WWW.JAMK.FI

Deciphering and Analysis of FreeNest Product Platform's testprocess

Tero Meuronen

Thesis
October 2013

Software Engineering
The School of Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) MEURONEN, Tero	Julkaisun laji Opinnäytetyö	Päivämäärä 02.10.2013
	Sivumäärä 91	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty (X)
Työn nimi FREENEST PROJECT PLATFORMIN TESTAUSPROSESSIN TULKINTA JA ANALYYSI		
Koulutusohjelma Ohjelmistotekniikka		
Työn ohjaaja(t) PIETIKÄINEN, Kalevi		
Toimeksiantaja(t) RINTAMÄKI, Marko		

Tiivistelmä

Työssä oli tarkoituksena tutkia käytäntöön tuotu testausprosessi sellaisella kohderyhmällä, kenellä ei ollut aikaisempaa kokemusta testauksesta. Lisäksi analysoitiin, mitä pitäisi lisätä, poistaa tai muuttaa. Esille tuodaan projektiprosessin yleinen teoria ja erityispainotuksena testauksen ja siihen liittyvän prosessin teoria.

Tavoitteena toimi se, että opinnäytetyö toimii dokumenttina FreeNestin testausprosessista, ja lisäksi siihen on tuotu analysointi, mikä meni väärin prosessin aikana ja miten sitä voisi parantaa. Työ on vaadittu englanniksi johtuen kansainvälisyydestä.

Tuloksena ja hyötynä toimii siis dokumentti ja analyysi testausprosessista, joiden avulla uudet työntekijät kykenevät oppimaan käytännön työn ja teorian FreeNestin kehityksessä. Näiden avulla SkyNest-projekti kykenee parantamaan suoritustaan tulevaisuudessa ja käyttämään tulokseksi syntyneitä dokumentteja opetusmateriaalina.

Avainsanat (asiasanat)
FreeNest, SkyNest, Testaus, Tutkimus, Analyysi

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s) MEURONEN, Tero	Type of publication Bachelor´s Thesis	Date 02102013
	Pages 91	Language English
		Permission for web publication (X)

Title
DECIPHERING AND ANALYSIS OF FREENEST PRODUCT PLATFORM'S TESTPROCESS

Degree Programme
Software Engineering

Tutor(s)
PIETIKÄINEN, Kalevi

Assigned by
RINTAMÄKI, Marko

Abstract

The purpose of this thesis was to inspect testing process in practice with a target group with no previous experience in testing. Additionally it was analyzed what is to be added, removed or changed. Issues such as the overall process for project are disclosed, and special emphasis is put on testing and the process related to it.

The goal was to make the thesis work as a document for FreeNest's testing process. In addition, it was analyzed what went wrong during the project and how it could be improved. The thesis was written in English, due to the international distribution of the documentation.

The document and analysis work as a result of the thesis and benefit the new employees in how to learn about the theory and the practical work of FreeNest development. With these SkyNest can improve their performance in future, and use the resulting documents as instruction material for students.

Keywords
FreeNest, Skynest, Testing, Research, Analysis.

Miscellaneous

Table of Contents

1 Introduction 6
2 Problem domain of thesis 7
3 Basic elements of software project 8
3.1 Project models 10
3.1.1 Traditional project models 10
3.1.2 Waterfall model 11
3.1.3 Prototyping 12
3.1.4 Iterativity 13
3.1.5 Scrum 14
3.1.6 Kanban 16
3.1.7 Changing from traditional to agile 17
3.2 Project Triangle 18
3.3 Software requirement management 21
4 Testing 25
4.1 Errors, mistakes and bugs 26
4.2 Seven principles of testing 27
4.3 Fundamental Test Process 28
4.4 Test levels 31
4.5 Choice of test cases 33
4.6 Ending test criteria 34
4.6.1 Complexity metrics 35
4.6.2 Coverage metrics 35
4.6.3 Error seeding 36
4.6.4 Tool support 37
4.6.5 Test-Driven Development 38
4.6.6 Test documentation 39
4.7 Developers as testers 41

5 ALM – Application life cycle management 43
6 About FreeNest Development 48
6.1 Three factors of project triangle 49
6.1.1 Time 49
6.1.2 Scope 50
6.1.3 Cost 50
6.2 Requirement 51
6.3 Development process 51
6.4 Scenario 55
6.5 Analysis 57
6.5.1 General 57
6.5.2 Testing process 58
7 REFERENCES 62

Figure Index

Figure 1: Waterfall model 12
Figure 2: Sprint structure 17
Figure 3: Project management triangle 21
Figure 4: The V-model 34
Figure 5: Factors of testing 44
Figure 6: ALM Areas 47
Figure 7: Requirements 58
Figure 8: Testing part 1 (design) 59
Figure 9: Testing part 2 (execution) 59
Figure 10: Bug status process 60
Figure 11: Developer's ticket progress 61
Figure 12: Results of lack of communication 66
Figure 13: Example test case 72

Figure 14: Example bug report 73
Figure 15: Testlink mainpage 78
Figure 16: Project creation 79
Figure 17: Creating a test suite 81
Figure 18: Creating a test case 81
Figure 19: Test case steps and other details 82
Figure 20: Creating a test plan 83
Figure 21: Creating a platform 85
Figure 22: Adding and removing platforms from a project 85
Figure 23: Adding test cases to the test plan 86
Figure 24: Creating a build 87
Figure 25: Executing tests 88
Figure 26: Automatically created test cases from a requirement 89
Figure 27: Requirement dependencies 90
Figure 28: Test report options 91
Figure 29: HTML test report 92
Figure 30: Resource plan 94
Figure 31: FreeNest Workflow as Swimlane chart 95

Terminology

FreeNest Project management environment developed within SkyNest -project.

SkyNest Project that is the development of FreeNest, hosted within JAMK.

Tekes The Finnish Funding Agency for Technology and Innovation.

JAMK (Jyväskylän Ammattikorkeakoulu) JAMK University of Applied sciences.

IPMA International Project Management Association.

PRINCE2 Originally developed for British public enterprises to be used as a project model.

PMI Non-profit association registered in the United States which has developed project models.

TDD Test-Driven Development (see page 21).

Season Factory

A period of time during a season in which a large group of students are taken into SkyNest Project as trainee's. Usually named with the corresponding season, such as summer factory.

Definition of Done

A precise definition for a task to fulfill to be considered done.

SDLC Software development life cycle.

ALM Application life cycle management.

Specification by example

a collaborative approach to defining requirements and business-oriented functional tests for software products based on capturing and illustrating requirements using realistic examples instead of abstract statements.

1 Introduction

FreeNest is designed to be a fast to deploy and an easy to use product development platform. It combines together many open source software commonly used in software development, creating a premade toolbox of easy to use tools for different phases in software product development. FreeNest is highly customizable, and can be adjusted to the users' needs. (FreeNest, 2012)

SkyNest is the project name for FreeNest development within JAMK, University of Applied Sciences and the Information and Communication Technology department. It is funded by Tekes and its main purpose is to develop solutions for Finnish software industry in order to improve competitiveness. The personnel consist of two or three regular employees, and a group of 5-20 students that who work in the project temporarily as interns. (Projects, Jyväskylä University of Applied Sciences)

The object for this thesis is from the field of software testing, analyzing the old test process for SkyNest Project, finding flaws in it and creating a new and improved test process. This subject is very relevant to current testing procedures in software development, as this thesis will benefit SkyNest Project itself, and also will give insight of the subject for the author of this thesis. It is important to always develop everything inside software engineering, as the whole software industry has to keep up with the new technologies and techniques that it itself constantly develops.

2 Problem domain of thesis

All the theories are relevant to development of FreeNest, becouse as a product brought into existence via project, it holds all the basic key elements that are explained during all the theories presented at this thesis. Their purpose is to create a basis for the actual practical task performed, and make the terms more clear. Theoretical framework is essential to any kind of research or project.

According to Escalada (2009), theoretical framework directs the research and provides a statistical relationship between theory and practice. She also states that when conducting research, theory guides the process.

Escalada (2009) continues about the general guidelines when developing the theoretical framework as follows:

- Examine the thesis title/topic and problem.
- Consider what the key variables are.
- Inspect related literature to find source material.
- Listing variables related to the research.
- Review the social science theories.
- Discuss the assumptions of the theory and point out their relevance to research.

As this thesis is about a research, it requires theory to support the points and arguments brought by the practical period.

3 Basic elements of software project

Every software project consists of certain elements that make them more clear and precise. This is no exception in the development of FreeNest, which contains both waterfall and agile elements. The waterfall method is on the grand scale, as a time line, while agile methods, Scrum and Kanban, are used for the actual development process. In this chapter a general theory of software project and its methods discussed in detail, and they are all related to the development of FreeNest.

Production of software is usually organized into projects, in which software is simply produced according to client's requirements. In reality, software projects do not come up out of nowhere, but are based on customers' transactional needs.(Haikala I. & Mikkonen T. 2011, 19)

The start for the project from the supplier's point of view, was to map the customer's requirements. The customer's requirements describe the project content and this procedure presents the start for a software project. If the requirements are detailed enough, a project can in principle be a description of the stages from requirements into implementation. In practice describing the requirements in such a detailed way is impossible, and the requirements always get additions, adjustments and changes during the whole project. Usually the developers are satisfied with the requirements that are ”precise enough”, and attempt to be prepared to changes that appear during the project. Possible exceptions to this are the most security critical software, such as the ones used in airplanes or nuclear power plants. In these cases, the process is very controlled and detailed. (Haikala I. & Mikkonen T. 2011, 21-22)

Software development is usually divided into different departments, which may be geographically separate. For example, testing could take place in another city, made into the correct environment. This creates possibilities of using the place of testing for multiple different projects. Another example would be dividing the development into different language versions for software, making the process need less software technical know-how. This distributed software development has created global software development, which means that software related work is made more cost-efficient by moving some parts of the development to multiple continents, timezones and cultural areas. This usually means a subcontract from another company, or international corporation, which tries to optimize its work by reorganizing. In practice, this has proved to be contradictory. Sometimes the division has been a great success, but there have also been bad experiences. (Haikala I. & Mikkonen T. 2011, 23-24)

Creating software differs from many other areas of industry in the way that the results of planning and implementation are immaterial. The process of creating it is expensive; on the other hand, copying the result is nearly free of cost. The Internet has brought an additional level to this, which enables easier distribution and development of communication across the world, in principle at least. (Haikala I. & Mikkonen T. 2011, 24)

The division of the industry, immaterial results and the Internet have created the possibility for the growth of open source software. When creating open source software, there are three principles which they follow: Development model, license and community. Development model is a constant and thorough code inspection. Licenses determine the terms of use, and these terms differ greatly from the ones required from the person making the changes and the distributor. A community consists of companies, research facilities and individual developers. Usually the community is divided into volunteer and business communities. Volunteers consist of

hobbyists, while business developers are actually paid. (Haikala I. & Mikkonen T. 2011, 24-25)

3.1 Project models

Development of FreeNest is contains several project models used in its development. This part discusses the most used project models as a universal clarification, and specifies the waterfall model, Scrum and Kanban more in detail as they are all used in the development of FreeNest. Waterfall model is used on the general planning level, the time line. The Scrum and Kanban methods are used on the actual development level, depending on which team's method is being inspected. In this chapter the methods are discussed in detail, and Chapter 7 provides more detail on the practical level of FreeNest.

3.1.1 Traditional project models

Uncountable public project models have been developed, of which some are completely common, and some are specifically designed for IT-projects. The business has grown around these models: they are tailored, trainings for them are organized, and many contain a certificate process, the purpose of which is to ensure that an individual who has received a certificate knows the model thoroughly. Very common ones are IPMA, PRINCE2 and PMI -based project models (Haikala I. & Mikkonen T. 2011, 34)

3.1.2 Waterfall model

When the first major software were developed, it was noticed fast that a systematic work process was required. Progress usually followed a familiar order: Define customer needs, plan the software, test the software and put the software into use. The waterfall model itself not only generally follows this, however in addition it iterates backwards, thus creating- ”inside loops” during the project, as seen in **Figure 1**. During the history of the model, industry lacked the iteration aspect completely, and many times it was claimed that the waterfall model cannot be practically applied for a given reason. By allowing the iteration and starting the phases even before the previous phase had ended, an adaptable project model can be acquired. When inspecting the newer project models, it can be found that the structure or parts of it may be based on the waterfall model. (Haikala I. & Mikkonen T. 2011, 36-37)

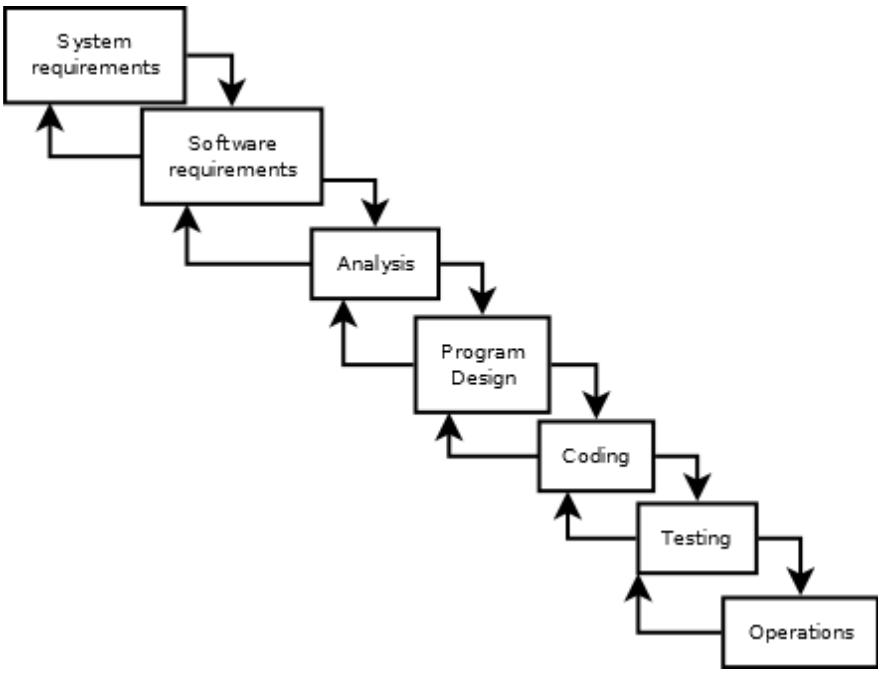

Figure 1: Waterfall model

3.1.3 Prototyping

Prototyping means in software production that in some way an incomplete prototype of the software is built for inspecting some parts of the software. Examples could be a new kind of user interface and its logical functionality, new kind of technology to implement or a more specific inspection for memory usage. Prototyping is divided into two options: evolutionary prototype and throw-away prototype. Evolutionary prototype is developed phase by phase into a fully complete product, while throw-away prototype is used to model the product, and after that the actual product is started over or with completely different tools. Naturally there are in-between versions of both prototyping, where there are some parts left to use, and some parts are to be replaced with a new implementation. (Haikala I. & Mikkonen T. 2011, 38)

Evolutionary prototypes work like actual components: they return sensible values and they strain the system like an actual component. These prototypes usually have much valuable code that can be used in the final product. Evolutionary prototypes have the problem that a badly working prototype is going to be implemented to the final product. A prototyped component might seemingly work correctly; however, it can actually be unrefined and have slower functionality. This may especially happen during the ending phases of a project, as it is a big temptation to leave such ”a minor” detail unattended. Even the testing department may miss this same thing, because it might not be noticed there as the large environment with all its incompleteness might get all the attention. This leads to the fact that the flaw will be noticed in the final product, by the customer. (Haikala I. & Mikkonen T. 2011, 38-39)

Throw-away prototypes are most commonly used in the system's user interface design. The simplest example is a drawing of a screen and then showing to the customer, what the software will look like.(Haikala I. & Mikkonen T. 2011, 39)

3.1.4 Iterativity

Repeated waterfall models have many flaws. Following is an example of creating a new version every year:

- **Efficient use of learning** – Software development is also a learning process. Waterfall models are used with lengthy projects spanning over many months or even years. When the developers have finished an older project, they might not remember what they did at the start during the next project, and the staff might have changed.
- **Complexity and unpredictability of risks** – Risks are not usually realized until in the test phase, which makes reactions to them really difficult and expensive.
- **Requirement adjustments and late customer feedback –** During a year the requirements change and are adjusted, and the system might be outdated when released.
- **Late start of testing** – Testing is started in the late part of a project, and testers have no possibility to learn the system beforehand. The typical amount of work is 20% from the total work time. In reality, this should be much more, such as 50%.
- **Self-deceit** – During Project planning, despite the quality assurances it is hard to determine if the defining documents are good enough.
- **Impatient customers** – Customers will get impatient when there are rarely not many tangible results to show about the work progress.

