Characteristics for Efficient Autonomous Cross-Functional Teams

A Grounded Theory Study of Software Engineering Teams

Andreas Hauge Standal

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

May 2019

Characteristics for Efficient Autonomous Cross-Functional Teams

A Grounded Theory Study of Software Engineering Teams

Andreas Hauge Standal

Characteristics for Efficient Autonomous Cross-Functional Teams

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

In this master thesis I aim to arrive at a theory which can answer the question of how software engineering teams can become more efficient, and how they can use cross-functionality and team autonomy to arrive at this goal. There has not been done enough research on this subject, as most research on software engineering teams tend to focus on a single part of software engineering such as frameworks like Scrum or mindsets like Lean. Therefore the scope of this thesis is quite broad, and envelops software engineering teams in practice as a whole.

The research method applied was Grounded Theory, which was useful due to the broad scope of the thesis. Grounded Theory was used to its full extent, using iterations of data collection and data analysis to arrive at an emerging theory. The research was conducted at a large Norwegian bank, in the spring of 2019. Two development teams were followed, observed, and interviewed - extensively.

The emerged theory has the main category *Team Efficiency*, together with six related central categories: *Flexibility*, *Autonomy*, *Cross-Functionality*, *Mindset*, *Morale*, and *Coaching*. These central categories' underlying topics, and relationships, are presented and detailed in depth, and aims to answer how to implement cross-functionality and flexibility to achieve team autonomy.

Acknowledgements

The work on this master thesis has been invaluable to me, and would not be possible without the help of several people. First and foremost I would like to thank my supervisor Viktoria Stray for continuous guidance, enthusiasm, and support throughout this whole process. Her knowledge and experience in both the industry and the research domain gave me a lot of valuable insight. Furthermore I would like to thank Nils Brede Moe and SINTEF for welcoming me on this project. I am also incredibly thankful to the teams and team-members who welcomed me with open arms and integrated me into the teams. This made the research process seamless, and their willingness and enthusiasm to contribute was paramount.

I am grateful for my friends at The Department for Informatics - having someone to discuss the thesis with and cooperate with has been of great importance. Lastly, I would like to thank my parents and my girlfriend, Larissa, for their never-ending words of encouragement and support.

Contents

List of Tables

List of Figures

1. Introduction

Autonomous cross-functional teams are becoming increasingly popular in software development projects. Breaching with the traditional hierarchical model for communication and decision-making, autonomous teams aim to decentralize the development process, from functionality and design choices to larger operational decisions, to increase the efficiency and effectiveness of the software development process. In order to achieve this level of broad competency, teams require their team-members to be cross-functional, meaning that they have significant knowledge about several central domains, such as programming, design, analysis, testing, user experience, operations and management, et cetera. This way of organizing teams stand out in stark contrast to the more traditional single-purpose teams or component teams that have been utilized for a long time, especially in plan-driven development.

1.1. Motivation

As organizations become more agile, and the use of autonomous teams become more widespread, the need for formal methods, backed up and proven by research becomes necessary. As it stands, there is a severe lack of research done on agile development processes and their effectiveness, and even more so in regards to autonomous cross-functional teams. The IT industry tends to adapt to newer processes on a whim, without putting much emphasis on whether the expected outcome is actually realistic. In addition to this, many organizations utilize parts of a development process, merged together with parts of other development processes, creating a hybrid of processes. This complicates the matter even further, as the combinations of different processes and mindsets (Extreme Programming, DevOps, Agile, Lean, Scrum, Kanban, etc) are essentially limitless - so how are we supposed to extensively validate all of these combinations and their real world application?

Instead of focusing on individual development processes, it seems appropriate to take a broader perspective, and rather focus on the general mindset that all of these development processes have, and their common interests in what they wish to accomplish. Autonomous cross-functional teams tick all of these boxes. They can generally be used with either Scrum or Kanban (or a mix of both), without diverging from the goal of being self-sustained, effective, and highly efficient. The finer details of autonomous teams and their development process therefore becomes irrelevant, as long as the general mindset is followed. This in turn makes it easier to generalize the findings of studying autonomous cross-functional teams, how they work, and how they can be implemented effectively, as the focus remains on their structure and place in the organization, rather than the intricate details of their chosen development process framework. However, it is still important

that agile principles are followed, and that essential topics such as Sense & Respond[28], UX-design, self-organization, cross-functionality etc. are a part of the iterative process.

1.2. Research Context

My research and thesis work will be in collaboration with an already existing project called A-Team (short for Autonomous Teams). I will be working with representatives from both SINTEF and The University of Oslo (UiO) - more specifically the Institute for Informatics. Several students before me have been a part of this project, from both UiO and NTNU (Norges Teknisk-Naturvitenskapelige Universitet), where they have researched different aspects of teamwork such as coordination and leadership models.

I will be working in collaboration with a major Norwegian bank, at some of their IT development teams, who provide software systems and solutions for their customers. There I will spend time observing and taking part in their meetings, I will observe their day-to-day work-flow and how this is impacted by their team-structure and development processes, as well as interviewing team-members and project leaders.

1.3. Research Questions

This thesis aims to answer the following research questions:

RQ1: Which characteristics are essential for efficient and successful software development teams?

The teams observed in this thesis are however not just regular teams, but teams built with cross-functionality and team autonomy in mind. Therefore an additional focus was central for this thesis, which can be summarized in *RQ2*:

RQ2: How does cross-functionality and focus on team autonomy impact the efficiency of software development teams?

1.4. Thesis Structure

Chapter 2: Background goes through relevant topics from the software engineering research literature. It presents topics that will be important background knowledge for the reader, in order to fully understand the aims and the results of this thesis. Additionally, some information regarding how these topics are used in practice by the teams I observed will be presented, as a comparison between theory and practice.

Chapter 3: Methodology details which research methods I have used - first and foremost *Grounded Theory*. Furthermore it will go into detail about *Data Collection*

and *Data Analysis*. Finally I present a literature review of grounded theory usage in Software Engineering, compared with previous work by Stol et al. [35].

Chapter 4: Research Context details the organization where the research took place, to make the reader familiar with the teams and their structure and processes.

Chapter 5: Results presents my final theory, which is grounded in the data I collected and analyzed through grounded theory, and which aims to answer the established research questions.

Chapter 6: Discussion compares my findings with relevant findings in the research literature. Selected topics of special importance are discussed in detail, and implications for practice and theory are presented.

Chapter 7: Conclusion summarizes and concludes the thesis in light of the process and results.

2. Background

Software Engineering is the act of developing software, either for a customer or for the organization itself. To perform this act, there has to be an established process in place to ensure that the project flows in an expected and streamlined way. This established process is called the software development process, and is a term that not only defines the process and "what to do", but also encapsulates the team structure and "who does what".

2.1. Plan-driven development processes

In the early days of software development, there was a lack of research done on the topic, so in order to specify a software development process, you had to look for inspiration in other domains. One of these were regular system engineering processes, which is where the original plan-driven software development method is derived from[29]. Similarities between regular system engineering processes and software development processes are clear, with a focus on phases that have to be completed before the next phase can start, such that planning, analysis, and design comes before production, integration, and testing. This sequential phasing, or cascade, of one phase to the next one became known as the "waterfall method". The waterfall method was adopted by most organizations, as software development became a rapidly growing industry.

The waterfall method generally consists of 5 phases[34]:

1. Requirements specification
2. System and Software design
3. Implementation and Unit testing
4. Integration and System testing
5. Operation and Maintenance

As previously mentioned, each of these phases had to be finished before the next one could start. As you can imagine, this led to a plethora of problems, specifically when issues where found during testing, or when a change request from the customer demanded a change in the functionality of the system. In those cases, the teams would have to go back to previous phases, creating a lot of unnecessary overhead, to fix the issues. This is problematic, as research over the years have shown that there is an exponential increase in change-request costs the further the project is in its life cycle[6].

The increased complexity of changes and the huge increase in cost, together with increased overhead, late deliveries and lack of risk management[23] led the industry towards changes in the late 1990's and the early 2000's based on some early thinkers like Barry Boehm, who already in 1988 had proposed the spiral model as a more iterative way of doing software development[5].

In 1995 the first official paper on Scrum was presented by Schwaber[31], and in 1999 Beck published the first version of Extreme programming[3], making way for the revolution of agile development processes.

2.2. Agile development processes

The need for a development process which could provide quicker incremental deliveries, require less managerial and hierarchical structures, and could handle less specification upfront was sought after as the industry had caught up with the inefficiencies of the waterfall method. Agile methods that usually follow an incremental development cycle, often using iterative time-boxes (Scrum[32, 30]) or task-boxes (Kanban[2, 40]) was proposed. These methods created a new way of thinking about software development, focusing less on requirement specifications, tools, phases, and plans, and more on teamwork, cohesion, responding to change, and customer relations/collaboration[4].

2.2.1. Scrum

Scrum has become one of, if not *the* most used agile development process framework of modern software development. It was first presented in 1995[31], and later versions were published in 2002[32] and 2004[30], predominantly by Ken Schwaber, with collaborations within the industry.

Scrum is an iterative *time-boxed* agile development process framework. Iteration duration should be constant, and normal values are 2 or 4 weeks. An iteration is called a Sprint, and each sprint should produce a working product, such that it can be implemented in the existing solution, and be shipped to the customer at regular intervals. This ensures that the customer has enough time with the system for acceptance testing throughout the project, eliminating the risk of a failed system validation at the end of the project[30]. It is the responsibility of the Scrum team as a whole to ensure that the goals for the sprint is met. This is important, to prevent work from sprint 'A' overflowing into the following sprint 'B'.

Instead of having a massive requirements specification detailed at the beginning of the project, the team(s) have a product backlog with items that have to be implemented in the final solution. Backlog items are estimated, and chosen for the particular sprint, while making sure the team hasn't bit off more than they can chew for the sprint.

There are certain central topics that Scrum defines, which I will cover in detail:

- Scrum Team

- Scrum Roles
- Scrum Ceremonies

Scrum Team and Roles

An essential part of Scrum is the Scrum team. Contrary to plan-driven development teams where each team had one purpose such as testing or design (often called Single-function teams[19]), Scrum teams are by definition *Feature teams*[19]. Feature teams are cross-functional, meaning that their team-members are knowledgeable and competent in more than one domain. This is beneficial because it ensures that there is no downtime for any of the team-members, as they can perform whichever tasks are available - increasing efficiency. A feature team has some distinct attributes that are highly valued in Scrum[19]:

- They are long-lived, meaning that they continue working together after the end of the current project.
- They consists typically of 7 +- 2 people (in Scrum)
- They are cross-functional, and made up of generalists that can do work in several domains
- And lastly they can be co-located, although this is a complicated matter, as some[42] present findings of great success of distributed Scrum teams, while others[16] present more varying results, depending on the approach taken.

There are two main roles in a Scrum team in addition to the regular team-members - the *Scrum Master* and the *Product Owner*.

The Scrum Master

One of the main focus points of Scrum is its flat "management" structure. The goal is to create a team, and a group of teams, that have little to no hierarchies. In order to do this, the removal of *project leaders/team leaders* is necessary. Although this flat structure does its trick in most situations, it was found that there was a need for a person that could facilitate the team, who could be a person that could bring guidance in difficult situations, and that could facilitate meetings[30]. This person is now known as the Scrum Master. His/hers job is not to lead or govern over the rest of the team, but rather to facilitate the formalities, talk with management if necessary, and generally be a guidance and a reference point for the rest of the team. Larman & Vodde writes: "One Scrum goal is an engaged self-managing team; thus, a good Scrum Master will avoid taking any management-like role or activity, including team representative."[19].

The teams at the observed organization in this thesis had so-called team-leaders. These team-leaders functioned however more like *coaches*, than team-leaders. Some team-members compared them to facilitators, and others to scrum masters, depending on the situation. I will outline the details of this in the *Results* chapter.

The Product Owner

As previously mentioned, the project has a backlog of items/functionality that is due to be implemented in the final solution, called the *Product Backlog*. At the start of each sprint during the *Sprint planning meeting* the team selects and estimates an appropriate amount of backlog items. Once the items have been selected, they are put in the temporary *Sprint product backlog*, containing only the items that are due to be implemented in the current sprint. This process is facilitated by the product owner[41]. He is in charge of the product backlog and is the person who prioritizes backlog items. The product owner often has ties to the customer, such that he can speak on their behalf. This creates a very streamlined process of picking/prioritizing and estimating what should be done in the coming sprint. This process may be more or less done by the team instead of the product owner, but the product owner remains accountable[41].

The product owner at the observed organization was a part of the team, and had considerable contact with the product owners of the other teams, the project leaders, and the customers. He would take care of project-related issues that the rest of the team did not need to worry about.

Scrum Ceremonies

Scrum consists of a few very important ceremonies, or more informally, meetings. Together with the Scrum team, they are the backbone of Scrum, and serves very distinct purposes throughout a project, both in regards to communication and cooperation. The meetings are[41]:

- Sprint Planning meeting
- Daily Stand-up meeting
- Sprint review meeting
- Sprint retrospective meeting

Sprint Planning Meeting

The Sprint Planning meeting marks the beginning of a new sprint. The whole team gets together for a maximum of 8 hours (for a 1-month sprint) to set the goals for the coming sprint. Schwaber and Sutherland specifies two questions that the planning meeting sets out to answer[41]:

- "What can be delivered in the Increment resulting from the upcoming Sprint?"
- "How will the work needed to deliver the Increment be achieved?"

It is the whole team's responsibility, together with the product owner, to ensure that the selected backlog items are of the right size and complexity to ensure that their completion is feasible within the time-limit of the sprint. To do this, estimation of the

backlog items is necessary. There are several ways this can be done, ranging from *expert estimation*[17] and previous experiences (collected data), to more informal methods such as *Planning Poker*[11], or a mix of both[22]. The planning meeting should also answer the second question, and as such "the Development Team should be able to explain to the Product Owner and Scrum Master how it intends to work as a self-organizing team to accomplish the Sprint Goal and create the anticipated Increment"[41].

Daily Stand-up Meeting

The daily stand-up meeting is an informal meeting that takes place at the beginning of each workday, lasting about 15 minutes. Its goal is to inform the team-members of the progress and what has been done since their previous meeting[41]. The Scrum master is the facilitator, and ensures that everyone gets to speak. He asks them three questions[41]:

- What has been done since the last meeting?
- What is going to be done until the next meeting?
- Did any issues arise that may inhibit the team in reaching the sprint goals?

However Stray et al. elaborates on findings, where in their observations only 24% of the time allocated for the meeting was used to discuss the three scrum questions, while as much as 35% was used for "Problem-focused communication"[37]. Thus an interesting phenomenon arises as even though daily stand-up meetings have been shown to be useful[39], it is not necessarily because of the three specific questions laid out in the agile literature, but more because of the overall information sharing and problem solving.

During my observations of the team's standup meeting, similar data was found. The teams focused a lot more on problem solving, and regular discussion, than the three distinct questions. Through the interviews I conducted, it also became clear that there is a lot of differing opinions on how to conduct the standup-meeting, as some feel the meeting is too short, while others (usually developers) feel that they do not get anything useful out of them due to the wide variety of specific topics that they might not be familiar with. The teams' standup meeting tended to last around 10 minutes, which is a decrease from the length the meeting lasted before they implemented changes to reduce the time-usage (more on this in *Chapter 3* on Data Collection from Retrospective reports). They also performed a prolonged "standup"/team-meeting on mondays, which seems to be more productive according to the team-members.

The importance of standing during the meeting has also been documented as it greatly decreases the overall duration of the meeting[39]. Whether or not the "daily" part of the definition should be taken literally is also a point of discussion, as some developers feel the cadence of the meeting is too short, and the duration too long[39].

The Sprint Review Meeting

The sprint review meeting is held at the end of the sprint. It is by many seen as a demo of the product/functionality developed in the previous sprint, but this is only one part of the meeting. In addition, the review meeting should focus on the product as a whole, and assess/validate whether or not the product is going in the right direction. Here it is important that the product owner is active, to ensure that the customer's interest is being considered and properly reflected by the product and its functionality[41]. As a result of the review, a revised product backlog is created, documenting any changes done to the backlog items, and a relatively clear path for the next sprint is available[41] (although the specifics will be handled during the next sprint planning meeting).

The Sprint Retrospective Meeting

The sprint retrospective meeting has a clear difference compared to the review meeting. The review meeting focuses on the product itself, whereas the retrospective meeting focuses on the development process and potential improvements to the work-environment[41]. Separating these two meetings is important to ensure that both the product and the process is evaluated as part of the agile philosophy[4].

Backlog Grooming meeting

The teams observed in this thesis conducted another meeting, which is the backlog grooming meeting. The purpose of this (weekly) meeting is to gather the team-members to take a look at the backlog. There they review new items and tasks, discuss them, prioritize their severity, and finally assign them to the appropriate person(s), either for further analysis, or to signal that it is ready for development. The meeting averaged approximately one hour, and every team-role was usually represented, allowing for a lot of cross-role discussion and leading to a lot of different perspectives to be heard. Due to the team's implementation of Kanban boards, this naturally deviates a bit from the time-boxing of scrum, and is more in line with the task-boxing of Kanban.

2.2.2. Kanban

Kanban dates back to Toyota's assembly line and production mentality, as part of Lean thinking[40]. Kanban can be described in many ways, but in regards to software engineering it has been adopted as an alternative agile development process, focusing on coordination, efficiency and elimination of waste[2]. It is task-boxed, meaning you have a backlog of prioritized items to be implemented or tasks to be completed, and the team-members pick and choose their tasks accordingly[2]. The tasks are worked on and tested until completion, as opposed to Scrum's time-boxed modell where the focus is more on how much can be done within the time-limit of the sprint. Kanban has been shown to improve quality of software, coordination and communication, and increased consistency of delivery[1]. Kanban has been used as a standalone development process, but recently

it is becoming more popular to include certain elements (such as Kanban boards) from Kanban into Scrum (much like XP has influenced Scrum and agile practices in general).

The teams I observed used Kanban extensively, especially with the usage of Kanban boards and task-boxing. The most used board had the following lanes/columns: *Enlisted* for new tasks, *To specification* for tasks needing further details, *Backlog*, and *Ready for development*. The *ready for development* lane has a limit of 15 tasks (for one of the teams), and they strictly enforce this to reduce clutter and to make it simple for developers to choose tasks based on prioritization and other factors.

2.2.3. Lean Thinking

Lean thinking is another way of approaching software development. It is comparable to the level of structuring as agile methodologies provide, as it's more focused towards the organization as a whole, and the overall mindset, rather than a specific framework like Scrum. Lean thinking stems from the old automotive industry, specifically in regards to Toyota[25]. The core of lean thinking is respecting the employees, allowing them to perform the work needed while letting them have the freedom to always look for improvements, leading to continuous improvement (*kaizen*). In order to accomplish this, company-wide principles have to be followed, such as[19]:

- Eliminating waste while seeking value.
- Kaizen - actively support employees in seeking improvements in the products and the processes.
- Just-in-time(JIT) - don't produce something you may not need, instead focus on producing exactly what you know you will need (also called a pull-system, as opposed to a push-system).
- Work towards achieving *flow* - fix problems as soon as they arises, eliminate waste, and work towards efficient and streamlined production and processes.

As with Kanban and XP, Lean is often implemented partially wherever it is seen appropriate. Larman and Vodde[19] argues against doing this, as Lean thinking needs to be a company-wide mindset in order to succeed (just like agile methodologies). There is however a lot of insightful and useful principles from Lean thinking that even the individual can keep in mind, such as eliminating waste, and striving towards improvement, but as Larman and Vodde stated[19], it is questionable how much of an impact this will have on the organization as a whole unless a significant portion of the teams (and team-members) adopt this way of thinking.

2.2.4. DevOps

DevOps, as the name suggests, aims to combine development and operations within the same team. The idea is that this will reduce the amount of overhead that is usually prevalent in organizations where there are a lot of hierarchies, middle-managers, and

inter-team cooperation. In principle, this sounds great, but there is a lot of variance in how DevOps is defined[20], and it seems to depend a lot on the culture in which it is deployed[33]. Lwakatare et al.[20] has identified four dimensions or characteristics of DevOps from relevant literature and the industry, but notes a lack of empirical evidence:

- Collaboration
- Automation
- Measurement
- Monitoring

Upon reading this, it seems very similar to principles and goals that agile methodologies (including frameworks like XP, Scrum, Kanban etc.) focus on, and the distinctions between them (seen on a high level) may not be clear. The lack of a clear definition and guidelines for implementation[33, 20] makes DevOps difficult to analyze. However, its focus on rapid deployment, quick iterations, and domain-crossing team compositions are promising for our focus on Autonomous cross-functional teams.

2.3. Autonomous Cross-functional Teams

The last 20 years of software development processes and practices, including team-structure and development frameworks seem to go in the direction of highly competent teams that are cross-functional and self-organized, or *autonomous*. Teams that are made up of cross-functional team-members, competent in several disciples, including both technical as well as managerial and operational skills, allowing the team to perform all actions from design to implementation to deployment and maintenance of software systems, autonomously.

In their research on grounded theory and self-organizing agile teams, Hoda et al.[13] specifies in detail this balancing-act performed by self-organizing Agile teams between:

- "Freedom provided by senior management and responsibility expected from [the team] in return."
- "Specialization and cross-functionality across different functional roles and areas of technical expertise."
- "Continuous learning and iteration pressure, in an effort to maintain their self-organizing nature."

"These three balancing acts were not easy to perform but, when done well, ensured the teams were able to sustain their self-organizing nature"[13]. Moe et al.[21], regarding self-organizing teams, found that "the most important barrier to be the highly specialized skills of the developers and the corresponding division of work" backing up the need for cross-functionality in such teams in order to avoid issues regarding division of work.

In another paper[14], Hoda et al. identifies several roles that are typically found within successful self-organizing agile teams; Mentor, Coordinator, Translator, Champion, Promoter, and Terminator. In general, these roles are descriptive of people who perform certain actions that are linked to positive results in self-organizing teams, such as:

- Providing guidance and encouragement towards adherence to agile methods.
- Managing customer expectations and coordinating customer collaboration.
- Securing and sustaining senior management support.
- Identifying and removing team members threatening the self-organizing ability of the team.

