Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time

Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
vrijdag 11 maart 2016 om 10:00 uur

Door

Petra Marianne HECK

Master of Computer Science
Eindhoven University of Technology
geboren te Sittard.

This dissertation has been approved by the

promotor: Prof. dr. A. van Deursen and
copromotor: Dr. A.E. Zaidman

Composition of the doctoral committee:

Rector Magnificus	chairman
Prof. dr. A. van Deursen	Delft University of Technology, promotor
Dr. A.E. Zaidman	Delft University of Technology, copromotor

Independent members:

Prof. dr. ir. R. van Solingen	Delft University of Technology
Prof. dr. F.M. Brazier	Delft University of Technology
Prof. dr. R.J. Wieringa	University of Twente, The Netherlands
Prof. dr. T. Gorschek	Blekinge Institute of Technology, Sweden
Dr. N.A. Ernst	Software Engineering Institute, USA

This work has been supported by the Netherlands Organization for Scientific Research (NWO) through the RAAK-PRO program under grants of the EQuA-project. It has has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Published and distributed by: Petra Heck
E-mail: petraheck@outlook.com
ISBN: 978-94-6186-600-4

Cover: 'Sticky notes on the wall of the Wikimedia Foundation office' by Sage Ross.
Printed and bound in The Netherlands by CPI – KONINKLIJKE WÖHRMANN B.V.

Dedicated to my beloved father

Acknowledgments

Before diving into the details of Just-in-time Requirements I would like to take the time to thank a number of people for their contribution to this thesis and even more important their contribution to my life and the process which has led to me defending my PhD thesis.

This is the third time in my life that I had the opportunity to start a PhD. I was always more drawn by other challenges, but this time Fontys Applied University gave me the opportunity to work on a topic of my own choosing within an area that had my interest since a long time (Early Quality of Software). Furthermore, they allowed me to do so while keeping my job as a lecturer in software engineering, working three days a week as a researcher. I immediately knew this was a unique opportunity for me. That is why I would like to thank the director of Fontys ICT, Ad Vissers, and my team lead, Sander van Laar, for allowing me to take this opportunity even though I only had been employed by Fontys 6 months before. I would also like to thank Jacob Brunekreef who was the project manager of the EQuA-project that I was part of during the first two years of my PhD. Jacob was always there for me with support and good advice.

The next group of people that I would like to thank is of course the team from the Software Engineering Research Group in Delft. First of all Arie van Deursen who gave me the opportunity to work with his group on a topic (Requirements) that is not so much in their core business. But most of all Andy Zaidman. Without your guidance I do not know if I would have made it in such a short period of time. You helped me to set my topic, to frame each of my papers and to work towards this thesis in a structured way. Your experience on setting up experiments/studies, writing papers and rebuttals has been invaluable to me. Next to that I greatly appreciate the time you took for working with me and answering all my questions. I am really going to miss our weekly virtual coffees. A final word of thanks to the rest of the group with whom I occasionally had the pleasure of sharing lunch and more specifically to Cuiting Chen who provided me with the template of her thesis and (her friends) cover design.

I did my day-to-day research behind my desk at Fontys Eindhoven. The long writing sessions were definitely more enjoyable through the continuous interruptions of my roommates (all the three different offices I was in :-) and the relaxing lunch and coffee breaks I spent with my colleagues. A special thanks to my col-

league Gerard Schouten who accepted to be the photographer during my defense ceremony.

Last but not least I would like to thank my family and friends. Their involvement in my PhD research might not have been very direct, but they have made me to the person that I am today. They have supported me in everything I have done and gave me the confidence to take up this challenge. Next to that, they offered me the invaluable moments of joy and laughter that inspired me to work even harder in the rest of the time. Thanks especially to my friends Judith and Kaate for agreeing to be my paranymphs; you are my dear friends, each symbolizing a different period of education (high-school and university) in my life. A special mention for my parents that have always supported me (both mentally and financially :-). It is sad that my dad is not able to see the end result of all of this but I know he would have been proud of me.

Thank you is not enough for my husband, Mark, who has always inspired me to take on new adventures and who is my best friend in life. You always listen to my stories, give me advice and support me in so many ways. I love you and I am really happy that we got the opportunity to start a family together with our lovely daughter Tessa.

Petra van den Broek - Heck
Eindhoven, March 2016

Contents

List of Tables

List of Figures

List of Acronyms

***C**	Creation time
***J**	Just-in-time
BRN	Bug report network
FRequAT	Feature request analysis tool
INVEST	Independent, negotiable, valuable, estimable, small, testable
IQ	Inherent quality
JIT	Just-in-time
LSA	Latent semantic analysis
OSS	Open source software
QC	Quality criterion
RE	Requirements engineering
RQ	Research question
SLR	Systematic literature review
SMART	1) Specific, measurable, acceptable, realistic, time-bounded; 2) System for the mechanical analysis and retrieval of text
SPCM	Software product certification model
SVD	Singular value decomposition
TF-IDF	Term frequency - Inverse document frequency
VSM	Vector space model

1 Introduction

Quality of requirements is considered important since the early days of software development. Already in the early eighties Boehm (1981) showed that finding and fixing errors in the requirements phase can be up to a thousand times cheaper than fixing them after delivery of the software. Since then, several guides (e.g. IIBA (2009)), standards (e.g. IEEE-830 (1998)) and even certifications (e.g. `www.ireb.org`) have been created that describe how good requirements engineering should be performed and what constitutes quality of requirements. For example, the IEEE 830 standard (for Software Requirements Specifications), states that requirements should be complete, unambiguous, specific, time-bounded, consistent, etc. According to Denger and Olsson (2005) it is important that each project or company defines their own minimal and optimal set of quality criteria for the requirements.

Since the beginning of this century a new way of developing software has become more and more popular. The principles of this so-called 'agile development' have been captured in the Agile Manifesto (Beck et al. 2001). The main implications for requirements engineering are: 1) software is being developed in short iterations with the requirements being detailed only just before the start of each iteration - Ernst and Murphy (2012) call this 'Just-in-Time Requirements'; 2) early feedback through heavy involvement of the customer and the use of prototyping leads to less written requirements details - we would call this 'Just-Enough Requirements'. The latter not being a widely used term, in this thesis we will use the more common term 'Just-in-Time (JIT) Requirements'.

Just-in-Time Requirements Analysis

Just-in-Time Requirements Analysis is a lightweight, adaptable approach to requirements analysis. It is an analysis process that expects and embraces change and is distinguished from other analysis methodologies in several ways (Lee 2002):

- Requirements aren't analyzed or defined until they are needed.
- Only a small initial investment is required at the start.
- Development is allowed to begin with incomplete requirements.
- Analysis and requirements definition is continuous throughout the project.
- Requirements are continuously refined as the project moves forward.
- Change is expected and easy to incorporate into requirements.

In this thesis we use the term **Just-in-Time (JIT) Requirement** to refer to any requirement created and managed through such an approach.

Not all quality criteria from the IEEE 830 standard (which has been designed for up-front requirements documents) are applicable to JIT requirements. JIT requirements are allowed to be incomplete or vague for at least a certain period of time. However, there is a limit to this vagueness. The requirement should be clear enough for the stakeholders to remember what this requirement was about at the start of the designated iteration. Otherwise the development team would need to spend extra time or effort in detailing it. In the same way it is good to have a high-level overview of the scope of the application before the first iteration starts. E.g. knowing that there is a reporting module will guide the development of other features and the basic architecture, but it might not be necessary to know from the start the contents of all reports in the module.

The goal of this thesis is to *obtain a deeper understanding of the notion of quality for just-in-time requirements*. In this thesis we focus on quality in the sense of informal verification: "ensuring that requirements specifications and models can be used effectively to guide further work" (IIBA 2009) without the use of formal methods. Verification activities as in this definition ensure that the requirements are specified in a correct way. This is opposed to requirements validation that "ensures that all requirements support the delivery of value to the business, fulfill its goals and objectives, and meet a stakeholder need.'" (IIBA 2009). According to IIBA (2009) verified requirements is a pre-requisite for validated requirements. That is why we will focus on this first step in this thesis. We will focus on informal verification as opposed to formal verification because the use of formal methods to prove requirements correctness is not common in agile development.

In this thesis we use open source feature requests as our main study object. They are a form of just-in-time requirements, with the advantage of having their whole history available on-line. Where applicable we generalize our results to other types of just-in-time requirements.

1.1 Background on Just-in-Time Requirements Engineering

According to Cockburn (2000), "the better the internal communications between usage experts and developers, the lower the cost of omitting parts of the use case template" (the type of requirements Cockburn discusses). This is exactly what JIT requirements engineering is about. The initial JIT requirements are "a promise for a conversation between a requirements expert and a developer" (Cockburn 2000). The conversation about the details of the requirements is postponed until as late as possible. This means that the full specification of a JIT requirement is only done just-in-time. What just-in-time specification means (i.e. how much is written down and when things are written down) depends on the way of working for the development team or project.

This section provides a brief background on both agile and open source requirements as they are the type of JIT requirements mostly discussed in this thesis.

Agile Requirements Engineering

Four of the 12 principles behind the Agile Manifesto (Beck et al. 2001) directly relate to requirements:

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Business people and developers must work together daily throughout the project.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- The best architectures, requirements, and designs emerge from self-organizing teams.

Although working together and face-to-face conversation are valued highly in the Agile Manifesto, many situations exist (e.g. distributed development, large development teams, complex projects) that do require documented agile requirements (Inayat et al. 2014). In the remainder of this thesis we will focus on the *written* requirements. Similarly we focus on the end product (the written requirement) and not so much on the process with which this end product was created (e.g. self-organizing teams).

Two popular methods of agile development (Matharu et al. 2015) are eXtreme Programming, or XP (Beck et al. 2001), and Scrum (Schwaber and Beedle (2001) and `www.scrumguides.org`). Of those two methods, Scrum is currently most widespread (see e.g. `stateofagile.versionone.com`. According to Leffingwell (2011) Scrum is "achieving widespread use because it is a lightweight framework, and–more

importantly–it works. It also has the added benefit of a training certification process...". Leffingwell then goes on to describe what he calls the "Scaled Agile Delivery Model": a model he uses to describe the big picture of agile requirements throughout the enterprise. This model is loosely based on the main principles of Scrum, but then generalized to the entire enterprise, not just the development team. In this thesis we will sometimes encounter XP as many older publications about agile investigate this method. To explain some basic concepts of agile requirements engineering we will use the model of Leffingwell (based on Scrum).

User Stories

According to Leffingwell, agile development teams work with 'User Stories' (Cohn 2004) to capture the needs of the user. A user story specifies an intent, not a detailed requirement. In its simplest form a user story consists of one sentence: 'As a <role>, I can <activity> so that <business value>'. However, in practice there are many situations where this one sentence needs more detailing. This can be done by adding comments or attachments to the user story. The detailing of user stories is done by continuous discussion with the (representative of) the customer (called 'product owner'). Each user story has a priority (determined by the product owner) and preferably also some acceptance criteria, specifying when the story will be satisfied. Some use more extensive formats, such as Dan North's approach (North 2006) which also provides acceptance scenarios by specifying them following a strict template: 'GIVEN ... WHEN ... THEN ...'.

Iteration and Backlog

An agile development process consists of iterations (called 'sprints' in Scrum), typically short - about two weeks. All user stories are placed in a so-called 'backlog': a task list with yet-to-be-implemented user stories. At the start of each iteration the product owner decides (with the advice of the development team) which user stories go into the next iteration by re-prioritizing them. The product owner is also allowed to come up with new stories or change existing stories. This is why detailing of a story is usually postponed until the iteration the story has been assigned to. Detailing of the user story right before the start of the designated iteration is necessary to estimate the effort needed to implement the story. These effort estimations are then again used to determine how many stories can be included in the iteration that is about to start.

Definition of Done and Definition of Ready

Most Scrum teams use a so-called 'Scrum board' to manage user stories. A scrum board is a physical task board where each story is written on a card. The cards can then be moved around between columns specifying the progress of the story (e.g. 'To Do', 'In progress', 'Testing', 'Done'). There are usually strict rules about when a story is allowed to move to the 'Done'-column. This set of rules is called the 'Definition of Done' (DoD). In many cases the DoD requires that all regression

tests have been successful. In the light of requirements engineering some teams also define a 'Definition of Ready' (DoR). This is a set of rules specifying when a user story is ready to be included in the iteration (Power 2014).

Tools

The Scrum board mentioned in the previous paragraph is one of the important tools for Scrum teams. However, this only works when the whole team is working in the same physical location and the product owner can also come in to discuss with the team. Therefore, most teams (also) use some kind of electronic tool to manage stories. There are electronic implementations of the Scrum board like Trello (`trello.com`) and VersionOne (`versionone.com`), but also e.g. Microsoft Excel is used to manage basic lists of stories. Traditional requirements engineering tools like IBM Rational Doors (`ibm.com/software/products/ratidoor`) or Jama (`jamasoftware.com`) and issue trackers like Jira (`jira.com`) and Mantis (`mantisbt.org`) often have plug-ins to enter stories directly in the tool or integrations with other tools to manage agile requirements.

Open Source Requirements Engineering

We described in the previous paragraphs how agile development copes with changes in the environment: new or changed requirements are allowed to enter the development process at any time; those requirements will be prioritized and assigned to one of the coming development iterations. As long as a requirement is not being developed in the current iteration, not much time is spent on detailing it, because there is a risk that the requirement will be changed or become obsolete in the mean time.

A similar way of developing software and dealing with requirements can be seen in open source projects (Scacchi 2009). Chapter 2 of this thesis shows that most open source projects (e.g. Netbeans, Eclipse, ArgoUML, Mono, and Android) use some sort of issue tracker to let users request new or changed features. Others can then comment on the feature request in that same issue tracker. The development team will prioritize those so-called 'feature requests' and decide which ones will be included in the next release of the software. Only when the feature request is being considered for development, more details will be asked from the author of the request to allow for proper implementation. Because open source projects usually have users and developers all over the world, the complete discussion between author, users and developers can be read on-line in the issue tracker.

Feature Request

In this thesis we will use the term 'Feature Request' (FR) to refer to a structured request "documenting an adaptive maintenance task whose resolving patch(es) implement(s) new functionality" (Herzig et al. 2012). Structured request means a request with a title, a description and a number of attributes. Some projects might

call FRs differently, e.g. 'change request', 'enhancement' or 'task'. Feature requests can be encountered in both open source and closed source environments (Alspaugh and Scacchi 2013).

1.2 Current State of the Research Field

IEEE has not published any update of their quality criteria since the rise of agile development. There are new publications about agile from practitioner organizations e.g. BABOK (Stapleton 2013) and IREB (Grau et al. 2014) but they are more focused on the process of agile requirements engineering and do not mention the specific quality criteria for JIT requirements. In the remainder of this section we will overview the research field.

For just-in-time (mostly from the area of agile development processes) and open source projects there is a body of work both on Requirements Engineering (Grau et al. 2014; Noll and Liu 2010; Paetsch et al. 2003) and Quality Assurance (Aberdour 2007; Huo et al. 2004). However, as Chapter 5 (a systematic literature review) will show, there is not much literature on the combination of both. In this section we will mention a few papers that do address quality of JIT requirements specifically. This mainly serves to illustrate that the current state of the research field is (was) quite immature and incoherent.

Size of Agile Requirements

Desharnais et al. (2011) and Dumas-Monette and Trudel (2014) focus on size measurement of agile requirements (using COSMIC). They claim that quality of those requirements is a necessary condition for accurate estimation. Quality for them mainly means completeness: descriptions of functional processes and data modeling artifacts are needed to base the COSMIC size measurements on. Dumas-Monette and Trudel (2014) describe a case study that also highlights other quality issues related to size measurement.

Quality Attributes of Stories in XP

Duncan (2001) analyses the quality attributes for requirements in Extreme Programming (XP, one of the agile methods). He does this by comparing *stories* (the main requirements artefact in XP) to the quality attributes presented by Davis et al. (1993). In this thesis we consider JIT requirements in general (not only XP) and construct our own list of quality attributes from different sources. In Chapter 4 we use the list of Davis et al. to validate our list of quality attributes.

Patel and Ramachandran (2009) focus on story cards (used for denoting stories) and promote a standard structure for such a card. Next to that standard structure they describe a list of guidelines for stories. This product-related guidelines of that list are also covered by our framework in Chapter 4.

Quality Attributes of User Stories

Lucassen et al. (2015) describe quality criteria for user stories. They have used our framework in Chapter 4 as a basis for their own framework. The difference is that they only consider the user story title (no comments or attachments), where we consider the complete JIT requirement specification with the full history in comments and attachments. They use Natural Language Processing techniques to identify the different parts of the user story title. This is a type of analysis that we did not apply in our feature requests dataset, but which might be interesting for future work.

Conclusion

All in all none of the above papers are wide-spread (cited by at most 25 other papers according to Google Scholar). This might have to do with the fact that two of them focus on XP, which nowadays less companies are applying (see e.g. `stateofagile.versionone.com`). The papers about COSMIC are very specific, focusing more on size measurement. The last paper is very recent and we do not know yet how this will evolve. However, it is based on our own framework and appeared only after our own work in Chapter 2 till Chapter 4. So this last paper was not part of the research field at the time that we started our own research.

1.3 Problem Statement

The limited body of related work (see previous Section) is what inspired us to this thesis.

To address our main goal (*to obtain a deeper understanding of the notion of quality for just-in-time requirements*), we started out with an initial exploration of open source projects to analyze the feature requests and the difficulties we could see in working with those feature requests (Chapter 2: ***[RQ2]***[1] *how can we assist the users as the main actors in the requirements evolution process in open source projects?*). This led us to three subsidiary questions:

> ***RQ2.1****: In what form do feature requests evolve in the open software community web sites?*
>
> ***RQ2.2****: Which difficulties can we observe for a user that wants to request some new functionality and needs to analyze if that functionality already exists or has been requested by somebody else before? Can we explain those difficulties?*
>
> ***RQ2.3****: Do we see improvements to overcome those difficulties?*

[1] Note that we number our RQs according to the chapter numbers of this thesis, hence numbering starts at 2

The analysis of feature requests in open source projects pointed us to the large amount of duplicates in those feature requests, which we investigated further. We argued that in order to avoid duplicate feature requests it might help if users have a way to visualize related feature requests when submitting new ones. For the visualization of such a feature request network it is necessary to first determine links between feature requests. Some links are included in the feature requests themselves, but our next research question was if we could use text-based similarity to detect additional horizontal traceability links for feature requests (Chapter 3: ***[RQ3]*** *Can TF-IDF help to detect horizontal traceability links for feature requests?*). We set out with a simple Vector Space Model (VSM) with TF-IDF as a weighting factor, to answer two subsidiary questions:

> ***RQ3.1****: Is TF-IDF able to detect functionally related feature requests that are not already explicitly linked?*
>
> ***RQ3.2****: What is the optimal pre-processing to apply TF-IDF focusing on feature requests?*

After that we checked if we could improve the results for [RQ3.1] when applying Latent Semantic Analysis (LSA), which led us to our final subsidiary research question:

> ***RQ3.3****: Does a more advanced technique like LSA improve the detection of non-explicit links?*

After this analysis we moved away from the feature request networks to answer the more general question of what would be applicable quality criteria for feature requests specifically and JIT requirements in general (Chapter 4: ***[RQ4]*** *Which criteria should be used for the quality assessment of just-in-time requirements?*). We developed a framework for informal verification of JIT requirements, based on the case of open source feature requests, following along the two subsidiary questions:

> ***RQ4.1****: Which quality criteria for informal verification apply to feature requests?*
>
> ***RQ4.2****: How do practitioners value our list of quality criteria with respect to usability, completeness and relevance for the quality assessment of feature requests?*

After that we applied our framework to existing open source projects to answer a third subsidiary question:

> ***RQ4.3****: What is the level of quality for feature requests in existing open source projects as measured by our framework?*

One way to confirm that our framework is not missing any important categories or quality criteria is by comparing our framework to literature on quality criteria for agile requirements. For this purpose we conducted a Systematic Literature Review (SLR) on quality criteria for agile requirements specifications (Chapter 5). This SLR answers the following research question:

> ***RQ5**: Which are the known quality criteria for agile requirements specifications?*

Note that this question is similar to RQ4.1. For RQ5 we did not focus on feature requests but on agile requirements.

1.4 Research Methodology

The main goal for this thesis is quite general: to obtain a deeper understanding of the notion of quality for just-in-time requirements. Such a broad goal calls for the use of different research methods: a so-called mixed methods design. The idea of mixed methods design is that by combining methods, the weakness of one method may be compensated by the strength of another method (Creswell 2013).

This thesis presents two different lines of research: 1) duplication and horizontal traceability of feature requests; 2) a quality framework for JIT requirements. Creswell mentions three broad strategies of which, for both lines of research, we followed the "exploratory sequential mixed methods": we started with a qualitative study to explore the area (Chapter 2 and the first half of Chapter 4 respectively) and then used the information to define a qualitative study (Chapter 3 and the second half of Chapter 4 respectively). For each of the chapters we applied methodological triangulation to have multiple sources to answer our research questions from. Chapter 5, the SLR, was an extension we added later in the process to strengthen our qualitative findings in Chapter 4. For each of our studies we followed the guidelines of Wohlin et al. (2000).

Throughout the chapters of this thesis different research methods have been used to answer the research questions. Each chapter extensively describes the research setup. We summarize our research approach in this section to give an overview of methods used in one single place and to show that we have used different types of methods. See also Table 1.1 for an overview of sources used. For each chapter we will discuss the limitations of the chosen methods below.

Chapter 2

To answer **RQ2.1 till 2.3** we performed an *exploratory case study* with 20 open source projects (see Table 2.1 for the list) to discover what 'open source requirements' are (RQ2.1). We use the term Feature Request (FR) as a common denominator for the different terms that are used in the different open source projects for the 'requirements'. We encountered many duplicate requests in those 20 projects

(RQ2.2). We performed a *small quantitative analysis* to plot the percentage of duplicate requests in a table. We subsequently performed a *qualitative analysis* on the duplicate requests of one single project to get a feeling for the reasons behind those duplicates. The categorization of duplications that we found, was double-checked and improved on a sample of duplicate requests of 5 other projects (30 duplicates from each project). Our work in this case study allowed us to come up with recommendations for users and developers of issue tracking tools (RQ2.3).

Limitations

We selected the sample of 20 open source projects based on our knowledge of active projects or platforms with widely-used software products. We deem this selection sufficiently heterogeneous to base our conclusions on. We can however not exclude that a different or larger selection of projects leads to different findings.

For the quantitative analysis on the percentage of duplicates we rely on the feature requests that have been marked as 'DUPLICATE' by the project members. Our qualitative analysis of 6 projects shows that on average 15% are not real duplicates. However, we can also assume that project members oversee some duplicates that remain unmarked. Nonetheless we must take the numbers in our quantitative analysis as indications, not as truth values.

The manual analysis of duplicates in 6 projects was done by the first author and thus is subject to researcher bias. Although part of the categorization is based on objective observations (e.g. "the duplicate is by the same author as the original request"), we cannot exclude that different researchers come up with a different categorization of duplicates.

Furthermore, all recommendations have been derived from the findings by the first author. This being an exploratory case study, we did not check our recommendations with members of the open source community. We could for instance have set up a survey or focus group to collect opinions on our recommendations. Such a survey might have led to a better view on the feasibility of implementing each of those recommendations.

Another option to collect feedback on the recommendations would be to implement some recommendations in (a prototype of) an issue tracker and have users experiment with it. This was our initial idea for future work, but after the experiment with horizontal traceability (RQ3) we decided to turn our attention towards quality criteria for feature requests (RQ4 and RQ5).

Chapter 3

To answer **RQ3.1 till 3.3** we needed a dataset with feature requests. We decided to use three of the projects from our previous exploratory case study for this *experiment*, see Table 1.1. We designed a tool to exactly measure the so-called 'recall rate' of different preprocessing steps of the feature requests (RQ3.2). Having found the best preprocessing configuration we applied it to the dataset for each of the three

projects to detect related feature requests with textual analysis (RQ3.1). We triangulated our findings by the presence of links in the feature request itself or by relatedness as indicated by the lead developers of two of the three projects. Next to that we applied our results to an existing feature request network to validate the outcome of our algorithm. We repeated this analysis with a more advanced algorithm and saw that the results of the advanced algorithm did not outweigh its longer processing time (RQ3.3).

Limitations

Our goal in this experiment was to show that it is possible to detect links between feature requests by textual analysis. We conducted an experiment with 3 projects which in our opinion provides an indication that this approach could work. A follow-up experiment would need a golden set of feature requests for which we know each of the links on beforehand. Which such a golden set we can better support our claim that we are able to retrieve traceability links with textual analysis. We mitigated the absence of such a golden set by using the information in the issue tracker and by verifying results with open source developers.

By choosing textual analysis as the technique, we aim to identify related feature requests by the overlap in their word bags. This implies that similar documents that have very few words in common are not retrieved. In our current experiment we only verify if the links we find are valid, we do not validate if we retrieved *all* links. To investigate techniques that do not rely on overlap in word bags we would need to set up an experiment with the aforementioned golden set of related feature requests and find ways to retrieve all links.

We tried to find closed source feature requests in industry that we could use for subsequent analysis to see if our results also hold in other cases, but the companies we contacted all had reservations to hand over their (confidential) feature requests. This absence of industrial cases, which also makes it difficult to validate graphical tools for visualizing feature request networks (which we initially thought to be our research goal), made us turn our attention towards the more general question of quality of feature requests (RQ4 and RQ5).

Chapter 4

To answer **RQ4.1** we used existing literature and our experience from previous work. This resulted in a framework for quality criteria for feature requests. To get a feeling for the applicability of the framework (**RQ4.2**) we conducted *interviews* with eight practitioners from the Dutch Agile Community. Finally we performed a *case study* where we applied the framework to 620 feature requests (**RQ4.3**) from the same three open source projects used in Chapter 3. The case study was done by letting 85 final-year software engineering students (Fontys Applied University) each score 20 feature requests, randomly selected out of the total set of 620 fea-

ture requests. We purposely let each feature request be scored by 2 or 3 students to also have a view on the subjectivity of the scorings (and to detect 'unwilling' participants). This resulted in 1699 scorings of the 620 feature requests, which we summarized in a table to show the general level of quality for the feature requests of those 3 projects according to our framework. The students also provided us with some qualitative feedback on the case study and the framework.

Limitations

We qualified the interview and the case study with the students as an evaluation, because they give us an initial feeling for the feasibility of the framework. The interview was limited in setup to not take too much time from the 8 participants (maximum 2 hours). The case study was executed by students on open source projects. To further validate the framework we would need to also apply it in an industry setting and formulate proper (quantitative) hypotheses about the value of using our framework.

With 2 or 3 students scoring each feature request we do not have the quantitative data to calculate a true inter-rater agreement. Instead, for each criterion, we use the percentage of feature requests for which 2 or more students give the same score, to indicate the subjectivity of each of the 16 criteria they had to score. It would be good to confirm these "subjectivity scores" in a more detailed experiment where each feature request is rated by more than 2 persons.

We also did not go back to the open source projects to verify if the scorings indeed give a good impression of the feature request quality. The goal of the scoring done by the students was an initial evaluation of the application of our framework. As such we must consider the exact scorings of the three open source projects more as anecdotal evidence than as absolute truth.

Chapter 5

To answer **RQ5** we conducted a *systematic literature review (SLR)*. For this we followed the guidelines of Kitchenham and Charters (2007). We use our framework from RQ4.1 to present an overview of quality criteria found in literature. This is a second way of confirming the viability of our framework. At the same time we include the list of quality criteria from RQ4.1 in the answer to RQ5 (as we included our own publication in the SLR). This is a second way of confirming that our framework instantiation for feature requests (from RQ4.1) is not missing important quality criteria.