Iterative method has been developed to fix these flaws, by changing a cycle into much less time, such as a month. This way the requirements may be divided into iterations and at the end of every iteration the progress can be shown. The customers

could receive deliveries every six months. Iteration holds many similarities with agile methods. Shortening the product development cycle is not without a cost, however. By reducing the time on a cycle, the product will require more testing. The small cycles may also need some sort of testing automation to be applied. Additionally, by adding small pieces one by one may collapse the architecture of the final design. This places a greater emphasis onto requirement management and product management.(Haikala I. & Mikkonen T. 2011, 40-42)

3.1.5 Scrum

Scrum is an agile method which contains very short cycles, usually from two to three weeks. It contains only three roles: product owner, Scrum master and a team. The product owner is closest to the product manager, while Scrum master is the project leader and a team is the project group. (Haikala I. & Mikkonen T. 2011, 47-48)

The product owner is in charge of the project's economical result. He/She will collect system requirements for a product work list, and upkeeps the product backlog in priority order. The backlog contains items, which each have a rudimentary time estimate. These items may be product features, use cases, user stories, requirements, error reports, document development or improving the architecture. (Haikala I. & Mikkonen T. 2011, 48-49)

Scrum master is best described as “a project chief without power”. He/she is responsible for everyone following the Scrum process, and acts as a trainer for the team and the product owner. He/she is in charge that during and after a sprint the results match with the **Definition of Done** for each item and ensures that unfinished ones are not considered done. Additionally his/her responsibilities include getting rid of any impediments, watch over the welfare of the team and to get rid of

inappropriate team members. (Haikala I. & Mikkonen T. 2011, 49)

The team's optimal size is around seven, of which everyone preferably should be from different specialty backgrounds. The team will organize itself, thus an actual project manager is not required, and the team chooses its own methods of working. The team itself organizes the tasks to be done and shares the work amongst its members. The workplace should contain an assignment board visible to everyone, in which all states of tasks are made clear. (Haikala I. & Mikkonen T. 2011, 49)

In Scrum the project proceeds as sprints, which usually last two to three weeks. Every sprint starts with sprint planning meeting that lasts the whole day. This meeting is participated by the product owner, Scrum master and the team. The day starts with the product owner introducing the product backlog items to be inserted into the sprint. After choosing the items each of them is valued by playing planning poker, making everyone in the team choose a number, which represents the amount of work for the item. The poker will be played until everyone has come to an agreement of a number. After choosing the items, the team divides them into smaller tasks which can be divided amongst the members to be worked on. **Figure 2** shows the structure of the Scrum process. (Haikala I. & Mikkonen T. 2011, 49-50)

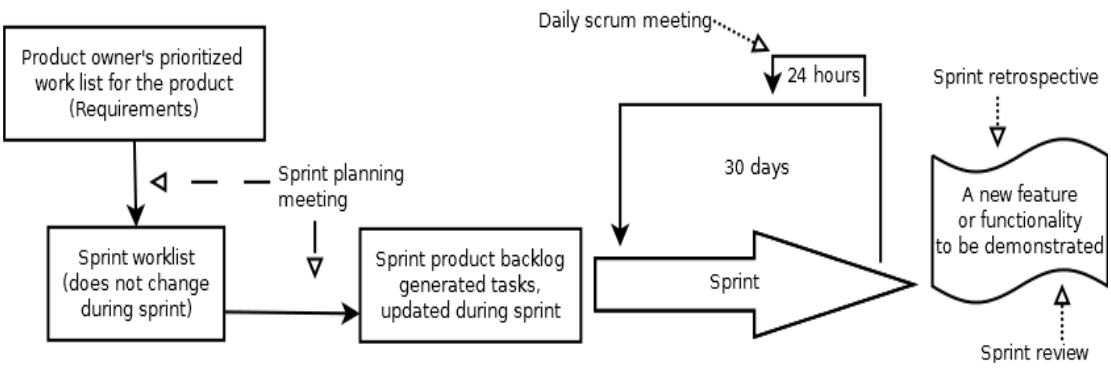

Figure 2: Sprint structure

During the sprint the list of tasks is not to be touched during the Scrum. This creates a requirement to change free time periods for the teams, which is in other models a normal workday and may include many requirement changes. During one sprint the team has a 15-minute daily scrum meeting, in which every team member answers three questions:

- What did I do after the previous meeting?
- What am I going to do next?
- Do I have any impediments?

The scrum master and the team usually take part in this meeting. The product owner may also participate, but he/she does not have any right of speech. (Haikala I. & Mikkonen T. 2011, 50-51)

After the sprint a review meeting is held, in which the team, scrum master, product owner and possibly co-team representatives participate. The team demonstrates the results of a sprint and collects the feedback. After this a sprint retrospective is held, in which the participants consist of the team, scrum master and product owner. This meeting lasts around two to three hours, and it is for evaluating the success of the sprint and thinking of improving the methods of work. Again, each participant should be ready to answer two vital questions:

- What went well?
- What should we improve in our next sprint?

(Haikala I. & Mikkonen T. 2011, 50-51)

3.1.6 Kanban

Kanban is a very simple working method, which includes three basic principles:

- Visualizing the work steps: practically this means a similar task board table to

the one used in scrum. The table contains its own columns for each work phase, such as planned, in progress, in check, and finished. Each task is in its own card and is moved from column to column according to its status in work.

- Column limit: each phase has a work limit for tasks
- Measuring the time: The development process can be optimized, if the work time in each phase is measured

The most notable difference between Kanban and Scrum is that Kanban has no sprints. A new task can always be started when a column has room for it, and does not exceed the limit, that has been established before among the team members working with Kanban. This type of work flow works especially well for a maintenance unit. The main benefit when compared to scrum is the ability to adapt immediately, whereas the scrum is based on untouched work periods. (Haikala I. & Mikkonen T. 2011, 50-51)

3.1.7 Changing from traditional to agile

The changing process from traditional waterfall model into more recent agile methods contain following important points:

- Implement **specification by example** as part of a wider process change.
- Focus on improving quality.
- Start with functional test automation.
- Introduce a tool for executable specifications.
- use test-driven development as a stepping stone.

(Adzic Gojko. 2011. 38-42)

When changing the team culture, following points are important take into consideration:

- **Avoid “agile” terminology:** Many terms within the agile software development methods are filled with terminology, such terms can be misunderstood and cause confusion
- **Ensure your management support:** Developers, testers and analysts start to get involved with each other more, which may lead to confusion. This requires management support for everyone.
- **Sell specification by example as a better way to do acceptance testing:** By bringing the idea of having realistic examples instead of abstract statements into the process, the participants will get a better idea of what is going to be done.
- **Do not make the test automation the end goal:** It leads into tests that are too technical and that are scripts rather than specifications, a common failure pattern. Long term effect is an impediment to change.
- **Do not focus on a tool:** This causes technical tests that cannot be used by business users and testers. Instead focusing on collaboration and process change are essential.
- **Keep one person on legacy scripts during migration:** Put one person in charge for distracting tasks, such as bugfixing. Meanwhile, the rest of the team may proceed without distracting impediments.
- **Track who is running and not running automated checks:** By tracking this in some way, management can focus on people who have problems and impediments on their work.

(Adzic Gojko. 2011. 43-49)

3.2 Project Triangle

This section discusses the project triangle, a very basic setting to consider when

starting any kind of project. Projects throughout history balanced between the scope/capability, cost and time of completion, and this theoretical framework refers not only to SkyNest Project, but every project ever made.

Any project is constrained by three factors:

- Product capability
- Cost
- Timescale

These factors are related as indicated in **Figure 3.** Changes affecting one factor will affect the other factors. Each tip of the triangle represents the best possible solution in that aspect (Capability, Cost, or Time). Every decision made in project positions the process within the figure. For example, if the project is moved toward the more cost efficient solution, the project's capability/result quality will go down and the time needed will go up.
(Hull, J and H. 2002. 162)

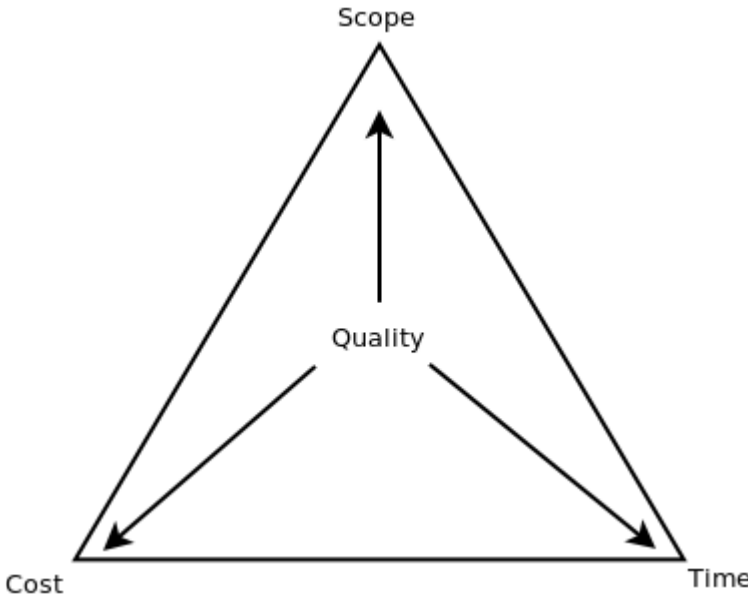

Figure 3: Project management triangle

Time

Project's activities can take a shorter or longer amount of time to complete. The amount of time required depends on a number of factors such as the number of people working on the project, experience, skills etc. Time is a crucial factor, however, it is completely uncontrollable. Failure to meet deadlines in a project can create negative effects. Often the main reason to fail in deadlines is the lack of resources. (Tutorialspoint. Project management triangle.)

Cost

It is essential for both the project manager and the organization to have an estimated cost when undertaking a project. Budgets will ensure that a project is developed or implemented below a certain cost. Sometimes project managers have to allow additional resources to be used in order to meet the deadlines. (Tutorialspoint. Project management triangle.)

Scope

Scope illustrates the project undertaken. This consists of a list of deliverables which need to be addressed by the project team. The project manager should know how to manage both the scope and any change in scope which impacts time and cost. (Tutorialspoint. Project management triangle.)

Quality

While it is not the part of the project management triangle, quality is the ultimate objective of every delivery. Hence, the project management triangle represents quality. (Tutorialspoint. Project management triangle.)

In software development the project's quality may be part of the triangle instead of scope, but in there it refers to the capabilities of the product that is produced from the project. In practice this means the same as the scope.

3.3 Software requirement management

Requirement is something that a product can do, or a quality, that a product should have. Requirements fall into three categories:

- Functional requirement. (ie. Software has the ability to import XML files)
- Non-functional requirement. (ie. Softwares XML import function contains color coded buttons)
- Constraints.(ie. This software must be designed into windows platform)

Creating software requirements starts from understanding the (customer) company's business goals. When this need is analyzed, it can be written down into a simplified sentence. This derived result is called a customer requirement. The customer requirement is then led into multiple software requirements that are in many cases software functionalities. Usually, this is more difficult to determine in practice.(Haikala I. & Mikkonen T. 2011, 61-63)

Requirement traceability is the ability to follow the life of customer requirements from definition into implementation and on the other hand backwards from implementation into customer requirements. In practice this means a thorough documentation of customer requirements and system dependencies, and keeping the connections between requirements up. This traceability is highly connected to change management. If something is changed during the original development plan, the effect chain can be more easily traced. Taken to extreme, it can be traced from a piece of code. (Haikala I. & Mikkonen T. 2011, 63)

Good requirement is error free and clear. Certain qualities are required from a good requirement. It needs to be::

- **Precise and understandable**: Precision guarantees that the requirement fulfillment can be measured.

- **Testable**: Ability to measure the requirement.
- **Traceable backwards**: Where does the requirement come from?
- **Traceable forwards**: Technical implementation and test cases linked to the requirement.

(Haikala I. & Mikkonen T. 2011, 63)

There are a many criteria that should be met in requirement statements (User Stories). They can be summarized as below:

- Atomic: statements carry a traceable element.
- Unique: statements are uniquely identifiable.
- Feasible: Technically possible within schedule and cost.
- Legal: legally possible.
- Clear: statements are clearly understandable.
- Precise: statements are precise and concise.
- Verifiable: statements are verifiable and it is known how they are verifiable.
- Abstract: do not impose a solution of design specific to the layer below.

(Hull, Jackson and Dick. 2002. 89-90)

In addition to these, there are also criteria for a set of requirements as a whole:

- Complete: all requirements are present.
- Consistent: no two requirements are in conflict.
- Non-redundant: each requirement is expressed only once.
- Modular: requirements statements that belong together are close to one another.
- Structured: there is a clear structure to the requirement document.
- Satisfied: the appropriate degree of traceability coverage has been achieved.
- Qualified: the appropriate degree of traceability coverage has been achieved.

(Hull, Jackson and Dick. 2002. 90)

In addition to the criteria, there are matters that should be avoided, since they make the statements unclear, vague and speculative:

- Avoid rambling: Conciseness is a virtue. The statements do not have to read like a novel.
- Avoid let-out clauses: such as “if that should be necessary” - they make the statements invalid.
- Avoid putting more than one requirement in a paragraph: often indicated by the presence of “and”
- Avoid speculations.
- Avoid vague words: Usually, generally, often, normally, typically.
- Avoid vague terms: user friendly, versatile, flexible.
- Avoid wishful thinking: 100 per cent reliable, please all users, safe, run on all platforms, never fail, handle all unexpected failures, upgradeable in all future situations.

(Hull, Jackson and Dick. 2002. 91)

Both Hull, Jackson and Dick, and Haikala I. And Mikkonen T. highlight the same key features in the requirements, although Hull, Jackson and Dick. go much deeper in their analysis and details.

Hull, Jackson and Dick refer to Standish group (1995, 1996) and scientific American (September 1994) for most common reasons for project failures:

Table 1: Most common reasons for project failure

Incomplete requirements	13.1%
Lack of user involvement	12.4%
Lack of resources	10.6%
Unrealistic expectations	9.9%
Lack of executive support	9.3%
Changing requirements/specifications	8.7%
Lack of Planning	8.1%
Did not need it any longer	7.5%

In addition they refer the same source again for most common reasons for project success:

Table 2: Most common reasons for project success

User involvement	15.9%
Management support	13.9%
Clear statement of requirements	13.0%
Proper planning	9.6%
Realistic expectations	8.2%
Smaller milestones	7.7%
Competent staff	7.2%
Ownership	5.3%

As can be seen, the requirements factor is for multiple reasons in both **Table 1** and **Table 2**, either directly (Incomplete requirements, changing requirements/specifications and their opposites on **Table 2**) or indirectly (unrealistic expectations, lack of planning and their opposites on **Table 2**).

4 Testing

This chapter details the general testing theory, related to the practical work conducted in this thesis. This chapter clarifies the general testing process, the testing theories and the test levels, which all get derived into testing process of FreeNest. This section also includes few additional procedures used commonly around the world, and even though they are not used in FreeNest testing, this should give an idea of new methods that might get incorporated into FreeNest development.

Test stages are test planning, creating the test environment, test execution and reviewing the results. These stages and their error tracing and debugging usually take half of the project resources. Because of this, testing requires great attention to become as good as possible. In software, testing is defined as planned error scouring by executing software or a part of it. In practice, testing unfortunately often takes place in a very random way by trial and error, especially if the tester is a programmer of the software. In this case, it is more of a purpose to prove the software works, rather than seeking errors. Testing grants the possibility to prove that software has errors; however proving that software has no errors is impossible. In practice testing usually covers only a fraction of the actual functions. This does not mean that testing effort should be minimal, it just means that the software is not without errors despite good test results. (Haikala I. & Mikkonen T. 2011, 205)

In this thesis, testing is the main focus and is emphasized throughout the whole document. This is why it is separated from **Basic elements of software project.**

4.1 Errors, mistakes and bugs

During testing, a bug is a deviation from specification. Consistent testing without specification is impossible, because the correctness of the end result cannot be defined. Usually most used specifications used in testing are functional specifications and technical specifications. Interpreting these documents might bring up disagreements and in a customer's mind, an error might be a feature. (Haikala I. & Mikkonen T. 2011, 205-206)

Bug severity varies greatly; it may completely block the whole software, or it may be just a minor cosmetic nuisance in the user interface. In practice even the best specification is always lacks something (for example: a feature, a section, a component), in which a conflict cannot be solved by looking at the specification. In software it is estimated that they contain one error for ten rows of code. Software that has been in use for long is estimated to have one error for 1000 rows of code. Another estimation is that around 5% of errors are not ever found, because a buggy function may not lead to an error output. Errorneous part of software may lead to a fault, which can repair itself because of some other part of the software, or its effect may be undone in result of another error. In the worst case, it may cause a failure, which is visible in the software's output. (Haikala I. & Mikkonen T. 2011, 206)

4.2 Seven principles of testing

Testing itself contains seven key principles that are important to understand:

1. Testing enables to see bugs; however, failing to find any does not mean that none exist. Tests must be designed to find as many bugs as possible. As it is assumed that a software contains bugs, a test that finds bugs is a better one than the one that does not.
2. Exhaustive testing is impossible, as there are too many variables. Testers should focus on the most critical priorities and risks.
3. Testing should be done as soon as possible after the product or application has been made, in order to get fast issue fixing. Errors identified later in the project are more expensive to fix.
4. Using defect clustering is useful because software problems cluster around narrow areas or functions. Identification and focus on these clusters enables the testers to efficiently test the sensitive areas while at the same time testing the non-sensitive areas.
5. It is advisable to use a variety of tests and techniques to find more defects inside the product. Using the same tests and methods end up finding less bugs and errors than by varying the testing methods.
6. Same tests should not be applied across the board as different products have different requirements, functions and purposes. As an example, a website and a company intranet should be not tested differently from each other.
7. Testers should not fall into the misconception of no errors found. It is not the same to not find any errors, and not have any errors in a software. It should always be assumed that a software contains faults.

