A guideline for requirements management in GitHub with lean approach

Risto Salo

University of Tampere
School of Information Sciences
Computer Science
M.Sc. Thesis
Supervisor: Timo Poranen
June 2014

University of Tampere
School of Information Sciences
Computer Science
Risto Salo: A guideline for requirements management in GitHub with lean approach
M.Sc. Thesis, 71 pages, 1 index page
June 2014

GitHub is an online platform for collaborating and sharing code. In recent years its popularity has increased widely and both people and different size organizations utilize its powerful features. One of those features is a lightweight issue tracker which is meant for handling features and identified errors of software. The issue tracker is intuitive to use and simplifies a lot of things, but what if it is to be used as a requirements management tool?

Requirements management is the last step of requirements engineering process. This process aims to identify, document and manage all the requirements valid for the software product. Requirements management focuses on tracking how requirements are fulfilled and keeping the information associated to requirements intact.

This M.Sc. Thesis represents a semi-formal guideline for handling requirements management in GitHub. The guideline is evaluated on a theoretical level by comparing how well it accomplishes requirements management objectives and fits in an agile software development environment. To assess the suitability for the agile approach, the guideline is compared against lean principles. Lean principles originate from Toyota's successful manufacturing practices and are converted to usable form in a software development. On a practical level a case study is carried out where the guideline is put into a real use.

Similar research hasn't been done before, making the results novel. Both the theoretical assessment and the practical case study point out that the guideline and GitHub are well-suited for requirements management in an agile environment.

Keywords: Requirements management, Lean Software Development principles, GitHub, agile

Preface and Acknowledgements

I have always been fairly interested in computers and technology involving them. At some point I thought that this interest would fade away when I grow older but it never did. Studying Computer Sciences has been exactly what I have had passion for. To be able to finish this academic study journey with M.Sc. Thesis that successfully achieves creating something new is like finishing a marathon with great time!

I would truly like to thank the members of the case study project group for accepting my guideline into trial. Your comments have inspired me to believe that this guideline has what it needs to continue its life outside Thesis.

My most sincere gratitude and thanks go also to my good friend Erkki Heikkonen. Thank you for introducing me to the Lean Software Devlopment and sharing your thoughts and experiences with me during the last few years.

Without a doubt Professor Mikko Ruohonen and Doctor Timo Poranen from the University of Tampere receive my deepest gratefulness for providing support and insights for my studies of Master Degree and the Thesis. Your feedback has given me invaluable knowledge through the whole journey.

Big thanks to all others – friends and colleagues – who have shared your sympathies with me when the writing process of the Thesis was struggling.

Last but not the least I want to express my uttermost admiration for my wife Pia. You have lit my path, helped tackle obstacles when needed and let me spend hours for polishing the Thesis so that I am happy with it. With the warmest love, thank you.

Ylöjärvi, June 4th, 2014

Risto Salo

Table of content

1. Introduction ... 1
2. Agile methodologies ... 3
 2.1. The aspects of agilism ... 5
 2.2. Lean Software Development ... 7
3. Version control systems ... 12
 3.1. GitHub terms and components ... 13
4. Requirements engineering ... 21
 4.1. Agile requirements engineering ... 25
5. Project management ... 28
6. Managing requirements in GitHub – The guideline ... 32
 6.1. Design Science Research approach ... 33
 6.2. How to use the guideline ... 34
 6.3. Separation of tasks and requirements ... 36
 6.4. Hierarchy between requirements and tasks ... 36
 6.5. Creating issues ... 38
 6.6. Status tracking and traceability of issues ... 43
 6.7. Combining version control and wiki to the issues ... 43
 6.8. Updating and maintaining issues ... 44
 6.9. A bug report and an enhancement proposal ... 45
 6.10. Changes in the guideline ... 46
7. Evaluating the guideline ... 48
 7.1. The case study ... 48
 7.2. The results of the case study ... 49
 7.3. Implications of guideline's practices and recommendations ... 51
 7.4. The main effects for requirements management and lean principles ... 53
 7.5. Suggestions for developing features of GitHub's issue tracker ... 56
 7.6. Suggestions for developing the guideline ... 57
8. Discussion ... 59

References ... 63
Appendix 1 – Compact introduction to the guideline ... 70

1. Introduction

GitHub is a rapidly growing Internet based service which describes itself as a social coding platform. In the beginning of 2013 GitHub had almost 5 million source code repositories, and in a year it doubled the figure to 10 million. An average of 10 000 new users subscribe every weekday. GitHub is not a place just for individual users; notable organizations have also figured out the power of GitHub. Such are for example Amazon, Twitter, Facebook, LinkedIn and even The White House.

It's no wonder that GitHub's powerful, lightweight and easy-to-use features have attracted so much positive attention. The features provided include an issue tracker and wiki. The problem however is that how could these features be used to manage requirements inside GitHub without a need to utilize another tool. This Thesis tries to solve this problem by introducing a guideline for requirements management process applying only GitHub's native features. The guideline was put to use in a case study project and also evaluated based on the objectives of the requirements management and lean principles.

Lean ideology derives from Japanese car manufacturing from the middle of 1950s. In the beginning of 21st century "being lean" has received a lot of hype from software engineering field after Poppendiecks' converted its principles to a suitable format for software development. The enthusiasm towards lean started to cumulate a little after agile methodologies hit through. Although agile methodologies originate from frustration of waterfall like methods that didn't do well in an era of rapid software projects and volatile requirements, their goals are aligned with lean principles. For this study lean principles offer an insight how well the guideline shapes into an agile environment that doesn't necessarily rely on a defined agile methodology like Scrum or Extreme Programming.

Similar topics couldn't be identified either within the scientific researches or from practitioners making the results novel. Although the problem domain and its solution are quite specific, the evaluation succeeds combining lean principles to the objective of requirements management. The case study reveals that the whole project team values the guideline's defined approach. On the other hand the evaluation points out that the guideline and GitHub achieve also in a theoretical level. Overall the results prove that GitHub can successfully be used for agile requirements management when a systematical guide is applied.

This M.Sc. Thesis is divided into 8 Chapters. Chapter 2 examines agile methodologies' roots and meaning. Lean Software Development is also presented. Version control systems especially GitHub and its relevant components are studied in the Chapter 3. Chapter 4 focuses on requirements engineering process and how it is applied to the agile methodologies. Project management is gone through in the Chapter 5. The product of this

Thesis – the guideline for managing requirements in GitHub – is represented in the Chapter 6. The practices, recommendations and suggestions of the guideline are discussed. The evaluation of the guideline from the theoretical and practical point of view is in the Chapter 7. The last Chapter summarizes discussion and conclusions of the Thesis.

2. Agile methodologies

Agile – according to dictionary [Dictionary.com, 2014] denotes "quick and well-coordinated in movement; active; lively; marked by an ability to think quickly; mentally acute or aware" – methodologies can be seen as a reaction to the cumbersome and strict software development processes utilized in the 1980s and 1990s [Cohen et al., 2004; Sommerville, 2007]. What is crucial to these new methods is the shift in emphasis and values to the neglected aspects of the traditional ways.

As Cohen et al. [2004] state "software improvement process is an evolution in which newer processes build on the failures and successes of the ones before them". So to better understand from where agile methodologies and the whole agile movement arose, one must go back to the 1980s.

Back then a process oriented or, as some practitioners called it, plan driven development was believed to be the right way to achieve a better software product [Sommerville, 2007]. Through 1980s to the early 1990s was a golden age for rigorous project planning, documenting and analyzing. A good example of a methodology that implements these aspects is the Waterfall Model. This model began with a full analysis of user requirements then continued with establishing a definitive list of features and both functional and non-functional requirements. After all those first steps were thoroughly documented, engineers could start working with the design collaborating with other experts to create architecture for the software. Finally the design was implemented, tested and shipped [Beck, 1999].

In theory this avenue of approach sounded good, but in the reality it didn't manage itself as well as advertised. For developing large, long-lived software systems which may have been made up of many individual programs, this solution was working, especially if system was considered critical [Sommerville, 2007]. However, when this approach was applied to small and medium-sized business projects, it quickly became too overwhelming and overall impossible to succeed with [Cohen et al., 2004; Sommerville, 2007].

The two main reasons could be identified for failures of the Waterfall Model in these situations. Firstly, customers couldn't make up their mind about what they wanted from the product. Extensive user requirements collection or mockups didn't help, if customer wasn't sure what they exactly needed. Secondly, requirements were prone to change in the middle of development, and incorporating the changes wasn't modeled to the Waterfall Model's processes. [Cohen et al., 2004; Sommerville, 2007] As such, the agile movement was a reaction for the traditional software development process that was in large degree dependent on documentation and heavyweight processes [Beck, 2000].

To counter these misfits incremental, iterative and spiral methodologies were brought up [Cohen et al., 2004]. In incremental method the requirement process is conducted before development, just like in the Waterfall Model, but instead of working requirements as a whole, the requirements are split into smaller, independent increments. The increments are then worked with and they can overlap, saving precious time. Iterative and spiral methodologies take a little different stance. Iterative development splits a project into iterations, which build on each other. Every iteration is like the Waterfall Model in a compressed form consisting of analysis, design, implementation and testing, in this exact order. The process starts from the bottom, delivering first the very basic features, adding more in subsequent iterations. The Spiral Model differs from the iterative in its prioritization model. When in iterative development prioritization occurs by functionality, in the Spiral Model prioritization is done by requirements' risk assessment. [Cohen et al., 2004; Sommerville, 2007]

Both of these approaches offer far better responsiveness than the plain Waterfall Model and are bowing more to the direction of agilism. Still, to some practitioners, these changes from the Waterfall Model weren't enough and they believed that things could be done even more optimally. Exhaustive planning and analysis coupled with the full documentation made iterative and spiral processes still too heavyweight to be called *agile* [Cohen et al., 2004].

The unhappy practitioners went on their own ways to find out how to better deal with the software development process. One of them was Ken Schwaber – founder of Scrum methodology – who realized [Cohen et al., 2004], that a truly agile process accepts change rather than tries its best to hang on to the predictability [Schwaber, 1996]. Mary Poppendieck was introduced to Lean Manufacturing, utilized by Toyota Company in Japan. Later she and her husband Tom converted those practices from car manufacturing to a usable form in software development [Cohen et al., 2004]. Kent Beck and Ron Jeffries went and saved Chrysler's almost failed payroll project (Chrysler Comprehensive Compensation, C3). The 180 degrees turnaround from almost a failure to a success was the first time Extreme Programming (XP) was used [Highsmith et al., 2000]. Alistair Cockburn, hired by the IBM Consulting group to come up with an object-oriented development method, developed the Crystal Methods based on the interviews and best practices of IBM's development teams [Highsmith et al., 2000]. Although these models were founded independently and separately from each other they lead a way to *Agile Software Development Manifesto* (from now on just Agile Manifesto) that would become the pinnacle of the agile movement and mark the new era influenced greatly by agile perception.

2.1. The aspects of agilism

But what went wrong with the Waterfall Model and other traditional models in the first place? Some studies take a very critical attitude towards these traditional development methodologies. Such studies are for example one conducted by Nandhakumar and Avison [1999] who state that traditional methodologies "are treated primarily as a necessary fiction to present an image of control or to provide a symbolic status." Truex and others [2000] go even further claiming the traditional methods to be "merely unattainable ideals and hypothetical 'straw men' that provide normative guidance to utopian development situations".

Others are more conservative on their views. For example Sommerville [2007] argues that although agile methods are suitable for certain cases, there are others – e.g. critical systems development – which processes like those in the Waterfall Model are better suited for. Overall we must realize that the business world where software development occurs has evolved from what it was in 80s or 90s. Internet software and mobile applications are currently hot trends and in their world. One must be able to handle the unstable requirements and sudden changes [Cohen et al., 2004; Sommerville, 2007], but still manage to deliver a functioning product in time. With traditional methods, it is likely that in a fast-moving business environment the finished product becomes immediately useless because business needs have changed [Sommerville, 2007]. Rapid development and delivery are key aspects for successful projects in this environment.

The Agile Manifesto, published by a group of software practitioners and consultants in 2001 is not a direct guideline on how to handle software development process. Rather, it declares the core principles and values that should be honored in agile methodologies. The manifesto itself contains twelve principles which can be summarized into four core statements about what to value:

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

These statements don't negate the value of the items on the right side, but the emphasis has moved from them to the items listed on the left. [Agile Manifesto, 2001; Cohen et al., 2004; Sommerville, 2007]

What the four core values really mean then? The first statement points out that the human beings behind the project are what matter instead of defined processes and tools. Current agile practices enforce this by emphasizing close team relationships and working space coupled with other team spirit boosters. Many practitioners talk about self-organizing meaning that developers should be left with relatively free hands to figure out how to work the best way. The second denounces that the goal of a software development

process is to deliver a defect free, working product. This can be achieved with frequent – usually bi-monthly or monthly - release intervals. The code base should be kept as simple and straightforward as possible to reduce the burden of documentation. The third statement highlights the importance of producing business value from the get-go of the project, thus decreasing the risk of going side rails. Contract negotiations should be considered as a way to build a working relationship with the customer, though contract itself shouldn't be neglected. The customer should be closely involved through the development. The final fourth urges the development team and the customer to be ready and authorized to do adjustments and changes when need for them arises during the development process. [Cohen et al., 2004; Sommerville, 2007]

The four core values wrap up the essence of agilism quite well. Sommerville [2007] and Miller [2001] would both add the incremental delivery and iterative approach as key aspects. Favaro [2002] also suggests that the iterative development paradigm is the dominating factor in agile processes. It should be observed that the ideals and paradigms behind the Agile Manifesto and agile methodologies are nothing new [Cohen et al., 2004]. As discussed earlier, agilism just emphasizes different factors and through that creates an opportunity to tackle the obstacles present in the traditional methods used in modern business world.

Common to all agile practices is simplicity and speed. Therefore a method can be considered to be an agile when: its development is incremental (quick releases of small batches of functionality), cooperative (close communication between developers and the customer), straightforward (the method itself is easy to learn) and adaptive (embrace the change). [Sommerville, 2007]

As with any new movement or ideology, the agile movement also has its advocates and adversaries. As Dybå and Dingsøyr [2008] have proved in their systematic review, the scientific evidences behind agile success stories are still very scarce, yet the "true" practitioners are highly confident to the powers of agile methods. In fact in the beginning of the 21st century, there was no literature describing a failed project that used an agile method. Poor projects were observed and reported, but their problem was a negligent use of agile methods, not the methods themselves [Cohen et al., 2004].

One interesting field is how agilism copes with the CMM(I) model. CMM (Capability Maturity Model) is a model that offers a way to evaluate company's processes and rank the company in a scale from one to five [Cohen et al., 2004]. CMMI (Capability Maturity Model Integration) is the updated version of CMM, and it has been made even more abstract in order to be more applicable. In the beginning of the agile movement part of the practitioners felt that the whole CMM model contradicted with agile practices. The criticism was that CMM was seen as a tool that wanted the software development to be a defined process in which agile practitioners didn't believe [Highsmith, 2002]. The heavy focus on documentation – the same aspect that was appalling in the traditional

development paradigms – didn't help the fundamental division between CMM and agile methods. However, the man behind CMM, Mark Paulk was already in 2004 quite positive about agile methods. His opinion was that agile methods address many CMM level 2 and 3 practices [Paulk, 2002]. Paulk was also aware of the problems caused by the heavy documentation and points out that documentation should be kept to "the minimum useful set" but it should exist nonetheless [Paulk, 2002]. The co-existing between agile methods and CMM is possible, if they are considered to adhere to different roles in software development. In fact, CMMI version 1.3 – released in 2010 – has included some, though not all, agile practices as part of the CMMI itself [Linders, 2011].

So called "traditionalists" have also questioned the essence of agilism. Some – like Steven Rakitin – even claim that it is a step backwards to disorder; an excuse to use hacker like practices [Rakitin, 2001]. The accusations are fought back with arguments that accusers are ignorant of the methodologies [Agile Manifesto, 2001]. It is also known that agile method tend to neglect the architecture of the system [McBreen, 2003]. Testing can become a hindrance due to the obstacles in setting up the test environment with all the dependencies [Svensson and Host, 2005]. On the team level, developers need to be highly qualified to get the most out of the agilism [Merisalo-Rantanen et al., 2005].

Either way, it is clear that the agile methods have opinions from the both sides. Hawrysh and Ruprecht [2000] have made an observation that should be noted: a single methodology can't work in every possible case, thus project management should first identify the nature of the project and then decide the best methodology to use. McCauley [2001] has stated the point even further by arguing that there is a need for both the traditional and the agile methods. Several experts agree with this [Glass, 2001]. Cockburn [2000] summarizes this to a punch line: "having multiple methodologies is appropriate and necessary". This is worth remembering because a single agile methodology doesn't necessarily cover the whole software development process. For example Scrum concentrates mainly on project management duties. With right choices for the right needs, it is possible to get the best result [Cockburn, 2000].

2.2. Lean Software Development

The ideology behind the Lean is considered to be started in the mid-1900s in Japan from the founder of Toyota, Kiichiro Toyoda [Bocock and Martin, 2011]. Toyoda had a vision of a company that could respond to the changing markets, produce its product in a right time and give control and responsibility to its employees. Unfortunately Toyoda passed away before his vision could be fulfilled. Taichii Ohno decided to continue the work Toyoda had started, and ten years later – in collaboration with Shigeo Shingon – he launched a new system called "Toyota Production System" (TPS). This system was based on the original ideas of Kiichiro Toyoda.

TPS was supposed to enhance Japanese competitiveness against American mass production lines, especially in the car industry. The method Americans used wasn't applicable to Japan where demand was greatly smaller and costs higher [Bocock and Martin, 2011; Poppendieck and Cusumano, 2012]. Americans tended to "push" as much as they could out of the production line in to the inventory. This is why TPS was designed with the customer in mind: every production step needed to produce value to the customer. Every step that didn't generate value, Ohno deemed unnecessary. Secondly, Ohno wanted to keep the inventory as low as possible, so that there wouldn't be any extra at any given time. On the other hand he also wanted to be able to deliver quickly [Bocock and Martin, 2011]. With these factors in mind the two key aspects of TPS were formed: deliver in the right time the right amount (Just-In-Time Flow, JIT) and stopping the work immediately when problem was observed in the production (Stop-The-Line, Jidoka). However, TPS was overlooked and its principles tried to apply without the grasp of the whole picture until 1970s. The oil crisis that stroke changed this situation. Toyota survived from the crisis quickly and it convinced other Japanese manufacturers to truly learn and use TPS [Poppendieck and Poppendieck, 2007].

