TAMPERE UNIVERSITY OF TECHNOLOGY

Mikko Pohja

Quality control in a startup software project

Master of Science Thesis

Examiners: Tommi Mikkonen
Examiners and topic approved in
the Information Technology
Department Council meeting on
04.05.2016

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
Mikko Pohja: Laadunhallinta startup-yrityksen ohjelmistoprojektissa
Diplomityö, 61 sivua
Toukokuu 2016
Pääaine: Ohjelmistotuotanto
Tarkastajat: Tommi Mikkonen
Avainsanat: Startup, Laadunhallinta, QA

Moderni ohjelmistokehitys on tehnyt mahdolliseksi nopeiden prototyyppien kehityksen pienellä budjetilla ja lyhyellä aikataululla. Tästä on seurannut startup-yritysten nopea yleistyminen. Startup-yritykset voivat kokeilla ja kehittää liikeideaansa nopealla aikataululla käyttäen edistyneitä ohjelmistokehyksiä ja noudattaen ketterän ohjelmistokehityksen periaatteita. Kokenut tiimi kykenee luomaan toimivan prototyypin jo muutamassa viikossa.
Nopea ohjelmistokehitys saattaa johtaa ohjelmiston lähdekoodin ja rakenteen huonoon laatuun, joka hidastaa ja vaikeuttaa kehitystä tulevaisuudessa. Ohjelmiston laatuun täytyy alkaa kiinnittää huomiota viimeistään siinä vaiheessa, kun liikeidea todetaan menestyväksi ja ohjelmiston kehitystä jatketaan prototyyppivaiheesta eteenpäin. Lisäksi startup-yrityksissä ohjelmiston laatu riippuu kriittisesti toteutettavan järjestelmän sopivuudesta kohderyhmälle. Tämän takia järjestelmän sopivuuden jatkuva validointi kehityksen aikana on tärkeä osa kehitystä.
Tässä diplomityössä on käsitelty perinteisiä menetelmiä laadunvarmistukseen sekä niiden tehokkuutta. Myös startup-yrityksiä sekä niiden laatukäsitystä on tarkasteltu yleisellä tasolla. Lisäksi ohjelmistojen laadun parantamiseen käytettyjä tekniikoita on tarkasteltu startupin näkökulmasta sekä yhdistetty perinteisiä laatumenetelmiä startupille tyypilliseen laadunvarmistukseen. Lopuksi esitellään startup-yritykselle toteutettu ohjelmistoprojekti sekä käsitellään sen laadun toteutumista tehtyjen haastattelujen pohjalta.
Työn tuloksena todettiin, että laatukäsitykset eroavat perinteisen ohjelmistokehityksen ja modernin startup-yrityksen ohjelmistokehityksen välillä. Myös laadun parantamiseen käytetyt metodit eroavat näiden kehitysmuotojen välillä. Perinteiset menetelmät luottavat enemmän teknisiin lähestymistapoihin, joilla löydetään virheitä, kun taas modernissa laadunparannuksessa keskitytään enemmän ihmisiin ja kehitysprosessiin. Perinteiseen laadunparannukseen liittyviä teknisiä toimenpiteitä, kuten katselmointeja ja erilaisia testausmuotoja, voi kuitenkin soveltaa modernin laadunparannuksen osana.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
Mikko Pohja : Quality control in a startup software project
Master of Science Thesis, 61 pages
May 2016
Major: Software Engineering
Examiner: Tommi Mikkonen
Keywords: Startup, Quality control, QA

Progress in modern software development has enabled tiny prototypes that can be implemented with small budget and short schedule. Since proving the business ideas with fast prototyping has become so easy, the number of startup companies has begun to grow. A team with sufficient experience can implement a simple working prototype as fast as in weeks.

Development of these simple prototypes can lead to poor quality of code and structure of the product, which can complicate the future development. This can become an issue if the business idea is indeed validated and proven to be successful. Additionally in startup environment, product quality usually means the value it brings to the customer. Because of this, continuous validation of quality is required throughout the life cycle of the product.

This thesis discusses software quality in both traditional software development and software startup environment. Methods traditionally used for improving quality and their efficiency are presented. These methods are also joined to the quality methods recommended for software startup environment. Finally, this thesis presents an example project done for a software startup.

Conclusions from this thesis include that the definition of quality and methods improving it vary between traditional software development and modern startup environment. In traditional software development, methods for improving quality are focused on technical activities discovering defects. In turn, modern methodologies concentrate more on people and processes. However, activities from traditional quality improvement can be applied to the methods recommended for startup environment.

PREFACE

Writing software is easy. Writing about writing software is hard. This thesis took more time and resources than it was allowed but here it is. Done and finished.

I would like to thank Professor Tommi Mikkonen for reading and commenting countless versions of this thesis. His advices and motivational comments helped me continue to the finish. I could not hoped for a better examiner.

My employer Futurice enabled me to take the time for writing this thesis by providing support and resources. Most of all I would like to thank John Laukkanen for guiding me through this journey. And everyone in Tammerforce: MorjestA.

Having a relevant project to reflect all the theory againts was really helpful. So I would like to thank MukavaIT and specially Petri Järvinen for participating in an interview and allowing the use of Päikky project as a case study.

Mom and dad, one has to thank mom and dad: thank you.

And finally I would like to thank my wife, who has been pushing me to finish my studies and helping in it as best as she can. And thank you my children for not helping me with this thesis but getting it out of my head when needed.

Tampere, May 22, 2016 Mikko Pohja

CONTENTS

1. INTRODUCTION

Recent progress in software industry has made developing new products and services in the field of software development increasingly easy. New and improved programming languages and frameworks are introduced constantly. These frameworks provide easy to start platforms for new software development removing the need for excessive planning and understanding of many low-level implementation details. In addition, the need for expensive hardware setup has come obsolete as several instances have appeared offering reasonably priced execution environments as a service.

These changes in the industry along with the progress in development processes have brought the possibility to create software business closer to every would-be entrepreneur. Developing software with Agile or Lean methodology with modern technologies have lowered the lead time and resources required. Experienced software developers can implement a prototype as fast as in weeks starting from scratch and ending in having a working product publicly available. Attempts to create the next big software product are on a rise as the stories of success spread around the globe.

Since the number of attempts for success are increasing, so is the number of companies succeeding. When a product built with minimum effort starts to prosper, the demand for high quality emerges. As products done by traditional software development are built with large amounts of effort used to improving the quality of the product, modern startup entrepreneurs may become confused when considering the quality of their product. The methods and activities used in traditional software development may seem separate from the rapid development of modern products. In addition, modern software development methodologies focus on different aspects of software quality than in traditional software development.

This thesis presents software quality in a startup environment and suggests the activities for improving it. Traditional quality improvement methods are presented and linked to the methods recommended in startup environments. Also a real-life software project for a Finnish startup is presented for illustration purposes. The project, executed in a Finnish software company Futurice, is evaluated by using the data gathered after the project from interviews of the project stakeholders.

The rest of this thesis is organised as follows. Chapter 2 introduces the background by defining software quality and startup environment. Chapter 3 describes the methods and activities improving software quality and divides them in phases of traditional software development. Chapter 4 redefines quality in context of a software startup and describes the methods of quality improvement applicable in startup software development. Chapter 5 presents the execution and phases of the case project and evaluates the quality achieved based on the interviews of the stakeholders.

2. BACKGROUND

This chapter describes software quality and startup environment. Section 2.1 describes software quality in general and the motivation behind improving quality. Section 2.2 explains the context of a startup environment and the phases in the life of a startup.

2.1 Software Quality

Quality is an attribute of the item in question. It is often described as a combination of qualitative and quantitative attributes. Quality is an ambiguous attribute, because it is subjective and thus differs when viewed from different perspectives. The American Society for Quality has two definitions for technical quality: 1. the product's ability to satisfy its needs; 2. the product's lack of deficiencies [19]. Software is one of the most used types of product in human history. At the same time it has one of the highest failure rates of any type of products mainly due to poor quality. With those facts, it is clear that the total influence of low quality software is considerable in both money and time. Still it is a known fact that a common practice to cut costs is reducing the effort used in software quality. Convincing the payer to allow using effort to achieve good quality can be difficult but crucial. The topic is widely researched and the results speak on behalf of quality. [11]

2.1.1 Motivation

Phil Crosby has made popular a concept that establishing a quality program will return in savings more than the program costs and thus "quality is free" [6]. Even though Crosby's concept is used mainly in the manufacturing sector, it has some truth that can be applied to the software business. In addition, the "cost of quality" is a slightly inappropriate term, considering that quality in itself does not create costs but the lack of it does.

Studies have shown that software quality has huge impact on project costs and success. Measurements on 10000 function point projects show that about 31% of projects of that size come to an end by cancellation. The average cost of these canceled projects is about $35,000,000. Successful projects of similar size with good quality have about 40% lower costs. These figures endorse the effort put towards the software quality and make it clear that at least in large scale projects, quality control should not be ignored. [11]

Capers Jones has listed several points that make high quality a major economic benefit. In the development of large systems, high quality from the beginning can reduce the probability of cancellations. Software projects can also benefit by achieving shorter development schedules. Shorter schedules with high quality also lower the costs of a project. Lower development costs, maintenance costs and warranty costs can add up to considerable amounts of cost savings. In addition to the quantitative benefits, high quality raises the satisfaction of customers, end-users and developers. [11]

Jones expresses concerns towards the poor measurement of software quality causing executives and even quality personnel to treat software quality as an expense. Those participants may also treat quality as an issue that is raising the development costs and increasing development schedules. On the contrary, Jones summarizes the benefits of high quality: "However, from an analysis of about 13,000 software projects between 1973 and today, it is gratifying to observe that high quality levels are invariably associated with shorter-than-average development schedules and lower-than-average development costs". [11]

2.1.2 Software Quality Defined

The word "quality" has many tones. This complicates defining quality and especially software quality. It can be understood as elegance or beauty, fitness of use, satisfaction of user requirements, freedom from defects, high reliability, and ease of use, among multiple other things. These descriptions appear even more complicated considering that quality and its attributes are bound to not just the observer but also the operation context in question. While a software component can have excellent quality in some context it can still be even dangerous in others. The same applies to several attributes of quality. A component can be fit for some use, but defective in different contexts. Some component can satisfy the users requirements in one environment, but can be useless in others. [11]

International quality standard in software is defined in ISO 25010 [9]. It was brought up to date in 2011 from ISO 9126 published in 1991 [10]. ISO 25010 introduces a quality model which classifies software quality in a set of characteristics each having a number of sub-characteristics. In the descriptions of the quality model's characteristics, it is assumed that the operation context is known and predetermined. Functional suitability is the degree to which the product provides functions that meet stated and implied needs. Reliability is the degree to which the product can perform specified functions for a specific period of time. Operability is the degree to which the product has attributes that enable it to be understood, learned, used and attractive to the user. Performance efficiency is the amount of resources the product uses under certain conditions. Security is degree of protection of information and data from unauthorized persons or systems trying to read, modify or access them. Compatibility is the degree to which two or more systems or components can exchange information, including, but not limited to, performing their required functions while sharing the same hardware or software environment. Maintainability is the degree of effectiveness and efficiency with which the product can be modified. Transferability is the degree to which a system or component can be effectively and efficiently transferred from one hardware, software or other environment to another. [17]

In addition to the quality model, the 25010 standard defines the model of software quality in use. This model consists of five main characteristics. Effectiveness is the accuracy and completeness with which users achieve specified goals. Efficiency is the amount of consumed resources in relation to accuracy and completeness with which users achieve goals. Satisfaction is the degree to which users are satisfied with the experience of using the product.Safety is the degree to which a product does not lead to a state in which life, health, property, or the environment is endangered. Usability is the extent to which product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction. [17]

2.2 Startup

The term became popular during the dotcom bubble, when a great number of companies were founded to do business on the Internet. Steve Blank lists some principles of a startup in his blog post "What's A Startup? First Principles.". Blank defines a startup as "an organization formed to search for a repeatable and scalable business model". The first goal for a business model can be revenue, profits, users or click-throughs. The business model must be quickly and constantly tested and iterated by using Agile approach. Blank also claims that the business model of most startups is changed multiple times. [3]

2.2.1 Lean Startup

Lean Startup is a movement pioneered by Eric Ries, which brings the principles of lean manufacturing to the context of entrepreneurship. Like the lean manufacturing is measuring progress also the lean startup measures its progress for discovering and eliminating the sources of waste. Lean manufacturing measures the progress by the production of high-quality goods, whereas in lean startup, the unit of progress is something Ries calls validated learning. This section is based on the book Lean Startup by Eric Ries [18].

