LAPIN AMK
Lapland University of Applied Sciences

AGILE METHODOLOGIES IN LARGE-SCALE SOFTWARE PROJECTS

Sana Grigoryeva

Thesis
Degree Programme in Information Technology

2015

LAPIN AMK
Lapland University of Applied Sciences

Abstract of Thesis

Technology, Communication and Transport
Degree Programme in Information Technology

Author Sana Grigoryeva Year 2015
Supervisor Erkki Mattila
Title of Thesis Agile Methodologies in Large-scale Software Projects
No. of pages + app. 62 + 6

Agile methods are widely used nowadays in software development both in small- and large-scale projects. However, it can be rather challenging to apply them to the bigger projects. Moreover, there are no precise instructions of how to use the agile methods in a big organisation. Therefore this topic is vital.

The goal of this thesis was to render the reader an overview of what the agile methods are, and how can they be applied to large-scale projects. In the theoretical part, the origin and the general concept of the agility in software development were explained. The statistics of the methods' usage in today's software development was presented as well.

In the middle part, the key points of scaling the agile methods were presented. After that, the most popular agile methods were compared and analysed regarding their suitability for large software projects.

The last section included the example from the author's own experience of how the agile methods were used in large projects, and based on theoretical and empirical knowledge certain improvements were suggested in order to raise the effectiveness of the methods.

A descriptive literature review research method was used when gathering and analysing the information from web resources, books and conference presentations. In the real case study, a qualitative method was used to describe the data from the project and to make conclusions. Possible further research could be conducted by implementing the suggested solutions to the real working environment and research the outcomes.

Key words agile, incremental and iterative, Scrum, Kanban

CONTENTS

LIST OF FIGURES

SYMBOLS AND ABBREVIATIONS

IID	Iterative and Incremental Development
DSDM	Dynamic Systems Development Method
XP	eXtreme Programming
FDD	Feature-Driven Development
ADT	Application Development Trends (McKendrick 2013)
RDM	Requirements Definition and Management
GUI	Graphical User Interface

1 INTRODUCTION

Nowadays software is required in most of the areas of production and everyday life. The market is large and constantly changing. Moreover, the technology is developing rapidly. In this environment, software development companies need to tailor their operating processes to correspond customers' demands, to reach the business goals and to be highly competitive.

Agile methods are created to meet these needs and to assist enterprises to cope with the challenges of nowadays prompt software development. That is why they are becoming prevalent among other development methods recently. However, large number of companies which adopted and successfully applied these methods is of rather small size. The guides for using agile methods are mostly applicable to non-large projects. Consequently, big organisations struggle and hesitate to adopt and use agile methods in their work.

It takes more effort to apply agile methods to the large enterprises, because more employees and structural levels are influenced by the changes. Therefore it is essential to investigate this topic. The author has own experience of working in an agile large-scale project, which gives a possibility to use real life experience. The main purpose is to explore how agile methods can be applied to large software projects effectively.

The objectives of this work are to investigate what is the nature of agile methods, how and why they appeared and developed. Because the concept is rather wide, the second objective is to collect various definitions of the software agility given by different authors. This is done in order to create one collective and fulfilling definition of what is agile software development. Furthermore, the objective of the work is to investigate if the methods are applicable to the large projects, and which of them are most suitable. It is also important to know, how to combine methods and tailor them to the company's needs. In order to explore this topic, the real company case is described and analysed in the end of the work.

2 FUNDAMENTALS OF AGILE SOFTWARE DEVELOPMENT

2.1 Roots of Agile Methods

The agile methods became the most noticeable change in software development thinking during last two decades (Fowler 2005). However, their history did not start then, but much earlier. Agile methods have strong roots, which go back to 1930's. The ideas which now form the backbone of agile methodologies were proposed long time ago as an alternative to the traditional methods. Those ideas appeared in different places independently, but many of them were not understood and were underestimated at that time (Laanti 2012).

According to Craig Larman, the foundation of modern agile methods was iterative and evolutionary development. He states that *"agile methods are a subset of iterative and evolutionary methods".* In his book "Agile and Iterative Development: A Manager's Guide" he presents a history of "iterative development, which lies at the heart of agile methods". (Larman 2003.)

Likewise, according to Larman and Basili (2003), the earliest recorded ideas which are related to agile methods were the iterative and incremental approaches. They grew from the work of an American engineer Walter Shewhart, where he proposed to use a short term "plan-do-study-act" cycles for quality improvement. (Larman & Basili 2003.)

The first record of using iterative and incremental approach in the project dates to 1957, as noted in the paper of Larman and Basili. That project was not developing the software, but it is significant for the present research because a few years later, in early 60's, the same team was working on the large software project of NASA called Mercury, which *"ran with very short (half-day) iterations that were time boxed"*. Interestingly, the team even used a few practices of *extreme programming* - test-first development and planning and writing tests before each increment. (Larman & Basili 2003.)

One of the engineers who were working on the Mercury project, Gerald Weinberg, describes the process of the project and developers' opinion on waterfall model as follows:

> *“We had our own machine and the new Share Operating System, whose symbolic modification and assembly allowed us to build the system incrementally, which we did, with great success.*
>
> *All of us, as far as I can remember, thought waterfalling of a huge project was rather stupid, or at least ignorant of the realities… I think what the waterfall description did for us was make us realize that we were doing something else, something unnamed except for “software development".* (Weinberg 2011.)

Since then, there were also a number of projects using the abovementioned and other agile related approaches in 70’s, 80's and 90's. Remarkably, the majority of them were working on developing large life-critical systems. Among them were software systems for USA submarine, for ballistic missile defense, US Navy weapon system; the primary avionics software system for NASA space shuttle, compilers for a family of application-specific programming languages. (Larman & Basili 2003.)

As a conclusion, the incremental and iterative approaches were the foundation for agile methods appearance. They were used primarily in large-scale projects over decades before the ideas and principles of agile software development were officially stated and recorded in the famous *Agile Manifesto* in 2001. (Beck *et al.* 2001.)

2.2 History of Agile Methods

Table 1 in Appendix 1 shows the history of the appearance and the development of the agile approaches. The table contains the descriptions of various (mostly large and life-critical) projects where the agile methods were firstly suc-

cessfully applied to. It contains also some publications where the agile development approaches and features were first described and formalized.

The *dates/duration* column of Table 1 (Appendix 1) shows the dates when the project was executed or when the publication was written. The column *project/author/publication name, short description, country* contains the name of the project or writer and his/her publication, its short description and country of origin.

The column named *methods/approaches applied* consists of the methods and approaches which were used or described in the particular work. And the last column *additional info* contains supplementary notes, whether the project was big and beneficial or not and other things. It can be seen in Table 1 (Appendix 1) that iterative and incremental development (IID) and agile practices of early years were mostly applied to large projects, because the complexity of the system made usage of the waterfall system inconvenient at some points.

There are a few tendencies which can be distinguished throughout the development of IID. The first method that appeared and later on was applied to the most of the projects is iterations. Their length varies among the projects – from half-day to 6 months or even longer. Some of them were not time-boxed at all. But the main tendency can be determined: the length of the iterations decreases with time, from several month long iterations to nowadays recommended 2-4 weeks iterations, even though it was chosen empirically for each certain project.

Second after iterations, the next tendency which appeared is using of increments. This tendency is distinguished in most of the projects since it appeared first time in 1970. First there were sequences of intermediate systems of code, and then teams started to produce independent versions, later on clear measures of success appeared. The biggest reason for using increments is the possibility to retreat.

Also the methods of specification and prioritization evolved throughout the time. In early projects the prioritization was done from top-level control structures

downwards. Later on it was done based on risks; core architecture was implemented on early stages.

Another important tendency is cycle-learning. Even the earliest projects were taking advantage of what was learned in previous iteration. The learning came from the development and using the system. The feedback has been always essential part of learning.

The emphasis on cycle learning was growing, and in 1987 the whole process of software development was regarded as learning and communication. That time there started to develop the focus on individual developers as people, their skills and communication between each other. About the same time contractor participation became an important part of development process.

In conclusion, there were no rigidly clear tendencies in agile methods development, because the new methods were implemented and tried out in real life, some of them became successful and some did not. However, it is possible to determine the main tendencies and the order of their appearance, which helps to have deeper insight into the history of agile software development.

2.3 Definition of Agile

Agile software development ideology includes 4 values and 12 main principles (Beck *et al.* 2001). Basically they do not dictate to development team what to do, they just point the direction. Agile software development is a multidimensional concept, it is significantly wide. When the definition of agile software development is given, depending on from which point of view it is described, the different aspects of agility are emphasized.

Table 2 (Appendix 2) contains systematized definitions of software development agility adapted from Laanti (2012) according to Kettunen (2009). The definitions are presented in chronological order. They were given by various authors at different times. They were analyzed and there was added third column that indi-

cates which aspects are stressed in each definition. Authors reveal different sides of the concept, although some of the aspects repeat in several definitions.

All the aspects that were indicated in definitions were studied and based on them collective definition of agility was composed. The definition is presented below and the aspects of agility in this definition appear in the order of their repetition frequency. The aspects were grouped according to the area of their application.

Agile software development is a time-boxed iterative and incremental lightweight approach, which is rapid, flexible and adaptive to changes in requirements or environment, oriented on individuals and communication among them. In agile development self-organized cross-functional teams in highly collaborative environment make own decisions; it implies constant planning, testing and integration; which uses constant user/stakeholders' feedback, quickly responds to it and constantly learns from it, improving the process.

When applied rationally, all aspects of agile development mentioned in the definition above provide faster software development and frequent deliveries, higher user satisfaction. They also allow the software to meet the needs of stakeholders, and the company to benefit in constantly changing business market.

2.4 Applying of Agile Methods to Modern Software Development

Agile methods are widely used today in software development enterprises as well as in industrial companies. According to many reports, they bring business values to the company and help to cope with nowadays market demands.

However, there are certain disadvantages of using these methods, and some companies have concerns about adoption of agile development tools. In addition, some of them are struggling to scale agile methods to bigger scale.

2.4.1 Software Crisis

Some time has already passed since the first computer was invented, and since then computing technology has been developing faster and faster, and nowadays it already grows exponentially. Along and together with technology, intelligent products and services are evolving. Moreover, digitalization of data is increasing all the time. All these factors generate growing demand for software.

It happened that hardware and its capabilities were developing rather fast, and as the result they required more and more complex software to be created. But existing methods for writing software was not sufficient and not rapid enough. This created a difficulty and certain problems for software engineering already in the early years of computer science. This phenomenon got a name of *software crisis as* early as 1968, and was described by Feller and Fitzgerald as situation when the software is too long to develop; it costs too much and does not work very well. (Feller & Fitzgerald 2000.)

This problem needed a solution. As the software product complexity increased, the complexity of software projects rose as well. Consequently, software projects demanded new software process models, which would bring the needed solutions, help to increase productivity and to minimize risks connected with software development.

Changing requirements posed one of the biggest risks for the software development. As software became more complex, there was much more lines of code in the program. Changes in requirements may appear at any point of the process, and the more software project is growing, the more difficult it is to make changes in it. Consequently, software engineering needed new process methodologies which are flexible and responsive to changes.

In conclusion, increase in software complexity and technology capabilities created a need for a light-weight, flexible ways to create software. It is a general tendency nowadays which can be seen not only in software development, but in other industries as well.