In 1990 TPS received a new name in the book *The Machine That Changed The World* [Poppendieck and Cusumano, 2012]**.** TPS became better known as Lean manufacturing. In 2005 Mary and Tom Poppendieck released a book *Lean Software Development: An Agile Toolkit* [2003] in which the seven principles of lean manufacturing eliminating waste, optimizing the whole, building quality in, learning constantly, delivering fast, respecting people and deferring commitment were modified to the boundaries of software development [Poppendieck and Cusumano, 2012; Poppendieck and Poppendieck, 2007].

According to the National Institute of Standards and Technology Manufacturing Extension Partnership's Lean Network, lean is: "A systematic approach to identifying and eliminating waste through continuous improvement, flowing the product at the pull of the customer in pursuit of perfection" [Kilpatrick, 2003]. This is pretty accurate summary of what Poppendiecks' lean means in the software development world. In the following paragraphs I'm going to present the aforementioned seven principles by Poppendiecks'. The principles are based on the articles by Bocock and Martin [2011], Poppendieck and Cusumano [2012] and books by Poppendieck and Poppendieck [2007] and Russo [2010].

Eliminating waste. In the lean world, waste is something that doesn't create direct value to customer or add knowledge about how to deliver that value more efficiently. This of course requires that companies have identified what their customers really want. After the value has been established the removal of waste can begin. In order to do that, companies must see the waste. Value stream maps and other such tools and methods can help in this process of making waste visible to all. Usually waste is found from activities that don't directly contribute towards the product. Poppendiecks' have listed the seven main reasons for waste.

1. Partially done work. Partially done software ties up unnecessary resources and investments thus causing waste. The way to cope is to reduce this kind of work.
2. Extra processes. For example paperwork can be very time consuming and usually it becomes obsolete or degrades during a time. If paperwork is a necessity, it should be kept as short and high-level as possible. Replacing methods are to be considered.
3. Extra features. Every line of code adds to the complexity and maintainability of the product and is a plausible risk for defect. That's why only the features needed should be implemented.
4. Task switching. Assigning people to overlapping projects is a big waste because they cannot concentrate on one thing and therefore have to split their time. The best way to deal with this is to apply one person to one project.
5. Waiting. Delays are a common thing, but they generate absolutely no value.
6. Motion. Moving from place to place causes interruptions and a loss of focus. Development teams are urged to work in a single open room, reducing the movement.
7. Defects. The waste caused by a defect depends on the impact of the defect and how long it has gone unnoticed. Testing and integrating from the beginning of the project are ways to tackle defects.

Optimizing the whole. Developers should see beyond the software they are implementing. It is rarely so that the software itself creates value to the customer: it is the context of a larger system that brings the needed value. For example the reason behind an ecommerce web site is to sell products. Identifying what really creates the value for customer is not easy job but it is crucial for succeeding. This could be summarized to one sentence: "seeing the whole picture". It is also valid for software engineering experts who may tend to optimize the subsystem they have best knowledge about, ignoring the optimization of the whole system. This should be the way the team thinks of every matter. When problem arises it is not enough to fix the symptoms, the team should be curing the root cause, so that it never happens again.

Building quality in. A high quality product should always be the target. This is a product that has no defects, works as it was intended to and satisfies the customer's needs. The goal is achieved by letting the communication flow through the developers to the customer and back, and also between the developers themselves. When people communicate with each other, knowledge and expectations transfer quicker. The forefront software development techniques that allow a defect free product are continuous integration and testing. Defects are caught earlier enhancing the overall performance of the development.

Learning constantly. Software development is in its essence all about creating knowledge and embedding this knowledge in a product. Poppendiecks' present two kind of learning approaches: learn first and learn constantly. In the "learn first" approach multiple options for expensive-to-change decisions are sought. Such decisions consider for example programming language and system architecture. Due to multiple options, the one that optimizes the best can be selected. The second way – "learn constantly" – means that the system is built with the minimum set of features and then it is continuously improved through short iterations and customer feedback. According to this principle it is also in the interest of the development team to learn collectively and share knowledge between each other.

Delivering fast. When fast delivery is the objective, it is wrong to think the software development process as a project. Rather it should be thought as a steady flow of small changes through the whole process (design, development, delivery). Rapid iteration and feedback cycles are the main tools to ensure the fast delivery. The more feedback, the further the product can be improved. This enforces the other principle – deferring commitment – because fast delivery allows customers to delay the decisions as long as possible. Once customer has made the decision, the team should try to implement it as soon as possible. In fact – as Russo points out – the easiest way of keeping customers from changing their minds is to deliver so fast that they have no time for pondering.

Respecting people. In a lean organization, due to the speed, decisions need to be made fast. To ensure this, the development teams should be highly involved in the decision making process: they often have better knowledge and experience than any other part of the organization. Teams should be empowered to make the decisions so that the constant speed can be kept up. In fact, in a lean organization, the central authority just cannot stay in the fast pace, thus it cannot run the activities. Rather, local signaling and pull technique – an agreement in lean world to "deliver increasingly refined versions of working software at regular intervals" [Russo, 2010] - should be utilized between the workers. This guarantees that everybody knows what needs to be done. When development team is the one calling the shots, project manager activities are a bit different. Managers are responsible for: identifying waste, taking care of the value stream map and the biggest bottlenecks in it, coordinating meetings, helping the team to get the resources it needs, coordinating multiple teams and providing a motivated environment for the developers to work in. This kind of self-organizing is common feature in other agile practices like Scrum, but in lean it is valued very high. Poppendiecks' [2007] have even said in their book that "if you implement only one principle – respect for people – you will position the people in your organization to discover and implement the remaining Lean principles".

Deferring commitment. Lean thinking emphasizes that decisions and commits should be made as late as possible. This is to ensure that decision makers have the most

knowledge before making the decision, thus the decisions are better and more reliable because they were made with enough knowledge, not just based on forecasting or approximating. Of course the latest possible time varies depending on many factors and circumstances: waiting too long can be as harmful as rushing the decisions. To allow this principle to be fulfilled the system itself should be built so that modifications and changes are easily doable. On the other hand also the decisions should be reversible. Iterative development process can help especially if the problem is evolving or there is a high-level of uncertainty. This kind of concurrent development means that developers can start working with still evolving requirements as soon as the high-level conceptual design is ready. Developers can try a set of options before committing to one. Every iteration provides the needed feedback to guide the product to the end.

There is also the eighth principle which is not directly listed in Poppendiecks' book but it covers in some way or other the aforementioned principles and thinking behind them: **continuous improvement** or keep getting better. For example in Toyota every work system is improved constantly, under the guidance of a teacher or tutor, at the lowest possible level of the organization [Poppendieck and Cusumano, 2012]. Everything is subject to improvement and adaptation. Lean thinking suggests that organizations starting with some agile methodology keep it as a starting point from which the methodology is improved and adapted by the people and teams doing the work.

3. Version control systems

Version control systems (VCS) play an important role in Software Configuration Management (SCM) [Malmsten, 2010]. SCM is a discipline that is considered to consist of four basic activities: configuration identification, configuration control, configuration status accounting and configuration audit [Kirk, 2002]. The main function for SCM is to give the project management a set of tools and practices on how to maintain software configurations and control the changes, thus preserving the integrity of the software [Kirk, 2002]. In software engineering this means tracking several files and changes made to them [Estublier et al., 2002].

Version control systems are the software tools used to control and track these files and changes. A need for such systems arose when developers started to work together in projects and a need for tracking was imminent [Malmsten, 2010]. Version control systems come in handy when multiple developers try to modify the same file: without a control mechanism, edits would just overwrite each other. Version control systems prevent this by offering ways to deal with these situations [Azad, 2007]. In cases where something foul or unwanted was changed or the change broke something, version control systems allowed to go back to an older version. It is also easier to back track what caused the problem with the version control systems due to information related to every change [Azad, 2007].

Currently there are two models on how the version control systems work: centralized and distributed. The centralized came first. It involves that the project has one repository in which all developers are working. If somebody changes the file, everybody can see it, the same goes for example with opening a new branch. All changes are visible to everybody. The first notable centralized version control system was Revision Control System (RCS) launched in 1982. It was a successor for Source Code Control System (SCCS) and as the predecessor; RCS could only keep track of single files. This is the reason why it was not suitable for projects that most often consisted of several files, though at that time it was free unlike SCCS [VCSHistory, 2010]. The follower - Concurrent Versions System (CVS) - was released four years later and it was the main version system control tool for many until year 2000, when Subversion (SVN) came forward. Other notable tools at the moment are IBM Rational ClearCase and Perforce. [Malmsten, 2010; Ruparelia, 2010]

In the distributed model every developer has a copy – or own branch – of the project that is visible only to the owner of the copy. When a change is made it is only reflected to the copy in which the change occurred. When developer wants to share this with everybody – like making it visible – he must publish the changes in order to get them to others. Others can then pull these changes to their own copy, thus keeping in sync with

what others have done. Tools that implement the distributed model are – but not limited to – Git, GNU arch [GNU arch, 2013] and Mercurial [Mercurial SCM, 2014].

For the purpose of this Thesis, I'll introduce in a little more detail the distributed version control model called Git and one of the platforms utilizing it, i.e. GitHub.

The development of Git began on April the 3rd 2005. It sprung out from the Linux kernel development world. Developers had used BitKeeper (first distributed version control system) to handle the Linux kernel project but due to copyright issues, which led to the revoke of BitKeeper's free-of-charge status in the project. This prompted the Linux developers – and especially Linus Torvalds – to come up with an own tool for the version control. They wanted the new system to be fast, reliable and support distributed model just as BitKeeper had. Their expectations were high and they negated the use of existing version control systems. At this stage, it was decided that a new tool was the only way to go. So began the journey of Git. Its existence is characterized by strong support for non-linear and distributed development, efficient handling of large projects and big quantities of data in a compact manner [About Git, 2014]. Diomidis Spinellis [2012] describes Git as "elevating the software's revisions to first-class citizens". In that article Spinellis highlights Git's ability to keep a complete record of the history of the file and bring the work process to higher abstraction level, changing for better the way of thinking. He even compares this transition from moving low-level programming languages like Assembly, to high-level languages. On 26th of July 2005, Torvalds turned over the maintenance to a major contributor Junio Hamano, who was later responsible for the release of the version 1.0. Hamano still remains the project maintainer. [A Short History of Git, 2014]

GitHub is one of the best known social coding platforms which utilize Git. It was founded by Tom Preston-Werner, Chris Wanstrath and PJ Hyett in 2008 and has since grown to the biggest code host in world [GitHub, 2014]. Other than Git version control system, GitHub offers a multitude of features from an issue tracker, to wikis, feeds and followers [GitHub New Features, 2014]. The Git itself has had some modifications allowing for example closing or commenting issues through commit messages. GitHub offers a free usage for open source projects and an array of hosted plans for those individuals who want to keep their repositories private.

3.1. GitHub terms and components

Throughout the guideline presented in the Chapter 6 a few central terms are used. These terms mainly come from the GitHub itself. In the following paragraphs I will go through these terms and definitions to establish an unambiguous meaning within the context of the guideline. After the definitions I will represent components of GitHub which are relevant from the guideline's point of view.

Issue. Within GitHub issue is a very vague concept. In its simplest, it is a task that should be done in order to achieve something with the software being developed.

However, issue can also be many other things, ranging from memos to requirements to sticky notes. Issue tracker is a component of a GitHub that is a lightweight tool for monitoring and tracking issues. For the guideline, the term issue is used as a higher level concept. It includes both requirements and tasks, which will be explained in the next paragraph.

Task and requirement. Tasks and requirements are both specialized issues. Task is a concrete item, whether it is implementation, designing or something else, that must be done in order to fulfill defined requirements. A very basic use case is that task represents a feature of the software, though in the guideline a task can be any concrete action that needs a work to be done. Requirements hold, in predefined formats, the goals and business objectives for the software. They are more abstract than tasks, and as such need tasks to accompany them. Both tasks and requirements can be divided to main level and sub level. Sub levels are to be used when a requirement or task is so large, that it is semantically wise to divide it to subcomponents. For example a task can have different sub tasks for designing UI, creating it, designing architecture and implementation. Sub level items specify their parent items in a more detail. Distinguishing requirements and tasks and their two levels, is not something that is recognized by GitHub. Therefore naming conventions, labels and references between issues must be used to achieve this.

Milestone. Milestones are basically iterations. In GitHub, milestones are used like a special filter which tells the due date for a set of issues. An issue can always belong to a maximum of one milestone, but the issue can also be without a milestone. A milestone can have a defined due date. Issues themselves don’t have such option available.

Label. Labels are small tags that can be assigned to issues. Every label in GitHub has a name and a color. The color is used as a background for label. Every single label also acts as a filter to the issues.

Filter. In the guideline filters refer to different filtering options available in the issue tracker. These filters can be divided to four categories. Personal, milestone, label and state filters. Personal filters are used to filter issues that have something to do with the current user. For example, issues that are created by the user can be filtered. Milestone filter is used to filter issues associated to a specific milestone. Only one personal filter can be applied at the time. The same situation is with the milestone filter. The label filter limits the issues to those that have all the selected labels assigned to them. Contrary to the personal and milestone filters, multiple labels can be selected for filtering purpose. The last filter – state filter - is for toggling the state of issues. Either open or closed issues can be displayed.

The central concept in GitHub is a repository. GitHub’s key aspect is version control, so it is natural that most of other components are built upon this feature. For the purpose of the guideline, we are only interested from this exact aspect and its subcomponents.

From the Figure 1 we can see the default view that becomes visible when a repository is opened in GitHub. In the center of the page lies the functionality related to the version control mechanism. The view shows a limited set of statistics like commit count. These statistics also act as links to more detailed information. For example behind commit count can be found a full list of commits, their change sets and commit messages.

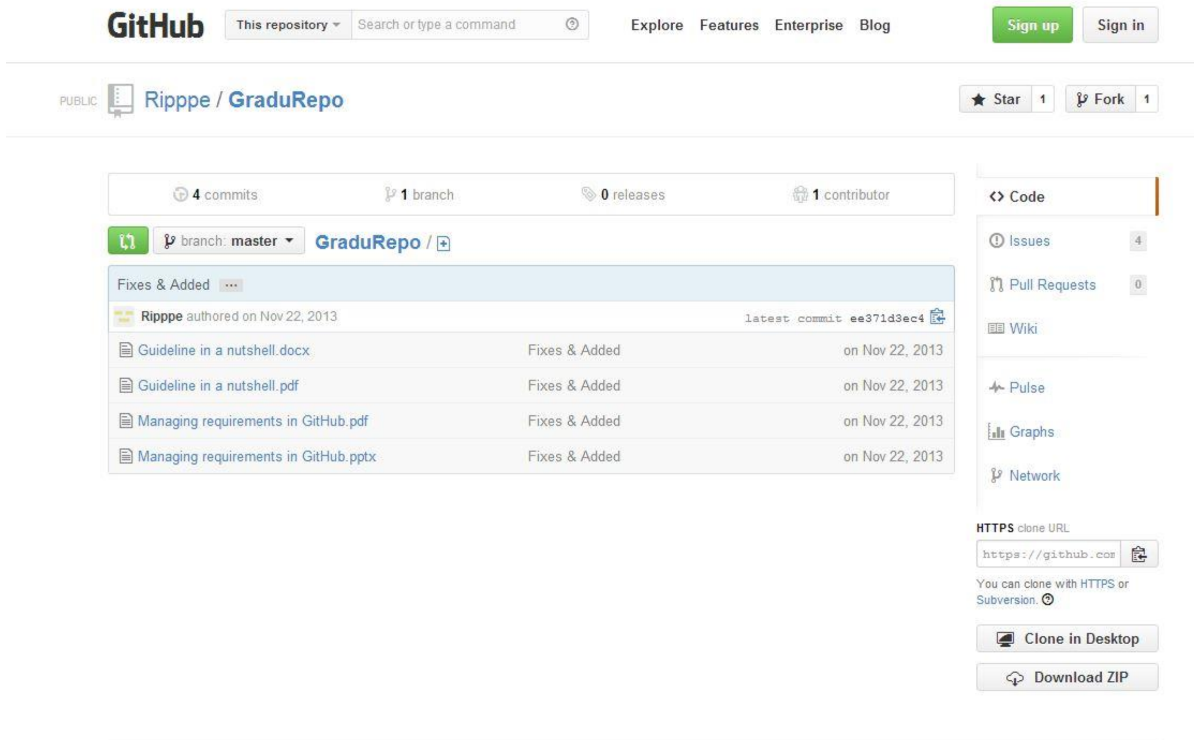

Figure 1. The repository view in GitHub.

The bulk of a repository's main page consists of the files and folders inside the version control system. Below them is a read me file, if such exists within the repository.

In the right side panel are navigation links to the sub components of the repository. "Code" – the selected one – is the repository view. Other navigation components are e.g. "Issues" and "Wiki".

In the top of the screen is the text search functionality. This can be used to search multitude of things inside GitHub's repositories.

The second Figure presents the "Issues" view, which is also called as an issue tracker. The guideline focuses mainly to this view and its functions. The issue tracker's main view is composed of five distinct areas labeled by the number in the Figure 2.

Figure 2. The issue tracker view in GitHub.

The first area is issue tracker's inner navigation. Provided options are the default view, milestone view and creating a new issue. The second area has exclusive general filters that affect the issue view (area 5). The third one lists possible actions with labels. From this area new labels can be created and the old ones edited. Name and label color are subject to editing action.

Area 4 provides the state filter. This area also contains a sort option for the issues. Area 5 lists every issue within this repository. Name, assigned labels, identification number, creator and elapsed time from the creation are shown. Every issue has a checkbox, and by selecting it, certain actions – for example assigning labels – can be carried out without the need to open individual issues one by one.