The second point, regarding customer relations, seems to be a particularly central aspect of the successfulness of self-organizing teams, as we have seen in previous sections of this review. Additionally, a study specifically on that topic was conducted [15], again by Hoda et al., further backing up the statement.

There are also several challanges related to self-organizing teams. Hoda and Murugesan[12] focused on the aspects of project management, and found challenges including:

- Delayed/changing requirements and eliciting senior management sponsorship at the project level.
- Achieving cross-functionality and effective estimations at the team level.
- Asserting autonomy and self-assignment at the individual level
- Lack of acceptance criteria and dependencies at the task level

Achieving this level of streamlined software development is the ultimate goal, but succeeding in figuring out the optimal processes and structure has proven difficult. Many methods have been proposed, some more influential than others, and the question is which attributes belong, and which don't. There are clearly available points that can be helpful in achieving successful autonomous teams, and the specific way of implementing such teams successfully should be the focus of future research on the topic.

3. Methodology

In this chapter I will outline the research design and plan for this thesis. It will include theories that I adhered to, how the data collection process was executed, and how I analyzed and made sense of the data I collected. Additionally, at the end I will present a literature review of Grounded Theory usage in software engineering.

3.1. Research Design

The overall aims of this thesis is to investigate autonomous cross-functional teams, and more specifically; how they work and operate in the industry. As such there are no established research questions beforehand. To deal with this situation, I required a theoretical framework to guide me to a valid result. Due to this, it seemed the most fitting to use the qualitative research method of grounded theory [10].

3.1.1. Grounded Theory

Grounded Theory is a term first coined by Barney Glaser and Anselm Strauss in 1967 [10]. It was developed for the social sciences to link their theories causally and validly to their research. During a time where quantitative research methods ruled, and were seen as the best course of action for empirical research, Grounded Theory aimed to legitimize qualitative research and their results to bring qualitative research up to the same level as quantitative research [36]. Grounded theory aims to create a causal link from research to theories. In order to do this, Glaser and Strauss [10] claims that theories must be *grounded* in data collected during the research (hence the name grounded theory). It is worth mentioning that "grounded theory" can mean both the *method* grounded theory, and the emerging theories themselves.

Grounded theory was developed foremost for the social sciences. During the following 20-30 years (after 1967) it increasingly gained popularity in other industries and research domains, like psychology and anthropology to name a few [36]. Strauss writes "because grounded theory is a general methodology, *a way of thinking about and conceptualizing data*, it was easily adapted by its originators and their students to studies of diverse phenomena" [36]. This has implications for studies in informatics, as it shows that grounded theory is not only limited to its original research domains. Glaser and Strauss had actually foreseen this evolution back in their original paper, where they wrote:

> "A grounded theory that is faithful to the everyday realities of a substantive area is one that has been carefully *induced* from diverse data...Only in this way will the theory be closely related to the daily realities (what is actually

going on) of substantive areas, and so be highly applicable to dealing with them [10].

Grounded Theory was mostly used in the information systems field, with a strong focus on technological (and more recently organizational and managerial) issues [24]. However, in more recent years, it seems as if the focus has been shifted quite drastically, and Grounded Theory has become more and widely used in Software Engineering research. This became clear through the research that Stol et al. did on Grounded Theory usage in Software Engineering [35], and my own literature review, which I have discussed in 3.4.1 and 3.4.2 respectively. The findings show a clear focus on qualitative research on agile practices, self-organizing teams, autonomy, and Software Engineering-specific tasks, see table 3.4.

In more recent years, the focus of research papers regarding the use of grounded theory in information systems research has somewhat changed from whether or not it is applicable, to *how* we can effectively utilize the method to its full potential. Urquhart et Al. in 2010 presents a set of guidelines for how to accomplish this [43]. They argue that the usage of grounded theory in information systems has promising results, but that the focus of grounded theory has been criticized for becoming somewhat synonymous with coding, and that the "theory" part has been neglected. The guidelines revolves around correct usage of central parts of grounded theory, such as *constant comparison*, *theoretical sampling and integration*, and the interplay between data gathering and analysis and its iterative nature. They note that the most successful research done in information systems with grounded theory is the one of Orlikowski [26], where Uruqhart et al. claims that the usage of their five guidelines is prevalent.

3.1.2. Choice and Grounded Theory Variant

It is the inter-personal cooperation and human-related parts of systems development that will be studied in this thesis, and as such I argue that the methodology of grounded theory is applicable by collecting relevant data from the autonomous teams, and methodically inducing and facilitating emerging theories from this data.

I will describe how grounded theory is performed in detail in section 3.2 and 3.3. Grounded theory is a qualitative research method, and as such it's fitting to use qualitative methods for gathering data. There is a special relationship between data collection and data analysis in the grounded theory method. Through the *constant comparative method* and *theoretical sampling*, which is quite unique to grounded theory, the researcher goes through iteration of data gathering and analysis of that data to achieve *saturation*. In short, it means that you have reached a point where there is no more knowledge to be learned, and further data gathering does not evolve the theory any further.

There are three predominantly used variants of grounded theory; Classical/Glaserian GT [10], Straussian GT (by Strauss & Corbin) [36], and the more recently developed Constructivist GT by Charmaz [7]. The three variants share several of the methods that is common for grounded theory research such as *constant comparison, theoretical sampling and saturation,* and *memo writing*. However they differ in three specific aspects:

Coding practices, philosophical positions, and their approach to the usage of literature. Weighing these differences against each other to arrive at a suitable GT variant was necessary for me. After much thought and considerations I ended up with choosing the Classical, or Glaserian, version of GT. I found the Straussian variation to have a much too rigid coding structure that would both be impractical, as well as it would limit the theory to a predefined structure that may not fit well with reality. Charmaz's constructivist GT focuses on the fact that there is not just one applicable theory to reality, but an unknown amount. This relativistic or postmodernistic philosophical mindset leaves too much room for personal biases in my opinion, and it also makes it difficult to follow the methods correctly if they are so flexible. Glaserian GT is however not without things to criticize, as it has a quite idealistic *positivistic* philosophical view. In short, a positivistic view states that there is an objective reality, which we in the case of GT can arrive at through unbiased research methods. Part of the Glaserian GT, which separates it from the other two methods, is its demand that you should not consult the literature before or during the research, but that your mind should be a blank slate. This is in my opinion infeasible, as I have both previous knowledge of the software engineering subject, as well as having consulted the literature on the usage of Grounded Theory in Software Engineering (see 3.4). In conclusion, I performed a Grounded Theory research following the Glaserian method, with the caveat that I had knowledge of the topic before conducting the research.

3.2. Data Collection

Data collection is an essential part of a research. It is important to have a clear picture of what kind of data you wish to gather before you start your research, as it directly impacts the types of results that you will end up with. The end goal of your research (or research questions) will drive you towards the optimal type of data for you. In this research, as I've previously stated in 3.1, the thesis will work towards the creation of a grounded theory, and as such I used appropriate methods for data collection, as I will outline next.

3.2.1. Qualitative Data and Grounded Theory

Data can be categorized as either qualitative or quantitative. Quantitative data is often characterized as metrics that are comparable and can be used for statistical analysis, whereas qualitative data is mostly data in textual form. This can make it difficult to analyze and deal with data that is qualitative, as the data is not comparative in the same way. There are several ways of collecting qualitative data, two of which I've chosen for this thesis, namely semi-structured interviews, and observation. I will outline why i chose these two, and how I performed them in detail in the following sections.

In grounded theory, the main activity for categorizing data is through different forms of *coding*. How I performed coding will be detailed in section 3.3. Coding is performed on textual data, which can be categorized as primary and secondary data. Primary data is data that is collected through the research, whereas secondary data is data that is

archived or collected before/outside of the scope of the research (often by other persons, for different purposes) [24]. In this thesis i primarily used primary data, collected through semi-structured interviews, participatory and non-participatory observations, and memos. However, I did supplement with some secondary data, such as older retrospective reports/notes, which I will detail in section 3.2.5.

3.2.2. Observations

The first method of data gathering I used in this study was observation. I followed two teams in charge of developing Software systems in the banking industry over a period of several months, from early January 2019, to late April 2019. During my time there, I spent my time observing in multiple ways, including:

- Observation of meetings
- Non-participatory observation of daily work-flow and team-member interaction
- Participatory observation in the form of both casual and formal conversations during work-time and lunch-hour

The main source of data came from the large amount of meetings I attended. The meetings ranged from daily Stand-up meetings, to larger meetings regarding the products, processes, or management. I categorized the meetings in separate categories, and recorded the amount of each, which can been seen in table 3.1. The *other* category consists mostly of techlead-meetings and team-leader meetings. There I got a lot of interesting insight into how the teams cooperate across team-borders, what kind of issues they face, and how they dealt with issues.

Meetings	
Stand-up	30
Backlog Grooming	5
Team-meeting	4
Other	6
Total	45

Table 3.1.: Number of Meetings

During the meetings I was making sure I was not interrupting in any way, and the goal was to ensure that the participants of the meetings did not notice me significantly to ensure that the meeting took place in a natural way. I spent my time in these meetings documenting as much as possible. Everything ranging from who was speaking and for how long, to what kind of problems were encountered, to what kind of frustrations were voiced by the attendees. In order to make this process as streamlined as possible, I used the following protocol for how to structure the data: I recorded the time and duration of the meetings, the amount of attendees (and their roles), and who (if any)

were attending the meeting remotely. Thereafter I noted who was talking, and the topics. I noted things that stroke me as interesting, or specific quotes that could become useful. After the meeting I would note down any thoughts or ideas I had gotten, if any, in memos or small paragraphs.

When it comes to the other two observation situations, the procedure for data collection had to be a bit different. In meetings, relatively expected situations arose which made it easier and more familiar to record the data, whereas in the open landscape any kind of situation could develop, whether it be serious or not. This created a larger variation of data types, and as such I needed to be prepared for unexpected situations. Every significant interaction and event was documented and logged with actors and time-stamps. The protocol had the following structure: I noted who was talking, about what topic and for how long. I noted whether or not it was a problem-solving issue, if someone needed help, or other similar classifications. This ensured that I could record the data in a structured way, similar to how I could record structured data from the meetings, which I later could easily code into relevant codes, as I will speak about in detail in the section on *Data Analysis* (3.3).

Lastly, to document casual and more formal conversations, the writing of the notes and memos would have to find place at a later time. This meant that the structure of this kind of data became more unstructured, and consisted more of a textual free-form. The documents and data gathered was stored in a systematic way, making the process of retrieving it for later use and analysis trivial.

I ended up with approximately 15 pages of "daily observations". At some point, the situations that arose did not give me any new data or insights, and thus I concluded that part of the data gathering. From there on I would only note if something new or special happened, which I consequently noted in memos.

3.2.3. Semi-structured interviews

The second main type of data gathering I conducted was semi-structured interviews. Where observations gave me a wide variety of data, the interviews allowed me to dig deep into topics and categories and get a deeper understanding. The interviews themselves were, as the section name states, semi-structured. A semi-structured interview is one of three interview types in qualitative research. The other two types are *Structured* and *unstructured/open* interviews, where the difference between the three is the rate of structure of the questions (and thus the resulting discussion/answers). A semi-structured interview aims to achieve a balance between strictness and openness. In reality, this meant that I would follow a relatively strict interview guide, where questions and topics where decided beforehand, but during the interview the discussions often overlapped with later questions/topics that I had not asked yet. Keeping this openness of discussion was important, as I was mostly interesting in getting accurate information, more so than getting the information in a strict form. The interview guide can be seen in Appendix B.

The interview process remained pretty static over the course of the study, with me and the interviewee alone in a room, with a voice recorder between us. The voice recorder

allowed me to focus on asking good relevant questions, and be active in the discussion, instead of spending my time writing and taking notes. After the interviews were done I would sit down and write a small summary, from memory, to give myself a small and concise reference of the interview before I got time for transcribing. This summary made me reflect on the interview, and gain insight into how I could improve the process.

I conducted nine interviews, with employees of different roles within the teams. The breakdown for this can been seen in table 3.2.

Interviews	
Developer	4
Architect	1
Tester	1
Test-lead	1
Designer	1
Director	1
Total	9

Table 3.2.: Number of Interviews

The transcription process possibly the hardest part of this process, however it was very valuable. It gave me a very solid overview and memory of what had been said, and by whom, which in turn made it a lot easier to code the interviews. This amount of interviews proved to be more than enough, coupled with the large amount of data I had previously gathered. Due to my many conversations with the team-members throughout the period, for example with the team-leader, I did not need to conduct more interviews to fill any holes in my data during analysis. More on theoretical sampling and saturation in the section on *Data Analysis* (3.3).

3.2.4. Memos

The third source of data came in the form of memos. Many probably have an idea of what memos are, but in this context they are related to grounded theory. The memos as data are quite different from the other two sources of data. Where the two other data collection methods gathered data directly from the team-members, the memos were created by me during the analysis parts of grounded theory. This meant that the data within the memos were quite different, and had to be used in a different manner, for a different purpose during further analysis.

The format and structure of the memos were quite vague, by definition, and for good reasons. The aim of a memo (in this context) is to serve as informal reflection notes where the researcher writes down ideas as they come to him [8, 10]. This meant that memos vary in length, content, and degree of conceptualization [8]. It was also important for me to write these memos as soon as I felt I had an idea or something that needed to be recorded. Strauss and Corbin notes [8]:

> "When stimulated by an idea, the analyst should stop whatever he or she

is doing and capture that thought on paper. This need not be a lengthy memo; a few generative ideas or sentences will suffice. Otherwise, important thoughts can be lost."

While memos are unstructured by definition (as I just outlined), they still have some common characteristics and features between them. These are outlined in Strauss and Corbin (1990) [8], and includes features such as dates, header content, reference styles, as well as some general guidelines for them. I gave each memo a classification, and a title. Table 3.3 shows the amount of memos for each classification:

Memos	
Reflections on Process	7
Code Definition	7
Theory and Relationships	6
Meta-memo	2
Final Report	7
Total	29

Table 3.3.: Memo Breakdown

3.2.5. Retrospective Reports

As previously mentioned, I got access to previous retrospective reports that one of the teams had archived. These six retrospective reports were divided into "positives" and "negatives" of the last period. By reading through these reports, I could sometimes spot things that had been in the "negative" column previously, be fixed and end up in the "positive" column. An example of this is that the team felt that the standup-meetings were taking too long, and were not productive enough. In the following retrospective report (approximately three months later), this had been fixed. The duration of the standup-meetings had been significantly reduced, and this improvement had been documented by being written in the "positive" column of the report. This additional data gave me more insight into how the team had performed before I arrived there, and gave me some talking points for the interviews that I would later conduct.

3.3. Data analysis

The section on Data Analysis is quite an interesting one, as the data analysis does not just take place once, but several times iteratively together with data collection. This means that a strictly sequential structuring of this chapter may not reflect 100% how the process took place in reality. Nonetheless, I will structure the sub-sections sequentially, and it is up to the reader to keep in mind that data collection was a part of each of these steps in the process, often multiple times in an iterative fashion. A depiction of how this process took place can be seen in figure 3.1, which highlights the steps in the

process as a timeline. The goal of the data analysis process was to reach a theory where I can claim that *theoretical saturation* has been approximately reached, meaning that no more information would positively impact the emerged theory. I will by the end of this section, outline to which extent this goal has been reached, and what the status was throughout the different facets of the data gathering and data analysis process. A timeline for this whole process, including data collection, can be seen in figure 3.1.

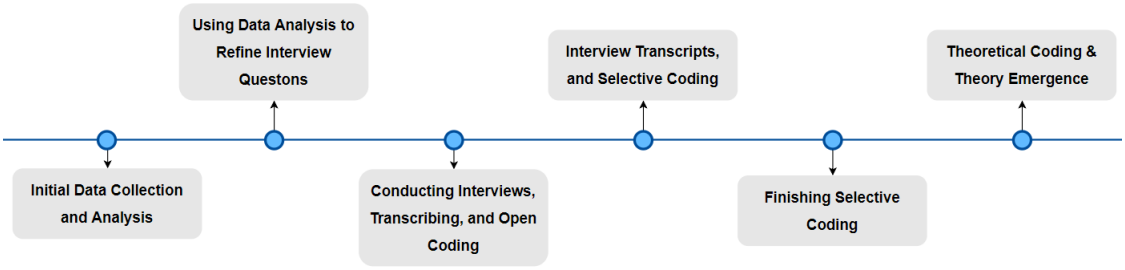

Figure 3.1.: Timeline of Data Collection and Analysis

3.3.1. Initial Data Collection and Analysis

I spent the first month or so collecting everything I could while being a part of the teams' daily work. I attended all meetings, all talks, and spent time getting to know the teams, how they communicate, inter-operate, and function. Throughout this initial period, I spent a lot of time brainstorming around what the scope and goal of the research should be. I noted down these thoughts into the aforementioned memos, as well as in documents and on paper where I would brainstorm, draw figures and put together initial relationships. During this period, I did not use any digital applications for coding, as I did not feel like it would result in any useful or coherent results. Instead, I spent time going over my data, highlighting interesting categories, issues, and characteristics, which could be viewed as an initial informal coding, or brainstorming. At the end of this period I had a pretty solid understanding of what the team was doing well, and what was problematic.

3.3.2. Using Data Analysis to Refine Interview Questions

At this point, I did not feel like I was getting any more useful data from daily observation, interaction, and meeting attendance. Thus, the next logical step was to figure out what I wanted to focus on, and to get more in-depth data and discussion with the team-members. I had previously, before the research period, created a tentative interview guide which served as a nice starting place for my brainstorming around scope and focus areas. However, this interview guide was way too broad, and did not focus on the tentative categories that had emerged from my data, and as such I had to quite significantly change the topics and questions for my interviewees. As mentioned in section 3.2.5, I also used retrospective reports that were archived for the prior year before

I arrived, to supplement my interview guide with questions that I otherwise would not have been aware of being relevant. This gave me some insight into what the history of the team was before I arrived, which turned out to be a useful addition to my data, especially regarding the history if their standup-meetings, which nicely transitioned into more general thoughts around meetings by the different team-members.

3.3.3. Conducting Interviews, Transcribing, and Open Coding

The first few interviews gave me further insight which helped me refine my questions even more. Some questions turned out to be relatively useless, as they had either been mentioned earlier in the interview, or the interviewees did not have any particularly good answers as the questions were too open. After the interviews, I spent about a week transcribing them. This process, while being quite demanding, was very useful. Not only because it gave me the interviews in textual form, but because it gave me time to really get to know my data, and to reflect on what was being said to an even larger extent than what I managed during the interviews. I took time to write down observations, or specifically good quotes which I knew could be used later. At this point, most of my major data collection was finished, and I had a solid idea of where the data would take me. This made me confident that starting to do line-by-line open coding on all my data would be the right thing to do. I used the application Nvivo 12 Pro for all of my digital coding. I made folders and structure for my data, and began open coding on each of the data types. First, starting with all daily observations, notes, memos, and meeting reports. In the beginning, this process was quite tedious, as I had to be careful and meticulous with how I named my codes, and how they would relate to each other. As my scope was quite large, I needed to capture a wide variety of topics, which meant that quite specific codes were created. During this process I wrote memos, as detailed in table 3.3, where I noted down any observations, thoughts, code definitions, ideas for theory, and possible sections for the final report. Once I was finished with this data (everything excluding the interview transcripts), I had 216 codes (including sub-codes), of which 75 codes had only one reference, 29 codes had two references, and 26 had three references. At the other end of the spectrum, I had five codes that had a significantly higher amount of references than the rest; 16, 17, 20, 23 and the most referenced code having 27 references. I will talk about how this large set of codes, and the vast difference in number of references per code, impacted me during further analysis, in the following section.

3.3.4. Interview Transcripts and Selective Coding

After coding all of my observational data, I began coding my interview transcripts - again line-by-line. By the time I had finished one and a half transcript, I realized that what I was doing was sub-optimal. I had so many codes that I was not able to keep them all in mind while doing my referencing. I decided to start doing selective coding, as I needed some structure to what I was doing, otherwise I would not be able to code the rest of my transcripts properly. I spent a long time looking through the codes I

had, looking for similar codes which could be grouped together, and similar code groups which could be arranged in an overarching category. I did this with pen and paper, brainstorming, until I was satisfied with a possible category. At this point, in Nvivo, I started grouping together my codes into hierarchies with sub-codes, and overarching categories. I ended up which six main categories, and an additional construct labeled "other" for the codes that I did not feel were relevant, or did not fit into any of the newly emerged categories. At this point it became a lot easier to finish open coding on my interview transcripts, as I had a structure with relationships between my categories which made it trivial to accurately code the rest of my data without need to keep all of my 200+ codes in my head. I finished with 299 codes, some of which were overarching categories and "super"-codes.

3.3.5. Finishing Selective Coding

I continued with selective coding after I finished coding the interview transcripts. I played around with different overarching categories, wrote memos of my thoughts and reflections on what kind of findings I had made, and how they fit together. The categories started to form well, and I was beginning to group together the codes within the categories into sub-categories that could serve as talking points and sub-sections for my theory presentation in the *Results* chapter. Once I was satisfied, I cross-checked my categories and sub-categories with my observational data, interview transcripts, memos, and any quotes I had saved. This lead to some new ideas and some restructuring of what I felt was important to talk about in my emerging theory. Some sub-categories were changed, and some were removed as they did not sufficiently fit into my category hierarchy. There were also some interesting categories that I left out as they did not fit in well with the other categories, such as "Relations with external suppliers" and "Testing Environments", which both had interesting problems that could be worth exploring, but did not fit with the rest of my findings. I had more than enough data on my final sub-categories, and as my scope is quite broad, encompassing the entire software development team, I did not have any glaring holes that needed to be filled with more data. That being said, there might be merit in the idea of running this whole process again, gathering more data with my current sub-categories and main categories in mind. This could potentially result in further findings, and additional main categories, but due to time limitations and the fact that this is a master thesis, this would not have been feasible. Theoretical saturation could thus not necessarily be claimed to have been reached, although the results suggest that I have reached a quite coherent and well put together theory, as I will expand on in the next section (3.3.6), and which will be presented in detail in Chapter 5 - Results.

