Jonna Kalermo and Jenni Rissanen

Agile software development in theory and practice

Master's thesis in
Information systems science
6.8.2002

University of Jyväskylä
Department of Computer Science and Information Systems
Jyväskylä

TIIVISTELMÄ

Kalermo, Jonna Maria
Rissanen, Jenni Karoliina
Agile software development in theory and practice / Jonna Kalermo ja Jenni Rissanen
Jyväskylä, Jyväskylän yliopisto, 2002.
188 s.
Pro gradu -tutkielma

Uusia ohjelmistokehitykseen liittyviä ajatuksia esitettiin vuosituhannen vaihteessa Agile Manifeston muodossa vastatoimenpiteenä perinteisille, ankarille ohjelmistokehitysmenetelmille ja prosessimalleille. Agile Manifesto koostuu neljästä arvosta ja kahdestatoista periaatteesta, joista tämän tutkimuksen tekijät muodostivat käsitteellisen viitekehyksen. Sen avulla voidaan arvioida ketterän (*agile*) ohjelmistokehityksen eri osaalueita ja niiden muodostamaa kokonaisuutta. Ketterä ohjelmistokehitys pyrkii asiakkaan liiketoimintaa tukevan ohjelmiston kehittämiseen nopeasti, kevyesti ja tehokkaasti. Ketterä ohjelmistokehitys ei kuitenkaan tarkoita kaaosta, vaan kultaista keskitietä kehittämistä tiukasti ohjaavien menetelmien ja täydellisen vapauden välissä.

Tutkimuksen tarkoituksena oli tarkastella Agile Manifestoa ja sen käytettävyyttä kirjallisuuskatsauksen ja empiirisen tapaustutkimuksen avulla. Agile Manifesto antaa ideologisen taustan erilaisille ketterille menetelmille, kuten Extreme Programming –menetelmälle (XP). Nämä menetelmät sopivat parhaiten pienille organisaatioille, jotka kehittävät innovatiivisia ohjelmistotuotteita. Tarkastelumme perusteella voimme todeta, että kirjallisuus tukee Agile Manifeston esittämiä ajatuksia, joskaan ne eivät ole uusia. Tutkimme tapaustutkimuksen avulla organisaatioiden sisäistä ketterää ohjelmistokehitystä corporate venturing –yhteydessä. Tulokset tukivat Agile Manifestoa ja vahvistivat väitteen, että ketteryys perustuu hiljaiseen tietoon, ammattitaitoisiin ja motivoituneisiin työntekijöihin sekä säännölliseen viestintään. Tämä asettaa monenlaisia vaatimuksia ketterää ohjelmistokehitystä tekevän yrityksen johdolle ja työntekijöille: heidän täytyy olla persoonaltaan avoimia, sosiaalisia, vastuullisia ja ammattitaitoisia.

AVAINSANAT: ketterä ohjelmistokehitys, Agile Manifesto, Extreme Programming, yritysten sisäinen ketterä ohjelmistokehitys, viitekehys ketterään ohjelmistokehitykseen

ABSTRACT

Kalermo, Jonna Maria
Rissanen, Jenni Karoliina
Agile software development in theory and practice / Jonna Kalermo and Jenni Rissanen
Jyväskylä, University of Jyväskylä, 2002.
188 pp.
Master's thesis

In turn of the millennium, new software development ideas were presented in the form of Agile Manifesto as a counteraction to the traditional, rigorous development methods and process models. Agile Manifesto consists of four values and twelve principles of which the authors of this thesis formed a conceptual framework. It assists in analysing different aspects of agile software development one by one but also as a whole. Agile software development aims at fast, light and efficient development that supports customer's business without being chaotic or following any rigorous method.

The aim of this study was to analyse Agile Manifesto and its applicability. This was done by conducting a literature review and an empirical case study. Agile Manifesto gives an ideological background for agile methods, such as Extreme Programming (XP). Such methods are most feasible for small organisations developing innovative software products. We analysed Agile Manifesto and found out that literature supported the ideas of it, although no revolutionary ideas were discovered. We used an empirical case study as an example of an agile in-house software development method in corporate venturing context. The case study supported principles and values of Agile Manifesto and confirmed the assumption that agility is heavily based on tacit knowledge, skilled and motivated employees and frequent communication. This sets several requirements for management and employees working in an agile mode: they have to be communicative, social, responsible and skilled.

KEYWORDS: agile software development, Agile Manifesto, Extreme Programming, agile in-house software development, agile framework

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Software plays a significant role in lives of individuals and in fierce competition between companies. It can be used as an application in a personal computer or as an embedded part of an industrial robot and it enables services and automates procedures. Software has been developed since the 1950's, and different methods, paradigms and process models have been invented to handle the complex efforts of development. Some of the development methods have become heavily documentation oriented or expect the developers to rigorously follow certain processes. Those can be called heavy or traditional methods, for example structured analysis and design. Additionally we include Capability Maturity Model (CMM) and Rational Unified Process (RUP) to heavy methods, although CMM and RUP can also be classified as process models.

In the turn of the millennium, new development method ideas were presented in the form of Agile Manifesto as a counteraction to rigorous, plan-driven software development (Boehm 2002, 64). Agile Manifesto gives an ideological background for agile software development. The word *agile* can be defined as "1) marked by ready ability to move with quick easy grace or 2) having a quick resourceful and adaptable character" (Merriam-Webster 2002). According to Cockburn (2001), "Core to agile software development is the use of light-but-sufficient rules of project behaviour and the use of human and communication-oriented rules." (Cockburn 2001, xxii) Fowler (2001) claims that the most significant differences between agile and traditional, heavy methods can be found in the emphasis on methods. Agile methods are less document-oriented than traditional methods. Instead, agile methods are more code-oriented and they emphasise working code over documentation. He also claims that traditional methods would resist change but agile methods would be more open to meet changes. Besides, agile methods are more people-oriented that process-oriented (Fowler 2001). According to Coldewey, Eckstein, McBreen and Schwanninger (2000), "most lightweight [i.e. agile] processes substitute interpersonal communication and teamwork for the extensive documentation and formal signoffs that are required by heavyweight processes." (Coldewey et al. 2000, 131) It is important to note, however, that agile does not refer to chaos.

Thus, agile software development aims to develop and implement software quickly in close cooperation with the customer in an adaptive way, so that it is possible to react to changes set by the changing business environment and at the same time maintain effectiveness and efficiency. Ways to accomplish this are for instance putting emphasis on tacit knowledge and sharing it via frequent face-to-face communication and by concentrating on producing working software instead of documentation. Agile Manifesto and agile methods claim to provide guidelines for such development efforts.

1.1 Research objectives and contributions

Because agile software development is such a new topic, it has not been thoroughly and objectively examined in the academic literature. Some analyses can be found but most of the articles and books about agile software development are written by the inventors of Agile Manifesto that at the same time are also promoting their own, commercial agile methods and consulting services. As a result, their writings sound like promotional material to some extent rather than objective analysis of the strengths and weaknesses of agile software development. The applicability and limitations for use of agile methods have not been comprehensively examined. Most agile methods were originally designed for small teams and projects. Some thoughts of their applicability for other contexts, for instance for a large group of people, have been presented but not thoroughly tested and proven. Additionally, agile methods have not been discussed in the light of different business modes. In this study we aim to analyse Agile Manifesto in theory and practice and provide a more objective and versatile view of the topic.

In our research, we use Agile Manifesto as an initial framework for explaining and evaluating different aspects of agile software development. We aim to give a more objective and comparative analysis on agile software development and develop a conceptual framework for agile software development. We also will study the applicability of agile methods. The following research questions can be derived from our research intentions:

- How can a conceptual agile framework be developed?

- What kind of enabling and limiting factors can be found for the use of agile methods?
- How do agile methods support the principles of Agile Manifesto
  - when using Extreme Programming?
  - when using in-house software development methods?

The contribution of this research will be assisting organisations in evaluating whether agile software development methods would suit their current situation and whether the agile framework could assist in making their software development faster, more agile and efficient. In previous research, agile software development methods have not been aimed to any certain business mode but in this thesis we study an organisation operating on COTS-business and using an agile in-house development method.

Other contributions of this study are establishing the concept of *skilled improvisation* and developing a conceptual agile framework and modifying it based on literature and findings of the case study. We found that the results of the literature review and empirical study support the principles of Agile Manifesto, which emphasise tacit knowledge and communication. The empirical study confirmed that skilled improvisation mode is agile and has several aspects similar with Agile Manifesto principles.

1.2 Research design

To answer our research questions, we will first take an interpretive approach to conduct a literature review. According to Tamminen (1992), the interpretive research process can be expressed as a hermeneutical circle. FIGURE 1 illustrates the hermeneutical circle. In the beginning, the researcher has a pre-understanding about the phenomena. After that, the researcher tries to gain more knowledge to expand his or her potential for interpretations. This is called the absorption phase. After the first round, the researcher finds out what are the most relevant research related factors and tries to build some kind of interpretations based on the findings. Based on these interpretations the researcher

gets a deeper understanding about the phenomena. He or she will also be able to sharpen some issues and theories and sum up the results as a report. (Tamminen 1992, 95-96)

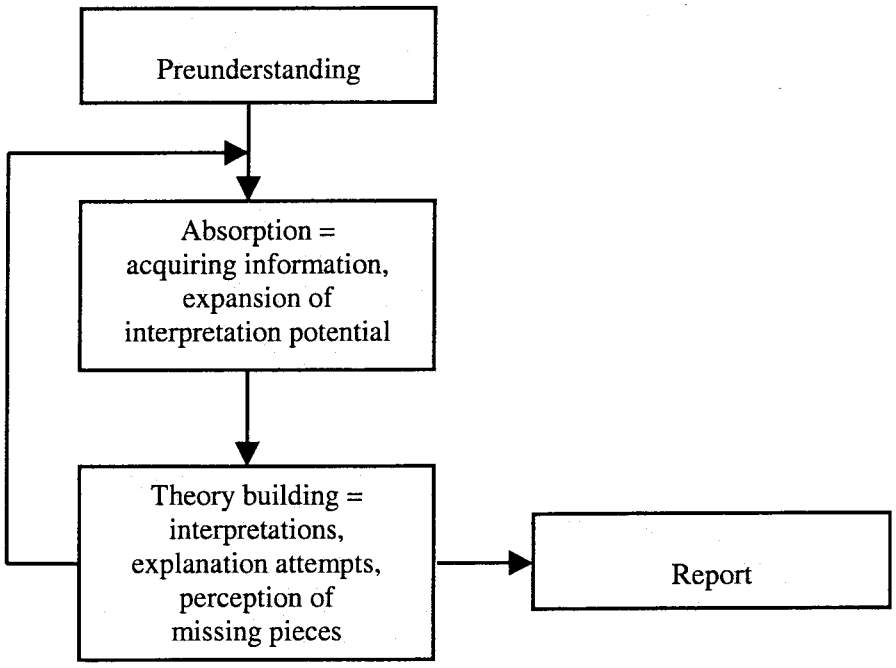

FIGURE 1 Hermeneutical circle (adapted from Tamminen 1992, 95)

This hermeneutical approach will be used to explore the evolution of software development to explain the background of the agile framework. The same approach will also be used to study Agile Manifesto and Extreme Programming. When studying agile in-house software development we will use empirical case study to thoroughly describe and analyse a corporate venture and its development method. According to Yin (1989),

> *A case study is an empirical inquiry that investigates a contemporary phenomenon within its real-life context; when the boundaries between phenomenon and context are not clearly evident; and multiple sources of evidence are used.* (Yin 1989, 23)

We aim to gain a thorough understanding about the case organisation's processes and their work. The purpose of our case study is to find answers to a question "how do agile methods support the principles of Agile Manifesto when using in-house software development methods?" Yin (1989, 18) suggests that case study is an appropriate research strategy when the research questions are of type "how" and "why" and thus exploratory by nature. This further supports our choice to conduct a case study.

1.3 Structure of this thesis

The second chapter introduces and defines the key concepts used in this paper. In the third chapter, we will clarify how information systems design methods and paradigms have evolved and why the need for using agile methods has risen. Fourth chapter will explore the background of Agile Manifesto and examine the values and principles of it in depth. In addition, supportive and contradicting arguments will be presented for each principle. This is done by reviewing literature. In this chapter, we also present a conceptual agile framework that assists in positioning Agile Manifesto principles in light of different dimensions such as technical versus social. To further explore the practical applicability of the agile approach, we study two agile methods. The first method, Extreme Programming, which is investigated based on contemporary literature and discussions, will be discussed in the fifth chapter. The second studied method is agile in-house development method. To study it, we conducted an empirical case study. The sixth chapter explains the empirical research setting for the case study in which a corporate venture was studied. The seventh chapter includes the case description and analysis of the case. In the eighth chapter, we develop a modified agile framework based on our findings from literature and our case study. Finally, in chapter nine, we will conclude and discuss our results of this study. In the final chapter, we will also describe the limitations of our study and propose topics for further research. FIGURE 2 illustrates the structure of our study.

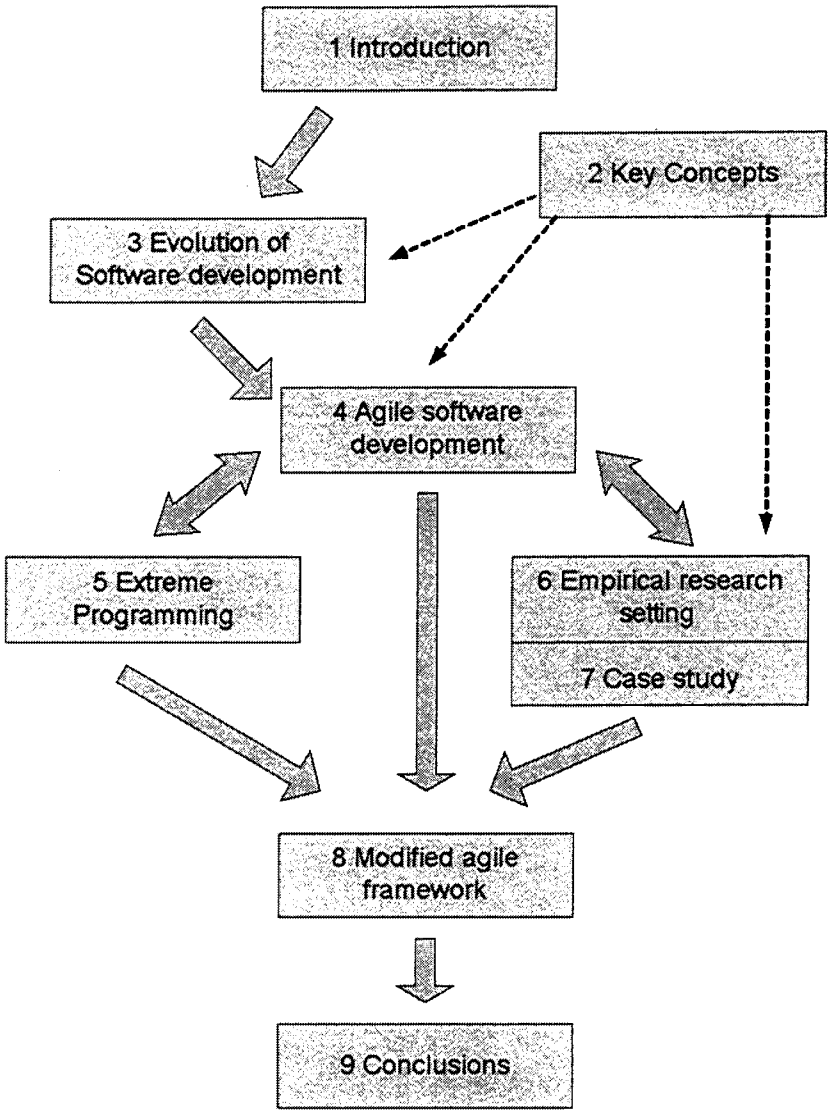

FIGURE 2 Structure of this thesis

2 KEY CONCEPTS

Here we present and define the key concepts to be used in this research. The terms *corporate venturing* and *skilled improvisation* will be used in our description and analysis of the case study, all the other concepts will be used throughout this study.

2.1 Information system, information systems development

In some contexts *information system* is regarded as "a mechanism used for storing and retrieving an organised body of knowledge" (IEEE std 610 1991, 106). In this paper, however, we will use a broader view. In the context of our research, information system encompasses people using information and software with hardware technology to serve organisation's goals in general, not only knowledge or information handling. More accurately said, we use the definition of Hirschheim et al. (1995, 11) stating that an information system is a "collection of people, processes, data, models, technology and partly formalised language, that together form a coherent structure which serves some organisational purpose or function". With *information systems development* we refer to a "change process taken with respect to object systems in a set of environments by a development group using tools and an organised collection of techniques collectively referred to as a method to achieve or maintain some objectives" (Welke 1981 in Lyytinen 1987b, 6).

2.2 Software engineering

Software is one element of the above-defined broader concept, information system. *Software engineering,* according to IEEE standard (1991), is:

> *(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).* (IEEE std 610 1991, 186)

Application of engineering principles has been seen as one way to control the complexity and development of objects and processes of software development.

2.3 Methodology, method

The use of concepts methodology and method is inconsistent in the research of information systems. According to Blokdijk and Blokdijk (1987, 5),

> methodology *means the science of method, a treatise or dissertation on method. A methodology can be materialised in one or even more methods. A* method *is a systematic procedure, technique or mode of inquiry, employed by or proper to a particular discipline, or a body of skills or techniques. It consists of intertwined or integrated* techniques, *which are procedures or a body of technical actions.*

To be more accurate, in this thesis we use the term method to refer to "the set of known methods adapted for the development of computer-based information systems" (Baskerville et al 1992, 242).

According to Lyytinen (1987b),

> *An information systems development method*[1] *is an organised collection of concepts, beliefs, values and normative principles supported by material resources. The purpose of an information systems development method is to help a development group successfully change object systems that is to perceive, generate, assess, control, and to carry out change actions in them.* (Lyytinen 1987b, 9)

Pressman (1994) presents a definition for a method in the software engineering context. According to Pressman, software engineering methods provide the technical guidelines for developing software:

> *Methods encompass a broad array of tasks that include:*
>
> - *project planning and estimation,*
> - *system and software requirements analysis,*
> - *design of data structure,*
> - *program architecture and algorithm procedure,*

---

[1] Lyytinen used *methodology* in the meaning of method in the original source.

- *coding,*
- *testing,*
- *and maintenance.* (Pressman 1994, 23)

Software engineering methods often introduce a new set of criteria for software quality and a special language-oriented or graphical *notation* (Pressman 1994, 23). A notation is a system of characters, symbols or abbreviated expressions used to express technical facts or quantities and usually a technique uses a notation (Blokdijk & Blokdijk 1987, 5). For example structured analysis and design, object-oriented analysis and design and prototyping are methods. Techniques of structured analysis and design are for instance data flow diagrams and entity-relationship diagrams that can be described by using a notation.

Hence, by a method we mean a collection of techniques and a set of rules to describe how and in which order they should be executed in order to achieve certain objectives. By techniques we do not only mean technical procedures, such as programming practises but also procedures relating to communication and collaboration between people. Methodology and method are often considered synonyms but as stated before methodology refers to science or study of methods. The literature we have studied and citations we use vary in the use of terms methodology and method but we will interpret and cite the material according to our definition above.

To further clarify our use of above-mentioned terms, it can be stated that in most cases within the scope of this study an organisation has an information system that includes development methods and knowledge of them. Those methods and knowledge are used to develop software for customers, or for own use. This research focuses mainly on development methods and software development. To keep our text more fluent, we often use the term *software development* alone. For example in "X, Y and Z were prerequisites for improving software development" software development refers to development methods but also techniques including communication and collaboration as defined above.

2.4 Paradigm

The term *(software engineering) paradigm* is often used to refer to a set of steps that consist of methods, tools and procedures (Pressman 1994, 24). A paradigm is also used in order to perceive the different phases in development. Phases are decomposed into tasks and activities and tools such as templates, forms and checklists are used to complete the tasks and activities (Pickering 2001, 4). One example of a paradigm is the classic life cycle paradigm that is also called waterfall model. FIGURE 3 illustrates the life cycle paradigm. The sequential steps of it are system engineering, analysis, design, code, testing and maintenance (Pressman 1994). The output of one step will be the input for the next step, for example analysis documentation will be the basis for design phase.

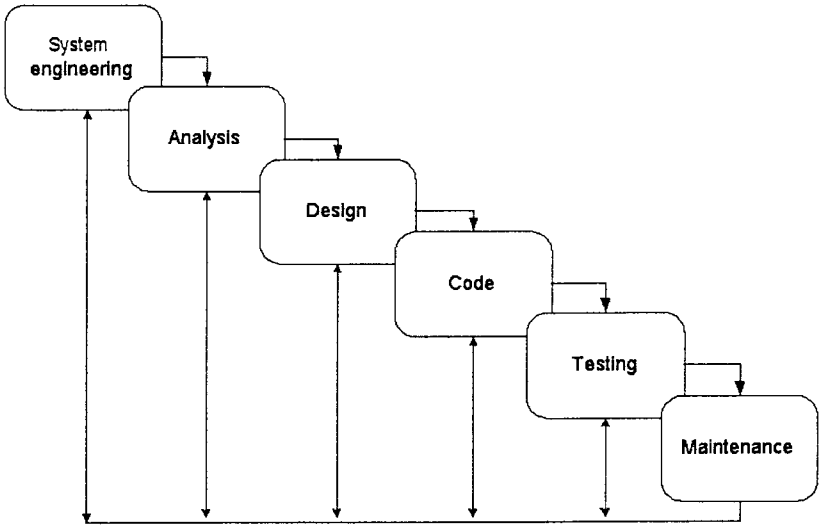

FIGURE 3 Waterfall model (Pressman 1994, 24)

2.5 Agile software development

The term *agile* can be defined as "1) marked by ready ability to move with quick easy grace or 2) having a quick resourceful and adaptable character" (Merriam-Webster 2002). The term lightweight is also used as a synonym to agile, referring to a method light in weight for instance (Agile Alliance 2002) but as the word can have other connotations as well, the word agile has been established. Agility has been researched from several different viewpoints, for instance organisational agility, enterprise agility (e.g. Wong & Whitman 1999), agile manufacturing and agile software development. In

this study, we will focus on agile software development, which not only guides software development but also includes some organisational related issues.

We find Kruchten's (2001) description for agility in software development adequate for this thesis:

> *Agility, for a software development organisation, is the ability to adopt and react expeditiously and appropriately to changes in its environment and to demands imposed by this environment. An agile process is one that readily embraces and supports this degree of adaptability. So, it is not simply about the size of the process or the speed of delivery; it is mainly about flexibility.* (Kruchten 2001, 27)

While Kruchten stresses adaptability and flexibility, Cockburn (2001) also points out the importance of following rules. According to him, "Core to agile software development is the use of light but sufficient rules of project behaviour and the use of human and communication oriented rules." (Cockburn 2001, xxii)

2.6 In-house development methods

Tolvanen (1998, 51) states that despite of the great number of development methods available, the methods do not meet the needs of organisations. Because of that, organisations develop their own methods, or modify existing methods to better suit their objectives. He (Tolvanen 1998, 14) defines *local method development* as "organisations' attempts to develop their own method or methods". We will use the concept *in-house development method* in the same meaning. In-house development is often carried out by combining and modifying existing methods. In-house development methods can further be adapted to organisational, project, or individual needs. (Tolvanen 1998, 14) Thus, in-house methods can be very technology, domain or team oriented and transferring them from one organisation to another, or even within an organisation from one project to another, can be impossible and useless.

According to Tolvanen (1998, 49), there does not seem to be any universal agreement about whether methods are useful in information system development at all. Numerous methods exist but companies do not use methods as such, or they have created their own variants. Although the significance of methods in improving the productivity and

quality has been acknowledged, systematic use of them in companies is low. Industry and academics have spent enormous efforts to create situation-independent methods but at the same time companies are developing their own situation-bound methods. (Tolvanen 1998, 14) Especially when innovative technology is used and rapid development is needed due to time-to-market pressure, a company might choose to develop its own, agile method and way of working to meet those demands. Also Cockburn (2000a, 27) supports this idea by stating "Each organisation must develop its own way of working, and each project must get its own tailored method, dealing with culture, location, criticality, and technology."

Tolvanen (1998, 15) makes an observation that little empirical knowledge on in-house method development and in-house methods use exists: how and why are in-house and adapted methods developed, and how do they work in practice compared with each other. A thorough examination of an in-house development method that Microsoft uses can be found in Cusumano and Selby (1998). That method can also be characterised as agile.

2.7 Skilled improvisation

When developing software or information systems without a defined method or process and without detailed plans, that kind of mode is known as *ad hoc*. According to (Merriam-Webster 2002), ad hoc as an adjective means:

> *1 a : concerned with a particular end or purpose* <an ad hoc investigating committee> *b : formed or used for specific or immediate problems or needs* <ad hoc solutions>
>
> *2 : fashioned from whatever is immediately available: IMPROVISED*

The latter part of definition for ad hoc highlights improvisation, which is the aspect on which we concentrate to develop a concept relevant to our study: *skilled improvisation.* According to (Merriam-Webster 2002), improvising means:

> *1: to compose, recite, play, or sing extemporaneously*
>
> *2: to make, invent, or arrange offhand*
>
> *3 : to fabricate out of what is conveniently on hand*

Improvisation then again is "the act or art of improvising" (Merriam-Webster 2002).

Improvisation does not follow processes or formalities, and it is based on existing knowledge, intuition and experience, thus tacit knowledge. We use the definition of Nonaka (1994) for explicit and tacit knowledge:

> *Explicit or codified knowledge refers to knowledge that is transmittable in formal, systematic language. On the other hand, tacit knowledge has a personal quality, which makes it hard to formalise and communicate. Tacit knowledge is deeply rooted in action, commitment, and involvement in a specific context.* (Nonaka 1994, 16)

We distinguish improvisation to *novice* and *skilled improvisation*. Novice improvisation is performed by inexperienced people that have limited and narrow knowledge or experience on the application domain or technical problems and solutions at hand. When no methods or process models are provided, novice improvisation is likely to produce poor results because the people involved do not have the adequate knowledge to make good decisions and proper implementations. If methods or process models are used, even the inexperienced and less skilled people are quite likely to produce fairly good results in standard tasks in which the predefined methods or process models can guide their work.

Skilled improvisation is performed by experienced people that are very familiar with the domain and used technology, and that have a broad and holistic picture of general development process and different phases and tasks related to it. They also have a good understanding of the business environment. If such people are in a situation where no methods or process models are in use (which often is the case for instance in start-up companies), they are likely to produce good results based on their previous knowledge and skills. If then again methods or process models are in use, employees know how to utilise them efficiently, which also includes adaptive and improvising use of them. In such cases, the results of skilled improvisation are most likely good, presuming that the technology and business environment stay at least partially stabile.

To further develop our concept definition, we will use the definition of *expertise* by Pfleeger (2001, 15):

> *...not simply the mastering of a body of knowledge. True experts live in a larger universe. They know how to recognise the relevance of what they have mastered and how to apply their knowledge and skills appropriately. Most important, experts can react to new situations, not merely to situations with which they are already familiar.*

Rasmussen (1986, 100-103) differentiates among three categories of behaviour: skill-based, rule-based and knowledge-based behaviour. Although Rasmussen uses term behaviour, these categories can also be discussed in terms of expertise, as Pfleeger (2001) defined it. According to Rasmussen, *skill-based expertise* is based on sensorimotor performance. In this case, automatic sensory and motor responses are used without much conscious effort or control. Such expertise is often used for instance in riding a bike or in playing music. (Rasmussen 1986, 100-102) In software context this type of expertise can occur in testing (looking for the same kind of faults in the same kind of code, for example) or cost estimates that are done based on previous experience from similar situations (Pfleeger 2001, 15). *Rule-based expertise* is based on stored rules, procedures and know-how established during previous occasions. Such expertise refers to awareness or the sequence of steps that need to be taken to accomplish a certain goal. These steps might not be explicitly formulated (Rasmussen 1986, 102) but for instance a good software designer usually knows and is able to take the needed steps when verifying interfaces or addressing performance issues. These two types of expertise can, however, become less useful or obsolete when problems or environment in which they are used, clearly differs from the familiar ones. (Pfleeger 2001, 15) The highest level of expertise, *knowledge-based expertise,* becomes essential when facing unfamiliar situations, in which no previous rules or know-how is available (Rasmussen 1986, 102). Knowledge-based expertise refers to recognising the goals of a decision instinctively, and making them explicit even in novel situations. After recognition, a set of procedures is shaped to attain the goal, and when such a plan is ready, rule-based expertise is used to finish the task. (Pfleeger 2001, 15-16)

Skilled improvisation derives from knowledge-based expertise. For example, a young person that has just graduated and has very little work experience will work in a novice improvisation mode if no methods or guidance is provided. It can hardly be expected that his or her results would be good in complex and demanding tasks. However, a person that has worked in the industry for several years and has participated in several

development efforts is most likely to succeed even in new and demanding situations with his or her intuition and tacit knowledge, and produce good results. The latter case refers to skilled improvisation.

Novice improvisation and skilled improvisation are presented in TABLE 1. The rows *novice* and *skilled* refer to people with different kinds of skills as described before. The columns refer to different types of situations. The first column refers to a setting where there are no defined methods of process models in use, which often is the case for instance when working with a new technology. Improvisation is used in such situations. The second column refers to a situation where defined methods or process models are used in developing software.

TABLE 1 Novice and skilled improvisation

	No defined method or process models in use / improvisation	**A defined method or process models in use**
Novice	Use of limited existing knowledge and experience → Highly varying, most likely poor results	Following methods or processes slavishly → Fairly good and repeatable results in standard tasks
Skilled	Use of existing knowledge and experience → Good results in familiar and new business and technology environments	Following methods or processes adaptively → Good results in partly familiar business and technology environments

When analysing agile software development in the context of skilled improvisation, we hypothesise that working in the skilled improvisation mode is similar to the agile way of working and assists agility. We will test our hypothesis when analysing the case study.

2.8 Software products

According to IEEE (1991), a *software product* can be defined as: "(1) A complete set of computer programs, procedures, and possibly associated documentation and data designated for delivery to a user, (2) any of the individual items in (1)." (IEEE std 610 1991, 186)

The segments and products in the information technology market can be divided into hardware related products and services, information processing services and software products and services according to Hoch et al. (2000). Software products and services can further be separated into embedded software including services, professional software services, enterprise solutions and packaged mass-market software. (Hoch et al. 2000, 27) It should be noted, that software is not a service itself but it rather enables a service to be provided or executed through it.

Warsta (2001, 34) discusses business modes that are divided into *tailored software*, modified-off-the-shelf (*MOTS*) and commercial-off-the-shelf (*COTS*). Although Warsta uses the term "business mode", his division is also valid for categorising software products, because the product focus also determines the business mode. The definitions of Hoch et al. (2000) and Warsta (2001) are very similar: professional software services correspond with tailored software, enterprise solutions are equivalent to MOTS and packaged mass-market software is similar to COTS.

In this thesis we will mainly use the software products categorisation by Warsta (2001), it will say focusing on tailored, MOTS and COTS products. Different types of products have different types of business modes and different development methods are used when developing products for a certain category. Actually most agile or traditional development methods have not been explicitly defined for a certain product or business mode. We will define later in this thesis, in which categories and under which circumstances the agile approach or certain features of it might be suitable. For instance the highly emphasised customer collaboration of Agile Manifesto cannot be interpreted as such in COTS mode where the customers are end-users in the mass-markets.

2.9 Corporate venturing

The concept *corporate venturing* will be defined here. The concept becomes relevant when we describe and analyse the case study in chapter 7. According to Sharma and Chrisman (1999),

> *Corporate venturing refers to corporate entrepreneurial efforts that lead to the creation of new business entities within the corporate organisation. They may*

> *follow from or lead to innovations that exploit new markets, or new product offerings, or both. [...] Internal corporate venturing refers to corporate venturing activities that result in the creation of organisational entities that reside within an existing organisational domain. (Sharma & Chrisman 1999, 11-27)*

Several aspects distinguish a venture from a conventional research or engineering project according to Vesper (1999):

- *A venture connotes a more complete business, including such elements as not only technical development but also profit responsibility, overall business planning, follow-through to market, production, selling, and servicing. In contrast, a project typically is more limited to particular functional specialties and lacks profit-and-loss responsibility.*
- *A venture usually involves a new direction for the firm or a more radical change in the way it does business, whereas a project connotes a more limited innovation, usually in line with the accustomed strategy and direction of the company.*
- *A venture needs greater autonomy than a project because it fits less well with the company's customary procedures. This autonomy may come about by "hiding" the venture, by separating it geographically, or by housing it in a special organizational unit capable of shielding it from the normal company activities.* (Vesper 1999, 29)

Block and MacMillan (1993) on their behalf set a line to distinguish a corporate venture from a project. According to them, a project is a corporate venture when it

- *involves an activity new to the organisation*
- *is initiated or conducted internally*
- *involves significantly higher risk of failure or large losses than the organisation's base business*
- *is characterised by greater uncertainty than the base business*
- *will be managed separately at some time during its life*
- *is undertaken for the purpose of increasing sales, profit, productivity, or quality.* (Block & MacMillan 1993, 14)

There are several benefits and reasons why corporate ventures are set up. According to Vesper (1999, 26), corporate venturing aims at making a combination of the reputation, talent and resources of a major established company with the innovativeness, flexibility,

aggressiveness and frugality of a start-up to move the big company smartly ahead in their business. Käkölä (2001, 7) adds:

> *Corporate ventures create, experiment with, and combine new technologies to come up with new business and product concepts for mass markets that may not be mature enough to adopt such products for years or ever. Corporate ventures can be highly successful from the viewpoint of the corporation even if their concepts fail. For example, ventures can be used to create and experiment with new partnerships with other companies.*

Thus, a corporate venture is more than just a development project. The venture will step out from the parent corporation's established procedures and experiment new technologies or new, risky business models aiming at making profit or at least gaining new knowledge and contacts for the parent corporation. The parent corporation supports the small venture in numerous ways, providing for instance material and financial resources. Symbolically the parent corporation can be seen as an ocean-going vessel that moves steadily forward over the sea but turning such a boat is slow and difficult. The corporate venture then again could be seen as a speedboat equipped by the mother vessel moving fast while it explores unknown waters in which it can find a treasure island or sink to the bottom if it hits rocks.

3 EVOLUTION OF SOFTWARE DEVELOPMENT

In the late 1980's Brooks (1987) presented essential problems concerning software development in his famous search for a silver bullet as an answer to growing development expenses and missed timetables. He suggested that complexity, conformity, changeability and invisibility were the main problems in software development. *Complexity* refers to different states that entities of for instance a program can have and to non-linear growth of complexity as the software is scaled-up. Complexity can cause problems in communication between development team members and difficulties in estimating and managing the workload. *Conformity* refers for example to the different interfaces a software needs to adapt to, as it often needs to conform to existing institutions, technical machines or working routines. *Changeability* means that software is constantly subject to pressures for change. As the technical or social environment changes, software needs to be changed, whereas for instance houses or bridges are seldom modified after their completion. Software is *invisible*, it is abstract: it is troublesome to try to visualise software and its components and functions. Brooks believed that some answers, if not a silver bullet, could be found in high-level languages (such as ADA), prototyping and incremental development. He also emphasised good developers as key success factors. (Brooks 1987, 11-18)

Compared to today, not much has radically changed in the nature of software and its development since Brooks' article was written. Prototyping, incremental development and heroic accomplishments of individual developers are used more and more in software development. Technology has become more developed, and development methods have become more feasible and effective but it all still boils down to some general problems of software development that Brooks highlighted.

The business environment has instead changed remarkably. Because of that software has to meet the requirements set by the new volatile business environment, and as a result systems are becoming more and more complex. In addition, software has to be integrated into several different interfaces, and the time-cycle of developing and completing software is becoming shorter.