From the milestone view, user can observe and manage the milestones. User can toggle open and closed milestones and also milestones with or without any issues assigned to them. For each milestone several options are available. Milestone can be edited, closed or deleted. Issues from a single milestone can be browsed, but this functionality is actually just a short cut to the default issues view that automatically enables the milestone filter. Milestone's name, description and due date are also present. The division between how many of milestone's issues are opened or closed is also given.

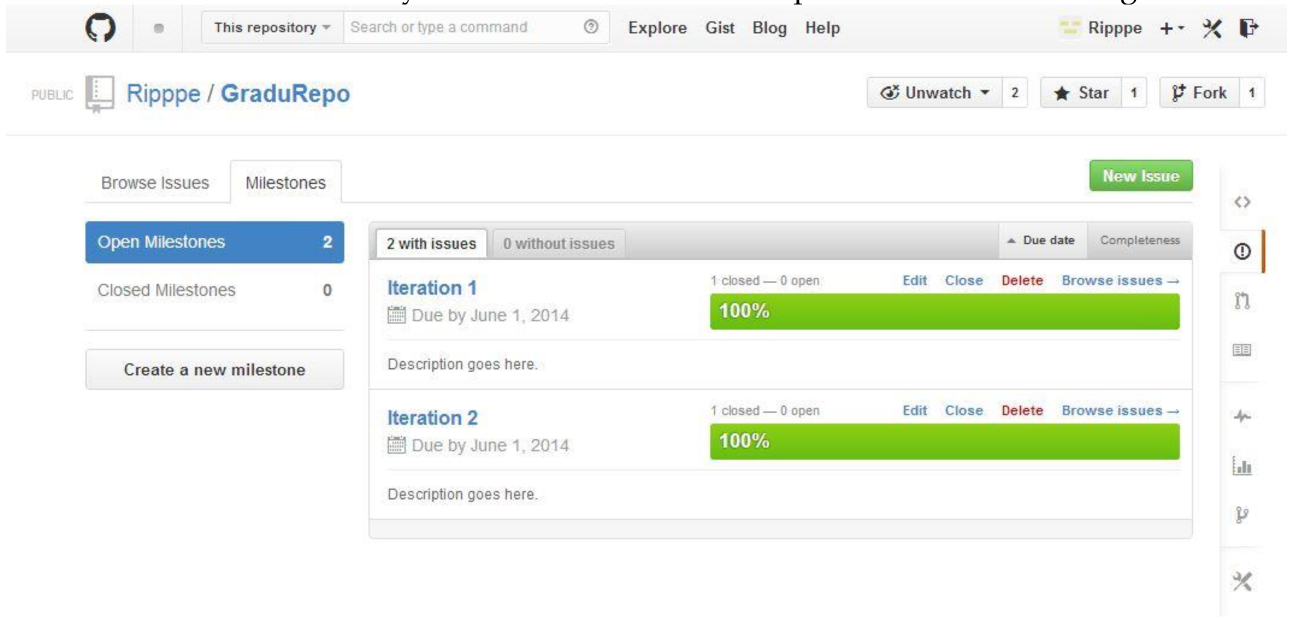

Figure 3. Milestone view in GitHub.

The issue creator view provides the choices for creating a new issue and is shown in Figure 4. User must give a name and a description, but he can also assign a developer for the issue, desired milestone and labels. The issue can be previewed before submitting it.

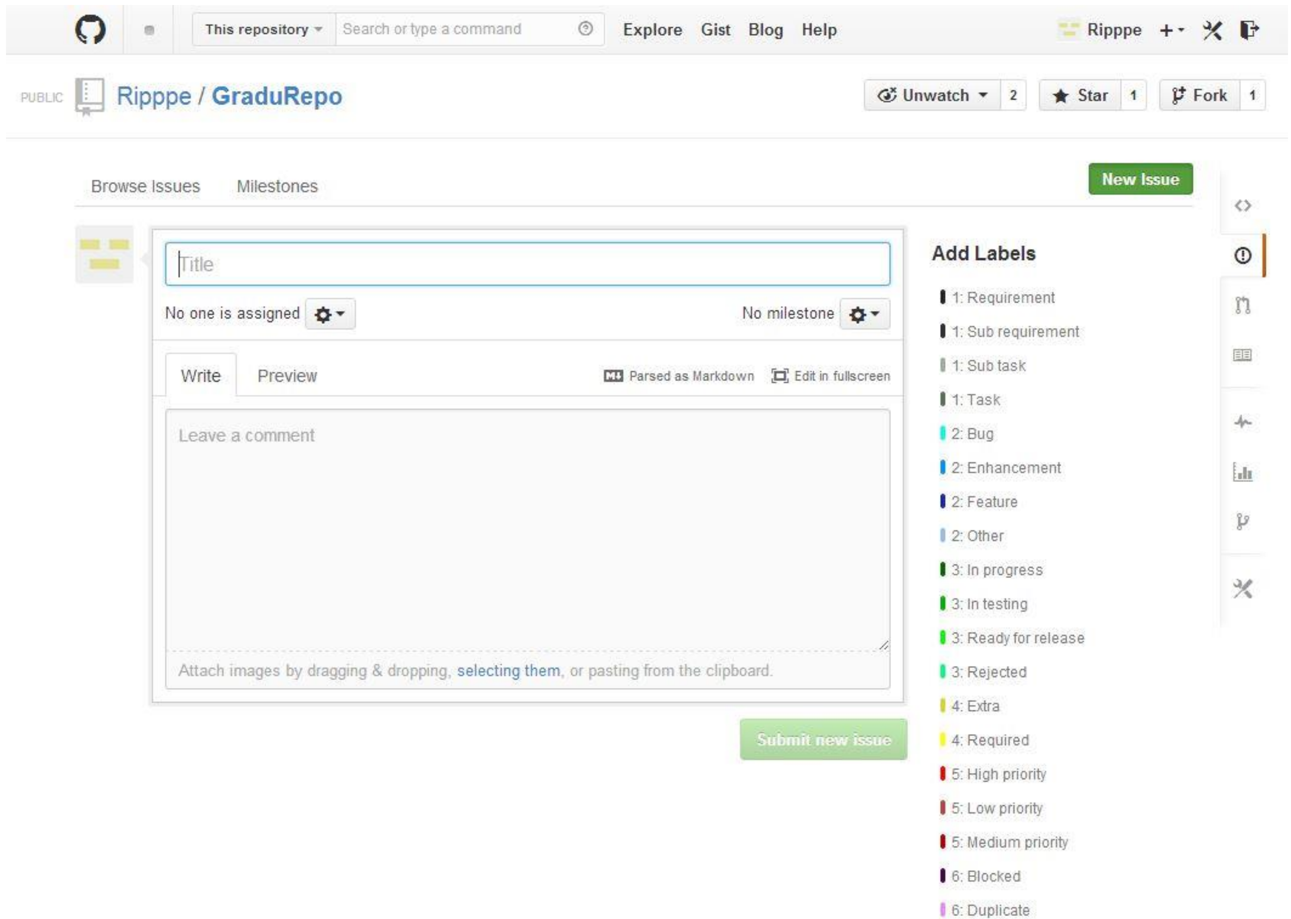

Figure 4. View of creating an issue in GitHub.

When an existing issue is opened, a detailed info view is displayed as shown in Figure 5. This view contains every piece of information that is related to the issue. All the data given in the creation view is visible, and also editable. The state (open or closed) is notified below the name of the issue. Other relevant information includes the ability to subscribe to the issue and listing all the participants of the issues. Comments and references to the issue are visible below the description in a chronological order. Subscribing to the issue means getting notifications about changes to the issue. The creator of the issue is automatically subscribed.

Figure 5. A view from an existing issue in GitHub.

The last sub component of the repository, which is referenced in the guideline, is the "Wiki". Each repository has an own wiki collection that acts like any ordinary wiki instance. It consists of one or more pages that are created using one markup from the list of possibilities.

This repository ▾ Search or type a command Explore Gist Blog Help Ripppe +▾

PUBLIC Ripppe / GraduRepo Unwatch ▾ 2 Star 1 Fork 1

Requirements for the project X

Edit New Page

Ripppe edited this page on Oct 9, 2013 · 10 commits

If requirements are divided under (for example) certain areas, put that one as a heading

- **(PRO-001)**: This is the first requirement
- **(PRO-002)**: This is the second one
- **(PRO-003)**: ...and the third

Here starts next area

- **(PRO-004)**: And it has a requirement!

Down below here (if you don't desire to make separate pages) you should open up the requirements more IF NEED BE.

PRO-002

Heres more info relating to the requirement (links, pictures, customer's wishes, etc.).

Alternatively you could keep requirement related stuff at the minimum in the wiki (meaning in practice only a list of initial requirements). If you want, you can only use issues for requirements and wiki pages to bring more information about that specific requirement (for example, GUI pictures).

▾ Pages (1)

Requirements for the project X

+ Add a custom sidebar

Clone this wiki locally

https://github.com/Ripppe/

Clone in Desktop

Figure 6. Wiki view in GitHub.

4. Requirements engineering

The requirements management is a part of the process called requirements engineering (RE). This process aims to study and identify the descriptions of services and its operational boundaries [Sommerville, 2007] that are needed to adequately solve customer's problem or to fulfill a contract. The essence of requirements engineering can be summarized to the statement by Paetsch and others [2003]: "The aim of RE is to help to know what to build **before** system development starts in order to prevent costly rework."

To understand this process and particularly the requirements management, we must examine what kind of requirements there are. As Sommerville [2007] points out, the term requirement is not used consistently across the field of software development. In some cases a requirement can represent very abstract description or definition of a system. In other cases it can be a strictly formal definition. Davis [1993] explains this contrast to be caused by contract negotiations. Requirements can't be too detailed if it is wanted that several contractors are able to bid for it. After the contract has been negotiated a more specific system definition must be written down so that different parties know what the system is exactly needed to do.

Requirements can further be divided to several categories. On the aspect of abstraction level, requirements can be distinguished between user requirements and system requirements. User requirements are written in plain language, possibly utilizing other tools – like diagrams or mockups – to illustrate them better. They are more abstract than system requirements which should be described precisely. User requirements should not take a stance on low-level design aspects they are for system requirements which line out functions and operational services the product is supposed to provide.

Requirements can also be divided to functional (FR) and non-functional requirements (NFR). FRs state the functional operation of the system, for example how it should respond to certain input. NFRs on the other hand are constraints on the services or functions and they usually govern the whole system. However, as the name suggests, they are not directly concerned with specific functions delivered by the system. FRs and NFRs can be divided to subcategories depending on the nature of the requirement. It is also possible to consider the third requirement level beside the FR and NFR: domain requirements. These requirements come from the domain itself – not from a need of the customer – and can affect for example how certain calculations are done, thus they often cross with FRs. [Sommerville, 2007]

The requirements engineering process consists of five parts: elicitation, analysis, documentation, validation and management [Paetsch et al., 2003]. Though, as pointed out by Sommerville [2007], requirements process can start with feasibility study. As mentioned in the beginning of this Chapter, the sole purpose of this process is to identify

the needs of stakeholders, analyze them and refine them as a specification [Jones, 1996]. The outcome – specification – states the requirements that the software must satisfy. I'm going to quickly cover the different parts of requirements engineering. Requirements management will be discussed in more detail.

Feasibility study. As discussed by Sommerville [2007], this should be the starting point for requirements engineering process for all new systems. The purpose of the study is to report whether or not it is worth – feasible – to continue the work with the requirements and the software development. The study must point out whether or not the system fulfills the business objectives set by the customer. This is achieved by information assessment, collection and report writing. Assessment phase identifies the information sources needed to answer following questions: 1) Does the system contribute to the business objectives? 2) Can the system be developed using available technology, budget and schedule? 3) Can the system be integrated with existing systems? Collection phase covers collecting the needed information from the identified sources to answer the above questions. The last step is to write the report to recommend continuing with requirements process or not. [Sommerville, 2007]

Elicitation. In this part of the process, software engineers must work in collaboration with customers and end-users to discover what services and features are expected from the system. It is up to software engineers to come up with the best ways of collecting the needed information. It may not be an easy task because people might have a hard time describing their needs, wants and knowledge. Practitioners have developed many methods for digging out the information, for example interviews, prototyping, brainstorming, observations, workshops and focus groups. There is no pattern on how to collect the information: it can be a combination of multiple methods. In the elicitation to it is crucial to understand the application domain, business needs, system constraints and the problem the system is intended to solve. In the end of elicitation a list of requirements should be made available. [Hofmann and Lehner, 2001; Paetsch et al., 2003; Sommerville, 2007]

Analysis. After elicitation has been carried out it is time to analyze the collected requirements. The requirements are checked against necessity (is the requirement needed), consistency (are there overlapping requirements), completeness (do requirements depict the system in its whole) and feasibility (are requirements within a budget and time constraint). If problems are detected, they must be solved to form the definitive set of requirements. Three main methods to solve the disputes are Joint Application Development (JAD) sessions [Paetsch et al., 2003], prioritization and modeling. JAD sessions are facilitated group sessions or workshops with a structured analysis approach. During the sessions different stakeholders discuss under the guidance of a session leader about the desired product features. JAD encourages to cooperative team work and mutual understanding between stakeholders. Requirements prioritization

focuses on prioritizing the most valuable features of the system. The customer is responsible for putting the requirements in a prioritized order, but developers should give feedback about the technical risks, costs and difficulties related to the requirements. This feedback might have an impact on the prioritization. The third tool is modeling. There are many approaches on how to do the task of modeling, but they all aim to create a model depicting the system. For example prototypes are operational models that stakeholders can directly experience with. [Hofmann and Lehner, 2001; Paetsch et al., 2003; Sommerville, 2007]

Documentation. The requirements documentation should be created and maintained to communicate requirements between stakeholders. It acts as a baseline from where to start and is a necessity for change control. This documentation is also called requirements specification (RS). IEEE Standard 830-1998 [IEEE Recommended Practice for Software Requirements Specifications, 1998] for software requirements states eight quality attributes for good RS. It shall be: correct, unambiguous, complete, consistent, prioritized, verifiable, modifiable and traceable. A correct RS consists of requirements that portray what the software is going to be. Unambiguousness guarantees that there is only one interpretation of every single requirement, i.e. every stakeholder understands them the same way. RS is considered to be complete when it lists every relevant requirement of the software, it defines responses for all inputs whether valid or invalid and all figures and other visualizations are included. When RS is consistent the requirements don't conflict with each other. Prioritization can be done either by importance or stability of requirements. Every requirement shall have an identifier to distinct the priority level. Requirements are verifiable if it possible to check whether the software meets the requirement or not. Modifiable RS accepts changes without breaking the format of the RS documentation and traceable requirements know their origin.

Validation. The purpose of the validation phase is to validate that the requirements really depict the system to be implemented. Information used for the validation is the requirements specification and organizational knowledge and standards. The output is a list of problems found and ways to cope with them. This phase can somewhat overlap with the analysis because with enough attention and care, these problems could be identified already in that phase. Validating the specification is an important factor for the success of the system. Errors found during the development can lead to extensive rework costs. [Hofmann and Lehner, 2001; Paetsch et al., 2003; Sommerville, 2007]

Requirements management can be seen as the last step of the requirements engineering process. This phase must be taken into a consideration since in the current software development world requirements are going to change [Sommerville, 2007] and it is something that customers nowadays expect from a modern software [Lam and Shankararaman, 1999]. The understanding of the problem can change during the development leading to a change of requirements. After the launch end-users get to

experience with the system and it is almost inevitable that new needs and priorities are discovered [Sommerville, 2007]. This means that requirements can be introduced, modified or removed almost at any point of the requirements management process [Favaro, 2002].

Multiple factors influence requirements evolution. In some cases requirements can be a compromise between two or more stakeholder parties. After the release it could be discovered that the balance of this compromise needs shifting to some direction. The customers may not be the ones using the system (e.g. in a case of ecommerce website), but they are the ones paying for it. Requirements that come from the customer – restricted by time and budget – can be contradictory with what the end-users want. Also the business or technical environment evolves. New hardware is bought, the system needs to be integrated with others, business objectives can change and new legislation or regulations could take place. All aforementioned factors cause turbulence to requirements. [Sommerville, 2007]

Requirements management itself is a process to understand, maintain and control requirements. In a traditional way this is quite a formal process, unlike in an agile way. From the management point of view, requirements can be either enduring or volatile. Enduring requirements are so called core requirements which stay relatively stable. They are usually derived from the domain of the system. Volatile requirements on the other hand are such requirements that are expected or likely to change during the course of the system development or after its release. [Sommerville, 2007]

Requirements management can be divided to four main activities: change control, version control, requirements tracing and requirements status tracking [Li et al., 2012; Paetsch et al., 2003; Wiegers, 2009]. The way these activities are carried out should be decided before the project starts [Sommerville, 2007].

Change control covers the situations when a need arises to make a change to requirements. When a problem with existing requirements is observed, it is usually brought up for discussion which may lead to a change request. When a change request is received, the effects it has to the system must be analyzed. One common tool is impact analysis, the purpose of which is to make the implications of the change more visible and concrete. The requirement can then either be rejected or implemented. Requirement documentation must be updated appropriately. This activity also includes measuring the requirements stability. [Li et al., 2012; Sommerville, 2007; Wiegers, 2009]

Version control defines how requirement documentation and requirements themselves are identified, versioned and kept together. Requirements tracing focuses on defining links to other requirements and system components, so that it is possible to see what kind of relations requirements have between each other. Requirements status tracking means that requirements are being observed about how they progress and if problems have occurred. Usually this implies that requirements have a defined status associated to make

the tracking easier. The status reveals at which stage of the progress the implementation of the requirement is going. For example an ongoing requirement could have status "In progress" and an implemented requirement "Resolved" or "Closed". Only one status at a time should be allowed for a requirement. [Wiegers, 2009]

4.1. Agile requirements engineering

Paetsch and others [2003] have noted that requirements engineering is often considered as a traditional software development process and as such, is not compatible with agile practices. One such conflict between these two is that requirements engineering is relying heavily on documentation whereas agile methods try to get rid of extra documentation and prefer face-to-face interaction with different stakeholders.

The questions of how requirements engineering is handled in an agile environment don't have unambiguous answers. As different agile methods embrace different aspects, so do researchers and practitioners who have studied the agile requirements engineering process. However some common, higher level, principles can be declared as we can see in the following list.

