View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at Tartu University Library

UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Andres Randmaa

Challenges to Architecture Decision-Making in Agile Development Environments

Master's Thesis (30 ECTS)

Supervisor(s): Dietmar Pfahl
Andrés Silva

Tartu 2016

Challenges to Architecture Decision-Making in Agile Development Environments

Abstract:

The goal of this master thesis is to explore challenges of architecture decision-making in agile environments and trying to find out what causes issues regarding agile architecture - the latter being a set of values and practices supporting the effective evolution of the design and system architecture, concurrent with the implementation of new business functionality. Although the topic has been existent for some time, there are still tensions between software architecture and agility that are not well understood by agile practitioners and researchers alike. While there are many active promoters of agile architecture design, there can be found equally many non-believers who state that agility and architecture cannot work together. To find answers to the research questions stated in this thesis, two research methodologies have been used: a literature survey and a case study. The literature survey covers related work on agility in software architecture and the challenges that come with it. The survey includes difficulties that have been observed regarding the use of agile development methods when building the architecture. In addition, recommended practices are discussed for applying agile processes associated with sound architectural principles. The second part of the thesis is a case study within a software startup company, Stagnation Lab, and the SMIT (Ministry of the Interior IT and Development Centre). In Stagnation Lab, two projects were analyzed. The main architects behind the design of each project were interviewed to look deeper into potential challenges when applying agile practices in architecture development. Similarly, at SMIT, the main architect responsible for the design of a monitoring information system for the Rescue and Fire Department was interviewed.

Keywords:

Agile, architecture, software, literature survey, case study

CERCS: P170

Väledates tarkvaraarenduskeskkondades esinevad väljakutsed arhitektuuriliste otsuste vastuvõtmisel

Lühikokkuvõte:

Käesoleva magistritöö eesmärk on uurida väljakutseid, mis võivad esineda tarkvaraarhitektuuri kombineerimisel väleda tarkvaraarendusega ning selgitada välja põhjused, mis neid esile toovad. Kuigi antud teema on olnud juba päevakorras mõnda aega, siis jätkuvalt esineb vastuolusid väleda tarkvaraarenduse ja tarkvaraarhitektuuri sobitamisel nii teadlaste kui praktikute seas. Väleda tarkvaraarenduse ning arhitektuuri toetajaid võib leida sama palju kui mittetoetajaid, kes väidavad, et neid kahte kokku sobitada pole võimalik. Käesolevas magistritöös püstitatud uurimisküsimuste vastamisel on kasutatud kahte uurimismetoodikat: kirjanduse ülevaade ning juhtumiuuring. Kirjanduse ülevaade käsitleb varem avaldatud uurimistöid ning raskuseid, mida on antud valdkonnas senimaani täheldatud. Lisaks sellele on arutlusse võetud kirjanduses välja toodud soovitused, mida järgida arhitektuuri arendamisel väledas arenduskeskkonnas. Magistritöö teine osa käsitleb Eestis tegutsevas tarkvara idufirmas Stagnation Lab ja Siseministeeriumi IT osakonnas (SMIT) läbi viidud juhtumiuuringut. Stagnation Labis analüüsiti kahte projekti ning intervjueeriti mõlema projekti rollis olnud tarkvaraarhitekte, et uurida, milliseid raskusi kohati süsteemi ning eelkõige tarkvara arhitektuuri arenduse käigus. Samal eesmärgil intervjueeriti ka SMITi projekti eestvedanud tarkvaraarhitekti.

Võtmesõnad:

Väle tarkvaraarendus, arhitektuur, tarkvara, juhtumiuuring

CERCS: P170

Table of Contents

1 Introduction 7

1.1 Research Questions 7

1.2 Motivation 8

1.3 Definitions 9

1.3.1 Software Architecture 9

1.3.2 Agile Methodologies 10

1.3.3 Scrum 11

1.3.4 Extreme Programming 12

2 Literature Survey 13

2.1 Plan and Goals 13

2.2 Performance of the Literature Survey 13

2.3 Literature Survey Results 14

2.3.1 Relevant Papers 14

2.3.1.1 Paper 1 - “What Does Research Say About Agile and Architecture?” 14

2.3.1.2 Paper 2 - “An Exploratory Study of Architectural Practices and Challenges in Using Agile Software Development Approaches” 15

2.3.1.3 Paper 3 - “Agility and Architecture: Can They Coexist?” 15

2.3.1.4 Paper 4 - “Agile Architecture IS Possible – You First Have to Believe!” 16

2.3.1.5 Paper 5 - “Peaceful Coexistence: Agile Developer Perspectives on Software Architecture” 16

2.3.1.6 Paper 6 - “Sustaining Agility Through Architecture” 17

2.3.1.7 Paper 7 - “How Much Up-Front? A Grounded Theory of Agile Architecture” 17

2.3.2 Summary 18

3 Case Study 19

3.1 Plan and Goals 19

3.1.1 About the Case Study 19

3.1.2 Overview of the Companies 20
3.1.2.1 Stagnation Lab 20
3.1.2.2 SMIT 21
3.1.3 Case Description 22
3.1.3.1 Klarvinduer 23
3.1.3.2 Simote 23
3.1.3.3 Rescue and Fire Department Monitoring Information System 23
3.1.4 Agile Software Development Practices 24
3.1.4.1 Agile Practices Regarding Non-Restrictive Architecture Definition 24
3.1.4.2 Agile Practices Regarding Restrictive Architecture Definition 26
3.1.4.3 Description of Practices and Relevancy to Architecture 26
3.2 Performance of the Case Study 30
3.2.1 Introductory Interview 30
3.2.2 Interview Questions for the Case Studies 30
3.2.3 Execution of the Interviews 32
3.3 Case Study Results and Analysis 33
3.3.1 Introductory Interview 33
3.3.1.1 Software Architecture in the Company 33
3.3.1.2 Challenges Encountered in Architecture Design 33
3.3.2 Interview at Klarvinduer 35
3.3.2.1 Addressing RQ1. To what extent do software architecture and agility support each other? 35
3.3.2.1.1 Agile Practices in the Project 37
3.3.2.1.2 Challenges in the Architecture Design 38
3.3.2.2 Addressing RQ2. How much up-front architecture is appropriate? 40
3.3.3 Interview at Simote 42
3.3.3.1 Addressing RQ1. To what extent do software architecture and agility support each other? 42

3.3.3.1.1 Agile Practices in the Project 44

3.3.3.1.2 Challenges in the Architecture Design 45

3.3.3.2 Addressing RQ2. How much up-front architecture is appropriate? 47

3.3.4 Interview at the Fire and Rescue Department Monitoring System 48

3.3.4.1 Addressing RQ1. To what extent do software architecture and agility support each other? 48

3.3.4.1.1 Agile Practices in the Project 50

3.3.4.1.2 Challenges in the Architecture Design 51

3.3.4.2 Addressing RQ2. How much up-front architecture is appropriate? 52

3.3.5 Summary of the Results 54

3.3.5.1 Agile Practices in the Projects 54

3.3.5.2 Agile Architecture and the Challenges Discovered 55

3.3.5.3 Amount of Up-Front Architecture Design in Agile Environments 58

4 Discussion 59

4.1 To What Extent do Software Architecture and Agility Support Each Other? 59

4.2 How Much Up-Front Architecture is Appropriate in an Agile Environment 61

4.2.1 Comparing Up-Front Architecture to Agile Architecture 61

4.2.2 Parameters Affecting the Appropriate Amount 63

4.3 Limitations 67

4.3.1 Literature Survey 67

4.3.2 Case Study 67

4.3.3 General Validity Threats 68

5 Conclusions and Future Work 69

6 References 70

Appendix 72

I. License 72

1 Introduction

The aim of this Master's Thesis is to explore how agility supports architecture decision making in software development projects. To do this, a thorough overview of the related work is made to explore known issues and suggested solutions. The first part of the thesis concentrates on the literature survey conducted to understand the subject in more depth. After analysing the main challenges, suggested solutions are presented.

The second part of the thesis is a case study in two organisations comprising three projects. The organisations are a small startup company, Stagnation Lab, and the SMIT (Ministry of the Interior IT and Development Centre), both operating in Estonia. Completed projects were explored in the organisations to try to find out how architectural decisions were made during development and how agile development methods were used during the process. Two projects were conducted in Stagnation Lab and the third project in SMIT. The first project was an e-commerce platform developed by the team from scratch without any previous architectural knowledge on said platforms and thus provided a valid example for exploring the architecture design phase of the system with hope to provide some answers to the research questions stated in the following chapter. The second project was a test automation tool developed for SmartTV and middleware apps, which included both hardware and software architecture development. In addition to the projects in Stagnation Lab, a monitoring software system developed for the Rescue and Fire Department by SMIT was also explored to not rely on a single company's agile adaption. To do this, data was collected through guided interviews with the lead architects of the projects and then further analysed for results. The case study results are compared to the previous literature review practices to find commonalities or differences related to the topic at hand.

Consequently, the results from both literature review and case study are then discussed in relation to the research questions. Conclusions and answers to the research questions along with limitations including validity threats of the study are found in the final parts of the thesis.

1.1 Research Questions

Agile development methods are used mainly to deliver software and provide adequate, non-heavy weight management during production. While it is proven to be effective in many cases (Isham, 2008), there are still arguably some parts of the development phase in which agile methods like XP, Scrum etc. are not used as effectively. For example, researchers have reported concerns that software architecture contributes to excessive big design up-front, specifically among practitioners involved in agile software development (Abrahamsson et al., 2010). A variety of methods have been proposed in order to balance trade-offs between agility and up-front design, (Boehm & Turner, 2004) including DSDM - the agile method where "just enough" architectural foundations are laid in the mandated "Foundations" phase.

Several industry-related studies (Murphy, et al., 2013) (Tripp & Armstrong, 2014) showed a certain reluctance of the management to buy-in agile methods, which we use as basis to assume that the use of hybrid approaches, such as the Water-Scrum-Fall reality (Theocharis et al., 2015), is caused by the reluctance. The primary driver for this type of integration is

that different stakeholder groups have expectations toward the "optimal" software development approach. To perform the standard project management tasks, such as estimation, planning, or controlling, the project managers require at least some stability. Moreover, project managers also have the responsibility to make sure that projects are aligned with the respective company strategy. For instance, projects must combine further business-related processes like human resource management, contracting, sales, and so forth. Considering agile methods traditionally address only system- or development related tasks, there is often a lack for such support. Thus, traditional approaches are regularly used to provide a framework and a basic structure for project organization. Then again, while developers are looking for approaches that support the development related tasks, they also prefer them to provide extensive freedom to select the best practices for the respective situation. This means developers prefer not to follow all practices enforced through management and would favour selecting methods and practices depending on the context, such as size of the project or the team working on it (Theocharis et al., 2015).

The purpose of this thesis is to explore how architecture design and decision-making is carried out in IT companies claiming to use agile development methods. The scope is interesting because there is a lot of controversy over agile practices in architecture design. (Abrahamsson et al., 2010) To look into the topic in more detail, a literature survey and a case study in two organisations may hopefully provide answers to the research questions selected for the thesis. The choice of research questions listed below were triggered by the conducted literature survey:

RQ1. *To what extent do software architecture and agility support each other?*

Although practitioners are stating more often that agility and architecture complement each other as propagated by studies (Abrahamsson et al., 2010), there is still a certain ambiguity as to what extent an architecture should be already there to continue development as propagated by literature on various agile methodologies such as Scrum and Extreme Programming. This question aims to explore how well agility complements architecture design and architecture evolution in organisations following agile methodologies.

RQ2. *How much up-front architecture is appropriate in an agile environment?*

According to an exploratory study (Falessi et al., 2010) of experienced practitioners conducted in the IBM Software Lab in Rome, project complexity was selected as a reason to focus agile development on software architecture by half of the respondents. Out of all options presented in the survey to be potential reasons for project complexity, the practitioners particularly perceived the lines of code or requirements as the leading indicator that requires a focus on software architecture with number of stakeholders following closely. This could be looked into further to find out in what circumstances does the necessity for architectural decisions come into light and what is meant by designing just enough architecture in agile environments before commencing with implementation.

1.2 Motivation

The motivation for addressing research questions RQ1 and RQ2 comes from the controversy of different opinions regarding how agility and architecture support each other. In addition, it is interesting to look deeper into what factors contribute to designing architecture up-front

in contrast to minimal planning prior to development. The motivation for choosing the topic for the Master's Thesis came from working at Stagnation Lab during the writing of this thesis. While the company viewed themselves as agile, there were differences to used practices supported by the agile methodologies. The beginning phases of projects conducted were followed more traditionally whereas later phases increased agility and delivery of software in iterations. Thus this was one of the starting points for understanding why these differences persist.

Prior research has indicated that the initial phases of software architecture development are problematic in agile environments. The suggested solutions have often been proven only in scientific settings or among focus groups involving students. This thesis aims to provide more answers how software architects concern the applicability of agile practices and how much planning is deemed necessary in various software projects. Hopefully the results can help the architects in question put their own work into more perspective for future purposes.

1.3 Definitions

This section contains brief descriptions and definitions of *software architecture* and *agile software development*. The reason for including these definitions is to precisely lay out the meaning of these terms in the overall thesis, as definitions may vary for some terms. This is especially true for agile methodologies *Scrum* and *Extreme Programming (XP)*, depending on the source used. In addition, the concept of software architecture can also be ambiguous with the reason being that software developers and architects define it based on their context. A clear reference model is necessary for both the agile methodologies and software architecture, as these are broadly used throughout the thesis.

The first subsection defines the software architecture and the explanations behind that. Next, the nature of agile software development is described, followed by depictions of two highly distinguished agile methodologies, Scrum (Cohn, 2010) and XP (Wells, 1999).

1.3.1 Software Architecture

Software architecture is the high level structures of a software system, the practice of constructing such structures, and documentation of these structures. Structures comprise of software elements, relations among them, and properties of both relations and elements. Analogous to the architecture of a building, the architecture of a software system is a metaphor.

There are various definitions of what a software architecture is (CMU, 2016). The reason is that people define architecture based on their context. There are also distinct opinions as to the scope of software architectures. In Software Architecture in Practice (2nd edition), architecture is defined by the authors as follows (Bass et al., 2003):

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation—are not architectural."

While some definitions such as the one above hint that software architecture encompasses strictly high level structural conceptions, there are others that stir confusion by providing more ambiguous definitions or instead including also some aspects of implementation. Consider Kruchtens definition of software architecture (Kruchten, 1994):

> "Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements such as scalability and availability. Software architecture deals with abstraction, with decomposition and composition, with style and aesthetics."

Nevertheless, these views are not mutually exclusive nor represent a fundamental conflict about what is considered as software architecture. Instead, they represent how much emphasis is placed on architecture in the research community. Taken together, the whole architecture entity along with its constituent parts, the behaviour when built, or the building of it form a unified view of software architecture.

1.3.2 Agile Methodologies

Agile methodologies came into prominence in the mid-1990s as an alternative to the traditional predictive approach to software development. As traditional software development was limited in ways such as generally not being adaptable to change and lacking communication between the specialized groups that complete each phase of work, an agile movement started appearing providing opportunities for assessing the direction of a project throughout the development cycle. It is achieved through routine iterations or sprints, at the end of which a potentially shippable product is presented by the teams. Early versions of renowned agile methods, like Scrum and XP, were introduced in the field of software engineering in 1995 and 1996 respectively.

In 2001, a group of software engineers gathered and formulated the Agile Software Development Manifesto (Beck, 2001), signed by representatives of Scrum, Extreme Programming, DSDM and others compassionate about searching for a substitute to plan-driven software development processes. The four main points of the manifesto are:

1. "Individuals and interactions over processes and tools"
2. "Working software over comprehensive documentation"
3. "Customer collaboration over contract negotiation"
4. "Responding to change over following a plan"

The common feature of agile methods is the frequent iterations. The development is divided into iterations spanning anywhere from a week up to a month. In these iterations, code additions or improvements are implemented by programmers. At the end of an iteration, newly implemented software is regarded as potentially releasable, meaning that the code developed during the iteration is both complete and tested in functional aspects as well as user acceptance. The new functionality is presented to customers or users in exclusive review meetings, where any stakeholder is welcome to attend. The general guideline for selecting the tasks to be performed is by prioritising the tasks according to their business value.

Agile development projects strive to be concentrated around highly motivated individuals. The developers should actively communicate with users or customers and feel a strong ownership of the code. Due to that, the teams are to a high degree self-organised and autonomous. Agile teams should be cross-functional, meaning that members complement each other's skills and personality within the team.

An important aspect of agile methodologies is that it is founded on an enduring commitment. Active participation is not only expected from the developers, but also the users and customers should be present during certain activities during the short iterations. Responding to change in agile environments means that projects receive constant input and feedback from their users. This is necessary for both to elicit and describe requirements and to maintain and clarify the existing functionality. While predictive methods such as Waterfall propagate development in a sequential fashion, with distinct phases, activities and transitions between these, agile methods combine all the activities into short iterations spanning from a week to a month.

1.3.3 Scrum

Scrum (Cohn, 2010) is an incremental and iterative agile software development framework for managing product development. It employs one or more self-organizing, cross-functional teams to deliver potentially shippable (properly tested) product increments in fixed-length iterations, called Sprints which are typically 1-2 weeks long but never more than 30 days.