3.3.6. Theoretical Coding and Theory Emergence

At this point, after selective coding, my category hierarchy was quite structured. Each of the bottom main categories have their own sub-categories, ending up with a total of 29 sub-categories, which are not a part of this figure. These sub-categories served as a basis for my theory going forward. I spent the following two weeks writing about each of these

sub-categories, which allowed me to get a deeper understanding for what each of them meant. At the end of this period, it became clear the the above-mentioned hierarchy needed some tweaking. I removed 2 categories, as I did not feel like they added anything to the theory other than generalization, which might not have made sense in the first place. Furthermore I removed the main categories in the third level, as I again felt like they did not accurately represent their sub-categories. I was now left with all of the 29 sub-categories, which were left unchanged as they simply represented the data they were grounded in. I spent some time reshuffling these sub-categories into approximately 6 main categories. These new categories, which can been seen in depth in Chapter 5 - Results, represented my data much more accurately. Additionally, relationships between these categories, horizontally, became possible, which I have detailed in section 5.8. I was then left with a coherent hierarchy of categories and sub-categories, and relationships between them that represents links from the research context (organization and development teams), which comes together as an emergent theory for how to achieve efficient and successful software development teams.

3.4. Grounded Theory and Software Engineering - A literature review

As someone with no previous experience with grounded theory, or in qualitative research in general, I wanted to make sure I had sufficient knowledge of the topic before conducting my own research. I started by reading through old literature of Glaser and Strauss, as well as Strauss and Corbin, but the more I read the more it became apparent that I needed more than just theoretical knowledge of the topic. I was introduced to an article by Stol et al. [35] by my supervisor, that could serve as a good starting point for further knowledge of grounded theory beyond theoretical knowledge. In their article, Stol et al. performs a (extensive)literature review of the usage of Grounded Theory in Software Engineering in practice. This was just what I needed to become familiar with the topic, and it gave me a deeper knowledge of how to both conduct, and how not to conduct, a grounded theory study in my field (or in any field for that matter, as the information in Stol et al. is quite universally applicable). Their article was a review of research articles up until mid 2015. To give myself further insight into the topic, I decided to continue in the footsteps of Stol et al., and performed my own literature review of grounded theory usage in software engineering between 2016 and the beginning of 2019.

I will thus in this section present the findings of Stol et al. first, and then continue with my own findings and reflections of the "do's" and "don'ts" of Grounded Theory research in the Software Engineering field.

3.4.1. Stol et al. 2016

Stol et al. wanted to explore and map the usage of Grounded Theory in the relatively new discipline of Software Engineering. They describe the different variants of GT, and identified the core set of GT practices. Through analysis of the use of grounded theory

in software engineering, they offer guidelines to improve the quality of both conducting and reporting GT studies. They emphasize the importance of the reporting of GT, as it is crucial that not just the results are presented, but also the specific details regarding methodology to allow for sufficient evaluation of the research.

From reading their article, it became clear that they felt that the devil was in the details. Their focus was on the widely varied reporting on the specifics of GT, which GT variant was used, the level of detail in the presenting of the articles' method usage, method slurring, and finally the type of output/results/theory that was presented as findings. I will go through each of these topics, and display Stol et al.'s findings, such that we can have a basis for comparison once I present my own literature review findings.

Article selection

To facilitate reproducibility. Stol et al. outlines their search process fairly detailed, however they do not go too much into the specific steps, but rather keep it on a surface level as an outline of their steps taken. They searched through several major research databases such as Scopus, IEEE Xplore, and ACM with the search string "'Grounded theory' AND 'Software Engineering'". This produced over 1700 articles, which obviously was infeasible to review. This lead to them implementing several restrictions on the set of articles, such as specific journals, and exclusion of peer-reviewed magazines and conference papers. In the end they were left with 98 articles, definitely a more manageable amount (presumably representing the top-end of research on the topic).

Data extraction

The next step was to inspect the research articles, and extract relevant pieces of information based on pre-defined guidelines and questions, which they outline in section 3.2 of their article (I mentioned most of these in 3.4.1).

In the following sub-sections I will present a condensed version of Stol et al.'s findings.

Grounded Theory "use" is ambiguous

Stol et al. begins by delving deeper into the resulting 98 articles, looking for statements regarding the literature's use of Grounded Theory. Almost half of the articles (46 to be precise) were found to simply borrow from parts of GT, without fully utilizing the method. Of these, 18 did not use the phrase "Grounded Theory" in the main text at all, 13 state they used GT "techniques" or "procedures", and 15 state they used something that "resembles", "adapts" or is "inspired" by GT.

Of the remaining 52 articles that explicitly claim to use GT, four are clearly deviating from GT and can thus not be acknowledged as GT studies.

Detail presentation

Stol et al.'s focus on detail is central from this point on. In this sub-section they focus on the detailing of *how* the GT study was conducted and *which* GT techniques were

mentioned. Of the previously mentioned 52 articles, 18 had no details at all regarding the use of GT or GT techniques, leaving us with 30 articles (after subtracting the four articles that deviated from GT significantly).

Of these 30 articles, 14 presented coding details only, 11 were comprehensive, and a meager five articles were "comprehensive and detailed".

These findings are quite surprising, and shows that most research either lack knowledge about how GT should be performed, or do not appreciate the importance of detailing and presenting their research procedures.

> "Many authors use GT techniques à la carte." [35]

As the above quote states, it seems to be prevalent between authors to pick and choose whatever parts of GT they want to use, and leave out what they deem unnecessary or perhaps tedious and time-consuming.

Grounded Theory variants

In earlier sections of Stol et al.'s article, they go through the three different GT variants, namely Glaser's, Strauss & Corbin, and Charmaz's constructivist GT, and detail the differences between these. They also state how important it is to stick to one of these methods and being aware of the specifics of your chosen method's limitations.

Of the 52 articles claiming to use GT, 39 did *not* specify which GT variant they used (or acknowledged the existence of different versions), and by looking at their citations it was found that 10 cited classic Glaserian GT, 13 cited Strauss & Corbin, 0 cited Charmaz, 13 a combination, and three cited others. Of the 13 that actually specified which GT variant they used, five claimed classical GT, and eight claimed Straussian GT.

Resulting Theory

Finally, Stol et al. focuses on the findings of these articles, and to which extent they present a resulting grounded theory. Here, in my opinion, they are being a bit ambiguous themselves in regards to how they define a "theory" (they do not). It seems like they rely on this quote by Glaser: "[A theory that] accounts for a pattern of behavior". Following this definition of a theory, they claim that only eight plus another one article present findings that constitutes as a theory:

> "Eight articles presented contributions that were clearly cohesive theories consisting of constructs and relationships, while a ninth article presented a set of hypotheses that could be considered a theory."

They continue, stating that the rest of the articles only present graphical representations of a theory, and gives examples such as frameworks, models, factors and categories, but claim that they to not constitute complete theories.

3.4.2. Post-2016 Literature Review

I want to first state that I did not conduct my literature review exactly the same way as Stol et al did. My guidelines for inspecting the articles were inspired by Stol et al. such that we could have comparable results, but as I do not know exactly how they went about their process, I obviously had to make assumptions. This is especially true regarding data extraction and how i collected and categorized the data (more on that in the following sections).

Article selection

As Stol et al. did their article search in the second part of 2015, I saw it fitting to limit my search to articles published in 2016 and later. To facilitate reproducibility, which Kitchenham [18] mentions is important in their set of guidelines for planning, conducting, and reporting a systematic review, and due to time-constraints, I limited my search to one research database. I chose Scopus, and used the same search string: "'Grounded Theory' AND 'Software Engineering'", with the range of 2016 to 2019 (inclusive). This resulted in 91 papers as of 21.01.2019. Approximately the same amount of papers as Stol et al. had. I chose therefore not to limit the search further, and thus included both articles and conference papers. I noted that as conference papers usually have a page limit, they are likely to present less detail, but I still chose to include them as they represented a significant amount of the 91 papers.

Further, I briefly went through each of the 91 papers to weed out the irrelevant ones, such as those that only happened to quote a GT article, without actually having anything to do with GT, as well as papers that fell outside of the domain of Software Engineering. This reduced the amount of relevant papers to 48. Of those 48 papers, two were inaccessible and were thus scrapped, leaving me with the 46 papers that I've used for this literature review.

An overview of this process can be seen in figure 3.2.

Data Extraction and Categorization

I downloaded and organized the 46 papers, and set up a spreadsheet where I documented my findings and relevant characteristics of each paper. At this point I had to make the decision of what kind of categorization I wanted, as Stol et al. did not go into detail. I settled on seven pieces of data in addition to any comments or quotes I found interesting. These were:

- Discipline within Software Engineering (SE)
- Overall rate of GT usage and detail (Fully, moderate, limited, or none)
- Specific GT techniques and details
- Which GT version was used (if specified)
- Type of result presented

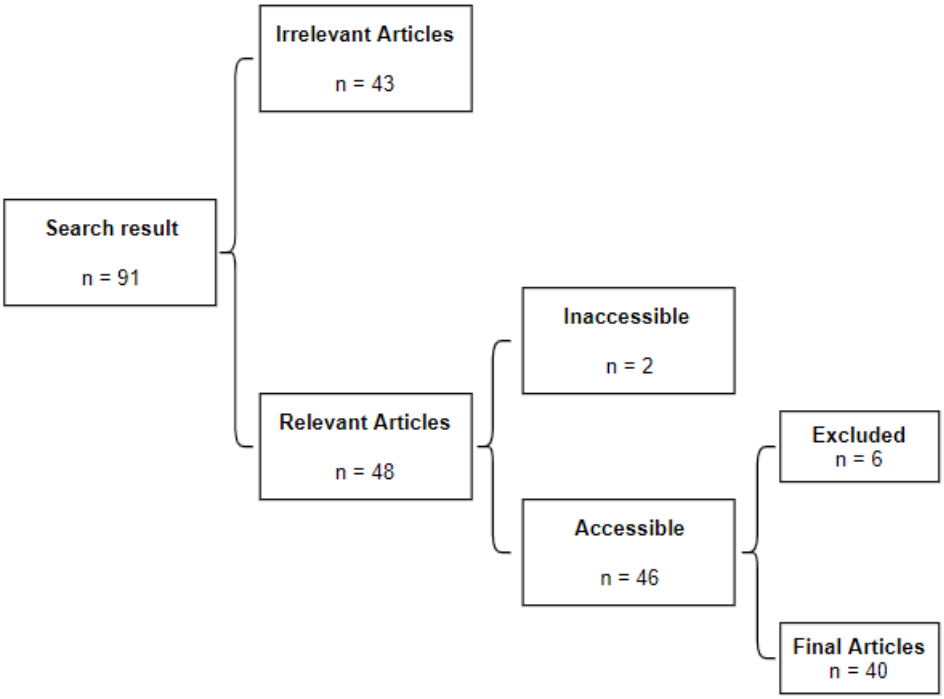

Figure 3.2.: Article Selection Breakdown

- Data Gathering methods
- Whether the study used primary(P) or secondary(S) data

During the review of the papers, I excluded an additional six papers. One being unfinished, one a research proposal that slipped through my initial filtering, two being meta studies that only mentioned GT but otherwise were not GT research, and two being significantly irrelevant. This resulted in the final number of **40** papers.

The most difficult part of this process was that the papers were structured so vastly different, and in addition to this, they had a tendency to mention relevant data in seemingly arbitrary places. Quite a few papers had a good overview in their abstract, but that's where the similarities stopped. Some papers had information in their "introduction", while others did not even mention GT until their "methodology" section. Mention of specific GT variants (if it was specified at all) could be anywhere from the "introduction", to "methodology", to the "results" or even "discussion" chapters. This made it time-consuming to extract the information I was seeking in a methodological way.

Disciplines and types of research

While all of the papers are in the Software Engineering discipline, they vary in terms of what their main focus area is. Some are focused on agile methodologies, self-organization, cross-functional, and development processes (which is highly relevant for me), while

others focus on requirements engineering, team management, testing, secure software development, etc. If it was not specifically mentioned, I noted down what seemed to be the main theme of the paper. See table 3.4 for a breakdown of the 40 papers' disciplines:

Discipline					
Agile	Tools	Management	SE tasks	Requirements	SE research
18	2	5	11	2	2

Table 3.4.: Table of paper disciplines

The discipline names in table 3.4 are aggregations, such that "agile" also encompasses self-organizing teams and focus on SE process improvements, while "SE tasks" can be things like software testing, developing secure systems, software architecture, and technical debt. The results are quite interesting, as 29 out of 40 papers have their main focus on Software Engineering processes, structuring, and tasks. If you include "management" too (which I use as a loose term for everything encompassing SE development, and not just 'leadership'), then it becomes 34 out of 40. This shows a distinct focus on what researchers believe is important to figure out and develop theories about.

Data collection and Data Types

As mentioned in *Article Selection*, I collected information about the articles' way of collecting their data, and from which sources these came from. I also noted whether this data was 'Primary' or 'Secondary'. In short, primary data is data collected by the researcher themselves, while secondary data is either aggregated data from before the research project, or previously gathered/created data such as old retrospective reports. Table 3.5 summarizes.

As we can see in table 3.5, the majority of the papers used data from primary sources, while a small minority used only data from secondary sources or a combination of both. As a caveat, I will mention that the distinction can be a bit blurred, and some papers did not outright state where their data came from (or when it was collected), so it would be logical to conclude that the "Combination" metric is in reality a bit higher, as some papers most likely used some secondary data together with their primary data without explicitly stating so.

As previously stated, I also collected data on what types of data collection methods were used. Here the variety is larger, but there is still a prevalent method, as we will

Data Source		
Primary	Secondary	Combination
33	3	4

Table 3.5.: Table of data sources

see. There is a lack of specificity in several of the papers, for example several papers said they conducted "interviews", without specifying whether they were structured, semi-structured or open. The same goes for "observations", as they should have been defined as "participatory" or "non-participatory" (as some articles correctly did). In addition, most papers used a combination of several data gathering methods, and as such there would be too many combinations to mention in a table. I have therefore generalized them into four categories, as well as created metrics for those papers who used more than one method to give an overall idea of what the norm is.

Data-Gathering Method			
Interview	Observation	Survey	Quantitative
32	18	3	7

Table 3.6.: Table of data gathering methods

As many papers used more than one method, the sum in table 3.6 exceeds 40 (many papers used for example interviews *and* observations).

The category "Survey" consists of two follow-up questionnaires and one survey, while "Quantitative" were things such as collected secondary data, raw data, logs et cetera. All interviews have been aggregated to one metric "Interview", but it should be noted that well over 50% of the papers stated they used semi-structured interviews. Lastly, the "Observation" metric consists of participatory observation, meeting attendance(of all kinds), and workshops. 19 of the 40 papers used more than one method (of those, the vast majority used two methods). Finally, the most common combination of methods were the use of (semi-structured)interviews together with observation.

The following sections will revolve around the Grounded Theory method (GTM)-related findings.

Grounded Theory Variants

Stol et al. focused on the three most widely used GT variants, namely Classical/Glaserian, Straussian (& Corbin), and Charmaz's constructivist GT, and thus I did the same. Surprisingly many stated which method they followed, but it should be noted that not all of those who did, acknowledged that there were several variants to choose from. This could imply that they merely chose the one they had heard of, without having given the topic any significant thought. A few papers stated they followed "Straussian" while one said only "Corbin", but I have generalized them into the same metric, as most mentioned both Strauss and Corbin. Interestingly, there were five papers that stated they used Charmaz's constructivist GT, which is an obvious increase from the amount that used that method in Stol et al.'s review (zero).

There is sadly still (compared to Stol et al.) a significant portion of research papers that do not even mention or recognize different GT variants.

Grounded Theory Variant			
Glaserian	Straussian	Constructivist	Not mentioned
10	11	5	14

Table 3.7.: Table of Grounded Theory Variants

Grounded Theory Details and Techniques

In this section I will present how prevalent the use of the different Grounded Theory techniques and methods were in the 40 papers. As previously mentioned, it was quite challenging at times to collect all the information contained in the papers, as they mentioned their usage of GT techniques in complete arbitrary sections. Commonly, I found a significant part of their methodology in their "Data-analysis"-sections, but additional information could be spread across the contents of the paper.

This is where the variance between the GT usage becomes most apparent, so I had to make sure that I did this part especially thorough. It was quite common for papers, especially those who only claimed to be "influenced by grounded theory" or said "we used techniques from grounded theory", to sprinkle their text with *theoretical saturation* or *theoretical sampling* without giving any more information. In addition, there seemed to be no limit to the combinations of different techniques. Some papers only used "coding" (with no mention of what kind of coding, or any outlining of their process) together with mentioning *constant comparison*, while others mentioned *open coding*, *memoing* and *saturation*. As a consequence of this, i will present in a table all the main GT techniques, together with a representation of how many papers utilized each of these methods (similar to how Stol et al. presented their data) in table 3.8.

Grounded Theory Technique Usage	
Iterative data collection and analysis	16
Theoretical Sampling	15
Coding	40
Memoing	18
Constant Comparison	30
Theoretical Saturation	16

Table 3.8.: Table of Grounded Theory technique usage

As a disclaimer, there are certain factors that were difficult to record with accuracy, such as the iterative nature of their data collection and analysis. Many papers did not mention this, and if they did, they did not necessarily use the word "iterative". Additionally, as I read the papers and did not use any "search" functions, there may have been mentions of techniques that were missed by me. As such, the numbers should be used relative to each other - not as absolute values.

Most papers who only used the bare minimum of GT typically stated they used *coding*

and *constant comparison*, which is why those metrics are significantly higher than the rest. A lot of papers seemed to be of the impression that only one iteration of data collection and analysis was enough, or they did not bother to mention it, which we can see with the low amount of mentions of theoretical sampling and any kind of iterative description of methodology.

Result and Theory Presentation

Stol et al. spent this section of their paper talking about the lack of a "theory" in most papers and only came with brief generalized examples of what the majority of the papers had presented. I will try to refrain from judging whether or not the papers in this review created a "theory" as I do not have the details for how Stol et al. defines theory, nor what the appropriate level of detail and structure is. Instead I will present the different ways the papers presenting their findings, and let the reader judge whether or not this constitutes a theory.

There were quite a lot of variation between the papers. Some presented their results as GT-typical models such as Glaser's Six C's, while the majority presented different categorizations, relationships, frameworks, and models. Results such as: "5 main dimensions of becoming agile", were very much used, whether they used 4, 5, or six as their numbers, and "dimensions", "categories", "taxonomies", or "core factors" as their structuring. These results must be inspected individually to decide whether or not they classify as a theory (and you would probably get different answers depending on who you ask). What I *can* say is that the level of detail is significantly different between the papers. Those that had a detailed description of their GT methodology tended to also have a GT-centered detailing of their results. This seems to correlate with what Stol et al. said; "We observe that studies that produce a 'set of themes' rather than a theory tend only to borrow discrete practices from GT—what we call grounded theory à la carte".

Overall Rate of Grounded Theory usage

After going through each paper, I gave them a score depending on my impression of their use of Grounded Theory, their details, and their results. As i mentioned in 3.4.2, these were "Full", "moderate", "limited", and "none". The "none" classification was given to the six papers that i filtered out from my original 46 papers(leaving me with the 40 papers I used in this review). I followed these general guidelines to arrive at a conclusion:

- **Full:** The paper must mention and detail to a certain degree all of the central Grounded Theory techniques, and detail the process and results in the context of the Grounded Theory Method.
- **Moderate:** Most of the Grounded Theory techniques must be mentioned, and preferably somewhat detailed. Papers that used more than the bare minimum of GT but did not fully utilize it.

- **Limited:** Papers who used the bare minimum of GT, mostly just mentioning coding and perhaps constant comparison, including little detail, and who seemed to only "borrow" from GT.

Paper GT Categorization	
Full	14
Moderate	18
Limited	8

Table 3.9.: Table of Grounded Theory paper categorization

I could probably have split the categories further, but at some point it becomes difficult to separate the papers from each other into a meaningful structuring. Table 3.9 shows the categorizations. That being said, there were some articles within each categorization that were significantly better than the rest, with more detail of methodology, research philosophy, and reasoning behind their actions.

Comparison with Stol et al. (2016) & Final Reflections

Stol et al. focused mostly on the specifics of Grounded Theory, the extent to which researchers follow precise methodological guidelines, and ultimately whether or not they use Grounded Theory or just claim to. They found that the vast majority of research papers included insufficient detail about their use of Grounded Theory to the degree that Stol et al. claim that they're not using Grounded Theory at all. This "method slurring" seems to go deep, as I encountered it time and time again during my own literature review. I found that most papers that passed my initial inspection only borrowed from, or used parts of Grounded Theory. Statements such as "using methods based on Grounded Theory" or similar were prevalent. This problem was made even more difficult as many papers seem to leave out important detail on methodology and how they performed GT in practical terms. Whether it was due to space-constraints or not is unknown. Stol et al. gave only five out of 52 articles the classification of being both "comprehensive and detailed", while 11 of 52 were "detailed" (totaling 16 papers). I found similar results, where 14 of 46 papers got the classification of using "full" GT, but as I said in the previous section, this category could probably have been split once more, leaving us with the approximately five or so top papers that stood out above the rest when it came to detail and completeness.

Due to Stol et al.'s little detail and description regarding what their classifications meant, I decided to simplify it and made my own classifications, as mentioned in detail in sub-section *Overall rate of Grounded Theory usage* on the previous page. Table 3.10 shows a comparison between Stol et al.'s findings, and my findings. The classifications can be compared horizontally, for example the "Comprehensive and/or detailed" classification is comparable to the "full" classification in my review.

The creation of theory is a difficult factor to assess. As i mentioned towards the end of section 3.4.1, Stol et al. were not specific in their definition of a theory. As

Stol et al. vs Post-2016			
Stol et al.		**Post-2016**	
Comprehensive/Detailed	16	Full	14
Coding Details	14	Moderate	18
No Details	18	Limited	8
Deviating	4	None	6
Total	**52**	**Total**	**46**

Table 3.10.: Comparison between the two literature reviews' results

such, it is difficult to arrive at comparable results regarding theory creation, as I do not know their definitions. However, I did incorporate it in my findings, in the sense that my impression of the article's detail, depth and relationship amongst their main categories indicates their success at creating a theory. Following my definitions of my classifications, an article would not be classified as using "full" GT if they did not arrive at a decently detailed theory. That being said, there was a large discrepancy between the articles, where some had much more thorough theories than the others. At this point it becomes quite arbitrary to draw a line for what constitutes a proper theory or not, but to present an educated guess, approximately 5-7 of the 14 articles with the "full" classification could be said to have a satisfactory theory creation and presentation. These articles had a significant amount of detail, they had presented logical and realistic relationships between the categories, and they had backed them up with explanations grounded in empirical data such as quotes and metrics.

