Requirements management in GitHub with a lean approach

Risto Salo, Timo Poranen, and Zheying Zhang

University of Tampere, School of Information Sciences, Tampere, Finland
risto.salo@gmail.com,{timo.t.poranen,zheying.zhang}@uta.fi

Abstract. GitHub is an online platform for hosting projects that use the Git revision control system to manage code. Its lightweight issue tracker helps to maintain lists of issues identified during the development process, including bugs, features, or other software artifacts. Although issue tracking software has been practically used in software industry, studies on using it to manage requirements remain insufficient. This paper tackles the issue by presenting a semi-formal guideline for managing requirements in agile software development projects using GitHub. The guideline is evaluated on a theoretical level by analyzing how well it guides to manage requirements and fits in an agile software development setting. It is compared against lean principles. In addition, the guideline is put into use in a case study. The studies indicate that the guideline and GitHub are well-suited for requirements management in an agile environment.

Keywords: Requirements management; GitHub; Lean software development; guideline

1 Introduction

GitHub [6] 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 10000 new users subscribe every weekday. GitHub is not a place just for individual users; its lightweight and easy to use features of managing source code and supporting collaboration with developers have attracted much attention, and been widely recognized by notable organizations, including Amazon, Twitter, Facebook, LinkedIn and even the White House. Although GitHub is mostly used for code, its issue tracker and wiki has been used for requesting and monitoring software features in ad hoc ways. How could these tools be used to manage requirements without a need to utilize another tool forms an interesting issue in software projects using GitHub. This paper tries to tackle it by introducing a guideline for requirements management (RM) using GitHubs native features. The guideline was put to use in a software development project and also evaluated based on the objectives of the requirements management and lean principles. This article is based on Salo's [13] thesis.

Lean ideology derives from Japanese car manufacturing from the middle of 1950s. In the beginning of the 21st century "being lean" has received a lot of hype in the software engineering field after Poppendiecks' [12] converted its principles to a suitable format for software development. The enthusiasm towards lean started to cumulate a little after agile methodologies hit through. Agile methodologies are a response to the failure of coping with changes in the waterfall-like methods, and their goals are aligned with lean principles. In this study lean principles offer an insight how well the guideline for RM shapes into an agile environment that does not 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 study novel. Although the problem domain and its solution are quite specific, the evaluation succeeds in combining lean principles to the objective of RM. The case study reveals that the whole project team values the guidelines 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 be used successfully for agile requirements management when a systematic guide is applied.

The rest of this paper is organized as follows. Next section gives an introduction to GitHub. Section 3 addresses common RM practices in agile projects. Section 4 describes the guideline and its usage. In Section 5 the guideline is evaluated. The last section concludes the work.

2 GitHub

The central concept in GitHub is a repository. GitHub's key aspect is its version control mechanism, so it is natural that other components are built upon this feature. The bulk of a repository's main page consists of the files and folders inside the version control system. There are navigation links to the subcomponents of the repository, and free-text search function to search multitude of things inside GitHub's repositories.

In GitHub all issues are created and tracked in an integrated issue tracker. An issue is a very vague concept; basically it is a task. However, issues can range from memos to requirements. In this study, the term issue is used as a higher level concept, and includes both requirements and tasks. A task is a concrete item that must be done in order to fulfill defined requirements. Requirements hold, in predefined format, the goals, business objectives and user requirements for the software. They are more abstract than tasks. Both can be divided to main and sublevel. Sublevels 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 subtasks for designing UI, creating it, designing architecture and implementation.

As shown in Figure 1, the issue tracker is composed of three components shown in three areas. The top of the screenshot is the issue tracker's inner navigation. Provided options are the default view, milestone view, labels, filters and

a link to create a new issue. Below this area are exclusive general filters. In the bottom are the issues. Name, assigned labels, identification number, creator and elapsed time from the creation are displayed. Every issue has a checkbox, for carrying out actions like assigning a label without a need to open the issue.