(UniversalExams, The 7 Key Principles of the ISEB/ISTQB Foundation Level Exam)

4.3 Fundamental Test Process

Fundamental test process comprises five activities: Planning, Specification, execution, recording and checking for test completion. The test process always starts by planning and ends with Checking for Test Completion. Any of these activities may be repeated or revised, and they can be started before the previous one has ended. (TestingExcellence, Fundamental Test Process)

Planning

By planning well, the testing itself becomes easier and better. It is good to have an overall test strategy, and possibly a project test plan. Test planning activity generates a specific test plan, for example a system test plan. While planning, a scope for the testing must be defined, and all the assumptions must be stated. All other components should be identified before the testing can commence and completion criteria should be stated. Some Examples of completion criteria are:

- 100% statement coverage
- 100% requirement coverage
- 100% test cases have been run
- 100% of high severity faults fixed
- 80% of low and medium severity faults fixed
- maximum of 50 known faults remain
- maximum of 10 high severity faults predicted
- time has run out
- testing budget is used up

Some of these are better than others depending on the software to be tested and using combinations is better than using just one. (TestingExcellence, Fundamental Test Process)

Specification

Specification is the process of designing the test cases according to the test plan, using the techniques and work flow presented. For each test case an objective, an initial state of the software, the input sequence and the expected outcome are specified. This process can be broken down into three specific tasks that help to understand the process:

- Identify test conditions
- Design test cases. Determine how the identified test conditions are going to be exercised.
- Build test cases. Study how to implement test cases (scripts, data)

(TestingExcellence, Fundamental Test Process)

Execution

In this activity all test cases are executed. This can be carried out either manually or with the use of a test execution automation tool. The order of the test case execution does matters in testing. The most important test cases should be executed first, generally because the most important test cases are the ones that are most likely to find most serious errors; however, they may also be the ones that concentrate on most important parts of the software. (TestingExcellence, Fundamental Test Process)

There may be situations in which not all test cases are to be executed. In the case of finding too many faults with few of the test cases, it may be considered unnecessary to perform the rest of the cases, until the faults have been fixed. In practice this may mean that within a limited time, some test cases are prioritized to be executed, and faults generated from those are more important to fix rather than executing test cases that are insignificant, or generate just too many minimal bugs. (TestingExcellence, Fundamental Test Process)

Recording

Recording process is carried out in parallel with the test execution. The first issues to be recorded are the versions of the software under the test and test specifications to be used. For each test case thereafter the actual outcomes and test coverage levels achieved are to be recorded. The document that is generated is called a test log. (TestingExcellence, Fundamental Test Process)

The documented expected outcome should be compared to the actual outcome gained from testing, and analyzed so that the source of the fault can be found. A test may have been executed incorrectly, and that is simply fixable by rerunning the test. Fault may also lie in the environment set-up or the reason may be having the wrong version under testing. Specifications may also be faulty; the expected outcome could be wrong, steps may be unclear or the fault is actually in the software itself. In these cases the fault should be fixed and the test case executed again. (TestingExcellence, Fundamental Test Process)

Checking for completion

This activity has the purpose of checking the records against the completion criteria specified during test plan. Specifying more test cases is necessary if the compared criteria are not met. There are plenty of different types of coverage measure and different coverage measures apply to different levels of testing. (TestingExcellence, Fundamental Test Process)

Comparison of the five activities

It is easy to see that the first two activities (Test Planning and Test Specification) are intellectually challenging. Planning how much testing needs to be done, determining appropriate completion criteria, and similar tasks. This section requires analysis and thought. Similarly specifying test cases (identifying the most important test conditions, designing good test cases) requires a good understanding of all issues involved and the skill to balance them. (TestingExcellence, Fundamental Test Process)

The next two activities (test execution and test recording) involve predominantly clerical tasks. Furthermore, executing and recording are activities that are repeated many times whereas the first two activities, test planning and test specification are performed only once (they may be revisited if the completion criteria are not met the first time around but they are not repeated from scratch). The test execution and test recording activities can be largely automated and there are significant benefits in doing so. (TestingExcellence, Fundamental Test Process)

4.4 Test levels

The relationship between development and testing is usually illustrated with the V-model (**Figure 4**), which indicates the testing planning is to take place against the corresponding test level. In the reverse, when the test levels are done, they are compared to the corresponding documents. According to the model, different testing levels are unit/module testing, integration testing and system testing.

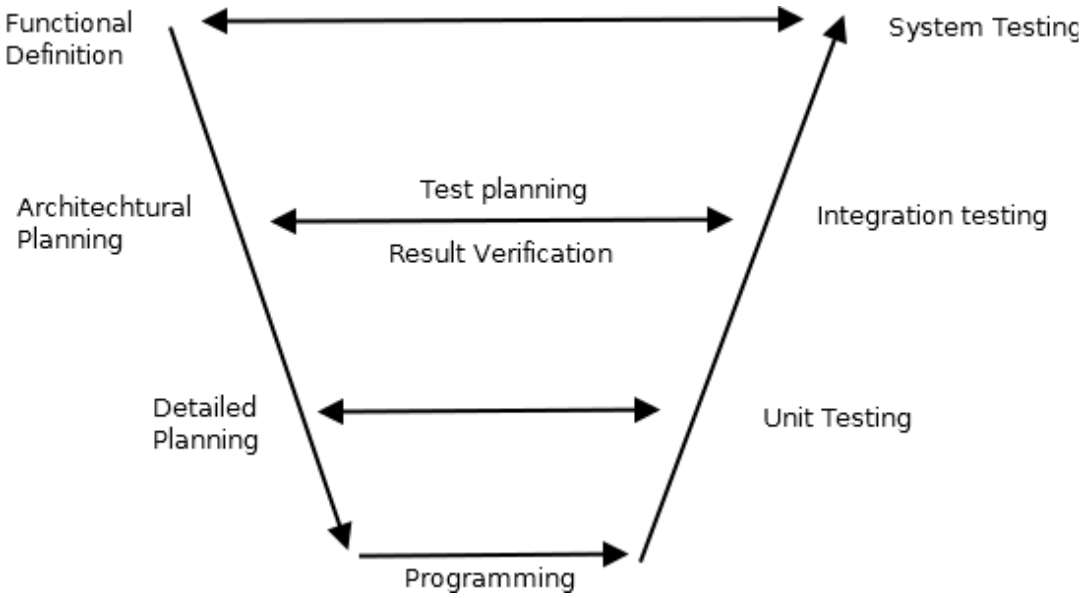

Figure 4: The V-model

Unit testing contains testing for the individual class or module that usually consists of 100-1000 lines of code. Class functionality is compared to the architecture and detailed plan results, usually the technical requirement document. Testing is usually performed by the developer. For the testing there might be need to create test beds, that can be used to test the class functionality. A test bed may include parts that simulate the environment, such as test drivers and test stubs. The test drivers make the class' calls and result inspection for service possible. The test stubs on the other hand, simulate the other classes that the testable class needs to be able to work, if they do not already exist. Sometimes for testing it is needed to create objects, which simulate required classes in some way. These are called mock objects. (Haikala I. & Mikkonen T. 2011, 207)

Integration testing follows when classes are combined, and it creates the complete software. The highlight lies in the junction functionality. The results are usually compared to the technical definition. System testing should be performed by people who have not been involved in the development itself. System testing may include

field testing and/or acceptance testing. Additionally, the non-functional abilities of systems are also tested:

- **Stress tests**: How well can a system deal with an expected amount of work?
- **Reliability tests**: How well does the system recover from errors, how long can it run without supervision?
- **Installation tests**: Will the system install itself as expected?
- **Usability tests**: How well can the design of the system be used in the end product?

The higher in the process of V-model a company currently is the more expensive it gets to fix errors. For example an error found in system testing phase may cause changes to multiple components. In the case of a need of change going unnoticed, it requires other components to be tested as well and to perform the system tests again. This re-testing, despite of the level, is called regression testing. (Haikala I. & Mikkonen T. 2011, 208)

If the same product is delivered to the customer in different versions, the need of testing grows because every deliverable version is to be tested individually. This is called release testing. Most commonly terms such as alpha and beta testing are used. Alpha testing means that the customer does the testing in the developer's environment, inside the company, while beta testing is for the customer testing inside customer's own environment. (Haikala I. & Mikkonen T. 2011, 208)

4.5 Choice of test cases

The test case choice has two basic approaches: white box testing (also known as structural testing or glass box testing) and black box testing (also known as functional testing). In the black box testing the test cases are chosen according to the testable

software's specifications without getting to know the software's execution. In the white box testing the execution is taken into account. Gray box testing is used when information about the principle of the software is needed. V-model and the nature of testing are usually connected, and moving the V-model from down to up the testing will gradually transform from white box into black box. (Haikala I. & Mikkonen T. 2011, 209)

The reason of choice of black box testing test case is to put the input space into equivalent classes. Equivalent classes are chosen on the principle of software working on a single equivalent class; it can work in all equivalent classes of the same category. This kind of test material choice is called equivalent partition. If test cases are considered to be in between classes, the choice process is called limit analysis. Partitioning equivalent classes has the problem of insecurity of validity. Testers might think that an equivalent class is individual, but it may actually consist of multiple classes. Developing the limit analysis is not an easy task, either. (Haikala I. & Mikkonen T. 2011, 209)

4.6 Ending test criteria

The amount of testing is a compromise between resources and the reliability of the product, also including remaining faults and possible delays. In addition, it is hard to evaluate how much testing is required, and especially in system testing the process may continue until time and/or money run out. Additionally the amount of testing is not the same as the efficiency of testing: few hours of carefully planned testing can lead to better results than a random trial and error spanned to multiple days. (Haikala I. & Mikkonen T. 2011, 210)

The ending of testing should always have acceptance criteria, which can be defined in

a test plan. System testing criteria can be related to cumulatively found error amount; when no more errors are found, testing can be stopped. The situation is not improved as usually a project has solid resources and a strict schedule, and these are informed to the customer. It may be difficult to evaluate the duration of the project, if the end of testing is considered to be a stabilized error curve, because the required amount of work for getting a steady error curve cannot be known beforehand. (Haikala I. & Mikkonen T. 2011, 210)

4.6.1 Complexity metrics

Complexity metrics are used to evaluate the complexity of software modules by analyzing the source code. With them, parts can be located that require testing from software. The most commonly used complexity metric is cyclomatic complexity, which is calculated for each function separately. The complexity is gained by adding one to function's control web branches. The resulting number describes the complexity of the function's control web. (Haikala I. & Mikkonen T. 2011, 210-211)

There are contradictory statements conerning the usefulness of complexity metrics. Sometimes complexity metrics correlate well with the found errors, and on the other hand, many professionals approach them very skeptically. Usually it is commonly understood that complex software has complex execution. In addition, the scaling of the results from the metrics is hard. (Haikala I. & Mikkonen T. 2011, 211)

4.6.2 Coverage metrics

Coverage metrics can be used to ensure that the test material used can cover all parts of the testable software. Coverage metrics can be used as evidence of a sufficient

testing. The metrics can be used for both functional and code coverage. Various code level coverage metrics are:

- **Statement coverage** indicate that at least each sentence in software is executed at least once.
- **Decision coverage** is for ensuring that any “if” -clause will at least once get both inputs (true/untrue)
- **Condition coverage** is for a decision to have all its values. This is not necessarily followed by decision coverage, however the coverages may get linked and are together referred as decision/condition coverage.
- **Multiple condition coverage** is to ensure that testing must be performed with all possible combinations of the conditions.
- **Path testing** is inspecting the software as a directed net, in which the nodes are the software's forks. This testing aims to cover as many different paths as possible. A complete path testing can only be done inside a single function, because a full software's amount of paths is practically infinite. (Haikala I. & Mikkonen T. 2011, 211-212)

4.6.3 Error seeding

Error seeding means adding errors into the source code on purpose, so that during testing it can be estimated how many errors there still are in the software. Assuming that software has X amount of actual errors, and Y amount of seeded errors, and during testing X' real errors and Y' seeded errors have been found. It is presumed that an approximately equal amount of errors have been real and seeded, and the amount of errors can be evaluated by $X = X' Y / Y'$. The efficiency of testing may also be estimated by making multiple versions of the software, where in each of them there is a seeded error. This method is called mutation testing. (Haikala I. & Mikkonen T. 2011, 212)

In practical software development error seeding is barely used, because the seeded errors are usually acted within limited resources and generating extra work is not sought after. A doubt rises if all the seeded errors have been removed, and what the validity of testing is, when the source code has been altered after testing. Additionally, finding the correct kind of seed is difficult. Nevertheless, error seeding may be applied to regression testing materials. It can also be used for document inspection method: the author may leave mistakes into the document on purpose to make sure that the people doing the inspections have had a thorough look into the material. (Haikala I. & Mikkonen T. 2011, 212-213)

4.6.4 Tool support

As for execution of regression and publication testing, methods should be as much as automated. Tools used for testing automation are test bed generators, comparers and test coverage tools. Test case automatic generation is in most of the cases impossible, because it requires formal specifications of the functionalities of the testable software. In some special cases this is possible. If for example a complicated interface is specified in a state diagram, the test material can be generated according to the state diagram and also the test result inspection is to be automated. (Haikala I. & Mikkonen T. 2011, 213)

Results gained from previous software versions can also be exploited by using comparison software. This way the result can be compared to previous test results. These are not without flaws, however. Using comparison software has the problem of constantly changing information inside the program, such as calculated time.(Haikala I. & Mikkonen T. 2011, 213)

The test coverage also needs its own tools to be used. By principle, the test coverage tools function as pre-processors that instrument the code of a program in a way that during the performance all the necessary information regarding coverage is collected. The same tool can usually be used to additionally measure the code row execution amount and hardware processor usage. Instrumenting increase the size of the source code, and slows down the performance, which in return hampers correct usage of the test coverage tools in system testing, real time systems and the smallest embedded systems. (Haikala I. & Mikkonen T. 2011, 213-214)

For locating performance issue profiling, software is usually used to give information about how program's use of time is divided between different components. With this information it is possible to concentrate on the actually problematic parts in the program when optimizing the performance. (Haikala I. & Mikkonen T. 2011, 214)

Using the test tools may influence the functionality of the software. Especially in embedded systems it is usual that the development hardware in the software works, while it may not function in the actual target hardware. Another problem is that after linking the test tool into the software problems may not occur. (Haikala I. & Mikkonen T. 2011, 214)

4.6.5 Test-Driven Development

Also shortened into TDD test driven development is a way to develop software in a way that the development cycles are extremely short. This method is originally introduced as a part of a larger entity, however lately it has become an independent method. In the method the test cases are planned and are executed even before implementation of new features, instead of a more traditional testing after the whole software is done. The development cycle forms in the following way:

- **Add a new test case that is connected to the new feature**. This phase is for the developer to understand what the idea in the new feature is.
- **Execute all cases and inspect if the newly added test case fails**. If the execution of a new test case succeeds, there is something wrong with the current software or in test procedures, and it has to be fixed before the development proceeds.
- **Implement the new feature**. Implementation does not have to be perfect. The most important issue is that it enables the new test case to be executable.
- **Execute test cases again, and see if they succeeded**. Because all tests are always executed, developers can be sure that the new feature did not break the older, already functioning parts of the software.
- **Re-factor the software according to the need**. Because tests can always be executed again, modifying the software is usually much easier. That is also the reason why large modifications that improve the basic structure are easier to make.

Test-driven development has many different variations of the original. The most traditional version presents that thinking beforehand is unnecessary, and that by refactoring the software it inevitably gains a structure. In the other extremity test cases are in a center role of all, even if the main idea of the software has been figured even before test cases. It may even be that tests and implementation are planned side by side. (Haikala I. & Mikkonen T. 2011, 214-215)

4.6.6 Test documentation

The test plan states what tests are done and when, how they are organized and what kind of end results are expected. Factors affecting the test plan are presented in **Figure 5**. Ending tests may vary greatly; it may be the deadline for testing, 100 % execution of test cases, 100 % of all errors have been repaired and after all repairs a

100 % successful regression testing has been performed. It should be noted that acceptance criteria are not the same as testing end criteria, as sometimes tests cannot gain accepted status. (Haikala I. & Mikkonen T. 2011, 215)

Errors during testing should be reported and analyzed, otherwise it is impossible to make out the time used for testing and development targets. Information to be documented in error reports are error description, seriousness level of the error , when and how it is found, how the error could have been found earlier, when the error has been made and how it could have been prevented. In practice errors are documented in a system test report and customer reclamation documents. (Haikala I. & Mikkonen T. 2011, 215-217)

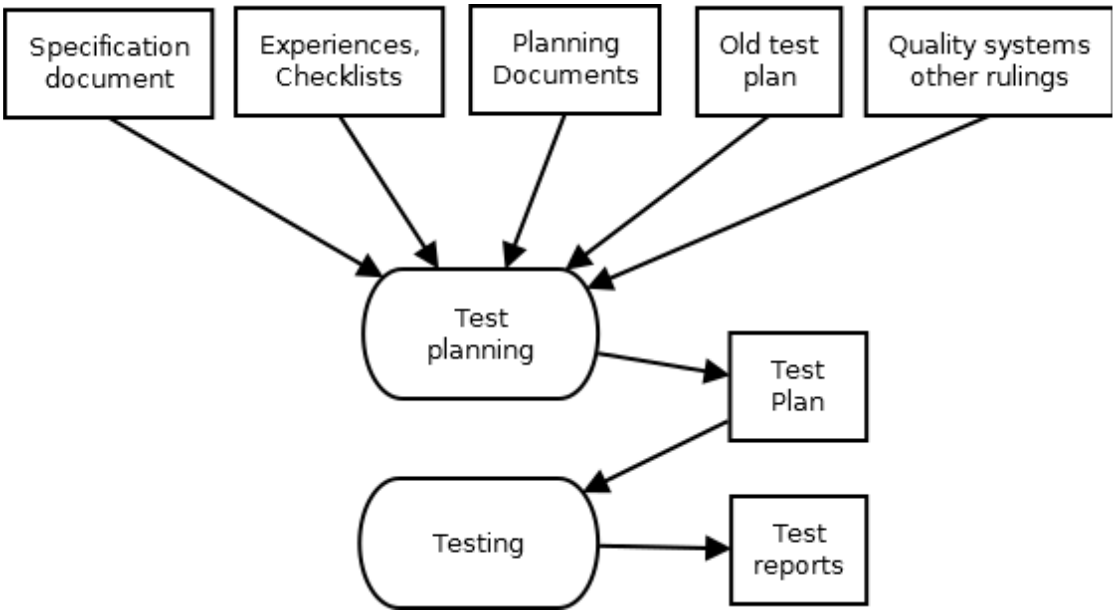

Figure 5: Factors of testing

Testing documentary may accumulate relatively much: system testing plan, integration testing plan of every integration test, unit testing plan from every unit test and a report from each executed test. In a small scale project usually one test plan is

enough. The test plan contains both integration and system testing. An introductory plan is made during the definition phase and it is filled in later on in the planning phase. Sometimes a test plan is included. (Haikala I. & Mikkonen T. 2011, 218)

4.7 Developers as testers

Developers have been poor testers, as they have many factors inhibiting themselves that affect the testing results. This chapter is to clarify the most essential ones.