Scrum introduces a predefined set of roles for the team members. Most important roles in Scrum are the *Product Owner*, the *Scrum Master* and the development team. The Product owner oversees the availability of sufficient information and resources about the tasks. The Scrum Master is accountable for removing impediments to the team ability of delivering the product goals and deliverables. It is important to note that the Scrum Master is not a traditional project manager or team lead, but serves as a buffer between the team and any distracting forces. Although Scrum focuses primarily on practices and organization within a team, it has also been successfully adapted to larger development projects consisting of multiple teams and distributed development, more precisely known as "Scrum of Scrums" (Paasivara et al., 2008).

During the time-boxed sprints, activities such as requirement planning and task estimation are performed by the development team in addition to coding and review activities. The initial stages of a sprint begin by arranging a sprint planning meeting, where a set of tasks is prioritized, broken down if necessary and estimated to sort out ambiguities by the Scrum team. Task estimation is often carried during the meetings by applying the *Planning poker* activity. Most of the time in a sprint is dedicated to coding and development of software. Short daily meetings, known as *daily stand-up meetings*, lasting up to fifteen minutes are performed to communicate in turns what team members are currently working on and what they plan to do for the rest of the day, as well as capture any impediments to the development the team might be facing.

The sprints end with meetings, known as *retrospective meetings*, where the team discusses both good and poor experiences encountered during the sprint. The meetings enable the team to assess which practices or activities worked well and where improvement is deemed

necessary as well as to discuss possible solutions to critical problems. In addition to the retrospective meeting, another meeting called the *sprint review* is also conducted to present the implemented tasks of the current sprint to the stakeholders.

1.3.4 Extreme Programming

Extreme programming, or XP in short, is another well distinguished agile methodology. It is similar to Scrum in aspects of short-time iterations, communication and close relationships between developers and customers. XP describes preferred work practices in great detail. The practices deemed most important include simplicity in design, pair programming, frequent code review and both unit testing and user acceptance testing.

While Scrum focuses more on the organisational aspects, XP is more focused on work methods than the general organisation of a project. The methodology propagates that team members should view themselves as equal in order to be able to implement any of the requirements. No specific roles are prescribed by the methodology. Criticism towards XP has been that it is deemed unable to scale up to large projects and has not been proven to be successful (not to the extent as Scrum at least) in projects with many participants (Cadle & Yeates, 2008).

2 Literature Survey

This chapter presents the literature survey conducted to look deeper into the research questions stated in section 1.1. Among the topics covered are the research methodology for the conducting the literature survey, the execution of the survey and the results gathered and discussed in more detail.

2.1 Plan and Goals

To gain more knowledge into the research questions stated in chapter 1.1, a literature survey is conducted and interesting papers in the field reviewed in more detail. In order to complete the literature survey, access to a selection of literature databases had to be granted. As the literature databases have vast amounts of research papers, it was necessary to produce search queries that would ensure that a sufficiently thorough result set would be returned. The relevancy of the papers gathered were then measured to extra inclusion-exclusion criteria to further narrow the result set. For the purposes of this thesis, the conducted literature survey was more lightweight as opposed to a systematic literature review, which means that some relevant papers could have been missed. Finally, a short conclusion was derived based on the papers to give an overview of what the current state is in regards to the research questions stated in this thesis.

2.2 Performance of the Literature Survey

The primary sources of material for this research come from the following databases: ScienceDirect, Institute of Electrical and Electronics Engineers (IEEE) Xplore Digital Library, Web of Science. Access to these databases is granted by the University of Tartu. The publicly available search engines Google and Google Scholar for research literature were used for collecting relevant papers.

The search terms were chosen taking into consideration combinations of software architecture and agility. From the agile methodologies, Scrum and XP were used in the search as two widely used and popular methodologies in software development. Taking into account the criteria, following keywords were used in search queries for collecting relevant papers in the aforementioned databases:

Q1. Agile AND Architecture
Q2. Software AND Architecture AND Agile
Q3. Agile AND Architecture AND Challenges
Q4. “Software architecture” AND Scrum
Q5. “Software architecture” AND XP
Q6. “Software architecture” AND “Extreme programming”

Longer search queries did not yield many more results without duplicates. Agile terms such as *feature-driven development* or *lean development* could have been included in the keywords, but compared to Scrum or XP, research on less renowned agile methodologies were likely covered by Q1-Q3. Altogether there were around 30 papers which were considered relevant to the topic and a selection was made as explained in more detail in section 2.3.1. In addition to the articles found by keywords, relevant articles were also collected from the references or bibliography sections on articles surveyed.

2.3 Literature Survey Results

This section presents the results of the conducted literature survey on relevant papers regarding the issue of software architecture and agility in more detail. At the end of this section, some general conclusions are derived from the research papers surveyed.

2.3.1 Relevant Papers

A selection of relevant papers concerning research and practices involving agile architecture design and development is provided here. Relevance to papers analysed is decided by how close the research is related to the current thesis topic of architecture design related challenges and proposals in agile environments. For the purposes of this study, all papers selected were in the field of software development and concerned about the design and development of software architecture following agile methodologies. Papers were not included for analysis that would only touch the subject partially or with only one or two mentions throughout the paper. This means if a paper was about agile practices and challenges in general with only brief mentions or relations to software architecture, it was excluded from review. The thesis also concentrates mainly on the agile methodologies Scrum and XP as two widely used and well defined methodologies related to development. Thus, papers about Crystal, DSDM, FDD and other lesser known methodologies were not taken into consideration. Books were also excluded from the survey as well as studies or papers not written in English. In the end, the following papers were analysed considering the criteria described:

1. “What Does Research Say About Agile and Architecture?” (Breivold et al., 2010)
2. “An Exploratory Study of Architectural Practices and Challenges in Using Agile Software Development Approaches.” (Babar, 2009)
3. “Agility and Architecture: Can They Coexist?” (Abrahamsson et al., 2010)
4. “Agile Architecture IS Possible – You First Have to Believe!” (Isham, 2008)
5. “Peaceful Coexistence: Agile Developer Perspectives on Software Architecture” (Falessi et al., 2010)
6. “Sustaining Agility Through Architecture” (Weitzel et al, 2014)
7. “How Much Up-Front? A Grounded Theory of Agile Architecture.” (Waterman et al., 2015)

2.3.1.1 Paper 1 - “What Does Research Say About Agile and Architecture?”

This paper presents the results of a literature survey for statements made in relation to the link between agile development and software architecture. The survey aims to take a look at two main research questions: Is architecture sufficiently emphasized in agile methods and do agile practices improve software architecture. Conducted as a systematic literature review, 36 studies out of a total 842 papers were reviewed and analysed.

The paper brings forward issues and proposals, for example statements including that there is a need to combine agile and architecture to ensure both quick response to changing market needs, and long-term survival of product assets. Two practices are explored in more detail in order to explore whether agile practices improve software architecture, namely test driven

development (TDD) and refactoring. In the case of TDD, the findings conclude that it seems to have a positive effect on architectural quality. In terms of refactoring, the general belief is that optimal up-front architecture design is preferred over minimal up-front design but even the optimal strategy delays inevitable problems like design erosion and architectural drift. However, there are few empirical results supporting these statements. This means that it is difficult to make any conclusions based on results supporting or disproving the assumption that software architecture is insufficiently emphasized in agile methods.

The main observation of the paper is that many of the claims made by the agile community lack scientific support. A lot of these claims are solely based on expert opinion and often only student projects or other artificial settings are tested for the adoption of an agile approach. Based on this, the researchers believe that additional research could provide value to understand the influence between agile and architecture before proposing supplementary practices.

2.3.1.2 Paper 2 - "An Exploratory Study of Architectural Practices and Challenges in Using Agile Software Development Approaches"

The paper written by Muhammad Ali Babar from the University of Limerick talks about a study exploring architectural practices and challenges using Agile Software Development approaches. In it, he reports a case study aimed at understanding and empirically identifying the architectural practices. He asserts that the findings of the study contribute evidence to support the reports that adopting agile approaches can cause modifications in architectural practices with possible negative impacts on the architectural artifacts and design decisions. The study also claims to have identified agile practices that are observed to have meaningful impact on architectural practices and artifacts by the participants in it. However the author suggests that conducting more studies on the basis of these findings would be invaluable to finding out if significant peculiarities are present for architectural practices within teams following agile principles because of different characteristics of the system to be developed within a company or distinct companies. He expects the results to further stimulate researchers to discover the fundamental reasonings and causes that point to the awareness and use of different architectural practices and challenges in agile development teams.

2.3.1.3 Paper 3 - "Agility and Architecture: Can They Coexist?"

IEEE Software Magazine published a special issue dedicated to Agility and Architecture (IEEE Software, 2010). For instance, one article concentrates on explaining the role of architects, architecture importance as a function of project context, and how much documentation is preferable in agile processes. It is suggested by the authors that tensions between architecture and agility might be a false dichotomy. The article states that software developers have equally important roles in successfully letting agile and architectural approaches coexist. Suggested practices for a successful agile architecture design include for example understanding the context of the system built, clearly defining the architecture and its scope and understanding when it's appropriate to freeze the architecture to provide necessary stability for developers to finish a product release.

2.3.1.4 Paper 4 - "Agile Architecture IS Possible – You First Have to Believe!"

Mark Isham's Agile 2008 conference paper (Isham, 2008) reports about the repercussions of a project that went astray and how the product team coped. The project sustained contrasting views between "fix it right" and "fix it now" ideals, which resulted in an architecture incomplete, inefficient and ultimately cancelled after months of working harder, not smarter. Arguably all factors contributing to the project demise were traced back to poor project management. Issues encountered included not enough project management for sprints, lack of accountability to the architects, time pressure and insufficient communication between the architecture team and the product team. The author describes how supporting an agile process later on the same project not only created a scalable and stable architecture but also improved the team morale and productivity. More importantly, it proved to a skeptical audience that the old ways of "rebuilding from the ground up" have become outdated and lacking in the modern Internet business environment.

2.3.1.5 Paper 5 - "Peaceful Coexistence: Agile Developer Perspectives on Software Architecture"

The article published in the same issue of IEEE Software Magazine described before (IEEE Software, 2010) provides a survey of 72 IBM software developers suggesting theoretical compatibilities between software architecture and agile methodology. The survey aimed to gather quantitative data by presenting the participants with different possible use cases to measure the relevance levels of software architecture among the cases. On average, the participants had around 18 years of developing practice with most of them already having adopted agile approaches. Results from that survey showed that only three out of 17 software architecture uses presented were more insignificant than significant to agile practices.

The second part of the survey was to look into when to focus on software architecture. The optional answers for the participants to choose from were always, never, and when the project is complex. With complexity being a broad term, the third option was divided into four leading causes: number of requirements or lines of code, number of stakeholders, geographic distribution, or other. Half the respondents chose project complexity as a reason for focusing agile development on software architecture. When asked why, the leading cause was perceived to be number of requirements/lines of code with number of stakeholders being close.

Further results from the study showed that since most of the participants were supportive towards agility and architecture approaches, the main problem in combining architecture-centric methods and agile is not related to theoretical issues, but rather in practical matters of adoption. For instance, non-agile developers appeared more pessimistic compared to agile developers, particularly by overestimating contrasts in architectural and agile approaches. However, the authors concluded by the results that agile developers recognize software architecture as important and complementary to agile values, as opposed to contrastive or neutral.

2.3.1.6 Paper 6 - *"Sustaining Agility Through Architecture"*

The paper reports the experiences regarding a long running Scrum project where a concept of an architecture design ("epic-architectures") for a coherent group of user stories was presented. The researchers state that by shifting the planning horizon further than single sprints helped them to create more stable and reusable concepts and also construct simpler, more elegant, and more maintainable solutions.

While agile development approaches do not deny the need for design activities to plan the realization of the system, there is no directly applicable guidance on how to do this in practice. To address this problem the researchers came up with an approach to integrate architecture design with the Scrum development approach they used. For epics or themes, epic-architectures were created. These were architecture designs with concepts that span across single user stories. From the epic-architectures the researchers derived story-architectures for the realization of single user stories. The experiences with the approaches were satisfying to the researchers with reports on reducing the necessary effort for refactoring and also for realizing new user stories. It led to a more consistent and stable system structure and more flexible system through more planning while sustaining the agility in the development process.

The reports hope to provide practically applicable guidance on how to integrate lightweight architecting in agile development processes and to sustain agility while creating high quality products. Further research is planned in order to provide better guidelines and better tool support to make the approach more generally applicable. Evaluation of those tools and guidelines is planned to be conducted in controlled experiments and industrial case studies.

2.3.1.7 Paper 7 - *"How Much Up-Front? A Grounded Theory of Agile Architecture"*

A more recent study aims at dealing with the problem of how much design up-front would be required with software architecture (Waterman et al., 2015). Many agile methodologies recommend some architecture planning, but little guidance is provided as to which architectural decisions should be made up-front and how those decisions are influenced. Even though many agile developers handle this absence in guidance by designing "just enough" architecture up-front in order to start development, it is difficult to determine in advance how to measure "just enough".

In the paper, a grounded theory of agile architecture is presented that describes how agile software teams come up with solutions to how much upfront architecture design effort is enough. The theory presents six forces that affect the team's context (requirements instability, technical risk, early value, team culture, customer agility and experience) and five strategies (respond to change, address risk, emergent architecture, big design up-front, frameworks and template architectures) that teams use to determine how much effort should be put into up-front design. The relationships between the forces and strategies are then explored to identify positive and negative changes one can bring on the other.

2.3.2 Summary

The conducted literature survey suggests that the debate about agility and architecture has been ongoing for some time, with practitioners for and against the methodology. While quite a few different methods and theories have been developed to facilitate the issue, there is still lack of empirical studies concerning the challenges or adoptions of those methods in industrial settings, requiring more research to fully reveal the benefits and drawbacks by an agile software development methodology (Breivold et al., 2010). Many of the tested theories and practices, such as the grounded theory of agile architecture or epic-architectures, have been advocated in theory and sometimes artificial settings which cannot be precisely conformed to actual practices. The studies that do exist are small and often diverging. Thus this thesis aims to further the empirical research done so far by studying architecting and agile practices in software companies to look into what challenges are there regarding architecture decision-making.

3 Case Study

Following the literature survey results in chapter 2, a case study in distinct organisations was conducted to explore the contradicting statements and findings in the researched literature. Section 3.1 describes the methodology for conducting the case study including the selection of participants. Section 3.2 lists out the interview questions and execution of the case study. Section 3.3 describes the results and analysis of the case study execution.

3.1 Plan and Goals

This section presents the plan and goals for conducting a case study of three software projects, where considerable effort was spent on software architecture and development was in agile environments. To conduct the case study, access to suitable software projects developed in agile environments along with significant effort spent on designing the architecture was needed. The company Stagnation Lab was a convenient option due to work commitments there during the writing of this thesis and as it had been involved in various projects developed from concept to finish. Having good connections with the lead architects in the company was a benefit also. The third project was selected from the company SMIT to not rely on a single company and to explore the architecture development in another agile environment. The projects selected for the purposes of this case study were an e-commerce platform for the retail business for windows and doors, a test automation tool for set-top boxes/smartTV-s and a monitoring information system for the Fire and Rescue Department. Prior to conducting the case study, interviews had to be arranged and reviewed for gathering data. This means that the interview guide had to be designed in such a way to collect necessary aspects of architecture design in an agile environment to provide answers to the research questions stated in this thesis. Due to having multiple definitions as to what software architecture is considered to be, two definitions from published bibliography were selected and relevant agile practices in both cases determined. The practices were used to map the actual practices used in projects to understand to what extent the companies considered themselves to be agile.

3.1.1 About the Case Study

The case study focused on three projects developed from concept to completion in two different companies. One of them was an online e-commerce project Klarvinduer which the company was working on in the first half of 2015. The customers behind Klarvinduer are involved in the retail business of windows and doors, operating in Denmark. The other project was the company's own product named Simote, which was developed from a need to automate IPTV black-box testing while being involved in a larger project led by Telia. The third project is a monitoring system developed for the Estonian Fire and Rescue Department by SMIT. The key architects in all projects were interviewed, in order to gather the necessary evidence for an agile evaluation.

The case study consisted of unstructured to semi-structured interviews with a set of questions for guiding the interviews in a direction relevant for the data collection. The case study data extraction was formalized into following steps:

1. Introductory overview of what the architects consider as software architecture and what agile methodologies are followed in the company;
2. Interview with the key architect of Klarvinduer;
3. Interview with the key architect of Simote;
4. Interview with the key architect of the Fire and Rescue Department monitoring system;
5. Presenting the results of the case study to the architects for possible added information.

In order to prevent inconsistencies due to problems recalling what was discussed, the interviews were recorded, except for the first introductory interview. In addition, all the interviews recorded were also transcribed for better data extraction. The collected data from the interviews was then analysed and discussed in relation to the research questions to provide results.

3.1.2 Overview of the Companies

This section presents the overview of the companies and the development methods used to provide some context about the environment where the case study was conducted in.