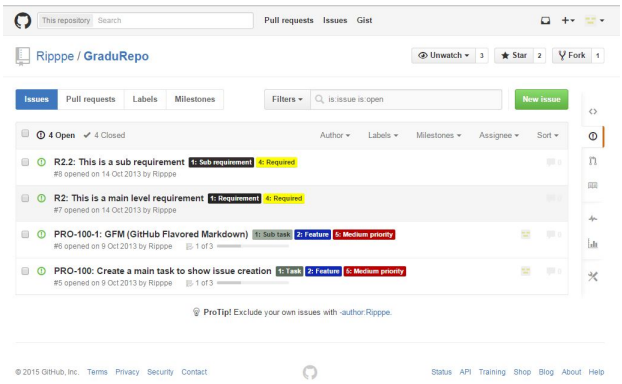

Fig. 1. The issue tracker view in GitHub.

The issue tracker provides a separate view for creating a new issue. User must give a name and a description of the new issue, but he can also assign a developer, desired milestone and labels. Labels are small tags that can be assigned to issues. Every label acts as a filter to the issues. Milestones are iterations and as such issues can be linked to them. An issue can always belong to a maximum of one milestone. User can observe and manage the milestones. User can toggle open and closed milestones and also milestones with or without any issues assigned to him. Every milestone can be edited, closed or deleted. Milestone's name, description and due date are also presented.

Filters refer to different filtering options available in the issue tracker. These filters can be divided to four categories: personal, milestone, label and state. Personal filters ("Assignee" and "Author" selections) are used to filter issues that are related with the current user. Milestone filter is for filtering issues associated to a specific milestone. The label filter limits the issues to those that have all the selected labels assigned to them. The state filter is for filtering open and closed issues. There is also a search box that provides a way to make more elaborated filter queries using a special syntax.

When an existing issue is opened, a detailed info view is displayed. All data in the view is editable. The state is notified below the name of the issue. Other relevant information includes subscription to the issue and participants of the issues. Subscribing to the issue means getting notifications about changes to the issue. The person creating the issue is automatically subscribed. Comments and references to the issue are visible below the description in a chronological order.

The last subcomponent of the repository is a 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.

3 Requirements management

Paetsch and others [11] have noted that the upfront documentation of requirements and the inflexible change management process in a traditional software development process and as such, is not compatible with agile practices. Agile practices emphasize the communication and collaboration with customers, the working software and responding to changes. The questions of how requirements are managed in an agile environment don't have unambiguous answers. As different agile methods embrace different aspects of agile principles, so do researchers and requirements analysts. However some common practices can be declared.

Stakeholder involvement forms one of the core principles in agile methods, and a key factor for project success [2, 4, 8, 11]. Though the traditional requirements engineering also embraces customer contribution, it is valued even more in an agile environment. Communication barriers often exist in the interaction between developers and stakeholders, due to different working experience, mind sets and background knowledge. Different techniques provide support for communication and collaboration with stakeholders. Besides the traditional ones such as interviews, face-to-face conversations, brainstormings, observation, etc., techniques such as prototypes, working software [2] or the tests and review meetings offer varying ways for stakeholders to understand their product and propose changing requirements. It also helps developers to gain an in-depth understanding of the application domain [8]. Such an intense collaboration in agile software development processes has been reported to lower the need for major changes in delivered products [2].

Replacing heavyweight documentation with a lightweight alternative forms another common practice. 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 [11]. Instead of a full requirements document, story cards or user stories form the most common form of user requirements [11, 17]. They are usually either physical or electronic representations of cards that include a few sentences describing how the user completes a meaningful task in interaction with the software to be built [3, 15]. The backlog or feature list can then be used to keep the track of stories and their progress [15]. The details of user stories are elaborated just in time, before they are about to be implemented.

As many agile methods take advantage of iterative software development process, the same practice is applied in RM [5, 10, 14]. Working on the increments based on user stories involves interaction with end-users, where, in fact, changing requirements come in the form of users' feedback. A key aspect of the iterative and incremental process mentioned by multiple studies [4, 8, 11] is prioritization. Requirements are ranked according to their priority, and the ranking can be adjusted often before the next implementation iteration [11]. The update and

adjustment is based on users' feedback on the latest increment, and to ensure that the most important and urgent requirements are tracked and implemented first. [8, 2]

To summarize aforementioned three factors, the key aspect for agile requirements engineering is to manage the requirements to implement the most important ones first in order to produce the best possible business value for the customer [5, 11].

As with agile methods in general, it should be remembered that agile requirements engineering does not guarantee success, although correctly used can greatly enhance chances for it. Cao and Ramesh [2] 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.

