jamk.fi

Agile Testing: Improving the Process
Case Descom

Mikael Huisko
Mikko Kyyrö

Bachelor's Thesis
September 2015

Business Information Systems
School of Business and Services Management

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s)	Type of publication	Date
Huisko, Mikael Kyyrö, Mikko	Bachelor's thesis	20.9.2015
		Language of publication: Enlish
	Number of pages 83 + 34	Permission for web publication: x

Title of publication
Agile Testing: Improving the Process

Degree programme
Business Information Systems

Tutor(s)
Kiviaho, Niko

Assigned by
JAMK University of Applied Sciences, School of Business and Services Management

Abstract

The thesis was assigned by Descom, a marketing and technology company based in Jyväskylä. The aim of the thesis was to research the current state of testing inside the organization, and to improve on the existing processes and practices. The thesis was carried out as a design research (applied action research), because the focus was improving already existing processes inside a company.

The theory base contains a wide range of subjects from agile development models, the testing process, and process improvement models to agile testing. Without a solid base of multiple aspects it would have been impossible to understand how the testing works as a process and how it could have been improved. As Descom uses agile development it was necessary to follow the same principles throughout the writing of the thesis and on results.

As a result information was provided for the company about the current state of testing procedures at Descom and how to improve the testing and processes in the future. The documentation already existing for testing such as the test plan and test report were updated. New documents such as a process improvement plan based on Critical Testing Processes, test strategy and testing policy were also created. Figures of the testing process, and the processes for all test types in use were created to be used as a visual aid for understanding the testing as whole at Descom.

Keywords/tags
Agile testing, Testing process, Scrum, Extreme Programming, Kanban, Process improvement, Critical Testing Processes

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Mikael Huisko Mikko Kyyrö	Opinnäytetyö	20.9.2015
	Sivumäärä 83 + 34	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Ketterän testauksen prosessien kehittäminen

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Työn ohjaaja(t)
Niko Kiviaho

Toimeksiantaja(t)
Jyväskylän ammattikorkeakoulu, Tietojenkäsittelyn koulutusohjelma

Tiivistelmä

Opinnäytetyön toimeksianto tuli Descomilta, joka on Jyväskylästä lähtöisin oleva markkinointi ja teknologia yritys. Työn tavoitteena oli tutkia testauksen tilaa organisaatiossa ja kehittää olemassa olevia prosesseja ja käytäntöjä. Tutkimusmenetelmäksi valikoitui kehittämistutkimus, koska painotus oli olemassa olevien prosessien kehityksessä yrityksen sisällä.

Teoriapohjassa käsiteltiin monia aiheita ketterästä sovelluskehityksestä, testausprosessista ja prosessi kehityksestä aina ketterään testaukseen asti. Ilman kattavaa pohjaa monille osa-alueille, olisi ollut mahdotonta ymmärtää miten testaus toimii prosessina ja miten sitä pystyy kehittämään. Descom toimii ketterän sovelluskehityksen mukaisesti projekteissaan, joten oli tärkeää seurata samoja ketteriä periaatteita läpi opinnäytetyön kirjoittamisen ja tuloksissa.

Tuloksena saatiin tietoa yritykselle, siitä miten testaus on toiminut Descomilla ja kuinka testausta ja prosesseja tulisi kehittää tulevaisuudessa. Myös aiemmin olemassa olleet testausdokumentit päivitettiin. Uusina dokumentteina laadittiin suunnitelma prosessikehitykseen, joka perustui Critical Testing Processes –malliin, testausstrategia ja testauspolitiikka. Prosessikuvaus tehtiin kaavioita käyttäen, joilla kuvattiin prosessi kokonaisuutena sekä käytettävät testaustasot.

Avainsanat
Ketterä testaus, testausprosessi, Scrum, Extreme Programming, Kanban, prosessikehitys, Critical Testing Processes

Muut tiedot

Table of Contents

Tables

Figures

Acronyms

N4S	Need for Speed
RAD	Rapid Application Development
XP	Extreme Programming
TDD	Test Driven Development
TMMi	Testing Maturity Model Integration
TPI	Test Process Integration
CTP	Critical Testing Processes
STEP	Systematic Test and Evaluation Process
ISTQB	International Software Testing Qualifications Board
CI	Continuous Integration
UAT	User Acceptance Testing
SQL	Structured Query Language
GUI	Graphical User Interface
RAM	Random-Access Memory
ROM	Read Only Memory
UI	User Interface
ROI	Return on Investment

1 Introduction

Software industry is a business changing constantly and a feature that might be required today could very well be something different tomorrow. The old waterfall development models were heavyweight and their development phases set in stone. Testing was usually done right in the end of the development and when the deadlines started banging on the door, the time was often cut from testing, which resulted in release of faulty software and displeased customers. In response to the problems caused by the old waterfall models, new agile and lean development models were created.

These agile development models strived for efficient and effective development by communicating with the customer frequently and making it easy to change something if the customer so decided. Agile helps in managing the changing requirements; however it does not itself solve the question, whether the new features are the desired ones. This is where software testing comes to play, a vital part of software development of which importance is ever growing. Software testing means the tasks and actions that help in ensuring the quality, validity and verification of software all the way from fundamentals of testing process to ever so topical agile testing (Crispin & Gregory 2009, 4-5).

The focus of this thesis was to provide information for the company of software testing process in agile environment and how to improve it using iterative methods and testing improvement models. The thesis contains a theory base for agile development, testing process, process improvement and agile testing which can be used for learning fundamentals of testing and agile development.

The theory was utilized to produce a testing process templates, testing strategy, testing policy and templates for Descom without forgetting the flexibility of agile development, meaning that the products of the thesis can scale effortlessly in different development projects.

2 Background for the Thesis

The thesis topic was assigned by Descom, a company working with the Need for Speed research program to improve their current state of testing. Testing has been performed in some way in most projects, however it was not necessarily carried out in agile ways, which the organization uses to run their projects. A lot of the testing was done manually in contrast with agile cornerstone, automation. The company has been growing very rapidly and their business has multiple aspects leading to a situation where different projects do not co-operate with each other as well as they could.

2.1 Objectives and Research questions

The research focused on studying on how it is possible to improve the testing process and if it could be standardized somehow for every project in the organization. The background information of Descom's testing process was acquired by interviewing the testing staff and surveys. The guideline for the testing process was to be defined by inspecting current processes and the staff's own opinions of what processes are or would be good. The requirements set by different development projects were also emphasized in the process creation. The theory was gathered to give validation for the information gained during the research.

The thesis aims to answer the following questions:

- How can the testing process and quality be improved?
- What kind of investment does Descom have to make in order to keep improving testing processes?

The thesis provides information on how company can improve their testing processes and find a flexible solution. The thesis also includes a theoretical background of testing, however, the main result of the study is applicable only for Descom. This thesis can also work as an info package for new testers at Descom.

Chapter 2 covers the agile development models that Descom is using in their projects. Along with Scrum and Kanban, RAD (Rapid application development) was selected because it is the model on which agile development is based. The fourth model XP (Extreme programming) was selected for its very complete practices and popularity.

Chapter 3 covers the agile testing process and the theory is written utilizing the old heavy waterfall model which consists of a much heavier testing process. The authors think it is important to go through everything concerning the process phases, so it can be decided which of them work in an agile environment and which are to be dropped out or modified.

Chapter 4 goes through different methodologies to improve testing, from which one was chosen for the testing improvement process at Descom.

Chapter 5 discusses how testing should work in agile development, agile testing policy and strategy.

2.2 Methodology

The thesis methodology was a design research (applied action research). In design research there is always a phenomenon, process or current situation in the background that needs to be improved by changes or improvements. There is usually a problem that requires solving but if the problem has multiple aspects, it can itself be a target for research (Kananen 2012, 13). Design research is not a research method of its own but a mixture of different research methods which are used depending on situation or target for improvement. Methods connect both qualitative and quantitative research methods. These researches are called “blended” or “mixed methodologies”. The research has always a theory background which is taken into practice (Kananen 2012, 19). Typical targets for design research are processes, products, services or state of affairs. All of the mentioned subjects are developed continuously in working life. What makes it a research is that improvement is documented and scientific methods are used which provide

reliable and validated information. One criteria for science is to produce new information. In addition to research there are also means to create change in the target environment. Goal is to influence the target by intervention. This may cause troubles of its own if the intervention is not a part of research targets normal behavior (Kananen 2012, 20-21).

2.3 Descom and N4S

“Small enough to care, big enough to carry through” (Descom, 2015).

Descom is a marketing and technology company and their main business income is focused around commerce. They focus heavily on offering tailored solutions for IBM platforms, however, also other technologies are used. Descom was founded in 1997 as a summer job project for four students, and the company has been steadily growing ever since. Descom emphasizes customer experience and the company is able to provide a competitive advantage for their customers by expertise in technology and marketing. (Descom, 2015)

Need for speed (N4S) is a co-operation consortium which is created to build a foundation for success for Finnish software companies in new the digital economy. The program takes on real time business models and brings them into practice. The models are based on deep expertise which enables instant value production. (N4S-program 2014.)

3 Agile Development Models

In 1980 the software industry started questioning the heavyweight old school development models when most software projects failed to deliver on time and budget, and customers were often dissatisfied with the results. New lightweight, lean and agile methods started appearing, striving for efficient and effective development by increasing communication, adopting smaller and manageable iterations, and being flexible and responsive to the customers' requirements and changing demands. (Watkins 2009, Chapter 3.1)

3.1 The Agile Manifesto

In 2001 a group of developers who practiced different agile development methods got together and wrote the manifesto to set the core values for agile development. (Highsmith 2001)

> ***Manifesto for Agile Software Development***
>
> *We are uncovering better ways of developing*
> *software by doing it and helping others do it.*
> *Through this work we have come to value:*
>
> ***Individuals and interactions*** *over processes and tools*
> ***Working software*** *over comprehensive documentation*
> ***Customer collaboration*** *over contract negotiation*
> ***Responding to change*** *over following a plan*
>
> *That is, while there is value in the items on*
> *the right, we value the items on the left more.*
> *(Manifesto for Agile Software Development, 2001)*

According to Shore & Warden (2008, 9), there is no such thing as an agile method. They understand agile as a way of thinking, a philosophy. To be truly agile you have to take action to implement agile values in to practice. Projects are different and situations unique. Therefore it is best to use an agile approach that is customized for the current situation (Shore & Warden 2008, 11).

3.2 Rapid Application Development

James Martin started to develop the rapid application development model (RAD) in the 1980s after the industry was growing dissatisfied of the old models such as the waterfall. Martin spent a decade refining the process and finally in 1990 released his thoughts of RAD in a book form. The main idea of RAD is to break down the heavyweight approach of waterfall into smaller manageable iterative steps and to increase the customers' involvement in the development. Watkins (2009, Chapter 3.2) brings up the importance of prototyping in RAD and defines this as one of the key aspects of the process model.

Watkins (2009, Chapter 3.2) explains the three key goals of RAD as such:

-High-quality systems

-Fast development and delivery

-Low costs

According to Novák (2011, Chapter 1) RAD is not a clear particular development methodology. He explains that it is a generic name for different methods that rely on iterations and prototypes rather than traditional methods. Models such as Scrum and Extreme Programming (XP) are based on RAD.

3.3 Scrum

"One of the fundamental principles of Scrum is "the art of the possible". That is, Scrum instructs teams not to dwell on what can't be done, but to think about what can be done." (Schwaber & Beedle 2001, 27)

According to Schwaber & Beedle (2001, 89) Scrum is founded on completely different foundations and way of thinking than other methodologies in systems development. Scrum is based on an empirical systems control model, meaning that timetables and assumptions of difficulty are based on experience and assumptions, and they will fluctuate with the project. Scrum is not a development model but rather a project management model, and it is often combined with other development practices such as Extreme Programming (XP). See Figure 1 Development process in Scrum (Mountain Goat Software 2005) for an illustrated guide of how the development process works in Scrum.

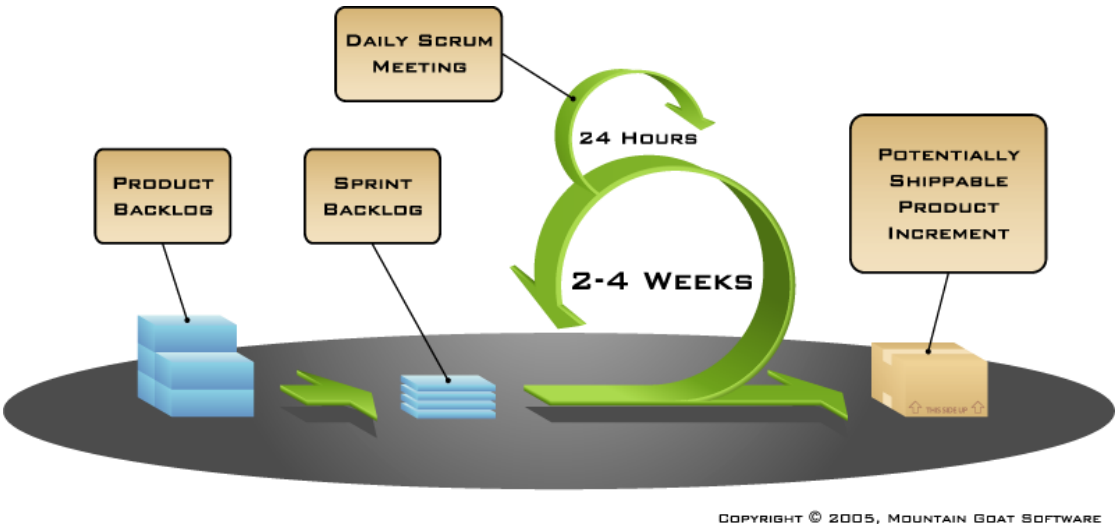

Figure 1 Development process in Scrum (Mountain Goat Software 2005)

Schwaber & Beedle (2001, 147-154) define the fundamental values of Scrum as follows:

Commitment

People need to be willing to commit to the project, and Scrum encourages this by giving the team autonomy and authority to do the sprint as they see fit.

Focus

Only the Scrum Master is allowed to communicate with the team during the sprint, so it is possible to focus on the work without distractions. A clear set of tasks will make focusing on reaching the goal better.

Openness

Everything about the process needs to be open and visible to everyone on the project or company. Anyone can come and listen to Daily Scrums and partake in review meetings. The Product Backlog also needs to be visible to everyone.

Respect

There are no individuals on a team and all its members should be treated with equal respect. All members have their own strengths and weaknesses, unique backgrounds and skills. Prejudice, resentment, quarrels and other negative attitudes do not have a place in a scrum team.