3.1.2.1 Stagnation Lab

Stagnation Lab is a startup that has been active for 3 years at the writing of the thesis. The current development work is based on different contract services focusing on multi platform web applications. Build once, deploy to platforms of choice as the company motto states. In addition to web applications, some employees are also actively interested in development of working hardware and software, such as robotics. Examples of the projects that Stagnation Lab has been involved in:

- Telia Pay TV services MinuTV and NutiTV. Solution was developed and implemented in an intimate relationship with Telia's IPTV team;
- World's first "set-top-box-free" IPTV service for Telia on Samsung SmartTV-s;
- Development of the E-residency application form in partnerships with Enterprise Estonia (EAS), SMIT and the Police and Border Guard Board;
- BigBank mobile credit card app;
- Klarvinduer - online retail platform for windows and doors.

Stagnation Lab has also developed their own product called Simote, which came from the need to automate testing for the Pay TV services listed above for set-top boxes and smartTV-s. It has also been successfully sold to TeliaSonera subsidiaries in the Baltics. It is also currently the only product where both software and hardware were developed from the very beginning and was commercially distributed.

Stagnation Lab currently has 10 employees out of which there are 5 founders, 4 of whom are with a technical background and 1 with a background in business. The roles in Stagnation Lab are divided into 5 back-end developers, 1 front-end developer, 1 front-end designer, 1 analyst, 1 junior developer and the COO who handles all the external production related work such as meeting with clients and future prospects. Of the 5 back-end developers in the company, 2 of them are mainly concerned with software architecture. For the purposes

of this thesis, both of the architects were interviewed in terms of the respective projects they were leading.

Development Methodologies Used in Stagnation Lab

The development process followed in the company is a mix of agile and traditional methods. Although the COO of the company has stated that they follow a Scrum based development method, it is hard to notice many of the related practices that go hand in hand with Scrum. As observed during the time worked there, whenever possible, the team follows a mostly autonomous process unless being involved in a project where certain methodologies such as Scrum or traditional methods are required. For example, in the case of the Bigbank mobile credit card app, the methodology followed was pure Scrum as it was decided by the customer and collaborators for the development approach in order to deliver the solution incrementally and fast. In the case of Klarvinduer, the development was done in a more relaxed environment with deadlines set for deliverables 3-4 months from the start of the architecture design and system implementation.

Generally there is no strict following of recommended rules for Scrum, XP, Kanban, etc. It is rather a mix of best practices followed from these methodologies with some adaptations coming from even traditional development practices. While there are elements of sprints, sprint retrospectives, and product backlogs present, the requirements, analysis and architecture design phase follows a more traditional waterfall-esque approach. In that sense, one could say that the company follows some variation of the hybrid Water-Scrum-Fall discussed in previous research (Theocharis et al., 2015). Generally however, once the development has passed the initial requirements engineering, prototyping and architecture design, the development follows a more agile approach with incremental deliveries, responding to change, etc.

3.1.2.2 SMIT

The Estonian Ministry of the Interior IT and Development Centre (SMIT) is an ICT (Information and Communications Technology) provider operating in the public sector. It started operations on March 1, 2008 with the purpose of delivering ICT services to the Estonian Ministry of the Interior and the administrative field along with it. The mission of SMIT is creating a development environment within the administrative area of the Ministry where internal security institutions would collaborate in devising and developing applications vital for providing internal security services.

The operation of ICT services for the Estonian Ministry of the Interior, The Rescue Board and the Estonian Academy of Security Sciences and The Police and Border Guard Board (PPA) are all ensured by SMIT by supporting the provision of public and integral services to the Emergency Centre. Examples of projects where SMIT has been involved in are launching the E-residency online application form in collaboration with EAS, PPA and Stagnation Lab, and the monitoring information system for the Fire and Rescue Department explored in the current thesis.

SMIT is a government agency deploying around 201-500 employees, depending on different projects and fields.

Development Methodologies Used in SMIT

Software development in SMIT has followed more traditional methods until recent years. The collaboration for developing the E-residency online application form was one of the first projects done in a more agile way while involving multiple parties and large amounts of cross communication. In the case of the monitoring system development for the Fire and Rescue Department, the project initially followed a traditional waterfall method but when it was decided to redesign the system, the team started following Scrum because of drawbacks experienced when working with previous projects. It is not precisely clear what the overall approach in SMIT regarding development methodologies is in large scale, but the general movement seems to go more towards agility in smaller projects.

3.1.3 Case Description

The case is based on three projects developed from concept to finish in the company Stagnation Lab and SMIT. A high-level overview of the projects is provided in table 1.

Table 1. Characteristics of the projects explored in the case study.

	Klarvinduer	**Simote**	**Monitoring system**
Domain	E-commerce platform specialising in the retail business of windows and doors	Black box test automation tool for set-top box/smartTV applications	Information system for conducting monitoring and reporting of different activities in the Rescue and Fire department
Number of developers	8	3	5
Duration	9 months + maintenance and support	1.5 years + maintenance and support	10 months + maintenance
Project type	Software	Software, hardware	Software
Programming languages	JavaScript, Node.js, HTML5, React	JavaScript, Node.js, HTML5, React	Spring MVC, Spring Data JPA, Spring Boot, Angular.js, Fat client
Chosen process method	No specific	No specific	Scrum

3.1.3.1 Klarvinduer

Klarvinduer is an online e-commerce shop for selling custom-made windows, doors and conservatories. Customers of the solution are operating in Denmark with plans of scaling more globally. The analysis phase of the project started late 2014 and majority of the development took place in the first half of 2015, launching in august for public.

The project was chosen as one example for this case study because it was built from start to end without any prior relevant knowledge or experience building e-commerce platforms. There are many ready-made platforms and architectures available for people to build online stores upon but in terms of this custom retail business, it was decided to build the whole thing from start.

3.1.3.2 Simote

Simote is a black-box test automation tool for SmartTV-s and set-top boxes. It works as an intermediary for a person controlling the switching of different views with an infrared remote control. A person can record a set of controls with the remote control and Simote plays the recordings on the target device(s). Useful info such as application errors, device restarts and application performance metrics are then provided for the tester for further debugging. The product was developed from the need to automate testing of the local pay TV service provider's smartTV and set-top box application. Formerly, testing of the application was largely manual, meaning a person had to test the application by using the remote control along with ridiculous amount of clicking.

The project was chosen as another example for the case study because it was developed from concept to finish with a key architect leading the development. On top of the software, hardware was also designed and developed by the architect behind Simote.

3.1.3.3 Rescue and Fire Department Monitoring Information System

The monitoring information system for the Rescue and Fire Department deals with mainly two things: identifying the causes of fires and facilitating the inspection of buildings or edifices, for example distributing building permits. Thus the monitoring system comprises of the state of different objects, such as fire detectors, and processing events or more precisely, for reporting details of events that required the involvement of rescue workers. Depending on the input from the rescue workers, the workers assigned had to inspect the area, fill out their reports and add the details to the information system for further processing.

The project was chosen because it involved another company in order to reduce bias by exploring only one organisation. Additionally, the project was conducted following the Scrum methodology and provided valuable input for the thesis. The opportunity to interview the architect of the system came from a co-worker in Stagnation Lab.

3.1.4 Agile Software Development Practices

In order to further identify what agile practices were being used more specifically in Stagnation Lab and SMIT in the software architecture design, a list of agreed Scrum and XP practices was used for determining the level of agility in the two cases explored. By using the selected methodologies, the mapping of practices does not rely too heavily on a single methodology. Scrum and XP were also used for a basis because they are both well-established, accessible and there is a general agreement with regards to their content and associated practices. For the purposes of this thesis, a list of agile practices was borrowed from a research paper about agility in scientific software development (Sletholt et al., 2011). In the original paper, 35 agile practices were identified but some activities were omitted because not all of those are relevant for software architecture design explored in this thesis. The practices dropped from the original list were as follows:

1. Team members volunteer for tasks (self-organizing team)
2. Move people around
3. All the production code is pair programmed
4. Only one pair integrates code at a time
5. Set up a dedicated integration computer

Tasks related to software architecture are delegated usually for the software architect as he is the architecture owner and responsible for delivering the design. In practice no. 1, the development team members volunteer for implementation related tasks but generally are not involved in tasks designated for the architect in a team. Similar reasoning is for dropping practice no. 2. Developers provide feedback to the architect regarding his design decisions but don't exchange roles with the architect. Practices 3-5 were dropped because they were related to pair programming - an activity related to coding only. Architecture design deals mainly with the conception and devising the high level structures of a software system. Although architecture design can include parts of implementation depending on the context, it is not the main focus.

As mentioned briefly in section 1.3.1, software architecture can be defined in various ways with differing opinions as to the scope of what it includes. For the purposes of the case study and depending on the architect's personal understanding of software architecture, two definitions are taken into account based on (Bass et al., 2003) and (Kruchten, 1994). These are named from here on as restrictive and non-restrictive architectures respectively.

3.1.4.1 Agile Practices Regarding Non-Restrictive Architecture Definition

Non-restrictive software architecture is regarded in this thesis as architecture that is not strictly related to conception and modeling high level structures and interfaces, but also incorporates some elements of design patterns and implementation. The architect participates actively both in architecture design and implementation of architectural elements and sets the baseline for the developers to build upon. Naturally, the architect is part of the development team and is also involved in overall development when the architecture is well defined but he is still primarily in charge for sustaining the architecture throughout the project lifecycle. The list of agile practices applicable to the non-restrictive architecture design is provided in table 2 below. Descriptions of the practices and the relevance to software architecture are listed in section 3.1.4.3.

Table 2. Mapping of agile practices for non-restrictive architectures based on (Sletholt et al., 2011). Practices marked with an asterix are XP practices, but are likewise recommended in the Scrum methodology.

#	**Agile practices**
1	Priorities (product backlog) maintained by a dedicated role (product owner)
2	Development process and practices facilitated by a dedicated role (Scrum master)
3	Sprint planning meeting to create sprint backlog
4	Planning poker to estimate tasks during sprint planning
5	Time-boxed sprints producing potentially shippable output
6	Mutual commitment to sprint backlog between product owner and team
7	Short daily meeting to resolve current issues
8	Burn down chart to monitor sprint progress
9	Sprint review meeting to present completed work
10	Sprint retrospective to learn from previous sprint
11	Release planning to release product increments
12	User stories are written*
13	Give the team a dedicated open work space*
14	Set a sustainable pace*
15	The project velocity is measured*
16	The customer is always available*
17	Code written to agreed standards*
18	Code the unit test first
19	Integrate often
20	Use collective ownership*
21	Simplicity in design*
22	Choose a system metaphor
23	Use class-responsibility-collaboration (CRC) cards for design sessions
24	Create spike solutions to reduce risk*
25	No functionality is added early
26	Refactor whenever and wherever possible
27	All code must have unit tests
28	All code must pass all unit tests before it can be released
29	When a bug is found, tests are created
30	Acceptance tests are run often and the score is published

3.1.4.2 Agile Practices Regarding Restrictive Architecture Definition

Rrestrictive software architecture is regarded in this thesis as architecture that is strictly related to conception and modelling high level structures and interfaces and does not incorporate elements such as design patterns and implementation. Code is kept separate from the architecture and is instead treated as the realisation of the architecture. The architect is responsible for designing and maintaining the high level structures and interfaces of the software architecture but the implementation is left largely to the development team. The architect can still also be a part of the development team and be involved in the implementation but it is not compulsory.

Taking the definition into account, certain agile practices listed in table 2 were omitted as they no longer were applicable for the architecture design. These are as follows:

1. Code written to agreed standards
2. Code the unit test first
3. All code must have unit tests
4. All code must pass all unit tests before it can be released
5. When a bug is found, tests are created

All the above practices were omitted as they were related to coding activities. The definition of the restrictive architecture excludes activities related to coding and deals solely with conceptualising and modelling the architecture components and interfaces.

3.1.4.3 Description of Practices and Relevancy to Architecture

1. Priorities (Product Backlog) maintained by a dedicated role (Product Owner)
Product Backlog is the artifact for keeping defined and prioritized tasks and requirements. It is maintained by the Product Owner, who represents the needs and demands of the customer. Both architectural and functional requirements and tasks are often listed in the Product Backlog.

2. Development process and practices facilitated by a dedicated role (Scrum Master)
The Scrum Master is regarded as the head of the Scrum team and closest to a traditional "project leader". His responsibility is to facilitate the focus on development and tasks in the sprint for the team. In addition, he handles all the impediments to the development.

3. Sprint planning meeting to create Sprint Backlog
A meeting in the initial stages of the iteration where the agenda is to figure out what functional or architectural requirements/tasks are to be finished during the sprint.

4. Planning poker to estimate tasks during Sprint planning
A technique for estimating the task complexity and duration during sprint planning meetings. Developers have a deck of cards with numbers corresponding to estimates. When estimating the task at hand, developers present the card simultaneously according to what they believe the task estimate to be. The statistical mean or mode is then calculated from all these individual estimates. The estimation is over when everyone agrees on the mean estimate. In the event of disputes, the activity is repeated and the tasks analysed more elaborately.

5. Time-boxed sprints producing potentially shippable output
Routine time-boxed iterations of fixed length to release a new version of software, often lasting from two to four weeks.

6. Mutual commitment to Sprint Backlog between Product Owner and Team
The Product Owner, Scrum Master and the development team are all involved in the sprint backlog and committed to completing the tasks within. In agile environments, software architects are usually also members of the development team.

7. Short daily meeting to resolve current issues
A brief informal meeting for the team members to describe very shortly what he/she has done the previous day and what will be done in further plans. Challenges or issues uncovered are discussed in the meetings.

8. Burndown chart to monitor sprint progress
The burndown chart displays how much work is finished according to time. Several lines can be used for displaying different effort spent but two lines depicting the line of actual burndown and ideal burndown are almost always included.

9. Sprint review meeting to present completed work
Meeting held in the final stages of the sprint, where customers and stakeholders are invited to present the tasks completed during the sprint.

10. Sprint retrospective to learn from previous sprint
Meeting held in the final stages of the sprint, where the development team reviews the practices and general issues that emerged during the course of the sprint. Discussion involves both good and bad practices/issues and improvements are suggested with the process altered accordingly.

11. Release planning to release product increments
Meetings held to plan the product increment releases. These can be planned by measuring the project velocity (how much work is completed in a single sprint) and using data from the burndown chart.

12. User stories are written:
User stories are created to define tasks. The pattern for writing the user tasks is along the following lines:
"As a <stakeholder, customer, role>, I want <goal, requirement, desire> so <motivation, purpose>"

13. Dedicated open workspace
The team should be situated in an open workspace environment to facilitate collaboration and communication.

14. Sustainable pace
The work planned for releases should be possible to complete without requiring the team to work overtime nor making shortcuts in functionality or quality.

15. The project Velocity is measured
The project velocity is a measure of work completed during a sprint. It is necessary to measure velocity in order to plan releases and to determine the feasible amount of work to be included in the iteration.

16. The customer is always available
In the event that issues need further input or clarification, the customer or a representative is always available at all times.

17. Code written to agreed standards
Code must have a consistent standard and formatting. This practice is not relevant to the restrictive architecture definition.

18. Code the unit test first
Test cases are defined prior to writing code. The initial test cases give the developer a clear picture of what is required in order to complete a task, as well as provide constant feedback to the developer. This practice is not relevant to the restrictive architecture definition.

19. Integrate often
Changes made should be uploaded often (at least once a day), as long as the code is written according to agreed standards and works. In regards to architecture, changes related to either code or design should be updated similarly.

20. Use collective ownership
In order to possibly contribute to all main aspects of the project, the developers should feel a strong, collective ownership of the software.

21. Simplicity in design
A simple design should be chosen that is suitable for solving the task at hand. Future enhancements are deemed superfluous to design up-front as they are not defined yet and will only lead to more complex code or architecture design than necessary.

22. Choose a system metaphor
An accessible metaphor for the system must be provided to be able to explain the system design to customers easily. The metaphor is fundamentally a logical representation of the architecture.

23. Use CRC cards for design sessions
CRC stands for “Class, Responsibilities and Collaboration”. CRC cards are used by the development team to evaluate the relationship and messaging between objects and classes. The purpose of the technique is to simply collect and evaluate design ideas.

24. Create spike solutions to reduce risk
Spike solutions are programs aimed to address very difficult technical or design-related problems. The programs are used to investigate and assess potential solutions. Spike solutions can be applied in architecture design by exploring various design alternatives for the system.

25. No functionality is added early
The development of functionality is postponed until it is needed. Adding extra flexibility will slow down the development and could be a potential waste as it may never be required, regardless of how it improves the system.

26. Refactor whenever and wherever possible
Improve the architecture and design whenever as long as it can be replaced by something more appropriate or better. Architecture and design must be adjusted as requirements can change quickly and to insure the software's maintainability and simplicity, constant refactoring is a necessity.

27. All code must have unit tests
A unit test is an assertion about a component of the software, such as a single method or function, which either fails or passes. To ensure code coverage, all code must be tested by dedicated unit tests. This practice is not relevant to the restrictive architecture definition.

28. All code must pass all unit tests before it can be released
No product increments are released if even a single unit test fails. This practice is not relevant to the restrictive architecture definition.