A common principle is that the developers shall not participate in to the test work. The reason for this is the fact that they will concentrate on the parts that will actually work, unlike an outside tester. In practice it is usual that the developers perform at least the unit testing and sometimes even the integration testing, and outsider testers do not participate until in the system testing phase. In addition it is natural for the developer to test the system with multiple inputs during development. This however is not actual testing, as it lacks the strict orderliness.(Haikala I. & Mikkonen T. 2011, 209)

Developers are emotionally attached to the code they write. In their eyes their work is good enough; however with outside view, the code might be really flawed. (Montvelisky. 2010. http://qablog.practitest.com)

Developers focus on the positive path. This means that the work of a developer takes the positive scenarios and enables them on the product. Most of their efforts are focused on how to make issues work right, effectively, efficiently, and further. Testers, on the other hand value a negative mind-set, a what-can-go-wrong state. (Montvelisky. 2010. http://qablog.practitest.com)

A developer's work is based on the principle of simplifying complex scenarios. The basic task a tester does as a part of his work is to look for complex scenarios (multiple actions simultaneously, make operations repeat over and over again, and further) in order to break the system and find the bugs. This, however, is in contrast with the developer's side, they break down complex processes into smallest possible components that will allow them to create a solution.
(Montvelisky. 2010. http://qablog.practitest.com)

Experienced testers develop a sense to detect what “does not fit” in the picture. Developers lack this sense as their focus has been on just looking at the code.
(Montvelisky. 2010. http://qablog.practitest.com)

Developers lack end-to-end and real-user perspective. While they have a vague idea what the end perspective should be, they only concentrate on a single component or feature in the product. Testers, on the other hand are required to have a wide perspective, and look for ways to simulate and test the final product.
(Montvelisky. 2010. http://qablog.practitest.com)

Developers have less experience with common bugs and application pitfalls. Even though developers gain experience on bugs and pitfalls during their development, testers acquire this experience much faster and on a much deeper level. For example, an experienced tester sees a form in a software, and will immediately start thinking about common bugs and failures he may find in it and starts testing for them.
(Montvelisky. 2010. http://qablog.practitest.com)

5 ALM – Application life cycle management

David Chappell (2008) in his source clarifies what ALM is by first stating that it is much more than just a SLDC, as ALM reaches from the first idea drafted, to the absolute end of a program, project or software. For example, from imagining a product starting idea,to the point where the company pulls it out of market and states that it is invalid and outdated. (Chappell. 2008.)

ALM can be separated into three distinct areas: governance, development and operations. How these three areas are handled can be seen on **Figure 6**.

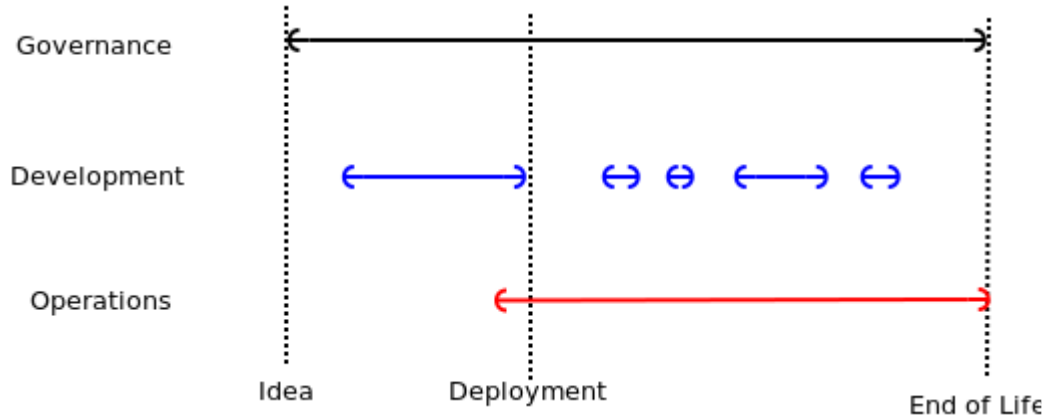

Figure 6: ALM Areas

(Chappell. 2008.)

Governance contains all the decision making and project management and it extends throughout the entire time. Development takes place first between idea and deployment, and after that it usually appears again because of upgrades and new versions. Operations required for running and managing the application start slightly

before deployment and lasts until the application is removed from service. (Chappell. 2008.)

ALM addresses build, configuration, deployment, release, test, quality, integration and requirement management, with comprehensive approach. (Huettermann. June 29, 2011.)

Life cycle of the project describes the phases and steps completed from the initial concept into post-deployment and project closure.(Garton, Colleen & McCullogh, Erika. 2004, 19)

Life Cycle can be divided into six phases:

- Planning
- Design
- Development
- Integration (including testing)
- Deployment
- Post-Deployment

All stages are critical for the project, and skipping any section will result in a failed project. Usually most ignored is Post-Deployment, and as always a project is never perfect and includes flaws that should be addressed in Post-deployment.
(Garton, Colleen & McCullogh, Erika. 2004, 19-20)

Badiru adds more phases into the previously mentioned ones, as he extends the management elements (Badiru, Adedeji B. 2012, 23-25):

- Problem Identification (before Definition)
- Project Definition (At the beginning of Planning)
- Project Organizing (During Planning)

- Resource Allocation (During Planning)
- Project Scheduling (After Resource Allocation)
- Project tracking and reporting (During development and integration)
- Project Control (During development and integration)

All these phases are put into the following chronological order, indicated by their titles.

Problem Identification

A need for a proposed project is identified, defined and justified. (Badiru, Adedeji B. 2012, 23)

Project Definition

In this phase the purpose of the project is clarified and a mission statement is given. Generally this phase gives guidelines on how to avoid common mistakes during project management. (Badiru, Adedeji B. 2012, 23)

Project Planning

Planning starts with the project concept and includes all meetings, research and reporting. The planning phase is complete when project definition, project plan and project approach are completed, reviewed and approved. It is important to get it right the first time because the whole project is performed according to the introduced plans. (Garton, Colleen & McCullogh, Erika. 2004, 20-21)

A plan represents series of actions needed to accomplish a goal. It may vary from being a statement of a project goal into being a detailed account of procedures of the project. Planning can be summarized as

- Objectives

- Project Definition
- Team organization
- Performance Criteria.

(Badiru, Adedeji B. 2012, 23)

Project Organizing

Specifies how to use all the functions of the personnel during the project. Organizing is performed during the project planning, and it contains guiding and supervising the subordinates. Employees perform better when expectations are clear, but they should also have some flexibility for independent choices. (Badiru, Adedeji B. 2012, 23)

Resource Allocation

Allocating resources for different parts of development is conducted in this phase. These resources may be e.g. money, people, equipment or information.

Project Scheduling

The main purpose of scheduling is to allocate resources. Scheduling is carried out when tasks are assigned to be completed within a certain time period. There are many factors that complicate the scheduling, such as urgency level, time limitations and work priorities. However, assigning a task to a certain time slot may not result in a satisfactory result, hence careful control is required.

Design

This phase is commonly neglected, at least to some extend and it includes reviewing functional and technical requirements and detailing how the components will be tested. Some prototyping may take place already at this phase, however, only as a base for the next phase. (Garton, Colleen & McCullogh, Erika. 2004, 21-22)

Development

This phase includes a detailed project schedule, writing the code and system building. During the planning phase this has been divided into milestones, which separate different accomplishments, and general deadline for the features. This section contains the unit testing and bug fixing, so that the code will be ready for the next phase. (Garton, Colleen & McCullogh, Erika. 2004, 22)

Project Tracking and Reporting

This phase is all about checking if the project results are in sync with project plans. This is a required phase for the next, the project Control phase.

Project Control

Control requires that required actions will be taken when the work itself starts to deviate from the original plan. The methods for these include measurement, evaluation and corrective actions. This phase includes smaller actions that take place during the project:

- Tracking and reporting
- Measurement and evaluation
- Corrective action (plan revision, rescheduling, updating)

(Badiru, Adedeji B. 2012, 24)

Integration

In this phase all the code or the components are integrated into a final product, which will lead into integration issues and errors. This is the most stressful phase during the Life Cycle, as deadline will come to a close, and finding critical errors will take time. (Garton, Colleen & McCullogh, Erika. 2004, 22-23)

Deployment

Few things are required before the project can be finished. The client will be trained

for the use of the software, critical bugs will be fixed and verified, after which the client will perform acceptance testing. (Garton, Colleen & McCullogh, Erika. 2004, 23)

Project Termination

As the last stage of project, the termination phase contains a definite conclusion to the project, such as delivery of the final report. Arrangements may be made for follow-up activities that extend or improve the extent of the project. Even though these should be treated as separate projects, they should still contain referrals to their parent project. (Badiru, Adedeji B. 2012, 25)

Post-deployment

During this period, the service company may have zero to many responsibilities, such as technical support, customer service and customer operations. (Garton, Colleen & McCullogh, Erika. 2004, 23)

All of the previously discussed steps may not necessarily run consecutively, as they often overlap and run concurrently at some periods of time. Project managers have the responsibility to ensure that every team has work during the whole project, this way minimizing project downtime. Also features require varying amounts of design, mostly depending on the size of the feature.
(Garton, Colleen & McCullogh, Erika. 2004, 24)

6 About FreeNest Development

This chapter discusses the case of FreeNest, and informs the reader about the actual details conducted in practice. All the theory explained before will come into practice

in this chapter, as it explains the details on which project development methods, testing procedures and such are used. Also, these are analyzed, which emerges thoughts about improvements and mistakes that should be attended to in the future.

6.1 Three factors of project triangle

By looking at the general picture of SkyNest Project, it can be seen that scope, time and price are different values, such as they are in any other software development project. By inspecting these factors a lead may be found quickly in what way the project is moved forward, with what aspects in it are highlighted and considered important. However there are also the features that are lacking from the project, and while they may be somewhat hidden, with a closer inspection they may reveal themselves.

6.1.1 Time

During the Summer Factory 2013, the schedule seemed to be really tied up together and having people worked constantly effectively. But to actually find out about the real time spent and the progress made, a person has to look at the whole annual time/progress relation. In fact, during fall, winter and spring the process slows down more heavily than one could first imagine. The employees do not work in any consistent method during the previously mentioned seasons, such as scrum to name an example. This generates inconsistency within the process. Students are the usual workforce, doing their practical training required in order to graduate from the university, they might not necessary know that much about software development, and may have too many questions. This problem is not as large as it sounds, as the

more experienced workforce encourages the inexperienced ones to ask about matters making the development process proceed at least at a slow pace, instead of coming to a stop altogether. During summers the Summer Factory catches up for this, and makes huge progress with a greater workforce of students. These students are guided for development tool usage and they gain the general idea in lectures at the workplace.

6.1.2 Scope

FreeNest seems quite massive, a large toolbox of various open source software development tools, it could even be described as messy in some parts. The product is huge, and keeping things in order can get difficult, especially when the components within are software products on their own. This creates huge numbers of various manuals and guides that have been built during the development. These are relatively easy to find on the FreeNest server maintained at JAMK, although as there are many people writing these guides, some may be named somewhat oddly, which in turn makes them not so easy to find. Nevertheless, despite being confusing at first, FreeNest contains the very essential tools for every kind of work within software development; Planning, testing, bug fixing, documentation and communication to name a few.

6.1.3 Cost

This is the part where SkyNest project is exceptionally good. While being a very huge project, collaborating with many different contributors, it is far cheaper than many other projects. This is because it contains open source material only, free to use and distribute. The majority of the workforce at JAMK is students in their practical

training period and they only get paid from the government in the form of student allowance. It is clearly shown in what part of the quality the management has put its effort, even though it makes it lack in other compartments, when inspecting overall quality.

6.2 Requirement

FreeNest requirement list is huge, as it contains all its components from current and previous versions. As the FreeNest is currently constantly developed, this feature list will grow during the years of development. All of the FreeNest features are listed inside its development Foswiki (a Wiki software within the server instance of FreeNest used by the SkyNest Project for development purposes), which is an open source Wiki component. While most of the features have detailed description within their own dedicated Wiki pages, there are some that lack in descriptive text, and it is unclear what the feature actually is.

6.3 Development process

There is a generic life cycle that can be applied to all projects. While the methods may vary greatly, all projects have the beginning, the middle and the end. FreeNest development has its own life cycle, which also has been depicted in a document-like swim lane chart. This swim lane chart contains the parts with different participants influencing their parts of the process. In addition, two topmost lanes represent storage and artifacts. The storage shows in which database the data is saved, and from where it can be later accessed. Artifacts are documents, guides and other important reported material in either electric or in paper form that is left behind

when moving onwards with the process. This swim lane chart be **Appendix 4**, and explained with a closer detail with smaller clips taken from it, shown as **Figures 7-11**.

The process starts with the requirements section, which represents interaction with the customer, when starting up with the requirements. This is a complicated process, with extensive discussions with the customer, making it way lengthier than one would first imagine. The requirements part is illustrated in **Figure 7**.

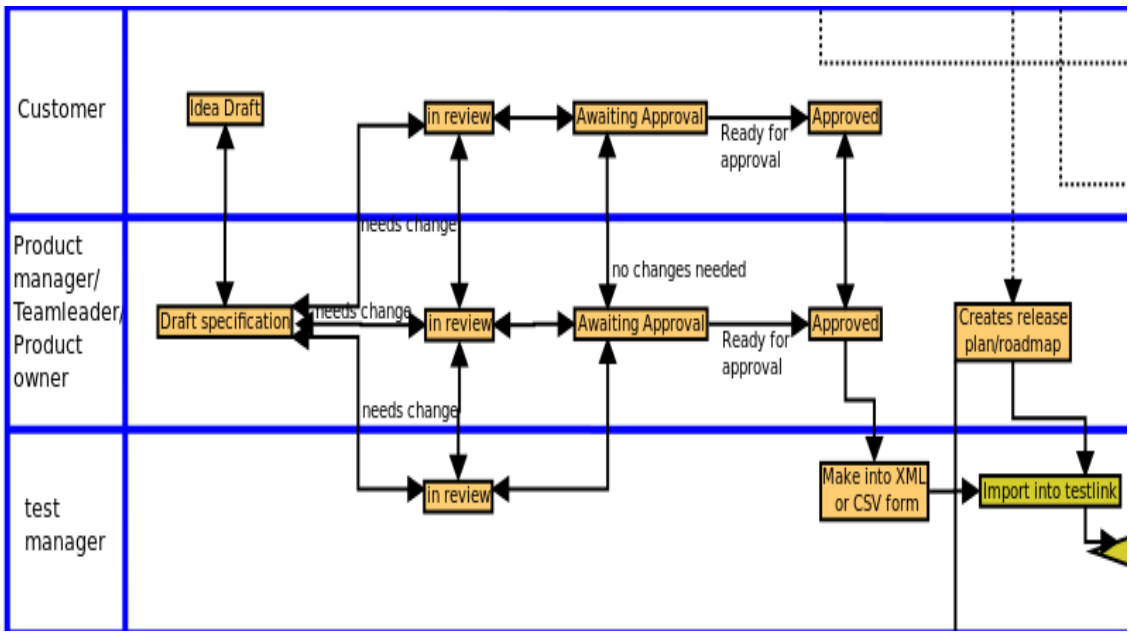

Figure 7: Requirements

Getting good requirements is really important, and everyone's feedback should be taken into account, whether it is from the customers, product manager, team leader, product owner or test manager. This is no exception in FreeNest, as all requirements are closely considered, and scheduled among different versions of FreeNest.

From the requirements two different procedures start; the testing process, and the development process. First stage includes the test planning phase, which comprises

creating a test project and new test cases for it. The planning phase is essential, as it creates the base for testing. In FreeNest, the test planning has been carried out many times, as there are constantly new features, upgrades and bug fixes of the code. Retesting the same issues might reveal something that could break down the system due to the new changes done into software. Because of the constant development, during a year FreeNest development goes through all forms of testing (integration, system, performance).

The following **Figure 8** shows how the beginning of testing is done in FreeNest, starting from the import of requirements into testlink, creating a new test project and further.

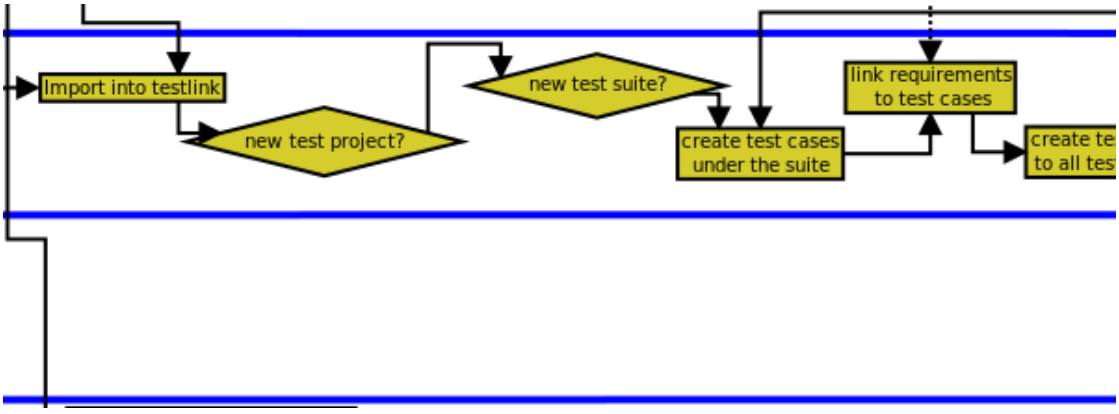

Figure 8: Testing part 1 (design)

After this initial planning phase, the test cases are next to be executed. **Figure 9** shows how the process is conducted after the planning phase, in the execution phase. First the tests receive a status of fail or pass, both of which have their own procedures to follow.