Courage

Team members need to have courage to work well with complete autonomy. They need to be able to trust their own judgement and not be afraid to ask for help and training if the need arises.

Scrum practices and roles

Scrum introduces new names and job descriptions for some of the traditional roles; the three new roles are Product Owner, Scrum Team and Scrum Master. The most important documents are Product Backlog and Sprint

Backlog. The three most important meetings are the sprint planning meeting, the Daily Scrum meeting and the sprint review. (Stober & Hansmann 2010, Chapter 3.4)

Product Owner

The Product Owner is officially responsible for the project. This one person is always the representative of stakeholders such as the customer or marketing, and decides the requirements and funds the project. (Stober & Hansmann 2010, Chapter 3.4) Product Owner is the only person who can maintain the Product Backlog. Everyone can add items to it, but he is the only one who can move them and assign their priorities. (Schwaber & Beedle 2001, 34)

Scrum Master

Scrum Master is usually the project leader or project manager under a different name. He guides the team to work following the values and principles of Scrum, helps to gather needed resources by removing impediments, by making decisions, and works as a representative to the management. No one else is allowed to disturb or give new directions to the team members in the middle of a sprint. All information from the management to the team must go through the Scrum Master. (Schwaber & Beedle 2001, 31-32)

Scrum Team

Scrum Teams are small teams that do all the development set for the sprint. Many different teams can work in parallel using the same Product Backlog. Scrum Teams are self-organizing and completely autonomous. (Schwaber & Beedle 2001, 9) They have complete authority to decide how they will tackle the items on the Product Backlog that they have selected for that specific sprint. A good number of members for a Scrum Team is around seven people, plus or minus two people. Teams with over eight people do not usually work efficiently. A Scrum Team should have the members needed to finish the sprint, including developers, testers and specialists. People in Scrum need to be able to work cross functionally, while everyone has their own set of skills,

sometimes they might need to help another team member and step out of their comfort zone. The team formations can change in the beginning of a sprint if the team does not work well together or if some specialists are needed to reach that sprint's goal. (Schwaber & Beedle 2001, 35-37)

Sprint

Sprint, also known as iteration, usually last around 30 days. The goal for every sprint is to produce a new executable product functionality. Architecture and design are added incrementally and modified within several sprints, rather than doing it all in the first sprint. There are as many sprints as needed for developing a finished product, taking into consideration the set requirements for cost, time, functionalities and quality. No one is allowed to disturb the team during the sprint, to change its objective, to add more functionalities or technologies, or determine how the team will do their job. The team can spend their sprint time as they wish, they can work when and how they want, and they can have meetings when they want without time limits. (Schwaber & Beedle 2001, 50-52)

Abnormal termination of a sprint can occur in case the goal of the sprint becomes obsolete. The company can change their mind or direction, market’s conditions shift, or technologies change so that the project is not relevant anymore. The Scrum Team also has a right to terminate the sprint if they feel that the sprint goal is not achievable, the goal has been reached already or if someone outside the team tries to change the goal and workload of the sprint. (Schwaber & Beedle 2001, 53)

Daily Scrum

Daily Scrum is a short 15-minute meeting in the beginning of the day led by the Scrum Master. Daily Scrum asks each of the team members three questions. What have they done after the last Daily Scrum? What will they do today? What impedes on doing that task? Daily Scrum is not a design session and no problems are solved at it. It is simply a status check so that everyone knows what they are and should be doing, and the Scrum Master finds out if

they need to clear out some of the impediments. Only the Scrum Master and the Scrum Team are allowed to speak during Daily Scrum. (Schwaber & Beedle 2001, 40)

Follow-up meeting

If a topic comes up that needs more discussion when answering the three questions of the Daily Scrum, a follow-up meeting can be arranged right after it. It is important to keep Daily Scrum and other working sessions separate, so a clear distinction of both should be made. Everyone who is interested in the topic can join this meeting and pitch in, whether it be by discussing the design or requirements further, or sharing relevant information about the subject. There is no time limit of how long a follow-up meeting should last and how deep into the topic it should go. (Schwaber & Beedle 2001, 46)

Sprint planning meeting

There are actually two parts to the sprint planning meeting and their main goal is to figure out what functionality to build on the next sprint and how. In the first part the Scrum Master, the team, Product Owner, customers, users and the management plan together the next sprint's goal. The Product Backlog has a large importance in a planning meeting, as the functionalities can be modified, deleted or added and their importance changed. Scrum Teams can also be changed during this time if needed, however, not in the middle of a sprint. (Schwaber & Beedle 2001, 48)

The second part of the planning meeting is held by the Scrum Team and often also the Product Owner will attend. The goal of the meeting is to figure out what needs to be done for the team to reach the set goal. Other people can be invited to this part of the meeting to provide technical or domain advice. The team will create the Sprint Backlog as the result of the meeting. The Sprint Backlog will have all the tasks needed to build the Product Backlog into a working software. The tasks are detailed and a time estimate of four to sixteen hours are assigned to them. The team member who tackles this task needs to keep updating the hours left on the task while doing it, and when it finally

reaches 0 the task should be done. The team members will decide what tasks they want to do and in which order. (Schwaber & Beedle 2001, 49)

Sprint review meeting

The sprint review meeting is held in the last day of each sprint. The review meeting is led by the Scrum Master, who also coordinates it and decides with the team who presents and what. The meeting should start with the Scrum Master giving an overview of the sprint, following by the team members' presentations. The meeting is very informal and no concrete presentations such as Power Point slides are displayed. The meeting is a working meeting and as such questions, observations, discussions and suggestions are allowed and encouraged. (Schwaber & Beedle 2001, 55-56) All the relevant and interested people are free to join the meeting. The meeting goes through the product increment, how the sprint was executed, if there were any problems during the last sprint, if the sprint reached its goal, whether there were contradictions in the requirements, how well the product and the code have been tested, how stable the release is and how well the team worked together. (Schwaber & Beedle 2001, 70) It is decided whether the team will continue to build on top of this increment, scavenge what can be reused, or if the whole increment should be scrapped. Usually the first solution is chosen, however, even in the worst case scenario the team has lost only a month of development time.

Product Backlog

As defined by Schwaber & Beedle (2001, 72), the Product Backlog consists of product features, functionality, infrastructure, architecture, and technology work. Anyone can add content to the Product Backlog, whether these are needed features or just ideas that someone feels would be good for the software. Everything needed in the project needs to be found in the Product Backlog. The Product Backlog is never done, it keeps changing and living throughout the lifecycle of the project. The items on the backlog are assigned priority numbers; the higher the number, the more important and desirable the

feature. Only the Project Owner can change the order and the priorities of the items on the list. The Product Backlog adapts at the end of each sprint when the new product increment has been released and which items on the list are be relevant for the next sprint. (Schwaber & Beedle 2001, 33-34)

Sprint Backlog

The sprint backlog has all the tasks needed to complete a sprint. The team maintains the list themselves, adding new tasks if new work is required, removing non relevant tasks and marking finished tasks as done. The tasks are detailed and a time estimate of four to sixteen hours are assigned to them. The team member who tackles this task needs to keep updating the hours left on the task while doing it, and when it finally reaches 0 the task should be ready. If the members do not have time to carry out all of the tasks during the sprint a new meeting takes place with the Scrum Master and Product Owner, where it will be decided if some of the tasks can be dropped or the amount of requirements decreased so that the goal of the sprint could still be reached. (Schwaber & Beedle 2001, 49-50)

Release backlog

The Release Backlog is a subset of the Product Backlog. The list consists of all the work needed for the product, however, the user stories are segmented into a probable releases and they are given release time estimates. As the sprints go further the Product Owner keeps empirically adjusting this list and the release estimates on it. (Schwaber & Beedle 2001, 71-72)

3.4 Extreme Programming

“Of all agile methods I know, XP is the most complete” (Shore & Warden 2008, 12).

Extreme programming (XP) has become an increasingly growing agile practice and multiple books have been written on the matter (Stephens & Rosenberg 2003, Chapter 1). According to Stephens & Rosenberg (2003) extreme programming emphasizes on four key values:

Communication

As Shore & Warden (2008, 18) have stated, XP gives a high value on face to face communication which is an effective way of eliminating delays in communication and misunderstandings inside the team. XP believes that the verbal communication is the most effective way (Stephens & Rosenberg 2003, Chapter 1).

Simplicity

Simplicity is the way to take when designing a software in extreme programming environment. The easiest way to achieve a certain goal takes usually least time and is therefore, most effective; however, sometimes the absolute simplest solution is not always the best. For example, sometimes a backup or monitoring systems for the software are required when simplicity would mean that only the simple version without backups is to be created. In some cases the non-functional requirements affect the design a great deal and the simple design might eventually turn out to be a very complex solution. Despite that, simplicity is a keyword to keep in mind. (Stephens & Rosenberg 2003, Chapter 1)

Feedback

Despite the fact that XP does not itself guarantee higher productivity, the process provides frequent feedback for the development team which enables easier refining and changing the plans (Shore & Warden 2008, 18). Also the

customer feedback is taken into account in several cases, which will be further explained in section Planning.

Courage

The developers should not be afraid to make changes in their project. Eventually making the right product may often mean radical changes along the way, however, they just need to be made. Fear is an enemy of extreme programming and it limits agility. (Stephens & Rosenberg 2003, Chapter 4)

How does it work?

One of perks of XP is eliminating the requirement, planning and testing phases and included documentation. Truth be told, software development need many requirements, which is why XP goes through these things every single day (Shore & Warden 2008, 18). An XP team works simultaneously on all the aspects presented in Figure 2. An approach such as this is very far away from traditional software development.

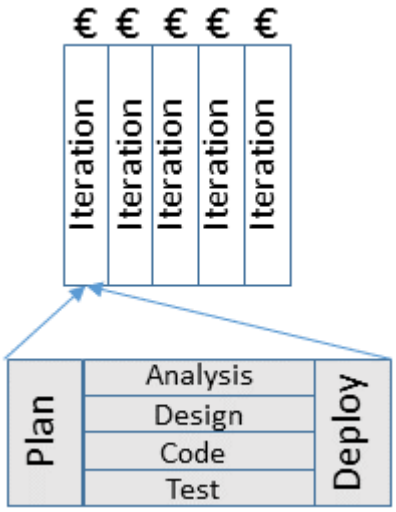

Figure 2 XP lifecycle (Shore & Warden 2008, 18)

Planning

XP team contains several business experts i.e., on-site customers whose task it is to make the business decisions. Their job is to guide a project to a correct

direction by visualizing the project, risk management, creating stories and making a release plan. Programmers offer estimate and suggestions which are mixed up with the customer expectations. This is a process called “the planning game”. (Shore & Warden 2008, 19)

Analysis

A separate analysis phase is not required in XP because the on-site customers sit with the development team full-time. The on-site customers may be actual customers or some other employees who possess most knowledge of how the software should work. On-site customers are also in charge of the requirements. To be able to set requirements the customers use their experience and traditional requirements-gathering techniques. They have to figure out the requirements early for the stories so that when developers inquire about details, they are ready to answer about these immediately. Some requirements, however, are difficult to understand or complex in other ways. To manage these situations, on-site customers create “customer tests” with the help of testers. Customer tests are examples which ensure that developers understand well enough that certain requirements are binding detail. (Shore & Warden 2008, 19)

Designing and programming

XP uses an incremental way of working, which means that the design is being refined and improved little by little. This way is being led using test-driven development (TDD), which binds together programming, design, architecture and testing. To support TDD, XP implements pair-programming. This enables to have more strength on every task and one of the pairs can usually think about the design in the larger scale. Developers are responsible for the maintenance of their own development environment. By using version control they are basically able to constantly produce builds that are ready for deployment. (Shore & Warden 2008, 20)

Testing

XP includes a wide range of different testing practices. Each team member puts his best effort into improving quality, and well-functioning teams produce only few bugs in they completed work monthly. The first shield for bug-free software is test-driven development. With TDD the team produces automated unit tests and integration tests. Also the customers' tests enhance the quality of the product. They review work made by developers and make sure that the outcome matches with the customer expectations. They provide examples to be automated by developers and the examples contain usually tricky business rules. Testers help the team to understand whether the produced code is in fact of high quality.

Exploratory testing is being used to find gaps and to point out unexpected behavior. Upon a bug discovery, tester performs a root-cause analysis which is used to improve process and hopefully prevent similar bug occurrence in the future. Testers perform also non-functional testing, such as performance testing. The team does not perform any manual regression testing. TDD is being used to create a comprehensive regression set. When a bug occurs, a new set is created which is used to ensure that the bug has been fixed. Regression sets are run after every integration to verify that nothing has been broken. (Shore & Warden 2008, 20)

Deployment

Development team prepares a complete deployment during every iteration, however, the actual deployment for real customers is scheduled after business needs. Weekly demos, however, are deployed to internal stakeholders. Depending on the organization, the maintenance is taken care of by a development team or a separate support team may take over. If maintenance is shifted for a different team, the development team creates documentation of the system and provides training for the new team. (Shore & Warden 2008, 20-21)

3.5 Kanban

Kanban was originally created in Japan during the Toyota Production Systems formation. Kanban is a method, based on visual cards, where it is determined that a new work item should be accepted in a system only after there are sufficient resources available to handle the task (see Figure 3). Unlike in traditional agile and especially in Scrum method, Kanban does not require implementing new roles such as Scrum master or product owner. Simplicity is the beauty and strength of Kanban, because the organization can maintain the existing roles and continue development even if the process is founded on waterfall. Kanban experts would not get worried when facing a waterfall project because the base of Kanban is the optimize concepts and techniques despite what development model is being used. (Pham & Pham 2013, Chapter 2)

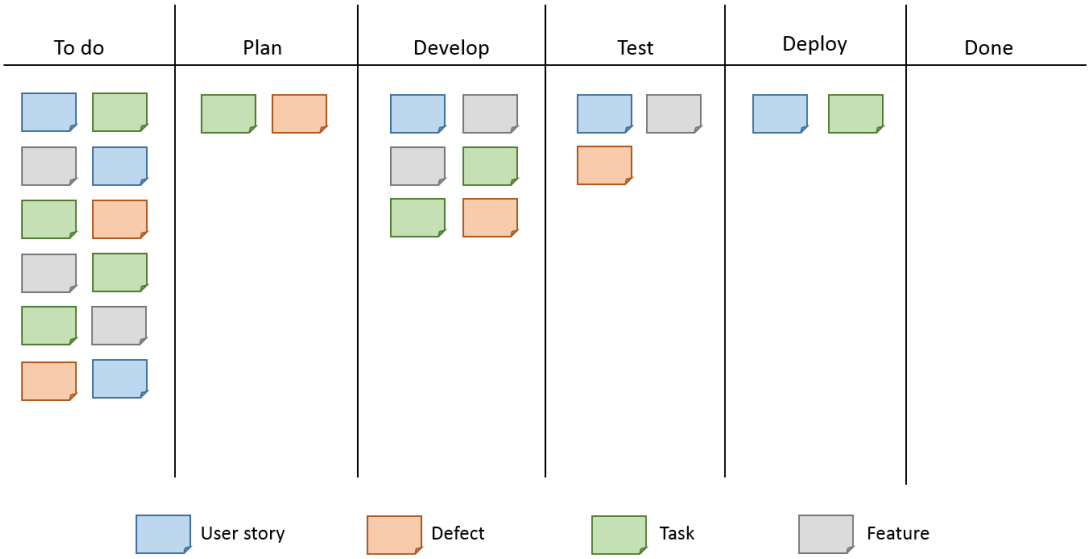

Figure 3 Kanban board (Leankit n.d.)