29. When a bug is found tests are created
To address the discovered bug, new tests are written and added to the test suite. This practice is not relevant to the restrictive architecture definition.

30. Acceptance tests are run often and the score is published
Acceptance tests are created based on user stories and used as a type of black box testing. The latter refers to testing the experience of the program, without focusing on the details of the implementation. Every acceptance test must pass in order for a story to be complete and resolved.

The practices and descriptions above were based on the paper on agile practices and effects on scientific software (Sletholt et al., 2011) along with some additional explanations how it could be applied to software architecture design.

3.2 Performance of the Case Study

This section presents the execution of the case study. Section 3.2.1 provides an introduction towards what is considered as software architecture by the architects in Stagnation Lab and some challenges encountered. Section 3.2.2 delves deeper into architectural practices in the context of the three projects explored. Questions were formed and organised so that the data gathered from the participants could be analysed to hopefully provide answers to the research questions set in the thesis.

3.2.1 Introductory Interview

In preparation for a more profound data extraction of agile practices in software architecture development in the organisations, an introductory interview was conducted with the architect of the Klarvinduer project to enquire about typical project lifecycles from the viewpoint of the developers and to ascertain what is considered software architecture in the company. In addition, the introductory interview gave some insight into what questions to formulate for the rest of the case study interviews. The interview was largely unstructured, with some questions prepared in advance in relation to the topic. The first two questions were about finding out how the architect understood software architecture and how it differs from implementation. The motivation for the third and fourth question was to find out what challenges the architect has encountered in architecture design in general and if the current processes set in the company have been positive towards architecture development. The prepared questions asked during the interview are listed as follows:

1. How do you define architecture in the company? *What does it consist of?*
2. How do you differentiate architecture from other implementation?
3. What problems have you encountered with architecture design? *What were the causes?*
4. Would you improve something in the current methods for architecture design and development? *Is there something you would do differently considering the environment?*

3.2.2 Interview Questions for the Case Studies

Following the introductory interview and taking into account insights gathered from it, a more profound interview was constructed for the key architects in the three projects to address the research questions stated in this thesis. Subsequently, the interview was divided into two sections, each one grouping together questions that would be relevant to the respective research question.

Interview questions regarding research question RQ1 were prepared to explore what agile practices were used in the organisations and how they mapped to recommended practices in Scrum and XP. This was required to evaluate the level of agility in the company in comparison to the understanding of what agility is perceived to be by the architects. It is easy for people to say that the organisation is agile. However, a number of architects and developers choose to exclude certain practices deeming them unnecessary for various reasons and the

outcome might be more towards traditional development than is believed. Questions listed towards the end of the RQ1 group, more specifically questions 9 and 10, aim to look into what challenges the architects encountered with software architecture design while working in agile environments.

Addressing RQ1: *To what extent do software architecture and agility support each other?*

1. What do you consider as agile in the way you develop software architecture or software in general? *Why do you consider these things agile?*
2. What practices did you use when designing the architecture? *What steps were used when designing the architecture of the system?*
3. How were the architectural requirements managed and prioritized? *Was there a technical backlog for architectural requirements?*
4. How was the information about the architecture design preserved and distributed for other developers (and new team members for future)?
5. How were architectural decisions communicated among the team?
6. How were architectural decisions communicated to the customer?
7. How much effort was put on defining functional and non-functional requirements in the architecture?
8. How was testing conducted for the architecture design and implementation?
9. What were the challenges encountered during architecture design and development?
10. Looking back, would there be anything you would do differently regarding the architecture design and development?
11. Is there something else you would like to comment or add?

Interview questions regarding research question RQ2 were prepared to explore how much architecture is designed up-front compared to the rest of the development. Although the companies have stated that they follow agile methodologies, the initial phases still follow a more traditional approach with up-front design to some extent without involving rest of the development team for early implementation. Questions 1 and 2 try to find out how much up-front design was believed to be appropriate by the architects in the projects and whether it caused any challenges. Question 3 aims at looking into when is it important to freeze the architecture to provide necessary stability for development to commence. Question 4 is aimed at finding out whether there were any requirements that were not handled well in the process but could have been with more up-front design. Questions 5 and 6 look into how much effort was put into making the architecture adaptable to change, if any. Question 7 tries to explore what the architects opinion is regarding best architectures emerging from self-organized teams.

Addressing RQ2: *How much up-front architecture is appropriate in an agile environment?*

1. How much up-front design was necessary in order for the other team members to start development? *Do you try to wait with freezing the architecture? How is the interplay with development?*
2. Were there cases where this caused problems*? Both in the case of minimal up-front and as much up-front as possible.*
3. Was it necessary at any point to freeze the architecture design so that development could continue? *If yes, then why? Do you try to put as many as possible architectural decisions up-front to be prepared for what comes?*
4. Were there any architectural requirements that were not handled so well but could have been done better if more time was allocated to up-front architecture design?
5. Was the architecture designed to accommodate changes likely to come up in the future?
6. How flexible is the general architecture with regards to changing functionality?
7. What is your opinion on the XP principle about emergent architecture? *Can architecture be developed with minimal or no up-front planning?*
8. Is there something else you would like to comment or add?

3.2.3 Execution of the Interviews

The interviews were conducted in 2016 from March to the end of April. Scheduling the interview dates was more difficult than anticipated as the architects in question were often occupied with work related tasks or had other errands to attend to. The interviews were planned and conducted on the dates in table 3 below. Average length of the interviews was around an hour. Interviews 2-4 were all transcribed afterwards for better data extraction and analysis.

Table 3. Planned interview dates

1	Introductory interview	March 1, 2016
2	Interview with the architect of Klarvinduer	April 4, 2016
3	Interview with the architect of Simote	April 27, 2016
4	Interview with the architect of the Rescue and Fire Department monitoring system	April 30, 2016

3.3 Case Study Results and Analysis

This section presents the results and the analysis of the case study. The summaries of the individual interviews are presented with some conclusions given from what was gathered. In addition, the agile practices from table 2 presented in section 3.1.4.1 were mapped to the actual practices in the companies along with the reasonings behind the answers.

3.3.1 Introductory Interview

This section presents the results from the introductory interview held with the architect of Klarvinduer.

3.3.1.1 Software Architecture in the Company

According to the architect interviewed, software architecture in the company is related to high level design with some elements of design patterns and implementations. Examples brought by the architect were API design, database schemas and setting up the linkage necessary for server to client-side communications. This relates to the non-restrictive definition of software architecture discussed in section 3.1.4.1.

The examples provided by the architect to answer the question how architecture differs from other development include setting up http methods and query parsing for the server so that queries from the server would display the message “Hello, World” to the user. Making the message look nice or the message placement would be considered as development. Another case would be setting up a system of file folders with keys for linking API calls whereas the specific content of the API call, such as defining the responses, would be regarded as development.

To generalise by the examples provided, software architecture is considered as starting from high level models of the system to providing a structured solution which is able to handle a server to client-side communication, database schemas related to the data to be collected and stored and API development as a means to providing the programmers an interface for further development of the system. Everything else related to making the system look nice, integrating web services, using the API and populating databases with new fields is implementation, not architecture design.

3.3.1.2 Challenges Encountered in Architecture Design

The architect mentioned certain challenges related to the architecture design phase. Some of them are summarised in the list below:

- **Minor or no documentation during the architecting phase.**
  As it often occurs in agile environments, the architect stated that documentation is deemed to be an excess waste and nobody really reads the documentation anyway. The developers in general opt to writing self-documenting code where possible instead. On the other hand, this often means that critical design decisions are exclusively in the head of the architect and only he knows the system fully. Memory is

not perfect and problems due to recall occur in time inevitably. It is also harder to bring new people in a project up to speed due to lack of documentation, which means experienced team members have to bring the trainee up to date mostly from their own time.

- **Agreeing on code conventions**.
  The architect of the project often decides the code conventions in the project and that may stir some disagreements among the members of the team during development.
- **Interruptions by team members.**
  The interviewee stated that architecture phase is something that would benefit from the least amount of interruptions as possible. Sometimes the communication overhead by other developers in development is inconvenient and disruptive to the architect. This tends to move towards the traditional architecting with ivory tower visions (Ambler, 2012), when feedback is not incorporated as much to the overall design.

As to the question, would the interviewee improve something with the current process methods in architecture design, the answer was that the present mostly autonomous environment is fine for the developers. He stated that in the initial phases, he would prefer working in solitude with occasional meetings involving the whole team for suggestions in improving the design. Ultimately, the architect would be the one deciding on the best possible architecture design. In addition to that, the interviewee mentioned that if possible, he would like even more time for architecture design so the development could proceed without having to wait for certain components not yet designed or implemented. However due to the nature of agile development, this would be considered anti agile, as the agile principles try to avoid long-term planning.

3.3.2 Interview at Klarvinduer

This section addresses the interview conducted with the key architect of Klarvinduer in terms of the research questions stated in this thesis.

3.3.2.1 Addressing RQ1. To what extent do software architecture and agility support each other?

The architect's view on agility and software architecture supporting each other was relatively mixed. There were aspects of the development that could be regarded as agile relatively well. There were also cases, in which the suggested practices were not followed as much with the architect's explanation on why he thinks that. The way the architect perceived agility is conducting development in iterations or sprints and keeping the customer in the loop at all times. An exception to keeping the customer in the loop for feedback would be during the beginning stages of architecture design, because there isn't much to show that would bring perceived value to the customer. Other than that, his thinking of agility didn't contrast the general opinion of agile development much.

On the account of following agile practices such as Scrum as propagated in the literature, the architect stated that he is not a fan of the practices. He felt that although there is most certainly a need for a project manager / product owner, taking it as far as planning sprints and estimating time for tasks is more negative than positive. In Klarvinduer's case, the project manager naturally acted as a product owner but there was no Scrum Master for facilitating the processes. Although there was a product backlog for the project, no elements of time-boxed sprints were used to deliver results. Naturally this excludes activities such as sprint planning meetings, planning poker, burndown charts for measuring the sprint progress and the mutual commitment between the product owner and team because there was no sprint backlog in usage. Tasks to do were not defined as user stories but they were based on requirements.

There were some elements similar to the practices in Scrum. For example there were no daily stand-ups throughout the project, but the team discussed what they had done and future plans in a more ad hoc way during the day. Similar aspects are with sprint reviews and sprint retrospectives, which were not conducted formally, but the team occasionally held meetings to review what had been done so far and what could be done better when moving forward with the development. Release planning for product increments was done in a relatively flexible way by versioning either by deadlines set by the customer or whenever the team felt enough work was completed for a release.

The architect's reasoning on why developers are not very fond of time-boxed sprints is that developers appreciate freedom to tinker with the implementation until it is at a satisfactory level. This is especially true regarding software architecture as it is the base of all development and should function as well as possible. Dividing the tasks to small increments every week or two brings added overhead of thinking how to break tasks into reasonable pieces as some tasks take more than one or two sprints. Sometimes closing tasks at the end of a sprint to reopen it at the start of a new one (perhaps with a new name) is deemed as needless formality. The architect also mentioned that involving the customer in the loop or versioning depends on whether the developers or architects deem it worthwhile to show or not (unless a strict development process is followed as required by the customer or a third party). It is believed that there is less need in demonstrating back-end development milestones, such as architectural improvements than front-end design changes for example.

Regarding suggested practices in XP, the architect's opinion was largely neutral. Some of the practices he deemed very useful while others not as much. The team had a dedicated open workspace, the pace was sustainable throughout the project, the customer always available when needed, collective ownership of the whole development work and code written to agreed standards, the last one taken very seriously in the company overall. The architect also agreed that extra functionality should only be developed when there is a demand for it. However, there was no mention of measuring the project velocity or using CRC cards for design sessions and no unit tests were used throughout the project. Acceptance tests were conducted occasionally to meet the overall system requirements, mostly nearing the end of the product release and whatever other testing was conducted in an ad-hoc way by the architect or developers. There was mention of building a test environment for the project by the architect which was never quite put to use.

Other than that, the rest of the practices were somewhat agile but not conducted to the full extent of what is stated in XP such as integrating often, using a system metaphor, creating spike solutions and refactoring whenever possible. Refactoring was done more according to need and not just for the sake of refactoring as much as possible. Simplicity in design was initially contradictory meaning that the architect believes more up-front design is necessary in the beginning regarding architecture but as the development commences, a more agile way to deliver architectural changes is followed.

3.3.2.1.1 Agile Practices in the Project

As the architect's understanding of software architecture incorporated both conceptualising and creating high level structures as well as low-level implementation such as design patterns, code structure and standards, the agile evaluation was taken based on the non-restrictive definition of software architecture in table 4.

Table 4. Mapping of agile practices to the activities in the Klarvinduer project.

#	**Agile practices**	**Klarvinduer**
1	Priorities (product backlog) maintained by a dedicated role (product owner)	Yes
2	Development process and practices facilitated by a dedicated role (Scrum master)	No
3	Sprint planning meeting to create sprint backlog	No
4	Planning poker to estimate tasks during sprint planning	No
5	Time-boxed sprints producing potentially shippable output	No
6	Mutual commitment to sprint backlog between product owner and team	No
7	Short daily meeting to resolve current issues	Somewhat
8	Burn down chart to monitor sprint progress	No
9	Sprint review meeting to present completed work	Somewhat
10	Sprint retrospective to learn from previous sprint	Somewhat
11	Release planning to release product increments	Yes
12	User stories are written	No
13	Give the team a dedicated open work space	Yes
14	Set a sustainable pace	Yes
15	The project velocity is measured	No
16	The customer is always available	Yes
17	Code written to agreed standards	Yes
18	Code the unit test first	No
19	Integrate often	Somewhat
20	Use collective ownership	Yes
21	Simplicity in design	Somewhat
22	Choose a system metaphor	Somewhat
23	Use class-responsibility-collaboration (CRC) cards for design sessions	No
24	Create spike solutions to reduce risk	Somewhat
25	No functionality is added early	Yes
26	Refactor whenever and wherever possible	Somewhat
27	All code must have unit tests	No
28	All code must pass all unit tests before it can be released	No
29	When a bug is found, tests are created	No
30	Acceptance tests are run often and the score is published	Somewhat

Overall out of 30 practices listed, there were 8 practices followed that were clearly agile, 13 practices that were not followed and 9 practices that were somewhat agile meaning that they weren't followed exactly as stated in Scrum or XP, but the activities were present in some aspect.

To measure the overall agility in the project based on mapping above, the answers were coded and scaled in a range for a better overview. Answers with *Yes* were coded as +1 for being fully agile, answers with *Somewhat* were coded as 0, meaning that the answer was somewhere between and answers with *No* were coded as -1 for being non-agile. Overall agility was calculated by subtracting the sum of non-agile practices from the sum of fully agile practices. The result was mapped on a scale consisting of values from *Not Agile*, *Slightly less Agile*, *Slightly more Agile* and *Fully Agile*. As seen in the Klarvinduer case in figure 1 below, the practices followed during the project were leaning towards being slightly less agile but were also close to having equally as many agile practices as well as traditional practices or the practices followed were determined as somewhere between.

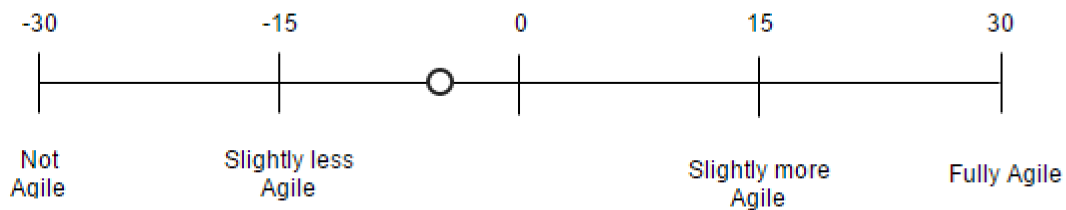

Figure 1. Overall agility measured in the Klarvinduer project.

3.3.2.1.2 Challenges in the Architecture Design

Insufficient architecture documentation

Although agile methodologies have the tendency to be against excessive documentation, there are cases where the absence of documentation can be problematic. In the Klarvinduer case, there were two documents produced for the architecture. One was a picture modelling the relations between the internal server-side components such as the API, database, file server and external components such as payment interfaces. The second one was the database schema model.

The architect expressed that he and the other developers value face to face communication more than documentation because writing and maintaining documentation is considered mostly a waste. On the one hand, you may write documentation that no one will read anyway. On the other hand, the documentation becomes obsolete as the application evolves, which requires time to keep it up to date. Most developers as well as the architect in question resort to self-documenting code instead. The code is broken down to components wherever possible and the components given reasonable names, which makes it relatively straightforward for navigating the source code.

Nevertheless the architect stated that the current policy of documentation may as well change with the introduction of new employees through time. So far every team member

has been involved in a project from the start but as new people are hired and added to a project, there could be instances where a central body of knowledge would be useful instead of having to spend the architects or developers time in explaining how something works. The architect himself stated that as time has passed, there are moments when he didn't really understand the system behaviour due to problems with recalling what was done. However he remained sceptical whether having an up to date documentation would be cost efficient enough to save time debugging the system.

