IT University of Göteborg

CHALMERS | GÖTEBORG UNIVERSITY

Master thesis in Applied Information Technology

REPORT NO. 2008:014
ISSN: 1651-4769

Department of Applied Information Technology or
Department of Computer Science

Bottlenecks in Agile Software Development Identified Using Theory of Constraints (TOC) Principles

ASTA MURAUSKAITE
VAIDAS ADOMAUSKAS

CHALMERS

UNIVERSITY OF GOTHENBURG

IT University of Gothenburg
Chalmers University of Technology and University of Gothenburg

Gothenburg, Sweden 2008

ABSTRACT

This master thesis identifies main bottlenecks in agile software development exemplified by research industry partner, the international advanced technology company, Ericsson. Theory of Constraints is used as an analytical tool. The research consists of three phases. First, high level bottlenecks of four agile software development methods: Lean software development, Extreme Programming (XP), Scrum, and Feature Driven Development (FDD) are identified. After that, theoretical model for identifying bottlenecks in Lean software development implementations is developed. At the end, this model is verified in a studied unit at Ericsson. Identified bottlenecks narrows down possible issues in agile software development implementations and allows focusing on the core problems. Companies working according to agile methods could benefit from using the results of the thesis to identify bottlenecks in their implementations.

Categories and Subject Descriptors

K.6.1 [**Management of Computing and Information Systems**]: Project and People Management - *Management techniques.*

General Terms

Measurement, Documentation, Performance, Theory.

Keywords

Agile software development, Lean software development, Scrum, Extreme Programming (XP), Feature driven development (FDD), Theory of Constraints (TOC), bottleneck.

1. INTRODUCTION

The increased speed and change in business world increased the need to develop software faster and cheaper as well as higher quality and more adaptable to constant change. New software development methods were developed and named as being agile.

Despite the fact that all of the agile software development methods have the same goal, each of them has a different approach. This was a base for researchers to look into differences and similarities of agile software development methods from various angles [10][14][15]. Many case studies were performed investigating if and when agile implementations work [23][24][25]. Researchers investigated how to fit agile methods for large organizations [26] and traditional development organizations [27]. Besides, they investigated even more specifically: e.g. how to manage requirements in agile processes [28].

Despite the variety of literature about agile software development, we could not find any that would discuss possible bottlenecks of agile software development. However, according to Goldratt [1] every process has a bottleneck – a weakest link in the chain that limits throughput. Identifying and eliminating it will increase throughput what leads to more profit. Therefore, our master thesis research will focus on creating a model that allows identifying bottlenecks in agile software development methods. Furthermore, Lean is chosen as a method to scrutinize in more detail. This choice is made due to a fact that our research industry partner, an international advanced technology company called Ericsson, is implementing an agile software development method in one product unit. Their method is following main Lean principles. Hence, we verify our theoretically identified bottlenecks in a studied unit at Ericsson.

The reasons above leads us to our research question:

- What are potential bottlenecks in agile software development?

As an analytical tool to achieve our research results, we choose to use Theory of Constraints (TOC). The main concept of TOC is to identify and exploit bottlenecks. Therefore, we use TOC thinking principles to identify possible bottlenecks in agile software development projects.

The thesis is organized this way. Chapter 2 describes theories and methods used in research. In chapter 3 we describe the research method and reasoning behind it. The results are described in chapter 4. The validity of the results is discussed in chapter 5. Finally, the conclusions are presented in chapter 6.

2. THEORY OVERVIEW

This chapter briefly describes all methods used in this master thesis. It consists of three parts. First, we present a short overview of Theory of Constraints (TOC) and motivation of choosing it as an analytical tool to identify possible bottlenecks in agile software development. Afterwards, we present a general definition of agile software development (subchapter 2.2). Finally, we describe analyzed agile software development methods (subchapter 2.3): Lean Software Development, Extreme Programming (XP), Scrum, and Feature Driven Development (FDD). These descriptions should help the reader to get broad overview of different agile methods as well as their similarities and differences.

2.1 Theory of Constraints (TOC)

Goldratt developed an approach for continuous improvement called Theory of Constraints (TOC), introduced in the book "The Goal" [2]. TOC was applied for production and manufacturing operations management. Goldratt's later books extended the application of the theory to other fields such as sales, marketing and production distribution [3]; project management [1]; and supply chain management [4]. We apply TOC thinking principles to identify potential bottlenecks in agile software development in this thesis.

TOC is a prescriptive theory [9], which means that it provides answer to the question what the constraint of the system is. Besides, it has developed tools to make logical decisions how to deal with them [5][6]. TOC enables managers to answer three fundamental questions about the change:

- WHAT to change?
- What to change TO?
- HOW to cause the change?

These questions are system-level, not process-level questions. They are designed to focus efforts on the whole system improvement. Undoubtedly, they will have impact on individual processes (positive or even negative), but the aim is to improve system as a whole.

A system is a project or a portfolio of projects in software development environment. This means that TOC focuses on bottlenecks which allow increasing throughput of a project or a

project portfolio. Exploiting identified bottlenecks will definitely affect internal activities within the project. It might even make them less efficient. Despite that, the throughput of the system as a whole (project or project portfolio) will be increased.

2.1.1 TOC Principles

Following paragraphs describe some of TOC principles, defined by Dettmer [9] and used by us to identify bottlenecks in agile software development:

1. *System as "chains".*

   TOC views every system as a chain or a system of chains (e.g. all tasks that have to be accomplished in particular order to finish a software project). This is essential way of thinking as it implies that every chain has the weakest link – a bottleneck. Furthermore, at the particular point of the time there is the only one weakest link, which enables clear focus. The weakest link (bottleneck) can be found and strengthened. Working only with the weakest links will improve the system (chain) as a whole.

2. *Cause and effect*

   Every system exists in cause-effect relations. Something happens (the effect) because something else has happened (the cause). TOC provides tools and a thinking process to employ cause-effect relations to represent our complex environments. They are visually presented as trees.

   For example, if our goal is to have an employee who can write the code, he has to be educated and he has to have tools. Educated person has to have theoretical knowledge as well as practical experience. This small example would be presented by TOC in the following cause-effect tree (see Figure 1). It means that in order to achieve higher branches in a tree, all lower ones must be implemented.

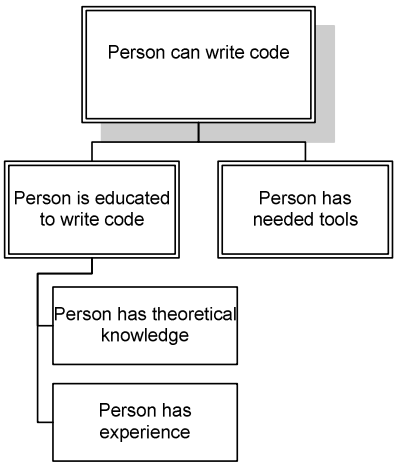

Figure 1. An example of TOC cause-effect tree

   In order to read the tree, if-then logics should be used. The tree presented in Figure 1 should be read: "IF a person has theoretical knowledge AND a person has experience THEN a person is educated to write the code". "IF a person is educated to write the code AND a person has tools THEN a person can write the code".

   We will use this cause and effect principle and trees to connect and visualize possible bottlenecks in agile software development (see subchapters 4.1 and 4.2).

3. *Undesirable effects and core problems*

   Almost everything found in a system as problems are actually undesirable effects. It is not the root of the problem (the core problem). Solving undesirable effects gives false security feeling that a problem is solved. Nevertheless, the existing problem has a tendency to appear again as a core problem still exists in a system. Only after the core problem is solved, the undesirable effect, that was a bottleneck in the first place (as well as the other undesirable effects that rose from the core problem), is actually solved and prevented from returning.

   For instance, we think that a person in our example presented in Figure 1 is not educated enough. This is an undesirable effect as a person cannot write the code. A core problem is either a person not having theoretical knowledge or a person not having experience. If a person does not have experience, but a company sends a person to a theoretical programming class the undesirable effect will remain. Only solving the core problem, training a person with practical exercises, will help us to achieve the goal: have employee who can write the code.

   Identifying core problems, not undesirable effects, in particular situations means identifying the bottlenecks. We will use this principle to identify possible bottlenecks in Lean software development (see subchapter 4.2).

4. *Physical vs. policy bottlenecks*

   Physical bottlenecks are relatively easy to find and break. However, most real bottlenecks that exist in systems are policy bottlenecks. Most commonly, physical bottlenecks are just a result of policies and rules in organization. Policy bottlenecks are much more difficult, but normally breaking them resolves in much larger improvements.

   Software development is not an exception. For example, developers decide to use a tool for writing standard comments in their integrated development environment (IDE) and then automatically transforming them to software documentation. All software documentation policies in company have to be reviewed and changed accordingly. If not, new policy bottleneck may be created: an old software documentation policy will require an old type of documentation at the same time when a new one is generated. This means that a new initiative will add more work to a project, rather than improve it.

   To follow this TOC principle, while identifying possible bottlenecks in agile software development implementation in a studied unit at Ericsson, we will look more carefully for possible policy bottlenecks rather than physical ones.

2.1.2 The Five Focusing Steps

Goldratt has developed TOC to enable a continuous improvement process [2]. When an organization knows its goal and understands the concept of a bottleneck it should follow the five focusing steps continuously to adjust improvements to changing environment [7]. We will not be using these TOC Five Focusing Steps for this master thesis research, as our goal is to identify possible bottlenecks (only step 1). Despite that, the ones that will use our research results should follow these steps in order to break identified bottlenecks and to continually improve their agile software development projects.

The five focusing steps are:

1. *Identify the system bottleneck*

   Find the weakest link in the system of chains. Remember, that there is only one weakest link at a given point of time. Look carefully for policy bottlenecks even if it is easier to find a physical bottleneck.

2. *Exploit the bottleneck*

   When a bottleneck is found it is essential to assure that it works 100% and all activities which do not directly add value to the tasks of a bottleneck has to be eliminated. This step enables to increase capacity of a bottleneck resource without additional investment.

3. *Subordinate everything else to the above decision*

   After performing step 2 (exploiting the bottleneck) all the rest of the system has to be adjusted to enable a bottleneck to operate at a maximum effectiveness. It might include changing rules, procedures, reassigning some tasks of a bottleneck resource for non bottleneck resources, and other possible subordination.