Despite the Kanban practitioners having different approaches to work flow, the base is the same as Pham & Pham (2013) have stated:

Visualize the workflow

Without clear understanding of the current process and work flow, all discussing remains on the level of speculation. This is why visualizing the work flow is the first important step to take as early as possible.

Capture metrics and rules

Without clear understanding of how the current process is being performed, all the effort for improvement is dangerous. Which is why it is important to capture rules of what has been done.

Identify bottlenecks

Interviewing the development team is an excellent way to identify risks for example finding out that the development team cannot perform reviews on time.

Establish a new service level agreement (SLA) and policy

The purpose behind SLA and policy is to identify the expectation of customers and users and what both parties should do to achieve the level of performance required for trusting relationship.

Limit work in progress (WIP)

As soon as SLA and mutual expectations have been reached, it is time to review the work flow of development team and find out how to determine limits for WIP. The idea behind WIP is to set limit for requests that can be accepted in the development teams' system without overflowing it.

Measure lead times and other metrics

Like in Scrum, Kanban includes the daily stand ups, however, unlike in Scrum, Kanban focuses on the work flow instead of the goals. Before each daily meeting the current progress should be updated to show the latest results of the development team.

Optimize

A step on process is a waste of time if it does not contribute to the agreed goal. Steps that do not add value for the product should be combined with the other steps in order to avoid unproductive work.

4 Testing Process

"A process is a series of activities performed to fulfill a purpose and produce a tangible output based on a given input." (Hass 2008, Chapter 2)

Testing process is defined to help testers achieve better results and be more efficient; documents of best techniques, guidelines and templates for different levels of testing will save time and money when testers can lean on predefined documents, terms and ideas, rather than having to learn and figure them out every time in a new project. (Watkins 2009, Chapter 2)

4.1 Planning and Control

In the planning phase the customers, stakeholders and the project's objectives we need to be understood with, the risks that testing will try to minimize. Based on this knowledge the goals and objectives will be formed for the testing itself, the approach on how to test will be chosen, the plan for tests will be formed as well as the specification of test activities. The test policy, master test plan and test strategy can be used to help with writing the level test plan. The plan made must adhere to the conditions set in the policy and strategy, and if deviated from them, it is necessary to consult the stakeholders beforehand and agree upon the changes. (Graham, Veenendaal, Evans & Black 2008, Chapter 1.4)

The planning will produce a tangible test plan, which should be modified to suit the organization and the project. This level test plan will be used as a reference to all the test processes that will be made in the following phases. As listed by Hass (2008, Chapter 2.2), a planning document should have the structure similar of following:

-Introduction (scope, risks, and objectives)

-Test item(s) (test object(s))

-Features to be tested

-Features not to be tested

-Approach (targets, techniques, templates)

-Item pass/fail criteria (exit criteria including coverage criteria)

-Suspension criteria and resumption requirements

-Test deliverables (work products)

-Testing tasks (analysis, design, implementation, execution, evaluation, reporting, and closure; all broken down into more detailed activities in an appropriate work break down structure)

-Environmental needs

-Responsibilities

-Staffing and training needs

-Schedule

-Risks and contingencies

Test planning and control are ongoing activities, and control is always carried out side by side with planning. In control the actual progress to the test plan is compared and the results are reported to the project manager, the customers and the project team. Control activities are needed if the plan is not followed. The results of control activities include: documenting all the changes and deviations from the actual plan, measurements and analysis about reviews and results of tests, passed and failed tests including the number, type and priority. It is important to keep all the members of project team informed on how much testing has been done, what the results have been and what risk assessment has been done. All the results of testing need to be visible and useful for the whole team. From the gained information corrective actions will be initialized, which include tightening the exit criteria, adding effort into debugging, prioritizing defects and even possibly changing the test plan. The

larger decisions about continuing or stopping testing, releasing the software or postponing it are based on the collected measurements and information. (Graham et al. 2008, Chapter 1.4)

4.2 Analysis and Design

Analysis and design is a part of a process where the details of what is going to be tested and how test conditions are associated with test cases are discussed so that as small as possible amount of test cases provide as much as possible coverage for the test conditions. This phase stands between the planning and test execution. It relies strongly on planning which means schedules, human resources and what will be tested. It also relies on test execution which means expected results and what environments or platforms are required. The purpose of test design is to predict how the software will perform under certain conditions. Sometimes test results are negligible but if the results are not estimated, there is a high risk to overlook some crucial aspect of the software. (Hambling, Morgan, Samaroo, Thompson & Williams 2010, Chapter 1)

Graham et al. (2008, Chapter 1.4) have divided test analysis and design to five tasks:

Review the test basis, which is being used to support building tests. The test basis makes it possible to start building certain tests already before the actual code exists because the documentation provides some understanding of what the system should do. Reviewing the test basis is also a tool for finding gaps and illegibility from specifications when trying to understand clearly what the different points of a system should do.

Identify test conditions. This helps building a high-level understanding of what is interesting for testing (Graham et al. 2008, Chapter 1.4). Identifying test conditions can be based on several aspects and it is always situational. Test conditions may rely for example on function, requirement or feature. A 100 % coverage cannot be expected and attention needs to be paid to

completion criteria, for example a percentage of the code coverage. (Hass 2008, Chapter 2.3)

Design the tests, which are connected to certain features that are particularly interesting or involve high risks (Graham et al. 2008, Chapter 1.4). It is possible to create the first high-level and low-level test cases by leaning on to test conditions. High-level test case is a case without specific input values or expected results, however, it contains logical operators or some other means used to defining what is being tested in general. According to Hass (2008, Chapter 2.3) high-level test cases have four rules of thumbs: effective, which means that the cases hold a high probability for finding errors; exemplary, which means that cases should be practical and contain little overlapping: cost effective, which stands for reasonable price and return on investment (ROI) and lastly, evolvable, which means structural, flexible and maintainable. Testing techniques help to identify required input values for high-level test cases which is the reason why they do not have to be defined in actual cases. The decision of how many high-level test cases and conditions are documented is based on strategy and involved risks. Low-level test case is a case where also an input and expected results are defined. Low-level case documentation must include at least following aspects: Unique identification, preconditions, input, expected results and postconditions. (Hass 2008, Chapter 2.3)

Evaluate testability for system and requirements. System testability can be defined by whether is it possible to test a system on an environment which equates the eventual operational environment and whether it is possible to test all configurations and ways the system can be used. For example on websites it is hardly possible to test the site with all the versions or all the browsers and different firewalls. Testability for requirements means that the requirements have been documented in a way that can be used for building tests. For example, the requirement “the software needs to respond quickly enough” is not testable because different people may understand the word “enough” very differently. (Graham et al. 2008, Chapter 1.4)

Design and build the testing environment and recognize the necessary tools. In other words, fix up all the systems and items that are required for executing the testing. (Graham et al. 2008, Chapter 1.4)

4.3 Implementation and Execution

At the test implementation phase the level test plan and other possible documents depending on the project such as the user manual, preceding test work, and templates for reporting and logging information are observed, as well as the test specification started at the analysis and design phase. The most important information to gather from the level test plan are: definition of the testable items or objects, scheduling and staffing of people included in the testing process, specification of the test environment for building, entry criteria to know when the testing of the objects can be started and exit criteria to know when the task is done. (Hass 2008, Chapter 2.4)

After the information about the test conditions has been collected, it will be transformed into the test cases, manual test procedures and automated test scripts. (Graham et al. 2008, Chapter 1.4) It should be taken into account who is doing the testing, an experienced tester or specialists will not need as detailed description on the procedure as an inexperienced tester. All procedures must have unique identification codes, general information, and the test cases, arranged into a specific order to run. One test procedure should have a limited amount of test cases, from 2 to 20. See Table 1 for an example of a test procedure template. (Hass 2008, Chapter 2.4)

Table 1 Test procedure template (Hass 2008, Chapter 2.4)

Test procedure:			
Purpose: This test procedure tests … Traces:			
Prerequisites: Set up …			
Expected duration: *x* minutes			
Execution information			
Test date and time:		**Initials:**	
Test object identification:		**Result:**	
Case	**Input**	**Expected result**	**Actual result**
1.			
2.			

By iteratively designing and organizing new test cases, procedures and test groups the test specification will be improved, this process will continue with the development until the satisfactory coverage is achieved. The test specification must be reviewed before it is used in the execution and the most important questions are if the specification is clear and easily understood, can the tests be automated, is it easy to maintain and if performing technical reviews will be easy? (Hass 2008, Chapter 2.4)

Setting up the test environment for the test execution is crucial, it should be described with precise detail in the specification and taken seriously so we can be confident about the results. According to Hass (2008, Chapter 2.4), the description of the environment must cover at least the following:

-Hardware—to run on and/or to interface with

-Software—on the test platform and other applications

-Peripherals (printers including correct paper, fax, CD reader/burner)

-Network—provider agreements, access, hardware, and software

-Tools and utilities

-Data—actual test data, anonymization, security, and rollback facilities

-Other aspects—security, load patterns, timing, and availability

-Physical environment (room, furniture, conditions)

-Communication (phones, Internet, paper forms, paper, word processor)

-Sundry (paper, pencils, coffee, candy, fruit, water)

If a proper environment for testing has not been set, testers might have to lean on pre-existing environments for executing tests such as the development environment, where the results can be unreliable and in worse case cause harm to the business. (Hass 2008, Chapter 2.4)

The actual execution of the tests can be started after we are certain that the entry criteria has been met. By performing the tests by following the procedures the results of testing can be relied on, the wanted results or defects can be repeated, time can be collected to make better time estimates and the progress can be measured. New ideas and cases need to be introduced through incident management system and taken into account in the next testing cycle. (Hass 2008, Chapter 2.4)

The results of testing need to be logged, as well as the identities, versions, test tools and testware. All this information should be combined wih the test procedure template, which is illustrated in Table 1 for example. From the template the results of testing can be compared to the expected result. (Graham et al. 2008, Chapter 1.4) Depending on how formal the testing is, the

reporting can range from an "ok" marker to detailed descriptions to screen shots and even video. (Hass 2008, Chapter 2.4) In case the test leads to a failure, it will be logged as an incident into the incident management system. The necessary details about the defect will be included in the report, such as how to produce it, identify what caused it, if it might have been caused by defective or insufficient test data, or mistakes done when executing the test. After a fix has been performed for the reported incident, the test activities must be repeated to confirm a fix and to see that no new defects have appeared from the fix. (Graham et al. 2008, Chapter 1.4)

4.4 Evaluating Exit Criteria and Reporting

Evaluating exit criteria is a phase in the testing process where executed tests are being compared against items' pass/fail criteria specified in planning phase. After test execution, the test manager will evaluate whether the criteria have been achieved (Hambling et al. 2010, Chapter 1). Exit criteria should be defined separately for each test level and the criteria always vary in the different projects. With the criteria it can be established that a certain test level or task has been completed (Graham et al. 2008, Chapter 1.4). Hass (2008, Chapter 2.2) has given examples of possible exit criteria:

> *-Specified coverage has been achieved*
>
> *-Specified number of failures found per test effort has been achieved*
>
> *-No known serious faults*
>
> *-The benefits of the system as it is are bigger than known problems*

If the determined exit criteria have not been met, testing cannot be finished. This is when the testing process is executed iteratively which means that a phase is returned to where testing can be repeated, which will ensure that the criteria will be met. This means often returning to the test design and analysis

phase because this is where new test cases can refactored and createed and the coverage increased. (Hass 2008, Chapter 2.2)

According to Hambling et al. (2010, Chapter 1) and Graham et al. (2008, Chapter 1.4) the exit criteria contain three key points.

Ensuring that the defined exit criteria have been achieved. Here the logs are compared with the criteria, which means evaluating in the light of evidence what tests have been executed and what defects have been discovered, fixed, re-tested or what tests are left undone.

Evaluate whether more tests are required or will the exit criteria demand changing. As mentioned above, increasing the number of tests or refactoring them may be necessary when the achieved coverage has fallen below the criteria or risks have increased during the project. One option is to refine the exit criteria when the demanded coverage can be lowered. In these situations lowering the criteria must always be agreed in mutual understanding with stakeholders. (Graham et al. 2008, Chapter 1.4)

Writing up the test summary report for stakeholders and other business sponsors. The test summary collects all the test activities and results. The document also includes evaluating the performed testing by comparing it with the exit criteria. The knowledge of test results is not sufficient itself if it is held solely by testers. All of the stakeholders ought to know what kind of testing has been performed and what results have been gained, which ensures that fact based decisions for the software can be made. (Graham et al. 2008, Chapter 1.4)

4.5 Test Closure Activities

The purpose of test closure activities is to gather and reflect experiences and store the test ware for the future. Hass (2008, Chapter 2.6) has divided test closure activities into three different categories.

Input

The activities require a certain amount of input and resources. To complete the activities, a schedule and staffing have to be made in order to achieve the goals. Input also requires basically all the test ware and deliverables produced during and after testing such as plans, specification, environment and reports. It is important for experiences to be taken into account because the participants often have a different view of what has been happening.

Overall procedure

The first thing to do is to check the completion again. Before finishing the actual testing it must be certain that the determined exit criteria have been achieved, which applies to the test coverage as well as to the test deliverables. If it does not, or some incidents are not sufficiently documented, it must be ascertained that they are before moving forward.

The second category is to archive the testware, usually into configuration manager, but if one does not exist it is sometimes necessary to determine some other secure location. The last thing to do is to have a retrospective meeting and report the experiences. This is where the information received during the testing is collected, analyzed and refactored to knowledge. This knowledge can be used for example in testing process improvement and these are the facts that are most capable of answering the question of what should be developed. Sometimes test closure activities include metrics which can be, for example, how high percentage of tasks has been completed. These metrics help the company to make estimates and schedules for future work.