2.4.2 Development of New Methodologies

As mentioned in previous section, there was a need for new software process models, and they started to appear and develop. There were two factors which contributed to it – increase in software product complexity and in software project complexity. Some heavy-weight methodologies already existed and were used at that time, and as can be seen from Table 1, first new lightweight elements were added to the existing approaches. As was mentioned before, the first iterative and incremental approaches appeared in 1959.

Methodologies and frameworks which are still used and popular nowadays appeared in 1990's. Among them are Scrum, Dynamic Systems Development Method (DSDM), eXtreme Programming (XP), and Feature-Driven Development (FDD). They all contain different techniques and approaches. However, there is something that they all have in common – they are all incremental and iterative, and as the result flexible and highly responsive. In 2001 an "umbrella" term *agile methods* was invented by a group of initiative software engineers of America. They described the main principles of agile software process, which was called Agile Manifesto.

When agile methods first appeared, they were applied to single projects. However, there was a need to scale methods to the large projects and big organizations. For that the method was changed or altered. For example Sutherland (2005) in his presentation in Agile Conference in Denver described how Scrum was used for a large-scale project. This type of Scrum has its own name, so-called C Scrum, and it included overlapping project phases with tightly integrated builds, which were made several times per working day. There were also tightly coupled teams, and the administrative control and pressure were significantly reduced.

To sum up, agile methods appeared as a solution for the problem of developing software and coping with increasing complexity and requirements. However, this process has not finished, today technology is developing even faster, but at the same time software development models are developing as well, they are

being complemented and edited. The same happens when already adopted agile methodology needs to be scaled to the bigger level.

2.5 Benefits of Agile and its Current Usage

These days software industry is highly competitive, the market is developing rapidly, and consequently, software enterprises have to work fast and be highly flexible. Agile methods' principles meet these objectives. They are said to be profitable for software companies. According to Schwaber, Laganza and D'Silva (2007), agile methods decrease time-to-market, increase quality of software and reduce the waste. They also increase predictability and minimize risks.

Figure 1 which is displayed below shows the difference between traditional (waterfall) and agile software development in four features: visibility of the project, its adaptability, business value and risks. As can be seen, in waterfall method visibility of the project is high at the start, but during the development process it decreases almost to zero, and the project becomes visible again only when it is ready.

Figure 1 illustrates that agile development allows the visibility to be high throughout all project process. Moreover, the project is able to adapt to changes throughout the entire project, while in waterfall model it decreases stably as the project advances. The business value of agile project grows since the start of the project and remains high all the time because of frequent releases. At the same time, the risks of the project decline hyperbolically already in the beginning of the project due to user testing and flexibility of the development process.

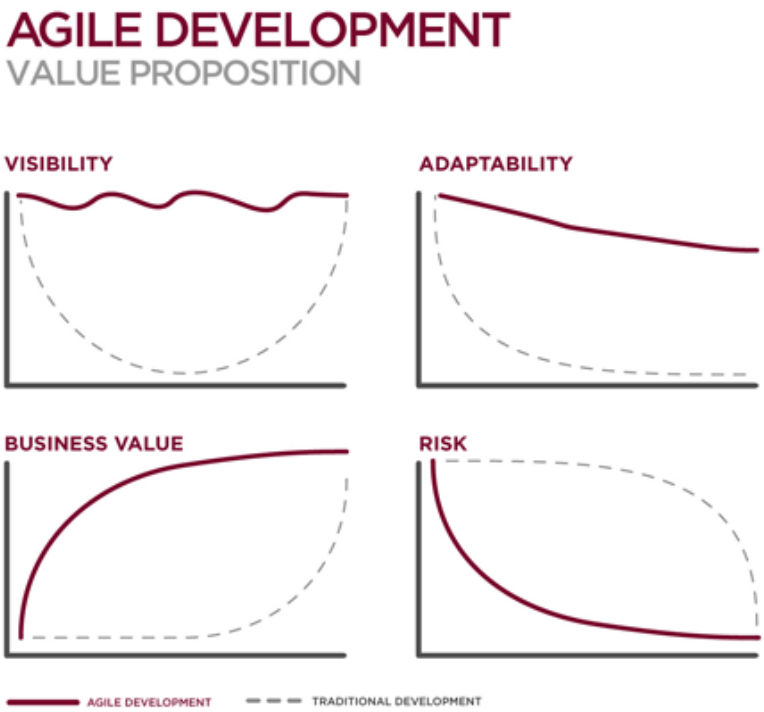

Figure 1. Difference between Agile and Waterfall Value Propositions (VersionOne Inc. 2014b)

The research into some up-to-date data was made to analyze the prevalence and usage of agile methods these days. The data was taken from Agile Survey 2013 (VersionOne Inc. 2014a). Overall, 3501 people took part in this survey, and most of them hold positions of project managers, scrum masters, and team leaders in software development departments of their organizations. What is important, ¾ of the respondents are coming from organizations where the number of employees is between 100 and 1000. This means that the respondents work in broad development environment and they are involved in large projects.

2.5.1 Adoption

The most significant number is general usage of agile methods by the IT enterprises. This number doubled since 2012, when only 35% of the respondents applied agile methods to their work. In 2013 there were already 76% of them.

More people are recognizing that agile methods are generally beneficial to their business in comparison with previous years. Particularly, 11% more than in last 2 years people admit that agile methods help them to complete their projects faster.

Moreover, people say that agile methods help to reach the target why they were applied to their business. They also distinguish 3 main benefits that they expected to acquire when they adopt agile methods: they want to be able to manage frequently changing priorities; they want to increase productivity and project visibility.

The respondents were given a list of the reasons to adopt agile, and the top 3 reasons they chose were to accelerate time to market (23%), more easily manage changing priorities (16%), and to better align IT and business objectives (15%). When an organization decides to adopt agile practices, initiative most commonly come from the management level (in 61% cases). Less often it comes from the developers themselves (17%) or from executives (16%).

Looking at these results, general current tendency is spreading of agile methodologies wider and wider, most of the software enterprises already use different kinds of them. Enterprises adopt agile methods because they help to solve problem arising in their performance, connected with today's changeable market situation. As the manager can see the development process both from inside and outside, initiative to use new process methods usually come from them.

2.5.2 Scaling

As survey concludes, software professionals have got wider knowledge about agile practices and now scaling them to the broader projects and within their organizations. According to the numbers, 88% of the survey respondents are knowledgeable with agile practices, which is 7% bigger than in the last year's survey. Majority of the participants use agile practices in their organizations already from 2 to 5 years.

During this time, they have seen some success in several teams where agile methods were applied. As the result, nowadays they scale agile practices more broadly in their workplaces based on the success they have got and on their wider knowledge. Today already 57% of respondents apply agile practices to the projects where are 5 and more distributed teams, and 38% have 10 and more teams. Last year this number was only 30%. These numbers indicate that there is a tendency of scaling agile methods to the bigger projects and embracing them to the enterprise level.

2.5.3 Most Popular Methods and Tools

The survey states that Scrum and its variants remain the most popular methodology used by companies during the last several years. In 2013 73% of the respondents use it. The second most popular methodology after Scrum is Kanban. It is rather new in software development industry and it has gained popularity during the last couple of years and continues to spread.

As the survey sums up, the respondents utilize a wide variety of different agile management tools and techniques. The most popular and wide-used in 2013 were daily standup meetings, iteration planning, unit testing, retrospectives, release planning, burndown/team-based estimation, velocity tracking, coding standards, continuous integration, automated builds, dedicated product owner, integrated development/quality assurance.

All these mentioned techniques were used by 50-100% of respondents in 2013. More than 85% of respondents use daily standups, and 75% are using iteration planning, retrospectives, and burndown charts. Furthermore, all of these methods has been used more than in past years, except for unit testing, usage of which has declined.

Kanban was used by 39% of respondents in 2013, and this number continues to grow. Interestingly, about half of all respondents who use Kanban or its variant called Scrumban said that they were primarily using these methodologies only for business processes in their organization.

To conclude, agile methodologies are flexible, because the developers may choose which methods and tools they would like to use and even alter them to the certain degree. It is often said that agile methodologies need customizing to be more suitable for the certain team or organization. However, at the moment Scrum and Kanban seem to be the most popular among various agile methodologies. Furthermore, statistics show that the key Scrum processes are the most used nowadays. Based on these results can be concluded that Scrum and Kanban meet the current needs of the organizations.

2.5.4 Positive Outcomes

When the agile methodologies had been adopted and a few projects were completed using them, respondents analyzed their enterprise's situation and distinguished the real improvements. In general, after they completed their first agile projects, most of the respondents agreed that the project was completed faster than when they used traditional approaches.

Among the other benefits named by the survey respondents are the following:

> *"Ability to manage priorities, increased productivity, improved project visibility, improved team morale, enhanced software quality, reduced risk, faster time-to market, better alignment between IT and business objectives, simplified development process, improved/increased engineering discipline, enhanced software maintainability/extensibility, and easier management of distributed teams."* (VersionOne Inc. 2014a.)

These answers display that there are real benefits of using agile methods, and it concerns large-scale projects as well. The methodologies are advantageous for big projects because there are typically distributed teams in them. Additionally, commonly the bigger the project is, the more complicated it is to develop, and agile methods simplify the development process, reducing risks and improving the product quality at the same time.

2.5.5 Failures in Adoption and Scaling

Even though there are many successful stories in adopting and using agile, there are also negative experiences. It is important to analyze them as well to have deeper insight into the problems and learn from them. Such analysis was done by Agile Survey (VersionOne Inc. 2014a), and the results will be summarized in this section.

Firstly, when the organization is considering adopting agile, they have certain concerns and doubts. The most common of them are the lack of up-front planning, which remains the top concern during the last few years (30% in 2013). The next biggest concern is the loss of management control. Among the rest are management and developers' opposition, lack of documentation, lack of predictability, and lack of engineering discipline, inability to scale, regulatory compliance, quality of engineering talent and reduced software quality. (VersionOne Inc. 2014a.)

It can be seen that some of the options were also mentioned in the success stories, which proves that some of the concerns are unjustified. For instance, one of the biggest concerns is the loss of management control. But in the data displaying actual improvements from using agile, it is mentioned that agile methods helped to improve engineering discipline and team morale, as well as it made management of distributed teams easier.

There was also a concern about reduced software quality, but from the real success answers can be seen that software quality actually enhanced. However, that is true that in agile it is much more complicated to make long-term predictions. But on the other hand, in traditional methods, when the project is nearly finished and it fits into schedule, there always might come some requirements changes or some urgent fixes, and in traditional methods fixing takes much longer and there is no chance for the project to be on time anymore.

The rest of the problems which are lack of up-front planning, lack of documentation remain open and there is no direct evidence that any outcomes balance this

lack. However, looking at the general benefits from using agile, it can be seen that productivity is increased and risks are reduced, and such results may justify the lack of other organizational things.

Finally, a concern about inability to scale needs a separate look within this work. As the survey states it is possible to scale agile to the organization level. Even though, there are some factors that become barriers to the further adoption.

Most respondents stated that their adoption or scaling of agile failed because of inability to change organizational culture. The next most common barrier after it was general resistance to change, followed by trying to fit agile elements into a non-agile framework. The other barriers are availability of the personnel with right skills, management support, project complexity, customer collaboration, confidence in ability to scale and budget constraints.

In conclusion, it can be seen from the real professionals' experience that agile methodologies most of the time meet the expectations of the people using them. Some of the issues which are concerned most before adopting agile methods seem to be unjustified when looking at benefits which were gained. When scaling agile methodologis to the larger workspace, enterprises meet different obstacles, which need to be concerned before taking actions, as well as the risk management should be done beforehand.