Ries' definition of a startup is "a human institution designed to create a new product or service under conditions of extreme uncertainty". This definition omits the size of the institution and thus implies that a startup can be ranging from an individual or a small group of people to a team or division inside a large company. The most important aspect of the definition is the extreme uncertainty, which needs constant measurement and steering of the process.

There are five principles in Lean Startup

1. Entrepreneurs are everywhere. Entrepreneurship involves everyone working inside a human institution in where new products and services are created under uncertain conditions. This means that Lean Startup approach can be applied in any size company from a single person working in a garage to even a large enterprise, leaving no industry out.

2. Entrepreneurship is management. Treating a startup as just a product is not a valid approach. A startup is an institution having constant uncertainty present, so it requires new kind of management.

3. Validated learning. Startups exist primarily for learning how to build a sustainable business. This learning can be validated using frequent experiments for testing the vision.

4. Build-Measure-Learn. The main function of a startup is to turn ideas into products, measure the customer response and then learn whether to pivot or persevere. All successful startup processes should be aimed to accelerate the feedback loop shown in Figure 2.1

5. Innovation accounting. Improving the outcome of entrepreneurs and holding innovators accountable means focusing on measuring the progress, setting milestones and prioritizing work. This requires new kind of accounting designed for startups.

2.2.2 Life Cycle in Lean Startup

The life cycle in a Lean startup software development does not match to the traditional life cycle of a software product. In building a successive product for a software startup, there are usually multiple short iterations surveying the context in which the product will be used. Building a software startup can be started without knowing what the customers want or even who the customers are, so the life cycle of the product has to be built from multiple consecutive version with significant and rapid changes for conforming to the customers demands.

The basic repeating process of development in Lean Startup begins from an idea. This idea contains usually several hypotheses about the customer behavior or the context of usage. Some of these are called leap of faith assumptions, which are the riskiest hypotheses of the plan. Using these hypotheses for validated learning, allows avoiding much of the waste usually present in startups. To utilize the validated

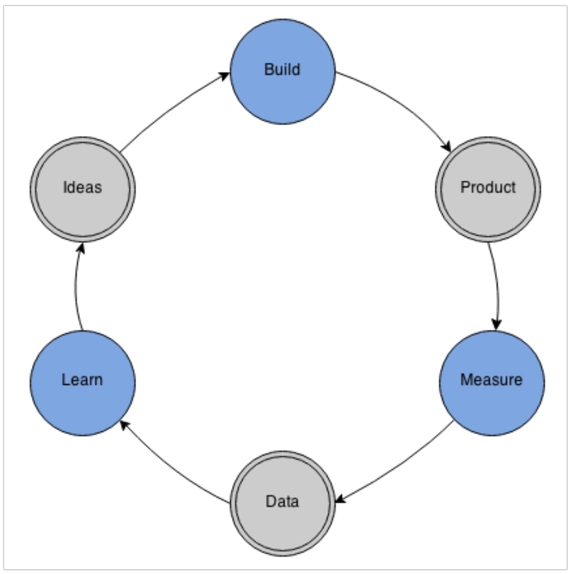

Figure 2.1: Build-Measure-Learn feedback loop

learning as a scientific method, there needs to be a selected group of hypotheses to test. The leap of faith assumptions should be the first ones to be tested, because they are the foundation on which the business is to be built. Testing the first hypotheses can be sometimes done with virtually nothing built. This has been done for example by the founder of the online shoe store Zappos, who sold the first pairs of shoes without having any inventory of his own but only some pictures of shoes taken in local shops.

When the first hypotheses have been proved true, hopefully with only a small effort, the Build phase can be initiated and the first minimum viable product (MVP) can be built. This product is a version which enables a full turn of the Build-Measure-Learn loop with minimum effort. This MVP is usually missing most of the features that will be proven essential later. The purpose of this version is that it must be able to measure its impact. The target audience of this product is not the development team or some business heads, but the potential customers, so it can evaluate the reactions of the market.

After the MVP is released, the startup enters the Measure phase. In this phase, the most important topic to get answer for is whether the product development efforts are leading to real progress. In other words: is the product something someone wants. If building something that nobody wants, there is no sense in using time and money on the development. Ries suggests doing the measuring with a method called innovation accounting. It is a quantitative approach for testing if the tuning is successful. It allows creating learning milestones, which are useful for the entrepreneurs for evaluating their progress accurately.

Once the entrepreneurs have learned from the measurements done, it is time for the most important step in the cycle. In this step, the entrepreneurs must assess the success of the hypotheses and based on these assessments, they must decide whether to pivot the original strategy or persevere. If any of the hypotheses proved to be false, it is time for pivot: to make a major change to a new strategic hypothesis. One of the most important objective of the Lean Startup is to allow the recognition of the time to pivot soon, wasting less time and money.

Even though this cycle is called Build-Measure-Learn according to the order of execution, the planning of the cycle is done backwards. The first thing to plan is what is needed to learn. Then using innovation accounting, the things needed to measure are figured out. The last thing is to design the product able to run the experiment returning the measurements.

Eric Ries reveals multiple success stories using Lean Startup, in where there are many similarities. One of the success stories is about a collaboration portal for voters. The first released version of the portal was achieved with 1200$ and three months of work. With that version, some of the assumptions could be tested in the real operating environment and the product could be developed further. After the partial success of the first version, there were five other version of the portal and the business model developed. After each of the launches, the leaps of faith were tested and the necessary changes were made for steering the product to the right direction. After 16 months of development the product had reached a sufficient state for a sustainable business. During these months, there had been several pivots and changes, which cannot be seen as a traditional product life cycle.

3. IMPROVING SOFTWARE QUALITY

In this chapter, the basic methodology and activities for improving quality are introduced. Section 3.1 divides the life cycle of a software into separate periods from the perspective of quality improvement. Section 3.2 introduces the defect prevention and related activities. Section 3.3 describes the period and activities prior to testing phase. Section 3.4 explains the testing phase and different methods involved. Section 3.5 describes software quality after the release. Unless stated otherwise, the content of this chapter is based on the work of Capers Jones presented in the book The Economics of Software Quality [11].

3.1 Quality Improvement Across the Life Cycle

Quality of a software product can be influenced throughout its life cycle with multiple approaches. Pursuing high quality means having methods in use for both decreasing the amount of defects and maintaining good structural quality. In addition to these technical approaches, projects should have methods to assure high quality of specification and implementation process.

The life cycle of a traditional software project can be divided to periods targeting different quality aspects. The life cycle includes periods concentrating on defect prevention, pretest defect removal, testing and post-release quality improvement as presented in Figure 3.1. Preventive and pretest periods contain actions such as reviews, inspections and audits. Testing includes various types of testing the functionality. Post-release methods focus on maintainability, defect discovery and defect repairing.

Solid foundation for high quality is built with good specification, requirements and planning in the beginning of a project. When development begins, there should be ways to prevent as much defects as possible. As defects appear anyway, they should be detected and fixed as early as possible. Detecting the defects early lowers the effort needed to fix them.

The hardest defects to remove are found in the requirements and design because testing and static analysis cannot find them. This is because these defects tend to be deficiency of features and errors of logic rather than errors in code. These defects are usually situated in the beginning of the life-cycle and thus require great effort to be removed.

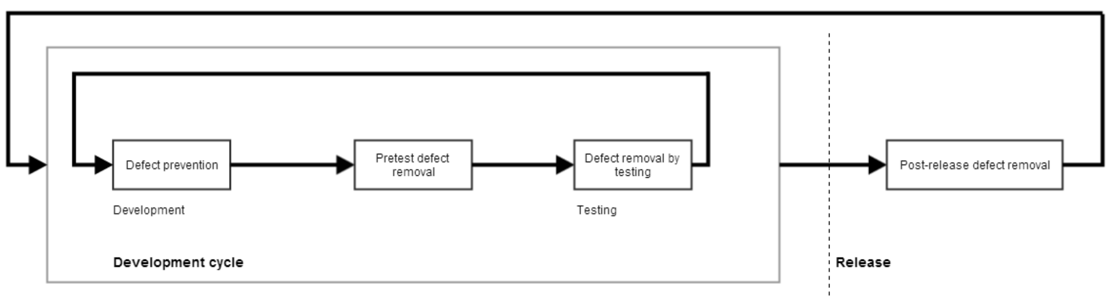

Figure 3.1: Phases of software quality improvement in the software life cycle

At some point, the project can initiate testing phase. Testing is used to systematically find defects in the software. Tests can be aimed to different areas of the software and the range of different tests used is dependent of the project. Big and critical projects should use comprehensive testing whereas smaller projects can get along with smaller amount of tests and lesser coverage.

Quality cannot be forgotten when the software is released. Because defect removal efficiency can never reach 100%, there are always defects after the release. Some of those defects may have been found in the previous phases, but not removed, and other defects were unknown to the project team at the time of the release. Quality methods after the release should include detecting and removing defects and increasing the design for increasing the maintainability. With this range of methods divided to the software life-cycle, every project should choose the most appropriate methods to be used in the project in question.

3.2 Preventing defects

Defect prevention is a set of methods used to lower the amount of defects coming from one or many of the defect origins. Software defects are originated from different parts of the project. Defect origins can be technical and nontechnical. Some nontechnical origins of defects include requirements and documentation. Technical origins include architecture, design and code. Because defects are originated from multiple sources, there is no single method for covering them all. Most methods are not effective against all sources. Usually, 1-4 defect prevention methods are used.

Most of the defect prevention methods are not primarily used for preventing defects, but for some other purpose. Preventing the defects is usually a secondary effect and can be sometimes incidental. These methods do not affect structural quality, but the defects of the software. For achieving high total quality of software, defect prevention should be combined with other types of quality improvement methods.

Defect prevention is one of the most difficult topics of software quality. It is hard to measure, improve and prove the economic value. The difficulty originates from the fact that defect prevention is a negative factor that reduces defect potentials. This means that reliable measurement of the efficiency needs multiple points of reference for both using the method and not using it. Despite this, a number of big companies has studied the topic with significant amounts of projects. IBM, for one example, has been studying this topic since its first studies in 1970s.

3.2.1 Formal Inspections

Formal inspections were inspected as a one line of research by IBM in the 1970s. The inspections were targeted at requirements, design documents, source code and other deliverables. In a short time they discovered that the defect removal efficiency with formal inspections could reach levels beyond 85%. At that time that was higher than any form of testing could achieve. In addition, the inspections seemed to affect the accuracy and completeness of the requirements and specification documents, which lead to raising the defect removal efficiency of testing by 5%. Combining formal inspections with formal testing could raise the efficiency even further to levels as high as 97%.

With these improvements in efficiency, the amount of defects in the beginning of testing was reduced significantly. The schedules and budgets of the testing could be cut to half and in some cases even more than half. That lead to about 15% decrease in combined schedule and cumulative effort compared to similar applications without inspections.

Another result from the studies was that using the formal inspections for a longer time, the project teams unconsciously started avoiding the kind of errors found in the inspections. This meant that the inspections not only removed defects but also prevented them from occurring.

3.2.2 Static Analysis

Static analysis is used to detecting syntactic and structural defects in source code without compiling or executing it. It is used in all sizes of software projects and with every type of applications. The concept is originally from the compilers, which performed syntax checking. Later in the 1970s, the features for detecting defects were improved in a tool called Lint. Static analysis has been further developed over time and nowadays it can comprehensively analyze system-level structure and even security vulnerabilities. Modern tools for static analysis can have defect removal efficiency of over 85%.

Static analysis tools are based on a library of rules defining the conditions to be examined. Some of the modern commercial tools contain over 1500 rules and allow the users to define their own rules for special conditions. Static analysis is effective in defect removal for using the rules to seek out and eliminate syntactic and structural defects. There are two reasons static analysis is also useful in preventing defects: the rule libraries are also useful for preventing defects and the tools can suggest corrections for defects to the developers. The latter enables the developers to see the effective solutions to the defects while examining the results of the analysis.

Automated static analysis of source code is used in both defect prevention and pretest defect removal. Statistics by Capers Jones show that the usage of automated static analysis exceeds 75% in most types of software projects. The first mentions of automated static analysis are from 1979 from the first release of Lint. In current modern software development, static source code analysis is automatically done by most of the Integrated Development Environments (IDE). IDEs, such as Eclipse and IntelliJ Idea, perform automatic analysis immediately after every minor change.