4. *Elevate the system's* bottleneck

   This step is reached in case steps 2 and 3 did not break the bottleneck (internal system adjustments were not sufficient to break the bottleneck). Elevating the bottleneck means doing whatever it takes to break it. That usually involves investment in money, time, energy or other resources. Therefore this step should be executed only after doing everything that is possible in steps 2 and 3.

5. *Go back to Step 1, but do not allow inertia to become a system* bottleneck.

   There is always the weakest link in a chain (a bottleneck). If a bottleneck is broken in step 3 or 4 it is a must to come back to step 1 and start looking for a new bottleneck. This is the process of continuous improvement which never ends. It provides with a strategy always to focus on current bottlenecks. It also reminds that it is important not to allow inertia to become a system bottleneck: even already broken bottlenecks might become bottlenecks again due to changing environment, so they have to be revised continuously as well.

2.1.3 Motivation to Choose TOC for the Research

There are four main motivation factors why we chose to use TOC and its principles described in paragraph 2.1.1 for this master thesis. First, TOC principles enable to view agile software development as system of chains. Second, they allow modelling agile software development principles and practices into trees with cause-effect relations. Third, TOC principles enable to identify main bottlenecks (core problems vs. undesirable effects, policy vs. physical bottlenecks). Finally, TOC allows to focus on core problems and "to channel improvement efforts for maximum immediate effect" [9]. It also provides tools to do that: TOC five focusing steps described in paragraph 2.1.2. This means that output of our master thesis research, identified possible bottlenecks, can be immediately reviewed and exploited in a company to achieve fast results. Furthermore, it will create a process of continuous improvement in a company.

This subchapter gave a short overview of a theory that is used to conduct a research. The following subchapter will present a short overview of agile software development in general.

2.2 Agile Software Development

Agile software development emerged as an alternative to document-driven, rigorous software development processes [14]. Software developers realized that processes which require many documents, artefacts, and procedures to follow is too slow to fulfil customer needs. Moreover, business needs nowadays change faster than software projects following old methods are able to implement them. Therefore, the focus had to switch from fulfilling well predefined project requirements to delivering up to date value to the customer.

2.2.1 Manifesto for Agile Software Development

A common ground for agile software development was defined in 2001, when 17 experienced and recognized software development "gurus", inventors and practitioners of different agile software development methods gathered together. Participants agreed and signed The Manifesto for Agile Software Development [14]. This manifesto declares the main values of agile software development:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions* over processes and tools
- *Working software* over comprehensive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

That is, while there is value in the items on the right, we value the items on the left more." [11]

2.2.2 Principles Behind the Agile Manifesto

Manifesto for Agile Software Development is followed by 12 principles. In this master thesis we assume, that these principles are important to consider for software development process to be recognized as agile. We do not question their validity or sufficiency and accept them as it is. We use these principles as a base for identifying possible bottlenecks in different agile software development methods (see subchapter 4.1).

Principles behind the Agile Manifesto are [11]:

1. *Satisfy the customer:*

   Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. *Welcome changing requirements:*

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. *Deliver working software frequently:*

   Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. *Motivate individuals:*

   Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

5. *Interact frequently with stakeholders:*

   Business people and developers must work together daily throughout the project.

6. *Communicate face to face:*

   The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. *Measure by working software:*

   Working software is the primary measure of progress.

8. *Maintain constant pace:*

   Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. *Sustain technical excellence and good design:*

   Continuous attention to technical excellence and good design enhances agility.

10. *Keep it simple:*

    Simplicity, the art of maximizing the amount of work not done, is essential.

11. *Empower self-organizing teams:*

    The best architectures, requirements, and designs emerge from self-organizing teams.

12. *Reflect and adjust continuously:*

    At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

We will use these principles to identify possible high level bottlenecks of agile software development methods (see subchapter 4.1).

Subchapter above shortly presented main values and principles of agile software development. The following subchapter will present four specific agile software development methods used in this master thesis research.

2.3 Agile Software Development Methods

There is a number of software development methods that follow the values and principles described above. They all fall under agile software development methods classification. For this master thesis research, due to time and scope constraints, we decided to choose four agile software development methods for further analysis. They are: Lean software development, Extreme Programming (XP), Scrum, and Feature Driven Development (FDD). We choose them due to different approaches they have to achieve agile goals. Lean development is about reducing the development timeline by removing all no value-adding wastes [29]. Besides, it is the closest method to current agile software development method Ericsson is implementing [29]. XP is one of the most agile methods that take common sense software engineering practices to the extreme level [31]. We choose Scrum because of its strong focus on self organizing teams, daily team measurements, and avoidance of predefined steps [15]. FDD, unlike other agile software development approaches, encourages an up-front architectural modelling and accomplishes core goals in different ways [14].

Further subchapters will shortly introduce these methods describing their proposed development processes and main principles they follow. In the end, principles are presented in a tree that is based on TOC "Cause and effect" principle described in paragraph 2.1.1. "If then" logics is used to read the tree. As an example see Figure 1.

2.3.1 Lean Software Development

Lean Software Development is an agile development method that applies Lean production principles which were created in Toyota Motor Company in 1980s to software development [16][22]. "Lean thinking focuses on giving customers what they want, when and where they want it, without a wasted motion or wasted minute" [21].

Lean Software Development suggests following iterative style of development, that creates incremental results at a steady pace. Lean Software development process is composed of four phases:

1. Preparation
2. Planning
3. Implementation
4. Assessment

At the beginning of development effort an initial backlog of prioritized desirable stories (features) is assembled. This is the preparation phase. Backlog items are usually features in terms of business goals since the Lean approach is to delay detailed analysis until the last responsible moment.

Planning meeting is held at the beginning of iteration. The whole team makes estimations how long the top priority stories from backlog will take to develop, test, document and deploy. According to these estimations and team capacity they pick the amount of stories they will be able to implement during the iteration. Team members decide and commit to iteration goal, which describes the theme of the feature set they picked for iteration.

During implementation phase a team develops, tests, documents and prepares for deployment the feature set they picked. Daily 10-15 minute team meetings are held to discuss what each team member has accomplished since the last meeting, what they will be doing till the next meeting, what problems they have, and where they need help. A story is not considered done until the team updates all associated artefacts (user documentation, design documents and other artefacts).

A review meeting is held at the end of iteration. The goal of the meeting is to show for the customer how much value was added to the product during the iteration. Feedback from customer is

collected to make changes if needed. After this iteration assessment, planning meeting starts for the next iteration.

Lean Software development has 7 main principles (see Figure 2). "As a group, these principles provide guidance on how to deliver software faster, better, and for less cost – all at the same time." [21]

- *Eliminate waste* – remove everything that does not create a clear value for a customer (product).
- *Build Quality In* – focus on eliminating defects as soon as they are detected; avoid creating defects in the first place.
- *Create Knowledge* – encourage systematic learning throughout the development cycle and make sure that tacit knowledge is shared.
- *Defer Commitment* – schedule irreversible decisions for the last responsible moment, that is, the last chance to make the decision before it is too late.
- *Deliver As Fast As Possible* – deliver software so fast that your customer would not have time to change their minds.
- *Empower the Team* – develop an organization where each person has an authority to prioritize, take responsibility and come up with solutions instead of having someone telling what to do and how to do it.
- *Optimize the Whole* – optimize the whole value stream from the time it receives an order to address a customer need until software is deployed and the need is addressed; avoid suboptimization.

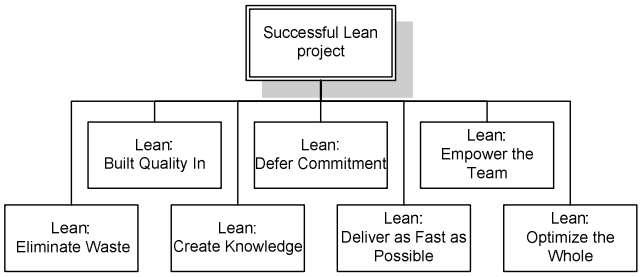

Figure 2. Lean Software Development Principles

2.3.2 Extreme Programming (XP)

Extreme Programming (XP) is an agile development method that value simplicity, feedback, community, and courage.[14] [15].

XP development process consists of three main iterative phases:

1. Planning
2. Development
3. Acceptance

At the beginning of a project Planning Game is held where the project is divided into iterations of 1 to 3 weeks. Story Cards that represent features are created and the project releases dates are set. Each release starts with a half day Release Planning Game where Story Cards are reviewed, estimated, and prioritized by a customer. Every iteration begins with Iteration Planning Game where the customer chooses which Story Cards should be implemented in the iteration. Furthermore, a task list is created and team members choose the tasks they want to work next.

Development phase starts with the high level design sketch on a whiteboard. Programming is held in pairs where both team members have the same responsibility for the code. All code is continuously integrated and tested on a separate machine.

In the acceptance phase all code is tested with automated acceptance test that is defined by a customer. A review meeting is held to get the feedback.

XP software development recommends these principles [14] [15] (see Figure 3):

- *Planning game* – a planning session where story cards are defined and prioritized together with a customer.
- *Test-first development* – a development culture where first a unit test is created and afterwards the code is written.
- *Simple design* – a design that has a main set of classes and methods and is created only when it is needed. No generalized components are created if not needed.
- *Stand up meeting* – a short 15-20 minutes daily meeting where each team member answers 3 main questions:
  - What is done so far?
  - What is planned to do until next meeting?
  - What are the obstacles to achieve iteration goals?
- *On site customer* – a working process where one or more customers are in the same room as a development team full time.
- *Continuous integration* – an integration activity where all code is continuously integrated in a common environment where the unit tests are run continuously.
- *Short releases* – an evolutionary delivery to increase suitability for business needs.
- *Acceptance test* – an automated acceptance test that is run with pass/fail result which is defined by a customer.
- *Collective ownership* – a development culture where any pair of programmers can improve any code creating an environment that no-one is blamed for mistakes in code.
- *Common room* – a working environment where the whole team is working as close as possible preferably in one room. Separate spaces are available if a team member needs it for a short while.
- *Frequent refactoring* – an effort to simplify the code to make it cleaner without changing its functionality.
- *Coding standards* – a coding style accepted by a company to ease coding and refactoring processes.
- *40 hours week* – a working culture where work is limited to working hours to increase creativity, health, and avoid overtime.
- *System metaphors* – memorable metaphors to enable better understanding about system in the design sketches.

- *Pair programming* – a development culture where all code is produced by two programmers at one computer.

