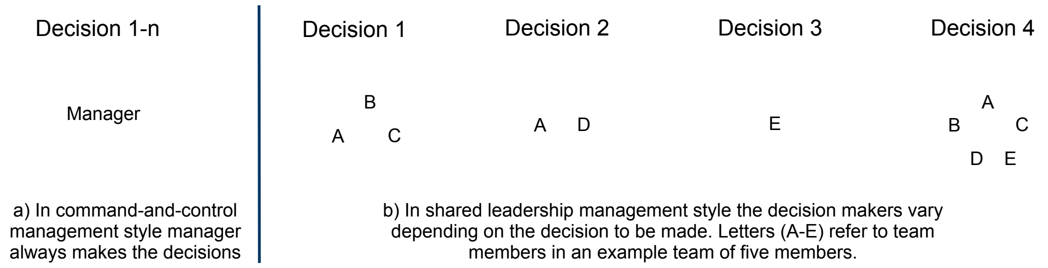
Decision 1-n
Manager
a) In command-and-control management style manager always makes the decisions
Decision 1
B
A C
Decision 2
A D
Decision 3
E
Decision 4
A
B C
D E
b) In shared leadership management style the decision makers vary depending on the decision to be made. Letters (A-E) refer to team members in an example team of five members.

Figure 3: Decision making in command-and-control management style (a) and shared leadership management style (b).

As an opposite to shared leadership Pearce defines vertical leadership to be something where one person is responsible and others follow him [Pea04]. He argues that although shared leadership is recommended leadership model for complex tasks requiring interdependency and creativity, there is a place for vertical leadership as well. Relevant tasks for vertical leaders would be clarifying task specifications, securing necessary resources, selecting team members, and identifying team member roles. A vertical leader should also emphasize the importance of shared leadership.

Some of the vertical leader tasks mentioned above can be questioned. For example it is possible to select team members using shared leadership methods [McC03]. Also why would the vertical leader have to clarify task specifications? Could a team do that together with the customer? And why is it important for a vertical leader to define the roles in the team, which contradicts to the idea of internal autonomy? Perhaps there really is a need for vertical leadership but one should be careful for not to give wrong kind of and too much responsibility for a vertical leader.

Establishing shared leadership in a team has certain requirements. The team needs training [Lan00], for example how to engage in responsible and constructive leadership, how to receive influence, and also in basic teamwork skills [Pea04]. It is also important that the organizational context (for example reward systems) and culture support self-management and thus shared leadership [CoB97, Pea04].

2.4.4 Redundancy

In a traditional work group individuals do their own work so that each person is specialized in some area. This does not work for self-organizing teams. For them it is crucial that the team members are able to do each other's work [NeB07, MDR09]. This means redundancy inside of the team, which in teamwork literature is often called as backup behaviour [MDD08]. In practice redundancy requires that the team members know what the others are doing and that they also have complementary skills [TaN86, NeB07].

Having specialists in a team is important but also dangerous [MoA08]. As mentioned above (Section 2.4.2), specialized individualists who do not interact with each others may regard personal goals more important than the team goals. This is a problem from the team orientation view. However, there is also a problem related to redundancy. For example if a member of the team is away, the rest of the team is not able to carry on his or her tasks unless there is redundancy.

When specialism is a typical way to organize tasks and the organization has no culture for redundancy, this may even affect to employees willingness to accept new tasks. Moe et al. found this in their study; employees were afraid that they have to cope with a task by themselves to the very end [MDK09]. If instead most of the team knows at least something about the task, then the person is not alone with it and not the only one who can solve problems related to it. This encourages employees in taking responsibility.

In the organization level redundancy is an especially challenging issue. Typically if for example two persons are doing the same task or the whole team is discussing something together, upper management sees that as a waste of time [MDD09]. The management should however understand that at the same time they would prevent learning and decrease the flexibility of the team. Instead of independence, the management should encourage the team to interdependence through overlapping job responsibilities [Jan98].

Again the research of two self-organizing teams made by Janz [Jan98] gives evidence about the importance of redundancy. In the study the group performing worse had no overlapping tasks and lacked of interdependence. They did not have cross-training and they avoided conflicts and giving feedback. The better performing team instead was very interdependent and the team members saw the success of the whole team important. They were also constantly trying to improve the skills with cross-training and gave plenty of constructive feedback. This is one evidence indicating the importance of redundancy in self-organizing teams. However, it is worth to notice that the example contains also important notes related to learning. This is the subject of the next section.

2.4.5 Learning

Like concluded in the previous section, learning is closely related to redundancy. If team members have to know what the others are doing, they must learn from each others. On the other hand, learning plays an important role also in other self-organizing team characteristics.

At first, if shared leadership is needed, there has to be a mechanism for learning [MDK09]. Otherwise team members are not able to make decisions together. Also without learning a team cannot make right decisions in changing environment [Hig00]. Another crucial factor is autonomy. If a team learns how to improve its behaviour but is not authorized to make changes, learning is useless [MDK09]. Another aspect related to autonomy is that the management has to tolerate mistakes because they are prerequisite for learning [RaA05].

Also team orientation relates to learning. If team members have own goals, the members do not probably need that much information from each others. However, if an individual's success is determined by the team's success, there needs to be cooperative learning [Jan98].

Self-organizing teams require different kind of learning. Takeuchi and Nonaka suggest that teams need multilearning [TaN86]. It means learning in different levels (individual, team, organization) and in different organizational functions. Team and organization levels are related to cooperative learning described by Janz [Jan98]. Team members can also learn from people outside of the team like from their customers [TaN86].

There are also different ways to learn [TaN86]: In an individual level people can learn for example by reading. In a group level they can learn by making group work. In a corporate level people can learn for example by participating into company-wide programs. Anyway, learning must be continuous. For example practises like project retrospectives or customer focus groups should be used constantly instead of having them only at the end of a project [Hig00].

Takeuchi and Nonaka emphasize that it is important that everyone in a team learns [TaN86]. Traditionally only a highly competent group of specialists learn so that knowledge is accumulated on an individual basis within a narrow area of focus [TaN86]. This is related to the specialist culture [MDD09]. In a self-organizing team members should rather be generalists [MDD09] who acquire the necessary knowledge from across all areas [TaN86].

Figure 4 visualizes the difference between independent specialists and interdependent generalists. Specialists (a) are good in certain specific areas but they have no overlapping knowledge and skills and thus no redundancy. In a team with generalists (b) on the other hand the knowledge and skills overlap, which enables redundancy.

Feedback is closely attached to learning; if a person or a team does not get any feedback, it cannot learn [MDK09]. In a team level this can be encouraged by constantly having

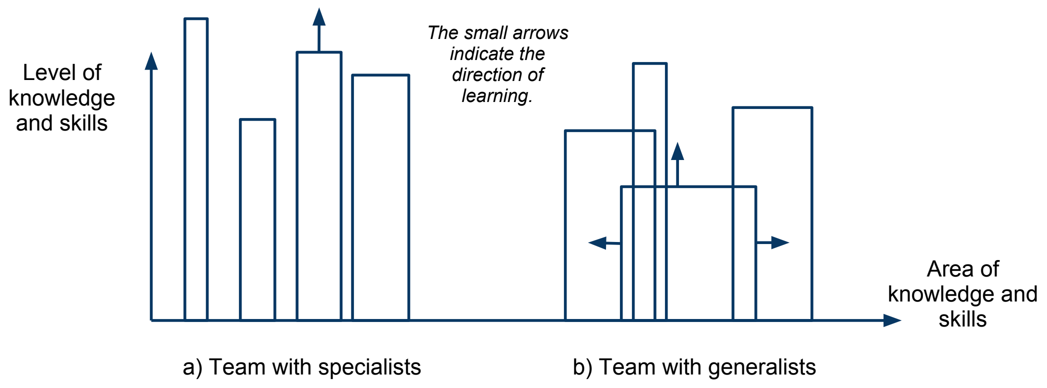
Level of knowledge and skills
The small arrows indicate the direction of learning.
Area of knowledge and skills
a) Team with specialists
b) Team with generalists

Figure 4: Independent professionals vs. interdependent generalists.

different kinds of workshops where the team tries to improve its processes [RaA05]. However, the communication must be open and honest, otherwise people do not give honest feedback [Bec99, p. 39].

Achieving learning and redundancy sounds very important but there is one drawback in it: it can be expensive. For example if individual work is replaced with pair work, the two persons are constantly able to give feedback and increase their knowledge of the same problem. But if the pair would split, they could do two tasks at the same time. So there is a risk that pair work is twice as expensive as single work. However, Moe et al. claim that even though this is expensive, the option would be even more expensive since it makes the organization more vulnerable [MDD09].

2.4.6 Communication and Collaboration

The previous five sections were based on the framework suggested by Moe et al. [MDR09]. They propose that the level of self-organizing can be measured by the five elements, which are autonomy, shared leadership, team orientation, redundancy, and learning. However, it seems that there is a need to discuss one more, which is communication and collaboration. Moe et al. also recognize the importance of them even though communication and collaboration are not explicitly part of their metric.

Communication and collaboration are two different issues [CoH01]. The first one means sending and receiving information. The second one is actively working together to deliver a work product or make a decision. Anyway, both are important for a self-organizing team.

Basically the idea of team work is that people who work together can achieve much more than if they would work individually [CoH01]. This is intimately related to communication, which should be continuous and informal [Whi01, pp. 118-119]. On the other hand, team size should not be too big so that communication and collaboration do not add too much overhead [Aug05]. In bigger projects one solution is to build sub teams with fewer people

[Aug05].

Communication and collaboration are important issues since they relate to all other aspects of a self-organizing team. Shared leadership requires that all the team members actively participate into decision making [MDR09]. This is not possible without collaboration between team members. Team orientation leans on communication. If people do not talk with each others, there can be easily misunderstandings what the goal of the team really is. Also the worst enemy of team orientation, individualism, can be avoided with better communication and collaboration.

Redundancy and learning require communication and collaboration as well, although learning can and should also happen in an individual level. Nevertheless, learning in a group level and redundancy, which is a group level matter itself, cannot occur without team members communicating and collaborating.

Even autonomy cannot work properly without communication. Autonomy is based on the upper management's trust that a team can organize itself and do the best possible work without having someone saying exactly what to do. Nevertheless, as concluded in Section 2.4.1, autonomy does not mean that the team should be left completely alone. Instead, there should be subtle control and periodic checkpoints [TaN86]. So a team needs to communicate for the upper management about its work. Otherwise the trust can gradually disappear.

It seems apparent that a self-organizing team needs to communicate and collaborate. In a way communication and collaboration are not the ultimate goal of a self-organizing team but rather a tool for accomplishing the other factors. This is taken into account in the theoretical framework that will be presented in Section 2.6.

2.5 Building a Self-organizing Team

The previous section listed certain components that a self-organizing team should have. However, it did not describe how to build such a team, which is a relevant aspect when considering the research question of the thesis.

Building a self-organizing team starts with a team creation. In the beginning the relevant question is who should be in the team. It is important to choose team members so that they complement each others [McC03]. The team members should be chosen based on technical, team, and leadership skills [Pea04]. On the other hand, Katzenbach and Smith suggest that personality should not matter, only the skills and the skill potential [KaS93].

After the team members have been chosen and the project starts, there is ambiguity and fluctuation without much information [TaN86]. As mentioned in Section 2.4.1, the upper management should have subtle control for the team [TaN86]. This means that the management should indicate trust towards the team [Pea04] and define challenging goals that are followed with constant checkpoints [TaN86]. On the other hand, the management should not have too rigorous control over the team but instead allow the team to create its own processes and concept [TaN86]. This refers to the external autonomy that the team should have.

One important issue seems to be training [HiM93]. The team should not be left completely alone. As mentioned in Section 2.4.4, team redundancy requires that the team members learn from each others, which can be done by organizing cross-training [Jan98].

Also Jian'an suggests training to be one component in building self-organizing teams [Jia08]. He defines two strategies how to bridge the gap of team empowerment, i.e. how to transform a traditional group into a self-management team. The other is demand-pulling approach, which means that training is provided to increase the team's level of commitment and competence and thereby increase the extent of empowerment. The other is supply-pulling approach, which means giving the team more challenging tasks. Jian'an proposes that there is an equilibrium between the team's competence and the tasks given for it. This indicates that even though a self-organizing team needs challenging goals, at least in the beginning they should not be too challenging.

Katzenback and Smith see that the very beginning of the project is important since it defines the direction for team behaviour [KaS93]. Management should define demanding performance standards and the team should get immediately some performance-oriented tasks. On the other hand, it is important that the team itself can define the performance metrics and has for example freedom to use its budget as it sees is best [HiM93].

Even though the team has a complementary set of skills, it does not automatically bring success. Only after the team members start to collaborate, the mix of skills brings advantages [TaN86]. One very practical way to make this happen is to co-locate the team in to a same room [Jan98].

The issues mentioned in this section support the ones described in the previous section. Additional aspects are organizing training and selecting the team members so that they have complementary skills. Otherwise it seems that giving a team a true autonomy is very important. When the team can organize itself, it also supports shared leadership. One the other hand, if the team format is predefined by the management, there is a risk that certain team members make the decisions only based on their hierarchical positions. The selection of people supports redundancy and learning, as does training.

As a conclusion, all the elements that have been explored previously seem to be important in building self-organizing teams and the elements seem to be interconnected as well. However, current theory does not provide with a clear recipe how the elements should be combined. The following section tries to meet this challenge by presenting the initial theoretical framework.

2.6 Initial Theoretical Framework of Building a Self-organizing Team

Based on the findings in the previous sections this section now presents the initial theoretical framework. The final one containing also the software development view will be introduced at the end of Section 3.

Figure 5 introduces the key concepts related to self-organizing teams. In the bottom there is autonomy. Without it self-organization is symbolic [TaP04] meaning that if a team has no autonomy, it cannot really act like a self-organizing work unit. This way autonomy is a basic presumption for a team to become self-organizing.

Another ground assumption is communication and collaboration that is basis for all other success factors, as decribed in Section 2.4.6. If a team does not collaborate and communicate, team orientation cannot be good, shared leadership is difficult to arrange, and there will be no redundancy nor learning.

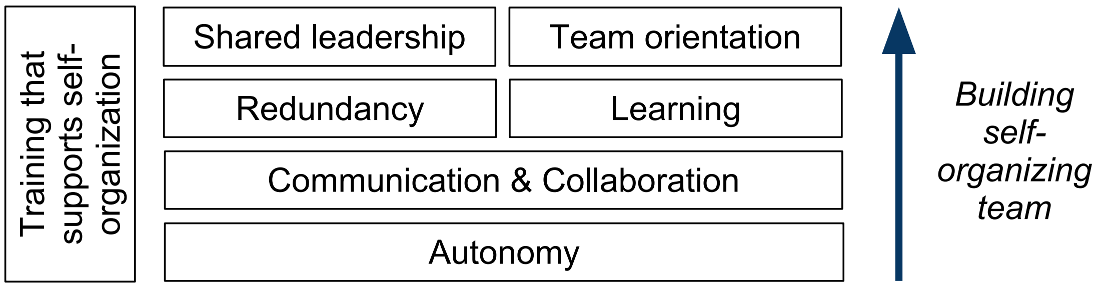
Training that supports self-organization
Shared leadership
Team orientation
Redundancy
Learning
Communication & Collaboration
Autonomy
Building self-organizing team

Figure 5: Initial theoretical framework of building a self-organizing team.

Autonomy together with communication and collaboration constitute the basis of the framework. Above them there are redundancy and learning, which are closely related to each others (see Sections 2.4.4 and 2.4.5). For example if team members do not learn from each other's work, it is difficult to have redundancy [Jan98, MDK09].

On the top there are shared leadership and team orientation. Shared leadership is needed for a team to make decisions with the best knowledge available instead of relying always for example to a team leader decision making [KaS93, Pea04]. Team orientation is required so that the individuals in a team consider the team goals more important than their own goals [Jan98, SSB05, MoA08, Whi08, MDD09, MDR09].

All the layers are supported by training, which is important especially for new teams [HiM93, Jan98, Jia08]. In the beginning a team is built of different kinds of people who may not necessarily have experiences of self-organizing teams. One purpose of training is to make sure that people in the team and also outside of the team understand what it requires for a team to be self-organizing.

The framework is also an answer to the research question 3: How to build a self-organizing team? It provides with a basic understanding of what must be there for a team to become self-organizing.

The next section concentrates more on software development. The target of the section is to find practices that support building of self-organization in software development teams.

3 Self-organizing Teams in Agile Software Development

This section concentrates on self-organizing teams from a viewpoint of agile software development. The agile approach is chosen since agile teams are characterized by self-organization [CoH01]. Section 3.1 discusses this in a bit more detail and then presents eight agile methods. The characteristics of these methods are then combined with the characteristics of a self-organizing team in Section 3.2. Finally the enhanced theoretical framework is presented in Section 3.3.

3.1 Agile Software Development Methods

Software development literature is full of different methods of which some are claimed to be agile and some not. At first this section briefly answers to the questions what makes a development method agile and why self-organizing teams are important in agile methods. Then it presents eight agile software development methods.

Before agile methods the traditional approach into software development has been plan-driven [Boe02]. The problem with plan-driven approach is that it assumes that the requirements are known beforehand [McC01]. As this is not always the case, the agile methods provide a more flexible and adaptable approach that allows developers to make late changes in the specifications [Abr02, p. 10].

Responding to change is one essential issue in agile software development but it is not the only one. Abrahamsson et al. suggest that common for all agile methods is that they are incremental, cooperative, straightforward, and adaptive [Abr02, p. 98]. In addition they say that agile thinking is a people-centric view to software development [Abr02, p. 99].

But why agile methods demand for self-organizing teams? The main reason is that since developers need to be able to react to change, they must be authorized and cabable for doing that. This is also what makes self-organizing teams more effective since the time between decision making and implementation is smaller than in traditional command-and-control approach [CoH01]. Another reason is that the agile approach emphasizes communication and cooperation [Agi01], which in self-organizing teams are very important (see Sections 2.1.6 and 2.6).

Abrahamsson et al. [Abr02] have examined eight agile methods and additionally describe two methods briefly. All of these are presented briefly in the following subsections except the Rational Unified Process (RUP), Open Source Software development (OSS), and Pragmatic Programming (PP). RUP is left out since as Abrahamsson et al. state, RUP is "not generally considered particularly 'agile"' [Abr02, p. 59]. OSS is left out since although it has several similarities with other agile software development methods [Abr02, p. 73], it is not an actual agile method. Similarly, PP is left out since it "is not a method *per se*" [Abr02, p. 18] but more like a collection of programming best practices [Abr02, p. 83].

On the other hand, lean software development is added to the list since it has recently received a lot of interest in the literature. Althought lean software development is not necessarily an agile method but a parallel paradigm [PeF07], it is worth of including in discussion.

Thus the eight agile methods described in the following subsections are: Extreme Programming (XP), Scrum, Crystal family of methodologies, Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM), Adaptive Software Development (ASD), Agile Modeling (AM), and lean software development.

3.1.1 Extreme Programming (XP)

Extreme programming (XP) is an agile method presented by Kent Beck [Bec99]. It collects together some practices that are not new as such but together form a complete methodology for software development [Abr02, p. 19]. The term extreme is based on the idea that the commonsense principles and practices are taken to their extreme levels [Bec99, p. 7].

XP consists of four basic values: communication, simplicity, feedback, and courage [Bec99, pp. 32-35]. Feedback can be from the system to a developer (especially from the tests written in it), from the customer to a developer, and from a developer to the customer. The values are supported with five basic principles and ten additional ones [Bec99, pp. 37-38]. The basic principles are rapid feedback, assume simplicity, incremental change, embracing change, and quality work.

XP defines seven roles: programmer, customer, tester, tracker, coach, consultant, and manager (called Big Boss) [Bec99, pp. 105-111]. For programmers Beck emphasizes the communication and coordination with other programmers and team members. Customer's task is to set the priorities, decide when each requirement is satisfied, and write stories and functional tests together with the tester. Tracker has an important role in giving feedback by tracing the estimates made by the team. Coach is responsible for the process as a whole and needs to have a sound understanding of XP. Consultant's role is to guide the team in specific technical problems and the manager makes the final decisions.

The values and principles are followed by eleven practices: the planning game, small releases, metaphor, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-hour week, and on-site customer [Bec99, pp. 47-58]. Some of these are worth of looking in more detail.

The planning game is meant for quickly determining the scope of the next release by combining business priorities and technical estimates. On the other hand, the plan can be changed afterwords. It is important to notice that the team is making the estimates since the team has the best knowledge for it.