Stakeholder – especially customer – collaboration and involvement. This is one of the core principles for the agile requirements engineering. Multiple studies [Cao and Ramesh, 2008; Ernst and Murphy, 2012; Hofmann and Lehner, 2001; Paetsch et al., 2003] have conducted that customer involvement with requirements through the whole project – not just in the beginning - is a key factor for a success. Though the traditional requirements engineering also embraces customer contribution, in an agile environment this is valued even more.

In order to achieve the needed communication between customer and other stakeholders, several ways can be used [Paetsch et al., 2003]. Interviews and face-to-face conversations provide a direct way of getting information and needed knowledge, but they should always be conducted without any middleman to minimize the chance for misunderstandings [Paetsch et al., 2003]. Similar to interviews, brainstorming and observation are good tools for this case. Especially observation can reveal such information that couldn't be collected through other means. Often this is because customers may leave something unsaid because they think it as self-evident [Paetsch et al., 2003]. Ad hoc modeling can temporarily be utilized to better visualize limited parts of the system [Paetsch et al., 2003]. The same purpose is with prototyping [Cao and Ramesh, 2008]. Acceptance tests and review meetings offer a way to validate the course of requirements, and they should be carried out continuously [Cao and Ramesh, 2008].

The aim of these practices is to deepen the knowledge of the developers about the application domain and enhancing the possibility for a successful requirements engineering process [Hofmann and Lehner, 2001]. An intense collaboration has been

reported to lower the need for major changes in delivered products [Cao and Ramesh, 2008].

As the agile requirements engineering depends on a stakeholder collaboration more than documentation, it could pose a risk to success, should the collaboration fail [Cao and Ramesh, 2008]. Failure in collaboration can be caused by the unavailability of the customer, conflicts inside customer's personnel and distrust to developers or to the agile processes.

Replacing heavyweight documentation with a lightweight alternative. Documentation should not be neglected because it is used for knowledge transfer, but it should be toned down to the minimum feasible level due to its cost ineffectiveness [Paetsch et al., 2003]. Alternatives for extensive documentation have been suggested and some are widely used. Story cards or user stories are good examples of these [Paetsch et al., 2003; Zhang et al., 2010]. They are usually either physical or electric representations of cards that include a few sentences describing user needs – business value - in a plain text [Cockburn, 2006; Waldmann, 2011]. They should correspond to a limited set of features or functionalities that can be implemented. Backlog or feature list can then be used to keep the track of stories and their progress [Waldmann, 2011].

Iterative requirements engineering process. As many agile methods take advantage of iterative software development process, the same approach can be applied to requirements engineering. In fact, as discussed earlier [Favaro, 2002; Miller, 2001; Sommerville, 2007], one of the reasons driving iterative software development in agile practices is that it gives a way to cope with rapidly changing environment, which also includes requirements.

A key aspect of iterative requirements engineering mentioned by multiple studies [Ernst and Murphy, 2012; Hofmann and Lehner, 2001; Paetsch et al., 2003] is prioritization. Requirements are ordered by the highest priority, and as such they are also implemented. Unlike in the traditional requirements engineering process, the priority status of a requirement can change during its lifecycle. It is not uncommon that requirements are specified in more detail just before they are about to be implemented. This kind of activity can occur each time before the next implementation iteration [Paetsch et al., 2003]. A case study by Hoffmann and Lehner [2001] supports this by discussing, that successful requirements engineering is driven by stakeholders' requirements prioritization. Ways for achieving this are constant prototyping and validation & verification of requirements. Customers' satisfaction can be ensured this way, especially if they can't explicitly tell what they want from the system [Cao and Ramesh, 2008]. Ernst and Murphy [2012] call this kind of practice as Just-in-time requirements in their article.

To summarize aforementioned three factors, the key aspect for agile requirements engineering is to produce the best possible business value for the customer through the whole requirements lifecycle [Favaro, 2002; Paetsch et al., 2003].

As with agile methods in general, it should be remembered that agile requirements engineering doesn’t guarantee success, although correctly used can greatly enhance chances for it. Cao and Ramesh [2008] have stated a wide array of possible problems with these practices. Communication is only as effective as its participants. It’s hard to create cost and time estimates with iterative requirements. Documentation is easy to be neglected, same as non-functional requirements because they don’t necessarily implement visible or otherwise concrete business value.

5. Project management

As Sommerville [2007] points out good management doesn't automatically mean a successful project, but bad management often leads to failure. As a supervisor for the software project, project manager's main responsibilities are to conduct the planning and scheduling of development, ensure it is carried out properly and monitoring progress. Because in most cases time and budget constraints act as boundaries to software development, it is imperative that somebody guides that process from start to finish.

Software project management has many similarities compared to other engineering project management aspects. However, there are some fundamental differences. Software as a product is more abstract [Coram and Bohner, 2005] than for example a house. If a house is missing a roof, it is clearly visible right away. Due to the nature of the software, this kind of remarks cannot be done by a software project manager rather he must rely for the documentation produced by others to observer the progress. Other distinction is a lack of standardized software processes. The process to build a certain kind of house is well defined and understood, but – though our knowledge has increased – it is hard to reliably estimate if the chosen software process is going to function without problems. Furthermore, software projects tend to change from project to project causing difficulties to oversee possible difficulties. The same goes to rapidly changing technological aspects of software projects. [Sommerville, 2007]

It is impossible to state a complete list of activities project managers are subject to carry out during the project, because these activities may change depending on the context of the project. Sommerville [2007] has identified five common activities that are most likely to occur in a software project.

Proposal writing. In some cases it is possible that project manager is responsible of writing a proposal to win a contract. This is considered a critical task as many software companies cannot live without won contracts. The proposal is basically a strategy guideline of how to achieve the objectives of the project. It may also include time and budget estimates.

Project planning. The effective management requires a thorough planning of a project. Project plan should be the best possible plan with the current information; it is used as guidance for the project. As the information increases the plan is revised to accommodate this, thus the project plan lives an iterative life and is only completed when the project finishes. The plan is constructed of constraints set to the project and project's parameters such as size and structure, milestones and deliverables. Well defined milestones are necessary because they can be used by a project manager to assess the progress of the project. Deliverables are the results of the project delivered to a customer. The project plan

may not be the only plan written for the project. Other plans are for example quality and validation plans, but they can also be incorporated into a project plan.

Project scheduling. The difficult task of project managers is to constantly estimate the time and other resources with the project's scope. Resources must be utilized and organized so that the activities from the project plan are achieved. What makes this task so devious is that projects may use different design methods, have a completely different domain area or be written with a changing programming language. Scheduling involves separating required work between the activities and assessing time required to carry them out. Estimates should always be pessimistic rather than optimistic, since problems are almost certain to come up. With a pessimistic plan, the project manager has some contingency to tackle these unforeseen obstacles. Resources are then coordinated to the activities. Multiple diagrams can be used to better visualize this diversion, and how well each activity is progressing. Such tools include bar charts and activity networks.

Risk management. A risk management is a key aspect of a project manager's work. It is a process of identifying, analyzing, preparing and monitoring risks that may occur during the project. Risks can be divided to three general groups: project, product and business risks. Project risks affect project's schedule or resources, product risks quality or performance of the product and business risks the organization developing the software. As software projects always contain uncertainty the risk management is an important step. The process for the risk management has four steps. First the risks must be identified. Then they are analyzed by the likelihood and consequences they might have. After analysis a plan is made to address how the risks can be minimized or avoided. Finally the risks are constantly monitored and the plan revised by the results and observations.

Personnel selection and evaluation. A project manager is also responsible of choosing the staff for the project. However, this is not a straightforward task. A project budget may not have room for more highly-paid staff or people with the right set of skills and expertise may not be available (neither internally nor externally). The organization may also wish to broaden the skills of its employees, assigning inexperienced staff to the project to gain the experience. With the constraints in mind, manager must conduct the team building.

Monitoring, reviews and reports. Almost every aspect stated so far requires some sort of monitoring from the project manager. In these cases monitoring could be described to be formal. However, a skilled project manager also practices informal monitoring, for example by exchanging information on a daily basis between the team members. An informal monitoring can reveal problems that are yet to surface, offering a shortcut for interruption before problems are concretized. Besides the monitoring, project manager carries out management reviews concerning the overall status of the project. The purpose

of these reviews – as with reports – is to keep stakeholders informed of how work is progressing and does it still meet the business goals assigned to it.

Agile practices have their own effects when it comes down to the project management. When the development style changes to more iterative and fast paced embracing the people behind the work, it definitely has consequences to the project management.

The broad spectrum of agile methods doesn't take a single uniformed stand on management activities. In fact, some methods don't instruct project management at all – like XP – while some others mainly concentrate to it like in the case of Scrum [Abrahamsson et al., 2003]. This doesn't negate the fact that the principles behind agilism and Agile Manifesto have an impact to management. The impact can be divided to three categories: people, process and project [Coram and Bohner, 2005]. As the following paragraphs will show many questions and challenges must be faced if the project management is wanted to be properly conducted in an agile manner.

Possibly the greatest impact within the people involved to the project concerns the developers. As the project manager is responsible of selecting the project team, he must be aware that agile methods are more suitable for experienced developers. The fast pacing of agile development requires quick problem solving, team working, talent and skills. Such developers are a rare commodity and projects often have to cope with a less ideal team, making it hard to fully utilize the best of agile practices. Testers need to have a closer collaboration with developers to accommodate the need for constantly ensuring the quality of the product. [Coram and Bohner, 2005]

From the management perspective, there exist two leaders: a team leader and a project manager. The team leader is responsible for leading people. This should occur by collaboration and mentor like approach. Agile practices embrace the team and suggest that the decision making power should be given to them. This might be a problematic if leaders come from a traditional command and control management world. The project manager handles the progress monitoring and business decisions. Their emphasis is in change control activity rather than scheduling and planning. [Coram and Bohner, 2005]

From a customer a more collaborative attitude is expected. The customer is involved more frequently and they have a greater influence. On the other hand this means that the customer's representative should be empowered, have a comprehensive domain knowledge, information about end-users' needs and be committed. Finding a customer willing for such highly involved process may prove to be difficult, and can therefore add to risks, which project managers should be monitoring and tackling. To the executive management agile practices may seem very unconventional since they focus on fast delivery and high quality, while leaving cost and schedule estimates to a greatly lesser role. Providing the best business value means volatile requirements so it leaves the end boundary of the project open. [Coram and Bohner, 2005]

The project management process changes with the agile practices. Planning is more informal and constant activity that must be carried out. Documentation is more lightweight, thus the manager must have other tools to share knowledge and information inside the project team and the organization. Code reviews are one such tool. Because the documentation shouldn't be fully neglected, the manager has to balance between what is enough and what is waste. Agile development processes rely on implementing nothing extra, integrating continuously and refactoring existing code. Such practices may be uncommon for developers and the manager must be careful when introducing them to not to cause change resistance. [Coram and Bohner, 2005]

The project manager decides the methodology used in the project. Choosing an appropriate method is crucial because if poorly selected, could hinder the development. Some agile methods and practices are more suitable to certain kind of projects, but they are not universally applicable to every project. For example safety- and life-critical systems are not ideal for agile approach. Also the business factors must be considered. If requirements are set in stone for contract, agile methods lose their strength to overcome change. Documentation may have strict requirements – for example in government's software projects – which don't suit to the agile world. If rigorous scheduling and cost estimation is needed, agile isn't the answer.

6. Managing requirements in GitHub – The guideline

Requirements management is an important topic for every project, yet it is also easy to go wrong with it. This is especially the case if you are not familiar with the tools you are using or there are no well-defined rules to follow. This is in some degree exactly what happened to me in two different software projects that I attended to as a part of two different university courses.

In the first project I was in the role of a developer whereas in the second I was part of the project management team. In the first project we had Redmine [Redmine, 2014] as our requirements management tool. In the second one our approach was a hybrid between GitHub and different online documents – mainly Google Drive [Google Drive, 2014]. I couldn’t help the feeling that in both cases, the requirements management (RM from now on) wasn’t as successful as it could have been. As I wasn’t involved to the RM process of the first project I can’t explicitly tell how it was handled. Nevertheless as a developer I felt that the control over requirements and tasks related to them wasn’t in the best shape.

With the second project I ran into same kind of problems, this time as the one responsible for the RM. In this project I was the contact person between the project team and the customer and therefore I was the one controlling the requirements. The project consisted of two project managers, so although I was mainly responsible of the RM, the second project manager was also involved. Our requirements specification was quite successful, considering the fact that this was a nine months long student project. At first, requirements were just in online document format. As they were done, I over lined them.

For a time this was working but soon we realized with the other project manager that we needed more detailed view of how requirements were progressing, what needed to be done to them and who was doing what. This happened at the middle of the project so we quickly converted the requirements to GitHub’s own system. This helped a lot, but left still much to be desired. The conversion was done in a rush, so we didn’t have extra time to think about how to utilize GitHub with requirements. We were already using GitHub as a version control system, but GitHub’s features used for our RM were new to us. The confusion about how we should handle the RM was lingering in the air. In the end, although the conversion helped us, the process was handicapped at the best.

During these projects GitHub had showed that it was well worth its promise as a social coding platform. However, the question was what could be done to improve the user experience. With the first project I realized that the more separate tools are used, the more confusing it is going to be. Tools take time to learn to use. If every tool is different, you probably can’t remember by heart how to do some specific task in a certain tool. Furthermore the processes may vary greatly from tool to tool, causing frustration and even negligence because people can’t remember them. If everything could be achieved in

a single tool, it could probably lessen the burden, and free more time for relevant activities.

The RM is one such process. As I discovered, there isn't a formal or semi-formal way of doing it in GitHub. Of course GitHub's own tools are quite lightweight so in simple cases they don't really need much of a documented process. It may not even be necessary to talk about RM but rather a plain task list. If the approach is closer to a traditional project – like in my examples – a defined guide would be helpful. This sprung an idea about creating a semi-formal guideline for RM in GitHub. Semi-formal in this occasion means that the guideline offers a set of practices and suggestions for RM in GitHub but it doesn't force users to use them precisely as described. However, they cover such conceptual topics that should be taken into a consideration when it comes down to RM process.

6.1. Design Science Research approach

Design Science Research (DSR) was chosen as the approach to qualitatively study the problem topic. DSR method focuses on designing IT artifacts to solve a specified organizational problem [Hevner et al., 2004]. The created artifacts are also evaluated. These two processes are also called *building* and *evaluating*. DSR combines behavioral science with design science: behavioral science seeks to develop and justify theories whereas design science is a problem-solving paradigm.

There exist several blog posts by different software practitioners who share their own thoughts about how they have utilized the GitHub for handling projects [Bitner, 2012; Hemel, 2013; Hilburn, 2013; Using GitHub Issues to Manage Projects, 2012]. These posts do give an insight to the matter, but their usability remains shallow. The posts don't go very far to generalize their ideas but rather state how in particular cases GitHub's different components have been used. Other valid point is that the posts don't discuss the RM as a whole, but rather approach the subject as if the desire is for a pure task list or a backlog. No comparison or evaluation is conducted whatsoever. The desire I had was a more systematic and better defined guide that could be generalized to be applicable for wider array of projects. As DSR suggests, this kind of artifact – a guideline in my case – was built and evaluated.

The initial draft for the guideline was made in the autumn 2013. It built around three items working together. The guideline itself was made in a form of a slideshow presentation. This was partly due to the occurring circumstances. The guideline was needed in a rapid succession for the use of the project team which served as the case study for this Thesis. The guideline also needed to be self-explanatory so that any team member could download it and get the core idea without a face-to-face introduction. A slideshow presentation fulfilled this purpose. To better demonstrate the examples and ideas behind the guideline, I opened a new example repository [Salo, 2014] to GitHub. The repository

holds examples and other complementary information. The third item was a brief referral – about two pages – to the guideline. As the main part of the guideline – the slideshow – was quite hefty, the referral was supposed to be a collection about the core principles, written as shortly as possible. The referral had references to the slides, so that reader could easily go and see a more comprehensive explanation if needed.

The three items were handed to the case study group who then put the guideline in the use. I offered my assistance when it was needed and made iterative changes during the case study to shape the guideline based on the feedback received directly from the case study group in their meetings and online discussions. No major changes were necessary during this time.

After the case study project finished a survey was conducted to gather the overall experiences of the team. The results of the survey and comparison of the guideline against lean principles and RM objectives formed the evaluation.

6.2. How to use the guideline

The purpose of the guideline was to offer a set of recommendations and practices for RM in GitHub's environment. Individual parts can be used to some extend as such, but doing this may not provide the best result. However, there are always a number of factors that differ from project to project. To make the guideline as versatile as possible it leaves room for customization.

What GitHub has to offer is mainly the issue tracker and issues themselves, though the version control and wiki components can be used for supporting activities. Therefore the guideline mainly focuses on aspects concerning issues: how they should be used, created and monitored to achieve a consistent RM process.

The recommendations and practices of the guideline aim for complementing the four areas of RM: change control, version control, requirements tracing and requirements status tracking [Li et al., 2012; Paetsch et al., 2003; Wiegers, 2009]. As GitHub's issue tracker is quite lightweight according to its functionalities, it is very useful for projects using an agile approach. As argued before there doesn't exist a guideline that is as thorough as this one. The blog posts mentioned are not systematical and they are not scientific, but rather experience reports. Due to the fact that there is no scientific reference point, I will use lean software principles for assessing the guideline and its compatibility to the agile environment. Lean principles are abstract enough so that they don't narrow down cases where the guideline would be useful, but still offer enough guiding and concreteness for assessment. Furthermore, the principles are like directions and as such serve boundaries for assessment. Of course – due to the abstraction level – the principles are subjective to the viewer and context.

I will start the presentation of the guideline by quickly going through how the guideline is used. As stated, the guideline is mainly a collection of practices and

recommendations that work together towards a consistent RM process, but can be used independently. Therefore there is no strict step-by-step process that must be followed.

The guideline expects that its users are familiar with the features and functions of GitHub. Knowing how GitHub works is imperative to get a good start with the guideline itself. Luckily GitHub is intuitive to approach and use, so familiarizing shouldn't take too long. Users don't need to master every feature, but a basic knowledge is required, especially about repository, issue tracker and wiki. GitHub utilizes a special hypertext formatting called GitHub Flavored Markdown (GFM). GFM inherits from a markdown [Gruber, 2004] and is enhanced with additional convenient features, such as task lists and automated references. This syntax is used in issues, but it is also applicable to wiki, making it a good idea to get to know the syntax.