Different kinds of software development methods have been developed to handle above-mentioned problems like complexity or time-to-market pressure caused by business environment. In this chapter we explain how software development and software business have evolved through recent decades. In addition, we aim to clarify the major changes in business environment that have had an impact on software development and software business in general. Finally, we aim to motivate why agile methods have been taken into use.

3.1 The eras of evolution

Computers were taken in commercial use in the 1950's and now they have been in use for half a century. During the first four decades the term describing computerised activities changed from data processing to management information and further to information processing (Somogyi & Galliers 1986, 1). Since about 1990's computers and information systems have been integrating businesses and are now one of the key success factors in competing in the rapidly changing markets.

We divide the past decades into eras according to Somogyi and Galliers (1986) and add a new era that started in 1990's. Names of the eras are characterised by the nature and the purpose of information processing. The eras of evolution are the following:

- Data processing (started in early 1950's)
- Management services (started in mid 1960's)
- Information processing (started in early 1980's)
- Business process integration (started in early 1990's)

Similar classification of these four eras can also be found for example in Pressman (1994, 4). Different classifications exist that vary country by country or by their emphasis (for example some technologies have become popular first in the U.S. and some years later in Europe) but the classification described above suits the extent of this research.

Next, we briefly explain the major trends of each era. We will concentrate mainly on the reasons for using technology, the nature of technology, the characteristics of system development, and we will point out the most significant technical innovations, of which some are still drivers in technical and theoretical development. FIGURE 4 gives an overview of the development of applications and methods since 1970's (the first era is not described in the figure due to its lesser significance for this study) and is used as our thread as we go through the history. We will not cover all the methods and applications mentioned in the figure in this thesis but we assume that the concepts are already familiar to our readers.

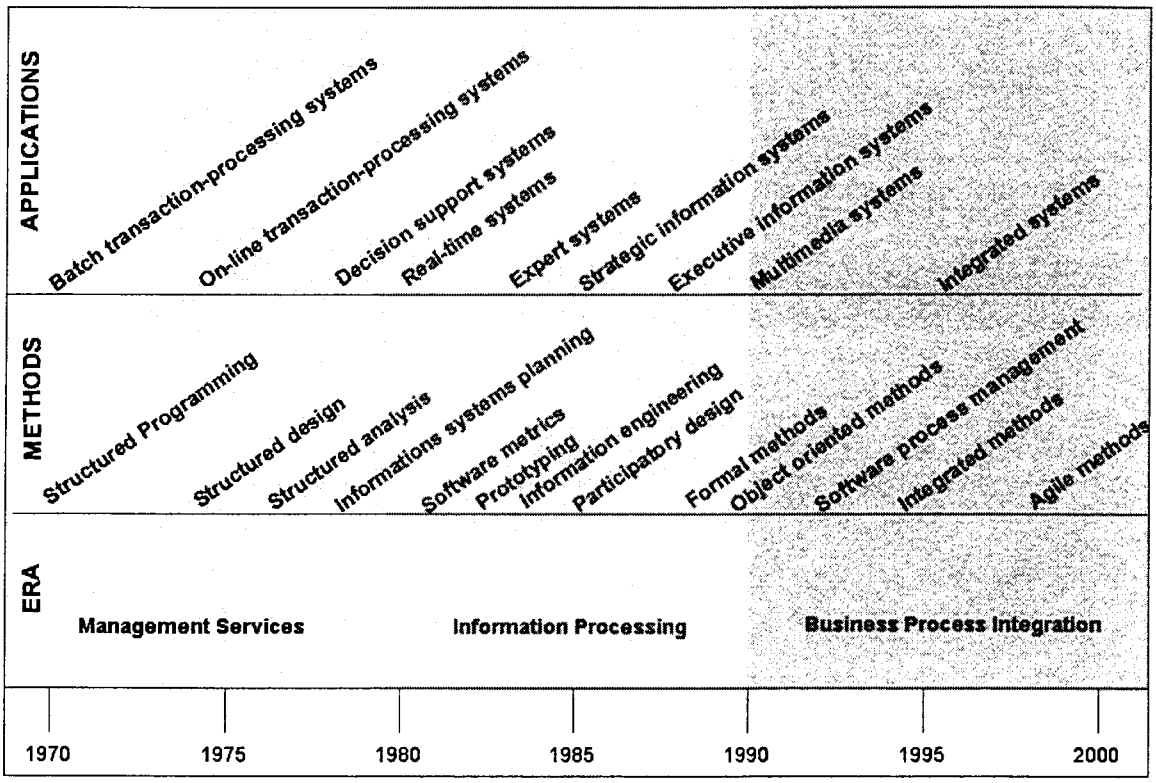

FIGURE 4 History time line (adapted from Norman & Chen 1992, 14)

Here we use both the terms information systems development and software development (defined in chapter 2.1) when describing the evolution, although the focus of this research is mainly on software development. We aim to emphasise that software is not a separate entity but the use of software as part of an information system has an

effect on organisations and individuals and vice versa, and that this conception has also evolved through the history.

3.1.1 Data processing

The first computers became available in early 1950's. Mainly scientists and code-breakers used the first gigantic computers as calculators in those days. When computers were taken into commercial use in a larger scale, they were used to automate routine work in companies. The typical systems developed and used at that time were for example payroll or general ledger systems. (Somogyi & Galliers 1986, 2-3)

Software development in this era was mainly ad hoc -based. The developers had to concentrate on making the computers run rather than on rationalising the development processes. From the present point of view, the developers had quite naïve thoughts of behavioural and organisational aspects of their work at that time. However, they were confident that their discipline would become a science. The most significant technological innovations and outcomes of this era were COBOL and FORTRAN programming languages, in addition to time-sharing and time-slicing concepts in computing. (Somogyi & Galliers 1986, 2-3)

3.1.2 Management services

In the 1960's computers were becoming pervasive and commercial systems were becoming more reliable and efficient. Most large corporations were using big mainframe computers, which the data processing departments were operating separated from the actual users. In spite of remarkable progress in hardware and data processing applications, companies started to examine the merits of the computerised systems that they were using. Although companies were able to decrease the number of moderately paid clerks, they had to hire highly paid data processing professionals, not to mention the high costs of hardware investments. Companies also noticed that maintenance of data processing systems was unexpectedly high. Dissatisfaction of the users increased as the systems were inflexible and arrangements in general highly formal. The term and discipline *software engineering* was introduced in the 1970's aiming to find solutions to

some of the above-mentioned problems and to develop methods for requirements specification. At the same time project management techniques were improved as a response to several miserably unsuccessful development projects that went far beyond their original budgets and time estimates. (Somogyi & Galliers 1986, 3-5)

Application of engineering principles is one way to try to control the complexity and development of objects and processes. However, applying engineering practises has not been unproblematic due to the complex and changing nature of software and its environment. Also paradigms (see chapter 2.4) were developed to assist understanding and handling complex software development. The classic life cycle paradigm originates from this era.

As more and more office operations became automated, management's need for cross-relating and cross-referencing data increased. Cumbersome, stand-alone batch systems could not fulfil these needs. This notion resulted in birth of some basic ideas in systems development: data should be de-coupled from the basic operations, and it should be timely, precise and available. This resulted in the creation of relational database and normalisation in the 1970's. Management information system supporting management decisions was a trend around mid 1970's. (Somogyi & Galliers 1986, 8-9)

In addition to the relational database, the most important technical accomplishments of this era according to Somogyi and Galliers (1986), were minicomputers that made utilisation of computing power possible for small companies too. In the field of programming languages, PASCAL was developed as a response to "spaghetti logic" of the previous programming languages to enhance the control, structure and logic of programs. Structure and modularity became important in programming and systems development. In fact, structure and modularity are still a major intellectual drive in both practice and theory associated with computer systems. (Somogyi & Galliers 1986, 4-6) Structured design and analysis followed structured programming around mid 1970's (Pickering 2001, 3).

In the first decades of information systems history software was tightly bundled together with the hardware as one product with one price. Big mainframe manufacturers undertook software projects for their biggest customers but they did not have enough

resources to serve the medium-sized customers. That is where the first software company CUC (Computer Usage Company) came in starting to offer their services for these customers. CUC and other American pioneer vendor-independent software companies offered mainly programming services in the form of development projects to the U.S. government and to individual enterprise customers in the 1950's - 1960's. (Hoch et al. 2000, Johnson 1998)

The first independent software products (see the definition on page 22) appeared around the mid 1960's although most computer executives believed that there would never be a significant market for stand-alone software (Hoch et al. 2000, 262). The first patented software product was ADR's flowcharting program. The first million-dollar software product was *Mark IV* (a file management system for IBM 360s), released by Informatics in 1967. (Johnson 1998, 37-39) ADR and Informatics had to face challenges such as intellectual property rights, pricing, customer support and maintenance related to their completely new and innovative products and business models (Hoch et al. 2000, 263). Those challenges are still key issues in software business.

IBM, a hardware and software company that dominated the market throughout the 1960's, sold software bundled together with their hardware. In addition to that, IBM provided customer service and access to their software library free of charge. IBM decided to separate hardware and software to independent products in 1970. It can be stated that IBM's decision was crucial to the software industry and opened a possibility to software business to evolve and grow. (Johnson 1998, 36) IBM's unbundling decision made it easier for independent software companies to develop and market their products since now customers had to pay for their software anyway: to IBM or to someone else. Emerging software products of this era (1970-1980) were tailored enterprise solutions, standard enterprise products (e.g. *SAP, Baan*) and database systems. (Hoch et al. 2000, 264-266)

3.1.3 Information processing

In the early 1980's microcomputers, also known as personal computers (PC's), were quickly entering the markets and causing major changes. First, the individual users were

able to use and buy microcomputers, and their skills and attitudes became more positive towards information systems and using computers. Now that computers were operated by anybody instead of educated professionals, usability became an important issue. Second, the low cost of the small computers highlighted the remarkable cost of human effort that companies were spending in software development and maintenance. Third, the idea of interconnecting small computers accelerated convergence of telecommunication, office equipment and computing. (Somogyi & Galliers 1986, 10) Tiny microprocessors were taken into use in several other equipments too. Computing power started to operate industrial robots, cars and even microwave ovens (Pressman 1994; Hoch & al. 2000). Ready-made applications and tools for system development became available and popular. New types of applications, such as voice, text and image processing and electronic mail, were taken into use in the personal computers. (Somogyi & Galliers 1986, 14)

In the field of systems development, it became clear that the linear life cycle view, introduced in the 1960's, was not feasible for several reasons. To name a few, misconceptions and errors moved on to the next steps and multiplied as errors caused more errors. Linear life cycle thinking also assumed that the requirements could be set in the beginning and that they will not change, which was a false assumption. As a response to these problems, new methods and paradigms were developed. Prototyping, quality management and regular tests and checks in form of walkthroughs and inspections were taken into use to cut down efforts and costs of information systems development. (Somogyi & Galliers 1986, 11-13) In prototyping paradigm the requirements are gathered in the beginning, which is followed by quick design and building a simple prototype. The customer evaluates the prototype and after that changes are made and the object system is completed. FIGURE 5 illustrates the prototyping paradigm. Pressman (1994, 27) states that prototyping ideally serves as an mechanism to collect requirements but other paradigms and methods suit the quality and maintainability oriented software engineering better.

However, the prototype paradigm did not seem to cover all the aspects of information systems development either. The iterative spiral model was developed to encompass the best features of both the classic life cycle and prototyping. Spiral model adds a new

important element, risk analysis, that is missing from the waterfall and prototyping paradigms. (Pressman 1994, 29) FIGURE 6 illustrates the spiral paradigm. The figure is a slightly altered version of the original model in Boehm (1988, 64). Due to limitations and focus of this thesis we will not concentrate on this paradigm further. See Boehm (1988) and Pressman (1994) for detailed descriptions of each iteration cycle and related tasks.

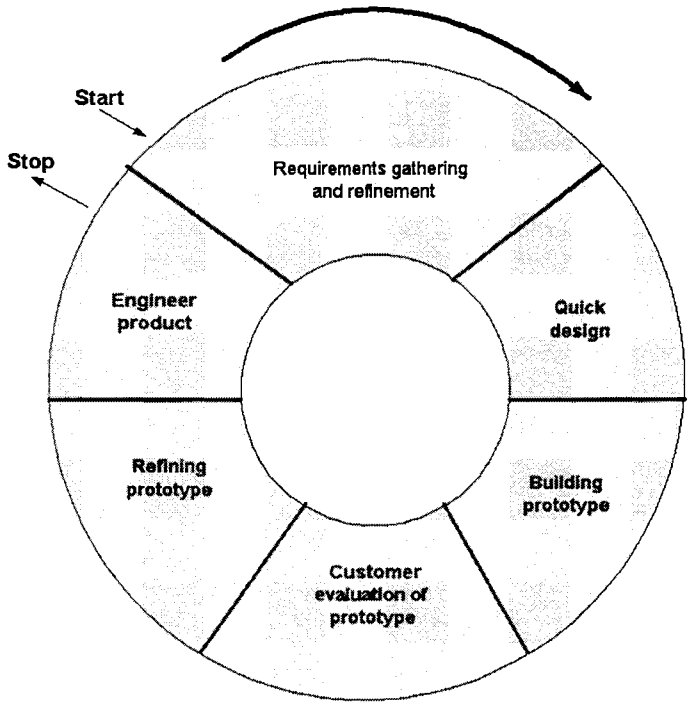

FIGURE 5 Prototyping paradigm (Pressman 1994, 27)

Real-time systems design and object-oriented analysis and design were developed during the 1980's to amend shortcomings of the structured techniques. In addition, CASE tools were developed to help the development process. Development of CASE tools started in 1970's and by the end of 1980's they were repository-based graphical tools that supported planning, analysis, design, programming, testing and maintenance. However, CASE tools depend heavily on a specific method and tend to support only certain types of application development, and thus are not feasible in all situations. (Norman & Chen 1992, 14-15)

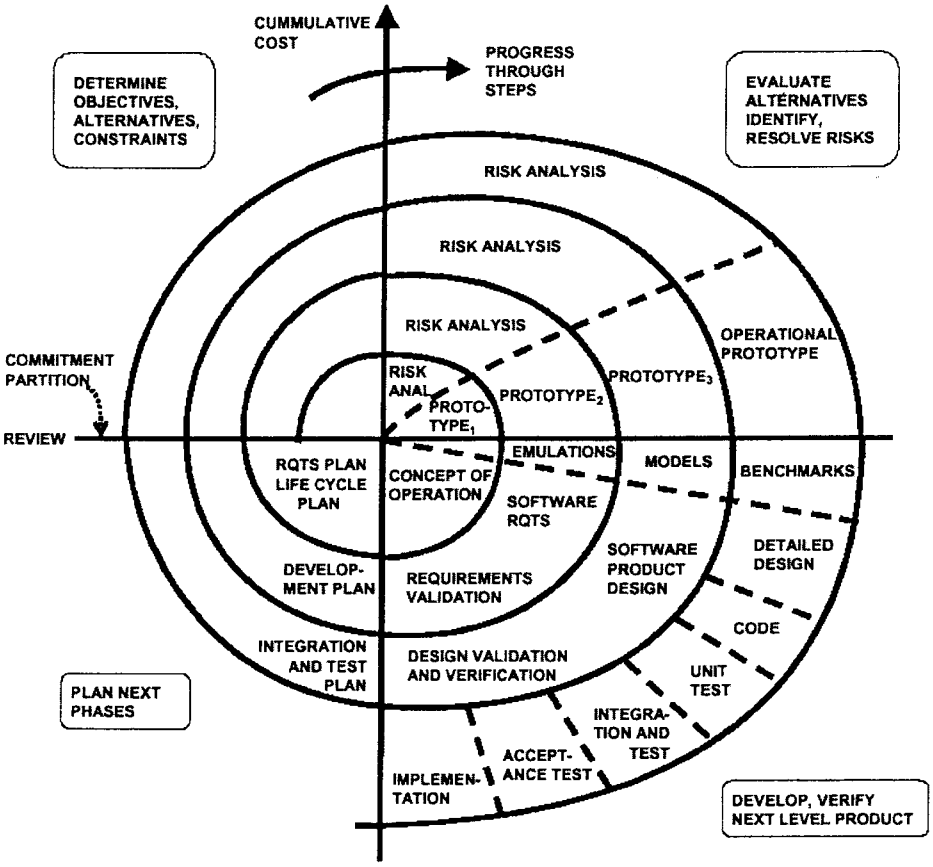

FIGURE 6 Spiral development paradigm (Boehm 2000, 2)

In the field of programming fourth generation languages (4GLs) emerged, and they were targeted to both professionals and amateurs. In addition to 4GLs, automated programming support environments, analysis and programming workbenches came into the markets. (Somogyi & Galliers 1986, 11) As the code generators and fourth-generation languages were simplifying implementation activities, the bottleneck in development shifted to the upstream activities, such as systems planning, enterprise modelling and requirements engineering. The quality of the upstream activities is determined by how well the systems personnel can get the users and managers involved in the development. (Norman & Chen 1992, 14) The conception about the role of organisation in information systems and their development changed radically around 1980's. Practitioners and researchers acknowledged that a computerised information system is like a two-sided coin, other side being the human organisation that uses the

information systems. The developers acknowledged that the end-users should be included in the development process and that the needs of organisation should be taken into account in the development process. (Somogyi & Galliers 1986, 13) On theoretical level the concept of information system was further broadened to include language as a field of information systems in addition to technology and the organisation (Lyytinen 1987a; Iivari 1989).

At the same time the role of information systems in business changed. Companies started to see information systems as competitive advantage and the need for linking information systems with the business and connecting the business strategy with the information system strategy was rising. (Somogyi & Galliers 1986, 14-15)

In the software business field the enterprise solutions kept evolving as mainframe computers were accompanied by PC's and new operating systems like *MS DOS* and *Windows NT*. PC's were also becoming increasingly popular in the consumer markets. *VisiCalc* for Apple II computer was the first killer application in this field. Later on IBM's platform together with Microsoft's operating system offered a PC standard on which several success mass-market software products, developed by companies like Adobe, Corel and Lotus, were build. The 1980's saw dramatic growth rates (up to 20% per year) in the American software industry. (Hoch et al. 2000, 268-269)

3.1.4 Business process integration

The need for the integrated information systems to span all business functions, all organisational levels and even global locations has been growing strongly since the 1990's. To enable the integration, several technologies need to be combined, and often the development work should be carried out in a very short time (Norman & Chen 1992, 14). In addition to the pressure for short time frame, the quality of information systems is the primary challenge and requirement for this era as systems are becoming more complex through distributed component use and reuse (Pressman 1994; Brereton et al. 1999).

In addition to the view that organisations and information systems have an impact on each other (Orlikowski 1992; Baskerville et al. 1999), it has also been acknowledged

that software and information systems affect the whole society (Brereton et al. 1999). This further increases the complexity of information systems: software failures may have significant implications if it is for instance used in national elections. Thus, the software services should be reliable, secure and stable. In addition to support to several technical platforms, information systems also need to support many different mental models in order to serve all users. (Brereton et al. 1999, 80-81)

Software process management, for instance in the form of CMM, was taken into use in the beginning of the 1990's (see FIGURE 4). As a counteraction to the heavy process models, agile software development methods were introduced about ten years later in the turn of the millennium. When it comes to the future of software development, Brereton et al. (1999, 83) made the following prediction about future development methods: "[...] it may be possible to evolve software by learning from biological models, through which evolution of incredibly complex structures has been achieved."

The biggest change of this era in both the business and application fields has been the rapid rise of Internet and World Wide Web (WWW) that started in the mid 1990's. There is no doubt about the fact that Internet and its services were truly causing remarkable changes. Those can also be classified as *discontinuous innovations*, which are "new products and services that require the end-user and the marketplace to dramatically change their past behaviour, with the promise of gaining equally dramatic new benefits". (Moore 1995, 13) Changes in communication, marketing and trade were indeed dramatic because of the rapid expansion of individual and enterprise users of the WWW and electronic mail.

The term *Internet-year* was soon launched. That does not refer to 356 days but rather to few months, within which new products and services need to be developed and introduced to the customers instead of several years of development time as in the past. Companies believe that first one in the market gets to set the standards, and to become the market leader. (Cusumano & Yoffie 1998; Pickering 2001) Cooper (1996, 13) states the following concerning speed and time-to-market issues:

> *Speed is the new competitive weapon. Speed yields competitive advantage: to be first on the market; it means less likelihood that the market or competitive*

> *situation has changed; and it results in a quicker realisation on profits. [...] however, speed is only an interim objective; the ultimate goal is profitability.*

Thus, the time-to-market factor has become very important. According to Cockburn (2001, xxii), deploying software to the WWW has intensified the software competition further than before: winning in business involves winning in software development competition. Software developers also need to track down continually moving user and marketplace demands. Nevertheless, Pickering (2001, 17) argues that information systems development projects have always been under time and market pressures, and that those pressures are not characteristics of the "e-projects" of today only.

The takeoff of the Internet opened possibilities to new kinds of businesses. Netscape was one of the great success stories of the 1990's, for a while at least, and other e-business players were for instance Amazon and Yahoo. (Hoch et al. 2000, 269-270) Many other companies rose like rockets and came down as quickly as they went up while the late 1990's was living the e-business hype. The form and usage of software has changed through the recent decades, and so has the idea of software. Traditionally software has been regarded as a product, now the notion is changing towards a service rather than a product (Brereton et al. 1999, 81). However, it should be noted, that software is not a service itself but it rather enables a service to be provided or executed through it. FIGURE 7 summarises the evolution of different types of software products through the eras we described in more detail above.

As we can see, software as a product and as business has gone through many transformations. Software projects are still made, totally or partially tailored but the mass-market products and services have become remarkably important. The above-mentioned issues like the time-to-market pressure have led to a situation, where the traditional software development methods have been found infeasible and new approaches have been developed. Next we will examine the limitations and shortcomings of the traditional methods aiming to motivate why the need for the agile development methods has risen.

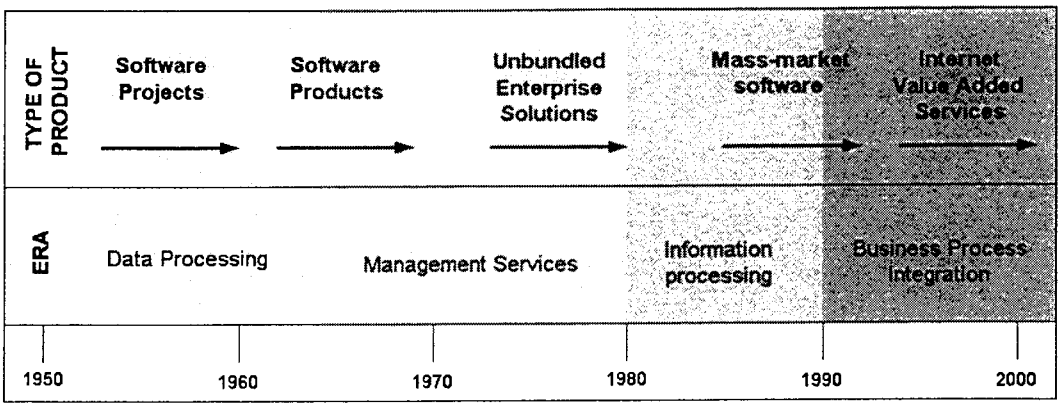

FIGURE 7 Different types of products on a timeline

3.2 Limitations of the traditional methods

By traditional or heavy methods we mean the process control or documentation oriented methods like structured analysis and design. Additionally we include CMM and RUP to the traditional methods although CMM and RUP can also be classified as process models. In this context, however, we consider them as methods, because for instance CMM for software (SW-CMM) exists that measures the maturity of software creation process. See further information about the process models for example in (Zahran 1998).

Several arguments against the traditional information systems development methods, and suggestions to improve them have been presented in the literature. Tolvanen (1998) mentions reasons why companies do not want to use development methods. According to his study, the improvements caused by the methods are modest, methods are considered labour-intensive, methods are difficult to use and learn, and methods have poorly defined and ambiguous concepts. Methods are also regarded to limit and slow down development, to generate more bureaucracy and to be unsuitable. (Tolvanen 1998, 51) However, there are of course successful examples of the use of the development methods too, the attitudes and experiences are not entirely negative.

Baskerville, Travis and Truex (1992) name numerous reasons why they think traditional information systems development methods are not suitable. They state that the post-modern business organisation is fundamentally different compared to the forms based

on the industrial organisational view of markets, competition and regulation, and that development methods do not acknowledge that. They also mention that systematic approaches to information systems development are oriented toward a few large-scale, long-term development projects but now technology and environment have brought incremental and short-term needs. Baskerville et al. suggest *amethodical systems development* to better fulfil requirements set by the post-modern organisations and by the changing technological and competitive environments. They say that the mode of development might be uncontrollably shifting from the structured, orderly and methodical mode to the unstructured, emergent and amethodical. They also make an observation that the role of system developers is evolving from system analysts into business analysts since the changes in information systems are not only technical but also affect the organisation and its business. (Baskerville et al. 1992) Although it seems that Baskerville et al. mainly discuss large, tailored software or information systems developing projects, their observations can be broadened to cover other kinds of software development efforts as well. Some of their ideas sound very similar to ideas of agile software development:

> *Rapid, small-scale information systems development will be critical if organisations are to adapt quickly to their environments, and thereby find success in the post-modern globalisation of commerce and industry.* (Baskerville et al. 1992, 248)

Avison and Golder (1992) suggest that the traditional, hard development tools like entity modelling and data flow diagramming do not take the disorganised world of people into consideration. However, *soft systems methods*[2] do that by acknowledging the importance of people in the organisations and by helping analysts to understand the role of people. They also point out that in the hard systems thinking there is an aim to reach a certain goal but in the soft systems thinking it is assumed that the system is more complicated than a simple goal that can be achieved and measured. (Avison & Golder 1992)

---

[2] Originally by Checkland, *Systems Thinking, Systems Practice.* 1981

DeMarco and Lister (1987) argue that a strict use of methods might encourage people to concentrate on documentation instead of the actual work and that it might cause absence of responsibility, because the methods do not give people enough freedom or responsibility. Besides, rigorous methods can lessen the use of imagination and creativity, and as a result decrease motivation. They do admit that convergence of working practises is needed in order to work effectively in teams but instead of using rigorous methods they propose training common techniques and practises, the use of compatible and efficient tools, and walkthroughs and inspections among peers. (DeMarco & Lister 1987, 116-118) Hence, DeMarco and Lister want to abandon the heavy methods in order to enhance people's motivation and efficiency, and close collaboration within teams.

Boehm (2002, 67) states that plan-driven (traditional) methods are most needed in critical high-assurance software for instance when developing life-critical systems. Boehm (2002, 67) adds:

> *Another set of objectives for the more traditional, plan-driven methods, and for the software CMM, has been predictability, repeatability, and optimisation. But in a world involving frequent, radical changes, having processes as repeatable and optimised as a dinosaur's may not be a good objective. Fortunately, the CMMI[3] and associated risk-driven methods provide a way to migrate toward more adaptability.*

We cannot analyse all the shortcomings or problems of the traditional development methods comprehensively within the scope of this thesis but we state that the main problems of the traditional development methods are their inability to face challenges set by changing organisational, business and technical environment, their insufficient emphasis on individuals and individual talent and creativity. Traditional methods can also be considered bureaucratic and restrictive. Agile methods aim to provide solutions to those problems by paying attention to reacting to change, individuals and

---

3 "Capability Maturity Model Integration (CMMI) provides guidance for improving organisation's processes and ability to manage the development, acquisition, and maintenance of products and services." Source: http://www.sei.cmu.edu/cmmi/general/genl.html [referred on 18.4.2002]

collaboration, and producing working results rather than documents. Eischen (2002, 36) claims that emphasis on craft-based and engineering approaches has been alternating for thirty years and that the contemporary discussion is another wave of this discussion in which the craft-based approach is now more popular instead of the engineering approach. For example CMM as a traditional, engineering approach was popular in the 1990's and now agile, craft-based approach is highlighted.

Although traditional methods have their drawbacks and shortcomings, they have brought many significant improvements to information systems development. Agile software development methods utilise the heritage of the traditional methods for instance by using iterative and incremental approach. Agile software development and Agile Manifesto will be closer explained and examined in chapter 4.

3.3 Summary

This chapter described the history in the field of information system and software development. We went through the main changes in the purpose of using information systems or software, mentioned the main technical inventions and also went through the history from evolution of the software business point of view. Ad hoc -methods in the 1950's and 1960's were replaced by systems life cycle thinking and standardised process models as a means to handle complexity and to add understanding of different aspects related to software development. However, in the course of time development methods became heavy, for instance as the amount of documentation grew enormous. In addition, the nature of business and technology has brought pressure to software development. Different kinds of goals, such as business intentions, security requirements and the increasing complexity of software, for example in the form of integration with several different interfaces, have resulted in the need for producing software quickly but while maintaining high quality. The tendency is partially changing from traditional, rigorous methods to agile development methods that were developed as a counteraction against traditional methods, which in some cases seem infeasible in the current rapidly changing business environment. In addition, changes in the technical and business environments have resulted in the invention and use of agile development methods.

4 AGILE SOFTWARE DEVELOPMENT

> *For many people the appeal of these agile methods*[4] *is their reaction to the bureaucracy of the monumental methods. These new methods attempt a useful compromise between no process and too much process, providing just enough process to gain a reasonable payoff.* (Fowler 2001)

As stated in the previous chapter, traditional software development methods are not always feasible in the rapidly changing business environment. However, the agile development approach has been claimed to be such. In this chapter, we will further examine the different aspects of agile software development using Agile Manifesto as our initial framework. We have chosen to use Agile Manifesto as the basis for explaining and evaluating different ideas related to agile software development, although Agile Manifesto mainly derives from practice, not so much from theoretical models. However, since we have found no approved theoretical framework for agile software development, we use Agile Manifesto and its values and principles for our analysis. We aim to develop a conceptual agile framework in this chapter. We will further modify it based on the analysis of two agile methods, XP and agile in-house software development, and present the modified framework in chapter 8.

The concept Agile Manifesto is often used in this thesis. These two words, agile and manifesto, both give descriptive input for this concept. As defined in chapter 2.5, the term agile refers to adaptability and ability to move quickly. According to Oxford dictionary, *a manifesto* is "a public declaration of principles, policy, purposes, etc by ruler, political party, etc or of the character, qualifications or a person or group" (Hornby 1974, 517). Several different manifestos challenging the status quo and proposing new ideas have been declared through the history, such as the revolutionary Manifesto of the Communist Party by Marx and Engels. Manifesto for Agile Software Development (Agile Manifesto) aims to question the current situation in software development, as it was created as a counteraction against the heavy, process and

---

[4] Fowler (2001) used *methodology* in the meaning of method in the original source.

documentation oriented methods and it aims to suggest new approach to software development.

Our primary aim in this chapter is to explore literature and research related to the principles and values of Agile Manifesto. Authors like Brooks (1987), DeMarco and Lister (1987), Cusumano and Yoffie (1998) and Sutton (2000) have expressed several ideas very similar to agile software development's ideas although they have written their articles and books independently not knowing about the agile movement.

In addition, we aim to indicate that the roots and best practises of agile methods originate from the traditional information systems development and implementation practises. We aim to explore whether there are new and innovative features in agile software development. We begin by explaining the background of Agile Manifesto and its creator Agile Alliance, and presenting the values and principles of Agile Manifesto in detail. We will reflect the principles of Agile Manifesto to the existing literature, and present favourable and contradicting arguments concerning the principles. Based on our literature review we will develop a conceptual agile framework. Finally, we will analyse the feasibility of agile methods.

4.1 Background and values of Agile Manifesto

Agile Alliance was formed when seventeen representatives of different agile methods, such as Extreme Programming (XP), Scrum and Crystal Family, met to discuss alternatives to rigorous, documentation driven software development. Agile Alliance did not want to specify detailed project tactics mainly because the members represent competing companies but rather agree on values that support agile or lightweight software development at a high level (Agile Alliance 2002). Thus, Agile Manifesto is a collection of values and principles, which can be found in the background of the most agile methods. Agile Alliance describes its intentions as follows:

> *The Agile movement is not anti-method*[5]*, in fact, many of us want to restore credibility to the word method. We want to restore a balance. We embrace modelling but not in order to file some diagram in a dusty corporate repository. We embrace documentation but not hundreds of pages of never-maintained and rarely used tomes. We plan but recognize the limits of planning in a turbulent environment. Those who would brand proponents of XP or Scrum or any of the other Agile Methods as "hackers" are ignorant of both the methods and the original definition of the term hacker.* (Agile Alliance 2002)

Boehm (2002, 64) adds to the debate whether agile methods are "hacker methods" or not, and believes that actually agile methods "are drawing many young programmers away from the cowboy role model and toward the more responsible agile methods".

Agile Alliance formulated their ideas into values and further to twelve principles that support those values. (Agile Alliance 2002; Cockburn 2001, 215) Values of Agile Manifesto are the following:

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

The authors of Agile Manifesto or other people writing supportively about agile software development (*agilists*) have not defined the concepts *traditional or heavy methods* against which some of them are heavily fighting. We found no explicit explanation to which exact methods agilists were referring to, they did not clearly point out the shortcomings of the traditional methods' in a detailed level. CMM was mentioned in several contexts but to be precise the Software CMM (SW-CMM[6]) can be compared with agile methods to some extent. The entire CMM is a much broader concept of process improvement. In addition, the concepts *method* and *process* are not

---

[5] Agile Alliance used *methodology* in the meaning of method in the original source.

[6] Summary of SW-CMM can be found at URL http://www.sei.cmu.edu/cmm/cmm.sum.html [referred on 16.5.2002]

clearly defined nor systematically used. Sometimes the agilists compare Agile Manifesto and agile methods to process models, sometimes they are compared with development methods.

The authors of Agile Manifesto and the inventors of agile methods do not distinguish in which different business modes (tailored, MOTS or COTS) agile ideas and methods are suitable. However, we will consider this division in our study. Most of the authors of Agile Manifesto work as consultants at different kinds of companies (Agile Alliance 2002). Although their business modes are not defined, we assume, based on their approach to Agile Manifesto, that they mainly work in the tailored business mode. Next we will discuss each value one by one.

4.1.1 Individuals and interactions over processes and tools

Traditional methods usually aim to develop a process that distinguishes human resources as roles, such as a project manager, an analyst or a programmer (see an example of CMM roles in Paulk et al. 1995, 59-65). Such processes emphasise roles, not individuals performing tasks related to a particular role. According to DeMarco and Lister (1987), managers often want to think that people are interchangeable. The managers believe that in order to keep the projects running a role can be filled by somebody else if an individual decides to leave the project. In a production environment such thinking may be convenient (DeMarco & Lister 1987, 9) but in creative work, such as designing software, individuality and personal competence is highlighted, and therefore it is difficult to replace an individual.

Agile methods reject such assumption that the people involved in software development are replaceable parts (Fowler 2001; Cockburn 2001, 217). Although process descriptions and organisation charts are required to get the project started, Agile Alliance wants to emphasise the individual people over roles and encourage interaction between the individuals (Cockburn 2001, 217). Interaction and communication between people are frequently addressed issues in the literature concerning agile software development.

This value statement also wants to abandon the use of strict process models such as CMM. Nevertheless, processes can also be beneficial to certain extent, and a total abandonment of processes should be avoided. Sutton (2000) discusses the role of processes in start-up context and we find some of his thoughts applicable for other, more established organisations that develop software in a fast-paced, reactive, and innovative world of commercial software development. Agility is often characteristic of a start-up company or a company operating in an environment described above, thus Sutton's ideas are also feasible in the agile context. According to Sutton (2000), defined processes can help ensure product quality and decrease process costs. They can also set expectations about development activities and provide a framework for process and project management. In addition, development groups and the broader organisation are likely to communicate important development practises more easily. Finally, if new people are hired to the company, defined processes can help inform them getting know the company and its working methods. (Sutton 2000, 36) Thus, processes can be very useful in many ways in guiding people instead of limiting their work.