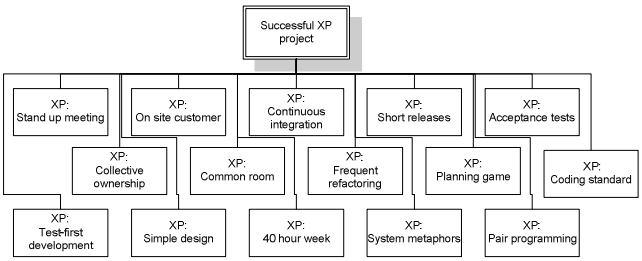

Figure 3. Extreme Programming (XP) Principles

2.3.3 Scrum

Scrum is an agile software development method that provides a management framework [14].

There are four basic phases in the process [15]:

1. Planning
2. Staging
3. Development
4. Release

The goal of the planning phase is to set the vision and expectations of a project and assign funding. This is done in pre-game planning. Moreover, the project is divided into iterations called sprints that are 30 calendar days.

In the staging phase requirements should be identified and priorities assigned for the iteration. This phase begins with sprint plan where the plan for iteration is created. External stakeholders are involved to prioritize the tasks in sprint. No more additional tasks can be added to sprint after the plan is created.

The development phase involves a system implementation in 30 days iterations and prepares it for a release. During this phase the work in sprint is divided into daily blocks that lead to daily builds. The development begins with high level design sketches. Every day a 15 minutes stand up meeting is held to update on the sprint status. During the meeting team members choose the tasks they will be working next.

During the last phase the system should be deployed. After each sprint the release meeting is held where a system to external stakeholders is presented to get feedback. After that future directions are set.

Scrum development process recommends these principles [14] [15] (see Figure 4):

- *Scrum meeting* – a short 15-20 minutes daily meeting where each team member answers 3 main questions:
  - What is done so far?
  - What is planned to do until next meeting?
  - What are the obstacles to achieve iteration goals?
- *Sprint* – 30-days iteration.
- *Pre-game planning* – a planning activity where the product backlog is created with list of features, use cases, and defects as well as product owner is assigned to ease future communication.
- *Sprint planning* – a planning activity that consists of two meetings: first, stakeholders refine and prioritize product backlog, second, team and product owners plan how to achieve iteration results and create task lists.
- *Common room* – a working environment where the whole team is working as close as possible preferably in one room. Separate spaces are available if a team member needs it for a short while.
- *Daily built* – at least one integration with a regression testing of the code in the system throughout a day.
- *Blocks gone in one day* – tasks that are finished in one day (from one Scrum meeting to the other).
- *Scrum master firewall* – Scrum master (manager) activity to assure that work in team is happening and no undesired activities exist (extra work added to sprint or any outside interruption) within the team.
- *Lock priorities within sprint* – priorities chosen at the beginning of sprint. No extra work that could be added to iteration is tolerated to maintain team focus on the goal. In case extra work is added some work should be removed.
- *Sprint review* – a meeting where a review for sprint is executed and demo of a product is presented at the end of sprint. Feedback and future directions are set during this meeting.
- *Decision is one hour* – decision making process that does not take longer than one hour. No decision is worse than a bad decision and a bad decision can be reversed.
- *High level design* – the sketch of design only to get basic understanding about the system.
- *Self-directed and self-organizing teams* – the culture where the team has authority and resources to choose the best way to achieve sprint goals, to prioritize work, and to solve its own problems.
- *Team of 7* – the team that consist of no more than seven people to assure efficiency and smooth communication.

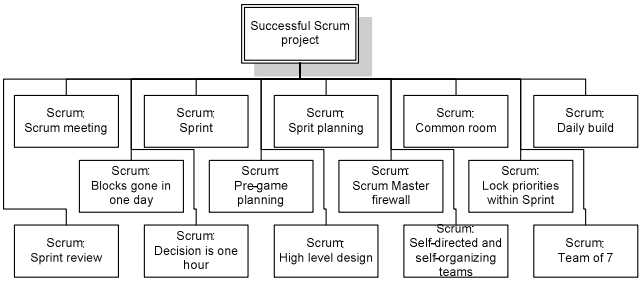

Figure 4. Scrum Principles

2.3.4 Feature Driven Development (FDD)

Feature Driven Development (FDD) is an agile software development method that values up front modelling and has "right first time" approach. [10]. It is a minimally described five steps process.[14]. First three steps are executed once in a project and

called “startup phase” while steps 4 and 5 are iterated for feature sets and called “construction phase” [17].

1. Develop an overall model
2. Build a features list
3. Plan by feature
4. Design by feature
5. Build by feature

First, the overall model is developed. The model is very brief and it contains only main classes and their connections (shape rather than the content). In larger projects the domain teams are formed and domain models are created by different teams. They are merged into overall model daily or every second day.

In the second step the complete and categorized features list is build. To compile this list the domain is decomposed into Subject Areas. Then Subject Areas are decomposed into Business Activities and the Steps (features) within each Business Activity. The size of a feature usually is from one to ten days work.

The goal of plan by feature stage is to produce a development plan. Planning Team consisting of Project Manager, Development Manager and the Chief Programmers is created. The team plans an order that features have to be implemented according to feature dependencies, complexity, and the load of development team. Chief Programmers are assigned to Business Activities and developers are assigned to own the classes.

Chief Programmer selects a feature set from his entire features list called Chief Programmer Work Package. Then he forms Feature Team by identifying the owners of the classes (developers) which will be involved in the development of a selected feature set. This team creates needed design for Chief Programmer Work Package which is refined against overall model created in the first step.

Finally, Feature Team implements Chief Programmer Work Package by following these steps: implement classes and methods, inspect the code, run unit tests, and promote to the build. After the build succeeds, new iteration from step 4 starts with a new Chief Programmer Work Package and a new Feature Team.

As every Agile software development method, FDD has main principles it follows [20] (see Figure 5):

- *Domain Object Model* – a process of creating the framework of problem domain within which features will be added.
- *Development by Feature* – a process where development is driven and tracked by decomposed list of small, client valued functions.
- *Individual Class (Code) Ownership* – a process where the consistency, performance, and conceptual integrity of each class is the responsibility of an assigned single person.
- *Feature Teams* – a process encouraging doing design activities in small, dynamically formed teams as well as encouraging evaluating multiple design options before one is chosen.
- *Inspections* – a process of defect-detection technique providing opportunities to propagate good practice, conventions, and development culture.
- *Regular Builds* – a process ensuring that there is always a demonstrable system available. It also helps to solve all synchronization issues as early in the process as possible
- *Configuration Management* – a process ensuring an easy way to identify/revert/change any version of completed source code files and other artefacts of the project
- *Reporting/Visibility of Results* – a process of frequent and accurate progress reporting at all levels, inside and outside the project, based on completed work.

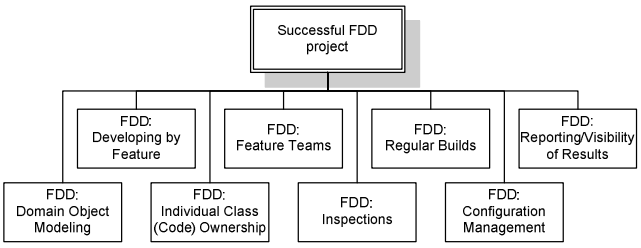

Figure 5. Feature Driven Development (FDD) principles

Chapter 2 presented a short introduction to TOC with motivation why we chose it for this master thesis research. Besides, a description of agile software development in general as well as four specific agile software development methods was given. The following chapter explains the research methods we chose to use, how we divided the research to answer our research question, and what results we expect.

3. RESEARCH METHOD

This chapter explains the methods that we use in this master thesis research. It explains how research is divided into phases, why specific methods are chosen, and how they contribute to find the answer to the main research question.

The main research question is:

- What are potential bottlenecks in agile software development?

To answer this question, the research is divided into three phases. Each phase has a question to answer. Answers to these three questions leads to the answer of the main research question.

- Question 1: What high level bottlenecks might exist in agile software development methods?
- Question 2: What bottlenecks might exist in Lean software development?
- Question 3: What bottlenecks might exist in agile software development implementation in a studied unit at Ericsson?

The research is based on Constructive research method [18] as the goal of the master thesis is to create a theoretical model using existing theory and verify it in a studied unit at Ericsson. Further subchapters explain each phase of research in more detailed. They provide descriptions and motivation for each question and the methods that are used to find the answers.

3.1 Phase 1: identify possible high level bottlenecks of agile software development methods

The goal of phase 1 is to identify high level bottlenecks of agile software development methods. In this research, we refer to high level bottlenecks as missing or not directly addressed principles and practices of agile software development methods.

To accomplish this goal, first, we have to get a clear understanding of the agile software development principles and their application in the specific agile software development method. We have to identify the main principles as well as try to understand the differences between various agile software development methods. An extensive theoretical study in books, articles, and websites of agile movement and different agile software development methods will be done. Summarized results in form of a short description of each agile software development method are presented in subchapter 2.3.

Having this knowledge allows us to define question 1:

- What high level bottlenecks might exist in agile software development methods?

To answer question 1 we make an assumption that each successful agile software development method have to address all general agile principles agreed by authors of Manifesto for Agile Software Development [11]. Despite that, we expect that different agile software development methods focus on different agile principles. We use Atlas.ti [8] software to code all principles and practices of analyzed agile software development methods. We use general agile principles agreed by authors of Manifesto for Agile Software Development [11] as codes.

Afterwards, TOC principles described in paragraph 2.1.1 are used to model the system and make visual presentation of the results. We transform coded data to a tree as described in paragraph 2.1.1 and visualized in Figure 1. Each principle and practice of each analyzed agile software development method is connected to general agile principle in it. We expect to find the gaps where specific agile development method does not directly address specific general agile principle. These gaps present possible high level bottlenecks (areas to look more carefully into) in a specific agile software development method.

Results of the phase 1 are presented in subchapter 4.1.

3.2 Phase 2: identify possible bottlenecks in Lean software development

In phase 1 we identify possible high level bottlenecks for different agile software development methods. In phase 2 we choose one agile software development method and identify possible bottlenecks for it. Furthermore, we define actions for each identified possible bottleneck. Defined actions should help us to measure if a possible bottleneck is a real bottleneck in a specific agile software development method implementation.

This master thesis research industry partner Ericsson is implementing agile software development approach [29]. This approach is mainly following Lean software development principles [12][13]. Therefore, Lean software development is chosen from analyzed agile software development methods for this phase.