Limitations

The main limitation of this SLR is that we might have missed relevant literature because of the choices we made in each of the steps of the process we followed. This is a risk inherent to each literature review which we carefully describe in Chapter 5.

Table 1.1: Overview of sources used for Chapters 2 till 4 (FR=Feature Request), for links to projects see Table 2.1

	Chapter 2	Chapter 3	Chapter 4
ArgoUML		1273 FR + Check with 2 devs.	210 FR
Mylyn Tasks		425 FR + Check with 1 dev.	100 FR
Netbeans	30 duplicate FR	4200 FR	310 FR
Apache HTTPD	28 duplicate FR		
Subversion	30 duplicate FR		
Firefox	30 duplicate FR		
Eclipse JDT	30 duplicate FR		
Novell Mono	30 duplicate FR		
Dutch agile community			Interview 8 practitioners
Last-year students software engineering			1699 scorings of 620 FR (from 85 students)

1.5 Contributions

The contributions of this thesis are as follows:

A categorization of duplicate feature requests in issue trackers in open source projects (Chapter 2). We perform an analysis of duplicate feature requests in six open source projects that use an issue tracker to manage the feature requests. This has led to a categorization of duplicate feature requests and related to that a set of recommendations for better use of issue trackers and for improvement of future issue tracking systems.

A method to identify horizontal traceability links for feature requests (Chapter 3). We have shown that a Vector Space Model with TF-IDF can be used to detect horizontal traceability links for feature requests. We have analyzed the best pre-processing options for the feature requests. We have also shown that Latent Semantic Analysis does not provide better results in our case study.

A framework for quality criteria for just-in-time requirements (Chapter 4). We have designed a framework to present quality criteria for different types of just-in-time requirements. We have instantiated this framework for feature requests in open source systems and we have evaluated the framework with practitioners.

A quality score for requirements in open source projects (Chapter 4). We have applied the aforementioned framework to three open source projects to score the quality of feature requests. We have translated low quality scores into concrete recommendations for practitioners to improve the quality of feature requests.

An overview of literature on quality of agile requirements specifications (Chapter 5). We have conducted a systematic literature review and classified the resulting papers along different axes. The main result is a list of quality criteria for agile requirements specifications.

Recommendations for practitioners working on quality of agile require-

ments (Chapter 5). The aforementioned systematic literature review also allowed us to collect a list of recommendations from the different publications and our own analysis of those publications for practitioners that need to perform a quality assessment of agile requirements specifications.

A research agenda on quality of agile requirements (Chapter 5). The aforementioned systematic literature review also allowed us to collect a research agenda from the different publications and our own analysis of those publications.

1.6 Thesis Outline

The outline of this thesis is as follows. Chapter 2 describes open source feature requests and the problem of duplication. Chapter 3 investigates text-based similary as a possible solution for detecting horizontal traceability links between feature requests. Chapter 4 introduces a framework for quality criteria for JIT requirements and a case study applying it to open source feature requests. Chapter 5 presents a systematic literature review on the quality criteria for agile requirements specifications. Chapter 6 concludes this thesis by revisiting the research questions and outlining future work.

1.7 Origin of Chapters

Each of the chapters in this thesis has been published as or is submitted for a peer-reviewed publication. Note that we have included each publication as is. This results in some duplications in the texts of the different chapters.

The following list gives an overview of the publications, all of which have been co-authored with Andy Zaidman. In each of the chapters, "first author" refers to Petra Heck and "second author" refers to Andy Zaidman.

Chapter 2 has been published in the proceedings of the *13^{th} International Workshop on Principles of Software Evolution (IWPSE'13)* under the title: *An Analysis of Requirements Evolution in Open Source Projects: Recommendations for Issue Trackers* (Heck and Zaidman 2013).

Chapter 3 has been published as *Horizontal Traceability for Just-in-Time Requirements: The Case for Open Source Feature Requests* in the *Journal of Software: Evolution and Process* (Heck and Zaidman 2014b).

Chapter 4 contains our work entitled *A Framework for Quality Assessment of Just-in-Time Requirements. The Case of Open Source Feature Requests* as it has been submitted to the *Requirements Engineering Journal* (Heck and Zaidman 2015a).

Chapter 5 comprises our findings submitted to the *Software Quality Journal* as *A Systematic Literature Review on Quality Criteria for Agile Requirements Specifications* (Heck and Zaidman 2016).

Additional Publications

The author of this thesis has also been first author of the following publications which are not directly included in this thesis:

- *The LaQuSo Software Product Certification Model*, which has been published in the *Software Quality Journal*. This paper is referenced as (Heck et al. 2010).
- *Quality Criteria for Just-in-Time Requirements. Just Enough, Just-in-Time?*, which has been published in the proceedings of the *1st International Workshop on Just-in-Time Requirements Engineering (JITRE'15)*. This paper is referenced as (Heck and Zaidman 2015b).

2

Just-in-Time Requirements in Open Source Projects: Feature Requests

While requirements for open source projects originate from a variety of sources like e.g. mailing lists or blogs, typically, they eventually end up as feature requests in an issue tracking system. When analyzing how these issue trackers are used for requirements evolution, we witnessed a high percentage of duplicates in a number of high-profile projects. Further investigation of six open source projects and their users led us to a number of important observations and a categorization of the root causes of these duplicates. Based on this, we propose a set of improvements for future issue tracking systems.[1]

Software evolution is an inevitable activity, as useful and successful software stimulates users to request new and improved features (Zaidman et al. 2010). This process of continuous change of requirements is termed requirements evolution (Li et al. 2012). Requirements evolution management has become an important topic in both requirements engineering (Li et al. 2012) and software evolution research (Ernst et al. 2009).

Both industrial experience reports (Scacchi 2001) and academic research have identified a significant set of software projects for which traditional notions of requirements engineering (RE) are neither appropriate nor useful (Ernst and Murphy 2012). In these settings, requirements still exist, but in forms different to what requirements textbooks typically characterize as best practice. These requirement

[1]This chapter has been published at the *13^{th} International Workshop on Principles of Software Evolution (IWPSE13)* (Heck and Zaidman 2013)

approaches are characterized by the use of lightweight representations such as user stories, and a focus on evolutionary refinement. This is known as just-in-time RE (Ernst and Murphy 2012).

This just-in-time RE is also found in open source projects (Ernst and Murphy 2012; Scacchi 2001; Mockus et al. 2002). Requirements in open source projects are managed through a variety of Web-based descriptions, that can be treated collectively as 'software informalisms' (Scacchi 2001) . Traditional requirements engineering activities do not have first-class status as an assigned or recognized task within open software development communities. Despite the very substantial weakening of traditional ways of coordinating work, the results from open source software (OSS) development are often claimed to be equivalent, or even superior to software developed more traditionally (Mockus et al. 2002).

Open source development has proven to be very successful in many instances and this has instigated us to explore how requirements are managed in open source projects. We expect to find a number of useful concepts that can be directly translated to more traditional software engineering trajectories as well, as these are slowly moving from the more traditional up-front requirements engineering to more agile RE (Cao and Ramesh 2008).

In successful and mature open source projects, many users get involved and start to request new features. The developers of the system receive all those feature requests and need to evaluate, analyze and reject or implement them. To minimize their workload it is important to make sure only valid feature requests are being entered. But the developers in open source projects have no control over what the remote users enter, so we need to analyze what happens at the side of the user that is entering the feature requests:

1. Is it easy for those users to see if the feature is already present in the system?
2. Is it easy to see if the same feature has already been requested by some other user?

Our idea is that by aiding the users in entering only new and unique requests, we can minimize the workload for developers that are maintaining the open source system. Our main research question is *RQ2: how can we assist the users as the main actors in the requirements evolution process, with the purpose of simplifying the maintenance of the system*.

This leads us to our three subsidiary research questions:

RQ2.1 In what form do feature requests evolve in the open software community Web sites?

RQ2.2 Which difficulties can we observe for a user that wants to request some new functionality and needs to analyze if that functionality already exists or has been requested by somebody else before? Can we explain those difficulties?

RQ2.3 Do we see improvements to overcome those difficulties?

This chapter describes an exploratory investigation into how requirements are

managed in open source software projects. In the projects we analyzed we witnessed difficulties that users have with entering only valid feature requests. We present recommendations to overcome these difficulties.

The structure of this chapter is as follows: Section 2.1 describes the common structure of open source requirements. In Section 2.2 we introduce and analyze the problem of duplicate feature requests. In Section 2.3 we provide recommendations for avoiding duplicate requests. In Section 2.4 we present related work. In Section 2.5 we discuss our results before concluding in Section 2.6.

2.1 Open Source Requirements

Our first step is to investigate the different layouts of the open software *community web sites* in more detail. These websites are termed 'informalisms' by Scacchi (2001) as they are socially lightweight mechanisms for managing, communicating, and coordinating globally dispersed knowledge about who did what, why, and how.

The open software community web sites that we have analyzed are listed in Table 2.1. These web sites were chosen based on three criteria: 1) they have a publicly available requirements database; 2) they have an active community (i.e. the requirements database is still growing); 3) The software they are developing is widely-used or the web site is a widely-used platform for code-hosting. We chose a large enough set (i.e. 20 web sites) amongst the projects and platforms that we were familiar with ourselves, such that we would be able to manually analyze them.

Table 2.1: Projects and platforms analyzed

Apache Subversion	`subversion.apache.org`
Apache HTTPD	`httpd.apache.org`
Mozilla Firefox	`mozilla.org/firefox`
Mozilla Bugzilla	`bugzilla.org`
Android	`source.android.com`
Drupal	`drupal.org`
Tigris ArgoUML	`argouml.tigris.org`
Tigris TortoiseSVN	`tortoisesvn.tigris.org`
Netbeans	`netbeans.org`
Eclipse BIRT	`projects.eclipse.org/projects/birt`
Eclipse JDT	`.../projects/eclipse.jdt.core`
Eclipse MyLyn Tasks	`.../projects/mylyn.tasks`
Eclipse GMF	`.../projects/modeling.gmp.gmf-tooling`
KDE	`kde.org`
Gnome	`gnome.org`
Mono	`mono-project.com`
SourceForge	`sourceforge.net`
Google Code	`code.google.com`
GitHub	`github.com`
CodePlex	`codeplex.com`

We have analyzed those 20 open source community web sites by browsing through their on-line repositories. For each of the web sites we have collected the requirements related elements, see Table 2.2. If a certain element (like pull requests for Github) only appears in one web site we have marked that in the table. If the element (like a list of involved persons) appears in more web sites with different names we have also marked that in the table. For each of the elements in the table we indicate how we think it is related to requirements engineering concepts.

From that analysis we found a common structure in the requirements part. See Figure 2.1 for a generic overview of requirements related elements in open source project web sites.

In the web sites we have seen there is always some sort of 'ticket management system' for users to request new features for the system. Github has a very simple system (comments that can be tagged) while Google Code's system is a bit more elaborate with type and status. Both sites include a voting system where users can vote for feature requests to emphasize their priority. The other sites use stand-alone issue tracker systems where the description in text is just one of the fields that need to be filled when requesting a new feature. Out of the sites we investigated, most of them use Bugzilla (a Mozilla open source project) as a ticket management system, see Figure 2.2. Note that Bugzilla is designed for managing defects ('bugs') so the way to indicate that this is a feature request is by setting the *Importance* to 'enhancement', although some projects (e.g. Apache Subversion) have a field called *Issue type* where 'enhancement' and/or 'feature' are values.

Well-organized open source projects require new requirements to be entered as a specific feature request, see for example `http://argouml.tigris.org/project_bugs.html` . The issue tracker is used to discuss the feature request, to assign the work to a developer and to keep track of the status of the request. Only in some smaller projects new features are implemented directly in (a branch of) the source

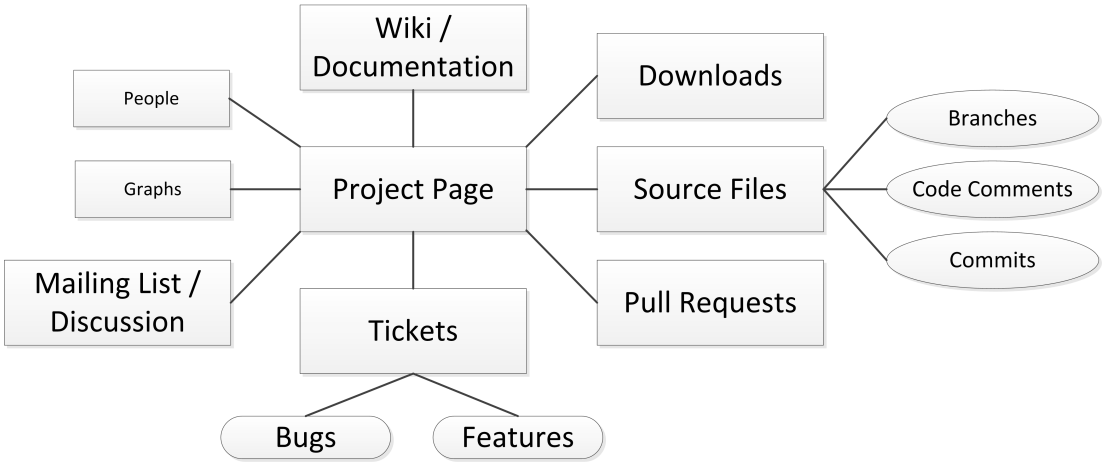

Figure 2.1: Open source requirements items

Table 2.2: Requirements elements in open source project websites

Item	Synonyms	Description	Requirements?
Project page		Main page for the project with explanation about the software and links to other pages	High-level mission of the software or releases of the software; Context description
People	People (Codeplex) Network (Github) Members (GoogleCode)	List of involved persons with roles	Stakeholders (including developers)
Graphs (Github)		Statistics about the project	-
Downloads	Files (Sourceforge)	Download of the latest executables/releases	-
Wiki / Documentation		Documentation for the project; can be on external web site; free format	System requirements from high-level description to detailed specification; degree of maintenance varies greatly with the projects
Source Files	Code (Github, Sourceforge) Source Code (Codeplex)	Source code for the project	Technical design or System requirements in code comments; comments are prerequisite for success
Branches (Github)		Different branches in the source code repository; each branch can result in maximum 1 pull request	Each branch implements different system requirements
Commits		Action of uploading new source code into the repository	Technical design or System requirements in commit comments
Pull Requests (Github)		Fork created by developer to solve issue; the developer requests for the fork to be merged with the trunk (main project)	Requirements are asserted by the developer and directly implemented and documented in the fork
Tickets	Issue Tracker (CodePlex) Issues (Github, GoogleCode) Tracker (Sourceforge)	Requests to the project to create, implement or correct something	Feature requests are requirements (synonyms: Enhancement, Feature); Github uses custom labels per project to separate bugs from features
Discussion / Mailing list		Archives of questions and answers posted by the project community; can be on-line forum or off-line mailing list with on-line archives	Usually the start of any requirement; people post message to see if a feature is needed or already being worked on; brainstorm about requirement
IRC / Chat		Live communication channel separate from the project page	Same as discussion / mailing list but no archives are kept!

code. Github uses *pull requests* to let users offer newly implemented features to the project (see Table 2.2) and automatically generates an issue in the tracker for each pull request to keep track of them.

To summarize we found that in most projects the requirements evolve in an issue tracker system.

Figure 2.2: Feature request in Bugzilla

2.2 Duplicate Feature Requests

While analyzing the Eclipse JDT Core project we noticed the huge amount of duplicate feature requests (see Table 2.3). Our first step was to see if the same is true for the other 13 projects that use Bugzilla as an issue tracker [2]. We could easily track the duplicate requests by filtering on 'severity = enhancement' and 'resolution = DUPLICATE'. It turned out that many projects, including very mature and well-organized ones, have a high percentage of duplicate feature requests, see Table 2.3. The ones where the number of duplicates is lower, either have a strict policy (Apache HTTPD and Subversion warn the user explicitly to search/discuss before entering new issues), are smaller (Eclipse MyLyn and GMF) or have a company behind the project (Mono and Android). One can easily argue that with the large number of issues in the database not even all duplicates have been marked.

Table 2.3: Duplicate feature requests in open source projects Jan 2012

Project	# Duplicate	# Request	%
Apache HTTPD	28	809	3
Apache Subversion	66	881	7
Mozilla Firefox	2920	8023	36
Mozilla Bugzilla	1166	5664	21
Android	283	5963	5
Drupal	1351	7855	17
Tigris ArgoUML	133	562	24
Netbeans	2896	10711	27
Eclipse MyLyn Tasks	16	403	4
Eclipse GMF	17	370	5
Eclipse JDT	1551	8314	19
GNOME Evolution	1843	6895	27
Mono Xamarin	11	477	2
Mono Novell	81	5277	2

So apparently users cannot or do not find out if the feature they request is really new or has already been implemented or requested before. Our next question was: what is the reason for those duplicates?

Research Strategy

The strategy we chose is to look at one of the projects with a strict policy (Apache HTTPD) and to see why still those 28 duplicates were reported. Out of the 28 duplicates one turned out to be a defect, not an enhancement. For each of the 27 remaining duplicates we analyzed the history of comments and tried to identify what the root cause for duplication was. We did this by answering the following questions:

[2] Mono has a split Bugzilla repository since the project went from Novell to Xamarin

- Is it a real duplicate? In other words: is the request of the author indeed the same as the original request?
- Could the author of the request have found the duplicate by searching in the issue tracker? With what search terms?
- Did the author submit some source code directly with the request?
- Was the duplicate submitted long after the original request?
- Who marked the duplicate? Was there an overlap in people involved in original request and duplicate?
- Do we see any way in which the author could have avoided the duplicate entry?

The analysis was done manually by the first author by reading the title and comments for each duplicate and analyzing what caused the reporter of the feature request to send in the duplicate; could the reporter have avoided that? After concluding this analysis for the Apache HTTPD project we had an initial categorization of duplicates by root cause.

Subsequently, we repeated the analysis for 5 other projects to validate the findings. These projects are Subversion, Firefox, Netbeans, Eclipse JDT and Novell Mono. Again this manual analysis was done by the first author. While we analyzed all duplicates for Apache HTTPD, the other projects had many more duplicates (see Table 2.3), so we had to select samples. For each of the projects we selected the 30 most recently reported duplicates between 01 Jan 2011 and 31 Dec 2012. For Mono Novell we just selected the 30 most recently reported duplicates without a time window (because of the smaller number of total duplicates).

This second round of manual analysis led us to do a slight adjustment to the initial categorization. The author category is now not only used for authors that enter the same request twice within a few minutes (we only saw this in the HTTPD project) but also for other situations where the author seems to be aware of the duplicate he/she created. We initially had a specific category for feature requests being duplicates of defects, but in other projects we saw situations where the *product* attribute did not match. We decided to group those two situations into 1 category for 'mismatch of attributes'.

Next to the detection of distinct categories of duplicates we did a number of other interesting observations which we discuss below.

Categorization

Grouping the duplicates with similar root causes resulted in the following categories:

Duplicate Solution [DS] This is not a real duplicate request. The request of the author is new but happens to have the same solution as a request that was posted before.

Table 2.4: Analysis of duplicate feature requests

	HTTPD	Subversion	Firefox	NetBeans	Eclipse JDT	Mono Novell	#	%
Explicit warning	Y	Y	N	N	N	N		
Solution [DS]	3	5	3	2	0	4	**17**	**10**
Partial [PM]	1	2	2	0	2	2	**9**	**5**
Patch [PA]	10	1	0	0	0	0	**11**	**6**
Author [AU]	4	5	0	2	2	7	**20**	**11**
Mismatch [MA]	1	0	20	3	7	4	**35**	**20**
Wording [WO]	7	6	2	5	5	3	**28**	**16**
No Check [NC]	1	11	3	16	14	10	**55**	**31**
(No duplicate)	0	0	0	2	0	0	**2**	**1**
#	**27**	**30**	**30**	**30**	**30**	**30**	**177**	**100**

Partial Match [PM] Only part of the request has been posted before; the main questions for original and duplicate are completely different.

Patch [PA] The author has submitted a patch directly with the request. Our assumption is that the fact that the author is proud of his own code makes him/her lazy in checking if the same has already been done before.

Author [AU] The same author enters his/her request twice or indicates in title or comments that he/she is aware of the duplicate entry.

Mismatch Attributes [MA] The original request has different values for important attributes (in our observations: *defect type/importance* or *product*) so the author might not have searched with the correct search values. An author that is entering an 'enhancement' might not think to include 'defects' in the search for existing duplicates. An author that is entering a feature request for the *product* 'Firefox' might not have included the 'Core' *product* in the search for existing duplicates.

Wording [WO] Different wording for the same problem is used in both original request and duplicate. Possibly the author did not detect the duplicate because he/she was not searching with the right terms.

No Check Done [NC] It is not clear why the author did not detect the duplicate while the duplication is easy to spot with a simple search.

For a complete analysis of the Apache HTTPD project see Table A1 and A2 in the appendix. Table 2.4 indicates for each of the analyzed projects the number of duplicates in each category.

Note that for the NetBeans projects two duplicates out of 30 ([216335] and [217150]) turned out to be no real duplicates and thus do not fall into any of the categories. In [216335] the 'DUPLICATE' link is misused to indicate some other type of relation (see **O8** below). In [217150] one person marks it as a duplicate (*"looks like another manifestation of a problem introduced in JavaFX SDK 2.2 by, as*

described in issue #216452") but two other persons later decide it is not.

As can be seen in Table 2.4 each category was present in at least two projects. Not each project shows the same division over the categories. A project like HTTPD has many duplicates because of patches added to the feature request, where as in other projects we did not find any patches. The FireFox project shows many problems with mismatching attributes because Mozilla uses one big Bugzilla database for all its Mozilla projects. FireFox is just one of the products in the database and thus it is easy to select the wrong product when entering a new request or to search for the wrong product when checking for duplicates. A bigger project with a wide user base like NetBeans shows more duplicates in the category NC. We can assume that reporters of new feature requests get easily discouraged searching for duplicates by the huge volume of existing requests. Furthermore NetBeans does not explicitly warn users to check first before entering new requests, as opposed to HTTPD.

The two Apache projects include an explicit warning at the beginning of the process, see Figure 2.3. The projects that are marked in Table 2.4 as 'N' under 'Explicit warning' do not have such an explicit warning at the beginning of the process. All of them however include a small notice or an optional step to search for previously entered issues or to ask on the discussion list first, but this is less compelling for the user than an explicit warning.

To summarize we can state that users do have difficulties in submitting unique feature requests, for different reasons. For each of these root causes we would like to offer support to the user to avoid duplicate requests.

Further Observations

While analyzing the six projects for duplicate feature request we did not only categorize the duplicates but also made some interesting observations:

[O1] Many of the 14 projects in Table 2.3 have a high percentage of duplicate feature requests. It seems to be the case that a project such as Apache HTTPD that explicitly warns the user to search and discuss before entering new requests can greatly reduce the number of duplicates. The Subversion project goes even further and explicitly asks users to file a bug report at a separate email address to get it validated before posting in Bugzilla, see Figure 2.3.

[O2] Given the high number of feature requests a manual search for certain terms can easily yield a high number of results. In our experiments it sometimes took a sophisticated search query with enough discriminating attributes set to obtain a set of search results that is small enough to manually traverse them looking for a duplicate. Even for us (who knew the duplicate we were looking for beforehand) it often took more than one step to come to the correct search query. This involved stemming of the search terms (e.g. "brows" instead of "browsing") and searching in comments of the issue instead of the summary. From our own experience we can say that the simple search screen (shown

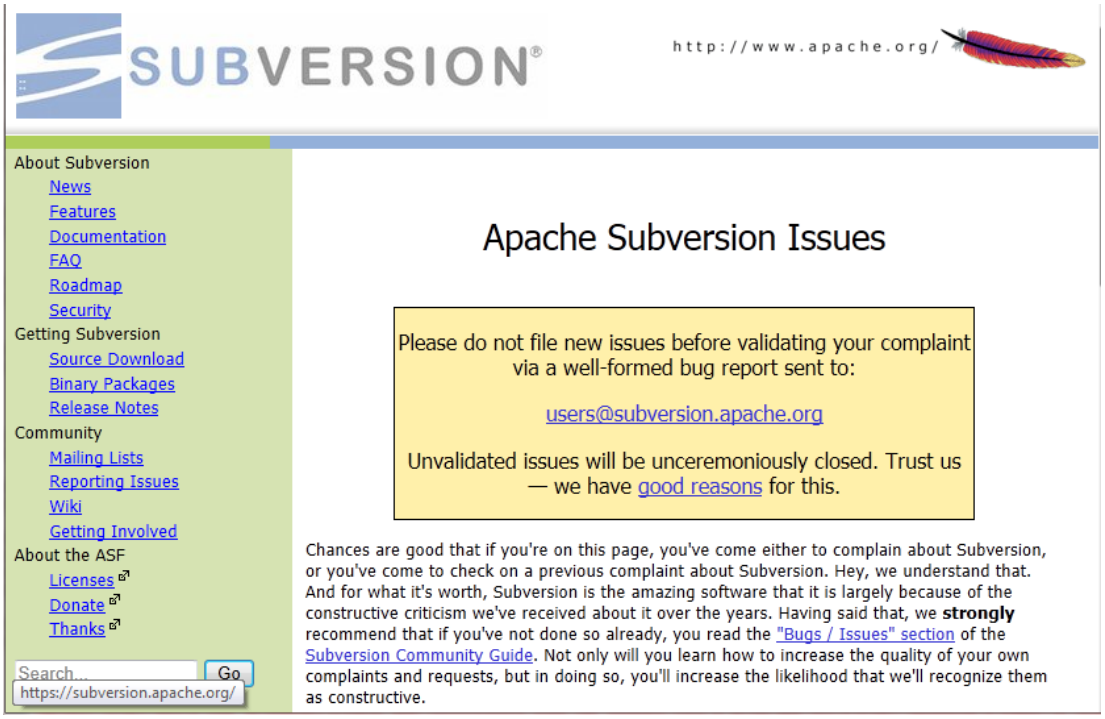

Figure 2.3: Warning on the bug report page of Subversion

by default by Bugzilla) is not enough; for most issues we needed to open the advanced search screen to find the duplicates.

[O3] Some feature requests were marked as a duplicate of a defect. Often there was discussion in the comments of an issue about whether the issues should be classified as a defect or an enhancement. Apparently this difference is not always easy to make for a user entering a new request (Herzig et al. 2012). Projects like Subversion make it even more complex by adding 'enhancement' and 'feature' both as separate types. Is it a completely new feature or an enhancement to an existing one? With a broad existing product like Subversion this question is extremely difficult to answer. One could argue that everything is an extension to the version control system and thus an enhancement. The risk is that users entering new requests will not search for duplicates of the type 'defect' and thus not find potential duplicates.

[O4] Marking of the duplicates within one project is done by a wide variety of project members: e.g. in Apache HTTPD the 27 duplicates were marked by 18 different user names. The users that marked the duplicates in this case were also not all of them part of the HTTPD core team. When we check the activity of those users in Bugzilla, we see that they are involved in 5 to 1338 issues (as Commenter or Reporter), with about half of the 'markers' involved in more than 400 issues and the other half involved in less than 75. This observation tells us that we can not assume that duplicates are easily filtered

out by a select group of developers that scans all new requests.

[O5] The time gap between the duplicate and the original request is arbitrary. In the Apache HTTPD project this ranged from 0 to 88 months. We expected to see short time gaps because we expected the user needs to be influenced by time-bounded external factors (e.g. the emergence of a new standard that needs to be applied), but this turned out to not be the case.
Note that the time gap between creation of the request and marking it as a duplicate is also arbitrary. In the Apache HTTPD project about half of the requests were marked within 2 months but the longest time gap for marking the duplicate reached up to 56 months. This indicates that some duplicates stay undetected for a long time.