4.1.2 Working software over comprehensive documentation

According to Agile Alliance, documents containing requirements, analysis or design can be very useful to guide developers' work and help predict the future. But working code that has been tested and debugged reveals information about the development team, the development process, and the nature of problems to be solved. Agile Alliance claims that running program is the only reliable measure of the speed and shortcomings of the team and gives a glimpse into what the team should really be building. (Cockburn 2001, 217) Although this statement might sound radical, in Boehm's (2002, 64) opinion "agile methods often appear less plan oriented than they really are", thus documents and plans are used in agile methods to a certain extent.

Agile Manifesto and agile methods mainly concentrate on implementation but also modelling can be done in agile way. See Appendix 1 for more information concerning agile modelling.

4.1.3 Customer collaboration over contract negotiation

According to Cockburn (2001), this value statement emphasises close relationships between the software development team and the customer. Cockburn (2001, 218) describes collaboration as follows: "Collaboration deals with community, amicability, joint decision making, rapidity of communication, and connects to the interactions of individuals". Attention to customer collaboration indicates an amicable relationship across organisational boundaries and specialities. Agile Alliance suggests that fruitful and close collaboration can make contracts unnecessary, and if a contract situation is in jeopardy, good collaboration can save the situation. (Cockburn 2001, 218) The basic assumption behind this value statement is customer satisfaction in general, which is the main driver in agile software development.

Customer collaboration and dedication are critical to all software development projects. Boehm (2002, 66) states:

> *[...] unless customer participants are committed, knowledgeable, collaborative, representative, and empowered, the developed products generally do not transition into use successfully, even though they may satisfy the customer. [...] Agile methods work best when such customers operate in dedicated more, with the development team, and when their tacit knowledge is sufficient for the full span of the application.*

Thus, if the customers do not see the software development as their highest priority and if they do not have enough knowledge about the subject, more rigorous, traditional methods should be used instead of agile methods.

We find this value statement suiting exclusively MOTS and tailored business modes. In COTS business the customer is an end-user in the mass-market and tight cooperation with them is very limited. Naturally, the wishes of and feedback given by the end-users can be taken into account to some extent but usually end-users do not participate in the development in a way as customer collaboration is understood from Agile Manifesto point of view. Customer collaboration differs also when subcontracting, and when using components of different partners or companies in development. The inventors of Agile Manifesto have not discussed this issue thoroughly according to our knowledge.

4.1.4 Responding to change over following a plan

The requirements for a new software system will not be completely known until after the users have used it. There have been findings in research, that uncertainty is inherent and inevitable in software development processes and products, and that it is not possible to completely specify an interactive system. (Sutherland 2001, 6) Requirements change constantly because of uncertainty and the interactive nature of software, and because of the fluctuating business and technology environments. Those changes should be taken into account in software development. Agile Alliance admits that plans are useful and planning is actually included in agile methods, which also have mechanisms for dealing with changing requirements. However, instead of following a plan rigorously, development teams should constantly reflect the plan to the current situation and flexibly change it accordingly. (Cockburn 2001, 218) Soft systems methods (see chapter 3.2) are in line with this thinking as they also assume that the requirements and the goal of development cannot be fixed in the beginning but instead they will change and that should be acknowledged.

Flexibility is essential for the development process of a start-up company (Sutton 2000) but similarly flexibility is relevant for organisations that work in an agile mode, and constantly need to adapt to changes. Sutton (2000) states that such flexibility is needed to accommodate changes for instance in the personnel and infrastructure, product specifications, resource levels, and release schedules. Although external conditions are very hard to predict and control, changes caused by them require timely and rapid responding. A defined but flexible process can assist in this, as a flexible process provides the employees continuity while facilitating adaptation. Flexibility assists organisations in handling process exceptions and deviations, and a flexible process definition provides an effective model to achieve desired results in occasional executions. (Sutton 2000, 37)

4.2 Principles of Agile Manifesto

The values described above are realised in the principles of Agile Manifesto. We will analyse each principle of Agile Manifesto in detail in the light of the established and contemporary literature. The principles are the following:

1. *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
4. *Business people and developers must work together daily throughout the project.*
5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
7. *Working software is the primary measure of progress.*
8. *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
9. *Continuous attention to technical excellence and good design enhances agility.*
10. *Simplicity – the art of maximising the amount of work not done – is essential.*
11. *The best architectures, requirements, and designs emerge from self-organising teams.*
12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.* (Agile Alliance, 2002)

In addition to the general discussion about Agile Manifesto principles, we will analyse each principle from the different aspects of software development in the end of each subchapter. We will construct a framework for this analysis. Firstly, we will take

following phases or tasks of software development into consideration: analysis and requirements gathering, design and architecture, implementation, testing and project management. Similar division of phases or tasks can be found in the waterfall model (excluding project management) on page 17 and in Pressman's definition of tasks related to software engineering methods on page 14. We have excluded maintenance from our framework because Agile Manifesto and agile methods do not directly address it. However, maintenance is not neglected in Agile Manifesto; if architecture design and implementation have been successful, also maintenance will be easier.

Secondly, we will broaden our framework and study principles from job satisfaction aspect. Mumford and Beekman (1994) define job satisfaction as

> *[...] a good fit between what a person does and has in his or her job and what he or she ideally wants to do and have. Most people want the following: to use the knowledge which they posses and to increase this; to get a sense of achievement from work; to have access to resources which enable them to work efficiently and effectively; to have an element of personal control so that they can take decisions and make choices, and to have a well designed job that provides the right mix of interest, variety and challenge.* (Mumford & Beekman 1994, 132)

Thus, with job satisfaction we refer to the motivation of employees, to the satisfaction that they get from their work, to the possibilities to be innovative and to develop oneself. The analysis of Agile Manifesto principles based on the framework of different aspect of software development (analysis and requirements gathering, design and architecture, implementation, testing, project management and job satisfaction) is summed up in TABLE 2 on page 76.

4.2.1 Principle 1

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

Delivering working software, at least parts of the planned system, to the customer at an early phase of the development project and regularly during it, will bring several benefits. Firstly, it enables the development team to get feedback on the system under development, and the feedback can further guide the next steps of the project to the right direction (Cockburn 2001, 218). The incremental and iterative

development approaches support this principle. Every now and then people need confirmation that they are heading in the right direction according to DeMarco and Lister (1987). Besides getting reassurance from the managers, customer feedback can act as such confirmation. Also organisations, not only individuals or development teams, need similar confirmation to reassure that a project is on the right track. The role of such confirmation is emphasised when the development project is large in size and financially significant. (DeMarco & Lister 1987, 152)

Secondly, early delivery of software also convinces the customers, because they can evaluate how their funding has been used. "Valuable" software refers to software, or parts of it, that are crucial to customer's business and are of high priority in development project plan. Early delivery can also guarantee revenue stream to the developing company. (Cockburn 2001, 219) Dividing the software to logical entities that can be delivered to the customer can also be problematic, see principle 3.

Glass (2001, 14) disagrees with this principle and states that at least the outcomes of the very early state of the development work should not necessarily be delivered to the customer. He means for instance test results of the early versions of the product, which might not have much relevance to the customer. It can also be questioned whether using the draft versions of the product would facilitate the development process, because it sets high expectations on customer's technical knowledge and understanding. If the customer representative is not a very technically oriented person, it is not probable that this principle could be fully followed in the development project. However, Glass (2001) acknowledges that working software can and should be used in communication and planning, especially among the members of the development group.

In our view, principle 1 strongly supports the analysis, design, and implementation phases as it reflects iterative software development. Early deliverables can be used when further defining requirements, architecture and programming solutions. Also project management is assisted, as deliverables can guide tracking the current state of the project and guide in planning the project further.

4.2.2 Principle 2

> *The hardest single part of building a software system is deciding precisely what to build. [...] the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements. For the truth is, the client does not know what he wants.* (Brooks 1987, 17)

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

As mentioned before, software requirements tend to change frequently and radically as the development effort proceeds. Principle 2 suggests that changes should be welcome to fulfil customer's wishes and support customer's business. Different agile methods offer different ways to react to the changing customer requirements. Delivering running software early and iteration are some ways Cockburn (2001, 218) mentions. We will return to iteration with the third principle.

However, Agile Manifesto is by no means the first or only context where reacting to changing requirements is discussed. For instance, Microsoft uses an approach known as "synchronise and stabilise", which helps to react to changes:

> *"Synchronise-and-stabilise" process moves away from rigid, sequential approaches to product development. It allows engineers to make a lot of changes in their designs until late in a project, while still keeping individuals synchronised and the evolving product more or less stabile.* (Cusumano & Selby 1998, ix)

Synchronise and stabilise is used in software development of a very complex product. According to Cusumano and Yoffie (1998, 60), "the goal is to balance an almost hacker-type flexibility and speed with professional engineering discipline". Several teams work separately on features of a single product (or product family) and make the development and changes independently. In the end of the day different code components of each feature are brought together as a whole product and the resulting "daily build" is then compiled and tested. This is done to keep the whole product entity synchronised so that the daily-modified code components will function properly together. In addition, the build should become increasingly stable during the process as

more tests are run on the interdependencies of the different components. (Cusumano & Selby 1998)

However, even when reacting to changes on one way or another, some decisions have to be made at quite early phase (for instance decisions concerning the used technologies, fundamental architecture solutions or programming environments). As we can see from the citation below, the requirements are frozen and no changes are implemented after a certain stage of the development process to get a whole, final product completed.

> *Far be it from me to suggest that all changes in customer objectives and requirements must, can, or should be incorporated in the design. Clearly a threshold has to be established, and it must get higher and higher as development proceeds, or no product ever appears.* (Brooks 1975, 117)

Thus, the market environment and consumer behaviour might change very quickly and companies need to be able to respond to those situations. However, if the developed software is very large and has for instance high safety requirements, it should be doubted whether this particular principle is applicable.

Glass (2001, 15) points out that realistic cost and schedule impacts should be required for all changes during the project, and that data should guide the decision-making. Glass (2001) interprets this principle so, that the development projects belong to the customer and they should be able to make changes but in the limits of agreed and to be agreed costs and schedules. If the customer thinks that implementing some changes is worth of investing a lot of time and money, they have the power to make such a decision.

According to Fowler (2001), the agile setting requires a certain kind of relationship with the customer, especially concerning pricing. Traditionally customers are apt to have a fixed-price for what they order. However, agile development and unstable requirements imply that adaptive pricing should be used. (Fowler 2001) Thus, reacting to changing requirements is essential, and that should be taken into account in the business models and pricing strategies.

Principle 2 can either facilitate different development phases or make them more complicated and difficult. Responding to changes should be done to satisfy the

customer. However, radical changes especially in the end of development project can result in problematic situations in analysis, design, implementation and testing.

4.2.3 Principle 3

> *Enthusiasm jumps when there is a running system, even a simple one. Efforts redouble when the first picture from a new graphics software system appears on the screen, even if it is only a rectangle. One always has, at every stage of in the process, a working system.* (Brooks 1987, 18 on incremental development)

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

The third principle refers to the length of development cycles. Employing short cycles or increments offers several advantages. Firstly, Cockburn (2000) argues that increments that last for more than four months reduce the possibility to repair the working process. With short increments the process itself gets tested and with rapid feedback it can be repaired quickly. Secondly, the requirements management becomes more efficient, as the requirements for the product can be tested and altered quickly. (Cockburn 2000, 32) Short increments and a possibility to get feedback after each cycle also support ideas presented in chapter 4.2.1 about how individuals need acknowledgement that they are doing right things.

Highsmith (2000) argues, that as the competitive environment tightens up by becoming more extreme, iteration (i.e. building, trying, succeeding, failing, rebuilding) can be a solution for succeeding in product development (Highsmith 2000, 23). Iterative and incremental development can be held as the main paradigms for agile software development.

Iterative development was presented in the form of spiral development paradigm in chapter 3.1.3. To be more precise,

> *Spiral model is a model of the software development process in which the constituent activities, typically requirements analysis, preliminary and detailed design, coding, integration, and testing, are performed iteratively until the software is complete. (IEEE std 610 1991, 189)*

Incremental development then again can be defined as follows:

> *A software development technique in which requirements definition, design and implementation, and testing occur in an overlapping, iterative (rather than sequential) manner, resulting in incremental completion of the overall software product. (IEEE std 610 1991, 107)*

Karlsson et al. (2000, 650) add a feature aspect to the definition:

> *Incremental development is the delivery of the functionality in increments with a well defined subset of the functionality (set of features) in each increment. These increments are planned on the project level. The frequency of increments can be rather different, down to one week for smaller projects.*

Incremental development can, however, be problematic. Dividing the software into pieces (increments) that can be developed independently and be integrated later on, and division to pieces that form logical entities and functionalities can be difficult. Efficient division sets high requirements for design, architecture and planning, and often is easier said than done.

The concept of *daily build* has been used for instance by Cusumano and Selby (1998) to handle the incremental development and synchronisation and stabilisation mentioned above. According to Karlsson et al. (2000, 649-650), daily build is:

> *[...] a software development paradigm that originated in the PC industry to get control of the development process, while still allowing the focus on end user requirements and code. [...] Daily build is the team/individual controlled delivery of code to a main codeline which is built and tested daily. Any functionality of the delivered code is completely controlled and used by the team/individual. The daily test only covers the functionality that existed in the system at the last stable baseline.*

Hence, this principle suggests that if the development cycles are short enough, the feedback gained after each iteration will guide the product requirements and the development process to be more suitable and precise. In addition, short development cycles can set enjoyable challenges for the employees if they are tight but not impossible to achieve. However, employees are likely to slip on their effort if they know that the deadline is impossible to meet. (DeMarco & Lister 1987, 137-138) Thus, setting a tight but realistic deadline can significantly enhance progress and efficiency of software development. When estimating the length and deadlines of a suitable time cycle, the developers should remember that the users might not be able to absorb to changes more often than every three months (Cockburn 2001, 220).

The feedback gained after each increment can strongly facilitate analysis, design and implementation. In addition, the feedback information can be used in testing and short increments can help in project planning as the progress of the project can be easily tracked according to the feedback.

4.2.4 Principle 4

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

Although often using the term *business people* in the articles and books concerning agile software development, the agilists have not explicitly defined the term. In this study we use it in two different meanings. Firstly, it can refer to a customer who is paying the developing company to make a tailored project or a software product. It should be noted, however, that especially in the case of mass-market COTS products end-users cannot be regarded as business people. Although end-user feedback should be used, and their wishes and needs should be respected, the close collaboration that this principle is suggesting is quite unlikely to take place.

Secondly, business people can refer to people or departments with business knowledge within the developing company, for instance marketing department, or it can refer to the parent corporation in the case of a corporate venture. Such business people can be regarded as stakeholders of the development project, and they might have certain requirements or needs that have to be fulfilled. For instance, the product needs to be in line with the corporate business strategy. There are also other possible viewpoints from which to consider business people; they might for instance refer to venture capitalists but we will exclude them from this research and concentrate on the two above-mentioned cases.

Business people are seldom working on development projects intensive enough, and as a result the outcome of the project might not meet their expectations and needs. However, if business people and developers would be working tightly together throughout the project, communicating and sharing ideas frequently, business people would be more satisfied with the outcomes. The word "daily" does not necessarily mean

literally every day but it rather refers to turning points, where important decisions are discussed (Cockburn 2001, 221).

If making a rough classification, there are two kinds of decisions to be made in the software development project: business decisions and technical decisions. According to Fowler (2001), business people (managers to be more precise) and developers should make the needed decisions together. Managers should give more responsibility to developers to make the technical decisions because they have expertise in the particular field. However, developers should also have the adequate business knowledge, so that they can use it to develop software that will meet the requirements set by the business environment. (Fowler 2001) This is in line with the observation of Baskerville et al. (1992) mentioned in chapter 3.2 that the role of developers is changing towards business orientation from pure technical orientation. Their suggestion about using rapid, small-scale information systems development (Baskerville et al. 1992, 248) sounds very similar to the ideas of agile software development.

Agile Manifesto assumes that business people and developers are different people but Microsoft aims to combine both qualities in one person. Microsoft puts a high emphasis on both business and technical understanding, especially when it comes to management. This is how Cusumano and Selby (1998, 65) describe the situation:

> *It is common practise for managers in all the functions to continue doing their functional jobs after becoming senior managers. Even development managers of large groups such as Word and Windows NT still spend one-third to one-half of their time writing code. The rationale is that managers need to remain directly involved in the technology in order to make the right decisions and solve problems that occur in their groups.*

Because technical skills and knowledge soon become obsolete in software field (Fowler 2001), this practise helps the managers to stay on top of technical developments. Nevertheless, such practise can lead to a situation where managers are capable of managing technology but not capable of managing people. That has also been a problem at Microsoft, especially in the middle management. (Cusumano & Selby 1998, 67)

Basically principle 4 is trying to prevent the problems caused by the communication gap between developers and business people by suggesting that they should work

collaboratively. Cockburn (2001, 221) summarises, "the longer it takes to get information to and from the developers, the more damage will occur to the project." This is valid both for customer-development group and stakeholder-development group settings described above.

Collaboration of business people and developers can strongly support analysis phase of software development because the requirements for the software are discussed from both business and technical point of views. In addition, project management is strongly facilitated by this principle because the project can be planned and managed cooperatively to meet the business objectives.

4.2.5 Principle 5

> *A positive [organisational] climate is one that supports and encourages intrapreneurs and risk-taking behaviour; where new product successes are rewarded and recognised (and failures not punished); where the team efforts are recognised rather than individuals; where senior managers refrain from "micromanaging" projects and second-guessing team members [...]. Idea submission schemes (where employees are encouraged to submit new product ideas) and open project review meetings (where the entire project team participates) are often facets of a positive climate.* (Cooper 1996, 11)

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

The agile approach places more emphasis on people factors, like amicability, talent, skill, and communication, than to project or process factors. Individual competency is regarded as a critical factor in agile project's success. Collaboration and communication further enhance the utilisation and development of individual skills, because individuals learn from each other more when working in an agile mode than when working alone. They also get better results for instance in problem solving when they work as a team. (Cockburn & Highsmith 2001)

Competent personnel assists software development process, or might even be one of the key factors for a successful software development project. According to Boehm (1981, 666), "personnel attributes and human relations activities provide by far the largest source of opportunity for improving software productivity". In order to maintain or increase productivity, companies must motivate their personnel and support their

careers and personal growth as professionals. When it comes to tasks and work division, Boehm (1981) mentions that for instance working on too small fragments of large software work is not motivating, neither is maintenance.

Brooks (1987, 18) suggests that software organisations should acknowledge that great designers are as important to its success as great managers are, and that they should be similarly supported and rewarded. Cockburn (2000) has noticed that experienced developers can speed up the project significantly compared to having less experienced developers. But he points out that if a team consists of a few experienced and several inexperienced developers, the experienced will have to spend much of their time helping the inexperienced. (Cockburn 2000, 32) Hence, if the number of inexperienced developers is too high compared to the number of experienced developers, the advantage of using experienced employees will vanish.

Fowler (2001) reminds, that when wanting to hire and retain good people, it is essential to recognise that they are competent professionals. He does not specify what he means by this but such recognitions might refer to rewarding mechanisms, respecting employees' competence and personalities, and giving them proper working conditions. Brooks (1987) believes that great designers can "grow". If a company wants to raise and utilise great designers, or other software professionals, it should first of all identify the potential individuals, then give them career guidance and education, and an opportunity to interact with their peers. (Brooks 1987, 18)

As Fowler (2001) and Brooks (1987) pointed out, the role of management is essential in software development. Next we will further discuss management issues from the agile point of view.

Managing in the agile context

> *...good management can promote both an efficient, well-coordinated software process and the high levels of staff capability, motivation and teamwork, which lead to outstanding software productivity.* (Boehm 1981, 675)

According to Cockburn and Highsmith (2001, 132), "agile companies practise leadership-collaboration rather than command-control management. They set goals and constraints, providing boundaries within which innovation can flourish." Agile

approaches want to abandon the authoritarian manager and give more decision power to the team, as the following citation states:

> *All too often, project teams are given the ultimate accountability for product delivery, while staff groups – project offices, process management, data administrators – are given the decision power. In order to innovate and react to change, agile teams reflect a better alignment of accountability and responsibility. It is once again an emphasis, at the team level, on competency rather than process.* (Cockburn & Highsmith 2001, 132)

Wegener (2001) defines the characteristics of a manager in agile projects: he or she should be very attentive, technically well educated, socially fit, tolerant and patient. Finding such a manager can, however, be quite challenging. As Brooks (1987, 18) states, "great designers and great managers are both very rare". Not every manager has the personal skills and capabilities of becoming a collaboration and team facilitator. Although collaboration and the feeling of working in a bonded team are very critical to projects' success, expecting such capabilities from everybody is unrealistic. Instead, those are personal characteristics that can be learned to some extent but still are very dependent on personality.

Managers are often outsiders in the development teams, they manage the team from a higher hierarchy level. However, well functioning teams should not need very much leadership from above but they should rather be given a possibility to make decisions within the team. According to DeMarco and Lister (1987), occasional leadership within good teams strives from different individuals who have competence in particular areas, and who start to lead the team for a while when their area of expertise is needed. This way there is no permanent leader, and the team is working in a network instead of a hierarchy. (DeMarco & Lister 1987, 155)

"Trusting them to get the job done" refers to the trust that managers should have towards their employees. Managers should let their employees make decisions and allow them to make errors. Thus, managers should trust their employees and give them responsibility. Lack of trust may poison the team spirit and hinder the formation of a bonded, close team (DeMarco & Lister 1987, 133-135). As an interviewee in (Cockburn 2001, 221) stated "We hire good people, give them the tools and training to get their work done, and get out of their way."

Staffing and recruitment are not directly addressed in Agile Manifesto but as this principle suggests, people working in the agile mode should not only be motivated but also skilled, educated and experienced. Boehm (1981) proposes some issues that should guide the selection of suitable personnel. First of all, companies should hire top talents. Although the most experienced workers are often the most expensive workers, their productivity can be exceptional. (Boehm 1981) Recruiting top talents, both managers and other employees, is also a strategy of Microsoft. However, Microsoft also hires graduates from the best universities and trains them to meet their particular needs. (Cusumano & Selby, 1998)

Secondly, according to Boehm (1981), skills and motivation of people should be taken into consideration. Instead of promoting a brilliant programmer to be a manager and burden him or her with bureaucratic tasks, they should also be given a possibility to become a senior programmer or such. Hence, the employees should be given recognition and promotions. However, when giving promotions the preferences and motivation of the employees should be taken into account. Boehm also reminds that due to the rapid pace of technology development, a chance for updating knowledge and training new skills should be provided. (Boehm 1981) In addition, DeMarco and Lister (1987) emphasised training among other things to substitute for rigorous, traditional methods (see chapter 3.2). As a practical example, Microsoft pays attention to the career development of their employees, and has developed rewarding and promotion systems that encourage employees to become better professionals (Cusumano & Selby 1998).

If employees are skilled and motivated, it naturally has a positive effect to all aspects of software development. According to our view, this principle mainly facilitates job satisfaction of employees. We will return to managers and teams in discussion of Principle 11.

4.2.6 Principle 6

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

According to Fowler (2001), frequent and open communication is necessary for the existence of an agile team. One of the main reasons for this is that the whole development team has to be ready to adapt to rapidly occurring changes, and therefore everybody should get instantly advice of the changes. (Fowler 2001) The sixth principle takes communication issues further by emphasising face-to-face communication. Face-to-face communication decreases the needed amount of documentation because the information and knowledge are transferred in personal discussions. In addition, face-to-face communication facilitates learning, when people are discussing and solving problems together. Face-to-face communication also enables instant feedback. If some questions or comments rise, those can be discussed immediately. Face-to-face conversation does not only refer to "chit chat" but such conversations can also take place when two people are discussing a piece of code while programming it. Although informal communication is emphasised (Cockburn 2001), formality can also be beneficial. Formal written proposals used in meetings help to focus on relevant issues and accelerate decision making (Brooks 1975, 67). Brooks (1975, 66) thinks that in addition to informal talks, formal meetings are in place: "Needless to say, meetings are necessary. The hundreds of man-to-man consultations must be supplemented by larger and more formal gatherings."

Frequency of communication is also important. Brooks (1975) favours regular weekly meetings. Several advantages can be gained from those meetings. As the group meets each other often, everyone is constantly up-to-date. Problems are presented and discussed within the whole group, and better solutions can be found when several people are working on them. (Brooks 1975, 67)

Employees who are expected to work tightly together should be given a possibility to be located close to each other to create an environment that facilitates informal face-to-face communication. Working in one location and encouraging informal communication is highly emphasised for example by Bill Gates at Microsoft (see Cusumano & Selby 1998, 285). Team closeness improves communication as the team can constantly share

information (Cockburn 2000, 31), whereas physical separation withholds casual interaction and hinders team bonding (DeMarco & Lister 1987, 136). If a team is dispersed in two or more rooms in the same building, there are still ways to achieve a sense of community. For instance, installing cameras on the team members' workstations and using on-line chat to instant queries can assist creating affinity. For teams that are dispersed in different locations this seems to be more difficult. (Cockburn 2000, 31) This principle is relatively easy to implement within a small development team but if a development team is large or distributed in different locations, even in different countries, other ways to facilitate efficient and effective communication have to be found.

Face-to-face communication is the best way to transfer tacit knowledge that plays a very important role in agile methods. Different types of knowledge and transferring them have been studied for instance by Nonaka (1995). According to Cockburn (2002), projects can get half of their agility from encouraging informal communication among team members instead of relying heavily on the external knowledge of group's organisational memory. He states that in fact teams often rely on tacit knowledge more than they even acknowledge. (Cockburn 2002, 6-7) Eischen (2002, 36) characterises agile development approach as craft-based approach and traditional methods as engineering approach. Crafts and skills are included in tacit knowledge as technical elements but tacit knowledge also includes cognitive elements, such as mental models and perspectives (Nonaka 1994, 17).

Nevertheless, relying on tacit knowledge can be quite problematic. Boehm (2002, 66) points out important issues concerning this problem:

> *When the team's tacit knowledge is sufficient for the application's life-cycle needs, things work fine. But there is also the risk that the team will make irrecoverable architectural mistakes because of unrecognised shortfalls in its tacit knowledge. Plan-driven methods reduce this risk by investing in life-cycle architectures and plans, and using these to facilitate external-expert reviews.*

Thus, agile methods assume that some people in the development team are highly professional and capable individuals. Nevertheless, practically it sets high, sometimes even unrealistic, expectations on employees. Constantine (2001, 68) states:

> *There are only so many Kent Becks[7] in the world to lead the team. All of the agile methods put a premium on having premium people and work best with first-rate, versatile, disciplined developers who are highly skilled and highly motivated. Not only do you need skilled and speedy developers but you need ones of exceptional discipline, willing to work hell-bent-for-leather with someone sitting beside them watching every move.*

Thus, emphasis on tacit knowledge can be fruitful but also very risky. As a result, employees working in agile mode are assumed to have extraordinary skills, strong tacit knowledge and be disciplined to keep the development agile. Such expectations, however, might be unrealistic. Face-to-face communication is emphasised in Agile Manifesto but transferring and sharing face-to-face expressed information over time is naturally impossible. Face-to-face communication brings many benefits but it cannot be the only way to share, collect and store information.

Face-to-face communication can assist in analysis, design and implementation as problems can be discussed and solved in rich communication mode with constant feedback. Testing, project management and job satisfaction can also be positively affected when following this principle.

4.2.7 Principle 7

7) Working software is the primary measure of progress.

This principle suggests that working software should be used as the measurement of progress. Other measures of progress can also be used but working software might tell more about the current situation than reports or such. (Cockburn 2001, 220) Principle 7 is closely related to principles 1 and 3 and again emphasises the importance of working software delivered to the customer at early stage.

---

[7] Kent Beck is one the inventors of Extreme Programming, in many articles considered as software development "guru".

Measuring is an important part of software development. Next, we will briefly explain software measurement and different metrics. Pressman (1994, 45) motivates the needs for software measurement as follows:

- to indicate the quality of the product
- to assess the productivity of the people who produce the product
- to assess the benefits (in terms of productivity and quality) derived from new software engineering methods and tools
- to form a baseline for estimation
- to help justify requests for new tools or additional training.

Pressman (1994) categorises software metrics into six groups. FIGURE 8 illustrates the different metrics. *Productivity metrics* focus on the output of the software engineering process (for example lines of code per person-month), *quality metrics* provide an indication on how closely software conforms to implicit and explicit customer requirements (software's fitness for use), and *technical metrics* focus on the character of software (for example logical complexity, degree of modularity) rather than the process through which the software was developed. *Size-oriented metrics* are used to collect direct measures of software engineering output and quality (for example lines of code, number of errors, pages of documents). *Function-oriented metrics* provide indirect measures of software and the process by which it is developed, and *human-oriented measures* collect information about how people develop computer software and about human observations about the effectiveness of tools and methods. (Pressman 1994, 46)

Seventh principle of Agile Manifesto emphasises the measurement of progress. This particular point refers to the output of the development process (productivity metrics) but also other measurement aspects are included in Agile Manifesto. Quality metrics (how the customer requirements are fulfilled) are addressed in the first and second principle; technical metrics (such as architecture and technical excellence) are addressed in the ninth and eleventh principles. Size-oriented metrics are to some extent covered in principle nine, and human-oriented metrics are related to the twelfth principle and improvement of the development process. Hence, several different measurement

considerations are actually included in Agile Manifesto, although no clear metrics are provided. It should be noted that metrics and measurement are more important to large projects. For small projects in general it is not necessary to have detailed metrics, working code can provide enough measurement information for such projects.

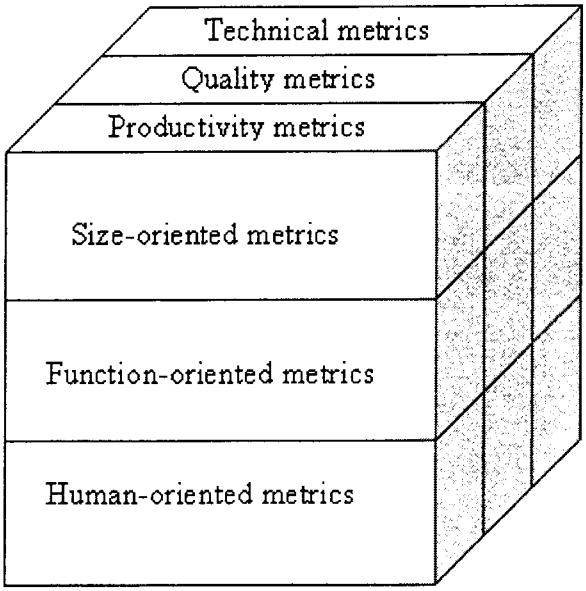

FIGURE 8 Metrics categorisation (Pressman 1994, 46)

Pressman's view on measuring is software engineering oriented and it does not take business point of view in account. Rifkin (2001) presents a complementary view to measuring software. He argues that traditional measurements supporting increased quality, increased programmer productivity and reduced costs only fit organisations that focus on operational excellence. *Operational excellence* as a business strategy refers to companies that usually offer a small and limited number of products but deliver them excellently and at a competitive price (for example McDonald's and Federal Express). Other strategies are customer intimacy and product innovativeness. *Customer-intimate* organisations offer a long, custom-made product menu for their customers (an example of such could be an organisation providing financial-services). *Product-innovative* organisations tend to maximise the number of turns they get in the market (for example Intel, Microsoft and Bell Labs). They measure their success by the number of new product introductions and the number of patents. Usually a company focuses on one of the three, nevertheless, it should have the essentials of all of them. (Rifkin 2001, 41-42)

Next we present which issues should be measured in different types of organisations and why they should be measured.

Important measures in the customer-intimate organisations according to Rifkin (2001, 44) are:

- Measures of flexibility to expand the product menu.
- Measures of comprehension and understandability about customer's strategy and business to estimate maintainability and changeability of the product menu in the future.
- Customer's architecture (counts of architecture checks and violations) in terms of how well can new components be added to it, and how many different interfaces there are.

Important measures in the product-innovative organisations according to Rifkin (2001, 44) are:

- Features are the most important deliverables – not quality, reliability, cost or flexibility.
- Quality goals should focus on thresholds, benchmarks and especially time-to-market. Some kind of measures for those can be found when comparing how the quality stacks up against those of the competitors.

Especially the product-innovative organisations that use new technologies and make innovative products could benefit from agile software development. Measurement issues described by Rifkin (2001) could add to the seventh principle, stating that also the number of innovative features is an important measure.

In our opinion using working software as a measurement of progress mainly supports project management as the current state can be more easily estimated.

4.2.8 Principle 8

> *Nobody can really work much more than forty hours [a week], at least not continually and with the level of intensity required for creative work.* (DeMarco & Lister 1987, 15-16)

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

Principle 8 has two sides: one relates to social responsibility, the other to project effectiveness. Employees should not be putting in long hours for too long. A tired and stressed staff is not an agile staff, because tired and stressed employees cannot make good results. Thus, projects should be organised in a way that everyone involved would have reasonable working hours enabling them to stay alert and engaged. (Cockburn 2001, 222) Workaholics are people who work extravagant hours. Their effectiveness probably declines and quality of their personal lives is likely to get worse. (DeMarco & Lister 1987, 16) Agile software development does not approve with workaholics. We did not find definitions for *sponsor* or *user* to whom this principle also refers. We assume that a sponsor is a synonym for business people (see Principle 4) but including users in this principle seems quite awkward.

Organisations often try to improve their productivity by pressuring people to put in more hours in work. Working some extra hours may be justified to meet a forthcoming deadline but in the long term it will not pay off. Compensatory "undertime" frequently follows overtime because people are catching up with their personal lives, and thus working less to balance the situation. (DeMarco & Lister 1987, 17)

Sustainable pace mostly affects job satisfaction. Sharp and not stressed employees naturally enhance software development more than workaholics.

4.2.9 Principle 9

9) Continuous attention to technical excellence and good design enhances agility.

If the design of the object system is well encapsulated and tidy, it is easier to make changes to it, which enhances agility. Designers should produce good designs from the very beginning and keep updating and reviewing them along the process. Iterative and

incremental development supports this. (Cockburn 2000, 32; Cockburn 2001, 222) Additionally, we have included testing as a factor that can improve the quality of the design.

According to Glass (2001), this principle is about continuous improvement of the software, which mainly refers to maintenance that consists of fixing errors but also refactoring (improving the code). He states that this principle is one of the many in Agile Manifesto that derive from programmers' point of view. For instance refactoring is often done by programmers to polish a working piece of code, which then again is seldom approved by the managers, because it can be considered as a waste of time and money. (Glass 2001, 16) Refactoring is often done by the programmers to fulfil their personal quality requirements:

> *The builders' view of quality, on the other hand, is very different. Since their self-esteem is strongly tied to the quality of the product, they tend to impose quality standards of their own. The minimum that will satisfy them is more or less the best quality they have achieved in the past. This is invariably a higher standard than what the market requires and is willing to pay for.* (DeMarco & Lister 1987, 20)

Thus, improving the design and code to a certain extent will increase agility but unnecessary refactoring is a waste of resources, at least from managers' perspective.

Testing issues have a very important role in technical excellence in our opinion. Testing in all phases of the development is also highlighted at Microsoft. The individual code components are tested before they are added to the big build, and additionally several other kinds of tests are run during the development. Microsoft has an interesting practise called "testing buddy". Every developer has a testing partner, with whom he or she works in very close cooperation. Testing buddies test the code produced by their development partner, and suggest improvements to the code, or sometimes even to the specification of feature. Testing buddies also write test cases based on the specification while the developer is programming a certain functionality. (Cusumano & Selby 1998, 297) This practise will most probably enhance technical excellence and good design, because at least two people are reviewing a certain piece of code. We will return to testing from XP point of view in more detail in chapter 5.

Technical excellence and quality can also be analysed from the human point of view. According to DeMarco and Lister (1987), technical excellence and quality can also facilitate the team closeness and personal satisfaction. Expecting a team to deliver no less than perfect work binds the team together, whereas asking a team to deliver lower quality product, considered good enough, may undermine employees' self-esteem and enjoyment of their work. For instance trying to deliver product in less time is often done at the expense of product's quality. (DeMarco & Lister 1987, 137-152).