Output

Test closure activities produce different deliverables. One important target is to produce a document, a test experience report, which is created during retrospective meeting. The second key part is the test ware which is documented in configuration manager or other location as mentioned above.

5 Testing Process Improvement

Just like testing that improves the software, methods to improve the development processes themselves can be utilized. These same methods can be used to improve the testing process. The methods strive to improve the process and its products by offering guidelines and bringing up areas that need improvement. (ISTQB 2012, 57)

5.1 Test Improvement Process

Most software development models like Capability Maturity Model Integration (CMMI®) do not pay much attention to testing, therefore some improvement models were designed just for the testing process: Test Maturity Model Integration (TMMi®), Systematic Test and Evaluation Process (STEP), Critical Testing Processes (CTP) and TPI Next®. The testing forms a large part of the development process and is often left insufficient, these models are designed to fix the lack of comprehensive testing in the most software development models. (ISTQB 2012, 57)

Improving the processes is an ongoing practice, because the processes cannot ever be perfect and there is always room for improvement. An old but still viable way for testers to improve a process is with the Deming improvement cycle PDCA "Plan, Do, Check, Act". Process models find the place where to start the improving by measuring the organization's process capabilities against the model. The models also offer a framework that can be utilized for improving the organization's processes based on the results of the assessment. (ISTQB 2012, 57)

According to the ISTQB syllabus (2002) the process improvement models can be divided into two categories:

1. The process reference model, where a maturity measurement will made and used to evaluate the organization's efficiency against the model and provides a roadmap for improving the process.

2. The content reference model, where a business-driven assessment will be made of the organization's opportunities for improving, sometimes it can include evaluating the organization against the industry averages by using objective measurements. This assessment can be used to create a roadmap for improving the process.

It does not mean that these models should be used in all cases and they are the only method of getting results, the testing process can also be improved by using analytical approaches and retrospective meetings.

5.2 Steps of Improvement

The IT industry can use the testing process improvement models to achieve a higher maturity level and professionalism. The example by ISTQB (2002) uses the IDEAL^{SM} model [IDEAL96] for how to improve the process step by step:

-Initiating the improvement process

-Diagnosing the current situation

-Establishing a test process improvement plan

-Acting to implement improvement

-Learning from the improvement program

See Figure 4 for illustration of the previous model as an iterative process.

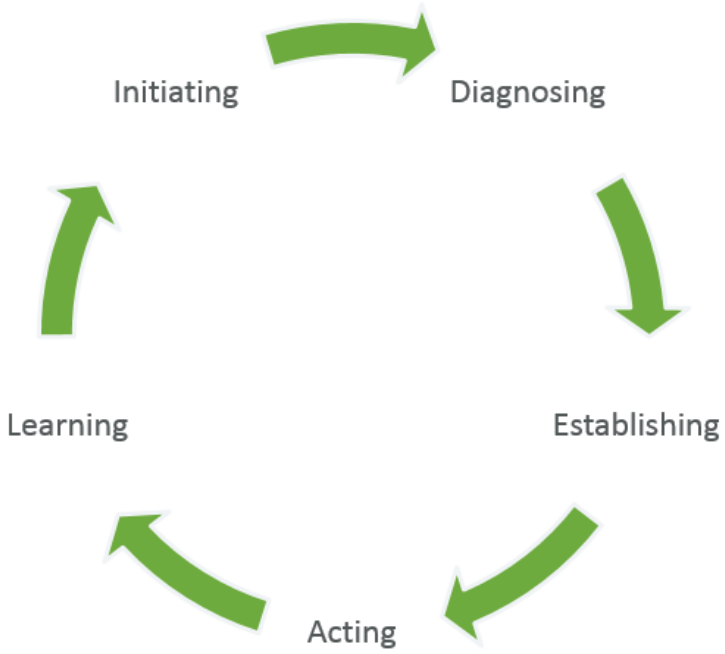

Figure 4 Iterative process improvement lifecycle

Initiating

Before the process improvement activities are started, the stakeholders must agree on the goals, objectives, scope and coverage of the process improvements. The process improvement model has to be picked out in this step, it can be one of the publically available options such as CTP, STEP, TMMi or TPI Next, or the model can be developed internally. The success criteria and a method which will be used to measure the improvement activities should be defined in this step.

Diagnosing

The appointed evaluation approach will be used and with it an assessment report will be formed, the report will include an estimation of the current testing practices and a list of possible process improvements.

Establishing

The list for possible process improvements will be prioritized. The prioritizing can be founded on returns gained from investment, risks, organization's

strategies and/or measurable quantitative or qualitative benefits. When the order of priorities has been settled, a plan for execution of the improvements will be developed.

Acting

The improvement plan for the testing process will be implemented. This might include arranging training or mentoring, piloting the processes and finally their full implementation.

Leaning

After the improvements has been fully implemented, it is crucial to verify the benefits achieved whether they were defined early on or unexpected benefits. It is also important to verify which steps of the process improvement has filled the success criteria.

Depending on the process model used, this is the step where we decide whether to start monitoring the next level of maturity, if the whole improvement process is started again, or if the process is stopped.

5.3 TMMi

Testing Maturity Model Integration (TMMi) is a detailed model for test process improvement and complementary to the CMMI. (TMMi Foundation, 6) TMMi is formed of five maturity levels and each of them includes predefined process areas. The levels have set generic and specific goals that have to be completed at least to 85% before the organization can move to the next level. (ISTQB 2012, 59)

The maturity levels of TMMi:

Initial

At level 1 the organization does not have any defined testing processes implemented. Most testing is done as ad-hoc testing after the code has been developed and is often considered as part of debugging. The idea of testing at

these organizations is just to show that the software works without any major failures and the success of testing depends solely on the testers' competence. Software is often released buggy, slow and has not filled the requirements and needs of the customer. The organization lacks of skilled testers, resources and tools for testing. (TMMi Foundation 2002, 10)

Managed

At level 2 testing is no longer considered as a part of debugging and process areas are defined and managed. Some types of testing are done, including component, integration, system and acceptance testing. All of the testing is planned and the set objectives are followed, the product fulfills the requirements of the customer. The following process areas are defined: Test policy and strategy, test planning, test monitoring and control, test design and execution, test environment. (TMMi Foundation 2002, 10)

Defined

At level 3 testing is no longer considered as a byproduct to follow coding, but an integrated part of the development lifecycle that starts early and happens at all stages. Code reviews are made across the lifecycle, depending on the type of code being reviewed, professionals specialized in certain types of testing can be included, for example security testers or domain experts. More types of testing are being done, including different types of nonfunctional testing. The process areas at level 3 are defined in a more detailed level, while they build on top of the processes made at level 2 and they also need to be revisited and improved with more detail. The following process areas are defined: Test organization, test training program, test lifecycle and iteration, non-functional testing, peer reviews. (TMMi Foundation 2002, 11)

Measurement

At level 4 testing needs to be a comprehensively defined, well-founded and measurable process. Measurements are being produced to help with decision making, to assess productivity, and to evaluate the quality and attributes of the

software on specific projects. Reviews, inspections and peer reviews are an integrated part of testing at level 3. The following process areas are defined: Test measurement, product quality evaluation, advanced peer reviews. (TMMi Foundation 2002, 11-12)

Optimization

At level 5 the testing processes throughout levels 1-4 have been successfully implemented, the information gathered from the testing process can be used to prevent defects and to improve and optimize already existing processes. Testing methods and techniques are improved when needed and the organization strives for constant fine tuning and improvement of the processes. The following process areas are defined: Defect prevention, quality control, test process optimization. (TMMi Foundation 2002, 12)

5.4 TPI Next

Test Process Improvement (TPI Next) is a practice based assessment model that consists of 16 key areas and four maturity levels. (ISTQB 2012, 59) The model is designed to improve the maturity of the organization by providing balanced business driven-improvement paths. The model provides step-by-step guide to improve multiple key areas. Each of the key areas covers a specific area of the testing process: stakeholder commitment, degree of involvement, test strategy, test organization, communication, reporting, test process management, estimating and planning, metrics, defect management, methodology practice, tester professionalism, test case design, test tools, test environments. (Aaltio 2013)

The four maturity levels are: initial, controlled, efficient, and optimized. (Aaltio 2013)

Improving the test process with TPI Next is done iteratively, when enough awareness has been raised inside the company about the problem, the scope, goal and approach will be determined. After the current situation has been assessed the plan for improvements needs to be defined. A plan of action will

be made and put into motion. After the implementation the results will be evaluated and depending what needs more work, the developers go back to either figuring out the goal, scope or approach again, or to assess the new current situation, or go back straight to defining improvements for the next iteration. (Aaltio 2013)

5.5 STEP

Systematic Test and Evaluation Process (STEP) is mainly a content reference model. Like in CTP, the model does not make the developers do the improvement actions in a certain sequence. STEP is based on the idea that testing should be done throughout the whole development lifecycle, from the beginning in the form of requirements specification to all the way until the software is retired. STEP is primarily prevention oriented, it supports the idea that tests should be made before the coding, so it is ensured that the code will always be testable and that it fills the set requirements. (ISTQB 2012, 60)

STEP is formed from specified tasks (individual actions), work products (documentation and implemented tests), and roles (defined responsibilities associated with groups of tasks). The four major roles and responsibilities in STEP are the following: manager (communicate, plan, coordinate), analyst (plan, inventory, design, evaluate), technician (implement, execute, check), reviewer (examine and evaluate). Depending on the size of the organization and the project, these roles do not have to be done by different individuals, but possibly they all can be done with just one person. STEP is not a tool dependent model nor does it expect the organization to have a certain staffing and test groups, but it does expect testers and developers to work together and to do their respective responsibilities. STEP tries to prevent bugs being born at all, and if defects happen, they are detected early on before they cause a large amount of damage. (Craig & Jaskiel 2002, Chapter 1)

5.6 CTP

Critical Testing Processes (CTP) model's basic idea is that some of the testing processes are critical and if they are carried out well they will support the testing team greatly. To the same extent if they are performed badly, it is unlikely that even a team of skilled testers and test managers will be successful. The model defines 12 critical testing processes. CTP is mostly a content reference model and adapts to all software development lifecycle models. Metrics are used in CTP to benchmark the organization against industry averages and best practices. (ISTQB, 60)

CTP was created to be a lightweight framework focusing on the most important areas of the testing process that should at least be done properly, unlike TMMi and TPI which are much more complex and comprehensive. Like most of the improvement models, also CTP is done in an iterative manner, however, unlike many of the other models it does not define in which order to do the needed improvements. CTP is a flexible model and it can be tailored to suit different projects by identifying the individual challenges and good quantitative and qualitative process attributes. The order of executing the improvement activities can be decided by the organization, the most critical areas that produce business value or cause a great deal of pain to an area might be the best places to start. (Black 2013)

According to the creator of CTP, Rex Black, the 12 testing processes defined as critical are: testing, establishing context, quality risk analysis, test resource estimation, planning, test team development, test system development, test release management, test execution, bug reporting, results reporting, change management. (Black 2003)

Figure 5 Critical Test Process steps (Bath & Veenendaal 2014)

CTP at the highest level is performed through four main steps as illustrated in Figure 5. Bath & Veenendaal (2014) have presented how each of the critical processes can be sorted in to specific steps, as seen through Table 2 CTP activities in the Plan-step (Adapted from Bath & Veenendaal 2014) to

Table 5 CTP activities in the Perfect-step (Adapted from Bath & Veenendaal 2014). Bath & Veenendaal (2014) have changed some of the process names from Black's (2003) list a bit to be more descriptive.

Table 2 CTP activities in the Plan-step (Adapted from Bath & Veenendaal 2014)

Plan	
Process	Aspects covered
Establishing context	• Life cycles used • Current or planned process improvements • Testware created (specific items and the value they add to the project and/or organization) • Existing testing practices • Stakeholders relationships (expectations, areas for improvement) • Management expectations of testing • The Role of QA as compared to testing
Establishing a risk-based strategy	• Current practices for risk management • Identifying stakeholders • Identifying, categorizing, documenting, and agreeing with stakeholders on product quality risks their mitigation strategies, and priorities
Estimate resources	• Developing a work breakdown structure (WBS) considering all test process activities • Project criticality • Scheduling the testing based on WBS • Estimating budget needs based on the WBS and schedule • Costs and benefits of testing (return on investment)
Develop a test plan	• Test planning process (create a test plan, distribute, review, negotiate, agree, adjust) • Good planning practices to apply (e.g., for outsourcing, for setting completion criteria, etc.)

Table 3 CTP activities in the Prepare-step (Adapted from Bath & Veenendaal 2014)

Prepare	
Process	Aspects covered
Establish a test team	• Required skills, attitudes, and motivation • Hiring and staffing • Career paths for testers • Skills development
Develop testware and set up testing infrastructure	• Required testware • Test coverage • Test conditions • Test designs to be used • Design of test cases • Use of test oracles • Combinatorial challenges • Design, implementation, and verification of test environment • Configuration control • Updating risk management information

Table 4 CTP activities in the Perform-step (Adapted from Bath & Veenendaal 2014)

Perform	
Process	Aspects covered
Install software required for testing	• Identifying and installing software under test • Managing the test release • Smoke testing
Assign, execute, and manage tests	• Test case selection • Assignment of test cases for execution • Execution of test cases • Recording results • Adjusting priorities and plans

Table 5 CTP activities in the Perfect-step (Adapted from Bath & Veenendaal 2014)

Perfect	
Process	Aspects covered
Document bugs found	• Defect reporting process • Testing and debugging • Communication of defects • Defect tracking tool selection
Communicate results	• Test results reporting process (steps and good practices) • Handling the presentation of results
Adjust to context changes and improve the test process	• Change management process (gather, select. review, plan, present, decide) • Attributes of a mature test process • Sources of formal test process • Overview of testing strategies • Incremental process improvement • Issues in implementing improvements

6 Agile Testing

Agile testing is not necessarily only performed in agile development. Some testing approaches are naturally agile whether they are being practiced in an agile project or not, for example exploratory testing (Crispin & Gregory 2009, 7). The purpose in agile testing is to deliver the quality that customer requires and each agile team member is committed to produce a high-quality product that provides business value (Crispin & Gregory 2009, 5). Agile testing can be divided into four quadrants, the numbering or the order of the lists do not define the order of the activities.