Before starting to work with issues themselves, the team must conduct some preparations. The label categories and labels themselves must be decided, so is the color coding with them. Overall naming conventions with issues must be agreed on. Generally it is ideal to go through the guideline's practices and decide how they are used in the project. As said, there is plenty of room for customizing the guideline to suit the project's needs. Beside the labels and naming conventions any other decision is not mandatory, but users are to be aware that if team doesn't use the guideline consistently, things can run out of control.

The team should also appoint somebody responsible for RM. Most often this is a project manager or other coordinator. This doesn't mean that the team itself is "free" from RM. On the contrary, the RM is something that belongs to every team members' responsibilities. The purpose of the appointment is to have a person who can attend to other team members' questions regarding the issue tracker, guideline and issues.

The guideline doesn't take a stand when it comes to requirements engineering steps occurring before the RM. The steps can be carried out by whatever means are necessary and suitable for the project's scope and type. As requirements are discovered and identified they are gradually created to the issue tracker. The optimal case is that requirements are immediately put into the issue tracker, but if there is a need to document them some other way first, the issue tracker can wait. It should be noted however that postponing creation of requirements to the issue tracker delays the moment when the team can begin the implementation. As soon as requirements are created the team can start splitting them to tasks, and when the first tasks are ready the implementation can begin. If requirements have a specific priority order, they must be split first to focus the work to the direction that brings the best business value. Creating an issue, whether it is a requirements or task, is covered in the Section 6.5.

Creating requirements and tasks is usually an ongoing action and can happen throughout the whole project. When first tasks are under work, they must be monitored, maintained and updated if needed. Issue's lifecycle is covered in the Section 6.8.

6.3. Separation of tasks and requirements

This guideline separates requirements and tasks. Why such distinction is needed? Requirements tend to vary in a size and detail level. It is not uncommon that a single requirement can cover a wide topic and so the amount of time, work and other resources needed for completing that requirement is different in every case. In order to trace requirements and track their progress, some kind of division is needed between what is wanted and what needs to be done to accomplish that. Both tasks – the concrete actions needed – and requirements – the business objectives – are kept in the issue tracker. This enables linking between them and keeping in one place. Everybody is kept up to date about what are the goals of the project and what needs to be done.

6.4. Hierarchy between requirements and tasks

Tasks and requirements can be even further split to smaller parts. This is a good way to eliminate unnecessary waste that goes for searching information: smaller issues contain less data to read through and are easier to identify. It also counts towards the status tracking. Splitting issues means that they must be thought thoroughly finding out the work really needed. The better the knowledge about an issue, the more likely it is that all of its aspects get covered.

This is why it is recommended to use sub tasks and sub requirements. This brings the number of different types of issues to four: requirements, tasks – a shorthand convenience names for main requirements and main tasks –, sub requirements and sub tasks. A good example of this occurred in the case study project. One of the project's thirteen requirements was “R2.2 There are a total of 10-15 puzzles”. This was in fact a sub requirement of “R2 Game has a Map which has sub sections”. A natural way of splitting R2.2 to tasks was to make one task for each different puzzle. The tasks could be further split to sub tasks representing implementing business logic, the UI and so on.

Of course a task for implementing business logic can still be quite large as a single task. However, there is a limit to how deep the hierarchy should go. For the guideline the maximum depth is four issues: requirement, sub requirement, task and sub task. To revise the example, an alternative to creating just one sub task for business logic would be to create multiple sub tasks to depict individual components of business logic. These sub tasks would be direct descendants to the main task – the original sub task simply wouldn't be created. For the sub tasks to be usable, they must not be in too detailed level. To give an example, a sub task that states “Implement function X” is too low on the abstraction level and probably causes more hindrance than benefit. The appropriate abstraction level for tasks is not set in stone, but rather it is something that must be decided separately in every project. The abstraction level is also somewhat covered in the Section 6.5 because it is highly related to information quantity associated with every issue, whether it is a requirement or task.

It is not always valid that the sub tasks are going to be helpful even if their abstraction level is appropriately high. Too small tasks cause more work with their creation and maintainability and offer minimal benefits. If developers are experienced, forcing them to create sub tasks might be waste of time: a main task can be sufficient coupled with a task list. On the other hand, in some cases it can help the developer further recognize the problem and its different areas, if he splits the current main task to sub tasks. So whether sub tasks should be created at all depends greatly on the situation on hand.

A good thing with sub tasks is that the components they affect are more easily identifiable and traceable. To give an example, it is possible that an UI component is implemented before the business logic to get quicker feedback from the customer. The business logic is to be implemented in the next iteration thus it is in a different milestone than the UI. With sub tasks, discovering this is easy, because the UI and business logic are separated to different sub tasks. If all information was in a main task, one would need to open the task in order to find it out. It is also possible that if everything is put in a main task, its description becomes clogged up due to the size of available data. If the project team has a lively communication, the comment area of a big main task can be quickly overrun, making it hard to find the relevant comments.

On the other hand, there are benefits for dismissing sub tasks. When an issue is referenced, the reference shown in a comment area has the information of how many of issue’s task list items are completed. With the sub tasks such information cannot be displayed due to the lack of automated hierarchy handling. This is a minor gain, since the information only shows how many of - not explicitly what – items are completed. The second benefit is negating the overhead caused by creating sub tasks. For example when a work begins with a sub task, the main task’s labels must be appropriately updated. It should also be noted that to get the best out of the issues, the whole team must be committed to use them. In some cases developers may find sub tasks to be just a nuisance and therefore the attitude towards the guideline might decrease causing negligent usage.

Overall it is impossible to say with a guarantee when sub tasks’ benefits overcome the slight waste caused by their upkeep. The decision about their use should be made case by case. The benefits gained from sub tasks shouldn’t be ignored just because they cause some more work. When information is separated to logical components, it is easier to find and interpret. The main point is that all the necessary information is present, logically structured and findable with minor effort.

Whether sub tasks are used or not, the following list contains the rules applied to the relations of issues:

- A main requirement must always exist.
- A sub requirement must always descend from a main requirement, same applies to tasks.
- A main task can be descendant of either a main or sub requirement.

- If both a main requirement and a sub requirement exist a main task must be descended of the sub requirement.
- A main requirement can have multiple sub requirement descendants. Multiple main tasks descendants are allowed to a single main or sub requirement and multiple sub tasks can be descendants of a single main task.
- All issues must be unambiguous (i.e. a single main task can be a descendant to only one main or sub requirement).

The Figure 7 demonstrates the above rules.

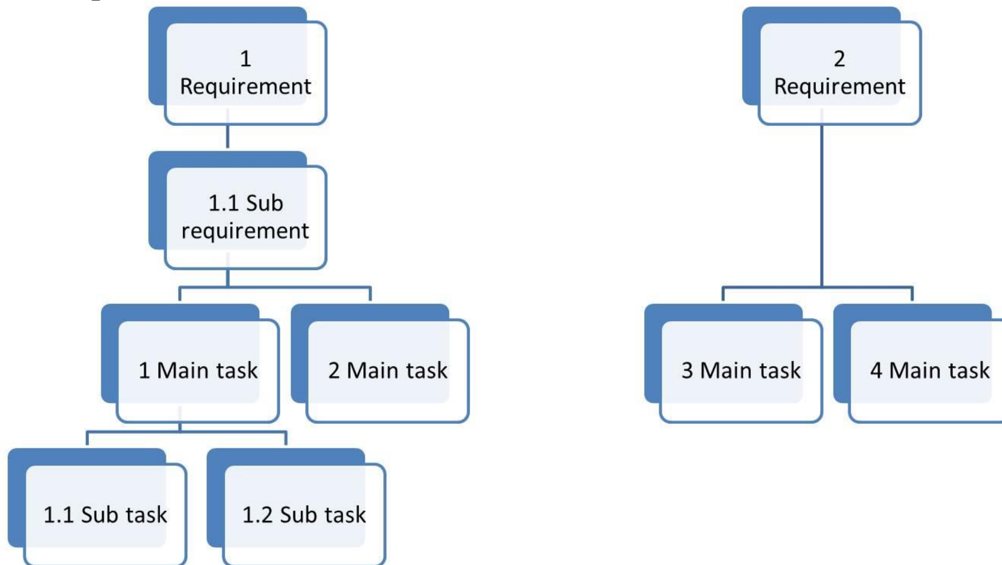

Figure 7. Different hierarchy combinations for issues.

To achieve this hierarchical structure in GitHub, issue references, labels and naming conventions must be used. These combined to the available filters are the most convenient ways for visibility purposes. They do require some manual labor and upkeep due to the fact that the issue tracker is very plain and lightweight, and does offer only a limited amount of automated functions. For example reference links between issues – i.e. referencing an issue form another issue – must be created manually.

6.5. Creating issues

Creating issues should be an activity for the whole team. Requirements and sub requirements may be composed with only a specified group of team members and customer representatives, but when it comes down to creating tasks, the team should be involved in its full. This has many benefits. Seasoned developers may already have suitable solutions in their mind or have such experience that is useful in some other way. Letting developers to contribute enhances the team spirit and takes them inside the decision making process. In the best scenario, the overview of the domain and its business goals is broadened within the team, thus building a stronger foundation for making better decisions.

As the guideline is directed towards agile environments, creating issues can be done iteratively. There are multiple ways of doing that. The team can start with only a handful of requirements splitting them to tasks and starting the implementation. Other setup might be that all baseline requirements are first created, then a set of those is selected for implementing and tasks are issued to them. The best approach is dictated by the size of the project and its domain. As the requirements evolve, so do the issues: new ones are created, old ones closed, modified or rejected, tasks are started and completed.

The creation process of an issue consists of seven steps. The first step is deciding the title of an issue. The title should include some kind of reference prefix that is different for requirements and tasks. Examples could be "(XXX-100): Title goes here" or "#123: Title goes here" for tasks and "REQ 2 Title goes here" or "R2: Title goes here" for requirements. The prefix format is up to the users, but it should support the hierarchy: if a main task has prefix format "XXX-100", the sub task could have "XXX-100-1" or "XXX-100.1".

The second step involves writing the description of the issue. This is a very crucial step and should be paid careful attention. The main question to answer is how extensive the description should be? There is no right answer here and it depends from a lot of factors. How experienced the project team is? Who creates and manages issues? Does the team use concurrently another documentation tool? If too much information is given, the issue becomes cumbersome and it might be hard to find the core information. On the other hand, too vague descriptions might cause guessing, misleading or a need to consult somebody with better knowledge thus wasting time. Of course, if the project team is highly experienced and proficient with the domain of the project, it might be a good idea to leave room for individual thinking and implementation. This endorses the team and acts towards agilism which expects that experienced developers reach the best outcome without handholding.

Requirements on the other hand should be as unambiguous as possible to make sure that every stakeholder understands them the same way. For example, developer can implement a feature the way he feels it is the most suitable, when customer had a very specific need, mostly likely for a good reason. In the worst case, the implementation is unusable and so precious time is wasted in vain.

The golden middle way can only be achieved by trial and error. The bottom line is that the issue should always have a description which is tangible. For tasks, anyone from the project team should be able to tell what the task truly means just by reading the description. The guideline suggests that as much of the task specific information as possible should be directly in the description. It should be noted that if a main task type issues contains such elements that could – but haven't - be split to sub tasks, the description must also cover them. By doing so, the guideline tries to make sure, that usable information doesn't get lost or ignored.

A large amount of pictures, documents or other relatively static material should be archived somewhere else. For example mockups of UI are usually related to multiple requirements or tasks, and therefore should be only in one place. For requirements it might be a good practice – especially if they have a lot of information themselves – to document them to the wiki. For a requirement issue, a short description and link is sufficient in this case.

The format of issue's description is not predefined, but the guideline highly recommends that some sort of logical structure is used throughout each issue. Readability is further enhanced when this is combined with GFM used as syntax for descriptions. GFM's absolute benefit is that the description doesn't need to be in a plain text, making it much easier to create aesthetically good documentation.

One way for creating a logical structure is dividing the description in four topics: description, information, task list and references. The description topic should always be present. It describes the issue in few sentences, bringing up the issue's core meaning. The information part consists of all the relevant information. If this data is stored somewhere else, links should be used to point out the other source. The task list topic should only be present in a task or sub task, and it is complementary. The task list is utilized when a task involves such steps, that are known or wanted to be monitored, but their size is small or the connection to the parent issue is strong, and therefore they are not split to own tasks. They should also be used as sub task replacements if sub tasks are not used. Users should be aware that task lists are just a cosmetic addition to the hypertext; they don't affect the issue – like closing it – in any way. The last topic is references to other issues listed one below another.

Otherwise the format of the description should be kept as simple as possible. Although GFM allows a wide set of text editing options, it is not necessary to use them all. An interesting addition is so called @-notation or "mentioning". It is part of GFM and because of that can be used in any text area that supports GFM. Mentioning somebody creates a filter for the one mentioned. This feature can be used to notify people that they are needed or that they can be contacted to get further information. Mentioning feature can also be directed not just individual persons but teams inside GitHub. Every user has their own account which basically is their alias. Every account can also belong to an organization created inside GitHub. An organization can be the company user works for. Inside an organization an account can belong to a team. Teams are great for dividing people to groups based on for example their job. There could be an own team for testers, UI experts, backend developers and so on. Mentioning a team notifies everyone in that team.

Because GitHub itself doesn't implement any kind of hierarchical constraints between issues, such behavior must be implemented manually. The three ways used in this guideline are naming conventions for issues, labels and issue references. An existing issue can be referenced from another issue establishing a link between them. To be precise

referencing an issue from another issue means that within the first issue's view, there is a hyperlink pointing to the second issue. GitHub has an automated feature and annotation for this exact purpose. When the reference is done this way, an automatic comment is listed to the issue's comment section to highlight this relation. What makes this procedure a little problematic is that the issue referenced must exist to get the link working. This causes inconvenience if it is desired that for example a main task refers to its sub tasks and every sub task backs to its main task. In such a case, if the main task is created first, the sub task references cannot be done until they are created. Most often the links are wanted both ways: from top to down of hierarchy but also from down to top. The guideline recommends that in order to accomplish this, references are updated when new issues are created. Referencing issues follows the same rules as the hierarchy of them. The exception is that if an issue has a close relation with another issue which is not a descendant or an ancestor to the original issue, a relational reference can be created. Otherwise the reference should always be to only a direct parent or child of the issue.

Steps 3 to 5 consist of assigning additional information to the issues: first labels, then people and lastly milestones. Labels are an essential part of the guideline and they are the main solution for creating visibility and status tracking. The guideline doesn't explicitly state what exact labels should be used rather it states possible categories for them. The categories are to be chosen based on the needs of the project.

There are a total of six categories suggested by the guideline:

- The type of the issue.
  - Values: Requirement, Sub requirement, Task, Sub task.
  - Explanation: States what higher level type the issue represents.
- The sub type of the issue (for task and sub task only).
  - Possible values: Feature, Bug, Enhancement, Other.
  - Explanation: The sub type of the task. This allows for example a quick filtering of bug reports. The list of values can be a lot longer depending on what kind of parts the tasks are split. The sub type also opens the nature of the issue without a need to go through the description.
- Status.
  - Possible values: In progress, In testing, In customer acceptance, Rejected.
  - Explanation: In what part of the process the issue is going. These are additions to GitHub's native statuses: open and closed. Values must be used based on the processes used in the project.
- Requirement level (for requirement and sub requirement only).
  - Possible values: Required, extra.
  - Explanation: In certain cases, a requirement can be closer to "nice to have feature" or "if there is time". In such cases, this label can be used to visually distinct these requirements. The aim of this label is to roughly

visualize whether the requirement really is needed to fulfil the software. This label is not to be confused with the "priority" label.

- Priority.
  - Possible values: High, medium, low.
  - Explanation: In conjunction with the requirement level, this label tells on how high of the prioritization list the issue is. The guideline suggests that this is mainly used with tasks and sub tasks, since these are the concrete actions needed to fulfill a requirement. If a certain requirement needs to be prioritized, the priority of its associated tasks should be raised to better reflect the urgency of the work. Of course the label can be assigned for a requirement or sub requirement as a reminder of requirement's prioritization but for example milestones are more suitable for this purpose.
- Miscellaneous (also called as flags).
  - Possible values: Blocked, duplicate.
  - Explanation: This label shares additional information regarding issue's status. For example flagging an issue as "Blocked" tells immediately, that there is something causing interference for the issue, and it must be investigated.

The guideline recommends that every category has its distinctive color. Every single label in that category should have an own shade to further ease the recognition. A short prefix in the beginning of each label – yet again depicting the category it belongs to – complements the color coding.

Generally there should always be a maximum of one label per category assigned to an issue, miscellaneous labels being an exception. Excluding the first category the matter is not whether these are the exact categories, but that categories are decided in the first place and then used accordingly.

After labels comes assigning people which basically means that the assignees are responsible for the implementation of the issue. They are linked directly to the issue.

The use of milestones is recommended and they complement the status and priority categories of the labels. Two use cases are identified by the guideline. The first one is an iterative approach, where one milestone includes issues (both requirements and tasks) to accomplish before a given date. The other is to use them for grouping issues, for example based on a bigger feature. An issue can only belong to one milestone, so the role of milestones must be decided beforehand. Of course milestones can be used with imagination. To demonstrate this, a project could have three milestones. One for the current iteration that holds every issue handled at the moment. One milestone could be used for issues that are ready to start work with, and one for issues that need further specification or other adjustments.

The sixth step is publishing the issue, i.e. pressing the submit button. The seventh instructs that the creator should now go and update the references of other issues.

When an issue is created and references updated successfully, it starts its own lifecycle. This will be covered in the Section 6.8.

6.6. Status tracking and traceability of issues

As stated, status tracking and tracing requirements are core activities of RM. We have already created a base for them in the previous Sections.

Traceability is mainly gained by the naming conventions and references, but also the labels from the type and sub type categories enhance it. The status tracking however relies on milestones and labels, like status, priority and miscellaneous.