2.6 Overview on Agile Methods' Formation and Spreading

To sum up, agile methodologies, which recently gained popularity among software developers, have their roots in the early 1930's. Separate elements which later became core in various agile methodologies, were used in single projects. However, most of these projects were large and even nationally significant in USA. Agile methods appeared as an alternative to traditional methods, and the main goal for creating them was to overcome software crisis.

Later on in Agile Manifesto all these ideologically related methods were systematized and they all got one collective name. They all include different processes and tools, but they all have certain common aspects and values. Therefore, the agile development has very complex definition, which was composed in this chapter from various points of view. Since then they continue to gain recognition by software developers.

Nowadays still many different agile methodologies are used, and the most popular at the moment are Scrum and Kanban. When an organization decides to adopt a new development process, or they have already adopted it and want to scale it to the further levels, they meet some barriers, because the changes in organization are significant. However, according to real software development organizations' experience, agile methods prove to be beneficial for them and they help to reach the goals that were expected to reach.

Agile methodologies have also one noteworthy common specialty, they should be followed quite strictly, but at the same time they give space for independent activities. It means that all the methods may and should be customized for the certain organization and even development team. Hence, used methods and appearing obstacles are individual for each enterprise which decides to apply agile methodology, and may lead to different outcomes.

3 AGILE METHODS IN LARGE PROJECTS

The previous chapter included references about big software projects, while describing the history of appearance of agile methods and their usage nowadays. This means that agile is used in big projects. However, it is not clear yet how it is used and how beneficial it is for the enterprise. Scaling agile methods to the large projects needs lots of consideration, and there should be some proof that it can work smoothly in large and geographically distributed project.

There are also certain disadvantages of using agile methods. Some of them are balanced by the advantages and benefits that the development model brings to the business. Consequently, there are some methodologies and tools which appear to be more and less suitable for the large-scale software development.

When adopting agile, all the stages of software development are affected: requirements collection, planning and design, implementation, testing and maintenance. These issues must also be considered when one is going to apply agile methods to the big project.

3.1 Can Agile Work?

From the survey conducted in 2013 it can be seen that agile methods have become prevalent in today's software development and in achieving the goals of the organization (VersionOne Inc. 2014a). However, there are still some companies, which have doubts and concerns about adopting new agile approaches and let them replace the older traditional methodologies. Part of these concerns makes the belief that agile methods would not work properly in large organizations and it is impossible to scale them to the bigger projects.

Agile methods are built around iterative principles, where small cross-functional teams are working closely communicating with each other. When imagining this scheme in larger scale, it may seem too complex and even chaotic, and makes

people feel unsecure about scaling it. It has been said that when agile methods are scaled, teams lose the sight of the entire project.

Consequently, there appears a question which requires a trustworthy answer before adopting agile – can it be scaled and work for big projects? American software and research company called Attunity, in their article “Agile development can be ideal for large-scale projects” argues that agile can scale upwards and it really brings benefits to the enterprise when scaled (Attunity Ltd. 2013).

In the beginning, agile methods were designed like this that they can be flexible and scalable, and they can be adjusted according to the needs from little to large projects. Scrum, Kanban and Lean methodologies appeared to strengthen the strategy of project management. The article argues that using leverage advanced processes and effectively adjusting them is the key to maximizing agile (Attunity Ltd. 2013).

Here are the real-life examples that agile methods can work in the big projects and enterprises:

One of the largest manufacturers of agricultural machinery in the world decided to transform their development department, which included hundreds of developers around the world into agile. As the result, the number of developers using agile started from 100 and grew to 1200 and their engagement improved significantly. The company also claims that they improved time-to-market, declined time to production and decreased warranty expenses by half. (McKendrick 2013.)

BMC Software, a company specializing in business service management software had over 900 developers which were geographically distributed from India to Houston to Israel. After a year they reported that they started to deliver the product to the market in shorter time and better quality than before, team productivity increased as well. The critical functionality was delivered to the customers more frequently. BMS also was nominated an ADT Innovator’s award in the Application Engineering category. (McKendrick 2013.)

Hewlett-Packard LaserJet Firmware decided to adopt agile to their software development process about 6 years ago. Their teams were situated in 6 different locations and there were more than 400 developers. At that time there was not a large variety of literature about agile methods, especially applied to large enterprises. The adoption was a success, and the company even published a book based on their own experience. In their presentation in Agile Leadership Conference they told that the costs for software development had been reduced considerably and the business goals were reached. (Gruver, Young & Fulghum 2012.)

These examples prove that agile methods are beneficial and used by large companies, and geographical distribution of the employees is not a barrier. Even more, it is believed by some people that agile is not only the desirable development approach nowadays, but it becomes a necessity. McKendrick writes that in our age of consumerization and rapidly accelerating changes, agile is capable of bringing success in long run (McKendrick 2013).

Nowadays the situation in information technology industry sector in many cases even creates the need for agile software. Such currently important benefits as faster time-to-market, high responsiveness, better collaboration with customers, supporting employee engagement were mentioned as vital nowadays for successful software business. Agile methods are implemented to reach these targets.

It is a trend nowadays as well that organizations scale agile to the higher levels of the organization and even to the managing and governing departments. Thus, agile methods not only for the software development. In fact, Kanban has originally appeared as technology process and was later on adopted by programmers. Agile at enterprise scale is becoming a mainstream (McKendrick 2013).

3.2 Disadvantages of Agile

The real life examples prove that agile methods bring considerable advantages to the large enterprises. However, along with the benefits, these methods may rise some problems or difficulties as well, which should be anticipated.

In large projects they might have slightly different effects. For instance, there is lack of emphasis on documentation. If the project is long-term, there may be changes in employee, managing or whole organization level, this creates difficulties for the new people to understand the job completed before they came to the company.

The other disadvantage that may appear is that the decisions upon the work are taken by the developers themselves, hence the main roles are given to the senior programmers, and there is no place for the juniors. It also might be difficult to assess the effort needed for completion of the project when the project is being launched, because of lack of planning.

Moreover, agile software development demands the active user involvement. That makes the project dependable on the user representative's time, as this is one of the key factors of success. If the customer is not sure in the beginning of the project which outcome they want to get, this might get the project to the wrong way. It is known, that user testing significantly increases the quality of software, but it needs to be done quite often and as soon as the new features were released. Therefore the project becomes dependable of the users' time as well.

When using agile in a large project, there is a change of it to become everlasting. One of the main features of agile is its flexibility. It can take the new requirements or changes into work any time. However, this creates problems, especially for large projects, because requirements tend to expand when the project outcomes can be seen even partly. As the result, it may not be completed in time and creates needs for new negotiations of prices and deadlines with the customer.

Integrated testing raises the costs for resources, because the testers needed not only in the end of the project, but throughout all software lifecycle. Furthermore, short iteration which requires 100% completion of features might be rather tiring for the developers. However, this problem is solved by finding an optimal sustainable development pace.

3.3 Key Points in Scaling Agile

After some real present-day examples were given, it is proved that agile methods can be scaled and be beneficial for large IT projects. Even though the process of scaling is distinctive for each company, there are some common issues that have to be taken into account.

First of all, agile software development is more a philosophy and a way of thinking rather than working instructions. The core of it is the main values of the organization. Therefore, the main point is to change the organization's principles and point of view on the software development.

Secondly, the right set of agile methods is chosen and applied to the project. These methods already exist and are explained in corresponding literature. However, it does not mean that when they are applied, they will be benefitial for the project. They need to be tailored for the current working process first.

3.3.1 Principles over the Methods

Gary Pollice in his article “Does agility scale? Wrong question!” states that the question "can agile be scaled to large projects" is raised incorrectly. The core principles of agile development, which are described in Agile Manifesto, are the values, not the methods of development. The document does not mention anything about the scale of the project or its complexity, therefore it can work effectively for any kind of software development companies. And therefore the question is formulated wrongly. (Pollice 2009.)

Likewise, in enterprises values and goals are defining the business, not the tools and methods. Agile principles were designed to assist in reaching the goals of the business, no matter how big it is. That means, when using or scaling agile methods, it is essential to concentrate on the specific project rather than on development process/technology. (Pollice 2009.)

3.3.2 Customization of Agile

All agile methodologies have common basic principles and values, but still there is a variety of them. Some methods appear to be more suitable for large projects than others. When the software enterprise grows bigger, the number of developers increases and the communication channels tangle.

Such a system needs more strict and formalized ways of work and management. Then there is a need for methods which allow developers to solve the arising problems quickly, to make decisions on the system fast, and to be able to communicate easily with each other.

Some examples of real enterprises were analysed to see which frameworks and methods brought success to the large projects. For instance, Salesforce.com applied Scrum with elements of XP to their software development process. The methods were customised to the projects as well, and this brought them success, which they told about in Agile 2007 Conference (Greene & Fly 2007). One more example is Norwegian Pension Fund software project, which ran in 2007-2011 and employed about 180 people. In their development process they used Scrum (Gjertsen 2011). One large project launched by Swedish Police was using Kanban, and was described in Henrik Kniberg's book (Kniberg 2011).

Gary Pollice pointed out the techniques which are more widely used in big scale projects (Pollice 2009). All of them belong to the XP methodology. Planning game makes the planning of iterations and sprints easier regardless of the tools of acquiring the requirements and length of iterations. Pair programming gives good results and quality of code until the developers are not distributed or have

own responsibilities. Refactoring, test-driven development, customer tests, coding standards, sustainable pace are also widely used in large projects.

There are some methods, which do not appear to be useful in large-scale usage. For example, so-called whole team method might create complications, because often the developers are distributed geographically, and it might be a problem to create enough space in one office to let all the team members work in one room. Teams might be divided into subgroups, and keep whole team meetings inside their subgroup.

Collective code ownership might also not be useful in large projects. It takes considerable effort to all developers to understand the code of other subgroups. Metaphors used in planning might be hard to create, when the project is too large and complicated as well.

To sum up, agile frameworks can be applied to the project no matter what its size is. However, adopting and scaling agile needs to be considered carefully beforehand and tailored to the process, because the main thing is to focus on principles over mechanics.

3.4 Software Lifecycle

There also appear questions as if the requirements specification, documentation and design exist in agile methodologies at all. However, if knowing the nature of agile projects, it can be seen that these stages of software development do not disappear from the software lifecycle. They appear in transformed way to be able to suit the agile principles of development.

Requirement specification and planning transform from being a stage of the process to the constant process. It means that they are executed consistently throughout the project development.

3.4.1 Requirements Specification and Documentation

One of the core values of agile software development is working software over comprehensive documentation. According to Christine Li (2012), many agile teams and followers argue that there is no need for formal documentation at all. Verbal communication and prototyping is sufficient, and creating documentation is just a waste of effort and resources. For instance, in extreme programming the requirements are conveyed verbally straight to developers, using some short notes to be able to memorize what was the requirement about.

Consequently, it becomes unsure if the project can work without documentation and if the small notes and index cards are sufficient. There are debates about this topic among the professionals and there is no unified opinion on it.

Tony Heap, an agile coach and concurrently a business analyst, states that modelling/specifications should be written so much that they are sufficient for the current situation, and not more. However, usage of agile methods and particularly writing specifications is very dependable on the specific project. Consequently, it is rather challenging to formalize how much specifications should be done, when and how. According to his own experience, he believes that early investment into requirements specifications can be a waste. He also suggests that requirements should be written as late as possible, because requirements specifications have a nature to be based on conjectures, not on knowledge. (Heap 2011.)