[O6] During the manual analysis of the duplicates we were often hindered by the fact that many issues include a lot of off-topic comments: comments that do not pertain to the issue itself but, e.g. to project management topics or for social interaction. The same problem was detected by Bettenburg et al. (2008a) for bug reports in general. Examples for feature requests in Apache Subversion:

> [Issue 3415] *This would be a very useful addition to the product IMO.*
>
> [Issue 3030] *I see. Good to know that the issue has will be resolved in the next release. I understand you suggestion about the mailing list - however joining a mailing list for one issue in 3 years seem an overkill. (Be proud: I use Subversion ever day and had only one problem which nagged me enough to raise a report) Personally I think Google-groups got it right here: they have a read/write web interface for casual users and mail distribution for heavy users.*
>
> [Issue 2718] *Except that that bug report has undergone the usual open source 'me too'/'let's add some irrelevant technical detail' treatment, which made me create a clear and concise bug report that someone could actually fix in the next 12 months.*

Having such useless comments in the issues makes it more difficult for a user or developer to quickly grasp the content of the issue and thus more difficult for the user to see if the issue is a duplicate of the one he/she is about to enter.

[O7] We saw cases of larger structures of master-duplicate relationships such as the example from Subversion in Figure 3.1. In this example many issues around 'improved error messages' are linked together. Currently the user exploring such structures only has the possibility to click on each of the links one by one and build such a graph in his/her head. The risk is that the user will get lost while doing this for larger structures.
Note also that in this entire list, only issue 434 is typed as an 'enhancement', the others are typed as 'defect', 'task' or 'patch'. This is related to observation

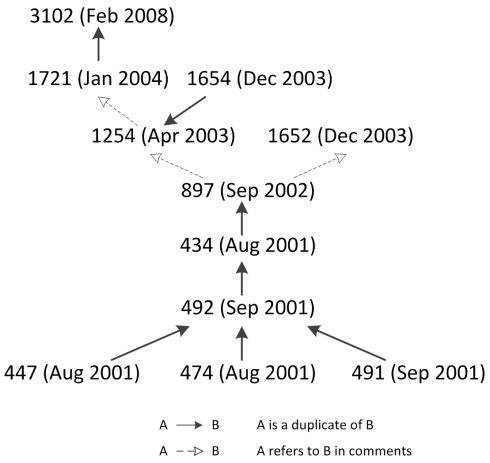

Figure 2.4: Network of 'duplicate' relations in Subversion

O3.

[O8] For the moment Bugzilla has only two explicit linking mechanisms: 'request A blocks request B' and 'request A is a duplicate of request B'. Only the first type can be visualized in a dependency tree. Another option is to implicitly link requests by entering comments with the ID of the other request included (Sandusky et al. 2004). This limitation of Bugzilla leads to misuse of the 'DUPLICATE' link as we saw already for the categories DS and PM. This misuse makes it difficult for users and developers to see the real link between two requests, which hinders comprehension of the work done or to be done. An example of this misuse is in NetBeans issue [216335]:

> *Although it is not exactly the same idea I am marking this as duplicate of issue #208358 as it addresses the same problem. Having the two enhancements logged in one place makes sense due to their close relation.*

[O9] Most users that request new features are developers themselves, as Noll states that the majority of requirements are asserted by developers (Noll and Liu 2010; Noll 2008). Developers implement what they need themselves: this will lead to on average the mostly needed features. In practice this means that part of the 'feature requests' in the issue tracker are already accompanied by a piece of source code in which the user that requests the feature implements the feature (see also **PA** before). The feature requests with patches included are often not well-described (the documentation is the source code). This makes it difficult to analyze the feature request when looking for duplicates.

All of these observations lead us to believe that things could be improved for the projects we have investigated and that those improvements could also benefit other projects. That is the topic of the next section.

2.3 Assisting Users to Avoid Duplicate Requests

In the projects we analyzed we observed some shortcomings of using a tool like Bugzilla for both defect and requirements management. Some fields for requirements (e.g. rationale, source) are missing (Robertson and Robertson 2000; IIBA 2009) leading to information loss already at the entry of a new request. To see how many users need a new feature a voting mechanism is one of the few online instruments (Dalle and den Besten 2010), but this is not always present. The main problem however is that it is difficult to indicate hierarchy or relations between requirements and to get an overview of relations that *have* been defined (Sandusky et al. 2004). This leads again to information loss (links that were known at request creation time are lost) and makes it difficult to get an overview of existing feature requests later on. Another problem is that the commenting mechanism is difficult for maintaining requirements while a comment is a free-text field. The user can enter anything he/she wants and Bugzilla enters some auto-comments for certain actions (e.g. marking as a duplicate). For the reader it will be difficult to get a quick overview of feature requests with many comments (What is the type of each of the comments? What is the final decision on the request?). Despite these disadvantages the usage of such issue trackers remains high because of the advantage of managing all development tasks (fixing bugs, writing documentation, implementing new features, ...) in one single tool.

With those drawbacks in mind we investigate the observations done before and come up with some recommendations to improve the system for the user. We start with recommendations for manual search and issue creation and end with implications for extended tool support by the issue tracker. For each of the items we refer to the category of duplicates (DS, PM, ... , NC, see Section 2.2) or the observation (O1 till O8, see Section 2.2) that has lead us to the recommendation.

Recommendations for Manual Search and Creation

[R1] Users that search for duplicates should include both defects and enhancement types in their queries (**O3**, **MA**).

[R2] Users that search for duplicates should not exclude issues before a certain date (**O5**). They could only do that if they know their request (e.g. implement a new standard) has an explicit date limit (nobody could have asked for the standard 1 year ago because it only exists for half a year).

[R3] Projects should include warnings to search first and to ask on mailing or discussion lists before entering a new request (**O1**, **NC**). Research has shown that most duplicates are reported by infrequent users (Cavalcanti et al. 2013a) so giving them a reminder of the procedure or explicit instructions could help filter out some of the duplicates. Furthermore, when the user submits a request that includes a patch an explicit warning should be repeated to remind the user that he/she should search for similar solutions first (**PA**).

[R4] Projects should include a link to clear guidelines on how to enter issues (e.g.

when is it a defect or an enhancement) to ensure that all fields are filled correctly and to avoid users entering new requests for new versions of the software (**AU**, **MA**)

[R5] Users entering new feature requests should only include issue-related comments; the same holds for the users commenting on existing feature requests (**O6**). For other types of comments the mailing/discussion list should be used (from where the user can easily hyper-link to the request). Projects could even go as far as removing unrelated comments from the request, to keep a 'clean' database.

[R6] Users should be told to split feature requests into atomic parts: one request per issue ID (**PM**). This makes it easier later on to follow up on the request and link other requests to it. When developers looking at the issue see that it is not atomic, they should close it and open two or more new ones that *are* atomic. The partial match example of Apache HTTPD shows that certain wordings can hint at non-atomic requests:

> *[29260 - Author] The base functionality of mod_vhost_alias for dynamic mass virtual hosting seems to work great, but there are* **a couple things** *that need to be changed to work with it.*
> ...
> *[29260 - Marker] Most of this is WONTFIX, but the stripping www thing is reported elsewhere.*
> **** This bug has been marked as a duplicate of 40441 ****

[R7] Projects should not accept patches or other source code as feature requests (**PA**). A patch is not a feature request, it is a request from the author that asks the project to look at something already implemented by the author. A mechanism like the pull request that GitHub (`https://github.com/`) uses is much more appropriate for submitting patches. In that way the patch is immediately available to all users (it sits waiting as a branch of the trunk repository until the owner of the trunk accepts and merges it) and this avoids users to enter the same patch twice. At the least a project could create a separate issue type for patches, making it easier to search for similar patches submitted earlier.

[R8] Projects should make clear what the hierarchy of products or components is within their issue database (**MA**). A user searching for duplicates should also include parent, children or siblings of the product he/she intended to search for, because a duplicate feature request might have been added for one of those. Especially for novice users the structure of the (group of) products might not be clear. This means they will not find some duplicate requests and unnecessarily enter a new request.

Implications for the Issue Tracker

The fact that in the investigated Apache HTTPD project many different developers are involved in marking the duplicates (**O4**) and many different users are involved in entering requests indicates that we need some kind of tool support in the issue tracker. Within such open source projects we can simply not rely on one small group of users or developers to keep an overview of the project and filter out inconsistencies.

We would like to prevent problems as early as possible: at the moment the user is entering new feature requests. Based on our observations we see different opportunities for extended tool support by the issue tracker:

Synonyms

The search tool could automatically suggest alternative search terms based on a list of synonyms or closely related words for the project, e.g. '.jpg' and 'image' (**WO**). This list could be compiled by language processing the existing feature requests and clustering them or could be manually composed. Each time a new duplicate request is found the developer marking the duplicate could verify if it was caused by a wording problem. If so, the different wordings could be added to the synonym list.

Duplicate Submission

The fact that a single author submits his/her request twice within a few minutes (we saw examples of this in the HTTPD project) could easily be filtered out automatically (**AU**). After submission, it should first be checked if the same author has already submitted a request with the same summary line, before the request is saved in the database.

My Requests

In the Subversion project we saw an example of an author 'forgetting' an earlier submitted request:

> *Wow. I *completely* forgot that I'd implemented this already (per issue #3748). Suddenly, I feel old.*

But there are more cases of the author being aware of some form of duplication (**AU**). It would be good that when a user logs in to submit a new request, he/she is first presented with a list of his/her previously submitted requests. This can serve as a reminder and as a status check. Instead of entering a new request the user can more easily go to one of the existing requests and extend it with additional information.

Comments

The issue reporting tool should offer some advanced support for discerning the different types of comments in a feature request (**O6**). In the examples we looked

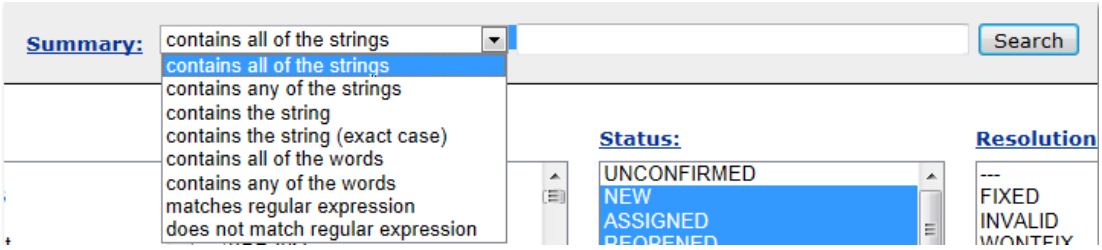

Figure 2.5: Search options in Apache HTTPD Bugzilla

at, we found comments on test cases, procedures, questions, answers to questions in other comments, automatic actions, special tags to search for a set of related issues, complaints, discussions, etc. It is not clear what type of comment it is and thus what the relevance is without reading the whole comment. And sometimes it is difficult to see which comment is answering which other comment. If there would be separate fields for some of the types (e.g. auto-generated comments could be displayed separately in some sort of history section) or tags/colors related to the type of comment this would greatly simplify the ability of a user to get a quick overview of a feature request. This overview is needed to judge if the request is a duplicate or not of the one the user is about to enter. Of course this would also demand from the users entering the comments that they would use the correct tag/color for their comment.

Linking

We need some more sophisticated linking mechanism between feature requests than the current ones in Bugzilla (**O8**, **DS**, **R6**). We could imagine newly added fields in a feature request where the user or developer can indicate links with a type and link comment, e.g. a “solution" link with comment “both can use the same library with mathematical functions mathlib". This would avoid misuse of the ‘DUPLICATE’ link and would keep users from entering comments that merely serve to indicate an informal link between requests. The extended linkage information can be useful for newcomers to get a grasp of the project.

Visualization

Currently Bugzilla only offers to visualize the so-called ‘dependency tree’. This is a graph showing which issues ‘block’ each other. This ‘blocking’ must have been indicated before by the user. For feature requests it would be more useful to be able to visualize the duplicate relations, as we did in Figure 3.1 (**O7**). And also other links if implemented (**Linking**) are a good candidate to visualize in such graph structures. We can even think of implementing tools that automatically process the feature requests to infer links between them from their texts and then visualize them.

Advanced Search

Last but not least we can think of more intelligent search engines for feature requests (**O2**, **R1**). The Bugzilla implementation of the HTTPD project searches on strings and regular expressions, see Figure 2.5. This is a good start but we suspect that natural language based search can greatly improve the search results. The search tool could be extended to automatically include the related products or components in the search (**R8**). Also the presentation of the search results to the user could be improved, e.g. with clustering or visualization techniques. A first attempt for better presentation of search results was made by Cavalcanti et al. (Cavalcanti et al. 2009) by highlighting search terms in the result set.

To summarize we see many opportunities for advanced support by the issue tracker. All of these functionalities will help in the process of understanding the current set of feature requests in a project. This serves to avoid duplicates but also in other situations where the user needs to get an overview of the existing feature set of a system, e.g. when documentation of the system's functionality is missing. What remains is to actually build that support into the issue trackers and validate it in open source and company-based projects.

2.4 Related Work

Requirements Evolution

There are many papers on the analysis of open source projects, but not so many cover requirements engineering topics. One of the first overview papers has been written by Scacchi (2001). The paper describes the open source requirements engineering process and the form of requirements in open source projects. Our analysis is based on the one of Scacchi, but proceeds to the next level of detail. For Scacchi 'open software web sites' is just one of the forms of requirements. In our analysis we dive into this by subdividing it into many different parts of requirements, see Figure 2.1.

Duplicate Detection

The fact that much duplication exists in the requirements of open source projects has also been detected by Cleland-Huang et al. (2009). In their research they focus on open forums, not on issue trackers. For the requirements-related messages on these open forums they propose an automatic clustering technique. In future work we could investigate whether this automatic clustering also applies to feature requests in the issue tracker.

Gu et al. (2011) use a similar clustering technique to automatically suggest duplicate issue reports to the author of a new issue report. Their recall rate is between 66 and 100%. Runeson et al. (2007) achieve a recall rate around 40% when analyzing defect reports using Natural Language Processing (NLP) with a

vector-space-model and the cosine measure. Sun et al. (2010) claim they obtain significantly better results by using a discriminative model. Wang et al. (2008) do not only use natural language processing but also use execution information to detect duplicates.

In our analysis we found an explanation why Gu and Runeson do not detect all duplicates: not all issues marked as 'Duplicate' are real duplicates in the sense that they could have been detected with natural language processing. This leads us to believe that next to experimenting with clustering as in (Cleland-Huang et al. 2009) and (Gu et al. 2011), we need some more sophisticated techniques like e.g. visualization, to support the author in getting an overview of the feature requests posted before.

Tian et al. (2012) extend the pioneer work of Jalbert and Weimer (2008) to improve the method of bug classification: is the bug a unique one or a duplicate? This classification could of course help in warning users that they might be submitting a duplicate, but considers only one aspect of the problem.

A different approach is used by Vlas and Robinson (2011) who have developed an automated Natural Language requirements classifier for open source projects. The tool classifies the individual sentences of a feature request into types of requirements according to an extended McCall model (McCall et al. 1977), e.g. 'operability' or 'modularity', with a pattern-based approach. A similar classification could also help in clustering complete feature requests, as we are looking for.

Cavalcanti et al. (2009) present a tool called BAST (Bug-reports Analysis and Search Tool) that provides an extended keyword-based search screen for issues. It shows issues with the same keywords, but additionally in the lower part of the screen a quick overview of important issue attributes. Their study shows that it worked better than the normal search tool for one company. The main drawback of their tool is that it is still based on keyword search and thus depends on the user entering the correct search terms. This contrasts our idea that synonym-based search should also be implemented. In a further paper Cavalcanti et al. (2013a) explore the duplication of bug reports in several projects and come up with recommendations. Most of those recommendations also match ours as they are also valid for feature requests. However, this chapter adds some feature request specific recommendations like the handling of patches, the improved linking mechanism and a better separation of comment types.

Where Bettenburg et al. (2008b) claim that duplicate bug reports are not always harmful, the same can be true for feature requests: two feature requests that are similar can help the developer in understanding the requirements. However this requires that the developer is aware of the duplicate before starting on the implementation. An unnoticed duplicate feature request can easily lead to two different developers implementing the same feature in different places in the system. This strengthens our claim that duplicate feature requests should be detected/prevented early on.

Our approach differs from the ones mentioned in this subsection because we focus on feature requests only. Feature requests are different from defects. They require different initial content (what the user needs and why vs. what is not working) and have different life-cycles. A defect stops living once resolved, but the description of a requirement is still valid documentation once implemented. We expect to extend or detail the approaches mentioned above with some requirements-specific analysis. With that we are not so much interested in the automatic detection of duplicates, but in supporting the user to get a better overview of the existing feature requests such that the user can more easily see which related (not necessarily duplicate) feature requests have already been submitted.

Visualization

Sandusky et al. (2004) studied what they call 'Bug Report Networks (BRN)' in one open source project. This BRN is similar to what we have drawn in Figure 3.1. In the bug report repository they studied 65% of the bug reports sampled are part of a BRN. They conclude that BRNs are a common and powerful means for structuring information and activity that have not yet been the subject of concerted research by the software engineering community. The continuation of this stream of research will result in a more complete understanding of the contribution BRNs make to effective software problem management. We support this from our own findings and plan to investigate what would be the best way to visualize the BRN's for our specific application domain of feature requests.

2.5 Discussion and Future Work

This section discusses the results and describes opportunities for future work.

The Research Questions Revisited

RQ2.1 *In what form do feature requests evolve in the open software community Web sites?* We analyzed the community web sites of a number of open source software projects and we found that while requirements are sometimes discussed on discussion fora or mailing lists, they are typically channeled towards the issue tracker. In particular, we observed that many open source web sites use Bugzilla as a requirement management tool to support requirements evolution.

RQ2.2 *Which difficulties can we observe for a user that wants to request some new functionality and needs to analyze if that functionality already exists or has been requested by somebody else before? Can we explain those difficulties?* We found that many duplicate feature requests exist within the projects. This indicates difficulties that user have to submit unique feature requests. We have categorized the duplicates by root cause; we created seven categories of duplicates which we have observed in six projects.

RQ2.3 *Do we see improvements to overcome those difficulties?* By analyzing the root causes of the duplicates we have suggested improvements and tool support to avoid duplicates, e.g. improved linking mechanisms, visualization of those links, clustering or advanced search. We think that all of these functionalities will help in understanding the current set of feature requests in a project. This serves to avoid duplicates but also in other situations where the user needs to get an overview of the existing feature set of a system, e.g. when documentation of the system's functionality is missing.

In future work we plan to investigate the tool support implications to see if indeed we can improve the requirements evolution of projects by providing extended issue tracking tools.

Validity

We did not try to falsify our assumptions by contacting the original authors or interviewing developers in other open source projects to see if they recognize the problems with duplicate requests. Based on the anecdotal evidence that we gathered from analyzing the issue tracker, we believe that projects would benefit from extra tools to get an overview of all feature requests but this must be validated in our future work. We could for example use a focus group of open source project members (both users and developers) to discuss the feasibility and value of each of the recommendations we do.

We are also aware of the fact that the issue tracker must stay a tool that users still want to use to report new feature requests. This means that we can not include to many obstacles for new users to enter requests. Practical experiments must be done to validate what is 'too many'.

Applicability

We have conducted our case study on an open source project. Also in companies there will be situations where end users have direct access to the issue tracker tool to enter new feature requests, so that the problem of duplicate entries is relevant. Furthermore our tool support will also help newcomer developers to get an overview of the project. In the above we have claimed that the results are also valid for company-based projects. In our future work we plan to validate this claim by applying the methods we develop also on company-based projects.

Issue Trackers

We have done a small comparison with two other widely-used issue trackers, namely *Jira* (`www.atlassian.com/JIRA`) and *Trac* (`trac.edgewall.org`), see Table 2.5. This shows us that the things we found missing in Bugzilla are also missing in the two other issue trackers. The only striking difference is the fact that Jira offers a free linking mechanism (like we recommended in 'Linking' in Section 2.3). How-

Table 2.5: Comparison with other issue trackers

	Bugzilla	**Jira**	**Trac**
Launch	1998	2003	2006
Custom Fields	In UI	In UI	In DB and config file
Labeling	With keywords	With labels	With keywords
Link	'Blocks' and 'Duplicate'	'Blocks', 'Duplicate' and 'Relates to' with link comment	'Duplicate'
Voting	Yes	Yes	No
Search	Built-in engine searches for keywords, substrings; simple and advanced search UI	Lucene engine; whole words only but can be told to stem words or do 'fuzzy' search; simple and advanced search UI;	Built-in engine searches for keywords and substrings; advanced search done with queries
Extensibility	Plugins & Open Source	Plugins	Plugins & Open Source
Interfaces	XML-RPC, REST	REST, Java API	XML-RPC

ever a newer tool like Trac does not offer this, so the recommendation in general is still valid. Jira also does not offer any visualization of those advanced links.

Other Questions

High quality feature requests could simplify the evolution of the system. But how do we define the quality of those feature requests? For regular requirements there are many characteristics that qualify a good requirement (e.g. atomic, no ambiguity, prioritized) (Heck and Parviainen 2008) but do they all hold for feature requests in an issue tracker such as Bugzilla? Can we observe interesting facts on the quality of feature requests? Do we see ways to improve the quality of feature requests? Bettenburg et al. (2008a) did similar work (including a tool) for bug reports in general, but not all their results are valid for feature requests.

2.6 Conclusion

In this chapter we have investigated requirements evolution in open source project web sites and saw that in most projects an issue tracker is used to evolve require-

ments. Within those web sites that use Bugzilla as a requirements management tool we have observed a high number of duplicate feature requests. We have made a classification of the causes for these duplicate feature requests. Using this classification we have given recommendations and implications for tool support to avoid duplicate feature requests.

Our main goal for future work is to improve tool support for dealing with feature requests in issue trackers. An important step in this direction is to give the users of these issue trackers an overview of the project, including relationships between already existing feature requests. Better search facilities and a hierarchical exploration of requirements are subsequent steps towards mechanisms to prevent duplicate feature requests from being entered. Our proposed tools will also benefit company-based projects, since a lot of them use Bugzilla-like tools for managing requirements evolution.

3:

Horizontal Traceability of Open Source Feature Requests

Agile projects typically employ just-in-time requirements engineering and record their requirements (so-called feature requests) in an issue tracker. In open source projects we observed large networks of feature requests that are linked to each other. Both when trying to understand the current state of the system and to understand how a new feature request should be implemented, it is important to know and understand all these (tightly) related feature requests. However, we still lack tool support to visualize and navigate these networks of feature requests. A first step in this direction is to see whether we can identify additional links that are not made explicit in the feature requests, by measuring the text-based similarity with a Vector Space Model (VSM) using Term Frequency - Inverse Document Frequency (TF-IDF) as a weighting factor. We show that a high text-based similarity score is a good indication for related feature requests. This means that with a TF-IDF VSM we can establish horizontal traceability links, thereby providing new insights for users or developers exploring the feature request space. [1]

Software evolution is an inevitable activity, as useful and successful software stimulates users to request new and improved features (Lehman 1984; Zaidman et al. 2010). A first step towards implementing these new features is to specify them. In projects developed using an Agile methodology this specification is typically informal, for example in the form of brief user stories which serve as conversation

[1] This chapter has been published in the *Journal of Software: Evolution and Process (JSEP)* (Heck and Zaidman 2014b).

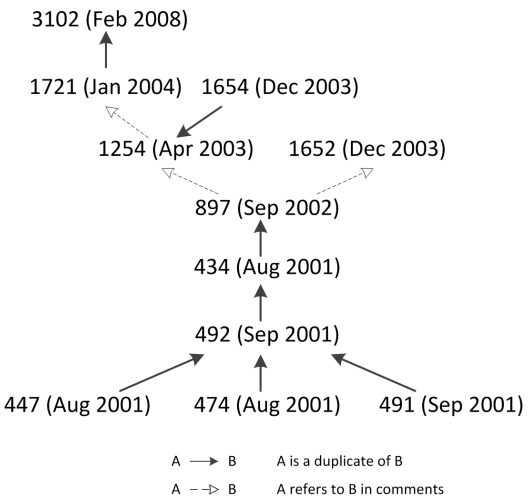

Figure 3.1: Feature request network in Subversion (feature request ID and creation date).

starters with stakeholders (Ernst et al. 2014a), or in the form of other 'software informalisms' (Scacchi 2001).

This use of more lightweight representations for requirements contrasts the more traditional notion of requirements engineering (RE) and is also known as just-in-time requirements engineering (Ernst and Murphy 2012). Ernst and Murphy (2012), Scacchi (2001) and Mockus et al. (2002) have previously observed just-in-time RE both in industrial projects and in open source projects. Mockus et al. even claimed that despite the very substantial weakening of traditional ways of coordinating work, the results from open source software (OSS) development are often claimed to be equivalent, or even superior to software developed more traditionally (Mockus et al. 2002).

Our analysis in Chapter 2 has shown that just-in-time requirements engineering typically employs an issue tracker to record *feature requests*. On the one hand we have observed that this potentially leads to a large number of requests, which are sometimes difficult to search and navigate. On the other hand, we observed that large structures of relations exist between individual feature requests, see Figure 3.1 for an example of a so-called feature request network from the Subversion project [2].

Both when developers are trying to understand the current state of the system *and* trying to understand how a new feature request should be implemented, it is important to know and understand all these (tightly) related feature requests, to avoid duplicate or inconsistent development efforts (Martakis and Daneva 2013). For users reporting feature requests having knowledge of existing feature requests can also be important to know what is already being requested. In Chapter 2 we

[2] `http://subversion.apache.org`

observed that some of these relations are explicitly mentioned (textually) in feature requests. In this chapter we focus on automatically constructing horizontal traceability links (Gotel et al. 2012) between feature requests, including the identification of additional links that are not made explicit in the feature requests, *e.g.*, because people are not aware of the related feature request. The more structure is provided, the easier it becomes for users or developers to follow these links and explore the related requests for the task at hand.

We focus on feature requests because as Cavalcanti et al. put it feature requests hold facts on the evolution of software, and thus can serve as documentation of the history of the project (Cavalcanti et al. 2013b). Discussions in some feature requests even show why the system has *not* evolved in a certain direction. Once a defect is closed ("fixed"), the information inside is not relevant for the current system anymore, because the defect does not exist anymore. However, once a feature request is closed ("fixed"), the information inside *is* relevant for the current system, because the feature still exists in its original or evolved form. This means traceability between feature requests remains interesting, even for closed feature requests.

In order to establish traceability links we use a *Vector Space Model* (VSM) with a *Term Frequency - Inverse Document Frequency* (TF-IDF) weighting factor to calculate the text-based similarity of feature requests. In the remainder of this chapter we will refer to the combination VSM - [TF-IDF] as "TF-IDF". TF-IDF has previously been applied to detect duplicate issue reports (Cleland-Huang et al. 2009; Gu et al. 2011; Runeson et al. 2007; Sun et al. 2010). Issue reports include both defects and enhancements (feature requests). In our research in this chapter we focus only on feature requests and are not only interested in duplicates, but more generally in requests that are functionally related.

We think that feature requests merit a separate text-based investigation with TF-IDF because we assume that the content of feature requests and defects is different, thus requiring a separate investigation. This assumption is supported by earlier work from Antoniol et al. (2008) who devised a classifier for separating feature request and defects, and by Ko et al. (2006) who used a decision tree algorithm to split feature requests from defects. An additional observation comes from Moreno et al. who claim that the terms used in bug reports are closely related to source code entities (Moreno et al. 2013), which might not always be true for feature requests as they can be described more abstractly.

This leads us to the **main research question RQ3** for this chapter: *Can TF-IDF help to detect horizontal traceability links for feature requests?*

Subsidiary research questions that steer our research are:

RQ3.1 Is TF-IDF able to detect functionally related feature requests that are not already explicitly linked?

RQ3.2 What is the optimal pre-processing to apply TF-IDF focusing on feature re-

quests?