The monitoring of these aspects requires taking the filters into use. Probably the most useful ones are the milestone and label filters. Although these filters are simple, they are quite powerful, especially since the filters are a part of the URL, so bookmarking the most used filter combinations is possible. Of course, there still are a lot of scenarios where the filters are no use. For example a filter combination that lists every task with low priority in a defined milestone is applicable. A filter for getting every main task which has a sub task is invalid, and cannot be created with GitHub's filter combinations.

This brings up the question about what kinds of filter combinations are then needed. There is no general answer here, since it again derives from the scope and size of a project. Some general statements are however mentioned in the guideline. Developers should be aware of all the issues that mention them and the issues that are directly assigned to them. Especially priority and milestone information is relevant to them. Project managers should pay attention to every task that is currently worked with. This includes following milestone deadlines and task list progression if issues contain them. Also issues needing special actions – like ones flagged "blocked" – must be responded without a delay to keep the work flow going. They should also be aware of bug reports and enhancement proposals.

6.7. Combining version control and wiki to the issues

The guideline focuses on the issues and issue tracker, but some thoughts are shared about the wiki and version control and how they could be used to complement issues.

The wiki has already been discussed in the issue creation. Depending on how much information is attached directly to the issue, the wiki is an excellent alternative for lengthier data masses. As it is within the repository, the convenience is increased due to the easy access. Should the wiki be utilized, establishing a logic structure is recommended. The structure itself doesn't concern the guideline as long as it is consistent and logical. Other recommendation is that every external document should be linked to the wiki. A collection page can be created for this sole purpose. The page lists every relevant link with

a short description, preferably grouped under headlines. Should the wiki contain information regarding requirements, careful attention is to be paid that this information doesn't conflict with the issue and tasks of the requirement.

The version control itself offers some interesting interaction possibilities. As it is possible to add a mention to the description of the issue or comments, the same is applicable to the commit messages. Referencing an issue is also made possible. Therefore to enhance visibility of what is going on developers are encouraged to always reference the issues their commit affects in commit messages.

6.8. Updating and maintaining issues

After an issue is created, its lifecycle within the issue tracker begins. During the lifecycle, the issue is going to be updated several ways. The most common forms of updates are: changing labels, assigning milestones, assigning people, commenting issue, referencing it from commit messages and closing it.

Stakeholder communication regarding issues is a common update action. Whether the issue is a requirement or a task, it is prone to some sort of communication. The challenge is that the communication can occur in multiple places: in emails, conference calls, face to face discussions and so on. These interactions can cause something to change within the issue or they may reveal some new information or insights that help the implementation. It is imperative that in such cases, the issue is immediately updated to reflect the changes.

Whatever is the medium for the communication, the descriptions of issues must always be up to date. This guarantees that the latest and best knowledge is easily findable without a need to go through every single comment or other documentation. The comment area of any issue should be actively utilized by every stakeholder. The benefit from moving discussion to issue's comments is that it always leaves a visible history. A hindrance is that an active conversation can make comment areas long. Especially if it involves debates – for example about the implementation – it is even more imperative that the outcome is updated to the description if it causes changes to it. Coupled with clarifying comment it is easier to follow issue's progression.

A crucial note is that users should be careful when deleting old information. It might be so that the new information becomes questionable at some point, and a need to compare it with the old information arises. If the old information is deleted, there may be no way to find it again. On the other hand, the description should not contain unnecessary information. If there is a chance that the old information is needed in the future, it can be moved from the issue to a suitable page in wiki.

An example of an issue's lifecycle could be the following. Project manager creates a sub task for an existing main task and assigns it to developer X. This involves the basic issue creation steps. After a while, the priority is changed by a request of the customer. The priority label is changed. Due to the increased priority, developer X starts to work

with the task. The status label is updated. In the middle of implementation developer X finds out a major problem so he adds the "blocker" label. The arisen problem is discussed in the issue's comments and more widely in the daily face-to-face meeting. Based on the input, the description is updated. The feedback helped X to solve the blocker and he continues the implementation changing the status label as needed. If the issue has a task list, X updates it to keep other posted on the status. Finally X is ready to make a commit. He references the issue in the commit message and mentions quality assurance person Y. The task is now ready for testing. Y starts the tests and changes the status if needed. When everything is in order the task is accepted and closed. It is possible to close an issue with a special syntax in a commit message but the guideline doesn't recommend this. Although it might save the developer from going to the issue tracker, it is prone to leave the issue outdated.

When an issue is closed, it is wise to submit the last comment that explains why the issue is being closed. Normally this is because the issue is done, but what if issue is closed for some other reason? Creating a short explanation takes a little time but helps keeping everybody informed and complements the discussion of the issue.

The example reflects some of the actions associated with the issue's lifecycle, but not all. For example a project manager should be monitoring the issue's progression, comments can be made without an arisen problem and a milestone could change multiple times.

As important as it is to keep the description updated, is to make sure that labels are used and updated. The importance of the labels is to visualize issues and different aspects of them in one view. Should labels be misused or not updated, an unnecessary waste is generated. For example a project manager who is interested about the progress of tasks needs to find this information some other way even though it could have been presented with a minimum effort in the issue tracker using up-to-date labels. Letting information get old causes mistrust among the team and may lead for rejecting the guideline's practices. It also interferes with the RM and its objectives.

6.9. A bug report and an enhancement proposal

Throughout this guideline presentation I have advised using certain processes ranging from simple to lengthier. One is yet to introduce.

It concerns creating a bug report or an enhancement proposal. Both of these are special issues that are to be used throughout the project. A bug report is used when somebody spots an incorrect behavior or an error in the software. Within the guideline the process for a bug report is simple: create an issue but assign only the "task" and "bug" labels. A bug should always relate to either a task or requirement and thus it is a good practice to reference that in the bug report. This poses a problem since bug reports should be doable by anybody, meaning that the reporter – for example the customer – may not explicitly

know or remember the issue from which the bug originates. If this is the case somebody who is more familiar with the existing issues, should update this information as soon as the bug report is received. The bug issues could also have an own prefix in the title to better recognize them, though the label is more important.

This is not the only way of handling bugs. The other way is to use the bug label as a flag that is given to the task when a bug is found. The description goes to the comments of the task. However, this is not recommended because now the bug is lost inside the task and so it loses its traceability. If the bug is minor one, like a spelling error, this may be acceptable, but otherwise it is usually unknown of how big the bug really is justifying the creation of a separate issue.

An enhancement proposal is the other special issue. In the guideline an enhancement proposal is a suggestion relating to a feature or requirement that in most cases comes from such stakeholders who are not authorized to make the decision by themselves or the decision is up for a discussion. These suggestions often consist of thoughts related to improving certain aspects or functionalities of the software. In an agile environment this kind of improvements can be done just by discussing with the customer who may immediately accept or reject it. Again, if a proposal is accepted and is only a minor one, there is no necessity to create an issue. If a proposal is larger or requires further discussion or other actions an issue should be created. An enhancement proposal is created like a bug report – expect with the label "enhancement" - and relates to either a requirement or another task. This depends about the nature of the enhancement. If such an enhancement proposal is accepted a new task or sub task is created for the actual work or implementation that is required. The issues that are affected by the enhancement should be updated to keep up with the new situation. After that the original enhancement proposal is closed and the newly created task continues on. Should an enhancement proposal get declined, it receives the "rejected" label and is closed.

6.10. Changes in the guideline

The guideline has evolved from the first version that was published in October 2013. The purpose was to iteratively develop the guideline based on the feedback it received. During a time period from October 2013 to May 2014 feedback mainly consisted of the case study that was conducted.

Overall the guideline has stayed relatively stable from the beginning and hasn't received major updates to its content. The single biggest update of the guideline is in which order it is presented in the slideshow. This overhaul update affects the whole guideline. The goal was to make it more logical and readable for any reader: the first version had quite much specific information for the case study project. Reformatting also affected the referral of the guideline. The order of items was changed to keep in synchronization with the slideshow. The references to the slides were also removed.

Some changes to the content itself were made after the case study, but also because of my extended personal experiences with different kinds of projects during the past half an year.

The first notable addition is that I broadened the discussion about whether sub tasks should be used. The motivation came from my own experiences. I realized that there are valid situations where a main task coupled with a task list is enough. Due to the realization, I made better arguments for and against of sub tasks, highlighting that the main objective is to get the relevant information easily displayed.

The second modification considered embracing the team. Although the guideline was meant to be applicable to a wide array of projects, the first version expected that the case study team had a strict division about who creates and handles issues. Retrospectively it would probably have been beneficial to point out from the beginning more clearly that the guideline and RM are everyone's responsibility and everyone's contribution is valuable. This is now more emphasized.

The third change handles milestones. They are an extremely powerful tool, and can be used very creatively. However the first version of the guideline failed to highlight this. Milestones were discussed but it was assumed that they are going to be used like traditional iterations. With added examples users should now be able to identify additional purposes for milestones that could help monitoring issues.

Updating and maintaining issues received a new note about preserving the old information. Even though the newest information is the one that is utilized, it's not always the case that the old information should automatically be discarded. The focus is presenting the best and newest information as well as possible, but the team should be aware about the old information. For example tracing requirements and how they have evolved becomes hard if the initial situation is lost.

Other new recommendation was that issues should be closed with accompanying comment to clarify why the issue was closed. In the case study there was a wide array of issues that were closed, but it was not clear whether they were completed, rejected or something else. This uncertainty was partially due to somewhat disorganized use of labels therefore up-to-date labels are even more emphasized.

Creating the special issues – bug reports and enhancement proposals – has been clarified to make it more clear how the guideline expects them to be used. Of course the proposed way of the guideline is not the only right way, but this is not the problem that is tackled with the clarifications. What makes these special issues problematic is if there is no systematic way to use them. By defining such a way the guideline tries to make these issues used more systematically, thus making them more useful for the RM.

7. Evaluating the guideline

7.1. The case study

To assess the usefulness of the guideline in a practical level, a case study was conducted. The project that was chosen for this purpose was part of a student project course in the Tampere University's School of Information Sciences. The projects of this course usually last around nine months and consist of two to even four project managers and average of four developers. The project is carried out during a normal university semester, meaning that team members are likely to participate to other course beside the project. Developers are required to spend at least 100 hours for the project to pass the course. The corresponding hour cap for project managers is 140 hours. Every project has its own customer who can be either from inside the university or completely external organization.

In this case, the customer came from inside the university. The project team consisted of four developers and three project managers from whom one had to leave in the middle of the project. None of the team had earlier experience with GitHub's issue tracker, though two of them mentioned having used GitHub itself in the past. The team was multicultural and used English as the communication language.

The goal of the project was to produce a mobile puzzle game which could be used to introduce Computer Science for student applicants. Due to the open source nature of the project, GitHub was chosen as a version control system before the project entered the case study. By the request of the customer, as much information as possible was stored solely on GitHub to make it possible for someone else to continue the work after the project finishes. This also concerned the RM.

The case study progressed according the following. In the very beginning of the project, I took part to one of the official meetings of the project. The official meetings were meetings to which also the customer was invited. These kinds of meetings were arranged from five to six times. At that specific meeting, I went through different views of GitHub, especially the issue tracker. The project had already begun their management processes in Redmine, so the next task was to convert these from there to GitHub. The recording of the working hours was kept in Redmine, because GitHub doesn't support such feature.

The conversion could begin few days after the meeting when the first version of the guideline was published and handed over to the team. A couple weeks later, the two-sided A4 summary paper was also published in hopes that it would help the deployment of the guideline. At this stage the responsibility of putting the guideline into use was transferred to the team.

My role as a researcher was to stay back and observe how the team utilized the guideline. Had problems arisen, I would have consulted the team on possible solutions.

The observation was conducted from three perspectives. I kept an eye on what actually happened in the project's issue tracker, I followed their internal discussion on the chosen platform (Facebook) and I attended all the official meetings to get first hand impressions.

This continued until the end of the project when an online survey was filled by the team members. The total number of participants to the survey was five: three developers and two project managers. The questions of the survey aimed to collect more information about how the team felt using the guideline and whether there was something they missed or disliked. The survey had two sections: common questions for all participants and an additional set for both project managers and developers.

Common questions tried to establish background information about whether participants had previous experiences with GitHub and its issue tracker. It was also asked how well they had studied the guideline and did they felt it was easy to use it. After the developer and manager specific questions every participant was given a free word and possibility to tell their overall feeling about the guideline

Developer specific questions approached the subject from two perspectives: what developers thought about their role in RM and how they perceived the guideline from this part. Developers were asked what they thought about their role in RM, was it difficult to use issue tracker or did it manage better than anticipated, what they found the most important in the issue tracker and was the guideline more helpful or devious.

Project managers answered first about their earlier experiences with requirements engineering. This was to clarify if it was a new area to them and could possibly explain actions they had taken with it, after that followed a set of guideline specific questions. The filters and labels played a key role in managers' question set. Labels are a crucial concept in the guideline and probably one of the most used feature within project managers combined with filters. Managers told their opinions about how well the label categories fit to their purpose, whether they helped bringing more visibility and had filters been useful. There were also questions about the overall processes of the guideline, how easy it was to follow them and had there been any problems. Lastly managers got to share their view about the most important features of the issue tracker from their perspective.

7.2. The results of the case study

Based on the observations and the survey a few things could be stated. The first observation was regarding how the guideline was perceived. It took the project managers – who had the responsibility of managing the issues – several weeks to convert the issues from the original requirements platform (Redmine) to GitHub. At this stage there were mainly just requirements, not specific tasks. This caused a new set of problems: when the issue tracker was finally taken into full use, the project was in a mid-way. Some of the traceability and status tracking were lost because of this. It also meant that the team didn't have extra time to familiarize itself with the guideline. Goals of the project were pushing

in hard, and as the team consisted of students with less experience in software development, they prioritized implementing the code over information documenting and systematic monitoring.

The team's low experience with the software development was reflected on how the issue tracker was perceived: *"I think the management of tasks and requirements is the role of the managers, developers should mostly care about doing the tasks and fulfilling the requirements.", "-- Managers should just fiddle with the requirements."* The guideline aims to embrace the whole team to adopt agile and lean practices. This means that the whole team is responsible of the guideline and its usage, even though there should always be at least one person responsible of the issues. If developers don't think they are supposed to be involved in requirements handling, it might affect how actively they take part for example in discussion within issues.

The rush with the issue tracker also meant that not all practices and recommendations of the guideline were used. Like one of the participants put it when asked about the problems of the guideline *"It probably was just this case, we didn't take it into our work fast enough to get benefits from it."* This is in fact clearly visible in the issue tracker. The most notable missing practices were: logical and systematic issue descriptions, ignoring issue references and taking liberations with the use of different label categories. This highlights the strength – but also the weakness – of the guideline. As it became clear throughout the observations and finally from the survey, the whole team felt that the guideline had helped them achieve a more coherent management experience. The guideline is not strict with its recommendations and practices and as such can be utilized almost in any kind of project. The aspects that are not suitable for the project out of the box are either completely left out or modified to the needs.

However, this is also a drawback. The practices and recommendations are built on top of each other. It is possible to cherry pick only the most suitable ones, but this might leave the chosen aspects handicapped. For example, if labels and their categories are used, but hierarchy of the issues is neglected, traceability suffers immediately. Other possible consequence is that what the guideline tries to solve – not knowing how to efficiently use the issue tracker – is in fact generated instead. For example labels lose their potential power if not used systematically. In such case labels can still be a valuable asset, but their effect is reduced.

What was interesting to notice from the survey was that the team didn't feel like they had skipped or neglected practices: when asked about following the processes both project managers answered that they were followed. This contradiction is most likely related to the rush with the guideline's deployment. One project manager shared a thought that the visibility improved when they started adding proper descriptions to the issues. This demonstrates the need for information sharing and collaboration, and how it benefits the whole team. Had the descriptions followed more closely the guidelines

suggestions, even better results would have been achievable. Yet again an important remark is that though practices were not used exactly by the book, the team felt that the guideline had helped them: *"I think the guideline worked quite well for us. I cannot really remember a case when we had serious issue with the guideline"*, *"I believe it was very efficient"*, *"Overall, the guideline was useful and logical, I did not find any inherent flaws in it."*

The difficulties the team faced with the guideline where mainly related to the limitations of issue tracker and inexperience with it: *"I didn't notice the "close" button next to the comment one. This was when issue was opened. I was little bit confused how it was going to be closed. The whole tracker kinda opened to me in the two last weeks."* Other developer reported problems with the overall concept of the issues in GitHub. This was not unexpected since none of the team had any previous experience with the issue tracker, but it could have affected the outcome: the guideline requires that the basic knowledge of the issue tracker is acquired beforehand.

As an agile approach offers the team a possibility to more freely choose what they will do in each iteration, there is also a chance that inexperienced team is not sure what to do next. This is a problem from both management's and developers' side. If developers lack the experience, it is in the project managers' duties to point out what needs to be done. A developer commented about this matter: *"-- at some point it would have been great to have more concrete goals for each week. Instead of just "I could make this functionality"."* This indicates that project managers were in some situations unable to see what needed to be done and with what priority. Making those calls requires good domain knowledge so that requirements can be split into doable parts. The guideline doesn't get involved in that process but after splitting is done the guideline helps with prioritization and overall situation monitoring by making requirements and tasks visible and more informative.

The most appraised feature of the issue tracker and the guideline was labels and practices used with them. Both managers thought that labeling system was meaningful and worked well in the project. Their efficiency was well recognized.

The results of the case study – both observation and team's feedback through the survey - led to minor fixes in the guideline. The changes were mainly putting more emphasis for practices that could have been utilized better. Especially a systematic use of labels, making sure that they are always up to date and providing enough information in the issues' descriptions were aspects emphasized more clearly. The observations resulted in a completely new addition to the guideline: closing issues with accompanying comment. These aspects were discussed in the Section 6.10.

7.3. Implications of guideline's practices and recommendations

The feedback from the case study was positive and speaks on the behalf of the guideline. But how well does the guideline cope with the objectives of RM and lean principles in a theoretical level? Already throughout the presentation of the guideline, I have shared

thoughts about effects of different parts. In this Section coupled with the Section 7.4 I will systematically go through them. The following paragraphs summarize how the different practices and recommendations relate to achieving successful results.