4 Managing requirements in GitHub - the guideline

4.1 A hierarchy between requirements and tasks

Tasks and requirements can be split to smaller parts. Smaller issues give a more transparent view into what is really wanted. The bigger the issue more likely it contains too wide a topic to fully grasp. This is why we recommend to use the subtasks and sub-requirements. For example the case study contained a requirement R2.2 "There are a total of 10-15 puzzles". A natural way of splitting R2.2 to tasks was to make a task for each puzzle. The tasks could be further split to subtasks representing implementing business logic, the UI and so on.

Splitting could be done endlessly so there must be a limit to how deep the hierarchy can go. For the guideline the maximum depth is four issues: a requirement, a sub-requirement, a task and a subtask. To revise the example, an alternative to creating just one subtask for a business logic would be to create multiple subtasks to depict individual components of the business logic. These subtasks would be direct descendants of the main task.

Subtasks have some drawbacks. Too small tasks cause more work with their creation and maintainability and offer minimal benefits. If developers are experienced, forcing them to create the subtasks might be a waste of time. On the other hand, in some cases it can help a developer further recognize the problem and its different areas. A good thing with subtasks is that the components they affect are easier to identify and trace. A big task could be clogged up with a long description and dozens of comments hindering the process of finding the relevant information.

There are benefits of not using the subtasks. 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 subtasks such information cannot be displayed due to the lack of automated hierarchy handling. The second benefit is negating the overhead caused by creating and maintaining the subtasks.

Overall it is impossible to say when the benefits of the subtasks overcome the slight waste caused by their upkeep. The decision about their use should be made case by case. The main point is that all the necessary information is present, logically structured and findable with minor effort. It should 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 subtasks to be just a nuisance and therefore the attitude towards the guideline might decrease causing a negligent usage. Overall it is hard to say when benefits from defining the subtasks overcome the slight waste caused by their upkeep. The decision about their use should be made case by case, although when information is separated to logical components, it is easier to find and interpret. The guideline emphasizes that the necessary information must be present, logically structured and findable with minor effort.

Issues should always follow a strictly specified hierarchy where the main level issues (requirements and tasks) must exist and the subissues (subtasks and -requirements) inherit from them. Every issue should always have only one parent, but the amount of children is not limited. Figure 2 depicts this hierarchy structure. To create this structure in GitHub issue references, labels and naming conventions are to be used.

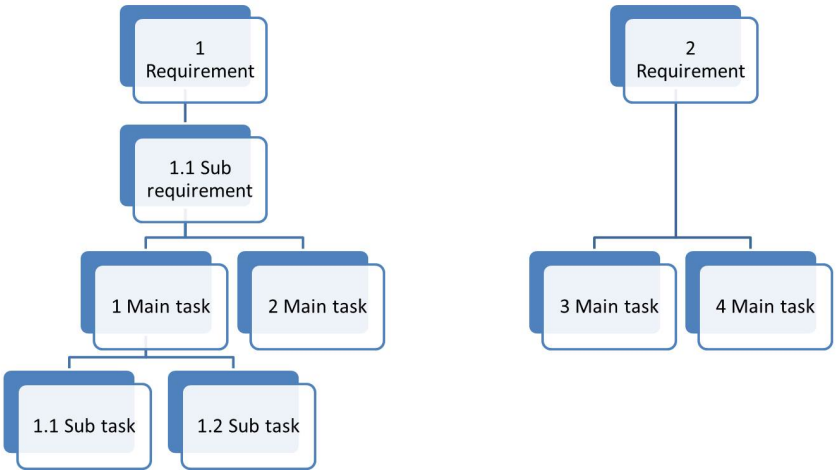

Fig. 2. Different hierarchy combinations for issues.

4.2 The steps of creating an issue

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 creating tasks should involve the whole team. This has many benefits. Seasoned developers may already have suitable

solutions or have such experience which is useful in some other way. Letting the developers 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 objectives is broadened within the team, thus building a stronger foundation for making better decisions.

As the guideline is directed towards agile environments, issues can be created iteratively. The best approach is dictated by the size of the project and its application 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 consists of seven steps. **The first step** is deciding the title of an issue. The title should include some kind of a reference prefix that is different for requirements and tasks to help identifying the issue and its place in the issue hierarchy.