Having the goal of phase 2 and agile software development method, question 2 is defined:

- What bottlenecks might exist in Lean software development?

According to Poppendiecks [13] Lean software development is based on 7 principles. To be able to achieve success in a Lean software development project, all principles must be fulfilled. To fulfil each principle, a set of practices must be executed. Therefore, first our task is to identify practices needed to implement Lean principles. Atlas.ti [8] software is used to code all identified Lean software development practices. Lean software development principles are used as codes.

Afterwards, TOC principles described in paragraph 2.1.1 are used to model the system and make visual presentation of the results. We transform coded data to a tree as described in paragraph 2.1.1 and visualized in Figure 1. Each identified Lean software development practice is connected to one of the seven Lean software development principles.

Following TOC principles described in paragraph 2.1.1, we know that core problems exist in lowest branches of the TOC cause-effect tree. The lowest branches in the cause-effect tree that we model in this phase are Lean software development practices. This means that possible bottlenecks in Lean software development might be each identified practice. Therefore, the output of phase 2, the Lean software development tree with possible bottlenecks is called a theoretical model of possible bottlenecks in Lean software development.

After we have the theoretical model of possible bottlenecks in Lean software development created, we define actions for each possible bottleneck. These actions help to identify if possible bottleneck is a real bottleneck in a specific Lean software development implementation. These actions will be the guidelines for the interview questions in phase 3.

Results of phase 2 are presented in subchapter 4.2.

3.3 Phase 3: identify possible bottlenecks in agile software development implementation in a studied unit at Ericsson.

In phase 2 we develop the theoretical model of possible bottlenecks in Lean software development. We also define actions for each possible bottleneck.

The goal of phase 3 is to apply the theoretical model developed in phase 2 for actual implementation of agile software development method. Ericsson is implementing agile software development [29], which is following the main principles of Lean software development [12][13]. Therefore, the question 3 for this phase is:

- What bottlenecks might exist in agile software development implementation in a studied unit at Ericsson?

To answer this question the method of semi-structured interviews [19] is chosen. This method allows us to focus interviews on bottlenecks as well as to keep them open. To prepare for interviews, we pre-select 7 most probable bottlenecks for agile software development implementation in a studied unit at Ericsson from all possible bottlenecks list identified in theoretical model in phase 2. We base our selection on the rule to have one bottleneck connected to each principle and our current knowledge

about situation in a studied unit at Ericsson. After that, we prepare a set of open questions (see Appendix A). These questions allow interviewees to discuss and decide whether possible bottlenecks exist in their environment. At the end of interviews, we ask interviewees to prioritize analyzed possible bottlenecks according to their influence on the whole project performance. The prioritized list of possible bottlenecks in agile software development implementation in a studied unit at Ericsson is the expected output of the interviews.

Due to non disclosure agreements we will not be able to present detailed results of the interviews. Therefore, only generalized summary of the results of the phase 3 is presented in subchapter 4.3.

Chapter 3 explained what research methods we chose to answer our research question and how we will use them. Results of this master thesis research are presented in the following chapter.

4. RESULTS AND ANALYSIS

This chapter contains results of master thesis research gained during all phases of the research as described in chapter 3.

The first subchapter (4.1) presents the comparison of principles of analyzed agile software development methods against general agile software development principles defined by the authors of Agile Manifesto [11]. It provides us with information about possible high level bottlenecks of each analyzed software development method. It is the output of phase 1 of this research as described in subchapter 3.1 and answers the research question 1: What high level bottlenecks might exist in agile software development methods?

The theoretical model for identifying possible bottlenecks in Lean software development is presented in the subchapter 4.2. The model includes descriptions of practices and bottlenecks, as well as actions that should help to identify if bottleneck exists in specific implementation. This subchapter is the output of phase 2 of this research as described in subchapter 3.2 and answers the research question 2: What bottlenecks might exist in Lean software development?

Finally, subchapter 4.3 presents analysis of interviews in a studied unit at Ericsson. The main result is the list of possible bottlenecks identified in agile software development implementation in a studied unit at Ericsson. It is the output of phase 3 of this research as described in subchapter 3.3 and answers the research question 3: What bottlenecks might exist in agile software development implementation in a studied unit at Ericsson?

4.1 Possible high level bottlenecks of agile software development

In 2001 creators and representatives of different agile software development methods gathered together and agreed on Manifesto for Agile Software Development [11] (referred as Agile Manifesto later in text). This agreement started the agile software development movement [14] and is considered to be the core definition of the values of agile software development.

Four value statements defined in Agile Manifesto are extended by 12 Principles Behind the Agile Manifesto [11]. Each principle is described in more detailed in subchapter 2.2. For the purpose of this master thesis we assume, that these 12 principles are important to consider while implementing an agile development method. We do not question their validity or sufficiency and accept them as it is.

Having two things in mind, assumption we just made and TOC principles (described in paragraph 2.1.1), we can state, that in order to have successful agile software development method all 12 agile principles mentioned above have to be addressed during implementation. Following this conclusion we analyzed a set of agile software development methods, identified their principles and practices, and mapped each of them to one of the 12 agile principles.

The output of the process was the agile software development methods comparison tree presented in Figure 6. Top horizontal row (boxes with double borders) presents 12 agile principles as defined by the authors of Agile Manifesto. They are described in more detailed in subchapter 2.2. Below them follows principles and practices of each analyzed agile software development method (described in paragraphs 2.3.1 - 2.3.4). These principles and practices of each method are grouped by a surrounding oval. Reading the tree vertically, you can identify how each agile principle is directly addressed in different agile software development methods. To summarize, the agile software development methods comparison tree (Figure 6) presents the comparison of principles and practices of analyzed agile software development methods against agile principles defined by the authors of Agile Manifesto.

In the tree we can see the gaps where no principle or practice of agile software development method is connected to one of general agile principles. We consider these gaps as possible high level bottlenecks of the specific agile software development method. It is important to note that these gaps might be addressed by agile software development method indirectly throughout other principles. Therefore, while discussing each possible high level bottleneck in the following paragraphs, we will mention the principles which address the bottleneck indirectly and can help to elevate it.

Further paragraphs will provide a short discussion about each identified possible bottleneck in analyzed agile software development methods.

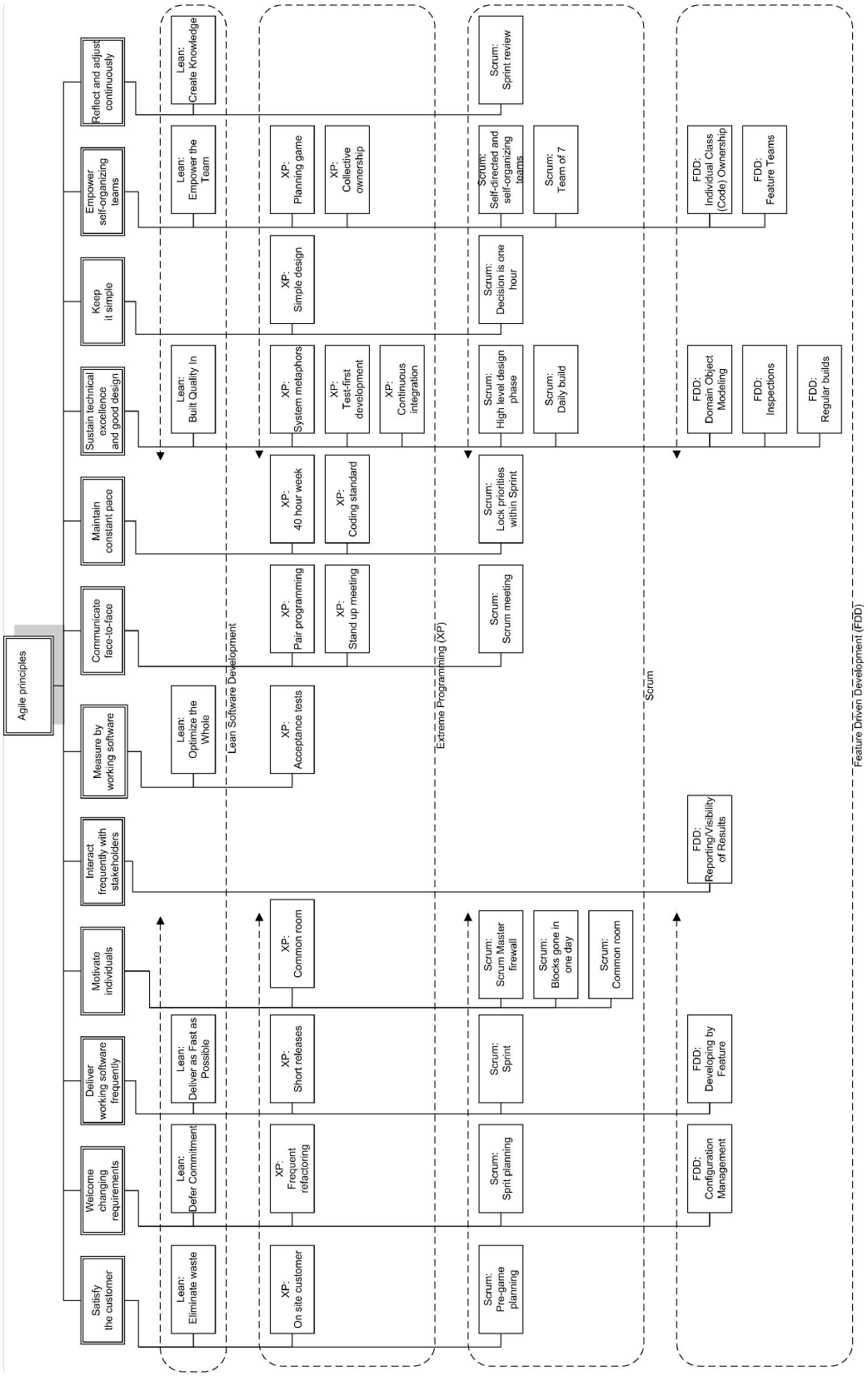

Figure 6. Agile Software Development Methods Comparison Tree

4.1.1 Possible high level bottlenecks of Lean software development

Agile software development methods comparison tree (see Figure 6) shows that Lean software development (referred as Lean further in this paragraph) has 5 possible high level bottlenecks:

- Lack of motivation for individuals ("Motivate individuals" principle)
- Lack of frequent interaction with stakeholders ("Interact frequently with stakeholders" principle)
- Lack of face-to-face communication ("Communicate face-to-face" principle)
- Lack of continuity ("Maintain constant pace" principle)
- Lack of simplicity ("Keep it simple" principle)

Further paragraphs discuss each identified possible high level bottleneck in context of Lean process and principles (described in paragraph 2.3.1) and how they indirectly can help to avoid these bottlenecks.