Small releases enable changing direction quickly and each release should contain only the most valuable business requirements. Testing aims at having automated unit and functional tests so that making changes to the system is easier. This is supported with continuous integration that gives fast feedback if there is something wrong in the code.

In XP all the code should be written in pairs, which should produce code with better quality. Another collaborative element is the on-site customer who is available for answering questions, resolving disputes, and setting small-scale priorities. Also open workspace encourages to better communication. Pair programming together with collective ownership increases the shared understanding of the code.

XP is a well-known agile method and there are many published papers on various aspects on XP [Abr02, p. 26]. There is also some evidence that companies who have taken XP into use have achieved good results in productivity, costs, quality, and time-to-market [Rei02].

3.1.2 Scrum

The term Scrum in literature was first used by Takeuchi and Nonaka [TaN86] who described the fast, flexible, and self-organizing product development process used in Japanese product development. Based on this the Scrum software development process was introduced by Ken Schwaber and Mike Beedle [Abr02, p. 27].

Scrum does not define any specific software development techniques but concentrates on how the team members should function in order to produce the system in a flexible way in a constantly changing environment [Abr02, p. 27]. Unlike traditional approaches (like the waterfall development method), Scrum admits that the development process is unpredictable and complex meaning that there needs to be maximum flexibility and appropriate control [Sch95].

The flexibility and adaptivity can be seen throughout the Scrum process that consists of pre-game, game, and post-game [Sch95; Abr02, pp. 29-30]. During the pre-game phase the product backlog is created and updated containing prioritized requirements with estimates made by the team. The game, i.e. the development phase, consists of multiple sprints that are iterative development cycles. The post-game phase contains the closure of the release and is entered when the requirements are completed.

Scrum defines three roles [Lef07, p. 42]: Scrum master, product owner, and Scrum team. Scrum master replaces the role of a traditional manager as his or her task is to remove impediments and to make sure that the team is working as productively as possible. It is notable that Scrum master does not organize the team but the team organizes itself [MoA08]. Product owner makes the final decisions related to the product backlog. Scrum team is a self-organizing team that has an authority to decide necessary actions and is involved for example in effort estimation [Lef07, p. 43].

Instead of providing specific software development methods, Scrum has a list of management practices: product backlog, effort estimation, sprint, sprint planning meeting, sprint backlog, daily scrum meeting, and sprint review meeting [Abr02, pp. 31-34]. Some of these were mentioned earlier but there is need to describe some of them in detail.

A sprint is started by creating a sprint backlog based on the most important items in the product backlog. The items are selected by the team, Scrum master, and product owner together in the sprint planning meeting. Even thought Scrum constantly adapts to the change, the sprint backlog is stable during the sprint.

There is a daily scrum meeting for the team to constantly discuss on the progress. If there are found any impediments, Scrum master should remove them. Another meeting type is sprint review meeting that is held after every sprint. Its purpose is to present the results and make decisions about the following activities.

3.1.3 Crystal Family of Methodologies

The Crystal family of methodologies by Alistair Cockburn includes a number of different methodologies for selecting the most suitable according to the project size and criticality [Abr02, p. 36]. The basic idea is that the bigger and more critical the project is, the more rigor methodology is needed. Abrahamsson et al. [Abr02, pp. 36-46] present two of these methodologies, Crystal Clear and Crystal Orange, of which the latter is more rigor.

Certain policy standards are applied to the Crystal development processes. Both the Crystal Clear and Crystal Orange suggest the following: incremental delivery, progress tracking, direct user involvement, automated regression testing, two user viewings per release, and workshops for product- and methodology-tuning [Abr02, p. 41]. There is one difference between the methodologies: Crystal Clear suggests deliveries in 1-3 months as Crystal Orange allows the length to be extended to four months at maximum. It is possible to replace the policy standards with equivalent practices of other methodologies, such as XP or Scrum, as long as the standards do exist in the process.

Both methodologies define certain roles [Abr02, pp. 42-43]. In Crystal Clear they are sponsor, senior designer-programmer, designer-programmer, and user. Crystal Orange adds a wide range of other main roles. Besides them, there is a range of sub roles defined.

All Crystal methodologies involve a number of practices [Abr02, pp. 43-44]. Staging is for planning the next increment so that the team selects the requirements to be implemented. Each increment includes several iterations and each of them contains revision and review. Monitoring is used for following the progress of the team. Parallelism and flux means that multiple teams can proceed at the same time.

The list continues with holistic diversity strategy, which is a method for splitting large functional teams into cross-functional groups. The main idea is to have different specialities in a single team. Methodology-tuning technique is used for improving the process as well as are the reflection workshops. The last practice is user viewings that should be organized in each release.

3.1.4 Feature Driven Development (FDD)

Feature Driven Development (FDD) is an agile method that does not cover the whole development process but concentrates on design and building phases instead [Abr02, p. 47]. The process has five main steps: develop an overall model, build a feature list, plan by feature, design by feature, and build by feature [FDD10]. The three first of them are sequential and done only once, the last two are done iteratively.

During the first phase a domain expert gives an overview of the domain to be modelled. The modelling team that consists of a chief programmer, a domain expert, and other project staff then develops the overall model. This is explored into a feature level in the second step. Each feature should have a lenght of two weeks at most. In the plan by feature phase a high-level development plan is created including the development sequence. Also different business activities are assigned to chief programmers and classes to developers, which makes them class owners. [Abr02, pp. 48-49; FDD10]

The last two phases, design and build by feature, are done in iterations that last from few days to maximum of two weeks. This enables quick adaptations to late changes in requirements and business needs. In the design phase the features are assigned for the chief programmer and in the build phase they are implemented by the class owners. [Abr02, p. 49; FDD10]

The key roles in FDD are project manager, chief architect, development manager, chief programmer, class owner, and domain experts [Abr02, pp. 50-52]. Besides them there are many other supporting and additional roles.

FDD consists of a set of best practices: domain object modeling, developing by feature, individual class (code) ownership, feature teams, inspection, regular builds, configuration management, and progress reporting [Abr02, pp. 53-54]. From these the individual code ownership is noteworthy since it contradicts with the shared code ownership in XP. In FDD the reason for this is to have someone to be responsible for the consistency, performance, and conceptual integrity of each class [Abr02, p. 53].

3.1.5 Dynamic Systems Development Method (DSDM)

Dynamic Systems Development Method (DSDM) is an agile method maintained by DSDM Consortium[1]. The main idea in DSDM is that instead of fixing the amount of functionality in a product and adjusting time and resources, the time and resources are fixed and

[1]www.dsdm.org

functionality adjusted [Abr02, p. 61].

DSDM consists of five phases, of which the two first are sequential and done only once and the last three are iterative and incremental. DSDM starts with feasibility study and business study. In the feasibility study phase the suitability of DSDM for the project is defined and additionally the technical possibilities and the risks associated to them are gone through. In the business study phase in workshops together with the customer's experts the essential characteristics of the business and technology are analyzed. [Abr02, pp. 62-63]

The content and approach of every iteration is planned separately. The first iteration phase is called functional model iteration, which produces prioritized functions, functional prototyping review documents, non-functional requirements, and risk analysis of further development. The second iterative phase, design and build iteration, is where the system is mainly built. The last phase called implementation is for transferring the system from the development environment to the production environment. [Abr02, pp. 63-64]

DSDM defines 15 roles for users and developers, of which here are the most dominant ones [Abr02, pp. 64-65]. The only development roles are developers and senior developers, which means that there are no distinct roles for design or testing, for instance. The technical coordinator defines the system architecture and is responsible for the technical quality. The most important user role is the ambassador user whose task is to be a bridge between the development team and the users. A visionary is such a user who has the most accurate perception of the business objectives. An executive sponsor has the related financial authority and responsibility and thus has the ultimate power in making decisions.

There are nine practices in DSDM [Abr02, p. 65]. These include active user involvement, empowered teams, focus in fast delivery, collaborative and cooperative approach, reversing any change during development, and fitness for business purpose, to name a few. Team empowerment is seen important so that the teams can make fast decisions. Reversing changes is allowed since in the development a wrong path may be easily taken. This is related to the fitness for business principle with purpose of building “the right product before building it right”.

3.1.6 Adaptive Software Development (ASD)

Adaptive Software Development (ASD) is developed by James A. Highsmith III [Abr02, p. 68]. It is based on cycles that contain three phases: speculation, collaboration, and learning [Hig00]. These phases replace the traditional plan, design, and build that are based on the assumption of relatively stable environment [Hig00].

The term speculation is used in ASD instead of planning since it acknowledges the reality of uncertainty. The idea is not to abandon planning, though. Collaboration on the other hand is highlighted since the amount of information in complex projects is much more than an individual can handle. This also means that teams must improve their joint decision-making ability and they must be authorized to make more decisions than traditionally. [Hig00]

Learning, the third phase of the cycle, is important since the decision making depends on it. Highsmith defines four basic categories for learning: result quality from the customer's perspective, result quality from a technical perspective, the functioning of the delivery team and practices they are utilizing, and the project's status. Customer focus groups can be used for the first one. In technical matters design, code and test plan reviews can be used. Project postmortems and end-of-cycle mini-postmortems are practices for evaluating the team functioning. It is important to constantly test team's knowledge. [Hig00]

ASD iteration, which is called an adaptive lifecycle, has six characteristics [Hig00]. A lifecycle should be mission focused, component based, iterative, timeboxed, risk driven, and change tolerant. Mission focused means that even though the final results may be unclear in the beginning of the project, an overall mission should guide the team. A well articulated mission provides boundaries rather than a fixed destination. Iterative cycle emphasizes re-doing as much as doing. Unlike in manufacturing where rework is considered as a cost, in softwared development rework is part of evolving. Timeboxing is needed so that the team and its customers have to make tradeoff decisions.

ASD does not define team structures in detail nor does it propose many practices for daily software development work [Abr02, p. 72]. Four practices mentioned are joint application development (JAD) sessions, iterative development, feature-based planning, and customer focus group reviews [Abr02, pp. 70, 72]. Abrahamsson et al. argue that this is the most significant problem in ASD; it leaves many details open [Abr02, p. 72].

3.1.7 Agile Modeling (AM)

Agile Modeling (AM) was introduced by Scott Ambler in 2002. AM is a practice-based methodology with scope in modeling and documentation, and it is meant to be used in conjunction with other agile methods like XP and Scrum [Amb02]. It encourages developers to produce advanced enough models to support acute design problems and documentation purposes while keeping the amount of models and documentation as low as possible [Abr02, p. 82].

AM adopts the values of XP: communication, simplicity, feedback, and courage [Amb02]. The values are used to derive ten core principles and eight supplementary principles [Amb02] that are also very close to XP principles. However, there are also principles related to modeling like model with a purpose, multiple models, and know your models.

The principles are followed by eleven core practices and eight supplementary practices

[Amb02]. Like many other agile methods, the practices encourage to customer presence, small teams, and close communication [Abr02, p. 83]. For example there are practices like active stakeholder participation, collective ownership, and model with others [Amb02].

3.1.8 Lean Software Development

The history of lean software development is based on lean manufacturing [Mid01; PoP03, p. xxii-xxiv]. Lean manufacturing is on the other hand based on Toyota Production System that has been developed in Toyota Motor Corporation after the Second World War [WRJ90]. It is built on five basic principles [WoJ96, Ram98]:

1. Define what is value.
2. Define the value stream that adds value. Eliminate all the waste that does not add value.
3. Make the value stream flow without additional interruptions, extra inventories, waiting, errors, or other kind of waste.
4. Do only what the customer asks for without producing something that does not add value for the customer.
5. Aim at perfection.

The lean philosophy has been started to apply to software engineering industry as well [Mid01, PoP03]. Poppendiecks have listed seven lean philosophies for software development [PoP03]. They also suggest practices, called tools, how to implement each principle in practice. The principles and tools are listed in Table 2.

Lean principle	Tools to implement the principle
Eliminate waste	Seeing waste, value stream mapping
Amplify learning	Feedback, iterations, synchronization, set-based development
Decide as late as possible	Options thinking, the last responsible moment, making decisions
Deliver as fast as possible	Pull systems, queuing theory, cost of delay
Empower the team	Self-determination, motivation, leadership, expertise
Build integrity in	Perceived integrity, conceptual integrity, refactoring, testing
See the whole	Measurements, contracts

Table 2: Lean principles and tools [PoP03].

According to Poppendiecks the most important principle is to eliminate waste [PoP03, p. 2]. In manufacturing waste can be divided in seven categories, and Poppendiecks suggest the corresponding ones in software development (Table 3).

The seven wastes of manufacturing	The seven wastes of software development
Inventory	Partially done work
Extra processing	Extra processes
Overproduction	Extra features
Transportation	Task switching
Waiting	Waiting
Motion	Motion (humans, handoffs)
Defects	Defects

Table 3: The seven wastes of manufacturing and software development [PoP03, p. 4].

One of the tools mentioned in Table 2 is to have short iterations with fixed time-box [PoP03, p. 29]. However, there is also such a lean method that has no iterations at all fitting well to the lean concept of just-in-time (JIT). JIT means that something should be produced only when it is needed and the production should flow smoothly [HiT00]. The method is called Kanban.

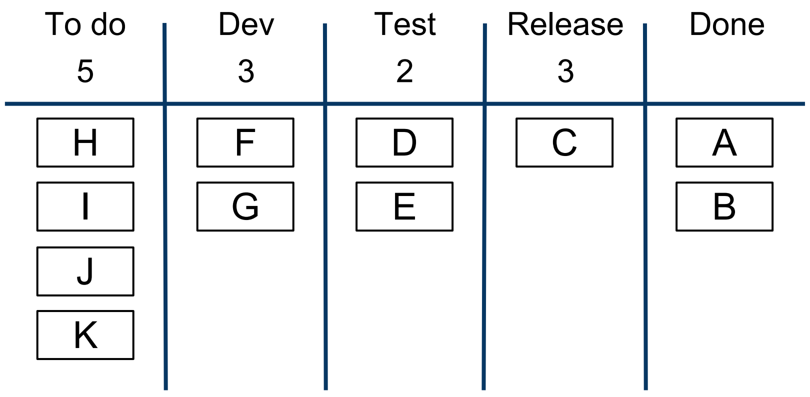

Figure 6: Example of Kanban board [KnS09, p. 21]. Numbers indicate work in process limits for each workflow state.

According to Kniberg, Kanban in software development is based on three rules [KnS09, pp. 20-21]:

1. Visualize the workflow. This can done by splitting the work into pieces, writing each on a card, and putting the cards on the wall (called Kanban board, see Figure 6). The wall should have named columns to illustrate where each item is in the workflow.

2. Limit work in process[2] (WIP). Each column should have an explicit limit telling how many items may be in process in that state.

[2]Kniberg uses the term work in progress but work in process is generally more often used.

3. Measure the lead time. This is needed for optimizing the process so that the lead time becomes as small and predictable as possible.

3.2 Self-organizing Team Characteristics and Agile Methods

Section 2.4 explored the six characteristics of a self-organizing team, which are autonomy, team orientation, shared leadership, redundancy, learning, and communication and collaboration. Section 3.1 on the other hand described briefly eight agile methods. The purpose of this section is to combine the both aspects, i.e. to see for example how XP or Scrum support for example autonomy or redundancy. The reversible viewpoint is considered also. As will be seen, some agile practices or principles may hinder self-organization.

This section utilizes the abbreviations used in the previous section: XP refers to Extreme Programming, FDD to Feature Driven Development, DSDM to Dynamic Systems Development Method, ASD to Adaptive Software Development, and AM to Agile Modeling. The remaining methods are Scrum, Crystal (family of methodologies), and lean (software development).

3.2.1 Autonomy

Autonomy appears in agile methods mainly in two ways. Teams are authorized to make decisions themselves and some agile methods define a role that should protect the team from external disturbance.

Authorization is emphasized in Scrum, which calls teams self-organizing teams. They have substantial decision authority and responsibility like planning, scheduling, work allocation, and decision making [Bar09]. Also one of the DSDM practices is to have empowered teams [Abr02, p. 66]. Moreover, empower the team is one of the lean principles [PoP03, p. 95-124].

An example of how empowerment actualizes in practical level is task estimation and selecting. In XP there is a planning game, and the technical people are responsible for estimating how long a feature will take to implement [Bec99, p. 48-49]. In Scrum the team participates in estimating as well and during the sprint chooses the order in which the tasks are to be done [Kni07, p. 15]. Crystal has the practice called staging where the team selects the requirements to be implemented [Abr02, p. 43]. Lean software teams should be able to decide what is feasible to accomplish in one iteration [PoP03, p. 30].

The sprint backlog in Scrum can be seen as one autonomy tool. Unlike the product backlog that can be modified anytime, the sprint backlog is fixed during a sprint [Lef07, p. 44]. However, in practice this is not self-evident; sometimes there may be a temptation to change sprint backlog for example by the product owner, which can cause problems in the project [MoA08].

Changing the sprint backlog during the sprint is an example of a situation when autonomy is broken. Scrum master in Scrum [MoA08] and project manager in FDD [Abr02, p. 51] are such roles that should prevent this to happen. Both should protect the team from outside distraction.

As defined in Section 2.4.1, autonomy can be divided in three levels: external, internal, and individual. Scrum master and project manager in FDD are meant for protecting the external autonomy. There is evidence that by introducing Scrum the developers can better concentrate on their primary project and the team resources are not so easily stolen [MDD08].

Some agile methods, for example DSDM and Crystal, define several roles [Abr02, pp. 44, 66]. If this leads to a situation where the team cannot organize itself as it sees best, then the roles deteriorate internal autonomy. On the other hand, both methods say that one role can be shared and one person may have many roles, which supports internal autonomy. Moreover, the only development roles in DSDM are developer and senior developer, and in Crystal designer-programmer and senior designer-programmer [Abr02, pp. 42, 64]. This indicates that both methods leave room for development team to choose itself more accurate roles, if needed.

The agile methods presented in this thesis do not contain any practices that would support individual autonomy. One reason might be that individual autonomy is not seen as an issue. Another reason might be that too much individual autonomy can be seen even as a threat for team self-organization [Lan00, MDD08]. Nevertheless, Barney et al. claim that even thought XP and Scrum can give the whole team a great deal of autonomy, individual autonomy can suffer [Bar09]. Individual autonomy is yet important for an individual's motivation [GaD05]. This means that there is a gap in agile methods when it comes to individual autonomy.

3.2.2 Team Orientation

According to Salas et al. team orientation can be supported with increased task involvement, information sharing, strategizing, and participatory goal setting [SSB05]. These elements can be found in some agile methods.

Participatory goal setting is present in XP (planning game), Scrum (sprint planning meeting), and Crystal (staging). In DSDM the business study and in FDD the building feature list can be seen as participatory phases assuming that the team really attends these. On the other hand, if only the management is involved, then the team orientation effect is difficult to reach.

Since team orientation is closely related to goal setting, priorities are important. Scrum product backlog is reprioritized at least before every iteration by the product owner. If the team disagrees with the priorities, it may deteriorate team orientation.

Information sharing is emphasized in many agile methods. Examples are daily scrum meetings (Scrum) and colocation and open workspace (XP, Crystal, AM). The assumption is that more the team members interact with each other, more team oriented they become. At least there is evidence of weak team orientation due individualism [MDD08]. More communication and collaboration related aspects are discussed in Section 3.2.6.

The adaptive development cycles in ASD are said to be mission focused [Hig00]. It means that the activities in each development cycle must be justified against the overall project mission [Abr02, p. 71]. However, this does not automatically mean that the team would also do that. One way to support this in ASD are the joint application development (JAD) sessions where developers and customer representatives discuss desired product features and enhance communication [Abr02, pp. 70-71]. The sprint review meetings in Scrum are quite close to this, assuming that the meetings are used also for looking into the future, not just reviewing the past sprint.

As mentioned in Section 2.4.1, team orientation and individual autonomy may conflict. In the previous section it was noticed that the agile methods do not take this into account that well. One solution is to organize special days for developers to do such projects that on the one hand are somehow related to the ongoing work but on the other hand inspire developers [Bar09]. According to Barney et al., this can increase the staff morale, bring advantages in hiring, and increase entrepreneurship in the organization [Bar09].

3.2.3 Shared Leadership

It is acknowledged that agile organizations need leadership-and-collaboration instead of command-and-control management [CoH01]. This refers to the idea of shared leadership, in which it is less important who makes a decision than getting the right people involved in the decision making [RaA05]. Nevertheless, agile methods do not provide many practices that actually would support this. They rather emphasize the idea of shared leadership in an abstract level, like the leadership practice [PoP03, pp. 111-117] in lean software development does.