Apparently, large agile projects have more developers involved which may be located in different places. Such teams need more detailed and formalized requirements specifications, and more coordination through the project. This implies that these issues and agile values should be combined and compromise should be found.

The gap which appears between the requirements start to appear and the actual development increase risks of the project. Moreover, in agile development requirements should always outpace the development team. To overcome the-

se difficulties there was created requirements definition and management (RDM) system in one of the most popular agile frameworks - Scrum. The system is presented in the figure 2.

The requirements which are directly used by development team for work are stored in the product backlog. The requirements are picked up from there when planning a sprint. It can be used also as a repository of requirements for the future use.

In the beginning of the process, the requirements backlog is created. It consists of the requirements which should be defined in order to fill the product backlog. They can be visualized, defined by user stories or as functional requirements. The requirements team also works in sprints as well as developers team. At this point prioritization of requirements is done. Jason Moccia says, describing the need of documentation at this point:

> *“In many cases, organizations have documents that need to be created to pass certain "tollgates", or organizational milestones. These items can also be put in the requirements backlog, but they may not end up in the product backlog. Instead, such documents often become reference materials for the development team to use.”* (Moccia 2012.)

When the requirement team composed a product backlog, the development team is involved in the process to refine the items in the backlog. That helps to reach more collaboration and this stage of RDM is called decomposition.

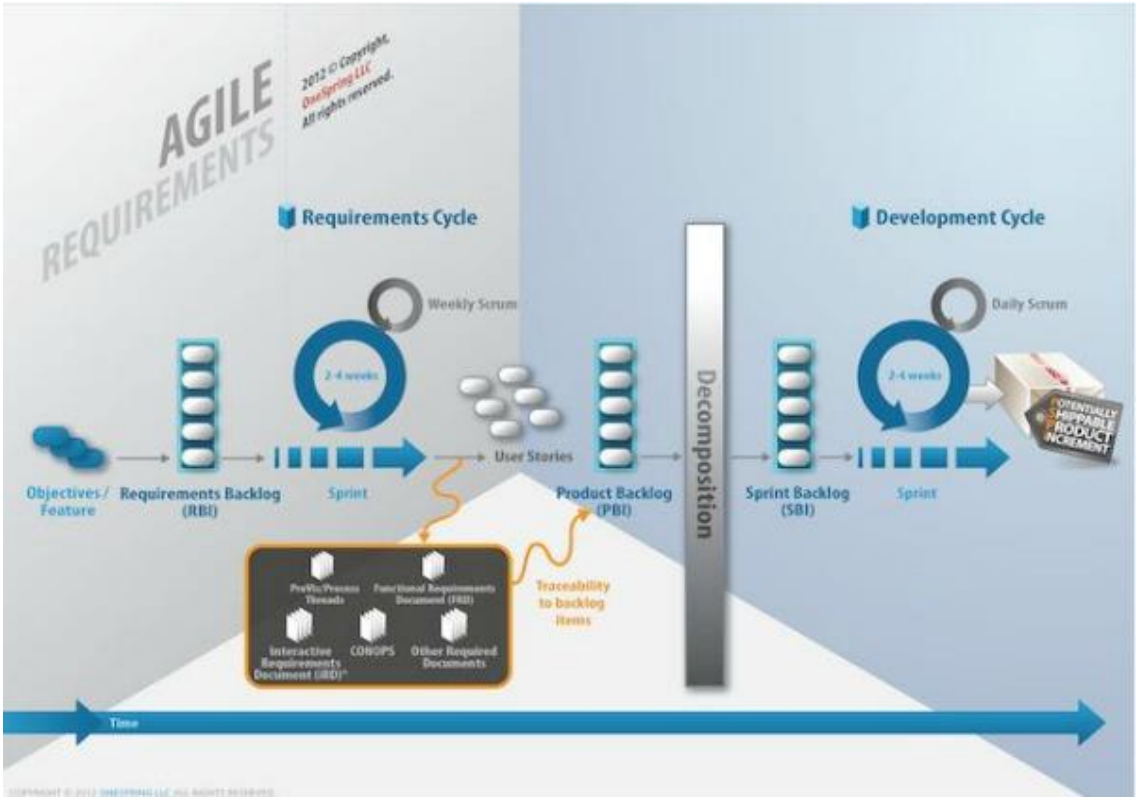

Figure 2. Agile Requirements Definition and Management (Moccia 2012)

There are also other methods and aspects that are used in requirements specifications. For instance, some professionals use Excel for product backlogs because it is rather simple to manipulate. This corresponds to the agile principle that the focus should be on content over the form. The files are placed to shared network drives in order that all the team members have constant access to them and they are able to modify it. It is also possible to filter the content in Excel files, which becomes useful at work.

In one example of managing requirements was described a functional specification (FS) method, in which all the requirements are described in an Excel document. There is a tab which contains all the requirements, and such fields as requirement ID, priority, short description are used at minimum. In the working process more tabs may be added. Then for each feature a spreadsheet is created, which describes the functionality in detail. They appear there in use cases form, the scenario is divided into two columns: "When..." and "Then...", which denote to the starting conditions and the response to these conditions. There may be several responses to one condition, then they are places in separate

cells one after another. The author notes that these use cases may be used by the testers as well, as there are ready tests results described. (Heap 2011.)

For large projects, all the changes that are added to the requirements or product backlogs should be noted. It allows to keep track of changes and increase process visibility.

3.4.2 Software Design

Similarly with requirements specification, the design in agile is done throughout the project and as much as it is sufficient for the current process stage. However, design is still an important issue, because poorly designed software is more expensive to repair in the future.

However, the incremental software architecture is not sufficient in large-scale projects. In the beginning of the project the design and modeling is done, although it uses Just barely good enough (JBGE) artifacts. This means that the design is kept as simple as possible. The common modeling practices are applied at this point as well.

In this stage the customer participation is important, some organizations are practicing user experience design, where the design is reviewed and commented by the potential users. This reduces the risks and increases collaboration. Prototyping is widely used in user experience design as well.

Then, during the development process, incremental design is applied. Because the design should be ready before each new iteration starts, design development is done in advance. The architect should know various modeling techniques to be able to adjust to the certain project. Among various practices, UML diagrams are applied in agile modeling, as well as acceptance tests, user stories, free-form diagrams, user interface prototypes, storyboards, class responsibility collaborators and other.

To sum up, agile methodologies allow all the methods which developers find useful if they add real value. They prescribe that workproducts including design patterns should not be developed for the sake of following a process formula. (Larman 2003.)

3.4.3 Software Implementation and Testing

There are various agile practices applied when writing the code in agile projects. They all correspond to agile principles and core values. The process of coding in large agile projects varies depending on which methodology the team uses. It can be pair programming, whole team working in one room, collective code ownership.

There are some common features referring to coding process which are applied in most agile methodologies. One of them is high collaboration. Developers find solutions by communicating and sharing ideas. Code refactoring is used often to keep the code clean, understandable and high quality. Some enterprises use code standards. The development happens in iterations, during which developers implement certain features.

Similar to the other stages of software development, testing is based on agile principles and is integrated into software development process. Therefore, testing is not a separate process as well. Testing in agile projects is said to be the main way to ensure the continuous progress of the project. All team members should be capable of module or unit testing, even though there are usually separate professional testers in the team, or special testing team.

Testing in agile projects allows feedback from earliest stage of the project, because the software is ready for testing almost from the beginning. The team employs different levels of testing to uncover various types of information.

In agile projects the use of test automation becomes rather important, one of the reasons is that they provide rapid feedback (from few minutes to few hours). Automated unit tests are used to check the behavior of individual features,

methods or objects and their iterations. Automated acceptance tests are used to check the performance of whole system. However, sometimes these tests check only the business logic, skipping the graphical user interface (GUI).

Manual testing is also important in agile projects, although it takes longer time to get feedback. It requires the developer to be on site. Usually manual tests are exploratory and are useful in finding problems in software quickly. They also help to discover opportunities to improve and missing features.

System integration and its testing are critical for large projects. That is why effective agile teams often include an independent test team which works in parallel with the development team and verifies their work constantly (Ambysoft Inc. 2015). They perform system integration testing. This allows more time for software implementation for software team. The test team typically has more sophisticated platform for testing.

3.4.4 Software Maintenance

It may seem that agile methods are not applicable to software maintenance portion of the lifecycle. However, David D. Rico proves that agile methods bring ponderable benefits to software maintenance. Among other, using agile practices allowed to reduce the personnel responsible for maintenance, to eliminate code complexity and stagnation obstacles, and to apply 67% defect reduction. (Rico 2015.)

Agile projects release software rather often, in large projects usually once in a few months. Consequently, maintenance stories start to appear in parallel with new features stories after each release. Therefore, just like other project stages, maintenance is done with the flow of the project. There are several techniques how support and maintenance are done in agile projects.

One of these techniques is a common sprint backlog, which includes both feature stories and bugs and support stories. All the stories are prioritized and implemented according to their priority. To be effective, this technique requires

that the product owner understands well the importance of features and criticalness of bugs to be able to prioritize the backlog.

If the product owner is not capable of finding compromises between features and bugs, another technique is applied, where two separate backlogs are created. These lists are prioritized separately and maintained by people who have wide knowledge about the area. When planning a new iteration, people responsible for both backlogs discuss the most important items and insert them into the iteration task list.

The other possible technique is allocating capacity by time or by sprints. The team may decide that they spend certain amount of days only for new features from the sprint, and rest of the days for bug fixes. As an alternative, they may only implement features for instance in the next two sprints, and the third sprint is allocated only for maintenance tasks. However, these techniques are usually avoided, because they allow implementing low-priority tasks first or deficient workload. One more technique, which is often avoided by the same reasons, is always to fix the bugs first.

Having a separate team for support and maintenance is a popular technique, which is suitable for large-scale projects. Large projects typically get many bug reports and also they have enough people to create a separate team, allowing the development team implementing new features without disruptions. However, the technique has its disadvantages. For example, learning loop is never closed, because the development team will not learn from its errors, as the other team fixes them. Moreover, real progress of the project is not known, and priorities are not managed properly.

Analyzing the abovementioned techniques, the first two techniques appear to be most reasonable to apply to large-scale projects. However, each technique is used in consideration with the project specialties.

4 COMPARATIVE DESCRIPTION OF AGILE METHODOLOGIES

All the projects are different, even if they are conducted within the same company. The number of developers teams and people in each team varies, the teams might be geographically distributed or located in one place. Each company also has own business and project goals. It means that the same methodology which effectively works in one project can lead to the failure in another.

However, initially agile methodologies were designed to be flexible. As mentioned before, the main agile principles are taken without alterations, but the actual methods and tools for planning, development and testing should be customized for the project, taking all the details into account. Therefore it is important to acquire a full understanding of various agile methodologies and be able to compare them.

The most popular and widely used methodologies were compared in different aspects in order to see which of them are most suitable for large projects. At first, core practices and values of the methodology were described. Subsequently, they were compared in the following aspects: iteration length and team size, strong and weak sides, which phase of the project the methods are focused on, scalability and advised project size. The methodologies are presented in the order of their popularity nowadays according to the Agile Survey (VersionOne Inc. 2014a).

4.1 Scrum

Scrum is the most used IID methodology these days. The features that distinct Scrum from the others is that it makes emphasis on self-directed teams, daily progress measurement. Scrum also tends to avoid the perspective process, which means that there is no planning or design made for the long perspective.