Lack of motivation for individuals. There is no principle that directly addresses motivation for individuals in Lean. As it is an important issue for achieving good results, individuals should be motivated and a motivation system should be created. Lean refers to motivation issue by implementing the "Empower the team" principle. This principle empowers team members to decide how to perform its best. The leaders of these teams should respect their team members and encourage them to self-organize their processes to complete the tasks. As a result motivation arises from individuals themselves. Nevertheless, lack of motivation might occur, so Lean managers should not ignore this possible bottleneck and continuously observe the motivation level as well as take actions.

Lack of frequent interaction with stakeholders. Lean principles does not define how often project members should interact with stakeholders. This step is a team responsibility as a team defines its working processes. A team decides on the frequency and type of communication with stakeholders (all people involved in the project). Despite that, Lean principle "Create knowledge" requires fast and frequent feedback for continuous learning. Therefore, it should force teams to establish frequent and close communication with stakeholders. Nevertheless, having poor processes of communication with stakeholders, as there is no described procedure of doing it, might still be a potential bottleneck

Lack of face-to-face communication. General agile principles encourage face-to-face communication over other communication channels. On the other hand, Lean suggests having daily short stand up meetings. This should maintain good face-to-face communication within the team. However, communication channels with stakeholders are up to the team to decide. Ignoring face-to-face communication with stakeholders or choosing time consuming methods might increase the impact of this possible bottleneck.

Lack of continuity. Agile principle "Maintain constant pace" promotes sustainable development. All project stakeholders should be able to maintain constant pace while using agile software development. As mentioned before, Lean principle "Create knowledge" focuses on continues learning, feedback and improvements, which should help Lean teams to maintain sustainable development. Despite that, Lean managers should make sure that gained knowledge is shared in a company to be able to deliver upcoming projects with the same pace.

Lack of simplicity. Agile development aims for simplicity. The principle encourages as simple methods and processes as possible. Therefore this might be the hardest possible bottleneck for Lean to break. Lean is relatively complicated method, focusing on improving many processes at the same time. It also addresses software development from highly managerial point of view, not getting into the technical details of software development as such and leaving it for self-organizing teams to manage. Therefore, Lean managers should keep in mind this possible bottleneck and make sure that all team members understand the value and processes of Lean and are committed to follow them.

4.1.2 Possible high level bottlenecks of Extreme Programming (XP)

Agile software development methods comparison tree (see Figure 6) shows that Extreme programming (referred as XP further in this paragraph) has 2 possible high level bottlenecks:

- Lack of frequent interaction with stakeholders ("Interact frequently with stakeholders" principle)
- Lack of reflection and adjustments to improve ("Reflect and adjust continuously" principle)

Further paragraphs discuss each identified possible high level bottleneck in context of XP process and principles (described in paragraph 2.3.2) and how they indirectly can help to avoid these bottlenecks.

Lack of frequent interaction with stakeholders. Agile principle requires interacting frequently with all project stakeholders (customers, product managers, sponsors, and other people involved in the project). XP focuses extensively on communication only with a customer. A customer has to be on site together with a development team ("On site customer" principle). He/she prioritizes what has to be developed first, does acceptance testing. However, other stakeholders (except customer) are almost not mentioned in the XP principles. This is definitely a possible bottleneck for the method.

Lack of reflection and adjustments to improve. There is no principle in XP that directly addresses how team should reflect and improve its processes. This is probably the hardest possible bottleneck for XP to break as XP is highly focused on technical software development activities. It advocates for simple self-organizing processes (e.g. "Simple design", "Pair programming" principles). Therefore it is relatively hard for managers to establish a stable ongoing reflection and improvement processes in such environment.

4.1.3 Possible high level bottlenecks of Scrum

Agile software development methods comparison tree (see Figure 6) shows that Scrum has 2 possible high level bottlenecks:

- Lack of frequent interaction with stakeholders ("Interact frequently with stakeholders" principle)
- No project progress measurement by working software ("Measure by working software" principle).

Further paragraphs discuss each identified possible high level bottleneck in context of Scrum process and principles (described in paragraph 2.3.3) and how they indirectly can help to avoid these bottlenecks.

Lack of frequent interaction with stakeholders. Agile principle requires interacting frequently with all project stakeholders (customers, product managers, sponsors, and other people involved in the project). Although Scrum does not require client to be "on site" as it is in XP, customer is responsible for prioritizing what has to go into Sprint backlog ("Sprint planning" principle) from the project features backlog. Despite that, interactions with other stakeholders (except customer) are almost not mentioned among Scrum principles. Therefore, this is definitely a possible high level bottleneck for Scrum.

No project progress measurement by working software. Agile principle states, that the only project progress measurement should be the amount of working software deployed in a production environment. Scrum measures project progress using Sprint Backlog Graph [14] which shows tasks finished by developers, but not features accepted by customers as general agile principle states. Therefore, "No project progress measurement by working software" is considered to be a possible high level bottleneck for Scrum.

4.1.4 Possible high level bottlenecks of Feature Driven Development (FDD)

Agile software development methods comparison tree (see Figure 6) shows that Feature Driven Development (referred as FDD further in this paragraph) has 7 possible high level bottlenecks:

- Lack of customer satisfaction ("Satisfy the customer" principle)
- Lack of motivation for individuals ("Motivate individuals" principle)
- No project progress measurement by working software ("Measure by working software" principle).
- Lack of face-to-face communication ("Communicate face-to-face" principle)
- Lack of continuity ("Maintain constant pace" principle)
- Lack of simplicity ("Keep it simple" principle)
- Lack of reflection and adjustments to improve ("Reflect and adjust continuously" principle)

Further paragraphs discuss each identified possible high level bottleneck in context of FDD process and principles (described in paragraph 2.3.4) and how they indirectly can help to avoid these bottlenecks.

Lack of customer satisfaction. Agile principle states that the main goal is to satisfy the customer by delivering valuable software frequently. Customer satisfaction should be the main drive for the project. FDD principles do not directly talk about customer satisfaction. The method focuses on implementing requirements (feature sets) and measures success by accomplished ones. Therefore, this is a possible high level bottleneck for FDD.

Lack of motivation for individuals. Agile principle considers motivating team members as very important part of project success. In FDD different feature teams are formed for each iteration ("Feature Teams" principle). People have to switch between different teams continuously. Keeping individuals motivated in such environment might be an issue. Therefore, lack of motivation for individuals is a possible high level bottleneck for FDD.

No project progress measurement by working software. Agile principle states, that the only project progress measurement should be the amount of working software deployed in a production environment. FDD measures project progress by accomplished feature sets. This does not mean that implemented feature sets are accepted by a customer. After the review they might require changes. Therefore, this is a mismatch what a general agile principle states and should be considered as a possible high level bottleneck for FDD.

Lack of face-to-face communication. Agile principle encourages face-to-face communication over other communication channels. FDD principles do not imply what communication channels teams should use. Moreover, FDD principle "Individual class (code) ownership" might unintentionally decrease face-to-face communication among FDD team members as each of them is responsible for his owned class and does not need to communicate with other team members often. Not considering face-to-face communication or choosing time consuming methods might increase the impact of this high level bottleneck.

Lack of continuity. Agile principle "Maintain constant pace" promotes sustainable development. All project stakeholders should be able to maintain constant pace while using agile software development. FDD principles do not propose how continuity should be ensured and learned lessons shared within the company to establish constant development pace in future projects. Therefore, implementing FDD the processes to ensure continuity should be establish to decrease the impact of this high level bottleneck.

Lack of simplicity. Agile development aims for simplicity. The principle encourages as simple methods and processes as possible. "If you want to be fast and agile, keep things simple. Speed isn't the result of simplicity, but simplicity enables speed" [30]. FDD has quite complex processes, as it is designed for bigger projects (there is a case study of using FDD in project with 250 people lasting 18 months [14]). Keeping simplicity in FDD is a possible high level bottleneck and should not be ignored.

Lack of reflection and adjustments to improve. There is no principle in FDD that directly addresses how a team should reflect and improve its processes. FDD principles define how to manage FDD project, but does not address how to reflect about principles, collect feedback and improve ongoing process to fit the needs of a specific environment. A reflection and improvements system should be created in FDD implementations to break this high level bottleneck.

This subchapter discussed identified high level bottlenecks (lacking principles) of four agile development methods. These bottlenecks present parts that selected methods lack or do not address directly. Very often, when implementing a specific method, the most effort is devoted for implementation, forgetting, that method itself can be not complete. If company wants to succeed in implementing agile software development principles, identified bottlenecks should be kept in mind when implementing a selected method.

4.2 Possible bottlenecks in Lean software development

In the previous subchapter we discussed possible high level bottlenecks of agile software development methods. We investigated which agile principles are not addressed in a specific agile method.

In this subchapter a theoretical model is presented to identify possible bottlenecks in Lean software development. We investigate Lean software development practices and define when they can become bottlenecks. Moreover, the actions that allow identifying each bottleneck are defined as well. In this subchapter we provide with the answers to the research question 2 described in subchapter 3.2: What bottlenecks might exist in Lean software development?

For the purpose of this master thesis we assume, that Lean principles (described in 2.3.1) are necessary and sufficient condition for software development process to be recognized as lean. We do not question their validity or sufficiency and accept them as they are.

Following the assumption we made above and using TOC principles (described in paragraph 2.1.1) we can state, that in order to have successful project all 7 lean principles have to be addressed. To achieve that, the practices that support each principle have to be implemented. If the practice is not implemented (not fully implemented) we consider it as a bottleneck. Following this conclusion we identified lean practices and mapped each of them to one of the 7 lean principles.

The output of the process was the tree presented in Figure 7. In the top horizontal row there are 7 principles of lean development. Below them identified practices are grouped according to which principle they directly address. In order to read the tree, if-then logics should be used. For example, IF we want to have a successful project using lean development THEN we need to address the principle "Eliminate waste". In order to address the principle "Eliminate waste" we need to implement the practice "Eliminate defects".