Internal communications regarding design decisions

There were instances where the architect thought he should have been more confident in design decisions. In Klarvinduer, the server-side and client-side architectures started as two separate systems because one of them was based on an older implementation developed by the second architect in an older project. The two systems were eventually merged into one but they continue to have slightly different behaviour and are not integrated perfectly. This could have probably been avoided by taking more responsibility in mandating the design decisions and communicating the decisions more with the other members of the team.

Finding agreements in communicating architectural decisions to the customer

Since architecture design often deals with non-functional requirements, it is difficult to involve the customer in the loop during the architecture design phases. Most of the architectural work is related to back-end development. When customers are paying for a solution, they expect to see results as much as possible. Refactoring the architecture design often yields no perceivable changes but can be critical in order to facilitate further development. In the Klarvinduer case the architect brought an example about some component rules, which defined what to display to the user. Oftentimes the components would be modified which meant spending around half a day to implement the changes. As this wasn't feasible, the interviewee spent a week to refactor the design so that the same changes would take minutes instead of hours. The customer's response to those changes was questioning why it took that much time for a seemingly simple change and it took effort to explain why it was necessary. This leads to changes sometimes being implemented as part of bigger tasks or implementing the changes without necessarily involving the customer until it is done. The architects reasoning being that sometimes it is easier to explain design decisions to the customer after they have been implemented and proven to simplify further development. Otherwise there's the risk of the customer denying any such development beforehand or opting the developer to cutting corners in design or implementation.

The issue related to customer collaboration is that implementing changes without involving the customer or communicating the decisions afterwards tends to be a more traditional approach. In agile development, one should communicate all decisions to the customer before implementation with both having mutual agreement. Inability to explain or prove up-front why certain improvements would be necessary and the customer rejecting the proposal should result in neglecting the development.

No automated testing

Testing is perhaps the most controversial practice in the company which is advocated in the agile methodology and in XP especially. Since the company was conceived, there hasn't been put much effort on automated testing practices. The architect stated that supposedly there has to be a test solution for every architecture. In the Klarvinduer case, the architect

had built it and made a few tests, demonstrated it to the others but not much interest was shown towards it by the other developers and it was quickly forgotten. This has been the case for a couple of years now. The subject is discussed at least once a year but the practice hasn't been picked up.

Part of the reason for omitting testing in development has been that there have been very few problems over the years. There have been instances when an automated test could have been useful for preventing a problem or two, but they haven't been severe enough to start testing regularly. The architect stated that one of the reasons why he doesn't like testing is that they take a lot of time to set up. It is similar to documentation, which has to be kept up to date consistently. Another statement the architect made is that they just do not develop bugs that often. When looking back at the issues that generally come up, there are some where an automated test would have proven useful, but there are more issues in which cases you don't write tests for them. In Stagnation Lab, the errors that come up are more related to code linking and style, rather than syntax errors for which writing test cases is more straightforward.

However, the architect didn't say that testing is necessarily a bad practice. There are times when the presence of tests is greatly missed. For example when conducting large-scale refactoring, like updating a framework version. You cannot be sure if the application is working as lots of methods are deprecated and rewrites will be necessary. Thus architecture of the system can also suffer. The architect concluded that it is something that will probably change as time goes on and new developers are recruited.

3.3.2.2 Addressing RQ2. How much up-front architecture is appropriate?

As stated earlier in section 3.1.2.1, the general process followed during architecture design in the company is more traditional than agile, as was also the case with Klarvinduer. This means that the architect is in charge of designing the software architecture in the initial phases and the rest of the team is involved mostly during the later phases of the architecture implementation. The interviewee stated that when designing the architecture in Klarvinduer, he didn't want to involve the rest of the team besides from occasional meetings until most of the architecture was more or less complete. Some tasks related to build and deployment were left near the release but other than that, there were no major improvements with the architecture in general after the rest of the team was involved. Most of them were related to changes unforeseen in the initial design and discovered during the implementation.

Some involvement from team members is still appreciated by the architect. Mainly during initial brainstorming sessions and for reviewing the architecture design. The architect stated also that some development work can be performed parallel to the architecture design, such as front-end development in HTML/CSS. Although the architect is in charge of the system architecture, input from the rest of the development team is still welcome when the implementation has commenced.

In the first introductory interview, the architect had stated that given the chance, he would have liked even more time to work on the architecture. During the second interview he agreed that he was relatively satisfied with the state of the architecture when the rest of the team was involved. More architecture design could have resulted in less issues during the implementation but then again it is similar to testing a prototype with actual users - negative

feedback in the later stages would mean that a lot of development was potentially wasted and has to be scrapped. When asked if he could have involved the team earlier than he did in Klarvinduer, the architect said that he wouldn't have preferred that. The architect believes that development goes faster if more design is done up-front than less. The rest of the team encounters less issues building on a more stable architecture. Stability in this case meaning that components perform as expected and the overall structure is intact. In the case of Klarvinduer, the developers had to be able to write API methods, create new views, run the application and have a database. Until these capabilities were implemented, it was risky to involve the rest of the team as the architecture went through various changes in the initial phases, which would have meant lots of redevelopment or having to wait for the architect to add a certain component for the developer to continue work. As the architecture is error-prone initially, this could also make the developer underperform as it would be difficult to determine whether faults in software behaviour are related to the developer or architecture.

While not being necessarily against agility, the architect doesn't believe that architecture can be developed with minimal planning as stated in XP for example. This approach could work in a one or two person project with possibly more effort spent on communication, but generally speaking if one would put two developers implementing the same system without any predefined structure, each developer would emerge with their own architecture. He agrees though that there is no point in thinking outside the scope of the current project. Planning for extra functionality is regarded wasteful as you either won't be needing it or when you need it, you learn that you couldn't anticipate the changes as they came anyway and have to either redesign it entirely or do major overhauls. Lastly, time is another factor that defines how much effort can be spent on architecture. As the deadline approaches, there comes a point where architecture design just has to be frozen to provide enough stability for the implementation to be completed.

3.3.3 Interview at Simote

This section addresses the interview conducted with the key architect of Simote in terms of the research questions stated in this thesis.

3.3.3.1 Addressing RQ1. To what extent do software architecture and agility support each other?

The architect interviewed was more supportive towards agile development than the first architect in Klarvinduer. His understanding of agility was responding to change in fast developing environments. As the development advances, changes are expected to be inevitable as the understanding of the domain increases and the customer needs evolve through time. Agile methods enable to adapt to those changes and reprioritise goals and ideas.

He also stated that in some cases, working fully agile in a sense that a team would follow all the suggested practices is not always reasonable. It would introduce unnecessary formality similar to what the architect of Klarvinduer had stated. When a team is very small with a unified vision of what they want to achieve, there is no significant need to having activities such as backlog grooming or planning poker. Provided that a team scales up by adding more people to the team, it makes sense to apply more formal activities. Thus in the case of Simote, most of the Scrum related activities regarding sprints were missing. Because the product built was the company's first owned product, the architect acted also as the product owner in charge of the development. Development was flexible meaning that there was no fixed date for the product to be finished even on the customer side. Some investment in the development was also spent by the company itself to improve the product. User stories were substituted by requirements and no sprint backlogs and related practices were present. Similar to the Klarvinduer project, there were elements from daily sprints, sprint reviews and sprint retrospectives conducted in a more ad-hoc way throughout the development. No effort was put on release planning and the product was released to the customer when development was more or less finished.

Regarding suggested practices in XP, the practices followed were similar to the Klarvinduer project. The environment provided an open working space and the pace was sustainable. Integrating was also conducted often and extra effort was allocated to standardizing the code. The architect approved the idea that no functionality should be added early as there is no knowing whether the functionality will be needed or not. However some high level up-front thinking in designing the architecture to accommodate future changes is considered by the architect to be a good investment. Refactoring was also considered by the architect a necessity and something not to fear.

Practices not followed included measuring velocity, using CRC cards for design sessions, collective ownership and unit testing. The architect brought out that he has throughout time described himself as a control freak in teams. He has preferred to be in charge of maintaining the architecture, structures and standards, although lately he expressed that it is not always reasonable and has started growing trust in the whole team. The reasoning for wanting to be in charge is that it enables a more unified vision opposed to the whole team setting their own standards and ending up with a mix. In a sense, it leans towards the concern for ivory tower architecture discussed in the software community (Ambler, 2012). No unit tests were used in the project but every team member was responsible for the correctness of their code and testing was conducted in an ad hoc manner.

Rest of the practices were somewhat agile but not conducted to the full extent of what is stated in XP. The customer was available when needed but as the scope of the architecture and development was agreed in the initial design sessions, the customer wasn’t involved in the process as often. Simplicity in design was important but extra effort was spent on isolating the components for example, which was not mandatory but practical for future development purposes or to facilitate adding new members to the project when needed. The architect mentioned structuring the architecture to conform to system metaphor but it is not fully clear to what extent and throughout the design and implementation, various ways to deliver the solution were experimented with that can relate to spike solutions. Acceptance testing was performed in the later stages of the project but not consistently.

The architect's opinion on whether architecture design and development support each other or not was generally positive. One does not exclude the other and taking a reasonable approach to incorporating agile techniques to architecture can improve the experience. However, some things such as reducing up-front planning in architecture cannot be entirely avoided and the architect believes it makes more sense to not involve the team in its entirety in the beginning phases of architecture design, regarding implementing functionality. Nevertheless feedback for the initial design is by all means welcome from team members, depending on the level of experience and predetermination.

3.3.3.1.1 Agile Practices in the Project

Similar to the architect in the Klarvinduer project, the architect's understanding of software architecture incorporated both conceptualising and creating high level structures as well as low-level implementation. Thus, the agile evaluation was based on a non restrictive definition of software architecture in table 5.

Table 5. Mapping of agile practices to the activities in the Simote project.

#	**Agile practices**	**Simote**
1	Priorities (product backlog) maintained by a dedicated role (product owner)	Yes
2	Development process and practices facilitated by a dedicated role (Scrum master)	No
3	Sprint planning meeting to create sprint backlog	No
4	Planning poker to estimate tasks during sprint planning	No
5	Time-boxed sprints producing potentially shippable output	No
6	Mutual commitment to sprint backlog between product owner and team	No
7	Short daily meeting to resolve current issues	Somewhat
8	Burn down chart to monitor sprint progress	No
9	Sprint review meeting to present completed work	Somewhat
10	Sprint retrospective to learn from previous sprint	Somewhat
11	Release planning to release product increments	No
12	User stories are written	No
13	Give the team a dedicated open work space	Yes
14	Set a sustainable pace	Yes
15	The project velocity is measured	No
16	The customer is always available	Somewhat
17	Code written to agreed standards	Yes
18	Code the unit test first	No
19	Integrate often	Yes
20	Use collective ownership	No
21	Simplicity in design	Somewhat
22	Choose a system metaphor	Somewhat
23	Use class-responsibility-collaboration (CRC) cards for design sessions	No
24	Create spike solutions to reduce risk	Somewhat
25	No functionality is added early	Yes
26	Refactor whenever and wherever possible	Yes
27	All code must have unit tests	No
28	All code must pass all unit tests before it can be released	No
29	When a bug is found, tests are created	No
30	Acceptance tests are run often and the score is published	Somewhat

Overall out of 30 practices listed in the table above, there were 7 practices followed that were clearly agile, 15 practices that were not followed and 8 practices that were somewhat agile meaning that they weren't followed exactly as stated in Scrum or XP, but the activities were present in some aspect.

Measuring was conducted similarly as in the Klarvinduer case in section 3.3.2.1.1. As seen in the Simote case in figure 2 below, the practices followed during the project were leaning more towards being slightly less agile, even though the architect of Simote was more favourable of agile methodologies than the architect of Klarvinduer. This is probably related to the overall team size being very small and consisting mainly of 1 architect and 2 developers. Like the architect had stated, formal following of the suggested practices was deemed unnecessary in such a case. In addition, collaboration with the customer was also less active than in Klarvinduer.

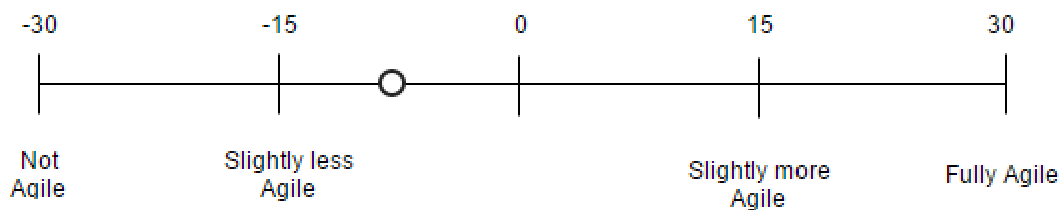

Figure 2. Overall agility measured in the Simote project.

3.3.3.1.2 Challenges in the Architecture Design

Managing the complexity of the architecture

This is mostly related to the XP practice of simplicity in design. The architect stated that one of the biggest challenges in designing the architecture is keeping the complexity relatively low or hiding the complexity in the components or modules. The idea is to spend more effort in keeping the relations and interfaces between the components as simple as possible. This eliminates the need for understanding the whole system for other developers also. In addition to that, it would make it easier to test the architecture and functionality with unit tests, which would, on the one hand, help discover issues and on the other hand serve as documentation for new developers, as the tests cover the functionality and help understand what the system should do. The architect mentioned also that it is a common sight for junior architects and developers to underestimate the complexity and while learning new methods and techniques, they tend to apply those complex methods or algorithms in their projects, which often might not be the simplest way of solving problems.

However keeping the design simple also means more effort is spent on designing the system up-front which leads to delays for involving developers to start with the implementation. As a consequence, this leads to the development being less agile at first.

No clear scope from the customer

There is a certain amount of danger related to agile development and architecture, when insufficient amount of analysis is done prior to development or the customer doesn't have a clear scope of what he wants to achieve. The problem with agile methodology is that while change is considered as inevitable, you should still have a general overview of what the goal of the project is. Otherwise when the customer has a vague idea of what is to be done and the development has commenced and modified as new information is required, it will also lead to major redesign and waste. This would be fine if the customer has endless resources but it is not the case in real life. He stated that oftentimes the customer requires a simple system with limited functionality and the architect is able to devise a reasonable architecture for the required system. However, when few months have passed, the customer decides that he wants to add a lot of new functionality mid-development, in which case the devised architecture would probably not be able to support it. Especially when the principle of simple design is followed and no extra effort is put on, for example, making the architecture more adaptable or dynamic to accommodate potential changes. Thus it is important to agree on at least a general scope in the beginning before commencing architecture design and development. The architect considers the best input for architecture and design to be a clickable prototype or a wireframe of the solution required.

No automated testing

Similar issue as in the Klarvinduer case. Testing was done in an ad-hoc way with the architect and developers each testing their own design or code. There were no automated nor unit tests in the project. Since the project was developed in basically the same environment as the Klarvinduer project, then the same reason was brought out by the architect that the whole team has encountered relatively few issues throughout time. Likewise he also stated that it is something that should be taken into more consideration in the future. The other factor being that since most of the projects have incorporated more user interface related design and development than applications mainly related to back-end development, then it is hard to write tests for proper user interface behaviour. The architect believes that the effort spent so far on quality development has been more justified than spending extra time writing quality test cases to cover the applications.

In the case of Simote, although the system developed was quite complex involving many components, it was easy to keep track of what people had contributed when the team consisted mainly of 3 members. If the project would have been conducted in larger scale, the architect would advocate incorporating more systematic testing to the overall development.

Another problem regarding testing is that it is costly for the customer or that completing the project for a specific date is more important than assuring that the application is working without issues. The customer sees testing as a too high cost and opts for the development to rather skip testing or allocate a small amount of resources to the testing of the whole application. The combination of not focusing on testing and pushing deadlines is a recipe for the architecture to potentially suffer also along with the rest of the development.

3.3.3.2 Addressing RQ2. How much up-front architecture is appropriate?

Similar to the architect of Klarvinduer, the architect interviewed prefers to make the major architecture design decisions in the early phases. The architect stated that as the development team members on the project were also not that experienced at the time, he had to design and implement most of the structure up-front for the others to join development. Had the developers been more experienced, there would have probably been more flexibility regarding architecture design decisions and team involvement.

The architect also discussed that depending on the project size and complexity, it is better to have a coherent vision from the start. In very small projects in a familiar field, one could postpone making any architectural decisions and get away with it by simply refactoring the design throughout time. In larger projects, it makes sense to design and make major decisions more up-front. He reasoned that if not much effort is put on up-front design and commencing with implementation, it will take a lot of time to redesign when learning that the current approach is not working. In addition, it is easy to complicate the design and implementation compared to spending effort on decent architecture design initially.

The architect also discussed that not only should one opt for the simplest design. He previously explained that hiding the complexity in different components as much as possible is one substantial challenges in architecture design. Naturally this increases the effort spent on architecture design but at the same time, involving the rest of the team is easier as they do not necessarily have to know how every component works as long as the interfaces work properly. It gives also the added benefit of testing the components. If one is not able to cover the entire complexity of the component with tests, then at least it is possible to cover the overall functionality of the component to validate its behaviour.