As a reminder, my own literature review consisted of not only research articles, but also conference papers, and as I wrote in 3.4.2, conference papers tend to have a page limit which means their likelihood of excluding details is higher. This can negatively skew some of the results compared to Stol et al.

For my own research purposes (in this thesis), this is very interesting, and has truly given me a deeper insight of how to plan, conduct, and present a grounded theory study. Being aware of what grounded theory variant I am using, making sure I follow the GT techniques correctly, and how to present a detailed description of the study is something that must be kept in mind throughout the entire research process.

4. Research Context

In order to get a sufficient understanding of the results of this thesis, it is important to be aware of the research context this thesis took place in. In this chapter I will expand on the information from the introduction, where I briefly mentioned the research context. I will detail the case description, the organization where the research took place, team-structure, roles, relationships, and work-flows to give an elaborate description that will enable the reader of this thesis to fully follow and understand the results.

4.1. Organization

The research was conducted at a major Norwegian bank in the winter and spring of 2019. The bank's aims are to serve both the private market (PM), and the business market (BM) through their 14 "development teams". These development teams are cross-functional, and consists of all the major roles within Software Engineering (which I will go through later in this chapter). The development teams have a mixed focus of management and administration of current banking services, and creation of new products. The ratio between these two are certainly mixed between the different teams. Some may have a main focus on one or the other, while other teams have a more 50/50 partitioning. Each team is in charge of a certain domain within PM or BM, consisting of a certain amount of "applications" that they are responsible for. Whenever dependencies between these change, or updates are made, the teams have to make sure they are coordinated.

The teams coordinate through different channels such as meetings for the team-leaders, meetings for the teams' tech-leaders, and meetings between the teams' product owners (PO's). For broader and more business-centered topics, there are additional meetings. In addition, for more informal knowledge sharing there are both scrum-of-scrum meetings and inter-team demo's at regular intervals. Messaging software such as HipChat and Slack are significantly used, both for intra-team communication, inter-team communication, and personal conversations.

The teams have common resources such as API's, test-environments etc. In addition, they have a common link in "operations" that deal with all of the sub-systems, core-systems etc. The teams are shielded from most of the customer relations as there is a separate department dealing with those issues. The teams do however get input from users through feedback-systems in their products, if relevant to their work (such as feedback-boxes on the webpages they are responsible for, and thumbs up/down).

4.2. The Business Market Teams

The Business Market (BM) teams are my focus for this thesis. Of the 14 previously mentioned teams, the vast majority of them have PM as their domain, whereas there are only 2 BM-teams. There is an additional third team that works on BM, but they do not closely work with the other two BM teams. I will from now on refer to the two teams as "BM1" and "BM2". BM1 and BM2 have, as I previously noted, a selection of applications that they are each in charge of, as well as a handful of applications that they share. These applications often have to communicate with each-other, and as such the teams are quite coordinated and well informed regarding each others' current work.

The bank as a whole (PM and BM) have for the past several years been restructuring their core architecture from a monolithic system to a modular one consisting of the aforementioned 'applications'. This is a process that PM has finished, while BM is in the final stages of. I was part of the BM-teams during their final months of phasing out this monolithic system, called the "portal". At the time I joined the teams in mid January, they had only a handful of use-cases left to implement in order to be completely independent of the 'portal'. Being part of this final stretch of their project gave me an expanded insight into their processes, how they work, and what can be improved on as it was a quite busy, and at times stressful, period for the teams. That being said, it was a very successful period with positive feedback from the bank, and their customers, as well as from upper management. This has a lot to do with well established and successful agile methodologies, communication frameworks, and knowledge sharing methods that I will elaborate on in this thesis.

BM1 and BM2 have quite similar processes and follow similar SE (Software Engineering) methodologies. They are both following an adaption of agile development practices, influenced by both Scrum and Kanban. They follow a Kanban-like "task-boxing" methodology, rather than a Scrum-like "time-boxing" methodology. This means that the teams have a backlog, consisting of a set of backlog items (BI), that are prioritized and assigned to the appropriate team-member. They have a backlog-grooming meeting every week where the full team is in attendance. There they decide on which BI-proposals should be accepted, which priority/severity the BI has, and who is in charge of dealing with it. Such BIs can range from being bugs that have to be fixed, new features to be developed, features to be tested, or administrative things that Team-lead (TL) or Product Owner (PO) should handle. Furthermore, the teams have a daily Standup-meeting at 9:15 that all team-members have to attend. Both of these meetings are separate for BM1 and BM2, however they were experimenting with an additional cross-team meeting for the developers where discussion and coordination between the two teams could be facilitated during my time there. A similar cross-team meeting was held for the testers. The team-leaders were present at both. A more in-depth detailing of what went on in these meetings can be found in my chapter on Data Collection in 3.2.

4.3. Team BM 1

At the time of my arrival in mid January, I was given a spot facing my 'mentor', and next to another developer, in the middle of the office landscape of BM1. This naturally meant that I ended up with a more detailed focus on BM1 than on BM2, as I was seated there. See Figure 4.1 for a graphical representation of the office-landscape. An additional "island" of 4 desks were situated right behind me (the "Observer" in 4.1), where both teams' designers were seated (both graphical and interaction designers).

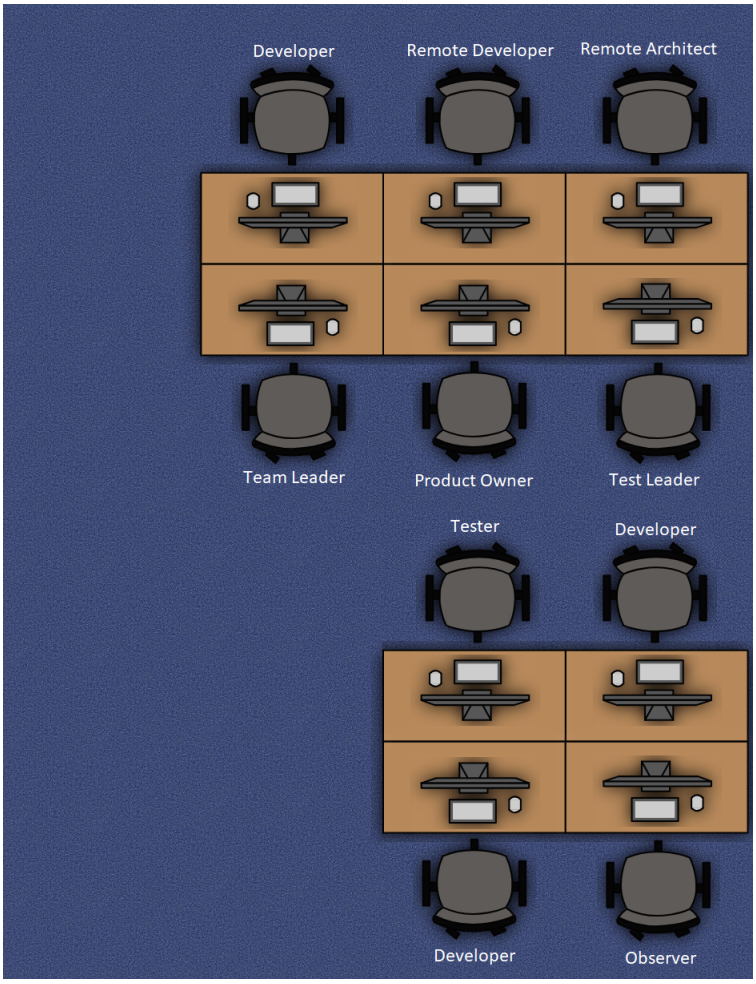

Figure 4.1.: Seating arrangement BM Team 1

BM1 is a cross-functional team (like most of the other teams) consisting of 5 developers, 2 designers, 1 tester, 1 test-lead, 1 team-lead and 1 product owner. In addition there is another developer who also has the role of the architect. The team allows for a lot of flexibility when it comes to working situations. It is very common to work from home every now and then, and in the case of family issues or illness it's never a

problem to report that the person will stay at home. One of the developers, and the developer/architect work from an office in a different city four days of the week, while once a week they travel to the office like the rest. This does not seem to impact the team negatively as the teams' workflows and meetings allow for distance. Team members who work remotely always attend meetings through online programs such as "appear.in" or Skype.

4.3.1. Workflow

The team's overall project is to phase out a monolithic system by replacing it with individual, modular, applications. The project is split into "use-cases" which represent a certain main functionality that the bank must offer to its customers. These use-cases are quite a bit larger and more complex than what one might think of when hearing "use-cases", and can be compared to deliverables or milestones regarding a certain system functionality. The team focuses on one of these use-cases at a time, unless there are multiple use-cases that are related in a way that they have to be developed in parallel. The use-cases are split into requirements that have to mirror the functionality of the monolithic system, while being made in a modular structure. Throughout the development, the team has a weekly meeting named the "backlog-grooming" meeting, where they form, decide on, assign, and prioritize backlog-items. Once a developer has finished a feature/bug-fix, and it has passed the review process (usually by another developer on the team), it goes to the testers who perform specific tests, automated tests, and system regression tests to ensure that the quality is up to standard. An amount of related features and bug-fixes are regularly grouped together, and released to the production environment. This release-process is very standardized, and goes through multiple stages of external validation and testing before approval. A use-case case has always many such released, and the process is repeated until all functionality from the monolithic system has been mirrored by the new modular applications. At that point, the team changes their focus to a new use-case until the project is completed.

4.4. Team BM 2

BM2 is the second team that I observed and interacted with during my research process and thesis work. They amounted to approximately 20% of my time and focus, mostly consisting of weekly progress meetings and cross-team meetings. BM2 is quite a bit larger than BM1. Their seating arrangement can be seen in figure 4.2. One interaction designer sits on island #1, with the rest of #1 and #2 being filled with developers and testers. The Team-leader and Product Owner sits at #3, while #4 also consists of a mix of testers and developers.

BM2 has in total 8 developers, 2 testers, 3 designers, 1 team-lead and 1 product owner. The test-lead from BM1 is a shared resource for BM2 as well, but is seated with BM1. The two teams sit adjacent to each-other, so any cross-team communication is easily achieved. Taking the figure of BM1 from 4.1, inserting a 4-person island for the designers, and placing it above the figure of BM2 from 4.2 would give the full picture of

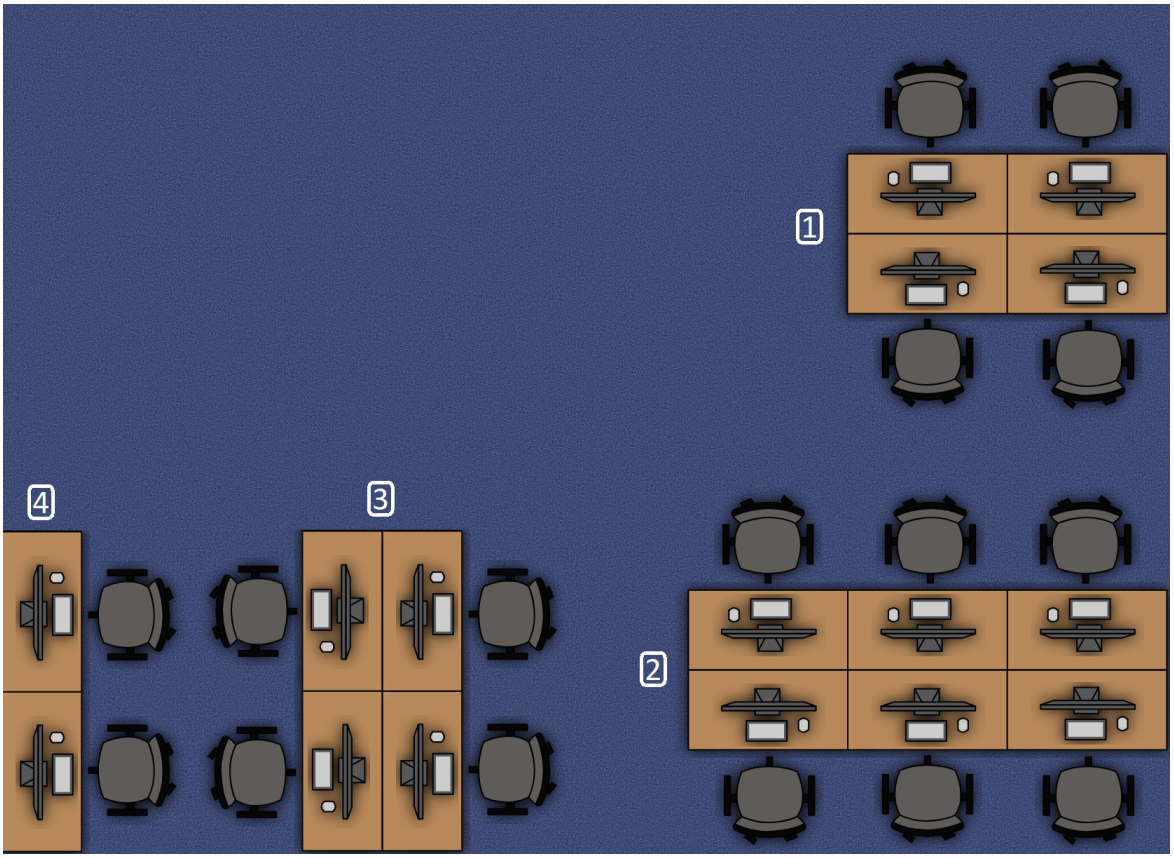

Figure 4.2.: Seating arrangement BM Team 2

the work-environment that the two teams are in. There is an additional island to the left of the teams where 2-4 people from external partners are seated, who are available as shared resources for both teams.

BM2 uses most, if not all, of the same processes, meetings, and work-flows as BM1 - see 4.3.

4.5. Team Member Roles

In this section I will give a brief description of the roles that can be found in the two teams. I will introduce their main work, who they interact with, and what their contributions are to the process as a whole. Additionally, I will include a couple roles that are outside of the teams themselves, but are still very much involved with the projects.

Development Leader

A role I haven't yet mentioned is the development leader. The 14 teams are split into departments based on similarities, and these departments have a leader named the development leader. This persons job is to oversee the projects that are ongoing in his department, and serves as a joint between the teams and the upper management.

Project Leader

Each project has a project leader who is external to the teams. Their role is to oversee the project, and work closely with the team leader, and the product owner to ensure that the team is developing the project satisfactory to the bank's wishes.

Team Leader

The team leader is responsible for the team, its team members, calling in meetings, being in contact with and coordinated with the other teams and the development leader.

Product Owner

The product owner role is the person who is in charge of the product, and hence the backlog. It is his responsibility that the correct features are developed, and that the product is in line with the requirements of the bank. During meetings such as the backlog-grooming meeting, he is responsible for knowing which backlog-items should be accepted, and then to prioritize their severity or importance.

Test-Lead

The test-lead is in charge of ensuring that releases are thoroughly tested and ready for QA and production. She often have to coordinate with other development teams, sysadmin, or other parts of the organization. The test-lead for this project was responsible for both BM1 and BM2. In addition to having the responsibility for this overview, she doubles as a regular tester.

Tech-Lead

The tech-lead is a senior developer with extensive technical domain knowledge. He attends weekly tech-lead meetings between all development teams where coordination is a central topic. While most teams work independently of each other, they often share underlying technologies or systems. In the case of updates or changes to these technologies or systems, it is important that all teams are "in the loop", in order to prevent any surprises or negative side-effects. Additionally, the tech-leads are expected to be up to date on new and exciting technologies, which can be brought to attention to the other teams for potential usage. The tech-lead works as a regular developer for the team for the vast majority of his time.

Developer

A developer is a software engineer that develops the features that are required for the project. They work off of Jira boards where their tasks (backlog-items) are listed and prioritized. Their job is to ensure that the right features are developed, and that they are implemented with as high quality as possible to make the testers' lives easier. Teams mostly have both backend developers and frontend developers.

Tester

A tester tests what the developers have developed. This can range from regular to critical bugs, or regular product features. Their job is to both run manual tests on the developed features, and automated tests. They use test environments that mimic the functionality of their production system, using test-data to emulate real users. They also run regression tests, which are tests that ensure that old functionality still works after new features have been implemented.

Interaction Designer

The interaction designer is a designer who has the main focus on UX (user experience). They are very aware of what the users want, how they use the systems, and small changes to layout can be done to improve the user experience. They spend their time both theorizing together about different ways of interacting with the system, implementing AB-tests, and doing field interviews and surveys with people to get a feel for which solution is the best one.

Graphic Designer

Graphic designers work closely with the interaction designers, but have their main focus on the presentation, and to a certain extent the layout, of the systems and applications. They use similar methods to the interaction designers to arrive at conclusions about product design.

4.6. Team Cooperation

Team BM1 and BM2 were both involved in the project of transitioning from a monolithic system structure to a modernized, modular system. As I previously mentioned in this chapter, both teams are responsible for a certain amount of "applications". In addition to these applications, the two teams share the responsibility of administering a couple applications related to their mobile banking solution. When making changes or adding features to applications that may interact with applications that are outside the responsibility of the team, it is important that they coordinate with the other team to prevent issues. To do this, the teams have different methods of communication and visualization, that I will briefly go through in this section.

4.6.1. Meetings

As mentioned in 4.2, the teams started experimenting with a cross-team meeting for developers. This meeting was held every other week, and resulted in a lot of discussion regarding common resources, and technologies that impacted several people. Discussions surrounding how to further develop and use certain systems and technologies arose, and many people were involved in the discussions. The meetings had a quite open format, but everyone got a chance to bring up topics that either bothered them, or they simply wanted to discuss with like-minded people.

Additional cross-team meetings were held whenever necessary. Such meetings could for example be related to the projects during startup, and towards the end, regarding topics that both teams had to be up to date on. Ad hoc meetings between developers or testers happened a few times, but it seemed like most of the communication needs of the teams were handled through face-to-face discussions or over a cup of coffee by the sofas.

4.6.2. Chat Applications

The two teams use two chat applications for inter-team communication, namely HipChat and Slack. They use these two applications for different purposes, as we will see in the following sub-section.

HipChat

HipChat is the main chatting application at the bank, and consequently at the BM teams too. The teams have established two chat rooms, that serve different purposes. The first, main, chat room is the general chat room for communication. It is has a few main purposes:

- **Coordination:** Whenever the teams are releasing a new version of a selection of their applications, and this new version may impact the other team, people often type in the chat room to keep the other team updated. This is especially true when both teams are deploying new versions.
- **Knowledge Sharing:** Whenever someone encounters a new solution, a new technology, or have some new information in general that might benefit members of both teams, it is posted in the chat room. This is usually done quite informally, with the intention of reaching as many relevant people as possible with this new/cool/relevant information.
- **Questions:** Whenever people have questions that are either aimed at the other team, or to both teams, it is posted in the chat room. Such questions can be related to the source-code, test environments, test data, underlying systems, contacts (internal or external), or other relevant questions. Questions almost always get answered, which allows everyone to see and learn from it, as another type of knowledge sharing.

- **Direct Messaging** is also used when specific knowledge is required or needed. People tend to learn, with experience, who in the organization has the required knowledge or competence, which simplifies the communication, and leads to less disturbances as you do not need to interrupt people who are not related to the specific conversation or topic of discussion.

The other chat room is less used, as it is more specific to solving problems that have come from the bank's customers (customers who encounter problems or bugs can report this to the central team that deals with such incidents. This team, which serves (and shields) all the cross-functional development teams, delegate incident reports to the relevant teams based on what systems and applications the feedback is related to). Whenever such an incident is reported to the BM teams, and it is relevant to both of them (or it is ambiguous who should fix it), the teams post and discuss the incident in this second chat room. The chat room also has a bot that automatically posts a message in the chat room, at the beginning of each day, detailing who is responsible to deal with incoming incident reports that day.

Slack

Slack is used to a significantly lesser extent than HipChat. Where HipChat is the main tool for chat communication, Slack is used primarily for messages sent when not at work. This means that people who are delayed for work, get sick, or have any similar relevant information to the teams during off-hours, post it on Slack. It has been mentioned that the bank might switch from HipChat to Slack, as Atlassian (the company who supplies Jira and Confluence) might become compatible with Slack.

4.6.3. Face-to-face Dialogue

The third and final method of cross-team communication is regular face-to-face dialogue. As the teams are situated right next to each other, see 4.3 and 4.4, it is trivial for anyone who has a specific question, or would like to discuss something, to walk over to the relevant person and have a chat. This usually happens several times a day, and the discussions are mostly quite brief. Often it allows for other people to join the discussion if they have any relevant input, and it helps to keep people up to date on what is going on, and what is potentially problematic. From time to time, an issue reveals itself to be more severe than expected, and the people involved can go to a nearby meeting room for a quick ad hoc meeting.

5. Results

Grounded Theory has two meanings. One is in relation to the qualitative research method, and the other meaning is the emerging theory which is grounded in data - in other words the results of applying grounded theory method. The grounded theory which will be presented in this chapter aims to answer the following research question:

RQ1: Which characteristics are essential for efficient and successful software development teams?

The teams observed in this thesis are however not just regular teams, but teams built with cross-functionality and team autonomy in mind. Therefore an additional focus was central for this thesis, which can be summarized in *RQ2*:

RQ2: How does cross-functionality and team autonomy impact the efficiency of software development teams?

The theory is built around a main category, with related central categories which impact the main category. These central categories are again built up of specific topics, structured into sub-categories. This 3-level hierarchy, combined with the relationships between the categories, will be presented in this chapter.