Scrum proposes empirical approach to software development. It assumes that it is impossible to define exactly what the customer wants. Moreover, they can change their mind during the project. Therefore, the team concentrates on the

quick responding to changes and on delivering the good quality software quickly.

4.1.1 Methodology Overview

Some of key Scrum practices include self-organising and self-directed teams with recommended amount of 3 - 7 people in a team. Also once the set of tasks for the iteration has been chosen, no additions to it can be done. The teams hold short stand-up meetings every day, where all the team members have to answer a set of special questions. At the end of each iteration the demo is presented to the external stakeholders. In addition, there is client-driven adapt planning done in each iteration. The key emphasis in Scrum is on empirical rather than defined process. (Larman 2003.)

Scrum lifecycle consists of four main phases: planning, staging, development and release. Planning stage is carried out in the beginning of the project. At this point, the vision of the project is created, the funding is found and budget is planned. The initial product backlog is created and needed estimations are done as well. After that, exploratory design and prototypes are created.

When the aims of the planning phase are satisfied, staging phase starts, where more detailed planning is done for the first iterations. Here more requirements are gathered and tasks are prioritised enough to start the iteration. More design and prototypes may be done as well.

After the iteration is planned, the development phase follows. The system is being implemented, and it is released in a series of iterations. Before each iteration the sprint planning is done, the tasks are taken according to their priorities from the product backlog and they are added to the sprint backlog. The team holds daily fixed-time (15-20 minutes per team) meetings every day discussing the sprint backlog. After each sprint, the sprint review is done. Quality assurance appears in every iteration. During the iteration team members update sprint backlog daily. After the system has been released and the needed docu-

mentation is written, marketing and sales tasks are carried out, as well as the other corresponding issues.

Core values of scrum are commitment, focus, openness, respect and courage. The team should be committed to reach the goals of the iterations, and they do the decisions how to reach these goals by themselves. Scrum master and managers commit not to add new work during the iteration and provide the team with needed resources, as well as to make sure that the blocks for work are removed.

Focus means the team should concentrate on reaching the goals of the iteration without distraction, and scrum master focuses on providing resources and removing blocks. Openness is expressed in daily scrum meetings, when all the team members get to know about the work of each other. In addition, the backlogs are open to all people who are involved in development with a possibility to modify it.

The next value is respect, which means that individuality of all developers is respected, and correlated problems are solved in self-organized teams. The last value, courage, means that the managers have the courage to trust the developers and not to tell them how to work. In their turn, developers are responsible for the decisions and organizational issues themselves.

The recommended iteration length in Scrum is 30 days. Iterations in Scrum are called sprints. Compared to the other agile methodologies, this length appears to be quite common.

4.1.2 Strengths and Weaknesses

The methods used in Scrum allow high collaboration and communication level throughout the development process. Among the disadvantages of the Scrum is weak documentation and poor management control of the project.

In some sources Scrum is referred as a framework rather than a method of software development. Volfram Boris (2012) calls it an agile management framework, which is often accompanied by practices from other agile methodologies. Therefore, it is hard to determine which phase on the project Scrum focuses on.

Scrum describes in detail the release cycle which is composed of 30-day sprints. It describes the scheme of delivering the software evolutionary. It also includes some methods of evolutionary planning and design. The workproducts of Scrum (including requirement, product and sprint backlog, burndown charts, etc.) are aimed to manage sprint planning and progress measure management. At the same time, it does not specify the integration and acceptance tests issues.

Scrum can be scaled to so-called "scrum of scrums". If there is a large project and many development teams are involved, the scrum masters of each development groups hold every day scrum meetings, where they answer the certain set of questions as well. Scrum is advised for use in any size projects, from small to very large and complex.

4.2 Extreme Programming

Extreme programming is a well-known agile methodology as well. According to Larman:

> *“It emphasizes collaboration, quick and early software creation, and skillful development practices. It is founded on four values: communication, simplicity, feedback, and courage.”* (Larman 2003.)

Extreme programming includes several precisely described techniques. They are used in a tandem in order to get the desired result.

4.2.1 Methodology Overview

The core XP values imply that the developers communicate with the customers and each other at work, where they locate in a common room, and the design is kept simple and clean. The feedback is acquired by testing, which is done from the first days of development, and the developers are able to respond to changes quickly. The developers are supposed to have the courage and motivation to develop software fast and adapt to changes.

Generally XP involves a constant practice of highly disciplined practices. In total XP recommends twelve core practices. They are: planning game, small and frequent releases, system metaphors, simple design, test-driven development, frequent refactoring, pair programming, team code ownership, continuous integration, sustainable pace, whole team together and coding standards. Larman states that many of these practices work in synergy, and therefore it is risky to combine XP with the other methodologies by eliminating some principles. (Larman 2003.)

Most evolutionary approaches avoid detailed up-front specifications, but they usually recommend recording some details or requirements at least for the next iteration. In contrast with them, XP emphasizes oral communication for planning and design stages of the project. In XP so-called story cards are used to write down the name of the feature that needs to be implemented in the system. When the iteration starts, the developers take the story card and ask the on-site customer for further details.

XP lifecycle consists of five phases. First of them is called exploration, where a few story cards are created, as well as prototypes and rough estimations. The second phase is planning, where the customers and developers complete story cards and estimations and plan the next release. The stories are discussed and picked for the next iteration based on the priorities and current status.

In implementation phase developers implement the agreed set of features during the iteration, actively collaborating and working in pairs in one common

room. They constantly do testing and get feedback from the customer. The following stages of lifecycle are productionizing and maintenance. In the first one, the documentation, training and marketing issues are dealt with. In maintenance stage enhances and fixes are made and major releases are built. The iterations in XP are usually relatively short, from one to three weeks.

4.2.2 Strengths and Weaknesses

XP is noticeable for the fact that it describes precisely the development practices. As Larman states:

> *"A refreshing quality of the original XP description was the statement of known applicability."* (Larman 2003.)

However, it also has its weak sides. While it is focusing precisely on specific programming processes, it gives less attention to overall view and management practices. Other disadvantages are weak documentation, lack of discipline and mandatory presence of customer on site.

XP is mostly focused on coding and testing stages of the project, as the name of the methodology refers itself. The design is very brief and oral in many cases.

XP is aimed at relatively small team projects, usually with delivery dates under one year. It had been proven on projects involving roughly 10 developers or fewer, and not proven for safety-critical systems (Larman 2003). Nevertheless, recently it has been applied to larger projects as well.

4.3 Kanban

Kanban is based on agile and lean software development, one of the key principles of which is flow. The main aim is to create a continuous and predictable delivery stream of the features. For this purpose Kanban visualize which fea-

tures are currently in the process or queued. It also contains mechanisms which allow decrease flow disruptions and waste of effort.

Kanban was invented in Toyota factory in Japan, and later on Kanban principles started to be applied to software development processes. Kanban is the least directive agile methodology as it contains only three rules. Hence, Kanban is not advised to be applied as a first agile experience of the enterprise. Moreover, because of its specialty, it is often combined with other methodologies, most often with Scrum (VersionOne Inc. 2014a). The team adopting Kanban should have certain level of self-discipline and self-organization.

4.3.1 Methodology Overview

First of three Kanban's concepts is to visualize the software development process. Usually a blackboard or projector is used for this purpose. The board contains a table, which displays the current state of the project in all levels. For example, there can be five columns: planning, analytics, development, testing and release. The tasks are written on cards and moved through columns throughout the project.

Second concept is the restriction of work in progress. Every column in the table described above should have restricted number of cards, which can be done simultaneously. This decreases the excessive switching among tasks and reduces the risks connected with this problem.

The third concept of Kanban is optimizing the process. The visualized process is constantly monitored and customized in order to make the workflow smoother and faster. For example, the restriction numbers can be changed, or it can be seen where the work process gets stuck, and more workers can be assigned to this stage of the project.

During the work process, the time which was taken by one task card when it entered the Kanban board and till it was moved do the “done” column. Then it is

analyzed and decisions on how to decrease this time are made. Iterations in Kanban are not mandatory; however they can be applied if needed.

4.3.2 Strengths and Weaknesses

Kanban increases the flow of work and it also makes the problems visible quickly, which arise in the working process. A Kanban team is only focused on work which is in progress at the moment. Kanban describes the planning and software implementation organization without specifying the actual practices for these phases.

The practices prescribed by Kanban cover the project workflow generally; they do not specify the actual testing and maintenance stages. Moreover, Kanban leaves many other issues unspecified, including the team size, leaving the space for creativity. However, as mentioned earlier, Kanban is often combined with other methodologies which define the actual working processes more detailed.

Kanban is advised to be used in larger projects, as it does not prove to be effective in short production run. It takes time before the team adapts to the continuous workflow, eliminates the bottlenecks and determines the suitable customizations.

4.4 Feature-Driven Development

The next agile methodology following Kanban and lean development by popularity is Feature-Driven Development (FDD). As its name states, it is a model-driven software development, which is based on object-oriented component.

The method proposes relatively more planning than the other compared methodologies. In FDD documentation is rather meaningful as communication way as well.

4.4.1 Methodology Overview

Feature-driven software development consists of five basic activities. First process is the development of overall model. The project starts with the high-level description of the whole system and its context. For this activity initial requirements and features are used. This stage of the project is controversial to most of agile methodologies, for instance Scrum and XP, where up-front design and analysis are avoided. However, this process is made iterative and highly collaborative in FDD rather than long and very detailed.

The main goal of the first process is to create shared understanding of the system domain, its key concepts, interactions and relationships. As Stephen Palmer states in his article “An Introduction to Feature-Driven Development”, the object model developed at this stage is concentrated on breadth rather than depth. Depth is added iteratively during the following stages of the project. The model created at this stage becomes essential in the software development, later discussions and requirement clarifications are made around it. (Palmer 2009.)

The second activity of FDD is building of detailed and prioritized feature list. The model created in previous stage is used for it. FDD feature list contains a three-level or more complicated hierarchy. First the domain is decomposed into so-called subject areas. After that, business activities are distinguished in each subject area. The steps in each business activity form a basis of a feature list.

A feature is referred here as a small, client-value function, which is expressed in the form of an action, a result and an object, for instance, “calculate the total of a sale” (Palmer 2009). The features should take no more than two weeks to implement. If it takes more time, it is divided into sub-features.

The next activity in FDD is planning by features. When the feature list is completed, the development plan is created and all features are assigned to the developers as classes. One more aspect that distinguishes FDD from the other agile methodologies is individual code ownership. But ownership is regarded as responsibility rather than exclusivity, according to Stephen Palmer (2009). The

other developers are allowed to change the owner's code if needed. However, the code owner is responsible to check if the change was made correctly.

After planning, design by features is done. The set of features that can be implemented within two weeks are chosen and the design packages are made for them by the corresponding class owners. Detailed sequence diagrams are created for each feature, class and method prologues are written, and after that design inspection is held.

After the design, the class owners write the code implementing the features. Then they do unit testing and code revision, and promote their code to the main build.

Iterations in FDD are usually rather short. Recommended length of iterations is between a few hours and two weeks.

4.4.2 Strengths and Weaknesses

There are a few aspects in FDD that distinguish it from the other iterative and incremental approaches. They may bring both advantages and disadvantages to the project.