The second step involves writing the description of the issue. This is a very crucial step and concerns the question of how long should the description be? If too much information is given, the issue becomes cumbersome and it might be hard to find the core information. Too vague descriptions tend to cause guessing, misleading or a need to consult somebody with better knowledge thus wasting time. 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 an individual thinking and implementation. This endorses the team and acts towards agilism which expects that experienced developers reach the best outcome without handholding.

The requirements on the other hand should be as unambiguous as possible to make sure that every stakeholder understands them the same way. The bottom line is that the issue should always have a description which is tangible. The guideline suggests that as much of the task specific information as possible should be directly in the description.

A large amount of pictures, documents or other relatively static material should be archived somewhere else to reduce the overhead of information. For requirements it might be a good practice to document them to the wiki. In such case a short description and a link are sufficient for a requirement issue.

The format of the issue's description is not predefined, but the guideline highly recommends that a logical structure is used throughout each issue taking advantage of GitHub Flavored Markdown (GFM). GFM is a special hypertext formatting syntax utilized by GitHub. It inherits from the Markdown [7] and is enhanced with additional convenient features, such as task lists and automated references.

The description shall comprise four parts: summary, information, task list and references. The summary explains the issue in few sentences. The information part consists of all the relevant information. If this data is stored somewhere else, links should be used to point out the source. The task list is utilized when a task involves steps, that are not split to own tasks yet need to be monitored. They should also be used as subtask replacements if the subtasks are not used.

Otherwise the format of the description should be kept as simple as possible. Developers should be aware of GFM's special features that are not included in

the basic Markdown. For example @-notation or "mentioning" can be used to notify a specific person or a team about the issue. Mentioning creates a filter into the issue tracker for the ones mentioned.

Because GitHub itself does not recognize any kind of hierarchical constraints between issues, such a behavior must be implemented manually. The way the guideline achieves this is by using labels, naming conventions for issues and issue references. An existing issue can be referenced from another issue establishing a link between them. In GitHub referencing an issue from another means that within the first issue's view, there is a hyperlink pointing to the second issue. The link is accompanied by a comment. The issue referenced must exist to get the linking to work. This can cause inconvenience when a main task is created before its subtasks. The guideline recommends that the 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 established. Otherwise the reference should always be aimed at the direct parent or child of the issue.

Steps three to five consist of assigning additional information to the issues: first labels, then people and lastly milestones. The labels are an essential part of the guideline and they are the main solution for creating visibility and status tracking. The guideline does not 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 and they may vary. There are a total of six categories suggested by the guideline: the type of the issue, the subtype of the task, status, requirement level, priority and miscellaneous. The guideline recommends that every category has its distinctive color with a unique shading for every single label. A short prefix in the beginning of each label 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, the miscellaneous labels being an exception but the important thing is that the categories exist and are used appropriately. After the labels comes assigning people (assignees) who 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. The exact way to use the milestones is up to the development team. The guideline suggests that they can be used to represent iterations or groups of tasks covering some wider topic or feature.

The sixth step is publishing the issue. **The seventh** instructs the creator of the issue to update the references of other issues. When an issue is created and references updated successfully, it starts its own lifecycle.

4.3 Status tracking and traceability of issues

Status tracking and tracing requirements are the core activities of RM. We have already created a base for them in the previous sections.

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

The monitoring of these aspects requires filters. Even though they can not cover every scenario they are quite powerful, especially since the filters are a part of the URL, so bookmarking the most used filter combinations is possible. The required filter combinations is determined by what kind of information is relevant to different team members. Developers should be aware of all the issues that mention them and the issues directly assigned to them. Especially the priority and milestone information is relevant. Project managers should pay attention to every task currently worked with. This includes following milestone deadlines and task list progression if the issues contain them. Also issues needing special actions - like ones flagged "blocked" - must be dealt with without a delay to keep the workflow going.

4.4 Combining version control and wiki to the issues

The wiki has already been mentioned 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 an easy access. Should the wiki be utilized, establishing a logic structure inside it is recommended. The structure itself does not concern the guideline as long as it is consistent and logical. A careful attention should be kept to avoid situations where the wiki's data contradicts issues'.

The repository offers some interesting interactions like referencing an issue from a commit message is possible. This is special feature of GitHub. Therefore to enhance visibility, developers are encouraged to always reference the issues their commit affects. This creates an automatic comment to the issue's comment section, making the link between the commit and the issue more visible and traceable.