5.1. Overview of Theory

Settling on the main category was central, as it is very related to the scope of the thesis. In my mind, during data gathering and analysis, there were two options for the scope of this thesis. I would either figure out something very specific that I would dig deep into, such as a specific part of software engineering or a specific part of the development teams' processes, or I could look at the software development teams as a whole, focusing on all aspects of them. As the process progressed, it became clearer and clearer that I wanted the broad scope of the teams as a whole - focusing on what makes them successful. However, the word "successful" can mean different things depending on the team, and as such I settled on *Team Efficiency* as my main category. Furthermore, I had a lot of potential topics which emerged through data analysis which I wanted to include, and some (still interesting) topics that I felt did not fit into the theory as a whole. I touched on this in *Chapter 3.3 - Data Analysis*. Through trial and error, I pieced together the topics (there were 29 in total) into six central categories (or concepts if you will). This resulted in the hierarchy, which can be seen in figure 5.1. Below each of the six central categories are the aforementioned topics/sub-categories. In the following sections

I will detail in-depth the central categories, and especially their underlying topics/sub-categories. In section 5.8 on *Category Relationships* I will go into more detail on how these central categories relate and influence each other and the main category *Team Efficiency.*

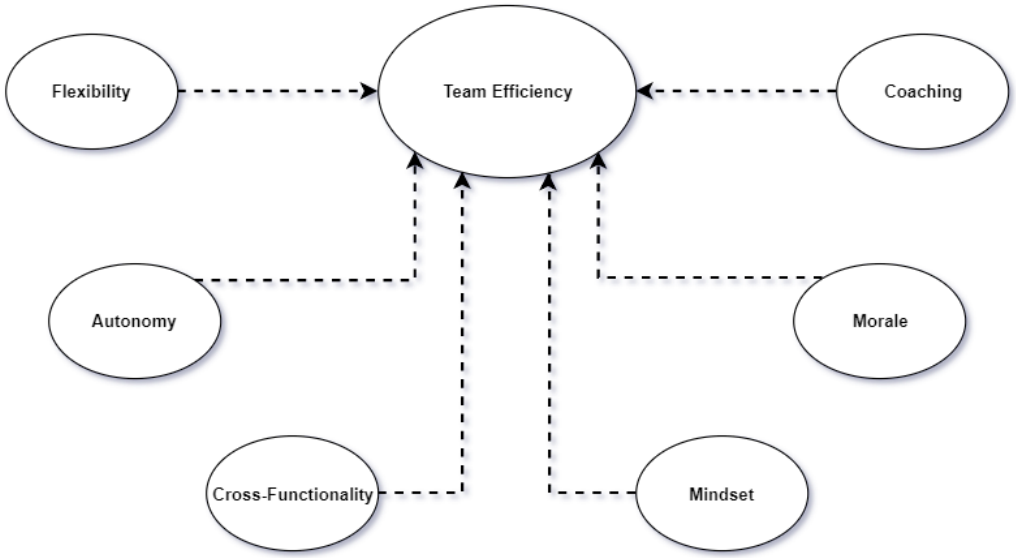

Figure 5.1.: Category Hierarchy

5.2. Flexibility

Flexibility emerged as a large category, highly linked to most facets of software development teams. The related sub-categories which I will detail revolves around how the teams foster this flexibility and use it to their advantage.

5.2.1. Freedom to Prioritize

One of the first things I noticed when becoming a part of this team, is how the task-allocation works. I will split this into two sub-categories; *Flexible Work* and *Responsibility Assignment* (see sub-section below). This sub-category will be about the freedom that each person has in regards to what they will work on, and how they will perform this work. As previously mentioned, the team works with a Kanban-board, structured into different lanes. Two of those are the complete "backlog" of things to do, and the much smaller "ready for development". The "ready for development" consists of approximately 15 tasks, where most of them are assigned (see *responsibility assignment*) to team-members of different roles. The designers have their own backlog. From this point on in the process, the team-members can pick and choose whichever tasks they wish to work on from this "ready for development" section, and sometimes also from

the large "backlog" section. Most of the time, there is no restrictions or orders made by the team-leader or product owner, so from here on the team-members are free to choose what they want to work on.

> *[...] Everyone is responsible for themselves. We don't have people who just wait for orders. So we look at the board, and think to ourselves "hmm I can take this one", and then we just take it and work on it.* - Developer

Only if there is a more important task, or an emergency, the product owner will chime in and ask if the developer can finish the more urgent task first.

One potential issue with such a process, is that if you are constantly only working on the tasks with the highest priority, the lower priority tasks will stay forever at the bottom of the "pile". I brought this up to a developer during an interview, and he said:

> *Usually the lower priority tasks are weekly and monthly tasks that are reoccurring. I usually spend Monday morning working on these, getting them out of the way [...]. When it comes to other low priority tasks, if they are left too long in the "backlog", we typically take it out of the backlog and re-prioritize it.* - Developer

Thus, this issue, if people are aware of it, will not become a problem for the team or the customer. As another data-point on this issue, I observed this exact procedure multiple times during the backlog-grooming meetings, where the team would stumble upon an old backlog-item from perhaps six months ago. They would then discuss whether the long wait-time of the task would increase its priority or not. In one of the following sub-sections, on *responsibility assignment*, I will go into more detail on the process that takes place before tasks are prioritized, assigned, and developed.

5.2.2. Responsibility Assignment

The sub-category *Freedom to Prioritize* (see above) comes as a consequence of this sub-category, *Responsibility Assignment*. The relationship between the two can be regarded as follows: Freedom to prioritize can only happen when the team has trust in each other, and when the upper management has trust in their teams to be smart and make good decisions through autonomy. In order to achieve this, the teams must be responsible, and the team-members themselves must take responsibilities seriously. This sub-category is about what actions and measures are taken to give responsibility to the team-members, and how this is done in a flexible manner through consideration of both availability and expertise.

While the teams strive to be autonomous, there are some restriction regarding their products, mostly in the form of *what* to make. This is supplied by the product owner, who works in tandem with product owners of the other teams, as well as being a close link to the customer. She informs the teams about what needs to be done, and what overarching goals should be set based on what the customer wants. This information is broken down especially during the backlog-grooming meeting. There tasks are broken

down, prioritized, and assigned. Most tasks that go to the "ready for development" section of the backlog are assigned to developers. Which developer is assigned to which task depends on the two things i mentioned earlier, availability and expertise. If a task falls within a category that a certain team-member has a lot of previous experience with, then it is natural that that person is assigned to the task. In other situations, only a couple developers may be available, as the rest have too much to do already, and in that case the available developers are assigned to the tasks. After this assignment and prioritization process, the developers are free to choose which tasks they want to work on, and in which order, and how they would like to implement it (within reason). This planning and structuring together before the freedom and flexibility of choosing takes place, seem to be a solid foundation.

> *Autonomy is probably what best describes us. Because me or other developers are not waiting for someone to tell us what to do. But it has to also be facilitated by the team-leader and product owner doing their jobs, making things available by prioritizing. We are not supposed to choose amongst everything, but we choose based on their prioritization.* - Developer

There is also a separate task that should be mentioned, which is *deployment*. As a result of their changes, the deployment-procedure can now be performed by anyone in the team. This is a huge improvement, as it used to be a very rigid process.

> *[On the topic of deploying] - I have written a script that makes it so anyone can do it. In one of the teams they rotate the responsibility, while the other team just has people doing it for a while until they get tired of it, and then someone else takes over.* - Developer

Making complex processes and tasks doable for everyone increases the autonomy that the team has by everyone being able to take responsibility and doing things that needs to be done, instead of relying on one or two people with the specific expertise to do it forever.

5.2.3. Flexibility in Release Schedule

A specific example of flexibility is during their release schedule. The teams have quite a structured and defined release-process which allows the teams to perform this in a very streamlined fashion. However, sometimes things happen resulting in a delay of one of the teams *applications* or similar issues, where the teams need some flexibility. The release process allows for this. Although it is very structured, it is flexible enough for things to be delayed for a day or two without creating problems for the release. So even in one of the most rigid/structured parts of the teams' processes, they realize that it is important to allow for some leeway and flexibility. This to me, says something about the mindset that the teams have regarding flexibility. They realize that things do not always go as planned, and that they must be flexible and able to make changes within a short amount of time. Whether this is an error correction that must quickly be pushed

to production, or a less serious matter, the mindset of the teams are calibrated in such a way that they are ready to change their processes at a moment's notice, whenever needed.

5.2.4. Communication Types

The teams use different communication types depending on the situation, who they want to reach, and the amount of formality that is warranted. One of the key things that were mentioned by team-members during interviews were the importance of having separate means of communication, and they talked at length about the upsides of each method. For example, on the topic of who to contact and through which means, a developer answered:

> *This is what's called tacit knowledge. Through experience you will learn who has the competence and knowledge that you seek, and from this experience you can use the appropriate communication tool.* - Developer

The main means of communication that the observed teams use were outline in section 4.6 about Team Cooperation, but the key question is how the team-members know which method to use for which purpose. That is what separates efficient from inefficient communication.

Emails are usually seen as a more formal way of communication, and might appear preferential when dealing with persons who are higher up in the hierarchy. However, using emails have drawbacks. There seems to be a higher threshold for answering emails, than for example chat applications. This may lead to a simple email with a simple question going unanswered for a day or two, whereas the same question could be answered immediately through chat. As one developer noted when comparing his situation to a previous job he had:

> *It was for the most part email. It took a lot of time, especially with weekends. Very ineffective, so it is a lot better here.* - Developer

As a result of this, the teams and the leaders above in the hierarchy have worked towards lowering the threshold for communication by having a flat organizational structure (as outlined in section 5.3.1). This also have positive implications for inter-team communication which is outlined in 5.3.5.

5.2.5. Coordination Mechanisms

Coordination takes place in many forms in different arenas throughout software development, and this sub-category represents these various methods. Coordination is central for how the teams perform software development, and having processes in place seems to greatly enhance and streamline the teams' performance. The first, and possibly the most crucial coordination mechanic, is **meetings**. Meetings can come in many shapes and forms, and thus provide the teams with a flexible format for coordination that can

be molded into serving the purpose which is most useful for the teams. One of the central meetings for cooperation is the *Backlog-Grooming*. This meeting takes place once a week, for an hour, with the goal being coordination of tasks. As previously noted in sections of this thesis, the teams use a variation of Kanban, focusing on tasks-boxing rather than time-boxing. As new tasks, *backlog items (BI)*, emerges, they are added to the "inbox" of the Kanban board on Jira. The main focus of the backlog-grooming is to go through all new BIs, further specify them if necessary, then prioritize them, and assigning them to the appropriate team-members. Lastly the BIs are either put in the all-encompassing backlog, or it goes straight to the backlog-section named "ready for development". Through this process, which most members of the team are a part of, the team-members become acquainted with the systems, informed of what people are working on, and coordinated on related tasks. From an observational point of view, the meetings are very productive, and this is supported by several team-members during interviews - for example:

> *Yes, I find the backlog-groomings productive. Some might find them a bit boring, but for me it's nice to get an overview over what people are working on. Similar to the [longer] standups on Mondays.* - Developer

> *I find the backlog-grooming quite nice. Coordination and information about what gets prioritized, what PO thinks, what we can skip, and what is important.* - Test-Lead

The results suggests that the key to a successful coordination meeting is that it is a dialogue between the team-members who might not talk as much the rest of the day. Through observation, it is clear that a lot of valuable discussion takes place. Whether it is about problem-solving a specific task, talking about the history of a bug/problem, or planning and prioritizing what different people think is the most important.

As meetings takes up a lot of time, they cannot serve as the sole mechanic for coordination. Various other processes and practices have been implemented to supplement. To facilitate coordination the teams use various software, but at the core of these processes are *Jira* and *Confluence*. Jira has already been mentioned, and serves as the teams' primary coordination mechanic for the specific tasks and items to be developed or completed. Whereas Confluence is used for documentation, planning, and as a wiki-like platform where teams and team-members can create pages for any objective. The need for coordination is not just on the task-by-task level but also in the broader sense of the project. To achieve this, the teams create **timelines** that serve as a broad overview over the project as a whole, the major tasks to be completed as milestones, and who are assigned to what part of the development. These timelines span several months, and are meant to be used for planning, as well as for having a common sense of where the project is headed.

Thirdly, the teams have **code-reviews**. Code-reviews allow developers and testers to review each other's work. This is an important step in the development process, as it weeds out a lot of mistakes before testing start. But possibly more important, it allows

team-members to become acquainted with and understand the code base that they have not written themselves. Being able to look at code with a fresh pair of eyes often reveal implementation mistakes such as architectural anomalies that could lead to problems down the line. Preventing this technical debt is very efficient in the long run. However, as one developer noted, code reviews can have some negative consequences:

> *Sometimes it can get tiring because it can lead to a lot of back and forth discussion, which takes time, and can be detrimental for a fix that is time-sensitive [...]. It can create some tension [correcting other people's work]. Not everyone is comfortable with it.* - Developer

5.2.6. Working Remote

This sub-category has implicit relationships with several other sub-categories, but will talk about it from the perspective of flexibility. There are two ways in which working remote allows for flexibility for the teams and team-members, and they differ only in the time-scale. The first is that several team-members of the teams have a permanent solution where they work remote three to four days a week, with the final day or two being at the office on-site with the rest of the team. This is a quite interesting situation, as it is something in between regular on-site team-structuring, and having cooperation with people who exclusively work remote (for example in other cities, or even countries). It turns out that having this solution requires some flexibility in terms of scheduling what should be done on the on-site days, and what should be done while working remote.

> *I am here two days a week, so I usually cluster together a lot of meetings those days. And then I'll join meetings on Skype or similar programs for meetings on the days where I am not at the office. But my two days on-site usually turn into meeting-days for me, so I don't get as much development done on those days. On the other hand, I get a lot more development done during the days I am at home, so perhaps it balances itself out.* - Remote Developer

Due to the flat structure, and how the teams are organized, this does not become problematic for the teams. On the contrary, it seems like the teams function just as well whether some people are working remote or not. The question become however, what is the limit for how many of your team-members can work from remote before it becomes problematic for the flow of the team-work? I asked this question to one of the team-members: "Assume that half of the team works remote. Would this pose any problems for the team?

> *Yes, it would probably be problematic. I usually work with [automatic testing software] during the days I work remote. That means if I have to talk with someone, they would be outside of the team anyways, so it is a good idea to work on something that doesn't require too much team communication* - Remote Tester

Further quotes are similar, where people who regularly work remote mention how they structure their days, and how they prioritize tasks depending on whether or not they work on-site or remote. The common way of doing it is to work on individual tasks such as development or automatic testing while remote, and to cluster team communication and coordination (such as meetings) on the days where they are on-site. Additionally, there is probably a maximum number of people who can work remote before the team suffers communication and coordination issues. What this number or proportion is, needs further research to be established.

As I mention in the beginning of this sub-category, there are two ways remote teamwork takes place. The second way is the flexibility that allows for team-members to work from home on a short notice. Whether this is due to illness that people want to take care of, or due to other private matters, does not matter as long as people inform the rest of the team that they will work from home that day(this is usually what Slack is used for by the teams). Usually this poses no problems for the teams, as people continue to work on their own stuff, and conduct meetings where the remote people can attend through Skype/appear.in. However sometimes the person missing had an important task or responsibility. If this is the case, the team is quick to find a replacement person who can deal with it, if the remote-person cannot. This is second-nature for the teammembers, both due to their good knowledge of each-others work and systems, but also due to their camaraderie and inter-personal relationships (more on this in section 5.6).

5.2.7. Remote Communication

The emergence of communication technologies have made it possible for team-members to work from a remote location. The teams I observed had several full-time members who worked remote several days a week, and had done that for several years. The teams have put in place systems and processes for streamlining communication between on-site and remote team-members such that the overall success of the teams are not negatively impacted by having remote team-members. This also allows people to work from home whenever needed, whether it's for personal reasons or otherwise.

> *I am personally dependant on [being able to work from home]. I have kids, and a significant commute, so I would have to waste a lot of time on travelling which I could have otherwise used on work [...]. It helps that we have a culture here that allows for this flexibility.* - Tester

The question that arises is whether or not this negatively impacts the team or the team-members themselves, and how this can be counteracted. Most of the day-to-day tasks that each team-members does, whether it is a designer, a developer, or a tester, are more or less easily done without the need for considerable interaction. However, appropriate measures have to be in place to help with communication. As previously noted, chat applications are sufficient most of the time. When the teams are having meetings, a simple process have been implemented; The team always bring a laptop and connect it to a portable camera/microphone gadget during meetings. They use a

program, similar to Skype, called appear.in where anyone who has the URL can join the video conference. Having this streamlined process that takes less than 30 seconds to set up, means that the teams are not wasting too much time. This works well most of the time, but it has been both noted and observed that from time to time there are technical difficulties:

> *A little too often there are problems with the technical equipment, resulting in poor sound, video, or screen-sharing quality. Appear.in usually have great sound and video, while poor screen-sharing quality, whereas Skype functions well but has more overhead.* - Remote team-member

When it comes to the productivity of the remote team-members, most, if not all, state that they get more done at home due to not being interrupted as often (see the section on *Over-Communication*). As all of the remote team-members work on-site at least a day or two per week, they do not feel like they lose out on information or coordination.

> *Even though there are limitations when working from home, it is a lot easier for me to work like this, balancing work and personal life. And as most communication happens on chat clients anyways, most people won't even notice whether I work from home or at the office.* - Remote team-member

5.2.8. Context-Switching

As it is with everything in life, too much of something can lead to negative consequences, and communication is not an exception. Increased types of communication, and improved and streamlined processes lead to better results, but there are some negative consequences that one have to be aware of. Some team-members have aired their thoughts around the "always-available" status that comes with chat clients, and that one might feel pressured to answer immediately instead of waiting because of expectations by the sender. This leads to a lot of *context switching* where your attention is constantly moved from one topic to another. This can be especially detrimental for developers, who often deal with complex problems that require 100% of their concentration. Being interrupted from this thought-process might set them back several minutes, and a lot of effort. One developer noted:

> *Over the course of a day, i might be working on something complex, and then I get interrupted. Once that is done, I'm left wondering "what was I doing?". And then all of a sudden something new pops up, and then there's a meeting, etc..* - Developer

But it is not only developers who struggle with context switching. On the topic of test environments and their complexity and their often lagging performance, it was said:

> *[...] and if you get a question about something then you have to check this and that environment, which takes long, and you have to wait for a lot of things to load for too long in the context switching.* - Test-lead

Most who noted such experiences did not mean that they necessarily wanted to change how the teams communicated, but merely noted that context-switching was a reoccurring problem for them. One person proposed a possible solution where a certain part of the day, say before lunch, was meeting-free and allowed for people to work in peace on their own tasks. However, this person also noted that this might be too rigid, and that it might create more problems than it would solve.

As with most things in software development, figuring out the right amount of communication and availability is a balancing act, which at the end of the day might leave some people unsatisfied without there being an obvious solution to the problem. That being said, it is important that people are aware of the set-backs that context-switching bring, and keeping this in mind might be the best solution available.

5.3. Autonomy

Autonomy is highly linked to flexibility, but with a slight differentiation on focus. The sub-categories found in this section revolves around how autonomy is facilitated through a flat organizational structure, horizontal team-communication, and on-boarding of new team-members into an already existing autonomous team.

5.3.1. Flat Structure

In the same sense that team-members do not work in a vacuum, the teams do not work in a vacuum either. They are a part of a greater organization, and outlining the different organizational factors and how they impact the teams is very important for an understanding of how the teams operate. The structure of this organization has been unanimously stated as *flat* or *quite flat*. This is the key to having autonomy in the teams, as removing middle-managers and letting the teams decide themselves how to attack a problem increases autonomy and the teams' ability to handle problems in the future.

> *I experience that it is more freedom with responsibility. Say, a developer gets a task that he knows how to solve, he doesn't have to confer with someone else. He can just go ahead and solve it, and send it to the testers. There is no hierarchy where solutions have to be approved, so there is a very flat structure.* - Developer

This is key, and shows how the mindset of the teams are. But this would not be possible if the organization did not allow for it to happen. The organization must trust its teams and its team-members to be responsible and honest in their work, and if they are, it leads to an increased sense of personal responsibility and team autonomy. Similar points can be made regarding the roles of the "test-leader" and the "architect". On my question regarding how they view their role, they answered:

> *We don't really have a test-lead, we have testers* - "Test-lead"

I was the "architect" at the previous project. But I view myself as a developer. But I think I've been here for so long, that I have accumulated a lot of knowledge overall, so I have an overview over how things work together [...]. Mostly I get asked to attend meetings or discussion regarding architectural decisions, but I don't view myself as ***the*** *architect* - Developer

What the teams have done is to give an increased responsibility to some people, whenever needed. So while on paper a person might be called "architect" or "test-lead", what actually separates them from the testers and the developers is an increased responsibility which usually comes due to their experience and overall knowledge of the systems and how things inter-operate. These people can be seen as "team-mentors" or sources of knowledge within their domain.

Outside of the teams, the structure keeps its "flatness", as there are very few middle-managers, if any at all. The teams communicate horizontally with each other, and leave the vertical communication to the team-leaders and product owners who communicate with the *development leader* (who is responsible for approximately six teams) and the project leaders. The teams are shielded from this interaction, as it involves information and bureaucracy that is necessary for the teams, but unnecessary for the individual team-members to have knowledge about (more about this in the sub-section on the *team-leader* in 5.7.1).

5.3.2. On-Boarding of New Team-Members

As the teams consist of a significant proportion of consultants, there inevitably is some variability in the teams, with some people leaving, and new people joining. These new team-members come both from other teams within the organization, as well as completely new people. Having a structured process in place for on-boarding new team-members will make the transition process easier for everyone involved. During my months with the teams, several people joined in different types of roles, including a new team-leader for one of the teams.

Being a new member of a team that works very much in an autonomous fashion can be difficult, as the amount of freedom and flexibility is quite high. When asked about this, the team-members said:

It is quite a lot of freedom here. More than you might think. There is no one who tells you that you have to do "this and that", or that you can just pick from the Kanban board. But once you ask, you realize that you can pick and choose what you want. - Team-member

and:

There is no one who directly follows-up on what tasks that we do here, which of course it is a good thing that people can choose their own tasks. But it might be difficult for new people who join if they aren't used to this, or unaware of the procedures - Team-member

This increased flexibility and autonomy that the team aims for can make it difficult for new team-members in the beginning. When asking relatively new team-members about their on-boarding process, it became clear that this difficulty was the case. This difficulty was exacerbated by a hectic period for the project, and the team-leader.

> *It has been a bit messy since I arrived. I don't think they were completely ready for me. This was due to them having a lot of things going on with the project etc. [...] At least in the beginning, I think you should get a deeper introduction into the systems and processes they use. This would make it a lot easier to get going. As it is now, you have to be proactive and ask questions, which can be a bit difficult when you are new.* - Team-member

This suggests that having a larger focus on structure and processes for new team-members can be helpful during their initial period when they are learning the systems, business, process, and getting to know the other team-members.