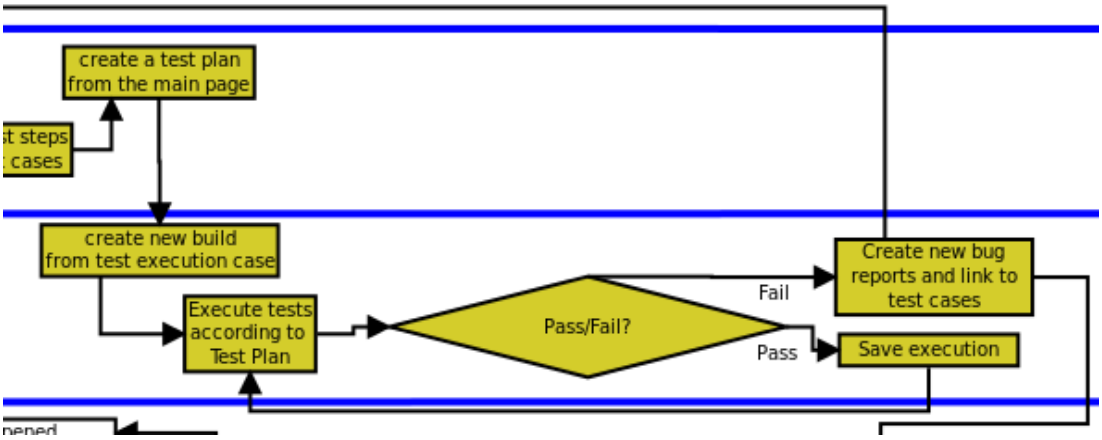

Figure 9: Testing part 2 (execution)

When all test cases are conducted, the bug fixing phase follows, which may even be carried out side by side with the testing, when enough errors have been found. **Figure 10** shows how the bug status and process changes. The bug status changes are very unrestricted, and can change from any status into any other status. During the usual work flow, however, the process is quite straightforward. The bug starts from unconfirmed, as one person has reported it. It needs to be verified by another (usually someone higher in hierarchy, or a development partner), which moves it then to confirmed status. Then after this, someone starts to fix the problem or the error found, and the status is transformed into In Progress. The person doing the bugfixing fixes the error, at least in his/her own opinion, which then leads the reported bug into status of Resolved. When this status is obtained, someone else will verify that the bug is fixed, who then changes the bug into verified status.

This basic flow is not necessarily always the case. During any point of bug fixing process the bug may return to any previous state, or skip a state. E.g. the seemingly resolved bug may be found to be faulty, and can be moved back into status of confirmed or a verified bug may be found to have a wrong fix and it is reopened.

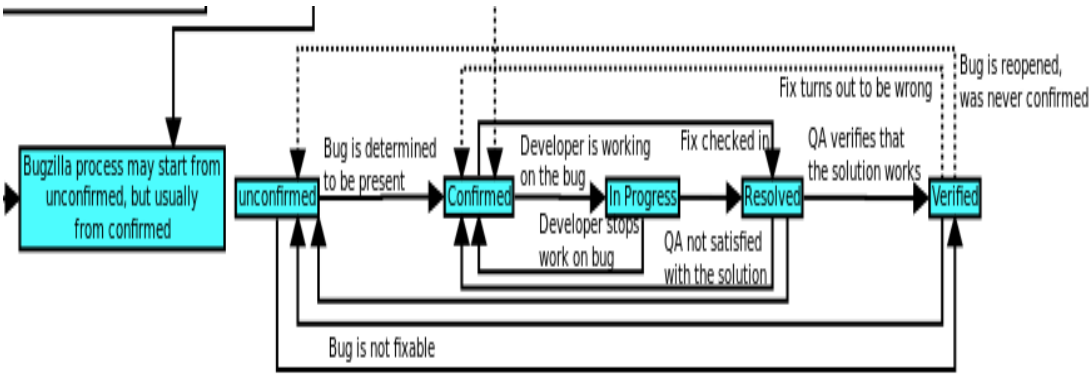

Figure 10: Bug status process

The developer's side is great deal simpler in the aspect of the process, while the major part of the work is done within the step itself. **Figure 11** shows how the developer puts tasks into different statuses during the development process.

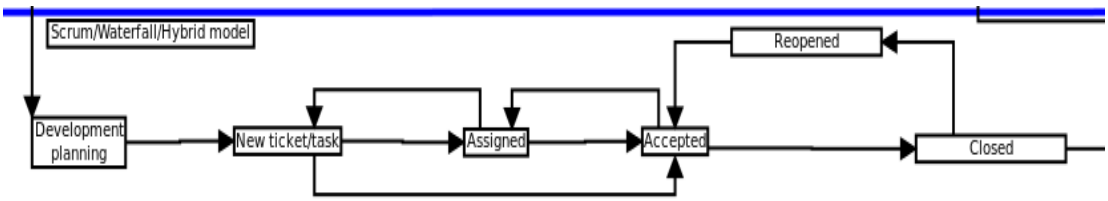

Figure 11: Developer's ticket progress

This part of the developer's process does not depend on the actual process method, be it scrum, waterfall or hybrid model as it only takes a stand on the ticket progress.

6.4 Scenario

The general working method for FreeNest during summer factory is usually scrum with development teams, and Kanban with design and maintenance teams. While

this may have seemed like a chaotic method of working, the teams acted separately and these work methods did not interfere with each other.

An original testing group (Hereby known as scribes, see **Appendix 3**) was trained to be familiar with testing, telling them about the functionalities of Testlink, Bugzilla and general testing procedures. The scribes were eager to ask questions, and seemed to learn really fast. After the basics, the scribes were taught the designing of test cases, which they started to do right away. On a steady pace they created a great number of test cases, which had a professional level quality in them.

After the training, the product owner introduced how a general testing period of work would be done, thus everyone in the summer factory would be a tester, at least for a while. This was carried out to teach everyone about testing, but also to actually find bugs and errors from the source codes of FreeNest. This thesis also states the facts and flaws that were found during this procedure. The theory was based on an idea of a group of people that have no experience of testing whatsoever who are to be included into executing tests and bug fixing. The basic premise was that originally the group of people was separated in different kinds of development teams: design, feature, maintenance and testing. These groups were then assembled together and separated into testing teams.

The group of testers, who already knew about testing procedures, started to create a resource list (**Appendix 3**). This resource list contains every available person listed, with their relevant information (person's names were replaced from the original version with person or scribe title, for privacy reasons). Teams were formed from persons on this list, and each team member had a priority number of changing from testing into bug fixing. According to their number, each team member was to change into bug fixing in the order of their given number. The people who were already experienced in testing were to set the statuses of each test case by the reports of the

testing team members. After these facts were introduced, each testing team gained different platforms to test, a combination of an operating system and a browser.

With this premise, the groups started working with their respective platforms, going through all the test cases individually that were handpicked to be shared among all the platforms. In total there were around 500 test cases, of which 300 were chosen into this testing work. At first, smaller group planning meetings were held, so that each member would know what platform(s) would be on their responsibility. After that each newly generated team members started working on their respective platforms and test cases. This continued throughout the sprint 7, and much process was made, both in learning and in development of FreeNest.

Appendix 1. and **Appendix 2.** include tutorials for a generic testing procedure, and a detailed guide to Testlink. These two are tutorials created for people attending the FreeNest Development, and are to be remain referral documents for years to come.

6.5 Analysis

6.5.1 General

The first matter to consider is the relation between time, scope and cost of FreeNest (Project Triangle). Even though the time consumed is the weak spot for the overall quality of FreeNest development, it may actually also be a strong asset. As it progresses relatively slowly, sometimes it may bring more stability, compared to a more rushed project. Components can be adjusted together in no hurry, more effort can be put into testing and overall planning.

The cost is something that can be hardly improved, as the cost is already really minimal, with SkyNest Project utilizing students as its workforce. However, there are some employees that get paid, which may cause more costs than one would first imagine. In addition though it is founded by JAMK, which has also provided some of the physical work environment.

The scope has been the main asset in the development of FreeNest, and is another strength of this project. Nevertheless it has its flaws too. The scope may be somewhat too large, making the FreeNest itself appear massive and unattractive to the beginner. It has a huge amount of features in it, many of which contain their own manuals, that might make it less used, as the future customers may find it intimidating. This has also resulted in a huge amount of requirements yet to be put into features, as the list just grows immensely over time.

6.5.2 Testing process

The testing process started off very smoothly. The phases were learning, planning, execution and error fixing. Exception to normal testing process was a quick learning phase, in which tutorial for Testlink was introduced to the initial testing group, who would later on act as scribes during the execution phase. This learning phase included the guides which are **Appendix 1** and **Appendix 2**. These guides were the main resource for learning aside the teaching, which was well received and was quickly shown to be assimilated into the practical work.

The planning phase included a great deal of tedious test case planning, and creating all sorts of relevant artifacts for future reference. These include the test cases

themselves, and a resource plan presented in the resource plan, which came into place during the execution phase; however, here is when some problems started to occur. There was confusion what was a resource plan to begin with. It was not clear to anyone what the idea behind the whole document was, until the product owner came back to explain it in more detail day or two later, which resulted in some wasted time. From this another problem arose; there should be some sort of a consistent plan which every student can also refer for general guidance later on, not just the regular employees on the project. To put it simple, the leadership lacks a constant guidance, which would bring it onto right track.

The leadership should also start to encourage more the communication between the basic groups, as it is very essential for the progress. There was a communication problem in this case also, as people did not communicate nearly as well as they should have. The basic job for scribes was to participate in the testing, and save the results from their respective testing teams into Testlink, and here is the situation where communication would have been essential. Only one scribe actually communicated with the team, making a note of what had been done, thus most of the platform results were not written in Testlink. The communication can be illustrated as in **figure 12**.

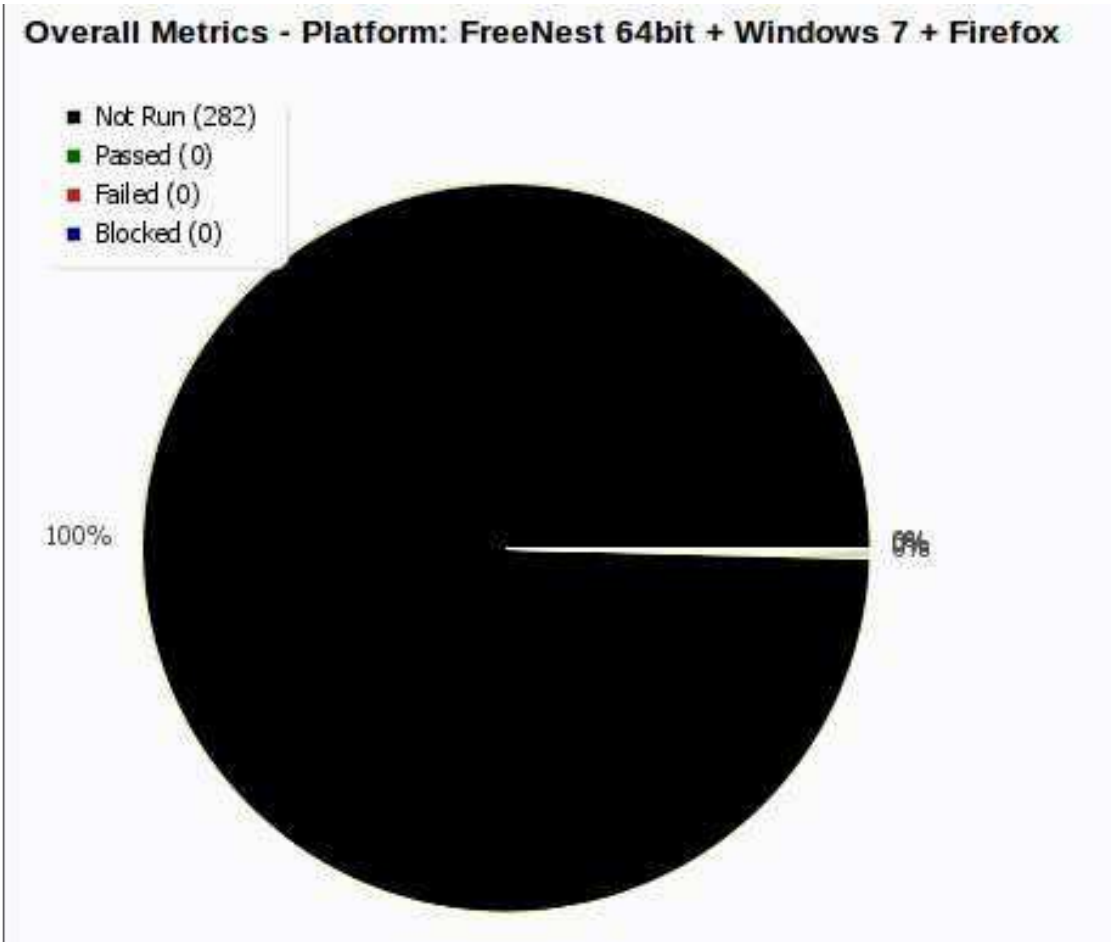

Figure 12: Results of lack of communication

Something went wrong during the testing process. It could be seen, by observation that the scribes were staying in one room, while the rest of their groups were in separate rooms. This created a threshold for going out and speaking to strangers. The leadership is not completely responsible about this, as this kind of behavior should have been left behind ages ago, and the scribes should have had a more mature approach, taken matters more seriously and been more self-imposed.

Another factor that should be considered, that while project owner and other employees on the project are usually busy with other business, either having meetings or out of town trips. The students working during the project tend to get

the mentality of the leadership rarely being present, and consider themselves lucky if they can get a hold of one person to get advice from. This may also result in the students not wishing to start seeking any of the standard employees, as running around the school campus without finding anyone is frustrating, which again leads to higher threshold for starting to seek guidance.

Besides these faults, the testing process is actually very well organized, and while there is some lack of communication, the results were, nevertheless, satisfactory, even if they were not recorded properly into any sort of form. Every time a project is developed, mistakes should be taken into consideration in the future, and learning from them may prevent them from happening again.

7 REFERENCES

Adzic, Gojko. 2011. Specification By Example. Manning Publications Co. Shelter island, NY.

Badiru, Adedeji B. 2012. Project Management Systems, Principles and applications. CRC Press, Taylor and Francis Group.

Chappell, David. December 2008. What is application lifecycle management? Referred on Sept 20, 2013. http://www.davidchappell.com/WhatIsALM--Chappell.pdf

Escalada, Monina. August 20, 2009. Writing the thesis outline: Theoretical framework. Referred on Sept 16, 2013. http://devcompage.com/?p=1419

FreeNest.org. 2012. Portable Product Platform Referred on April 24, 2013 http://freenest.org/about

Garton, Colleen & McCullogh, Erika. 2004. Fundamentals of Technology project management. First printing December 2004. MC Press Online, LP.

Haikala I. & Mikkonen T. 2011. Ohjelmistotuotannon käytännöt. 12th revised print. Hämeenlinna: Talentum Media Oy.

JAMK. Projektit. SkyNest. Referred on April 24, 2013. http://www.jamk.fi/projektit/1233

Huettermann, Michael. June 29, 2011. Agile Application Lifecycle Management (ALM). Referred on September 20, 2013. https://today.java.net/article/2011/06/29/agile-application-lifecycle-management-alm-0

Hull, Jackson and Dick. 2002. Requirements Engineering. Springer-Verlag Lonson Limited. Great Britain.

Montvelisky, Joel. May 5, 2010. Why can't developers be good testers? Referred on August 5, 2013. http://qablog.practitest.com/2010/05/why-cant-developers-be-good-testers/

Project management triangle. Tutorialspoint. n. d. Referenced on July 9 2013. http://www.tutorialspoint.com/management_concepts/project_management_triangle.htm

Royce W. 1970. http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf

TestingExcellence. November 8, 2008. Fundamental Test Process. Referred on June 3, 2013. http://www.testingexcellence.com/fundamental-test-process/

UniversalExams. January 5, 2011. The 7 Key Principles of the ISEB/ISTQB Foundation Level Exam. Referred on May 31, 2013. http://www.universalexams.com/iseb-istqb-certification/the-7-key-principles-of-the-isebistqb-foundation-level-exam

APPENDIX 1. BASICS OF TESTING

Description

Planning/Designing

Planning a test case requires a definite grasp on the concept and functionality of the testable component. A test designer should know exactly how the software should work, preferably from a manual. Now to actually create content is a bit harder. A test case should contain a brief headline/name that defines it and it is generally identifiable what it does. For example "Verify remove account" headline gives very clear idea of what is to be tested in it. After the headline, it is preferable to write a summary of what the test does. In this phase you can explain a bit more, but do not make it too long because it will just look like a big wall of text.