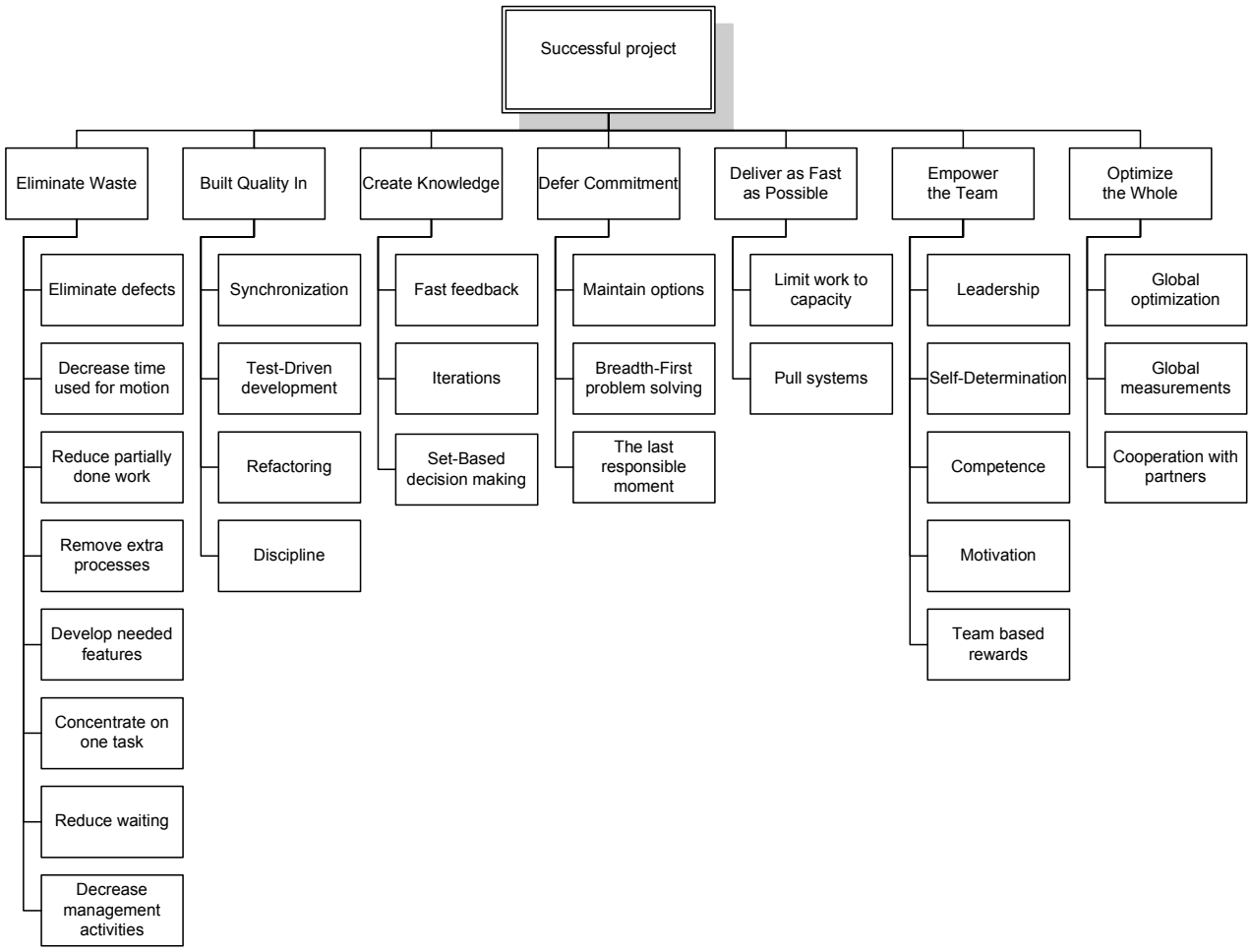

Figure 7. Lean Software Development Tree

Basic understanding about lean software development principles is provided in paragraph 2.3.1. Each principle has practices that directly address it. A thorough description of practices as well as possible bottlenecks is provided in following paragraphs 4.2.1-4.2.7. We discuss each principle one by one with corresponding practices.

The structure of 4.2.1-4.2.7 is as follows. First, the figure is presented where each lean principle is connected to a possible bottleneck. The motivation behind it is to point possible bottlenecks as TOC offers to look at root problems of each process. If we want to implement a practice successfully the bottleneck that stops the practice from implementation should be broken. That means, if the bottleneck will be eliminated the practice will be implemented successfully. If the practice will be implemented successfully, the principle will be implemented successfully as well.

Second, each practice is described in a structure: practice, bottleneck, and action. Practice description shows the aim of the practice. Bottleneck is described as an activity or process that stops implementing the practice. Action is defined as checklist that enables to measure weather a bottleneck exists or not.

4.2.1 Eliminate Waste

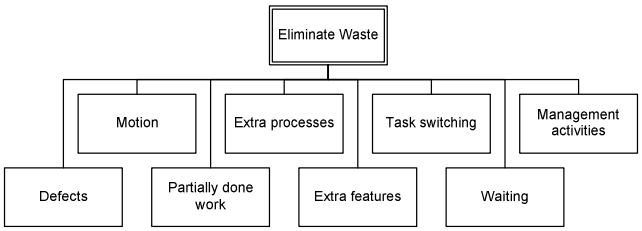

Figure 8. Possible bottlenecks of the Eliminate waste principle

Eliminate defects. The practice encourages looking for defects as early as possible and eliminating defects as soon as they are tracked. This should reduce the waste of time in late system fixes.
Bottleneck "Defects" – defects that are identified late in the system.
Action: List most common defects and indicate the ones that could be avoided/detected and fixed in earlier product lifecycle stages.

Decrease Time Used for Motion. The practice encourages reducing needed movement time for documents, artefacts or people. For example, sending an architecture document in parts would enable developer to start developing a part of the system; working in a common room would decrease the time to get answers from the colleagues. .
Bottleneck "Motion" – the time that a person, a document, an artefact is in motion.
Action: List most common activities that require motion and indicate the ones that the time needed for motion could be reduced.

Reduce partially done work. The practice encourages to do short releases to reduce the amount of work that is currently in a pipeline as it has a tendency to become obsolete.
Bottleneck "Partially done work" – is amount of work in a pipeline (from idea to deployment in production).
Action: Estimate the work that is currently in the pipeline and compare to the pipeline capacity.

Remove extra processes. The practice encourages reviewing all processes in company, prioritizing them, and removing the ones that add the least value for a customer (product).
Bottleneck "Extra processes" – the processes that do not add value for the customer (product).
Action: List all tasks performed by employees and indicate the ones that add the least value for a customer (product).

Develop needed features. The practice encourages prioritizing the features according to customer (market) needs and developing only the most important ones.
Bottleneck "Extra features" – the features that do not add (add very little) value for the customer (product).
Action: List all feature candidates for the product and indicate the ones that add the least value for a customer (product).

Concentrate on one task. The practice encourages working on one task at once.
Bottleneck "Task switching" – a resource has to switch between two or more tasks.
Action: List parallel tasks executed by the same resources and indicate the ones that switching could be avoided.

Reduce waiting. The practice encourages reviewing a process of a product lifecycle in a company and checking where the waiting time can be reduced.
Bottleneck "Waiting" – time spent on waiting for things to happen.
Action: Create a value stream (throughput) map and indicate the longest waiting times.

Decrease management activities. The practice encourages reviewing management activities, prioritizing them and removing the ones that add the least value for a customer (product).
Bottleneck "Management activities" – management activities that do not add value for the customer (product).
Action: List management activities and indicate the ones that add the least value for a customer (product).

4.2.2 Build Quality In

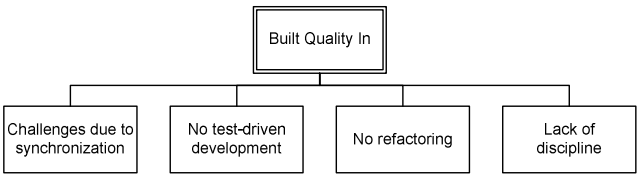

Figure 9. Possible bottlenecks of the Build quality in principle

Synchronization. The practice encourages integrating the code frequently into the system and testing the system as soon as it is integrated to decrease integration problems and to reduce amount of defects during the final release.
Bottleneck "Challenges due to synchronization" – infrequent or troublesome code integration into the system.
Action: List all synchronization activities (daily builds, builds by feature, system builds) and the most common problems that appear due to synchronization.

Test-driven development. The practice encourages developing defects free software that corresponds to specification that is written in form of executable tests.

Bottleneck "No test-driven development" – a development method where specification is not written in form of executable tests.
Action: List most common defects and identify the ones that could be found if specification would be written in form of executable tests.

Refactoring. The practice encourages improving code, by making it more readable and simplifying the design yet not changing the functionality of it.
Bottleneck "No refactoring" – the absence of time allocated for refactoring.
Action: List refactoring activities and estimate the time that is spent.

Discipline. The practice encourages creating the rules/instructions that people should follow (such as coding standards, naming conventions) to assure better quality.
Bottleneck "Lack of discipline" – the absence of rules/instructions that people should follow.
Action: List current rules/instructions and problems that arise due to absence of some of them.

4.2.3 Create Knowledge

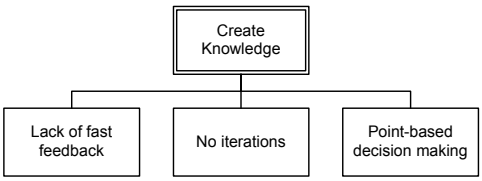

Figure 10. Possible bottlenecks of the Create knowledge principle

Fast feedback. The practice that encourages getting feedback as soon as the chance appears in different stages of a product lifecycle.
Bottleneck "Lack of fast feedback" – no feedback is collected during different stages of a product lifecycle.
Action: List all feedback sessions and the problems that are discussed most often during them.

Iterations. A practice that encourages developing software in short fixed timeframes.
Bottleneck "No iterations" – the software is developed in non iterative way and there is only one delivery to the client at the end of development.
Action: Identify the number and the length of iterations used in product development cycle.

Set based decision making. The practice that encourages a decision making process where a decision should be chosen from a set of possible options.
Bottleneck "Point-based decision making" – a decision making process where a decisions are proposed and refined with everyone until consensus is reached.
Action: List all decision making processes and indicate the ones that are point-based.