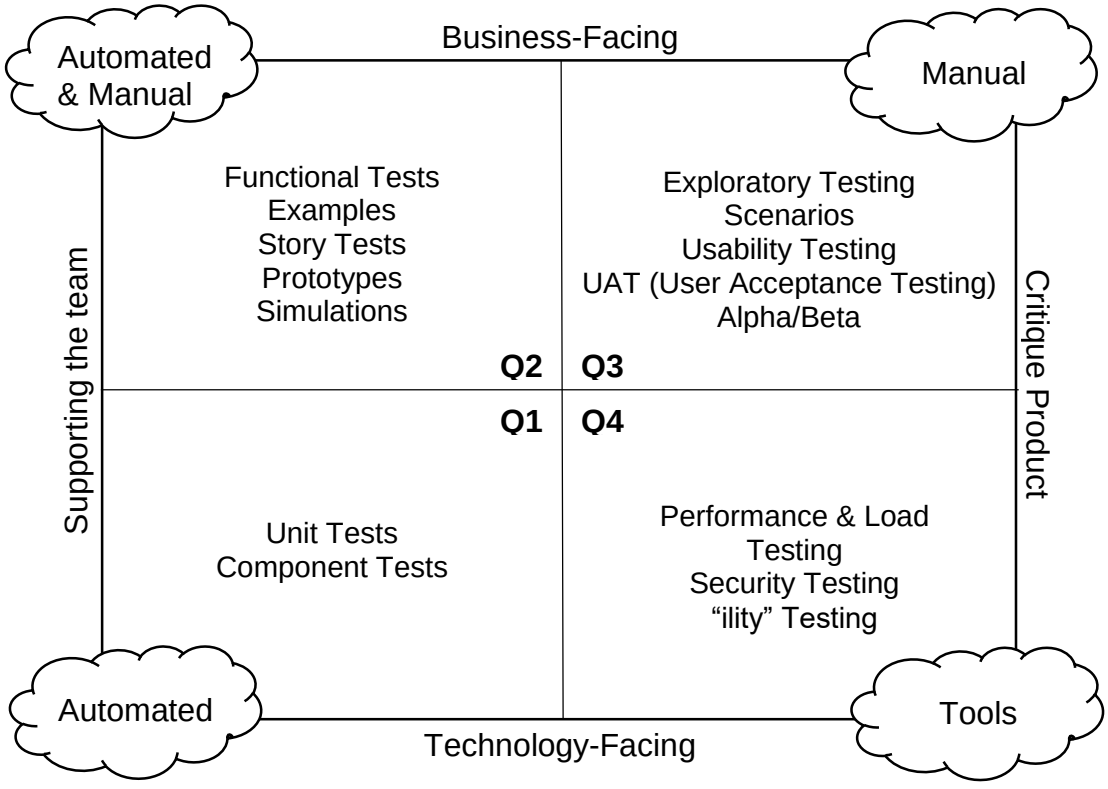

Figure 6 Agile testing quadrants (Crispin & Gregory 2009, 98)

6.1 Team Supporting Technology-facing Tests

Team supporting technology-facing tests are the base for agile development and testing (Crispin & Gregory 2009, 111). Other quadrants (see Figure 6) cannot compensate for insufficient implementation of quadrant one (Crispin & Gregory 2009, 110). The purpose behind unit testing is to ensure that the software units are reliable and they fulfill the requirements, however, in the big picture the goal is to recognize defects in the program logic. Typically the programmer who has built some component will perform the unit tests for that component right after finishing coding it (Watkins & Mills 2001, Chapter 5.1). When using TDD in agile, the programmer writes units or component test first to cover some small part of code and begins programming until the test passes. By making the units small and as simple as possible programmer has a chance to think about the functionality that might be essential for the customer. If post-development testing is busy with finding and fixing bugs, there is no longer time for discovering serious faults that could affect business. Most of unit level bugs can be prevented by writing tests before code (Crispin & Gregory 2009, 111). Unit and component tests contribute for quality assurance by providing guidance for planning, which helps programmers to understand how code should really work (Crispin & Gregory 2009, 110).

Automated unit and integration tests make is possible for programmers to refactor frequently. This helps in making code more maintainable and provides the best value for the time that is being used. So called “technical debt” is being kept as low as possible (Crispin & Gregory 2009, 112). Automating unit and component tests prevents the testers from having to spend too much time on finding low-level bugs. One of the greatest benefits of unit tests is the speed of feedback. By wasting little time on basics, the testers can focus on exploratory testing or high-level business functionality and explore unexpected situations or scenarios that programmers have not had time to think about. This is a good way to learn more of how the software should really work. (Crispin & Gregory 2009, 118). Unit tests are always individual and test one dimension at the time. That is why programmers can easily recognize and fix

the bug if some test fails (Crispin & Gregory 2009, 120). When a bug is discovered, it will be presented to programmer who writes a unit test after which he fixes the code so that the fresh unit test will pass. (Crispin & Gregory 2009, 113)

If the unit tests are left unnoticed, eventually all the time of testers will be spent on discovering so many defects from “happy path” that there is no possibility to test more complex scenarios or borderline cases (Crispin & Gregory 2009, 112). Good source code management, configuration management and continuous integration (CI) are essential so that tests performed by programmers provide benefits that guide the design. Idea is that development team is always aware of what is being tested and the software modules can be tested every time new code is submitted. If the tests will not pass, the programmer who added the code earlier can fix the problem rapidly. CI is good for time saving and motivating coders to perform tests before submitting code. If these core practices are missing, the agile projects can easily turn into “mini-waterfall” projects. This means that development cycles are shorter than in traditional waterfall, however, the code is being dropped off to testers who do not have enough time to test because the code quality is questionable. (Crispin & Gregory 2009, 111-112)

Writing team supporting technology-facing tests can be difficult. Often adapting to agile practices start with no unit test automation or other core practices (Crispin & Gregory 2009, 120). Eventually it is a matter that whole development team should address. The tests from quadrant one are the base for rest of the testing done in the project. By doing ineffective work with these tests it is more difficult to gain maximal benefit from other tests but it is more difficult because the internal quality of code is not as high as it could be. (Crispin & Gregory 2009, 123)

6.2 Team Supporting Business-facing Tests

This chapter discusses about the 2nd quadrant seen on Figure 6. In agile development we cannot spend time making traditional requirement documents to tell the team what to test, because it would cost too much time and documents are not flexible enough in the often changing fast paced incremental environment. Instead examples and business-facing tests are to be used to let the team know what to code. (Crispin & Gregory 2009, 150)

Iteration will often start with limited information and we need to start making the program as soon as possible, so tests are formed to understand what to code from story cards. The customer team will write their desired features as stories or groups of stories. The story cards are short and describe a feature the customer wants, as seen on Figure 7. Testers, programmers and other technical personnel can take part in the story creation and help the customer with technical stories, limiting story sizes and suggest better alternatives for implementation and mitigating risks. The stories will be expanded until the development team have enough information of what the customer needs. Testers will provide examples and context for stories and help the customer to write tests for their stories, from which the programmers can start doing their job. The features and requirements of the program can and usually will change many times during the development as the customer might change his mind or figure out new requirements. By making only a small part of the software in one iteration, the customer will see what was done and how, and if

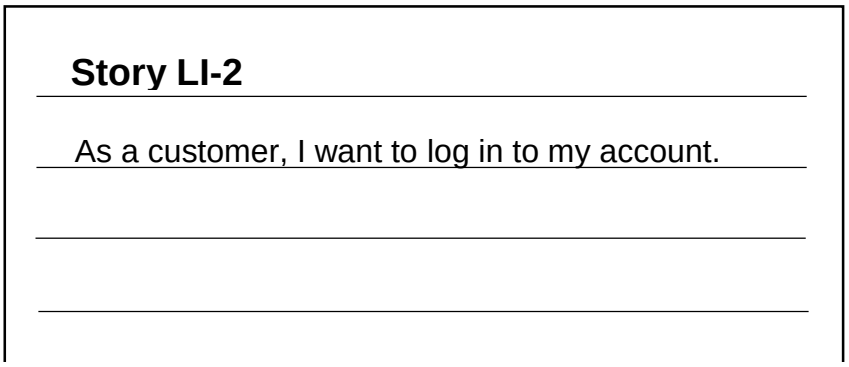

Figure 7 Story card with a feature (Crispin & Gregory 2009, 130)

they change their mind at this point, the change or a new feature is easy to implement in the next iteration. (Crispin & Gregory 2009, 129-133)

Often the customer cannot explicitly tell all the wanted features or does not yet know them, so it is important especially for the testers to help the customer to figure them out in a form that will help the developers. (Crispin & Gregory 2009, 135) Testers can help in the process by asking questions, forming the worst and best case scenarios, using examples like pictures, flow diagrams, spreadsheets and prototypes, and in the future iterations with prototypes and simulations. Asking about the best case scenario usually forms the happy path test, which tells how the software should work. (Crispin & Gregory 2009, 134, 136) After the happy path is known, tests should also be made for the high risk scenarios: the worst things that can happen and negative cases that have a good probability of happening. (Crispin & Gregory 2009, 147)

In agile development and testing, the focus is on one feature at a time, thus their dependencies to other features need to be checked and made sure they all work together. It is important to keep the big picture in mind and make tests confirming that one feature does not break the functionality of others. (Crispin & Gregory 2009, 143) A good idea is to start the development with a "thin slice" that forms the happy path from end-to-end. This forms the "steel thread" for the software that connects all the components together, it can also be used to verify the architecture, and it is easy to add more functions on top of it incrementally. (Crispin & Gregory 2009, 144-145) It is also important to keep track of the time used for designing the tests. After an iteration it is checked if the team had enough tests to develop the software through them, did the stories cause misunderstandings, or if the tests or stories lacking detail. This way we know if more time and effort need to be invested into creating tests in the next iteration. It might be wise to write only the high level story tests before the coding, do the detailed test cases when the coding starts and exploratory testing after the code has been written. (Crispin & Gregory 2009, 149)

The techniques in this section test business requirements and will not touch the technicality of tests and code; the main objective here is to define how and

what the program should do. All members included in the development should be able to understand the requirements and examples, so a language and format should be used that everyone can understand. (Crispin & Gregory 2009, 129-131)

Business-facing tests that drive the coding should be automated early on, they should be easy to understand and run, and provide quick feedback, so people will use them often to check the functionality of the code. (Crispin & Gregory 2009, 149) Later these same tests can be used in the regression test suite without having to make new tests. (Crispin & Gregory 2009, 131) The business-facing customer tests give a guideline for the programmers to know what unit tests to write, so they can be sure they meet the customer's requirements. (Crispin & Gregory 2009, 149-150)

6.3 Business-facing Tests that Critique the Product

The 3rd quadrant discusses the business-facing tests that critique or evaluate the product, which can be seen on Figure 6. System assessment and critiquing is based on end users' behavior adaptation in testing. Understanding different scenarios and business situations or rules contributes to achieving accurate experience. Business-facing tests that critique the product are very difficult to automate because they rely on human instinct and experience. (Crispin & Gregory 2009, 190)

Scenario testing

The user activities can be assessed by defining different scenarios. Real-life domain knowledge is essential in order to be able to mimic end users behavior and create accurate scenarios. The idea is to test the system end to end, however, this does not necessarily mean black box testing in this case. (Crispin & Gregory 2009, 192)

Testers have the tendency to create the test data themselves and usually very simple data so that it is easy to see the results afterwards. When different scenarios are being tested, the test data and flow has to be realistic.

Therefore we have to know whether the data comes from external system or is it being inserted manually. Sometimes sample data needs to be requested from customer to enable scenario testing so that the real data will flow through the system (Crispin & Gregory 2009, 193). Tools present effective support for defining scenarios, and workflows work very well for this purpose. Flow diagrams help with identifying usual events and the diagrams of scenarios can be helpful for thinking through highly complicated issues (Crispin & Gregory 2009, 194). Figure 8 illustrates an example of a flow diagram.

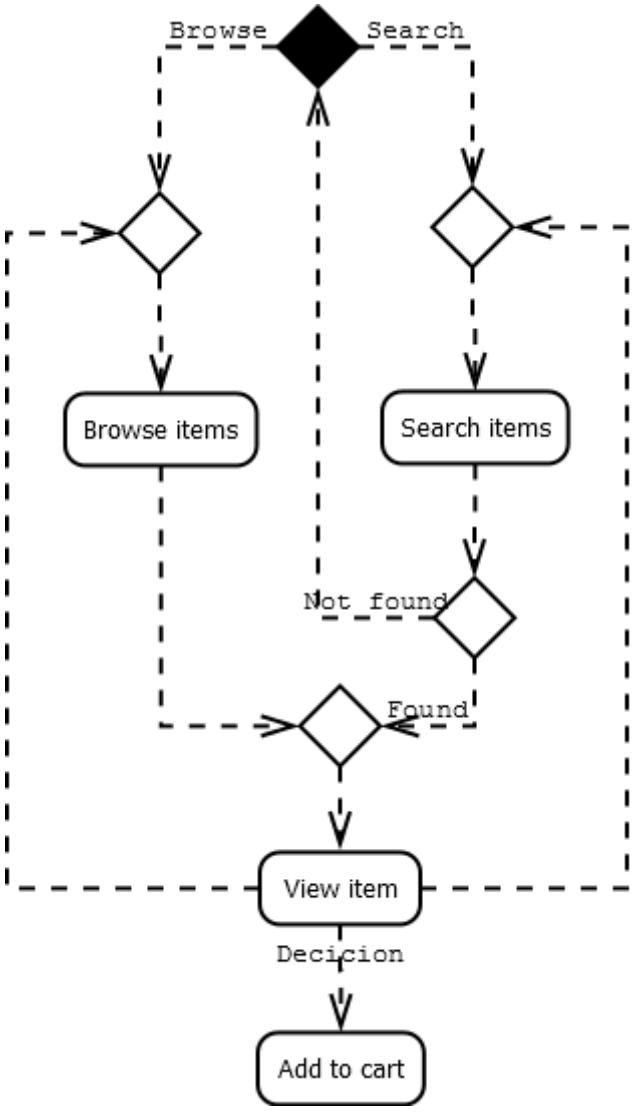

Figure 8 Example of simple online shopping flow diagram

Exploratory testing

Exploratory testing is often mistaken as ad-hoc testing, however, these are two different things. Inexperienced black-box testers do not always really know how exploratory testing works. (Crispin & Gregory 2009, 198)