Whereas TF-IDF only matches words that are literally the same, Latent Semantic Analysis (LSA) (Deerwester et al. 1990) takes into account words that are close in meaning by assuming that they will occur in similar pieces of text. However, LSA requires more calculation time because it adds additional calculation steps compared to TF-IDF. This situation intrigued us to see if the results for [RQ3.1] improve significantly enough when applying LSA to our application domain to merit this extra calculation time, which leads us to our final subsidiary research question:

RQ3.3 Does a more advanced technique like LSA (Latent Semantic Analysis) improve the detection of non-explicit links?

The remainder of this chapter is structured as follows: Section 3.1 contains background with some major concepts and the related work. Section 3.2 explains our experimental setup. In Section 3.3 we describe the results of our experiment. In Section 3.4 we explore an example of a feature request network to illustrate our results. We discuss our results in Section 3.5. Section 3.6 concludes this chapter.

3.1 Background

In this section we discuss the background of our study. We start with a general background on traceability and then focus on horizontal requirements traceability. We further explain why we think the focus on feature requests (as opposed to issue reports also including defects) is necessary. We then discuss the use of TF-IDF/VSM for duplicate detection. We end by explaining both TF-IDF and LSA in a nutshell.

Traceability

"Traceability is the potential to relate data that is stored within artifacts of some kind, along with the ability to examine this relationship" (Gotel et al. 2012). Over the last 20 years or so, the software engineering research community has investigated numerous approaches to establish, detect and visualize traceability links (Cleland-Huang et al. 2012).

An important branch of research has busied itself with so-called *requirements traceability* or "the ability to describe and follow the life of a requirement in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases)(Gotel and Finkelstein 1994; Gotel et al. 2012)". Requirements traceability can typically be described as a case of vertical traceability, where artifacts at different levels of abstraction are traced (Gotel et al. 2012).

The investigation described in this chapter however, can be catalogued as *horizontal traceability* as we are seeking to relate artifacts at the same level of abstraction.

De Lucia et al. have proposed to classify the traceability recovery methods according to the method adopted to derive links (De Lucia et al. 2008): heuristic based, IR based, and data mining based. Recently, machine-learning methods have also been proposed to recover links between code and product-level requirements (Cleland-Huang et al. 2010).

Many authors have successfully applied IR techniques including TF-IDF, e.g. (Antoniol et al. 2002; Marcus and Maletic 2003; De Lucia et al. 2006). Typical applications of IR include: concept location (Maarek et al. 1991; Poshyvanyk et al. 2007), impact analysis (Antoniol et al. 2000), clone detection (Marcus and Maletic 2001), software re-modularization (Maletic and Marcus 2001) and establishing traceability links (Qusef et al. 2014; Antoniol et al. 2002; Lucia et al. 2007; Marcus et al. 2005; Lormans et al. 2008). See Binkley and Lawrie (2011) for an overview.

The investigation in this chapter concerns recovering horizontal traceability links between artifacts of the same type, i.e. requirements.

Tracing Requirements to Requirements

We specifically discuss three works on tracing requirements to other requirements.

Hayes et al. (2003) use IR techniques for tracing high-level requirements to low-level requirements. Candidate links generated by IR algorithms were to be confirmed by requirements analysts to measure performance of the algorithms. They extend TF-IDF with a simple thesaurus (manually made) to improve recall and precision of generated traceability links.

Natt och Dag et al. (2004) use IR techniques for linking customer wishes to product requirements. They use a VSM with a different weighting factor [1 + 2 log (term frequency)].

Cleland-Huang (2012) identifies three types of traceability that might be useful in agile projects. The third type is "requirements to requirements": to track dependencies between user stories. This can be useful during the planning process. She claims that there is no need to retain such traces once stories have been implemented. We argue that this is different for feature requests in open source projects: the traces should be retained as they form an important part of project documentation.

We are not only interested in "traceability links" between feature requests, but also in related feature requests in general (e.g. about the same functionality). From our experience in open source projects there is not a habit of splitting high-level feature requests into low-level feature requests and we do not see any distinction in types of requirements, so with our TF-IDF analysis we can not make use of such an existing structure to find related requests.

Clustering

Cleland-Huang et al. (Cleland-Huang et al. 2009) propose an automatic clustering technique based on TF-IDF for the requirements-related messages on open forums. Their goal was to create threads of related messages. We focus on feature requests in issue trackers.

Networks

Kulshreshtha et al. (2012) studied literature about dependencies between system requirements and abstracted this into four dependency types: Contractual, Continuance, Compliance, Cooperation and Consequential. They modeled a dependency network from a set of 50 requirements pertaining to a Hotel Front Office Reservation system. Their study is on relationships between traditional requirements, where we focus on agile requirements (feature requests).

Sandusky et al. (2004) studied what they call "Bug Report Networks (BRN)" in one open source project. This BRN is similar to what we have drawn in Figure 3.1. In the bug report repository they studied, 65% of the bug reports sampled are part of a BRN. They conclude that BRNs are a common and powerful means for structuring information and activity that have not yet been the subject of concerted research. The continuation of this stream of research will result in a more complete understanding of the contribution BRNs make to effective software problem management.

Feature Requests vs. Defects

As claimed in the introduction feature requests and defects have different characteristics. In this section we provide further support for that assumption from our own dataset and from existing literature.

The datasets we use are the feature requests as listed in the issue trackers of Mylyn Tasks, ArgoUML and Netbeans; more details on these projects can be found in Section 3.2. However, we start of with an investigation into who submits defects on the one hand and feature requests on the other hand. Our data is depicted in Table 3.1. This table shows the total number of feature requests and defects, the number of submitters for feature requests and defects, and the total number of submitters. What we see is that the intersection of submitters of feature requests and submitters of defects is relatively small (between 13.5 and 21% of the total number of submitters)[3]. This shows that the group of submitters for feature requests and the group of submitters for defects contain different persons. Those different persons might also be using different vocabulary.

Herzig et al. (2013) investigated the relationship between issue reports and source code change sets. They found that this relationship is different for defects

[3]It is known that sometimes the same person might submit under different user names, but we assume this holds only for a small number (possibly zero) of usernames in our large dataset.

Table 3.1: Number of submitters for feature requests and defects

Project	Feature requests		Defects		Submitters	Submitters
	#	# Submitters	#	# Submitters	total	feature req. $\bigcap$ defects
Mylyn Tasks	452	112	583	158	**223**	**47**
ArgoUML	1323	399	4847	1226	**1426**	**199**
Netbeans	29023	5017	204827	23071	**24737**	**3351**

and features. Defect fixing change sets seem to change older code while feature implementations are based on newer code fragments. They also find that feature implementing change sets have more structural dependency parents in the change genealogy graph than defect fixing ones. Above all defect fixing change sets show smaller impact on code complexity than feature implementations.

Herzig et al. (2012) investigated the classification of existing bug reports (as indicated by the project members) in five open-source Java projects. They find that 34% of reports classified as "bug" is actually not a bug and only 3% of reports classified as feature requests is actually a "bug".

TF-IDF and Duplicate Detection

Several authors use Vector Space Models and/or TF-IDF similar weighting functions in the area of duplicate detection of bug reports (Jalbert and Weimer 2008; Tian et al. 2012; Runeson et al. 2007; Sun et al. 2010, 2011; Gu et al. 2011). Runeson et al. (2007) achieve a recall rate around 40% when analyzing defect reports using Natural Language Processing (NLP) with a vector-space-model and the cosine measure. Wang et al. (2008) do not only use natural language processing but also use execution information to detect duplicates. Sun et al. (2010) claim they obtain significantly better results (a relative improvement of the recall rates of 17-43% compared to the techniques used in (Jalbert and Weimer 2008), (Runeson et al. 2007) and (Wang et al. 2008) by using a discriminative model.

We only use duplicate detection to tune our pre-processing step. Our main research question is not focused on the best duplicate detection, but on the identification of horizontal traceability links for feature requests.

TF-IDF in a Nutshell

We use TF-IDF to convert a collection of documents into a collection of vectors (our Vector Space Model, VSM) by calculating the relative importance of each word in each document. The TF-IDF value increases with the number of times a word appears in a document, but decreases with the number of documents in which the word appears. This helps in filtering out words that are common in the entire collection of documents.

The formula that we have used for TF-IDF is as follows:

$$TFIDF(w,d,D) = TF(w,d) \times IDF(w,D) \tag{3.1}$$

with w a word in a document d that belongs to the collection of documents D. The Term Frequency $TF(w,d)$ can be computed in several ways. We choose to correct for longer documents, to prevent a bias towards longer documents:

$$TF(w,d) = \frac{f(w,d)}{max\{f(w,d) : w \in d\}} \tag{3.2}$$

with $f(w,d)$ the number of times that word w appears in document d.

$$IDF(w,D) = log_e \frac{|D|}{|\{d \in D : w \in d\}|} \tag{3.3}$$

with $|D|$ the total number of documents in the collection and $|\{d \in D : w \in d\}$ the number of documents where word w appears.

For a collection of documents D we first determine the complete set of unique words W. This set can be taken as is or can be normalized in many different ways (see section 3.2). Then we transform each document d that belongs to the collection D into a vector v_d. The dimension of this vector is equal to the size of the set of unique words W. The ith element of the vector v_d corresponds to ith word in the set W (w_i). For each d and i we calculate:

$$v_{d,i} = TFIDF(w_i,d,D) \tag{3.4}$$

Once we have each document vector we can calculate the so-called cosine similarity between two documents n and m.

$$\cos(v_n, v_m) = \frac{v_n \cdot v_m}{\| v_n \| \| v_m \|} \tag{3.5}$$

This yields a value between 0 and 1 indicating how similar the text of the two documents is: closer to 1 meaning more similar, 0 meaning no words in common at all.

Latent Semantic Analysis (LSA) in a Nutshell

Latent Semantic Analysis (LSA) (Deerwester et al. 1990) takes into account words that are close in meaning by assuming that they will occur in similar pieces of text. The input for LSA is a matrix. As described in (Dumais 2004) this is a weighted term-document matrix: the output matrix of the TF-IDF weighting algorithm as defined above (by combining all vectors v_d). This results in a matrix M of documents D against unique words W:

$$M[i,j] = TFIDF(w_i,d_j,D) \tag{3.6}$$

LSA uses a mathematical technique called singular value decomposition (SVD) to transform the so-called 'hyperspace' M into a more compact latent semantic space.

This is done by maintaining the k largest singular values (also called rank-lowering of M). The optimal value of k is unique for each application domain. If the value of k is too low, the decomposition may end up under-representing the hyperspace M. Alternately, if the choice of k is too high, the decomposed sub-space may over-represent the hyperspace by adding in noise components (Santhiappan and Gopalan 2010).

After rank-lowering we have a new matrix M'. In this matrix M' each column represents a document vector. We calculate the so-called cosine similarity between two documents n and m in the same way as we do for TF-IDF above.

To summarize, the LSA technique adds reduced-rank SVD as an extra step compared to TF-IDF. It is believed that in the vector space of reduced dimensionality, the words referring to related concepts, i.e., words that co-occur, are collapsed into the same dimension. Latent semantic space is thus able to capture similarities that go beyond term similarity (e.g. synonymy) (Santhiappan and Gopalan 2010).

3.2 Experimental Setup

In order to answer [RQ3.2] *What is the optimal pre-processing to apply TF-IDF focusing on feature requests*, we decide to make all pre-processing steps (see section 3.2) optional to find out which of these configurations yields the best results on our datasets. We define 'the best results' as the configuration that succeeds best at finding the known set of duplicates. Known duplicates are feature requests marked as 'DUPLICATE' for the field *Resolution*. We use known duplicate feature requests from three mature open source projects. All three projects use Bugzilla[4] as an issue tracker to manage feature requests. We download these feature request as XML-files from the following three open source projects[5]:

- Eclipse MyLyn Tasks `projects.eclipse.org/projects/mylyn.tasks`, 425 feature requests, 10 'Duplicate';
- Tigris ArgoUML `argouml.tigris.org`, 1273 feature requests, 101 'Duplicate';
- Netbeans `netbeans.org`, 4200 feature requests, 342 'Duplicate'.

We select these specific projects based on four criteria: 1) they are mature and still actively developed; 2) they differ in order of magnitude in terms of number of feature requests; 3) they have a (substantial) number of known duplicate feature requests; 4) they use Java as a programming language (important because some feature requests contain source code fragments).

We implement the TF-IDF calculation with possible preprocessing configurations in a C# tool, called FRequAT (Feature Request Analysis Tool)[6]. FRequAT automat-

[4] `www.bugzilla.org`

[5] Datasets can be downloaded from `http://dx.doi.org/10.6084/m9.figshare.1030568`

[6] Available from first author through email.

ically verifies the similarity score against known duplicates (see Section 3.2). We use Mylyn Tasks and ArgoUML for the tuning of our algorithms and then attempt to confirm our findings with the Netbeans project.

Our FRequAT tool produces an Excel file that contains the pair-wise cosine similarity score for the complete set of feature requests. We do not have a golden set of all horizontal traceability links in the set of feature requests, thus we cannot use standard performance measures as recall and precision to evaluate TF-IDF for finding links. We therefor devise two different ways to answer [RQ3.1] *Is TF-IDF able to detect functionally related feature requests that are not already explicitly linked*:

1. The top 50 most similar pairs of feature requests (*i.e.*, the pairs with the highest cosine similarity score) are manually analyzed by the first author. For the pairs that do not have a physical link in the Bugzilla repository confirmation of our findings is requested from one main developer from the Mylyn Tasks project and two main developers from the ArgoUML project.

2. We investigate an existing feature request network (one large example found during manual exploration of the repository of the Mylyn Tasks project) to validate that the existing links are supported by a high cosine similarity score and to see whether we can extend the existing network with additional links based on high cosine similarity scores.

To answer [RQ3.3] *Does a more advanced technique like LSA improve the detection of non-explicit links* we repeat the step of manually analyzing the top 50 most similar pairs of feature requests (see [RQ3.1] above) to see if we find more related requests with LSA than with TF-IDF only. We only do this for the Netbeans project because we deem the other two projects too small in terms of number of documents.

We explain some more details of the experimental setup in the following sections.

Preprocessing Feature Requests

The main step of the tokenization (*i.e.*, parsing the feature request text into separate words) is done through regular expressions. After that a number of configuration options are available to arrive at the final set of words to be considered for each document:

- *Include All Comments [AC].* The main parts of a feature request are: title, description and comments. This option configures if only the title and description or the complete feature request are used to build the VSM.
- *Set to Lowercase [LO].* This option sets all words to lower case.
- *Remove Source Code [SC].* This option uses extra regular expressions to remove different types of source code. The source code is completely removed, including comments, names of variables and identifiers.

- *Remove Spelling Errors [SP].* For this step we use a list of project-specific abbreviations and spelling mistakes that the first author manually compiled from the Mylyn Tasks project (we do not repeat this for the other projects because of their much larger vocabulary). If the option [SP] is on, each word is checked based on the Mylyn Tasks list and replaced by the alternative word before further processing.
- *Remove Stop Words [SR].* This option excludes known stop words from the VSM. We construct a list of stop words starting from the SMART list[7]. We manually add 66 stop words from the Mylyn Tasks project to this list (like 'afaik', 'p1', 'clr')[8]
- *Stem Words [SM].* This option reduces words to a common base: *e.g.*, 'browsing' and 'browse' become 'brows' and 'files' becomes 'file'. We use Porter's Stemming Algorithm (Porter 1980).
- *Create Bi-Grams [BG].* This option builds the VSM based on sequences of two consecutive words, instead of based on single words.
- *Set Title to Double Weight [DW].* This option sets a double weight for the title as opposed to the description and comments (Gu et al. 2011; Runeson et al. 2007) .

Our FRequAT tool allows to switch the above options on or off. We define the best configuration as the one that yields the highest ranking for the known duplicates in our 3 projects. When we calculate the pair-wise similarity between a known duplicate and all other feature requests, then rank by similarity in descending order, the master of the known duplicate should be in the top of this ranking. A similar duplicate detection recall rate is used by Runeson et al. (2007), who state that the standard information retrieval definitions of *recall* and *precision* do not really work for duplicate detection. The *Recall rate* is defined as the percentage of duplicates for which the master is found for a given top list size. We are interested in the TF-IDF configuration with the highest recall rate. In our evaluation we use top list sizes of 10 and 20.

We try several combinations of options on the Mylyn Tasks project and the ArgoUML project. We calculate the top-10 (T10) and top-20 (T20) recall rates for those combinations. Then we repeat the best combinations on the larger Netbeans project to get a confirmation of the highest recall rate found.

Most Similar Requests

We use the TF-IDF configuration with the highest recall rate for duplicate detection to see, for a given set of feature requests, whether we can get new information about related feature requests by ranking them by pair-wise similarity. Our idea is that two feature requests with a high pair-wise similarity should be about the same topic and thus related.

[7] SMART list: `ftp://ftp.cs.cornell.edu/pub/smart/english.stop`
[8] Extra stop words `http://dx.doi.org/10.6084/m9.figshare.1030568`

The relatedness of two feature requests is determined in several ways:

1. The feature requests are related because they have one of the three existing link types in Bugzilla:
   - The 'blocks/depends' link. This is a manual link that is indicated by one of the project members, indicating that one should be resolved before the other.
   - The'duplicate' link. This is a manual link that is indicated by one of the project members.
   - The feature requests are related because one refers to the other in comments. This is manually done by one of the project members.
2. The feature requests are related because we as authors find them to be about the same topic.
3. The feature requests are related because one of the main project members considers them as related.

We calculate the pair-wise similarity for all feature requests from each project. Then we rank them and extract the top-50 most similar pairs of feature requests for each project. We choose 50 as a cut-off because 50 (times three for three projects) seems a reasonable amount to base conclusions on, and because we want to limit the effort for manual investigation. The top-50 is manually verified for the first three options above (existing links in Bugzilla) by the first author by looking at feature requests in Bugzilla. For the last two options we need a more extensive manual verification. The authors validate the remaining feature request pairs as being about the same topic or not. Lastly we send the remaining feature request pairs (meaning the ones in the top-50 not already linked in Bugzilla) to a prominent project member asking for his/her opinion about the relatedness. For this purpose we do not define 'relatedness', thus leaving it to the interpretation of the project member what he/she considers as 'related'.

LSA

We use the TF-IDF configuration with the highest recall rate for duplicate detection to further apply SVD (see Section 3.1). For application of the LSA algorithm we determine the k-value to use empirically by scanning the range (0 to 4200, i.e. the number of feature requests for Netbeans) in steps of 50. We use the k-value with the highest top-10 and top-20 recall rates for retrieving known duplicates. This is the same logic as described before to determine the optimal pre-processing steps for TF-IDF.

With this optimal k-value we calculate LSA-based cosine similarities and repeat the step of manually analyzing the top-50 most similar pairs of feature requests (see Section 3.2 above) to see if we find more related requests with LSA than with TF-IDF only.

3.3 Results

This section describes the results of our experiments[9]. We start off by describing the results of our investigation into the best pre-processing configuration for applying TF-IDF by using the known duplicate feature requests (Section 3.3). Subsequently, in Section 3.3 we analyze whether we can identify related feature requests, looking both for relationships that are already known and relationships that have previously not been documented explicitly. In Section 3.3 then, we compare the results that we have obtained with TF-IDF to the results obtained with LSA.

Best Pre-Processing Configuration

In order to establish the best pre-processing configuration(s), we present an overview of the different configurations and their recall rates in Table 3.2. We observe that for the smaller Mylyn Tasks project several configurations score 90/100 recall rates, but when also taking the ArgoUML and Netbeans project into consideration, we observe that a single configuration scores best among all three projects.

In what follows, we discuss the influence of the pre-processing configurations.

Table 3.2: Recall rates for the three projects

	Mylyn Tasks		ArgoUML		Netbeans	
Options	T10	T20	T10	T20	T10	T20
BG	30	40	-	-	-	-
SR	50	50	58	71	40	49
None	**50**	**70**	**55**	**73**	**42**	**51**
LO	50	60	63	78	51	59
SP	50	70	55	73	42	51
SC	50	70	55	73	42	51
SM	40	70	57	74	42	53
DW	50	70	67	73	45	51
AC	80	90	60	72	57	65
AC_SC_SP	90	100	61	71	61	69
AC_DW_SC_SP	90	100	72	78	61	70
AC_DW_SC_SP_SM	90	100	73	79	60	71
AC_DW_SC_SP_LO	90	100	75	80	67	75
AC_DW_SC_SP_SM_LO	**90**	**100**	**79**	**85**	**67**	**77**
DW_SC_SP_SM_LO	50	80	75	83	55	63

As Stop Word Removal (SR) is often performed as a standard step in research using TF-IDF (Gu et al. 2011; Jalbert and Weimer 2008; Sun et al. 2011), it is surprising to see that it yields worse results when applied. Apparently the built-in

[9]Raw cosine similarity files as produced by our FReQuAT tool can be found on `http://dx.doi.org/10.6084/m9.figshare.1030568`

correction for common words (Inverse Document Frequency) in TF-IDF performs better than automatically removing all stop words from our manually compiled list. An option for future work would be to investigate the alternative technique presented by De Lucia et al. (2013) to use a smoothing filter to remove "noise" from the textual corpus.

The Bi-gram option (BG) produced very low recall rates. The options DW, SC, SM, SP improve the recall rates.

Regarding the All Comments (AC) option, from our first experiments with Mylyn Tasks, improved recall rates are achieved by including all comments. With the ArgoUML project the improvement is not clear. However, when adding Source Code Removal (SC) and Stemming (SM), the results for AC improve to the highest recall rates. This makes sense because when adding all comments we get a lot of diversity in the text of feature requests. Replies often include source code to provide a solution or hint for a solution. By removing the source code and stemming the words, the long feature requests get more similar. To be sure about this setting we repeat the experiment with AC off (last line in Table 3.2). From this it is clear that overall we get better recall rates with All Comments.

To summarize, the best configuration (last bold line in Table 3.2) is to activate all options except SR (stop word removal) and BG (bi-grams).

A Note on Categories of Duplicates

In Chapter 2 we defined different duplicate types or categories: duplicate solution [DS], partial match [PM], different wording [WO], author knows [AU], no check done [NC], patch [PA], mismatch of attributes [MA]. In Table 3.2 it can be seen that with the best configuration 15% (ArgoUML) to 23% (Netbeans) of the duplicates is not detected within the top-20. We now analyze the duplicates that remain undetected within the top-20 of Table 3.2 to determine in which duplicate category they can be situated. It turns out that most non-detected duplicates are from three categories, see Table 3.3:

- *Duplicate Solution [DS].* The two requests are not duplicate, but their solutions are. Developers tend to link feature requests as 'DUPLICATE' because they could be solved in a similar way. However the requests and their wordings are sometimes (completely) different. As such, text-based similarity sometimes does not detect this type of duplicates.
- *Partial Match [PM].* The master and the duplicate are not exactly the same because one contains more requests than the other. Only one of these requests is a duplicate of the other feature request. We then do not easily find this type of duplicates with text-based similarity analysis, because one of them contains extra text about an entirely different topic. An option for future work would be to investigate a different similarity measure such as the asymmetric Jaccard index to see if it better takes into account documents of different lengths.
- *Wording [WO].* The master and duplicate describe the same feature using

Table 3.3: Categories of undetected duplicates. Categories taken from Chapter 2.

	ArgoUML	Netbeans	Netbeans LSA
Not detected in top-20	**15**	**78**	**74**
Partial Match [PM]	7	28	25
Wording [WO]	5	23	24
Duplicate Solution [DS]	-	17	15
No Check Done [NC]	3	8	8
Author [AU]	-	2	2

entirely different terminology for the most important concepts. Examples we saw are 'nested class' vs. 'innerclass' or 'move' vs. 'drag/reposition'. Using TF-IDF we cannot detect such similarities without manual intervention. This motivated us to also investigate LSA as an alternative technique.

Two duplicates are of the category 'Author' [AU], meaning the author was aware of the duplicate. Normally we would hope to find duplicates from this category with TF-IDF because of use of similar words by the same author. In these 2 cases this does not work because of a long comment that is present in only one of the two requests, in one case, and because of the spelling of 'outline view' versus 'outlineview', in the other case.

The rest of the feature requests were of the category 'No check done' [NC], meaning it is not clear why the author did not detect the duplicate: searching for the main word in the title of the duplicate already results in finding the master. It is not immediately clear why using TF-IDF does not work well for these feature requests. It could be due to the fact that by using All Comments [AC] some of the feature requests become very long, increasing the chance that many other feature requests achieve a high similarity score. In that case, the real duplicate would not stand out any more.

Our analysis shows that out of the non-detected duplicates a number of them could not have been detected by (TF-IDF) text-based similarity and we can clarify that by the category they belong to. This means that a recall rate of 100% is not achievable for most projects (like ArgoUML and Netbeans in our study). Previous work on duplicate detection, e.g. (Jalbert and Weimer 2008; Tian et al. 2012; Runeson et al. 2007; Sun et al. 2010, 2011; Gu et al. 2011), did not investigate the duplicates that have been manually marked by the project. Our analysis shows that the data itself can be polluted (*e.g.*, the PM category are not real duplicates, although they have been marked as such), yielding lower recall rates than expected. This pollution of data could be solved by providing the project members with better linking mechanisms for feature requests, because now often the 'DUPLICATE' link is misused for other purposes (see also Chapter 2).

Related Feature Requests

As explained in Section 3.2 we extract and rank the top-50 most similar pairs of feature requests per project.

Table 3.4 summarizes the results of the top-50 analysis. The table shows, for each of the three projects, how many feature request pairs are physically linked (block/depend, duplicate, in comment) in the Bugzilla repository. This category of pairs is clearly related (we have evidence in Bugzilla) and does not need to be verified by the developers. As can be seen from the table, around 50 percent of the pairs is 'linked'. The rest of the pairs needs to be considered manually. For the Mylyn Tasks project we contact the project leader and for the ArgoUML project we get feedback from two lead developers. We do not contact any developers for the Netbeans project. The 50th most similar pair for the Netbeans project still has a similarity score just under 0.70, which is higher than for the first 2 projects (0.46 and 0.48). The higher the similarity score, the 'closer' the text is, the easier it is to determine relatedness. In Table 3.4 we mark a pair as 'Related, confirmed' when both we and the project developers have indicated them as being related. As can be seen from the table a low percentage of pairs is not linked at all, around 10%.

Table 3.4: Related pairs from top-50 most similar.

	Mylyn Tasks	ArgoUML	Netbeans	Netbeans LSA
Linked	20 (40%)	28 (56%)	29 (58%)	25 (50%)
Related, confirmed	15 (30%)	19 (38%)	-	-
Related, not confirmed	8 (16%)	1 (2%)	15 (30%)	17 (34%)
Not related	7 (14%)	2 (4%)	6 (12%)	8 (16%)
Unmarked duplicates	-	-	5	4

We analyze why some of the feature requests are considered as related by the authors, but not by the developers. This mainly happens because of the difference in interpretation of 'related'. For the authors 'related' means 'on the same topic' and for the Mylyn developer it means 'in the same module of the software'. The developer definition should be investigated differently, *e.g.*, by analyzing commit logs. We stick to our definition of related because we think it is useful to have a link between feature requests that are about the same topic or feature. This can help, *e.g.*, new users that would like to ask for an extension to an existing feature but first want to see what is already there, or it could help developers that are new to the project and would like to get an overview of which functionality is there from a user point of view. With our topic-based definition of relatedness we have around 90% related feature request pairs in the top-50.

For the Netbeans project we detect in the top-50 five pairs of feature requests that are indeed duplicates of each other. Their title and description indicate that the two are real duplicate requests asking for the same feature. But those pairs have

not been marked as such in the Bugzilla repository. These feature request pairs are called 'Unmarked duplicates' in Table 3.4. It shows that our method can also find new duplicate requests, i.e. not having a 'duplicate' link in Bugzilla yet.