Static analysis tools are not only very quick and effective but also fairly inexpensive. Because of this, static code analysis has become one of the most used quality methods in software industry. There are tens or even more tools for source code static analysis in the market, both open source and commercial. For such an inexpensive and effective method, one could imagine the market penetration being close to 100%. Jones suggests that the reason for this not being true is that humans have a natural resistance for new ideas even though the turn out to be valuable.

3.2.3 Test-Driven Development

Test-driven development (TDD) is a development process where the development consists of short repetitious cycles of development. A cycle comprises an initially failing test case, minimum amount of code to pass that test and refactoring of the code. The development process embraces the phrase "clean code that works" by Ron Jeffries. Kent Beck analyses the benefits of that statement in his book Test-driven Development: By Example. [1]

Writing clean code that works can help developers by allowing a predictable flow of development. Using tests to define the finished state of a task helps developers know when the task is finished. This is contrary to a common way of development, where developers may be uncertain whether the task is finished or is there still some trail of bugs to fix. Another benefit is that when the developers aims to clean code, instead of building the first thing they think of, they can learn different sides of the problem thinking about another solutions. These benefits lead to enhancing the lives of the users and developers and the whole team. The project team can achieve better trust between the developers and the individual developers can feel better when writing clean code. [1]

Writing clean code that works is not such an easy task. Anyone working in the software development can admit that there are many forces driving the development further from clean code and even from code that works. One solution is using automated tests as the driving force of the development. This is called Test-Driven Development. There are two cornerstones in TDD: new code is written only if an automated test has failed and duplication is eliminated. These rules seem simple enough, but they can actually produce complex behavior for individuals and the whole team. The team must be able to choose between decisions by getting feedback from the running code. Every developer must write his own tests in opposite to waiting for someone else to write the tests. The development environment must be quick enough to provide instant feedback on small changes. The design of the software must allow easy testing by using simple, loosely coupled components. [1]

These complex requirements imply a specific order of activities in development. TDD defines the steps of development and the order of executing them as Red, Green and Refactor. Red and green are the colors of the test success. First a simple test is written so that it will not succeed. The test will not sometimes even compile. Then the test is made green, successful, by not avoiding any means necessary. After the green is achieved, the code is refactored to eliminate all of the duplication created while still keeping the test green. [1]

3.2.4 Agile Approach

Agile approach is a set of guidelines based on a publication made by 17 software developers. The developers had gathered to discuss about lightweight development methods and the result was the published in The Agile Manifesto. The people in the signature of the manifest formed the Agile Software Development Alliance.

The Agile Manifesto reads as follows:

> "Seventeen anarchists agree:
>
> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan.
>
> That is, while we value the items on the right, we value the items on the left more.
>
> We follow the following principles:

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

These methods are based on the actual methods present in the work done by the 17 developers in that time. The purpose of the manifest is not to give definite answers but some guidelines of how to prefer the aspects of software development. It is not trying to tell how things are done but help developers with agile approach. When listing the things to value, the purpose is not to underrate the aspects with lower priority but give some hints about what brings the most value to the project. [2]

Embedded users. Embedded users is an Agile method where one or more user representatives are embedded into the project team. The purpose of the user representative is to work in cooperation with the developers creating the critical requirements which are then implemented in short sprints. The idea is to build the specific business critical features and get those running as quickly as possible.

The embedded customer representatives are also used to give support in reviewing the features and requirements. The main purpose of this method is to improve the requirements definition.

This method is proven to be useful in small software projects under 2500 function points and effective in projects with under 100 users and size below 1000 function points. In larger scale applications, over 10000 function points or more than 1000 user, a single representative cannot effectively provide enough requirements. This method can however be scaled up by using multiple user representatives, but like usually with Agile approach, this is most effective in smaller projects.

3.3 Pretest Defect Removal

Capers Jones suggests that in every software project there should be multiple pretest QA methods used. Jones lists a combination of methods for both small and large projects. A small software project, in this context, is described to have a maximum amount of 1000 function points or 50 000 source code statements. These small projects are generally executed by a team with less than 6 software developers. These teams usually have no specialists for any quality methods, but the developers are generalists handling requirements, design, coding and testing. In many cases with Agile approach, there is a users representative embedded in the team providing requirements and customers viewpoint in real time. Jones reminds that removing defects with high efficiency requires trained and technically skilled software engineers instead of generalists. However, this is not as necessary in small projects, since fortunately these projects have usually low defect potentials.

The origins of defects in small studied projects are split into five categories. Source code is the most common origin of defects. About 1.75 defects per function point are found in source code and this leads to 1750 defects in whole projects. Software design is the second most common source of defects. Design is the origin of 1.00 defects per function point. Requirements are causing 0.75 defects per function point and documentation nearly as much with 0.65 defects. Poorly executed fixes are the origin of 0.27 defects per function point. All together these five are the source for 4420 defects in a whole 1000 function point software project. These figures represent the approximate averages and the actual values can be as much as 25% lower or higher for every source.

Jones presents a suite of pretest defect removal activities and their efficiencies for small projects. This suite includes:

1. personal desk checking (subsection 3.3.1)
2. Scrum sessions(subsection 3.3.4)

3. client reviews of specifications(subsection 3.3.2)
4. informal peer reviews(subsection 3.3.3)

Each of these forms of defect removal activities are targeted towards a specific type of defects, but other types of defects can be found during the activities. Jones gives several figures for the efficiency of each activity. These figures can only be created by companies that have complete accurate defect measurement programs. Because of this, these figures can vary from context to other and thus are indicative. These figures still illustrate two major problems in the software industry: the removal efficiency levels are comparatively low for most of the removal activities and the defect removal is much harder for requirements and design. The first one leads to a need for numerous kinds of defect removal activities. The latter means that a significant amount of effort must be used to assure the quality of requirements and design. Defects in requirements and design must be removed prior to testing, because the testing cannot find them. Also static analysis is incapable to finding them, because the defects are not bugs found in the code.

3.3.1 Personal Desk Checking

Personal desk checking is a manual operation in where the logic of an algorithm is checked by the creator of the algorithm . The logic used in the desk checking is presented as a pseudo-code rather than the implemented actual program code. The algorithm is executed by a person acting as a computer. The person performing the desk checking carefully follows the algorithm while filling a table of notes with pen and paper.

The notes form a table which include columns for: line number, variables in use, conditions, input and output. Line number is necessary to identify the line being executed. All variables have a column in alphabetical order. As the value of the variable changes, the appropriate column is filled up. Conditions columns include a column for every condition in the algorithm which shows the result of the condition in either true (T) or false (F). The condition column is updated whenever the condition is evaluated. Input and output columns are used for the inputs got from the user and the output from the program. [14]

Desk checking is the oldest form of software defect removal. It has been in use since the beginning of the history of computers. In the early days of computing, testing the programs was difficult because of the limited numbers of computers. The computers worked on production work in the daytime and often the testing had to be done at night. In those days, testing had only an efficiency of 70% in finding bugs, because of the primitive test case design and limited time available for testing. Therefore the desk checking were a necessary addition to testing. Nowadays the desk checking is still a common activity for removing defects prior to testing. Desk checking today can be enhanced by using static analysis for program code and spell checkers and complexity tools for text documents.

Personal desk checking is used mainly in low-level code. Approximately over 75% of low-level code and under 30% of high-level code in projects are checked using personal desk checking. Additionally, personal desk checking is used for over 75% of text documents, such as requirements. The execution time of desk checking is around 80% of normal reading speed of text. This leads to about 5 logical statements per minute for source code.

The efficiency of defect removal for personal desk checking is between 25% and 50% averaging 35%. The reason for these relatively low figures can be found in human nature. Humans have a natural tendency to ignore their own mistakes. A developer making an error usually does the error thinking that the action was correct. Therefore when the developer checks the code for defects, the train of thought can remain the same and the defect is not found. This could be avoided by using proofreaders or copy editors, which is a rare habit, but could be profitable for software projects. Another solution for avoiding the blindness to own mistakes is to use peer reviews.

3.3.2 Client Reviews of Specification

Client reviews of specification is as well among the oldest of defect removal methods. It is still not enough used in average software projects. One of the authors of "The Economy of Software Quality" has got experience in lawsuits for canceled or defective projects. In most of the lawsuits the supplier of the software accuses the customer for failing to review the designs and other documents of not making a remark about any problems during the reviews. In some cases the customer has even accepted the materials quoted in the trial.

The products reviewed by customers usually do not include inner workings of software applications, like source code, test cases or detailed design. Capers Jones presents a list of 12 major items in directly funded software projects, which usually are reviewed:

1. Requirements.
2. Requirements changes.
3. Architecture.
4. High-level design, user stories, use cases.
5. Data flow and data structure design.

6. Development plans.
7. Development cost estimates.
8. Development quality estimates.
9. Training materials.
10. Help text and help screens.
11. Features of COTS packages.
12. High-severity bug or defect repairs.

Client reviews are an important practice as the clients are paying for the software. There are many clients being active participants in reviews and paying serious attention to the deliverables of software project. Simultaneously some clients are overlooking the reviews and falsely assuming that the software teams know what they are doing. The lawsuits speak on behalf of the reviews, having passive or partial reviews as a worrying feature.

The statistics show that the usage of client reviews of specifications are used in under 50% of U.S. software applications. The defect removal efficiency varies greatly being at lowest under 15% and sometimes reaching over 45%. The average efficiency is around 25%, which Jones calls "marginally adequate". The effort taken by client reviews is over 2 hours per participant for preparation and over 4 hours per participant for execution. Client reviews work best when the client is directly available, and the applications having indirect clients cannot take full advantage of it.

3.3.3 Peer Reviews

Peer reviews are almost as old as desk checking. The idea of peer reviews is close to the idea of desk checking. The essential difference is that in peer reviews, the code, documentation or other product under review is checked by another person. This prevents the defects from being hidden by the human tendency to ignore their own mistakes. Peer reviews are best suited in small projects with under five developers.

In larger projects, because of the informal nature of peer reviews, the efficiency of defect removal is better with formal inspections. This leads to informal peer reviews to being the secondary method of defect removal in large projects. In large projects using Agile approach without formal inspections, peer reviews can be highly important.

Peer reviews are targeted to find technical, structural and logical defects. This means that peer reviews are not to be thought as the same as proofreading or copyediting. However, the findings from these can partially overlap each other.

In addition to removing defects in the pretest phase of projects, peer reviews can benefit projects other ways. Peer reviews can achieve the same effect as formal inspections: the participants of the review tend to unconsciously avoid the problems found in the review. Reviews can also be useful for learning. Novice developers can learn while reviewing the work of more experienced developers. Furthermore, experienced developers can remark the problems the novice members need to understand. Even beginner reviewing other beginners work can be better than nothing. Reviews done by expert for the work of expert can be highly efficient, but these cases can sometimes lead to social problems with big egos colliding when mistakes are being pointed out.

Informal peer reviews are used in under a half of software applications. The achieved defect removal efficiency is usually between 35% and 65% and the average removal efficiency is 45%. Reviews can take up to 30 minutes to prepare and the execution pace is around 70% of the normal reading speed for text, and around 3 logical code statements per minute for source code. Best results for peer reviews can be achieved with small projects using Agile approach.

3.3.4 Scrum Sessions

The development teams in Scrum are usually formed by a Scrum master, embedded user representative or stakeholder, possible specialists and three to five developers. In average software development, the team usually has around five software engineers plus one or more specialists as needed. Specialists may include technical writers, business analysts or database specialists. As defined in Agile and Scrum, teams should be self-organized and consist of generalists. Thought in many cases having specialized requirements, some specialists are needed.

Projects using Agile and Scrum guidelines are split into small units of work. These units are called "sprints" and the development work needed to complete the unit can be achieved in two-week period. The embedded user in the team is responsible to provide the requirements for each sprint. The end result of a sprint should be the source code and supporting documentation ready to be published in production.

One of the principles of Scrum is to have daily Scrum meetings or "stand ups". The latter name comes from the idea that the people attending the meeting should be standing up so the meeting can stay within the time limit of 15 minutes. During these meetings, every member of the team should describe three things: what was done yesterday, what will be done today and are there any problems that will slow things down. The most interesting topic in the context of defect removal, is listing the issues, bugs and problems there is and simultaneously sharing the knowledge among the team.

Agile and Scrum methods are popular and successfully applied in the relatively small, up to 2000 function point in size, projects. In larger software projects, the need for personnel and time raise and the work is more difficult to split in two-week long units. While Agile does have methods for scaling up to larger projects, there are also alternative approaches available.