Separation of tasks and requirements. The most important benefit from this practice is to get issues split to more comprehensible pieces. When all information related to a requirement is not one big chunk, everything is made easier. This also forces the project team to thoroughly think each issue and its implications. The drawback is that the splitting is only as good as is the team's experience and knowledge – especially about the domain.

Hierarchy between issues. Hierarchical dependencies and relations work closely together with the separation of tasks and requirements. It increases the dependency visibility when each issue can be traced at least upwards, preferably also downwards, should references between issues be in both directions. It is also easier to comprehend the whole picture when relations are easy to follow within the issue tracker. What makes this part a little devious however, is the fact that the issue tracker doesn't support automated hierarchy structure, thus it must be manually created and maintained. This takes extra time, and is easy to broke e.g. by not updating the references. Still, it is very beneficial. When considered from the RM's point of view this practice pretty much helps with every objective.

Naming conventions includes both the issue titles and labels. This recommendation tries to make it even more convenient and straightforward to spot what the issue is all about. Without the prefixes in issues' titles, viewer may need to spend more time to figure out how the issue he is looking at relates to others. Again some manual work is needed, because prefixes must be given by hand, and for example running number part must be recalled from somewhere.

Issue descriptions. This is probably the most two-edged sword of the guideline's practices. The question is what is enough? There is no golden mid-way, because so many factors influence it. If too much detail is given in the description, it wastes time and might leave developers unmotivated, because it can seem as handholding and not trusting the team. On the other hand, too vague description can cause the task to be completed without the best knowledge, possibly leading to misassumptions or other consequences. One of the elements of a good team in a lean world is knowledge sharing, and although heavy documentation should be avoided, it must not be neglected completely. Clear, convenient and thoroughly thought, yet not over long descriptions enhance the knowledge of the team and make it easier to follow what is truly wanted. Creating the business value is the main goal, and knowing what is wanted is a key to it. Syntax suggestions are there to make descriptions more readable and less time consuming to digest.

Label usage. The labels are a major factor for providing better status tracking and traceability. They clearly state what type the issue is and what is going on with it. All of this is viewable by just one look, saving time. However, to make labels beneficial they must always be up to date. By neglecting their usage is the same as not using them at all.

Assigning people and milestones. Issues are easier to track and monitor when they are included in a milestone. Assigning people makes it easier to ask questions, for example how the task is progressing.

Utilizing filters. If there are a lot of issues, the filters are the only sensible way of coping with them. When suitable filters are found, users can narrow down the issue list with ease.

Combining version control. Referencing issues from commit messages, makes it convenient to follow straight from the issue itself, what has been accomplished with it. No need to go through commits themselves to find it out.

Combining wiki. When everything is found from one single web page, nobody needs to try and remember what was the correct URL or login info. The wiki is a very powerful tool and offers a wide variety of formatting options, suitable for most of the basic scenarios occurring in today's software development documentation.

Creating, updating and maintaining issues. This aspect truly tries to embrace and commit the whole team. Although there should be always somebody responsible for the issues, it is the job of the whole team to handle them as agreed. The lifecycle of issues offers a great way for developers to express their views and thoughts in comments. Everybody should be encouraged to actively take part and thus create an environment where an individual person feels appreciated and has power to make changes.

Specialized issues. Practices with bug reports and enhancement proposals aim to make these special issues more used by establishing how they can be utilized systematically. Bugs are a critical aspect of building quality in, and they must be handled efficiently. On the other hand enhancement proposals make needs for change more visible and concrete when they are not lost in random conversations.

7.4. The main effects for requirements management and lean principles

In the previous Section I showcased what are the implications and limitations of the practices and recommendations. The following list will summarize the key aspects that try to enhance objectives of the RM and the principles of Lean as introduced by Poppendiecks' [2007]. The first four paragraphs cover the RM objectives, the rest paragraphs are for lean principles.

Change control. Working and successful change control requires that there is a way to get an overview of how different requirements and their implementations and components relate to each other. This greatly helps solving the impacts of the proposed change. A specialized issue – the enhancement proposal – gives a convenient tool for

formally proposing such changes. Of course, in an agile environment, changes don't always follow a formal path and they can be accepted for example in face-to-face meetings with the customer. The guideline enforces that issues and the information they contain is always up to date. It is something that change control requires. However, the team is responsible that the descriptions are updated causing a potential risk, if such activity is deemed unworthy.

Version control. The issue tracker itself holds all requirements together. Every issue is given an unambiguous identification number by GitHub that can be used referencing the issue. Versioning itself is not directly supported so the guideline suggests that the old information is not erased, but somehow tracked as the team sees fit in the project's scope. The naming conventions and type labels are there for identifying purposes. The worst enemy is the team itself: if for example labels are not used systematically and prefixes from titles are missing, version controlling is flawed.

Requirements tracing. Separating requirements and concrete tasks and establishing a hierarchy between each issue creates a clear structure for tracing requirements and links they have. This strongly requires that the references in descriptions are used and maintained properly. Without such actions, the traceability links are easily broken. The aforementioned naming conventions also help identifying relations between issues.

Status tracking. The labels are absolutely the best way to accomplish status tracking. The suggested categories are crafted so that tracking requirements and their tasks is as straightforward as possible. Although the label system is very plain, it is still an efficient tool. The drawback is that labels are mainly toggled with tasks, not with requirements. Due to the limitations of the issue tracker the requirement must be opened and the related tasks fetched through there. Well-picked naming conventions can ease this, but it is still problematic. If the number of requirements is relatively low, the problem is not as bad as with higher requirement numbers.

Eliminating waste. The guideline tries to solve the problem of using too much time on activities that don't produce value. However, it is impossible to say how well this is achieved because waste is always relative to the project and its context. Overall it is not easy to claim what parts of the RM are wastes. The guideline takes the stand that a successful RM is not waste, and if it can be completed with ease, it is efficient.

Descriptions with enough information and encouragement for free speech in comments aim to generate more knowledge for the whole team. This contributes towards learning how to produce value more efficiently. On the other hand, "enough information" is a thin line and it is easy to go over or underneath. Unnecessary waiting often occurs when developer is not sure how to continue or has a problem that he cannot solve in a reasonable time. Sharing knowledge and using labels tackle this inconvenience. The processes of the guideline are narrowed down to the minimum and are inherently quite simple. They are also flexible, which makes them more suitable for various situations.

Splitting requirements to small pieces increases the chance that not-needed extra features are detected and rejected.

Optimizing the whole. This is more like an attitude of the team and cannot be created by this kind of guideline. However, I believe that sharing knowledge and discussing it increases the insight to the customer's mind and thus makes it easier to see the whole picture.

Building quality in. A very important factor of this principle is to understand the customer's needs and openly collaborate and communicate about them. Information sharing, keeping everything visible and actively using commenting functionality are ways the guideline proposes. Although face-to-face conversations and ad-hoc interaction situations are quicker than commenting, GitHub has made it very convenient to utilize the comments.

Learning constantly. Sharing and absorbing both information and knowledge are the keys for lean principles. These practices are already discussed in the above paragraphs. The splitting of requirements in small pieces forces the team to truly get a better understanding of the domain. This of course applies if splitting is done thoroughly.

Delivering fast & deferring commitment. Small tasks take less time to implement. If those tasks happen to be logical components, a fast delivery is made possible. Feedback and communication should occur all the time further improving the delivery and its quality. When the feedback is almost instant, corrective maneuvers can be done without a delay. This demands an iterative approach and the milestones are just the right tool for helping that.

Deferring commitment on the other hand requires the best available information. Keeping issues up to date and openly discussing them enhances this. What causes a conflict with this principle is that it suggests starting implementation work as soon as higher level conceptual design is ready. Though the guideline doesn't directly imply it, splitting requirements to tasks is needed before the work can begin. The problem can be avoided if the whole team is involved in the splitting operation. It should be remembered that a requirement is quite quickly converted to one or more main tasks that developers can start working with. These tasks can further be refined, if such need or desire arises.

Respecting people. The guideline aims for empowering the whole team. They are the ones that use the issue tracker, and they all should have at least some kind of interest on what is happening there. After all, it is their tool for viewing what needs to be done. The guideline doesn't restrict who can do what, though it expects that there is somebody on the team that is more interested about monitoring progress. Usually this has been the project manager, but it depends on how self-organizing the team really is. How well the empowerment succeeds is based on the abilities of the parent organization and whether it supports giving the decision power to the development teams.

7.5. Suggestions for developing features of GitHub's issue tracker

GitHub and its issue tracker are powerful tools. This doesn't mean that they are ideal for every single case but nonetheless with defined processes and practices they should be considered at least a viable option for any project.

During the development of the guideline I came to observe few aspects that GitHub could further develop. Before listing my observations, it must be discussed about the essence of the issue tracker. GitHub aims to be intuitive and lightweight to use. It offers convenient features and functionalities that user can take advantage of, hopefully enhancing their user experience and work flow in GitHub. A tool that is fun and easy to use is prone to be an efficient one. That is why the limitations of its features have to be put into a right context. It is unreasonable to ask them to implement features that would contradict the lightweight approach that is chosen by GitHub. This is especially good way in an agile environment that is designed to minimize all the extra effort and focus the time that is freed by doing so to the activities that bring more business value to the customer. It is about how to use GitHub with its features to achieve this. I have identified a total of five features that would benefit this guideline.

Automatic relationships between issues. At the moment, the guideline users must waste some of their time to manually create and maintain the relations between issues. This could be avoided if issues themselves had a functionality to automate this process. A lightweight solution could be the following. A "relations" header could be added to right sidebar of every issue. Under the header user has an option to select an issue to relate and the type of the relationship. Three types would exist: parent, child and related from which related is the default value. Every established relation is also displayed with the type and issue's title. When such relation is made from issue A to issue B, issue B's relations are automatically updated. Now user wouldn't need to manually go back and forth between the issues and update the references manually. New relation types could also be used to show issues in other formats than list. This leads to the second development subject.

Different views for issues. Currently the issue tracker provides only one way of viewing issues. Although this is in most cases good enough, adding few more options would customize the issue tracker for the user needs. If the above automatic hierarchy generation is implemented, one such view could somehow visualize the relations between issues.

Text filter. Although the issue tracker offers quite a nice assortment of filters one very relevant seems to be missing. This is a plain text based filter that would display only those issues that have the given text in their title. This filter could utilize some basic regular expression kind syntaxes for more powerful usage. GitHub does have a universal text search that can be used to search for issues. The downside is that the search results are

shown in a separate search result view, and thus it isn't very efficient in the guideline's context.

Separating references from comments. This feature could be implemented in the combination with automated relations handling. The problem is that now, when issue is referenced from anywhere, the reference is displayed in the comment area. Reference can quickly get lost there, especially if comments are used on a regular basis. My suggestion is that references are separated from the comments. A solution would be to split the comment area in two: the other half having description and comments, the other listing the references and their timestamps. Should automated relations handling not be implemented, every reference should be displayed on the designated column. Otherwise, relational references are displayed only under their own header on the right sidebar.

Labels visible to references. Currently references from an issue to issue don't display the label information associated to them. Only the open or closed status, issue's title and a possible task list status are visible. As the guideline relies heavily on labels, it would be very good to be able to see the current labels of every reference. That way, viewer can quickly see what is going on with referred issues. Even more important this would be if automated relations handling is not implemented and relations are handled manually by references. Now, had viewer been looking at a requirement he would have immediately seen the label information of related main tasks.

7.6. Suggestions for developing the guideline

The guideline tries to benefit from as many features of GitHub as is rationally wise. GitHub is constantly evolving and new features and updates are presented regularly. The guideline's practices were designed to work with the features that were available during the time period from autumn 2013 to spring 2014. Maintaining the guideline's usability requires keeping up with GitHub's development and modifying the guideline as needed.

One feature that wasn't taken into consideration in the guideline was pull request, which is one of the ways GitHub tries to make social coding more desirable [Using pull requests, 2014]. As stated before, GitHub offers free usage, but every repository made without a paid subscription is automatically public and visible to everybody. This is not limited to just users with GitHub account but literally any Internet visitor can read and see everything in a free repository. Users can also download the source codes or – if they have a GitHub account - clone an own copy of the repository; this is called forking. These factors influence especially open source software projects, because controlling them in GitHub is made so straightforward. In such cases the pull request feature comes in handy.

The pull request is basically a notification that somebody has made changes in their cloned version of the original repository (fork) and would like to publish those into the original repository. Pull request needs a fork from where the changes are then pushed to the original repository. This means that collaborators don't need access privileges to the

original repository in order to make changes, the changes just need to be accepted by the author of the original repository.

Pull request comes with a nice set of its own functionalities that each make it very convenient to use. Every pull request contains a title and possible description. This helps identifying what the suggested changes really do, without a need to go through the exact changes in the code level. The creator of the pull request can also determine from which branch in his own fork to which branch in the original repository the changes are going. One implication for this functionality is to have two branches in the original repository: a development and production branch. The production branch is the stable branch and direct changes to it are strictly controlled. Every update is first committed to the development branch. Once the development branch is stable, contains the wanted changes and is accepted by the quality assurance, the merge to the production branch is conducted. The third – and probably the most important functionality – is that when a pull request is received, contributors can see the changes in the code level and discuss them at the comments of the pull request. It is even possible to make inline comments to the code.

The guideline doesn’t utilize the pull request functionality in any way. The major reason is that in a traditional software project, every contributor has equal rights to make changes to the code base as they please. Branches can still be used and quality assurance and code reviews take place, but without the defined process of the pull request. Therefore the pull request is not likely to be used. Open source projects benefit from this feature but their RM process may be very different than what is expected by the guideline.

The guideline was designed mainly for the purposes of a traditional software project were the RM is likely to play an important role and it should be handled with a coherent way. It should also be noted that even though the pull request indirectly affects the RM, it mainly relates to how the implementation, testing and releasing processes of the project are planned to go.

8. Discussion

The aim of this M.Sc. Thesis was to create a semi-formal guideline for conducting RM process using only the tools provided by GitHub. To prove that the guideline is applicable and working, a case study was carried over and the guideline was also evaluated on how well it supports or conflicts with the RM objectives. Evaluation was also done against the lean principles which are well suited for assessing if the guideline is suitable for agile environments.

As far as my research shows, this is one of a kind topic: similar guidelines for GitHub platform with a properly surveyed case study were not found. There exist several blog posts about thoughts and suggestions of using issue tracker in GitHub, but none of them approached the topic as systematically as this Thesis. These blog posts also didn't have any real evaluation of how well their suggestions worked: the positive results were based on the mere feeling of the author [Bicking, 2014]. My conclusions are based on a case study and observations, not just vague opinions.

From the science's point of view, my research is interesting because it combines the lean principles into the RM. This is also something that has very few studies, if any at all. Although lean is something that should be utilized through the whole organization and its processes to be the most effective, the evaluation of the guideline gives more insight on how the RM can be coupled with the lean principles. While this is of course constrained by the choice of platform, GitHub, and its features, it does examine how the objectives of RM and lean combine and even support each other.

For the practitioners the guideline provides considerable, systematic and well documented instructions for how to use issues to handle the RM in a project that utilizes GitHub. The guideline is also evaluated by the case study, and though the study consisted of only one student project, its feedback was very positive. The personnel of the case study project felt that the guideline helped them achieve a more consistent RM process than they could have had without it. By comparing the guideline against the lean objectives I showed that the guideline is well suited and can even enhance an agile approach in a software project.

The guideline was created to be usable in a wide array of projects, though it is required that the project utilizes an agile approach. As the guideline is backed with lean – and therefore agile – principles it doesn't suit projects that use some other development paradigm. However it could be argued that in such cases GitHub's lightweight issue tracker might not be the best RM tool to use in the first place. Therefore the biggest factor for not using the guideline is the chosen development process of a project. The practices and suggestions of the guideline are designed to be easily understood and adopted in hopes for establishing a good base for taking RM in a serious use in any project. This is

especially the case in agile projects that tend to bury requirements engineering process below iterative development. As GitHub could be considered a programmer-friendly platform, handling RM in the same places can lower the threshold of developers to participate and take more active role in it.

Making this kind of general guideline means that, although there is room for customization without interfering the core ideas, the guideline may not be suitable for projects that strictly follow a defined agile method.

Like with any agile practices, the users are left with the responsibility to follow the instructions as they see fit. This can cause a problem, if users try to cherry pick the aspects that are suitable and easy, and leave out other aspects that may be deemed "wasteful" or "too heavyweight". This kind of selective use can greatly diminish the usability and results achieved by the guideline.

There are some limitations concerning this Thesis. The topic is very specific which makes it unique but at the same time difficult to generalize. The RM has certain objectives and accomplishing them depends on the tools at one's disposal. With GitHub we have a limited set of features but in some other platform the tools and their usage may be completely different. Although the evaluation combines the RM and lean principles, it wraps around the guideline's practices and tools offered by GitHub.

As argued in the Section 7.5 GitHub is evolving constantly. Features are updated and new ones implemented with a steady pace meaning that the features utilized in the guideline may change. Should the guideline stay usable, it must keep up with GitHub making sure that the practices are valid. Other point to keep in mind is that the guideline establishes certain processes and practices that fulfil its core ideas. Room for customization is left around them, but it may not be enough, if the core ideas are in contradiction with other processes of the project in which the guideline is used. On the other hand due to the possibility for the customization exists every possible situation cannot be taken into account. This can also leave unnecessary vagueness to the instructions, because instructions that could have been strict are now mere options and suggestions.

The case study was relatively small and limited in scope: only one project was involved into it. The project was also a student project that lasted only around nine months. The problem with students as developers and managers is that they tend to lack in the experience that is present in paid software projects. As the project was carried out during the university semester the developers and managers also had other course to attend to, leaving relatively small amount of time to work. An average time spent for the case study project was 100-140 hours, which is only 2.5 to 3.5 weeks' worth of full time job. The team was multicultural and this posed a minor problem with communication. English was used as the communication language but it was not native to any of the project

members. This was reflected for example to the survey results: some of the feedback was hard to interpret.