First you should create test cases as a concept, just the headline/name for them. You should consider to naming them very simply. Starting with the word "Verify" is mandatory in most cases and it helps to start thinking about the functionality to be tested. Changing name, moving an object, creating new object or writing text should each contain their own test case (ie. verify change name, Verify Object drag, Verify Create object, Verify Text input). Remember to include what is required for the test case to be executed (ie. Test case TC-15 has already been done, Pre-Created test data).
Creating steps is a tedious task, but needs to be done accurately and into an easily understandable text. Each step is a small procedure that a tester does when executing the test. Compare it to a cookbook, in which each recipe contains steps what to do and in which order. Writing a test step should also

always contain an expected result, so that the tester can then see what should take place after each step.

An example screenshot taken from Testlink, contains a test case on verifying a password change:

FN-7:Verify Password change

Edit | Delete | Move / Copy | Delete this version | Create a new version | Deactivate this version | Add to Test Plans | Export | Compare versions | Print view

Version 2
Created on 19/06/2013 09:09:31 by AdminUser
Last modified on 19/06/2013 09:09:49 by AdminUser

Summary
Changing user's own password in FreeNest 1.4

Preconditions
A fully installed FreeNest 1.4

#	Step actions	Expected Results	Execution
1	Open site with your browser: 192.168.42.121	Should see FreeNest login page	Manual
2	Log in with credentials: AdminUser and adminuser	Should see the new dashboard	Manual
3	Click Home -> User Details	Should see user details	Manual
4	Fill in the new password: Checking2013		Manual
5	Retype new password		Manual
6	Fill in current password: adminuser		Manual
7	Click Save Changes	Should see text: User details changed!	Manual
8	Fill in new password: adminuser		Manual
9	Retype new password		Manual
10	Fill in current password: Checking2013		Manual
11	Click Save Changes	Should see text: User details changed!	Manual
12	Log out	Should go back to Freenest login page	Manual

Figure 13: Example test case

This screenshots shows how detailed the steps, the name, summary and preconditions should be. As you can see, not each step contains an expected result, in this case on the writing steps.

Executing

When executing tests, baseline of determining success or failure is that if it does anything that is not written on the test case, it should be marked as a failure. In Testlink, there are statuses of not run and blocked. Not Run means

that the test is not executed. Do note that this status cannot be selected when executing tests in Testlink. The blocked status is a rare occasion when a test cannot be performed for any given reason, such as component not yet existing or a precondition missing.

After you have followed the steps instructed, and if the test fails at any point, you need to report into Bugzilla (Work Collaboration -> Defect Management). In there, click on the new tab, and a page looking like this opens:

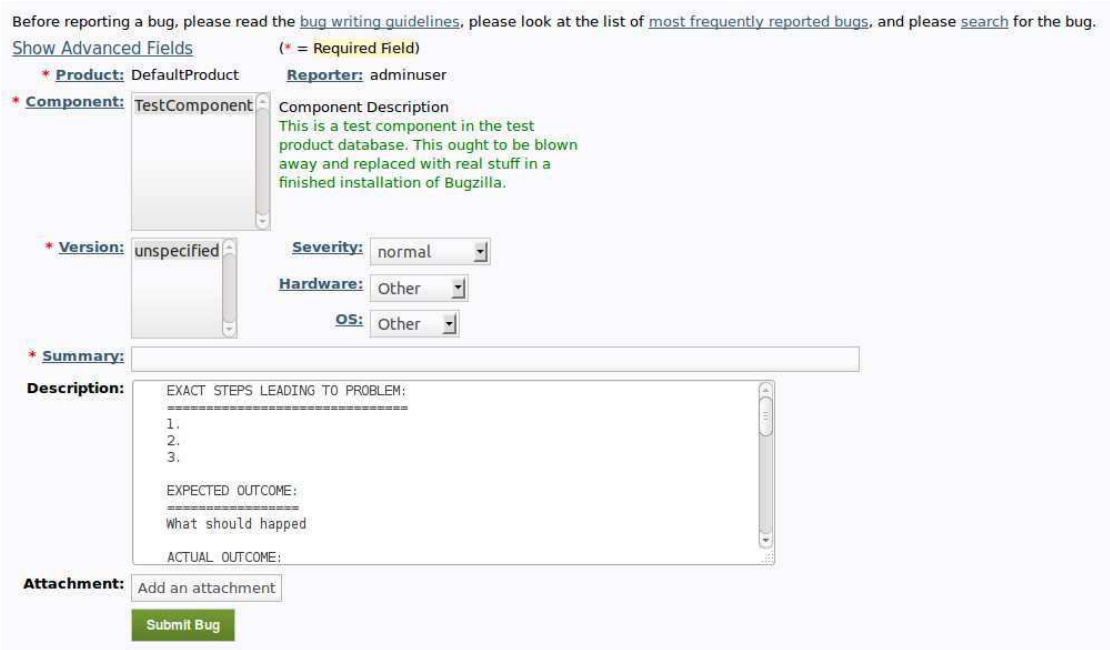

Figure 14: Example bug report

It may be possible that you need to select the product before you enter into this, but in this example, it is called "DefaultProduct". First you select the component in which you found the error, then you select the version where you found the bug, such as week 25. Select the according severity, hardware and operating system from the dropdown menus. Summary is mandatory to

write for the test, and it is similar to the name for the test case. As with the test case name, keep it simple, and descriptive. After it the description section appears, where there are prefilled sections for your data input. Fill out the description sections (you may add your own if you think it is necessary). The software itself thinks that it is optional to put a description, but actually that is the most important part of the whole bug report.

Without the steps written, it is really hard to repair the component, as the person doing it will not know how the bug can be repeated. Expected outcome and actual outcome are self-explanatory. Under user impact you should write how the bug affect the usability of the component. Under reproducibility you should write how often the bug occurs. Extra software/configuration installed is to report any attached component that may affect to the cause of the bug, this is especially related to when first time doing the integration testing. In the package version the task is to report what version the component in which the bug occurred is. Other tool versions are the versions of the other components mentioned in "Extra software/configuration installed" section.

Remember to verify the test multiple times in the case of a bug, so that you can be sure that the bug actually occurs and how many times it occurs.

Exploratory Testing

The previously explained workflow is a planned test process, in which the testing group plans the test process together, and starts testing from there. There is a concept known as exploratory testing, in which a tester looks at the

bug reports, or generates bugs himself, and builds test cases based on that. This method is usually used by more experienced testers.

You can start with two ways: You can either go to the software component itself to find bugs or you can go to Bugzilla to see already made bug reports and add test cases based on those. The latter however may not work, as usually testers report a bug whenever they find one when doing test execution. But there are some that are reported by developers, who do not usually used test cases for them.

APPENDIX 2. HOW TO USE TESTLINK

Contents

- Contents
- Overview
- Workflow
- Creating new Test Projects
- Test Specification
- Test Plans
- Builds
- Test execution
- Requirement based testing
    - Advantages of requirement and risk based testing
    - Requirement dependencies
- Test Reports and Metrics
- Administration

Overview

TestLink is a tool that basically covers everything needed for software testing, and it might seem a bit complicated at first sight. This page is here to help a person that has never seen TestLink before to grasp the essential basics how to use it. TestLink is found by clicking Work Collaboration -> Test Management. All red colored text is a later on edited onto the pictures, and is just to clarify certain things.

Work flow

The basic work flow is:

-> Test Project management**[Create]**

-> **(**Requirement Specification**[Requirement Create, Import, Auto-create Test cases])**

-> Test Specification**[Create Test Cases]**

-> **(** Platform Management**[Create Platforms])**

-> Test Plan Management**[Create new]**

-> **(**Add / Remove Platforms**)**

-> Add / Remove Test Cases**[into test plan]**

-> Builds / Releases**[Create Build]**

-> Assign Test Case Execution**[Who Does Tests]**

-> Execute Tests**[Do Testing, attach to bugs, create bugs from failed test cases]**

-> Test Reports and Metrics**[report of test execution]**

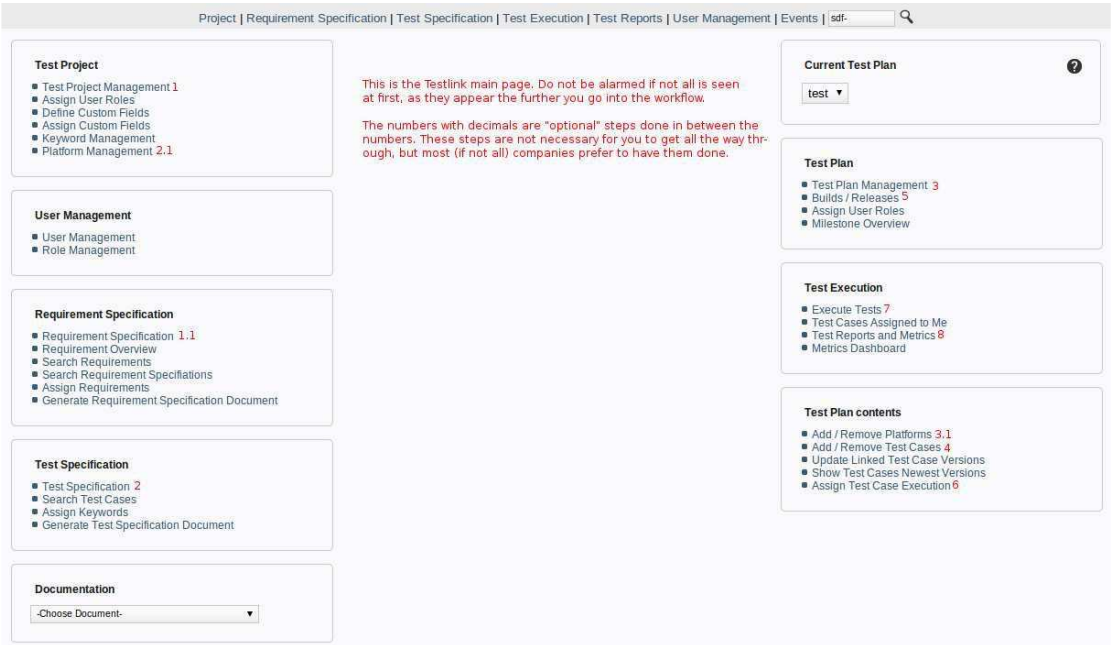

Figure 15: Testlink mainpage

Do note that this is the grand work flow of things, and includes many users

involved doing different things.

Creating new Test Projects

Creation of test project requires "administrator" rights. Start by Selecting "Test Project Management" link on the main page, and press create button.

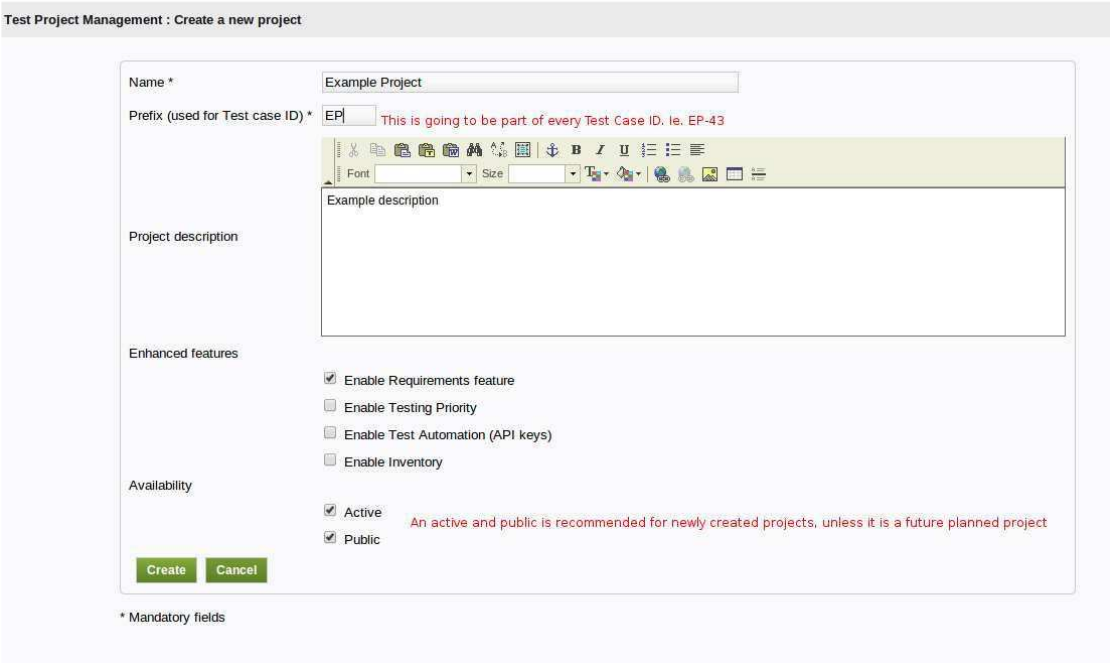

Figure 16: Project creation

Remember, that each test project must have a unique name. The text area is for describing the project object. Several other features are also included

- Administrator can enable Requirements functionality.
- Test prioritization could be enabled to select appropriate testing set in limited time. §
- Test automation support could be enabled as well.

Things to note when creating new Test Projects:

- Deleting test projects is not recommended as the procedure orphans or deletes a large number of test cases.
- Test Plans are the ones that indicate testing at a certain point of time, not test Projects. It is not recommended to create separate Test projects for versions of one product.
- TestLink supports XML or CSV data importing into a Test Project. (more info on this later)

Editing and deleting test project require also "administrator" rights. User can change the name and functionalities mentioned before. User can inactivate the Test Project if it is obsolete, this will make the Test Project invisible to non-Administrator users.
Deletion of a Test project is not recommended! It is not a reversible action. Use inactive instead.

Test Specification

TestLink breaks down the test specification into Test suites and test cases. These levels are persisted throughout the application. One test project has just one test specification Users organize Test Cases into Test Suites. Each suite consists of a title, underlying test cases and possibly other Test Suites. TestLink uses tree structure for Test Suites, much like files are done in most operating systems. You can reorder by drag and drop on the navigation tree.

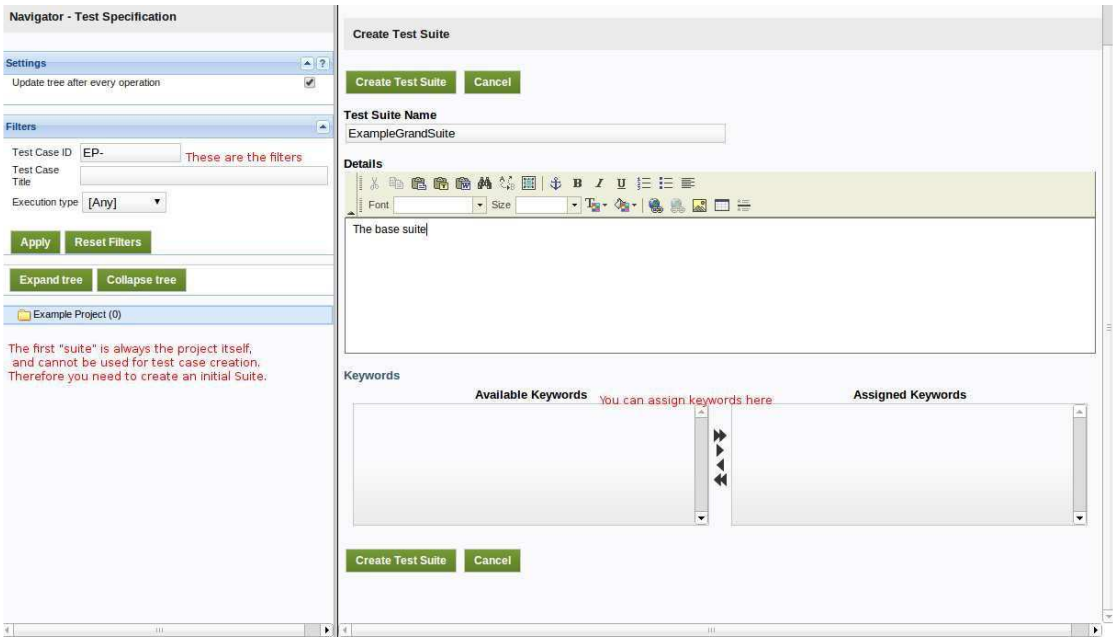

Figure 17: Creating a test suite

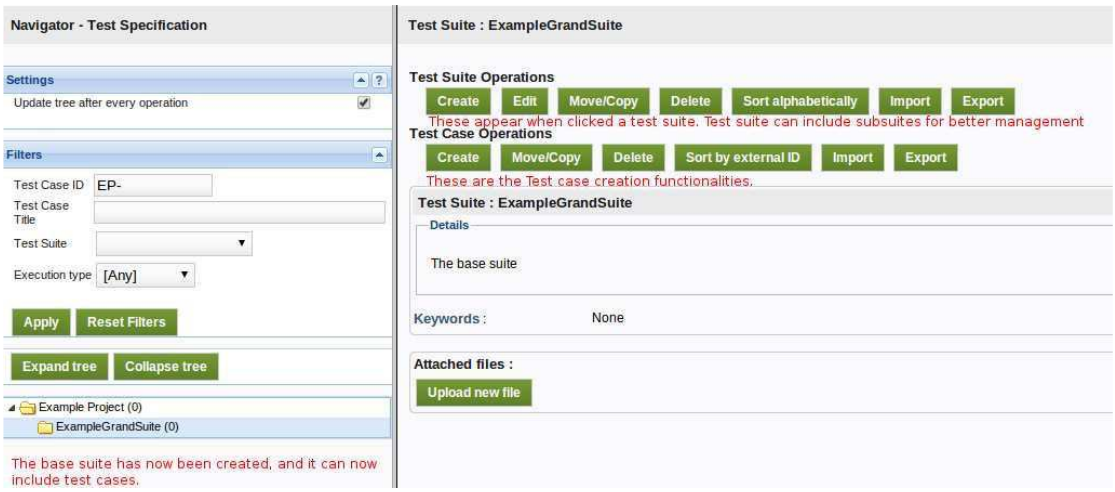

Figure 18: Creating a test case

Test Case is a set of inputs, execution preconditions and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. Test cases consist of:

- Identifier is automatically assigned by TestLink, and cannot be changed

by users. It composes of test project prefix and a counter number.