The statistics by Capers Jones show that Scrum sessions are used in over 90% of Agile applications and up to 20% in non-Agile applications. The sessions take under 15 minutes per participant to prepare and optimally not much longer than the limit of 15 minutes per participant to execute. The defect removal efficiency ranges from under 35% to over 70% averaging 55% statistically. However, the statistical efficiency can be somewhat lower than the actual efficiency achieved . This is because the Agile and Scrum teams are usually not very strict on collecting the data for defect removal.

3.4 Testing

Testing is one of the oldest forms of software defect removal. It has been the most important category of defect removal since the beginning of the software industry, and in many cases even today, it is the only defect removal activity used. Several aspects of testing is covered widely in literature such as testing itself, test case design, test libraries and others. There are also a variety of standards and certifications offered by several companies and groups. Considering the penetration and importance of testing, there is surprisingly low amounts of quantitative data available on testing and test results. Quantitative data in this context means information about numbers of test cases used, numbers of defects found and other information that can be presented in numbers. In addition to the amount of data, the variety of business sizes is not as wide as it could be. The reason for this is that small companies rarely evaluate or benchmark let alone document the results with sufficient precision.

3.4.1 Subroutine Testing

Subroutine is a small piece of code that may have only a few lines of code. Testing subroutines is the lowest level of testing introduced by Capers Jones. It is a very informal way of testing and is performed almost spontaneously by compiling and executing a subroutine just created. The goal of testing the subroutines immediately after creating them is to verify the correct behavior of the algorithm before the integration of the algorithm to the larger module or application.

Subroutine testing is a glass box form of testing. It is used in almost every custom-coded software and over 90% of defect repairs. The defect removal efficiency is between 25% and 75% and in average 55%. Because subroutine testing is such a natural process and is such an efficient way to prevent defects, it is often omitted in testing literature.

3.4.2 Unit Testing

Unit testing is aimed at small code modules ranging from around 100 to 1000 source code statements. Units are tested by executing the new or repaired code. In case of developing new features, also the surrounding modules can be unit tested. The testing is usually run by the developer who wrote the module. This leads to poor data collection lowering the amount of data available for unit testing.

Unit testing contains often bad test cases which are either false positives or not finding defects. When using unit testing, a significant amount of bad fixes and new bugs are introduced while repairing defects.

The unit testing is often measured by code coverage, the degree of code a certain test suite covers. Aiming for high code coverage is usually a natural objective for test suites, but sometimes a high cyclomatic complexity of the module under test can prevent achieving high coverage. Modules with complexity under 10 can be tested thoroughly but when complexity raises over 20, the removal efficiency of the unit testing will decrease.

Unit tests can be executed manually but also automatically using a test runner connected to triggers actuating the testing sequence. The usage of automatized unit tests is becoming more common, while the popularity of Continuous Integration systems increase. These systems can be bound to version control systems allowing the automatic execution of tests whenever the source code changes.

Unit testing is considered as glass box testing. It is used in over 85% of projects using waterfall and in over 80% of defect repairs. Unit testing removes from under 25% to over 55% and in average cases around 35% of defects. Unit testing can benefit from the usage of static analysis, which is in most cases performed before the unit testing. In development of complex systems, unit testing can also benefit from code inspections.

3.4.3 New Function Testing

New function testing is a way of testing in where tests are written for evaluating the correct functionality of new features. These features can be introduced from modification or updating of an existing application. New function testing is often combined with regression testing.

In an entirely new software project, the new function testing is also known as "component testing". This is because usually the subject under test is a work of a group of developers, combining multiple code blocks into a one functioning component in a large system.

Because of the multiple contributors, the testing is frequently executed by separate testing specialists. Major new functions can exceed 10000 statements of source code, or 100 function points, when added to an existing system. Usually the new function testing is aided by a formal testing plan, planned test cases and a full configuration control. New function testing can be both black box and glass box testing. One of the main targets of new function testing are the errors in the interfaces between modules and in the movement of data through the application.

Like with many other testing method, a high complexity of the code can have a negative impact on new function testing. Both the defect removal efficiency and test coverage tend to decrease as the complexity raises. By using mathematical models for designing the test cases, the efficiency level of the testing can be improved without the need for infinite amount of test cases.

New function testing can take advantage of static analysis and formal code inspections. A usual flow with these three begins from the static analysis of the source code, followed by formal code inspections of the most critical parts and finally performing the new function testing. This combination can reach over 99% in defect removal efficiency, omitting the defects in requirements. Also using regression testing with new function testing can be beneficial to each other.

New function testing is used in over 99% of new software projects and also in over 99% of enhancements to legacy applications. The defect removal efficiency is in average 40% and ranging from under 30% to over 55%.

3.4.4 Regression Testing

Regression testing is a method of testing targeting the opposite of new function testing. In regression testing, the subjects under tests are old functionalities and features. The word "regression", in the context of software development, means an unintentional damage done to existing features while introducing new functionality. Regression testing also aims to make sure the known defects, repaired before the implementation of the new features, do not reappear.

Regression testing can be initiated during the development, when a sufficient amount of modules have been implemented. It continues through the whole development phase and further over to the post-release phase. Preventing the regression damage is very important in the systems already in maintenance phase.

Testing the regression damage is one of the most extensive forms of testing. This is because the evolution of a software application usually consists of multiple releases taking place over the years. With regression testing, the library of available tests continues to grow over the releases. These libraries involve the whole code base. In large systems the code base can even exceed a million lines of code.

Test libraries concerning a big amount of source code are at times problematic. They can have both useless test cases and test cases containing errors in themselves. Studies about these kind of libraries are rare, but an IBM study of a regression test library found both of the aforementioned. These erroneous test cases can raise the testing costs and lower the defect removal efficiency.

Regression testing is usually done in an application under full configuration control. It can be performed by programmers themselves, testing specialists or quality assurance personnel. Regression testing can be black box of glass box testing. Regression testing can benefit from the usage of static analysis tools on the legacy code under change before implementing the changes or refactoring.

High cyclomatic complexity can be harmful to regression testing. Cyclomatic complexity over 20 can lower the test coverage and defect removal efficiency.

Over 95% of new application development use regression testing. It is used also in over 97% of legacy application enhancements and in over 85% of software defect repairs. The defect removal efficiencies can vary from under 25% to over 45%. Average defect removal efficiency of regression testing is 45%.

3.4.5 Integration Testing

Integration testing is method normally used in relatively large applications having several modules connected to each other. Generally these large applications are over 1000 function points of size. As the name implies, integration testing is testing a number of modules assembled together to form a single software system.

The pace of integration testing is usually more or less steady as the integration tests usually target a single release or build. These builds can come in different cycles depending on the organization and the development practice. The interval between the builds can be for example a month or a week. As an example, Microsoft integrates the software projects in daily basis and thus performs daily integration tests.

Since the number of modules under test can be significant, the test suites can contain significant amounts of test cases. This leads to high amounts of work needed to design the test cases. However, using mathematical test case design methods can produce high test coverages and high defect removal efficiency with relatively small amount of test cases.

With integration testing, there are in most cases other supportive tools and practices used. Testing is usually done to an application under formal configuration control. Formal defect reporting procedures and test plans, planned test suites and test library support tools are also commonly used with integration testing.

Integration testing can occur as black box or glass box testing. The execution of the testing is the most effective when it is performed by professional testing personnel. Still, in addition to testing specialists, the testing can be performed by the programmers or quality assurance personnel.

Integration testing can benefit from static analysis, formal inspections and formal development practices. High cyclomatic complexity of over 20 can reduce the efficiency of integration testing.

Integration testing is rarely used in small, under 100 function point, projects. In these projects only under 10% of projects use integration testing. In medium size projects, over 1000 function points, integration testing is used in 85% of projects. Development of large, over 10000 function point systems use integration testing in over 99% of projects. The defect removal efficiency of integration testing can be from under 35% to over 65% averaging 50%.

3.4.6 System Testing

System testing is in most cases the last form of internal testing. After the system testing, the system is usually tested with real customers in field testing or beta testing.

Formal system testing for large systems is a critical testing stage and can require large teams of professional testing personnel and programmers involved in the development. This kind of large formal system testing can take several months, consisting of repairing the discovered defects and re-testing. The expression "system testing" dates back to large applications in the 10000 function point range. Since then, the term is widely used to describe the final stage of testing of applications any size.

System testing requires a formal configuration control of the application and usually a formal defect tracking is used. System testing is generally executed using the principles of black box testing, but sometimes there is also glass box type of testing. Testing can be performed by the developers, professional testing specialists or quality assurance personnel. However, when the testing is performed in large companies or for large systems, professional testing specialists performing the tests is the most common case.

If the system under test contains controls over physical devices, the term system testing can include simultaneous testing of the hardware devices. In these cases the group performing the tests can involve other engineering and quality assurance personnel dealing with the hardware.

System testing can become degraded if the system under test contains error-prone modules. These are modules, that encompasses the majority of the existing defects of the system. Error-prone modules can be extremely harmful for large systems and there is a need for continuously analyzing the existence of these. It is often necessary to precisely remove the error-prone modules and rewrite them with better methods.

IBM did a frequency distribution research in the 1970s for customer-reported defects. For several large applications, the distribution of defects was extremely uneven. With one product, there were 425 modules in the application. In this product, 57% of all reported defects were found in 35 of these modules. No reported defects at all were found for around 300 modules. All of the 35 error-prone modules had no inspections done and the testing was truncated due to schedule pressures. In addition to highly skewed distribution, bad-fix injections for these error-prone modules was higher than 35%, so fixing every third bug would generate one new bug.

Error-prone modules has been researched by other companies as well and these analyses confirm that these modules are alarmingly common in large applications. These modules have typically a high cyclomatic complexity and thus have reduced test coverage and defect removal efficiency. Repairing these existing modules is difficult and expensive, but preventing creation of new error-prone modules is possible with a combination of formal inspections with pretest static analysis. In the IBM study, inspections were 100% effective in preventing new error-prone modules.

System testing can benefit from static analysis and code inspections. IBM studies show that when error-prone modules were removed and inspections were used, the testing costs and schedules decreased by 65%. Also the customer satisfaction increased by 75% and development schedules were shortened by 15%.

System testing is used in over 99% of systems over 10000 function points. 75% of projects over 1000 function points and 50% of projects over 100 function points are using system testing. The defect removal efficiency ranges from under 25% to over 95% averaging 55%.

3.4.7 Agile Testing

Agile testing is a special form of testing and among the newest forms of testing. As Agile development contains embedded users, these users can define the requirements for a sprint or iteration. Embedded users can also define the proper test cases for an iteration. These test cases are primarily black box test cases, where the embedded user defines the functionality under test and the expected behavior.

In the development of larger applications that might have thousands of users, no single user can produce adequate information for gathering the requirements or specification of test cases. In the testing of normal Agile projects, under 1000 function points, the embedded users can provide effective assistance for the specification of test cases. Also the validation of test results can be usually aided by the embedded users.

In a certain type of Agile development, extreme programming, the test cases are developed prior to the implementation of the features. However, this procedure is not present in all Agile development.

Agile testing is used in over 90% of Agile application development. Defect removal efficiency is in range of under 40% to over 65% and in average 50%. Agile testing can benefit from static analysis and extreme programming testing.

3.5 Post Release

After a software product has been released to the market, it practically always still has defects in it. IBM calls these defects latent defects, because before the release, these defects have not yet been found as problems to customers. The existence of these defects is due to the imperfect effectiveness of the defect removal. Usually the defect removal efficiency is around 85% and virtually never reaches 100%.

3.5.1 Latent defects

Some latent defects can be defects found during the development or testing but ones that have not been repaired before the release of the software. Other defects were present in the application, but not discovered by the developers or test personnel. Furthermore, some defects can be originated from new development or other defect repairs in the form of bad fixes. The last two weeks before the release can bring in from about 1% to even 5% of delivered defects.

In a traditional development of commercial software, most of the latent defects found in after the release were those that had not been found and removed in the development and testing phase. In the more recent history, some vendors have started to release software with remarkable amounts of known, but not removed, defects. In small applications, below 1000 function points, there might be a handful of latent defects present. In larger systems, hundreds of latent defects can be released with the software. Moreover, in massive applications, like Windows 7 or SAP, the amount of known latent defects can sometimes be counted in thousands.

The motivation behind releasing a software with known latent defects appears to be compiled from three factors: first, the aspiration for achieving earlier release dates. Second, an assumption that a quick subsequent release will fix the defects. Third, the utilization of the skills of the customers for finding and repairing defects. The latest of the three can include a strategic offer for customers to get a compensation for repairing or identifying flaws.