4.5 Updating and maintaining issues

During the lifecycle of the issue, it is going to be updated in several ways: changing labels, assigning milestones, assigning people, commenting, referencing it from commit messages and closing it. Stakeholder communication regarding issues is a common update action. The challenge is that the communication can occur in multiple media. These interactions can cause changes to the issues' content. It is imperative that in such cases, the issue is immediately updated to reflect the changes: the descriptions of the issues must always be up to date. This guarantees that the latest and best knowledge is easy to find. It is encouraged to use the comment area of an issue for discussion.

A crucial note is that users should be careful when deleting old information. By doing so the traceability is compromised. However the descriptions should not contain unnecessary information. The old information should be moved from the

issue to a suitable page in the wiki. As important as it is to keep the description updated, is to make sure that the labels are used and updated. The importance of the labels is to visualize issues and different aspects of them in one view. Should the labels be misused or not updated, an unnecessary waste is generated. Letting the information get old causes mistrust towards the guideline and may lead for rejecting its practices. It also interferes with the RM and its objectives.

5 The evaluation of the guideline

The guideline was evaluated in a student project given in the School of Information Sciences at the University of Tampere in 2013. The project lasted nine months, and had four developers and three project managers from whom one had to drop out during the project. None of the team had earlier experience with GitHub's issue tracker, though two of them had used GitHub. The team was multicultural and used English for the communication.

The goal of the project was to produce a mobile puzzle game which introduces the concept of computer science to school applicants. Due to a requirement of an open source development GitHub was chosen as a version control platform. At the beginning of the case study the team was accustomed to the platform. They also received the first version of the guideline. Couple of weeks later the team received a shorter document summarizing the key aspects of the guideline. The team was responsible of deploying the guideline. We observed the usage from three perspectives: watching the team's actions in GitHub, surveying their internal communication and attending the meetings the team had with the customer. In the end of the project an online survey was filled by the team with different sets of questions for developers and managers.

Several important observations were made from the case study. The managers were little lazy to bring the requirements to the issue tracker and convert them to tasks. When this was finally conducted the project had already gone a good way. This meant that some of the traceability and status tracking were inherently lost. As the team consisted of mainly inexperienced students, they put more emphasize on the actual implementation than on the RM. This was also reflected on how the team perceived the whole issue tracker. One of the developers commented "I think the management of tasks and requirements is the role of the managers, – Managers should just fiddle with the requirements." The rush with the deadlines also meant that the team skipped or neglected some parts of the guideline. For example issue descriptions were incomplete, issue references were neglected and the label categories were used too liberally. However the whole team felt that they had followed the guideline and it in fact did help them to achieve a more coherent RM process: "I believe it was very efficient", "Overall, the guideline was useful and logical, I did not find any inherent flaws in it.".

The guideline was evolved from the first version published in October 2013 in a slideshow format. The purpose was to iteratively develop the guideline based on the feedback we received from the case study from October 2013 to May

2014. A few things were altered. The structure of the guideline was reformatted to make it more logical and readable. The discussion whether subtasks should be used was widened. We realized that there are valid situations where a main task coupled with a task list is enough. Therefore better arguments for and against of the subtasks were made, highlighting that the relevant information must be easily displayed whatever the chosen approach is. The first version leaned towards the expectation that there is a strict division between who creates and handles issues and the rest of the team. The guideline and RM are everyone's responsibility and everyone's contribution is valuable, so the guideline was changed to put more emphasis on the whole team. The guideline failed to note the powerfulness of the milestones. With added examples users should be able to identify additional purposes for them that could help monitoring issues. Maintaining the issues received a new note about preserving the old information, especially about disregarding it. Tracing requirements becomes hard should the initial situation get lost. A new recommendation was that the issues should be closed with an accompanying comment to clarify why the issue was closed. In the case study there were a lot of issues closed without an apparent reason. This uncertainty was also partially due to somewhat disorganized use of the labels. Therefore the up-to-date labels are even more highlighted. Creating the special issues like enhancement proposals and bug reports has been clarified to distinct how the guideline expects them to be used.