4.2.4 Defer Commitment

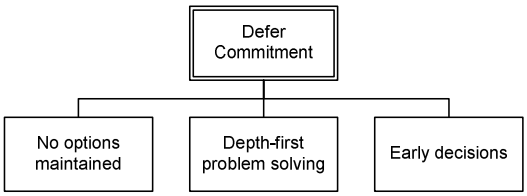

Figure 11. Possible bottlenecks of the Defer commitment principle

Maintain options. The practice that encourages a process where few options are maintained and a decision to chose the best one is made as late as possible.
Bottleneck "No options maintained" – a process where a decisions to choose one option are made early in the lifecycle.
Action: List made decisions and indicate the ones that prevented maintaining several options till later in the lifecycle.

Breadth-first problem solving. The practice that encourages a problem solving process based on breadth-first attitude.
Bottleneck "Depth-first problem solving" – a problem solving process based on depth-first attitude.
Action: List all problem solving processes and indicate the ones that are depth-first.

The last responsible moment. The practice that encourages a decision making process where decision is taken at the last possible moment (a moment when the absence of decision creates loss or eliminates an important alternative).
Bottleneck "Early decisions" – a decision making process where a decision is taken as soon as it is possible.
Action: List all decision making processes and indicate the ones that could be postponed.

4.2.5 Deliver as Fast as Possible

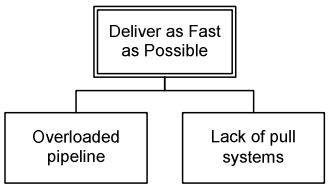

Figure 12. Possible bottlenecks of the Deliver as fast as possible principle

Limit work to capacity. The practice that encourages a work organization process where an amount of work in a pipeline equals to an amount of work resources can execute.
Bottleneck "Overloaded pipeline" – a work organization process where an amount of work in a pipeline exceeds an amount of work resources can execute.
Action: Estimate all work in a pipeline and compare to an amount of work the resources can execute.

Pull systems. The practice that encourages creating processes which enable developers to decide work processes without a management direction.
Bottleneck "Lack of pull systems" – every task for developer has to be assigned by a manager.

Action: List processes of tasks assignment and define the ones that require constant management direction.

4.2.6 Empower the Team

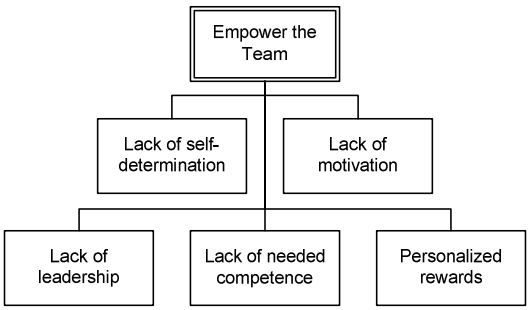

Figure 13. Possible bottlenecks of the Empower the team principle

Leadership. The practice that encourages having a team leader (not manager) who leads a team, motivates individuals in it, and sets a direction for a team.
Bottleneck "Lack of leadership" – no team leader responsible for leading and motivating a team as well as setting a direction for it.
Action: List team leader's/manager's responsibilities and indicate the ones that encourage the role as a manager rather than a leader.

Self-determination. The practice that encourages a culture where an individual has authority to choose tasks and prioritize them as well as organize the way of executing them and solve the problems along the way.
Bottleneck "Lack of self-determination" – a culture in the company where an individual get the tasks assigned by a manager and the way is already set on how to do it as well as there are clear procedures how to solve problems along the way.
Action: List procedures of task assignment and identify the ones that do not require self-determination.

Competence. The practice that encourages having ability (knowledge and skills) within a team to perform needed tasks to reach a goal.
Bottleneck "Lack of needed competence" – lack of the ability within the team to perform needed tasks to reach a goal.
Action: List all needed competences within the team to accomplish the tasks and indicate the ones that are missing.

Motivation. The practice encourages an engagement to perform a specific task.
Bottleneck "Lack of motivation" – lack of engagement to perform a specific task.
Action: List all motivating factors and indicate the ones that are missing.

Team based rewards. The practice that encourages team incentives for a well performed job over the personal recognition.
Bottleneck "Personalized rewards" – an incentive for a well performed job is based on personal recognition rather than a team one.
Action: List all rewards systems and indicates the ones that are based on rewarding personal achievements rather than a team performance.

4.2.7 Optimize the Whole

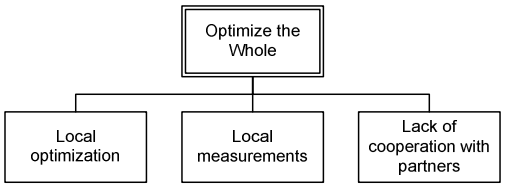

Figure 14. Possible bottlenecks of the Optimize the whole principle

Global optimization. The practice that encourages improving the whole system rather than a part of it.
Bottleneck "Local optimization" – an improvement process that optimizes a part of the system but not necessary the system as a whole.
Action: List all improvements and indicate the ones that focus on local optimization that does not improve the system as a whole.

Global measurements. The practice encourages value stream (throughput) measurements that lead to a global optimization.
Bottleneck "Local measurements" – value stream (throughput) measurements that focus on a local optimization.
Action: List all measurements and indicate the ones that focus on a local optimization.

Cooperation with partners. The practice encourages close communication and cooperation with partners (people and companies).
Bottleneck "Lack of cooperation with partners" – the absence of close cooperation with partners based on reaching a common goal.
Action: List all partners and indicate the ones that do not seek for the same goal as your company does.

In this subchapter we developed a theoretical model that enables to identify bottlenecks in Lean software development implementations. We explained each Lean practice and defined a possible bottleneck for it. Besides, we proposed actions that help to identify if a particular bottleneck exists in real Lean software development implementation. Companies that implement Lean software development could use this theoretical model to identify what bottlenecks exist in their implementations.

We verified this model in an agile software development implementation in a studied unit at Ericsson. The results are presented in the following subchapter.

4.3 Possible bottlenecks in agile software development implementation in a studied unit at Ericsson

In previous subchapter we presented our theoretical model for identifying possible bottlenecks in Lean software development. We defined main Lean practices, possible bottlenecks, and actions that help to identify if a bottleneck really exists in a specific implementation.

The goal of research phase 3 was to verify our theoretical model at our research industry partner Ericsson. Using the model we found possible bottlenecks in agile development implementation in a studied unit at Ericsson and answered our research question 3: What bottlenecks might exist in agile software development implementation in a studied unit at Ericsson?

To perform full analysis, each possible bottleneck in the theoretical model had to be discussed. Due to time constraints, first, we had to simplify the model. We reviewed all possible bottlenecks in it and preselected 7:

1. Waiting (follows Eliminate waste principle)
2. Challenges due to frequent synchronization (follows Build quality in principle)
3. Lack of fast feedback (follows Create knowledge principle)
4. Early decisions (follows Defer commitment principle)
5. Overloaded pipeline (follows Deliver as fast as possible principle)
6. Lack of needed competence (follows Empower the team principle)
7. Local optimization (follows Optimize the whole principle)

The selection was based on the rule to have one bottleneck connected to each principle and according to our current knowledge about situation in a studied unit at Ericsson and our own judgment if possible bottleneck might exist there.

We prepared open questions (see Appendix A) and conducted three semi-structured interviews with representatives from Ericsson. At the end of the interviews we asked each interviewee to select the bottleneck, which he thought was the most important one at the moment.

Summary of possible bottlenecks identified in a studied unit at Ericsson is presented in

Table 1. The organization of the table is as follows. First two columns ("Lean principle" and "Lean bottleneck") present the possible bottlenecks and principles they address preselected from our theoretical model. The third column ("A possible bottleneck in a studied unit at Ericsson") describes identified possible bottleneck. "Prioritization by interviewees" column identifies which possible bottleneck the interviewees choose as the main one at the end of the interview.

Due to non disclosure agreement with Ericsson we cannot present detailed descriptions of identified possible bottlenecks in this paper, therefore only generalized discussion will follow.

We found that 6 out of 7 our preselected possible bottlenecks might exist in a studied unit at Ericsson. Two of them link to the same possible improvement in testing procedures. Despite the fact that possible bottlenecks, prioritized by interviewees did not match exactly, we found very close connections between them. The bottleneck "Visibility of the global measurements" (considered as the main by interviewee 1) means, that we did not find formal measurements to measure the impact of decisions to the system as a whole, which leads to local optimization. If the cost of lead time on the system level is fully known, it would make a more clear case for addressing the other bottlenecks. Data of the interviews and prioritization of the possible bottlenecks by interviewees show that Ericsson is aware of the possible bottlenecks we identified and is working on solving them.

This subchapter presented results of our research phase 3. We verified our theoretical model developed in phase 2 and identified possible bottlenecks in agile development implementation in a studied unit at Ericsson. After this case study we are firm to state that the model can be used to identify bottlenecks effectively in Lean software development implementations in other companies as well.

Table 1. Summary of potential bottlenecks identified in a studied unit at Ericsson

Lean principle	Lean bottleneck	A potential bottleneck in a studied unit at Ericsson	Prioritization by interviewees
Eliminate Waste	Waiting	Time spent by product managers evaluating and documenting low priority features	
Build Quality In	Challenges due to frequent synchronization	Lead time of testing procedures	Interviewee 2 Interviewee 3
Create Knowledge	Lack of fast feedback	Lack of official designer to designer communication process	
Defer Commitment	Early decisions	None	
Deliver As Fast As Possible	Overloaded pipeline	Testing procedures require more time than testing resources can handle	Interviewee 2
Empower the team	Lack of needed competence	Different and not completely matching competence development models that are encouraged by different organizational structures.	Interviewee 2
Optimize the Whole	Local optimization	Visibility of the global measurements	Interviewee 1

The feedback about the model was positive from representative from Ericsson. He stated that our "model is relevant especially if you are new to the agile software development methods and about to deploy it in your organization. However in a running agile development, some bottlenecks are very easily found in practice, (e.g. the physical ones like Test lead time). When you try to address them you will reveal more hidden bottlenecks such as policy bottlenecks. Your method has a potential to put attention to the more hidden bottlenecks at an earlier stage, trying to avoid them to appear in the first place. If there is a practice not used from a certain method, your model can find arguments from Theory of Constraints (TOC) on why that practice should be implemented or not depending on the specific conditions in the particular development unit. I find the method very natural because the agile methods share the same goal as TOC, i.e. to bring high throughput, high flexibility and fast time to market. They can be regarded as method frameworks for how to achieve it in software development. Lacking practices or practices that can be improved are, therefore, an indicator that a bottleneck might appear."