As mentioned also in the Klarvinduer case regarding freezing the architecture, this is again one reason why designing more up-front is justifiable as the deadlines may also rush the architect to make hasty design decisions later in the development when delaying design decisions. If the architecture performs as required, then there is no more point in spending extra time on the architecture and it is best to preserve the architecture stability for development. The architect did say though that although there is no point in developing extra functionality as it might not be needed, it is good to invest some time in designing the architecture to be more adaptable. One should not implement extra functionality but thinking the design through in terms of possible future development is believed to be good investment by the architect.

3.3.4 Interview at the Fire and Rescue Department Monitoring System

This section addresses the interview conducted with the key architect of the Fire and Rescue Department Monitoring System in terms of the research questions stated in this thesis.

3.3.4.1 Addressing RQ1. To what extent do software architecture and agility support each other?

The architect's understanding of agility was about involving the customer throughout the whole project lifecycle as often as possible. He stated that agile methodologies suggest practices to keep the customer in the loop and deliver product increments in short development cycles. The main idea is to deliver results as quickly as possible and iterate based on the feedback received from the customer. When asking if he is supportive of the agile methodologies, he responded that he absolutely supports agile development and stated that traditional development is significantly more expensive and inefficient.

Most of the Scrum related practices were followed in the architecture design and development in the project. The architect described that the team had chosen Scrum because previous experiences with more traditional waterfall methods had been problematic for some time. The development was conducted in sprints lasting up to two weeks. The team held sprint planning meetings where tasks were taken from the prioritized product backlog. In sprint planning meetings, tasks were written down as user stories and estimated by the planning poker activity to come up with a sprint backlog for the upcoming sprint. In addition, release planning was also conducted at the start of the sprint. When the sprint started, daily meetings were conducted either at the workplace or through call conferences when team members were distributed at times in different offices. The sprints ended with a sprint review and a sprint retrospective respectively. The only Scrum practices not followed were the absence of a Scrum Master and monitoring the sprint progress with a burndown chart. There was also some conflict with maintaining the product backlog by the product owner that the architect brought out. He stated that more effort should have been put on prioritizing the backlog and keeping it up to date and the product owner in the project wasn't very fond of the role and was also occupied tending to other work assignments.

XP practices followed throughout the project included having a sustainable pace, collective ownership to the software, keeping the design as simple as possible and also adding no functionality until needed by the customer. Acceptance testing was also performed throughout the project by the product owner and analyst in the team. Only using CRC cards for design sessions was not followed.

Rest of the practices were somewhat agile but not followed to the full extent. Most of the time, the team had an open working environment but on some occasions, the team was split between two cities. There was no explicit mentioning project velocity but it was assumed to be somewhat present and although integrating was certainly important, it remained unclear in the interview how often exactly it was conducted. Regarding the customer being always available, there was some contradiction. The architect stated that the customer was involved at the end of sprints to provide feedback but it would have been better if the customer had been involved even more during work with the backlog. That is because on several occasions, many tasks were performed during sprints and the customer desired something else instead when seeing the completed work. The architect discussed also using metaphors in the architecture design but it was not clear to what extent exactly and different designs were conceived and discussed with development team members to assess whether something was

reasonable to implement or not, which can be related to spike solutions. Refactoring was done more according to planned iterations. The architect stated that although refactoring is necessary and inevitable in changing environments, it should still be kept minimal as it is costly and inconvenient for the development team. This means that refactoring the architecture induces rework not only for the architect, but for the developers also.

When discussing the question whether architecture and agility go well together, the architect had differing opinions. While being supportive of agile methodologies, he stated that architecture should be regarded as a separate process and not bundled in with other development. In the project, the architect was also a member of the development team and was part of implementing the system in addition to designing the architecture. Nevertheless he stated an architect shouldn't be the member of a Scrum team or there should be another Scrum process for architecture design. In his opinion, Scrum should involve activities and tasks related to implementation of the solution, but he has seen processes where even activities such as recruiting new employees or vacations are listed in the backlogs. In an ideal scenario, he would move architecture design to be part of requirements elicitation and analysis, but in Estonia, there is too little effort put on planning. He did not imply that it should be done in a traditional way but more effort should be spent for planning in general in agile environments. However, this does not mean architecture design should be done in ivory tower settings but should be communicated to the stakeholders early and often. In that regard, agile development in principle supports the architecture design.

3.3.4.1.1 Agile Practices in the Project

In this case, the architect's understanding of software architecture was that it must be separated from code, thus the architecture design deals strictly with conceptualising and modeling the high level structures of the system without regard to implementation of those structures. The rest was deemed as the realisation of the software architecture. The agile evaluation of the project was based on the restrictive definition of software architecture.

Table 6. Mapping of agile practices to the activities in the Fire and Rescue Department monitoring system project.

#	**Agile practices**	**Monitoring system**
1	Priorities (product backlog) maintained by a dedicated role (product owner)	Somewhat
2	Development process and practices facilitated by a dedicated role (Scrum master)	No
3	Sprint planning meeting to create sprint backlog	Yes
4	Planning poker to estimate tasks during sprint planning	Yes
5	Time-boxed sprints producing potentially shippable output	Yes
6	Mutual commitment to sprint backlog between product owner and team	Yes
7	Short daily meeting to resolve current issues	Yes
8	Burn down chart to monitor sprint progress	No
9	Sprint review meeting to present completed work	Yes
10	Sprint retrospective to learn from previous sprint	Yes
11	Release planning to release product increments	Yes
12	User stories are written	Yes
13	Give the team a dedicated open work space	Somewhat
14	Set a sustainable pace	Yes
15	The project velocity is measured	Somewhat
16	The customer is always available	Somewhat
17	Integrate often	Somewhat
18	Use collective ownership	Yes
19	Simplicity in design	Yes
20	Choose a system metaphor	Somewhat
21	Use class-responsibility-collaboration (CRC) cards for design sessions	No
22	Create spike solutions to reduce risk	Somewhat
23	No functionality is added early	Yes
24	Refactor whenever and wherever possible	Somewhat
25	Acceptance tests are run often and the score is published	Yes

Overall out of 25 practices listed in the table above, there were 14 practices followed that were clearly agile, 3 practices that were not followed and 8 practices that were somewhat agile meaning that they weren't followed exactly as stated in Scrum or XP, but the activities were present in some aspect.

Measuring was conducted similarly as mentioned in the Klarvinduer and Simote cases in sections 3.3.2.1.1 and 3.3.3.1.1 respectively. As seen in the monitoring information system case in figure 3 below, the practices followed during the project were slightly more agile compared to previous projects. The architect had stated that he fully supports agile methodologies but as the size of the project was also relatively small with mainly 2-3 developers up to 5 developers at peak times, it seemed more reasonable to not opt for all practices suggested in Scrum.

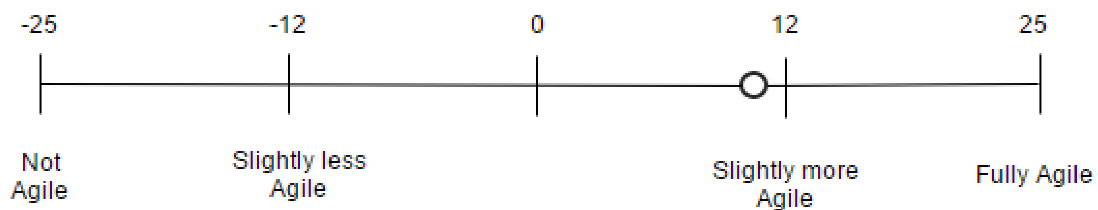

Figure 3. Overall agility measured in the Fire and Rescue Department monitoring system project.

3.3.4.1.2 Challenges in the Architecture Design

Insufficient effort spent on managing the Product Backlog

Although the project was developed in a relatively agile environment, the architect expressed concern that more work should have been done by the product owner in regards to planning and grooming the backlog. This often resulted in vague or incomplete requirements from the customer and there was much redesign and development both for architecture and implementation. The architect stated that the 10 months it took to deliver the solution was fine, but it could have also been done in just half a year, had there been more effort spent on the backlog and planning the development. However, the architect did not say that the planning should have been done necessarily in a more traditional way and up-front. If an extra person would have been allocated to spend more effort on the backlog, it would have helped deliver the solution faster. Then again, oftentimes keeping the backlog up to date relied also on the availability of the customer, who couldn't invest as much time in the project due to being also tied to other work assignments in the Fire and Rescue Department.

Getting the team to accept change

A challenge in agile environments regarding architecture and development is getting the whole team to accept the fact that change is inevitable. To the architect, change and subsequent refactoring is normal and part of the process in agile software development. To the developers, change is inconvenient as it means a lot of effort is wasted. There were some occasions in the project where it was difficult getting the development team to understand and agree with what was being done or suggested. The architect noted that it sometimes lead

to disputes in the team even. The challenge for the architect was to keep refactoring to a minimum as changes in the architecture would in turn induce changes in the implementation also.

The way the architect tries to solve this is by dividing the architecture into different levels. The higher level structures are set usually in the beginning of the architecture design process and will not be changed. As the organisation is working in a field highly regulated by laws, the changes are introduced at higher levels usually when the laws are changed. The changes that affect development appear when going towards lower level modules and components. Similarly to what the architect of Simote had said, some effort is spent on isolating the modules so it would have minimal impact on the overall architecture and implementation, should refactoring become a necessity. The benefits of this approach is that sometimes modifications in architecture mean minimal changes in code, such as changing a configuration file or renaming files.

No automated testing

Similar to the first two projects conducted in Stagnation Lab, there was again little effort spent on testing. As the organisation was relatively large and had to conform to organisational standards, the architecture design of the project was tested or more specifically validated in a central technical architecture review committee. Unless the architecture was reviewed, no permission was given to start implementation. At the project level, however, there was little testing conducted throughout the project. This was due to two reasons: management not allocating enough resources and the developers being relatively passive towards the activity. Although the architect also justified the lack of testing by stating that they encountered very few issues. Oftentimes testing is deemed too costly for the management or there is no understanding of the importance of tests. The architect stated that to some extent, acceptance tests conducted by the customer are fine, but if there was a need to test the software after every release, it would become tedious and time wasting without automated tests.

3.3.4.2 Addressing RQ2. How much up-front architecture is appropriate?

Since SMIT is a larger organisation operating in a public infrastructure, systems developed have to conform to certain standards. To that reason, the organisation employed a technical architecture review committee. This committee oversaw the whole organisation and every project had to gain approval from the committee before commencing with implementation. Thus it was necessary to design the high level architecture up-front and the architecture was enclosed after review. Should there be a need to redesign at a high level, further feasibility studies are conducted to determine if the changes to be performed are beneficial. If yes, then the committee can allow high level architecture redesign. The architect also stated that work on the rest of the architecture was approximately only 5% of the entire development. This was due to having an existing system in place and the main task was just updating the system to new technology.

Regarding decisions on a lower level, the architect had the opinion that decisions don't have to be taken as early as possible. What matters is that the design decisions are communicated as often as possible with the most relevant stakeholders. Since architecture delivers more value to the developer than the customer, then design decisions are communicated mostly

with the developers. This is necessary to gather feedback about the design fast and find out if what the architect has designed is reasonable to implement or takes too long for example. Likewise with the architect of Simote, the architect interviewed said that he tries to design his architecture to be relatively adaptable by isolating the components. This brings the added benefit in future refactoring so that changes affect implementation minimally, although taking more time to design initially. When asked what the architect thinks about the XP principle, where architecture emerges through implementation and refactoring, the architect said that he hasn't seen it in practice and doesn't believe in that. It may be applied to very simple systems with one or two people but large information systems are too complex to just start building without thinking the design through first.

Like the architect of Simote had said, there are no rules to determine when the architecture is enough and has to be enclosed for the implementation to commence or continue until release. When the architecture enables the implementation of all requirements, then the rest is more about fine-tuning the details. For the most part, the architect stated he doesn't design architecture further than classes and relations between them. Attributes are left for the implementation. When asking if the architect had any other comments regarding the topic of agility and architecture, he had the opinion that too little effort is put on planning in general in organisations. He believes that in Estonia, people don't specialise in software architecture. People called architects in larger organisations are titled as principal or senior developers. A software architect should be a person dealing strictly with conception, a process manager for example. He feels that more distinction should be made between software architects and developers in general whereas in Estonia, people tend to jump into implementation as fast as possible. This approach brings both benefits and drawbacks. Benefits being faster releases and drawbacks including unstable architecture evolution.

3.3.5 Summary of the Results

This section summarises the results of the conducted case studies. A table of practices relevant to the architecture design along with the accounts of the architects interviewed is listed in section 3.3.5.1 followed by summaries of the research questions in this thesis.

3.3.5.1 Agile Practices in the Projects

The agile mapping chart of the three projects is presented in table 7 below. While the first two cases were less agile, the third project followed majority of the recommended practices. The blank cells in the monitoring information system case indicate that the practice did not apply due to the architecture definition applied.

Table 7. Summary of agile practices in the projects.

#	**Klarvinduer**	**Simote**	**Monitoring System**
1	Yes	Yes	Somewhat
2	No	No	No
3	No	No	Yes
4	No	No	Yes
5	No	No	Yes
6	No	No	Yes
7	Somewhat	Somewhat	Yes
8	No	No	No
9	Somewhat	Somewhat	Yes
10	Somewhat	Somewhat	Yes
11	Yes	No	Yes
12	No	No	Yes
13	Yes	Yes	Somewhat
14	Yes	Yes	Yes
15	No	No	Somewhat
16	Yes	Somewhat	Somewhat
17	Yes	Yes	
18	No	No	
19	Somewhat	Yes	Somewhat
20	Yes	No	Yes
21	Somewhat	Somewhat	Yes
22	Somewhat	Somewhat	Somewhat
23	No	No	No
24	Somewhat	Somewhat	Somewhat
25	Yes	Yes	Yes
26	Somewhat	Yes	Somewhat
27	No	No	
28	No	No	
29	No	No	
30	Somewhat	Somewhat	Yes

3.3.5.2 Agile Architecture and the Challenges Discovered

All the projects explored in the case study followed agile practices to some extent. Cases involving Klarvinduer and Simote didn't follow a formal process whereas the Rescue and Fire Department monitoring system team followed Scrum with few modifications. All teams were relatively small, averaging from 3-5 team members. During peak time, the Klarvinduer project involved up to 8 people when implementation had commenced. Two architects out of the total three were supportive of agile methodologies and had stated that architecture and agility can be combined in the sense that whatever work finished should be communicated to the customer to receive feedback as early as possible. The architect in the Klarvinduer case had a mixed opinion related to agile development. He was not opposing the methodologies in general but he felt that the managerial practices in Scrum produce too much overhead for the development. Excessive meetings such as recurring sprint planning meetings, sprint reviews, sprint retrospectives and also daily stand-ups are considered somewhat waste to the developers. The daily stand-ups bring little value to a few members of the team whereas taking up hours when taking into account the whole project lifecycle. He stated that if there was some knowledge that he would need, it would be easier and quicker to just ask the appropriate team member during the day or when the need presents itself. Needless to say, this approach would be more complicated if the team was divided between different locations.

XP practices were generally more favoured at least to some extent compared to Scrum practices. Every team worked at a sustainable pace, had a dedicated open workspace and everyone agreed that no functionality should be added early. Every project had at least reasonable involvement by the customer. Reasonable means that no work had to be put on hold due to the customer involvement. Refactoring was also regarded as necessary due to changing environments. The architects in Klarvinduer and the monitoring system mentioned though that refactoring shouldn't be done for the sake of it, but only when really necessary and preferably even kept to a minimum as it induces changes both to architecture and implementation.

There were notable practices avoided also in every project. None of the projects employed a Scrum Master and there was no mention of using burndown charts for measurement in the projects. Regarding XP practices, there was no mention of using CRC cards for design sessions and no unit testing was applied in any project. The teams tested their corresponding work and made sure that what was implemented or designed worked as required in an ad hoc manner. However, acceptance testing was performed mostly during the end of projects in Stagnation Lab and more often in SMIT.

Various challenges were brought up by the architects when working with architecture in agile environments. Prior to discussing the issues stated, it should be mentioned that overall the architects described few problems that affected the project outcome crucially in some way. The general consensus was that the projects were completed rather well with only minor disturbances along the way. Similarly there were not many architectural challenges encountered and challenges described applied for systems and architecture development in general, not necessarily in terms of the projects discussed. The challenges along with the number of mentions are listed in table 8 below:

Table 8. Challenges raised by the architects interviewed regarding agility and architecture.

#	Challenge	Occurrence
1	No automated testing	3
2	Managing the complexity of the architecture	2
3	Finding agreements in communicating design decisions to the customer	2
4	No clear scope initially from the customer	1
5	Insufficient effort spent on managing the Product Backlog	1
6	Internal communications regarding design decisions	1
7	Insufficient architecture documentation	1
8	Getting the team to accept change	1

None of the projects applied much formalized testing effort in the projects, such as unit testing advocated by the Scrum practices although it didn't induce notable challenges in the projects explored. Common justification for omitting these practices were that the teams had had relatively few issues throughout time, both in Stagnation Lab and the project explored in SMIT. Nevertheless, all the architects agreed that the practice should be followed more often as it can eventually lead to negative consequences. Architecture can suffer from the lack of unit tests in large scale refactoring, such as updating frameworks, where many former functions and methods get deprecated for instance and can go unnoticed.