Making the design and implementation better and better (i.e., refactoring) will strongly facilitate design and implementation, and support analysis and testing. Iterative approach enables utilising design and implementation as input for the analysis of the next iteration round. High-quality code then again facilitates maintenance. In addition, if the employees are asked to deliver products of high quality, it can increase their well-being as they can use and demonstrate their personal technical expertise.

4.2.10 Principle 10

10) Simplicity -- the art of maximising the amount of work not done -- is essential.

This principle emphasises simplicity. The main point is that developers should only implement features that have been agreed upon with the customers, nothing more. This principle acknowledges that in programming it is more difficult to make simple design than cumbersome solutions. Yet it is worth noticing that the notion of *simple* is very subjective, and giving practical guidelines what is simple and how to accomplish that is impossible. (Cockburn 2001, 222-223) Glass (2001, 16-17) warns that simplicity should not mean neglecting design by starting programming as soon as possible. Thus, simplicity and maximising the work not done should not exclude highly important phases of the development. Simplicity is further discussed in chapter 5 on page 96.

The requirements are hardly simple in the beginning but as the project goes on and analysis will become clearer, this principle supports the analysis phase. If the design and implementation are simple, testing is easier and more effective. This principle most strongly supports the design of high-quality architecture and implementation.

4.2.11 Principle 11

> *Self-organising teams are not leaderless teams; they are teams that can organise again and again, in various configurations, to meet challenges as they arise.* (Cockburn & Highsmith 2001, 132)

11) The best architectures, requirements, and designs emerge from self-organising teams.

Cockburn (2001) points out that the members of Agile Alliance did not totally agree on the definition of *self-organising teams* and they discussed the word *emerge* too. His explanation of this principle is a little bit vague but the bottom-line is that development teams should allow architectures, requirements and designs to adjust over time, become better and more accurate gradually, and not to fix them too accurately in the beginning. (Cockburn 2001, 221) Glass (2001) is of different opinion concerning the focus of this principle. He believes, "[...] this principle is not about architecture, or requirements, or design. It is about organisational approach, self-governing teams." (Glass 2001, 17) Since we have not found a more precise definition for the architecture side of this principle, we will concentrate on Glass' interpretation of this principle. Finding motivated and skilled individuals (see Principle 5) is not straightforward; neither is building a healthy organisation, in which the team chemistry can flourish. We will explore and explain the concept of a team and especially the concept of an agile team. We will also examine how a group of people appointed to take care of a certain task together might start to form a close team and what it can mean to the success of a development project.

Agile team

"Agility requires that teams have a common focus, mutual trust, and respect; a collaborative but speedy, decision-making process; and the ability to deal with ambiguity." (Cockburn & Highsmith 2001, 132) As mentioned before, individuals are highly emphasised in the agile approach but so is teamwork. The concept of self-organising team is emphasised in Nonaka's (1994) research of organisational knowledge creation. Self-organising teams have an important role in creating knowledge according to Nonaka. Knowledge creation can happen in two ways:

> *The self-organising team triggers organisational knowledge creation through two processes. First, it facilitates the building of mutual trust among members, and accelerates creation of an implicit perspective shared by members as tacit*

> *knowledge. The key factor for this process is sharing experience among members. Second, the shared implicit perspective is conceptualised through continuous dialogue among members. [...] The two processes appear simultaneously or alternatively in the actual process or knowledge creation within a team.* (Nonaka 1994, 24)

Thus, self-organising team has a significant part in knowledge transfer and learning.

DeMarco and Lister (1987) present the concept jelled team. According to them, "a jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts". (DeMarco and Lister 1987, 123) Often heard term for the same phenomenon is synergy. Covey (1994, 262-263) defines synergy as follows:

> *What is synergy? Simply defined it means that the whole is greater than the sum of its parts. It means that the relationship which the parts have to each other is a part in and of itself. It is not only a part but the most catalytic, the most empowering, the most unifying, and the most exciting part.*

Usually a group of employees are gathered around to a common goal, or rather assigned to accomplish it. If the group members are committed to a common goal, it starts to bring the members closer to each other, and "jelling" to form a team begins. Setting goals is not always easy; the corporate values and goals do not always seem motivating for the employees, instead they might find their inspiration elsewhere. Once a team begins to become tighter, it is likely to become more successful, and people can be more productive when they work in a well collaborating and close team than working solo. A jelled team is characterised for instance by low turnover of workers. A strong sense of identity prevails in such teams; in practice they may have colourful names and share inside jokes. Having a sense of eliteness, feeling that one is part of something unique, is also characteristics of a jelled team. A jelled team has a strong ownership of the product. Last, "the final sign of a jelled team is the obvious enjoyment that people take in their work." (DeMarco & Lister 1987, 124-127)

DeMarco and Lister (1987) list several issues that can facilitate or hinder the formation of a jelled team. Firstly, they suggest that employees should be allowed to focus on one project at a time. Being a member of many projects and groups simultaneously is very difficult. It requires employees to communicate and interact with people of all those different projects. It reduces the team efficiency because the members might be

occupied with issues of one project when they should concentrate on another project. Secondly, allowing and encouraging heterogeneity can also assist in strengthening the team. Addressing that not everybody needs to be the same can be important for the team members. Further issues that can hinder team formation are for instance lack of trust, bureaucracy instead of concentration on the actual work, or unsuccessful office arrangements. (DeMarco and Lister 1987) Boehm (1981, 671) suggests, that selecting people who will complement and harmonise with each other also facilitates team formation, although it can be very challenging to pick the right people in technical and psychological aspects.

Sometimes managers may feel threatened and insecure because of jelled teams, as the teams might be so tight, that the managers are totally excluded from them. Therefore, the managers might be willing to separate teams. As a reason for the separation, the managers may argue that if the team members feel very special and being "elite", they may consider others as being less important, and might be unwilling to cooperate with other teams. However, DeMarco and Lister (1987) suggest that a successful team should not be broken if they function well together. They should at least be given an opportunity to work together in another project. (DeMarco & Lister 1987, 155) This is a very important point, especially from the agile software development point of view. If a particular team has successfully worked together once, they should stay together during the next project as well if possible. It always takes some time before the team members learn to know and trust each other, and this time could be reduced to zero, if the same team would work together. Instead of spending time learning each other's names and working habits, the team could concentrate on the problems of the software immediately. This would be self-organising as Cockburn and Highsmith (2001) understand it.

When it comes to a team or a team member that has not been successful, Boehm (1981) suggests that such people should be transferred to other tasks. By this he emphasises that managers should have the courage to even fire their employees if it is necessary.

It is important to notice, however, that not everybody enjoys working in an agile mode. As Sutherland (2001, 9) states "Not everyone feels comfortable in an open and agile working environment, and some people want to find other jobs and other teams where

the work is done according to rules and processes." This issue might be worthwhile to consider already when recruiting personnel.

Emerging and gradually enhanced architecture mostly supports design and analysis and requirements gathering but also implementation. Additionally, self-organising teams can support job satisfaction but as mentioned, not everybody prefers to work in an agile team, thus this principle can also decrease well-being.

4.2.12 Principle 12

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

It is not only the requirements that constantly change; also the development process is changing over time. The last principle suggests that a development team should meet at regular intervals, for instance once every two weeks and discuss their working habits and tune those habits and methods to be more and more agile (Cockburn 2001, 223). The team will find their own ways of working in the course of time and alter the process to fit their work better. Reviewing the process regularly improves agile and adaptive processes. (Fowler 2001) This approach is facilitated by iterative development. This principle is very project and team specific, and teams should take responsibility of this themselves. (Cockburn 2001, 223)

Although this is an important issue, it can be doubted, however, whether a team will take the extra time to do this. Even though this might improve their outcomes and decrease their workload, sparing extra time for adjusting their behaviour might be impossible.

This principle seems to be targeted to development teams but agility can also be increased on the organisational level. Although organisational agility (see for instance Shafer et al. 2001) is not the focus of this research, the issue will briefly be discussed here. It is naturally important that a development team tunes its performance and the methods that they use during the development project. But it also important to do that in a wider, organisational context. That has partly been addressed in page 49 where the collaboration between business representatives and developers was discussed. Development teams should communicate and work well in an agile mode with other

people and units within their organisation, and constantly improve that. This issue can be broadened to the concept of *organisational learning*. Cusumano and Selby (1998, 327-328) list issues that can be improved or learned in an organisation:

> *Organisations have many opportunities to improve what ever they do: They can reflect on their operations, study their products, listen to customers and encourage different parts of the organisation to share knowledge as well as the results of their separate efforts, such as in designing products and components.*

Thus, organisational learning is improvement of the above-mentioned issues by the employees within their company. Agile Manifesto encourages organisational learning on a team level but it could be expanded to the whole organisation. However, Agile Manifesto does not provide any specific guidelines for how to do that. Some ideas of Microsoft (Cusumano & Selby 1998, 327) could be utilised here, for instance writing post-mortem documents after a completed project (something that is often mentioned but more often left undone), gathering and analysing feedback from the organisation and from the customers, and promoting linkages and sharing of information between different teams and units.

Continuous improvement that this principle suggests is a main driver in software process models, for instance in SW-CMM (see page 44). As SW-CMM is such a wide area, we will not describe it in detail in this thesis. See for instance (Zahran 1998) for further information.

Discussing performance and how to enhance it within a team can facilitate project management. In addition, it can support job satisfaction of employees.

TABLE 2 sums up how the principles of Agile Manifesto support the framework of different aspects of software development (analysis and requirements gathering, design and architecture, implementation, testing, project management and job satisfaction). Character + means that a principle supports the mentioned issue, ++ means strong support, and +/- means that a principle can either support or not support the mentioned issue. An empty cell means that a principle does not address mentioned issues.

TABLE 2 Aspects of software development versus Agile Manifesto principles

Principle / Aspect	1	2	3	4	5	6	7	8	9	10	11	12
Analysis and requirements gathering	++	+/-	++	++		++			+	+	++	
Design and architecture	++	+/-	++			++			++	++	++	
Implementation	++	+/-	++			++			++	++	+	
Testing		+/-	+			+			+	+		
Project management	++		+	++		+	++					+
Job satisfaction					++	+		++	+		+/-	+

To conclude, all the mentioned aspects of software development are covered by Agile Manifesto. In addition to the more traditional aspects such as design and analysis, job satisfaction is addressed in several principles.

4.3 Agile framework

Next we will further analyse Agile Manifesto. We begin by presenting the dependencies of the values and principles according to our view, which is based on the above analysis of each principle.

Individuals and interactions over processes and tools

Interaction and communication between people are frequently addressed issues in agile software development. Principles 1, 4, 5, 6, 8 and 12 reflect this value statement.

Working software over comprehensive documentation

Emphasis of working software can be found in principles 1, 3 and 7. In addition, principles 9, 10 and 11 emphasise technical excellence, simplicity and architecture design that further enhance producing high quality and low-defect software.

Customer collaboration over contract negotiation

This value statement emphasises close relationships between the software development team and the customer. Collaboration and communication are reflected in principles 1, 2, 4 and 8.

Responding to change over following a plan

Ideas concerning responding to change can be found in principles 1, 2, 7, and to some extent in principle 3 too. Principle 12 also encourages active response to change to some extent, although this linkage is not visually presented in the framework. The dependencies of Agile Manifesto values and principles are summed up in TABLE 3.

TABLE 3 Dependencies of Agile Manifesto values and principles

Principle / Value	1	2	3	4	5	6	7	8	9	10	11	12
Individuals and interactions over processes and tools	x			x	x	x		x				x
Working software over comprehensive documentation	x		x				x		x	x	x	
Customer collaboration over contract negotiation	x	x		x				x				
Responding to change over following a plan	x	x	x				x					(x)

To continue our analysis, we have analysed the principles of Agile Manifesto, evaluated their weightings and placed the principles in relation with two dimensions: internal versus external and social versus technical to form a conceptual framework. *Internal* refers to the development team and *external* to the customer (see our definition for customer in Principle 4 on page 56). Nonaka (1994) presented model for organisational knowledge creation, and one part of his model consists of detailed levels of interaction in which knowledge is created. Nonaka (1994, 17-20) distinguishes the following levels: individual, group, organisation and inter-organisation (e.g. suppliers, customers

and distributors). Compared with his model, our dimension for the framework is not that specific: our external dimension refers to Nonaka's inter-organisational level and our internal dimension includes all the other levels Nonaka presented.

The socio-technical approach aims to optimally combine both technology and people to gain efficiency and produce a more humanistic work design (Mumford & Beekman 1994, 53-55). Although socio-technical approach has been used to study for instance manufacturing (Mumford & Beekman 1994), the approach has also been used in studies of information systems development. We find this approach suitable for our analysis. By *social issues* we refer to human well-being, job satisfaction (see definition for job satisfaction on page 50), communication, team building and team spirit. *Technical issues* are related to more technical aspects of software development (analysis and requirements gathering, design and architecture, implementation, and testing). Technical issues are related to the outcome of the development project (software artefact) and social issues are related to the social factors of the development project and to the development process. Other dimensions could have been possible, for instance technology versus business, or dimensions for tacit versus external knowledge (Nonaka 1994, 20), which could be used if the analysis had been more knowledge management oriented. However, these other dimensions are beyond the scope of this study.

We positioned the principles according to their emphasis on these dimensions. However, the authors of thesis were not able to develop a systematic way to measure or value different principles. Instead, positioning of each principle was done intuitively. Based on that analysis the authors of this thesis developed a conceptual framework for agile software development. The framework can be used to get a more organised picture of Agile Manifesto and its focus areas, which guide agile software development in general by giving a higher level view on different aspects of it. The framework is presented in FIGURE 9.

The first principle is related to all dimensions of our framework. The feedback from continuous delivery facilitates development team's internal technical work but also gives the employees reassurance that they are doing the right things, which can support their job satisfaction. As the main objective of this principle is customer satisfaction, the first principle is also external.

We have positioned the second principle to be external and technical because many requirements are likely to be derived from certain standards, thus they are quite technical by nature. On the other hand, the principle was partly positioned in the social section because the market or end-user demands and wishes are seldom technical but rather practically oriented wishes about features or services that would be nice to have. Usually the demands from the markets come with only little attention to the technical implementation.

Principles 3 and 7 emphasise iterative and incremental development, delivering working code and using it for measuring. The principles are quite technical, and are mainly related to internal implementation, although the external parties can use the outcomes to give feedback and monitor the progress. Feedback gives positive reassurance like in principle 1, so these principles are positioned to be partly socially oriented as well.

Collaboration of business people and the development team (principle 4) is mainly social and management related. As business people can be external, we have positioned this principle equally between internal and external axes.

Principles 5 and 12 are mainly for internal team building and for developing working practises of the team. In addition, they are more socially oriented than technically oriented. Principle 6 mostly emphasises internal communication but also communication with external parties. Sustainable pace of principle 8 is one of the key factors in human well-being. Although the principle suggests that developers, business people (sponsors) and users should avoid working overtime, we find that this is mainly development team's internal issue.

Principle 9 that strives for refactoring and technical excellence is positioned to be technical and internal. Additionally technical excellence and producing high-quality might increase job satisfaction, and thus it is also a socially oriented principle.

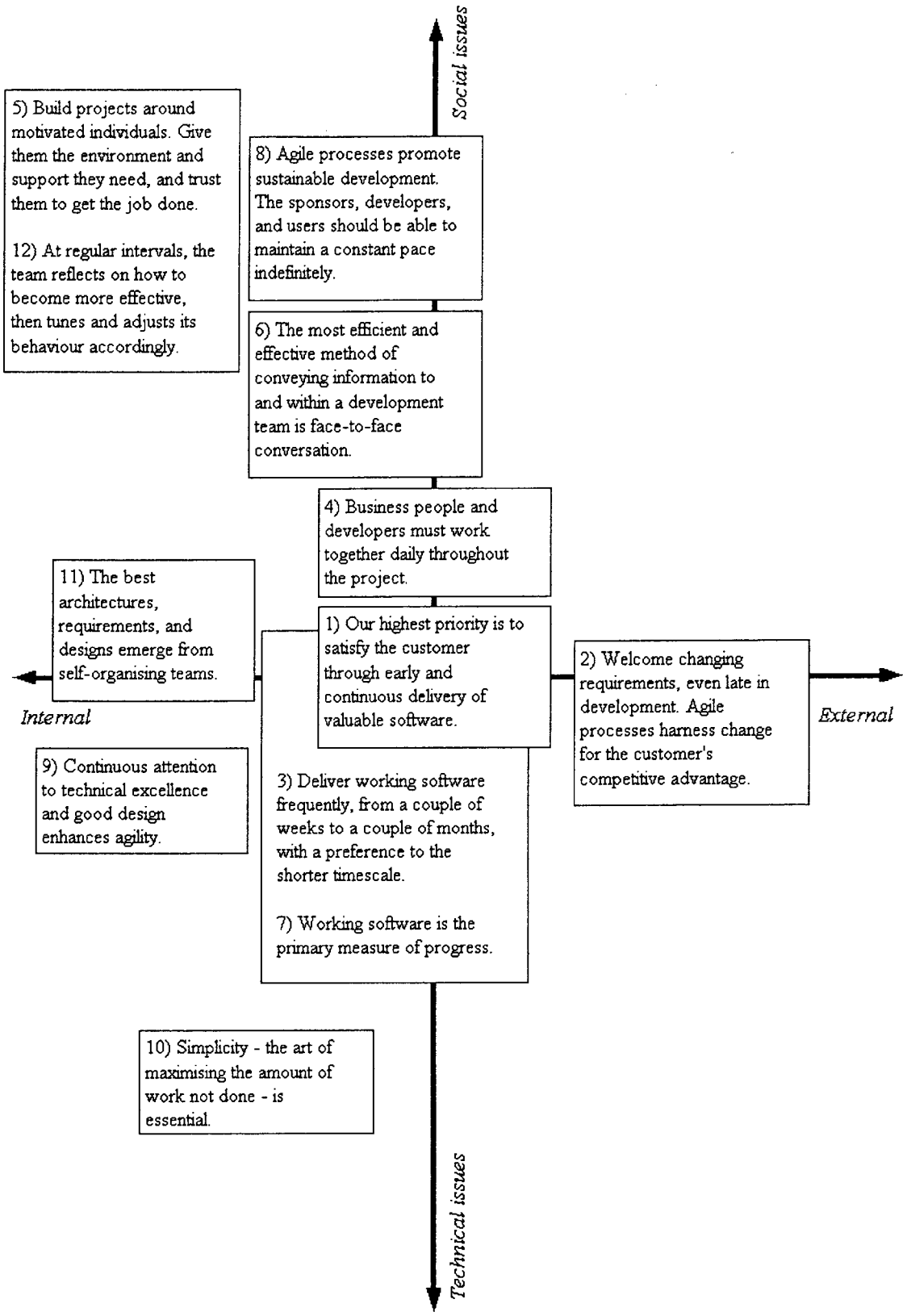

FIGURE 9 The conceptual agile framework

Implementing only the features or requirements that have been agreed upon with the customer is quite a technical principle (principle 10). Even though the customer sets the requirements, the internal development team will do the actual implementation and realisation of this principle. Thus, we have positioned it to be internal and technical.

Although 11th principle highlights the architecture development, which is a rather technical issue, we have analysed the principle mainly from the team building point of view. Thus, we have positioned the principle to be internal and somewhat technical but mainly social.

Based on TABLE 3 and FIGURE 9 we mapped the Agile Manifesto values to similar dimensions. FIGURE 10 visualises the mappings.

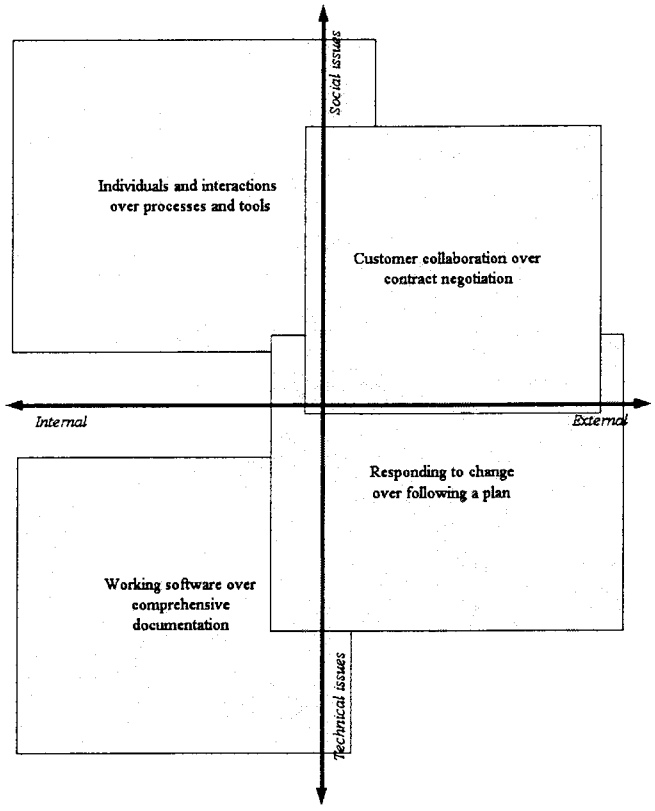

FIGURE 10 Mapping the values to the agile framework

Next, we will examine the feasibility of agile software development for different environments and situations.

4.4 Enabling and limiting factors for the use of agile methods

In several articles, agile and traditional development methods are set against each other and authors are trying to prove which is the best (e.g. Charette 2001a). However, as Glass (2001) states, there is no need for a war or competition between those two. Both types of methods have their benefits and drawbacks, which are additionally subject to certain conditions.

> *After all, both the traditionalists and the agilists are bright and dedicated people, people who want us to perform software engineering in better ways, resulting in better results. Why do we assume, over and over again, that one camp is right and one is wrong?* (Glass 2001, 12)

Hence, instead of announcing a winner in the method competition we should examine in which situations agile methods would be more feasible than traditional methods and vice versa and how they could be combined. Also Boehm (2002, 63) is in favour of this opinion. Furthermore, Boehm (2002) suggests that risk management could assist in deciding in which situations these methods would be most suitable. If the risks are high, more planning is required than when the risks are low, and the possibility to use agile methods exists. Paulk (2001, 26) states that combining traditional and agile methods is important. He focuses on combining one agile method (XP) and SW-CMM:

> *XP provides a systems perspective on programming, just as the SW-CMM provides a systems perspective on organisational process improvement. Organisations that want to improve their capability should take advantage of the good ideas in both, and exercise common sense in selecting and implementing those ideas.*

Several researchers agree with the statement that a certain development method will not suit the varying needs of an organisation. Instead, methods have to be chosen (for example using the contingency approach), modified or even created from scratch (e.g. Tolvanen 1998; Baskerville et al. 1992) to better suit organisation or project specific needs. Thus, an organisation should choose a proper method for their development projects. Lyytinen (1987b) points out that a single method does not usually cover all aspects of systems development, and states:

> *Methods are partial in their focus but can be used to identify problematic situations and object systems for change, to generate and analyse correctness*

> *of change actions (analysis and design methods), to assess and evaluate effectiveness and efficiency of change actions (cost benefit and assessment methods), or to carry out and implement changes (programming methods, organisational implementation methods).* (Lyytinen 1987b, 10)

When choosing a suitable method the size of the organisation and the nature of the development project (for instance criticality in security or time performance) should be considered. Glass (2001) proposes some further issues that should have an impact on the method selection: Differences in application domain, system criticality and innovativeness should be examined before choosing a proper method. Tight schedule and problems in hiring motivated and skilled people might also influence the selection. (Glass 2001, 18) When deciding whether a company will use agile or traditional methods, these considerations should take place. It should be noted that different methods might be used for different subprojects of a development project.

There is little research available concerning the actual use and feasibility of agile methods. However, we found results of one survey, which was conducted by Cutter Consortium (Charette 2001a, b, c, d) to find out in which circumstances agile methods were used in companies and in which situations traditional methods were preferred. The results are based on opinions of 200 information systems or information technology management level respondents representing different industry categories, different-sized companies in different countries. (Charette 2001a, b, c, d) The sample in (Charette 2001a, b, c, d) was small and the survey was not academic. Thus, the results and their generalisability should be examined critically.

The survey results indicated that agile methods were preferred (52% of the respondents) when the primary objective of the development project was to complete the project "on time". Heavy methods (such as RUP, CMM) were preferred when high quality and high functionality were the main objectives. (Charette 2001a, b, c, d) However, Charette (2001a) adds that the survey provides "a truer indication of preferences" when both primary and secondary objectives are taken into account. Combining primary and secondary objectives of completing the project "on time" raises the percentage of preferring agile methods up to 79% whereas the number for preferring heavy methods goes down to 22%. This further reinforces that agile methods were preferred when the objective is to bring the project in "on time". If the percentages of primary and

secondary objectives of completing the project "on budget" are added together, agile methods was preferred by 50% and heavy methods by 41%, thus with this objective the difference of preferring either agile or heavy method is not very notable. See the exact data of the survey results in FIGURE 11.

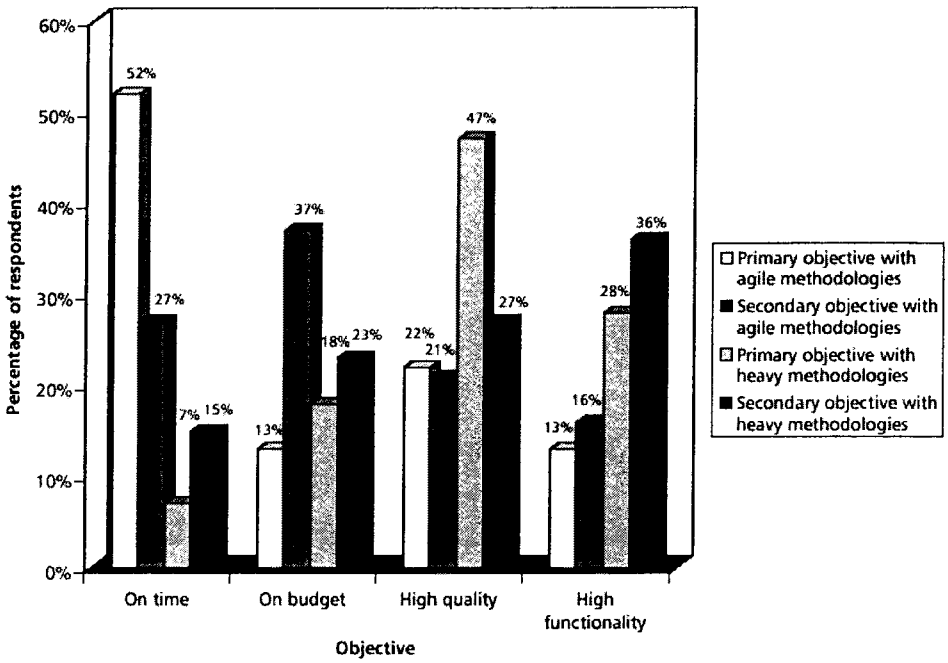

FIGURE 11 Objectives for use of agile or heavy methods (Charette 2001a, 1)

Charette (2001d) claims that companies will continue to use heavy methods in large, long lasting projects that have special safety, reliability, or security requirements and that agile methods are used when the markets require rapid and innovative software development and products. He also believes that more and more companies will start using agile methods in the next few years. However, Charette points out a possible problem with agile methods. He claims that project teams are very happy to use agile methods but managers might have the feeling of loosing control or insight to the progress of the projects and thus resist using agile methods. (Charette 2001d)

Boehm (2002, 66) adds to the discussion concerning the choice between agile and traditional methods:

> *Plan-driven methods work best when developers can determine the requirements in advance -including via prototyping- and when the requirements remain relatively stable, with change rates on the order of one percent per month. In the increasingly frequent situations in which the requirements change at a much higher rate that this, the traditional emphasis on having complete, consistent, precise, testable, and traceable requirements will encounter difficult to insurmountable requirements update problems. Yet this emphasis is vital for stable, safety-critical embedded software.*

Thus, traditional methods pay heavily attention to stable requirements, and if that is not possible in a development project, traditional methods are not feasible.

Hence, according to the points concerning the general method selection and the survey results of Cutter Consortium, we state that large organisations and organisations that are undertaking massive development projects with high quality and safety requirements are most likely to use the heavy methods. Small organisations and those that develop innovative products are most likely to use agile methods. Charette (2001d, 3) estimates that more and more projects will use agile methods in the future. However, he also admits that it took a long time for heavy methods to get established and that might also happen with agile methods and their use. Agile methods seem also to be useful when new, innovative technologies are used. Unstable and changing requirements are often related to innovative software development. Frequent feedback and close customer collaboration used in agile methods facilitate dealing with those changing requirements.

Agile methods were initially designed for small projects, for instance Extreme Programming is explicitly designed for small to medium projects (Glass 2001, 14). Issues concerning scaling up agile methods are still to be solved, thus the use of agile methods in large projects and companies needs to be further examined. Some experiences about operating in a quite agile mode in a large company have been gained from Microsoft and Netscape. According to Cusumano and Yoffie (1998, 68),

> *The synchronise and stabilise process works well for the Internet's very fast cycle times and high degree of uncertainty in market requirements. [...] But this development style, aided by home-grown tools and a few rigid process rules such as daily builds and the periodic stabilisations that accompanied alpha and beta releases, also had advantages. This process gave Netscape and Microsoft an effective mechanism to coordinate large number of developers and testers, and it provided great flexibility in controlling even late design changes.*

Additionally Cusumano and Yoffie (1998, 68) make an important observation that both companies eventually became more and more disciplined:

> *They were flexible and fast as well as creative when these characteristics were important to success, and they paid more attention to quality and schedules when and where these became more important goals [...]. Both Netscape and Microsoft had the ability to adapt rapidly to change and to introduce more process discipline over time.*

4.5 Summary

This chapter described and analysed the values and principles of Agile Manifesto one by one, as well as the background for Agile Manifesto. Agile Manifesto emphasises customer satisfaction, individuals and flourishing teamwork, flexible and fast reaction to changes and the iterative and incremental production of working software instead of piles of paper. The values and principles together form the conceptual agile framework that was presented on page 80. The framework developed by the authors of this thesis assists in mapping the principles to the values and gives a general idea of how different aspects of software development are related to agile software development. In addition, the framework can help analyse which principles are most valid in team's internal activities and which principles are more targeted to external activities. Other dimension of the framework is social versus technology. Socio-technical approach aims at combining technology and people to gain efficiency. Another objective of it is to produce a more humanistic work design. We can see from our framework that both social and technical aspects are taken into consideration in Agile Manifesto. As Agile Manifesto is in accordance with the objectives of socio-technical approach, we assume that following all the principles can improve software development. In addition, job satisfaction and other social aspects related to individuals are acknowledged and respected. This not only facilitates agile software development in short projects but also the long term operations of companies.

Agility derives to large extent from individuals and their competence and tacit knowledge. This sets high expectations for the developers and also for the managers. Finding such talents and personalities can, however, be difficult or even impossible.

Thus, if agile software development relies on such foundation, this can limit the applicability of agile methods.

Further limitations for applying agile methods can be found for instance concerning the demand to react to changes, even in the late stage of the project. This can be fulfilled in projects that have a rather small number of employees and a well-functioning team, and when the software is not very complex. Then again large and critical projects have to freeze their requirements and quit making changes at some, relatively early point.

Traditional methods are most suitable for large organisations and for organisations that are undertaking massive development projects with high quality and safety requirements. Agile methods then again are more suitable for small organisations and those that develop innovative products. Agile approach is also feasible in projects where new and innovative solutions are developed with very tight schedules and changing requirements. However, combining traditional and agile methods is also possible.

Fruitful and frequent communication and collaboration are other main drivers of agile software development. The development team should work closely together but also different kinds of stakeholders should be tightly involved in software development. Communication and collaboration facilitate organisational and individual learning and sharing of knowledge. Frequent feedback and close customer collaboration also facilitate dealing with changing requirements, resulting in better customer satisfaction.

One of our aims for this chapter was to explore whether these agile ideas are fresh and innovative. We found no revolutionary ideas in Agile Manifesto. When we analysed each principle, we discovered that the most thoughts behind them can be found in the literature. Most findings supported the agile principles, some were contradicting. The issues Agile Manifesto addresses have been discussed widely in the literature concerning software development. As such, those issues were well known already before the declaration of the manifesto. We found that many different aspects of software development are covered by Agile Manifesto. In addition to addressing the traditional tasks of software development (such as analysis, design, implementation, project management), job satisfaction is emphasised in Agile Manifesto. Nevertheless,

the emphasis and collection of issues in form of the agile framework can be valuable to software development even though the individual principles were not revolutionary.

To conclude, despite the limitations of Agile Manifesto, it seems quite extensive in many aspects. It does not provide detailed guidelines how to implement the ideas suggested in the principles but it rather gives an overview of several very important issues in software development, which should be taken into consideration holistically when developing software. In the following we will step down from the conceptual level to more practical level as we explore two agile methods: Extreme Programming in the next chapter and agile in-house software development in chapter 7.

5 EXTREME PROGRAMMING

Agile Manifesto with its values and principles gives an "ideological" background for agile software development. Those ideas are materialised in agile methods. According to the survey conducted by Cutter Consortium concerning the use of methods in software development, 54% of the respondents reported that they use own in-house developed methods that can be described as agile. For those who use defined agile methods, Extreme Programming (XP), Feature-Driven Development and Adaptive Software Development were the most popular. (Charette 2001d, 2) In the following we will introduce Extreme Programming, which according to Charette (2001d) is the most often used defined agile method. Other agile methods, such as Adaptive Software Development, Scrum, Crystal Clear, Feature Driven Development and Dynamic System Development Method, are briefly introduced in Appendix 1. Agile in-house software development will be studied in the following chapters.

5.1 Background of XP

> *Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practises, with enough feedback to enable the team to see where they are and to tune the practises to their unique situation.* (Extreme Programming 2002a)

Kent Beck, Ron Jeffries and Ward Cunningham developed XP in the mid 1990's (Paulk 2001, 20). While working on DaimlerChrysler Comprehensive Compensation (C3) project in 1996, they were eager to find better ways to develop software and use new working practises when they restarted a failed software development project. During this project they found out that improving communication, seeking simplicity, getting feedback and proceeding with courage were prerequisites for improving software development. This was the basis for a new software development method called Extreme Programming. (Extreme Programming 2002b, Cockburn 2001) It should be noted that XP was developed before Agile Manifesto was written, and it has widely influenced the principles of Agile Manifesto.

There are different opinions whether XP is an actual method, whether it is agile or how extreme it actually is. Paulk (2001) argues that most of the XP's techniques (called practises in XP) are actually commonsense practises that are part of any disciplined process and thus not so extreme at all. From Haungs' (2001, 118) point of view "XP is not a theory but a cogent descriptive body of successful praxis", thus XP is marked with practicality. XP is also quite fresh, and as one of the inventors of XP (Beck 1999, 77) states:

> *XP is by no means a finished, polished idea. The limits of its application are not clear. [...] My strategy is first to try XP where it is clearly applicable: outsourced or in-house development of small- to medium-sized systems where requirements are vague and likely to change. When we begin to refine XP, we can begin to try to reduce the cost of change in more challenging environments.*

Nevertheless, in this thesis we consider XP as a software development method since it consists of a systematic process and a set of defined techniques (practises).

5.2 XP practises

Tacit knowledge and communication among all team members are highlighted in XP. XP practises such as pair programming and extensive testing further reinforce this insight, as well as minimising documentation. The importance of tacit knowledge also raises a problem: if for instance the turnover of the workers is high, the likelihood that tacit knowledge disappears increases. However, pair programming and intense customer collaboration are ways to share and maintain tacit knowledge. (Cockburn 2001, 168)

XP puts a high premium on customer satisfaction. Taking the customer within the team and receiving feedback frequently are ways to accomplish it. This way customer's suggestions can be taken into account throughout the development project. Another important source of feedback is starting testing in the very early phase. The customer also participates in testing. Thus, XP is developed to provide a favourable setting for programmers to be able to respond rapidly to changing customer requirements. (Extreme Programming 2002b)

XP does not encourage "hacker style" programming that neglects documentation or procedures but requires control at all levels: "project planning, personnel, architecture and design, verification and validation, and integration" (Beck 2000, 22). That control, however, is not rigorous but, as Beck (2000, 22-24) describes it, it is emergent and robust following the wishes of the customer and working style of the development team. Testing has a very important role in XP; testing the produced code at early point and having a running build of the software at all times is another feature of control and quality assurance. Especially unit testing is emphasised.