- Title: Could include a short description or abbreviation
- Summary: Should be really short.
- Steps: describe test scenario (input actions); can also include precondition and clean-up information here.
- Expected results: describe checkpoints and expected behavior of a tested product or system
- Attachments: could be added if configuration allows.
- Importance: Test designer could set importance of the test [HIGH, MEDIUM and LOW]. The value is used to compute priority in Test Plan (must be enabled on Test project management)
- Execution type: Test designer could set automation support of the test [MANUAL/AUTOMATED].
- Custom fields: Administrator could define own parameters to enhance Test Case description or categorization.

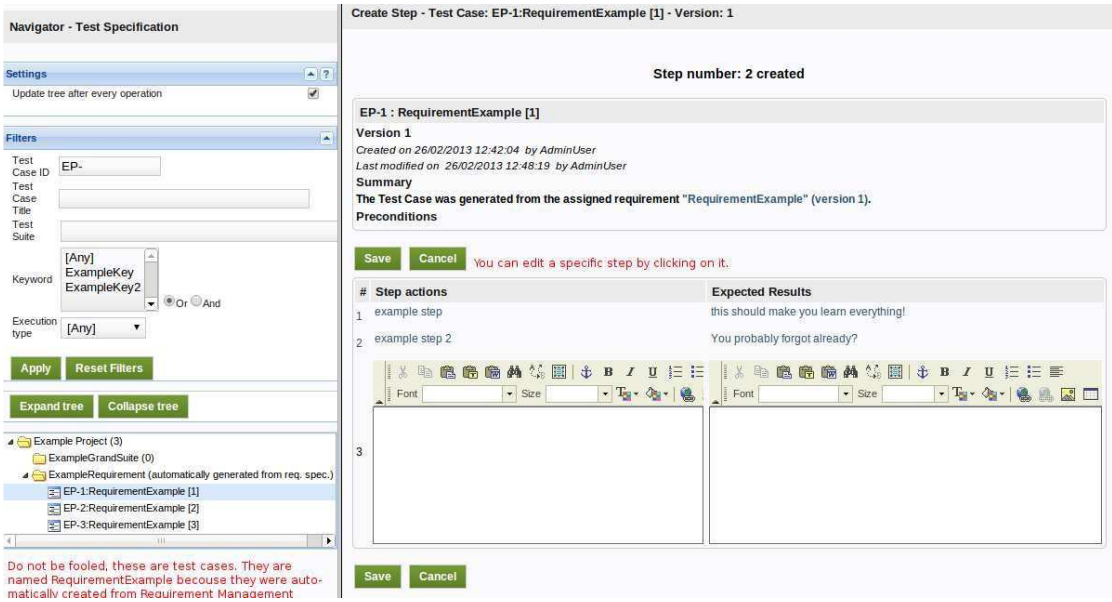

Figure 19: Test case steps and other details

(More later on about Requirement Test Cases)

Removal of Test cases and Test suites can be done by users with "lead" permissions. Do note that removal of a test case will cause the loss of all results associated with them. Extreme caution is advised when deleting cases or suites.

Test cases can be searched by using various search criteria: Identifier, Test case version, Keywords, requirement ID.

Keywords are a categorizing level for test cases. By giving several test cases the same keyword, they can be searched easily by looking for the keyword. First a keyword needs to be created, and then assigned to the test case.

Test Plans

Test plans may be created from the "Test Plan management" page by users

with lead privileges for the current Test Project. Press "Create" button and enter data.

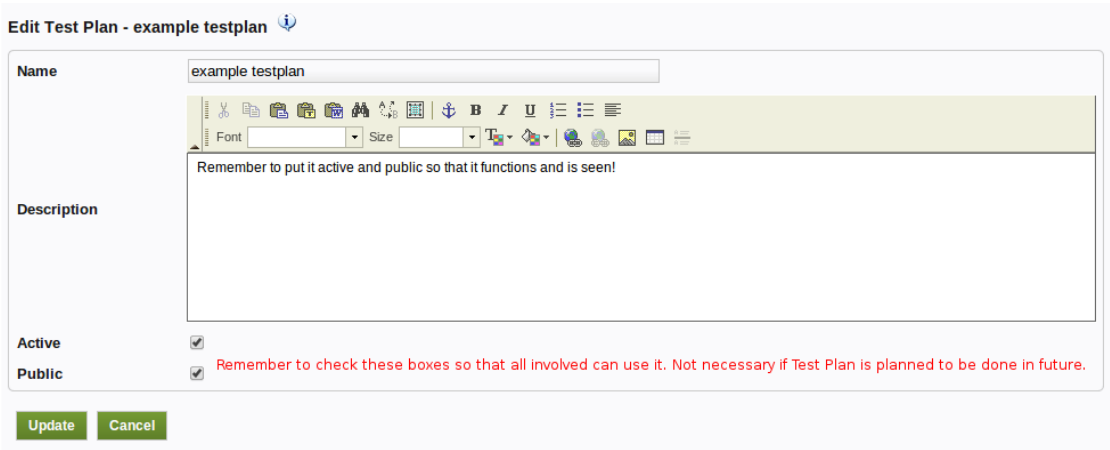

Figure 20: Creating a test plan

Test Plan definition consists from title, description (htm format) and status "Active" and "Open" check boxes. Description should include the next information with respect to company processes:

- Summary/Scope
- Features to be tested
- Features not to be tested
- Test criteria (to pass tested product)
- Test environment, Infrastructure
- Test tools
- Risks
- References (product plan or Change request, Quality document(s), etc.

Test Plans are made up of test cases imported from a test specification at a specific point of time. Test plans may be created from other test plans. This allows users to create test plans from test cases that exist at a desired point in

time. This may be necessary when creating a test plan for a patch. In order for a user to see a test plan they must have the proper rights. Rights may be assigned (by leads) in the define User/Project Rights section. This is an important thing to remember when users tell you they cannot see the project they are working on.

Test Plans may be deleted by users with lead privileges. Deleting Test Plans permanently deletes both the Test Plan and all of its corresponding data, including Test Cases (not in Test Specification), results, etc. This should be reserved only for special cases. Alternatively, Test Plans may be deactivated on the same page, which suppresses display on selection menus in the “main” and “Execute” pages.

Platforms bring the basic idea of testing the test cases on various platforms. The first step is to add them to a test plan by doing it within link "Add / Remove Platforms". Afterwards, simply enter "Add / Remove Test cases" to see that you can add the same test case to each individual platforms.

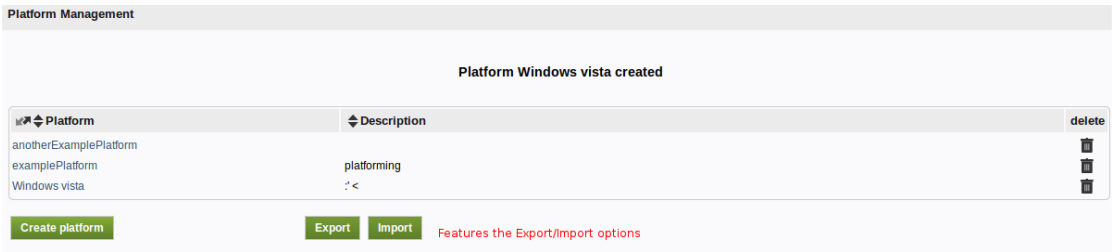

Figure 21: Creating a platform

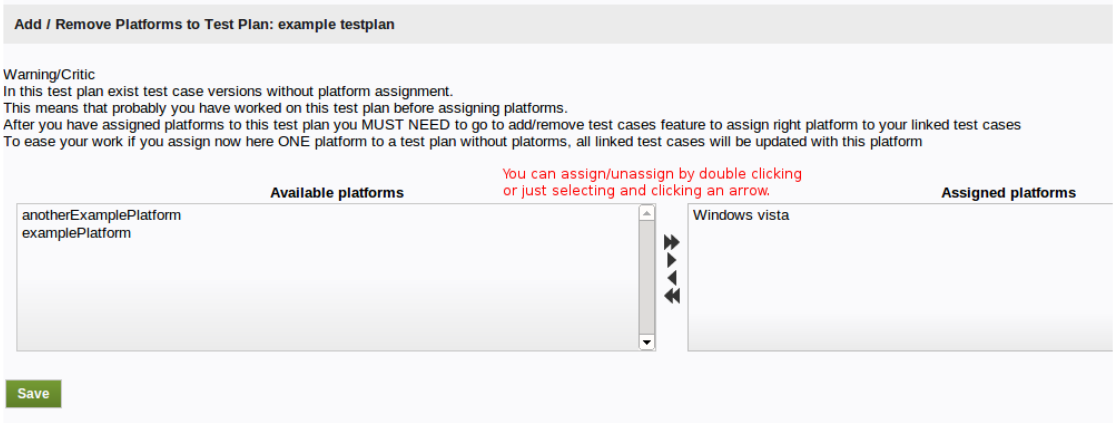

Figure 22: Adding and removing platforms from a project

Test cases are usually added into a test plan by using the "Add / Remove Test Cases" link on the main page. User can choose Test cases via check-box and hit "Add selected" button to define Test set. Clicking on the "check" icon selects all test cases in each test suite. Added test cases are added as their current version, so if it is modified, it needs to be added again.

Removal can be done by users with lead permissions from the same place as they are added to the Test set. Caution: removing test cases from the set will cause the loss of all results associated with them.

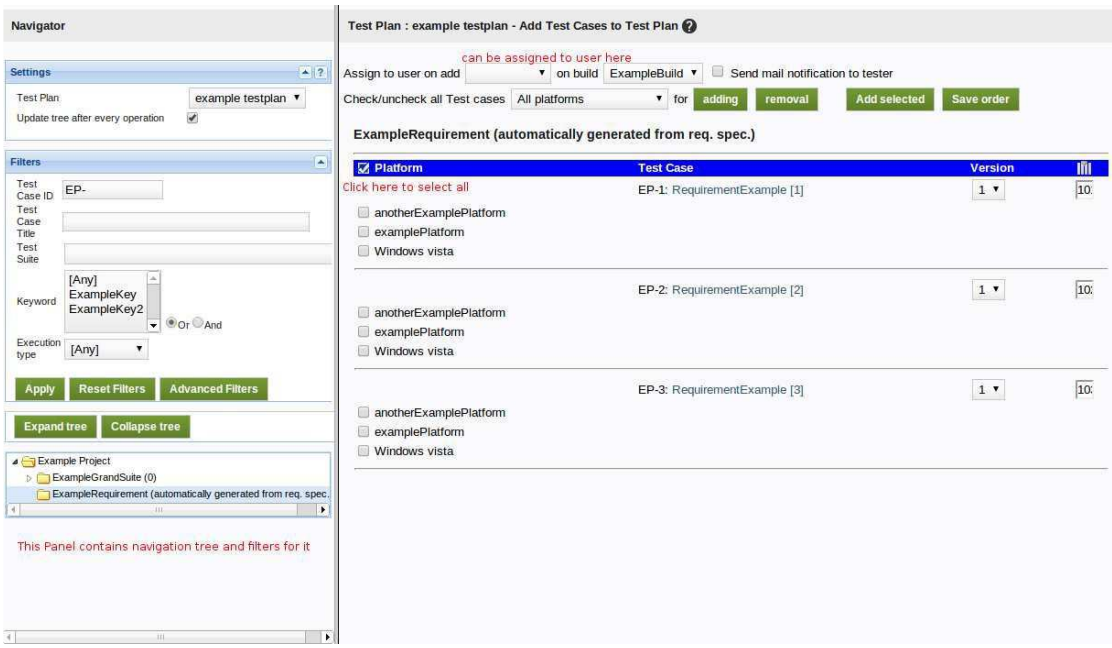

Figure 23: Adding test cases to the test plan

The ones who are to carry out the testing of each test case can be done from the "Assign Test Case execution". Execution assignment affects both the execution and reports, by selecting from the left treemap a test suite, the right side shows all the test cases held within. First you need to check the test cases that you want to assign to someone (checking is done by similar fashion to adding/removing test cases to/from a test plan). Now you can use bulk user assignment to assign a bunch of test cases quickly to a single user. By clicking the test suite checking you can check all test cases contained within. If you do want to add only a couple of individual test cases to varying users, you can assign them to the user from a drop down list at the end of each test case. Click save, and assignments have been stored.

Builds

A user with lead privileges could follow the link "Build management" in the main page.

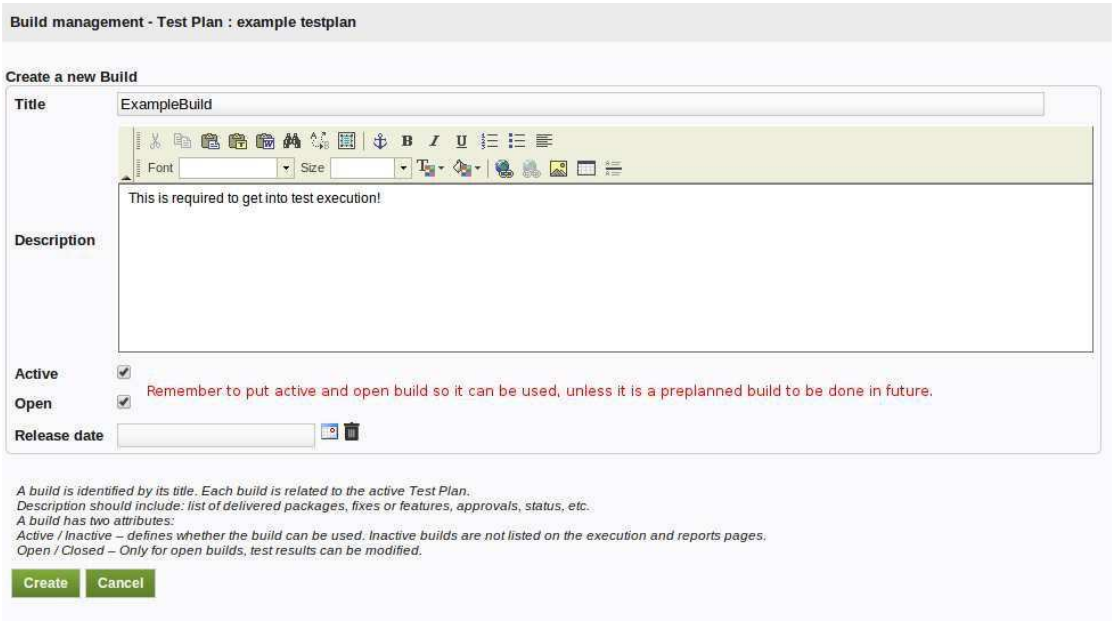

Figure 24: Creating a build

Builds are a specific release of software. Each project in a company is most likely made out of different builds. Testlink uses builds with test cases within to create execution. If no Builds exist for the project, the execution screen will not allow you to execute.

Builds are identified by their titles, and include two states: Active/inactive and Opened/Closed. Active defines if the build is available for Testlink functionality. Inactive is not listed in execution or report pages.

Test execution

Available after:

- A Test specification is written.

- A Test Plan is created.
- Test Cases are added into Test Plan.
- At least one Build is created.
- Testers have appropriate rights for execution to work with this Test Plan.

Select the "Test Execution" from top of the page or "Execute Tests" from the main page. This navigates into the "Test Execution" window, which has a left panel with treemenu Test cases. This treemenu shows the test cases that you've added into the Test plan. Clicking a testcase opens up all data of the test case. Of course this also contains a filter by keyword, Tester, Result or Test priority. The tree menu items will be colored depending on which a test result has occurred on each test case. The test cases have statuses of not run, pass, fail, or blocked. Blocked status is put when Test case is not possible to test for some reason.

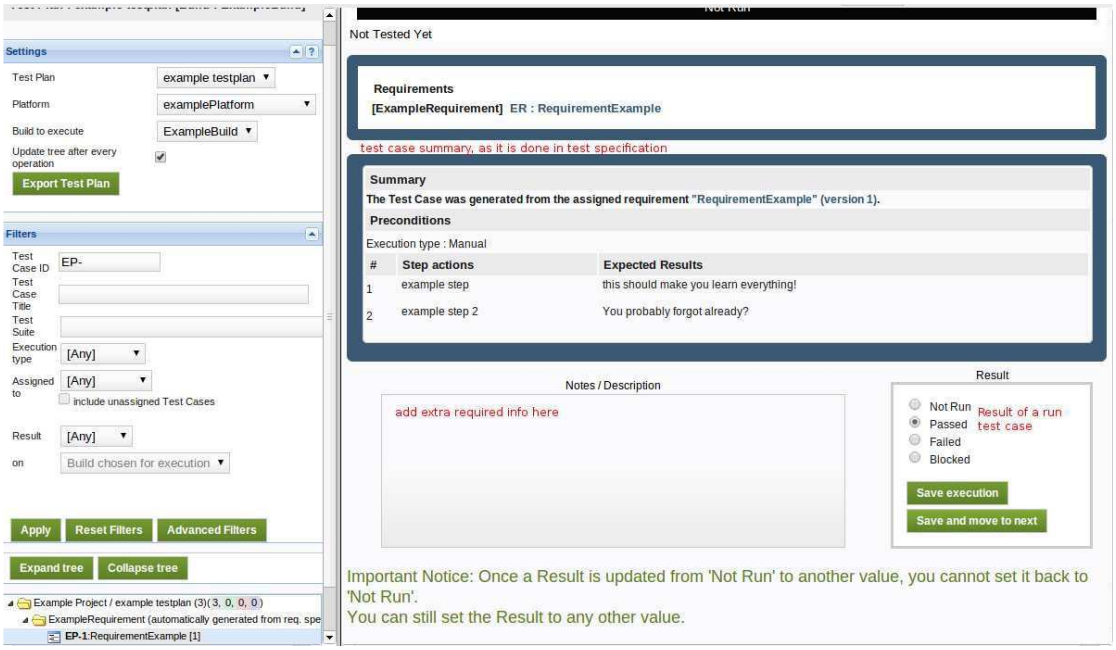

Figure 25: Executing tests

Requirement based testing

To prove that a system is built as specified, testers use requirement based testing. For every requirement, they design one or more Test cases. At the end of the test execution, a test manager can report on the tests that are executed and the requirements are covered. This way client and/or stakeholders can decide whether a system can be proceeded to next phase or can go live. Using a combination of risk and requirement-based testing it is ensured that a system is built as specified from the customer and stakeholders perspective.