An interesting side-effect of analyzing the top-50 most similar feature request pairs is that we see that the same feature request appears in several pairs in the top-50. This means that we can start to see small groups of related feature requests. Intuitively, these small groups should contain feature requests about the same topic. This is an example of such a group from the Mylyn project:

[102854] add bug editor support for **voting**

[156742] Add search criteria for tasks/bugs having **votes**

[256530] [upstream] **voting** for a bug capability missing from editor when no votes present

[316881] provide **voting** API in BugzillaClient

The Mylyn group about "Voting" can only be found completely when searching for "vot". This search for the stem of a verb is not natural to an end-user and he/she risks not to find the complete set. That is why our TF-IDF approach can add value, even if it is just text-based on literal strings.

We also see examples of feature requests that are related according to TF-IDF scores (0.64) and according to the developer, but that do not have related titles, e.g.

[Mylyn 283200] [Bugzilla] Support querying over custom fields

[Mylyn 339791] Bugzilla search page did not store chart settings

A simple search for 'Bugzilla' in the Mylyn Tasks feature database yields 40 results among which the two related ones are hard to find based on the rest of the title. This example shows that the VSM with TF-IDF can find related feature requests that are hard or impossible to find with simple search in the issue tracker.

What we can conclude from this experiment is that high values of cosine similarity can be a good indication of relatedness between two feature requests. We will explore this further in Section 3.4 when looking at a feature request network.

LSA

With the same pre-processing as for the TF-IDF application (AC_DW_SC_SP_SM_LO) we determine the optimal k-value to be 1700. The top-10/top-20 recall rates for k=1700 are only slightly higher than for TF-IDF: 69/78 instead of 67/77. This could be due to the fact that our dataset is relatively small (only 4200 'documents'); when considering applications of LSA, most applications use 10,000 documents or more (e.g., see (Deerwester et al. 1990)). Nevertheless, we consider Netbeans with its 4200 documents as a large project. Another possible explanation of why LSA does not outperform TF-IDF is that the community of users of issue trackers use the same kind of terms to refer to concepts, whereas LSA would be better at linking different terms that have a similar meaning. This assumption needs to be checked in future work.

If we look at the categories of non-detected duplicates, we even find one more non-detected duplicate of the category 'Wording' [WO], see Table 3.3. Furthermore we still do not detect any extra duplicates of the 'No check done' [NC] or 'Author' [AU] categories. This means LSA in our case does not perform better than TF-IDF for detecting those categories of duplicates, contrary to the expectation that we expressed in Section 3.3.

In the analysis of the top-50 most similar feature request pairs, we find many of the feature request pairs that were also there with TF-IDF (36 out of 50 are the same, although their position within the top-50 differs for some of them). Noteworthy is that the 50th most similar pair has a significantly higher cosine similarity than with TF-IDF: 0.79 instead of 0.69. When we analyze the top-50 most similar feature request pairs (see Table 3.4) we see that for TF-IDF 88% of the top-50 feature request pairs are related (coming from the categories Linked and Related), compared to 84% in the case of LSA. Additionally, the TF-IDF result-set contains 6 unrelated feature requests, while for LSA this is 8. This indicates that TF-IDF results are slightly better for our dataset.

We do not invest time in optimizing the performance for both TF-IDF and LSA. Currently the processing time for calculating the output term/document matrix for the Netbeans project is approximately 3 minutes with TF-IDF and 900 minutes with LSA. Even with optimization the processing time for LSA would likely still be considerably higher than that for TF-IDF, because LSA requires large matrix multiplications.

3.4 Extending a Feature Request Network

Knowing that VSM with TF-IDF can find new related feature requests, we want to look into what this means for the feature request networks as presented in the introduction. One would assume that the feature requests within such a network have a high pair-wise similarity.

Figure 3.2 shows a feature request network from the Mylyn Tasks project. Each node is a feature request identified by a six-digit unique ID. There are three types of links between the feature requests: 1) 'blocks/depends', 2) 'duplicate' and 3) comments. These three types are also explained in Section 3.2. For each link in the figure the pair-wise cosine similarity is indicated. If the feature request is linked to a defect, the cosine similarity is not calculated, indicated with a 'D'. Indeed we can see that this similarity ranges from 0.33 to 0.69 with a few low outliers. Our explanation for those low numbers:

- *0.04/0.06/0.09*: this is due to a partial match (see also Chapter 2). Feature request [354023] is about the implementation of a web service between Mylyn Tasks and Bugzilla. The other 3 feature requests linked to [354023] need this web service to be implemented (hence the links), but are not really about the same topic.

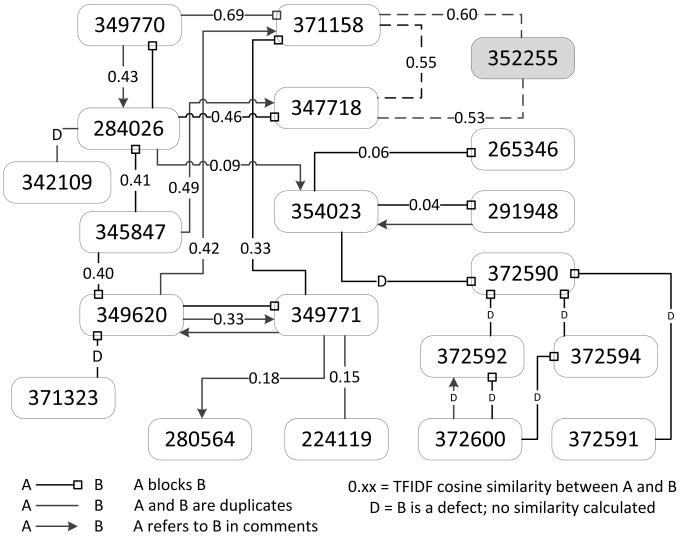

Figure 3.2: Feature request network for the Mylyn Tasks project

- *0.15*: feature request [224119] describes a new functionality in Bugzilla that needs to be available in Mylyn Tasks from the Bugzilla viewpoint. Feature request [349771] describes the same, but from a Mylyn viewpoint with much more detailed discussion. That is why the two feature requests are not so similar. However, they are known duplicates. In our duplicate analysis the duplicate is still found within the top-20 because other feature requests are even less similar to [224119].
- *0.18*: again a partial match. Feature request [280564] is about creating scalable icons in general. Feature request [349771] is about the lock icon, that needs to be scalable as well, but the scalability is just one of the many comments in this feature request. However, if we rank the most similar feature requests for [280564] then [349771] is on the 7th place, thus within the top-10 most similar.

This analysis teaches us that we should not only look at absolute similarity numbers to find related requests, but more specifically include the relative similarity. This could be done by, *e.g.*, taking into account the mean m and standard-deviation s of the pair-wise similarity of feature request A with all other feature requests, when calculating the relative similarity between feature requests A and B.

In Figure 3.2 we also add feature request [352255] with dotted lines. This feature request is in the top-50 of highest pair-wise similarity for the Mylyn Tasks project. In this top-50 it is linked to two other feature requests, [371158] and [347718], with a high cosine similarity. Interesting to see is that [347718] and [371158] also have a high pair-wise similarity while they have no direct link in the issue tracker. We are thus able to extend the physical feature request network in the issue tracker with a new feature request and new links through our TF-IDF calculations.

3.5 Discussion

Revisiting the Research Questions

In the introduction we set out to investigate whether TF-IDF can help to detect horizontal traceability links for feature requests. Answering this step is part of our bigger research ambition to visualize related feature requests that are stored in issue trackers. Before answering this high-level research question, let us first consider the subsidiary research questions.

RQ3.1 *Is TF-IDF able to detect functionally related feature requests that are not already explicitly linked?* We show that a VSM with TF-IDF can be beneficial to detect new related feature requests. We cross-check our results with feature requests that are already marked as related in the issue tracker, with our own opinion while reading the feature requests, and for two out of the three projects (Mylyn Tasks and ArgoUML) also with the help of experienced project members. We confirm this by means of an existing feature request network from the Mylyn Tasks project. This analysis teaches us that we should not only look at absolute similarity numbers to find related requests, but more specifically include the relative similarity.

RQ3.2 *What is the optimal pre-processing to apply TF-IDF focusing on feature requests?* Through case studies with the Mylyn Tasks, ArgoUML and Netbeans projects we determined the optimal pre-processing does not include stop word removal, while removal of source code is beneficial. Additionally, we determined that all comments of the feature request should be included. We attribute this to the fact that important words concerning the feature are repeated in the comments.

RQ3.3 *Does a more advanced technique like LSA (Latent Semantic Analysis) improve the detection of non-explicit links?* The results of our experiment with LSA show us that our initial assumption (that LSA would improve results as it takes into account e.g. synonyms) does not hold for the Netbeans case. We are not interested in achieving higher recall rates, but in finding related requests. LSA scores lower on finding related request based on the top-50 analysis. This makes us even more satisfied with the results achieved with TF-IDF (also because the LSA calculations take much more time).

Main research question RQ3 *Can TF-IDF help to detect horizontal traceability links for feature requests?* We can confirm TF-IDF finds related entries in an issue tracker. In our experiment, on the one hand we retrieved already known relations between feature requests, while on the other hand we retrieved feature requests pairs that were previously not marked as related, but that are confirmed to be related by project members after we confront them with our results. We explain our results for one example of a feature request network.

Threats to Validity

We now identify factors that may jeopardize the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for

case studies research (see (Runeson and Höst 2009; Yin 2013)) we organize them into categories:

Construct validity We evaluate the best configuration of our text-based similarity approach via the recall rate of previously known duplicate feature requests. These duplicates were marked by the project team. Similarly, for evaluating the related feature requests, we rely on information in the issue tracker entered by the project team, on our own opinion and on extra insights obtained from project members. What must be noted in this case is that for the ArgoUML project the two different developers do not always agree on related or not related (discussing between 'yes' and 'somewhat'), although they only completely disagree on 1 item (out of 14). As one of the developers says this difference is because "it is subjective whether two feature request are related" (*e.g.*, related in terms of implementation or in terms of topic). This could mean that if we would ask different developers we would get different answers.

External validity In this study we investigate feature requests of three software projects: Mylyn Tasks, ArgoUML and Netbeans. We choose them to be sufficiently different, in terms of domain and size. Yet, with only three data points, we cannot claim that our results generalize to other systems.

Reliability In this chapter we rely on our FRequAT tool, which we have thoroughly tested and which we consider reliable.

3.6 Conclusion

This chapter describes three case studies with feature requests from open source projects. Our main contributions are:

1. TF-IDF can be used to detect horizontal traceability links for feature requests, something which we validated for 2 out of the 3 projects with the help of developers.
2. Configuration of the pre-processing step is analyzed separately, because we focus on feature requests. For our three projects stop word removal is not beneficial, whereas including 'All Comments' and removing source code is yielding better recall rates.
3. LSA does not provide better results than TF-IDF in detecting horizontal traceability links for feature requests in our case studies.
4. When using 'Duplicate' links in issue trackers to base recall rates on, one should be aware of data pollution caused by misuse of the 'Duplicate' link, yielding lower recall rates.

Our results will help others to properly set up pre-processing for information retrieval techniques with feature requests, and to get insight in feature request networks. Especially those feature request networks play an important role in understanding the evolution of the specification of the system (recorded in the form of feature requests).

The results in this chapter show that TF-IDF can help to detect horizontal traceability links for feature requests. A next step is to get measures on thresholds, recall and precision for the retrieval of those links, by using (industrial) case studies where we are able to identify all those links on before hand.

A subsequent step is to create tool support for automatically creating feature request networks so that users can maximally benefit from the horizontal traceability links.

Another avenue for future research is to compare additional information retrieval approaches for our particular problem domain, in similar vein to Oliveto et al. (2010). Part of that investigation should also determine why LSA is currently unable to outperform TF-IDF.

4

Quality Criteria for Just-in-Time Requirements: Open Source Feature Requests

Until now quality assessment of requirements has focused on traditional up-front requirements. Contrasting these traditional requirements are just-in-time (JIT) requirements, which are by definition incomplete, not specific and might be ambiguous when initially specified, indicating a different notion of 'correctness'. We analyze how the assessment of JIT requirements quality should be performed based on literature of traditional and JIT requirements. Based on that analysis, we have designed a quality framework for JIT requirements and instantiated it for feature requests in open source projects. We also indicate how the framework can be instantiated for other types of JIT requirements.

We have performed an initial evaluation of our framework for feature requests with eight practitioners from the Dutch agile community, receiving overall positive feedback. Subsequently, we have used our framework to assess 550 feature requests originating from three open source software systems (Netbeans, ArgoUML and Mylyn Tasks). In doing so, we obtain a view on the feature request quality for the three open source projects.

The value of our framework is three-fold: 1) it gives an overview of quality criteria that are applicable to feature requests (at creation-time or just-in-time); 2) it serves as a structured basis for teams or projects that need to assess the quality of their JIT requirements; 3) it provides a way to get an insight into the quality of JIT requirements in existing projects. [1]

[1] This chapter is submitted to the *Requirements Engineering Journal (REJ)* (Heck and Zaidman 2015a).

It is increasingly uncommon for software systems to be fully specified before implementation begins. As stated by Ernst et al. (2014b), "The 'big design up front' approach is no longer defensible, particularly in a business environment that emphasizes speed and resilience to change". They observe that an increasing number of industry projects treat requirements as tasks, managed with task management tools like Jira or Bugzilla. A similar task-based approach is seen in the agile movement and in open source projects (Koch 2004; Warsta and Abrahamsson 2003). In an earlier paper Ernst and Murphy (2012) use the term 'just-in-time requirements' (JIT requirements) for this. They observed that requirements are "initially sketched out with simple natural language statements", only to be fully elaborated (not necessarily in written form) when being developed. This indicates that the notion of quality for JIT requirements is different from the notion of quality for traditional up-front requirements.

Requirements verification "ensures that requirements specifications and models meet the necessary standard of quality to allow them to be used effectively to guide further work" (IIBA 2009). Verification activities as in this definition ensure that the requirements are specified in a correct way. In this chapter we focus on the informal verification of JIT requirements, which we will call *quality assessment*. Standards such as IEEE-830 (1998) define criteria for 'informal correctness': requirements should be complete, unambiguous, specific, time-bounded, consistent, etc. However, this standard focuses on traditional up-front requirements. These are requirements sets that are completely specified (and used as a contract) before the start of design and development. We have not found a practical implementation of quality assessment for JIT requirements.

There is some evidence that correctly specified requirements contribute to a higher software product quality (Génova et al. 2013; Kamata and Tamai 2007; Knauss and El Boustani 2008). The question is: does the same hold for JIT requirements? After all incorrect JIT requirements will be spotted early on because of short iterations and can be more easily corrected because of high customer involvement. Our hypothesis is that, at the least, early verification helps to save time and effort in implementing the requirements. After all, even if the work can be redone in a next iteration to correct wrong implementations, it still pays off to do it right the first time. We would like to investigate which quality criteria define 'doing it right' for JIT requirements. This leads us to our **main research question RQ4**: *Which criteria should be used for the quality assessment of just-in-time requirements?*

We use a quality framework from previous work (Heck et al. 2010) to present the JIT requirements quality criteria in a structured way. Ernst and Murphy (2012)

describe two types of JIT requirements: *features* and *user stories*. User stories are the requirements format typically used in agile projects (Leffingwell 2011). A feature or feature request is a structured request (issue report with title, description and a number of attributes) "documenting an adaptive maintenance task whose resolving patch(es) implement(s) new functionality" (Herzig et al. 2012). Most open source projects use this type of structured requests (also called 'enhancements') for collecting requirements (see Chapter 2). For an example, see Figure 4.2. For the purpose of this chapter, we focus on open source feature requests documented in on-line issue trackers because of their public availability and their structured (i.e. the fields of the issue tracker) nature. This leads us to the detailed research question: **[RQ4.1]** *Which quality criteria for informal verification apply to feature requests?*

As the first version of our quality framework for feature requests is based on literature and our own experience we deem it necessary to evaluate the resulting criteria with practitioners. This leads to the following research question: **[RQ4.2]** *How do practitioners value our list of quality criteria with respect to usability, completeness and relevance for the quality assessment of feature requests?*

Once practitioners deem our framework valuable, we apply it to existing open source projects. In that way we get both a) experiences in applying the framework in practice and b) insight into the quality criteria of feature requests in open source projects. This constitutes our last research question: **[RQ4.3]** *What is the level of quality for feature requests in existing open source projects as measured by our framework?*

The remainder of this chapter is structured as follows. Section 4.1 explains the quality framework used. Section 4.2 instantiates the framework for feature requests. Section 4.3 indicates how to customize the framework for other situations and types of JIT requirements, with an example for user stories. Sections 4.4 and 4.5 describe our evaluation of the framework. Section 4.6 highlights the findings from the application of the framework to the existing projects. Section 4.7 discusses the research questions, including recommendations for practitioners working with feature requests, while Section 4.8 contains related work. Section 4.9 concludes this chapter.

4.1 A Quality Framework

In previous work we have performed an in-depth study on quality criteria for traditional up-front requirements, which we collected from an extensive list of standards (ISO/IEC/ IEEE/ESA) and a literature review (see (Heck et al. 2010) for more details). The resulting quality criteria are included in the Software Product Certification Model (SPCM), a quality framework for software products with traditional up-front requirements.

The SPCM divides a software product (including all design, documentation and

tests) into so-called 'elements'. For traditional up-front requirements the elements according to SPCM are: use cases or functional requirements, behavioral properties (e.g. business rules), objects (in e.g. an entity-relationship diagram or a glossary) and non-functional requirements.

The SPCM furthermore structures the quality criteria for all parts of a software product in three groups, called Certification Criteria (CC):

[CC1] Completeness. All required elements are present. Group CC1 contains quality criteria for three different levels of detail (or formality) in those elements: required, semiformal or formal.

[CC2] Uniformity. The style of the elements is standardized. Group CC2 contains quality criteria for three different levels of standardization of those elements: within the project, following company standards, following industry standards.

[CC3] Conformance. All elements conform to the property to be certified. For requirements this property typically is "Correctness and consistency": each element in the requirements description is described in a correct and consistent way. Furthermore, the relations between the elements in the requirements description are correct and consistent. The quality criteria in group CC3 for correctness and consistency of traditional up-front requirements from SPCM are (see Heck et al. (2010) for more details):

1. No two requirements contradict each other
2. No requirement is ambiguous
3. Functional requirements specify what, not how
4. Each requirement is testable
5. Each requirement is uniquely identified
6. Each use-case has a unique name
7. Each requirement is atomic
8. Ambiguity is explained in the glossary

A Quality Framework for Just-in-Time Requirements

Through an analysis of JIT requirements we evaluate which of the quality criteria from the SPCM are also applicable to feature requests (see section 4.2). Based on the SPCM we define the same three overall criteria for JIT requirements. We just rename them to Quality Criteria (QC):

[QC1] Completeness. All required elements should be present. We consider three levels: basic, required, and optional. In that way we differentiate between requirement elements that are mandatory or nice to have.

[QC2] Uniformity. The style and format should be standardized. A standard format leads to less time for understanding and managing the requirements, because all stakeholders know where to look for what information or how to read e.g. models attached to the requirement.

[QC3] Conformance. The JIT requirements should be consistent and correct.

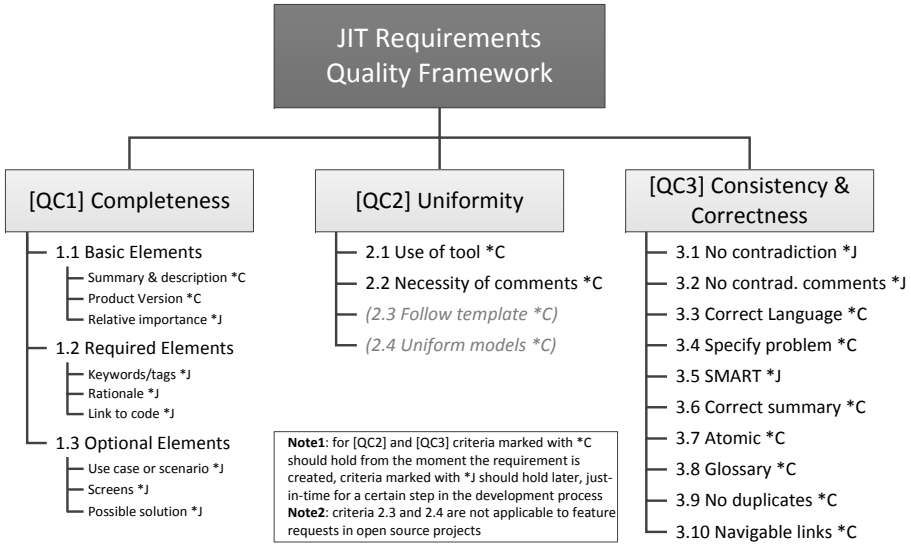

Figure 4.1: JIT Requirements quality framework, see also Tables 4.6 and 4.7

The overall QCs are detailed into specific criteria [QCx.x] for each type of JIT requirements. Figure 4.1 shows the instantiation of the framework with the specific quality criteria for feature requests. These specific criteria will be explained in the next section.

There is however another dimension to JIT requirements that clearly differentiates them from traditional requirements, namely the observation that JIT requirements are "initially sketched out with simple natural language statements" (Ernst and Murphy 2012), only to be fully elaborated when being developed. This difference points to the need for a notion of *time* in our quality framework. For each of the quality criteria we indicate when it should hold:

***C** At creation-time. This criterion should hold as soon as the requirement or the requirement part is created.

***J** Just-in-time. This criterion does not necessarily have to hold when the requirement (part) is created. However, it should hold at a later moment, just-in-time for a certain step in the development process. This could be further detailed by specifying which step is the latest moment for the criterion to hold.

In that way the framework can be used to give a structured overview of requirement qualities that should be there from the beginning and requirement qualities that should be there just-in-time, see also Figure 4.1.

4.2 Specific Quality Criteria for Feature Requests

To answer *[RQ4.1] Which quality criteria for informal verification apply to feature requests?* we evaluate which of the quality criteria from SPCM (see Section 4.1) are applicable to feature requests (in open source projects). Next to that we try to

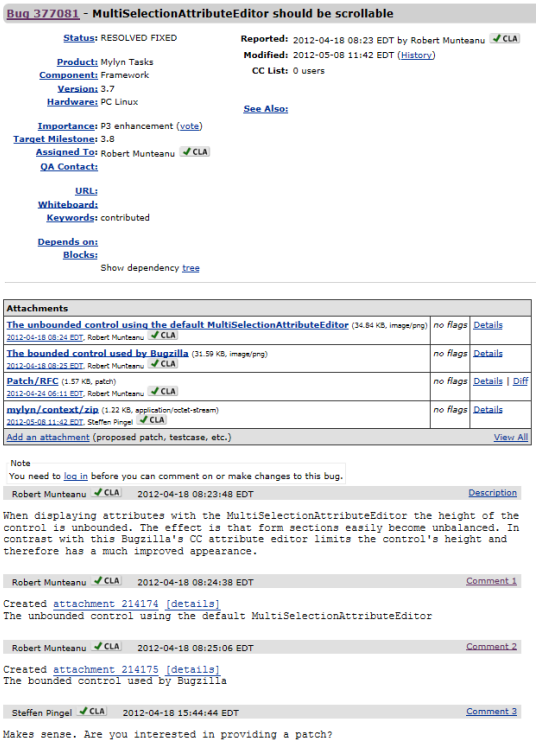

Figure 4.2: Feature Request in Bugzilla (Mylyn Tasks project)

come up with new specific criteria based on existing literature about just-in-time requirements and based on our own experience with feature requests in open source projects. Section 4.3 indicates how the framework could be instantiated for other types of JIT requirements.

Feature Requests in Open Source Projects

A feature request typically corresponds to one requirement ("a documented representation of a condition or capability needed by a user to solve a problem or achieve an objective" (IEEE 1990)). In open source projects they are used as JIT requirements: a feature request is specified by users or developers at any moment and the development lead(ers) decide(s) if and at what moment it is implemented. They select the feature requests to be implemented based on priorities set by the developers and/or the users. The initial specification of the feature request might be 'incomplete' or 'incorrect'. With JIT requirements this is acceptable, as long as the specification is corrected once the feature request is selected for implementation. This 'correction' of an open source feature request is done by adding comments to the original request. In this way a discussion is created that continues until all parties are satisfied with the implementation of the request. This is different from traditional up-front requirements that are specified as a complete, prioritized, correct and consistent set. This set of requirements is usually collected in one big 're-

quirements document' before the implementation starts. Correction of traditional up-front requirements is usually done by producing a new requirements document.

The below sections analyze what the just-in-time specification of feature requests in open source projects means for the quality assessment of those feature requests. The complete list of specific criteria for feature requests can be found in Figure 4.1 and is explained in more detail in Table 4.6 and Table 4.7. In the below analysis each of the specific criteria is clarified and indicated with "(QC xx *J/*C)". This is the identifier of the specific criterion and an indication if the criterion should hold at creation time (*C) or just-in-time (*J). This indication is also repeated in Table 4.6 and Table 4.7.

Completeness for Feature Requests

Completeness (QC1) in our framework means that all elements of the specification are present. This should not be confused with the completeness of the content of the specification ('did we specify the complete user need?').

The SPCM (see section 4.1) considers a requirement specification complete if it includes use cases or functional requirements, behavioral properties (e.g. business rules), objects (entity model or glossary) and non-functional requirements.

Alspaugh and Scacchi (2013) find that the overwhelming majority of requirements-like artifacts in open source projects may be characterized as what they term *provisionments*. Provisionments "state features in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. Most provisionments only suggest or hint at the behavior in question; the expectation seems to be that the audience for the provisionment is either already familiar with what is intended, or will play with the cited system and see the behavior in question firsthand."

This form of specification lacks most of the elements that are considered in the SPCM for traditional up-front requirements, so we cannot judge completeness by these elements. Instead we look at the attributes of a feature request. Which fields need to be filled for a feature request to be complete? The basic elements (QC1.1) are the ones that define a feature request, such as title (= unique name) and description. The required elements (QC1.2) are the ones that are necessary for management of the feature request: keywords to organize them, a rationale to determine importance and a link to the source code once implemented for traceability. The optional elements (QC1.3) are the ones that add value to the developer when specified by the author, but can also be clarified later on in the process (e.g. by prototyping or asking questions): scenarios, screen mock-ups or hints for a solution.

Based on what we observed while analyzing a large number of open source feature request we determine that at creation time (*C) of the feature request, the author only needs to fill the summary (= title) and description (QC1.1a *C) of what

he/she requires and the product (also which version of the product, QC1.1b *C) for which they require it. As can be seen in Table 4.6, all other fields/attributes can be filled in at a later moment during the development cycle (*J). Relative importance (QC1.1c *J), rationale (QC1.2b *J), scenarios (QC1.3a *J), screen mockups (QC1.3b *J) and hints for a solution (QC1.3c *J) need to be present just before coding starts because they determine when and how things get implemented. Although keywords (QC1.2a *J) should be added by the author of the feature request at creation time, they can be updated during the entire lifecycle of the requirement (just-in-time), because new topics can emerge in the discussion of the feature request. For the link to the source code (QC1.2c *J) it is obvious that it can only be added once the feature request is implemented.

Uniformity for Feature Requests

Uniformity (QC2) means all requirements have the same format. For traditional up-front requirements, the SPCM (Heck et al. 2010) defines three levels of uniformity: all elements have the same format, all elements follow company standards, all elements follow industry standards. For feature requests in open source projects, company or industry standards usually do not apply. For example, feature requests are text-only, so no modeling language is used that can be compared to industry use. Most format choices for feature requests are determined by the issue tracker being used (QC2.1 *C). Issue trackers have a number of pre-defined fields that must be filled in and that are always shown in the same way. We recommend to use the issue tracker from creation-time of the feature request (*C) such that all information on the complete life-cycle of the feature request is logged in one place. Note that although company standards do not apply, open source projects might have specific uniformity criteria on top of the use of an issue tracker. These uniformity criteria (like "All titles start with a code for the module of the software that the request is for") should also be included as specific criterion under QC2, in addition to the general specific criteria as mentioned in this chapter.