Exploratory testing is an important part of agile testing. Exploring prefers customer association over negotiated contracts and therefore is highly based on agile software development (Crispin & Gregory 2009, 198). This is a careful approach towards testing without scripting and works to reinforce automated regression suites and story tests. Exploratory testing makes it possible to dig deeper than obvious variations. Exploratory testing combines learning, test design and test performing in to one approach. As a valuable by-product new aspects of software are discovered which might require new regression tests, new features or refactor already existing features which can lead to new stories (Crispin & Gregory 2009, 195). Some tests may be automated in future while others only answer to one question and are sidelined after one execution. Manual exploratory testing can be a very powerful way for discovering functional defects, however, without having sufficient business-facing regression test suite, all the time would probably be wasted trying to desperately regression test manually (Crispin & Gregory 2009, 280). Agile team has to be critical to what is been learned from the software and improve the tests accordingly. A rule of thumb could be that repeatable tasks are automated and human resources are exploited in matters where people are good at, such as seeing, thinking and handling the unexpected scenarios. (Crispin & Gregory 2009, 199)

It is important to remember that exploratory testing is not at all an independent testing technique but an approach to testing or a mindset what can be connected to any testing technique. Secondly, is has to be remembered that exploratory testing approach can be implemented for example in test design and analyzing, therefore it is not only for test execution. Thirdly, exploratory testing is not careless or without a plan. Exploratory mindset can require a

very long and mature devotion and the knowledge is developed during the years. (Crispin & Gregory 2009, 196)

Exploratory testing usually begins by designing a chart or list of what functionalities are going to be explored. Testers write notes during the testing so that they are able to reproduce possible errors and continue exploring them if necessary (Crispin & Gregory 2009, 198). Agile team has many possibilities to perform exploratory testing since a version of software is being developed during every iteration. When explorative testing is begun in the beginning of every iteration it can be based on the following guidelines (Crispin & Gregory 2009, 198):

Risks (analysis) meaning the critical features that development team or customer believe can fail or potentially arise issues. Models (mental or something else) of how software should work. There are expectations of how a new feature should work so it is better to test it. Past experience of what kind of issues similar software have had or what has been good about them. It is best to use predictable patterns so that the test can be designed and explored. The knowledge of development team has to be taken into account. It must be clear what is really important especially for the team. What can a tester see and observe? As a member of agile team, a tester must continuously learn more about the product, development team and customer. Through continuous learning, the mistakes that the team is making and found quickly along with good or bad features in product and customers' needs. (Crispin & Gregory 2009, 198-199)

According to Crispin & Gregory (2009) the following qualities are required from successful exploratory tester:

Exploratory tester is systematic but pursues to discover inconsistent pieces of software. The tester learns to identify problems and is able to choose an emphasis, theme or a role for testing. They can think about how experience affects software for example how experts' behavior differs from amateur. The

tester also explores similar or competitive software together with domain experts. (Crispin & Gregory 2009, 201-202)

Usability testing

Usability testing means testing that critiques the product by looking at it from end users' perspective. To support testing different invented characters can be used. These characters have different skill levels for using software and different purposes for it. They are especially helpful for testing scenarios because the scenario is tested from each of the character's perspective. The results then direct the product to right direction. For example, if there are many amateur users accurate introductions for software might be required. (Crispin & Gregory 2009, 203)

Navigation is a large part of usability testing. For example it is essential to test links and tabs from navigation. Navigation is the first thing fresh users usually need to use and if the first experience is poor, there is high probability that the user dismisses the product. Although parts of testing navigation should be automated, it is important to test from user experience point of view. (Crispin & Gregory 2009, 204)

When evaluating the systems usability we can benefit from other similar systems by asking questions about their qualities. How do they work? Are they user friendly? What is good or bad? If possible we can try using competitor system and reflect it against the own system. Usability is quite specialized field of testing and implementation depends on project. If we are building small software for internal use and user group is small, usability testing is going to probably play a small role; however, when building an assistance system for phone company, usability may have an essential role. In this case it would be important to learn as much as possible from system or import an external usability expert. (Crispin & Gregory 2009, 204)

User acceptance testing

User acceptance testing (UAT) is especially important for large customized systems or internal systems. The goal is to ensure system's usability and confirm business functionality for already existing and added features. UAT is performed by all the business departments that the system affects. Customers are those who actively operate with the system. Therefore they are the ones who have to take care that the system is working as expected. (Crispin & Gregory 2009, 464-465)

UAT begins usually when development team decides that the quality of software has reached the required level to be released. Although sometimes the schedule dominates on release cycle. Working with the customers is vital, so that the customers understand process, their roles and what expectations are focused on them. If UAT does not work smoothly, there is an increased risk that high level support will be needed. An experienced UAT team may design some test cases but most commonly performed testing is ad hoc. Customers can approach testing just as if they were doing their daily work but focusing on some certain feature. This is also powerful way of observing how people are using the system and to receive feedback of what is good and what improvements could be needed. Testers can support customers who are performing UAT by reviewing the tests, logged defects and by tracing defects. (Crispin & Gregory 2009, 465-466)

Alpha/Beta testing

Organization with large customer base may not have opportunity to perform UAT. In these cases it is recommended to do alpha or beta testing. The objective for development team is to gather feedback from real users and this is how it is possible. From tester's point of view it is important to understand how customers see the software because it may have an effect on how the software is being tested. Alpha and beta testing can be the only time when development team gets to interact with end users and therefore alpha and

beta testing should be exploited as effectively as possible to find out how the software matches with customers' needs. (Crispin & Gregory 2009, 466-467)

Alpha testing means early distribution of software. In this phase the software is likely to contain major defects and alpha testing customers should be selected with caution. When the selected approach is alpha testing it must be ensured that the customers understand their role and expectation. The goal is to get feedback on features instead of reporting on bugs. (Crispin & Gregory 2009, 466)

In beta testing the idea is to release a nearly finished software which can really be used. It might not be completely ready for all the customers but many of them may find the benefits of new features greater than the risks. What customer should understand is that beta release is not complete product and users are expected to test it and report bugs. (Crispin & Gregory 2009, 467)

6.4 Technology-based Tests that Critique the Product

The fourth and last quarter seen on Figure 6 deals with the nonfunctional requirements of the software (Crispin & Gregory 2009, 218). The tests are technology-facing and such requirements are written with programming domain vocabulary rather than with business terms. The purpose of the tests is to critique the product's performance, load capabilities, memory management, scalability, security, maintainability, interoperability, compatibility, reliability and installability. (Crispin & Gregory 2009, 102-103)

It is important to remember to define the nonfunctional technical requirements with the customer team, as often the customers think that the developers will define them themselves during the development (Crispin & Gregory 2009, 103). Not all the aspects of nonfunctional testing are necessary for every product or story, however, it is a good idea to have a checklist and go through all of them with the customer team, so nothing relevant gets forgotten. Risk areas should be addressed in the test plan, including the tools and resources needed to fulfill the testing (Crispin & Gregory 2009, 218).

The usage of tools is encouraged and often mandatory for technology-facing tests that critique the product. Automation has to be used for load and performance testing, and tools should also be used to help create test data, used to run security testing, set up scenarios for manual testing, and make sense of the results. (Crispin & Gregory 2009, 103)

This quadrant needs more specialists than any other of the quadrants. While all members should be able to work cross-functionally in an agile team, nonfunctional features often need specialization and cannot be done with expertise by everyone. If the team does not have anyone who can reliably create and run these tests, it is wise to get outside help and call in experts from other sections of the company or outside of it (Crispin & Gregory 2009, 103). One of these critical fields that needs a specialist is the security and how to block hackers from exploiting the software (Crispin & Gregory 2009, 220). Programmers can change unit tests into performance tests by using a multi-threaded engine (Crispin & Gregory 2009, 102) and most of the other tests can be done by the programmers and testers collaborating and with the use of suitable tools. A team with a diverse skillset will work more efficiently and will not have to lean on outside consultants, but with or without them, your team has to make sure at least all of the minimal testing has been done for the software. The results of running these tests will help the testing in the next iteration in a form of improvements, bugs and problems will be caught early on and new stories and tasks can be written. (Crispin & Gregory 2009, 220-221)

The quality of the software will be better if the nonfunctional requirements have been defined in the beginning before the coding starts and it will be built with those in mind, as things like security and response time for example will affect critically the user's safety and willingness to use a software. The tests on this quadrant should be done at every step of the development and not be left until the end (Crispin & Gregory 2009, 103). The earlier the problems are identified, the cheaper it will be to fix them. Some tests cannot be run until some functionality or even the application has been built, such as some of the performance testing. If the technical requirement is an important feature, you should figure out ways to test it in small chunks and add more to the tests

when the function has been built up more (Crispin & Gregory 2009, 222). All of the test techniques explained in the quadrants should be used to build up the confidence of the software and let you know when the features have met the customer's requirements. Not every story will need every single type of testing, but it should be thought through which ones are necessary. (Crispin & Gregory 2009, 104-105)

"Ility" Testing

The "ility" tests include qualities such as security, maintainability, interoperability, compatibility, reliability, installability, configurability, auditability, portability, robustness and extensibility. The most important "ilities" are discussed further in detail as follows.

Security

The methods of appraising the security aspects of the software are done in a similar matter to the other “ility”-tests, so it will be included in this segment. (Crispin & Gregory 2009, 223) Security should be a major concern for most businesses, one security breach can be enough to ruin a company depending on its scale and damage (Crispin & Gregory 2009, 226). The level of effort put into security should be evaluated from which type of product we are making. If there are user accounts, interactions between users or money involved, it is important to make sure the authentication confirms each user's identity, authorization to access certain features is checked, and the software cannot be intruded from outside. (Crispin & Gregory 2009, 223-224)

Automated tests should be used to help with verification, static analysis tools can be used to analyze the code without running it to verify the code is written correctly and to detect potential flaws. Dynamic analysis tools can be used to test SQL-injections and cross-site scripting. Risk-based exploratory testing should be done by a tester specialized in security, as they can recognize attack patterns, abuse or misuse cases and architectural risks, focusing on weaknesses that a machine would not recognize. (Crispin & Gregory 2009, 225)

Maintainability

Maintainability can be achieved by making sure all of the team follows certain standards for writing the code, test frameworks and writing the tests. A good idea for the team is to create their own standards and guidelines which to follow, because people are more willing to use them if they have created the rules themselves, not to mention they are easier to memorize that way. Pair programming is also a good way in agile for reviewing the code and to notice any discrepancies from standards. Standards support the shared code ownership by making the code easily understandable by any programmer and following the same style, so anyone can pick it up and start developing it further. (Crispin & Gregory 2009, 227)

The standards that will make the code easier to maintain should not be complex and difficult to follow; one of the simplest rules that will make maintainability much better is naming conventions for methods and tests. Crispin & Gregory (2009, 227) also give the following examples for maintainability of the code: "Success is always zero and failure must be a negative value", "Each class or module should have only one single responsibility", "All functions must be single entry, single exit", "Use names for all GUI objects rather than defaulting to computer assigned identifier" or "You cannot have two fields with the same name on a page". Standards for the GUI will help testers be certain that it does what it is supposed to and automating tests will be easier. The database needs also to be easily maintained regularly so it does not get cluttered with old useless information. (Crispin & Gregory 2009, 228)

Interoperability

With interoperability is ascertained that the necessary systems and organizations can work together and share information. When testing interoperability can be found out if the end-to-end functionality of two or multiple communicating systems are working as intended. (Crispin & Gregory 2009, 228)

Interoperability testing can be done at the end of each iteration when a new deployment is up and running. It might not be possible to test the software's full functionality until the end if the system is supposed to work together with external systems. To create a similar setup in test environment stubs and drivers need to be used to simulate the behavior of other systems or equipment. (Crispin & Gregory 2009, 229)

Compatibility

The amount of effort you need to put into compatibility depends on what type of project is made and for what audience. Web applications need to work on multiple different browsers as they are used by people all around the world, however, with business applications you can state the rules of what browser for example the employees need to use. With software you need to also take into consideration which operating systems can run them. If using third-party applications you need to make sure of their compatibility and interoperability. When automating tests, it is wise to use a program that can run the same script on different browsers, operating systems and hardware. (Crispin & Gregory 2009, 229-230)

Reliability

Reliability testing gives confidence that the software works as intended in routine and unexpected circumstances. Reliability is measured keeping statistics of the average or mean time to failure or malfunction and the mean time between failures, to find out how long the software can run before it breaks. To test reliability automated tests need to be run multiple times; these include unit and acceptance tests. Crispin & Gregory (2009, 231) give an example of how stories can be written for reliability tests: "Functionality X must perform 10000 operations in a 24-hour period for a minimum of 3 days." Make tests that depict how the program will be used daily and in a long term and specific tests to demonstrate that the customer's requirements are met even in peak times. (Crispin & Gregory 2009, 230-231)

Installability

Installability should be automated as much as possible for, builds to be easily deployed into test environments on daily basis. Continuous integration plays a large part in this, meaning that the build is testable at all times. The amount of testing depends on the risks, however, at least the minimum amount should be performed to build confidence that the build can be deployed successfully in multiple different environments, including the customer site. (Crispin & Gregory 2009, 231-232)

Testing the Performance

To evaluate the overall performance of the program in real life conditions we need to do technology focused testing for performance, load, stress, scalability and for memory problems. Like "ilities", these often need specialized skills, however, the minimum can usually be performed by any agile team. (Crispin & Gregory 2009, 233)

Scalability

Scalability means that the program can remain reliable when more users are introduced by being able to scale up to growing user base. Often scalability problems cannot be resolved by fixing the application code. The network and hardware are more likely problems which the agile team cannot themselves fix, so we need to go outside the team for a solution with scalability concerns. (Crispin & Gregory 2009, 233-234)

Performance and Load Testing

How the software should work in a daily normal situation needs to be addressed when figuring out the requirements of the software; without these its performance cannot be tested reliably. How can we know what is an acceptable response time and how many users the software should be able to handle concurrently if they have not been defined anywhere? With performance testing these technical properties that might cause bottlenecks in the software are looked into, confirm that the technical requirements and goals

are upheld, and a baseline should be created which we compare to the new versions of the software to see if the performance is starting to drop for example in response times.