The important issues mentioned above are included in XP practises, which are the following:

- metaphor
- whole team, on-site customer
- sustainable pace
- small releases
- planning game
- test-driven development, customer tests
- simple design, design improvement, refactoring
- pair programming
- continuous integration
- collective code ownership, coding standard (Extreme Programming 2002a).

Some of the practises are like values (for instance sustainable pace) and others give more precise suggestions how to proceed in the software development (for instance planning game, test-driven development). The development is done iteratively and phases sometimes overlap, thus it is difficult to provide a comprehensive step-by-step description of an XP project and the use of the XP practises. Next, we explain the practises further but more information about their dependencies and relations can be found at (Extreme Programming 2002b).

Metaphor

> *Extreme Programming teams develop a common vision of how the program works, which we call the "metaphor". At its best, the metaphor is a simple evocative description of how the program works, such as "this program works like a hive of bees, going out for pollen and bringing it back to the hive" as a description for an agent-based information retrieval system. (Extreme Programming 2002a)*

Thus, metaphor is used to give an overall picture and common understanding of the system and its functions, and how it should be built. Also Nonaka (1994, 20) presents the concept of metaphor as "One effective method of converting tacit knowledge into explicit knowledge is the use of metaphor." According to Nonaka (1994, 21),

> *Metaphor is not merely the first step in transforming tacit knowledge into explicit knowledge; it constitutes an important method of creating a network of concepts which can help to generate knowledge about the future by using existing knowledge. Metaphor may be defined as being 'two contradicting concepts incorporated in one word'. It is a creative, cognitive process which relates concepts that are far apart in an individual's memory. While perception through prototype is in many cases limited to concrete, mundane concepts, metaphor plays an important role in associating abstract, imaginary concepts.*

Hence, metaphor assists in getting a common view on the development project and helps in transforming tacit knowledge into explicit knowledge.

Whole team, on-site customer

Teamwork is strongly emphasised in XP. Usually XP is set up for small teams. A recommended size for the project team is between 2 and 12 people, though also larger projects have reported success, even up to 80 people in Simons (2002). (Extreme Programming 2002a; Extreme Programming 2002b) Here is a definition of an ideal XP team according to (Extreme Programming 2002a):

> *This team must include a business representative -- the "Customer" -- who provides the requirements, sets the priorities, and steers the project. It's best if the Customer or one of her aides is a real end user who knows the domain and what is needed. The team will of course have programmers. The team may include testers, who help the Customer define the customer acceptance tests. Analysts may serve as helpers to the Customer, helping to define the requirements. There is commonly a coach, who helps the team keep on track, and facilitates the process. There may be a manager, providing resources, handling external communication, and coordinating activities. None of these*

> *roles is necessarily the exclusive property of just one individual: Everyone on an XP team contributes in any way that they can.*

Office arrangements further facilitate teamwork: XP suggests that a team should be located in one, open workspace close to each other. (Beck 2000, 23)

As mentioned above, in an XP project the customer is seen as part of the development team. The term *on-site customer* derives from this idea. The advantage of being constantly near the customers or usage experts enables the programmers get rapid feedback for the questions raised during the programming. (Beck 2000, 23-24; Cockburn 2000, 31) This can also assist the development team to get a deeper understanding of the user needs and habits. In addition, as not much time is wasted to reach the customer, more ideas can be experimented. This is also a step towards a better and more usable final product and more satisfied customer. (Cockburn 2000, 31)

Sustainable pace

> *It's pretty well understood these days that death march projects are neither productive nor produce quality software.* (Extreme Programming 2002a)

"Death march project" refers to a situation where employees are under a lot of pressure and working very hard and are most likely to be stressed. XP principles suggest that team members should not work overtime, at least not for more than one week at a time (Beck 2000; Extreme Programming 2002a). As we mentioned in chapter 4.2.8, tired and stressed out people make more mistakes, thus normal working hours are more efficient in the long run.

Small releases

> *The development team needs to release iterative versions of the system to the customers often. The release planning meeting is used to discover small units of functionality that make good business sense and can be released into the customer's environment early in the project. This is critical to getting valuable feedback in time to have an impact on the system's development.* (Extreme Programming 2002b)

Small releases play an essential role in XP: employing iterative development can increase agility in the development process. XP suggests dividing the development schedule into about a dozen iterations that last from one to three weeks. (Extreme

Programming 2002b) As iteration cycles in XP are very short, only a relatively small set of functionality can be provided in a new release (Wiki 2002). To succeed in today's software business environment time-to-market is essential. This goal can be facilitated by frequent production releases (Taber & Fowler 2000, 13).

Planning game

> *XP planning addresses two key questions in software development: predicting what will be accomplished by the due date, and determining what to do next.* (Extreme Programming 2002a)

XP planning emphasises steering of the project as it proceeds, rather than trying to give exact, fixed predictions of what will be needed and in how much time in the beginning. Additionally, planning game can reduce problems related to transfer or potential disappearance of tacit knowledge, because in planning game tacit knowledge is transformed to explicit knowledge through writing story cards and prioritising them. (Cockburn 2001, 168) Planning game consists of two steps: the first step is *release planning* and the second step is *iteration planning*. In release planning the customer presents the desired features. The programmers then estimate the difficulty of desired features and make cost estimates. The customer can prioritise and plan the project further using that information. (Extreme Programming 2002a) Release planning is more high level planning, whereas iteration planning is done approximately every fortnight. As business people (often the customer) and technical people work together, versatile competence can be utilised. The purpose of release planning is that a project may be quantified by four variables: scope, resources, time and quality (Extreme Programming 2002b).

During iteration planning the customer sets the requirements for the next desired features by writing story cards. Those are simple explanations of the desired features and they should be "business-oriented, testable and estimable" (Beck 1999, 71). Story cards can be added to the requirements pool as the project goes on, and in the beginning of each iteration round the customer chooses the ones with the highest priority to be implemented in the next round (Beck 1999). The programmers break story cards down into tasks and estimate their cost (at a finer level of detail than in release planning). Based on the amount of work accomplished in the previous iteration, the team signs up

for what will be undertaken in the current iteration. Running software ought to be delivered at the end of each iteration (Extreme Programming 2002b), and usually each iteration contains new features based on story cards (Taber & Fowler 2000, 13).

Test-driven development, customer tests

Frequent testing provides feedback that is highly emphasised in XP. XP's testing practises differ considerably from methods that suggest testing in the end of the development or that prohibit programmers from testing their own code (Beck 1999, 74). For instance Microsoft has separated programming and testing, see chapter 4.2.9. Instead, XP emphasises starting testing at early point and encourages programmers to test their code by themselves.

XP's *unit tests* enable quick changes in rapid software development projects. Jeffries (1999) suggests writing unit tests for every feature to ensure that new features work. In addition, all unit tests ought to be run in the entire system before releasing any code to make certain that nothing else is broken. (Jeffries 1999, 24) XP's "test-first development" means that programmers write test cases and actual tests before they start programming (Beck 1999; Extreme Programming 2002a). The test results offer programmers feedback of the functionality and design of their code and also enable making the software work straightaway. All the produced code should clear the tests at all times to ensure a functioning build. Tests are gathered together, and a test written for a particular feature should be cleared by the particular piece of code but also the main build should clear the test (Beck 1999).

Unit tests enable fixing small problems within a short timeframe instead of fixing large problems later on. Furthermore, unit tests should be automated. Automated unit tests are expected to pay off their cost of creation (mostly due to the time that they require) quite quickly in the course of project. (Extreme Programming 2002b) In addition, automated tests eliminate bypassing manual tests, which easily occur in the time of stress (Extreme Programming 2002a).

When presenting the desired features, the customer also defines one or more *acceptance tests* beforehand to examine whether the feature is working. The development team

builds and runs those tests. The results again guide the team and the customer to further design and implement the software (Extreme Programming 2002b).

> *[...] "programmer tests", or "unit tests" are all collected together, and every time any programmer releases any code to the repository (and pairs typically release twice a day or more), every single one of the programmer tests must run correctly. (Extreme Programming 2002b)*

Testing practises also facilitates learning:

> *Programmers write their own tests and they write theses tests before they code. If programming is about learning, and learning is about getting lots of feedback as quickly as possible, then you can learn much from tests written by someone else days or weeks after the code.* (Beck 1999, 73-74)

Simple design, design improvement, refactoring

Simple design refers to developing only as much code as is required, adding only the planned functionality at a time. Keeping design simple is not an easy task. However, Extreme Programming (2002b) suggests keeping things as simple as possible as long as possible, and adding only functionality or piece of code when it is really relevant.

Simple design is being improved as the process goes further and iteration rounds continue. According to Extreme Programming (2002a), the design improvement process focuses on removal of duplication and on increasing the cohesion of the code, while lowering the coupling. They also use the term *refactoring* for this, which basically means improving the code. Refactoring is facilitated by unit tests, as they can verify that changes in structure have not caused changes in functionality (Extreme Programming 2002b). High cohesion and low coupling have been recognised as the characteristics of well-designed code for at least thirty years. Beck (2000, 23-24) adds that this offers an opportunity to the programmers to first solve small problems and later enlarge the solution to maybe more complex problems.

> *Refactor mercilessly to keep the design simple as you go and to avoid needless clutter and complexity. Keep your code clean and concise so it is easier to understand, modify, and extend. Make sure everything is expressed once and only once. In the end it takes less time to produce a system that is well groomed.* (Extreme Programming 2002b)

Stephens (2001), who subjectively criticises XP on his informal WWW-site points out that constant refactoring might also mean unnecessary "polishing" of the code and should only be done occasionally so that no extra time is wasted on code that is working already. Refactoring was also discussed in chapter 4.2.9.

Pair programming

> *All production software in XP is built by two programmers, sitting side by side, at the same machine. This practise ensures that all production code is reviewed by at least one other programmer, and results in better design, better testing, and better code.* (Extreme Programming 2002a)

Pair programming is a practise, in which two programmers work side by side at one computer on the same design, algorithm, code, or test. The other one is writing the code and the other is giving comments, reviewing the results and looking for defects. A pair can also make pair design and pair analysis in a similar mode. (Williams et al. 2000, 19) Pair programming provides a favourable setting for sharing communication throughout the team: pairs can learn from each other (Extreme Programming 2002b; Brooks 1987, 18), and switching pairs further leverages the diffusion of knowledge (Extreme Programming 2002b). Pair programming brings social control as it facilitates focusing on the issues at hand; when a pair is sitting in front of one computer, they will most likely concentrate on programming instead of reading email for example.

According to Williams et al. (2000, 21), some programmers view pair analysis and design as more critical than pair implementation (programming). Thus, it is not necessary for a pair to sit together all the time but rather discuss and solve the most difficult problems together, and do the simple and routine tasks individually. The idea of a pair working together can also be utilised in other kinds of tasks, for instance Microsoft uses pair testing, see chapter 4.2.9. Microsoft's pairs, however, consist of one developer and one tester and thus they do not write the code simultaneously together as in XP's pair programming.

Pair programming, however, is not a new invention. Brooks (1975) mentions a setting similar to pair programming. In his model there is a chief programmer and another person working with her or him as an assistant, who is able to give advice and

comments on chief programmer's work. Pair programming is further discussed in chapter 5.3.

Continuous integration

The parts of the developed system are kept fully integrated at all times (Extreme Programming 2002a). Continuous integration constantly reduces the chance of conflicting changes (Beck 2000, 24). According to Extreme Programming (2002a), weekly or daily builds are not enough; XP teams integrate their accomplishments more often, some times several times a day. (See explanation about weekly build in chapter 4.2.3) Microsoft uses daily or weekly builds:

> *Microsoft uses a range of build-cycle frequencies depending on the particular needs of a project and the amount of time required to complete a build successfully. Systems products generally take longer to build because of their size and the number of files and interdependencies included. Microsoft builds Excel, Word, and a testbed version of Office daily; it builds the full version of Office at least weekly.* (Cusumano & Selby 1998, 275)

Integrating code several times a day seems like a quite ambitious goal. It means that the programming pairs have to deliver several pieces of working code tremendously often and sometimes it might not be possible to produce logical entities within a very short time. Stephens (2001) is of the following opinion concerning this practise:

> *Of course there's the inevitable keel-hauling if your code doesn't compile and pass all the tests when integrated. Therefore, no developer is going to attempt anything complex enough to take more than a day. What if they are writing a module that is going to take three days?*

Collective code ownership, coding standard

Collective code ownership means that any pair can improve any code in the system (Beck 2000, 24). When this is done in pairs, the problem of using or altering a piece of code that is not fully understood is likely to be minimised. Collective code ownership also enables changing the code quickly, while the changes do not have to be made by a certain individual assigned to own the code but anybody can make changes. (Extreme Programming 2002a) In practise this works as follows. The programmers release all source code to a source code repository. One development pair can integrate, test and release changes at a time. Pairs can integrate their own changes with the latest version at

their own workstation any time but integrating them with the team requires teams to wait for their turn. This practise requires some kind of locking system. An example of such is a single computer dedicated to this purpose. This may work well if the development team is co-located. (Extreme Programming 2002b)

XP suggests using same *coding conventions*. This reduces for instance duplicate work that may occur if programmers are using different names for similar classes. (Wiki 2002)

> *XP teams follow a common coding standard, so that all the code in the system looks as if it was written by a single – very competent – individual. The specifics of the standard are not important: what is important is that all the code looks familiar, in support of collective ownership.* (Extreme Programming 2002a)

In addition, coding standard provides rules that prevent developers from writing their own short cuts or unacceptable solutions.

However, XP does not provide further instructions on how to create a coding standard within a XP team but it is left for the team members to decide or to use a defined standard. Common coding practises once again seem like commonsense practises, and are highly emphasised at Microsoft (Cusumano & Selby 1998, 285) for instance. There it is regarded as "speaking the same language", which of course has a very good motivation: when people use common programming languages and naming conventions, problems in communication and in code compatibility are likely to decrease. Coding standard can to certain extent be regarded as part of Lyytinen's (1987b) language context and formalisation of language:

> *In this [language] context, the IS [information system] change is about coming to an agreement on conventions that govern language use: its form, signification, use-intention and so on. Information systems development qua language change involves a language development and formalisation process. During systems development languages evolve: new meanings and words are defined, new uses established and so on. Language formalisation means that its use is made more systematic and institutionalised: syntax of language is restricted to preselected forms, and its use-patterns and contexts are fixed.* (Lyytinen 1987b, 14)

Formalisation and thus coding standard can increase common understanding and assist communication within an organisation.

5.3 XP experiences

Pair programming has been among the most popular topics in articles analysing XP in practice. First we discuss pair programming and then present some other viewpoints of XP use in practice that have been documented in the literature.

Pair programming has been researched in other than XP context in the mid 1990's, and the results then and in the recent years show that pair programming improves software quality and reduce time-to-market (Williams et al. 2000, 19-20; Nawrocki & Wojciechowski 2002). In addition to improving efficiency and high quality, it has also been reported in Williams et al. (2000) that 90% of the programmers felt happier and more confident about themselves when working in pairs.

If programmers do not have any previous experience of pair programming, starting such a practise can be quite problematic. Often programmers have prejudices against programming in pairs and believe that pairing with someone would be uncomfortable and inefficient (Williams et al. 2000). Some programmers adjust to and enjoy pair programming but as Beck (1999, 76) states, "XP is an intensely social activity, and not everyone can learn it." He even suggests that people unsuitable for pair programming should be fired or transferred to other tasks because the quality and efficiency of their work is often not satisfying.

DaimlerChrysler's Comprehensive Compensations (C3) project with 15 employees and a large-scale payroll system to develop has been used as an example of XP in practice in Haungs (2001) and Beck (1999). Haungs states that depending on the type of design and implementation, pair programming can be a very useful practise. "Pair programming forces you to make explicit your implicit assumptions. Our combined efforts produced a tool that was much greater than the sum of its parts." (Haungs 2001, 119)

Stephens (2001) has several opinions about this XP practise too. He believes that in some cases a pair that includes a senior and a junior programmer might waste the

senior's time if the senior keeps correcting the junior's work all the time. He also points out that acting as a mentor to a junior programmer should be optional, because not everybody wants or is able to act as a mentor. Pair programming and all other XP practises as well are actively debated in several newsgroups[8]. Cockburn (2000, 32) agrees with Stephens: the less experienced developers can slow down the project as the experienced developers have to spend their valuable time helping the inexperienced.

DaimlerChrysler representative Chet Hendrickson (in Beck 1999, 75) describes his experiences about XP in C3 project quite enthusiastically:

> *When we have driven our development with tests, when we have written code in pairs, when we have done the simplest thing that could possibly work, we have been the best software development team on the face of the earth.* (Beck 1999, 75)

Hendrickson admits that the project has not been completely successful but states:

> *Looking back on this long development experience, I can say that when we have fallen short of keeping our promises to our management and our customers, it has been because we have strayed from the principles of XP. When we have driven our development with tests, when we have written code in pairs, when we have done the simplest thing that could possibly work, we have been the best software development team on the face of the earth.* (Beck 1999, 75)

Taber and Fowler (2000) were working on a demanding leasing application built with Enterprise Java and they describe their experiences after eight months of using XP. The project team was quite big, 50 members, of which 25 were developers. (Taber & Fowler 2000, 13-14) They found the use of XP to be fruitful in general and in addition listed some specific issues of how XP practises had helped their work:

- The estimation of how much could get done in a given iteration was very accurate. Close communication and short iterations helped setting realistic expectations for the project's continuation.

---

[8]For instance in http://groups.yahoo.com/group/extremeprogramming/ [referred on 15.3.2002]

- There was consistent, quick, and demonstrable process every month as the story cards became reality as working code.
- Communication within the team and to the client and sponsors was vastly improved. The planning meetings were good opportunities to get everybody updated on the project status and discuss the next iteration goals.
- Customer participation was very active.
- The team itself initiated improvements for the process, which lead to the process iteratively improving itself.
- Working in an XP way also increased the team morale. (Taber & Fowler 2000, 18-20)

Simons (2002) reports an XP project of 80 people to have worked well. When doing XP with a large number of people, he suggests that first the project should be started in small scale and gradually add more people to it as the common understanding of the object system prevails. According to Simons (2002), pair programming is an excellent way to get more people quickly involved and updated about the project.

5.4 Limitations of XP

We discussed general limitations for using agile methods in chapter 4.4. Here we will focus on limitations concerning applicability of Extreme Programming.

Paulk (2001) criticises XP for its general focus on technical work and lack of managerial issues (such as in SW-CMM). However, Paulk (2001) points out that those two could also be combined to get all the issues covered.

Discussion about the customers of software development projects was presented in chapter 4.1.3 on page 47. Customers should have strong, comprehensive knowledge of the developed software and they should be committed to the project (Boehm 2002, 66). Working code is emphasised over documentation in XP, and therefore no extensive documentation usually exists. Instead, the customers are expected to understand the produced code and give feedback and acceptance based on it. XP also emphasises

customer tests and expects the customers to write tests. However, in our opinion this kind of thinking can be questioned. If the customer representative does understand the code, he or she might be a highly technically oriented person and might be incapable and unauthorised to make strategic business decisions. If the customer representative is a business-oriented person instead, it cannot be expected that he or she would understand programming solutions thoroughly. Thus, he or she may be unable to base decisions regarding the next phases of the development on realistic facts. Hence, it can be questioned whether XP is applicable at all in cases where the customer does not have the sufficient technical knowledge. For instance, if a company assigns a subcontractor to develop software for their needs, they most probably do it because they do not have the required knowledge and skills themselves. Instead, they want to utilise the best possible technical expertise by outsourcing the project to specialist.

As mentioned, XP was designed for small teams and scaling up these practises including pair programming can be very challenging. Brooks (1975) proposed that the "chief programmer and assistant" model he presented, quite similar to pair programming setting, can be scaled up to a large group of people and to a large project by dividing the work among several teams. Several teams have their own chiefs and assistants then, and mainly the chiefs communicate with each other. However, communication will become a challenge in the case of scaling up, as well as controlling architecture decisions. Apparently, XP has been mainly used in tailored projects (see for instance Beck's statement in chapter 5.1 Background of XP). Using XP in MOTS and COTS business modes has not been discussed.

Continuous integration several times a day might bring several problems (see page 98) and issues to consider. If the developer pairs are required to deliver working code too often, their work might become fragmented and less focused, and as a result the quality of code might be poor. Also collective code ownership might be problematic in our view. If the software under development is large and complicated, it is rather improbable that every single person or even every pair would have the required knowledge of all the functionalities and dependencies of the code. Thus, developers' ability to understand and change the code written by others decreases as the size of the software increases. Microsoft does not expect everyone to know the whole code

(Cusumano & Selby 1998, 264), although each developer has to make sure that the piece of code they are working on, can be synchronised with the big main build. Additionally a build master controls the stability of the build (Cusumano & Selby 1998). Pair programming and testing aim to spread knowledge about the code throughout the development team and enable the collective code ownership. However, such ways to spread the knowledge about the code seems inadequate in a case of a very large software, and thus this practise can be problematic.

Pair programming and also other XP practises are very social, thus participants are expected to be communicative and collaborative. This indicates that XP is not suitable for everybody. Some people might find working in an XP project very enjoyable, others then again might find it highly uncomfortable.

Automated testing of large systems takes a long time, and thus using automated testing might not be feasible in the development of large software.

5.5 XP compared with the principles of Agile Manifesto

Here we analyse XP from Agile Manifesto point of view, comparing XP practises and Agile Manifesto principles to find similarities and dissimilarities. Refer to Agile Manifesto principles on page 80. In addition, we evaluated XP practises in a similar way that we did with Agile Manifesto principles, positioning the practises of XP according to social versus technical, and internal versus external orientation. This categorisation is also discussed below, and the results of it are illustrated in FIGURE 12 on page 109. When categorising XP practises, we focused on the direct implications and emphasis of the practises. For instance the small releases practise was categorised as a technical practise, although indirectly small releases might affect social issues, such as motivation, as the product is becoming more complete and the requirements are becoming clearer, and it can also affect communication as it enables frequent feedback.

Customer satisfaction is a main driver both in Agile Manifesto and XP, and it is directly or indirectly involved in every principle of the manifesto and practise of XP, and thus not repeated in the following paragraphs.

Metaphor

Metaphor is supported by the general Agile Manifesto idea that emphasises communication and mutual understanding within the team. No principle directly addresses the metaphor.

As the metaphor describes the system in a way that everybody should be able to understand, we have positioned it to be equally internal and external. In addition, it is supposed to give an overall view of the system and it functions, which makes it technical but somewhat approaching the social orientation.

Whole team, on-site customer

XP's whole team practise is in line with Agile Manifesto principles 4, 6, 11 and 12. The XP customer is encouraged to work as a member of the team, so the idea of business people and developers working together is taken even further in XP than Agile Manifesto principle 4 suggests. XP's idea of teamwork also includes face-to-face communication of principle 6, as well as the recommendation for co-location.

Agile Manifesto principles 11 and 12 also refer to teamwork. Although those issues are not directly addressed in XP practises, self-organisation and the team tuning their own performance to become more agile are prerequisites for a successful XP project in our opinion. Furthermore, XP facilitates such behaviour.

These XP practises are quite socially oriented, and equally taking both internal and external dimensions into account.

Sustainable pace

XP and Agile Manifesto's principle 8 totally agree on maintaining a sustainable pace. Keeping up sustainable pace facilitates human well-being, thus we have positioned it to be very social. As XP customers are taken into the team, practise of sustainable pace is both internally and externally oriented.

Small releases

Principle 3, bringing customer satisfaction though early and continuous delivery of valuable software, is in line with XP's small releases. This practise is technically oriented and takes internal and external dimensions into account.

Planning game

Planning game is one way to react to changing requirements that are welcome in principle 2, thus XP and Agile Manifesto agree on this one too. Actually, reacting to changes is one of the main drivers in both Agile Manifesto and XP. Planning game is partly linked to principle 4 as both the customers and technical people participate in the planning process.

This practise is technical but somewhat approaching the social dimension as well, as the customers are expected to express their wishes to guide the development to meet their requirements. Thus, it is positioned to in the middle of internal–external axis.

Test-driven development, customer driven tests

Testing is widely described in practises of XP but only briefly mentioned in Agile Manifesto principle 9. Both aim at technical excellence through testing but in XP receiving feedback through testing is highlighted. The feedback from testing is used to make changes based on changing requirements that is addressed in Agile Manifesto principle 2. Customer driven tests are trying to ensure that the customer gets the functionality he or she wants. Tests are a way to measure the progress, thus this practise can be linked to principle 7.

As pursuing technical excellence is the main driver of testing, we have placed it to be technical. Then again, the customer driven approach makes it equally internal and external according to our view.

Simple design, design improvement, refactoring

As being closer to practise, XP again takes the issues suggested in Agile Manifesto principles 9 and 10 further. Agile Manifesto emphasises simplicity ~~and~~ technical

excellence on a higher level, and XP takes the ideas of how to accomplish simple design and improving it to a more practical level, although XP does not provide detailed guidelines how to accomplish that either. Refactoring and implementing simple and tidy code makes these practises technical. As the developers are the ones to implement the code, these practises are internal.

Pair programming

Pair programming could be regarded as "extreme" face-to-face communication, which is more than the Manifesto principle 6 suggests. As the pair works together, they can support each other and learn from each other, and pair programming has been found motivating in studies (see chapter 5.3). Those issues are in line with principle 5 of the manifesto. Pair programming also facilitates the quality of the software while two people are working together, which enhances technical excellence mentioned in principle 9.

We have placed pair programming to be internal. The pairing approach makes this issue social but programming is also very technical in nature. Hence, we positioned this practise in the middle of the social–technical orientation axis.

Continuous integration

Continuous integration meets Agile Manifesto principle 3 that suggest delivering software frequently. Integrating continuously also enables implementing changes even in a late stage as demanded in principle 2. This practise is very technically oriented, and also refers to team's internal tasks.

Collective code ownership, coding standard

Collective code ownership and coding standard are in line with Agile Manifesto principle 9, which promotes technical excellence and good designs. Issues related to coding makes these practises technically oriented but to some extent they approach social orientation, as everybody is expected to have access to the code. These are also internally oriented practises.

TABLE 4 recaps the links between XP practises and Agile Manifesto principles.

TABLE 4 XP compared with Agile Manifesto

XP practise \ Agile Manifesto principle	1	2	3	4	5	6	7	8	9	10	11	12
Metaphor	x											
Whole team, on-site customer	x			x		x					x	x
Sustainable pace	x							x				
Small releases	x		x									
Planning game	x	x		x								
Test driven development, customer tests	x	x					x		x			
Simple design, design improvement, refactoring	x								x	x		
Pair programming	x				x	x			x			
Continuous integration	x	x	x									
Collective code ownership, coding standard	x								x			

It would be possible to find other linkages as well because several XP practises and Agile Manifesto principles can be interpreted in different ways to be directly or indirectly related. For instance, whole team and customer's presence on-site can indirectly increase developers' motivation, and thus that XP practise would support Agile Manifesto principle 5. These indirect connections indicate that both XP practises, just like Agile Manifesto principles, are interrelated propositions. Following one practise or principle will cause something, and might enable following some other practise or principle. For example, following continuous integration practise can result to better customer satisfaction. At the same time that practise is a prerequisite for small releases practise.

As a result of our comparison it can be stated that XP practises are to great extent in line with the principles of Agile Manifesto. XP practises are more closely related to practical work, whereas principles of Agile Manifesto are often on a higher, more general level. Hence, XP practises complement and support Agile Manifesto. When comparing the above XP practises framework with the agile framework, we can see that XP places more emphasis on the internal and technical related issues than Agile Manifesto. More principles placed in internal–social factor can be found in the agile framework than in XP practises framework. However, the framework comparison also confirms the

statement that XP and Agile Manifesto complement and support each other because in general quite similar issues are emphasised in both frameworks.

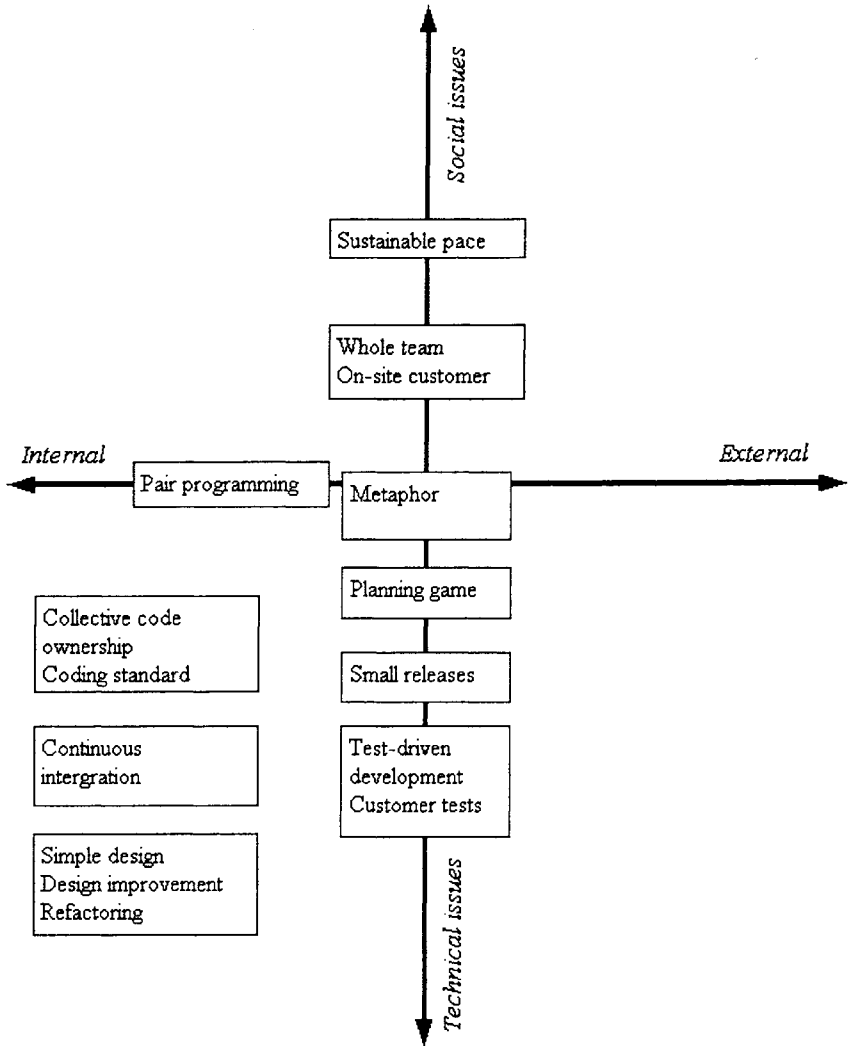

FIGURE 12 XP practises framework

5.6 Summary

In this chapter we described Extreme Programming, which is the most commonly used defined agile method according to a survey by Cutter Consortium (Charette 2001a, b, c, d). First we analysed the most important XP practises one by one. We also explored how XP has been utilised in practice, and what kind of experiences and results were reported concerning XP projects. Using XP was reported to be a success in projects with varying number of employees. The reports revealed that XP was mainly used in tailored

projects. Pair programming in particular seemed to be a very good practise that increases efficiency and individual's confidence and well-being, although it should be noted that it does not suit everyone.

Although XP was found very useful and efficient in software development, there are also limitations for its use and applicability. One of the main limitations is the presumption that the customer should be highly technically oriented, which is not always possible. Scaling up XP for larger teams can be difficult, although success in scaling up was also reported (for example in Simons 2002). When scaling up, for instance communication and planning have to be in more formal and systematic level than what XP suggests. In addition, collective code ownership can be challenging especially in a case of a very complex software because it is improbable that every developer would know the code well enough. Continuous integration several times a day might result in fragmentation of developers' concentration and worse quality of software.

Finally we compared the practises of XP and principles of Agile Manifesto, and established that they complement and support each other very well. We organised the XP practises in a similar framework with internal-external and social-technical oriented axes and found out some differences with the agile framework. However, this framework comparison further confirmed our findings that XP and Agile Manifesto complement and support each other. Next we will introduce our empirical research setting and then we will study agile in-house software development using a case study.

6 EMPIRICAL RESEARCH SETTING

Here we will describe the empirical research setting, but first we will briefly explain the background of our case study.

We studied a development project in which a corporate venture (defined chapter 2.9) developed a software product. Originally this case study was done from a process modelling point of view to assist the case organisation to enhance their processes, not actually from agile software development point of view. However, this case is very interesting in many ways within the scope of the agile software development research too, and the data were suitable for this different viewpoint. Although the case venture did not consciously use agile methods or follow the principles of Agile Manifesto, their way of working and their environment was agile in many ways. Time-to-market pressure existed, which is characteristic for agile development, and in addition they created an innovative product.

The product development project of the case venture included all the steps from developing an entire software product from design through implementation to launching it. The corporate venture did not use any beforehand described or given process models (such as CMM) in their development work but rather their work was done in skilled improvisation manner. However, they acknowledged that to become more efficient, they would need to study their processes and form some kind of more structured procedures and processes to assist their future work. The corporate venture did not use any particular software development methods either. They created their own, in-house development method for developing software for this particular project and product based on their early experiences and resulting tacit knowledge. In this research the case is considered as an example of an agile in-house development method.

What makes this case even more interesting is the fact, that the case organisation was a corporate venture within a large corporation. According to Käkölä (2001):

> *[Corporate] ventures can overcome most hurdles faced by traditional independent start-ups more easily than start-ups provided that their parents have solid resource bases such as highly appreciated and well-recognized brand image, solid financial situation, innovative technology strategy, and*

> *well-functioning technology development and marketing. For example, corporate ventures can create solid businesses relatively quickly by innovatively combining existing core knowledge and technologies within the corporation. After all, brand image as well as marketing and distribution channels and expertise exist already and market creation or penetration need not be invested in as heavily as in traditional start-ups.*

Our case venture also reflects Käkölä's description of different aspects of corporate venturing. The studied corporate venture operated partly quite autonomously like a start-up company but was assisted by the resources of the parent corporation. The corporate venture could for example get assistance in marketing and technical issues, and receive financial aid by the parent corporation.

6.1 Research methods

We described the general research design in chapter 1.2. Here we describe how the empirical case study was carried out and explain how the research data were gathered and analysed.

The goal of this empirical study was to gain understanding and a thorough insight of the undefined processes of the case organisation and find out the weaknesses and strengths of the venture. As we came from outside the corporation, we had no previous knowledge about the case venture, nor about their processes and working methods. Additionally, as we have mentioned, agile software development is a fresh topic and thus not widely studied in academia. Moreover, the venturing concept in particular has not been addressed in the research of agile software development. According to Creswell (1994, 145-146),

> *The qualitative study approach is considered an appropriate method when little is known about the phenomenon under investigation and the concepts are immature due to lack of theory and previous research and a need exists to explore and describe the phenomena.*

Thus, as we were exploring a phenomenon unfamiliar to us and analysing it from the rather immature and unestablished agile software development point of view, we chose to use qualitative research approach.

According to Yin (1989, 25),

> *[...] case studies have a distinctive place in evaluation research. There are at least four different applications. The most important is to* explain *the causal links in real-life interventions that are too complex for the survey or experimental strategies. A second application is to* describe *the real-life context in which an intervention has occurred. Third, an evaluation can benefit, again in a descriptive mode, from an illustrative case study - even a journalistic account - of the intervention itself. Finally, the case study strategy may be used to* explore *those situations in which the intervention being evaluated has no clear, single set of outcomes.*

We used the case study approach to describe and explore the software product development in the case organisation (chapter 7) and different issues related to that.