In shared leadership the role of a manager must be different than traditionally. Instead of telling the team what to do, the manager should be more like a coach or a facilitator, like Scrum master in Scrum [MDD08]. This means that the team should organize itself and make the decisions concerning what to do [MDD08]. As a matter a fact, the team leadership in Scrum should be divided among the product owner, Scrum master, and the team [MDK09].

One aspect of shared leadership is that those who have the best knowledge decide. When task durations are estimated, the best knowledge is among developers. This is taken into account in XP's planning game, Scrum's effort estimation, and Crystal's staging phase.

If shared leadership is to be developed, a team must have the right people meaning that the team members should have mission-critical knowledge, skills, or abilities [Pea04]. This is related to the idea of cross-functional teams in Scrum [Lef07, p. 41] and in Crystal [Abr02, p. 44]. To make it simplified, if all the team members have skills only from the same area, then the one who is the most expert should always decide. However, if the team knowledge is shared among members, then the need for shared leadership increases.

In order to improve shared leadership it is also important to avoid individualism. A study by Moe et al. identified that when the team members did not have interaction with each others, there was no shared leadership [MDD09]. Basically all the agile methods highlight communication and collaboration and many of the methods also provide with practices related to them. These are explored below in Section 3.2.6. However, before that one medicine against individualism, redundancy, is discussed in the next section.

3.2.4 Redundancy

Agile practices supporting redundancy can be divided into four categories: working together, improving communication, agreeing on uniformity, and sharing responsibilities.

Working together supports redundancy since that way the knowledge is shared for multiple team members [MDD08]. Pair programming in XP is such a practice [Bec99, p. 51]. AM has quite a similar idea with the practice of modeling with others [Amb02]. A third working together practice is the sprint planning meeting in Scrum. Since the whole team usually participates in it [Kni07, p. 15], everyone is aware of all the tasks for the next sprint at least in some level.

It is easier to work together if the team members are colocated. This is encouraged by XP [Bec99, pp. 63-65] and Crystal Clear [Abr02, p. 38]. XP also specifically mentions open workspace as one of its practice [Bec99, pp. 63-65]. All of these go under the category of improving communication. Other practices improving communication and thus increasing redundancy are the daily scrum meeting in Scrum [Lef07, p. 45] and displaying models publicly practice in AM [Amb02].

The third category for redundancy is uniformity. This means that the team agrees on certain principles about for example how the code is written. If all the team members use the same conventions, they can more easily continue the work done by the others. Practices supporting uniformity are the coding standards in XP [Bec99, p. 53] and applying modeling standards in AM [Amb02]. Also Crystal Orange proposes selecting notation standards, design conventions, and formatting standards to be used in the project [Abr02, p. 41].

Instead of having specialists working with their own components or modules, there should be generalists that can do each other's work [MDD09]. This requires that the responsibilities are shared among the team members. One way to encourage this is to have collective ownership like in XP [Bec99, p. 51] and in AM [Amb02]. On the other hand, FDD introduces individual ownership and class owners [Abr02, pp. 51-53], which is against the idea of sharing responsibilities.

3.2.5 Learning

It is easy to find several agile practices and principles that support learning. The adaptive lifecycle of ASD (speculate-collaborate-learn) [Hig00] in a way summarizes the built-in principle of learning in agile software development. Amplify learning principle in lean software development [PoP03, pp. 15-46] is another example. The learning categories discussed in this section are short iterations, learning from customer, working closely with the other team members, continuous feedback and information flow, feedback from technical work, tracking, and end-of-iteration sessions.

As Abrahamsson et al. [Abr02, p. 98] define, in order a software development method to be agile, it must be incremental with rapid cycles. All agile methods discussed in their book promote this. In lean software development this goes even further as in Kanban there is a continuous flow of tasks but no prescribed iterations at all [KnS09, p. 13]. In any case, an important advantage in short development cycles is the fast feedback loop [MDD08]; the sooner the team gets feedback, the faster it can learn [Whi08].

Having short iterations gives a team a possibility to learn about the customer domain [PoP03, pp. 27-31]. Learning from a customer is supported also in many other ways in agile methods. In XP the team should have a real user sitting with the team [Bec99, pp. 52-53]. Similar kind of direct user involvement is emphasized in Crystal, DSDM, and AM [Abr02, pp. 39, 66; Amb02]. FDD defines a role called domain expert who can be a user, a client, a sponsor, a business analyst, or a mixture of these [Abr02, p. 52].

Many agile methods also define some kind of customer session. In XP there is planning game [Bec99, p. 70]. In Scrum there are sprint reviews [Lef07, p. 47], which are close to user viewings in Crystal [Abr02, p. 39]. Likewise, in ASD there are JAD sessions [Abr02, pp. 70-71] and customer focus group reviews [Hig00].

Besides working closely together with the customer, many agile methods encourage working closely together with the other team members. This is related to the need for redundancy described in the previous section. XP and pair programming [Bec99, p. 51], AM and modeling with others [Amb02], and colocation both in XP [Bec99, pp. 63-65] and Crystal Clear [Abr02, p. 38] go under this category. The basic idea is to achieve knowledge sharing among team members [LaC05].

Continuous feedback and information flow is one category of learning in agile methods. The daily scrum meetings are an example of a practice that encourages this [Lef07, p.

45]. Instead of having for example a weekly meeting, the team discusses its progress every day. Design, code and test plan reviews are another way to provide feedback for team members [Hig00]. Feedback or feedback loops are considered important also in lean software development [PoP03, pp. 22-27].

Continuous feedback goes to its extreme level when the actual technical work is considered. If the code is developed in a test-oriented way like XP and DSDM suggest [Bec99, p. 50; Abr02, p. 66], the developers get immediate feedback if they break the code. Automated regression testing, continuous integration and regular builds in XP, Crystal and FDD are related to this [Bec99, p. 51; Abr02, pp. 39, 54]. Under the technical feedback category goes also the AM practice prove it with code [Amb02].

One way to learn is to track the progress of development. XP even has a special role called tracker for this [Bec99, pp. 108-109]. The idea is to learn by measuring against what was predicted to happen [Bec99, p. 61]. Similarly Crystal encourages progress tracking and monitoring [Abr02, p. 43]. Tracking can be also visualized. This can be done by using Kanban board in lean [KnS09] or a burndown chart in Scrum or lean [Kni07, pp. 51-52; PoP03, p. 33].

In lean software development the work in process limits are one way to track progress. The idea in using them is to find bottlenecks in the work process [KnS09, pp. vii, 19]. For example if the testing phase is constantly blocked, it indicates that the team must modify that part of the process somehow. This is related to the queuing theory, which is also one tool in lean software development [PoP03, pp. 77-83].

The last category for learning are end-of-iteration sessions, like reflection workshops in Crystal [Abr02, p. 39], retrospectives in Scrum [Kni07, p. 67-71], and postmortems in ASD [Hig00]. The purpose of them is to learn from the past and improve the team processes. Based on his own experiences, Kniberg [Kni07, p. 67] states that for some reason teams do not always seem inclined to do retrospectives but still they seem to agree that retrospectives are extremely useful. He suggests that the most important thing in retrospectives is actually to make sure that they happen. Moe et al. support this concluding that if the retrospectives are skipped, teams will have problems in learning [MDR09].

Lamoreux [Lam05] agrees that team reflections are important for many reasons. She however emphasizes that it is not meaningless how the reflections are held. For example having a reflection with too many people or repeating every time the questions “what went well?” and “what needs improvement?” is not very beneficial. She suggests that the reflection techniques should be improved all the time to make sure that the team can get the best out of them.

3.2.6 Communication and Collaboration

Communication, collaboration and interaction are in the heart of agile software development. As Agile Manifesto states, "individuals and interactions over processes and tools" and "customer collaboration over contract negotiation" [Agi01]. The reason is that people working together and having good communication and interaction can operate at noticeably higher levels than when they use only their individual talents [CoH01].

No surprise, also the agile methods provide with many principles and practices for increasing interaction between people. The ASD's speculate-collaborate-learn cycle [Hig00] shows one principle towards collaboration. Almost all the practices and principles that are mentioned in this section have already been introduced in the previous sections, especially related to redundancy and learning. The main reason is that without communication and collaboration redundancy and learning cannot take place.

Like customer and other team members were two important target groups in learning, in the same way cooperation with customer and among the other team members can be identified here. Third category in communication and collaboration is visualization.

Cooperation with customer is emphasized in XP due to the on-site customer practice [Bec99, pp. 52-53]. Crystal suggests having user viewings [Abr02, p. 39] and ASD customer focus group reviews [Hig00] as well as JAD sessions together with the customer [Abr02, pp. 70-71].

The second category includes practices that improve communication among team members. Pair programming and open workspace in XP belong to this [Bec99, pp. 51, 63], as well as colocation in Crystal Clear [Abr02, p. 38]. Similarly sprint planning meeting, daily scrum, and retrospectives increase communication in Scrum [Kni07, p. 67-71; Lef07, p. 45]. Model with others, display models publicly, and model to communicate are practices from AM [Amb02]. All of them should enhance communication in the team. Lean software development that concentrates on reducing waste provides with one more advice that increases communication in the team. The team should avoid handoffs, which means for example that instead of an analyst sending a specification document for a developer, they should rather have a discussion about the specification together [PoP03, pp. 7-8].

The Scrum board including burndown chart [Kni07, pp. 51-52] and Kanban board [KnS09] are communication practices that visualize the progress of the team. They are information radiators that increase the awareness and certainty regarding the project activity [Whi08]. Sometimes these kinds of tools are good for identifying problems in the process. They can for example show if the team is doing too many tasks at the same time [MDK09]. Seeing the progress visually in a board can also be a motivational factor [QuK09].

Notifiable in communication in agile methods is the continuity of it. For example daily scrum meetings are held every day and each iteration should have some kind of customer and team review. Colocation encourages team to talk anytime and pair programming makes the communication between two developers inevitable. Also use of information radiators reveals the status of the project for the team all the time.

3.2.7 Summary of Self-organization and Agile Methods

Table 4 summarizes the foundings from the previous sections. It lists all the agile methods described in Section 3.1 and shows how they support the characteristics of a self-organizing team described in Section 2.4.

Many of the principles, roles, or practices can be found in several columns. For example the Scrum master role is in the autonomy column since the role protects the team's autonomy. On the other hand the role is in the shared leadership column as well since Scrum master should support shared leadership instead of being a traditional command-and-control manager.

Method	Autonomy	Team orientation	Shared leadership
XP	planning game	planning game	planning game
Scrum	Scrum master role (team protection), effort estimation, sprint backlog	sprint planning meeting, effort estimation, daily scrum	effort estimation, cross-functional teams, Scrum master role instead of traditional manager
Crystal	staging (team selects tasks)	staging (team selects tasks)	staging (team selects tasks), cross-functional teams
FDD	project manager (team protection)	build feature list	-
DSDM	developer roles, empowered teams	business study	-
ASD	-	mission-driven, JAD sessions	-
AM	-	-	-
Lean	empower the team principle, self-determination practice, task estimation	-	leadership practice

Method	Redundancy	Learning	Communication and collaboration
XP	pair programming, collective ownership, open workspace, coding standards	tracker role, planning game, short releases, testing, pair programming, continuous integration, on-site customer, open workspace	programmer role, pair programming, open workspace, on-site customer
Scrum	sprint planning meeting, daily scrum, cross-functional teams	Scrum team, sprints, daily scrum, sprint review, retrospective, cross-functional teams, burndown chart	Scrum board, burndown chart, sprint planning meeting, daily scrum, sprint review, retrospective
Crystal	colocation, test cases, notation, design conventions, formatting, cross-functional teams	colocation, progress tracking, direct user involvement, automated regression testing, user viewings, workshops, iterations, monitoring, cross-functional teams	colocation, direct user involvement, user viewings
FDD	~~class owner~~ (neg.), ~~individual ownership~~ (neg.)	overall model, domain experts, inspection, regular builds	-
DSDM	-	ambassador user, active user involvement, fast delivery, fitness for business purpose, iterative, all changes reversible, testing	collaborative and cooperative approach, active user involvement

ASD	-	learn (in adaptive cycle), JAD sessions, iterative, customer focus group reviews, design reviews, code reviews, test plan reviews	collaborate (in adaptive cycle), JAD sessions, collaborating team work, customer focus group reviews
AM	collective ownership, display models publicly, model with others, apply modeling standards	active stakeholder participation, create several models in parallel, iterate to another artifact, model in small increments, prove it with code	active stakeholder participation, display models publicly, model with others, model to communicate
Lean	-	amplify learning principle, feedback loops, measure lead time, WIP limits, short iterations [PoP03, p. 29] or "0-length" iterations (Kanban)	visualize the workflow, avoid handoffs

Table 4: Cross tabulation of agile methods and characteristics of a self-organizing team.

The table shows that the agile methods strongly support communication and collaboration and that way also learning and redundancy. However, it is more difficult to identify practices that increase shared leadership in a team.

The table also reveals two practices that deteriorate a self-organization element. Class owner role and individual ownership practice in FDD are againts the idea of redundancy.

3.3 Theoretical Framework of Building a Self-organizing Software Development Team

This section presents the final theoretical framework that will be used as a basis of empirical analysis. It is based on the initial theoretical framework (Section 2.6) and is enhanced by the foundings from the previous section (3.2).

Figure 7 shows the theoretical framework of building a self-organizing software team. The same blocks that were in the initial framework (Figure 5) are also in this one. The blocks are complemented with different practical guidelines that can be found in agile software development literature. The abstraction level of these practices is higher than in the cross-tabulation (Table 4 in Section 3.2.7). For example instead of reflection workshops, retrospectives, or postmortems the figure contains a general guideline for having end-of-iteration review sessions.

The theoretical framework is a theory based answer to the research question: How to build a self-organizing software development team? The basic idea is to follow the practices listed in Figure 7.

Building self-organization starts with authorizing the team. In addition there should be someone who makes sure that the team autonomy remains. Such a person can be for example Scrum master.

The next thing is to make sure that there is enough communication and collaboration in

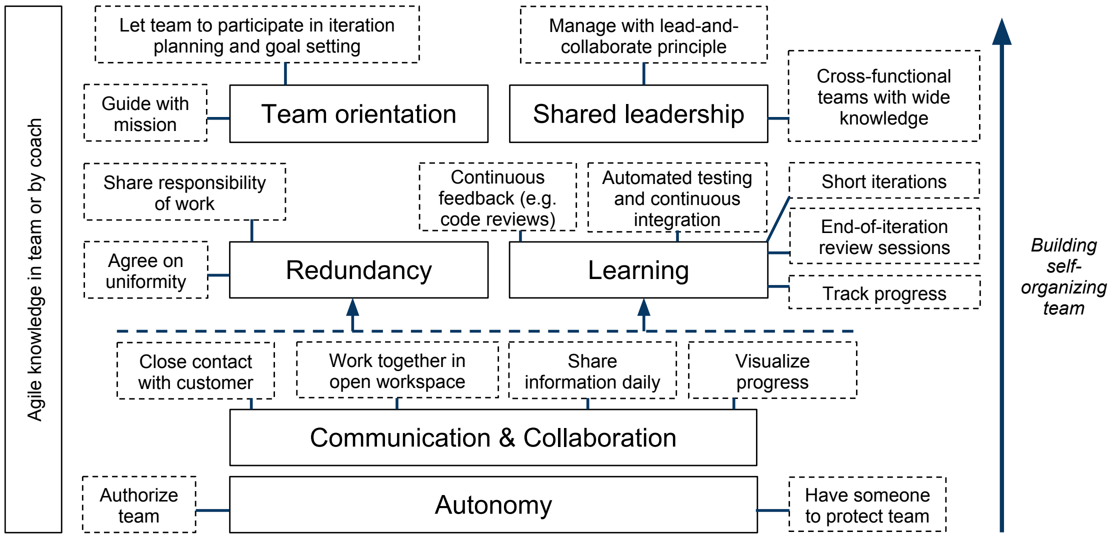
Agile knowledge in team or by coach
Let team to participate in iteration planning and goal setting
Manage with lead-and-collaborate principle
Guide with mission
Team orientation
Shared leadership
Cross-functional teams with wide knowledge
Share responsibility of work
Continuous feedback (e.g. code reviews)
Automated testing and continuous integration
Short iterations
End-of-iteration review sessions
Agree on uniformity
Redundancy
Learning
Track progress
Building self-organizing team
Close contact with customer
Work together in open workspace
Share information daily
Visualize progress
Communication & Collaboration
Authorize team
Autonomy
Have someone to protect team

Figure 7: Theoretical framework of building a self-organizing software development team.

the team. The practices related to this are close contact with customer, work together in open workspace, share information daily, and visualize the progress. If these are not done properly, the other necessary characteristics of a self-organizing team cannot evolve.

The communication and collaboration practices are especially important for redundancy and learning. Additionally redundancy can be increased by agreeing on uniformity and having shared responsibilities, like collective code ownership. Learning is enhanced with providing continuous feedback, having automated testing and continuous integration as well as having short iterations, end-of-iteration sessions, and progress tracking.

Team orientation can be increased by allowing a team to participate in iteration planning and goal setting. This enhances the team's understanding of what is most important to do next. Another practice is to guide team with mission. The purpose is to provide the team with a high level understanding of project goals so that the team can decide what are the best ways to reach the goals.

Practices for supporting shared leadership are managing with lead-and-collaborate principle and having cross-functional teams with wide knowledge. The first encourages to such leadership that lets all the team members participate in the decision making so that the decisions would be made with the best knowledge available. The second is important since the more there is knowledge from different areas, the more useful shared leadership becomes.

All the practices are supported by having someone who understands the agile values, principles, and practices. Such a person can be for example an XP coach [Bec99, pp. 60-61] or Scrum master [Lef07, p. 42]. This kind of person is needed since otherwise the team may stop following certain practices, like Kniberg's experience related to retrospectives

[Kni07, p. 67] shows.

Next the theoretical framework is validated against empirical data. This is done in Section 5. However, before that the empirical research design and the methodology used in it are described in Section 4.

4 Research Design and Methodology

The purpose of the empirical part of the thesis is to test and improve the theoretical framework. This section describes the study context, the research methodologies, and the data collection methods used.

4.1 Study Context

The empirical research was made in two Software Factory[3] (SF) projects. SF is initiative in the Department of Computer Science, in University of Helsinki. It was launched in the beginning of 2010 and this research was made in the first two projects in SF.

	Project 1	Project 2
Time	January-February 2010	March-April 2010
Duration	7 weeks	7 weeks (totally 8 weeks but one week Easter holiday in between)
Team participants	13 (incl. author)[4]	9 (of which three[5] participated to the first project as well)
Additional participants	1 coach	2 coaches (the author and the coach from the first project)
Customers	2 persons (the other was the inventor of the business idea)	An entrepreneur behind the business idea
Business description	Web platform that connects small companies seeking funding and people who want to invest in small companies.	Pluggable and generic recommendation system for online stores; "people who bought this item were also interested in these items".
Product status in the beginning	A private company[6] had made a prototype of the web pages communicating the idea for stakeholders.	The entrepreneur had implemented the core component of the product.
Main technologies	Ruby on Rails	Ruby on Rails, Redis key-value storage
Lines of code	production: 3400, test: 4600	production: 2500, test: 600
Role of the author	Team leader	Team coach
Interviews	6 + 1 customer	4 (of which two participated to the first project as well) + customer

Table 5: Project characteristics.

[3] www.softwarefactory.cc

[4] A 14th team member joined the first project in the last week, and the same person participated the second project as a whole. Nevertheless, most of the time the first team had thirteen members, of which three were in the second team as well.

[5] See the previous footnote.

[6] Leonidas Oy, www.leonidasoy.fi

One SF project lasts typically seven weeks and during that an alpha version business prototype should be built [Abr10]. The team members are mainly students from the department but there can be participants from other organizations as well. For example the first project had three students from Helsinki Metropolia University of Applied Sciences. Table 5 summarizes the project characteristics.

Table 6 shows the team members' previous experience in programming or design work. As can be seen, many of the participants were quite inexperienced. The author had the longest experience of all participants, eight years.

Experience in programming or design work[7]	Project 1	Project 2
none	1 (8%)	2 (22%)
1-2 months	3 (23%)	0 (0%)
4-7 months	3 (23%)	1 (11%)
1 year	2 (15%)	2 (22%)
2-3 years	2 (15%)	3 (33%)
8 years (author)	1 (8%)	0 (0%)
unknown	1 (8%)	1 (11%)

Table 6: Team members' previous experience in programming or design work.