The organization of features in the list in FDD is more sophisticated than in XP or Scrum, which use the flat list of items. This aspect of FDD helps to manage larger systems and gives advantages in large projects, which need higher organizational level. When the domain model and feature list are created, the team starts to create own vocabulary, which is called ubiquitous language (Evans 2003). As Palmer believes:

> *"The ubiquitous language the model provides helps phrase features consistently. This helps reduce frustration in larger teams caused by different domain experts using different terms for the same thing or using the same terms differently."* (Palmer 2009.)

Individual class ownership brings certain advantages as well. For instance, there is always an expert in the team, who is able to explain how the class works. Class owner can also make urgent changes to the class faster than anybody else, and to make sure that the class' integrity was not damaged after the changes made by other programmers. Own class can be a reason to be pride of for the developers. However, individual code ownership may be a disadvantage too, depending on the project context.

FDD promotes stronger documentation than other agile methodologies, which allows multitasking. Documentation is used as a communication tool among the team members. Customers are involved through reports as well. FDD also uses UML diagrams contrasting to the other popular agile methods.

FDD approach focuses detailed on design and implementation of the software, and other project phases need other supporting approaches. As mentioned earlier, FDD is advised to be applied to large-scale projects, it is said to be not applicable to small projects due to its basic activities and principles.

5 CASE STUDY: CHALLENGES AND SOLUTIONS FOR SCALING AGILE

The author of this paper has own experience of working in an agile team project. The project was executed by a company, which remains anonymous due to the legislation issues.

The project scale was large, there were several developer teams, separate testing team and the teams were distributed geographically. The project had adopted agile and was looking for ways to discover the most effective methods of work in agile framework. Theoretical basis and research conducted in previous chapters will be used to determine possible methods to enhance the software development process in agile environment, complemented by real life experience acquired by the author.

5.1 Project Description

The project was using a mix of Scrum and Kanban methodologies. Particularly, there were two developer teams, about ten people in each, Scrum master and product owner. The project used one common backlog, which contained both new features and bugs that needed fixing. Backlog and sprint backlogs were maintained in a form of Excel file situated on shared network drive. In this case, Excel file replaced the big blackboard typically used in Kanban.

Backlog items had a few states, through which each item was going during the development process. “Blank” status means that the task is free to be taken. When the task is chosen and developer is assigned to it, the task is moved to “ongoing” state. If the task implementation process is stuck, it is moved to “hold” state. When the task is implemented and tested, it gets “completed” state. All the items in the backlog were prioritized.

The teams were holding daily meetings, which were restricted to fifteen minutes in time per team. In the meeting, each team member told the progress of his/her work and most problematic issues were discussed as well. Monthly meetings

were held to present demo to the client. The teams were working in a few working rooms which were close to each other.

While the author was working there, the experimental test was done in order to see how new methods could be implemented and which obstacles might appear. Two developers participated in the experiment as a small team. Totally there were three sprints. Firstly, the backlog was created for each sprint, containing the main features needed to be implemented. When planning the first sprint, the features were decomposed into small tasks which can be completed in a few days. The tasks were prioritized and assigned. After that, each task was pre-estimated by hours, and the sprint was formed.

During each sprint, the burndown chart was used to monitor the progress. Figures 3 and 4 represent the burndown table and chart for the sprint 1. As can be seen from the figure 3, the length of the sprint was one week. Leftmost column contains decomposed tasks, where first developer was responsible for module 2 and second for module 1. The initial estimations for each task are shown in second column and measured in working hours. In the end of each working day, both developers entered the estimated time left to complete each task. The progress row is the sum of all the hours left to complete the sprint. The ideal burndown row shows how many hours are left for completing the sprint to gain the optimal work progress, regarding that each developer works 7.5 hours per day.

	20.5.2014	21.5.2014	22.5.2014	23.5.2014	26.5.2014	27.5.2014	28.5.2014
module2: 1.1 Creation of the build	4,5	0	0	0	0	0	0
module2: 1.2 Assignments	3	3	1	0	0	0	0
module2: 1.3 Package check	8	8	0	0	0	0	0
module2: 1.4 Package creation	7	7	2	0	0	0	0
module2: 1.5 Load_mode paramete	5	5	5	0	0	0	0
module2: 1.6 Request to module 1	5	5	5	0	0	0	0
module2: 1.7 File info inquiry	5	5	5	2	0	0	0
module1: 1.1 Receiving download re	6	0	0	0	0	0	0
module1: 1.2 Asking for file info	5,5	5,5	0	0	0	0	0
module1: 1.3 Check file type	10,5	10,5	2	0	0	0	0
module1: 1.4 Check naming	10,5	10,5	8	2	0	0	0
module1: 1.5 Download	5	5	5	0	0	0	0
Progress	75	64,5	33	4	0	0	0
Ideal burndown	75	60	45	30	15	0	0

Figure 3. Burndown Table for Sprint 1

Based on the table in figure 3, burndown chart was created. In figure 4, blue line represents the “ideal burndown” row from the figure 3, and red line shows the actual progress of work, taking data from the “progress” row. Figure 4 shows that first two working days had slower progress, but unfamiliarity of developers with the modules could be a reason for it. After that, the speed of progress was rather high. In the end, both progress and ideal burndown lines decelerate a lot because the days 24.5 and 25.5 were a weekend.

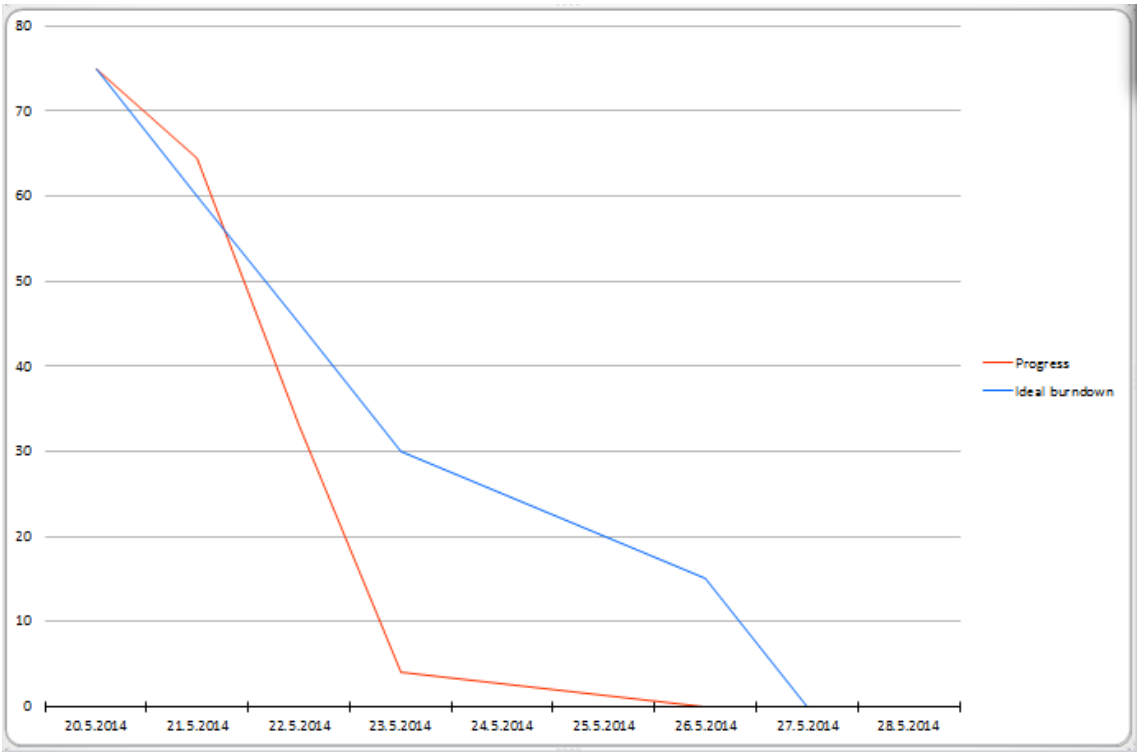

Figure 4. Burndown Chart for Sprint 1

Generally, the burndown chart shows fairly rapid progress. However, it discloses problems in estimation; the time needed for tasks was overestimated. Looking at figure 3, it is possible to observe which exact tasks were overestimated. However, when planning the second sprint, this fact was not analysed. The burndown chart for the second sprint is displayed in figure 5. Comparing the burndown charts of the first and second sprints, it can be seen that estimation accuracy problem was not solved.

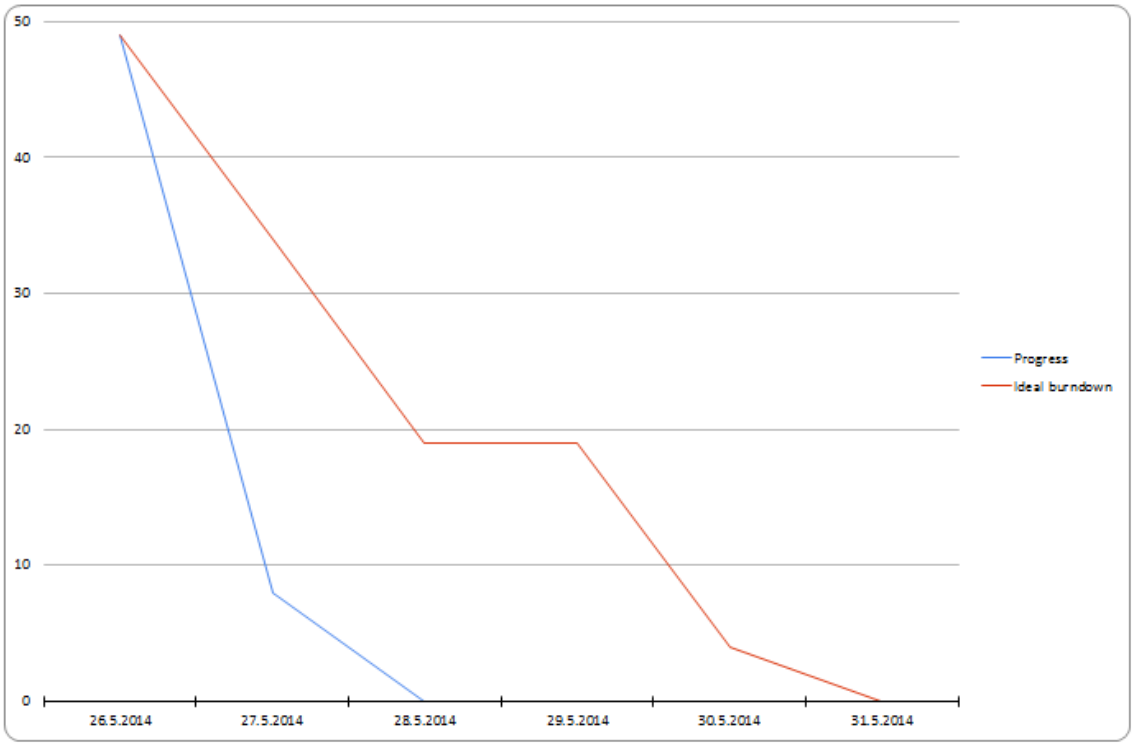

Figure 5. Burndown Chart for Sprint 2

The team also held daily meetings. The work was done in one common room, which was very helpful. It facilitated communication and sharing of knowledge a lot. Each team member kept abreast of developments and knew what the other one is doing at the moment. They also helped each other to solve problems and when the trouble was overcome, they shared the ways of solving it.

Collective code ownership and close collaboration were especially useful in critical situations. Once one team member got ill, and the other developer was able to continue his work after finishing his own without obstacles, because he had sufficient knowledge about other member's module, its current state and problems.