Load testing is used to assess the system's behavior when many people are using it at the same time, and stress testing takes it even further by adding higher than normal loads on to the system for testing its robustness. The system has to be able to scale for more users in the future and its critical characteristics will be evaluated during this testing, such as response and load times. (Crispin & Gregory 2009, 234-235)

Test Environments

Before the software is handed to the customer, final performance tests need to be run in an environment similar to the production environment, which means similar hardware, operating systems and other possibly related equipment. Often teams will use weaker machines and make generalized conclusions if the performance is good enough for the customers' needs. These tests will help the customer to make a decision for accepting the software. (Crispin & Gregory 2009, 237)

Memory Management

Usually memory usage is described in amounts (mostly minimum or maximum) which means how much memory is being used, for example RAM, ROM and hard drive. Attention should be given to memory usage and the developers should watch out for memory leaks because these can have critical consequences when software in production is actively used. Some programming languages are more vulnerable to memory issues than others. Identifying their strengths and weaknesses help with predicting possibly occurring issues. Memory management testing can be combined with performance, load and stress testing. When the development team is working on some specific story it is worth asking programmers whether they can expect some kinds of memory issues. Specific testing is possible when the possible danger zone is acknowledged. Studying memory management

issues that might affect software can be beneficial. By learning about these issues, tests can be designed to ensure that those problem will not occur. (Crispin & Gregory 2009, 237-239)

6.5 Testing Strategy

Testing strategy describes general testing methodology inside an organization. This includes the way how testing is used in controlling product and project risks, how testing levels are divided, and testing based higher level tasks. Organizations may have different strategies for different situations such as phases on software development life cycle. Testing strategy and processes it defines must be equivalent with testing policy. It must describe entry and exit criteria for one or more project. (ISTQB 2012, 32)

A traditional testing strategy document might be strange for agile development since the key value is preferring working software over documents. In some cases traditional document would not necessarily create mutual understanding of testing strategy. This is because the creators of the testing strategy and probably the testers might be only ones actually reading the document. Despite that, testing strategy plays a valuable role in quality assurance. Automatized unit tests narrow the old gap between programmer and tester. It is important to make sure that everybody understands what has been tested in order to minimize the risks and achieve the necessary quality. Other key point is that testing strategy can help with determining quality criteria for functional and nonfunctional requirements. (Smith, 2011)

In a nutshell, agile test strategy provides guidelines for the agile teams and a structure for the development team to follow. Test strategy often contains some mission statement for example business goals or objectives. Ghahrai (2015) explains test strategy content by using agile testing quadrants (See Figure 6). Basically for each testing quadrant the testing strategy answers following questions: Why, who, what, when, where and how. Short example answers for the questions could be for unit testing: To ensure quality, developers, all the new or refactored code, immediately after code is written,

locally and automation. Besides the quadrants, it is also important to remember agile development tasks and their impact on testing strategy. Following items should be noted in the strategy in order to gain complete understanding of how strategy can be implemented for each aspect of development: Product backlog, acceptance criteria, story workshops/sprint planning, development, developer testing, acceptance tests, nonfunctional tests, regression testing, UAT and exploratory testing and done criteria. (Ghahrai, 2015)

6.6 Testing Policy

Testing policy is used to describe why testing is being performed inside an organization. It defines general testing goals which the organization wishes to achieve. Composing testing policy is a task that should be attended by test managers and stakeholders. In some cases testing policy may supplement or be a part of larger quality policy. The quality policy describes usual values and goals directed on quality by management (ISTQB 2012, 31-32). Implementing testing policy can be used to ensure that strategic value is maximized for every project (Ghahrai 2009).

According to ISTQB (2012) a written testing policy should be a short, high level document which summarizes the following items:

Summarizing benefits gained from testing. Determining why testing is being done in general. When understanding why testing is required, it is possible to specify what testing goals an organization has (Ghahrai 2009).

Determining testing goals, such as ensuring reliability for software, discovering defects and reducing risks. What kind of quality criteria and what kinds of exit criteria exist (Ghahrai 2009).

Description of how testing can be evaluated. How test results can be evaluated? How can test effectivity be ensured in project? (Ghahrai 2009)

Description how typical testing process. A clear vision towards testing process must be built and different subtasks and phases accounted. What kind of roles are required? What kind of documents apply or are required? (Ghahrai 2009)

Determining how organization strives to improve testing processes. How often and what ways or tools are used to evaluate usability of current processes. What kind of elements or techniques require improvement or changing? (Ghahrai 2009)

Testing policy on the whole should also address testing tasks focusing on the development as well as on the maintenance. The policy may also refer to internal or external standards considering test deliverables and terminology (ISTQB 2012, 31-32). Ghahrai (2009) describes testing policy as a document, sitting on the throne of all testing documents.

6.7 Automation

As emphasized in the agile testing quadrants, automation is the cornerstone of agile testing and there are numerous reasons to automate.

The logical reason for automation is that manual testing is just too time consuming. As the software keeps growing over the time, naturally the testing time increases. When in agile development a new software version is produced during each iteration and a team has a deployable software after each working day, the automated regression set is a practice that cannot be overlooked. If the regression testing is executed manually the amount of time required for perform regression increases exponentially leading to the point where manual regression testing is impossible. The only way to solve this is to automate or hire more testers and make developers test. Either way, the technical debt will keep growing and frustration flourish. Testing scenarios can take plenty of time since setting up the data for complex situations can feel like an impenetrable obstacle. (Crispin & Gregory 2009, 258-259)

Manual processes expose testing to errors. Manual testing tends to become repetitive after a while which can make it boring. Even the most thorough testers can make mistakes during tiresome task and simple bugs might go unnoticed. When a team has a tight schedule and the deadline is approaching, it is easy to skip some phases or tasks. Automated builds make the development process more consistent and play a large role in preventing risks. (Crispin & Gregory 2009, 259)

Automated unit and regression tests free people to focus on more interesting things, exploratory testing for example. By automating the exploratory testing setup enables even more time to be spent on focusing potentially vulnerable parts of the system. Not having to spend time on numbing tasks, there is energy to actually think through different scenarios and learn more about the software in question. (Crispin & Gregory 2009, 259)

Having a regression suite that covers the code on a necessary level provides confidence towards software. Of course, a possible change in the software may lead to an unexpected defect; however, unit level tests notice it in few minutes and on a deeper functional level in few hours. Unexpected failures in automated tests provide feedback rapidly which makes troubleshooting easier because the programmer still has a fresh memory of code instead of finding out about the bug after some weeks. Without having the safety net of automated tests, the programmers may easily start viewing testers as their own safety net; however, this just adds weight for testing tasks. Without an automated regression suite, the development team can hardly ever make any changes in the code without worrying about something breaking. (Crispin & Gregory 2009, 261-262)

Automated tests act as documentation for software and they describe how a system should work. However, with automated test suite providing feedback of passes and failures there is no room for argument. While in agile development there is no emphasis on wide range of documentation, the tests and examples guide the development. It might be difficult to keep static documentation up to date but when an automated test suite comes out clean, there is evidence that

the build process is working and the code works. Evidently the end goal is to return the investment and pay back the contribution. Automation enables to have a certain consistency in development and people can try different testing methods or really take a shot on the software's limits. Lastly the payback that automation provides is the way how defects and failures are approached. Without automation the attitude could go towards fixing the most critical bug daily and ignoring the root cause of the bug. Running the automated suite in a programmer's own environment and recognizing bugs right away enables a programmer to redesign code accordingly before passing it on. This is how technical debt is kept low and profit high. (Crispin & Gregory 2009, 263-264)

Obstacles for Automation

Learning testing automation can be difficult, especially when learning how to do it in a way that provides a maximal benefit. The team has to choose the tools and frameworks for testing, which means that they have to invest plenty of time on making the right decisions. Needless to say that the return of investment does not happen right away. Ability to design tests makes a huge difference on how automation pays off since the amount of required research is far smaller. In addition, inexperienced development teams with insufficient training and skills might find the benefits of automation lesser than the perks and decide it is not worth of their time. (Crispin & Gregory 2009, 268). Adapting to automation can be described well with a term "Hump of Pain" (see Figure 9). The term describes the battle of beginning to automate when development team has to face the most difficulties. Without good support and encouragement, there is a high probability that the transition to new practices will be unsuccessful. Getting the traction on automation may feel almost impossible, especially when working on poorly designed legacy code. Usually it is easier to adapt automation when starting off fresh with new code and with design that is paying attention on testability. When a development team has to refactor legacy code, even the unit level automation can be an overwhelming task when the code is not designed for testability (Crispin & Gregory 2009, 269). Once the automation becomes a part of natural development process,

there is a clear sign that the hump can be overcome. (Crispin & Gregory 2009, 266-267)

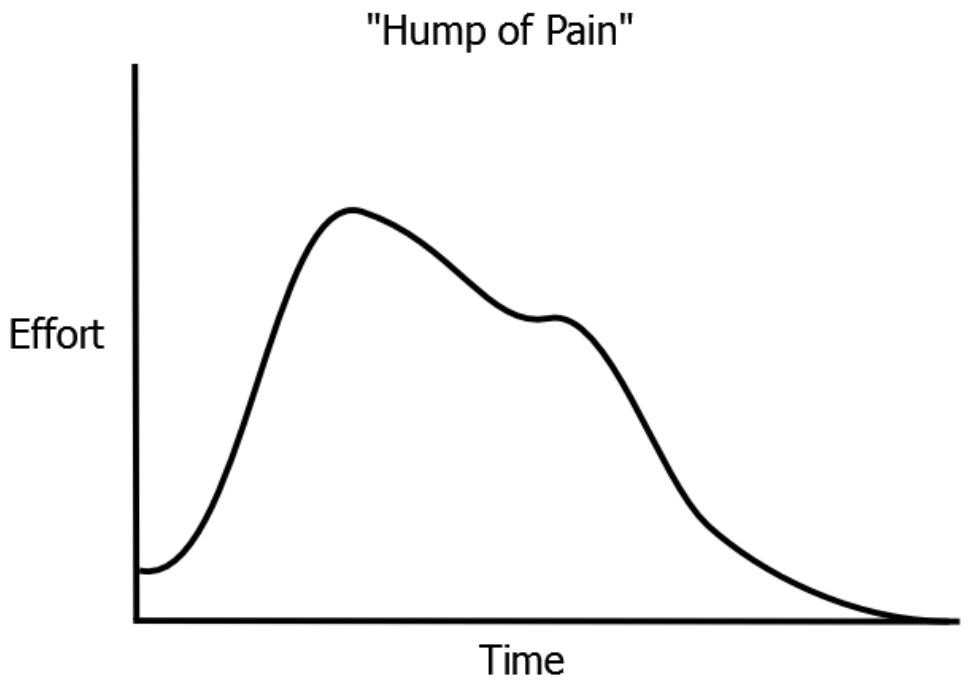

Figure 9 Hump of Pain (Crispin & Gregory 2009, 266)

Automation might be difficult if the development team does not achieve the wanted design with business logic, database and UI. In these cases it could be difficult to keep up with automation since the tests require changing just the same as the design. In agile development the code is often changing; however, the purpose of the code, the intent rarely changes. To tackle an excessive amount of test refactoring, the tests should be organized by the intent, rather than implementation. Programmers might be inexperienced in writing automated tests despite programming background and testers might not have strong programming skills and they lack confidence to build automation. Being scared of automation is only natural but eventually it is a team problem and everybody has a role to play. Unexpected problems during development are dangerous foes for automation. When stress starts to build and people panic they tend to fall back to old habits even if they were terrible to begin with. A quick decision might be that it is better to do just as much manual testing as possible and hope for the best. However, this may be the

way towards damnation. Some testing can be done, however, lacking the important factors of agile testing can lead to overlooking a bug which may have a huge impact on business. And because there were no finished automation tasks, they will follow to next iteration and reduce the business value that a team can deliver. As the iterations pass, the situation keeps on escalating. (Crispin & Gregory 2009, 269-270)

6.8 Metrics

Metrics divide opinions and they raise plenty of conversation. They can be a waste of time, money and energy and often numbers for the sake of numbers. Metrics can be exploited terribly, however, they do not necessarily have to be an ill thing. They can be used to guide the development team towards the right direction in order to achieve mutual goals. (Crispin & Gregory 2009, 74)

Lean development constantly seeks ways to decrease the amount of metrics and instead invest in effectivity for measurements, striving for correct methods. For example, metrics of how long does it take to build a working software from “concept to cash”. This is used to measure the cycle time which means producing working software continuously and reliably. Furthermore, this measurement may help with improving processes attached to cycle and strive for shorter cycle time. When using metrics that concern the whole development team instead of individuals or small groups, metrics are more likely to provide value. They can also encourage internal collaboration for making improvements. For example by measuring the time spent on fixing defects, the team can be encouraged to address the issue of how that time could be abbreviated. (Crispin & Gregory 2009, 74-75)

Why do we need Metrics?

There are good reasons for collecting and tracking metrics, however, also some extremely poor ones. Anybody could follow some team member’s personal performance by using metrics. However, progress cannot be followed without them. Metrics are well worth using when we wield them to let

the development team know when they are off the track or providing positive feedback when they are facing a correct direction. It can be measured whether the amount of unit tests increases daily or why code coverage dropped suddenly. Metrics can help us with discovering problems such as decrease in code coverage, however, perhaps the coverage fell because we got rid of unused code that was covered by tests. Metrics that follow milestones during the product's development cycle are very useful for the development team. If the goal is to increase code coverage percentually will the coverage be observed more carefully during the development? Probably. It is important to focus on mutual goals instead of stalking individuals. (Crispin & Gregory 2009, 75)

Metrics can be an effective way of tracking the progress of each iteration. For example using burn down chart is an effective method for progress feedback. If the chart is advancing upwards instead going down there is a fair reason to stop and think what is being done wrong. Perhaps the team did not understand the story completely. When metrics are being used, there is importance in understanding accurately, what the problem is or what things to improve. When understanding the problem correctly, a goal can be placed and action be taken to achieve the goal. For example, by setting a goal that the system has to have response time of X seconds when it includes X amount of people there is a clear goal at hand. When again we just say that response time has to be short, there is nothing clear to measure. When the goals are measurable the details and information to track the metrics with are often obvious. (Crispin & Gregory 2009, 76)

It is important to remember that if metrics do not provide extra value for the team or help in understanding progress towards goals, wrong metrics may be in use. The main thing is to focus on goals instead of metrics and use metrics when they provide actual benefits. If the amount of defects has increased, it will not necessarily mean that quality of code has decreased but maybe the testing has improved. (Crispin & Gregory 2009, 76)

7 Research Results

7.1 Current State of Testing at Descom

The information of the current status of the testing process was gathered by interviews, questionnaires (see Appendix 1. Webropol questionnaire for testers) and by mingling with the people and conversing about the status of testing and the processes.

The main project management models were Scrum and different variations such as mixture of Kanban and Scrum. All the projects are continuing to emphasize benefits of agile development and customers have also beginning to understand the perks of agile.

The way testing was implemented in different projects varied a lot. In some projects testing was only just beginning while others had already taken it to quite thorough level. One highly testing focused project had found the necessary resources for testing and customers were being supportive towards them. The way testing could have been implemented more effectively for every project would require planning and good internal communication. One solution could have been increasing activity in testing community, which could spread the word about good testing practices.