As I previously mentioned, one of the teams got a new team-leader during my observations, and I got to see and document how the team functioned during this period. While waiting for a new team-leader, the current team-leader for BM1 took over the job at BM2 as well, meaning he acted as a team-leader for both of them. This period lasted approximately three weeks. What is interesting is how well the team functioned during this period. It was a true testament to their autonomous nature, that even during the most hectic period of the project (this was during the final month of the project) the team continued to function well as if nothing had changed. People knew their roles, and they had processes in place for everything they did, so the fact that they did not have a dedicated team-leader did not impact them significantly. Speaking with an old-timer in the team about this, it was said:

> *[It worked well] because we are quite a mature team. People are responsible for themselves, and we don't have people who have to be told what to do in order to do their job. Having an active team-leader might be more useful if you don't have autonomous teams. Here people just pick relevant tasks, and if something important pops up, the PO can come in and tell us to prioritize that. So sometimes a bit of guidance, but mostly autonomy* - Developer

5.3.3. Coordination of Deliverables

As the teams work with an agile mindset, frequent deliveries of functional code to the production environment is central for the cooperation with the customer. The teams have a well established time-based framework for how they deal with deliverables, or *releases* as they call it. Each *application* that they manage have release-versions. As the product itself is built up of these applications, it is crucial that the correct version of each application is ready for release with the other related applications. Coordinating this release-schedule happens on a weekly basis, where code goes from testing to QA to pilot and finally to production. This process is as mention quite rigid and well established to make the process streamlined. The single most common topic of discussion

is coordinating these releases. This is done at meetings, especially standups, through chat applications, and with face-to-face dialogue. Having this rigid regiment for the release-schedule works as a contrast to the agile mindset that development has regarding most other activities. This helps everyone involved by being a very predictable process. It would have taken a lot more time and coordination if the process was more flexible, as the chance of misunderstandings and mistakes would increase. However, we will see in section 5.2.3 on *flexibility in release-schedule* that this release process has some flexibility to it, fitting in with the flexible nature of the teams whenever necessary.

5.3.4. Inter-Team Coordination

Continuing with the topic of releases from the previous sub-section, there is an important point to be made regarding coordination between the teams when it comes to releases. The streamlined release-cycle allows the teams to trust each other on having the related applications ready at the specified time. This increased trust removes the need for a lot of the coordination that the teams otherwise would have needed. Most of the coordination between the teams regarding releases is to inform each other when applications are ready, and so it is only on the off-chance where an application is not ready that there is need for a delay which has to be coordinated. However, if a delay *is* needed, the release-schedule is flexible enough to allow for a day or two delay between *code-freeze* and QA.

5.3.5. Inter-Team Communication

Observing a large organization with a significant amount of teams made it obvious that not only communication within the teams is important, but across team borders as well. There are two ways that inter-team communication took place; by walking over to the relevant team and have a chat, or through chat/email - and it turns out that there is a stark difference between the two when it comes to efficiency. The two teams I observed sat next to each other, which allowed for easy communication by simply walking over to the person you needed to speak with, and figure out the problems immediately. If this was not possible, the team-member would have to either send a message through chat or through email, which would lead to some of the issues I described in the subsection on *Communication Types* above. This would lead to problems, as one developer noted:

> *Sometimes I've experienced that all interaction takes place through HipChat, and sometimes I don't get a reply fast enough. It can take a day for a simple correction/change [...]. I like face-to-face meetings, and I'd like people to sit close to me so I know whether or not they're available for a chat.* - Developer

Speaking with a developer about how he feels about interacting with other teams, he stated:

> *I don't view us as teams [in this context]. Our applications do not live in a vacuum* - Developer

Especially the communication between the two BM teams is equally efficient as any communication within a team. They are seated in the same office space, and they are close enough inter-personally that the threshold for going over to the other team is no different than talking to someone on their own team. This promotes good communication between the teams, which in turn results in efficient solving of problems, and a quality end product.

Several team-members noted that from time to time problems arise as a result of poor communication across the teams. This is usually caused by one team making changes to the code without including, nor communicating, these changes to the affected teams. Sometimes this is impossible to control, as it is difficult to keep track of everyone who potentially could be affected by a change you make, but other times it is due to negligence by the person who made the change, who should have known the implications of that change.

> *Sometimes APIs for example have changed without us being notified of it. This often results in errors in our code as return values/codes might have been changed, or endpoints have been completely reworked [...]. This might just be the nature of the beast, though, when you have multiple applications made by different people.* - Developer

As an antidote to this, the teams have established cross-team meetings such as the *teach-lead meeting* where each team sends a representative; their tech-lead or one of their senior developers/architects for example. There discussion and information sharing takes place to ensure that whenever a significant change in the architecture, APIs, or technologies happens, it is communicated to the other teams so they can work around it and implement changes accordingly.

5.4. Cross-Functionality

The related sub-categories to cross-functionality details cross-functional team-structures, communication between roles in a cross-functional team, and an important distinction that can lead to misunderstandings if the organization is not vigilant.

5.4.1. Cross-Functional Team Structure

The teams are structured with an approximate distribution of 30% developers, 20% testers, 20% designers and 20% Team-Lead and Product Owner, and 10% variability. Mixing all of these roles into one team, requires them to be seated together in the office (as opposed to different areas of a building), as otherwise cooperation would be infeasible. This is one of the key factors that several team-members brought up regarding what is important about their structuring, namely that they **sit together**. Being seated next to each other allows for very easy communication and coordination across roles.

> *The fact that we're sitting like we are, close to each other. It makes it easier to ask about what you need, and the more you are here the more you know*

which person knows what, and so you know who to ask with experience. You get an answer straight away. - Tester

The daily interaction is important. Dialogue between people, where they can point and discuss things on the screen etc. I often see people sitting together, cooperating and helping each other. It helps having a lot of knowledgeable people too. - Test-Lead

Being seated together, working together, and following each step of the process makes people more invested in their work. It is often mentioned how you as a developer often can get stuck working "in a vacuum", not having much interaction regarding what happens before and after development, but when working with a cross-functional team you get a sense of increased ownership as you following your code from design throughout development, testing, and release.

If you just code and code without knowing what happens with your code out in production, it can get quite boring for a developer. So you can see the whole picture [by following the whole process]. - Developer

Another thing the teams do is to visit customers around the country, seeing how they interact with the systems, speaking with them, and hearing about their positives and negatives. This, which is a typical designer-task, has become something everyone on the team has to do at least once, leading to several improvements like increased ownership, seeing the big picture, becoming cross-functional et cetera.

Developers who usually never go out of the "lab" can go out and visit customers across Norway, seeing these people and talking with the customers, advisors and end-users. Which is great. Not only do you become more motivated and invested, but also cross-functional. - Developer

5.4.2. Coordination Between Roles

Possibly the most important consequence of increased coordination in a cross-functional team is coordination between different team-roles. As many interviewees touched on, being able to quickly discuss, plan, or cooperate on a task is a significant improvement from how some of them have previously experienced so-called "single-purpose" teams (e.g. a team of only testers) in previous organizations.

For me [being cross-functional] is great. We have great testers and designers, and we work together on functionality. - Developer

I have experienced problems working other places where we work separately. Here you sit together in the same meetings and discuss the same solutions and you can immediately flag an implementation that you know will not work, saving everyone a lot of time. - Designer

The cross-functionality leads to *cooperation* between roles too. It is one thing to coordinate between roles, but having it lead to cooperation between roles strengthens the positive outcomes of cross-functionality.

> *The workflow is to just pick an item [from the prioritized backlog], and work on that task. We can include the testers, and go back and forth between us until it is good enough to ship.* - Developer

Team-members would often during a day confer with each other on solutions. Not only between testers and developers, but everyone in the team. This would take place during meetings, after meetings (especially after standups), and throughout the day. This cross-functional partnership is central for the teams' success in delivering quality products on time.

5.4.3. Cross-Functional Teams vs. Cross-Functional Team-Members

Throughout the process of working with a cross-functional team, interviewing the team-members, and thinking about what cross-functionality actually means, I realized that there are two ways of measuring cross-functionality. The first is the classic way of talking about cross-functionality, which is cross-functional *teams*. Cross-functional teams are made up of people with different knowledge and roles who work together closely in software development. I will talk in depth about the advantages of this in the later sub-categories. However, you can also talk about cross-functionality in regards to cross-functional *team-members*. With cross-functional team-members, you don't just have a team made up of different roles, but the team-members themselves inhabit skills and competence which ranges across multiple roles, creating a much different dynamic for the team (this is also sometimes called *Redundancy*). This distinction is not as often made, neither by the theories, nor by the practitioners themselves. Bringing this up during interviews sparked a lot of interesting conversations.

> *I think a lot of people have the mindset of: "We sit together, therefore we are cross-functional". But often the process is that UX starts, then development takes over, then testing, etc. That is not cross-functional, then you are just acting like it cause you are sitting together.* - Team-member

> *[When making the distinction,] I am not cross-functional as I am only a developer, but the team as an entity is cross-functional* - Team-member

Not making this distinction can lead to misunderstandings if people have different definitions of what "cross-functional" means. At the very least, people should be aware of the differences such that when discussing structure and processes and roles, people are aware of what is being discussed, and what is considered cross-functional for the current team or organization. These words and definitions can turn into *buzzwords* in the organization and for the business-people, instead of having any weight or specific meaning to them.

5.5. Mindset

Mindset represents the thoughts and the attitude that the organization and the team-members have towards key factors such as change and continuous improvement. Some sub-categories revolve around exactly this mindset towards change, some detail how information and knowledge is shared and the importance of this, and lastly a couple examples of how the teams and the organization have made significant changes to structure and process.

5.5.1. Organization's Leaders' Mindset

Having an organization, and more specifically leaders of an organization, that are open and willing to have a focus on structure and process is very important. It was said by several interviewees that the mindset of the organization shapes the structure and the way work is done by the teams in the organization.

> *I like to think of the bank as a dream customer. They are very open and they love to hear proposals from us. So if I speak with someone and say that we might want to try "this or that", they are very enthusiastic about it, and they implement it if they think it is a good idea.* - Developer

The organization seems to be very up to date on matters that are cost-sensitive, and they quickly realize if they are doing something that is inefficient or wasteful. However, they are not afraid of spending money where they see potential.

> *I think they realize it is a waste if people go to meetings that aren't productive. But at the same time, they are great at arranging seminars, hackathons, and the like. So they are not stingy with their money. They know that learning and improvement is important. They view it as an investment.* - Developer

A key area where the organization's mindset is paramount is when it comes to change and having a culture of improvement. I will go into detail about this in the following sub-categories.

5.5.2. Mindset for Change

First of all, there is a mindset that can be found amongst the teams that is a precursor to a culture of change, namely being aware of waste, inefficiencies, and potential improvements. Once you are aware of these discrepancies, you can begin to solve the problems if you have the time, resources, and competence. There are more factors to take into consideration, however. As you do not work in a vacuum, but as a part of a team, you have to ensure that the changes that are made either will not impact the rest of the team negatively, or the rest of the team has to agree with you before you begin to make the change. An interesting example was found when I was digging through older retrospective meeting notes from before I arrived at the team. There, approximately

six months prior to my arrival, the teams mentioned in two consecutive meeting notes that the standup meeting was taking too long. Then, the following meeting report, it was noted that they had successfully reduced the time of the standup meetings considerably. I asked the various team-members about this during the interviews, and they noted a couple interesting thoughts. The consensus seems to be that it is a positive change to reduce the time spent on standups, however some feel like they get more out of standups than others do, indicating that they would liker longer meetings. It seems like the team is still searching for an optimal solution to it, which to me shows that they are still thinking about ways to improve stuff even after they have made corrections and changes. The examples of changes are plenty, ranging from meeting duration and productivity, to automation of the deploying-process, to improvements and research on AI in relation to automatic testing software.

There is also a focus on improvement that comes from above, which the team-leaders have to keep in mind. An example of this is how they are aware of two metrics, namely *cycle time* and *time-to-market*. Time-to-market meaning how long it takes from a task enters the backlog to its completion and implementation to the production system, and cycle-team meaning how long it takes from a task is started until it is implemented into production.

> *A key factor that makes these metrics worse is through inter-team communication which eats up time. This is something we are actively trying to reduce, to improve the metrics over time.* - Team-leader

For a view of how the teams deal with inter-team coordination and communication, see sections 5.3.4 and 5.3.5 respectively.

5.5.3. Promoters and Driving Forces

When asking the interviewees about having a culture of change, and how they view it, several of them mentioned that it often comes down to people figuring stuff out on their own. That there is a certain mentality that some people have, where they see a problem and they spend time and energy to be the driving force to fix these problems. The initiative comes from the autonomous individual.

> *I think there are some central people who are quite technically adept who act as the promoters and evangelists for how they wish things were, in a way that inspires others. For example at our tech-lead meeting, you have central people with vast technological knowledge discussing and showing potential new technologies and implemented features to the rest of the teams, influencing them.* - Developer

So there are, within most teams, some individuals who take on an extra burden and seek to solve problems and make things more efficient. At the same time, there are others who do not feel like taking this responsibility, and would rather wait for someone else to do it.

> *Sometimes when a developer encounters a problem, I see him taking the time to fix or improve it immediately. On the other hand, there are others who encounters problems, and think to themselves "I can not be bothered to fix this, I'll wait for someone else to do it", so it varies from person to person.* - Developer

However, when a person actually wants to make a change, and the time comes for when he has the resources to do it, the person is likely to be encouraged by both the teams and the organization to figure out potential solutions, and even implementing them without needing a grand process involving a lot of bureaucratic nonsense.

> *There is a very low threshold. You just do it, and then it usually spreads by itself to other people and other teams. There is usually a handful of these people with the technical know-how who implement it, and then it spreads to everyone else by them promoting it.* - Developer

5.5.4. Knowledge Sharing

On a related note, there is a lot of knowledge sharing taking place amongst the teams, and within the organization as a whole. As mentioned in the previous sub-category, when a new solution has been implemented in a team, the people are usually very eager to show this to everyone else. There are several avenues where this takes place, such as the demo-presentation which takes place a couple times a month. There is another workshop-like meeting which takes place from time to time where people are introduced to new technologies, and there are hackathons with regular intervals. All of these arenas are used to promote and introduce each other to new technologies, new ways of thinking, and solutions that potentially benefits the organizations as a whole.

Additionally, there is knowledge sharing taking place at a smaller scale within the teams and related teams. Often someone airs an idea at one of the team-meetings, talking about a new idea or a new technology. This often leads to other people chiming in with the same idea or the same mindset who wish to learn more. Then you often see people sharing resources and things like online-courses over the chats, helping and introducing each other to new ways of thinking.

5.5.5. Sharing of Information

Information sharing is highly linked to coordination and communication. The separation I chose occurs with the differentiation between information that is shared for a specific purpose (Coordination), often related to a specific task or goal, and information that is shared simply to inform or make other people aware of something (Communication). This section is about the latter of the two. The situation where such information sharing predominantly took place includes, but is not limited to:

- Standup-meetings, team-meetings, backlog-groomings.
- Informally through chat.

- More formally through email (usually team-wide or company-wide emails).

Of these, the standup-meeting was the platform that is the most central for information sharing. The standup is coincidentally one of the more divisive topics, with differing opinions depending on who you ask. The key aspects concerning information sharing and standup-meetings which different roles and people in the teams disagree on is; the usefulness of such information sharing, the standup-meeting's successfulness in sharing this information, and the appropriate length of the standup as a result of this. One team-member said:

> *I like to get an overview of what the developers are doing. My job as a tester is to work with the work of other people, and thus i would like to be up to date. For this reason the standup might be too short in my opinion* - Tester

Another said:

> *I think they do [standup] well here. It's almost mechanical, though. "I did this, you did that, she did this". Sometimes to the extent where I don't feel like I get anything out of it [...]. It's almost so quick that you don't feel like you got any insight or information.* - Developer

And a third said:

> *Most of the time it's not productive in a way that helps me with anything, but sometimes I get some knowledge about what people are working on. But everyone is working on different things, so I don't usually get anything out of it. But from time to time a nugget of useful information is shared* - Developer

Several team-members, with different roles, mentioned that the standup might be more useful to the Team Leader, Test Leader, and the Testers, as they directly benefit from knowing what everyone is working on, and any information they have to share. Regardless, several team-members noted the usefulness of a longer standup/team-meeting on Mondays, where deeper discussion could take place, and information relevant to everyone was shared. The combination of efficient standups most days, coupled with an extended standup/team meeting on Mondays might be a solid middle-ground which balances time and productivity for everyone.

5.5.6. Responsibility and Ownership of Work

This final sub-category is one that could fit in several of the overarching categories. The relationship between autonomy and ownership has been stated over and over by participants in interviews and through observations and discussions. When people feel like their work is meaningful, that their work has a significant impact on the product as a whole, it becomes way easier to be motivated to do good work, and to work good for the betterment of the team as a whole.

[On the topic of cross-functionality's impact on results] - It definitely impacts results. When everyone has a common mindset, people are willing to work extra hard when they feel like they have an impact as a part of a greater whole. If you have made something bad or wrong, then you will work way harder to try to fix it when you have a sense of ownership to your work, than if you were just "pushed" to make something by someone. - Developer

With a mix of increased responsibility, increased trust, and being part of the whole process, the team-members state that they feel an increased ownership of their code, and even the product as a whole. This is key to keeping motivated team-members, and having a team-autonomy where people actually care about their work, which is so essential.

I feel like the majority wants to make something that has a positive lasting effect. Only a minority wants to work in a vacuum. And by working together from the get-go, you feel like you have a much greater impact on the results, which leads to increased feeling of ownership. - Developer

The following two sections will detail two ways that the teams and the organizations have implemented large scale changes in team-structure and System composition for the betterment of processes.

5.5.7. Increased focus on UX

Something worth mentioning is the increased focus on UX and interaction design within the teams and at the organization as a whole. Using hypothesis-driven development and user-data the teams often have great insight into what the user wants before it is developed. This increased focus over the last decade has led to the organizations being aware of the cost-savings this can lead to.

[Talking about previous experience] - You used to have to be a very good salesman when working as a designer, in order to convince people about design-choices. Now, however, it is a lot easier because you have gotten a good insight into the user, and so you have concrete proof, empirical numbers, and facts that show "this is what the users do/want". That has made things a lot easier. - Designer

Additionally, due to this not being a completely new thing, enough time has passed such that the organizations have seen the positive improvements this has lead to, and if the organizations are open to new ideas, then they quickly realize that there are important cost-savings to be made here, which quickly changes their opinion.

Since the change happened, enough time has passed so that we can document the effect of it. So it is a lot easier to show that user-centered design brings in good money. At this point the businesses quickly agree to change if you show them the potential savings. - Designer

5.5.8. System Reconstruction

Lastly, it seems appropriate to mention a large example of this focus on change. The organization has gone through a massive restructuring of a monolithic codebase to a modular one for the past five years. This process was in its final stretch as I joined the teams in January. During the following three months, the teams successfully finished the restructuring of the new architecture, and shut off the monolithic-based system on time in the beginning of April. This was a massive undertaking for the organization, but they knew that in order to be competitive in the market in the coming decades, they needed to have a system that was made up of modular pieces, as opposed to a mammoth. Even though this lead to a decrease in the new features they could offer their customers during this time period, it was a necessary evil that they were willing to do.

> *The impression is that it costs to live in the past, and the bank is aware of this. For example in regards to the [monolith], the bank is very aware that it will be costly to keep it alive for longer, so they are eager to shut it off as soon as possible. The downside is possibly for the end-users, who do not necessarily understand why things are changing for no apparent reason, but they don't understand the underlying architecture* - Developer

The bank being aware of, and removing technical debt is a strong indicator that they are willing to sacrifice in the short term, in order to have a healthy code-base, and thus product, and thus business in the long-term. It has already made improvements for the developers, who instead of having to push and deploy the whole monolith every time they made a change (which meant that the teams would only do a new release every quarter of the year), now can push a small change in one of the application, as long as they ensure that it does not negatively affect any other applications.

5.6. Morale

The *Morale* category represent topics and sub-categories that relate to inter-personal relationships, social factors in the teams, and how this positively impacts the results of the teams, both in terms of quality and efficiency.

5.6.1. Low Threshold for Asking Questions & Help

Different team-members have different sets of competences. No one knows everything, and there will inevitably come a time when a team-member is stuck on a task. This could be due to them being new to the team, or new to the industry, or an experienced developer working on a new piece of technology. Whatever it might be, it is important for efficient teams to have a low threshold for asking each other questions, and helping each other. A developer noted:

> *I won't use more than 30min on a task if I'm stuck. I won't sit there trying and trying forever. I would rather ask someone straight away, whom I know have the knowledge I seek.* - Developer

From my observations, this seems to be the case for most of the team-members, and it is corroborated through several other interviews as well. A culture with a low threshold for asking people have been mentioned as a key reason for both more efficient work, as well as a more friendly and cooperating workplace.

> *[...] If you're stuck on something and ask someone for help, people never say "I'm busy, can you come back later?". People are always open and welcoming when someone needs help.* - Developer

This interaction is often across different roles, and team-members seem to have an equally easy time interacting with other roles than their own. As a tester noted:

> *If I don't understand something, I'll ask a developer. For example if something is poorly explained or detailed in Jira, I'll ask the person responsible directly.* - Tester

Integrating the different roles in the teams have significantly increased the ease of communication between different roles. This is practically unanimously stated by all interviewees, and supported by my observations of how the teams work in a day-to-day setting. Couple this with a low threshold for asking (anyone) questions, you achieve a very open and coherent team with a high grade of willingness to help each other. However, this culture can also have downsides, which I detailed in section 5.2.8 on *Context-Switching*.

5.6.2. Good Personal Relations

Seeing the members of a team being friendly towards each other, and knowing a little bit about each others personal life, goes a long way towards a workday that feels a lot easier and manageable than if the opposite was true. Having this friendly tone amongst each other reduces any chance of conflicts, and if conflicts do arise, they can quickly be sorted out. As a matter of fact, I do not think I witnessed a single negative interaction or conflict during my observations, and no person mentioned any of the sort in the interviews. What I *did* witness was people being respectful of each others work, and being friendly towards each other. Surely coming into work knowing you will meet like-minded people who enjoy working together is a much better feeling than the alternative, and so fostering good personal relations should be a priority for any team. Having people who arrange social events outside of work, or even a small get-together for someones birthday in the common-place can go along way to bond with each other, and make everyone feel welcome, as my research suggests.