Another reason why testing is omitted is that it is not backed by management enough. Testing is deemed costly and thus often neglected. Implementing a system with test driven development (TDD) for example would cost twice as much as implementing without tests. What is done is that the customer usually allocates a fraction of the budget for testing purposes but the risk remains that issues will not be uncovered as effectively. More important than the quality is meeting the deadline (preferably even sooner than anticipated). The architects agree that when the projects are scaled, this approach would be deemed rather negative.

The architects in Simote and the Fire and Rescue Department monitoring system mentioned managing the complexity of the architecture as one of the challenges in agile environments. Both preferred to spend extra effort on designing the architecture to isolate the components and hide the complexity within the components, whereas agile practices suggest to keep the overall design as simple as possible. This is somewhat contrasting to the agile principle but it is done for better comprehension of the architecture and helps to achieve two things. One is the easier management of the architecture itself and recruitment of new people in the project as the new developer doesn't need to understand the whole system to join the project. Second reason being the fact that it helps minimize the effects of refactoring the architecture or the implementation. The architect of the monitoring system explained that sometimes

refactoring would only mean minimal changes in code, such as renaming or modifying a configuration file for example.

The third challenge introduced by two architects was related to communicating the architectural decisions to the customer. Agile methodologies advocate to keep the customer in the loop as often as possible. Whereas implementation of functionality brings value to the customer, architecture design brings value mostly to the developers. Thus it can be complicated to communicate architectural decisions to the customer. For example the architect of Klarvinduer explained that occasionally it is easier to improve the architecture and communicate the decisions after improvement. Otherwise there can be instances where the customer would not allow spending as much effort on architecture and recommend "hacking" or finding shortcuts to meet the requirements, instead of allocating more resources to fixing issues properly. Customers with a more technical background understand that such an investment is redeemed in the long term whereas nontechnical customers would have a harder time approving more costly architectural improvements.

Challenges 4-8 in table 9 were mentioned fewer by the architects but were still interesting. The architect of Simote brought out one threat in agile environments regarding architecture design when the customer does not have a more or less clear scope from what the expected project outcome is to be. He explained that in some cases, there would be a customer who initially wants a simple solution with a certain set of functionality. The architect is able to devise an appropriate architecture for the solution but a couple months later the customer would decide that he requires a more complex solution than originally described. Subsequently the architecture would suffer also by adding the required functionality and overcomplicating the design. This is also one reason the architect advocates more up-front analysis and design than commencing with implementation as soon as possible.

The architect in of the monitoring system encountered two more challenges during the project. One was insufficient effort spent on managing the product backlog and the other justifying the importance of changes through refactoring. As the dedicated product owner did not take his role too seriously, it led to numerous redevelopment that could have been avoided if the product owner had been collaborating with the customer more. Naturally the added refactoring was often not received well by the team. He explained that although change in agile environments is inevitable, too much refactoring can be frustrating for the morale of the development team. Thus it is important to minimize the amount of refactoring needed. There were a few more challenges affecting architecture brought out by the architect in Klarvinduer such as communicating design decisions internally among the team. While he had stated that he preferred to mostly work on the architecture himself with occasional feedback from other team members, this led to some inconsistencies in the system that could have been avoided with more communication with the developers. He also described that fixing these architectural inconsistencies is hard to communicate later with the customer also as is the case with the challenge of finding agreements discussed previously. Finally it was mentioned that some issues may arise from insufficient documentation in agile environments. The agile principle of working software over comprehensive documentation is at times taken too far by providing almost no documentation at all. This means that oftentimes, the architect is the only one with a full overview of the system. Even the architect himself stated that the details of the system are forgotten over time and extra effort would have to be spent when returning to the project after a certain period of time. However, he was not sure whether maintaining a more thorough documentation would be cost efficient enough.

3.3.5.3 Amount of Up-Front Architecture Design in Agile Environments

When discussing the issue of how much up-front design is appropriate in an agile environment, every architect felt that there needs to be a certain amount of architecture designed up-front before involving the development team. Multiple factors contribute in what is meant by certain amount. In general, all architects mentioned that involving the implementation in a very early phase would produce a number of changes both for the architecture design and consequently also implementation. Both of the architects in Stagnation Lab stated that they would prefer to design the architecture and implement the basic architecture structure themselves before involving the rest of the development team.

In Klarvinduer's case, the architect stated that building the architecture in smaller increments would hinder the developers work as the architecture would be missing components the developer might need. In addition, there is a high chance that the minimalistic architecture would contain bugs. It could be difficult for the developer to trace the issues during implementation as behaviour of the software would be compromised by faults in the architecture structure for example. The architect of Simote explained that one of the main reasons he opted to work on the architecture by himself was that the rest of the development team members assigned to the project were relatively inexperienced at the time. He did state that if the skill level would have been higher, he would have been more flexible involving other team members.

In the case of the Fire and Rescue Department monitoring information system, the architect was more open to involving the rest of the team earlier. He explained that whatever the outcome is from a process, whether it be architecture design or implementation of functionality, it should be communicated as early as possible to get feedback and improve the design. Then again, as he had stated earlier that software architecture design should be regarded as a separate process in agile environments. He mentioned that in Estonia, too little effort is spent on initial planning and design. Agile environments usually consist of teams where an experienced developer is put in charge of the software architecture as opposed to the ivory tower architects in traditional development. This adds the collaborative benefit of faster releases but also brings with it the risk of not enough thinking but only doing.

When asking what the architects think about the XP principle that the best architecture, requirements and design emerge from self-organized teams, none of them believed in it much. All of them agreed that it may work only in a very small project or team but larger systems are too complex to just postpone architectural decisions while proceeding with development. It was believed that this approach would induce issues with integration, stability and communication overhead. In addition, the amount of refactoring would increase along with the risk that at some point, refactoring is not enough and a complete redesign would have to be done in order to meet the required needs of the customer.

4 Discussion

In this section, the results from the literature survey and the case study are discussed taking into account the two research questions. The first research question concerned how software architecture and agile development methods support each other. The case study projects are reviewed and the differences and similarities between the projects, with regards to practices applied are discussed. The second research question concerned how much up-front design is deemed necessary in agile environments and is discussed in light of the case study results and surveyed literature.

4.1 To What Extent do Software Architecture and Agility Support Each Other?

The case study results show that every project used practices from the agile methodology to produce software and many of those practices were incorporated to architecture development. The projects in Stagnation Lab did not follow a set methodology and was at times a mix between agile and traditional plan driven development. Measuring the amount of Scrum or XP practices detected in Klarvinduer and Simote, the projects were leaning towards being slightly less agile than is generally perceived by the company. The project in SMIT was following Scrum with most of the recommended practices present fully or to some extent. Thus, the measurement leaned between slightly more agile and fully agile. However, since the overall project in SMIT was part of a bigger infrastructure in the organisation, a traditional development process was naturally also used for the initial high-level architecture design. This is because project architecture designs had to be approved first by the central technical architecture review committee before implementation was to commence. Rest of the lower level architecture design along with development was following Scrum for the most part.

The architects of Simote and the monitoring system were supporting the fact that agile development and architecture can support each other. The reasoning being that whatever outcome of a process should be communicated with relevant stakeholders as soon as possible, whether it be the customer or the development team. Architecture design is no different. The architect of Klarvinduer had a mixed opinion. While not entirely against the methodology, he felt that some practices related to Scrum including meetings such as daily sprints and sprint planning meetings produce too much overhead in the project lifecycle. The architect explained that he and the developers prefer to tinker with the design and implementation without having to figure out how to divide the tasks into small increments and assign them values. Sometimes this leads to assigning random numbers to define the complexity of a task as the architect or developer has a vague understanding of the actual complexity which might in turn induce issues during the sprint for example. He understands though that planning is necessary and agrees that there should definitely be a qualified project manager setting up the tasks and goals. Taking it further, however, is more important to management but not as valuable for the architects and developers.

Through the interviews collected, there are certain challenges regarding architecture design that the architects brought out. The architects in Simote and the monitoring system both mentioned the issue of managing the complexity of the architecture. Scrum and XP advocate keeping the design as simple as possible and refactoring to ensure that. While simplicity is

important, the architects argue that some effort should be spent on making the design not just simple but to try to isolate the components to hide the complexity within the components. As too much refactoring is wasteful both for architecture and implementation, managing the complexity helps reduce the impact of refactoring in case there is a need. Literature results also indicate this as a problematic factor, where features were developed against an architecture constantly changing, causing developers to “shoot at a moving target” (Isham, 2008). The architect in SMIT mentioned that one challenge he encountered in the project was getting the team to accept change also, sometimes leading to disputes between the architect and developers. The development team was understandably not pleased regarding why certain functionality would have to be scrapped or redeveloped due to changes ordered by the customer. Thus it is believed to be a good investment to spend extra effort on hiding the complexity while keeping the interfaces between the components as simple as possible.

It is interesting to note that while none of the projects employed formal testing practices advocated in the agile practices for various reasons (time to develop test cases, insufficient management backing or architects/developers simply not encountering enough to issues to justify more testing), all of the architects agreed that more effort should be put on testing activities regarding future projects, especially when the project scales in size and complexity. Testing in agile environments, particularly TDD having a positive effect on architectural quality is also mentioned to have scientific validity in the literature surveyed, although coming with the price of increased time developing test cases for example, mentioned likewise by the architects in the cases (Breivold et al., 2010).

Based on the case studies, a fully agile approach to architecture design and development is believed to not be possible whereas most of the practices do not influence the architecture design much if followed as recommended. In fact, they can be helpful in delivering working software faster as opposed to traditional development. Agility and architecture work together best in smaller teams and small to mid-size projects. All the architects mentioned that when scaling up in team size or the project complexity increases, it is notably harder to apply the practices effectively, whereas in the case of very small teams such as in the case of Simote, following all practices is simply deemed not as necessary when the basics are provided, such as a dedicated open workspace to begin with. Team members prefer to collaborate often without a formal process which is believed to be faster and reduce unnecessary formality. Few practices were considered harmful if followed as recommended by the literature. Too much refactoring is believed to lead to less stability and more rework by the whole development for example. The principles still remain largely the same in every project meaning that emphasis is put on customer collaboration, delivering value quickly and simplicity in design. However it is hard to verify what happens when the team size or the project complexity grows as all the projects explored in this case study were relatively small and conducted with team sizes averaging around 3-5 people during most of the development lifecycle.

It is also somewhat difficult to draw definite conclusions based on the case studies as all architects mentioned that they had very few problems throughout the project (or have had very few problems working as a team throughout time). Thus, the impact of some of the challenges brought out by the architects was simply not severe enough to understand if an approach was working or not. For example the fact that none of the projects applied systematized testing such as unit tests propagated in XP was not deemed as big of a challenge because there were very few issues throughout the development. Even then, all architects agreed that more effort should be placed on the practice in general as the presence of well

written unit tests is in some cases greatly missed, such as in large scale refactoring. As architecture refactoring can also have a direct impact on the overall implementation, unit tests can be helpful in uncovering issues early. The practice comes with drawbacks though as double the amount of effort and cost is needed to cover all the functionality or structure. More so, architecture deals largely also with non-functional requirements, such as scalability and behaviour, in which case unit tests are not best approach.

4.2 How Much Up-Front Architecture is Appropriate in an Agile Environment

When discussing the subject of up-front architecture design, all architects interviewed agreed that a certain amount of planning is needed to produce a working and stable architecture. The fully agile approach of minimal/no up-front planning is believed to produce issues in the overall implementation and consequently the amount of refactoring would have to be increased throughout development. In addition, there is a higher risk in reaching a state where refactoring is not sufficient to deliver the requested requirements and the whole system would have to be redesigned. Architecture up-front planning tries to reduce the risk of system redesign and minimise the amount of refactoring throughout development. Up-front planning gives the architect also the opportunity for designing the architecture to be more adaptable. That means it is easier to add new functionality without compromising the whole system or making refactoring affect the overall system less.

When looking deeper, it is difficult to understand what is the appropriate amount of architecture an architect can design that would be more beneficial compared to traditional plan driven architecture or minimal to no planning at all as it is affected by a multitude of variables. The literature tends to give vague suggestions such as one needs to design just enough architecture before commencing with development. However, little explanation is given to what just enough means as it is almost always related to the context. In the literature survey conducted, one paper addressed the issue directly by identifying forces that affect the team's context and five strategies used by teams to determine how much effort they should put into up-front design (Waterman et al., 2015).

4.2.1 Comparing Up-Front Architecture to Agile Architecture

Consider two scenarios where software architecture is designed up-front compared to no up-front architecture design at all. In the first scenario depicted in figure 4 below, a lot of effort is spent on designing the architecture initially, which is then frozen when commencing with implementation corresponding to the designed architecture. In all development, effort is spent on both writing tested production code and doing rework, no matter how good it is planned due to internal and external factors, such as developer experience and changes requested by the customer for example.

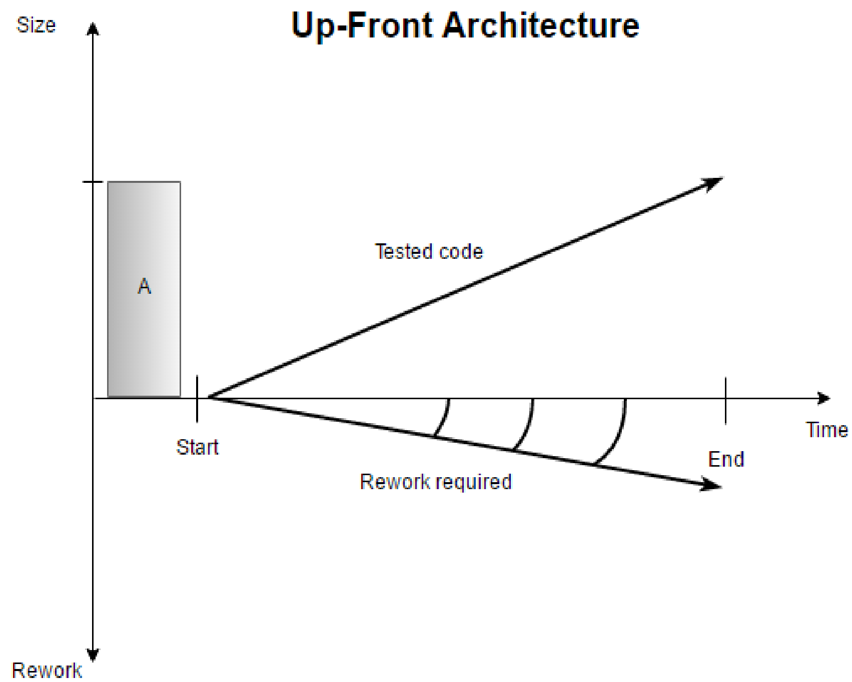

Figure 4. Up-front approach to designing architecture as advocated in traditional development. *A* represents software architecture.

In the second scenario depicted in figure 5, no architecture is planned up-front ideally, which would be an extreme case of agility. Instead, it is designed gradually in every sprint and then frozen for the duration of the sprint.

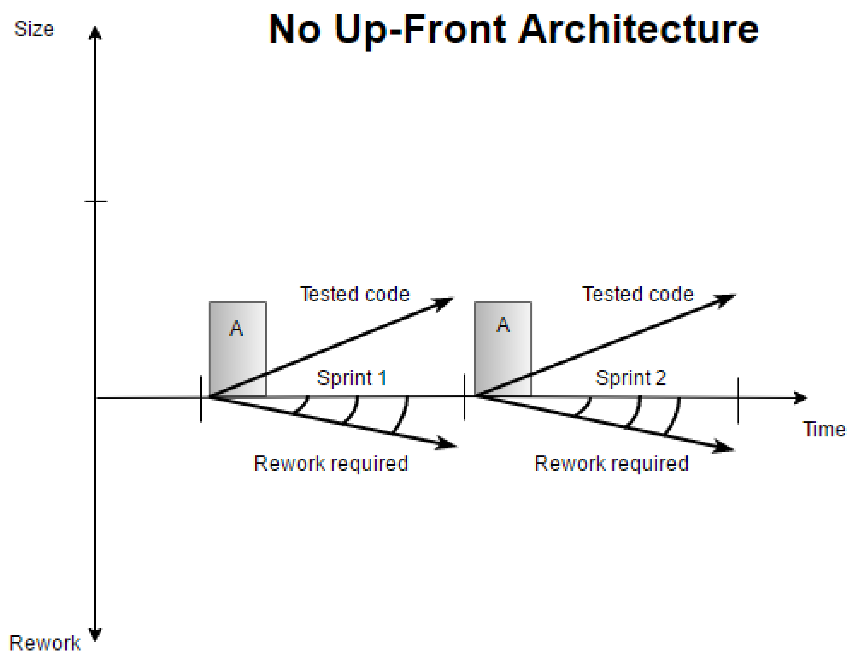

Figure 5. Minimal up-front approach to designing architecture as advocated in agile development. *A* represents software architecture.

Assuming that the amount of the architecture is the same in both figures with the difference that in an agile setting, it would be divided into two (or more) increments, the question remains is it really so that the sum of the amounts of rework in figure 5 is smaller than the amount of rework in figure 4, as is claimed by agile proponents.