6.1.1 Research design and procedure

The case study was conducted to investigate the product development process of a corporate venture. The case study consisted of a current state analysis to gather information about the software product development process, about the involved parties and their relations, as well as their working methods and working environment. The current state analysis consisted of collecting material and conducting interviews to gain understanding of the development of the first released product. The collected material consisted for instance of a project plan and other material created within the project. Additionally, organisation-specific information and information of the domain (such as products developed, technologies used, and services offered) from the WWW. The interviews were conducted as focused interviews. Kidder and Judd (1986) state that in focused interviews the interviewer has a list of selected topics he or she wants to cover but can flexibly choose how and when to ask the questions. According to the interviewer's evaluation, he or she can change the course of the interview accordingly. Interviewees are given the freedom to openly express their thoughts but the interviewer is in charge of the direction of the interview. Further, focused interviews can be broadened to cover:

> *[...] any interview in which interviewers know in advance what specific aspects of an experience they wish to have the respondent cover in their discussion, whether or not the investigator has observed and analysed the specific situation in which the respondent participated. (Kidder & Judd 1986, 274-275)*

The employees of the venture as well as other people relevant to the project were interviewed. According to Yin (1989), there are six sources of evidence that can be the focus of data collection. Those are: "documentation, archival records, interviews, direct observations, participant-observation, and physical artefacts" (Yin 1989, 85). Of those six, we mainly used interviews as a source of data, but as mentioned we also used documentation (e.g. project plan) and archival records (e.g. organisational information). Observation was not used and the artefact produced was not analysed, although it might have brought some specific information for example about the quality of the software.

The theme interviews were conducted as follows. First, the background information such as name and area of responsibility in this project was collected. The actual interview was mainly divided in two parts.

The structure and questions of the first part were adapted from the framework of El Sawy (2001, 79-89). El Sawy's framework is designed for business process reengineering (BPR) and it provides a setting for defining enterprise processes for BPR purposes. However, we found his framework suitable for our study because our intention was also to explore and distinguish processes.

The second part of the interview handled the processes of this particular venture in depth. It consisted of more detailed questions about how, why and when different tasks and activities were performed. In addition to the current state description, the interviewees also pointed out weaknesses in their work, and suggested how those should be managed. The aim was to gather as much process specific information as possible. Besides such detailed questions, in the end of the interview the interviewees were asked to evaluate the whole product development process in general.

The first part of the interview was the same for almost all the interviewees but the second part varied according to the interviewees' area of responsibilities and tasks. The interview forms were sent by electronic mail to the interviewees beforehand to provide them a possibility to prepare in advance. This was important, because some of the events and issues discussed had happened several months earlier, and the interviewees needed to recollect what had happened. Sending the forms in advance also enhanced

fluency of the interviews. An example of one interview form can be found in Appendix 2.

Nine interviews were conducted and altogether fourteen people were interviewed. Usually two to three people participated in an interview. All members of the venture team were interviewed. In addition, other involved people from venture's stakeholders were interviewed. When interviewing the stakeholders, at least one member of the venture team was present. This facilitated thorough understanding, because the venture team members could set further questions to the interviewees if necessary. This assisted in capturing data as accurately as possible. The interviews lasted from 45 minutes to four hours. To avoid bias the interviews were recorded.

6.1.2 Analysis of data

The analysis of data began by verbatim transcription of the interviews. Verbatim transcribing enables using citations in analysis. The meaning of citations is to bring the reader closer to the target group and describe certain things in words of respondents. (Solatie 1997, 62) The interviews were conducted in Finnish and we translated the direct citations (used mainly in case analysis) into English. Translating word by word was naturally impossible but we rather aimed to emphasise the main points than making exact translations.

The interview data are confidential and not included in this paper except for some short citations.

The interview data were used to describe processes of the development project. Processes were modelled using process modelling software, visualising process models as charts and writing clarifying descriptions of them. The venture team members assisted in modelling by reviewing and commenting the process models that were drawn. The actual motivation for the current state analysis was software process improvement i.e., to find ways to enhance and rationalise corporate venture's processes. The venture did not strive for a rigorous process model but rather wanted to get an overview of all the procedures and tasks they had performed, and wanted to eliminate unnecessary elements and tasks but still keep the process light.

Although the primary objective of this study was to assist process enhancement, we discovered that the data were also suitable and interesting for this study of agile software development.

The case description and case analysis were based on the interview data, process models and other collected materials. For analysis, it was impossible to organise the data statistically because the interview questions had a process-centred point of view instead of agile software development point of view, and because not all the interviewees were asked the same questions. Instead, we studied the transcripts carefully several times and used the conceptual agile framework for analysing the data.

6.2 Reliability and validity of the study

One of the most difficult issues in reviewing qualitative research is to prove the validity and reliability of research. In qualitative research it is not easy to differentiate data analysis and review of reliability (Grönfors 1982, 173). The concepts of reliability and validity are derived from quantitative research. Therefore their applicability to qualitative research can be questioned. (Hirsjärvi & Hurme 2001, 186) This, however, does not mean that the validity examination could be omitted. The structural validity (thus researcher has to report how he or she classified the findings and why) has more importance than other forms of validity in qualitative research. Reliability in qualitative research refers to how well all the data have been taken into account, whether it is transcribed correctly, and the how well the results reflect respondents thoughts. It should be noted that the results of interviews are always consequences of the co-operation of researcher and respondents. (Hirsjärvi & Hurme 2001, 188-189)

The goal of qualitative research is not to make statistical generalisations, it is rather to understand a certain incident, action or research field and make meaningful theoretical interpretations of those. Generalisations of qualitative research are based on rational and logical data collection, carefully considered sample and the review of research method. It is important to study the research subject and data collected of it so well that theoretically solid perspectives can be created. (Eskola & Suoranta 1999, 66)

According to Yin (1989), case studies provide very little basis for scientific generalisation but he adds, "case studies are generalisable to theoretical propositions and not to populations or universes" (Yin 1989, 21). Eskola and Suoranta (1999) are of same opinion. According to them, the generalisations of qualitative research are not made based on the actual data but on the interpretations of it. The analysis of data and interpretations are only the basis for generalisations. Generalisations in qualitative research can be considered through transferability. It means that some of the theoretical concepts used for example in this study can be used in analysis of some other agile in-house development method. (Eskola & Suoranta 1999, 65-68) Elements and relations found in this study can therefore be transferred as hypothesis to account for other cases of agile in-house software development.

Because the original focus of the case study was software process improvement and modelling, not agile software development, it affected the question setting and also the data that were collected. We might have missed some relevant pieces of information because they were not directly addressed in the interviews, thus this might set limitations for the validity of our results. Nevertheless, we had altogether several hundred pages of transcripts, so the amount of data was relatively large, thus providing a sufficient basis for making our conclusions. Moreover, as the data were collected without using the agile framework, the data were unbiased and based on how the corporate venture really worked and behaved. Thus, the interviewees did not have the change to polish their answers to give an unrealistically agile impression of their work. The interviewers did not have knowledge of the agile framework either, so the question setting did not cause bias. These issues strengthen the validity of our results.

7 CASE STUDY

As stated in chapter 5, the most commonly used agile methods are developed within companies, totally from scratch or by modifying existing defined methods according to Charette (2001d, 2). To study in-house development from agile point of view, we conducted a case study, which we will use to explore different aspects related to agile in-house software development.

First, we will first describe the case organisation: the development team and its stakeholders and then we will describe the product and the development process. The case description includes chapters 7.1-7.3. Chapter 7.4 starts the case analysis based on the agile framework.

7.1 Organisation

Sometimes it is difficult to draw a line between a development project and a corporate venture. However, in the light of definitions by Vesper (1999) as well as Block and MacMillan (1993) in chapter 2.9 we state that the development effort we researched in this case study was a corporate venture rather than a development project. For instance, the studied venture's tasks included technical development but also productising the software and taking care of marketing and legal issues. Such a broad view for one small group was not necessarily a new but uncommon activity to the parent organisation. The venture was rather autonomous. Only the fact that the venture did not have a profit-and-loss responsibility speaks more for being a conventional development project than a venture. Hence, we consider the studied development effort a venture, which was working on a software product development project.

The parent corporation of the corporate venture was a large multinational high technology company. Like its parent corporation, the venture too had to face the challenges of this high velocity business environment. Being a corporate venture is advantageous in a sense that parent corporation is likely to provide different kinds of resources to the venture. Such can be material (e.g. facilities, hardware and software, marketing material) or financial resources. In this case, the corporate venture had a

chance to utilise parent corporation's expertise on various areas, such as marketing or web-functions. The parent corporation also assisted the corporate venture in legal issues, such as contracting. In addition, the parent corporation provided office facilities.

7.1.1 Venture team

The corporate venture consisted of six team members. One person, the team leader, was responsible for financial issues and he also handled the contracts with subcontractors and partners. The product manager was responsible for product management and product launch activities, such as issues considering product definition. Three team members coordinated programming work and did some implementation work too. They all had strong technical background, which was closely related to and well utilised in this development project. One of them was also responsible for the project management. One team member focused on technical research, such as investigating possible installation tools for the developed product for example. All the team members had various other tasks to take care of, for instance one team member organised the development of sample applications for the product. From now on we will use the term *venture team* when referring to this corporate venture and its employees.

The venture team collaborated with other business units inside the parent corporation, and additionally they practised outsourcing. The venture team's work was mostly to coordinate the product development process but they also did some software design and implementation, although the majority of the programming work was done outside the venture team. As there were many parties involved, it was essential to keep the collaboration effective. FIGURE 13 illustrates the structure of the organisation. The objects within the triangle belong to the parent corporation. In the following subchapters we will discuss different parties in more detail.

7.1.2 Product

The venture team developed a software product with Java technology. The main distribution channel for the product was WWW; customers that were located worldwide could download it from there. In addition, the software was distributed on CD-ROM's.

The customers were divided in two categories. There were general customers that could download the product from WWW without any charge, whereas "superior customers" paid a fee for the product, and were provided with some extra service such as help desk services. Financial resources from the parent corporation were essential for the venture team, given that they did not receive much income from selling the product because only few customers paid an annual fee for the product, whereas others downloaded it without charge. Thus, there was no revenue model in use at that time.

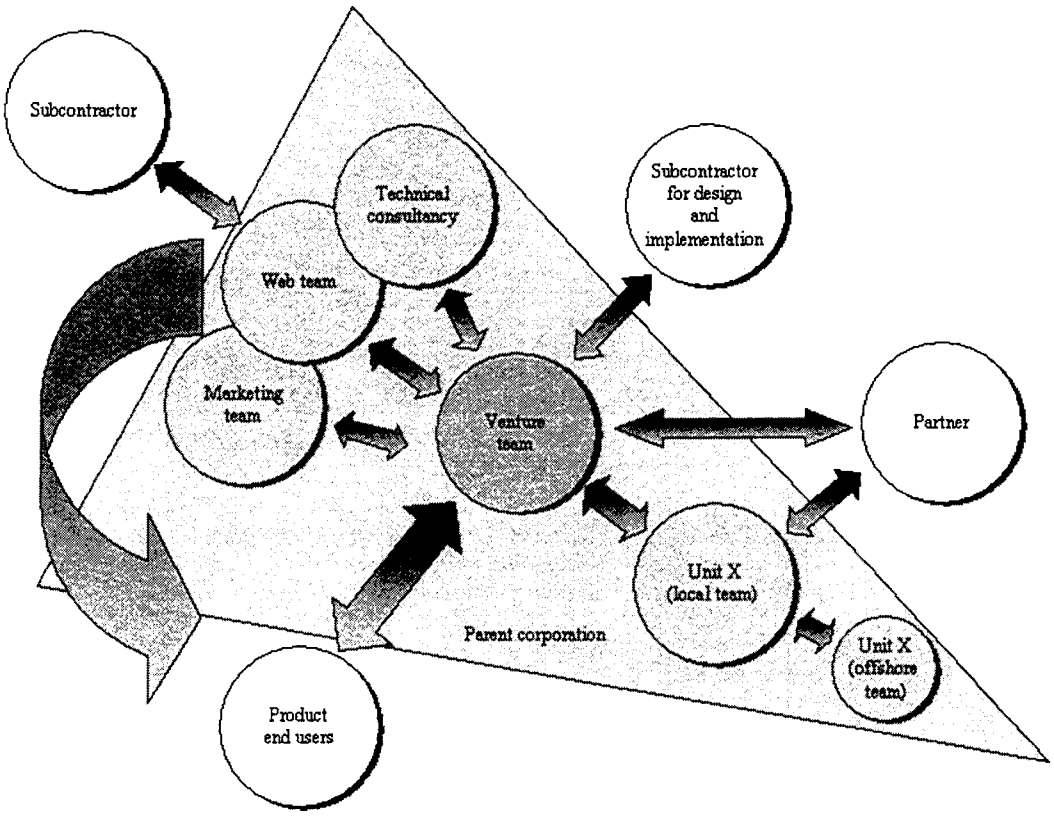

FIGURE 13 Organisation structure

The product that the venture team produced was a joint product development effort in collaboration with a well-known technology provider and software developer located abroad. From now on we will refer to this company as *the partner.* The product consisted of three separate parts. Development of the product core was subcontracted outside the corporation but the work was done in very close cooperation. The product core had some functionality in it and in addition, the two other parts were integrated to

it to form the whole product. Another business unit (hereinafter we will call this *Unit X*) within the corporation developed the first major exterior part of the product. Unit X's local team took the main responsibility for this part, closely collaborating with its offshore team. There was one person working in Unit X's local team and he mainly did programming work of the first exterior part for this project. The second exterior part was developed by the partner. The following picture illustrates the composition of the product (FIGURE 14).

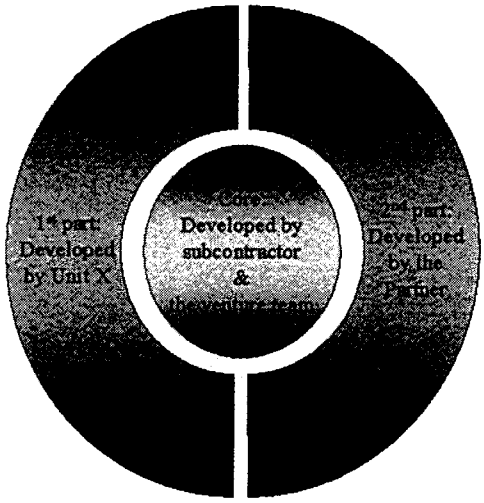

FIGURE 14 Composition of the product

Technical requirements and limitations for the product core were drawn to rather large extent from specifications of a particular technology that was used as the platform of the product core. Requirements for different parts of the product were captured for each part separately. Requirements for integration of these different parts to fit together through interfaces were discussed collaboratively between different stakeholders. The venture team had collected a list of requirements for the first exterior part of the product in the beginning of the project. The venture team members as well as Unit X's programmer added new feature proposals over time. Unit X's programmer then decided whether to implement feature proposals or postpone them to later releases. The development process was rather open to changes, and changes were implemented even in quite late

stages of the project. As the first part had to be integrated to the second exterior part developed by the partner, change requests were also discussed with that company.

7.1.3 Project stakeholders

Here we describe the project's stakeholders and their relationships and work distribution with the venture team (see FIGURE 13).

Web team

The parent corporation provided the venture team assistance in web functions. As the main distribution channel for the product was WWW, it was natural for the venture team to collaborate with the web team that belonged to the same corporate department as the venture team. The web team took care of issues related to web publishing, for instance releasing the product in the web, and developing and maintaining the web page. The web team cooperated closely with the marketing team and technical consultancy that also assisted the venture team. This collaboration further facilitated the cooperation between different stakeholders involved in the project. The web team was rather busy, as it was responsible for web-functions of several other units of the corporation. They outsourced some of their functions.

Marketing team

The marketing unit of the parent corporation assisted the venture team in marketing and other issues related to productisation. For instance, the marketing team took care of product's visibility with press releases and newsletters, by taking care of marketing issues on the web site and by producing promotion slides for various events. They also planned suitable distribution channels for the product, and took care of internal marketing.

Technical consultancy team

The technical consultancy team provided technical support and consultancy both for the product development and for the end-users of the product. Their task was to develop the technical solution for publishing the product in the web in collaboration with the web

team. In addition, the technical consultancy team gathered the customer feedback and responded to it if possible, or otherwise redirected feedback to the party that was responsible for the concerned issue.

Other stakeholders

The venture team had contacts with other units of the parent corporation too. They got assistance in handling intellectual property rights and other legal issues. This was very useful especially because of the difficulties in legal issues, as the legislations and the working styles of the lawyers differ in the venture team's home country and the partner's home country.

A department of the parent corporation specialised in usability checking checked the usability of the product. People from other units participated in field-testing, whereas system testing was subcontracted to an external company. Spelling and language checks for the product documentation (such as user manuals and tutorial texts) were also subcontracted to an external company.

7.2 Communication

The venture team members knew each other well, and the personal relationships between them were good. The venture team members were working in an open workspace, so communication between team members was rather straightforward. In addition to informal discussions, the venture team held weekly meetings in which relevant topics, such as current state and next steps of the project, were discussed. According to the team members, there was not enough time to document what was decided in the meetings. Instead of writing documents, the team members started to implement the discussed issues right away.

> *Because of our tight schedule, our meetings felt like fire fighting. We didn't have time to document stuff [between the meetings]. After the meeting we went and implemented something, discussed it together and went implementing the agreed changes.*

Thus, the communication was based on face-to-face meetings and tacit knowledge, not documentation.

Besides having face-to-face meetings, phone calls and email were frequently used in communication between the venture team and project's stakeholders. If the cooperation was running smoothly, meetings were not all that regular but if there were problems with the partners or subcontractors, tighter meeting and communication rules and practises were demanded by the venture team.

The venture team's task was mainly to coordinate the whole product development process. The progress of different stakeholders' outputs was constantly one of the venture team's main concerns. Different parties implemented each of three parts of the product that the venture team then integrated. In addition, the venture team had to productise the product, so also tutorial documents and sample applications among other things needed to be produced. The venture team had to take care of intellectual property rights, and marketing had to be organised and coordinated. Moreover, as the product was published and delivered via WWW, web-related issues needed to be coordinated as well as technical support considering the product.

Web team, marketing team, and technical consultancy team operated in the same building as the venture team. Being relatively close to each other naturally enhanced communication and cooperation, and meetings were easy to organise. Communication between the venture team and these other teams was mostly very informal but efficient.

The venture team collaborated closely with the subcontractor that developed the core part of product. This collaboration was facilitated by the nearby location of the subcontractor, as subcontractor was working approximately one kilometre away from the venture team. It was relatively easy to organise face-to-face meetings when ever necessary. The subcontractor used prototyping approach, and development was done iteratively. The progress of development was monitored in one-week cycles, as the venture team and the subcontractor got together in regular weekly meetings, in which the status of the project was checked. In addition to going through what had been done last week, a plan for the next weeks work was discussed. Tacit knowledge played an important role here, as no actual action plans were written. However, the subcontractor delivered a status report in every two weeks to the venture team's project manager, explaining for instance what had been done in last two weeks, and what was planned for the next two weeks.

It was not possible to arrange regular meetings with the partner. In addition to distance constraint, time zone differences caused inefficiency in communication. The venture team often had to wait for an answer to an email query until the following day. On the other hand, the time difference also provided some benefits. The venture team worked during the day in Finland, and by the end of the day delivered their work to the partner whose employees were just starting the workday. By the next morning, the venture team often had a reply waiting. In a way, this was working in two shifts, which increased efficiency.

7.3 Product development

The initiative to develop the product was made in the higher management level as a part of the corporate strategy. A clear need for a product like this was found on the market, and this was one of the main motivations to develop the product. Additionally, it was seen that this need was related to a certain technology that was expected to become very commonly used in the future. Thus, the venture team developed a product that supported this technology. In addition to coordinating the software development activities, the venture team had to coordinate marketing, plan and design the look and feel of the product, and take care of intellectual property rights issues. Productisation also included organising the product launch, which took place in a large international conference.

The venture team started the product development from scratch, and they did not have any formal, documented process to follow. A project plan set the goals that the venture team pursued. The parent corporation had in-house development methods available but those did not suit the needs of the venture team. The working methods of team members arose from the previous projects they had worked in, thus they had an unwritten, implicit process model based on their tacit knowledge to follow instead of a formal process. Thus, based on our definition of *skilled improvisation* (in chapter 2.7), the working mode of the venture team can be characterised as skilled improvisation. The venture team's main goal was to be flexible, and develop the product within the strict schedule as effectively as possible. One interviewee stated: "We want a process that guides our work but is loose enough to give us enough freedom". Based on the

conducted interviews, process models were drawn of the venture teams work. An example of process models is presented in Appendix 3.

The venture team did not use any particular method in software development. Although no formal process or methods existed to assist their work, the venture team had some templates from parent corporation that they were able to utilise to support their development efforts. For example, there was a document template that could be used in creating customer documentation, and a template for legal disclaimers.

7.3.1 Project management

The venture team's team leader said that when the business decision about developing the product was made, he knew a rough estimate about the budget and the schedule for the project. The goal was to launch the product in a large international conference that was organised approximately six months from the project start:

> *I guess our business decision was something like this: in [month n] we knew approximately how much money was allocated for this [project], and that it [the product] should be completed in six months. And we knew that the product would be this required software product.*

Team leader added that they knew what some of the competitors had done in this area, and an estimation of what those competitors could be doing during the next six months. No real business plan was written; it was rather derived from parent corporation's strategy. The approaching deadline set time constraints and pressure that had to be met. Project plan was written but most of the planning concerning different activities or technical solutions was made in the meetings when discussing about what should to be done. Not much of those decisions or plans were actually documented. Already from the beginning the product was planned in a way that would facilitate developing other products that share similar qualities in the future. Thus, idea of deploying a product line in the future existed. However, this case study concentrates on the development of the first release of the product.

As being a corporate venture, some decisions required acceptance from higher management levels but the venture team had still quite a lot of autonomy to make decisions. Bigger decisions had to be approved by the upper managers of the

corporation; small decisions could be made independently. For instance in contracting issues, the financial significance of the decisions, and whether the external parties were trusted and well known or not, had an effect on who could make the final decision to contract with the concerned party. The venture did not have the power to decide for which markets their product was targeted at, the markets and other strategic objectives were decided by the parent corporation. Signs of the venture team's autonomy were the freedom to define their own way of working and division of tasks within the team and the possibility to choose their employees.

7.3.2 Requirement and change management

The venture team had an agreed procedure to manage requirements, although it was undocumented. They did not use any specialised requirements management software system. Instead, they used excel-sheets for capturing and communicating requirements throughout the project. These excel-sheets were circulated among the involved stakeholders one at a time, and they added new requirements and feature proposals to the file. Then it was sent to the next stakeholder and finally back to the venture team. After this the venture team got together in a meeting in which they discussed the proposed requirements, and decided which ones should be implemented next. As circulating information via email is sequential in nature, the venture team was not able to utilise parallelism in this case.

Throughout the project there was one person responsible for managing these excel-sheets. This procedure was considered to have worked well for this project, as the software under development was not very large or complex. Nevertheless, the venture team indicated that they would possibly benefit from the use of a requirements management system, especially if the developed software becomes more complex and more stakeholders and partners become involved in the future.

In addition to the requirements, the bugs found in the software were documented in the same excel-sheet, as well as issues concerning change management. The team did not prioritise bugs based on criticalness or importance but fixing them or making other changes was rather done intuitively based on the knowledge and experience of the team

members and their capabilities to distinguish the relevant needs from the irrelevant. Thus, no specific change management system was used but the small size of the software and small number of involved parties reduced the need for such software.

> *Interviewer: How do you feel about using excel-sheets for reporting the found bugs? Would you have needed a software tool, software for version or change management?*
>
> *Subcontractor: I don't think we would have needed a tool because the number of people involved was limited. There were only a few people who made changes to the file. And circulating it via e-mail worked OK.*

However, it became clear in the interviews that some more formal procedure for managing changes effectively would be necessary for future projects:

> *Interviewer: So the excel-sheet worked well?*
>
> *A team member: In this type of project it did. But if the software is really massive, then you'd need a tool because you'd have hundreds or thousands of bugs.*

Potential requirements management tools or other tools should also be available for the stakeholders outside the parent company. The subcontractor stated:

> *I don't know whether some tools would have helped. But a shared network drive with access from different locations would have been useful. Because of all these firewalls it was difficult to arrange that. But I think it could have been useful.*

Improvement suggestions concerning the changes and new requirements came up along the product development process, and the feedback was constantly taken into account to the extent possible. As the schedule was tight, the venture team found it important to prioritise the feedback. All the change requests and requirements were viewed but not all were implemented. The most critical changes and requirements were implemented first, whereas the "nice-to-have" changes and requirements were postponed to be implemented in later versions of the product.

There was no version control system in use, which was seen as quite a big weakness. The team did not document details of any build (e.g. which versions of the different components work well together), so tracking changes was very difficult.

> *Let's say [we would be working with the fourteenth build and] we would try to figure out which version of the Unit X's part, which version of the core and which version of the second exterior part worked together in the tenth build. No idea! I'm sure you could find the information somehow but some kind of documentation should exist.*

Thus, the venture team did not use any specialised software systems for requirements management, change management or version control. However, they considered such tools important for future projects.

> *Version control would have been absolutely necessary [when integrating different parts to form a new build]. [...] If the new build didn't work properly, it would have been useful to return to a previous build.*

7.3.3 Implementation and integration

The subcontractor developed the software iteratively and used prototyping, too. First he developed a prototype; functionality came along as the development work went on. There were no clearly separated design and implementation phases but they were done in parallel. The venture team was closely involved in this development process, and the progress was monitored in weekly meetings.

Integrating the different parts of the software was the venture team's task. This was an iterative process to some extent, although "not the kind of iterative process that might be presented in schoolbooks", as one interviewee said, and it can also be described as ad hoc. A new build of the product was finished in the end of each iteration round. Given that the product was a combination of different components produced by different stakeholders, getting a new version of any of the components led to making a new build. About fifteen main builds were made within two weeks.

In practice the integration was organised as follows. A lab was built in a room near the venture team's open workspace. Two powerful computers were set up in that lab and connected to personal workstations via network. New versions of different parts of the product were usually delivered to the venture team members via electronic mail. They were then transferred to a network drive so that it was possible to access them from the lab. All tasks that were linked to the integration were done in this lab.

The venture team used a shared network drive, which enabled each team member to access the files. When a new build was made, it was recorded on a CD-ROM and then delivered to the involved stakeholders (e.g. field-test group). Several CD-ROM's were recorded while making the builds, which was very time-consuming.

7.3.4 Testing

Field tests, usability checks and system tests were made for the product in different phases of product development. Field-testing was started in quite an early phase whereas system testing was done in quite a late phase, just before the product launch.

Field-testing

Field-testing was started as soon as it was seen necessary. The decision and elementary planning for testing were made in weekly meetings. These plans made were not documented in any test plan.

First separate parts of the product were tested independently, which can be considered as unit testing. Unit X tested the part it developed without formal testing procedures or plans. The software of this part was rather small and Unit X's programmer tested the code along the implementation. As bugs were found, the improved code was immediately tested. No clear test reports were written from these tests either. Lacking decent reporting method among the stakeholders and inadequate communication led to some overlapping testing and ineffective use of resources. Also the partner tested the part they developed. The partner wrote some reports that were difficult to interpret, however. Thus, even though the testing information was shared between stakeholders, the results were not very useful for the venture team or Unit X.

Several field test rounds were organised to test the compatibility of different parts. First such testing was quite informal but as the separate parts of the product became more developed and improved, it was possible to organise more comprehensive testing. In this case these testing practises worked quite well but the venture team members stated that a testing template and other guidance would have been very useful to guide the testing process.

Interviewer: Do you have any ideas how the testing could have been made more efficient?

Unit X's programmer: The first step would probably have been documenting a basic collection of tests that should be done for each release. I mean, certain applications or operations or stuff. [...] Then you'd remember to perform all the same tests every time. [...] And the next step would have been automating them [the tests] somehow, so that running the tests would have been easier... you'd just press a button and see whether the code passes the tests.

First field-test rounds were made within the venture team and by a few people from the technical consultancy team that assisted the venture team. More people got involved in testing as the product became more developed. The venture team got testing assistance and resources from other units of the corporation, and also some external companies took part in field-testing. The venture team categorised some loose test cases and directed them to different people, some people for instance tested the product in different versions of operating systems.

System testing

System testing was subcontracted, and it was much more systematic than field-testing. System test plan was written and a comprehensive test report delivered to the venture team. Three weeks were allocated for system testing. First there was a basic testing round in which the functionality was tested. Initial feedback from the first round was given on excel-sheets and transferred to the venture team by email. The first two weeks of system testing were considered as the most critical ones, since the bugs and defects found during that time were possible to be fixed before the launch date. Finally a regression test was conducted. The venture team did not review the results of the regression test prior to product launch. The bugs found on this round were documented but not fixed for the first release. Additionally, the validity of documentation in respect of the functionality and for instance sample applications was tested.

Checking usability

Usability checking was subcontracted to another department of the parent corporation, as the department had expertise on this area. Cooperation with this department worked fine. While the schedule to launch the product was tight, usability checking had to be

started while the software was not very developed yet. Usability checking was performed as the software was constantly evolving. However, starting testing and usability checks early was considered necessary.

Next we analyse the venture team's work and relationships with its stakeholders in the light of the agile framework presented in chapter 4.3.

7.4 Individuals and interactions over processes and tools

Principles 1, 4, 5, 6, 8 and 12 of Agile Manifesto emphasise paying attention to individuals and interactions rather than slavishly following certain process and using tools. In our case study, all the individuals involved in the product development were highly competent. If some of them had left the project, the venture team would most probably have been able to find some other people to replace the people who had left, which is partly consistent with the traditional role oriented thinking.

> *In this [subcontractor's] case, being dependent on more or less one person doing the programming work wasn't a [serious] risk [because we had back-up]. Since there are other projects done with the subcontracting company simultaneously and other competent people working [there], who would have been able to continue the subcontractors work if he had been for example shifted to some other project.*

More importantly, however, the interviews revealed that the project's success was highly dependent on individuals' competences, and thus this agile value statement was valid in this case.

7.4.1 Working environment and motivation

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

All the venture team members had worked in the parent corporation before, and they were familiar with the corporate culture. One venture team member described the parent corporation environment as being quite open in nature: employees were encouraged to express their ideas. Thus, this environment supported creativity and sharing of ideas.

The venture team was given relatively lot of autonomy for their work. The working setting for this the venture team was rather different compared to the setting team members had experienced in their previous projects in the parent corporation. Many of them had worked in different departments participating in large projects, in which significantly less autonomy was provided. The interviewees found the given responsibility, freedom and autonomy highly motivating and enjoyable.

> *It should be more like... something like in 'concurrent engineering'... letting people participate starting from an early phase [of a project] and give people responsibility for their own work. It might sound like a funny comment but that's the way it is. It shouldn't be taken for granted.*

Nevertheless, even though the venture team had quite much autonomy, it was still part of a large corporation that had the ultimate decision-making power, thus decision-making and agility that can be gained from being autonomous were limited to some extent.

> *We made many decisions on our own, we didn't ask anyone for authorisation. We presumed that if we'll launch a proper product, it'll be OK. But when we felt that there was a lot of money or high risks involved, we had to find the person who could make such decisions [that we couldn't make ourselves].*

As we mentioned, the team members did not follow any tight process, nor were they given strict demands concerning the productisation. This flexibility gave them a chance to make different kinds of decisions concerning the technical solutions and details of the product launch for example. The venture team members found that challenging but also rewarding because it forced them take a broader view on product development, instead of just concentrating on the technical implementation or other specific and narrow areas. Hence, the broad view of product development and productisation and freedom to make decisions related to it were motivating factors. However, the venture team members were of opinion that in the future more control and management would be needed to enable proceeding that is more effective.

> *To have a quite loose process is really good because it will not suffocate the employees. However, later on some control and management is needed to become more efficient. I've being doing this kind of work for about four years and this [working in a corporate venture] has been quite exceptional, not having a clear plan, not having everything given but you'll have to find it yourself. I think it's been really valuable.*

The venture team members developed a product that was small in size and targeted to end-users whose needs the venture team members could understand. Previously the team members had worked in large projects and they were implementing tiny parts of massive entities. Now they were responsible of a whole entity, a product that was small enough to comprehend and handle. The team members found it appealing to work on something they could entirely understand and relate to.

> *You can imagine a developer using our product on his or her PC, so it feels quite concrete.*
>
> *The end-user is incredibly close in this work. And we are closer to the other companies [and stakeholders], too. When comparing with the past, we just implemented a small part of a massive software for some company customers. Here we are a few steps closer to the end-user.*

In addition, all the venture team members were personally cooperating with different stakeholders, both corporation's internal and external parties. The venture team members considered this motivating because they could very closely follow the progress of the different stakeholders.

To conclude, our case study shows strong support to this principle as motivation and environment were found very important success factors.

7.4.2 Communication

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

Frequent and continuous communication among the venture team members enabled their agile way of working. The team worked in an open workspace, which facilitated straightforward communication. In addition to informal and occasional communication, the venture team held weekly meetings. Also the stakeholders whose presence was considered important participated. Weekly meetings were considered very suitable, one team member stated:

> *At least I considered [the weekly meetings] as an easy way to monitor the project's progress. We met weekly, face-to-face, and we discussed together what to fix and things like that. These kinds of follow-ups...nothing formal... We just had a good feeling about it, we made quick decisions about what kind of changes or features we wanted to implement [next].*

Face-to-face communication did not only facilitate the venture team's internal communication but also communication with stakeholders. The progress of subcontractor's work was also monitored in the weekly meetings. Face-to-face conversations provided a favourable setting for such monitoring. In addition, the subcontractor had meetings with the usability expert. This way the subcontractor got direct feedback and could continue working based on it. He also received feedback by email. The usability expert emphasised the importance of personal contact. He said:

> *Personal contact is always important. Even if there had been a shared database for gathering feedback, or other tools that do not necessarily require human interaction, I would have insisted having personal contact to give feedback.*

As mentioned, requirements management was handled using excel-sheets. The person responsible for managing these excel-sheets often discussed personally with people who wanted to fill in new requirements or report bugs. He said:

> *But that's the way it is, you have to discuss those [requirements] anyway. It is not enough to fill in a description into a database [...] it still requires some further discussions of the details. People are more willing to do it on the phone than type it with an awkward tool. Maybe such tool might have given some assistance but it wasn't at any way necessary in this [case].*

Communication between the venture team and Unit X was truly intense throughout the project. Emails and phone calls took place nearly everyday. No regular meetings were organised but occasional and informal face-to-face meetings took place when necessary. One venture team member stated:

> *Communication between Unit X and us was tight and effective... It was vital for the success of this project.*

Electronic mail and phone were also actively used with the partner because it was impossible to organise regular face-to-face meetings with them. Communication with the partner was mostly handled via electronic mail, and conference calls were organised weekly. Unit X's representative said, that conference calls played an important role in this project. He stated:

> *In this kind of project in which you have to work quickly, and the schedule is tight, I think it [making conference calls] was necessary. If it's only email, it easily happens that not every issue is mentioned. In a phone conversation*

> *somebody may slip something important that the other party might have ignored by mistake. It was good.*

One possibility to get a bit closer to the face-to-face setting would have been organising videoconferences. The issue was only discussed with two interviewees, and they, however, did not consider videoconferencing useful. The interviewee from Unit X said:

> *I don't really see that videoconference would have brought any added value to this. Of course it's nice to see the faces of those people once but I don't see what extra it would have brought to the conversation... In my opinion, in a technical discussion videoconferencing is not very important.*

To summarise, face-to-face communication took mainly place within the venture team, and occasionally with the other stakeholders. In addition, phone conversations were intensively used. Hence, face-to-face, or at least "voice-to-voice" on the phone, communication was an important, if not a crucial factor to the venture team's success, and thus this principle of Agile Manifesto was strongly supported by this case.