The other thing to look at is the 'uniformity of comments' (QC2.2 *C). A feature request is entered with summary and description by the author. Then other persons (users or developers) can add comments to the feature request. This is done for discussion of the request or for tracking the implementation of the request. The comments in the different feature requests should be uniform, meaning they should be 'necessary to understand the evolution of the feature request'. This is a subjective criterion but it definitely rules out comments like "I am sorry for losing the votes." (Netbeans FR #4619) or "Wooo! Party :) thanks!" (Netbeans FR #186731). Uniformity of comments is needed from creation-time of the comment (*C), because no unnecessary comments should be created at all.

All uniformity criteria should hold from creation-time (*C) because at any given moment in the development cycle the team or project profits from things being specified in a uniform manner. For example, if a team or project uses a specific template

for specifying JIT requirements, then it makes no sense to create the requirement in any other format than with this template.

Consistency and Correctness for Feature Requests

Consistency and correctness (QC3) indicate those criteria that state something on the quality of an individual feature request (correctness), or on the quality of the link between two or more feature requests (consistency).

For each of the eight SPCM quality criteria from Section 4.1 we discuss if, and how, they apply to feature requests. The resulting quality criteria for feature requests are mentioned between round brackets (QCx.y *J/C), see Figure 4.1 for an overview and Table 4.7 for the description of those criteria.

SPCM CC3.1 No two requirements contradict each other.
Does this hold for feature requests? For a complete set of up-front requirements contradictions can more easily be established then for the ever-growing set of feature requests in an open source project. As feature requests are typically submitted by many different authors, they often do not have a good picture of the feature requests that have been submitted before, resulting among others in many duplicate requests (see Chapter 2). The identification of related and possibly conflicting feature requests (QC3.1 *J) is important for developers to determine the correct implementation. Another check that can be done is to see that the comments of a single feature request are not contradicting each other (QC3.2 *J). Ideally the creation of conflicting requests and conflicting comments is avoided all-together, but since this is very hard with an open source on-line community where every user can submit feature requests and comments, we require that at least just before development starts (*J) all conflicts should be resolved.

SPCM CC3.2 No requirement is ambiguous.
Does this hold for feature requests? As stated by Philippo et al. (2013) there are many factors that can decrease the effect of ambiguity and most of them are accounted for in JIT environments. For feature requests it is not such a problem if the description is ambiguous because there is a habit of on-line discussion before implementation (Scacchi 2009). Another method that is frequently used in open source projects is prototyping (Alspaugh and Scacchi 2013). We however require a basic level of clarity from the author of a feature request at creation-time (*C): write in full sentences without spelling/grammar mistakes (QC3.3 *C).

SPCM CC3.3 Functional requirements specify what, not how.
Does this hold for feature requests? As indicated above the author of a feature request may include hints for implementation of the feature request. As mentioned in Noll and Liu (2010) the majority of features is asserted by developers. This makes it more natural that some feature requests are stated in terms of the solution do-

main (Alspaugh and Scacchi 2013). They should however at creation-time (*C) *also* specify the problem that needs to be solved (QC3.4 *C), for developers to be able to come up with alternative solutions.

SPCM CC3.4 Each requirement is testable.
Does this hold for feature requests? As Alspaugh and Scacchi (2013) state, an open source product that is evolving at a sufficiently rapid pace may be obtaining many of the benefits of problem-space requirements processes through solution-space development processes. This means that the fact that some feature requests may not be specified in a testable way can be compensated by follow-up discussions in comments, extensive prototyping and involving the author of the feature request as a tester later in the process (*J). However, the author is required to submit verifiable feature requests and make the statement as precise as possible (QC3.5 *J): e.g. "I cannot read blue text on an orange background" instead of "I need more readable pages".

SPCM CC3.5 Each requirement is uniquely identified.
Does this hold for feature requests? A unique identifier is added automatically for each new feature request that is entered in an issue tracker (IQ1, see Section 4.2).

SPCM CC3.6 Each use-case has a unique name.
Does this hold for feature requests? Each feature request should have a unique name ('Summary' or 'Title', QC1.1a *C). The summary should be in the same wording as the description and give a concise picture of the description (QC3.6 *C) from the moment the feature request is created (*C).

SPCM CC3.7 Each requirement is atomic.
Does this hold for feature requests? For feature requests in an issue tracker it is very important that they are atomic, i.e. describe one need per feature request (QC3.7 *C). If a feature request is not atomic from creation-time (*C) the team or project runs into problems managing and implementing it (a feature request cannot be marked as 'half done'). There is a risk with non-atomic feature requests that only part of the feature request gets implemented because the comments only discuss that specific part and the other part gets forgotten.

SPCM CC3.8 Ambiguity is explained in the glossary.
Does this hold for feature requests? An open source project (like any project) is likely to use some specific terminology (like 'DnD' means 'Drag and Drop') but the bigger and older the project gets, the more likely that new persons arrive, unfamiliar with that terminology. It is a good practice to maintain a glossary (e.g. wiki-pages) for such project-specific terms and abbreviations (QC3.8 *C) and add any unclear terms in the feature request from the moment it is created (*C). The advantage of on-line tools is that one can easily link terms used to such a glossary.

Issue trackers offer functionality to mark feature requests as 'DUPLICATE' such that users and developers are always referred to the master discussion. There are open source projects with many duplicate entries in their issue trackers (see Chapter 2). This is a risk because discussions on both duplicate feature requests might deviate if the duplication-relationship goes unnoticed (QC3.9 *C). Worst case this leads to two different implementations of the same feature. From the very first moment that a duplicate is detected (*C) it should be marked as such, to avoid duplicate work being done.

A last item is about the linking of feature requests. Each link to another feature request should be clearly typed and navigable (QC3.10 *C) from the moment it has been created (*C). To refer to another feature request the author of a comment should insert a URL (some tools do this automatically when using a short-code) and give an explanation why he/she is linking the two requests.

Inherent Qualities of Feature Requests in Issue Trackers

As indicated above, this chapter only considers open source projects that use issue trackers to store requirements. Those issue trackers by design fulfill the following quality criteria. These *inherent qualities* [IQx] are not explicitly included in our quality framework.

[IQ1] Unique ID: as stated above an electronic tool will automatically assign a unique ID to each added requirement.

[IQ2] History: electronic tools automatically track all changes to a requirement. This can be viewed directly from the tool's GUI or in the database.

[IQ3] Source: electronic tools automatically log the author of a requirement and the author of each comment.

[IQ4] Status: electronic tools have a separate 'Status' field where the status of the requirement can easily be seen. Most tools support a work-flow in which the status field is updated (manually or automatically) based on the work-flow step the requirement is in.

[IQ5] Modifiable (see Davis et al. (1993)): electronically stored requirements are by definition modifiable because the tool provides a structure and style such that individual requirements can easily be changed.

[IQ6] Organized (see Davis et al. (1993)): electronic tools offer an easy way to add attributes to requirements. With built-in search options this allows the tool user to locate individual requirements or groups of requirements.

4.3 Instantiating the Framework for Other Types of Just-in-Time Requirements

The previous sections describe the specific criteria for feature requests in open source projects as presented in Table 4.6 and Table 4.7. This chapter explores the applicability of the framework for other types of JIT requirements.

The framework consists of the following elements: 1) the three overall quality criteria (QC1, QC2 and QC3), 2) a time dimension (*C/*J), 3) a list of application specific criteria (QC1.x till QC3.y). For the application of the framework to a new type of JIT requirements the three overall quality criteria and the time dimension remain the same and just the list of application specific criteria should be adjusted to the specific situation for the team or project.

An example of this is to customize the specific criteria for the tool that the team or project is using. Teams or projects that use the quality framework for their JIT requirements should check if their tool also defaults the six quality criteria in Section 4.2. If not, it makes sense for them to include the not supported criteria as extra check in [QC1] or [QC3]. An example of customization is demonstrated in the next paragraph where we analyze which specific criteria would apply for another important type of JIT requirements as discussed in Ernst and Murphy (2012): User Stories.

For a team or project to customize the specific criteria for their specific JIT environment is simple in theory (the team or project decides on the specific checks and metrics) but at the same time difficult in practice (on what grounds would they decide this?). Our advice is to start with the criteria list in Tables 4.6 and 4.7 and first decide which criteria are (not) relevant. Then missing criteria can be found by interviewing members, by re-evaluating old requirements (why do we think this requirement is good/bad?), or by just applying the framework in practice and improving it on-the-fly.

Then the team or project should decide if they need to get an absolute scoring for the JIT requirement, or need to obtain a professional opinion. In most cases it is more important to find violations (e.g. "Do I see not-SMART statements" or "Do I see irrelevant comments?") and improve the requirement based on that, than to get absolute scorings for the requirement. As such exact metrics might not be needed; a simple Yes/No answer for each criterion with the goal to answer all criteria positively could be enough.

Specific Quality Criteria for User Stories

As a feature request can also be described with one or more user stories (Leffingwell 2011), we investigate whether the same quality criteria apply. A user story is the agile replacement for what has been traditionally expressed as a functional requirement statement (or use case). A user story is a brief statement of intent that describes something the system needs to do for the user. User stories usually take a standard (user voice) form: 'As a <role>, I can <activity> so that <business value>' (Leffingwell 2011).

In Table 4.1 we detail the differences for user stories with the specific quality criteria we defined for feature requests (see Figure 4.1) .

[QC1.x′] indicates that the criterion has the same title as for feature requests,

Table 4.1: Specific criteria for user stories

[QC1.1′ *C]	Basic Elements: Role, activity, business value ('Who needs what why?') instead of summary and description
[QC1.2′ *J]	Required Elements: acceptance criteria or acceptance tests to verify the story instead of rationale (already as business value in QC1.1′)
[QC1.3′ *J]	Optional Elements: the team could agree to more detailed attachments to certain user stories (e.g. UML models) for higher quality
[QC2.3 *C]	Stories Uniform: each user story follows the standard user voice form
[QC2.4 *C]	Attachments Uniform: any modeling language used in the attachments is uniform and standardized
[QC3.5′ *J]	INVEST: User stories should be Independent, Negotiable, Valuable, Estimable, Small, Testable (Wake 2003)

but with different elements that should be part of the user story. [QC2.3] is added for user stories because the user voice form as mentioned before is specific for user stories. [QC2.4] is added for user stories because we did not see any feature requests in open source projects that have attachments with detailed specification models, but we have spoken to many companies that use this mechanism to provide more detail for their user stories; this form of specifying user stories is also mentioned by Leffingwell (2011). Leffingwell introduces INVEST(see (Wake 2003)) as the agile translation of SMART, hence [QC3.5′]. The other [QCx.x] are valid for user stories without changes.

A team working with user stories should decide which specific quality criteria apply to their practice. If e.g. the product owner is in a remote location, then the quality criteria for documented user stories should be applied. If e.g. user stories are only documented as a 'user voice statement' and comments are discussed off-line, then [QC2.2] and [QC3.2] do not apply. The quality criteria could be incorporated into the team's 'Definition of Ready' (see (Power 2014)) that determines when a user story is ready to be included in the planning for the next development iteration.

4.4 Empirical Evaluation of the Framework for Feature Requests: Setup

The result of our analysis on *[RQ4.1] Which quality criteria for informal verification apply to feature requests?* is the framework as presented in Section 4.2, see also Figure 4.1.

To answer *[RQ4.2] How do practitioners value our list of quality criteria with respect to usability, completeness and relevance for the quality assessment of feature requests?* and *[RQ4.3] What is the level of quality for feature requests in existing open source projects as measured by our framework?* we followed two different approaches.

Section "Interview Setup" describes the setup of an initial evaluation of our framework that consisted of interviewing eight practitioners. Section "Case Study Design" describes the setup of a case study in which we applied the framework to 620 feature requests from three open source projects.

Interview Setup

To explore the applicability (usability, completeness and relevance) of our framework for the quality assessment of feature requests the first author interviewed eight practitioners from the Dutch agile community. They were sourced through our personal network. We chose the agile community to get a first idea of whether the framework would also be useful for companies using JIT requirements. The participants are not necessarily experienced in open source projects, but are familiar with both the traditional up-front requirements engineering and the JIT requirements engineering. This makes them well-suited to comment on the underlying principles of our framework. The interview consisted of two parts:

1. General questions on JIT requirements quality, including an exercise to evaluate feature requests from the Firefox (`www.mozilla.org/firefox`) project;
2. An exercise to use our quality framework on feature requests from the Bugzilla project (`www.bugzilla.org`), followed by questions to rate the quality framework.

The first part of the interview was done with minimal introduction from our side and above all without showing the participants our framework. For the second part, the first author has turned the quality model into a checklist (in MS Excel) for the participants to fill in. For each check the answer set was limited. When each check is filled in, the spreadsheet automatically calculates a score for each of the quality criteria and an overall score for the quality of a single feature request (LOW/MEDIUM/HIGH), see Section "Scoring Setup" for the inner-workings.

The feature requests used for the exercise were manually selected by the first author using the following selection criteria: a substantial but not too big amount of comments (between 7 and 10) in the feature request, feature request has been implemented, contents of the feature request are not too technical (understandable for project outsiders). This last criterion is also why we selected the two projects: both Firefox and Bugzilla are well-known (types of) tools such that project outsiders should be able to understand or recognize the features. The feature requests were accessed on-line.

The data sets (five feature requests from Firefox, ten from Bugzilla), Excel

checklist and interview questions can be found on-line (Heck 2014).

Case Study Design

Our framework also allows us to get an insight into the quality of feature requests of existing open source projects. As such, we asked a group of 93 software engineering students to apply the checklist (as described in the previous section) to a large number of feature requests. As a side-effect we also get additional qualitative feedback on the practicality of the checklist.

The three open source projects that we used in this case study are:

- Eclipse MyLyn Tasks `projects.eclipse.org/projects/mylyn.tasks`: 100 feature requests selected out of around 400 total
- Tigris ArgoUML `argouml.tigris.org`: 210 feature requests selected out of around 1275 total
- Netbeans `netbeans.org`: 310 feature requests selected out of around 4450 total

The projects (see also Chapter 3) were selected because: 1) they are mature and still actively developed; 2) they differ in order of magnitude in terms of number of feature requests; 3) they use Java as a programming language (important because some feature requests contain source code fragments); 4) they use Bugzilla as an on-line tool to manage feature requests. The feature requests were selected randomly from among those with status "CLOSED" (since then we have the complete feature request history) and between 7 and 12 comments (because this yields a proper text size to analyze manually). For Mylyn Tasks, being a smaller project, we had to extend the criteria to status not equal to "NEW" and between 3 and 12 comments. All feature requests were accessed on-line in Bugzilla.

The application of the checklist was assigned to a group of 93 final-year computer science students majoring in software engineering at the Fontys Applied University in the Netherlands as part of one of their courses. During their studies they have gathered a proficiency in Java programming and worked with Eclipse, UML tools and Netbeans. Therefore we assume that they possess sufficient background knowledge to have a high-level understanding of the feature requests presented to them. Each student was assigned 20 specific feature requests. Furthermore, each feature request was assigned to 3 different students to be able to compare answers from different raters.

For the purpose of easy on-line data collection we have transformed our framework for feature requests into a Google Forms questionnaire. Some specific criteria were not enclosed in the questionnaire because the answer would always be the same for all feature requests (e.g. "does the feature request have a title?"), resulting from the fact that all analyzed feature requests are entered in the Bugzilla issue tracker that has mandatory fields. The criterion "Contradicting requirements" [QC3.1] was excluded from the questionnaire because it would be too hard for the

participants to check this (for that they would have to check all other feature requests). The remaining criteria were transformed into multiple choice questions with a short explanation for each question. We have included an additional comment box ("Opmerkingen" in Dutch) for the students to fill in any free-format remarks. So each student had to fill in 20 questionnaires, one questionnaire for each feature request assigned to them.

We received 1699 filled in questionnaires from the students through Google Forms: 83 students completed the assignment for 20 feature requests, 2 students did only 19 feature requests and 1 student did only 1 feature request. We corrected wrongly entered student numbers and feature request numbers (typos and use of network ID instead of student number). We also corrected small mistakes that four students reported by email (because students were not able to resubmit already sent questionnaires). We transformed this Excel file back into our original criteria list from the framework for feature requests by adding the criteria previously not enclosed (because of standard Bugzilla as explained in previous paragraph) and we added the scoring algorithm explained below. In that way we obtained 1699 'scorings' of the 620 feature requests. On average each feature request was scored 2.7 times; 10 feature requests were scored only once.

The list of feature requests used, the Google Form questionnaire and the resulting feature request scorings can be found on-line (Heck 2014).

Scoring Setup

In this section we explain the scoring model that was used.

In Table 4.6 and Table 4.7 it is indicated for each criterion what the outcome can be (column 'Metric'). For each specific criterion the answers are translated into percentage scorings. For a criterion with two possible answers the score is either 0% (low quality) or 100% (high quality). For a criterion with three possible answers there is also a 50% score (medium quality). As indicated in Table 4.6, QC2.2 directly results in a percentage (of relevant comments).

Higher-level scorings are calculated as indicated in Table 4.2. In particular, the overall score for [QC3] is calculated by taking the simple average of all percentage scores, because in our opinion no single criterion is more important for correctness than the other criteria. The final score first depends on the [QC1] score. If QC1.1 or QC1.2 score below 100% (meaning that basic or required elements are missing), the final score is always 0%. Otherwise, the final score is a weighted average of [QC1.3], [QC2.2] and [QC3]. QC3 has a weight of 3 in this average as we feel that the 'Correctness' is the most contributing factor to the overall quality of the feature request. [QC1.3] are 'optional elements', comments (QC2.2) are just a small factor for uniformity and [QC3] really looks at if everything that *has* been written is written in a correct way. The overall quality score is considered 'HIGH' when equal to or above 75%, 'LOW' when below 55% and 'MEDIUM' otherwise.

Table 4.2: Quality score calculation

Individual Scores	
Yes, Very much, N/A	100%
A little bit	50%
No, Not at all	0%
Overall Scores	
QC1.1, QC2.1	always 100% for open source feature requests
QC1.2, QC1.3	Average([QC1.x a] till [QC1.x c])
QC2.2	percentage of relevant comments
QC3	Average(QC3.1 till QC3.10)
TOTAL	IF ((QC1.1 < 100%) OR (QC1.2 < 100%)) THEN 0% ELSE $\frac{[QC1.3]+[QC2.2]+3*[QC3]}{5}$

This scoring algorithm is based on our professional opinion on what is appropriate for feature requests in open source projects. For other situations different rules or a (different) weighted average might be more appropriate.

4.5 Interview Results

As mentioned in Section 4.4 we first wanted to explore the applicability (usability, completeness and relevance) of our framework to the quality assessment of feature requests. We interviewed eight practitioners from the Dutch agile community. We summarize these interviews in what follows. [2].

Part One: Background and JIT Requirements Quality

All participants are experienced IT specialists with good knowledge of JIT requirements engineering. They work for five different Dutch companies in the area of software development and quality consulting; their roles in agile projects vary from coach, to trainer or consultant. Most of them also have experience as analyst or tester in agile projects. All participants mention user stories as a format for JIT requirements, but also use cases, features, and wireframes (i.e. screen blueprints). Some participants mentioned that they also consider informal communication as being part of 'the requirement'. We made clear that for the purpose of our framework we only consider the *written* part.

All participants agree that JIT requirements should fulfill certain quality criteria. This helps the understanding within the team or project and is important for traceability or accountability towards the rest of the organization. When asked for

[2]Complete interview transcripts in Dutch are available from the first author

a list of quality criteria the participants do not only mention quality criteria like the ones in our framework (SMART, not ambiguous, sufficiently small, atomic, following company standards/template), but also include process-oriented criteria like "approved by the product owner", "estimated in hours".

When asked to score 2 feature requests from the Firefox project (175232 and 407117) as HIGH/MEDIUM/LOW quality (without prior knowledge of our framework, just based on professional opinion), the participants do not always agree on the exact score, but they consistently score 175232 lower than 407117.

Part Two: Our JIT Quality Framework

Each participant filled in the checklist for at least two different feature requests from the Bugzilla project to get some hands-on experience with the checklist. The goal of this exercise was not to collect quantitative data, but to get qualitative feedback from the participants on the checklist.

Four participants mention "# of relevant comments" (QC2.2) and 2 participants mention "SMART" (QC3.5) as checks that are unclear or difficult to fill in. For [QC2.2] they find it difficult to determine if a comment is 'relevant' or not and for [QC3.5] they have difficulties determining the overall score on 5 criteria (Specific, Measurable, Acceptable, Realistic, Time-bound) in one check. We agree that these two checks are quite subjective, but we chose not to objectivize them in further detail. As one participant remarks: "I am in favor of checklists but quantifying in too much detail triggers discussions on scores and weighing. The discussion should be on the content of the requirement.". This is what we also conclude in our section 4.3 about Customization of the framework.

When asked to rate the score calculated by the Excel sheet for each feature request (LOW/MEDIUM/HIGH) the opinions vary. On a scale from 1 (no match at all with my personal opinion) to 5 (great match) all ratings have been given, although 6 out of 8 participants rate 3 or higher. This shows that most participants consider the final score of the model to be relevant. Yet, we also accept that our initial weighting scheme for the checklist requires fine-tuning for future use.

For example, in the checklist used in this interview a feature request always scores LOW if one of the basic (QC1.1) or required (QC1.2) elements is missing and not all participants agree with this choice. They for example argue that a feature request with a missing 'Rationale' (QC1.2b) can still be a correct feature request if it is self-explanatory enough. We agree. We added a scoring algorithm to help the participants in judging feature request quality, but the scoring algorithm should not be taken as an absolute judgement (one participant: "A practical checklist like this always helps, but I am not sure how useful it is to calculate a final score from the individual checks."). As stated before the checklist is very useful as a reminder of what to check when looking for good feature request quality. It is the reviewer or author of the requirement that can still decide how serious a violation is in the given situation, e.g. by marking it as 'N/A'.

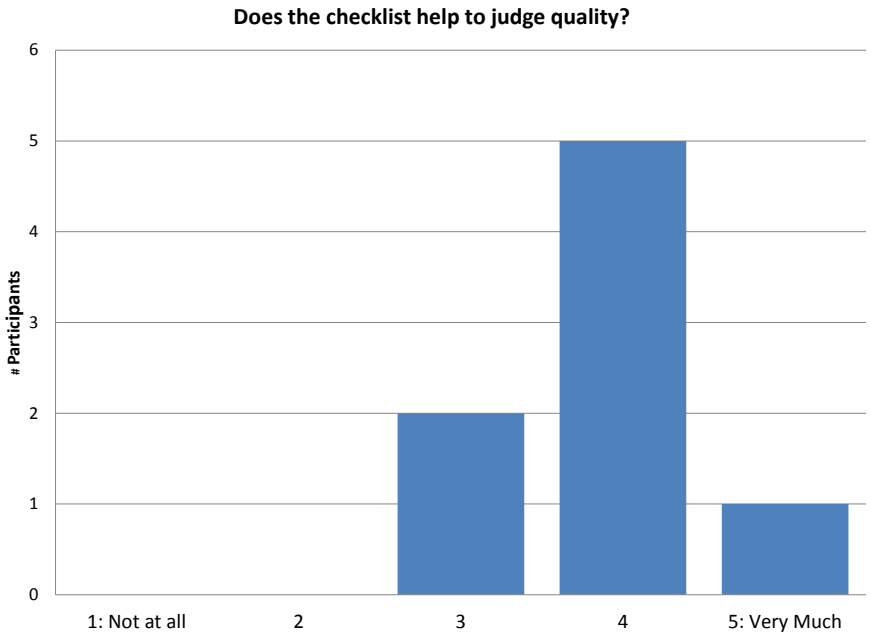

Figure 4.3: Checklist rating by interview participants

Some participants answered that they would like to add topics such as: non-functional impact (e.g. usability, performance), business value, domain models. We see this as valuable suggestions for practitioners customizing the checklist for their JIT projects. We feel that these topics are not applicable for feature requests in open source projects. As one participant mentions "It is refreshing that this checklist is tailored for this specific situation. The ultimate result would be to know how to construct such a tailored checklist.". Section 4.3 shows how this customization could be done.

But why would teams or projects do the effort of including such a checklist in their development process? All participants rated the checklist as helpful when judging the quality of a feature request (compared to using 'gut-feeling'), see Figure 4.3. They valued the help of the checklist to not forget criteria, to base their opinion on facts, to use it as an education for new team or project members, to standardize the review process. One participant (that rated the checklist as 'Very Much' helpful) nuances this by stating "It is not always the case that high-quality requirements lead to high-quality products. The checklist is helpful but just a small part of all factors that influence final product quality.". This is a valid point. Our study shows that also in JIT environments requirements quality is considered important, but that there is no one-size-fits-all solution. All participants confirmed that our framework is a good starting point to get to a tailored process for quality assessment of JIT requirements.

4.6 Case Study Results: Findings on Quality of Feature Requests

To collect experiences from the use of our framework in practice we applied it to 620 feature requests from the Netbeans, ArgoUML and Mylyn Tasks open source projects. For details on this setup see Section 4.4. In this section we highlight the aggregated findings. First we explain how we cleaned the data. Then we present some aggregated findings on feature request quality. Lastly we describe what we learned about our framework for feature requests.

Data Cleaning

When collecting data through surveys, data quality can be a concern. In particular under conditions of obligatory participation, so-called careless responses can be a worry (Meade and Craig 2012). To this end, we performed two checks on the response to identify suspect participants. These suspect participants could either not have taken their participation in the survey seriously, or they might lack a basic understanding. The two checks are:

1. We used criterion [QC3.9] (No duplicate requests) as a 'control question'. The answer to this question is not subjective (each feature request in Bugzilla is clearly marked as "CLOSED DUPLICATE"). We remove all 17 participants that have answered this question wrongly in 1 or more cases.
2. For each participant we calculated the absolute distance of his questionnaires to the average questionnaires for the same feature requests. The resulted in absolute distances between 73 and 308 (where maximum distance is 340 if a participant always answers completely the opposite of the other participants that have scored the same feature request) with an average distance of 136. We remove the 5 outlier participants (that have a distance of more than twice the standard deviation from the average, i.e. more than 214) and note that 3 out of these 5 participants are also included in the 17 removed participants from the previous check.

In the aggregated analysis in the next paragraph we present the results after removing all of the suspect participants (19 in total). This might be more than needed for the purpose of our analysis, but makes us more confident that we are working with valid data. This leaves 67 participants filling in 1319 questionnaires for 570 feature requests (200 ArgoUML, 80 Mylyn Tasks, 290 Netbeans). Out of these 90 feature requests have been scored by only 1 participant. On average each feature request is scored by 2.3 participants.

Aggregated Results

The aggregated answers and resulting scorings (see Section 4.4) can be found in Table 4.3. A lot of interesting observations can be made from this table. We highlight

a few observations in this section:

Overall Score The overall score (= the average score of all feature requests in the project) for the three projects is quite low (5% for Netbeans and ArgoUML, 15% for Mylyn Tasks). This is mainly due to the fact that a lot of feature requests score 0% because of missing keywords, missing rationale or missing link to source code. Note that the QC3 "correctness" score is quite high for each of the three projects. This led us to conclude that the way we calculate the overall score (scoring 0% if basic or required elements are missing) does not provide particular insight into overall feature request quality.

Completeness As indicated in the previous observation not all required elements are present: on average 54% (QC1.2). For optional elements on average 36% is present (QC1.3). For the individual elements we see that for example the use of mockups and keywords is not so common in the three projects. Mylyn Tasks does clearly better then the other two projects here. This explains the higher overall quality score of 15% for Mylyn Tasks.