4.2 Research Methodology and Data Collection Methods

The empirical part consists of two case studies. In case studies it is possible to use only quantitative data, only qualitative data, or both of them [Yin03, p. 33]. Although some quantitative data is used, the main approach in this thesis is qualitative. The reason is that the primary purpose of the research is to study qualitative, not numeric data [Uus91, p. 79]. Additionally, with qualitative methods it is possible to get closer to the meanings that people give for phenomena [HiH00, p. 28]. Moreover, qualitative methods are appropriate for describing mechanisms [Uus91, p. 104], which in this thesis relate to the factors that make a team more self-organizing.

In qualitative research possible data collection methods are usage of existing material (like documentation or archival records [Yin94, p. 80]), different observation techniques, and interviews [Uus91, p. 82]. In this thesis mainly the last two of them were used, altough for example the wiki documentation of the projects was used as well. The author observed the both projects in different level of participation (Section 4.2.1). Additionally, totally ten team members and two customers were interviewed after the projects by the author (Section 4.2.2).

[7]Based on the questions asked by another researcher, PhD student Marko Ikonen

4.2.1 Participant Observation

Interviewing (see Section 4.2.2) is a research method for finding out what people think. Observation on the other hand can reveal whether people act like they claim to act. The biggest advantage of observation is to get direct information of the activity and behaviour of individuals, groups, and organizations in their natural environments. It is also an excellent method for studying interaction of people. [HRS00, pp. 199-200]

The advantages were reason why observation was part of the research. More precisely, in the both case studies participant observation was used instead of direct observation. Participant observation is a special mode of observation in which the researcher is not merely a passive observer [Yin94, p. 87]. The main advantage is that the researcher can get a better insight from interpersonal behavior and motives [Yin94, p. 80].

There are also risks in participant observation. At first, the observer may lose his or her objectivity by engaging emotionally to the group or situation to be studied [HRS00, pp. 200-201], and the observer may become a supporter of the organization or the group being studied [Yin94, p. 89]. Another challenge in participant observation is that capturing information may be difficult, and the observer may have to trust on his or her memory [HRS00, p. 202]. In other words, the participant role may require too much attention relative to the observer role [Yin94, p. 89].

The risks were reduced by using multiple sources of evidence [Yin94, pp. 90-93]. At first, the author wrote daily notes in the both projects. The notes contained thoughts about what had happened, how people had behaved, and how a project was progressing. Secondly, some of the team members and a customer from both of the projects were interviewed after the projects (see the following section). Usage of three sources of data (participant observation, notes, and interviews) refers to the triangulation of data sources aiming at corporating the same fact or phenomenon from different perspectives [Yin94, pp. 92-93].

The participation and the observation in the second project were minor than in the first project. Among another person the author's role was to coach the team for example by helping them especially in the beginning and by pointing out some problems that the team was facing. This is close to how Beck [Bec99, pp. 109-110] defines XP coach. Due to the less observation the author asked one team member and the customer to write notes about project. Additionally, the author wrote notes as well.

There is a relevant difference between the two case studies. In the first project the author had no systematic understanding of self-organizing teams, only some experience from the previous work life. This means that as the author had the team leader role in the project, he was not concentrating on making the team more self-organizing but to make it perform better. In the second project, on the other hand, the author had studied literature of self-organizing teams and most of his interventions could be argued by the theory. For example when the team was not reluctant to have its first retrospective, the author explained them why it is good to have such. The intervention in this case could be argued by the importance of learning when building a self-organizing team.

The second project approach is close to the action research method. In action research, the researcher tries out a theory with practitioners in real situations, gain feedback from this experience, modify the theory as a result of this feedback, and try it again [Avi99]. The difference is that in this research there was no action research iterations, there was no "try it again" phase.

4.2.2 Thematic Interviews

To support multiple sources of evidence [Yin94, pp. 90-93], interviews were used as another primary data collection method among participant observation. There are several types of interviews. In a survey an interviewee reads and fills a questionnaire by him- or herself, in an interview study the interviewer asks the questions [Uus91, pp. 90-92]. In this research the latter is more appropriate for several reasons [HiH00, p. 36]: the interviewer has more chances to interpret the answers, he can make specifying questions, the interviewee can provide with descriptive examples, and the validity of the information can be checked for example by observing. Moreover, when meaning constructions are studied, people have to speak with their own words instead of using the terms selected by the researcher [Ala95, p. 73].

Interview study can be divided into three subcategories: written interview, thematic interview, and unstructured interview. Written interview is most appropriate when the material is easy to quantify and facts are gathered. The format of the questions in written interview is rigid. The questions in the opposite method, unstructured interview, are always based on the previous answer. [HiH00, pp. 44-46]

For this research the most suitable method is however thematic interview. In a thematic interview the interviewer concentrates on certain themes instead of detailed questions [HiH00, pp. 47-48]. This means that the themes are same for all interviewees but the questions and details inside the themes may vary. In this research the themes are characteristics of a self-organizing team, described in Section 2.4. The purpose of the interviews was thus to find out for example how learning took place in the projects or what kind of communication there was.

In qualitative research the target group is typically selected based on certain judgment instead of random selection [HRS00, p. 155]. In this research the interviewees were selected based on the author's own judgment. The purpose was to interview people with different personalities, background, and roles in the projects.

Interviews were done right after the projects to increase the reliability of them as participants still remembered the projects very well. In the first project the team members were interviewed in the following week after the project. However, the customer of the first project was interviewed three and half weeks after the project had ended. In the second project the customer was interviewed in the last day of the project and the team members during the following ten days after the project.

The reliability of empirical material is dependent on its quality, which in thematic interviews can be increased by creating a good framework for interviews and by thinking optional supplementary questions [HiH00, pp. 184-185]. In this research the author created a question framework around the themes including several possible questions. There were different question frameworks for team members and customers (see appendices 1 and 2). Another ways to ensure quality is to make sure that the technical equipment works properly and to transcribe all the interviews [HiH00, pp. 184-185]. In this research all the interviews were recorded and later transcribed.

5 Empirical Results

This section describes the findings of the empirical research. During the section the both projects are analyzed and compared at the same time. The empirical analysis proceeds so that at first Section 5.1 briefly presents the both projects in a general level giving an idea how the work was done in the projects, what were the goals in the beginning, and how satisfied the customers were at the end. Section 5.2 identifies how the characteristics of a self-organizing team were applied in the projects. This is done by seeing whether the teams used the practices that were proposed in the theoretical framework and by identifying the consequences of their decisions. For example, if a team had short iterations, did it increase their learning? Or if a team did not have close contact with the customer, what kind of consequences there were in redundancy, learning, or in other aspects of a self-organizing team?

Sections 5.3 and 5.4 summarize the empirical findings. First Section 5.3 compares the teams and all the practices mentioned in the theoretical framework and concludes how self-organizing was built in the projects. Then Section 5.4 identifies the primary empirical conclusions.

The text contains quotes from what the interviewees said in the interviews. After each quote their is a reference telling who said the comment. The references are declared in Table 7. As can be seen, C refers to a customer, TL to the team leader, and TM to a team member (other than team leader). The number in a reference tells the project and letters A-F differentiate the team members.

Role	Project 1	Project 2
Customer	C1	C2
Team leader	(author)	TL2
Team member 1 in both projects	(not interviewed)	TL2
Team member 2 in both projects	TM1A	TM2A
Other team members	TM1B, TM1C, TM1D, TM1E, TM1F	TM2B, TM2C

Table 7: Explanations of interviewee references.

The same references are used inside of the quotes as well when an interviewee mentions such a team member who was also interviewed. All the names have been removed from the quotes. For example the customer names are replaced with the words “the customer”. Some of the interviews were held in Finnish and the quotes from them are translated to English by the author.

5.1 Contextual Project Information

5.1.1 Customers' Expectations

The customer of the first project stated that the initial goal was very clear:

> "The crystal clear goal was that in March 4th[8] we are ready and then we present the goal for a wide range of different stakeholders - - and the company gets funding and is ready to start working." [C1]

In the beginning of the project the customer was a bit surprised of the big team size (13 people) but he was still quite confident:

> "I had a good feeling. . . And especially since there were certain key people in the team that I knew beforehand. That increased my confidence. At least there are a few people who are worried about the project result." [C1]

One thing that increased the customer's confidence was the ready made prototype of the product. He also felt that the domain is simple to understand.

In the second project the customer seemed to have a pretty clear high level goal as well. However, he also had some doubts related to the short planning time he had:

> "The first worry was that I knew about this project about two weeks before it started. I hadn't planned or anything like that, this was just an idea what I'd like to do. So of course then I'm in a situation where I don't know all the details beforehand. I mean I've got this idea but I don't know the details and then I have to come to explain to a group of people what I want done." [C2]

The customer had also some other worries:

> "Initial fear for me was that I end up with piece of software with different modules and components built but that they would never actually work as whole. - - Of course I'm all the time thinking: is it even possible? So is there some technology that it just won't work this way?" [C2]

5.1.2 Team Members' Expectations

The team members' expectations are described briefly in this section. This is useful information since it gives a better picture of what was the starting point for the projects and for example what kind of learning requirements there were in the beginning of the projects.

[8]The second last project day in which the project results were presented for about 70-80 people in the opening ceremony of Software Factory.

As the projects were also university courses, the participants (students) had voluntarily registered to them. For that reason their personal expectations were related to learning, mainly in technical and team level:

> “The personal goal one was learning of this technique and the second was the way of work.” [TM1B]
>
> “I had such an attitude that whether we succeed in making software or not is secondary. Of course that would be nice but the main thing is to learn things. That was my main focus.” [TM1C]
>
> “I was interested in modern operational environment. Besides, as a course it is such which adds people’s commitment, motivates them easily.” [TM2B]
>
> “I wanted just to check how in these days the projects are like. What is agile development, what is Ruby on Rails?” [TM2C]

Before the projects started the team members had very little information. The course web site described the main purpose of the projects but not in a detailed level. As one team member in the first project stated:

> “Well, as it was written on the web site, it would be a business prototype so I was just thinking that there would be some specification and we just need to write some application that will match to that specification. And that’s it. I wasn’t much aware of what the goal is.” [TM1F]

The situation got better when the customers visited the teams in the beginning and described what they were supposed to develop. The first project team had an existing prototype that clarified the idea quite well. However, the second team seemed to be in a confusion after the customer had explained the idea to them.

> “At the beginning we understood that we have to develop a web service that can be used by online web stores. But there was quite a lot ambiguity and we were not sure how we will perform. Everyone had many questions in their minds.” [TM2C]

One team member in the second project, the team leader, however understood the concept pretty fast:

> “The first reaction when I heard about this was that do we ever get this ready? But when I started to think that what actually needs to be done, I was thinking that there is not that much to do. We will get this done ahead of time.” [TL2]

5.1.3 Project Timelines and Work Processes

This section gives better understanding how work was done in the projects. It is important information to be used in Section 5.2.

Figure 8 describes the project timelines based on the author's and a participant's notes. The first week in the both projects was spent increasing the understanding of the customer domain and learning the core technologies, Ruby on Rails as paramount. The reason for latter was that only some of the team members had experience on it. In the first team four people of thirteen had taken a course in Ruby on Rails and in the second team three of nine had some previous experience on it.

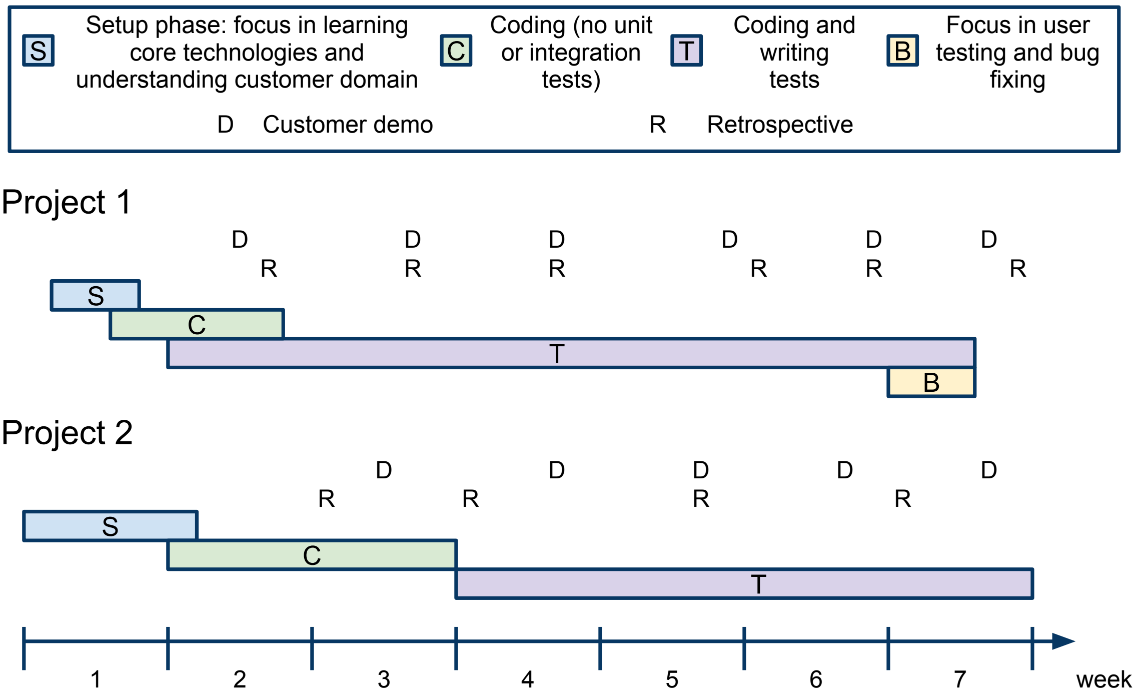

Figure 8: Project timelines (based on the author's and a participant's notes).

Coding of the actual product started at the end of the first week in the first project and in the beginning of the second week in the second project. The second team had to use more time understanding the customer domain and designing the architecture, which delayed the start of actual coding. The first team on the other hand was creating a standard web application, and Ruby on Rails provided them with a ready-made architecture.

In both projects the team members did not have much experience on writing tests or doing test-driven development. That was the main reason why in the beginning the code was written without unit or integration tests. In the second project another reason was that the team had an idea to first create a prototype that contains all the functionality without focusing to the quality of the code, and in later stage to write the actual product with better code quality. For this reason writing tests started two weeks later than in the first project.

The beginning of the last week in the first project was spent on testing the application thoroughly by the team members and fixing bugs and inconsistencies. In the second project product development was done even on the last day of the project. The second team did testing and bug fixing as well but the whole team was not concentrating on that in a same way as in the first project.

Both projects had similar kind of workflow. Based on the experiences in the first project the author had written a guide for Software Factory teams [Kar10]. This was available for the second team and it probably influenced how the team works. Another reason for similar work methods was that three people participated in the both projects and brought ideas from the first project to the second one.

As Figure 8 shows, the teams had customer demos for presenting the tasks they had completed during the previous week. In the demos the teams also got new features to implement or refinements to the existing requests. The new tasks were listed in the task list, which was then reprioritized. During the week the teams developed as many features as possible and then presented the achievements in the next customer demo. This workflow is visualized in Figure 9.

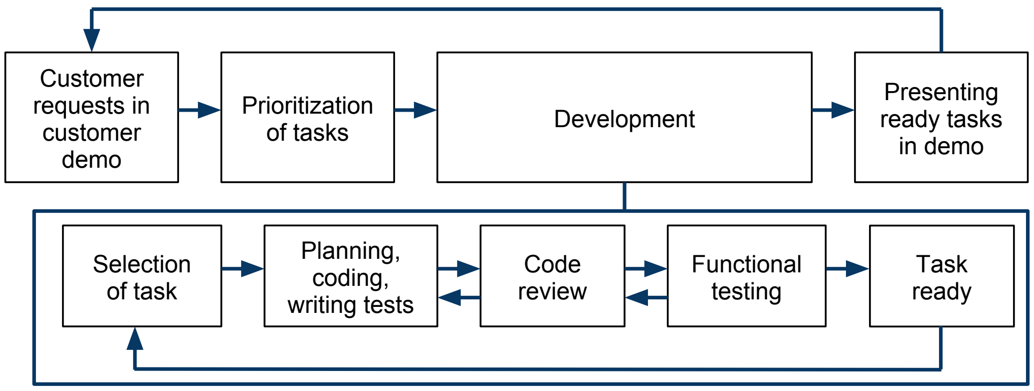
Customer requests in customer demo
Prioritization of tasks
Development
Presenting ready tasks in demo
Selection of task
Planning, coding, writing tests
Code review
Functional testing
Task ready

Figure 9: Workflow in projects.

Figure 9 shows also the development lifecycle. After selecting a task a developer self or together with someone else planned it and wrote the necessary code for it. The tasks were reviewed and tested[9] then by another team member. If the quality in either phase was not as the reviewer or tester wanted it to be, the task was returned back under work. Otherwise the task was considered ready and the developer could take another task. Both teams had definition-of-dones defined for each phase, which helped the reviewer and tester

[9]The first team called the functional testing phase as “in QA” (quality assurance) and the second as “user acceptance testing”. Both terms are a bit misleading since at first, code review stage also goes under the category of quality assurance and secondly, there was no user or customer who tested and accepted the work done. Both teams nevertheless conceptually referred to the idea of functional testing.

to make the judgment whether the task was acceptably done.

The teams had every morning a standup meeting at 10 o'clock. In addition, the teams had end-of-iteration review sessions, called retrospectives (see Figure 8). The first team had totally six of them, the second one had four.

Figure 10 reveals one more aspect to the work done in the projects. It shows how many commits to the version control system were done each week on average. For example when the second team did about 90 commits during the fourth week, on average it was ten commits per person as nine people were in the team.

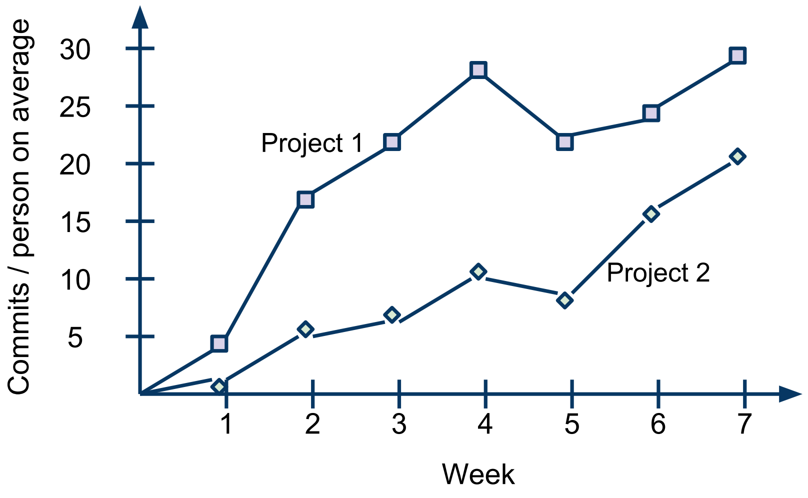

Figure 10: Commits to version control system per person on average in each week. Values are calculated by dividing the total amount of commits by the team with the number of team members.

The curves in Figure 10 should not be necessarily compared with each other since the teams might have had different committing habits (the first team emphasized commit often approach). Nevertheless, it shows some difference between the teams that will be assessed later in more detail. In addition, it reveals that the first team was able to achieve a high level sooner than the second team.

5.1.4 Project Results According Customers

Both customers seemed to be satisfied with the results of the projects. The first customer commented:

> "It [the result] was extremely good. It was brilliant. It contained a lot of things that I didn't expect that would have been possible. - - I would say that the team exceeded my expectations multiple times." [C1]

The customer of the second project was satisfied as well but perhaps a bit more cautious. He was basically afraid that there are some problems that will be revealed after the project:

> "It [the system] was just there where it was supposed to be. I think it has got to the stage where the whole system works." [C2]
>
> "I think now I'm happy. I would say even very happy. But I hope I wouldn't be too optimistic until the stage that everything's checked. But ok if I had an expectation or sort of hope at the beginning of the project, I think I've gotten there. And everything is there, it has all the things that I asked for plus a little bit more. Which I think is quite impressive in short period of time." [C2]

The both customers had had previous experiences on projects where more planning was done beforehand. Based on experiences in these projects they liked the flexibility and adaptation in the projects:

> "It tells that as a customer I learned during the journey, and it was good that I wasn't committed to a certain implementation model. I felt good that there was no contract saying that we should do something else than what I later realized that actually should be done." [C1]
>
> "So if I was gonna do a software project on my own, this is the way I would do it. I would just start writing things and trying things out and then build up small modules that contain the functionality that I know that I need and then start trying to put them together." [C2]

The customer testimonials indicate that the teams have done correct things during the project. It automatically does not mean that the teams were self-organizing. However, if Section 5.2 shows that the teams really were self-organizing, it gives some evidence that self-organizing teams can produce good results.