Architects would probably say that it is not true because there is other rework induced if not enough architecture is planned ahead. Going further it is possible to define two types of rework. One is due to insufficient planning and the other due to external change requests. A more generic definition would be to classify them as internally induced rework and externaly induced rework. This leads to the question of how should the combination of up-front investment and time horizon be chosen to reach the optimal point. However defining this optimal point is not trivial as it may depend on the type of product, project, skills, etc.

In an ideal scenario without various internal and external parameters affecting the development, it should not matter whether development is done in one big yearly leap, such as following a waterfall method, or it is divided into 12 increments, each one developing the same amount of architecture, the same amount of code and even producing the same amount of rework or waste. However the inclusion of various parameters make the finding of the optimal point in up-front design difficult due to different forces working against it in different directions. Finding the right balance between the external and internal parameters while taking into account the lengths of the development periods might be the clue to understanding what may constitute as the optimal investment in architecture.

4.2.2 Parameters Affecting the Appropriate Amount

The previous section tries to conceptualise the essence of the problem in defining what constitutes as the optimal amount of architecture in a given setting. Further complicating the issue is determining what different variables can affect the choice between more architecture up-front or less.

Internal Parameters

Perhaps the simplest is starting with internal variables over which the architect and developers have control over. One can assume that up-front architecture reduces the amount of thinking a developer has to do and this saves time due to having a well-defined structure to build upon. The question is how much time is saved by spending more time by the architect to think through the design so that there would be less need to understand what has to be done by the developers. In addition, each developer would have a learning curve for the initial architecture. Assuming that each developer is equally experienced, the time required to learn about the architecture would then be multiplied by the number of developers. If the architecture stays the same, then in sprints, the need to recall about the architecture decreases until all developers understand the system and no time is spent on learning or recalling the architecture.

In the minimal up-front scenario, the developers would have to spend some time in every sprint to study the architecture to build upon. Now assuming that an architectural increment consists of the same number of components with the same complexity every sprint, then developers would in turn have to spend the same amount of effort on familiarizing with the new increment plus possible added time recalling what was done during the previous sprint.

Another variable to consider would be the experience between the developers. The time spent on learning the architecture in both up-front and no design at all would vary as every developer would have a different level of skill while in the previous example, it was assumed that every developer would be equally experienced which is not the case in reality. In addition, the skill of the architect also plays a very big role in affecting the outcome, as it directly contributes to the quality of the architecture designed. As the quality varies, so does the implementation in the sense that more experienced developers can discover the issues faster and perhaps work their way around whereas novice developers would probably be interrupted at one point. The architect of Simote in the case study had also pointed out that one reason why he had designed more up-front by himself was because of the experience of the rest of the team. It should also be noted that the quality of the architecture is not only dependent on the skill of the architect but can also be directly linked back to the stability of requirements affected by external parameters. In some instances requirements are fairly stable, yet complete understanding of the requirements usually comes later in development and cannot be developed fully up-front. On the other hand, when requirements are incomplete or not sufficiently detailed due to the customer not initially knowing what they want, the risk for the architecture to not support actual needed requirements is higher.

Perhaps the biggest contributor to deciding on more up-front architecture or less is the architecture complexity, whereas it should be differentiated from size. A developed product can be equal size in terms of lines of code for example but the complexity within vary a great detail. Authors of the grounded theory of agile architecture (Waterman et al., 2015) highlight that complexity can be caused by three things: challenging architecturally significant requirements (ASRs), having to integrate with other systems and including legacy systems. It can be assumed that scaling the complexity influences how much planning should be done as was also agreed with the architects in the case study. Even then, it must be clear to what extent this risk should be addressed as too much planning would be wasteful nevertheless when external parameters such as change requests come to play.

The projects explored in the case study were all relatively small but even there, some challenges were related to either internal or external communication. With larger teams the communication overhead increases even more. It can be expected that with no up-front planning, team members would have to consult with each other regularly to avoid implementing increments that do not integrate well with the rest of the product. One architect had brought up the example that when two developers commenced implementation with no initial planning, the end result would be two different architectures. Up-front planning would help reduce the amount of internal communications which can be associated to waste if not managed properly. Especially if there is no documentation whatsoever also, which could lead to misunderstandings due to human errors (problems with remembering what was discussed in detail). In the minimal up-front planning example, the sheer amount of ad-hoc communication could induce waste in the form of too many interruptions and the product may suffer due to misinterpritation in what was agreed. More up-front would lower the risks but too much prove problematic as requirements could have changed in between.

One more important parameter to consider in determining the optimal amount of up-front architecture design is the (planned) duration of the iteration. The architecture stability is dependent on the time horizon to which the planning is applied and during which the architecture is supposed to be frozen for implementation. It can be assumed that a shorter duration of the sprint during which a certain amount of architecture is designed and then frozen for the rest of the sprint is less prone to inducing the need for architecture refinement.

However as brought out multiple times by the architects in the case study also, this approach leads to issues with architecture integrity in the longer term through refactoring and is also supported by previous research (Breivold et al., 2010). Sprints during which the architecture is designed and frozen for a longer period has a higher risk of discovering issues in implementation due to failures in the architecture. Determining the optimal timeframe for sprints and subsequently the amount of time spent on architecture and the amount of time it is frozen for implementation would require knowledge of some probabilities. For instance, the occurrence of the need to refine the architecture in relation to the length of the sprint, the amount of architecture designed and the amount of time the architecture is frozen for implementation percentagewise.

External parameters

Whereas internal variables can be affected a lot by the people inside the organisations either by conducting more training sessions to ensure the developers are more skillful or choosing the right approach for the current context, external variables are harder to counter as they happen anyway and more spontaneously.

Typical example of an external variable leading directly to development waste would be change requests from the customer, which can always happen no matter how much is planned. The question here is whether the risk is higher when planning too much. Knowing that changes are introduced with a certain regularity, the team can continuously learn to minimize the extra planning which would become obsolete, hopefully arriving at a state where the team has the right amount of architecture to commence with development but at the same time not be affected by the unexpected changes as much.

The unpredictability of the environment can also account as a factor. If there is only one fixed customer that the team collaborates with for a long time and the product is clear, then there is probably a smaller risk for unexpected changes. On the other side for products with high familiarity, there probably is no need for that much up-front planning because everyone knows what is due to happen in development.

An example from a less direct external variable would be changes in the market trends, which could suddenly render the product worthless and is hard to anticipate. It could also be that change in a policial ground could affect the design such as introductions of new legislations. For example there might come a requirement passed by a law that state all new products must be conformed to accessibility requirements.

Predicaments in extracting the parameters

Looking back at the different scenarios described, it would require a substantial amount of effort to define how much architecture is appropriate in agile environments. Even in the smallest settings involving just one architect and a developer, it would require the understanding of various parameters to measure the level of just enough design. These parameters can range from the architect/developer experience to the trends in developing markets. To gather data about these parameters, interviews would have to be conducted not just with software architects but also developers using the architecture. This sets a huge limitation on this thesis as the case study involved only the opinion of architects in distinct projects. Even involving a larger selection of cases and qualified people to interview would probably not be enough to produce correct measurements as there may be contradicting statements between architects and developers. Statements given in a qualitative study tend

to be subjective whereas precise measurements would require hard facts. Thus the detailed measurement of how much up-front architecture is appropriate is out of the scope of this thesis, but it helps conceptualise the issue at hand.

However, there is little information in literature regarding this issue of finding out the optimal point other than the expert accounts of similar qualitative studies, which suggests that nobody has really conceptualised this issue properly and thus nobody knows exactly what is best to do. The research conducted so far suggests strategies to apply in different scenarious based on the context, for example (Waterman et al. 2015). Due to not having a clear answer to what is considered appropriate amount from the literature nor the case study results, the discussion tries to conceptualise the issue more and could be used for future reference. In any case, when starting to gather facts, a proper reference framework should be defined up-front.

4.3 Limitations

In this section the limitation of the thesis are discussed, both for the literature survey and the conducted case study. This includes threats to validity and the actions undertaken to limit the effects of such threats.

4.3.1 Literature Survey

The survey in this thesis was primarily carried out individually. To prevent the risks of a single-reviewer, central parts of the survey were reviewed by the supervisors to ensure that its activities were executed properly and material was consistent the topic at hand. Validity issues can be encountered in the general exectution and many stages, such as selecting the databases and lacking search queries Since the survey wasn't conducted as a systematic literature review and is shorter in essence, it may be missing papers that could provide more hindsight into the challenges discussed but were not as clearly identified through the survey.

4.3.2 Case Study

The case study is relied mostly on interviews as this is the one of the few types of evidence available. To better understand the case, other evidence sources would be recommended, such as observations of an ongoing project because the case study in this thesis concentrates on a project that was already finished. Interviews as an evidence source have few common risks (Yin, R., 1994) which are outlined below with precautions to limit risk factors.

- **Bias due to poorly constructed questions**
  To ensure that the interview guide is as precise as possible, thorough reviews are made regarding the questions by the supervisors of this thesis. However there might be misunderstandings in terms of what is being asked from the interviewee. The interviewer and the interviewee may have different understandings of certain technical terms used. In addition, as new knowledge was acquired after the analysing the conducted interviews, additional questions could have been formed for more thorough data extraction.

- **Response bias**
  Some interview questions might be giving a guide for answers. To avoid this, the questions asked are reviewed by supervisors and as open as possible while at the same time having the necessary level of detail for the interviewee to understand what is asked.

- **Inaccuracies due to poor recall**
  This doesn't necessarily affect the case study as the interviews are recorded digitally, thus eliminating any issue with recall.

- **Reflexivity**
  Interviewee gives what the interviewer wants to hear. To avoid this, insight about the questions asked should be kept to the minimum and only provide necessary context for the interviewee to answer and not suggest answers in the questions.

The fact that the thesis deals with only three cases can also be a threat to validity as the findings may not correlate to other similar projects. Generally multiple-case studies are preferred as these offer robust analytical conclusions increasing the external validity. (Yin, R., 1994).

4.3.3 General Validity Threats

It can be somewhat unclear what agile practices directly contribute to the architecture design and development. Certain practices such as *Simplicity in design* and *No functionality is added early* have a considerable impact whereas practices like *Use CRC cards for design sessions* might be too specific or even completely irrelevant. The absence of a *de facto* standard in both Scrum and XP methodologies makes it hard to know which reference is the most reliable or “correct”. Thus the agile evaluation of the projects in the case study can also be biased to some extent due to no following of the not particularly relevant practices for example.

Another limitation is that the interviews produced only expert opinions on the topic. The answers to the research questions are very subjective and could be biased depending on the person interviewed. Making conclusions based on expert opinions related to how much architecture is appropriate can be too generic or vague and requires more facts from specific attributes which contribute to the overall design.

Finally, a very big limitation was only including the architects of the organisations when conducting the case study as it is later implied in the discussion, numerous parameters affect the outcome of searching for the optimal amount of architecture to design at various points in the development lifecycle. Among the internal parameters are the overall developer experience and skill which is a contributing factor to consider when building upon the existing architecture. For a more comprehensive understanding of the issue, both accounts from architects and developers should have been explored. Although architects in agile environments are also often developers themselves, it still is biased to some degree as the mindset of architects leans more towards architecture and vice versa.

5 Conclusions and Future Work

The findings of this study suggest that agile development can be combined with architecture but it does not come without drawbacks. The overall principles followed in agile methodologies such as customer collaboration, simplicity in design and delivering solutions in short increments are welcomed for architecture. However, following a fully agile approach with minimal planning is believed to lead to unstable architecture, increased amount of refactoring and a higher risk of having to redesign the system in time. Thus traditional plan driven practices are more preferable in the early stages of development while increasing agility in the later stages.

It is difficult to determine the right amount of architecture appropriate for involving the whole team in development. The architects in the cases had differing opinions and none could describe apart from their gut feeling what is right as it depends a lot on context. Looking deeper, measuring the appropriate amount of architecture is difficult as it involves a number of parameters ranging from the complexity of the architecture designed and architect experience to the learning curve needed from the developers building upon the architecture. Extracting the necessary parameters for measurement requires factual data and conducting a greater number of studies. Using a qualitative study to collect data could be biased as the architects and developers could have contradicting statements and subjective opinions. A contribution of this thesis is trying to conceptualise the issue of finding the optimal amount of architecture in a given setting taking into account these various parameters.

The case study results were also presented to the architects interviewed. The benefit was understanding better how architecture is perceived by different people and the various challenges encountered in the different projects. In Stagnation Lab, one outcome is to incorporate testing practices more effectively in future projects as the current approach is not feasible in the long term.

As the current study involved only three projects, further studies should be conducted in various settings to draw more conclusions on the applicability of software architecture development and agility. A systematic literature review should be conducted to construct more detailed questions for better data extraction and controlled experiments could be used to assess the effect of combining agile practices with architecture design. Perhaps the most interesting is measuring the amount of up-front design vital in agile environments. Various internal and external parameters should be defined and measured in distinct projects to be able to calculate approximately what is deemed just the right amount of architecture. These parameters could then be compared taking into account their probabilities of occurring during the design and when they carry more risk than. This could be interesting to provide a more clear understanding of what the most important factors are to the initial architecture design, as opposed to the vague understanding of just enough architecture in literature.

6 References

Abrahamsson, P., Babar, M. A., & Kruchten, P. (2010). “Agility and Architecture: Can They Coexist.” *IEEE Software*.

Ambler, S (2012). “Agile Architecture: Strategies for Scaling Agile Development”: http://agilemodeling.com/essays/agileArchitecture.htm (07.05.2016)

Babar, Muhammad Ali. “An exploratory study of architectural practices and challenges in using agile software development approaches.” *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on* 14 Sep. 2009: 81-90.

Beck, Kent et al. (2001) “Manifesto for agile software development.” http://www.agilemanifesto.org/ (04.05.2016)

Boehm, B., & Turner, R. (2004). “Balancing Agility and Discipline.” *Addison-Wesley*.

Breivold, H. P., Sundmark, D., Wallin, P., & Larson, S., “What does research say about agile and architecture?” Fifth International Conference on Software Engineering Advances, 2010.

Cadle, James, and Donald Yeates. *Project management for information systems*. 5th Edition. Pearson education, 2008.

Clements, Paul et al. *Documenting software architectures: views and beyond*. Pearson Education, 2002.

Cohn, Mike. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

Falessi, D., Cantone, G., & Sarcia, S. A. (2010). “Peaceful Coexistence: Agile Developer Perspectives on Software Architecture.” *IEEE Software*.

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., & America, P. (2007). *A general model of software architecture design derived from five industrial approaches.*

IEEE Software, April 2010: www.computer.org/web/computingnow/archive/april2010 (01.01.2016)

Isham, Mark. “Agile architecture IS possible, you first have to believe!” *Agile, 2008. AGILE'08. Conference* 4 Aug. 2008: 484-489.

Kruchten, Philippe B. “The 4+ 1 view model of architecture.” *Software, IEEE* 12.6 (1995): 42-50.

Len, Bass, Clements Paul, and Kazman Rick. “Software architecture in practice.” *Boston, Massachusetts Addison* (2003).

Murphy, B., Bird, C., Zimmermann, T., Williams, L., Nagappan, N., & Begel, A. (2013). “Have agile techniques been the silver bullet for software development at microsoft.” *International Symposium on Empirical Software Engineering and Measurement.*

Nuseibeh, B. (2001). “Weaving together requirements and architectures.”

Paasivaara, Maria, Sandra Durasiewicz, and Casper Lassenius. “Using scrum in distributed agile development: A multiple case study.” *Global Software Engineering, 2009. ICGSE 2009. Fourth IEEE International Conference on* 13 Jul. 2009: 195-204.

Sletholt, Magnus Thorstein et al. “A literature review of agile practices and their effects in scientific software development.” *Proceedings of the 4th international workshop on software engineering for computational science and engineering* 28 May. 2011: 1-9.

Software Engineering Institute (CMU) - Published Software Architecture Definitions: http://www.sei.cmu.edu/architecture/start/glossary/published.cfm (05.05.2016)

Theocharis, G., Kuhrmann, M., Münch, J., & Diebold, P. (2015, December). “Is Water-Scrum-Fall Reality? On the Use of Agile and Traditional Development Practices.” *LECTURE NOTES IN COMPUTER SCIENCE*.
Tripp, J., & Armstrong, D. (2014). “Exploring the relationship between organizational adoption motives and the tailoring of Agile methods.” *Hawaii International Conference on System Sciences*, (pp. 4799–4806).
Waterman, Michael, James Noble, and George Allan. *How Much Up-Front? A Grounded Theory of Agile Architecture*. 2015.
Weitzel, B., Rost, D., Scheffe, M.“Sustaining Agility through Architecture” Software Architecture (WICSA), 2014 IEEE/IFIP Conference
Wells, D. 1999. The Rules of Extreme Programming: http://www.extremeprogramming.org/rules.html (04.05.2016)
Yin, R. (1994). Case study research: Design and methods. Beverly Hills. *1*.

Appendix

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Andres Randmaa**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Challenges to Architecture Decision-Making in Agile Development Environments,

(title of thesis)

supervised by Dietmar Pfahl,

(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **19.05.2016**