The guideline would greatly benefit from a larger scale case study that would involve projects with different settings, scopes and sizes. The problem here is how the projects would adopt the guideline and follow its instructions and not just cherry pick what to use. Of course a selective behaviour could indicate that there is a problem in the guideline but it could also mean that the team has not yet fully embraced the ideologies behind the guideline. If certain practices are systematically dismissed the study could reveal why so and present an alternative solution. Nonetheless a wider case study would make it easier to further evaluate the core ideas of the guideline and how well they really achieve their objectives. It could also highlight in what kind of situational contexts the guideline manages the best.

Projects with more experienced team members could also give better facts to support different kind of approaches. To give an example, using sub tasks versus not using them is sure to arouse conversation and arguments for and against. The guideline gives both of them as an option, but an interesting question is how the benefits and possible disadvantages would split. Professional teams could also bring up such aspects that the guideline hasn't taken into a consideration.

The guideline built and evaluated in this Thesis achieves its goal, that is: creating instructions for agile requirements management process utilizing only GitHub's own tools. To prove this a case study was conducted and the practices of the guideline and GitHub's features were critically assessed.

The results from case study were only positive. The project team from the case study felt that the guideline had given them a systematical way to approach requirements management and it had worked for them. This justifies that the guideline is applicable in a real life circumstances. The evaluation argues that the practices and recommendations of the guideline combined with GitHub's features succeed on supporting the four objectives of requirements management - change control, version control, status tracking and requirements tracing – in a theoretical level. As there exists none as systematical and well assessed guideline, the results of this Thesis are novel.

GitHub could further develop certain features to better suit the needs of the guideline. Such features are: automating relationships between issues, offering different views for displaying issues, providing a text filter for issues, separating issue references from comments and showing labels currently associated labels on issue references. These development focuses would benefit current practices of the guideline.

The guideline itself would benefit from a larger case study that involves projects with different sizes, scopes and backgrounds. Getting more feedback from experienced development teams might point out possible problem areas within the guideline or aspects that need refining. It is likely that GitHub is going to change the features utilized

by the guideline or add new ones, therefore the guideline should stay in a constant build & evaluate cycle to always provide intact instructions.

At the moment the guideline is the best solution when it comes to systematically designed and evaluated artifacts describing how to handle requirements management process in GitHub.

References

[A Short History of Git, 2014] A Short History of Git. (2014). Retrieved March 5, 2014, from http://git-scm.com/book/en/Getting-Started-A-Short-History-of-Git

[About Git, 2014] About Git. (2014). Retrieved March 5, 2014, from http://git-scm.com/about

[Abrahamsson et al., 2003] Abrahamsson, P., Warsta, J., Siponen, M. T., & Ronkainen, J. (2003). New directions on agile methods: a comparative analysis (pp. 244–254). IEEE. doi:10.1109/ICSE.2003.1201204

[Agile Manifesto, 2001] Agile Manifesto. (2001). Retrieved April 25, 2012, from http://agilemanifesto.org/

[Azad, 2007] Azad, K. (2007). A Guide to Version Control. Retrieved June 4, 2014, from http://betterexplained.com/articles/a-visual-guide-to-version-control/

[Beck, 1999] Beck, K. (1999). Embracing change with extreme programming. *Computer*, *32*(10), 70–77. doi:10.1109/2.796139

[Beck, 2000] Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[Bicking, 2014] Bicking, I. (2014). How We Use GitHub Issues To Organize a Project. Retrieved May 11, 2014, from http://www.ianbicking.org/blog/2014/03/use-github-issues-to-organize-a-project.html

[Bitner, 2012] Bitner, J. (2012, June 20). Managing Projects with GitHub. Retrieved May 11, 2014, from http://www.lullabot.com/blog/article/managing-projects-github

[Bocock and Martin, 2011] Bocock, L., & Martin, A. (2011). There's Something about Lean: A Case Study (pp. 10–19). IEEE.

[Cao and Ramesh, 2008] Cao, L., & Ramesh, B. (2008). Agile Requirements Engineering Practices: An Empirical Study. *IEEE Software*, *25*(1), 60–67. doi:10.1109/MS.2008.1

[Cockburn, 2000] Cockburn, A. (2000). Selecting a project's methodology. *IEEE Software*, *17*(4), 64–71. doi:10.1109/52.854070

[Cockburn, 2006] Cockburn, A. (2006). *Agile Software Development: The Cooperative Game* (2 edition.). Upper Saddle River, NJ: Addison-Wesley Professional.

[Cohen et al., 2004] Cohen, D., Lindvall, M., & Costa, P. (2004). An Introduction to Agile Methods. In *Advances in Computers* (Vol. Volume 62, pp. 1–66). Elsevier.

[Coram and Bohner, 2005] Coram, M., & Bohner, S. (2005). The impact of agile methods on software project management. In *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the* (pp. 363–370). doi:10.1109/ECBS.2005.68

[Davis, 1993] Davis, F. D. (1993). User acceptance of information technology: system characteristics, user perceptions and behavioral impacts. *International Journal of Man-Machine Studies*, *38*(3), 475–487. doi:10.1006/imms.1993.1022

[Dictionary.com, 2014] Dictionary.com. (2014). Retrieved May 24, 2014, from http://dictionary.reference.com/

[Dybå and Dingsøyr, 2008] Dybå, T., & Dingsøyr, T. (2008). Empirical studies of agile software development - A systematic review. *Information and Software Technology*, *50*(9-10), 833–859. doi:10.1016/j.infsof.2008.01.006

[Ernst and Murphy, 2012] Ernst, N. A., & Murphy, G. C. (2012). Case studies in just-in-time requirements analysis. In *2012 IEEE Second International Workshop on Empirical Requirements Engineering (EmpiRE)* (pp. 25–32). doi:10.1109/EmpiRE.2012.6347678

[Estublier et al., 2002] Estublier, J., Leblang, D., Clemm, G., Conradi, R., Tichy, W., van der Hoek, A., & Wiborg-Weber, D. (2002). Impact of the Research Community on the Field of Software Configuration Management: Summary of an Impact Project Report. *SIGSOFT Softw. Eng. Notes*, *27*(5), 31–39. doi:10.1145/571681.571689

[Favaro, 2002] Favaro, J. (2002). Managing requirements for business value. *IEEE Software*, *19*(2), 15–17. doi:10.1109/52.991325

[GitHub, 2014] GitHub. (2014). Retrieved March 5, 2014, from https://github.com/about

[GitHub New Features, 2014] GitHub New Features. (2014). Retrieved March 5, 2014, from https://github.com/blog/category/ship

[Glass, 2001] Glass, R. L. (2001). Agile Versus Traditional: Make Love, Not War!, 12–18.

[GNU arch, 2013] GNU arch. (2013). Retrieved May 21, 2014, from http://www.gnu.org/software/gnu-arch/

[Google Drive, 2014] Google Drive. (2014). Retrieved May 18, 2014, from https://www.google.com/intl/fi/drive/

[Gruber, 2004] Gruber, J. (2004). Markdown. Retrieved May 6, 2014, from http://daringfireball.net/projects/markdown/

[Hawrysh and Ruprecht, 2000] Hawrysh, S. P., & Ruprecht, J. (2000). Light Methodologies: It's Like Déjà Vu All Over Again. *Cutter IT Journal*.

[Hemel, 2013] Hemel, Z. (2013). Using Github Issues Effectively. Retrieved May 11, 2014, from http://www.stateofcode.com/2013/06/using-github-issues-effectively/

[Hevner et al., 2004] Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105.

[Highsmith, 2002] Highsmith, J. (2002). What is Agile Software Development? *CrossTalk*, 4–9.

[Highsmith et al., 2000] Highsmith, J., Cockburn, A., & Orr, K. (2000). Extreme Programming. *E-Business Application Delivery*, 4–17.

[Hilburn, 2013] Hilburn, B. (2013). Using the GitHub Issue Tracker for Large Projects. Retrieved May 11, 2014, from http://hokietux.net/blog/blog/2013/11/05/using-the-github-issue-tracker-for-large-projects/

[Hofmann and Lehner, 2001] Hofmann, H. F., & Lehner, F. (2001). Requirements engineering as a success factor in software projects. *IEEE Software*, *18*(4), 58–66. doi:10.1109/MS.2001.936219

[IEEE Recommended Practice for Software Requirements Specifications, 1998] IEEE Recommended Practice for Software Requirements Specifications. (1998). *IEEE Std 830-1998*, 1–40. doi:10.1109/IEEESTD.1998.88286

[Jones, 1996] Jones, C. (1996). *Applied Software Measurement (2Nd Ed.): Assuring Productivity and Quality*. Hightstown, NJ, USA: McGraw-Hill, Inc.

[Kilpatrick, 2003] Kilpatrick, J. (2003). Lean principles. *Utah Manufacturing Extension Partnership*, 1–5.

[Kirk, 2002] Kirk, R. (2002). Software configuration management principles and best practices. In *Product Focused Software Process Improvement* (pp. 300–313). Springer. Retrieved from http://link.springer.com/chapter/10.1007/3-540-36209-6_26

[Lam and Shankararaman, 1999] Lam, W., & Shankararaman, V. (1999). Requirements change: a dissection of management issues. In *EUROMICRO Conference, 1999. Proceedings. 25th* (Vol. 2, pp. 244–251 vol.2). doi:10.1109/EURMIC.1999.794787

[Li et al., 2012] Li, J., Zhang, H., Zhu, L., Jeffery, R., Wang, Q., & Li, M. (2012). Preliminary results of a systematic review on requirements evolution. In *16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012)* (pp. 12–21). doi:10.1049/ic.2012.0002

[Linders, 2011] Linders, B. (2011). CMMI V1.3. Retrieved March 4, 2014, from http://www.benlinders.com/2011/cmmi-v1-3-summing-up/

[Malmsten, 2010] Malmsten, C. F. (2010). Evolution of Version Control Systems-Comparing CENTRALIZED against DISTRIBUTED Version Control models. Retrieved from http://gupea.ub.gu.se/handle/2077/23474

[McBreen, 2003] McBreen, P. (2003). *Questioning Extreme programming*. Boston: Addison-Wesley.

[McCauley, 2001] McCauley, R. (2001). Agile Development Methods Poised to Upset Status Quo. *SIGCSE Bull.*, *33*(4), 14–15. doi:10.1145/572139.572150

[Mercurial SCM, 2014] Mercurial SCM. (2014). Retrieved May 21, 2014, from http://mercurial.selenic.com/

[Merisalo-Rantanen et al., 2005] Merisalo-Rantanen, H., Tuunanen, T., & Rossi, M. (2005). Is Extreme Programming Just Old Wine in New Bottles: A Comparison of Two Cases. *Journal of Database Management*, *16*(4), 41–61. doi:10.4018/jdm.2005100103

[Miller, 2001] Miller, G. G. (2001). The Characteristics of Agile Software Processes. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)* (p. 385–). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=882501.884709

[Nandhakumar and Avison, 1999] Nandhakumar, J., & Avison, D. E. (1999). The fiction of methodological development: a field study of information systems development. *Information Technology & People*, *12*(2), 176–191. doi:10.1108/09593849910267224

[Redmine, 2014] Overview - Redmine. (2014). Retrieved May 13, 2014, from http://www.redmine.org/

[Paetsch et al., 2003] Paetsch, F., Eberlein, A., & Maurer, F. (2003). Requirements engineering and agile software development. In *Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings* (pp. 308–313). doi:10.1109/ENABL.2003.1231428

[Paulk, 2002] Paulk, M. C. (2002). Agile methodologies and process discipline. *Institute for Software Research*, 3.

[Poppendieck and Cusumano, 2007] Poppendieck, M., & Cusumano, M. A. (2012). Lean Software Development: A Tutorial. *IEEE Software*, *29*(5), 26–32. doi:10.1109/MS.2012.107

[Poppendieck and Poppendieck, 2003] Poppendieck, M., & Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Boston, Mass.: Addison-Wesley Professional.

[Poppendieck and Poppendieck, 2007] Poppendieck, T., & Poppendieck, M. (2007). *Implementing Lean Software Development: From Concept to Cash* (Vol. 2006). Addison-Wesley Professional.

[Rakitin, 2001] Rakitin, S. (2001). Manifesto elicits cynicism. *IEEE Computer*, *34*(12), 4.

[Ruparelia, 2010] Ruparelia, N. B. (2010). The History of Version Control. *SIGSOFT Softw. Eng. Notes*, *35*(1), 5–9. doi:10.1145/1668862.1668876

[Russo, 2010] Russo, B. (2010). *Agile technologies in open source development*. Hershey, PA: Information Science Reference. Retrieved from http://site.ebrary.com/id/10318238

[Salo, 2014] Salo, R. (2014). GraduRepo. *GitHub*. Retrieved May 13, 2014, from https://github.com/Ripppe/GraduRepo

[Schwaber, 1996] Schwaber, K. (1996). Controlled chaos: living on the edge. Advanced Development Methods, Inc. Retrieved from http://cf.agilealliance.org/articles/system/article/file/786/file.pdf

[Sommerville, 2007] Sommerville, I. (2007). *Software engineering*. Harlow, England; New York: Addison-Wesley.

[Spinellis, 2012] Spinellis, D. (2012). Git. *IEEE Software*, *29*(3), 100–101. doi:10.1109/MS.2012.61

[Svensson and Host, 2005] Svensson, H., & Host, M. (2005). Introducing an Agile Process in a Software Maintenance and Evolution Organization. In *Ninth European Conference on Software Maintenance and Reengineering, 2005. CSMR 2005* (pp. 256–264). doi:10.1109/CSMR.2005.33

[Truex et al., 2000] Truex, D., Baskerville, R., & Travis, J. (2000). Amethodical systems development: the deferred meaning of systems development methods. *Accounting, Management and Information Technologies*, *10*(1), 53–79. doi:10.1016/S0959-8022(99)00009-0

[Using GitHub Issues to Manage Projects, 2012] Using GitHub Issues to Manage Projects. (2012). Retrieved May 11, 2014, from http://www.smashingboxes.com/using-github-issues-to-manage-projects/

[Using pull requests, 2014] Using pull requests. (2014). Retrieved May 17, 2014, from https://help.github.com/articles/using-pull-requests

[VCSHistory, 2010] VCSHistory. (2010). Retrieved March 5, 2014, from http://code.google.com/p/pysync/wiki/VCSHistory

[Waldmann, 2011] Waldmann, B. (2011). There's never enough time: Doing requirements under resource constraints, and what requirements engineering can learn from agile development. In *Requirements Engineering Conference (RE), 2011 19th IEEE International* (pp. 301–305). doi:10.1109/RE.2011.6051626

[Wiegers, 2009] Wiegers, K. E. (2009). *Software Requirements*. Microsoft Press.

[Zhang et al., 2010] Zhang, Z., Arvela, M., Berki, E., Muhonen, M., Nummenmaa, J., & Poranen, T. (2010). Towards Lightweight Requirements Documentation. *Journal of Software Engineering and Applications*, *03*(09), 882–889. doi:10.4236/jsea.2010.39103

Guideline in a nutshell

1) Overall idea
   a) How to agile handle requirements managing in a GitHub with a semi structured way.
      i) Lean Software Development principles aid in agilism.
   b) Guideline mainly focuses on Issues –tool, but some thoughts about wiki and version control are also discussed.
   c) In order to make this guideline to work: users should know the basics of Issues –tool (those who manage requirements, must have a deeper understanding than others), GitHub and GitHub Flavored Markdown (GFM) syntax.
2) Issues
   a) Depict requirements and tasks. Both of those have two levels: main requirement (or just requirement) and a sub requirement + main task (or just task) and sub task.
      i) Whether to use sub tasks or main tasks coupled with task list depends on the situation and circumstances on hand: the bottom line is that enough relevant information is always there.
   b) Hierarchical dependencies between issues must be handled manually.
      i) A quick run through of valid scenarios: requirement can have sub requirements and/or main tasks, sub requirements can have main tasks, and main tasks can have sub tasks. A task can belong to only one requirement and it must always follow the aforementioned structure (i.e. sub task must always belong to a main task).
      ii) Minimum effort with hierarchy: child issues refer to parent issues. Preferred way: also parent issues refer to child issues. Issues can refer to other issues that are not ancestors or descendants of the referencing issue and if there is a logical connection between them.
   c) Use labels and keep them up to date with every issue (concerns all team members on their behalf). They are a critical part of the visibility and monitoring.
      i) Checkout the suggestion for label categories and individual labels. Remember category prefix and color coding for every category.
      ii) Requirements should only have labels from type, status and miscellaneous categories. Priority can be assigned as a reminder but it is better to give it to the tasks themselves.
   d) Utilize filters. Create bookmarks to the filter combinations you use the most.
      i) Developers: be aware of issues that mention you and those that are assigned to you. Project managers should pay careful attention especially to the following: issues currently being worked with, task lists, milestones, blocked issues, bug reports and enhancement proposals.

ii) Milestone is a ‘special filter’ which has a deadline and issues can be assigned to it (issue can belong to only one milestone!).

e) Update the description of an issue when needed! *The description must be up-to-date.*

f) Task lists should only be used in the description of an issue.

g) Developers: always reference the issue your commit handles in the commit messages. Remember also to update the labels and task lists.

h) For a recommendation of syntax for issues, see the open example issues in https://github.com/Ripppe/GraduRepo/issues?state=open (especially the task –type issues).

i) Bug reports and enhancement proposals should be used appropriately.

3) Issue creation step-by-step.

a) Create title including possible reference prefix (recommended).

b) Write description.

i) Check https://github.com/Ripppe/GraduRepo/issues/5 and https://github.com/Ripppe/GraduRepo/issues/6

c) Assign labels.

d) Assign people.

e) Assign milestone (if needed).

f) Create the issue.

g) If the issue was supposed to be referenced from another issue, go and update that issue now!

4) Where to start?

a) Go through this paper. Refer to the actual guideline when needed. Check also the example repository https://github.com/Ripppe/GraduRepo/ -> live examples with explanations can be found there.

b) Decide what label categories to use, what are the colors for them, what are the labels themselves. Then create the labels.

c) Create first requirement and sub requirement issues to the issue tracker. Remember the hierarchical structure. Follow the steps for creating an issue.

d) Decide what kinds of tasks are needed to complete each of these requirements.

e) Decide if those tasks should be split further to smaller pieces and create main tasks and sub tasks accordingly. Remember the hierarchical structure. Follow the steps for creating an issue.

f) If milestones are needed, create them and assign issues.

g) Start utilizing other principles of the guideline!