5.2 Characteristics of a Self-organizing Team in Project Contexts

This section identifies how different characteristics of a self-organizing team appeared in the projects. Each characteristics is discussed in its own subsection. The idea is to examine how the different practices defined in the theoretical framework (Figure 7 in Section 3.3) were applied and what kind of consequences there were.

5.2.1 Autonomy

According to the theoretical framework, autonomy is the basis for self-organization in a team. If there is no autonomy, self-organization cannot evolve and there will be problems with the other components of a self-organizing team. The framework suggests two practices related to autonomy: authorize team and have someone to protect the team.

The projects were done under Software Factory initiative that has own management and that uses university courses as a main instrument to get work force. In the first project the management decided that the core technology will be Ruby on Rails and the team should use Kanban as their process model. In the second project the team was able to decide these by themselves. Even though the technology and the process model had major impact on the team's work, they can be considered as normal constraints that are defined by the management before a project begins. More important is how the management influenced the teams' work during the project.

The both teams were very autonomous. The management did not intervene to the daily work of the teams. As the projects were university courses at the same time, there were no either teachers deciding how the teams should work. This is also what the team members acknowledged when they were asked who were the people outside of the team influencing it:

> "We were pretty much alone. The only who directly influenced us were the customers." [TM1C]
>
> "Well, the customers affected us most. They gave us the biggest 'commands'." [TM1D]

Besides the customers there were team coaches, one in the first project and two in the second one, including the author. They also were seen as people influencing the team:

> "At least the customers and to some extent the coach. There probably weren't any other outsiders." [TM1F]
>
> "The coach, you [the author], and the customer. That's about it, I guess." [TM2B]

This means that the first practice in the theoretical framework, authorize team, was clearly fulfilled. On the other hand, it might have been that the customers would have affected the teams so that the external autonomy would have been broken. However, as will be seen in later sections, this was not the case.

The second practice related to autonomy is to have someone protecting the team. In these projects that was not necessary since there were no external distractions. On the other hand, there is something that can be seen as such. Namely in the first project the customers usually gave pretty high level requirements in the customer demos. If the customers started to design the implementation in too detailed level, the team prevented it and in that way protected its own autonomy. Also the customer was pleased with this approach:

> “I liked that when I started to talk things in too technical level, the team or someone in the team said that ’sounds technical, we will get back to this in a week’.” [C1]

In the second project the approach was pretty much the opposite. According to the customer the discussion in the demos went often to quite a detailed level:

> “They [the questions] were mostly technical but not big questions. More like ’do we do this?’ or ’do we need..?’. Well, it was not this question but ’do we need this green button or red button?’ type of question.” [C2]

The customer felt this a bit uncomfortable since in some cases the team was not able to handle simple details by themselves. When this is compared to what the first project customer said, it shows that team autonomy can be a positive thing for external parties as well, as long as the team uses the autonomy. In other words, the management or the customer of the team can feel relieved when the team is able to manage the details. In addition, this saves the management’s or the customer’s time.

As a conclusion, the both teams were very autonomous. As the theoretical framework suggests, the teams were authorized. On the other hand, there was no need to have people protecting the teams.

As defined in Section 2.4.1, besides external autonomy there are two other autonomy levels as well: internal and individual. Internal autonomy is discussed more below in the shared leadership section (5.2.3) as these two are closely related to each other. Similarly individual autonomy is dealt together with team orientation in the following section. The reason for the latter is that in the projects the major factor increasing individual autonomy was that the team members were allowed to choose tasks on their own. However, this does not automatically mean that the selected tasks were important for the teams. In other words, individual autonomy has to be considered in the same context with team orientation.

5.2.2 Team Orientation

Team orientation is important for a self-organizing team so that all the team members focus on the same goal instead of each having their own goals (see Section 2.4.2). The theoretical framework suggests two practices to enhance team orientation: guide with mission and let the team to participate in iteration planning and goal setting. However before getting to these, first the relationship of individual autonomy to team orientation is described, as mentioned at the end of the previous section.

The most visible sign of individual autonomy was that the team members in the both projects were able to choose the tasks they did. However, the selection was not made only based on an individual’s interests but also the team goals were considered:

> "Well, everyone individually, they are taking tasks from the ticketing system. And they were just checking what are the priorities of the tasks and they are assigning these tasks to themselves and then everything is going." [TM1A]
>
> "If there was something that I was interested in. If none, then I asked others if there's something I can do. - - Of course I picked the tasks with as high priority as possible." [TM1D]
>
> "We generally used to pick up tasks on our own. No-one used to assign that you do it this way." [TM2C]

Especially the second project interviewees mentioned one important aspect related to task selection. They said that the tasks were selected based on how each individual feels that is capable of doing a task. So even though there are tasks available with a higher priority, a team member leaves them untouched if he or she feels that the tasks are too difficult. On the other hand, those who were comfortable with their skills mainly focused on more difficult tasks:

> "Usually I took relatively critical components under work so that we get the important things done. During the project I noticed that I have most trust on these two team members. If they selected some important component, I was confident that they manage to do it. Otherwise I tried to take it myself." [TL2]
>
> "It was a bit difficult task and I felt that nobody wants to start thinking about it. And it was a critical task so I just started doing it." [TM2B]

Not taking the most important tasks may sound like the team orientation was not considered. However, it might be better that an individual does such tasks that are not too difficult. This way all the team members can be utilized in their maximum capacity. On the other hand when building of self-organization is considered, it is important that the team members are willing to take challenging tasks and learn that way.

Another risk in that the most important tasks are not taken is that no-one is taking the complex tasks. In the second project this occurred at least once. After one customer demo the team had two most important tasks. The other one was to do some sort of stress testing for the application. However, that was not done during the week.

The team members did not always select tasks by themselves. Sometimes the team members asked the team leaders what should be done next and sometimes the team leaders proactively searched developers for critical tasks:

> "I'm not completely sure how independently I chose the tasks. At least many times I looked that this task would be nice and I asked if I can take it. I think in some cases I got also tasks from you [the author] as well. But there were never tasks that I wouldn't have wanted to do." [TM1E]

> “Some tasks I got so that ‘hey, that looks interesting, and it is in a pretty high priority, let’s take that’. Some I got so that you said: ‘This should be done, would you like to do it?”’ [TM1F]

> “There were tasks that were especially signed to me by the team leader because he thought there are two or more people [available] but they are not very good with these things. He assigned two or three tasks to me that I could do. And the other people always tried to take their own tasks from the ticketing system. Whatever they think they have trust.” [TM2A]

If the team members followed the prioritization, the next question is how the tasks were prioritized? Both teams used pretty much the same practice. In the customer demos the teams wrote notes based on the new requests and the notes were transformed into the tickets, usually by the team leader. The team leader also mostly set the priorities, but it is important to notice that after the customer demos it was rather obvious for the teams what the most important tasks were.

> “So in the demo once we have demonstrated the features we had developed, we discuss these things with him [the customer]. We created the list that these are the next most important tasks that the customer wants to have. And then [TL2] used to create new tickets based on our notes and then we started working on those tickets from the next day.” [TM2C]

Especially in the first project the prioritization during the customer demos was emphasized by the team. Usually the team leader (the author) asked at the end of the demo what are the most important things to be done during the next week. The customer considered it as a good practice:

> “The team leader asked that on what they should concentrate during the next week and this was asked from the both customers. I felt it was actually a good thing to do. It makes you to summarize the thematics into a couple of sentences.” [C1]

In the second project the same question was sometimes asked but not as consistently. Nevertheless, as the customer demos were an important source for both new customer requests and prioritization, it was useful that all the team members participated in them. In the first project one of the team members was not able to participate in all demos and he commented:

> “I had such a feeling that it would have been useful to be in the demos. I would have known better what is happening in the project. - - Maybe the biggest disadvantage was when I did code review or QA for others. I didn’t have a clear vision if they were doing the right thing.” [TM1E]

The theoretical framework suggests that the team should participate in iteration planning and goal setting. As the customer demos were the main events for planning and goal setting and as the majority of the team members were present in the demos, the practice was followed. It was also very useful for the team orientation.

Another practice to increase team orientation suggested by the theoretical framework is to guide with mission. The purpose is to provide the team with a high level understanding of project goals so that the team can decide what are the best ways to reach the goals. The practice was used in the both projects to some extent. In the first project it was emphasized in the demos and in the second project it was present in the beginning of the project. As was mentioned in the previous section, the first team usually received rather high level requirements from the customer, although there were small detailed requests as well. One example was a feature called reputation building. The customer asked for this in the demo but was not able to give clear instructions what it actually means. He was nevertheless satisfied with the result:

> "For example reputation building was analyzed in a triangular model, which in my opinion was a very robust solution. And the team was able to demonstrate it with the smart board very systematically - - I was very convinced." [C1]

As came out in the previous section, in the second project the discussion in the demos was in a lower level. However, the guide with mission practice was there in the beginning of the project when the customer presented his idea about the system to be developed. The customer had drawn a model that represents the system but there were no implementation details in it. Instead, the team managed to create its own solution, which it then showed for the customer. Afterwards the customer felt happy about this approach and appraised the team as well:

> "I think it was much more efficient way just throw the idea out, this is generally what I want to happen. And they managed to figure it out. I think there was of course a couple of iterations where everything wasn't completely understood and there was some confusion but I think, I suspect that the fact that there was a team working together, actually helped quite a lot. They were able to take a lot of information and then sort of simulate it quite quickly and start working with it more or less right a way." [C2]

The empirical data supports the theoretical framework. When a team can participate in iteration planning and goal setting, it increases team orientation. The reason is that it makes the team members to understand what is important, which is taken into account in the task selection. The freedom to select tasks according to an individual's preferences, assuming that the priorities are followed, is on the other hand a practice that has a positive impact on an individual's autonomy without jeopardizing the team orientation. Guide with mission practice increases a team's understanding of high level goals and gives the team a possibility to select such an implementation that best meets the goals.

5.2.3 Shared Leadership

There are two core issues in shared leadership (see Section 2.4.3): those who have the best knowledge decide and everyone should have a possibility to participate in decision making. The theoretical framework suggests that there are two practices supporting this, manage with lead-and-collaborate principle and have cross-functional teams with wide knowledge. In order to better understand how the teams were led, first it is described what were the team structures and how they evolved.

In the both projects there was a team leader. However, neither of them was appointed beforehand, rather these persons took the roles for themselves. In the first project the author became a team leader in such a way that he just started leading the team in the beginning when most of the team members were somewhat confused. Probably one significant factor was that the author had much more previous work experience than any other team member.

In the second project one team member became a team leader in a similar way. This person had participated in the first project as well and thus had experience on a similar project, even though he had not much previous other work experience. When an interviewee in the second project was asked how the team leader was selected, the answer was:

> "It [the team leader role] was not given but it happened in a way that he became our team leader. One of the main reasons was that he was present for the first project for the full time so he knew all the processes very well. Second thing, he was technically good, he could visualize the whole product picture better than the others initially. - - The third thing was that at some point it happened that people committed to few tickets but then they could not finish it. At the time he took responsibility on his own, he worked hard late at night or anyhow. - - So he had all the qualities of taking the lead role, guiding the team, and at the same time giving enough time that a team leader would do." [TM2C]

It is noteworthy that in the first project the same person was rather quiet and did not take any leading role. Nevertheless, the examples from both of the teams emphasize the high external autonomy the teams had; the roles in the teams evolved internally instead of external decisions.

In both of the teams there were certain other key roles as well. In the first team it was decided that the people who have taken the Ruby on Rails course, and are thus the most experienced in that area, are called as senior developers. A senior developer was for example responsible of the technological training in a sub team in the beginning of the project and was allowed to do code reviews. In the middle of the project the latter limitation was removed so that anyone could review the others' code. The senior developers also had a major role in making certain key decisions in the beginning of the project:

> "I think I've been working on small issues. There was a layer of people who could handle big issues. I think it was you [the author] and the senior developers." [TM1A]
>
> "I think the biggest single thing was the definition-of-done. - - It was so that the most experienced of us were discussing about it and the others quietly accepted it since they didn't have any better ideas." [TM1E]
>
> "In the beginning we decided things like definition-of-done. - - I think the decisions were made in a rather democratic way considering that part of the team, including me, had no much idea about it. There wasn't much to say since you didn't know things." [TM1F]

Even though not everyone was giving their input to the important decisions, it does not mean that there was no shared leadership. Actually quite the opposite since the examples show how those who had the best knowledge made the decisions. This is exactly what shared leadership means. In addition, in the important decisions everyone was involved, like in shared leadership should be.

Similar roles and behaviour could be seen in the second team as well. There were three or four people who were more experienced and took bigger role in the project than the others:

> "I feel that me, [TL2], and [a third team member] were somehow the driving force there. The others perhaps did not have technical knowledge or courage to make so much decisions." [TM2B]
>
> "And that was one thing that these four people, they are like excellent team members and contributers to the team." [TM2C]
>
> "I think that me, [TM2B] and [a third team member] participated planning most actively, like how the system should work. Or we had a clear vision. The others were more cautious and I felt that some of them did not have much idea how it should work." [TL2]

The retrospectives were events that were used in a team level decision making. They were also seen as collective events where the whole team discusses equally the project. This is supported by the interview data since the interviewees did not name any particular person or people who suggested or decided something in the retrospectives.

> "I believe that all the decisions in the retros were done correctly in a way that everyone accepted the decisions. And it had an impact [on team behaviour]." [TM1C]

> “[Until] second or third week we did not do test cases. And then we had one retro and we decided that now we will strictly follow them [the rules of writing test cases]. And after that retro in one week the progress was very good. Because in one week whatever, whoever wrote, they started doing test cases as well.” [TM2A]
>
> “First two weeks we just used to implement things. - - In the next week in the retro we all decided that it should be a good idea that people will write test cases only if we make some kind of rule. Otherwise everyone is just trying to finish the tickets and make it to the demo and then we will lack of testing. To avoid that lack of testing we made a rule that we will not pass the code review stage until we have the test cases.” [TM2C]

Even though the teams decided together about something, it was not always followed. The second team had problems with absence of people and people being late from the daily meetings. This was discussed in the retrospectives but the situation did not get much better:

> “In the second project we probably discussed in every retro that people are away a lot and they do not come in time, and that never got better.” [TL2]

As in the first project certain people did not much participate in decision making since they “had not much own experience in these things” [TM1D], the same appeared in the second project as well. One team member tried to involve the people with less experience because she felt that everyone should take part in important decisions:

> “[My role was] sometimes involving all the team members to take the decisions. It was not like a leader role but it was more like getting everyone together and taking the decisions so that they are not pending. Like one day we were discussing that Kanban process was not well working. And we were three people discussing: me, [TL2], and you [author]. And then the others were there but they did not start participating it. The process was not just for we three but it was for all of us who were there in the room. So before taking a decision I asked everyone: ’What do you think, is this the correct number that we are going to change?’ Then on some other occasions like we were discussing that how to create this prototype and everyone had different thoughts. And in some people. . . It used to happen in our team that few people have strong views and the other people do not say their views. So I used to ask each one of them that do you agree on this and what do you think of this? And then we used to come to conclusion for many of tickets.” [TM2C]

The examples indicate that the teams were using lead-and-collaborate principle as the theoretical framework suggests. People were involved in decision making and everyone was able to take part. There was no single person making all or most of the decisions.

However, the second suggestion by the framework, cross-functional teams with wide knowledge, is not that obvious. In the both teams certain people had enough knowledge and skills to make important decisions but some did not. This means that shared leadership was not used as much as it could have been used. On the other hand, shared leadership also means that those who have the best knowledge decide. That was clearly done in the projects.

5.2.4 Redundancy

Redundancy means that team members have to be able to do the other team members' tasks (see Section 2.4.4). This requires that they have to have necessarily skills and relevant information related to the tasks. The theoretical framework suggests two practices enhancing redundancy: agree on uniformity and share the responsibility of work. In addition, the communication and collaboration practices should support redundancy as well.

The interviewees were asked how well they knew what the others were doing. Especially in the first team, perhaps a bit unexpectedly, they denied it:

> "Very badly. In practice this appeared to me so that the layouts changed in certain places and there were new links everywhere. I mean I wasn't actively aware of what we had done." [TM1C]
>
> "Not that well." [TM1D]
>
> "I wasn't aware practically at all. - - I felt a bit guilty that I should know better where this project is going but in practice there was not much harm." [TM1E]
>
> "Actually not very well. Well, I could have find out but usually it wasn't that important related to what I did. There were after all so many fields." [TM1F]
>
> "I wasn't usually aware, I was focusing on my own tasks. Elsewhere happened something but it wasn't usually related to what I did." [TM2B]

When the same interviewees told about the daily meetings, the message was a bit opposite:

> "I like very much the daily meetings. - - For example in the beginning when the people didn't yet know how to ask help, in the daily meetings there was many times 'I am struggling with. . . ' and then someone said that 'ok, we can look that after the meeting'. - - And I think there were many times that someone said: 'I am going to do this'. And then someone else said: 'Would you rather do this since it is more immediate or important?' - - And this way I think there was a lot of communication." [TM1C]

> “The daily meetings were a good way to see what the others had done. Since during the work day there was no time and interest to ask around: ’What are you doing there?’ And there was no time to see what each one is doing and start to investigate the code. But in the morning you found out what someone had done in the previous day, what they are going to do today and that way understand what is happening there.” [TM1D]

It might be that people do not recognize that they actually are quite well aware what the others are doing. Another explanation is that the first team was quite big with thirteen people in it. It is difficult to remember what the rest twelve are doing at the same time. Moreover, it is a matter of details. Whether one constantly knows what each and everyone is doing or whether one has a generic idea what is going on in the project. In addition, the team members seemed to be aware of such people’s tasks that were close to their own tasks. As two team members in the first project stated:

> “If I’m just doing my task and in the morning I will hear people telling what they are doing, it will click on my mind then. Ok, these features are like somebody is doing that. If I’m more interested, I can ask them.” [TM1A]
>
> “If somebody’s doing something similar than what I am, I try to be up to date.” [TM1D]

The level of redundancy can be measured by defining how many is able to do the same tasks. In the first project there seemed to be no problems with this. Only with few challenging tasks there were less people capable of implementing them, at least in a similar time frame. The interviewees did not mention any task that only one person would have been able to do.

> “There were many people, majority of people I should say, that they were able to work on the task what I had been working.” [TM1B]
>
> “Well, considering that Ruby on Rails was completely new for me, I’m sure that probably almost anyone could have done them.” [TM1F]
>
> “In the beginning it was a bit difficult to say whether anyone could have done the same things but later probably anyone. Probably the only ones that everyone could not have done were the bigger features, the newsfeed and the map, with which I personally didn’t have anything to do. In some tasks you could see that it requires skills to do them.” [TM1D]

In the second project, on the other hand, there seemed to be fewer people that were able to implement the core functionalities. These were the same team members who had the biggest roles in the team (see the previous section):

> “Well I would say that [TM2B] and [another team member] could have done them, and perhaps [a third team member] some of them. The others would have required perhaps a bit more training. . . ” [TL2]
>
> “Difficult to say how many other. But at least [TL2] and [another team member] could have done it.” [TM2B]

Code review was part of the work process in the both teams (see Section 5.1.3). In the first team the team leader and the senior developers (totally four) did the code review and later in the project anyone was allowed to do it. In the second team it was however sometimes difficult to get people reviewing the others’ code. The tasks in the code review column in the team’s Kanban board remained there rather long time. In addition, the same team members were burdened by the code review responsibility as there were not many people who could do it:

> “My code review was done by [TM2B] mostly.” [TM2A]
>
> “It was a bit that I was ready to review others’ code but I felt that [TL2] was the only one who wanted to check my code.” [TM2B]
>
> “I think that [TM2B] reviewed all my tasks. ” [TL2]

Both the certain people being responsible on the core tasks and the same people being the only one reviewing code indicates that the responsibility of work was not well shared in the second project, like the theoretical framework would suggest. Besides that the team members used to constantly work with the same features:

> “Sometimes it happens that if we present it [a task] in a demo and it comes with revised requirements, then I continue working on that.” [TM2C]
>
> “It [transferring a task to someone else] probably wouldn’t be easy because I did pretty much, and everyone else as well, individually. Even though we sometimes asked advice and discussed, the implementation was done alone.” [TM2B]

This refers to a certain level of individualism or specialism that is not seen as good for redundancy (see Section 2.4.4). In the first project individual work was done as well but the team members were not that much concentrated on the same features. If the customer asked changes to a certain feature in a customer demo, the change was not necessarily implemented by the same person who initially had wrote the feature.