> *If you see the team here, everyone is very welcoming, especially to new people [...]. I think people laugh a lot, and it is enjoyable to work here. A lot of joking around too. It is almost like you are working with friends, and when you work with friends, you do not want to only do the least amount of work needed. You want to contribute, and you want to do more stuff together.* - Developer

This friendliness also makes it easier to ask people for help or consultancy, as people might not be as afraid to interrupt.

> *It is easy to ask here - a low threshold.* - Test-lead

My results suggests there are plenty other things that create a more pleasant workplace. I observed small things such as going for a coffee with a coworker, or having a chat about something off-topic to get a little break. It feels a lot better if you can do that without feeling like you are slacking off.

5.6.3. Collective Lunch

Eating together is something the team strives towards. Most days, the majority of the team eat together in the canteen, typically around or slightly before 11 a.m. It allows people to talk about something other than work, which I find to be true as there is very little work-related talk during lunchtime. Most people can relax and chat about whatever they want. Several people mention that this positively impacts their day and their relationship with the rest of the team-members. Additionally, from the aforementioned retrospective reports, I found that "eating lunch together" was a change they wanted to implement, and a goal which was reached. It also helps a lot for newcomers such as myself, or new team-members, to become acquainted and get to know each-other a bit better, which makes people feel a lot more secure in general.

5.6.4. Praise

Lastly, I want to mention positive reinforcement, for example in the form of praise. It is something that often is done by the team-leader or the product owner e.g. after an important release or a significant milestone. They often do this in plenary, either at meetings or through the chat if they are not on-site. It is noticeable in the faces of the team-members that this praise is much appreciated, even if people do not necessarily admit it. It might not be a particularly difficult or central topic of software engineering, but having respect for each others work, building personal relations, and all around togetherness helps people enjoy their time at work, which inevitably impacts the end-product in a positive way.

5.7. Coaching

The final category consists of only one sub-category, but I feel like it is warranted due to its importance. The topic is on the role of the team-leader, and what the team-leader does for the team (and what they do not do).

5.7.1. The Coaching Team-Leader

The role of the team leader has a changed a lot from the tradition, old style, team-leader who practically ruled over the rest of the team. The modern team-leader, especially for

cross-functional teams who strive for autonomy, takes on a very different approach to the role. Their responsibility changes from delegating and ordering people around, to acting as a coach. A person responsible for the team and its success, and who reports to the people above in the hierarchy, but other than that serves as a helping hand for the team-members wherever they might need it. As an example, the team-leader shields the team from non-relevant information and bureaucracy that otherwise would have have wasted a lot of team-members' time if they had to deal with it. Acting as a link between the team, and any relevant upper management, the team-leader can do a lot of administrative things, attend overarching goal-setting meetings, and other organizational meetings that are necessary for cohesion across the teams.

> *I think it works great. The team-lead guides the team, does administrative work, logs hours, etc. He serves as a mediator and middle-man for communication with the employer and customers. Not everyone in the team has to know about everything, and so the team-leader can involve the people he feels are most relevant for the situation.* - Tester

The part about the team-leader "guiding" the team forward is an important thing to note. Several team-members answer that the team-leader pushes people, in a good way, towards improvement and increased responsibility. He should be able to spot when someone has the ability to take on more responsibility.

> *He has put a lot of extra responsibility on me throughout the period I have been here. He is great at challenging me, even if I have to go outside my comfort-zone a bit. In the beginning most things seem a bit scary, but as soon as you get used to it, you realize that you can do a lot more than you thought.* - Test-lead

Having a team-leader that serves as a teammate, who is on your side and does what he can do to help you increases the morale in the team, and is something I touch on in the section about *Morale* in 5.6.

Several team-members talk about how important it is that the team-leader not only have leadership capability and business interests, but that he has the technical background and understanding on the same level as for example the team's developers. This ensures that he can be a part of any discussion, and can help the team with difficult situations, should they arise.

> *What's so special about our team-leader is that he was previously a developer here [...]. You need a team-leader who understands both the functional aspects of development, as well as business. That's were our team-leader came in, shielding us from unnecessary information, but can always take part in technical discussion or give relevant information.* - Developer

Not only does the team-leader seem as a facilitator from an outsider's perspective, but the team-members view him as that too. This understanding of what his role is, and what he can do to help the team in their activities is very positive for everyone involved.

When it comes to the work itself, then he seems like a facilitator. He makes sure we have enough to do, and the right thing to do. Whenever we need clarifications, he can fix that, and make sure we get the right information. So for us, a team-leader is a facilitator. - Developer

5.8. Category Relationships

While it is crucial to have a significant structure to your theory, it is just as important to detail the relationships between the categories that make up the theory. As a theory aims to explain a situation in reality, its categories and concepts are naturally going to be significantly related, and not exist in a vacuum. Some of the most central relationships between categories and related sub-categories will be further detailed in this section.

Figure 5.2 shows the category hierarchy as seen in section 5.1 at the beginning of this chapter, with some modifications to show additional relationships between the central categories themselves. As you can see, some relational arrows are bidirectional, while some are unidirectional. This indicates whether both categories are influencing each other, or if the influence is only in one direction (speaking abstractly).

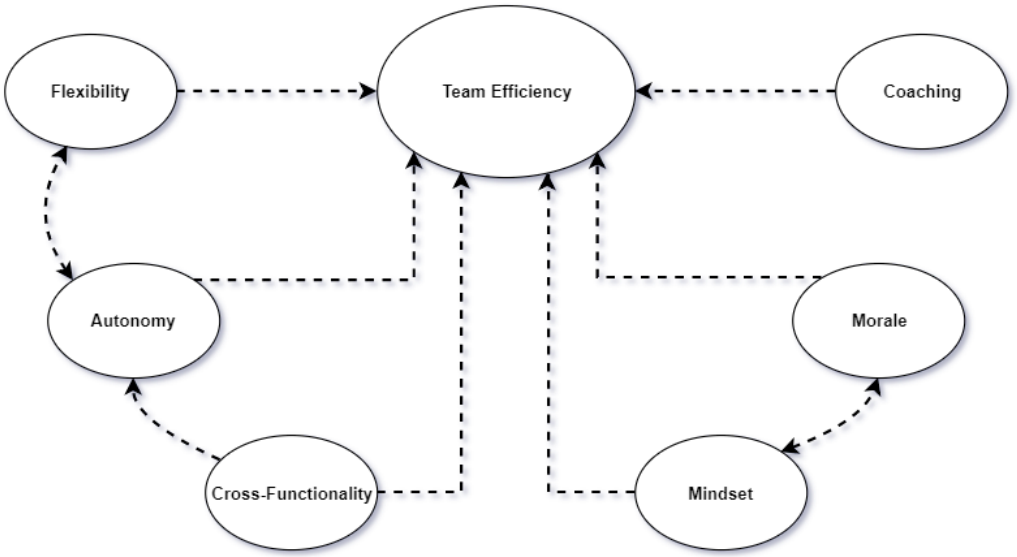

Figure 5.2.: Category Relationships

5.8.1. Coaching

While figure 5.2 shows *Coaching* as if it is isolated from the rest, the contrary is actually the case. Coaching is highly related to most, if not all of the central categories in this theory, however for ease of viewing, these links are not represented in the above figure. To highlight the importance that the coaching team-leader plays in the overall software

development process, I will start with this category. The obvious starting place is the relationship between coaching and autonomy - specifically *flat structure* (5.3.1). Not having a bossy leader, but rather a motivating and highly competent asset as a coach is essential to achieve autonomy. If this is not the case, then autonomy can not be achieved, as the team will follow a hierarchical decision-making practice.

Furthermore, the coaching team-leader acts in some ways as a facilitator for *flexibility* in the team. This is especially true for the sub-categories within *flexibility* that are related to freedom and autonomy to choose tasks, and take responsibility. Having a coach who pushes the individual team-members to take on more responsibility, as mentioned in topics within both *flexibility* and *autonomy*.

One of the jobs of the coaching team-leader is to serve as a link between the organization and the team. Thus he is an important source that can impact both the *mindset* of the organization, and the *mindset* of the team. Positively influencing both entities require an understanding of the overall goals of the organization, while representing the interests of the team, to create quality products for the customer. The coaching team-leader is thus central for breaking down borders, and establishing a shared mindset for the organization and the teams.

Lastly, the coaching team-leader influences *morale* in the sense that he is seen as a respected figure in the team. He should be available whenever team-members need someone to talk to, and he should motivate and *praise* the team when they perform quality work.

5.8.2. Autonomy & Flexibility

There is an argument for having *Autonomy* as the main category. However, I would argue that the goal of having autonomy is to achieve further efficiency with the team, which is why *Team Efficiency* is the main category. While *Autonomy* have quite few sub-categories, I would argue that *Flexibility* is highly related to autonomy, and several of *flexibility's* sub-categories could easily have been under *autonomy* instead. Sub-categories regarding freedom and responsibility under *flexibility* are highly related to *autonomy*. To represent the overlap between these two central categories, see the Venn-diagram in figure 5.3.

Another way to view it is an invisible hierarchy where *Flexibility* is a sub-set of *Autonomy*, but due to their similarities in practice, a Venn-diagram should suffice to portray the relationship. Both of these central categories are tightly related to the topic of *Agile* software development. Ties can be linked between the agile manifesto, and these two categories, and I will expand on this in *Chapter 6 - Discussion*.

5.8.3. Autonomy & Cross-Functionality

Cross-Functionality is what enables a team to be autonomous. It is central to cooperation that the team consists of multiple roles. If this was not the case, then the team would rely on significantly more coordination with other teams (say, one team for developers and one for testers). Instead, having all roles within one team allows for decision-making

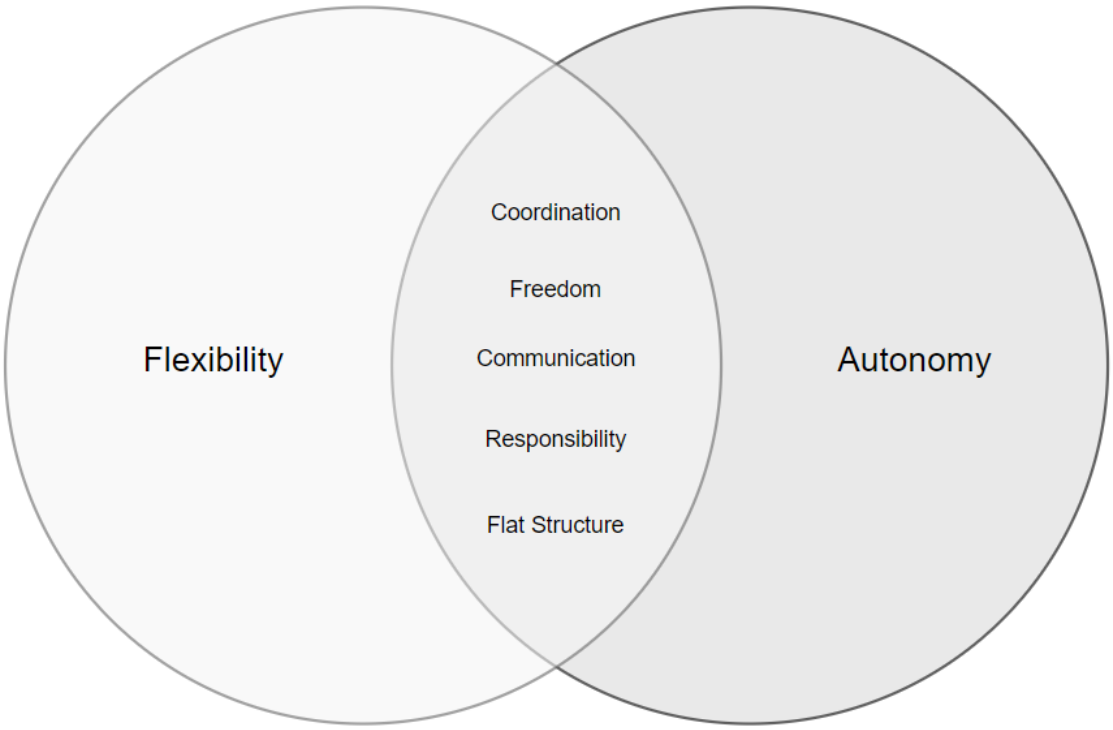

Figure 5.3.: Flexibility-Autonomy Overlap

to happen within the team, without inter-team communication being a necessity. That being said, in some cases the team would still need to communicate outwards, as the teams do not exist in a vacuum, but the amount of this outward communication and coordination is significantly reduced due to the cross-functional nature of the teams. *Cross-Functionality* is very much a structural attribute of the teams, whereas *autonomy* and *flexibility* exploits this structure, creating processes that is suitable and more efficient. *Cross-Functionality* can therefore be seen as an enabler for *autonomy*, which is why the relational arrow in figure 5.2 is unidirectional.

5.8.4. Morale & Mindset

Mindset is similar to *Coaching* in the sense that one could argue it impacts all of the other central categories. This might be correct, as it is the mindset of the organization and the teams which decides to have *Flexibility* and *Cross-Functionality* which both allows for *Autonomy*. However, as it would clutter the diagram in figure 5.2, I only linked it to *Morale*, as morale and mindset could be seen as two sides of the same coin. *Morale* is mostly related to social relationships within the teams, whereas *Mindset* is an overarching mentality that permeates through the entire organization. I would argue that *morale* is a necessary precursor to *mindset*, as you cannot expect to have a positive mindset in your organization if your organization is full of teams with poor morale.

6. Discussion

This chapter is related to discussing the results presented in the previous chapter. I will begin by talking about my results in light of the previous research done on selected central topics. Following that I will talk about implications for implementing my theory in practice, how this should be done, and how the relevant organizations can benefit from it. Then I will talk about *implications for theory* where I will detail how my theory can be used by other researchers. Finally I will talk about *validity* and *limitations* of the results.

While there seems to be a lack of grounded theory research done that present a theory for software development teams as a whole, there have been research done on more specific parts of the process. Due to this, these following sections will compare my specific findings to other research's specific findings on selected key topics such as autonomy, agility, and waste.

6.1. Autonomy & Self-Organization

In this section I will revisit the topics presented in section 2.3 in the *Background* chapter, regarding autonomy and self-organization, and reflect on to which extent my results are comparable. In their article on challenges and topics for future research regarding agile, autonomous teams, Stray et al. mentions multiple topics which are relevant to my findings [38]. The first topic Stray et al. discusses is that of leadership, and the importance of having a coach. They propose the following question: "How to design, support, and coach autonomous teams?". The entirety of the theory presented in the results in this thesis can serve as an answer to this question, with central categories such as having a necessary *mindset* to enable structural and procedural changes, which can be further facilitated and *coached* by the team-leader.

The next topic Stray et al. discusses is that of coordination. They ask what are effective intra- and inter-team coordination mechanisms, which I have several topics on under *flexibility*, *autonomy*, and *mindset*. Furthermore they ask how system architecture can best support coordination of autonomous teams. This, I have detailed in *System Reconstruction* under *mindset*, where the organization spent years restructuring their monolithic system to a modular system, which is much more suitable to autonomous teams with flexible and independent releases.

Thirdly, Stray et al. discusses the organization, and pose the question: "How to change the mindset of the wider organization to adopt agile autonomous teams". How this is performed in my observed organization is extensively documented in the category *mindset* in the emerged theory.

Following this, they discuss the design of autonomous teams, and ask what the effective team structures of autonomous teams are, and "how should agile practices be adjusted to promote effectiveness in cross-functional teams?". The topics on team structure, and specifically on cross-functional teams are detailed in the category on *Cross-Functionality*.

Lastly, Stray et al. discussed team processes and "how can communication tools such as Slack improve collaboration and coordination?", which I have discussed in *Communication Types* under the category *Flexibility*.

It is important to mention that I was not aware of this article, nor any of these questions until after I had completed and presented the resulting theory. This is important to state because the resulting theory in this thesis emerged organically, and the fact that it answers several of these proposed research questions, without me being aware of them from the beginning, strongly suggests that the theory itself is solid.

In their paper on "Developing a grounded theory to explain the practices of self-organizing agile teams", Hoda et al. describes several balancing acts between facets of software development. "Freedom provided by senior management vs. responsibility expected from the team", "Specialization vs. cross-functionality", and "continuous learning vs. iteration pressure" [13]. These topics are also to be found in my results, where both *freedom*, *responsibility*, *cross-functionality*, and *continuous improvement* are important sub-categories in the theory. Their interplay, and this "balancing act" as Hoda et al. mentions is very clearly an important aspect of the organization that took part in this research as well.

Furthermore, Hoda et al. in a different article regarding "self-organizing roles on agile software development teams" [14], speaks about observations that lead to the emergence of different abstract roles that each agile self-organizing team had. These are roles such as: Mentor, Coordinator, Translator, Champion, Promoter, and Terminator. If we were to compare these abstract roles to the results of my research, we would find several similarities. The "Mentor" is similar to the Coaching team-leader, a central category in my theory. The "coordinator", whose role is to "manage customer expectations and coordinating customer collaboration" [14], is similar to the role of the PO, who coordinates with the customer. The "promoter" can be found in the sub-category regarding promoters of change and improvements. However, the "terminator" was not observed by me and was not evident from my results. This "terminator" is a person responsible, according to Hoda et al., for removing people from the team who are threats to the self-organizing nature of the teams [14]. This may be due to it not being a problem at my observed teams, or it could be that this is a weakness that my observed teams have. To what extent this is detrimental to the teams is hard to say.

Lastly, Hoda and Murugeson's article on self-organization and project management [12] found challenges that self-organization brings. They broke these problems down in levels, namely at the project-level, the team-level, the individual-level, and the task-level. These challenges were:

- Delayed/changing requirements and eliciting senior management sponsorship, at the project level.
- Achieving cross-functionality and effective estimations at the team level.

- Asserting autonomy and self-assignment at the individual level
- Lack of acceptance criteria and dependencies at the task level

Let us go through each of these four levels, and see if my results have any of the same challenges. On the project level, regarding changing or delayed requirements were not an issue for my observed teams. However, eliciting senior management sponsorship is an important topic, which I detailed in category *Mindset*, where I talk about how the mindset of the organization is crucial to the overall success of autonomous teams. My findings suggests that the overarching organization must be equally aware and supportive of the autonomous nature of their teams, as the teams' mindset themselves. On the team-level, the question of cross-functionality is obviously central to my theory, and talked about in depth in the category *Cross-Functionality*. On the individual-level, regarding autonomy and self-assignment of the individual, we have again similar results, where my findings suggests that this increased freedom and responsibility is crucial, and leads to increased sense of ownership, which in turns leads to motivated team-members and quality products. Finally, on the task-level, the biggest issues seem to be in regards to dependencies. The difficulty with having autonomous teams and individuals is that whenever inter-team coordination and cooperation is needed, then it is important that changes in code dependencies are properly documented and communicated such that the teams do not run into compatibility issues down the line. It is important to be proactive here, and parts of my results suggest that this might be a necessary evil as a consequence of autonomy.

6.2. Agile

While you wont necessarily find a lot of the keyword *agile* throughout my results, the actual results themselves are often quite closely related to the agile principles laid out in the agile manifesto [4]. The scope of the thesis is similar to that of the agile manifesto, in the sense that you take a step back, and look at things at a higher level of abstraction. Not only is the abstract nature of the manifesto a similarity to my results, but the actual topics and categories have much in common. I will detail a couple of values and principles from the agile manifesto, and see how my results are comparable. One of the key parts of my research results is the topic of autonomy or self-organization. The results suggest that autonomy is key to efficient development of quality products. Requirements are discussed and improved by the autonomous team, leading to better designs and better quality. This correlates well with Principle 11 from the agile manifesto [4].

> *The best architectures, requirements, and designs emerge from self-organizing teams.* - Principle 11

Furthermore, my findings overlap quite extensively with the remaining principles. "Customer satisfaction through early and continuous software delivery" (Principle 1), and "Deliver software frequently, [...] with a preference to shorter timescales" (Principle

3) both relate to my findings regarding the importance of being able to release as often as needed, by having automated parts of the deployment such that anyone can do it, and restructuring the whole codebase to a modular system such that the whole organization does not have to release at the same time. "Accommodate change, even late in development" (Principle 2) is easily done by the teams due to their usage of Kanban with prioritized backlog-items and an iterative process, coupled with flexible releases. "At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly" (Principle 12) - this is highly related to my findings on *mindset for change* and to a lesser extent *knowledge sharing* as a means to achieve this improvement. "The sponsors, developers, and users should be able to maintain the pace indefinitely" (Principle 8) is achieved by the teams' use of Kanban and a prioritized backlog ensuring that there is always work available and ready to be done. Weekly backlog-grooming meetings facilitate this, where prioritization and discussion takes place between the team-members and the different roles.

The agile principles, and their overarching values, from the agile manifesto [4] creates a condensed focus that my results are very related to. This speaks to both the impressive processes and structures of the observed teams, as well as the relevancy of the grounded theory presented in *Chapter 5 - Results*.

6.3. Mutual Respect

My results suggest that one of the key factors for *morale*, is to establish good personal relations, and have respects for each other and each others work. This is helped by having a *flat structure* with few "bosses" and little hierarchy, together with a coaching team-leader. Gittell talks about the importance of having mutual respect [9], and one of the ways to achieve this is through eliminating status differences (or status boundaries as Gittell calls it) and hierarchies. Creating good personal relations, and not taking other peoples work for granted are additional factors that Gittell mentions are needed to achieve this mutual respect in the workplace. All of this is addressed in the findings in my results, and suggests that the *morale*-component of the theory is important for the overall success of the teams.

6.4. Reduction of Waste

The book *Lean Software Development: An Agile toolkit* [27] presents seven types of wastes, and how to prevent them. I want to especially focus on two of them, namely *Context Switching* and *Handoffs*.