5.2 Challenges and Possible Solutions

During the experiment the team members worked without major outside disruptions, as there were no bug reports or changes in specifications. However, there still were challenges in the development process. For instance, at some point

both team members got sick and the work progress stopped completely. There were also wrong time estimations which can be seen from the burndown charts. Furthermore, in all three sprints the work halted in the testing stage. One of the reasons was lack of testing automation, and the fact that some existing tests were obsolete.

The problem which arises when team members get sick or cannot work and as the result work is delayed is rather common in agile projects. In this case, both team members could not work and the progress stopped completely. As the sprint was rather short, only one week, the team did not manage to finish it in time. That is why short iterations should be avoided in large projects as well.

However, this is rather rare situation in large projects when the work stops totally, as the teams are typically bigger. But even if one or two team members cannot work, it is a risk for the project, and such situations should be prepared for. As learnt from personal experience, close developers' collaboration, working in common room and collective code ownership are alleviating the problem. When the other team members know exactly what the missing worker was doing and which problems he was solving, they are able to take his duties. Moreover, the team should contain professionals who are able to do various tasks and have expertise in several areas of the project rather than only one.

The wrong workload estimation was a clear problem in the experiment. There was too much time set for certain tasks, which is, however, better than when the time is underestimated. Accuracy in estimation needs practice and comes with experience. Cohn states that estimation should be done collaboratively by the team, including the people who will actually implement the estimated tasks (Cohn 2005). In our experiment, the estimation was done only by the two team members, without the team leader or mentor.

There are also several existing practices which are used for prioritization and estimation. According to Cohn, the most popular of them are expert opinion, analogy and disaggregation. He also believes that they work best when combined.

Despite the fact that expert opinion is an important opinion, there are wide expertise areas and large variety of tasks in big projects, which creates a need for opinion of different professionals when estimating. Analogy would be useful in the case under consideration, because the team does estimation through many months and years. They gain experience with time and become able to compare the difficulty of tasks. As they empirically know, how much time the tasks actually took, the accuracy of estimation rises.

Disaggregation, which means decomposing the feature into smaller tasks and estimating them separately, was a part of the experiment. It appeared to be useful for the future feature implementation. When decomposing, the developer understands the nature of the task better and an approximate plan of its implementation appears in his head. This technique could possibly be adopted for the company's project. However, the estimators should be careful not to go too much into details, especially in large agile project, because this wastes time and is against nature of agile methods.

Planning poker is a method, which combines all three abovementioned techniques, and is the best estimation practice according to Cohn (2005). In planning poker, the whole team is taking part in estimation; each member gets a card with numbers from an estimation scale. One member explains the feature to be estimated to everybody, and all members ask questions to make the issue clear. After that, each member chooses the number card and uncovers it simultaneously with the others. If someone has too low or high number than most of the team, he explains his position. After agreement is done, the poker round is repeated until most of the members have same average estimation.

In the case under consideration, planning poker can be fairly time consuming. Because the project has been running for rather long time, the team members know each other and the project quite well, estimation can be done faster and more effectively just by the group discussion, where each team member has to say his opinion and quickly justify it. In case that people struggle with telling

their opinion to everybody, planning poker or some other “gaming” techniques might be useful.

In agile projects the estimation should not be precisely accurate, as it consumes too much time and effort. In the experiment, it would have been less work disruptions, if the team would have done pre-planning and roughly plan the next sprint beforehand. If the team completes all the tasks before the end of the sprint, they can pick new tasks with the highest priority from the sprint backlog. This is one of Kanban principles.

Moreover, one of the important methods of agile software development is cycle learning. It should have been done after the first sprint in order to discover the ways of more accurate time estimation.

It is visible from the burndown tables and charts that the work progress decelerated in testing stages in all three sprints. Testing automation effectiveness was discussed in the previous chapters. In this case it would help to solve the problem, and the solution was being developed already.

One more issue that declined work progress during the experiment was the unawareness of the new workers of how the modules work. The project is large and new people in the team struggle to understand something that was created several years ago. In this case close team collaboration helps. Moreover, despite the tendency of agile methods to avoid writing documentation, in such situation the documentation is needed and sometimes essential. However, the principle of agile development which prescribes to write “just enough” documentation should be followed.

Ethical and moral issues are also important in agile software development. During the experiment, there was a case when the team member could not complete a task, and was behind the schedule. He stopped entering his work progress into the burndown table, the table was not checked by a team leader for some time, and the developer did not ask for help. As a consequence, the sprint progress considerably decelerated.

The situation reveals communication problems within a team, low morale and lack of control from the management. As was mentioned before, even though agile methods promote less control and more team's independence, large projects require certain level of control. One of possible solutions is to control that the project visibility is constantly high. It is still recommended that team is self-decisive and trusted, because it raises developers' morale. Furthermore, it is important to raise team's communication level and encourage people to work. Team members should not hesitate to ask for help or speak about their problems, and this requires good relationships within the team.

One of the challenges that the developers teams were meeting when they adopted agile was that bug reports were coming rather often in the middle of the sprint, and some of them were urgent. One of the rules of Scrum is that there should be no interruptions and task additions during the sprint. But in this case Scrum is combined with Kanban, and also each method can be tailored to be more effective.

In Kanban methodology, there is a limited set of tasks, which are urgent and should be picked by developers as soon as they finish their current task. The sprint can be arranged in the way that there is some spare time in it, for instance, 30 hours of 2-week sprint remain free when the sprint is planned. During the sprint, if urgent bug reports come, these hours are used to fix them. If there are no unpredicted situations during the sprint, the team uses this time to do the first few tasks with highest priority from the backlog. At the same time, sprints should become more flexible to allow this modification. To acquire this, decomposition of tasks can be done, which, however, requires some practice in the beginning.

Geographical distribution of the teams appears to be a challenge in certain cases. High collaboration and communication between team leaders is supposed to smooth the difficulties. Here might be used so-called "Scrum of Scrums", where the team leaders or Scrum masters of each team are holding daily meetings with each other, where they share the progress of their teams' work, discuss the

responsibilities and other possible issues. Video conferences would be useful in this case. When using Scrum of Scrums, the company becomes agile in higher levels, and it becomes important to keep the whole company's morale at high rate and to make the company's goals understood and respected by all the employees.

In conclusion, it takes lots of effort to make the whole company become agile. It means large-scale reorganization and requires careful consideration and wise decisions. Some methods work for the certain enterprises and projects and some of them fail and need to be changed and customised. However, company's business goals should be clear and inspiring, and employee morale should be built correctly. The analysis of real example shows that it is essential to analyse the challenges which appear during the work, learn from them and find solutions, improving the development process constantly.

6 DISCUSSION

The present work introduces the arguments showing that agile methods can be scaled to large companies and be applied effectively. Official statistics and several examples from real life were given in order to prove it.

The research of the literature about agile methods and their origins showed that the ideas which now form the basis of agile methods are not new. They emerged already in the second quarter of the twentieth century. These ideas were used in the form of iterative and incremental development approaches by the companies creating complex and life-critical systems. Various techniques were appearing throughout time and along the development of IT industry. The agile methodologies which are widely used nowadays appeared mainly in 1990's – 2000's, and the main agile principles and values were summarized in Agile Manifesto in 2001.

The projects being developed by different enterprises vary a lot. Consequently, the methods used for each project should be chosen according to the business goals, project boundaries and other specifications. Moreover, agile methods should be tailored for the certain project and altered according to the needs. For instance, the length of the iterations and the size of the teams can be chosen regarding the situation. However, the main principles of agile software development should be followed.

During the research, the most suitable methodologies for the large projects were determined. They are Scrum, Kanban and FDD. Statistics and real experience show that hybrids of them are effective in software development. Furthermore, the work analysed the agile techniques used in different project stages and determined which of them suit for the large projects better than the others. However, the research covered only some agile methods and techniques which are most popular nowadays. Moreover, the statistics presented by VersionOne (2014a) may have inaccuracies, and the number of respondents in the surveys was limited.

The description and analysis of the real case in this work were based only on the author's own knowledge and experience. Moreover, the solutions proposed in that chapter are based both on theory and practice. Some of them were already being implemented when the author was working on the project, for example testing automation. However, some of the proposed solutions would need to be executed in real project environment in order to see how they would work. There are other ways to reorganize work and make it more effective as well. Agile software development requires trying and learning from own experience and mistakes.

Nowadays the software industry develops rather rapidly, as well as the market demands. The companies have to improve their development processes constantly, adapt new technologies and development techniques and methods. New methods are appearing often and it is important to be keep track of the new tendencies and be able to apply them effectively.

Currently there are not many guides and manuals about applying agile methods to large-scale projects. This work researched the key points of adopting and scaling agile methods to the large software enterprises. However, the conclusions made in this work might be more efficient in the project described in the paper and less efficient in different environments and projects, because every company has own specific situation.

REFERENCES

Ambysoft, Inc. Roles on Agile Teams: From Small to Large Teams. Referenced 20 March 2015. http://www.ambysoft.com/essays/agileRoles.html.

Attunity, Ltd. 2013. Agile Development Can Be Ideal for Large-scale Projects. Referenced 22 December 2014. http://www.attunity.com/learning/articles/agile-development-can-be-ideal-large-scale-projects.

Beal, V. Agile Software Development. Referenced on 3 December 2014. http://www.webopedia.com/TERM/A/agile_software_development.html.

Beck, K. et al. 2001. Manifesto for agile software development. Referenced 23 November 2014. agilemanifesto.org.

Cohn, M. 2005. Agile Estimating and Planning. Addison-Wesley.

Evans, E. 2003. Domain Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley.

Feller, J. & Fitzgerald, B. A Framework Analysis of the Open Source Software Development Paradigm. Presentation in Proceedings of the Twenty-First International Conference on Information Systems in Brisbane, 2000 by Association for Information Systems.

Fowler, M. 2005. The new methodology. Referenced 25 November 2014. http://www.martinfowler.com/articles/newMethodology.html.

Freedman, R. 2009. The roots of agile project management. Tech Decision Maker. Referenced 21 November 2014. http://www.techrepublic.com/blog/tech-decision-maker/the-roots-of-agile-project-management/.

Gjertsen, M. How to get at multi team agile project going a presentation based on the experience from PERFORM. Presentation of Statens Penjonkasse on 23 March 2011. Doi: http://konference2011.agilia.cz/data/mette-gjertsen.pdf.

Greene, S. & Fly, C. 2007. Salesforce.com Agile Transformation. Agile 2007 Conference. URI: http://www.slideshare.net/sgreene/salesforcecom-agile-transformation-agile-2007-conference.

Gruver, G., Young, M. & Fulghum, P. 2012. A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware. Addison-Wesley Professional.

Heap, T. 2011. An Agile Functional Specification. Referenced 17 March 2015. ://www.its-all-design.com/an-agile-functional-specification/.

Kettunen, P. 2009. Agile Software Development in Large-Scale New Product Development Organization: Team-Level Perspective. Helsinki University of Technology. Faculty of Information and Natural Sciences. Department of Computer Science and Engineering. Doctoral Dissertation. http://lib.tkk.fi/Diss/2009/isbn9789522481146/.

Kniberg, H. 2011. Lean from the Trenches: Managing Large-Scale Projects with Kanban. USA: The Pragmatic Programmers.