5.1 How to use the guideline

The purpose of the guideline was to offer a set of recommendations and practices for the RM in GitHub's environment. Individual sections from the guideline can be used as such but it is advisable to use it as a whole. Some room is left for customization to preserve the versatility. The guideline aims to complement the four areas of RM: change control, version control, requirements tracing and requirements status tracking [9, 11, 16] by utilizing mainly the issue tracker. As the issue tracker is lightweight to use, it is well suited for the agile RM. As argued before there does not exist as thorough a guideline for the RM in GitHub. There are blog posts covering the issue tracker but they are not systematic nor scientific, but rather experience reports. The lean software principles are used for assessing the guideline and its compatibility to the agile environment. These principles are abstract enough so that they don't restrict guideline's usefulness, yet still offer enough concreteness for assessment.

The guideline is a collection of practices and recommendations working together towards a consistent RM process. The guideline requires a knowledge of using GitHub, especially the repository, issue tracker and wiki. Using GFM is advised in the issues and wiki.

Before starting to work with issues, some preparations must be made. The labels, their categories and their color coding are to be decided. Overall naming conventions with the issues must be agreed on. Generally it is ideal to go through the guideline's practices and decide how they are applied into the project. Without a commitment from the whole team, the guideline loses its effectiveness.

The team should appoint somebody responsible for the RM. This does not exempt the team from the RM. The purpose of the appointment is to have a person who can attend questions regarding the issue tracker, guideline and issues.

The guideline does not take a stand when it comes to requirements engineering processes occurring before the RM. They can be conducted by whatever means necessary for the project's scope and type. As new requirements are identified they are gradually created into the issue tracker. The prefered way is that the requirements are immediately put into the issue tracker. Postponing this delays the implementation. As soon as the requirements are in the issue tracker the team can start splitting them into tasks. When the first tasks are ready the implementation can begin. The requirements are to be split in the order of priority to generate the best business value.

Creating requirements and tasks is usually an ongoing process and happens throughout the whole project. When the first tasks are under work, they must be monitored, maintained and updated.

5.2 Evaluating the guideline against the RM & lean principles

The following list will summarize the key aspects enhancing the objectives of the RM and the principles of the Lean as introduced by Poppendiecks' [12]. The first four cover the principle's of the RM, the rest focuses on the Lean.

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 special issue type the enhancement proposal gives a convenient way for formally proposing changes and tracking their life span: does the proposed change get implemented, is it further evaluated or even completely rejected. Of course, in an agile environment, changes don't always follow a formal path. The guideline enforces that information contained in the issues is always up to date.

Version control. The issue tracker itself holds all the requirements together. Every issue is given an unambiguous identification number by GitHub that can be used for referencing the issue. Versioning itself is not directly supported so the guideline suggests that the old information is not erased, but preserved somewhere else. The naming conventions and type labels are there for identifying purposes.

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 the descriptions are used and maintained properly. The aforementioned naming conventions also help identifying the relations between the issues.

Status tracking. The labels are the best way to accomplish the status tracking. The suggested categories are crafted so that tracking requirements and their tasks is as straightforward as possible. The drawback is that the labels

are mainly toggled with tasks, not with requirements so the status tracking of requirements must be carried out through tasks.

Eliminating waste. Descriptions with enough information and an encouragement for a free speech in comments aim to generate more knowledge for the whole team. This contributes towards learning how to produce value more efficiently. Avoiding unnecessary waiting sharing knowledge and using labels are suggested. The processes of the guideline are narrowed down and are inherently simple. They are also flexible, which makes them more suitable for various situations. Splitting requirements to small pieces increases the possibility that unrequired 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. Sharing knowledge and discussing it increases the insight to the customer's mind and thus makes it easier to comprehend the whole picture.

Building quality in. The customer's needs must be understood and openly collaborated and communicated. Information sharing, keeping everything visible and actively using commenting are ways the guideline proposes.

Learning constantly. Sharing and absorbing both information and knowledge is best way to fulfill this principle. The splitting of requirements in small pieces forces the team to truly get a better understanding of the domain.

Delivering fast & deferring commitment. Small tasks take less time to implement making the fast delivery possible. Feedback and communication should occur all the time further improving the delivery and its quality. Deferring commitment requires the best available information. Keeping issues up to date and openly discussing them enhances this.