Some projects were writing test plans whereas others do not. However this was mostly dependable of the customer. This meant that plans were being made if customer had requested it, which was not the case usually. Also in the most projects an official testing process had not been determined. In some cases specifying was started but it was never quite finished. During the thesis most conversations were had with the testing specialists whose projects had had some kind of processes. The lack of documenting seemed troublesome but there was not actual reason to force some practices into projects if they and the customers did not see the need for those documents. However, a test plan is a document that could radically increase quality of testing and play an important role in quality assurance.

Test levels were implemented differently between projects. It has to be remembered that when building an e-commerce system on an existing platform, there is not much need for unit testing, because the assumption is that complete platform will work as expected on unit level. However unit testing could be done in theory but it had not been yet seen to be necessary. Almost in all the cases, there was a need for increasing regression testing, especially in automation. In most projects regression was done almost completely manually, which drained excessively large amount of time and the required time only keeps exponentially growing as more regressions tests are added for new functionalities. In many cases acceptance testing and user acceptance testing had been mistaken with each other and eventually we agreed a statement of how both of the testing levels differ. UAT was done quite often by customers, some putting more effort in it than others, however, internal acceptance testing was not done almost at all. It is clear that there is not just one way that would work for all the projects in acceptance testing.

In the beginning there was discussion about trying to pick testing tools for all the projects. This goal was abandoned after research because projects are already running and it would be difficult to change during the project and especially because tools divide opinions. After researching and discussing the matter, the decision was made that each project got to pick whatever tools they wanted and were available, however, preferred tools would be written down on the high level documents.

Automation had existed on certain level and the amount differed a lot in the projects. There had been a lot of requests for increasing automation. The important aspect to be noted in automation is that it has to be implemented all the way from the beginning of the project. The more the effort that is put towards automation in the beginning will result in the far cheaper product and smaller technical dept. The automation had often been started in the projects but usually it had not been finished due to lack of resources or time.

Most of the customers felt that testing is important part of software development although sometimes it was still viewed as the last link before

releasing complete product for customer. Most customers however understood the increasing importance of testing.

Neglecting testing had almost always been caused by too tight schedule or poor planning alongside with lack of resources. According to Descom testing staff, increased resources were necessary for almost every project. Only some persons found resources to be sufficient partially because they had hired an external testing team to fight internal shortage of testing. Also the resources should be backed up somehow meaning that one person should not be solely responsible for one area so that nobody else knows anything about it.

Reporting had been diminishingly small and in many projects there is no testing reporting at all. According to staff, the increase in reporting is required especially from bugs and coverage point of view. Agile can be sometimes an excuse to stop all the documentation, however nobody will be able to remember every single thing so some documentation would be important. Due to lack of documentation, the decisions were often made in chat and if people were not currently watching it they might have never heard about the matter until it popped up somewhere.

Largest issues during testing had been inaccuracy in specification documents, keeping specification up to date, issues concerning test data, customers' lack of interest to test system before it is complete, resourcing and know-how.

7.2 Process Improvement

In the beginning of the thesis process a clear vision was set by Descom. The idea was to create high level and agile processes that do not interfere with the everyday practices of different projects. The purpose of the processes was to provide guidelines and value for projects rather than trying to change existing practices. For the further agile improvement of the new processes, CTP was the only logical choice. Descom is using a Management 3.0 –model in their company, which means, for example that they do not have a dedicated testing

manager. Therefore a heavyweight process improvement model such as TMMi would not be suitable as testing community does not have the resources to invest enough. By using CTP, the Management 3.0 will encourage testing staff for process improvement without taking too much time from their daily work.

The improvement model had to be lightweight and non-restrictive to work well in an agile environment, as well as being implemented incrementally over time. Implementing the process improvement method requires great effort from especially the testing people inside the organization and CTP focuses only on the most important testing processes that need to work well for testing to be efficient and effective.

Improving the process with CTP can be as simple as making a checklist of the required aspects that should be carried out, which can be seen in the tables through Table 2 CTP activities in the Plan-step (Adapted from Bath & Veenendaal 2014) to Table 5. When one aspect has been filled, it can be checked out, however, as the improvement process isn't linear and doesn't happen in an instance, returning and improving further all aspects can be done in the next iterations. CTP-checklist document can be found in the appendices with instructions of how it should be used inside Descom.

7.3 Testing Policy

The testing policy document describes why testing is performed inside organization and overview how to do it. It also goes through testing process and the aspects that are linked to it. Policy document (see Appendix 3. Testing Policy) is customized after TestingExcellence-website and thought that Rex Black has on the policy.

Content:

1. Summarizing benefits gained from testing. Determining why testing is being done in general.
2. Determining testing goals, such as ensuring reliability for software, discovering defects and reducing risks.
3. Description of how testing can be evaluated. How test results can be evaluated? How can test effectivity be ensured in project?
4. Description how typical testing process. A clear vision towards testing process must be built and different subtasks and phases accounted.
5. Determining how organization strives to improve testing processes.

7.4 Testing Strategy

The testing strategy (Appendix 4. Testing Strategy) was constructed in a way that will scale to different projects. The document is customized after strategy contents introduced by TestingExcellence-website and ISQTB materials. The produced strategy is a high level document that can be used as guidelines for planning testing to individual projects. Emphasis was on the agile development where traditional testing strategy is uncommon. Therefore high level document will work better when it does not limit the options on a project. The strategy has included introduction for some parts which are dependable on the project that strategy will be applied on. These are the parts that can either be filled in or removed.

Content:

1. Purpose
2. Guiding agile principles
3. Quality attributes

4. Test approach
5. Test environment
6. Test tools
7. Test execution
8. Test data management
9. Defect management

8 Conclusions and Discussion

The goals that were set when thesis process started heavily focused on researching the current state of testing and using existing processes and methods for standardizing testing in general at Descom. The scope of the thesis grew almost too wide and the focus was difficult narrow down. This decision was made consciously, because it was necessary to address multiple subjects in theory, which were closely linked to final product.

It became clear quite quickly that existing processes will not be sufficient alone and agile practices should be emphasized more. The consensus was that there necessarily was no need for refining small details, rather than drawing high level guidelines for future.

Some details of the final products were gathered from projects; however, most of them were theoretical models adjusted to the current state and future of testing. The high level documents and process models were quite successful and the received feedback was positive. The feedback was mainly received from the assigned support team. Surprisingly rest of the testing community in Descom hardly commented on the products at all during the progress or after the products were presented. The feedback from Descom community was requested several times, however, eventually almost all of the co-operation happened with three projects.

Perhaps the processes will not be one only truth or fit every project without customizing, nevertheless, building processes for every situation would have been an impossible task to address. The differences in terminology or the way testing techniques were implemented got solved during meetings and the process was built in a way that it can scale for most testing types that were used inside organization.

Some limitations were met especially in scheduling meetings and in some occasions the process was just waiting for next meeting to receive feedback in order to make necessary corrections. It could have been useful to pilot

processes in some project, however, that was a task for future. The information was gathered sufficiently by interviews and a questionnaire from almost every project that had included testing. However, the interviews did not have a fixed setup, although questions were quite similar in every interview. Handwritten notes of the results were made but it would have been better to record the interviews and transcribe them afterwards for more validity. However, the iterative and light way of refining final products seemed to be more flexible and result in faster improvement.

The process improvement guidelines were built in a way that Descom testing community can with some effort improve the state of testing further and especially achieve the long desired internal communication between the projects. There is not one correct solution for every situation especially remembering the range of Descom projects. It is up to the testing community to question the previous practices and drive their development teams to strive for higher quality and more refined processes. Critical Testing Processes-method will suit Descom perfectly due to its flexibility which enables the testing community choose some targets for improvement and refine those for a while little by little and then move to the next target. The resources will not be wasted in banging heads on a wall but process will improve from the critical points naturally instead of trying to force the change.

Descom already had documents for test plan and rest reporting, but these documents had been deemed too heavy and not many projects had used them. Upon inspecting the documents it did not seem like there was anything unnecessary in the content, so the authors decided to modify them only a little and fill out some parts that are universal for most projects.

New high level documents that provide guidelines for all testing on the company level were written in the form of testing policy and testing strategy. Writing the documents was challenging, because theory does not give you a straight example how they should be formed and neither of the authors of the thesis had any prior working life experience of testing and had never seen such documents made for agile development. The old heavy development

model testing documents were inspected as well as some documents for agile testing, of which it was decided what would be necessary and useful for the company. In the future, further improvement rests on the company’s testing community and project groups.

Sources

Aaltio, T. 2013. Test Process Improvement with TPI NEXT - what the model does not tell you but you should know. Accessed on 29.7.2015. Retrieved from http://www.slideshare.net/VLDCORP/test-process-improvement-with-tpi-next-what-the-model-does-not-tell-you-but-you-should-know

Bath, G & Veenendaal, E. van. 2014. Implementing Improvement and Change - A Study Guide for the ISTQB Expert Level Module. Sebastopol: O'Reilly Media. Accessed on 21.8.2015. Retrieved from https://books.google.fi/books?id=oT-4BAAAQBAJ&printsec=frontcover&hl=fi

Black, R. 2003. Critical Testing Processes: An Open Source, Business Driven Framework for Improving the Testing Process. Accessed on 3.8.2015. Retrieved from http://www.rbcs-us.com/images/documents/critical%20testing%20processes.pdf

Black, R. 2003. Critical Testing Processes - 12 Testing Processes and Why They Matter. Accessed on 3.8.2015. Retrieved from http://store.rbcs-us.com/images/documents/Critical-Testing-Processes.pdf

Crispin, L. & Gregory, J. 2009. Agile testing: a practical guide for testers and agile teams. Boston: Pearson Education.

Craig, R & Jaskiel, S. 2002. Systematic Software Testing. Artech House Publishers: Boston. Accessed on 3.8.2015. Retrieved from http://library.books24x7.com.ezproxy.jamk.fi:2048/toc.aspx?bookid=6411

Descom. 2015. Accessed 12.8.2015. Retrieved from https://www.descom.fi/en/

Ghahrai, A. 2009. Test Policy Document. Accessed on 23.7.2015. Retrieved from http://www.testingexcellence.com/test-policy-document/

Ghahrai, A. 2015. Agile Test Strategy Example Template. Accessed on 23.7.2015. Retrieved from http://www.testingexcellence.com/agile-test-strategy-example-template/

Graham, D., Veenendaal, E. van, Evans, I., Black, R. 2007. Foundations of Software Testing—ISTQB Certification. London: Thomson Learning. Accessed on 15.6.2015. Retrieved from http://library.books24x7.com/toc.aspx?bookid=26179

Hambling, B., Morgan. P., Samaroo, A., Thompson, G. & Williams, P. 2010. Software Testing: An ISTQB–ISEB Foundation Guide. 2. edition. United Kingdom: British Computer Society. Accessed on 15.6.2015. Retrieved from http://library.books24x7.com/toc.aspx?bookid=38031

Hass, A. M. 2008. Guide to Advanced Software Testing. England: Artech House. Accessed on 22.6.2015. Retrieved from http://library.books24x7.com/toc.aspx?bookid=27207

Highsmith, J. 2001. History: The Agile Manifesto. Accessed 10.6.2015. Retrieved from http://agilemanifesto.org/history.html

International Software Testing Qualifications Board. 2012. Advanced Level Syllabus Test Manager. Accessed on 3.7.2015 Retrieved from http://www.istqb.org/downloads/finish/46/96.html

International Software Testing Qualifications Board. 2012. Sertifioitu Testaaja. Jatkotason sertifikaattisisältö Testauspäällikkö. Accessed 3.7.2015 Retrieved from http://www.fistb.fi/sites/fistb.ttlry.mearra.com/files/Advanced%20Syllabus%202012%20-%20TM%20Final.pdf

Leankit. N.d. What is a Kanban Board? Accessed on 20.8.2015. Retrieved from http://leankit.com/kanban/kanban-board/

Manifesto for Agile Software Development. 2001. Accessed 10.6.2015. Retrieved from http://agilemanifesto.org/

Mountain Goat Software. 2005. Scrum Overview in Agile Software Development. Accessed on 20.8.2015. Retrieved from https://www.mountaingoatsoftware.com/agile/scrum/overview

N4S-program. 2014. N4S-Program: Finnish Software Companies Speeding Digital Economy. Accessed on 5.8.2015. Retrieved from http://www.n4s.fi/en/

Novák, I. 2011. Beginning Visual Studio LightSwitch Development. Indiana: Wiley Publishing, Inc. Accessed on 25.6.2015. Retrieved from http://library.books24x7.com/toc.aspx?bookid=44286

Pham, A. T. & Pham, D. K. 2013. Business-Driven IT-Wide Agile (Scrum) and Kanban (Lean) Implementation: An Action Guide for Business and IT Leaders. Boca Raton: CRC Press. Accessed on 27.6.2015. Retrieved from http://library.books24x7.com./toc.aspx?bookid=51916

Schwaber, K. & Beedle, M. 2001. Agile Software Development with Scrum. New Jersey: Prentice Hall

Shore, J. & Warden, S. 2008. The Art of Agile Development. Sebastopol: O'Reilly Media.

Smith, A. 2011. Agile Test Strategy Template. Accessed on 3.8.2015. Retrieved from http://ennova.com.au/blog/2011/05/agile-test-strategy

Stephens, M. & Rosenberg, D. 2003. Extreme Programming Refactored: The Case Against XP. Berkeley: Apress. Accessed on 26.6.2015. Retrieved from http://library.books24x7.com/toc.aspx?bookid=8227

Stober, T. & Hansmann, U. 2010. Agile Software Development: Best Practices for Large Software Development Projects. New York: Springer-Verlag Berlin Heidelberg. Accessed on 22.6.2015. Retrieved from http://library.books24x7.com/toc.aspx?bookid=36069

TMMi Foundation. 2002. Test Maturity Model Integration (TMMi) Accessed on 28.7.2015. Retrieved from http://www.tmmi.org/pdf/TMMi.Framework.pdf

Watkins, J. 2009. Agile Testing: How to Succeed in an Extreme Testing Environment. England: Cambridge University Press. Accessed on 15.6.2015. Retrieved from http://library.books24x7.com/toc.aspx?bookid=32497

Watkins, J. & Mills, S. 2011. Testing IT: An Off-the-Shelf Software Testing Process. 2. edition. England: Cambridge University Press. Accessed on 24.6.2015. Retrieved from http://library.books24x7.com/toc.aspx?bookid=41437

Appendices

Removed from public release because of security classification.