7.4.3 Business people and developers

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

In this particular case, we can analyse this principle from different angles. The decision to develop such product came from the higher management level of parent corporation, and the need for such product was in a way technical since the product was developed to support a certain technology that was expected to become commonly used in the future. Thus, there was a customer (parent corporation) that demanded that such product should be developed, and that it should support a certain technology but did not provide further detailed requirements for the product. Besides this technology requirement, the customer was not involved in development. This was not seen problematic at all; on the contrary, it gave the venture team free hands to make autonomous decisions.

If then again, we consider business knowledge in general and ignore the customer setting we can interpret this principle from the venture team's internal viewpoint. As mentioned, most of the venture team members had a strong technical background but also business knowledge was found within the team. The team leader was responsible for financial issues, and also the product manager handled mostly business-related

issues. Thus, the venture team consisted of developers and business people, who worked together daily throughout the project.

Interpreting this principle can also be done from the stakeholder point of view. The web team and technical consultancy team can be considered as technical people, whereas the marketing team was more business-oriented. The collaboration with the marketing team took place in the late phase of the project. The representative of the marketing team said:

> *I don't find it [being involved in the project from the very beginning] necessary. In the beginning of the project it is mainly technical stuff that is discussed, so I don't see any added value for us to be involved in that. Still, it would be good if they'd inform us about new projects. It is good for us to know that a new product will be developed. So that we can prepare ourselves for it but to sit down with those guys in the meetings gets us nowhere. When the product starts to take form that is already something, one can shape what it is, and to whom it is targeted, and what features and other stuff... then it is worth for us to sit there and start thinking about it.*

Thus, the venture team did not collaborate with the marketing team throughout the project but on the other hand it was not seen necessary either.

The actual end-users of the product could be held as customers and thus business people. However, the end-users were not actively involved in the development of the first release but their feedback and wishes will be used in the future releases.

Hence, this principle is ambiguous and it can be interpreted from different viewpoints. Although this case did not strongly confirm or contradict this principle, we find that the fourth principle is indeed important, and should be acknowledged and followed especially in cases where a customer and a software supplier in a business relation can be clearly distinguished.

7.4.4 Sustainable pace

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

The venture team members had to work overtime every now and then, and that contradicts with the eighth Agile Manifesto principle. The product manager mentioned:

In this project [lacking people] wasn't really a resource constraint, since everybody was committed and ready to work overtime if necessary.

Other team members reinforced this statement. However, the venture team members did prioritise their tasks in order to limit the overtime, so the notion of the importance of keeping up sustainable pace was recognised but also quite easily disregarded in the pressure of getting the product done within the schedule. One venture team member stated:

> *It [workload] was pretty extreme at some point. We wouldn't have been able to go on like that much longer.*

Unit X's programmer worked really hard for this project. Even though the part he developed was not very complex, the workload for this employee was heavy and workdays were long. In addition to this project he was working in other projects at the same time. He was strongly committed to get the part done in time but as he stated, the workload was too heavy for one person concurrently working in other projects as well:

> *It [workload] was so much, that it would have been a whole-time job for somebody. But I had also other things to do, so I then had too much work altogether. [...] It wasn't good at all. It wasn't nice.*

To conclude, the importance of sustainable pace was recognised but not followed. Although this project was tight, it was done in a relatively short period of time and the pressure did not burden the venture team's members too long. However, as it seems that there will be new releases of the product and new projects, they cannot keep the hectic pace on in the future.

7.4.5 Tuning performance

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

Principle 12 of Agile Manifesto suggests teams to observe their own work, and tune and adjust the behaviour accordingly. As mentioned before, team members found weekly meetings important for monitoring the state of the project. Getting together weekly enabled the team members discuss their work, what they had done and what they were going to do next, and at the same it provided them a possibility to improve their work. This principle receives strong support by our case study, too.

7.5 Working software over comprehensive documentation

Working software over comprehensive documentation is emphasised in the agile framework. Working software is seen as the most reliable measure of progress. Agile Manifesto principles 1, 3, 7, 9, 10 and 11 support this value statement. In the following we analyse the case through these principles.

7.5.1 Delivering working software frequently

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

Design and implementation of the software was done iteratively. The people involved in the implementation process, both from the venture team as well as from the subcontractor, found this iterative method a good way to work. Weekly follow-up meetings made it possible to constantly keep the team up-to-date on the project's status while the subcontractor demonstrated his latest work. One interviewee stated:

> *In weekly meetings we discussed what had been done and what ought to be done by the next week's meeting. With these week-long cycles, we did not have time to forget what was planned, and we could see the progress quickly for example when we brought in new features... It was easy to monitor the project; you did not need to read long texts in the documents. Seeing concretely what is happening... that makes it easy.*

7) Working software is the primary measure of progress.

Frequently delivered software was also used as a measure of progress in the weekly meetings. This enabled gathering feedback that was used to guide the next phases of the development. As a whole, this procedure was seen as a very suitable and effective way of working. One team member stated:

> *We saw the results of tasks that were started last week [in the weekly meetings]... we didn't forget anything. We were on top of things all the time... When we brought in new features and so on... It was easy to monitor the [progress of the] project, you didn't have to read any papers. When you concretely see what's going on... that's what I mean by easy!*

Thus, our case study supports principles 3 and 7.

10) Simplicity -- the art of maximising the amount of work not done -- is essential.

Simplicity, referring to implementing only the agreed and important features was not directly supported by the interview data. But we observed a problem related to fast pace of development and simplicity. Usability expert said that the rapid evolution of the product caused some extra work.

> *Interviewer: The product was evolving pretty quickly. Did this cause any problems in usability checks?*
>
> *Usability expert: A bit, of course, since some extra work was done [which later turned out to be obsolete as the software became further developed]. However, the nature of this project [tight schedule] set the working environment, and altogether the project went fine.*

Thus, the fast pace development and testing an evolving product caused extra work. In a more general level, keeping the focus of the venture narrow was seen important. The venture team leader stated:

> *A narrow focus is very important, like for a start-up in general. You can think of a business idea very quickly but productising something into big business will increase the work-load hundred or thousand times more than when making the first demo version. [...] Communication between two people is quite simple but when there are more people, it gets more and more complicated. It's pretty much the same with developing competence for people that you've recruited... at the same time you get more problems to solve. You're running and running but your speed and efficiency are decreasing all the time.*

Thus, keeping the focus supports this Agile Manifesto principle, although some extra work was done due to fast development.

7.5.2 Technical excellence: competence and testing

9) Continuous attention to technical excellence and good design enhances agility.

The venture team members were highly skilled employees. The venture team's product manager said that when they recruited employees they wanted to keep high standards, so they only hired competent people.

> *We have tried to keep up a [high] level of competence. We have insisted that the people we hire are not just anybody.*

Recruiting competent people offered better possibilities to accomplish the product within the schedule but also to develop a product of good quality. We did not analyse the product and its quality but the product manager stated:

> *Of course the quality of the product is a high priority... Customer feedback has mainly been positive... We haven't received many notices of defects from the technical consultancy team, which indicates to relatively high quality of our product. [Even from the beginning] we believed in it [that we could develop a high quality product] because the project was quite small and there were experienced people involved.*

The venture team consisted of a combination of technical, business, and research competence. The tasks of the project were divided between the venture team members so that personal competences of each person were taken into account. For instance, persons with a strong technical background concentrated on the technical issues.

The subcontractor who implemented the core part of the product was an extremely competent programmer who wrote extraordinarily bug free code. His competence was seen as a very important factor to the success of the project. Hence, the competence of the individuals enhanced agility and the quality of the product in this case. In addition, the competences of the team members seemed to complement each other, which provided firm conditions for successful teamwork.

Agile Manifesto does not mention testing whereas XP puts a heavy emphasis on continuous and thorough testing. In addition, we included testing in technical excellence

principle in chapter 4.2.9. In this project testing was done partly ad hoc. One of the interviewees commented:

> *The main bugs could be found quickly, which was good but one cannot reach faultlessness with this method. However, the tight schedule in this case set the limits, so ad hoc testing could be evaluated to be the most effective method in this case.*

As testing was not thoroughly planned and executed, technical excellence principle from testing point of view was not completely fulfilled in this case. This was also acknowledged by the venture team:

> *Generally said, we need considerably more systematic testing.*

Nevertheless, these testing practises were found sufficient. Thus, we state that technical excellence was emphasised in this case and it increased agility although systematic testing did not take place. This principle was supported by the case study.

7.5.3 Self-organising team

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

Communication was rather straightforward due to the small size of the team. The venture team members considered steering groups unnecessary for their work, one team member stated: "This group guided itself". From "jelled team" point of view we can state that this principle was strongly supported by the case study: the good team spirit and excellent personal relationships increased agility.

7.6 Customer collaboration over contract negotiation

Agile Manifesto values customer collaboration and states that close relationships between the software development team and the customer can be more important than contracts. Principles 1, 2, 4 and 8 of Agile Manifesto refer to this value statement. Once again, the customer perspective can vary (see chapter 4.1.3). End-users can be seen as customers but as the corporate venture was developing a COTS product, this value statement is not valid for this viewpoint.

7.6.1 Contracting

When looking at the venture and its stakeholders, the venture was the customer, who specified what the stakeholders should do for this project. This setting included the web team, technical consultancy, the marketing team, Unit X, the subcontractor, and the partner. The venture team leader said that there were no rules for choosing a subcontractor or partner:

> *I don't think that choosing partners is science. There are no rules how to create partnerships. What you can do, is to make a guess how the future ecosystem will look like and which players it will have and then think with which partners you could end up in a win-win situation. After that you can approach the [potential] partners and try to see how it goes. Maybe 80% of those attempts will fail, hopefully 20% will succeed.*

Contracting was more case-specific. Some subcontractors were chosen based on personal contacts, some were found on the basis of customer references and some subcontractors were determined based on chosen technologies. In many cases, work was started before the contracts were negotiated and signed, so trust played an important role in subcontracting and partnering issues. The venture team leader stated:

> *It's always so that there is a certain amount of risks that both parties take as well as a certain level of trust at the same time from both sides. We had discussed with this subcontractor that we'd pay a certain amount of money if they started working. I heard that it was a week ago when the contract was signed. There was a gap of few weeks [between starting to work and signing the contract] but I know that they'd taken a risk and started working already.*

Nevertheless, managing style differed with stakeholders. If the venture team saw a risk considering some unit, subcontractor or partner, this collaboration was managed with tighter control. If there was enough trust, less control was required. Hence, the collaboration and working methods with different stakeholders varied case-by-case. The venture team's product manager stated:

> *To understand how others are working is the key to happiness. When you understand it and are able to adapt to it, things will work.*

Thus, in the product manager's opinion, recognising the wishes and working methods of collaborative parties was essential to be able to proceed efficiently.

7.6.2 Customer satisfaction

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

Principle 1 speaks for delivering valuable software to the customers frequently to promote customer satisfaction. The development cycle for the venture team's first release was six months and they stated that the forthcoming releases should be delivered within three to six months. Thus, the venture team followed this Agile Manifesto principle. However, from the COTS perspective delivering parts of the system is not valid. Rather a whole product, or at least a beta version like in this case, should be delivered first. A beta version can be complemented later with bug fixing modules or small features that can be downloaded from the WWW.

This principle can also be discussed from a different perspective, referring to the venture team being the customer when considering their collaboration with different stakeholders. The venture team had close relationships with most of the project's stakeholders. The subcontractor delivered his products (pieces of code) weekly, and according to the venture team members, the quality of the code was excellent. This speaks for delivering valuable software continuously. Also Unit X delivered new versions of their part very often, which further reinforces following the principle.

The support to the first principle can be discussed from many angles. Although continuous delivery could not be followed as this principle suggests, satisfaction of the end-users was indeed important. As the close collaboration with the end-users in capturing requirements was impossible, the venture team was assisted by the other units in this task. The venture team communicated daily with some stakeholders and at least weekly with others. As described in chapter 7.1, the organisation culture supported open communication, and the units operating within the corporation were free to express their opinions and change requests:

> *It's actually quite common here [in the parent corporation] to encourage people to give comments on pretty much everything. At least that's how I feel. It's a different story, who makes the final decisions but you can always give comments.*

The open communication culture and frequent interactions with different stakeholders facilitated collecting requirements and also "welcoming changing requirements".

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

Iterative software development enabled adding requirements and their implementation incrementally. Feedback was also gathered throughout the implementation from field tests, and it affected the requirements.

Thus, the venture team was willing to listen to other people's suggestions to be able to develop a better product. The suggestions were respected but with some reservations, as it was not possible to implement every feature proposal in project's tight schedule. The second principle was strongly supported by our case study. We will return to this principle in chapter 7.7.2.

7.7 Responding to change over following a plan

Agile Manifesto on no account neglects the importance of planning. It emphasises that development teams should make plans and follow them, while constantly reflect the plans to the current situation and change them accordingly. This value statement is supported by Agile Manifesto principles 1 and 2, and principles 3 and 7 are related to reacting to change through frequent delivery of code. Principle 12 also partly relates to this value statement. Here we will analyse the case most thoroughly from principle 2 point of view because the other principles have been discussed earlier in this chapter.

7.7.1 Planning

A project plan was written in the beginning of the project. It set high-level goals and schedule for the venture team's project. More specific planning was made in the weekly meetings. This procedure worked fine within the venture team but when it came to collaboration between some other teams, more planning would have been necessary. An interviewee from the web team mentioned that a systematic milestone procedure presenting clear "web points" would have assisted the web team to plan their work better. The web team was busy with numerous projects, so getting information about

forthcoming deadlines in advance would have been essential for coordinating their work. Interviewee from the web team stated:

> *Milestones would help a lot. We would need to have some general information in the beginning and more detailed information towards the end. That would help our own planning and resourcing, and we could inform our subcontractors as well. If our subcontractor can't do it, we can inform the project that this schedule doesn't work. We don't want those "we gave out a press release of our new product today, do something!" Then all our other projects will be delayed. In addition, we'd like to have a meeting face-to-face or on the phone every week or every other week to discuss the state of the project. Just for 15-30 minutes.*

As we have pointed out, agile does not mean chaos. Being agile does include planning to certain extent. In this case there was obviously need for more detailed planning when collaborating with the stakeholders. Thus, it can be concluded that adaptive planning can assist the collaboration between different parties but does not have such an important role within a small project team.

7.7.2 Changing requirements

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

Agile Manifesto principle 2 suggests welcoming changing requirements, even late in the development. Sometimes such requirements might come from the customers; sometimes they can be technology dependent. This development project had to face an unexpected change in technology that they were using. A part of the product was developed using a commercial platform, which changed during the last few months of the project. The software had to be changed accordingly in the final stages of the project. The subcontractor stated:

> *Right at the end we noticed that oops, this platform was not the same as a few months ago. We dealt with it then. We managed to fix it and deliver a package that worked on that new platform configuration.*

Another example of responding to changing environment in this case was naming the product. The venture team had to change the product name due to intellectual property right issues in the very late phase of the project. This was a laborious process, since all the documentation and code had to be updated. The venture team members stated, that it

would be ideal but almost impossible, to fix the product name in an early phase of the project. Hence, the venture team members noticed the importance of being flexible and were able but under these circumstances also forced, to accept the change and act upon it. Thus, this case study strongly supports the second principle.

7.8 Summary

To sum up the above discussed issues we have gathered the results in TABLE 5. Character + means support and ++ strong support. +/- means that the case partly supported the principle but was partly contradictory.

TABLE 5 Case study's support to Agile Manifesto principles

Principle	**Support found in the case study**
1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.	+
2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	++
3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.	++
4) Business people and developers must work together daily throughout the project.	++
5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	++
6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	++
7) Working software is the primary measure of progress.	+
8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.	+/-
9) Continuous attention to technical excellence and good design enhances agility.	+
10) Simplicity – the art of maximizing the amount of work not done – is essential.	+
11) The best architectures, requirements, and designs emerge from self-organizing teams.	++
12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.	++

As the case analysis reveals, the values and principles of Agile Manifesto were observable, although the venture team did not choose to follow them deliberately. This confirms the ideas presented earlier that agile development is very much based on practises and ways or methods of working that experienced people have. Those practises arise from tacit knowledge and intuition if the circumstances are favourable. In chapter 2.7 we hypothesised that skilled improvisation is agile. Our case study supports this hypothesis: according to our findings, working in skilled improvisation mode can be regarded as agile. The parent organisation and excellent team spirit in the venture team supported agility. A lightweight process with a remarkable amount of face-to-face communication between the different stakeholders made this product development a success.

Although all the Agile Manifesto principles were observable in the case, not all of them were followed in practice. For instance obeying sustainable pace was recognised but neglected.

As the team's task was to coordinate software development that brought more pressure to communication and coordination between different stakeholders. Collaboration with several stakeholders increased the need for detailed planning and specific division of tasks. The venture team members were satisfied with having little rules and little management but they would have wished for more guidance and defined processes to be more effective. Also testing was found problematic but very important in this case study. Ad hoc testing was seen suitable for this small project but the venture team members considered more systematic testing necessary especially for future releases.

Thus, in this case skilled improvisation was seen inadequate in planning and testing. Skilled improvisation assisted the agile way of working but was not enough and should be complemented with some more formality in for instance planning. Agile Manifesto does not address these matters directly. XP provides rather detailed guidelines for testing which could have been utilised in this case.

8 MODIFIED AGILE FRAMEWORK

In this chapter we discuss the main limitations and shortcomings of Agile Manifesto and the agile framework (presented in chapter 4.3). We aim to enhance the conceptual agile framework and reformulate some Agile Manifesto principles according to our findings from literature and our case study.

8.1 Limitations of Agile Manifesto and the agile framework

Several concepts concerning agile software development were not clearly defined nor systematically used in existing literature. Terms such as methodology and method were used as synonyms, and agile software development was compared to process models and to development methods although in our opinion agile software development is mainly related to development methods. Additionally, some values and principles of Agile Manifesto seem quite idealistic and vague, not that realistic. Principle 11 is most imprecise in our opinion. Maybe Agile Alliance tried to say too many things in one statement without being too specific and ended up saying nothing.

Such vagueness derives partly from the fact that the agilists have not really defined a context for their statements. For instance the terms *business people* and *customer* were not well defined, although those terms or roles can highly vary in different business modes. In addition, "the big, bad enemy" of the agilists, traditional methods, was not explicitly defined and identified in the literature.

Agile Manifesto and literature concerning agile software development have not thoroughly discussed the use of software tools, for example requirements management software, and their role in agility. The venture team in our case study used excel-sheets for managing requirements, reporting bugs and changes. The team members were of opinion that in this project that was sufficient but if the software becomes more complex and larger and feedback would be received from multiple sources, software to manage requirements and changes, and version control software would be necessary. The lack of software tool discussion in literature does not necessarily mean that the topic is neglected by the agilists. However, we want to emphasise this issue because it

was highlighted in our case study. Software tools are important for agile software development but development teams should carefully consider what kind of tools would be most useful for their particular projects. As in our case study, excel-sheets and circulating them via electronic mail can be feasible but in some other circumstances, such as operating with multiple subcontractors and partners or developing large and complex software, more formal and comprehensive tools might be needed.

In our case study, we found that when software development is performed by several parties, it brings more pressure to communication and coordination. If there are several different stakeholders and complex dependencies between their contributions to the development, more formality is needed in management and communication. Even in such circumstances face-to-face communication is important but by no means enough. In addition, collaborating with multiple parties sets higher requirements for planning and documentation, as information has to be transferred from one party to another, and as all the activities have to be coordinated to be as efficient as possible.

8.2 Developing the modified agile framework

We start developing an enhanced agile framework by defining a clearer context for agile software development. Agile Manifesto and agile methods are most suitable for tailored projects in which the customer is another company or another department from the same organisation. Especially the customer perspective distinguishes the use of Agile Manifesto in different business modes. Early and continuous delivery of valuable software is not possible as such in COTS mode. Pieces of software are not useful to an end-user whereas whole products or even beta-versions are.

Business people perspective is also a distinguishing issue. In COTS mode, business people most likely refer to people responsible of marketing and sales within the same department or company. In tailored and MOTS modes, business people can have above mentioned roles or business people can be representatives of the customer.

For our modified agile framework principles 1 and 2 do not need to be changed. In principle 3 we shorten the length of the minimum development cycle to one or two days instead of a couple of weeks. Such short cycles were emphasised in XP, Microsoft's

daily build practise and also in our case study. The third principle should also highlight iterative and incremental development, and we add the statement "Use iterative and incremental approach to accomplish this" to the principle. The 4th principle is left as it is. The 5th principle should further emphasise experience and skills of the employees, and thus we add "highly skilled" as one important criterion for selecting employees for agile teams. As our case study revealed, collaborating with multiple parties requires more control and formalism in communication in order to coordinate activities and spread information, face-to-face communication alone is not sufficient. Hence, we change the 6th principle by adding a new statement: "When collaborating with multiple parties, more formal communication is necessary". 'Parties' refers to project's stakeholders including subcontractors and partners. We leave principles 7 and 8 unchanged. In principle 9 we emphasise testing as a means to accomplish technical excellence. We leave 10th principle as it is. The 11th principle is so vague that we omit the principle. Architecture and good design are already discussed in principle 9. 'Self-organising team' is now mentioned in the new 11th principle (the former principle 12). We also add using of (software) tools to principle 11. The feasibility of the tools that a development team is using should be given thought in regular intervals. For example, an excel-sheet for requirements management can be fine when the software is small but if the software becomes more complex, a more advanced software tool should be taken into use.

Based on the discussion above, we alter the principle of Agile Manifesto to be more precise. The changes are marked with **bold font**.

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

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

3. Deliver working software frequently, from a couple of **days** to a couple of months, with a preference to the shorter timescale. **Use iterative and incremental approach to accomplish this.**

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

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

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. **When collaborating with multiple parties, more formal communication is necessary.**

7. Working software is the primary measure of progress.

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

9. Continuous attention to technical excellence, good design **and testing** enhances agility.

10. Simplicity – the art of maximising the amount of work not done – is essential.

11. At regular intervals, the **self-organising** team reflects on how to become more effective, then tunes and adjusts its behaviour **and tools** accordingly.

Positioning of the principles on the two dimensional framework changes so that the new 11^{th} principle becomes also technical as the software tool perspective is closely related to technical issues. The former 11^{th} principle (The best architectures, requirements, and designs emerge from self-organising teams) will be omitted. The new positioning is illustrated in the following figure.

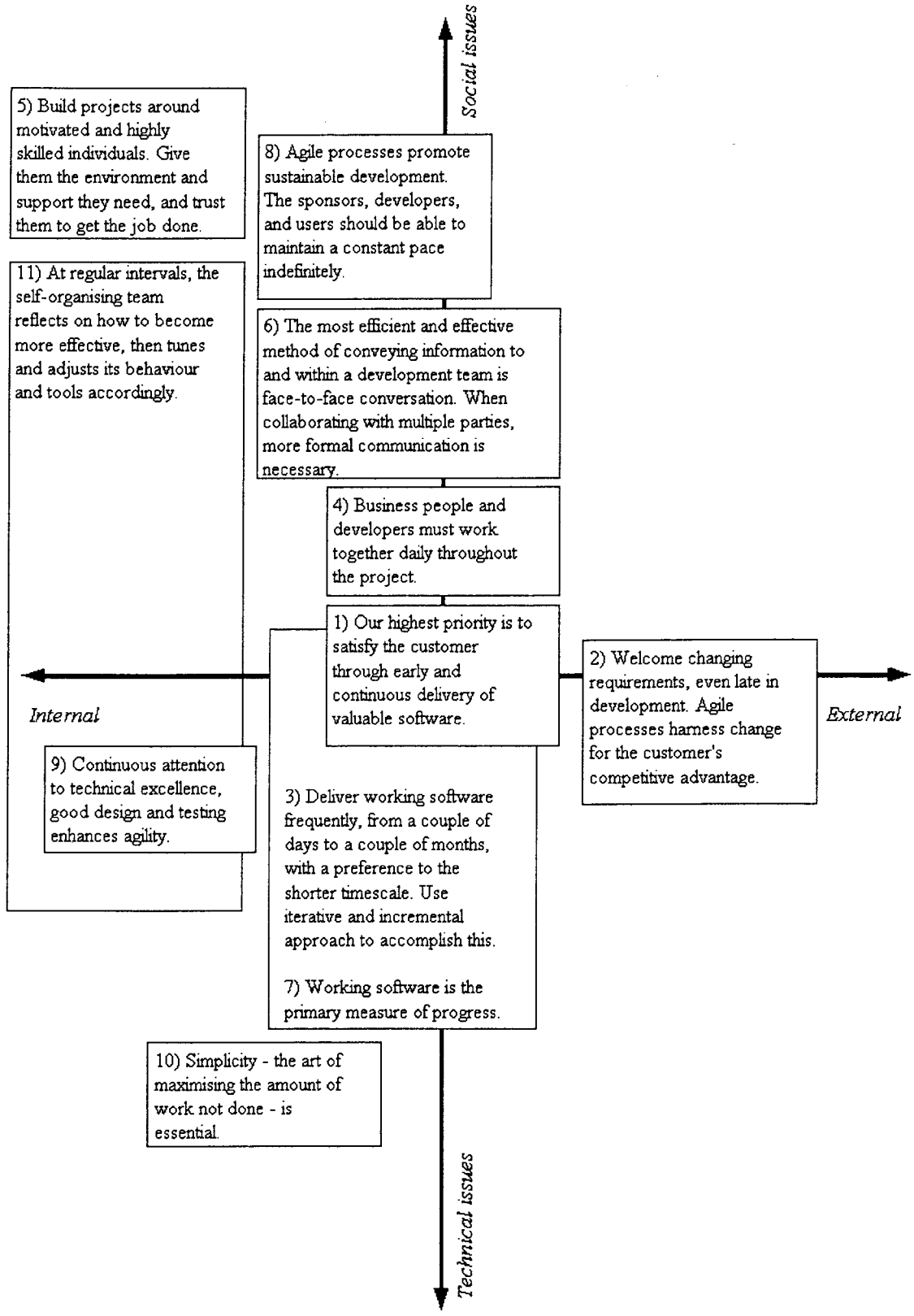

FIGURE 15 The modified conceptual agile framework

9 CONCLUSIONS

We studied the evolution of software development in the third chapter. Systems life cycle thinking and standardised process models among others replaced ad hoc type of development of the 1950's and 1960's. Traditional development methods became heavy and infeasible in certain situations as for example the amount of bureaucracy increased. In addition, traditional methods and process models (such as CMM) have been considered unable to respond to changes in organisational, business and technical environments, and unable to acknowledge individual talent and creativity. Since the turn of the millennium, the trend is to some extent changing from traditional methods to agile development methods. Those were developed as a response to heavy methods and additionally changes in technical and business environments resulted in the use and invention of agile development methods.

Agile Manifesto is a collection of values and principles concerning agile software development. Those were analysed in chapter four. The authors of this thesis positioned the values and principles to a conceptual agile framework. The framework can be utilised to get an overview of Agile Manifesto, its values, principles and their interdependencies. In addition, the agile framework can assist in evaluating the principles from two dimensions: teams' internal versus external activities and social versus technology oriented issues. Agile Manifesto and the agile framework are realised in different agile methods, such as in Extreme Programming (XP).

The underlying objectives of Agile Manifesto are customer satisfaction, frequent communication, emphasis on individuals and response to change. We found out that actually none of the ideas presented in Agile Manifesto were truly innovative. The most obvious heritages of traditional methods in agile software development are iterative and incremental development. However, Agile Manifesto gives a quite holistic view on software development, in which not only technical issues are of importance but also customer satisfaction and human well-being are strongly emphasised. In our opinion, the main contribution of Agile Manifesto is its holistic view on software development that aims at fast and light software development while maintaining high quality.

One of the purposes of this thesis was to explore enabling and limiting factors for the use of agile methods. This was done based on literature review. In chapter 4.4 we found out that agile methods can especially be used in projects where new and innovative solutions are developed with very tight schedules and changing requirements, and in small organisations that develop innovative products. Large organisations and organisations that are undertaking massive development projects with high quality and safety requirements are most likely to use traditional methods in stead of agile methods. However, traditional and agile methods can also be combined to gain the best results. When making decisions concerning which methods to use, general method selection considerations, for instance criticality in security and time performance, are valid. In addition, the nature of requirements, innovativeness and the possibility to hire skilled personnel should affect the method selection.

We analysed two agile methods to explore how the agile ideas are realised in these more practically oriented methods. The two analysed methods were XP and agile in-house development. In the analysis of XP we found that XP practises are very well in line with the Agile Manifesto principles. They support and complement each other so that Agile Manifesto gives a more general view on different issues and XP provides more practically oriented guidelines and suggestions. XP has a more technical orientation than Agile Manifesto, which was revealed for instance by the comparison of the agile framework and the XP practises framework. We also studied experiences of XP in practice, and we discovered that the most experiences were quite positive and successful, especially those concerning pair programming.

In our analysis of agile in-house software development we empirically studied a corporate venture that developed a software product. The venture team of this case study did not use defined software development methods, neither did they have process models to follow. Instead they based their work on their skills and knowledge gained from previous projects, thus their working mode was skilled improvisation. We analysed the case study in comparison with the principles of Agile Manifesto and discovered that although the venture team did not consciously follow Agile Manifesto, all the principles of it were supported by the case study. Thus, it can be concluded that

working in skilled improvisation mode is similar to the agile way of working. Working in skilled improvisation mode has several aspects in common with Agile Manifesto.

In chapter 8 we discussed the main limitations of Agile Manifesto and agile software development and developed the modified agile framework. We provided a more detailed context definition for agile software development and altered some of the principles of Agile Manifesto to be more precise or comprehensive.

9.1 Practical implications

Since agile methods highly emphasise tacit knowledge and face-to-face communication, it sets many personality related requirements for the development team and for the management. The individuals participating in agile mode of development have to be communicative, collaborative and willing to discuss and share ideas. This should be considered when starting a project and recruiting the development team. It appears that agile methods do not facilitate employment of inexperienced people very well for example due to the lack of documentation and heavy emphasis on tacit knowledge. Although for instance XP's pair programming in particular can support hiring new, inexperienced people, the whole team's agility is most likely to decrease as the more experienced have to spend their time on helping the inexperienced instead of making productive results. These reasons indicate that inexperienced employees would not be very valuable to an agile team.

Process descriptions and strict plans are often ways for the management to control the status of the project. In agile mode control and evaluation have to be done by trusting the development team and by evaluating their results based on code. This then again is based on the presumption that the manager is a very technically oriented person in addition to being a facilitator of communication, collaboration and teamwork. Evaluating the produced results (pieces of code or a running build of several code components in most cases) is also expected from the customer. Thus, special technical skills and understanding are also expected from the representatives of customers.

Because agile methods aim to minimise the number of documentation and plans, they put a lot of pressure on individuals. Individuals are expected for instance to write their

code in a way that it can be easily understood without documentation. There is also a lot of responsibility, developers have to be very apt to evaluate the current state of the development project and how to proceed. On the other hand, responsibility might encourage developers, while they have more control and responsibility over their own work.

Neglecting the need for sustainable pace (Agile Manifesto principle 8) prevails in many companies and departments within the IT-business today and actually in the whole society. Too often work pressure forces people to the edge of burnout even on the cost of their own mental and physical health. The team members of the venture team of our case study acknowledged the need for sustainable pace but despite that worked overtime in practice. This seems to be a general tendency, people know that workaholism is not healthy but keep pushing themselves into their limits.

Corporate venturing and agile software development made a good match in our case study. A large parent corporation can provide circumstances that support working in an agile way, for instance by assigning highly competent employees to the agile team or by giving financial support. Then again a corporate venture can, by being agile, obtain several goals that the parent corporation has set to it. A venture might be able to start the business more quickly and start bringing income to the parent company compared to a traditional start-up company.

9.2 Theoretical implications

Although Agile Manifesto did not present any truly innovative ideas, many ideas of it are valuable and important in software development. Agile Manifesto has not been thoroughly discussed and validated by the academia. However, this thesis is an objective and versatile study of agile software development. Additionally, the discussion has started (for example in Beck 1999, Boehm 2002, Cockburn & Highsmith 2001, Paulk 2001) and it is becoming increasingly active and deep, and such discussions are most likely to bring progress to agile software development. Ideas closely related to agile software development have also been presented in other contexts (for example Baskerville & al. 1992, Brooks 1975, Cusumano & Selby 1998, DeMarco & Lister

1987, Nonaka 1995). Those somewhat different but complementary viewpoints should be more thoroughly studied and utilised in agile software development.

The main theoretical contributions of this thesis were establishing the term *skilled improvisation*, the study of Agile Manifesto and developing the agile framework. In addition, we compared Agile Manifesto with XP and with agile in-house development, and found that ideas of Agile Manifesto are used in practice. Moreover, we presented a conceptual agile framework and revised it based on findings from literature and the case study.

9.3 Limitations of this study

Literature concerning agile software development is quite limited. In addition, many articles and books concerning agile software development were written by the inventors of Agile Manifesto who also represent companies that market and sell commercial products and services related to agile software development. Thus, their opinions probably were not entirely objective but rather promotional and commercial. This limitation was kept in mind when analysing literature but it may have influenced our objectivity.

When formulating the conceptual agile framework we used two dimensions: internal versus external and social orientation versus technology orientation. We mentioned that business oriented versus technology oriented dimension could have been one other possible way to analyse Agile Manifesto. In addition, other topics like complexity of software could be included in the analysis, and they would affect the positioning of the principles in the framework.

As we conducted a single case study, it limits the extent to which the findings can be generalised. A multi-case study or survey would be more likely to provide results that could be more generalised. In addition, there are great differences in corporate venturing settings in different companies, thus our results cannot be generalised very widely.

9.4 Implications for further research

This research has proposed certain circumstances in which agile approach might be suitable. Scaling up agile approach to large teams and large projects is an interesting and challenging topic, which was not thoroughly analysed in this thesis. A vital question for numerous software companies is how to scale up to reach larger markets but to sustain agility. The following questions can be derived from this viewpoint: How can agile software development be utilised when the development is done in several different locations instead of one site? Thus, how does agile approach suit multi-site and multi-systems software development? Agile approach is mainly constructed from R&D (research and development) basis. How could agile approach be utilised in other parts and functions of an organisation, for instance in marketing?

Agile software development methods emphasise tacit knowledge. As our case study revealed, informal face-to-face discussions are not sufficient or possible when there are several stakeholders involved in development. This question is also relevant for scaling up aspect. Following question can be derived: How can the balance between tacit and explicit knowledge and their diffusion be found in agile software development when there are several parties involved?

We made a narrow evaluation of how some issues suggested in Agile Manifesto would suit different business modes (COTS, MOTS or tailored settings). The agile framework could be studied more thoroughly from the perspective of these different kinds of software products and their development. We also studied the suitability of agile methods for different aspects of software development (see TABLE 2 on page 22). However, this analysis was not very thorough, and thus would be an interesting topic for further research. How does agile approach support different stages of the software product life-cycle?

The agile framework was developed intuitively and no metrics were used to position the principles in different dimensions. How could principles be more precisely measured or valued? How could a more enhanced framework be developed? What other dimensions could be used in addition to mentioned technology versus business or complexity of the product?

Some people are not willing or capable of working in agile mode but rather want defined processes and bureaucracy to feel comfortable and to be efficient. How could agile approach be taken into consideration when recruiting personnel and allocating people into projects? Agile software development methods have been developed in the western society. It emphasises individuals and communication as well as collaborative skills. Such qualities are often associated with for instance North-Americans. Would agile methods be applicable for example in China or India?

We developed the concept of skilled improvisation. A more precise definition for it should be developed. This more comprehensive concept could be further studied in the agile software development context.