5. THREATS TO VALIDITY

In this chapter we will discuss threats to validity of our master thesis research. As our research was divided into 3 phases we will discuss threats to validity of each phase separately in following subchapters.

5.1 Possible High Level Bottlenecks of Agile Software Development

First threat to validity is that we compared only 4 agile software development methods (Lean, Extreme Programming (XP), Scrum, and Feature Driven Development (FDD)). There are more agile software development methods that could be compared and analyzed. The analysis focused on each method separately; therefore, we claim that it is valid for the analyzed methods. Moreover, the same approach could be used to analyze other agile software development methods.

Second, the analysis was performed based on data found in books and articles. The existence of high level bottlenecks could be checked in real implementations in companies. That would verify and extend the theoretical analysis.

5.2 Possible bottlenecks in Lean software development

First threat to validity for the model is that it was based on data found in books and articles. Moreover, the authors of most used literature are Mary and Tom Poppendiecks. Although they are considered to be gurus of Lean software development, additional check in real implementations is needed to verify the model. We did it partially in the third phase of our research at Ericsson. Nevertheless, full validation of the model (including all possible bottlenecks) was not conducted.

Moreover, we defined actions how to identify if possible bottlenecks really exist in a real Lean implementation. These actions helped us to formulate questions for the interviews in the third phase. On the other hand, we verified only part of actions (the ones that we investigated).

5.3 Bottlenecks in Agile Software Development Implementation in a studied unit at Ericsson

First threat to validity for identified possible bottlenecks is that we performed only 3 interviews. That represents very small part of people working with agile software development in a studied unit at Ericsson. Despite the fact, the interviewees were people working directly with agile software development (the process itself or using it to create the product) and had different roles and positions in the company. Therefore, we can state that we collected data that represents the opinion of wide range of people.

Second, due to time constraints, we evaluated only part of our theoretical model as we preselected possible bottlenecks by ourselves. Therefore, there is a good chance that more possible bottlenecks might be found in agile software development implementation in a studied unit at Ericsson. Despite that, we believe that the ones we found are important, and if elevated, could help to improve current processes.

6. CONCLUSIONS

Agile software development emerged as a need to respond to the rapidly changing market. Creating software using document-driven, rigorous software development processes became too slow. Many agile software development methods were developed and successfully implemented in different organizations.

Despite the fact that all agile software development methods follow the same values, they address them in different ways. They all have bottlenecks (week parts) that should be carefully monitored while implementing the method. In this master thesis research, using principles of Theory of Constraints, we identified these bottlenecks in different level of detail.

In the first phase of our research, we identified possible high level bottlenecks (lacking principles) of four agile software development methods (Lean software development, Extreme Programming (XP), Scrum, and Feature Driven Development (FDD)). These high level bottlenecks present general agile development practices that analyzed methods do not have or do not address directly. They should be kept in mind while implementing the selected method. As a result, it is not enough to focus on implementing a method itself. What a specific agile software development method lacks (according to agile principles) is also important and should not be forgotten.

In the second phase of our research, we selected to investigate Lean software development method deeper. The decision to choose Lean was made because our research industry partner, Ericsson, is implementing the agile software development method that follows the main Lean principles. We developed the theoretical model that could be used to identify bottlenecks in Lean software development implementations. The theoretical model includes descriptions of possible bottlenecks as well as actions that enable to identify if a bottleneck exists in a particular Lean implementation.

During the last phase of the research we verified the theoretical model developed in the second phase. We interviewed people involved in the agile software development implementation in a studied unit at Ericsson, identified, and prioritized the possible bottlenecks. Only generalized results of this phase are presented in this document. The case study proved that our theoretical

model developed in research phase 2 is valid and can be used in other companies implementing Lean software development to identify bottlenecks. Moreover, representative from Ericsson mentioned that our model put a focus on policy bottlenecks that might be hard to notice from the beginning and helps to avoid them to appear in the first place.

Our master theses research as a whole expands the knowledge area of agile software development and implementation of different methods. Limited number of identified bottlenecks narrows down possible areas of issues and helps to focus on the core problems. Moreover, this was the first attempt (as far as we could find) to use Theory of Constraints principles to examine agile methods. The theory proved to be very useful as analytical tool in this kind of investigation. Its principles could be further used to find out how to eliminate identified bottlenecks and how to create a process of continuous improvement in an organization.

The purpose of this master thesis research was to identify high level bottlenecks of four agile software development methods and create a theoretical model for identifying bottlenecks in Lean software development implementations. Further research could follow in couple different ways. It could identify high level bottlenecks or develop theoretical models for other agile software development methods. It also could investigate and create guidelines how to elevate each possible bottleneck.

ACKNOWLEDGEMENTS

The authors would like to thank interviewees from Ericsson as well as supervisors Miroslaw Staron and Helena Holmström Olsson from IT University of Gothenburg for their time. We greatly appreciate your effort, involvement, and support.

REFERENCES

[1] Eliyahu M. Goldratt: “Critical Chain”, The North River Press, 1997

[2] Eliyahu M. Goldratt: “The Goal”, 2nd revised ed. Great Barrington, The North River Press, 1992 (1st ed., 1984,2nd ed., 1986)

[3] Eliyahu M. Goldratt: “It's Not Luck”, Aldershot, England, Gower, 1994

[4] Eliyahu M. Goldratt, Eli Schragenheim and Carol Ptak: “Necessary but Not Sufficient”, North River Press, 2000

[5] Eliyahu M. Goldratt: “What is this thing called the theory of constraints?”, NY, The North River Press, 1990.

[6] Eliyahu M. Goldratt: ”The haystack syndrome”, NY, The North River Press, 1990

[7] Lawrence P. Leach: “Critical Chain Project Management”, Second Edition, Artech House Publishers, 2004

[8] ATLAS.ti, http://www.atlasti.com, last accessed: 2007-11-25

[9] H. William Dettmer: “Goldratt’s Theory of Constraints. A Systems Approach to Continuous Improvement”, ASQC Quality Press, 1997

[10] David J. Anderson: “Agile Management for Software Engineering. Applying the Theory of Constraints for Business Results”, Prentice Hall, 2006

[11] Agile Manifesto, http://www.agilemanifesto.org, last accessed: 2008-03-21

[12] Mary Poppendieck and Tom Poppendieck: “Implementing Lean Software Development: From Concept to Cash”, Pearson Education, 2007

[13] Mary Poppendieck and Tom Poppendieck: “Lean Software Development: An Agile Toolkit”, Addison-Wesley, 2003

[14] Jim Highsmith: “Agile Software Development Ecosystems”, Addison-Wesley, 2002

[15] Craig Larman: “Agile and Iterative development: a Managers Guide”, Addison-Wesley, 2004

[16] http://www.netobjectives.com/training/lean-software-development, last accessed: 2008-04-04

[17] http://www.nebulon.com, last accessed: 2008-04-06

[18] http://en.wikipedia.org/wiki/Constructive_research, last accessed: 2008-05-13

[19] http://en.wikipedia.org/wiki/Semi-structured_interview, last accessed: 2008-05-13

[20] Stephen R. Palmer and John M. Felsing: “A Practical Guide to Feature-Driven Development”, Pearson Education, 2002

[21] Mary Poppendieck: “Lean Software Development”, IEEE Computer Society, 29th International Conference on Software Engineering (ICSE'07 Companion), pp. 165-166, 2007

[22] Roy Morien: ”Agile Management and the Toyota Way for Software Project Management” 3rd IEEE International Conference on Industrial Informatics (INDIN), pp. 516-522, 2005

[23] Jonathan Rasmusson: “Agile Project Initiation Techniques – The Inception Deck & Boot Camp”, Proceedings of AGILE 2006 Conference (AGILE’06), 2006

[24] Sanjiv Augustine, Fred Sencindiver, Susan Woodcock: “Agile Project Management: Steering From The Edges”, Communications of the ACM, Vol. 48, No. 12, 2005

[25] Damon Poole: “Breaking the Major Release Habit”, ACM Queue, October, 2006

[26] Mikael Lindvall, Dirk Muthig, Aldo Dagnino, Christina Wallin, Michael, Stupperich, David Kiefer, John May, Tuomo Kähkönen: “Agile Software Development in Large Organizations”, IEEE Computer Society, December, 2004

[27] Barry Boehm, Richard Turner: “Management Challenges to Implementing Agile Processes in Traditional Development Organizations”, IEEE Software, 2005

[28] Lan Cao, Balasubramaniam Ramesh: “Agile Requirements Engineering Practices: An Empirical Study”, IEEE Software, 2008

[29] Piotr Tomaszewski, Patrik Berander and Lars-Ola Damm: “From Traditional to Streamline Development – Opportunities and Challenges”, Software Process Improvement and Practice, vol. 13, pp. 195-212, 2008

[30] Jim Highsmith: “Agile Project Management”, Addison-Wesley, 2004

[31] Mark C. Paulk: “Extreme Programming from a CMM Perspective”, IEEE software, 2001

Appendix A

Eliminating waste (Waiting)

1. Do you experience that artefacts have waiting periods? Where usually is the longest artefact inactivity moment? (Waiting for resources, waiting for approvals, and other waiting periods.)

Build quality in (Synchronization)

2. How do you perform synchronization? Do you encounter any problems due to continuous synchronization? If so, could you name the top problems?

Create knowledge (Fast feedback)

3. Does your team have regular feedback sessions? If yes, how often? Which problems are discussed most often? Is a customer involved into these feedback sessions?

Defer commitment (The last responsible moment)

4. Is there a practice in a company to create several solutions (or one adaptable) for the complex problem? If yes, when and how? If no, why not? When is the final decision made?

Deliver as fast as possible (Limit work to capacity)

5. Do you have a backlog of features prepared for iterations? How much work (in person hours) is there in the list? How much time do you spend on managing the backlog of features?

Empower the team (Competence)

6. Have you identified the competences of each team member in your teams? What processes do exist to share their knowledge with others?

Optimize the whole (Global optimization)

7. How do you decide which processes to improve? Are there any specific measurements that influence the decision? Do those measurements focus on local optimization? Are they evaluated against the impact to the whole system (project)? Could you exemplify both?