This trend of releasing a software knowingly with defects has made customers skeptical about buying or installing the first release of a new software. Some customers prefer to wait for a second release, assuming the latter versions have many of the latent defects removed.

3.5.2 Defect severity levels

Because of the potential high amount of defects combined with limited amount of resources for removing them, some system for categorizing the defects on the basis of seriousness is needed. One of the oldest methods for assigning severity levels to defects is the IBM severity scale, which dates back to 1950s. It is still probably the most used severity scale.

The IBM severity scale contains four levels of severities and four other categories of defects. The defects in the first severity level cause that the software does not operate at all. Level 2 defects are disruptions or errors in major features. Level 3 contains minor disruptions, with which the software is still usable. Severity level 4 defects cause cosmetic errors that does not impact the operation of the software.

The other categories in the IBM severity scale consist of four levels existing for convenience. Invalid defect level contains problems that are caused by hardware or other software. Duplicate defect is a category for additional reports of a known defect. Abeyant defect contains defects that cannot be reproduced. Improvement category is for reported defects which are actually suggested improvements.

The usage of the severity scale is for arranging the defects to an order in which they are repaired. Defects in the higher levels are more important to customers than the low-severity defects. Because of this, the group responsible for removing post-release defects use more effort to the higher level defects. The defects in the highest levels may even require temporary fixes to allow the continued usage of the software.

3.5.3 Maintainability

The maintainability of software can be evaluated with several different metrics. According to Capers Jones, this is researched by IBM by interviewing a wide range of maintenance programmers about what makes the software maintainability better or worse. The three most useful metrics are introduced below.

Maintenance assignment scope is the amount of source code or functionality that a single programmer can handle for one year. The range of the maintenance assignment scope indicated by the interviews is from 300 function points to about 3000 function points. Measured in source code statements of java-like language, the range is from a low 15000 statements to a high 150000 code statements. These figures are assembled from the sizes of applications under maintenance and the sizes of the maintenance teams. These ranges seem to work well for maintenance programmers, but the same idea can be applied to other kinds of maintenance specialists.

Cyclomatic complexity can be measured automatically with both commercial and open source tools. For easier maintenance, the cyclomatic complexity levels should be under 10. With levels over 25, the maintenance is getting increasingly harder. Furthermore, higher levels than 25 are virtually never necessary, so high levels can indicate reckless programming practices.

Rate of structural decay is the velocity in which the small changes to the software raise the cyclomatic complexity levels and degrade the original software structure. The velocity can be lowered by running complexity analysis tools frequently. The average velocity of entropy or structural complexity seems to be a little more than 1% per year. This means that applications with long periods of use become increasingly harder to maintain or change safely.

The interviews by IBM revealed several topics that were mentioned by the dedicated maintenance personnel, which have a positive impact on the maintainability.

Training. Training of the maintenance personnel can improve the maintenance effort. In many cases, the maintenance personnel getting the maintenance responsibility are not properly trained for the application. For large applications, the training should include the general functionality and also the specifics of the components assigned to a single maintenance person. One cost-efficient way of training the maintenance personnel into the application is to include the personnel in inspections during the development.

Structural diagrams. Diagrams can support the understanding of the application structure in large systems. These diagrams can visualize the control flow, branches and other paths through the application. Structural diagrams can be created prior to the development of the system, which makes a requirement for keeping the diagrams up to date. If the diagrams do not exist, they can be generated using a variety of tools that analyzes and visualizes the code.

Comments clarity. The clarity of the comments can affect both positively or negatively on the maintainability. Good comments are clear, accurate and complete. Having too much comments can degrade the understanding of the code and can be almost as bad as too few. Comments should include the purpose of the module and explanation of the calls, branches and error messages.

No error-prone modules. Error-prone modules are one of the biggest topic degrading maintainability. These modules have usually high levels of cyclomatic complexity, are difficult to understand and can contain opaque dependencies to other modules. Perhaps the biggest problem with error-prone modules is that existing error-prone modules can be almost impossible to fix. Because of this, the main defense against error-prone modules is to prevent them from forming in the development phase by using static analysis and inspections.

Maintenance tools and workbenches. Software maintenance can be assisted by several kinds of maintenance tools. Actual maintenance workbenches can assist in analyzing code structure; Static analysis tools can analyze the code continuously;

Visualizing tools can automatically generate diagrams from the code structure; Testing tools can execute automated tests; and code refactoring tools can aid with fixing the code structure.

Programming languages. The choice of programming language can be made for business or technical reasons, but maintenance in mind, the readability and ease of understanding can affect positively. Some languages, like Assembly and API, are hard to understand and therefore hard to maintain. Other languages such as java-like languages are readable and straightforward and can be maintained more easily than the harder ones. With several thousand programming languages, sorting them into categories based on their easiness is not easy.

Considering these topics, the worst case scenario would include a long-time running software with high levels of cyclomatic complexity and several error-prone modules. There would be only messy comments or no comments at all and the software would be written in a hard or even dead language. The maintenance would be done by a novice untrained maintenance personnel with no aiding tools. In these circumstances, the maintenance assignment scope would be under 300 function points or under 15000 statements of code.

Mutually, the best case would have a fresh, low complexity software with well commented source code in easy language. In this case the maintenance personnel would consist of well trained maintenance professionals having good tools for maintaining the software. In this situation, the maintenance assignment scope could be over 3000 function points or over 150000 source code statements.

4. IMPROVING QUALITY IN SOFTWARE STARTUP PROJECT

This chapter describes the differences between quality improvement in traditional software development and modern software startup. Section 4.1 redefines the meaning of software quality in a startup environment. Section 4.2 describes the basis of quality. Section 4.3 introduces the methods for improving quality in a startup environment and combines the activities from traditional and modern software development. Section 4.4 introduces common things to avoid in quality improvement. Finally, section 4.5 describes the practices used in the company responsible for case project described in the next chapter.

4.1 Quality in a Startup project

Improving quality in a startup environment differs from the QA activities in traditional software development. This is because the requirements and goals of a startup project are usually considerably different from a traditional software projects. Furthermore, even the term software quality can be partially redefined for a better fit in a software startup. Eric Ries describes that the development of Minimum Viable Products questions the traditional notions of quality and this is the most vexing aspects of the development. [18]

In the beginning of the chapter about role of quality, Eric Ries states that "The best professionals and craftpersons alike aspire to build quality products; it is a point of pride". This is a good summary about the attitude towards software quality in many movements about modern software development. Measuring and defining quality in modern software projects with constant changes and uncertainty can be difficult, but the development personnel should have the pursuit for high quality. [18]

Ries claims that modern production processes seek to boost efficiency by relying to high quality. The belief that the customer is the most important part of the production process means that all effort should be focused to producing results that the customer finds valuable. This view can be beneficial in an environment where the company knows the opinions of customers. However, in a startup development, assuming the opinions of customers is a risky thing to do. Often in the startup, it is not even sure who the customer is. Thus, Ries introduces a quality principle for startups: "If we do not know who the customer is, we do not know what quality is". [18]

In a startup developing an MVP, the quality of the product can be a fluid concept. Even if the quality of MVP is low for customers, it can bring great value in building a high-quality product. If the customers find the product low on quality, this can be used as an opportunity to learn what customers care about. This is infinitely better than mere speculation, because it provides empirical information on which to build future products. [18]

When working with a 3D chat software called IMVU, Ries and his colleagues decided to leave a critical sophisticated feature done with only minimum effort. They were embarrassed to release the moving of the avatars without any animations or other modern visualizations. The avatars just reappeared to another location. The response of the customers was surprising as the customers were thrilled from the new feature, which allowed an immediate change of location without waiting. From the customers point of view, the released feature was more appropriate than the option which would take more time and money to implement. In the end, the quality of the released feature was probably higher than the one planned. The lesson behind the story is that customers do not care how much time something takes to build. [18]

Lean Startup method is aiming for the goal of winning over customers and not opposed to building high-quality products. Thus, it is necessary to set aside some professional standards to enable the validated learning as soon as possible. This is not supposed to allow operating in an undisciplined manner. This is important as there are some quality problems that can slow down the Build-Measure-Learn feedback loop. In addition, defects complicate the evolution of the product and interfere with the ability to learn. Helping the development of the MVP means removing any feature, process or effort that does not lead directly to the learning sought. [18]

4.2 Foundations of high quality

In a startup environment, it is particularly important that the quality of the software is sufficient to allow the further development. Compared to a company running a successful business already, a startup company usually cannot afford many mistakes in the product life cycle. Implementing high quality software starts from the beginning of the project. The foundations of high quality have to be solid for the overall high quality to emerge. These foundations are at the structure of the code and other deliverables.

4.2.1 Clean code

Every programmer with experience for more than two or three years have probably been slowed down by messy code. Over a couple of years of development, teams that were moving fast at the beginning of the project can be slowed down to a really slow pace. Development begins to include more and more non-trivial changes, which are about resolving the existing knots and tangles and adding new. Eventually, the mess can become so big that it can not be cleaned up at all. The total cost of owning a mess is being pondered by Robert C. Martin in his book Clean Code: A Handbook of Agile Software Craftmanship. [12]

Martin reminds that "code is really the language in which we ultimately express the requirements" and thus code is in the very foundations of every software project. A common mistake done in software projects is to increase the measure of developers in the team. This is usually done when the code has already reached a too messy state and the pace of development has slowed down. Adding new staff to the development does not improve the situation, but underlines the meaning of messy code. This new staff is not familiar with the existing system, so they easily complicate the system even further, driving the productivity even worse. [12]

A bitter fact for the developers is that usually the cause of messy code is the fault of the developers. While the management and sales staff are defending schedules and requirements, the developers should be defending the code. Many developers previously slowed down by messy code still feel the pressure of deadlines and the temptation to make messes to achieve them. Martin says that true professionals know that the only way to go fast is to keep the code as clean as possible at all times. A programmer able to write clean code is an artist, who can use all the little techniques in a disciplined manner through the sense of cleanliness. [12]

4.2.2 Integrity

One of the principles of Lean Software Development (LSD) is "Build Integrity In", which has its premises in high-performing automotive companies in 1980s. A study "Product Development Performance" by Kim Clark found out that the product integrity was a key difference between average and high-performing companies in developing superior products. Clark found out that integrity has two dimensions: external integrity and internal integrity. [4]

LSD renames these two types of integrities as perceived integrity and conceptual integrity. Perceived integrity is a balance of function, usability, reliability and economy observed by the customer. It is affected by the whole experience of the system beginning from advertising and delivery to intuitiveness and the ability to solve the problem. An analogy to the measure of perceived integrity comes from the question:

which ones of the bookmarks of your browser would you add back immediately, if you wiped out them all? These are the products with perceived integrity. [16]

Conceptual integrity is smoothness and uniformity of the whole system's central concepts. The architecture of the system has an effective balance between flexibility, maintainability, efficiency and responsiveness. Conceptual integrity is a prerequisite to perceived integrity. To be able to achieve perceived integrity, the system must have a consistent design. As the system evolves and matures, conceptual integrity emerges. Although conceptual integrity is needed for perceived integrity, the existence of the former does not ensure the existence of the latter. This is because conceptual integrity is not sufficient for a successful customer experience, if the system does not meet the users' need. [16]

4.3 Aiming at high quality

Achieving high quality requires discipline in both team work and individual development. A team of motivated experts with solid leadership is able to make the required decisions and keep the pace of development high. This kind of team can execute the development in a way that enables the emerging of integrity in the product.