One way to measure the level of redundancy is to see how absence of people affects to team performance. When the interviewees were asked to speculate what would have happen if some team member would have gotten sick, one interviewee in the first team took up a noteworthy point:

> "But again there was this splitting of tasks so we did have trust in our team. When the coworker was missing and his task was a little bit huge or something, we had the task in different pieces." [TM1B]

The idea is that if the tasks are small and a developer doing a task gets sick, not much work is lost even if another developer would start the task from scratch. This seems to be good "insurance" against absence of team members.

Agree on uniformity is another practice to increase redundancy suggested by the theoretical framework. Both teams agreed on coding conventions and in the code reviews it was supposed to make sure that these were followed. As defined in the definition-of-done of the both teams, the developers were supposed to have "code done in 'Rails way"' and the code was supposed to be "readable code (no useless comments, good variable and method names)". In the second project this was however not always followed. The team leader commented:

> "At first I checked that it was something of the kind, like it was supposed to do in Rails or in Ruby, based on the experience I have. In some cases it was that people had done it like it was not supposed to be done. If it was somewhat understandable, I let it pass. Maybe I should have been more rigorous especially in the beginning of the project. It would have perhaps reduced at later stage that there will be more of that [bad code]. - - I was too kind perhaps." [TL2]

There is no evidence how much it affected to redundancy that not all team members followed the coding conventions. The team members did not report that they would have had troubles with the other members' code. On the other hand, it was not asked directly and as was seen above, the second team members worked often quite individually. If a same person always works with the same classes for example, the need for code conventions becomes less important, as long as the person remains in the project. Nevertheless, when the code review was not rigor enough, it had an impact on learning, as will be seen in the next section.

As a conclusion, the empirical evidence shows that the suggested practices support redundancy and if the practices are not followed, it can cause problems. For example if the responsibilities are not shared, it may burden some of the team members. The empirical data also showed that keeping tasks small is one way to improve redundancy.

5.2.5 Learning

A self-organizing team must learn in many levels so that they can better reach the project goals. The theoretical framework suggests five practices that enhance learning in a team:

continuous feedback, automated testing and continuous integration, short iterations, end-of-iteration review sessions, and track progress.

Providing continuous feedback is important because without feedback it is difficult to learn (see Section 2.4.5). In the previous section the team leader of the second project told that in the code reviews he was not guiding the more inexperienced team members to the right direction. Modest feedback in the code reviews was confirmed by another team member:

> "Code review was very important but I don't know actually... Because when I worked in my previous projects, every time when we send our code for the review, we had very strict standards defined. And when someone was doing the code review, we always used to get comments, some new things. But when it came to this project - - It was more like that when the ticket is in the code review stage, - - our main goal was to close the ticket. So probably we compromised on the quality of the code." [TM2C]

Compromising on the quality of the code is one issue, avoiding learning is another. Code review is not meant only for making sure that the code base remains a good quality but especially when the team has inexperienced developers, code review is for learning as well, providing feedback. This was recognized in the first project:

> "I think the code review stage was extremely good because you got immediate feedback." [TM1C]
>
> "Then if someone noticed that this is going completely wrong, you got honest feedback. And then you asked how should I do this and what would be a better way." [TM1D]

In the first project the author noticed sometimes while reviewing code that a developer had spent lot of time doing a task in a wrong way. Although rework is part of agile software development [Hig00; PoP03, pp. 19-20], it is not something that should be pursued. For this reason the author suggested for the second team that besides code review they would include plan review to their process as well (Figure 11).

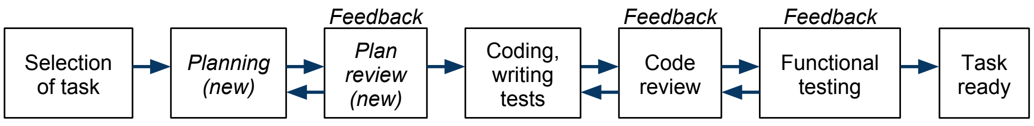

Figure 11: Author's suggestion of improved development process including separate planning and plan review phases (in the second project).

The idea in plan review was that the sooner a developer gets feedback, the better it is. For example instead of wasting one day for coding a feature in a wrong way, the developer would already in the beginning hear that there is a better way to do it. The author considered this important since the team had many inexperienced developers.

The second team changed the Kanban board by adding a new column to the beginning, called Planning and plan review. However, they did not deploy the change to their daily work:

> "It was effective thing but I've seen very less use of it in this project." [TM2A]
>
> "I'm not completely sure what the others did since at least I didn't review plans even once. I think it was such a column that the people did not quite get it and so it was silently ignored." [TM2B]
>
> "I would not say I did not follow it but we all as a team could not follow it properly. Because it used to happen that our ticket is there in Plan and plan review and we don't know how we plan it. And it also came to one of our retro meetings - - that we created the column but we could not follow it the way we should have. Otherwise we could have avoided a few rework tasks. - - Probably not all of us knew how to plan it. And then they didn't bother to take advice from the persons who could advice them." [TM2C]

The second comment indicates that perhaps plan review was not used since some of the team members did not know how to plan their tasks. In a way this even emphasizes the need for increasing communication before coding is started. Of course sometimes a developer knows how to write some piece of code only after he or she has done it. However, with better communication it is possible to save the developers' time.

Both teams spent some time for learning in the beginning of the project (see Figure 8 in Section 5.1.3). This meant that for example during the first week the developers created an excercise application. The main purpose was to study Ruby on Rails and Git version control system that were new technologies for most of the team members. The first team also used time for studying how to write tests. In the beginning this slowed down the progress but there was a belief that it will pay back later:

> "I pair programmed the whole day. We wrote very thoroughly RSpec tests and it is really slow. Admittedly there is little suspicion whether this is a right decision - I mean learning how to write tests - but on the other hand I have to believe in the big ball theory: We try to get the big ball moving so that it will have great forces when it achieves its full speed. The beginning is slow but the reward is at the end. We could get a small ball moving quickly but in the long run we would not get as much done." [author notes, project 1]

The first team also organized training sessions inside of the team. For example the author spend half of a day with four other people by teaching them test-driven development

(TDD). A senior developer held a session for the whole team where he showed how to do behaviour-driven development (BDD). Some of the team members were absent at that time and the author had a similar session with them later on. Probably this was one reason why in the first project the whole team was able to contribute especially at the end of the project. The second team did not have similar training and the certain team members were responsible for the most of the development (see the previous two sections).

The theoretical framework suggests automated testing and continuous integration as one feedback tool and that way as a practice to enhance learning. Both teams used continuous integration but there was a big difference how much tests were written (see Table 5 in Section 4.1). One reason was that the second team started writing tests much later (see Figure 8 in Section 5.1.3) and in practice they did not do test-driven development:

> “We discussed TDD but it was never done in practice. - - I think there was such a pressure all the time that we need to get something done. So let’s not concentrate on TDD since many of the team members don’t even have a clue of it.” [TM2B]

The reasoning of the team member in the second project is in contradiction with the reasoning of the author during the first project (see the “big ball theory” above). This indicates that when building of self-organization is considered, it is important to invest in learning in the beginning. Otherwise the team is forced to make compromises, and possibly not all the team members are able to contribute even later as much as wanted.

There is no quantitative evidence how many defects were found and fixed in the projects. Nevertheless, the author’s experience was that the first team struggled very little with the quality problems. As the test coverage was good, there was immediate feedback if something was broken. The second team on the other hand faced problems with the system during the last days as the “last minute changes” were done before the final customer demo.

The third learning practice in the theoretical framework is to have short iterations. The goal is that a team and a customer can learn during the project and adapt to the changing requirements if needed. Both of the teams used Kanban process model that does not prescribe iterations [KnS09, p. 13]. However, in the first team a weekly rhythm evolved containing a customer demo once a week. The second team followed the same rhythm (see Figure 8 in Section 5.1.3) meaning that the both teams in practice had one week iterations.

The short iterations were especially important for the first team. The customers did not have a clear picture what the application will be like and every week the direction was somewhat changed:

> “We had this practice, these customer demos, that became critical communication points. - - I call them occasions to mediate feelings, at least I felt that

> I'm extremely fired up and excited about what has been done, and I honestly commented what it looks like. What is the direction and have they done what critically was supposed to do? - - In this kind of initiative the goal is crystal clear but still extremely movable." [C1]
>
> "There was some revision after each demo." [TM1A]
>
> "The customer was the one who showed the direction, I mean pretty much every week the direction was changed." [TM1D]
>
> "I think they were useful. You got to know the new direction, what this should become." [TM1F]

The short iterations gave both the team and the customer a possibility to learn what is important in the product. During the project the customer requested several new features based on what seemed to be the most important thing to do at a certain point of time.

In the second project, however, the plan was mostly created in the beginning and the rest of the time was used implementing it. The customer did not introduce any big features during the project, rather there were small adjustments to the details:

> "[The customer said that] there are some more things that you need to add in the admin panel or... I think not very big but small things that you need to add these as well." [TM2A]
>
> "Then we had such a starting implementation phase so that we get the basic things ready and I don't think we needed much feedback from the customer because it was rather clear what we are going to do. Then in some phase it went to the details, how this should exactly work." [TL2]
>
> "It [the customer demos] was pretty much the same as in the first project. First we present what we have accomplished during the previous week and then the customer can comment. And especially at the end... I mean because part of the software is in the customer's side, so how to get these two working together. That's what we mainly used time in the customer demos. Mutually asking question how this was done and how we should do it." [TL2]

Even though the customer demos in the second project were more fine-tuning than introducing new feature requests, the customer considered them valuable:

> "Then there was just the checkpoints [the customer demos] to see that they were actually doing what I asked them to do and that we both sort of agreed that what it was." [C2]

The theoretical framework suggests having end-of-iteration review session to enhance learning. The customer demos in the both projects were such. Another kind of end-of-review session were the retrospectives. They helped the both teams to improve their processes:

> “I think the retros were valuable events since we could in a way share feelings so that the others could reply. For example many times there was this issue that ’I have this problem’ and you had to say it over and over again that you just need to ask for help. And it got better, it clearly had an impact that at the end there was no more such that ’I get stuck to this’. Everyone knew what they had to do then.” [TM1C]
>
> “In the retros we noticed that we are not doing some things well enough. So let’s try to get better testing routine for everyone.” [TM1D]
>
> “In the retros the people said that the BDD session was very useful. Since everyone was not present, we will have another one for them.” [author notes, project 1]
>
> “People could tell their own viewpoints of what had happened. It opened my own eyes. When you thought these things that you are uncertain about, you could see them from the others’ perspective.” [TM2B]
>
> “For me they were the most interesting thing. Because in retros every time I used to find out new things that were wondering people’s minds but they never speak up. Initially no-one was happy with the Kanban [board], and that we got to know in the first retro. - - Then in retro people said that we should try these processes. Whenever we changed any processes or made any rules, those... The inputs came from the retro.” [TM2C]

The examples show that the retrospectives helped the teams to learn. However, not all retrospectives were equally useful. The author followed the second project retrospectives mainly as an external observer. One observation was that sometimes the team had difficulties to get to the root of a problem. Rather the problem was just recognized but the team did not always really tried to understand the reasons behind it or to find solutions for it. In the first project the challenge was to keep the retrospectives short enough, which some team members pointed up:

> “Especially when we were able to shorten them, we could get more out of them.” [TM1D]
>
> “They could have been shorter. Or the statements could have been shorter, at least some of them. When the retros were so soon after the demos, there was already a tingle to get back coding.” [TM1E]

Based on the empirical evidence the retrospectives are very useful for a team to improve its processes. However, there is a challenge to keep them short and interesting and still be able to go to the detailed level in discussion.

The fifth practice to enhance learning is to track the progress. This means that by measuring how fast the tasks are done, a team can learn. However, neither of the teams tracked their progress. In the first team the author as a team leader reasoned that by keeping the tasks very small there is no need for tracking. On the other hand, it might have revealed problems for example by showing that some team members are doing too big tasks. In the very beginning of the second project the team marked the beginning date of a task to the tickets in the Kanban board but quite soon stopped following the practice. Nevertheless, based on the empirical evidence it is quite difficult to argue whether tracking is important for learning and thus for self-organizing or not.

The empirical evidence supports all the five practices suggested by the theoretical framework, except track the progress, for which there is not enough empirical data. Continuous feedback is valuable for team members to learn but if it is avoided, people keep doing the same mistakes. Automated testing and continuous integration provide useful feedback for the developers, albeit there is no clear statistical evidence of the amount of defects in the projects. Short iterations are useful so that both the team and the customer can learn during the project. Finally, end-of-iteration review sessions are important at least in two ways. First, especially customer demos increase the team members' understanding of what is important in the project. Second, retrospectives are very useful for the team to improve its processes.

In addition, the empirical data shows that teams should invest in learning in the beginning. This is important since otherwise the team is forced to make compromises in later stages of the project. Also especially if there are inexperienced team members and they do not learn the critical tools and techniques, they are not able to contribute as much as wanted.

According to the theoretical framework there are four communication and collaboration practices that support learning and the other self-organizing team components as well. These will be explored next.

5.2.6 Communication and Collaboration

Besides autonomy, communication and collaboration consist the second basic building block in the theoretical framework. The four practices suggested by the framework are close contact with customer, work together in open workspace, share information daily, and visualize the progress. Communication and collaboration are rather a tool to support the other self-organization components than the goal itself (see Section 2.4.6). For this reason the impact of each practice to the other components is evaluated in this section. In other words, how either existence or absence of certain practice influenced to building self-organization in the teams.

In the both teams the customer demos were the main communication points with the customer. Otherwise there were relevant differences between the teams how they communicated with the customer.

The first team seemed to have a good and open relationship with its customers. However, some team members felt that they would have needed more customer feedback:

> “The customers were active when they were present and they had a lot of good ideas. And they acted typically like a good, active customer acts. But I would have hoped that the customer would have been more present and available.” [TM1C]
>
> “I think at some stage when neither [of the customers] was present, then we had something. There was this period when we had to a bit longer do like ’I guess it goes like this’. - - The customer doesn’t have to be there all the time but somewhere comes a limit that when it’s not enough there, you have to start guessing.” [TM1F]
>
> “At latest now you can state that this cycle [one week iteration and demo] works well. The only thing that could be improved is if we could discuss with the customer more often than once a week. Especially this week when we had to do such a prototyping that definitely demands for customer feedback. Luckily the customer was here on Tuesday so we got a confirmation for our view and were able to proceed.” [author notes, project 1]

Also the customer recognized the importance of presence afterwards. However, he notified that being present is not always possible in practice and there is actually a good reason of not being with the team constantly:

> “When in the last week I was all the time present and visited there many times a day, many things that would have waited still a few days for the traditional customer demo, were solved there.” [C1]
>
> “Probably some things we could have done faster [if the customer would have been present more often] or if we could have participated in the smart board conversation. On the other hand, it is quite unrealistic arrangement. The customer presence, it has to be real and stable. But even the customer can’t always analyze the answer right away. - - So it is perhaps better to let things sit for a while.” [C1]

Nevertheless, as the first team did not have as close contact with the customer as wanted, the team was not able to advance and learn as quickly as possible. The second team had similar problems but the starting point was a bit different. The customer explained his idea about the communication:

> “The initial communication was as efficient as possible communicating the problem and then figure out quickly whether people were actually getting into their

> idea and talk. - - And then after that I think it was very simple: I'm available anytime, I'll come whenever you want me to come or you can call me or you can Skype or send emails or anything like that if you have a problem." [C2]

The team however did not use the possibility offered by the customer. During the first half of the project there was very little other communication than the customer demos. In the third retrospective the author asked about this:

> "The third thing [that the author wanted to discuss more] were unclear requirements. I asked how often the team contacts with the customer. It was found out that during the last two weeks there has been two demos and one email that was followed by a Skype conversation." [author notes, project 2]

Perhaps in the retrospective the team understood the problem they had since at the end of the project there was much more communication with the customer:

> "He [the customer] said that he is always available through Skype or emails but we didn't contact him much initially. But at later stages whenever we felt like asking him, we used to ask him. - - The interaction increased with him. We used to send him mails every two days or in the last week probably every day." [TM2C]

> "Especially at the end we sent quite many emails and there were a few Skype conversations. Probably in the last week, I don't know how many emails." [TL2]

There was also another problem that the second team had in the relationship with the customer. The relationship was somewhat tensed. This affected to learning since for example when a team did not understand something that the customer explained, the team could not admit it:

> "Perhaps we didn't manage to say that it is not clear. We just said together with the other team members that 'yeah, yeah'. It probably slowed down. And then we perhaps couldn't also ask from the customer. That here we have an unclear thing. - - And then we just implemented it somehow and then demoed. And then it wasn't like the customer would have wanted." [TM2B]

> "I guess there's a bit difference in the position [between the team and the customer]. And it's probably also that you don't dare to say that 'I did not understand'. Well, it makes it a bit tensed. I mean tensed so that you are a bit uncertain. Not tensed like in a sense of conflict." [TM2B]

The examples from the both teams show that if a team is not able to have a close contact with the customer, it cannot learn as fast as possible. The empirical data thus supports the theoretical framework.

The second communication and collaboration practice suggested by the framework is to work together in an open workspace. This was done by the both teams since they had one room where all the team members were located. Colocation had a clear positive effect, for example when help was needed:

> "Well if I get stuck, I start like taking some help from internet or from anyone near me. And if I see that [two team members] or someone is not capable of that, I call you [the author] or [TM1C] who really can solve that problem and mostly they solved it." [TM1A]
>
> "Usually it [problem solving] went so that I asked if someone knows something." [TM1F]
>
> "And whenever we tried to implement something and if we are stuck at some point, we always used to get help from others. And we can just shout anyone's name that ok, I have a problem. And then the person used to come to my desk and ok, let's try to do it this way." [TM2C]
>
> "[TM2B] was sitting next to me, so usually there were some problems and we discussed together. That way I could solve the problems." [TL2]

Colocation also increased collaboration. There was a smart board in the room and the first team used it many times. One team member commented how this had an effect on collaboration:

> "It [use of the smart board] worked very well especially collaboratively. Someone could for example speak that what is the meaning of it [a drawing] while some other was drawing a feature to it." [TM1C]

Colocation makes it easier to share information daily, which is the third practice suggested by the theoretical framework. The purpose of the practice is that when information is constantly shared, the team can learn, redundancy evolves, and the team orientation gets better. The empirical data supports all of these. In Section 5.2.4 it was described how the daily meetings increased redundancy. The daily meetings also enhanced learning since the team members were able to get help and understand the project status better, as well as they reminded the team members about the team goals:

> "We were in the daily scrum, meeting we had in our team. We had to summarize. - - And it helped me to know what the other people are doing. So again in that way it helped me to take decisions in that team because - - I had to

> see the team environment, I had to customize the decision according it. - - I should take a proper decision. Not only from my point of view." [TM1B]
>
> "And I think there were many times [in the daily meetings] that someone said: 'I am going to do this'. And then someone else said: 'Would you rather do this since it is more immediate or important?"' [TM1C]
>
> "It was perhaps such a cycle that after the demo, possibly after the retro, we set the goals for the next demo. And in the daily meetings we looked them more detailed." [TM2B]
>
> "That [the daily meetings] was the place that we come to know that who is working on what thing. We used to know that this person is doing this thing. And this person is free or is planning to take up a new task. So that way we were not working on isolation that we don't know who is changing what and then we have conflicts suddenly. - - That never happened with us." [TM2C]

However, the daily meetings were not always completely successful. Some of the interviewees complained:

> "They could have been a bit faster as well [like the retrospectives]. My concentration is so short that if they would have lasted five or ten minutes, then I would have perhaps gotten a better overall picture but when everyone is talking too long, you forget what the previous was saying." [TM1E]
>
> "I think it was a little. . . maybe we concentrated too much on individual work. It would have been better to discuss the team issues together." [TM2B]
>
> "They were sometimes that we just go around. - - Everyone is probably not paying attention and it's just a routine that has to be done. But sometimes, especially now when I was in a leading position, it was useful to listen what was the situation. Although I quite well knew what it was." [TL2]

The first team used a same format in the daily meetings during the whole project. The second team, on the other hand, discussed in the first retrospective that everyone is not able to follow what is happening in the team. For that reason the team started to held the daily meetings in front of the Kanban board. The idea was that when the team members were telling about their tasks, they point the ticket on the board and that way make it more explicit.

> "But then some people said that we cannot understand what others are doing. We are not aware of the fact. So we changed the format [of the daily meetings], we used the explain our tickets on the Kanban board. That on this ticket we are working on. So then it was more clearer." [TM2C]

> “At some stage we moved, so that we do not stay on our own places but go in front of the Kanban board. At least in the beginning I would remember that we had good conversations.” [TL2]

The theory supports the share information daily practice. However, it also shows that it matters how for example the daily meetings are held. Nevertheless, when the team is made to discuss its progress every day, it increases the level of self-organization.