Uniformity The percentage of relevant comments (QC2.2) is similar for all three projects: around 60%. This means that out of every three comments (in total more than 4500 comments were read by the participants) one is irrelevant. We assume this has implications for the understandability of the feature request.

Creation-time The feature requests that have been selected are all in status "Closed". This means that the QC3 overall scorings in Table 4.3 are based on both the creation-time (*C) and the just-in-time (*J) criteria. When we only average the seven creation-time criteria for QC3 we see a clear difference in scoring:

	Mylyn Tasks	**ArgoUML**	**Netbeans**	**TOTAL**
QC3 *C	83%	80%	84%	**82%**
QC3 *C + *J	80%	75%	78%	**77%**

This shows that the quality of the feature requests slightly deteriorates for all three projects from the moment it is created until it is closed. In the projects we analyzed this is mainly due to the fact that according to our framework a feature request does not need to be SMART (QC3.5 *J) when initially created. However, a lot of feature requests (on average 46%) have been scored 'not SMART at all', leading to a lower just-in-time quality score.

Our overall conclusion from the data gathered is that although sometimes important elements are missing (QC1) and comments are not always relevant (QC2) in fact the overall correctness/consistency of the feature request is quite high (based on [QC3] scorings).

Feedback on the Use of the Framework for Feature Requests

The participants provided us additional insights as well: they submitted remarks by email or in the comment field of the questionnaire. The most important things we learned about our framework from these remarks are:

Table 4.3: Scorings from open source projects (NB = Netbeans, AU = ArgoUML, MT = Mylyn Tasks)

Specific Criterion	Answer	MT	AU	NB	Total
QC1.1 Basic elements	Average score	**100%**	**100%**	**100%**	**100%**
QC1.2a Keywords	Not at all	53%	87%	88%	**82%**
	A little bit	44%	11%	11%	**18%**
	Very much	3%	1%	0%	**1%**
QC1.2b Rationale	No	32%	33%	30%	**31%**
	Yes	68%	67%	70%	**69%**
QC1.2c Link to source code	N/A	19%	17%	19%	**18%**
	No	41%	13%	13%	**18%**
	Yes	39%	70%	68%	**64%**
QC1.2 Required elements	Average score	**51%**	**54%**	**54%**	**54%**
QC1.3a Scenario	No	59%	61%	47%	**54%**
	Yes	41%	39%	53%	**46%**
QC1.3b Mockup	No	77%	90%	94%	**90%**
	Yes	23%	10%	6%	**10%**
QC1.3c Solution	No	46%	48%	47%	**47%**
	Yes	54%	52%	53%	**53%**
QC1.3 Optional elements	Average score	**39%**	**34%**	**37%**	**36%**
QC2.1 Use of tool	Average score	**100%**	**100%**	**100%**	**100%**
QC2.2 Relevant comments	% relevant	**66%**	**61%**	**60%**	**61%**
QC3.1 Contradicting requirements	(Not scored)	N/A	N/A	N/A	N/A
QC3.2 No Contradicting comments	Not at all	0%	2%	4%	**3%**
	A little bit	17%	22%	26%	**23%**
	Very much	82%	76%	70%	**74%**
QC3.3 Correct language	Not at all	5%	6%	6%	**6%**
	A little bit	27%	24%	29%	**27%**
	Very much	68%	70%	65%	**67%**
QC3.4 Problem stated	No	24%	23%	18%	**21%**
	Yes	76%	77%	82%	**79%**
QC3.5 SMART	Not at all	33%	55%	45%	**46%**
	A little bit	52%	37%	46%	**44%**
	Very much	15%	8%	9%	**9%**
QC3.6 Title correct	No	16%	30%	16%	**21%**
	Yes	84%	70%	84%	**79%**
QC3.7 Atomic	No	14%	10%	8%	**10%**
	Yes	86%	90%	92%	**90%**
QC3.8 Clear terms	Not at all	3%	6%	5%	**5%**
	A little bit	31%	30%	30%	**30%**
	Very much	66%	64%	65%	**65%**
QC3.9 No duplicate	No	12%	28%	20%	**22%**
	Yes	88%	72%	80%	**78%**
QC3.10 Links clear	No	13%	11%	9%	**10%**
	Yes	87%	89%	91%	**90%**
QC3 Consistency and correctness	Average score	**80%**	**75%**	**78%**	**77%**
TOTAL	Average score	**15%**	**5%**	**5%**	**7%**

- Five participants are confused about the term “Title" we used in the questionnaire for [QC3.6]. This is not a problem of our framework itself, but we should have better explained in the questionnaire that by “Title" we meant the statement in bold just after the identifier of the feature request (see Figure 4.2) or for the ArgoUML project the field called “Summary”.
- Five participants comment that they consider the feature request they evaluated to be a bug report. This misclassification is a well-known phenomenon. According to Herzig et al. (2012) only 3% of feature requests is misclassified. This means that our results are not greatly influenced by this.
- Three participants were confused about the question about keywords (QC1.2a) because the ArgoUML project does not have this field in the feature request. We could have mentioned this to the participants beforehand. This is again not a problem of our framework.
- We learned that some questions are more subjective than others, see Table 4.4. For example on the question about mockups (QC1.3b) all students agree in 90% of the cases. On the question if the feature request is SMART all students agree in only 49% of the cases. As explained in Section 4.3 this subjectiveness is not a problem, because we are usually not interested in absolute scorings but in finding shortcomings in the JIT requirements. Even if only one person thinks there is a shortcoming, it might be worth to look into the details of it.

The remarks could all be avoided by a more detailed explanation of the structure of the different feature requests in Bugzilla and by a more detailed explanation of the questions in the questionnaire. We did organize a general meeting for this explanation, but participants that were not present during that meeting had to rely on the explanation in the questionnaire itself. Overall the remarks from the participants did not make us change our underlying framework or quality criteria.

4.7 Discussion

In this section we highlight some recommendations for practitioners, based on our findings. Subsequently we revisit the research questions and discuss limitations of our research approach.

Practical Consequences for Practitioners

Looking at the data we collected from the three open source projects, we come to a number of recommendations for practitioners authoring feature requests. When we look at the lowest scoring quality criteria in Table 4.3, these are the top-5 to focus on for improving feature request quality:

1. Enter keywords/tags for each feature request, making them more easily retrievable for future reference. This will likely at the same time help to reduce the number of duplicate feature requests.

Table 4.4: Subjectivity scores per question

Question \ Unanimity	2 students agree	All students agree
[QC1.2a] Keywords	94%	84%
[QC1.2b] Rationale	85%	67%
[QC1.2c] Codelink	81%	62%
[QC1.3a] Scenario	83%	59%
[QC1.3b] Mockup	95%	90%
[QC1.3c] Solution	82%	58%
[QC2.2] Relev. comm.	20%	20%
[QC3.2] Contr. comm.	79%	58%
[QC3.3] Corr. lang.	72%	52%
[QC3.4] Problem	88%	72%
[QC3.5] SMART	73%	49%
[QC3.6] Summary	85%	68%
[QC3.7] Atomic	93%	85%
[QC3.8] Terms	73%	50%
[QC3.9] Duplicate	97%	95%
[QC3.10] Corr. Links	90%	81%
Total	**81%**	**65%**

2. Indicate the problem that needs to be solved and/or include a rationale for the feature request (why is this feature needed?). This will help developers to better understand the feature request and thus will increase the chance of the solution matching the actual need of the feature request author.
3. Further increase the understandability of the feature request by adding one or more of additional items: screen mockups, descriptions of use case scenarios, or possible solutions.
4. Be as precise as possible. There is no need to fully specify all aspects at creation time of the feature request, but what is written should not be unnecessarily vague or ambiguous (e.g. avoid the use of abbreviations or terms that are not quantified). Being precise from the start avoids wasting time on discussions to clarify statements during development.
5. Avoid irrelevant comments. This clutters the discussion on the feature request and thus hinders its correct implementation. Even better would be if the issue tracker used has a way to 'categorize' or 'hide' comments, for easy retrieval of the relevant ones for the task at hand. We did not see such an option in the projects we considered.

Research Questions

In this section we revisit the research questions one by one.

[RQ4.1] We started by asking: *which quality criteria for informal verification apply to feature requests?* We have developed a framework for quality criteria for JIT requirements based on earlier work on traditional upfront requirements, our experience with feature requests in open source projects and analysis of literature on just-in-time requirements. We have instantiated this framework for feature requests in open source projects.

How does our framework compare to quality criteria for traditional requirements? To highlight the differences with traditional requirements, we compare it to the list of Davis et al. (1993). Davis et al. performed a thorough analysis of qualities of a software requirements specification (SRS, an up-front document). Table 4.5 shows that four quality criteria apply to traditional requirements, but not to just-in-time requirements (we have marked them as 'N/A' in Table 4.5):

1. *Externally consistent = no requirement conflicts with already baselined project documentation.* The provisionments (see Section 4.2) are not specified with respect to external documents (Davis et al. define this as 'already baselined project documentation'), but with respect to existing systems. If in specific situations external documents are relevant, the team or project should add one or more criteria to [QC3] to check the consistency.
2. *Executable = there exists a software tool capable of inputting the SRS and providing a dynamic behavioral model.* In open source (JIT) projects this is not done by up-front extensive specification, but by prototyping or frequent releases.
3. *Annotated by relative stability = a reader can easily determine which requirements are most likely to change.* Open source (JIT) projects have embodied change as a known fact. They solve this with short iterations and reprioritization of requirements for each iteration. That is why in open source projects we do not need a special attribute to specify change-proneness up-front.
4. *Reusable = sentence, paragraphs and sections can easily be adopted or adapted for use in a subsequent SRS.* Since open source feature requests are necessarily incomplete ('provisionments'), it makes no sense to reuse them.

All criteria from our resulting framework, see Tables 4.6 and 4.7, are in one way or another present in the work of Davis et al. (1993), see Table 4.5. However, we have adjusted the description of each criterion to JIT feature requests, e.g. for the criterion 'Design-independent' Davis et al. explain that a maximum number of designs should exist to satisfy user needs. This means that according to the definition of Davis et al. the requirement should just describe the problem and not already present a design solution because that would decrease the number of possible designs. We have included '[QC3.4] - Specify Problem' but we specifically

Table 4.5: Mapping between Davis et al. (1993) and our framework

Davis et al. (1993)	JIT Requirements Quality Framework
Unambiguous	[QC2.4], [QC3.3], [QC3.8]
Complete	[QC1]
Correct (contributes to satisfaction of some need)	[QC3.4], [QC1.2] - Rationale
Understandable	[QC1.3], [QC3.3]
Verifiable	[QC3.5]
Internally consistent	[QC3.1], [QC3.2], [QC3.6]
Externally consistent	N/A
Achievable	[QC3.5]
Concise	[QC2.2]
Design independent	[QC3.4] (Solution might be included)
Traceable (facilitates referencing of individual req.)	[QC3.7], [IQ1]
Modifiable (table of contents and index)	[IQ5]
Electronically stored	[QC2.1]
Executable (dynamic behavioral model can be made)	N/A
Annotated by relative importance	[QC1.1] - Relative importance
Annotated by relative stability	N/A
Annotated by version	[QC1.1] - Version
Not redundant	[QC3.9]
At right level of detail	[QC1]
Precise	[QC3.5]
Reusable	N/A
Traced (clear origin)	[QC1.2] - Link to code, [IQ2], [IQ3], [IQ4]
Organized	[QC1.2] - Keywords, [QC3.10], [IQ6]
Cross-referenced	[QC3.1], [QC3.9], [QC3.10]

allow the user to *also* specify design solutions (QC1.3c), as this is common practice in open source projects where users are also developers.

Did we introduce any new criteria for just-in-time requirements? If we do the comparison with the list of Davis et al. (1993) the other way around we also see a few differences:

1. *Complete* Davis et al. refer to the completeness of the set of requirements (before development starts). Since this is not a goal for just-in-time requirements engineering, we use completeness in the sense of "are all basic/required/optional elements of one single requirement present" (QC1.x).
2. *Concise* For open source feature requests comments are added to the original requirements. We have translated the conciseness of a requirement as de-

scribed by Davis et al. to the demand that the comments that are added are concise (QC2.2).

3. *Internally Consistent* In a similar manner we have added a demand for internal consistency of the comments that are added to one single requirement (QC3.2). And we also added a demand for consistency between the feature request and its summary title (QC3.6).
4. *Unambiguous* Instead of this one single criterion from Davis et al. we have three related criteria: use of correct language (QC3.3), use of a glossary (QC3.8) and use of uniform attachments (QC2.4).
5. *SMART/INVEST (QC3.5)* Some of the separate components of SMART and INVEST have been mentioned by Davis et al. (Verifiable = Testable, Achievable = Realistic, Precise = Specific), but most of them are new in our model. INVEST is specific for user stories, but in our opinion it can also be used for other types of just-in-time requirements.
6. *Atomic (QC3.7)* The demand that a single requirement should be atomic is not mentioned explicitly by Davis et al., but of course it helps to make requirements traceable.
7. *Cross-referenced* We have translated the cross-referenced criterion of Davis et al. into two separate criteria about 'linked duplicates' (QC3.9) and 'navigable links' (QC3.10)

For our user stories customization (see Section 4.3) we added the aforementioned '[QC2.4] - uniform attachments' and one more criterion:

8. '[QC2.3] - Follow template' ("As a <role>, I can <activity>so that <business value>" (Leffingwell 2011)). This is unique for user stories since traditional requirements and feature request do not follow standard templates (methods that advocate this are not widely used).

Overall, our analysis of literature in just-in-time requirements and our experience with just-in-time requirements led to 8 instances of 'additional criteria' or 'new' interpretations of existing criteria. And of course we have added the time dimension to each of the criteria by specifying creation-time (*C) or just-in-time (*J).

[RQ4.2] Our second question was *How do practitioners value our list of quality criteria with respect to usability, completeness and relevance for the quality assessment of feature requests?*

The overall evaluation of the framework for open source feature requests was positive. The interviews with practitioners have made it clear to us that specific situations need some fine-tuning of the specific criteria and the scoring. Our framework caters for that kind of specific tailoring and we have given some hints om how to approach this. The questionnaires filled in by the practitioners only resulted in some minor remarks. We concluded that all of them could be solved by better ex-

plaining the questions. The feedback from practitioners did not make us change our basic framework.

Although we have used open source feature requests to evaluate the framework, we see the value of our framework not so much for open source projects to apply it, because in open source projects it is hard to make the whole community comply to quality standards. The interviews with the practitioners indicated to us that a translation of the quality criteria to specific JIT industry settings is both feasible and useful. The practitioners valued our framework as a structured approach for doing this.

[RQ4.3] Finally, we answered *What is the level of quality for feature requests in existing open source projects as measured by our framework?*. We presented a table with average scorings for each of the quality criteria. These scorings were collected from over 550 feature requests that were rated using our framework. The overall impression is that the score for [QC3] is quite high (77%) , but there is more room for improvement in the scorings for [QC1.2] (54%), [QC1.3] (36%) and [QC2.2] (61%). We also concluded that the quality at creation-time (*C, 82%) was slightly better than at the time of scoring (*J, 77%). From our results we have distilled a set of recommendations that makes this research actionable for practitioners.

Limitations

In this section, we discuss the threats that can affect the validity of our study and show how we mitigated them.

Internal validity regards threats inherent to our study. We assume that the results gathered in our case study with the 86 final-year software students are reliable because any negative effects from participants not understanding or not cooperating would average out over such a large number of entries (randomly selected and randomly divided over the software engineers). In order to minimize the risk of data pollution we cleaned the data (see Section 4.6) and we have investigated how unanimous the answers from the students are (see Table 4.4).

With regard to the interview setup that we have detailed in Section 4.5 it might be that the participants were influenced by how we approached them or how we explained our framework to them, the so-called *observer-expectancy effect*. We tried to be as neutral as possible towards the interviewees and clearly explained them that we were expecting them to provide honest feedback, which would be most beneficial for our investigation.

Also note that for the scorings of the feature requests in open source projects we assume that all communication around the feature request is logged in the issue tracker. We might have missed some data related to the feature requests that was documented e.g. on mailing lists or discussion forums. However, if we missed the data, every reader of the feature requests would have missed it, since there is no

link to it in the feature request. So from a quality assessment perspective it makes sense to only look at the data in the issue tracker and judge the feature request quality solely based on that.

Construct validity concerns errors caused by the way we collect data. A possible issue is that the criteria list that we deduced for open source feature requests does not give a good representation of the quality of those feature requests. This would influence the observed quality of open source feature requests in Table 4.3. While we acknowledge that we still need to further investigate our criteria list, we also want to stress that the industry participants acknowledge that the criteria are relevant.

Although all industry participants acknowledge that the criteria are relevant we did ask them to judge the criteria for open source feature requests. These open source feature requests might be different from what they are used to in their industry projects. As such an important step in future work would be to repeat the analysis with open source developers or with industry requirements.

External validity threats concern the generalizability of our results. In the interview that we describe in Section 4.5 we only have 8 participants from Dutch industry. As such, a possible threat to validity is that other participants might have a different opinion on the usefulness of our framework. However, the participants are from sufficiently different companies and backgrounds to get a first overall impression of community feedback. Although all participants come from the Netherlands we have based our framework on international literature and experiences with international (open source) projects. All participants in the interviews were well aware of international literature and international best practices in agile development and using Scrum (Schwaber and Beedle 2001) as a development approach. With all participants originating from a single country we cannot exclude cultural bias. Case studies with participants from different countries are called for in future work.

Similarly, for the cases study that we describe in Section 4.6, we have 86 final-year software engineering students participating. A possible threat here is that all students have a similar background (they all study at Fontys University of Applied Sciences) and that another group of students with another background might react differently. However, we plan for replication studies in future work.

We can also not be sure that we would have had the same results if we would have involved practitioners from industry or open source. To mitigate this we used final-year students, who are close to becoming practitioners themselves. We also found several papers indicating that students can be good proxies for practitioners, see e.g. (Salman et al. 2015).

We have investigated three open source projects in the case study. We cannot be sure that the results presented in Section 4.6 can be generalized to more open source projects. In order to try to mitigate this threat, we have selected the three

projects to be sufficiently different in size and domain.

In this chapter we focus on *open source* feature requests because of their public availability. This means we cannot be sure that feature requests that are created as part of a closed source project adhere to the same standards. However, according to Alspaugh and Scacchi (2013) "closed source software bug reports and feature requests and the process for managing them look much like those for open source software", so we expect our results to hold in both cases.

4.8 Related Work

For JIT (mostly from the area of agile development processes) and open source projects there is a body of work both on Requirements Engineering (Grau et al. 2014; Noll and Liu 2010; Paetsch et al. 2003) and Quality Assurance (Aberdour 2007; Huo et al. 2004). In this section we discuss the few papers that specifically mention quality criteria for JIT or open source requirements.

Duncan (2001) analyses the quality attributes for requirements in Extreme Programming (XP, one of the agile methods). He does this by comparing *stories* (the main requirements artefact in XP) to the quality attributes presented by Davis et al. (1993). This is the same approach that we followed in our Discussion, see section 4.7.

Dietze (2005) describes the agile requirements definition processes performed in open source software (OSS) development. Quality aspects are not part of his analysis. However he does mention meta-data of a change request corresponding to our [QC1].

Scacchi (2009) argues that requirements artifacts in open source software development might be assessed in terms of virtues like 1) encouragement of community building; 2) freedom of expression; 3) readability and ease of navigation; 4) and implicit versus explicit structures for organizing, storing and sharing. Virtue 3) and 4) above are covered in our framework, whereas virtue 1) and 2) should be achieved by a correct setup and management of the open source project.

Bettenburg et al. (2008a) conducted a survey in open source projects on what makes a good bug report, revealing a mismatch between what developers need and what users supply. Most developers consider steps to reproduce, stack traces, and test cases as helpful, which are at the same time difficult to provide for users. These three items are somewhat similar to the scenario's and screens we have included in [QC1.3].

Génova et al. (2013) describe a framework to measure the quality of textual requirements. They have defined metrics and implemented those metrics in a tool to automatically verify the quality. A requirement is scored as bad, medium or good. The tool is commercially available and users report benefiting from using it. They use formal requirements documents as input data, making some of their quality criteria (such as completeness - covering all user needs - and traceability - explicit

relationship with e.g. design documents) less relevant for JIT requirements. See also our analysis of the earlier work of Davis et al. (1993) in Section 4.7. We do however value the idea of automating certain quality checks as was also requested by one of the participants. This is something we plan to do in the future.

4.9 Conclusion

Summary In this chapter we have developed a framework for the quality assessment of JIT requirements that caters for tailoring to different situations. We have instantiated this framework for feature requests (in open source projects) and indicated how to do this for other situations. The framework is based on literature on the quality of traditional requirements as well as literature on JIT requirements and open source projects. The framework structures quality criteria and includes a time dimension: some quality criteria should hold at Creation-Time (*C) and others should hold Just-in-Time (*J) for implementation of the JIT requirement.

We have performed an evaluation with practitioners. The framework was positively evaluated by all of them. They also confirm our assumption that quality assessment of JIT requirements is important, the same as for traditional up-front requirements. We have also quantitatively evaluated the framework with 86 software engineering students. This resulted in even more confidence in the usability, completeness and relevance of the framework.

Next to that we were able to present quantitative results for the feature request quality in three open source projects:

- A lot of feature requests score “LOW” on overall quality because of missing keywords, missing rationale or missing link to source code.
- The percentage of relevant comments is surprisingly low: around 60%.
- The average “correctness” score is quite high: 77%
- The correctness of the feature requests slightly deteriorates for all three projects from the moment it is created until it is closed (from 82 to 77 % correctness).

Our analysis of these quantitative results led to a top-5 of recommendations for practitioners to improve feature request quality.

Key Contributions This work makes the following key contributions:

1. a framework for quality criteria for JIT requirements;
2. instructions for practitioners how to customize the quality criteria to their specific JIT environment;
3. an instantiation of the framework for feature requests in open source projects;
4. findings on feature request quality from three open source projects;
5. recommendations for practitioners on improving feature request quality.

In this chapter we have demonstrated the value of our framework in the following three ways: 1) we have used it to give an overview of quality criteria that are applicable to feature requests (at creation-time or just-in-time); 2) we have indicated how it can serve as a basis for teams or projects that need to assess the quality of their JIT requirements; 3) we have used it to get an insight into the quality of feature requests in open source projects.

Future Work In future work we will extend our analysis of open source feature requests to confirm the findings from the case study with the student participants and to get a more detailed view on feature request quality. Another useful addition for future work is the automation of some of the checks to increase the usability of the checklist. We also plan to validate our framework in case studies in closed source and/or agile JIT environments.

In this chapter we have conducted interviews and a case study. To answer future research questions we would like to gather more data suitable for statistical analysis by using controlled experiments. Our idea is to investigate related topics that require more quantitative evidence such as the relationship between feature request quality and software product quality (design, tests, code, defects) or between feature request quality and productivity.

Table 4.6: JIT quality framework for feature requests - [QC1] and [QC2]

ID	CRITERION	DESCRIPTION	METRIC	*C/*J
[QC1] Completeness				
QC1.1 Basic Elements				
a	Summary and Description	The 'Description' contains the provisionment and the 'Summary' (or 'Title') gives a clear short version of the provisionment	yes/no	*C
b	Product Version	Indicates for which software product and version the provisionment holds	yes/no	*C
c	Relative importance	The relative importance of the feature request should be clear. Examples of this are a 'Priority' or 'Severity' field or a voting mechanism (feature requests with more votes are more important)	yes/no	*J
QC1.2 Required Elements				
a	Keywords / tags	For organization purposes (easily finding related requests) a feature request should be tagged with keywords	yes/no	*J
b	Rationale	Each feature request has a justification. The author should specify why this feature request is important for him/her. This helps the developers in deciding on the priority of the feature request	yes/no	*J
c	Link to source code for fixed requirement	For solved feature requests indicates in which version of the source code it has been solved. This can be done through a manual comment, or through an automated one generated from the source code management system. Ideally the feature request has a separate field to track this.	yes/no	*J
QC1.3 Optional Elements				
a	Use case or Scenario	The author specifies the exact steps that he/she is missing in the current version of the software. What is the trigger for the missing functionality and what are scenario's in which the functionality would be useful?	yes/no	*J
b	Screens	The author could clarify the screens in the existing system that he/she wants to be changed by adding screen shots of the current situation and/or screen mock-ups of the desired situation.	yes/no	*J
c	Possible solution	The author could add a complete solution as 'Attachment' (patch), but could also specify hints for a possible solution in comments	yes/no	*J
[QC2] Uniformity				
QC 2.1	Issue tracker or other tool should be used	An issue tracker ensures that feature requests are stored in an uniform way, at least with respect to the attributes that are present for the feature request. Of course it remains up to the author to correctly fill those fields	yes/no	*C
2.2	All comments are necessary	All comments are necessary to understand the evolution of the feature request. The addition of irrelevant comments (e.g. thank you notes or instructions how to behave in the project) makes it more difficult for people to understand the totality of it	The percentage of relevant comments	*C

Table 4.7: JIT quality framework for feature requests - [QC3]

ID	CRITERION	DESCRIPTION	METRIC	*C/*J
[QC3] Consistency and Correctness				
QC 3.1	No contradicting feature requests	It is difficult to completely avoid conflicting requests, but they should not go unnoticed. A link can be made through comments and one of the feature requests should be 'Closed' to avoid implementation of the wrong request	yes/no	*J
3.2	No contradicting comments	Each contradicting comment is clarified in later comments. It should be clear what the correct interpretation of contradicting comments is	Very much, a little bit, not at all	*J
3.3	Correct language	Feature requests should be written in full sentences without hindering spelling or typing errors	Very much, a little bit, not at all	*C
3.4	Specify problem	Feature requests may include (hints for) a solution, but should always describe the problem that needs to be solved. This helps developers to think about alternative solutions	yes/no	*C
3.5	SMART	A feature request can be more quickly and easily resolved if it is Specific, Measurable, Acceptable, Realistic and Time-bounded (SMART, see (Doran 1981))	Very much, a little bit, not at all	*J
3.6	Correct summary	The summary should be a brief statement of the needed feature. It should be clear from the summary what the feature request is about. The description should give added value to the summary.	yes/no	*C
3.7	Atomic	Each feature request should contain only one requirement	yes/no	*C
3.8	Glossary	Each unclear term or abbreviation should be explained in the feature request or in a separate 'glossary'	Very much, a little bit, not at all	*C
3.9	No duplicate requests	Due to the nature of open source projects there will always be some duplicates (also users not so familiar with the project get the rights to enter requests). At least the duplicates should be identified, properly linked, and the master should be indicated.	yes/no	*C
3.10	Navigable links	Links to other feature requests should be navigable (by clicking on the link) and it should be explained what the type of the link is	yes/no	*C

5

A Systematic Literature Review on Quality Criteria for Agile Requirements Specifications

The quality of requirements is typically considered as an important factor for the quality of the end product. For traditional up-front requirements specifications a number of standards have been defined on what constitutes good quality: requirements should be complete, unambiguous, specific, time-bounded, consistent, etc. For agile requirements specifications no new standards have been defined yet and it is not clear yet whether traditional quality criteria still apply. To investigate what quality criteria for assessing the correctness of written agile requirements exist, we have conducted a systematic literature review. The review resulted in a list of 16 selected papers on this topic. These selected papers describe 28 different quality criteria for agile requirements specifications. We categorize and analyze these criteria and compare them with those from traditional requirements engineering. We discuss findings from the 16 papers in the form of recommendations for practitioners on quality assessment of agile requirements. At the same time we indicate the open points in the form of a research agenda for researchers working on this topic. [1]

Requirements engineering has been perceived as one of the key steps in successful software development since the early days of software engineering (Sillitti and

[1] This chapter is submitted to the *Software Quality Journal (SQJ)* (Heck and Zaidman 2016).