4.3.1 Build Integrity In

In the study by Kim Clark, a primary claim is that integrity is achieved through superior information flow. Perceived integrity is a projection of the integrity of the information flow between users and developers. In turn, conceptual integrity is a projection of the integrity of the upstream and downstream of technical information flow. When reaching a system with high perceived and conceptual integrity, an excellent information flow is necessary both between customer and development team and between the upstream and downstream processes of the development team. Information flows must take into account both the current and potential uses of the system. [16]

Perceived integrity. In traditional software development, perceived integrity is transmitted to programmers through a multistage process. In this process, requirements are collected and processed through analysis and design. Then the design is transfered to the programmers implementing the code. These multiple hand-offs will cause the loss of considerable amount of information, key details and future perspectives. Fortunately, an alternative for this multistage process is to establish a superior customer-developer information flow by other means. With smaller systems, the development should be done by a single team having direct access to the people capable of judging the systems integrity. This team should use short iterations in where the system is demonstrated to a wide range of people giving feedback about the integrity. This way the feedback can be used for a rapid steering of course. Another technique is to use customer tests purposely created for getting feedback. [16]

Conceptual integrity. Conceptual integrity in automotive manufacturing is pursued by using existing parts and integrated problem solving. In integrated problem solving, understanding and solving the problem happen simultaneously, instead of sequentially. The initial information is released early and not only after complete information is available. Information flow happens in two directions instead of one and it contains multiple small batches of information instead of a single large batch. The preferred communication style is face to face instead of documents. This kind of problem-solving is ideal for achieving conceptual product integrity, particularly in development of complex systems such as automobiles or software systems. [16]

In software development, using integrated problem solving means that the development must be started before the design is finalized. In addition, the developers should be able to access customers or customer representatives for getting the answers as soon as possible for any questions raised. Customer tests should be developed and ran along the iterations and not just at the end. [16]

Finally, using experienced developers in their own areas of expertise can help in achieving conceptual integrity. Not all of the developers have to be with plenty of experience, but for the complex areas, experience brings understanding of the technical details and patterns widely used to manage with the complexities. One of these experienced developers should be a master developer facilitating the effort over multiple teams. For example, integrating the decisions and trade-offs among multiple developers and customers would be the responsibility of the master developer. [16]

Refactoring. When the integrity of a product is building, the iterative development brings continuous improvement as the product is evolving. In software development, the system must be continuously improved by the developers. Otherwise the internal structures of the software will become calcified and fragile. In time, the system will even stop working. Refactoring is done to prevent this disintegration of the software structures over the development. [16]

The need for refactoring comes when new features are added to the system one at a time and the architecture and requirements of the system change gradually. Often it would be better to think of new related features as a set and build an architectural capability to support them. If the features are added in multiple different locations of the code, the system will lose conceptual integrity. Refactoring regularly keeps the system healthy. [16]

Testing. In a traditional software development where implementation is done single module at a time, there are numerous types of testing used. There are for example unit testing, system testing, integration testing and acceptance testing used.

The purpose of testing is to check that the intention behind the design is achieved and the system does what the customers want it to. In LSD, the development is done by implementing entire features instead of single modules, so the distinction between the different kinds of testing is more artificial. Instead, testing is divided into two parts: developer tests and customer tests. Developer tests are testing that the implemented features work as intended and that all the pieces work together. Customer tests are developed to verify that the system does what customers expect. [16]

Tests act in several different roles in the development process. First, they present the exact functioning that the features are supposed to have. Second, they also supply information on whether the features actually work as supposed. Third, tests act as a scaffolding which provides support for tools such as refactoring. This allows the developers to make changes during the development. Fourth, after the development is done, the tests provide a representation of how the system was built. Finally, developing sufficient tests for all systems in production, making changes to these systems can be verified by running the tests for all related applications. [16]

Tests can also work as a tool for communication. Customer tests can be written within the conversation with the customer about details of desired functionality. This way the tests can give the most accurate description of how the features should work for a customer. Likewise, developers can document the planned design by writing tests testing exactly what the design should do. There are other tools allowing the communication before the code is written, but there is no substitute for testing whether the system does what it is supposed to do. With this in mind, using tests for both tasks can be beneficial. [16]

Receiving immediate feedback about the correct operation of code is important to a developer. The developer usually somehow tests the code as soon as its written, so the test could be captured as a written test allowing the reuse. Another opportunity to catch necessary experiments as tests is the review to customer within an iteration. Demonstrating the current system to a customer can be done with a set of demos or scripts, which can be captured as customer tests. These captured tests should be run as often as possible, so the team should ensure that the tests will not take too much time and are automated as much as possible. [16]

Using automated testing with extensive test suites for both developer testing and customer testing can be an efficient way to implement scaffolding to support the development. The scaffolding is a structure supporting the developers when they are making major changes to the system. While developing software in iterations, refactoring and other tools will include changes, which tend to have unintended effects. Having these effects in the system without the scaffolding provided by automatic tests can be an unsustainable situation. Often it may seem that writing tests slows down the development, but it pays back both during development and over the systems life cycle. [16]

4.3.2 Empower the Team

Lean thinking emphasized the importance of intelligent, disciplined and motivated frontline workers. In LSD, it is believed that the critical factor in motivation is empowerment. This means moving the decisions to the lowest possible level in the organization while assuring that those people have the ability and resources to decide wisely. [16]

Motivation. 3M is a company which is regularly meeting its goal that 30% of each divisions sales is generated by products introduced in the last 4 years. This stream of new products has been keeping the company continually renewed for over 75 years. In the core of the company is a formula that allows the entrepreneurial spirit prosper. This vision includes that the heart of the company consists of small, self-organizing groups who are passionate about a possibility and are allowed to realize the possibility. This kind of invention forms a company which evolves from the creativity of the individual employees rather than from the planning of managers. [16]

In the book Lean Software Development, roots of motivation are described as follows: "Insintric motivation comes from the work we do, from pride in workmanship and a sens of helping a customer." Intrinsic motivation is specifically strong in a team which has a commitment for reaching towards a purpose the team cares about. [16]

The sense of purpose can be achieved with several ways. In the beginning, there should be a clear and compelling purpose. This compelling purpose brings commitment and passion for bringing out the product. The purpose should be achievable. It should be made sure that the team has the capability and resources required for accomplishing the goal within itself. The team should be able to access the customers. Communicating with real customers can help understanding the purpose and also give insight into their individual work. [16]

The team should make its own commitments. The amount of work fitting into the iteration should be a call made by the team. Commitment in a team is a commitment to each other. Managements role is to remove interference. A leader does not need to tell a motivated team what to do, but listen for how the team could advise the leader. This helps the leader to remove obstacles and thus maintaining the momentum of the team. The team should not have skeptics around, because people who know plenty of reasons why things cannot be done, can kill the purpose really fast. [16]

There are four building blocks of motivation. First, a feeling of belonging can be achieved in a team where members are equal and respect each other. Having winners and losers created in the team can seriously harm the feeling of belonging. Second, feeling of safety is about the atmosphere of making mistakes. Tolerating no mistakes will kill all initiative as in creative processes tolerance of mistakes cannot be zero. Third, a sense of competence comes from success in challenges, positive feedback, knowledge and skill. In addition, a good leader must justify that the team is on a right track and having the resources needed to be successful. Finally, sense of progress will be maintained when the team has the feeling of accomplishment. Developing software in iterations is supporting this feeling as in every iteration the team gets to put its work in front of the customer for feedback. [16]

There are some downsides in passionate attitude towards a purpose. First, long working days and nights can be productive for a some time, but they are not a part of sustainable working culture. Second, different levels of passion are to be found inside teams. This means that there is a risk that the members of a team are expected to work long days. It is not fair for those not choosing them voluntarily and this can affect the spirit inside the team. [16]

Leadership. Innovative teams at 3M are led by a passionate leader called "product champion". This leader is usually the person behind the initial product concept. Getting support from the management and forming the team are also in most cases done by this leader. The leader is supposed to represent the customer by interpreting the product vision to the team. In Toyota, there is a similar position called "chief engineer". He has complete responsibility for the vehicle and the power to make all the decisions. It is common in these positions that the product is named after the bearer of this position. Another similarity between these companies is that these leaders do not have direct authority over the people working in the team. They are the leaders of their teams, not managers. In LSD, these respected leaders are called "master developers". [16]

The position of a master developer is not usually designated, but it is emerged. It is not necessary that there is a known master developer in the beginning of a project. A person with deep experience and understanding on both technical and customer issues usually rises to this position. Master developer is part of the team working towards the purpose and empowering the team. Architects and other advisory roles are not likely in a role of master developer, because they are not a part of the team. [16]

Expertise. In software development, there are two major areas of expertise: technical knowledge and domain knowledge. Technical knowledge creates experts in areas such as database administration, user interface design and embedded code. Domain knowledge includes understanding of the domain of the business, for example health-care or security. [16]

Expertise in different areas evolves best in communities of expertise, where people responsible for same kind of activities meet. A traditional method of developing these communities inside a company is to divide the company into functions by core competence. These functions can include program leaders guiding the teams in product development. This structure is called a matrix organization. [16]

In a successful matrix organization the two managers each person has, view their jobs correctly. Functional managers should act as mentors and teachers. They should supply the inexperienced members of the team with support and coaching through progressive series of assignments. Value adding managers should view their jobs as enablers and motivators who remove impediments and supply resources to the team. [16]

Companies not using matrix organization should also preserve communities of expertise. These communities can be gathered by identifying the areas of competence in both technical and domain knowledge. The members in formed communities should be available for communication and information sharing using repetitive meetings or other media reaching everyone in the community. If the amount of experts in a critical area is not sufficient for forming internal community, external communities are usually available for support in the expertise. [16]

4.3.3 Activities

The Lean Software Development focuses on building integrity in and doing the development in short iterations with continuous validation. These guidelines try to assure high quality more as preventing the defects from incurring than finding and repairing the defects after their occurrence. Traditional quality assurance does not fit to startup environment as is, but the principles behind the activities can be applied.

Defect prevention. Traditional defect prevention activities can be applied in Lean Software Development. Agile approach and embedded users can support the iterative development and observing of the customers. Test Driven Development can help in achieving integrity, using tools such as refactoring and testing as a whole. Static analysis can prevent defects in the code and it should be used in modern software development where ever possible.

Pretest. Activities used in pretest defect removal were studied and presented by Capers Jones in The Economics of Software Quality. The study was done to small software projects, which were defined as an application below 1000 function points or 50000 statements in a Java-like language. This applications were developed by a team of under six developers. The results of the study divided the defects found by the origin of defect and the activity it was found with. Defect origins distinguished were requirements, design, code and documentation. Used activities were personal

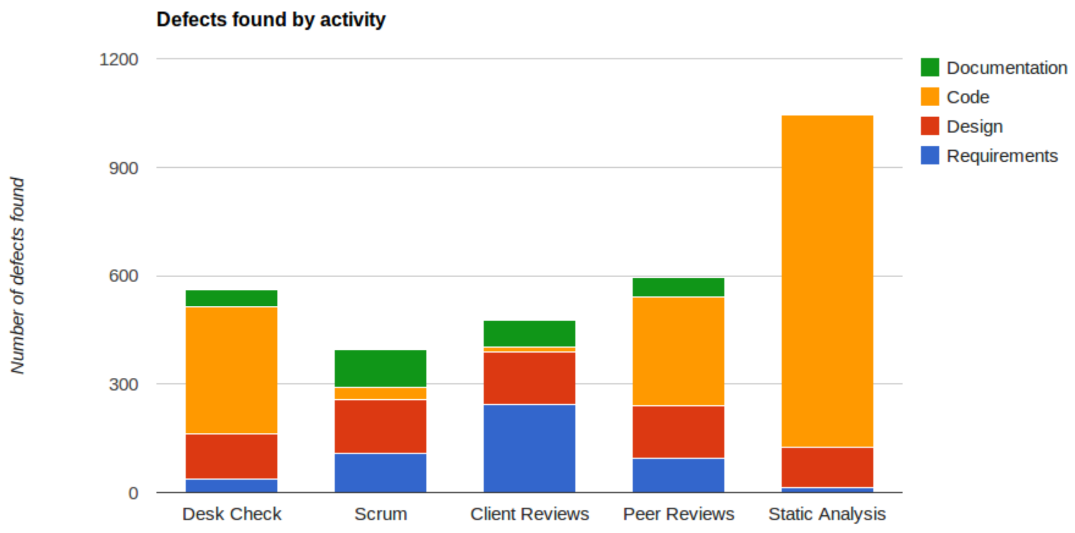

Figure 4.1: Efficiency of pretest defect prevention

desk checking, Scrum sessions, client reviews of specifications, informal peer reviews and static analysis of source code. The results show that over half of the defects were found from the code. The most effective activity for finding the defects prior to testing was static analysis. Considering the almost automatic nature of static analysis, it can be seen as a predictable result. The results presented in Figure 4.1 can aid in choosing the pretest defect removal activities.[11]

Using these pretest defect removal methods can be beneficial also when doing LSD. Personal Desk Checking can help when trying to write clean code and when reaching for conceptual integrity. Informal Peer Reviews can be a powerful tool supporting things such as clean code, conceptual integrity and refactoring. Also empowering the team can be done with peer reviews, because the reviews can include elements raising motivation, strengthening leadership and improving expertise. Client reviews of specifications can support in achieving perceived integrity and getting overall understanding of the customer's needs. Scrum sessions are a natural part of iterative development.