The fourth communication and collaboration practice in the theoretical framework is to visualize the progress. The both teams had a Kanban board that contained the ongoing and finished tasks. Neither of the teams however used burndown charts.

In the interviews of the first team members the Kanban board was not much discussed. Nevertheless, the author felt that as a team leader the board visualized very well the current status of the project. It was for example easy to see who needs code review and who does not have any task at certain moment. Seeing an overall task status was very useful especially before the customer demos. As the project direction was always somewhat changed in the demos (see the previous section), it was important to get the Kanban board cleared before a demo. That gave the team better possibilities to react to the changing needs of the customer:

> “One important thing came into my mind. It is very essential to keep the tasks small and get them moving quickly through the Kanban board. If for example now [right after a demo] the developers had ongoing tasks that still require a few days work, it would be more difficult to change the direction.” [author notes, project 1]

In the first project the author had understood the importance of fast feedback of code reviews. In the second project the code reviews were however done rather late (see the previous section). This was revealed by the Kanban board where the tasks stayed quite long time in the code review column. The initial work in process limit for code reviews was four, and the author recommended that the team should lower it. When they dropped the limit to two, it forced the team to review code faster:

> “I asked from the team how long the tickets stay in the code review column. They answered that rather long, as I suspected. I suggested dropping the WIP limit to two, which would force the team to act faster.” [author notes, project 2]
>
> “First of all we made a big decision on the Kanban board. Because we people are working on the task and then we are just putting the task for code review. - - And every task, it is going to code review and it is not going to pass further. Then we decided the limitation on the Kanban board, only three or four [two

> actually] code review should be here and this way the flow of the Kanban board will go smoothly. And it was a good decision.” [TM2A]
>
> “For me the most important thing in this Kanban was that we get, and you know code review, that we get the process moving. That we get the tickets all the way to the done.” [TL2]

The both teams also had nine screens on the wall where they could for example show the status of the continuous integration, which was useful for the both teams. One screen in the second project was showing the number of commits (see Figure 10 in Section 5.1.3). For the team it was motivating to see the progress they were making, especially after they had had less progress during the fifth week.

The four practices suggested by the theoretical framework are supported by the empirical data. A team should have a close contact with the customer since otherwise there are problems in learning. Working together in an open workspace clearly makes the team to communicate better, which has a positive effect for example on how team members help each others. Sharing information daily enhances learning, redundancy, and team orientation. Visualization of the progress can reveal problems in the process and also motivate the team members.

5.3 Summary of Theoretical Framework Practices in Projects

Table 8 summarizes how different practices suggested by the theoretical framework were used in the projects and what kind of consequences there were. The table contains also three new practices that will be added to the framework based on the empirical evidence: invest in learning, keep tasks small, and prioritize clearly.

Component	Practice	Project 1	Project 2
Autonomy	Authorize the team	The team had strong external autonomy, which gave it a good possibility to improve its processes and take the necessary actions to satisfy the customer needs.	The team had strong external autonomy, which gave it a good possibility to improve its processes and take the necessary actions to satisfy the customer needs.
	Have someone to protect team	Not basically needed. The team protected its autonomy in the customer demos when the customer was about to go to the implementation level in the tasks.	Not needed
Communication & collaboration	Close contact with customer	Open and honest relationship but the team would have needed more mutual time. Lack of it affected to learning.	Somewhat tensed relationship with the customer; the team could not admit the lack of understanding. The customer was available but in the beginning the team did not use the possibility to interact. Both affected to learning.
	Work together in open workspace	Colocation in the same room, which had a positive effect e.g. on getting help from the team members.	Colocation in the same room, which had a positive effect e.g. on getting help from the team members.

	Share information daily	The team had useful daily meetings increasing team orientation, redundancy, and learning. However, some problems in effectiveness, which the team was not able to fix.	The team had useful daily meetings increasing team orientation, redundancy, and learning. Changing the format of the daily meetings made the team to communicate better.
	Visualize progress	The Kanban board showed beneficial information of the progress helping the team to learn and improve its processes. The team had useful continuous integration screens.	The Kanban board showed beneficial information of the progress but the team did not see it without the help of coach. The team had useful continuous integration screens and a motivating commit progress screen.
Learning	Continuous feedback	The code review stage in the process was very beneficial source of feedback.	The team had problems with code reviews: they were not done as soon as possible, the quality of code reviews prevented some team members from learning, and there were not many team members capable of doing code review.
	Automated testing and continuous integration	The good code coverage and continuous integration provided useful feedback for the developers.	The tests and continuous integration were useful but the team started writing unit and integration tests rather late.
	Short iterations	The team had one week iteration that ended to a customer demo. The short iteration gave the team and the customer an important possibility to learn during the project.	The team had one week iteration that ended to a customer demo. The short iteration was a tool for a customer to make sure that the progress goes to the right direction.
	End-of-iteration review sessions	The customer demos were an important communication point with the customer and very important for adjusting the project goals. The retrospectives helped the team to improve its processes but there was a challenge to keep them short enough.	The customer demos were an important communication point with the customer. The retrospectives helped the team to improve its processes but there was a challenge to go to the root of the problems and find solutions for them.
	Track progress	Progress was not tracked anyhow. The team wanted to keep the tasks small, which reduced the need for tracking. Not enough empirical evidence available for or against.	Progress was not tracked anyhow. Not enough empirical evidence available for or against.
	Invest in learning (new)	The team studied the technology in the beginning and arranged later TDD and BDD learning sessions. This gave all the team members better possibilities to contribute at the end of the project.	The team studied the technology in the beginning but did not support the inexperienced team members enough so that they could contribute for the team. Since for example TDD was not studied properly and early enough, the team had to make compromises.
Redundancy	Share responsibility of work	The team members were capable of doing others' tasks, many people did code reviews, and there was not much individualism in task selection.	Only certain key members were capable of doing core tasks, few people were burdened with code reviews, and the team members specialized in certain tasks.
	Agree on uniformity	The team agreed on coding conventions and writing code in “Rails way”, and the rules were followed quite well. However, no strong evidence how important this was for redundancy.	The team agreed on coding conventions and writing code in “Rails way” but did not follow the rules very well. However, no strong evidence how important this was for redundancy.
	Keep tasks small (new)	The team aimed at keeping the tasks in a small level, which prevented specialism. This was one reason why absence of people did not cause problems.	No clear evidence how small the tasks were and how this affected redundancy.

Team orientation	Let team to participate in iteration planning and goal setting	The customer demos were the most important iteration planning sessions and almost the whole team participated in them every time. This increased the team members' awareness of the most important tasks.	The whole team spent plenty of time in the beginning to design the first version of the product. This helped the team members to understand what needs to be done. The customer demos had a similar effect although there was a case when no-one did the most important task of the week.
	Guide with mission	The team was given high level goals by the customer and was able to implement the tasks in a way that the team saw was the best.	The team was given an overall idea of the product in the beginning by the customer and was able to design the implementation as they saw was the best.
	Prioritize clearly (new)	Based on the "what are the most important features next week?" question the team members had a good understanding of iteration goals. As a motivational factor individuals were allowed to choose tasks themselves but they also followed the priorities.	The team members had quite a good understanding about the most important tasks. As a motivational factor individuals were allowed to choose tasks themselves but they also followed the priorities. However, once no-one did the most important task of the week.
Shared leadership	Manage with lead-and-collaborate principle	The team roles were shared based on individuals' knowledge and skills. The people with the best knowledge made the decisions instead of the team leader making them. The whole team was able to participate in decision making but the inexperienced team members could not contribute.	The team roles were shared based on individuals' knowledge and skills. The people with the best knowledge made the decisions instead of the team leader making them. The whole team was able to participate in decision making but the inexperienced team members could not contribute.
	Cross-functional teams with wide knowledge	Shared leadership did not fully evolve since only certain people had knowledge that could be used in making the major decisions.	Shared leadership did not fully evolve since only certain people had knowledge that could be used in making the major decisions.
Training	Agile knowledge in team or by coach	Some key team members had understanding of agile practices. As the practices were followed, the team was able to be quite self-organizing.	The team itself was not very experienced on agile practices but the certain members' experiences from the first project and the author as a coach helped the team.

Table 8: Summary of how theoretical framework practices were followed in projects.

The last row in Table 8 describes how agile knowledge was present in the teams. There was no own subsection above for it but there were some examples how in the second team the author as a coach helped the team. For example the author suggested changes to the work processes of the second team to fasten the feedback loop and improve learning. Similarly, the retrospectives were very useful for the team but the team was willing to have such only after the author explained the importance of them.

5.4 Primary Empirical Conclusions

Based on Sections 5.2 and 5.3 the primary empirical conclusions (PEC) can be drawn. These will be refered to in the following section when the empirical results are compared with the existing literature and when the managerial implications are presented. Totally fourteen PECs can be found.

PEC 1: Strong external autonomy enables building self-organization but does not automatically mean that the team will use all the autonomy. If the autonomy is used, it relieves the management and the customer from thinking all the details.

PEC 2: Individual autonomy can be increased by giving team members a possibility to choose the tasks they do. When the tasks are prioritized clearly and the team members are encouraged in considering the priorization while choosing the tasks, it is possible to combine individual autonomy and team orientation.

PEC 3: Customer demos are useful checkpoints for the customers but they also increase the team's understanding of what is important.

PEC 4: Major decisions in a team can be made with shared leadership; vertical leadership is not required in it. Team members' experience and knowledge are nevertheless important since otherwise it is difficult for a team member to be part of shared decision making.

PEC 5: Like constantly sharing information increases redundancy, in a same way sharing tasks and switching responsibilities are important so that specialism and individualism do not evolve.

PEC 6: Getting continuous feedback is important for learning. A good method for developers is code review. The importance is highlighted when the developers are inexperienced.

PEC 7: A team should invest in learning in the beginning of the project, especially if it is not familiar with the technology. Otherwise the team is forced to make compromises for example in quality, and the inexperienced developers will not be able to contribute as much as wanted.

PEC 8: Short iterations are useful for learning, both from the team's and the customer's point of view. They also enable changing direction fast but this requires that the tasks are small so that new tasks can be started right away.

PEC 9: Retrospectives are useful events for a team to learn and improve its processes but for enhancing shared leadership as well. It is important to keep the retrospectives interesting and short enough but still to be able to get to the roots of the problems.

PEC 10: Close contact and open relationship with the customer is important. If the customer is not available, the team cannot get the feedback needed and is not able to support learning. If the team and the customer are not able to discuss openly, it deteriotates the team performance as well.

PEC 11: Daily information sharing is very important since it supports communication, team orientation, redundancy, and learning. On the other hand, the team needs to pay attention how the time for example in daily meetings is used.

PEC 12: Visualization of the team progress is useful because it enables the team to find problems in its processes.

PEC 13: Agile software development methods provide with many practices that are useful for building self-organization. For this reason it is important to have someone in the team or for example a coach that understands the purpose of different practices and makes sure that they are properly followed.

PEC 14: Self-organization can be built in a very short period of time.

6 Discussion

This section discusses the results of the thesis. The discussion concentrates mainly on the primary empirical conclusions (PEC) listed in Section 5.4, and what kind of theoretical implications (Section 6.1) and managerial implications (Section 6.2) they have.

6.1 Theoretical Implications

The theoretical framework leans on the five characteristics of a self-organizing team suggested by Moe et al. [MDR09]. In addition, communication and collaboration are included in the framework as they have an important effect on the other components. Based on the empirical evidence it seems that these six characteristics provide a good framework to understand how self-organizing teams can be built.

As novel information the thesis proposes a model that combines the characteristics of a self-organizing team and the practices of agile software development methods. The empirical data gives initial support for the model but there is need to test the model with additional empirical data.

Previous literature says that team members should emphasize the team goals over their own goals [Jan98, Whi08, MoA08]. However, this may be in contradiction with the individual autonomy, which is important for an individual's motivation [MDD08, Bar09]. PEC 2 provides with one solution for this problem. Individual autonomy and team orientation can be combined if the priorization is clear and the team members have a possibility to choose the task themselves.

One empirical finding related to the task selection is that not everyone in the team should necessarily do the most important tasks. If the team members are very inexperienced, it is better to choose tasks that are challenging enough but not too difficult. This supports the idea of equilibrium between the team's competence and the tasks given for it proposed by Jia'an [Jia08].

PEC 4 emphasizes the importance of shared leadership over vertical leadership. According to Pearce vertical leadership is needed at least for clarifying task specifications, securing necessary resources, selecting team members, and identifying team member roles [Pea04]. From these the task specification and identification of team member roles were relevant in the case studies. However, the empirical evidence is in contradiction with what Pearce suggests since vertical leadership was not required in either of these.

Constantly sharing information and switching responsibilities in tasks in order to prevent specialism and individualism is mentioned in PEC 5. All of these conform to the previous studies [MoA08, MDD08, MDD09]. Same way PEC 6 and the meaning of continuous feedback in software development supports the existing literature [PoP03, pp. 22-27].

Short iterations were very useful in the case studies (PEC 8). All the agile software development methods encourage using them (see Section 3.1) but usually they suggest longer than one week iterations. The case studies however show that one week can be a proper period length.

The usefulness of retrospectives is recognized in PEC 9 and in the existing literature as well [Lam05; Kni07, p. 67-71]. The empirical evidence emphasizes the Lamoreux's argument that it is not meaningless how retrospectives are held [Lam05]. First, the team should try to keep them interesting. Secondly, the team should really try to get to the roots of the problems instead of just mentioning them.

The theoretical framework suggest that a team should have a close contact with its customer. PEC 10 supports this and that way conforms to the agile software development methods like XP [Bec99, pp. 52-53]. However, PEC 10 also states that the relationship between the team and the customer should be open, as in the second case study there were problems with this. Unfortunately, the case study did not reveal how the relationship would have changed if the customer would have been present all the time, as XP suggests.

PEC 11 states that daily information sharing is important, which conforms to the purpose of daily meetings suggested by XP and Scrum [Bec99, p. 102; Lef07, p. 45]. As new information it however shows what kind of effects they have for building self-organization.

The usefulness of visualization comes out in PEC 12. The empirical data shows how Kanban board helped the teams to find problems in their processes and how varying work in process limits helped the second team to modify their process. This supports the existing evidence that by visualizing the progress a team is able to identify problems in their process [MDK09]. The empirical evidence is nevertheless relevant information since there is not much research done related to usage of Kanban in software development.

PEC 13 concludes that agile software development methods provide with many practices that are useful for building self-organization. This is an important finding since typically the methods like Scrum ask for self-organizing teams, but there are no clear instructions how self-organization can be actually built [Lef07, p. 43]. It seems that when the best practices from different agile methods are combined, self-organization can evolve "automatically".

The last primary empirical conclusion (PEC 14) says that self-organizing teams can be built in a very short period of time. The evidence is based on the short projects (seven weeks), in which rather high level of self-organization was reached and in which the customers were also satisfied with the results (see Section 5.1.4). The evidence contradicts with the existing literature that claims that transformation of a work group into a self-organizing work team takes years [Jia08].

This thesis has concentrated on a single team level but the empirical evidence also reveals a relevant result from an organizational level. Both projects studied were under the same organization, and the second project got evident advantages from having team members who participated in the first project as well. This supports the argument of Takeuchi and Nonaka who suggest that for an organization it is important to transfer knowledge and best practices from one project to another [TaN86].

As final theoretical implication the empirical data shows that self-organizing teams can produce good results. The empirical study did not contain comparison groups so it is difficult to say how effective the teams were compared to the traditional working groups. Nevertheless, the customer testimonials in Section 5.1.4 indicate good customer satisfactory, which conforms to the results of many previous studies [BHS92, Win94, Jan98, Jia08].

6.2 Managerial Implications

Based on the primary empirical conclusions it is possible to suggest relevant managerial implications as well. At first, providing a team with true external autonomy gives it a possibility to become self-organizing (PEC 1). If the team uses the autonomy, it can save the management's time as the details of work are left for the team.

PEC 2 states that individual motivation can be increased by giving individuals a possibility to select tasks on their own. This however requires that it is clear for the team what are the priorities of the tasks. One way to increase the awareness of priorities is to have end-of-iteration review sessions like customer demos that at the same time can be useful checkpoints for the management as well (PEC 3).

Managerial implication of PEC 4 is that there is no need to emphasize vertical leadership in teams. Rather the management should encourage the teams towards shared leadership. In a same way there is no need for the management to define the roles of the team members; it is better to let the roles evolve.

PECs 5 and 6 highlight the importance of communication and collaboration inside of the team. Collaboration sometimes means overlapping work but the management should recognize how important it is for building self-organization. In a same way retrospectives take time away from the daily work but they are very valuable events for the team, and the management should recognize this as well (PEC 9).

According to PEC 10 a software development team should have a close contact with its customer. The management's task is to make sure that the team is able to have such a relationship. Likewise, the team requires understanding of agile software development methods (PEC 13), and the management should make sure that such understanding is available for the team or in the team.

7 Conclusions

This thesis has examined how self-organization can be built in software development teams. Section 2 first described the six characteristics that define the performance of a self-organizing team and introduced an initial theoretical framework. Section 3 discussed how agile software development methods support self-organization and presented the final theoretical framework. The study context, research methodology, and data collection methods were explained in Section 4 and the empirical results described in Section 5. Section 6 discussed the empirical results and identified theoretical and managerial implications.

This section concludes the thesis. First, Section 7.1 provides with an answer to the reseach question. Section 7.2 discusses the limitations of the thesis. Finally, Section 7.3 suggests what further study is still needed.

7.1 Answer to Research Question

The research question of the thesis is: How to build a self-organizing software development team? In order to answer to the question, a theoretical framework was first created and then analyzed against empirical data. This section provides with an answer to the research question.

The empirical evidence shows that the most of the practices of the theoretical framework are useful for building self-organization in teams. It also shows that if some of the practices are not followed, it harms the building of self-organization. For some practices there was not enough empirical data available, but on the other hand there is no evidence either that the practices should be removed from the framework.

Based on the empirical evidence there is a need to add three more practices to the theoretical framework. First, it is important to invest in learning since otherwise the team is forced to make compromises in the project and possibly not all the team members are able to contribute for the team. Secondly, keeping tasks small increases redundancy since then the lost work is small if a team member has to be suddenly away. Thirdly, when the tasks are clearly prioritized, individual team members know what tasks are the most important and can take this into account while choosing tasks. As they can choose the tasks themselves, it increases their motivation as well.

Additionally the close contact with customer practice is rephrased to close and open relationship with the customer. This is based on the empirical data that showed problems in the communication between the team and the customer even though the customer was available for the team practically all the time.

Figure 12 shows the final model, which at the same time is the answer to the research question. It should be interpreted so that at first, there are certain characteristics that are

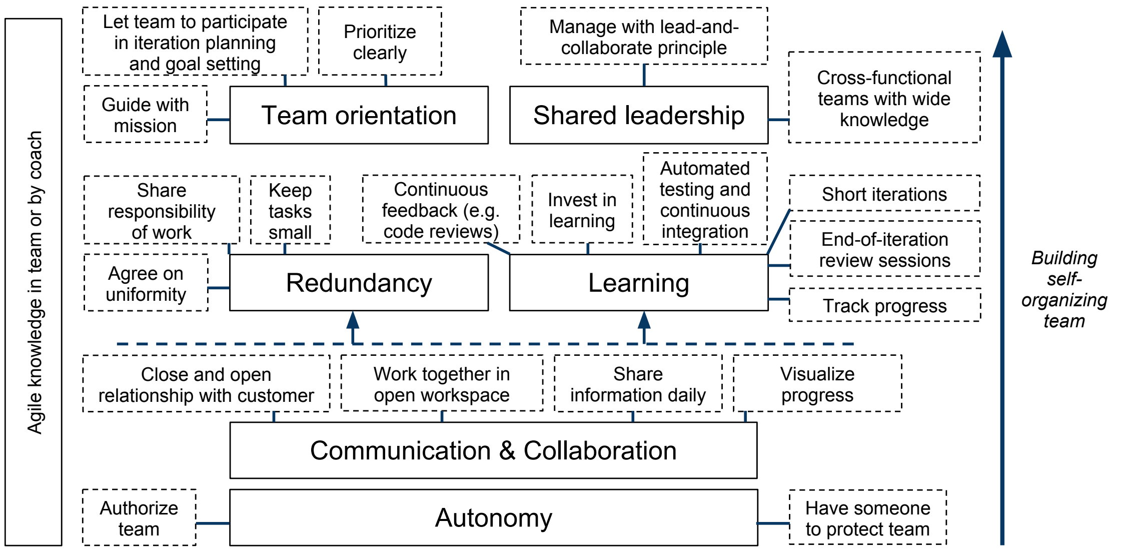
Agile knowledge in team or by coach
Let team to participate in iteration planning and goal setting
Prioritize clearly
Manage with lead-and-collaborate principle
Guide with mission
Team orientation
Shared leadership
Cross-functional teams with wide knowledge
Share responsibility of work
Keep tasks small
Continuous feedback (e.g. code reviews)
Invest in learning
Automated testing and continuous integration
Short iterations
End-of-iteration review sessions
Agree on uniformity
Redundancy
Learning
Track progress
Close and open relationship with customer
Work together in open workspace
Share information daily
Visualize progress
Communication & Collaboration
Authorize team
Autonomy
Have someone to protect team
Building self-organizing team

Figure 12: Answer to research question: How to build a self-organizing software development team?

needed in order a team to become successfully self-organizing. The primary characteristics are autonomy, and communication and collaboration. The additional ones are redundancy, learning, team orientation, and shared leadership.