Handoffs represent whenever something is handed from one entity to another [27]. This can mean between people, and it can be between teams. Extra work is created from this transition, and that is assuming that no information gets lost during communication. Reducing handoffs is thus an important waste-reduction method in lean methodologies, and my results suggest similar findings. Being autonomous and cross-functional ensures that the team does not have to hand off information, code, or products to other teams for

further development, but instead they can do everything within the team. The amount of handoffs could be further reduced if everyone in the team could do development and testing (*redundancy*), but as an interviewee mentioned it is not necessarily a good thing that people test their own code, as they are likely to miss bugs and problems because they only view it from their own angle. Having someone else test your code increases the chances of bugs being found, and is worth this extra handoff-cost.

Context-Switching represent interruptions that lead to a team-member having to drop focus on one thing in order to focus on something else [27]. The problem arises when the team-member goes back to working on the thing she was working on beforehand, as it requires a lot of work mentally to get back to where they were before the interruption. If there is a lot of such context-switching throughout a workday then the team-members can become exhausted from this increased mental workload, as well as a significant amount of time will be wasted for nothing, as my results indicate. Limiting the amount of context-switching should be on everyone's mind, as it is quite prevalent due to the *open communication nature* that the teams has, coupled with the *flat structure* and *low threshold for asking question and help*. Balancing the amount of necessary communication against the goal of reducing context switching is a difficult but important task to reduce waste.

6.5. Implications for Practice

Figuring out which characteristics positively impacts the efficiency of software development teams fueled my motivations for this thesis, and is what the scope of this thesis revolves around. Thus the results of this thesis are highly relevant for the industry, and are presented in such a way that they can be implemented in a way that suits the teams' current situation. There are natural steps that can be followed, based on the results in the previous chapter.

First, the teams and the organizations have to instill and adopt the right *mindset*. This mindset is essential if the organization wishes to adopt changes that serve the organization positively in the long run. Once such a mindset has been sufficiently established, the teams need to have people who pushes for change. These people, or promoters, are important to start the process. They are often highly knowledgeable people, who have a specific interest in continuous improvement wherever they see it. These promoters have to be nourished and appreciated, as they are the driving forces of change in an organization. Once they have started the process of implementing change, it is easier for other interested parties to join, but someone have to take the first step. Furthermore, there needs to be arenas for knowledge sharing and information sharing, such that people can influence each other and improvement can be facilitated. Lastly, a way to motivate team-members and employees is to give them increased responsibility. This enhances their feelings of ownership of their work, which in turn leads to better quality, as humans are proud people and if other team-members rely on you then you are more likely to do a thorough job.

Once such a *mindset* has been established, structural change needs to happen. If

the goal is higher autonomy, then the teams will have to become cross-functional, as otherwise autonomy will not happen. Teams must become self-governing entities, to the extent that is possible in the given organization, and for this to happen the teams must have expertise within all roles of software development. An important point to remember is to differentiate between a team as an entity being cross-functional (meaning they have team-members of all roles), and for team-members themselves to be cross-functional (also called *redundancy*). This is important to ensure that everyone is on the same page, and that everyone has the same goal in mind to prevent misunderstandings of scope. Once cross-functionality is in place, the teams can start to become autonomous.

Moreover, teams must become *flexible* in order to become autonomous. Freedom to choose tasks, to prioritize, and to take responsibility (all within reason), are important steps related to flexibility. However, flexibility is not a static set of characteristics that fit for all teams, on the contrary, flexibility means that each team must figure out where their needs are, and ensure that they have flexibility where it matters for them. As such, these main categories (or concepts) are abstract in the sense that they have to be applied differently depending on the relevant organization and teams. That being said, one characteristics that the results suggest is paramount is having a flat structure. If this is not the case, then autonomy becomes a very difficult goal to achieve.

All of this must be supplemented with a coaching team-leader who serves the team with whatever the team needs, and acts as a link between the organization as a whole, the customer, and the team itself on specifically administrative matters. Alongside the coaching team-leader, the team itself must have a good *morale*. How this is achieved depends on the team-members and their social preferences, but it is important that people enjoy coming in to work, that they are friendly towards each other, and last but not least that they *respect* each other's work and as people.

6.6. Implications For Theory

For people and researchers looking to use this theory for their own research needs, it is paramount that they keep in mind that the six central categories (flexibility, autonomy, cross-functionality, mindset, morale, and coaching) are abstract. This means that the sub-categories below them reflects the current organization, and that different organizations will have different needs, and in turn use different methods to achieve the same goals. Thus the central categories are somewhat malleable as they have to fit different types of businesses, organizations, structures and company traditions. Furthermore, the categories do not exist in a vacuum. They influence each other, and as such the theory is not complete if one of them is left out. Ignoring parts of the theory, will lead to poor results, as they all serve a purpose that positively influences the overall goal of increasing *Team Efficiency*. Lastly, the researcher must keep in mind the vast amount of different methods and practices that software development teams use, and not become overwhelmed by this large variation. The central categories are general and abstract enough to accommodate this variability, and the theory does not demand that a certain framework (like Scrum or Kanban) is used by the teams in order for the theory to be

applicable.

6.7. Validity

In this section I will detail how I took *reliability*, *internal validity*, *external validity*, and *construct validity* into consideration throughout the work on this thesis.

6.7.1. Reliability

Reliability is important to ensure that your evidence is reproducible, and can be made so by having multiple sources of evidence for the claims that you make. I followed this line of thinking, as all of my significant claims in the results-chapter are backed up by several quotes, together with observational data. Reliability is also important as a counter-measure to bias by the researcher, and ensures objectivity (to the extent that objectivity can be achieved by one researcher, in one thesis).

6.7.2. Internal Validity

Somewhat related to reliability, is internal validity. It means ensuring that the results of the research is valid to "internally" to the specific research (in this case this thesis). To protect against threats to internal validity I have used multiple sources of evidence before any claims have been made. Additionally I used different kinds of evidence, such as both interview data (quotes), and observational data.

6.7.3. External Validity

The question of external validity, or generalizability, is a difficult one within software engineering. This is because different organizations are so vastly different, with different team structures, different processes, and different mindsets. One of the measures I took to reduce threats to external validity is to present a theory with abstract and general categories that can be implemented by different kids of organizations. I do not require organizations to use specific frameworks, like Scrum or Kanban. However, a way to properly reduce threats to external validity which is not possible for me, is to conduct the research across multiple organizations. This would positively impact external validity.

6.7.4. Construct Validity

By using widely used evidence-gathering methods such as observation and semi-structured interviews, coupled with a thorough usage of the Grounded Theory Method, I improved the construct validity. A potential threat to construct validity is that I did not have the opportunity to verify my findings with the team-members. To counteract this possible threat, I put a higher value on quotes and interviews data, than observational data, as observational data could potentially have a higher risk of subjectivity or bias by me. Additionally, I was very careful not to mischaracterize or abuse the data, and did not

twist it to fit a certain narrative. If any interview answers were ambiguous, I refrained from using it as evidence.

6.8. Limitations

The most significant limitation is in my opinion time-limitations. Had this been a multi-year grounded theory study then a lot of interesting doors would have opened. I could have been part of multiple groups of teams within the organization and gotten a broader data-set. With this I could have drawn parallels and comparisons between the different teams in a larger scope than I was able to by being part of only two teams, as I was, which could have lead to a stronger theory. Furthermore, if I had more time I could have brought back the final theory and tested it more thoroughly on the teams, to let any holes or insufficiencies emerge and be filled. More rounds of interviews could also have provided me with more topics and examples, although this is merely speculation, as it is difficult to know when proper *theoretical saturation* has been achieved.

7. Conclusion

In this thesis I conducted a grounded theory study of two software development teams at a major Norwegian bank, during a period between January 2019 to May 2019. The scope of the thesis was to figure out characteristics, practices, processes, and structures for how software development teams become efficient. I have presented background information with research relevant to the thesis objective, followed by a detailing of the methodology usage of grounded theory coupled with a literature review on grounded theory usage in software engineering. Furthermore I described the organization in which the thesis work was done. It became important to have a broad scope, which is reflected by the broad theory presented as the results of this thesis. This theory is hierarchical in nature, made up of categories/concepts that represent the underlying sub-categories which are detailed in depth, together with a detailed relationship between the central categories themselves. Then I discussed my findings in light of relevant research literature, and presented implications for theory and practice. Lastly I reflected on validity and limitations of the study.

The theory presented is an answer to the first research question (RQ1) which asks which characteristics of a software development teams are necessary for increased efficiency and future success. *Team Efficiency* is seen as the ultimate goal which all of the central categories aim to positively influence together. There are six of these central categories, and each of them have several underlying sub-categories/topics (in total 29). This hierarchy, together with the relationship between the central categories comes together as a theory for how to implement efficient software engineering teams. Furthermore, the theory aims to answer the second research question (RQ2), which asks how cross-functionality and team autonomy influences the teams' success, and how they can be effectively implemented. These two factors naturally emerged as central categories from data analysis, which significantly backs up the claim that cross-functionality and team autonomy are essential for increased team efficiency and improvement.

In short, the theory can be summarized as follows: The teams and the organization must have a *mindset* that is open and welcoming towards new ideas, change, and improvement. This is essential to be able to successfully implement the necessary changes that are required. These changes include a structual change to become *cross-functional* and implementing a *flat organization structure*, as well as a change in processes and procedures with a focus on *flexibility*. These two (cross-functionality and flexibility) in turn impact and allow for team-autonomy to take place. All of this should be facilitated by the *coaching* team leader who serves as a link between the administration and "higher-ups" in the company, and the team itself. The coaching team-leader should guide the team forward, give increased responsibility to those who can handle it, and push the team forward while shielding the team from bureaucracy and administrative

issues. Maintaining a positive team morale throughout, and after, this process is an important factor, and the final central category.

There is a central topic which needs further research and understanding which I did not have time to dig into, and this is the role of the Product Owner (PO). Whether the PO should be part of the team or not, what his responsibilities should be, and what his role should be in general are highly relevant and important questions that needs further research. This is not only my observation and impression, but at a meeting for team-leaders that I attended where the topic of discussion was improvements and clarifications, the number one question that the team-leaders had was what the proper role of the PO should be. There was a lot of variability in what the PO did for the different teams, which only fueled the confusion surrounding the PO's proper purpose.

On a more general note, future research can use the theory presented in this thesis in two ways: Either they can use the theory as a framework for case studies or ethnographic research (like I detailed in the section on *Implications for Theory* in *Chapter 6 - Discussion*), or they can attempt to verify or build upon the theory in other organizations and businesses, using similar methods.

A. Literature Review Articles

1. Abad, Requirements engineering visualization: A systematic literatur review, 2016
2. Alsahli, Toward an agile approach to managing the effect of requirements on software architecture during global software development, 2016
3. Barke, Some reasons why actual cross-fertilization in cross-functional agile teams is difficult, 2018
4. Bass, Artefacts and agile method tailoring in large-scale offshore software development programs, 2016
5. Bass, Large-scale offshore tailoring: Exploring product and service organisations, 2016
6. Baum, Factors influencing code review processes in industry, 2016
7. Bhatti, Global monitoring and control: A process improvement framework for globally distributed software development teams, 2017
8. da Cunha, Decision-making in software project management: A qualitative case study of private organizations, 2016
9. de Franca, Characterizing DevOps by hearing multiple voices, 2016
10. Fonseca, Describing what experimental software engineering experts do when they design their experiments, a qualitative study, 2017
11. Gandomani, Agile transition and adoption human-related challenges and issues: A grounded theory approach, 2016
12. Garousi, Challenges and best practices in industry-academia collaborations in software engineering: A Systematic literature review, 2016
13. Ghanbari, Seeking technical debt in critical software development projects: An exploratory field study, 2016
14. Ghobadi, Risks to effective knowledge sharing in agile software team: A model for assessing and mitigating risks, 2017
15. Giardino, Software development in startup companies: The greenfield startup model, 2016

16. Gralha, The evolution of requirements practices in software startups, 2018

17. Hoda, Becoming Agile: A grounded theory of agile transitions in practice, 2017

18. Hoda, Multi-level agile project management challenges: A self-organizing team perspective, 2016

19. Jovanović, Transition of organizational roles In agile transformation process: A grounded theory approach, 2017

20. Krancher, Key affordances of platform-as-a-service: Self-organization and continuous feedback, 2018

21. Küpper, The impact of agile methods on the development of an Agile culture - research proposal, 2016

22. Lavallée, Are we working well with others? How the multi team systems impact software quality, 2018

23. Márcuez, Identifying emerging security concepts using software artifact through an experimental case, 2015

24. Martini, A multiple case study of continuous architecting in large agile companies: current gaps and the CAFFEA framework, 2016

25. Masood, Self-assignment: Task allocation practice In agile software development, 2017

26. Murugesan, Overcoming challenges in self-organizing agile software teams

27. Prechelt, Quality experience: a grounded theory of successful agile projects without dedicated testers, 2016

28. Rahman, Characterizing defective configuration scripts used for continuous deployment, 2018

29. Ralph, Toward methodological guidelines for process theories and taxonimies in software engineering, 2018

30. Rashid, Discovering "unknown known" security requirements, 2016

31. Regassa, Agile methods in ethiopia: An empirical study, 2017

32. Salleh, Recruitment, engagements and feedback in empirical software engineering studies in industrial contexts, 2018

33. Sánchez-Gordón, Understanding the gap between software process practices and actual practice in very small companies, 2016

34. Santos, Towards a theory of simplicity in agile software development : A qualitative study, 2017

35. Schramm, Implementations of service oriented architectuer and agile software development: What works and what are the challenges?, 2016

36. Sedano, Lessons learned from an extended participant observation grounded theory study, 2017

37. Sedano, Practice and perception of team code ownership, 2016

38. Sedano, Software development waste, 2017

39. Sedano, Sustainable software development through overlapping pair rotation, 2016

40. Song, How to support customization on SaaS: A grounded theory from customisation consultants, 2017

41. Stray, The daily stand-up meeting: A grounded theory study, 2016

42. Tyagi, Adopting test automation on agile development projects: A grounded theory study of indian software organizations, 2017

43. Vishnubhotla, Designing a capability-centric web tool to support agile team composition and task allocation: a work in progress, 2018

44. Würfel, Grounded requirements engineering: An approach to use case driven requirements engineering, 2016

45. Zayour, A qualitative study on debugging under an enterprise IDE, 2016

46. Zieris, Observations on knowledge transfer of professional software developers during pair programming, 2016

B. Interview Guide

- Background Questions:
  - How long have you been on this project?
  - Can you tell me a bit about your role?
  - Can you tell me about your daily work-activities?
  - Do you have any relationship with the words *autonomy* or *cross-functionality*?
  - If yes, how do they impact the team?
- Teamwork and Coordination:
  - How do you achieve team-orientation in autonomous teams? Meetings etc.
  - What do you think about meetings?
  - What can be improved? Are some meeting unproductive?
  - In the retrospective report at 10.10.18 you said that standups were too long, then you fixed it. Can you tell me about this process?
  - What do you think about the standups now?
  - If you are stuck at a task, what do you do?
  - Who do you coordinate with on a daily/weekly basis?
  - What is the difference between autonomous and cross-functional teams, and other team structures you have been a part of?
- Leadership:
  - How do you define leadership?
  - How would you describe the leadership in this team, and in this organization?
  - How do you feel about having autonomy, but also having a team-leader?
  - How is the balance between autonomy and leadership?
- Knowledge-Sharing and learning:
  - How do you facilitate for knowledge sharing and learning in autonomous cross-functional teams?
  - How do the team-members react to input and criticism?
  - How open are team-members to changes in structure, process, technologies and strategy? (One at a time)

- Customer Relations:
  - To what extent do you communicate with the customer?
  - Do you feel like you as a team are being shielded too much or too little from the customer?
- Agile Processes:
  - In what way do you have agile processes?
  - Have you changed the way you are working, and if so, what do you feel about the result?
- To Finish:
  - How do you feel in general about being a part of this team? (very open question)
  - Whats the biggest source of waste? if any.
  - Sources of frustration? Regarding tools, processes, structure etc.
  - Any final remarks that you want to mention?

Bibliography

[1] Muhammad Ovais Ahmad, Jouni Markkula, and Markku Oivo. "Kanban in software development: A systematic literature review". In: *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*. IEEE. 2013, pp. 9–16.

[2] David J Anderson. *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010.

[3] Kent Beck. "Embracing change with extreme programming". In: *Computer* 32.10 (1999), pp. 70–77.

[4] Kent Beck et al. "Manifesto for agile software development". In: (2001).

[5] Barry W. Boehm. "A spiral model of software development and enhancement". In: *Computer* 21.5 (1988), pp. 61–72.

[6] Barry W Boehm et al. *Software engineering economics*. Vol. 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[7] Kathy Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis*. Sage, 2006.

[8] Juliet M Corbin and Anselm Strauss. "Grounded theory research: Procedures, canons, and evaluative criteria". In: *Qualitative sociology* 13.1 (1990), pp. 3–21.

[9] Jody Hoffer Gittell et al. "Relational coordination: coordinating work through relationships of shared goals, shared knowledge and mutual respect". In: *Relational perspectives in organizational studies: A research companion* (2006), pp. 74–94.

[10] Barney G Glaser and Anselm L Strauss. *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 1967.

[11] Nils Christian Haugen. "An empirical study of using planning poker for user story estimation". In: *Agile Conference, 2006*. IEEE. 2006, 9–pp.

[12] Rashina Hoda and Latha K Murugesan. "Multi-level agile project management challenges: A self-organizing team perspective". In: *Journal of Systems and Software* 117 (2016), pp. 245–257.

[13] Rashina Hoda, James Noble, and Stuart Marshall. "Developing a grounded theory to explain the practices of self-organizing Agile teams". In: *Empirical Software Engineering* 17.6 (2012), pp. 609–639.

[14] Rashina Hoda, James Noble, and Stuart Marshall. "Self-organizing roles on agile software development teams". In: *IEEE Transactions on Software Engineering* 39.3 (2013), pp. 422–444.

[15] Rashina Hoda, James Noble, and Stuart Marshall. "The impact of inadequate customer collaboration on self-organizing Agile teams". In: *Information and Software Technology* 53.5 (2011), pp. 521–534.

[16] Emam Hossain, Muhammad Ali Babar, and Hye-young Paik. "Using scrum in global software development: a systematic literature review". In: *Global Software Engineering, 2009. ICGSE 2009. Fourth IEEE International Conference on*. Ieee. 2009, pp. 175–184.

[17] Magne Jørgensen. "A review of studies on expert estimation of software development effort". In: *Journal of Systems and Software* 70.1-2 (2004), pp. 37–60.

[18] Barbara Kitchenham and Stuart Charters. "Guidelines for performing systematic literature reviews in software engineering". In: (2007).

[19] Craig Larman. *Scaling lean & agile development: thinking and organizational tools for large-scale Scrum*. Pearson Education India, 2008.

[20] Lucy Ellen Lwakatare, Pasi Kuvaja, and Markku Oivo. "Dimensions of devops". In: *International Conference on Agile Software Development*. Springer. 2015, pp. 212–217.

[21] Nils Brede Moe, Torgeir Dingsøyr, and Tore Dybå. "Understanding self-organizing teams in agile software development". In: *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE. 2008, pp. 76–85.

[22] Kjetil Moløkken-Østvold, Nils Christian Haugen, and Hans Christian Benestad. "Using planning poker for combining expert estimates in software projects". In: *Journal of Systems and Software* 81.12 (2008), pp. 2106–2117.

[23] Nabil Mohammed Ali Munassar and A Govardhan. "A comparison between five models of software engineering". In: *IJCSI* 5 (2010), pp. 95–101.

[24] Michael D Myers et al. "Qualitative research in information systems". In: *Management Information Systems Quarterly* 21.2 (1997), pp. 241–242.

[25] Taiichi Ohno. *Toyota production system: beyond large-scale production*. crc Press, 1988.

[26] Wanda J Orlikowski. "CASE tools as organizational change: Investigating incremental and radical changes in systems development". In: *MIS quarterly* (1993), pp. 309–340.

[27] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit: An Agile Toolkit*. Addison-Wesley, 2003.

[28] Nicholas Roberts and Varun Grover. "Investigating firm's customer agility and firm performance: The importance of aligning sense and respond capabilities". In: *Journal of Business Research* 65.5 (2012), pp. 579–585.

[29] Winston W. Royce. "Managing the development of large software systems". In: (1970).

[30] Ken Schwaber. *Agile project management with Scrum*. Microsoft press, 2004.

[31] Ken Schwaber. “Scrum development process”. In: *Business object design and implementation*. Springer, 1997, pp. 117–134.

[32] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River, 2002.

[33] Jens Smeds, Kristian Nybom, and Ivan Porres. “DevOps: a definition and perceived adoption impediments”. In: *International Conference on Agile Software Development*. Springer. 2015, pp. 166–177.

[34] Ian Sommerville. *Software engineering*. New York: Addison-Wesley, 2010.

[35] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. “Grounded theory in software engineering research: a critical review and guidelines”. In: *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE. 2016, pp. 120–131.

[36] Anselm Strauss and Juliet Corbin. “Grounded theory methodology”. In: *Handbook of qualitative research* 17 (1994), pp. 273–85.

[37] Viktoria Gulliksen Stray, Nils Brede Moe, and Aybüke Aurum. “Investigating daily team meetings in agile software projects”. In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE. 2012, pp. 274–281.

[38] Viktoria Stray, Nils Brede Moe, and Rashina Hoda. “Autonomous agile teams: Challenges and future directions for research”. In: *arXiv preprint arXiv:1810.02765* (2018).

[39] Viktoria Stray, Dag IK Sjøberg, and Tore Dybå. “The daily stand-up meeting: A grounded theory study”. In: *Journal of Systems and Software* 114 (2016), pp. 101–124.

[40] Y Sugimori et al. “Toyota production system and kanban system materialization of just-in-time and respect-for-human system”. In: *The International Journal of Production Research* 15.6 (1977), pp. 553–564.

[41] Jeff Sutherland and Ken Schwaber. “The scrum guide”. In: *The definitive guide to scrum: The rules of the game. Scrum. org* 268 (2013).

[42] Jeff Sutherland et al. “Distributed scrum: Agile project management with outsourced development teams”. In: *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. IEEE. 2007, 274a–274a.

[43] Cathy Urquhart, Hans Lehmann, and Michael D Myers. “Putting the ‘theory’back into grounded theory: guidelines for grounded theory studies in information systems”. In: *Information systems journal* 20.4 (2010), pp. 357–381.