The agility of the case study can also be analysed from corporate venturing point of view. Based on our case study, we found that corporate venturing can strongly support agile in-house software development. First of all the parent corporation had the possibility to choose the members of corporate venture from their most skilled, experienced and motivated employees and form a highly skilled and motivated team that is essential for agile software development. Secondly, the parent corporation can offer a favourable business environment for the corporate venture as the venture can be supported for example financially, and the venture does not need to face the risks that for example start-up companies face in the beginning of the life-cycle. Thirdly, autonomy in decision-making is usually higher in corporate ventures than in usual departments or projects, and that autonomy can further increase motivation. Motivation can further increase agility. Thus, agile software development can, according to this case study, be suitable for corporate ventures for several above mentioned reasons. However, more case studies are needed to validate these statements. In addition, venturing and agile software development should more thoroughly examined. Can working in an agile mode assist a corporate venture in achieving good results early, in starting business, and in bringing income for the parent company? As corporate ventures usually go to new business areas and work with new technologies, they are most likely unable to utilise existing commercial or parent corporation's in-house development methods. Could Agile Manifesto and agile methods be a good starting

point for the corporate venture to start their development effort towards their own, efficient agile in-house software development method?

REFERENCES

Agile Alliance. 2002. *Agile Manifesto.* http://www.agilealliance.org/ [referred on 10.1.2002]

Agile Modeling. 2002. www.agilemodeling.com [referred on 5.3.2002]

Avison, D. E. & Golder P. A. 1992. *A tool kit for soft systems methodology.* Proceedings of the IFIP WG 8.2 Working Conference on the Impact of Computer Supported Technologies on Information Systems Development / Kendall, Lyytinen & DeGross (Eds.) Elsevier Science Publishers B.V., North-Holland. Pp. 273-287.

Baskerville, R., Travis, J. & Truex, D. 1992. *Systems Without Method: The impact of New Technologies on Information Systems Development Projects.* The Impact of Computer Supported Technologies on Information Systems Development / K. E. Kendall et al. (Editors). North-Holland, Amsterdam, pp. 241-269.

Beck, K. 1999. *Embracing change with Extreme Programming.* Computer, 13, 10. Pp. 70-77.

Beck, K. 2000. *Emergent Control in Extreme Programming.* Cutter IT Journal, 13, 11. Pp. 22-25.

Block, Z. & MacMillan I. C. 1993. *Corporate Venturing. Creating New Businesses within a Firm.* Boston: Harvard Business School Press.

Blokdijk, A. & Blokdijk, P. 1987. *Planning and Design of Information Systems.* Academic Press, London.

Boehm, B. 1981. *Software engineering economics.* Prentice-Hall, Englewood Cliffs, NJ.

Boehm, B. 1988. *A spiral model for software development and enhancement.* Computer, 21, 5. Pp. 61-72.

Boehm, B. 2000. *Spiral Development: Experience, Principles, and Refinements.* Special Report CMU/SEI-00-SR-08, ESC-SR-00-08, June [on-line]. Available at URL http://www.sei.cmu.edu/activities/cbs/spiral2000/february2000/SR08.pdf [referred on 20.2.2002]

Boehm, B. 2002. *Get Ready for Agile Methods, with care.* Computer, 35, 1. Pp. 64-69.

Brereton, P., Budgen, D, Bennett, K., Munro, M., Layzell, P., MaCaulay, L., Griffiths, D. & Stannett, C. 1999. *The Future of the Software.* Communications of the ACM, 42, 12. Pp. 78 – 84.

Brooks, F. P. 1975. *The mythical man-month: essays on software engineering.* Addison-Wesley, Reading, MA.

Brooks, F. P. 1987. *No Silver Bullet: Essence and Accidents of Software Engineering.* Computer, 20, 4. Pp. 10-19.

Charette, R. 2001a. *Fair fight? Agile versus heavy methodologies.* Cutter Consortium e-Project Management Advisory Service, 2, 13.

Charette, R. 2001b. *The fight is on? Agile versus heavy methodologies.* Cutter Consortium e-Project Management Advisory Service, 2, 14.

Charette, R. 2001c. *Fists are flying? Agile versus heavy methodologies.* Cutter Consortium e-Project Management Advisory Service, 2, 17.

Charette, R. 2001d. *The decision is in: Agile versus heavy methodologies.* Cutter Consortium e-Project Management Advisory Service, 2, 19.

Cockburn, A. & Highsmith, J. 2001. *Agile Software Development: The People Factor.* Computer, 34, 11. Pp. 131-133.

Cockburn, A. 2000. *Balancing Lightness with Sufficiency.* Cutter IT Journal, 13, 11. Pp. 26-33.

Cockburn, A. 2001. *Agile Software Development.* Addison-Wesley, Boston, MA.

Cockburn, A. 2002. *Agile Software Development Joins the "Would-Be" Crowd.* Cutter IT Journal, 15, 1. Pp. 6-12.

Coldewey, J., Eckstein, J., McBreen, P. & Schwanninger, C. 2000. *Deploying Lightweight Processes (Poster Session).* Conference on Object-Oriented Programming, Systems, Languages, and Applications on Addendum to the 2000 proceedings. Pp. 131-132.

Constantine, L. 2001. *Methodological Agility.* Software Development, June. Pp. 67-69.

Cooper, R. G. 1996. *New Products: What separates the winners from the losers.* In: PDMA Handbook of New Product Development. Rosenau, M. D. Jr. et al. (Eds.) John Wiley & Sons, NY.

Covey, S. R. 1994. *The Seven Habits of Highly Effective People: Restoring the Character Ethic.* Simon & Schuster, NY.

Creswell, J. W. 1994. *Research design: Qualitative and Quantitative Approaches.* Thousand Oaks: Sage Publications, Inc.

Crystal Family. 2002. http://crystalmethodologies.org/ [referred on 24.1.2002]

Cusumano, M. A. & Selby, R. W. 1998. *Microsoft's Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People.* Simon & Schuster, NY.

Cusumano, M. A. & Yoffie, D. B. 1998. *Competing on Internet time: Lessons from Netscape and its battle with Microsoft.* The Free Press, NY.

Cusumano, M. A. & Yoffie, D. B. *Software Development on Internet Time.* Computer, 32, 10. 1999. Pp. 60-69.

DeMarco, T. & Lister, T. 1987. *Peopleware: productive projects and teams.* Dorset House, NY.

DSDM. 2002. http://www.dsdm.org/ [referred on 4.2.2002]

Eischen, K. 2002. *Software Development: An outsider's view.* Computer, 35, 5. Pp. 36-44.

El Sawy, O. A. 2001. *Redesigning Enterprise Processes for e-Business.* McGraw-Hill, Singapore.

Eskola, J. & Suoranta, J. 1999. *Johdatus laadulliseen tutkimukseen.* 3. painos. Tampere, Vastapaino.

Extreme Programming. 2002a. http://www.xprogramming.com [referred on 18.1.2002]

Extreme Programming. 2002b. http://www.extremeprogramming.org/ [referred on 20.2.2002]

Fowler, M. 2001. *The New Methodology* [on-line]. Available at URL http://www.martinfowler.com/articles/newMethodology.html [referred on 18.1.2002]

Fuggetta, A. 2000. *Software Process: A Roadmap.* Proceedings of the 22nd International Conference on the Future of Software Engineering 2000. Limerick, Ireland. Pp. 25-34.

Glass, R. J. 2001. *Agile Versus Traditional: Make Love, Not War!* Cutter IT Journal, 14, 12. Pp. 12-18.

Grönfors, M. 1982. *Kvalitatiiviset kenttätyömenetelmät.* 2 painos. Porvoo: WSOY.

Haungs, J. 2001. *Pair Programming on the C3 project.* Computer, 34, 2. Pp. 118-119.

Hawrysh, S. P. & Ruprecht, J. 2000. *Light methodologies: It's Like Déjà Vu All Over Again.* Cutter IT Journal, 13, 11. Pp. 4-12.

Highsmith, J. 2000. *Retiring Lifecycle Dinosaurs.* Software Testing & Quality Engineering, July/August. Pp. 22-28.

Hirschheim, R., Klein, H. K. & Lyytinen, K. 1995. *Information systems development and data modelling: Conceptual and philosophical foundations.* Cambridge University Press, Cambridge.

Hirsjärvi, S. & Hurme H. 2001. *Tutkimushaastattelu.* Helsinki: Yliopistopaino

Hoch, D. J., Roeding, C. R., Pukert, G., Lindner, S. K. & Müller, R. 2000. *Secrets of Software Success: Management Insights from 100 Software Firms around the World.* Harvard Business School Press, Boston, MA.

Hornby, A. S. 1974. *Oxford Advanced Learner's Dictionary of Current English.* Oxford University Press. London.

IEEE Std 610. 1991. *IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries.*

Iivari, J. I. 1989. *Levels of Abstraction as a Conceptual Framework for an Information System.* In Information Systems Concepts: An In-depth Analysis / Falkenberg & Green (Eds.) Elsevier/North-Holland, Amsterdam. Pp. 323-352.

Jacobson, I. 2002. *A Resounding "Yes" to Agile Processes – But Also to More.* Cutter IT Journal, 15, 1. Pp. 18-24.

Jeffries, R. E. 1999. *Extreme testing.* Software Testing & Quality Engineering. March/April. Pp. 23-26.

Johnson, L. J. 1998. *A View From the 1960s: How the Software Industry Began.* IEEE Annals of the History of Computing, 20, 1. Pp. 36-42.

Karlsson, E-A., Andersson, L-G. & Leion, P. 2000. *Daily build and feature development in large distributed projects.* In Proceedings of the International Conference on Software Engineering, 4-11 June 2000, Limerick, Ireland. Pp. 649-657.

Kidder, L. H. & Judd, C. M. 1986. *Research Methods in Social Relations.* Holt, Rinehart and Winston, NY.

Kruchten, P. 2001. *Agility with the RUP.* Cutter IT Journal, 14, 12. Pp. 27-33.

Käkölä, T. 2001. *Research Program on Software Business* [on-line]. Available at URL http://www.cs.jyu.fi/%7Etimokk/ResearchProgramOnSoftwareVenturing17102001.pdf [referred on 25.4.2002]

Lyytinen, K. A. 1987a. *Implications of Theories of Language and Information Systems.* MIS Quarterly, 9, 1. Pp. 61-75.

Lyytinen, K. A. 1987b. *Taxonomic Perspective of Information Systems Development: Theoretical constructs and Recommendations.* In Critical Issues in Information Systems Research / Boland & Hirscheim (Eds.). John-Wiley, Chichester. Pp. 3-41.

Merriam-Webster. 2002. *WWWebster Dictionary and Thesaurus* [on-line]. Available at URL http://www.m-w.com/home.htm [referred on 9.4.2002]

Moore, G. A. 1995. *Inside the Tornado: Marketing Strategies From Silicon Valley's Cutting Edge.* HarperCollins Publishers, NY.

Mumford, E. & Beekman, G. J. 1994. *Tools for Change & Progress. A Socio-Technical Approach to Business Process Re-engineering.* CSG Publications, Netherlands.

Nawrocki, J. & Wojciechowski, A. 2002. *Experimental Evaluation of Pair Programming* [on-line]. Available at URL http://www.pairprogramming.com/ [referred on 1.3.2002]

Nonaka, I. 1994. *A Dynamic Theory of Organizational Knowledge Creation.* Organization Science, 5, 1. Pp. 14-37.

Nonaka, I. 1995. *The Knowledge Creating Company: How Japanese Companies Create the Dynamics of Innovation.* Oxford University Press, NY.

Norman, R. & Chen, M. 1992. *Working together to integrated CASE, Guest Editors' Introduction.* IEEE Software, 9, 2. Pp. 12-16.

Orlikowski, W. J. 1992. *The duality of technology: rethinking the concept of technology in organisations.* Organisation Science, 3, 3. Pp. 398-427.

Palmer, S. 2000. *Feature Driven Development and Extreme Programming.* Coad Letter 70 [on-line]. Available at URL http://www.togethercommunity.com/coad-letter/Coad-Letter-0070.html [referred on 5.3.2002]

Paulk, M. C. 2001. *Extreme Programming from a CMM Perspective.* Software, 18, 6. Pp. 19-26.

Paulk, M. C., Weber, C. V., Curtis B. & Chrissis, M. B. 1995. *The Capability Maturity Model: guidelines for improving the software process.* Carnegie Mellon University, Software Engineering Institute. Principal contributors and editors: Mark C. Paulk et al. Addison-Wesley. Reading, MA.

Plfeeger, S. L. 2001. *Effective Decisionmaking on Software Projects.* Cutter Consortium e-Project Management Advisory Service Executive Report, 2, 5.

Pickering, C. 2001. *Building an Effective E-project Team.* Cutter Consortium e-Project Management Advisory Service Executive Report, 2, 1.

Pressman, R. S. 1994. *Software Engineering: a practitioner's approach* (adapted by Darrel Ince). McGraw-Hill, London.

Rasmussen, J. 1986. *Information processing and human-machine interaction: an approach to cognitive engineering.* North-Holland, NY.

Rifkin, S. 2001. *What Makes Measuring Software So Hard?* Software, 18, 2. Pp. 41-45

Schwaber, K. 2000. *Against a Sea of Troubles: Scrum Software Development.* Cutter IT Journal, 13, 11. Pp. 34-39.

Scrum. 2002. http://www.controlchaos.com/ [referred on 26.3.2002]

Shafer, R. A., Dyer, L., Kilty, J., Amos, J. & Ericksen, G. A. 2001. *Crafting A Human Resource Strategy To Foster Organizational Agility: A Case Study.* Human Resource Management, 40, 3. New York. Pp. 197 – 227.

Sharma, P. & Chrisman, J. J. 1999. *Toward a reconciliation of the definitional issues in the field of corporate entrepreneurship.* Entrepreneurship Theory and Practice, Spring. Pp. 11-27.

Simons, M. 2002. *Big and agile?* Cutter IT Journal, 15, 1. Pp. 34-39.

Solatie, J. 1997. *Tutki ja tiedä: Kvalitatiivisen markkinointitutkimuksen käsikirja.* Helsinki: Mainostajien liitto.

Somogyi, E. K. & Galliers R. D. 1986. *From Data Processing to Strategic Information Systems - A Historical Perspective.* In Towards Strategic Information Systems. / Somogyi & Galliers (Eds.). Abacus Press, Tunbridge Wells.

Stephens, M. 2001. *The Case against Extreme Programming* [on-line]. Available at URL http://www.bad-managers.com/Features/xp/case_against_xp.shtml [referred on 19.3.2002]

Sutherland, J. 2001. *Agile can scale: Inventing and Reinventing SCRUM in five companies.* Cutter IT Journal, 14, 12. Pp. 5-11.

Sutton, S. M. Jr. 2000. *The Role of a Process in a Software Start-up.* IEEE Software. July/August. Pp. 33-39.

Taber, C. & Fowler, M. 2000. *An Iteration in the Life of an XP Project.* Cutter IT Journal, 13, 11. Pp. 13-21.

Tamminen, R. 1992. *Tiedettä tekemään! Opas yritysten tutkijoille.* Jyväskylä: Atena Kustannus Oy.

Tolvanen, J-P. 1998. *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence.* Doctoral Dissertation. University of Jyväskylä, Jyväskylä.

Vesper, K. H. 1999. *Internal Ventures.* In: The Technology Management Handbook. Dorf, R. C. (Ed.) Boca Raton, FL: CRC Press LLC. Pp. (1-) 26-31.

Warsta, J. 2001. *Contracting in software business: Analysis of evolving contract processes and relationships.* Doctoral Thesis. University of Oulu, Oulu. Available at URL http://herkules.oulu.fi/isbn9514266005/ [referred on 15.1.2002]

Wegener, H. 2001. *Human issues of Agile processes. Programming* [on-line]. Available at URL http://www.coldewey.com/publikationen/conferences/oopsla2001/agileWorkshop/wegener.html [referred on 20.3.2002]

Welke, R. J. 1981. *IS/DSS: DBMS support for information systems development.* ISRAM WP-8105_1.0, McMaster University, Hamilton.

Wiki. 2002. http://www.c2.com/cgi/wiki? [21.2.2002]

Williams, L., Kessler, R. R., Cunningham, W. & Jeffries, R. 2000. *Strengthening the case for pair programming.* Software, 17, 4. Pp. 19 -25.

Wong, S-P. & Whitman, L. 1999. *Attaining Agility At The Enterprise Level.* Proceedings of The 4th Annual International Conference on Industrial Engineering Theory, Applications and Practice, San Antonio, TX.

Yin, R. K. 1989. *Case Study Research: Design and Methods.* Sage Publications, CA.

Zahran, S. 1998. *Software Process Improvement. Practical Guidelines for Business Success.* London: Addison-Wesley.

APPENDIX 1: Agile Methods

Here we will briefly present Adaptive Software Development (ASD), Scrum, Crystal Clear, Feature Driven Development (FDD), and Dynamic System Development Method (DSDM) that are often cited to be agile. In addition, we shortly describe Agile Modeling. We will not conduct a deep analysis or comparison of them but rather present their main features.

Adaptive Software Development

Adaptive Software Development (ASD) is a lightweight software development method that accepts continuous change as the norm. The method follows a dynamic lifecycle, *Speculate-Collaborate-Learn,* instead of the traditional, static lifecycle, *Plan-Design-Build.* ASD emphasises continuous learning. It is characterised by constant change, re-evaluation, peering into an uncertain future, and intense collaboration among developers, testers, and customers. ASD was designed for projects that are characterised with high-speed, high-change, and uncertainty. (Highsmith 2000, 23)

Planning requires a certain degree of certainty to enable gaining the desired results. As Highsmith states, adaptation is a necessity in the new world in which change, improvisation and innovation reign. In such an environment it is difficult to plan to manage projects into innovative directions. This is why Highsmith prefers the concept *speculation.* Speculation consists of project initiation and cycle planning. Speculation does not exclude planning but it takes uncertainty into account, thus encouraging exploration and experimentation. For instance, carrying out short delivery cycles and iteration can do this. (Highsmith 2000, 23-24)

Collaboration is another concept that ASD's lifecycle emphasises. When it comes to complex applications developed in a turbulent environment, the volume of information flows and demands for knowledge requirements are high. This kind of environment strives for collaboration to manage project lifecycles. In ASD, collaboration and concurrency are the key words to develop working components. (Highsmith 2000, 24)

The third component of adaptive lifecycle is *learning* that also affects decision-making. "To become an adaptive, agile organisation, one must first become a learning organisation" (Highsmith 2000, 25). Learning comes as a result of feedback captured from quality reviews.

Adaptive lifecycle is characterised by being mission focused, component based, iterative, time boxed, risk driven, and change tolerant. A *mission* does not provide a fixed destination but rather boundaries for the project. In the context of adaptive lifecycle, *component* defines a group of features that will be developed. Components evolve as customers provide feedback by ongoing *iteration* cycle. *Time boxing* means setting fixed delivery times for iterative cycles and projects. The purpose of setting timely deadlines is to get people focus on making hard business decisions, not to burn employees out. Time boxing also urges the project team as well as the customers to monitor the project's state constantly. Analysing critical *risks* drives the plans for adaptive cycle. Being *change tolerant* at its best could be seen as being able to incorporate change as competitive advantage. (Highsmith 2000, 25)

Scrum

> *Scrum is a method that aims to help teams to focus on their objectives. It tries to minimise the amount of work people have to spend tackling with less important concerns. Scrum is a response to keep things simple in the highly complicated and intellectually challenging software business environment.* (Schwaber 2000, 35)

Scrum is based on two pillars: team empowerment and adaptability. *Team empowerment* refers to teams being relatively autonomous. Management gives them work to do but teams can choose their own ways to do that given work within each increment. Management's task is to provide teams freedom to work and remove the inefficiencies that restrict teams to improve their productivity. (Scrum 2002) Scrum teams get together daily in short meetings. This provides the whole team with direct feedback, thus reducing the amount of managerial bureaucracy. In Schwaber's opinion observing the interaction between team members is the best way to understand the complex interaction of people, technology, needs, competing interests, and satisfaction. In addition, bringing people together encourages collaboration and communication. These factors together strengthen the team as well as individual team members. Along

daily meetings the projects progress is constantly visible. This transparency provides benefits such as making the project's goals and their individual goals within the contexts of the project clear to all team members. In addition, transparency reduces the amount of time wasted on politics. Eventually Scrum may make teams becoming completely self-organizing and self-regulating. Finally the teams begin to own the projects that they work on. (Schwaber 2000, 36-37) Sutherland reports about a study of five projects employing Scrum. According to that study Scrum project planning enables teams achieving project goals faster than any other form of project planning reported to date. In addition, to achieve those project goals with Scrum project planning require also less administrative overhead than do other methods. (Sutherland 2001, 6)

Scrum supports iterative development. The projects are divided into "sprints" that form iterations, each of which last approximately one month. (Schwaber 2000, 37-38) *Adaptability* refers to equilibrium that the team maintains during each increment (i.e. sprint), as teams are insulated from outside disturbance (Scrum 2002). After each sprint the increments are punctuated, as the team and other interested stakeholder (such as users who usually are members of marketing, management, and alpha customers) gather together in a meeting in which the iteration is reviewed and what will be done during the next increment is planned (Schwaber 2000, 37-39).

Sutherland states that in all cases he studied, the results showed that Scrum improved radically communication and delivering working code. Scrum also proved to be scalable into programming in large, and that it fits to organisations with different environments. (Sutherland 2001, 10)

Scrum requires employees to be committed to the team and the project, while managers are expected to trust the employees. Thus, employees can concentrate on their work, as there is less need for meetings, reporting, and authorisation. Open working environment in which feedback is encouraged is one of Scrum's characteristics. Thus, tolerating mistakes is also recognised. An issue that may limit teams ability to adopt Scrum is that every team member is expected to understand every problem and all of the steps in developing a system to solve it. The aim is to utilise everybody's skills, intelligence, and experience. However, the odds are that this may limit the size of the system developed using the method. (Scrum 2002) On the other hand, companies might be willing to use

different method to develop a planned system than limit the size of the planned system because the method could not support developing it.

Crystal Clear

Crystal Family is a collection of methods developed by Alistair Cockburn. Different projects require different methods, and Crystal aims to provide a solution to these divergent requirements. Crystal is a human oriented approach to software development. It is based on a thought that people are the ones to make project succeed. Regardless the size of the project or its priorities, the amount of documents is supposed to be as low as possible, which on its behalf should minimise the level of bureaucracy. (Crystal Family 2002) People and communication centric approach is supported by the viewpoint that tools, work products and processes exist only to support the human component. Crystal methods also courage high tolerance, that is supported by recognising varying human cultures. High tolerance of Crystal also allows teams to adopt parts of other methods, for instance from XP. (Cockburn 2001, 201)

Crystal Family distinguishes methods that are suited to different subsets of project types. They are labelled as clear, yellow, orange, red, maroon, blue, and violet. Each of these methods is designed to different sized projects, yet enabling every team to set their own policies. However, as team size increases, team needs to shift their coordination and communication strategies. Crystal Clear is a method designed for small and agile teams. (Crystal Family 2002) We will briefly present the characteristics of Crystal Clear in the following.

Crystal Clear supports small teams that work in the same or adjoining offices, so dispersed teams cannot benefit from it. Separate people are needed in various roles that are sponsor, senior designer-programmer, designer-programmer, and user who participates in the project at least part-timely. These roles include tasks such as coordinating the project, being a business expert, gathering the requirements, and designing and programming. Thus, both technical and business skills are included in the roles. (Cockburn 2001, 202-203)

Incremental development aims to provide frequent delivery of software. The length of one increment is two to three months. (Cockburn 2001, 202) Iteration reviews are strongly emphasised in Crystal. In these iterations people can find problems early, which enables early correction of them. This requires the employees to monitor and improve the process, which further encourages processes to be self-improving. (Fowler 2001) Crystal Clear promotes tracking progress through milestones that consist of software deliveries of major decisions instead of written documents. Project documentation is required but the content of the documentation is indefinite. This provides team members a possibility to decide how they present their design notes. In addition, Crystal Clear promotes some amount of automated regression testing of application function. (Cockburn 2001, 203)

Direct user involvement is also encouraged in Crystal Clear. Users are expected to participate in two user viewings per release. As soon as upstream is considered to be stable enough to review, downstream activities start. In addition, product and method tuning workshops are held in the beginning and in the middle of each increment. (Cockburn 2001, 203)

Feature Driven Development

Feature Driven Development (FDD) is a model-driven process that focuses on short iterations. FDD consists of five processes: The process starts with domain and development team members working together with an experienced component/object modeller who develops an overall model for the development project. The established model is adjusted along the way. After this the development team builds a comprehensive feature list. Working together with domain experts, development team prioritises features and identifies which features must at least be included in the whole product. The features are then sequenced into a high-level plan, based on which the programmers can start working. Next two steps are done in approximately two-week iterations. *Design by feature* activity is followed by *build by feature*. This procedure promotes concurrent development within each increment. Iterations include design inspections, unit tests, integration, and code inspection before the chief programmer accepts the promoted features in the main build. Thus, FDD supports tracking and reporting progress with rather good accuracy. (Palmer 2000)

Feature Driven Development is highly iterative and result oriented process that aims to enable teams to deliver results quick without compromising quality. The focus is more on people than on documentation. FDD was designed to rather small teams but it scales up to larger project teams also. The teams consist of people having divergent backgrounds and competencies: business people (e.g. business experts/analysts and users) and technical people (such as domain experts/analysts and developers) work together. (Palmer 2000)

Dynamic System Development Method

Dynamic System Development Model (DSDM) follows a life cycle approach. DSDM lifecycle is iterative and incremental, consisting of five phases: *feasibility study, business study, functional model iteration, design and build iteration,* and *implementation.* (DSDM 2002)

Underlying principles of DSDM include active user involvement that takes both developers and users into the team, and empower the teams to make decisions. DSDM supports frequent delivery of products that meet the business requirements. In order to fully utilise feedback from users, DSDM calls for iterative and incremental development. Changes during development need to be reversible so the evolution of the products can be controlled. DSDM suggests requirements should be baselined at a high level. This means "'freezing' and agreeing the purpose and scope of the system at a level, which allows for detailed investigation of what the requirements imply." (DSDM 2002) More detailed requirements can be specified later but the primary scope should not be changed significantly. Incremental development allows also incremental testing. Thus, DSDM principles integrate testing throughout the lifecycle. Finally, DSDM encourages collaboration and cooperation between all stakeholders involved in the project at any way. (DSDM 2002) Fowler points out that DSDM emphasises high quality and adaptivity towards changing requirements (Fowler 2001).

Agile Modeling

Agile Modeling is a collection of values, principles, and practises for modelling software that can be applied on software development projects in an effective and

lightweight manner. This method provides an add-on on projects that use an agile approach such as Extreme Programming, Dynamic Systems Development Method, or Scrum. (Agile Modeling 2002)

Agile Modeling focuses on effective modelling and documentation. Other activities such as programming, testing, and project management are not included in Agile Modeling but those are derived from other methods, for instance XP. A project is started with some base process, and then it is tailored with Agile Modeling and other techniques that are considered appropriate. (Agile Modeling 2002)

The values of Agile Modeling are similar to those of XP, with a little extension. Effective *communication* between all project stakeholders is essential for modelling success. *Simplicity* is striven to develop the simplest solution possible. Early and frequent *feedback* regarding to the efforts made, and having *courage* to support holding on the made decisions are desirable characteristics. In addition, Agile Modeling values *humility* to admit that others may have value to add to one's project efforts. (Agile Modeling 2002)

APPENDIX 2: Interview sheet – questionnaire guide

Aim: To gather information about processes during the first development cycle of the [product X].

Designed to interview [N.N] and [N.N].

PART I

Background information:

Name:

Date:

1. The name of your project group:
2. How big is your project group?
3. What is your main responsibility?
4. How long have you been working in your current position?
5. How is the work divided inside your project?
6. In your opinion, what are the personal competencies of each person of your group?
7. How long have you worked at [Corporation X] overall?

Process redesign goals and performance targets *(adapted from El Sawy 2001)*

- What are the goals of redesigning this project?
- For each goal, give the level of importance (low – medium – high)
- What are the related process performance targets for each goal? Please, identify the targets in concrete terms (reducing costs, improving quality, etc.).
- How would you measure the goals for each process target?

- What do you see as major resource constraints related to the work environment of the process?

Defining process boundaries *(adapted from El Sawy 2001)*

- Who are the customers of this process?
    - What are the different categories or types of the customers?
- What are the outputs of this process?
    - What are the different types of deliverables?
- What are the inputs of this process?
    - What are the different types of triggers that start it
- What departments inside the organization does the process interact with?
- What other business processes in the organization does the process interface/interact with?
    - What do you see as threats and opportunities considering each business process?
- What external entities (suppliers, service provides) does the process interface with?
- What are the few big chunks that the process can be divided into (sub processes) that make intuitive sense?
- Where do you get the revenues?

PART II

Process specific information:

Business decision

1. There is no business plan written. Why not?

2. Has this disturbed the project somehow?

    2.1 If yes, how?

3. How were the involved people informed about the business decision? (E.g. who were the involved people? What channel was used?)

4. How soon was the "agreements phase" started after business decision had been made (i.e. searching for the partners etc.)?

Agreements (Partnerships, licenses)

1. How were the partners chosen?

    1.1 What was the search process like in each case?

    1.2 Who were involved in searching involved possible partners?

    1.3 Who made the final decisions about partners?

    1.4 How were other people informed about partners?

2. What was the process of making agreements like? (E.g. how was communication organized, the negotiations, etc.)

3. How much time did this phase take altogether? (I.e. getting things in the state that the real work could be started)

4. What are the templates involved in this phase?

    4.1 Evaluate the templates used (easy/difficult to adapt in each case, require only little/too much work etc.)

5. Did you use any tool(s) during this phase?

    5.1 If yes, what?

    5.2 Evaluate the used tools?

Brand conventions and trademark policy

1. How much time was planned to complete this phase?
2. What tasks did this phase include?
3. Were there some problems in any task(s)?
   - 3.1 If yes, describe the problems?
4. What were the parties involved in this phase?
5. Were there some template(s) used in this phase?
   - 5.1 If yes, what?
   - 5.2 Evaluate the effectiveness of the template(s) used?
   - 5.3 If not, do you think some template(s) might have been useful?
   - 5.4 Can you suggest any template(s) that might be good for this task?
6. Were there any tool(s) used in this phase?
   - 6.1 If yes, what?
   - 6.2 Evaluate the effectiveness of the tool(s) used?
   - 6.3 Can you suggest any tool(s) that might be good for this task?
7. How much time did this phase take altogether?

IPR

"Light IPR check will be done by the end of May" (source: [Project x]: Status report)

1. How much time was planned to complete this phase?
2. Who was mainly responsible of this task?
3. The keyword list was done first – how were keywords chosen?

4. The results are ranged in sense of importance – on what is this evaluation based?

5. What parties were involved in this phase?

   5.1 Who took care about visual check and on what was it based?

   5.2 What parties were informed about the results of IPR check?

   5.3 How were these parties informed (i.e. what channel was used, by what time)?

6. Were there some template(s) and/or tool(s) used in this phase?

   6.1 If yes, what?

   6.2 Evaluate the effectiveness of the template(s) and/or tool(s) used?

   6.3 If not, do you think some template(s) and/or tool(s) might have been useful?

   6.4 Can you suggest any template(s) and/or tool(s) that might be good for this task?

Naming

1. How much time was planned to complete this phase?

Task 1: Define method (the name should be based on [parent corporation's] naming policy, there was a name competition organized, source: [Project x]: Status report)

2. How would you evaluate the chosen method (name competition, i.e. strengths/weaknesses)?

3. Can you suggest any other method that might have been good for this task?

4. How were people informed that there was a naming competition?

5. How much time did the competition take?

Task 2: Filtering

6. How was filtering organized?

7. Who were informed about the filtering results?

Task 3: IPR check (Parent corporation's IPR section checked the candidates; source: [Project x]: Status report)

8. How much time did this task take?

9. What parties were informed about the results of IPR check?

10. What was the used communication channel in informing other parties?

    10.1 Evaluate the effectiveness of the communication channel used

    10.2 Can you suggest any other communication channels that might be good for this task?

11. Were there any tool(s) and/or template(s) used in this task?

    11.1 If yes, what?

    11.2 Evaluate the used tool(s) and/or template(s)

    11.3 If not, do you think some tool(s) and/or template(s) would have been useful for this task?

    11.4 Can you suggest any tool(s) and/or template(s) that might be good for this kind of task?

Task 4: Select

12. When was the final decision of the name made?

13. Assuming that the selection was made in quite a late phase before launch, how critical do you evaluate this issue (documents had to be updated etc.)?

Task 5: Inform

14. What parties were informed about the selected name?

15. What channel was used in informing other parties?

16. How much time was used in the informing task?

17. Were there any other tasks involving this phase (naming)?

    17.1 If yes, what tasks and how were they handled?

Look&Feel (based on design planned in autumn 'yy, source: [Project x]: Status report)

1. Who was responsible of this task?

2. Validity check was one task in this phase. How was this task handled?

3. What other tasks did the phase have?

    3.1 Please, describe the other tasks.

4. How was communication about Look&Feel issues organized?

    4.1 Who were the people informed?

    4.2 How were the involved people informed?

"The actual Look and Feel project was not completed for the [product x's first] release"

5. On what was the layout of [this release] based?

Customer documentation

1. How much time was planned to be used for this phase?

2. How was the input gathered?

    2.1 How much time did this task take?

2.2 What communication channels were used with input sources?

3. Who did the writing work?

3.1 How much time did this task take?

4. How and by whom were the documents reviewed?

4.1 How much time did this task take?

4.2 If a text did not pass the review, how was the writer informed about it?

4.3 How was versioning of reviewed (failed, passed) texts organized?

5. How much time did the law check take?

6. How much time did spelling take (assuming, that it was outsourced to [subcontractor X])?

6.1 How was communication between [venture team] and [subcontractor X] organized?

7. Who checked the validity?

7.1 How much time did the validity check take?

8. The layout was based on templates used. What templates were used?

9. Did you get any feedback about the templates used from the writers (assuming that writers were people outside the [venture] team)?

10. Evaluate the templates used (proper/improper for the task, easy/difficult to use, require good amount of/too much time, etc.)

11. Were there other tasks in this phase?

12. How much time was used altogether?

13. Do you have any improvement suggestions for this phase?

Productisation of [product X]

[Venture team] took care of packaging and distributing the final software package (source: [representative of Unit X]; [Project description])

1. Did you get any requirements/suggestions from other parties (e.g. [Partner], [Unit X]) considering the final package?

   1.1 If yes, what?

   1.2 What was the communication channel for the requirements/suggestions?

2. Through what channel(s) did you get all the parts of the package? (E.g. [Unit X's part], [Partner's part])

3. How much time did the packaging and distribution take?

4. Were there any tool(s) in use for packaging?

   4.1 If yes, what tools?

   4.2 Evaluate the effectiveness of the tool(s)

   4.3 If there were not any tools in use, why not?

   4.4 Do you think some tool(s) would have been useful?

      4.4.1 If yes, what tool(s) and why?

5. If it was necessary, how were other parties involved ([Partner], [Unit X]) informed about packaging & distribution?

Delivery and web site

1. How was the delivery organized?

   1.1 What channel was used?

   1.2 Was there a schedule defined for this task?

2. How was it made sure that the latest versions were delivered?

3. Were there any tools used in this phase?

   3.1 If yes, what?

   3.2 Evaluate the effectiveness of the tools used

   3.3 If not, do you think some tool(s) would have been useful for this task? Can you suggest any tool(s) that might be good for this kind of task?

Evaluating the whole project

1. Which phases do you find were the most critical in this project? Why?

2. In your opinion, what phases were the most difficult to complete? Why?

3. In your opinion, what phases were the easiest to complete? Why?

4. Can you mention some tools that would have helped in overcoming the confronted problems?

   4.1 Why do you think the tool(s) mentioned would be good?

5. Can you mention some templates that would have been helpful in some tasks?

   5.1 Why do you think the template(s) mentioned would be good?

6. Evaluate the personal workload during this project.

7. This project has basically followed the small company model.

   7.1 What have you learnt during working in the project?

   7.2 Has the working mode in a small company model influenced on your personal competence development?

8. What do you see as critical factors for this projects future?

APPENDIX 3: Example of a process model