Respecting people. The guideline aims for empowering the whole team. They are the ones to use the issue tracker, and they all should have an interest on what is happening there. The guideline does not restrict who can do what. How well the empowerment succeeds is based on whether the parent organization supports giving the decision power to the development teams.

6 Conclusions

The aim of this article was to present a semi-formal guideline for conducting the RM process using only the tools provided by GitHub. To prove that the guideline is applicable and working, a case study was conducted. The guideline was evaluated on how well it supports the RM objectives and the lean principles.

As far as the research shows, this is an unique topic: similar guidelines for GitHub platform with a properly surveyed case study could not be identified. There exist several blog posts about thoughts and suggestions of using the issue tracker, but none of them approached the topic as systematically. These blog posts also lacked the evaluation of their usefulness [1]. Our conclusions are based on the case study and observations, not just vague opinions.

For the science, the guideline is interesting because it combines the lean principles into the RM. This is also an area that is covered by very few studies.

The evaluation of the guideline gives more insight on how the RM can be coupled with the lean principles.

For the practitioners the guideline provides considerable, systematic and well documented instructions for using the issue tracker to handle the RM in a project that uses GitHub. The guideline is also evaluated by the case study, and though the study consisted of only a student project, its feedback was very positive. The case study project group felt that the guideline helped them achieve a more consistent RM process. By comparing the guideline against the lean principles we have showed that the guideline is well suited for the agile approach.

The guideline was created to be usable in a wide array of projects, though it is required that the project utilizes some agile approach. 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 for establishing a good base for the RM. This is especially beneficial for agile projects that tend to bury the RM below an iterative development. As GitHub could be considered a programmer-friendly platform, handling the RM in the same place can lower the threshold of developers to participate and take more active role in it.

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 concepts and leave out others. A selective use can greatly diminish the usability and results achieved by the guideline. The guideline leaves room for customization to preserve versatility but this means that instruction contain some vagueness.

There are some limitations concerning this article. 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 disposal. With GitHub we have a limited set of features but on 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.

The case study was relatively limited in scope. The project was a student project that lasted nine months. The problem with students is that they tend to lack the experience present in professional software projects. As the project was carried out during the university semester the students also had other course to attend to, leaving relatively small amount of time to work with the project. 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.

The guideline would greatly benefit from a larger case study involving projects with different settings and sizes. A wider case study would make it easier to further evaluate the core ideas of the guideline and how well they achieve their objectives. It could also highlight the situational contexts the guideline manages the best. Projects with experienced team members could give better facts to support different kind of approaches.

Overall the guideline achieved its goal: creating instructions for an agile requirements management process utilizing only GitHub's own tools. This is backed up by the case study and critical assertion of guideline's practices and GitHub's features.

References

1. Bicking, I. (2014). How We Use GitHub Issues To Organize a Project, `http://www.ianbicking.org/blog/2014/03/use-github-issues-to-organize-a-project.html`
2. Cao, L., & Ramesh, B. (2008). Agile Requirements Engineering Practices: An Empirical Study. IEEE Software, 25(1), 60–67.
3. Cockburn, A. (2006). Agile Software Development: The Cooperative Game (2nd edition.). Upper Saddle River, NJ: Addison-Wesley Professional.
4. Ernst, N. A., Murphy, G. C.: Case studies in just-in-time requirements analysis. IEEE Second International Workshop on Empirical Requirements Engineering, 25–32 (2012)
5. Favaro, J. (2002). Managing requirements for business value. IEEE Software, 19(2), 15–17.
6. GitHub - Web-based Git repository hosting service, `http://www.github.com`
7. Gruber, J. (2004). Markdown, `http://daringfireball.net/projects/markdown/`
8. Hofmann, H. F., & Lehner, F. (2001). Requirements engineering as a success factor in software projects. IEEE Software, 18(4), 58–66.
9. 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).
10. 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).
11. 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).
12. Poppendieck, M., & Poppendieck, T. (2003). Lean Software Development: An Agile Toolkit. Boston, Mass.: Addison-Wesley Professional.
13. Salo, R. (2014). A guideline for requirements management in GitHub with lean approach, University of Tampere, School of Information Sciences, master's thesis, 71 pages.
14. Sommerville, I. (2007). Software engineering. Harlow, England; New York: Addison-Wesley.
15. 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).
16. Wiegers, K. E. (2009). Software Requirements. Microsoft Press.
17. 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.