Secondly, the figure presents practices, with which the characteristics can be supported in a software development team. Assuming that the team follows the practices, it can build self-organization. This can happen even in a short period of time, as the empirical data shows.

The practices related to autonomy are authorize team and have someone to protect team. The communication and collaboration practices are close and open relationship with customer, work together in open workspace, share information daily, and visualize progress. Learning can be supported with invest in learning, continuous feedback, automated testing and continuous integration, short iterations, end-of-iteration review sessions, and track progress practices.

The redundancy practices are share responsibility of work, agree on uniformity, and keep tasks small. The three team orientation practices are let team to participate in iteration planning and goal setting, guide with mission, and prioritize clearly. Finally, the shared leadership practices are manage with lead-and-collaborate principle, and cross-functional teams with wide knowledge. In addition a team needs agile knowledge that can be in the team or provided for example by a coach.

7.2 Limitations

There are certain limitations that may threaten the validity of the research made in this thesis. At first, as the author participated in the first case study as a team member and in the second as a coach, there is a risk that the interpretations made are biased. The risk was however reduced by using multiple sources of evidence as participant observation was supported with interviews and notes written by the author, one customer, and one team member. On the other hand, the participation gave the author such insight to the projects that might have not been possible with direct observation.

Another limitation is that the empirical phase of the thesis contained only two case studies. This means that the model proposed by the thesis should be tested with further research. However, the model suggested is novel information and the empirical study was an important initial test for it.

A third limitation is that the case studies were made with students, of which some were rather inexperienced. On the other hand, not all of them were completely lack of work experience in software development. In addition, using students is quite typical in software development research.

Finally, the empirical study was made in a somewhat artificial environment instead of a real workplace. However, the work in the environment was considered very close to real work life by the participants. Moreover, the work was not regulated by the external parties anyhow. Besides the author's participant observation there was only direct observation, which did not affect to the team members.

7.3 Further Study Suggested

There are still issues that require further study. First, the proposed model should be validated with additional research. It would be especially interesting to see how well it works with more experienced software engineers. The case studies of the thesis were done with students, of which some did not have previous work experience at all.

In this study the both teams had a good level of external autonomy. The model suggests that there should be someone protecting the team but in the case studies it was not necessary. A study made in real work life would perhaps better test this part of the model.

Furthermore, the model suggests colocation as one practice to increase self-organization, and close communication and collaboration are seen generally important as well. For that reason it would be important to see how the model needs to be changed if the teams are geographically distributed and even in different time zones.

The case studies lasted only seven weeks. Altough the empirical evidence reveals that self-organization can be built in such a short time period, it would be valuable to see how self-organization evolves in longer projects.

Finally, the focus of the thesis has been in software development. However, the main components of the model and many of the practices in it are not restricted to the software development industry only. For that reason it would be very interesting to see how the model could be adjusted for the needs of other industries using team work.

References

Abr10	Abrahamsson, P., Unique infrastructure investment: Introducing the software factory concept. *Software Factory Magazine*, 1, pages 2–3.
Agi01	Agile manifesto, 2001. URL `http://www.agilemanifesto.org`.
Ala95	Alasuutari, P., *Laadullinen tutkimus*. Vastapaino, Tampere, 1995.
Avi99	Avison, D., Lau, F., Myers, M. and Nielsen, P. A., Action Research. *Communications of the ACM*, 42, pages 94–97.
Amb02	Ambler, S., Lessons in agility from Internet-based development. *IEEE Software*, 19,2(2002), pages 66–73.
Aug05	Augustine, S., Payne, B., Sencindiver, F. and Woodcock, S., Agile project management: steering from the edges. *Communications of the ACM*, 48,12(2005), page 89.
Abr02	Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J., *Agile software development methods*. VTT, 2002.
Bar09	Barney, H. T. e. a., Balancing individual and collaborative work in agile teams. *Agile Processes in Software Engineering and Extreme Programming*, Pula, Sardinia, Italy, 2009.
Bec99	Beck, K., *Extreme Programming Explained: Embrace Change*. 1999.
BHS92	Behnke, L., Hamlin, R. and Smoak, B., The evolution of employee empowerment. *Semiconductor Manufacturing, IEEE Transactions on*, 6,2(1993), pages 143 –155.
Blo03	Blotner, J., It's more than just toys and food: leading agile development in an enterprise-class start-up. *Agile Development Conference, 2003. ADC 2003. Proceedings of the*, 25-28 2003, pages 81 – 91.
Boe02	Boehm, B., Get ready for agile methods, with care. *Computer*, 35,1(2002), pages 64 –69.
CoH01	Cockburn, A. and Highsmith, J., Agile Software Development, the People Factor. *Computer*, 34,11(2001), page 131–133.
CoB97	Cohen, Susan; Bailey, D., What Makes Teams Work: Group Effectiveness Research from the Shop Floor to the Executive Suite. *Jounal of Management*, 23,3(1997), pages 239–290.

DyD08 Dybå, T. and Dingsøyr, T., Empirical Studies of Agile Software Development: A Systematic Review. *Information and Software Technology*, 50,9-10(2008), pages 833–859.

FDD10 The latest FDD processes, 2010. URL `http://www.nebulon.com/articles/fdd/download/fddprocessesA4.pdf`.

Fen98 Fenton-O'Creevy, M., Employee involvement and the middle manager: evidence from a survey of organizations. *Journal of Organizational Behavior*, 19,1(1998), pages 67–84.

GuD96 Guzzo, R. and Dickson, M., Teams in organizations: Recent research on performance and effectiveness. *Annual review of psychology*, 47,1(1996), page 307–338.

GaD05 Gagne, M. and Deci, E. L., Self-determination theory and work motivation. *Journal of Organizational Behavior*, 26,4(2005), pages 331–362.

GSK05 Gemünden, H. G., Salomo, S. and Krieger, A., The influence of project autonomy on project success. *International Journal of Project Management*, 23,5(2005), pages 366 – 373. Selected papers from the Sixth Biennial Conference of the International Research Network for Organizing by Projects.

HiH00 Hirsjärvi, S. and Hurme, H., *Tutkimushaastattelu. Teemahaastattelun teoria ja käytäntö*. Yliopistopaino, Helsinki, 2000.

Hig00 Highsmith, J., Retiring Lifecycle Dinosaurs – Using Adaptive Software Development to meet the challenges of a highspeed, high-change environment. *Software Testing and Quality Engineering*, May/June, page 22–28.

HiM93 Hirschheim, R. and Miller, J., Implementing empowerment through teams: the case of Texaco's information technology division. *SIGCPR '93: Proceedings of the 1993 conference on Computer personnel research*, New York, NY, USA, 1993, ACM, pages 255–264.

HoP06 Hoegl, M. and Parboteeah, P., Autonomy and teamwork in innovative projects. *Human Resource Management*, 45,1(2006), page 67–79.

HRS00 Hirsjärvi Sirkka, R. P. and Paula, S., *Tutki ja kirjoita.* Kustannusosakeyhtiö Tammi, Helsinki, 2000.

HiT00 Hines, P. and Taylor, D., *Going lean.* Lean Enterprise Research Centre, Cardiff Business School, Cardiff, 2000.

Jan98 Janz, B. D., The best and worst of teams: self-directed work teams as an information systems development workforce strategy. *SIGCPR '98: Proceedings of the 1998 ACM SIGCPR conference on Computer personnel research*, New York, NY, USA, 1998, ACM, pages 59–67.

JCN97 Janz, B. D., Colquitt, J. A. and Noe, R. A., Knowledge Worker Team Effectiveness: the Role of Autonomy, Interdependence, Team Development, and Contextual Support Variables. *Personnel Psychology*, 50,4(1997), pages 877–904.

Jia08 Jian, C., Research on Strategies and Empowerment Process to Achieve Self-management Team. *Area*, pages 25–29.

Kar10 Karhatsu, H., Guide for Software Factory Teams, 2010. URL `http://www.softwarefactory.cc/guide`.

KLJ89 Kleeper, R., Litecky, C. and Jones, T. W., Self managed teams and MIS productivity: literature, model and field study. *SIGMIS Database*, 20,1(1989), pages 36–38.

Kni07 Kniberg, H., *Scrum and XP from the Trenches.* C4Media, 2007.

KaS93 Katzenbach, J. R. and Smith, D. K., The discipline of teams. *Harvard Business Review*, 71,2(1993), pages 111–20.

KnS09 Kniberg, H. and Skarin, M., *Kanban and Scrum - making most of both.* InfoQ, 2009.

Lam05 Lamoreux, M., Improving agile team learning by improving team reflections. *ADC '05: Proceedings of the Agile Development Conference*, Washington, DC, USA, 2005, IEEE Computer Society, pages 139–144.

Lan00 Langfred, C. W., The paradox of self-management: individual and group autonomy in work groups. *Journal of Organizational Behavior*, 21,5(2000), pages 563–585.

LaC05 Law, A. and Charron, R., Effects of agile practices on social factors. *ACM SIGSOFT Software Engineering Notes*, 30,4(2005), page 1.

Lef07 Leffingwell, D., *Scaling Software Agility: Best Practices for Large Enterprises.* Addison-Wesley Professional, 2007.

Lep93 Leppitt, N., The path to successful empowerment. *Engineering Management Journal*, 3,6(1993), page 271–277.

MoA08 Moe, N. B. and Aurum, A., Understanding Decision-Making in Agile Software Development: A Case-study. *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 216–223.

McC01 McCauley, R., Agile development methods poised to upset status quo. *ACM SIGCSE Bulletin*, 33,4(2001), page 14.

McC03 McCarthy, C., Participative leadership in team formulation. *Engineering Management Conference, 2003. IEMC '03. Managing Technologically Driven Organizations: The Human Side of Innovation and Change*, 2-4 2003, pages 566 – 570.

MDD08 Moe, N., Dingsøyr, T. and Dybå, T., Understanding self-organizing teams in agile software development. *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, 26-28 2008, pages 76 –85.

MDD09 Moe, N., Dingsøyr, T. and Dybå, T., Overcoming barriers to self-management in software teams. *IEEE Software*, 26,6(2009), page 20–26.

MDK09 Moe, N., Dingsøyr, T. and Kvangardsnes, O., Understanding shared leadership in agile development: A case study. *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*, 5-8 2009, pages 1 –10.

MDR09 Moe, N. B., Dingsøyr, T. and Røyrvik, E. A., Putting agile teamwork to the test — an preliminary instrument for empirically assessing and improving agile software development. *Agile Processes in Software Engineering and Extreme Programming*, Pula, Sardinia, Italy, 2009.

Mid01 Middleton, P., Lean software development: two case studies. *Software Quality Journal*, 9,4(2001), page 241–252.

NeB07 Nerur, S. and Balijepally, V., Theoretical reflections on agile development methodologies. *Communications of the ACM*, 50,3(2007), page 83.

Pea04 Pearce, C., The future of leadership: Combining vertical and shared leadership to transform knowledge work. *Academy of Management Executive*, 18,1(2004), page 47–57.

PeF07 Perera, G. and Fernando, M., Enhanced agile software development — hybrid paradigm with LEAN practice. *Industrial and Information Systems, 2007. ICIIS 2007. International Conference on*, 9-11 2007, pages 239–244.

PoP03 Poppendieck, M. and Poppendieck, T., *Lean Software Development - An Agile Toolkit*. Addison-Wesley, 2003.

QuK09	Qureshi, M. R. J. and Kashif, M., Seamless long term learning in agile teams for sustainable leadership. *2009 International Conference on Emerging Technologies*, pages 389–394.
RaA05	Ramler, R. and Auer, D., Encouraging self-organization: reflections on a quality improvement workshop. *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, 2005, page 284–291.
Ram98	Raman, S., Lean software development: is it feasible? *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, volume 1, 31 1998, pages C13/1 –C13/8 vol.1.
Rei02	Reifer, D., How good are agile methods? *IEEE Software*, 19,4(2002), pages 16–18.
RCL03	Reilly, R. and Lynn, G., Power and empowerment: the role of top management support and team empowerment in new product development. *PICMET '03: Portland International Conference on Management of Engineering and Technology Technology Management for Reshaping the World, 2003.*, pages 282–289.
Sch95	Schwaber, K., Scrum development process. *OOPSLA Business Object Design and Implementation Workshop*, volume 27. Citeseer, 1995, pages 10–19.
SSB05	Salas, E., Sims, D. E. and Burke, S., Is there a "Big Five" in Teamwork? *Small Group Research*, 36,5(2005), pages 555–599.
TaN86	Takeuchi, H. and Nonaka, I., The new new product development game. *Harvard Business Review*, 64,1(1986), page 137–146.
TaP04	Tata, J. and Prasad, S., Team Self-Management, Organizational Structure, and Judgments of Team Effectiveness. *Jounal of Managerial Issues*, XVI,2(2004), pages 248–265.
UFC96	Underwood, J., Fitzgerald, M. and Cassidy, M., Self-directed teams in power electronics manufacturing. *Applied Power Electronics Conference and Exposition, 1996. APEC '96. Conference Proceedings 1996., Eleventh Annual*, volume 1, 3-7 1996, pages 64 –68 vol.1.
Uus91	Uusitalo, H., *Tiede, tutkimus ja tutkielma - Johdatus tutkielman maailmaan.* WSOY, Helsinki, 1991.
Whi01	Whitehead, R., *Leading a Software Development Team – A Developer's Guide to Successfully Leading People & Projects.* Addison-Wesley, 2001.

Whi08 Whitworth, E., Experience Report: The Social Nature of Agile Teams. *Agile 2008 Conference*. IEEE, 2008, page 429–435.

Win94 Winter, R., Self-directed work teams. *Proceedings of 1994 IEEE/SEMI Advanced Semiconductor Manufacturing Conference and Workshop (ASMC)*, 00, pages 123–125.

WoJ96 Womack, J. and Jones, D., Beyond Toyota: how to root out waste and pursue perfection. *Harvard Business Review*, September-October 1996.

WRJ90 Womack James P., Jones Daniel T., R. D., *The Machine that Changed the World*. Rawson Associates, 1990.

Yin94 Yin, R. K., *Case Study Research: Design and Methods.* Sage Publishing, Beverly Hills, CA, second edition, 1994.

Yin03 Yin, R. K., *Applications of Case Study Research.* Sage Publishing, Beverly Hills, CA, second edition, 2003.

Appendix 1. Team member interview questions

Forewords: In this interview think me as a person who never actually was in Factory. You may wonder why I am asking something that I already know but I would like to hear these things from your point of view. If you need to refer to me as a person, you can say "you" or use my name, whichever you feel is natural for you.

If the interviewee participated also the first project, tell him that he can also make comparison between the projects if he likes.

Background

Could you tell me first briefly how much you have done software development before this project?

Why did you get in to this project?

What is important for you when doing software development?

How was this project different compared to the ones you have had before it, in work or studies?

Theme: Individual/internal autonomy

How would you describe your role in the project?

Tell me about the team structure. What kind of roles did the others have?

I'm sure you did lots of things in the project but tell me what was "your thing".

- Mention a couple of tasks you did.

Based on what the interviewee mentioned:

Tell me more about *this task (or tasks).*

- How did you get the task?
- Was this the usual way to get / take tasks?
- What happened before you started coding (if anything)?
- Tell me about the coding phase.
- What happened after the coding?

What kind of practices you had in your work?

Can you remember such a situation that at first you had an idea how something should be done but finally it was decided that it will be done in another way?

- What was the reason for this?
- How did you react to the changed plans?

Theme: External autonomy

Who were the people who influenced on your own or team work outside of the team?

- What was the influence like?
- Benefits / drawbacks?

If and when the customer is mentioned: What was the cooperation with the customer like?

If the coach(es) is mentioned: What was the cooperation with the coach(es) like?

Theme: Shared leadership - how did the team make decisions?

Can you think if there were some "big and important things" in the project?

- Something that affected the whole team or many team members.
- Some big feature

Tell me more about *this*.

- How were the decisions made in this matter? Who decided?
- Was this usual in this project? Can you think a different kind of situation?
- Did you feel that you have a possibility to participate in decision making?

Did you often notice situations that the others had decided something important without you?

- What do you think was the reason for this?

Theme: Learning

What did you learn during the project?

Are you able to tell me what different ways you had for learning?

Did you often get feedback from your work?

- What was the feedback like?
- Was it useful for you? Did it help you to improve?

Think some problem you had (for example in the task you mentioned) and tell how you were able to solve the problem.

- Was this the typical way to solve problems?

How did the team improve its work during the project?

If retrospective is mentioned:

Do you think that the retros were useful for you or for the team?

- Can you give an example of something that changed your behaviour due to a retro?
- Can you give an example of something that changed the team's behaviour due to a retro?

Theme: Redundancy

Think about your own tasks. How many in the team in your opinion could have done the same tasks?

Let's assume that you got sick. What do you think would have happened to your tasks?

- Why do you think this would have happened?

Let's assume that some others in the team would have been sick. How would have this influenced on the rest of the team?

- Would have it been the same situation for all persons in the team?

How aware were you what the others were doing?

If daily meeting is mentioned: Tell me more about the daily meetings.

- What were the benefits (for you)?
- Did they have an effect on your work somehow?

Theme: Team orientation

Let's go back to the beginning of the project. What do you think was the project goal then?

Did the goals change during the project?

Based on the goals mentioned: How did the team goals affected in your daily work?

How did the team decide what will be done in the upcoming days?

How did you choose the tasks for yourself?

- Was it clear for you what tasks were more important than others?

Based on personal goals mentioned in the beginning: You mentioned in the beginning that *xxx* is important for you in software development. Do you think that the team goals were in contradiction with this at any stage?

Did you have a possibility to do interesting tasks?

- Why? / Why not?

Did you propose your own ideas for the team? Give me an example.

- How did the team like it?
- Did you do tasks based on your ideas afterwards?

Did you help others in their work?

Theme: Communication and collaboration

In case these were not mentioned before this, ask the interviewee's opinion about them: What were they like? How did they affect your own / the team's work?

- Daily meeting
- Customer demos
- Retrospectives
- Code review
- Plan review (in the second team)

Appendix 2. Customer interview questions

Background

What kind of projects have you had previously so that you have been the customer?

What kind of things do you expect from the team as a customer?

- How do you define value in software development?

This project

What kind of expectations and goals did you have before the project?

Tell me about your first contact with the team.

Tell me about the communication between you and the team. What was it mainly like?

Customer demos

Tell me more about the customer demos. What kind of events they were?

Tell me some change requests that you proposed in the demo.

- How detailed were them?

What kind of feedback (positive, negative) you gave for the team in the demos?

When you think about the demos, how well did the team manage to do those things that you asked in the previous demo?

- What do you think was the reason for this?
- Did the team sometimes skip some relevant things?
- Did you notice situations that the team did something irrelevant from your point of view?

Did you have any negative feelings during the demos?

Could you choose some bigger feature you asked the team to do? Tell me more about it.

- What was the background for the idea?
- How did you present it for the team?
- How precise instructions did you give for the team?
- How did the team succeed during the following week?
- When the team proposed its implementation, what did you have to change in it?

Time between demos

In the beginning you told about your *expectations*. How did these change during the project?

How actively did you participate the team work between the demos?

- Did the team contact you between the demos?

If you had been constantly (daily) involved with the team between demos, what would have you changed in how the team worked?

Did you hear anything such that the team would have needed more time with you or that there was not enough mutual time?

Project end

Now when the project is over, can you compare your initial estimate to what you actually achieved?

- In the beginning you defined what the value is. Was the team able to add value?

What did you learn from the project as a customer?