Testing. Using traditional testing methods in startup environment is a topic which has to be planned per project. As LSD divides testing in developer and customer tests, each project has to decide which of the traditional methods of testing have the best result when applied. Plain unit testing and subroutine testing can be relevant in some situations, but the use of excessive unit testing in pursuit for full test coverage can be waste of time and resources. In startup environment, the development is usually done as a cross-application development covering full features. In this kind of development the testing should be focused on testing the application as a whole. Though, testing single parts of the application can be useful and even necessary depending on the software.

4.4 Things To Avoid

When aiming for satisfactory quality, one common mistake is to put the effort available on wrong target. Lean methodology appoints this as waste and seeks to eliminate it. Mary and Tom Poppendieck have interpreted waste in LSD as anything that does not add value or "anything that interferes with giving customers what they value at the time". Some main forms of waste highlighted in their book are partially done work, "churn" and extra features. Partially done work can get obsolete, hide quality problems and ties up money. Churn means excessive amounts of unproductive work. Churn can include requirements churn, churn in testing long after coding or churn created by delayed integration. These software development churns are usually associated to large inventories of partially done work. Finally, extra features are a major factor in increasing the cost of development. In the book there is an estimate that two-thirds of all features and functions of typical software are only rarely used. These features just increase the complexity of the code. [15]

A typical activity in quality assurance is the use of defect tracking system. Found defects are reported in this system, where they are stored until a developer takes them under work. The use of these trackers assumes that the time of discovering the defect is prior to the repair of the defect. Tom And Mary Poppendieck mention that Shigeo Shingo has divided inspections in two: inspections after defects occur and inspections to prevent defects. Reaching for quality means controlling the conditions to not allow the emerging of defects. Thus the usage of defect tracking systems can be seen as a source of waste consisting of partially done work. [15]

Lisa Crispin considers the topic of using defect tracking systems in agile development in her article based on her presentation in STAREAST conference. She introduces a well-tried practice for handling with defects: immediately after a bug is identified, an automated test is written, the bug is fixed, and both the fix and the test are pushed to repository. With this approach, the tests are instantly written so that they catch the defects if they reappear later and the tests also document the bug. Also when this "fix and forget" approach is used, there should be no need for tracking the defects. Crispin also reminds that using a defect tracking system is not a good way to communicate the defects and it can even get in the way of direct communication. Defects recorded in the tracking system can often suffer from poor and inadequate descriptions. Also the tracking system can be polluted with invalid defects which are never going to be repaired because of their low priorities or their location inside a module waiting for rewrite. [5]

4.5 Software Quality in Futurice

Futurice is a lean service creation company founded in Finland. There are currently around 200 employees in four offices located in three countries. The employees of the company are passionate multi-talents in areas from technology, business consultancy and service design. [7]

Futurice does not have strict processes or common practices in quality assurance. The main idea present in all activities in the company is that smart people make smart decisions. With software quality, this means that the developers and other members of the project teams are able to decide their own approach to software quality. This freedom of choice leads to experimentation of multiple activities and practices, from which the good and bad experiences can be shared with others. Some practices of course become more common than others, but the use of those is not forced in any way. A common way of executing projects is to use methodologies from Lean and Agile software development.

In the core of Futurice, is communication and transparency. All the decisions and experiences are available for everyone and this brings the possibility to make decisions to every employee in the company. There are also informal communities built for areas of interests, such as mobile development and web development. These communities have recurring events where experiences are shared.

For the longer-term projects there is a separate team named Life Cycle Management team. Members from this team are present in a project from the beginning and they live through the product's life cycle participating in the development and maintenance of the product. This ensures that there is a solid amount of experience present in the team from the beginning of the project to the post-release phases.

5. CASE: PÄIKKY

This chapter presents phases from a project done before the writing of this thesis. The execution of the project was done with no intention of it being a part of a thesis work. All the data shown here are gathered afterwards and thus do not meet the requirements of a proper scientific study. The project is presented here as an example project of a small budget startup. Section 5.1 describes the objectives and starting point of the project. Section 5.2 introduces the execution of the phases in detail. Section 5.3 describes the methods used for improving quality in the project. Section 5.4 evaluates the quality achieved in the project based on the interviews of the project team and customer.

5.1 Project objectives

The project was built on an idea from MukavaIT. The big picture and the goals of the project were specified in advance and introduced to the developing team. Lower level specifications and implementation details were designed in cooperation between MukavaIT and the development team [13].

The objective of the project was to create a new kind of recording system for kindergartens. The system would allow the tracking of the arrival, departure and absences of the children in the groups. The main goal was to make a user-friendly system which connects parents, employees of the kindergartens and the administrative personnel in the municipality. The connection between the stakeholders would enable the planning of the need for day care and the real-time tracking of the nurses and children. Eventual goal of the system would be dynamic hour-based billing models. In addition to this, the system would produce several types of reports for administrative personnel for optimizing the daycare system.

The most important functionality of the system would be that nurses are able to record the presences of the children with mobile devices. The records will be collected to a database serving different services for nurses, parents and administrative personnel in the kindergartens. The services would include a mobile client for nurses, a desktop client for parents and another desktop client for administrative personnel. A common goal in which these clients are aimed at, is to bring the daycare system towards a genuinely transparent process, which encourages the parents to participate more in the early childhood education.

During the development, a better understanding of the market was achieved. It was learned that reaching towards the hour-based billing models and the report generation was not as necessary as it first seemed. The stakeholders became interested already as the first version of the system was released into pilot use. The features that allowed the parents to communicate with the nurses were something the parents were excited of. In addition, the recording and checking the presences, communicating with the parents and storing the information about the children all in the same system were inspirational for the nurses.

This understanding of the market was not present when deciding the priorities of the upcoming features, so some of the effort put to reporting and other features could have been allocated to more appealing features.

5.2 Project Execution

The project was divided into multiple phases. In the scope of this thesis, five phases are observed. Each of the phases had predefined objectives and a time frame. The project was executed using agile practices, so the phases were divided in sprints lasting usually a week. A weekly session, with both the development team and the customers present, was arranged in where the previous sprint was reviewed and the next sprint was planned. The contents of the sprints and the backlog were managed with Pivotal Tracker, a tool for agile project management.

The development team consisted of a few developers, a user experience and design specialist and a project manager. The size and structure of the development team stayed rather consistent, but the individuals belonging to the team were changed several times during the project. Also the pace of the development varied somewhat between the phases.

In the description of each of the phases, the amount of functionality is approximated by using a conversion from the amount of source code lines to function points [8]. The user interface is implemented with web technologies, so the amount of Javascript code is used. For the backend, the implementation language is Groovy, which is assumed to be close to Java in function points.

Phase one. The feature content of the first phase included the implementation of the mobile client and administration interface. The mobile client was for recording the presences of the children and the nurses. The administration interface included the principal editing of kindergartens, care groups, children and nurses.

The phase lasted 87 calendar days and the team in the first phase consisted of three developers, a UX specialist and a project manager. Budget of the phase was 26% of the total budget of the five stages presented here. The phase included 101 new features registered in the Pivotal Tracker. 17 bugs were found during this phase. New source code were introduced worth of 130 function points.

In the first phase, the project quality was affected by the introduction of the continuous integration environment and a user interface testing framework. The continuous integration was handled with a product called Bamboo. It was configured to track the changes in the repositories of both the mobile client and the desktop client. Whenever a change was pushed into the repository, the continuous integration system would run the tests and update the testing environment. In addition, Bamboo could update the production server by running the job manually. Testing of the mobile client was started with a small set of tests running with CasperJS.

Phases two and three. These two phases shared some goals so they are merged in the project records. During these phases, the development targets were in the communication, planning and administration. Communication between the kindergarten and children's home was implemented to the mobile client and the desktop client for the parents. For the parents, a tool for planning the future presences of the children was implemented. These presences were intended to help the personnel of the kindergarten plan the future shifts of the nurses. The last big feature for these phases was a desktop client for the administrative personnel of the municipality. This manager client for the kindergarten management included features for browsing and editing the kindergarten information, including the information about the children and the nurses. In addition, this manager client could generate reports describing the recent history of the kindergarten.

Phases two and three took 105 calendar days in total. At the beginning of the phase two, the team consisted of two developers, a UX specialist and a project manager. After a one third of the period, a third developer was included in the team. These two phases took approximately 56% of the total budget. According to the Pivotal Tracker, 69 new features were implemented and 2 bugs were fixed. New source code was introduced worth of 285 function points, totaling 415 function points for the project so far.

During these phases, the quality of the project shifted in different directions. A staging server environment identical to the production server was created. The staging server was integrated with the continuous integration server. The staging server had two purposes: the customer could demonstrate the application to potential customers and some system testing could be done in the staging server.

In addition to the introduction of the staging server, the quality of the project was affected by problems with testing. Testing the user interfaces of the application became an issue, which eventually lead to total disabling of the user interface tests. The issues were related to gradually degrading tests, which in the end impeded the usage of the continuous integration environment.

Phase four. In the fourth phase the main goal was to implement the support for the features and APIs for family daycare. These included several exceptions to the ordinary kindergarten operation, for example calculating the compensation of expenses for the care provider.

The time frame for the fourth phase was 40 calendar days. The team included two full-time developers, one part-time developer, a UX specialist available when needed and a project manager. The budget of the fourth phase took only 6% of the total budget. The history data from Pivotal Tracker reveals 8 features and 3 bugs for this phase. The amount of source code raised by 50 function points to a total of 475 function points.

No major changes to the quality aspects of the project in the fourth phase. User interface tests were kept disabled and testing of the new features was handled manually.

Phase five. In the fifth phase, the main focus was in refactoring some critical parts of the application. The most important targets for this was the implementation of presence markings and handling of time zones. In addition, some refactoring was done to others parts and some new features were implemented.

The fifth phase lasted 32 calendar days. The budget for this phase was around 12% of the total budget. Three developers, an on-demand UX specialist and a part-time project manager aggregated the team for this phase. Pivotal Tracker contains 5 features and one bug allocated for this phase. The veracity of these numbers can be questioned and it is possible that the usage of the tracker was neglected with the work done in refactoring the existing implementations. A total of 25 function points were implemented as new source code and after this phase the total function points reached an even 500 function points.

Some major refactoring tasks of the application were done in this phase, so the structure of the application was improved. Thus the overall quality of the application should have became higher.

5.3 Quality Assurance in the Project

Quality assurance in the Päikky project was not formally specified or planned beforehand. Methods and processes were selected primarily by individual developers and talked through with the whole team. Methods and tools were taken to use when necessary.

Agile development. The project was executed with agile approach. The development was done in a week long sprints, which included a weekly review and planning session. The customer representatives were present in the weekly meetings. This enabled that the implementation of the features in the finished sprint could be assessed against the understanding of the customer.

Static analysis. The development of the application was mostly done by using some integrated development environment (IDE), although some of the developers involved in the development used only a source code editor without any intelligent features. These modern IDEs provide automatic tools for running continuous static analysis of source code. Using this kind of static analysis tool can prevent some defects from getting into the application.

Peer reviews. With some parts of the application, ad hoc informal peer review sessions were used. These reviews were done in situations where the developer responsible for the implementation of a component had questions about the implementation details. In these brief discussions, parts of the code and thoughts about the implementation were reviewed by another developer. The end result of these session would usually be either an actual decision about the implementation or a joint decision that these details must be discussed with the whole development team or with the customer.

Testing. During the project, testing was one of the most problematic forms of quality assurance. In the beginning of the project, all the testing was done manually by using the system via the user interface. The effort for implementing the automatic tests were postponed in favor of the development of new features. Some critical features in the backend were tested with integration tests and a set of user interface tests were implemented during the first two phases of the project. However, in later phases the user interface tests were disabled because of the problems with the test runner. Alternative options for test runner were investigated, but no good enough solution was found with the effort put into the studies. Also the amount of integration tests did not increase at the same pace as the number of critical features. The attitude towards testing was not neglecting, but the amount of new features to implement did depress the allocation of effort on the testing.

5.4 Achieved Quality in The Project

The quality achieved in the project was evaluated by interpreting the views and opinions of the product stakeholders. A separate informal interview sessions were arranged with the development team, project manager and the customer representatives. In addition, notes from a retrospective session after the fifth phase were considered.

5.4.1 Development Team

The overall feeling of the development team was that there should have been more effort put into the quality assurance during the project. The actual development suffered from the lack of effort put into the quality aspects. The shortage of testing and other methods of quality assurance seemed to affect negatively more and more as the project advanced.