Laanti, M. 2012. Agile Methods in Large-scale Software Development Organizations: Applicability and Model Adoption. University of Oulu. Faculty of Science, Department of Information Processing Science. Academic dissertation. http://herkules.oulu.fi/isbn9789526200347/isbn9789526200347.pdf.

Larman, C. 2003. Agile and Iterative Development: A Manager's Guide. Addison-Wesley Professional.

Larman, C. & Basili V. R. 2003. Iterative and Incremental Development: A Brief History. Referenced 20 November 2014. http://www.craiglarman.com/wiki/downloads/misc/history-of-iterative-larman-and-basili-ieee-computer.pdf.

Li, C. 2012. What Is The Best Structure For Agile Software Requirements? Referenced 10 March 2015. http://www.excella.com/blog/what-is-the-best-structure-for-agile-software-requirements/.

McKendrick, J. 2013. Yes, Agile works in larger enterprise projects, too. Service Oriented. Referenced 25 November 2014. http://www.zdnet.com/article/yes-agile-works-in-larger-enterprise-projects-too/.

Moccia, J. 2012. Agile Requirements Definition and Management. Referenced 18 March 2015. https://www.scrumalliance.org/community/articles/2012/february/agile-requirements-definition-and-management.

Palmer, S. 2009. An Introduction to Feature-Driven Development. Referenced 2 April 2015. http://agile.dzone.com/articles/introduction-feature-driven.

Pollice, G. 2009. Does agility scale? Wrong question! The Rational Edge. Referenced 24 January 2015. https://www.ibm.com/developerworks/ru/library/r-edge/jun09/agilepractices/.

Rasmusson, J. What is Agile? Agile in a Nutshell. Referenced 28 November 2014. http://www.agilenutshell.com/.

Rico, D. F. Agile Methods and Software Maintenance. Referenced 23 March 2015. http://davidfrico.com/rico08f.pdf.

Schwaber, K., Laganza, G. & D'Silva, D. 2007. The Truth about Agile Processes: Frank Answers to Frequently Asked Questions. Forrester Report.

Sutherland, J. 2005. Future of Scrum: Parallel Pipelining of Sprints in Complex Projects. Agile Development Conference Proceedings. Presentation in 2012 Agile Conference in Denver on July 24 2005. URI: http://doi.ieeecomputersociety.org/.10.1109/ADC.2005.28.

VersionOne, Inc. 2014a. 8th Annual State of Agile Survey. Referenced 20 December 2014. http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf.

VersionOne, Inc. 2014b. Agile Software Development Benefits. Referenced 7 December 2014. http://www.versionone.com/agile-101/agile-software-development-benefits/.

VersionOne, Inc. 2015. 9th Annual State of Agile Survey. Referenced 7 April 2015. http://info.versionone.com/state-of-agile-development-survey-ninth.html.

Volfson, B. 2012. Agile Software Development. Version 1.2. Referenced 15 November 2014. http://adm-lib.ru/books/10/Gibkie-metodologii.pdf.

Weinberg, G. M. 2011. Iterative development: Some history. Referenced 27 November 2014. http://secretsofconsulting.blogspot.fi/2011/11/iterative-development-some-history.html.

APPENDICES

Table 1. History of Development of Agile Methods and their Roots (Larman & Basili 2003)

Dates/ duration	**Project/author/publication name, short description, country**	**Methods/approaches applied**	**Additional information**
1959-1963	Mercury project, NASA, USA	*IID:* - very short (half-day) iterations; - time-boxed iterations; - development team conducted technical review of all changes. *Extreme Programming:* - test-first development; - planning and writing tests before each micro-increment.	One of the first recorded projects using IID in software development, with successful outcome.
1970	IBM FSD, USA	Developing from top-level control structures downward. Building the system via iterated expansions. Generating a sequence of intermediate systems of code and functional sub-specifications so that at every step, each intermediate step can be verified to be correct.	Another early IID proponent. However, it was not avoiding a large up-front specification step, had no specified iteration length, no emphasize on feedback and adaptation-driven development from each iteration.
1972	IBM FSD	Project organized in 4 time-boxed iterations about 6 months each. Significant up-from specification effort. Feedback-driven evolution in requirements. IDD used as a way to manage complexity and risks of large-scale development.	IID was a key success factor of the project, which was high-visibility life-critical system of more than 1 million lines of code—the command and control system for the first US Trident submarine.
1972	TRW, USA $100 million TRW/Army Site Defense software project for ballistic missile defense.	Five relatively long iterations. Significant up-front specification work. Iterations were not strictly time-boxed. Each iteration is refined in response to preceding iteration's feedback.	
Middle 1970's	IBM FSD. Part of the US Navy's helicopter-to-ship weapon system, a four year 200-person-year effort involving millions of lines of code.	System incrementally delivered in 45 time-boxed (each is one month) iterations.	The earliest example found of a project that used an iteration length in the range of one to six weeks. As Mills wrote, "Every one of those deliveries was on time and under budget."
1975	Development of extendable compilers for a family of application-specific program-	17 iterations in 20 months (a little bit more than a month per iteration). Developers take advantage of what was learnt during earlier development. Learning comes from both.	They analyzed each iteration from both the user's and developer's points of view and used the feedback.
	ming languages on a variety of hard-	development and using the system. Incremental, deliverable versions of	to modify both the language

	ware architectures.	system. Key steps: start with a simple implementation of a subset of the software requirements and iteratively enhance the evolving sequence of versions until the full system is implemented.	requirements and design changes in future iterations.
1976	Tom Glib (worked on Mercury project), "Software Metrics"	Suggests to implement a complex system in small steps with clear measure of successful achievement. Each step has a “retreat” possibility to a previous successful step upon failure, which gives the opportunity of receiving feedback from the real world before throwing in all intended resources, and to correct possible design errors. Continuous user participation and replanning, design-to-cost programming within each stage. Development in a closed loop with user feedback between iterations.	Material was probably the first with a clear flavor of agile, light, and adaptive iteration with quick results, similar to that of newer IID methods. "The danger in the waterfall approach is that the project moves from being grand to being grandiose, and exceeds our human intellectual capabilities for management and control."
1977	IBM FSD	Integrating all software components at the end of each iteration into its software-engineering practices ("integration engineering").	Integration engineering spread to the 2,500 FSD software engineers, and the idea of IID as an alternative to the waterfall stimulated substantial interest within IBM and its competitors.
1977-1980	IBM FSD NASA’s space shuttle software.	Series of 17 time-boxed iterations over 31 months (1.8 month per iteration). Feedback-driven refinement of specifications.	Avoided the waterfall life cycle because of the frequently changing requirements.
1982	The $100 million military command and control project.	An IID approach that does not usually include time-boxed iterations.	The earliest reference to a very large application successfully using evolutionary prototyping.
1987	TRW A four-year project to build the Command Center Processing and Display System Replacement (CCPDS-R), a command and control system.	The team time-boxed six iterations, averaging around six months each. The approach was consistent with what would later become the Rational Unified Process: attention to high risks and the core architecture in the early iterations.	
1987	Bill Curtis and colleagues “On Building Software Process Models under the Lamppost”	Successful large softwaredevelopment emphasizes a cyclic learning process and communication. High attention to people’s skills, common vision, and communication issues, rather than viewing the effort as a sequential “manufacturing process.”	The publication reported results on research into the processes that influenced 19 large projects.
1988	DoD-Std-2167A	The contractor is responsible for selecting software development	

		methods (for example, rapid prototyping) that best support the achievement of contract requirements.	
1988	Tom Gilb "Principles of Software Engineering Management"	The Evo method: - frequent evolutionary delivery, emphasis on defining; - quantified measurable goals and then measuring the actual results from each time-boxed short iteration.	The first book with substantial chapters dedicated to IID discussion and promotion.
1993	Easel Co	Projects using small, cross-functional teams produce the best results.	Appearance of Scrum.
Early 1990's	Project to build a new-generation Canadian Automated Air Traffic Control System (CAATS)	Using a risk-driven IID method, a series of six-month iterations.	The project was a success, despite its prior near-failure applying a waterfall approach.
Middle 1990's	Rational Corp.	Promotion of the daily build and smoke test, a widely influential IID practice institutionalized by Microsoft that featured a one day micro-iteration.	Creation of Rational Unified Process.
1996	Chrysler C3	Using pair programming, releases in short development cycles, code review, unit testing.	Appearance of Extreme Programming.

Table 2. Definitions of software development agility (adapted from Laanti 2012, according to Kettunen 2009)

Publication	Definition	Focused attributes
Aoyama 1998	Quick delivery, quick adaptations to changes in requirements and surrounding environments.	- frequent deliveries - adaptive to changes
Cockburn 2001	Being effective and maneuverable; use of light-but-sufficient rules of project behavior and the use of human and communication-oriented rules.	- flexible - light methods - people- and communication-oriented
Highsmith 2002	Ability to both create and respond to change in order to profit in a turbulent business environment.	- adaptive to changes
Larman 2003	Rapid and flexible response to change.	- accelerating development - adaptive to changes
Schuh 2004	Building software by empowering and trusting people, acknowledging change as a norm, and promoting constant feedback; producing more valuable functionality faster.	- people-oriented - adaptive to changes - accelerating development - taking advantage of feedback
Subramaniam & Hunt 2005	Uses feedback to make constant adjustments in a highly collaborative environment.	- communication-oriented - feedback-driven changes
Ambler 2007	Iterative and incremental (evolutionary) approach to software development which is performed in a highly collaborative manner by self-organizing teams with "just enough" ceremony that produces high quality software in a cost effective and timely manner which meets the changing needs of its stakeholders.	- iterative - incremental - highly collaborative environment - self-organizing teams - adaptive to changes
IEEE 2007	Capability to accommodate uncertain or changing needs up to a late stage of the development (until the start of the last iterative development cycle of the release).	- adaptive to changes - iterative
Wikipedia 2007	Conceptual framework for software engineering that promotes development iterations throughout the life-cycle of the project.	- iterative
Wikipedia	Group of software development	- iterative

2012-13	methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams.	- incremental - self-organizing, cross-functional teams - highly collaborative environment
Larman & Vodde 2009	Be agile …. the Merriam-Webster dictionary defines agile as a ready ability to move with quick easy grace …. Agile means agile — the ability to move with quick easy grace, to be nimble and adaptable. To embrace change and become masters of change — to complete through adaptability by being able to change faster than your competition can.	- accelerating development - flexible - adaptive to changes
Appelo 2011	Agility is about staying successful in ever-changing environments (page 376). Agile has never been some specific set of practices (page 377) but rather has its roots in complexity theory (page 11) — and solutions depend on the problem's context.	- adaptive to changes
Leffingwell 2011	Adaptive (agile) processes … assumed that — with the right development tools and practices — it was simply more effective to write the code quickly, have it evaluated by customers in actual use, be "wrong" (if necessary), and quickly refactor it than it was to try to anticipate and document the requirements up front.	- fast - frequent deliveries - taking advantage of feedback - quickly responsive to the feedback
Rasmusson 2014)	Agile is a time boxed, iterative approach to software delivery that builds software incrementally from the start of the project, instead of trying to deliver it all at once near the end.	- time-boxed - iterative - incremental - frequent deliveries
(Beal 2014)	Agile development is a phrase used in software development to describe methodologies for incremental software development. Agile develop-	- incremental - highly collaborative environment - team decisions

	ment is an alternative to traditional project management where emphasis is placed on empowering people to collaborate and make team decisions in addition to continuous planning, continuous testing and continuous integration.	- continuous planning - continuous testing - continuous integration