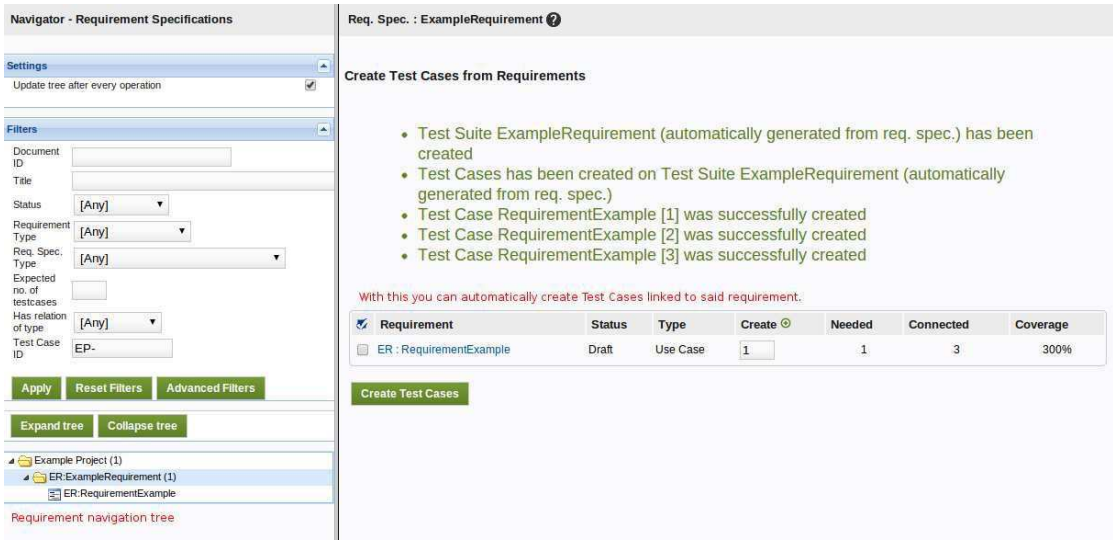

Figure 26: Automatically created test cases from a requirement

Advantages of requirement and risk based testing

- Linked Risks and requirements reveal vague or missing requirements.
- Testing can be focused on the highest risks at first.
- Communicating with the same "language" with the client and the stakeholders. This enables easier reporting.
- Risks and their priority make negotiating on the test project easier when under pressure. The process is streamlined, testing priority is easier.

Requirement dependencies

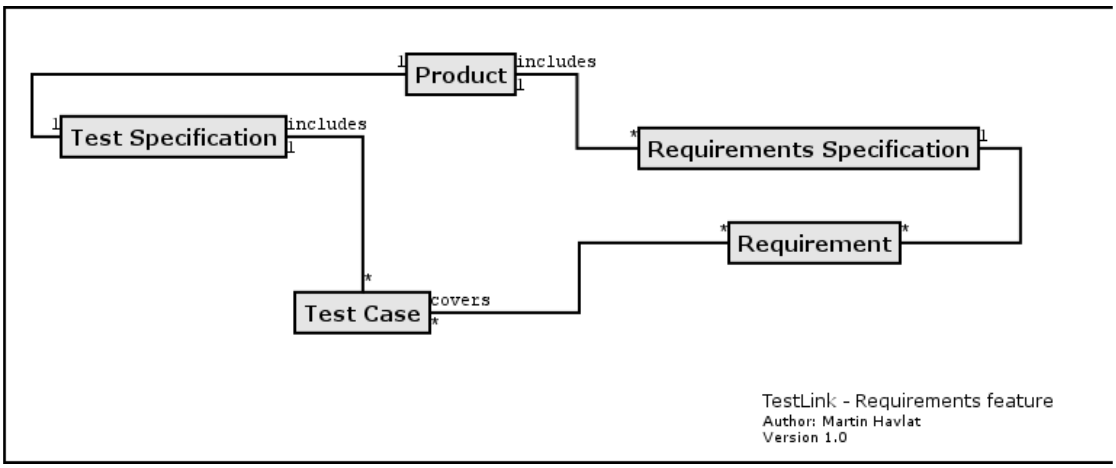

Figure 27: Requirement dependencies

Creating a document with requirements (Requirements feature must be enabled from the Project management):

- Click requirements specification in Main window. The list of requirement specifications is shown.
- Press Create button to create a document
- Adjust Title, Scope and eventually Count of Test Cases. The last parameter is used for statistics. Use only if you have a valid Requirement document but not all requirements are available at the moment in Testlink. Default value '0' means that the current count of requirements in a specification is used.
- Press Create button to add data to database. You can see the title of your new created document in the table of list of requirement specifications.
- Click the title of document for next work.

Requirements can be created/modified or deleted manually within Testlink interface or imported as CSV file.

Test Reports and Metrics

There is no use to perform test cases unless you can report them, and show statistics of a run test plan. You can access this by "Test reports and Metrics" link on the main page, or from the "Test Reports" link always shown on top of the page. The page that is shown includes on the right panel instructions on how to use and report, left panel is used for navigating and filtering (again).

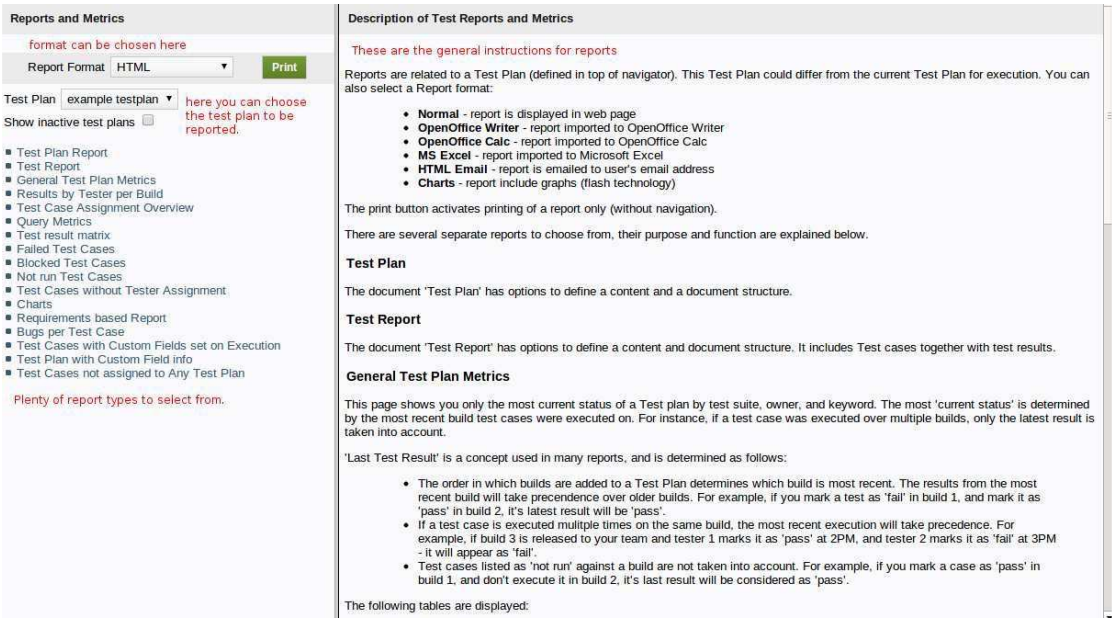

Figure 28: Test report options

Test reports can be generated in one out of six different formats:

- Normal - Report is displayed in a web page (html)
- OpenOffice Writer - Calls OO Writer to show a report
- OpenOffice Calc - Calls OO Calc to show a table based report
- MS Excel - calls microsoft Excel to show a table based report
- MS Word - calls microsoft word to show a report
- HTML Email - send report to user's email address

Currently there are 16 separate reports to choose from.

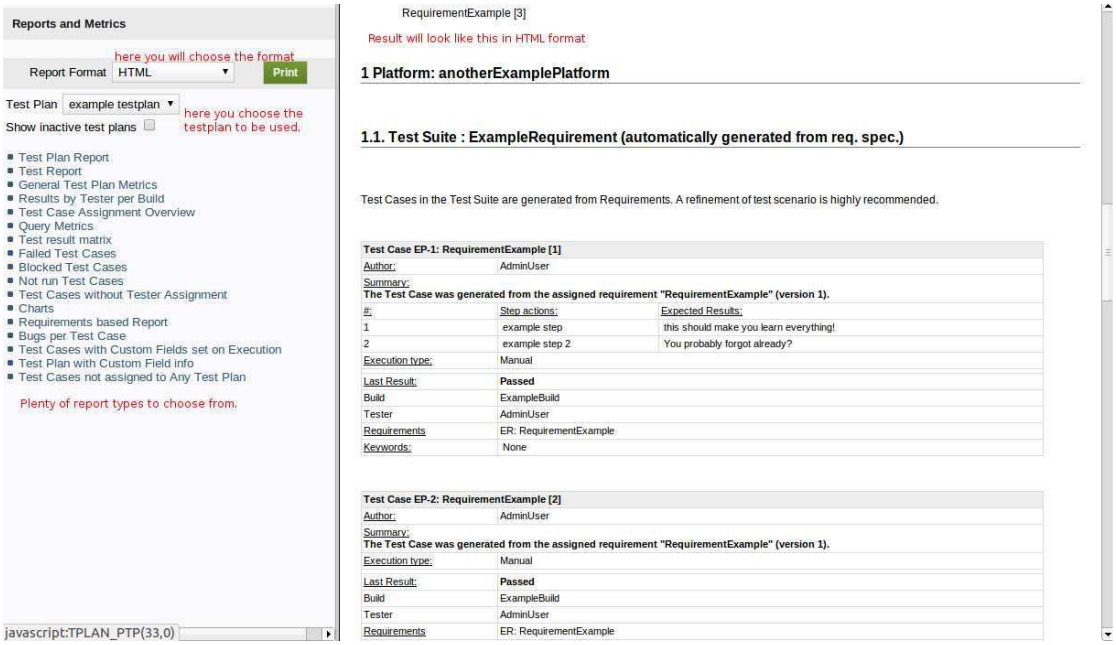

Figure 29: HTML test report

Administration

Every guest can create their own account on the login page. Every user is able to edit their own information via Account setting. The difference to a normal password is created on LDAP inside the FreeNest, and cannot be modified on Testlink itself.

APPENDIX 3. TEST PROCESS RESOURCE PLAN

Team 1	Bug	Team	Platform
scribe1	5	Testing	Ubuntu 12.04 Chrome
person1a	1	Maintenance	Windows 7 Firefox
person1b	2	Feature	Mac Firefox
person1c	3	Feature	
person1d	4	Maintenance	

Team 2			
scribe1	5	Testing	Windows 8 IE 10
person2a	1	Maintenance	Ubuntu 12.04 Firefox
person2b	2	Feature	Android Chrome
person2c	3	Feature	Mac Safari
person2d	4	Maintenance	

Team 3			
scribe3	4	Testing	Windows 7 IE 10
person3a	1	Maintenance	Ubuntu Chromium
person3b	2	Feature	Windows 7 Chrome
person3c	3	Automation	

Team 4			
person4a	4	Automation	Ubuntu Opera
person4b	1	Feature	Windows 7 IE 9
person4c	2	Feature	Android Chrome
person4d	3	Automation	

Bugfix Priority	#		
person1a	1		
person2a	2		
person3a	3		
person4b	4		
person1b	5		
person2b	6		
person3b	7		
person4c	8		Ei mukana
person1c	9		
person2c	10		
person3c	11	2pv	X
person4d	12	2pv	X
person1d	13		
person2d	14		
scribe1	15		
scribe2	16		
scribe3	17		
person4a	18		

Test configuration	Comment	Team
Android 4.2.2	Asus Nexus 7	Team 4
FreeNest 32bit + Ubuntu 12.04 + Chrome		Team 1
FreeNest 32bit + Ubuntu 12.04 + Chromium		Team 3
FreeNest 32bit + Ubuntu 12.04 + Opera		Team 4
FreeNest 32bit + Windows 7 + Chrome		Team 3
FreeNest 32bit + Windows 7 + IE 9		Team 4
FreeNest 64bit + Ubuntu 12.04 + Chrome		Team 1
FreeNest 64bit + Ubuntu 12.04 + Firefox		Team 2
FreeNest 64bit + Windows 7 + Firefox		Team 1
FreeNest 64bit + Windows 7 + IE 10		Team 3
FreeNest 64bit + Windows 8 + IE 10		Team 2
Mac OS + Firefox		Team 1
Mac OS + Safari		Team 2
Android 4.0.x + Chrome	Asus Transformer	Team 2
FreeNest 32Bit + Windows 7 JAMK + FireFox? ESR 10		NarsuMan?
FreeNest 32Bit + Windows 7 JAMK + IE 9		NarsuMan?
Android 4.0.4 + Chrome	SAMSUNG GT-P7500	

Figure 30: Resource plan

APPENDIX 4. WORKFLOW CHART

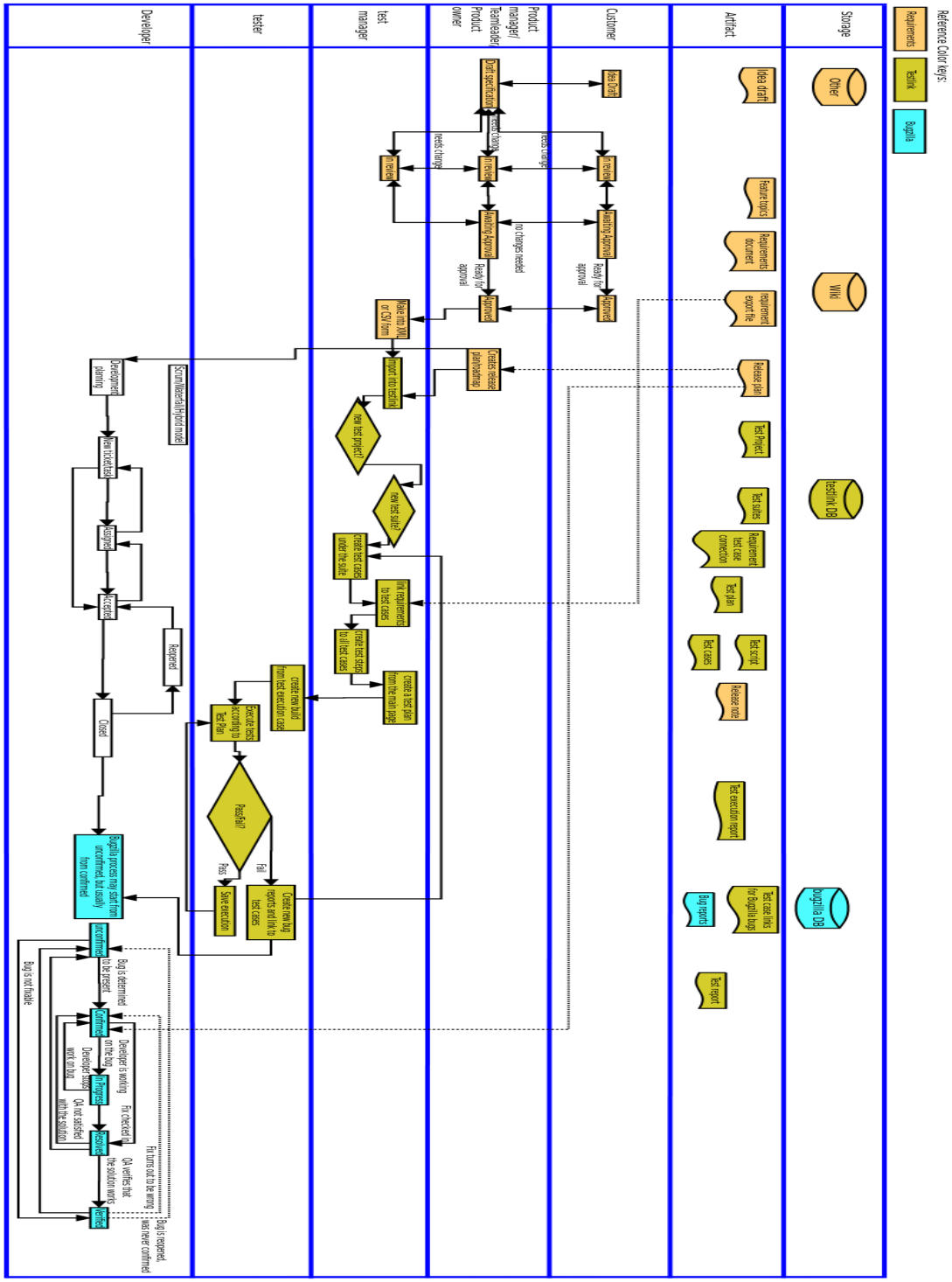

Figure 31: FreeNest Workflow as Swimlane chart