Testing. Testing was one of the most important themes brought up in the interviews with the development team. In the beginning of the project, testing was taken into consideration and some effort was put into bringing testing tools in use. Testing tools for testing both the backend and frontend were studied and selected. Some tests for testing the most critical features in the backend were written through the whole project. For the frontend, only a few tests were written and the execution of those tests were unstable from the day one. The final setback for the frontend tests was got when the chosen test runner began to freeze during the execution of the tests.

Biggest problems with testing was that there were no common practices or clear allocation of effort to the writing of the test cases. According to the understanding of the development team, there were defects present that could have been prevented with proper tests. The general feeling was that putting more effort to testing would have helped save time and money.

Proposals for improving testing suggest that in the first place, the infrastructure for the testing should be repaired. The test runners should execute the tests reliably and the tests should be run in the CI environment. In addition to fixing the tools, the team and the customer should agree on a common practice for systematically testing both frontend and the backend of the system.

Effort distribution. Another wider issue mentioned by the development team was the distribution of effort between the quality assurance, bug fixes and new feature development. From the developers point of view, new features were prioritized too high and the velocity was kept too fast. In this constant hurry, some of the implementation details had to include shortcuts. These shortcuts led easily to taking technical debt and even to defects.

Another problematic topic with effort distribution was the lack of separation between further development of features and the maintenance of the system in production. Most of the time, at least in later phases, the discovery of a defect could lead to an interruption for one or more of the developers. This affected the performance of the developers, because the simultaneous work on the new features and the running system meant that the developers faced multiple context switches during the ordinary work.

Opinions for improving the situation included using more effort to quality related activities, including planning, documenting, implementing and testing the features. Allocating more effort to these topics would have first led into a slower pace of delivery with new features. In the long run though, it could have helped keep up the average delivery times of new features. Moreover, with this approach the structural quality could have been better. With better structural quality, the amount of defects and the work required to repair defects could have been lower.

The maintenance of the running system could have also been done better. Using dedicated personnel for the defect repairs would have disengaged the developers implementing the new features, so the focus could have been kept in the implementation. Also a more flexible prioritization of the defect repairs may have had allowed more efficiency in working while still keeping the defect repair schedules sufficient.

Shortcuts. Taking shortcuts and making compromises in planning and implementing the features was seen somewhat necessary in this kind of project. There was a constant uncertainty of the requirements and expectations of the end users, which caused a need for experimenting with the features.

A problem with these shortcuts was that the causes and existence of these were easily forgotten. As a consequence, the estimates of the tasks were easily skewed and the expectations from the features often lacked the facts caused from these compromises. In most cases there would have been a clear demand on refactoring the implementation, but frequently the refactoring was forgotten or left out because of the need for quick delivery of new features.

There were no clear opinions on how to improve this, but there seemed to be a shared concern on the lack of refactoring the structure. The general feeling was that using more time on planning and implementing the features would have prevented many problems from occurring. Also the pace of implementing the features seemed to slow down in time, which was seen as an outcome of the poor structure. Proper restructuring could have solved this slowdown.

Common practices. There were virtually no forced processes in the development used. The development team found that this partially led to careless behavior in the project. In particular, mutually agreed ground rules about coding practices were wanted. Ideally these rules would act as guidelines for the development and not as a bureaucratic burden.

These practices could include things like a proper definition of done, guidelines for code style and other practices now having differentiated styles among the developers. In addition to helping keep the code uniform, these guidelines would help new and inexperienced developers to become familiar of the practices used by others.

Also one practice in particular was mentioned several times: informal reviews or inspections of each others code. There was some concern about giving negative feedback to other developers. As an improvement to this, there was a suggestion that a mutual agreement would be required that every member of the team would agree to receive both positive and negative feedback, without finding it offensive. Reviews could be done with pull requests or other lightweight tool for automatic reviews.

Communication. In the beginning of the project, good communication helped to achieve good quality of the specifications and delivered features. As the project progressed, communication between the development team and customer was gradually degraded. Later on, communication inside the team was also weakened because a few of the developers were part-time employees with separate working days.

There were no actual proposals for improving the communication between the team and customer. Also no solutions were presented for solving the communicational challenges when working with part-time employees.

5.4.2 Customer and End Users

The main tone in the interview with the customer representative was that the realized quality in the project was close to what the customer had expected. Although there were no clear problems with the quality mentioned, the need for improving the application quality in the future was acknowledged.

Overall quality. The customer was generally quite satisfied with the achieved quality in the phases one to five. It was evident that some problems were present in the project, but the approach had been in emphasizing the effort put to the delivery of new features. The customers viewpoint was that implementing and delivering new features would open opportunities for obtaining new end customers for their product. Using less effort on testing and other quality assurance, the team would produce new features more quickly and that in turn would eventually lead to more revenue.

The success of using this approach was proven by the feedback from the end users: the end users expressed interest in the system and were excited of the possibilities it could provide. The end users were satisfied with the system regardless of the quality issues which reached the end users. There was an assumption that, for the end users, the main interests regarding quality would be simplicity, logic and usability of the user interfaces. This seemed to be correct.

Causes of problems. It was seen that the fast pace of development and the low focus on quality aspects in the earlier phases were boomeranging to the later phases. Many of the large problems being fixed in the later phases could be traced back to the shortcuts done before. These were partially caused by the loose specification of the features and the regression formed by the new feature implementation. Some gaps in the specifications were partially consciously taken risks, as all the time spent specifying features and having meetings cost money.

Some reference was given towards the changes in the team personnel. Some defects caused by regression could have been avoided if the developers implementing new features would have been familiar with the implementation of the existing features. The customers view on this lack of sufficient specification was that a constant communication about these ambiguities should have corrected most of the issues with understanding.

Some mistakes were made during the project, which could have been done differently. In the manager client meant for the kindergarten management, not enough attention was put into the things the end users cared the most: simplicity, logic and usability. The manager client was implemented quickly without a proper knowledge of the actual usage it was going to.

Another clear mistake was the effort put to generating automatic reports for the kindergarten personnel. After the feature was delivered, the end users expressed that they preferred to export the data to an external spreadsheet software. The effort put to this feature could have been saved and put into more efficient use. Eventually these generated reports are going to be implemented more properly, so the effort was not entirely wasted.

Practices. Some practices the customer was satisfied with were in the area of communication. The weekly meetings were mentioned as an important part of the development. Most importantly, in these meetings the whole team should agree on what tasks are to be done in the following sprint. Also the tasks would be discussed through to assure that everyone in the team would understand the principles of the features.

In addition to the weekly meetings, communication through the digital channels were mentioned as a good practice. However, the customers view was that there could have been even more communication. The customers thoughts on this were that the development team probably would not want any more interruptions on their daily work, so the situation could have been in balance after all.

Areas in need for improvement. There were three topics the customer mentions that probably had room for improvement. First, more testing should have been executed using the data from production environment. Some testing had successive results when executed in either development or testing environment, but still some issues were found in production environment. The data from the production should have been duplicated to the environments the test were executed in.

Second, the migrations to the database should have been tested more thoroughly. There were some issues caused by the multiple migrations done to the production database. The customers opinion was that these issues could have been avoided by using the actual production data for testing the migration.

Third, the customer thinks there could have been more testing in cooperation with the development team. By testing together with the customer, the development team could have acquired deeper understanding on the requirements of the customer and end users.

5.4.3 Discussion

The opinions of the developers differed clearly from the views of the customer and end users. It appears that the attitude towards software quality differs between these groups of stakeholders. Achieving a good balance between assuring good quality and implementing new features can be crucial in building a new product. The customer was confident that in this project, the balance was quite good.

In spite of having many negative observations about the quality of the project, the success of the application so far has been good. The customer has gotten several new clients with pleased end users, which matters more than the perceived quality in the project. As there was no single big issue impairing the quality of the project, fixing multiple issues would have taken significant effort. With this effort, the pace of delivering new features could have been compromised. Using more effort on testing could have prevented some defects, but as some of the defects were caused by incomplete specifications, there would have been defects present after proper testing. In any case, some of the problems with quality were caused by the lack of time of the customer, so perfect quality could not be reached.

An estimate about the sufficient quality achieved in the project was that having a team of developers with high motivation and a desire of being proud of their own work will produce decent quality for the resources in use.

6. CONCLUSIONS

In this thesis, quality improvement in a software startup environment was studied. Startup environment was examined from the viewpoint of Lean Startup methodology. Software quality was described both in traditional sense and in a software startup context. Thesis defined and evaluated software improvement methods and activities that were then linked into the startup environment and its view of software quality.

A case project was also presented by describing its execution and phases in detail. Also the quality improvement methods used in the project were described. Quality achieved in the project was assessed with the data from interviews with both the development team and customers. These opinions were summarized for a view to overall quality. Some suggestions were also made for improving the quality and the development processes in the future.

Improving quality in a software startup development emphasizes the importance of defect prevention over defect removal. This is common with traditional software development. In traditional software development though, the effort put to defect removal activities and testing has larger share of total effort than in startup environment. Another difference between these two environments is that in modern startup environment, the development is usually done in short iterations. The patterns in traditional development are not applicable in this kind of development as such, but the activities can be applied by adjusting them for use in short iterations. These patterns also usually require too much effort for a startup environment.

When traditional quality improvement has more focus on the technical aspects, methodologies of modern software development quality focus on the people and processes of development. Both the team and individuals are considered important and the emphasis is on the motivation, expertise and leadership of the personnel. The development should include plenty of communication and the team should have all the resources and authority to make all the important decisions. The technical aspects to consider in software startups are the integrity of the product and the iterative development containing constant improvement.

In the future, topics regarding the human aspects of the development could be further researched. Building a motivating environment which embraces expertise and leadership seem to improve the efficiency of the team and eventually raise the quality of the software developed. In the further work the effect of these human factors could be examined in real projects. This examination could include good practices used in successful projects and pitfalls found in lower quality projects.

In addition, individual activities used in modern software development could be inspected. Peer reviews and other means improving communication are widely used. In an experienced team with skilled professionals, communication acts an important part in achieving high quality. In addition, consistently more automatic tools aiding the development are used, which should make reaching high quality easier. The usage of these activities and processes in a successful startup environment could be an interesting topic for a research.

REFERENCES

[1] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[2] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. [WWW]. [accessed at .1].2014. Available at: http://www.agilemanifesto.org/, 2001.

[3] Steve Blank. What's a startup? First principles. [WWW]. [accessed at 26.12.2013]. Available at: http://steveblank.com/2010/01/25/whats-a-startup-first-principles/.

[4] K.B. Clark and T. Fujimoto. *Product Development Performance: Strategy, Organization, and Management in the World Auto Industry.* Harvard Business School Press, 1991.

[5] Lisa Crispin. Stareast: Agile testing and defect tracking. [WWW]. [accessed at 29.3.2014]. Available at: http://searchsoftwarequality.techtarget.com/tip/STAREAST-Agile-testing-and-defect-tracking.

[6] P.B. Crosby. *Quality is free: the art of making quality certain.* Mentor executive library. New American Library, 1980.

[7] Futurice. Futurice - Fact sheet. [WWW]. [accessed at 28.3.2014]. Available at: http://www.futurice.com/wp-content/uploads/2014/02/Fact-Sheet-English2.pdf.

[8] Quantitative Software Management Inc. Function point languages table. [WWW]. [accessed at 12.3.2014]. Available at: http://www.qsm.com/resources/function-point-languages-table.

[9] ISO 25010. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.* ISO, Geneva, Switzerland, 2011.

[10] ISO 9126. *Software engineering – Product quality – Part 1: Quality model.* ISO, Geneva, Switzerland, 2001.

[11] C. Jones, O. Bonsignour, and J. Subramanyam. *The Economics of Software Quality*. Addison-Wesley, 2011.

[12] R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2008.

[13] mukavaIT Oy. mukavait - mukavaa päivää. [WWW]. [accessed at 20.5.2016]. Available at: http://www.mukavait.fi/.

[14] Campion College O'Neil Hibbert. Desk check guide.

[15] M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley Signature Series (Beck). Pearson Education, 2006.

[16] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[17] Roelof Reitsma. Software has a new quality standard: ISO 25010. [WWW]. [accessed at 27.11.2013]. Available at: http://ryreitsma.blogspot.fi/2011/07/software-has-new-quality-model-iso.html.

[18] E. Ries. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. Crown Business, 2011.

[19] American society for quality. Quality glossary. [WWW]. [accessed at 23].11.2013. Available at: http://asq.org/glossary/q.html.