Succi 2005). Sillitti and Succi mention several standards for requirements elicitation and management, such as IEEE-830 Recommended Practice for Software Requirements Specification (IEEE-830 1998), that have been developed for traditional requirements engineering. They claim that agile methods do not rely on standards. Inayat et al. (2014) observe that agile requirements engineering solves the initial 'vagueness' of agile requirements not by documenting according to standards, but by e.g., face-to-face communication or prototyping. On the other hand they remark that the practice of less documentation poses a challenge in many situations (e.g., distributed teams, large teams, complex projects) that require documented agile requirements that are fully elaborated in writing. As soon as agile teams cannot rely on face-to-face communication the 'correctness' of written documentation becomes more and more important.

Standards such as IEEE-830 (IEEE-830 1998) define criteria for this correctness: requirement specifications should be complete, unambiguous, specific, time-bounded, consistent, etc. However, this standard focuses on traditional up-front requirements specifications. These are requirements sets that are completely specified (and used as a contract) before the start of design and development. As said, agile methods do not tend to follow such standards. However both research (Eberlein and Leite 2002; Inayat et al. 2014) and findings in practice (see e.g., Heck and Zaidman (2014c) and the empirical study of Kassab (2014)) suggest that quality of requirements specifications is also an important topic for agile requirements engineering.

In previous work we have developed a framework for quality criteria for agile requirements specifications (Heck and Zaidman 2014c). This resulted in a list of possible quality criteria for e.g. agile user stories. Working on this list we realized that we could not find any updated standards or best practices for agile requirements specifications, nor an extensive overview of quality criteria for agile requirements specifications. At the same time we felt the need to validate our list of quality criteria with what others have published on this topic. In our previous work (Heck and Zaidman 2014c) we focus on open source feature requests and thus only included related work in the area of open source development. This is what made us see the necessity of conducting a systematic literature review (Kitchenham and Charters 2007) to make an inventory of quality criteria for agile requirements specifications that have been mentioned in literature. The result of such a literature review would be that we have a more thorough analysis of which of the traditional criteria are still applicable and which new quality criteria have been presented for agile requirements specifications. Furthermore, we take this opportunity to discuss the found literature from the viewpoint of both practitioners and researchers.

This leads us to the driving research question for our systematic literature review:

[RQ5] Which are the known quality criteria for agile requirements specifications?

Note that we focus on verification (have the requirements been written in a correct way) and not on validation (do the requirements correctly reflect the need of the user). Moreover we consider formal verification methods (that use mathematical models to derive specification correctness) out of our scope, as we assume that not all agile teams would have the skills to apply such formal methods.

The remainder of this chapter is structured as follows. Section 5.1 introduces some background and related work. Section 5.2 details the selection process that we followed, while Section 5.3 presents the classification of the resulting papers according to different dimensions. Section 5.4 summarizes the quality criteria for agile requirements specifications that are mentioned in the selected papers. Sections 5.5 and 5.6 include recommendations for respectively practitioners and researchers in the area of quality of agile requirements specifications. Section 5.7 discusses our findings and Section 5.8 concludes the chapter.

5.1 Background and Related Work

This section sketches a short background on the role of requirements in agile development and introduces some related work. The absence of papers with quality of agile requirements as their main topic is what motivated us to conduct our research in the first place.

Background on Agile Requirements Agile development is a collective name for a family of software development processes that follow the so-called 'Agile Manifesto': "Individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; responding to change over following a plan" (Beck et al. 2001). The main implications of this manifesto for agile requirements engineering are (Sillitti and Succi 2005; Inayat et al. 2014): software is developed incrementally with requirements being detailed and prioritized just before every iteration, requirements documentation is reduced in favor of face-to-face communication and prototyping. Grau et al. (2014) characterize agile requirements engineering as "collaborative, just enough, just in time, sustainable". Ernst and Murphy (2012) use the term 'just-in-time requirements' (JIT requirements) for this. They observed that requirements are "initially sketched out with simple natural language statements", only to be fully elaborated (not necessarily in written form) when being developed. In this chapter we use the term 'agile requirements', because this is the most widely used.

Related Work on Agile, Requirements and Quality For the area of agile development processes there is a body of work on either Requirements Engineering (Grau et al. 2014; Paetsch et al. 2003; Ramesh et al. 2010) or Quality Assurance (Bhasin 2012; Huo et al. 2004), but not specifically on the combination of the two. In this section we discuss some relevant papers.

Ramesh et al. (2010) present the results of a qualitative study of 16 organizations, to answer two questions: What Requirements Engineering (RE) practices do agile developers follow? What benefits and challenges do these practices present? Their study also included subjects in quality assurance roles. Their study presents a good overview of agile requirements engineering practices and challenges. One challenge they specifically mention is the absence of adequate requirements verification. This supports the main motivation for our research.

Inayat et al. (2014) conducted a systematic literature review on agile requirements engineering practices and challenges. They mention a few examples of other reviews on agile methods. None of them focus on requirements engineering. Inayat et al. (2014) focus on practices (process) instead of on the requirements (product) themselves. However, they mention minimal documentation and neglecting non-functional requirements as a challenge for agile requirements engineering.

Sfetsos and Stamelos (2010) conducted a review on quality in agile practices. However, they focus on quality aspects of the end product (maintainability, usability, etc.), not on quality of the requirements.

Grau et al. (2014) see that documentation formats in agile are a "continuous evolution of well-known requirements engineering concepts" (such as scenarios). They see this same continuous evolution in the definition of quality attributes for agile requirements. To illustrate this continuous evolution they mention SMART (Specific, Measurable, Achievable, Relevant, Time-Boxed) and INVEST (Independent, Negotiable, Valuable, Estimable, Small, Testable) (Wake 2003) as examples of quality attributes that have been defined in practice for agile requirements. However, they do not explain these acronyms and do not go into the subject to thoroughly investigate the applicable quality attributes for agile requirements.

Related Work on Test- and Behaviour-Driven Development There is a recent stream of agile development called 'Behavior-Driven Devevelopment', or BDD (North 2006). North suggested a template that takes the agile requirements one step further by specifying them following a strict template: 'GIVEN ... WHEN ... THEN ...'. Nowadays a number of tools, like Cucumber and JBehave, exist that help to translate this format into executable test cases. Using test cases as requirements is also done in the related area called Test-Driven Development, or TDD (Beck 2002).

Melnik et al. (2006) report on an experiment where customers used executable acceptance test (storytest)-based specifications to communicate and validate functional business requirements. To evaluate the quality of the acceptance test specifications they use the following criteria: credible (contain realistic and reasonable set of operations to be likely performed by the customer), appropriate complexity (involve many features, attributes, workflows, etc.), coverage of the major functionality, easy to read and informative, easy to manage (packaged in meaningfully structured suites, subsuites etc.).

This is an example that shows that using test cases as requirements results in quality criteria primarily related to the test cases themselves. Furthermore, in TDD verification of requirements is often done by implementing automated regression tests (Bjarnason et al. 2015), thus removing the need for further informal verification based on quality criteria. In fact, the creation of test cases is in itself a way of verifying the requirements, because if the requirements are not clear, we would not be able to specify them as test cases (Bjarnason et al. 2015). For this work we focus on quality criteria for agile requirements that are not in the form of test cases and consider BDD and TDD as out of scope.

Related Work on Requirements Quality Criteria Wake (2003) introduces SMART and INVEST in the context of Extreme Programming (XP). According to him tasks should be Specific, Measurable, Achievable, Relevant, Time-boxed and stories should be Independent, Negotiable, Valuable, Estimable, Small and Testable. Both acronyms have been taken over by several other papers and are being used in agile practice (see e.g. (Leffingwell 2011; Grau et al. 2014)). Our work takes these acronyms and places them in a larger framework of quality criteria.

5.2 Method

To answer our research question (*Which are the known quality criteria for agile requirements specifications?*) we need to select articles that are relevant for the topic of quality criteria for agile requirements specifications. For the selection of the relevant articles we followed a structured process, according to the guidelines of Kitchenham and Charters (2007). The structured process they propose contains the following steps:

1. Define inclusion and exclusion criteria
2. Identify query string
3. Identify databases and other sources to search
4. Select candidates based on title and abstract
5. Refine candidate list based on full paper
6. Extend result set based on citations
7. Classify resulting papers

Candidate selection and refinement was executed by the first author and repeated by the second author with a random sample of the articles. Where differences were found the outcome was adjusted according to the discussion between the two authors. The below paragraphs describe each of the steps in detail.

Step 1: Inclusion and Exclusion Criteria

Inclusion Criteria The inclusion criteria were defined at the start of the review process based on the research question (*Which are the known quality criteria for agile requirements specifications?*) and on the type of literature we wanted to include:

- The paper is about *software* products. This is to make the distinction with agile/just-in-time in the areas of just-in-time manufacturing and systems engineering (more focused on hardware).
- *Agile* or similar just-in-time *requirements specifications* are the central topic of the paper. The paper can be focused on specific formats for requirements such as user stories.
- *Product quality* aspects of requirements specifications are an important part of the paper (we consider traceability also as a quality aspect in this sense); case studies can be included if they might deliver anecdotal evidence of requirements quality judging from the abstract.
- The quality aspects are discussed in a setting of *informal verification*. As described in the introduction we consider papers focused on validation and papers about formal methods for verification out of scope.
- The paper is a self-contained article published in a journal or in the proceedings of a workshop/conference or as a book chapter.
- The paper went through an external peer review process.
- The paper is written in English.
- The paper is published between 2001 (Agile Manifesto) and 2014 (search was conducted in January 2015).

Exclusion Criteria During the selection process we sometimes ran into papers on the topic of requirements quality and agile, but with not exactly the right focus. Since it is difficult to define the exact focus with inclusion criteria only, we enhanced them with the following exclusion criteria to indicate which topic areas we considered out of scope for our review:

- does not meet inclusion criteria
- only contains a tool description
- focus on test driven development (TDD), where tests are used instead of requirements
- focus on User Experience (UX) requirements

- focus on architecture requirements
- focus on requirements prioritization
- focus on requirements (size or effort) estimation

Step 2: Query String

Based on the research question (*Which are the known quality criteria for agile software requirements specifications?*) we also defined a search string that includes the keywords and their most important synonyms. We took care of not making the search string too restrictive. We did not want to miss relevant papers on beforehand based on a difference in terminology. This makes the key words for the search string "agile", "requirement" and "quality".

We included "just-in-time" as it has been coined by Ernst and Murphy (2012) as a term for requirements approaches that are characterized by the use of lightweight representations such as user stories, and a focus on evolutionary refinement. They note that the notion of "agile requirements" is in many ways analogous to "just-in-time requirements".

Some papers might not be discussing the general "quality" concept. Although we focus on informal verification, we explicitly extended the query string with both "verification" and "validation" to ensure that we do not miss any candidate papers based on the confusion between verification and validation.

We did not explicitly include specific requirements formats such as "user story", "feature" or "epic" because we assume that any paper with the quality of those items as central topic would also mention "requirement".

> ((agile OR "just-in-time" OR "just in time") AND requirement AND
> (quality OR verification OR validation))

Step 3: Source Selection

Based on Kitchenham and Charters (2007) we selected five digital databases which index the most important venues in the software engineering research field. For the digital databases we executed the query string on title and abstract and noted the number of returned search results. We also determined for each digital database the number of unique search results (January 2015), resulting in 630 unique items in total:

- IEEE Xplore (`http://ieeexplore.ieee.org`) 271 items
- ACM Digital Library (`http://dl.acm.org`) 268 items, 120 unique items
- Scopus (`http://www.scopus.com`) 365 items, 219 unique items
- DBLP (`http://www.dblp.org`) 0 unique items

- ScienceDirect (`http://www.sciencedirect.com`) 38 items, 20 unique items

In this initial result set of 630 unique items we saw a number of venues that related to requirements or agile specifically. We cross-checked this list of venues with the list from the review of Davis and Hickey (2009). This resulted in a list of journals and conferences which have requirements engineering or agile as their main topic and which have digital publications:

- Agile Alliance Agile Conference (AGILE), 2003-2014
- International Conference on Agile Software Development (XP), 2003-2014
- IEEE Requirements Engineering Conference (RE), 2001-2014
- International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), 2007-2014
- Requirements Engineering journal (REJ), 2001-2014
- First Agile Requirements Engineering Workshop (AREW'11), 2011

We decided to include all papers published in those venues (instead of executing the search string) for our manual search step described in the next paragraph, too reduce the chance of missing relevant papers based on the search string only. The starting year is the first year of digital proceedings for the conference or 2001 for other sources (see inclusion criteria).

And of course we include our previous work itself. This consists of a technical report (Heck and Zaidman 2014c) which is being extended into a journal paper (i.e. Chapter 4 of this thesis).

Step 4: Candidate Selection

We conducted a manual search through the sources mentioned in the previous paragraph.

A first step was to exclude items from the 630 unique items returned from the digital database that, based on their abstract and title, fulfilled the exclusion criteria. In case of doubt we included the paper in our candidate list. With this step we narrowed down the digital database items from 630 to 113, see Table 5.1.

A second step was to test the candidates for the inclusion criteria, based on title and abstract. We did this for the 113 candidates of the digital databases but also for the other sources mentioned in the previous paragraph (the agile and requirements engineering forums and our previous work). In case of doubt we included the paper in our candidate list. Based on the inclusion criteria we decided to include 55 candidates from the digital databases and added 10 new ones from other sources, see Table 5.1.

Both steps were executed by the first author and a sample was validated by the second author (without knowing the results of the first author in advance). For the first step 20 papers from IEEE Xplore, 20 papers from Scopus and 10 from ACM (the first 10 percent of the query results in the order they were returned by each digital database) were handed to the second author. For the second step also all 2014 (i.e. the most recent year) publications of the RE conference (67 papers) and XP conference (27 papers) were handed to the second author. Table 5.2 shows the different ratings that were given. This shows that in 93% of the cases both authors agreed on the in- or exclusion of the papers. In the cases where the rating was different a discussion lead to the judgment of the first author being kept. This was due to the fact that:

- The second author was more strict than the first author;
- The second author was not aware of the meaning of the term 'Definition of Ready';
- Upon reading the abstract a second time, the second author recalled his decision in the one case in which he included a paper that the first author excluded.

We found many papers on agile and (requirements) quality, that were related to our research question but did not contribute in the area of quality criteria for agile requirements specifications. Some of these out of scope topics have been mentioned in the exclusion criteria. For reference purposes we would like to note that we also

Table 5.1: Filtering publications on quality criteria for agile requirements

Source	**Step 3 Query** (Abstract)	**Step 4a Exclusion** (Abstract)	**Step 4b Inclusion** (Abstract)	**Step 5 Inclusion** (Full Paper)	**Step 6 References** (Full Paper)
ScienceDirect	20	1	0		
IEEE Xplore	271	57	26	4	5
ACM DL	120	18	9	0	
Scopus	219	38	21	4	6
Subtotal	**630**	**113**	**55**	**8**	**11**
REJ			0		
RE			4	0	
REFSQ			1	0	
XP			4	2	4
AGILE			0		
AREW			0		
Heck and Zaidman (2014c)			1	1	1
Subtotal			**10**	**3**	**5**
TOTAL	**630**	**113**	**65**	**11**	**16**

observed that a number of papers that were **not** included can be grouped around the following topics:

- The process of introducing agile methods in company or project XYZ
- Comparison of agile methods and CMM(I) or ISO
- Validation of the software product against user needs (by prototyping or active customer involvement)
- Quality assurance in agile methods (not product-oriented, but process-oriented)

This shows other topics within the area of quality and agile that have received attention in literature. This could be a starting point for other specialized systematic literature reviews.

Step 5 and 6: Candidate Refinement and Citation Snowballing

Step 5 and 6 were executed simultaneously as we decided that the list we obtained by selecting candidates based on title and abstract was not too long (65 papers) to manually review all of them. By performing step 6 with 65 papers we increase the chance of finding relevant papers during this step.

We took the remaining 65 papers and applied so-called 'snowballing' (according to the guidelines described in Wohlin (2014)). This means we checked all references from the 65 papers, but also checked all citations of these 65 papers. For the backward checking ('this paper cites...') we used the full version of each paper and for the forward checking ('this paper is cited by...') we used digital databases (IEEE and ACM include this info in their digital database and we used Google Scholar for the other papers). We repeated this snowballing process until no new papers are added.

In order to find the references in the 68 papers, we had to obtain the full paper. In doing so we at the same time reviewed the full papers for the inclusion criteria (would the paper help to answer our research question about quality criteria for agile requirements?). Based on this final review we decided to include only 11 out of 65 papers in the final result set. In cases where we had multiple papers on the same research from the same authors, we decided to include only the most relevant one (i.e., the most recent one).

Table 5.2: Interrater agreement for candidate inclusion

		Second Author	
		Include	Exclude
First Author	Include	2	9
	Exclude	1	132

Table 5.3: Interrater agreement for final inclusion

		Second Author		
		Include	Exclude	Possible Include
First Author	Include	2	1	1
	Exclude	1	4	1

The snowballing process added another 5 new papers to this final set, resulting in 16 papers in total (see Table 5.1).

To be more confident of our decision to only include 11 out of 65 papers in the final set, we also handed the top-10 most recent papers to the second author. Table 5.3 shows the results. In four cases the judgment of first and second author were different. After discussion the judgment of the first author was taken as the final judgment for each of the ten papers. In two cases (first author exclude, second author include) it turned out that the paper mentions quality of agile requirements, but does not elaborate on quality criteria. Therefore the decision to exclude them was kept. In one case (first author include, second author exclude) the paper ([P14]) mentions some elements that should be present in user stories. The first author considers this as part of 'quality', while the second author thought no quality aspects of agile requirements were mentioned. The decision was made to keep the paper. In the last case (first author include, second author probably exclude) the paper ([B14]) was indeed judged as low relevancy by both authors, as it only briefly mentions priority as a quality criterion. We decided that this brief mention was enough reason for us to keep it in the final set.

Step 7a: Classification on Meta-Data

To provide the reader with more background on the selected papers we classify them on several meta-data:

1. Author name, author affiliation, author country;
2. Year and venue of publication;
3. Publication type: Journal (J), Workshop (W), Book chapter (B), Conference (C), Other (O);
4. Number of pages;
5. Agile method: XP, Scrum or general;
6. Requirements format: user story or general;
7. Research type: see below;
8. Evaluation method: see below.

Research Type One aspect to know about the selected papers is the type of research that is included in each of them. This answers a number of questions we can ask about the selected papers: is it a new solution that is proposed? does it contain a full-blown validation? is it personal experience? is it just the author's opinion? This helps readers to understand the value of the paper for their own practice. For each paper we classify those aspects that touch on our topic (quality of requirements).

For the classification of the research type we follow the framework of Wieringa et al. (2005):

- **Evaluation Research**: investigates a problem in Requirements Engineering (RE) practice or an implementation of an RE technique in practice.
- **Proposal of Solution**: proposes a solution technique and argues for its relevance, without a full-blown validation.
- **Validation Research**: investigates the properties of a solution proposal that has not yet been implemented in RE practice.
- **Philosophical Paper**: sketches a new way of looking at things, a new conceptual framework, etc.
- **Opinion Paper**: contains the author's opinion about what is wrong or good about something, how we should do something, etc.
- **Personal Experience Paper**: the emphasis is on what and not on why. The experience may concern one project or more, but it must be the author's personal experience.

Evaluation Method For each paper we also classify the type of evaluation that was done for the quality-related aspects. To give an impression of the depth of evaluation we present a number of dimensions on the evaluation method used (adapted from Cornelissen et al. (2009)):

- **Preliminary**: Evaluation of proposed solution is of preliminary nature, e.g. toy example or informal discussion
- **Regular**: Evaluation is not of preliminary nature
- **Human Subjects**: Evaluation involved human subjects, e.g. questionnaires, interviews, observations
- **Industry**: Evaluation was performed in an industry setting
- **Quantitative**: Evaluation resulted in some quantitative data on the proposed solution
- **None**: No evaluation

Step 7b: Classification on Quality Criteria

To classify the papers based on quality criteria we build upon our previous work (Heck and Zaidman 2014c). We structure the inventory according to the three overall quality criteria (QC) explained in (Heck and Zaidman 2014c). We think that this is a general classification that holds for written requirements (in fact, for all written documentation): it is firstly important that we have all documentation that needs to be checked (completeness), secondly that the writing follows certain templates or standards (uniformity, to aid the reviewer in understanding documentation), lastly, that we know which criteria to check against (consistency and correctness).

Overall we divide the quality criteria from the selected papers into three categories:

[QC1] Completeness. In this category we place criteria that specify elements that should be present in (the description of) the agile requirements. An example would be the rule that each requirement should have a unique identifier. Note that our definition of completeness (all structure elements of a single requirement should be there) is different from the notion of completeness as in 'specifying all user needs'.

[QC2] Uniformity. In this category we assign criteria that pertain to a standardized style or format of the agile requirements specification.

[QC3] Consistency and correctness. This category contains all other criteria that state something about the correctness of individual requirements or the consistency with other requirements.

We identify any mentions of quality criteria for agile requirements specifications. We include each of the mentioned quality criteria in our classification, regardless of the length of the discussion in the paper or the strength of the evidence presented in the paper. In the same way we also include all elements of agile requirements that are mentioned in the papers (QC1). We take the decision to include everything regardless because we want to provide a broad overview which points to specific papers for detailed information.

Step 7c: Classification on Recommendations for Practitioners and Researchers

As a final step we classify recommendations for practitioners or researchers found in the papers.

For practitioners we look at recommendations found anywhere in the 16 resulting papers about how to apply the quality criteria in practice. Then we summarized these recommendations in a few paragraphs, to structure them and to connect similar recommendations.

We present the recommendations for researchers in the form of a research agenda. To construct this research agenda we use the future work as indicated in the selected papers and our own analysis of what we see missing in the selected papers.

5.3 Results: Meta-Data Classification

In this section we characterize the 16 selected papers based on meta-data such as author, venue, agile requirements format, research- and evaluation type. This sets the ground for a detailed analysis of the quality criteria in Section 5.4. Each paper is identified with a unique identifier [XXnn] based on the author and year of publication. For an overview of the unique identifiers see Table 5.8 in the appendix of this chapter, that presents each of the 16 papers in more detail.

Author, Affiliation, Country

The 16 selected papers have been written by 34 authors. Two of those authors have co-authored 2 papers. These papers are similar but have both been included because their contributions slightly differ. Authors of eight papers originate from Europe, 7 from North-America, 1 is written by authors from both continents. Two papers are written by authors with an industrial affiliation, 14 are from researchers. This shows that the topic attracts interest from both researchers and practitioners.

Year, Venue, Type, Pages

The publication date of the selected papers is spread over different years between 2001 and 2014 (see Figure 5.1). However, the bulk of the publications come after 2009. It looks like the topic is gaining in popularity, but we can only be sure of this in the years to come. It is also worth noticing that the types (workshop, conference, journal, book chapter, other) of publication are very heterogeneous.

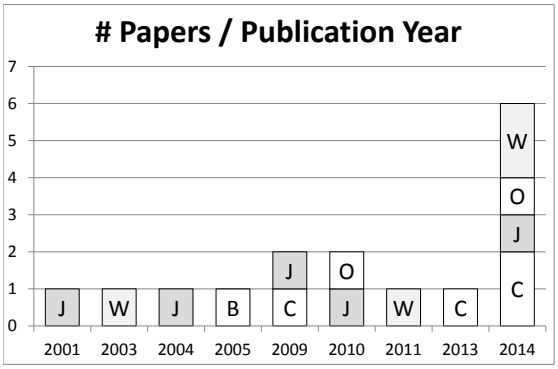

Figure 5.1: Meta-data of selected papers (W=workshop, C=conference, B=book chapter, J=journal, O=Other)

Table 5.4: Venues

Venue	Type	Paper
Requirements Engineering Journal	Journal (J)	[B14]
Journal of Emerging Technologies in Web Intelligence	Journal (J)	[DQ10]
Workshop on Software Measurement (IWSM-MENSURA)	Workshop (W)	[D11] + [DT14]
Journal of Defence Software Engineering	Journal (J)	[D01]
IEEE Southeastcon	Conference (C)	[FM13]
Journal of Object Technology	Journal (J)	[F04]
Better Software Magazine	Other (O)	[GG10]
Workshop on Traceability in Emerging Forms of Software Engineering	Workshop (W)	[L03]
Conference on Extreme Programming and Agile Processes in Software Engineering (XP)	Conference (C)	[L14a] + [P14]
Workshop on Cooperative and Human Aspects of Software Engineering	Workshop (W)	[L14b]
Journal of Software	Journal (J)	[PR09]
Engineering and Managing Software Requirements	Book Chapter (B)	[SS05]
Conference on Information Technology: New Generations (ITNG)	Conference (C)	[SL09]

Table 5.4 shows the different venues the selected papers have been published in. This is important to know for persons that are new to the field and want to know where to start reading or publishing themselves. Unfortunately we cannot give them a definite answer. The number of different venues is almost as large as the number of papers. The Workshop on Software Measurement (IWSM-MENSURA) has 2 publications, but these come from the same research group. Only the Requirements Engineering Journal and the Agile Conference (XP) have two distinct publications. This at least shows that not only the traditional requirements engineering community is working on this topic, but also the agile community. This indicates that the topic is also considered important within agile environments. For the rest the communities are quite diverse, ranging from 'Human Aspects' to 'Web Intelligence'. Note that in Table 5.4 we have not included [HZ14] (technical report).

Table 5.8 in the appendix of this chapter shows the number of pages for each paper. Most of them are longer papers of 8 pages or more. Half of the papers even have more than 10 pages.

Agile Method and Requirements Format

As described by De Lucia and Qusef (2010) many different agile methods exist and they all have slight differences from a requirements engineering perspective. That is why it is important to know for each of the selected papers which agile method they discuss, as this might influence their perspective on quality criteria. The same

Table 5.5: Classification of selected papers

	Agile Method				**Research Type**			**Evaluation**					
	General	Scrum	XP	(User) Story	Evaluation Research	Proposal of Solution	Validation Research	Preliminary	Regular	Human Subjects	Industry	Quantitative	None
[B14]	x					x			x				
[DQ10]	x			x	x								x
[D11]	x			x		x		x				x	
[DT14]		x		x			x		x		x	x	
[D01]			x	x	x								x
[FM13]	x					x		x			x	x	
[F04]	x			x		x							x
[GG10]	x			x		x							x
[HZ14]	x			x		x		x		x	x		
[L03]	x					x							x
[L14a]	x			x		x			x	x	x		
[L14b]		x		x	x				x	x			
[PR09]			x	x		x		x		x	x		
[P14]	x			x	x				x	x	x		
[SS05]	x			x	x								x
[SL09]		x		x	x				x	x	x		

goes for the requirements format discussed, since not all quality criteria apply to all formats (e.g. there exist specific templates for some formats).

Most papers do not describe one specific agile methodology (see Table 5.5). Two papers specifically address eXtreme Programming (XP) and three papers specifically address Scrum.

According to Ernst and Murphy (2012) "just-in-time requirements refer to higher-order organizational constructs, including features to be added to the project, agile epic and user stories, improvements to software quality, and major initiatives such as paying down technical debt". In our selected papers, those papers that mention a specific type of requirements, all mention (user) stories. User stories (US) are short sentences that represent the customer requirements at a high level, and the documentation for these stories includes the explanations of the requirements [D11]. Table 5.5 shows which other papers treat user stories. Note that some of them, such as [SS05], only mention user stories briefly.

Table 5.5 indicates that User Stories are not the only requirements format being discussed. Researchers investigating quality of agile requirements specifications

should thus not only investigate user stories and practitioners working with agile requirements should bear in mind that there is no "obligation" to use user stories as their format.

Research Type

Out of the 16 selected papers, 9 of them have been classified as 'Proposal of Solution', 6 of them as 'Evaluation Research' and only 1 as 'Validation Research' (see Table 5.5). This shows that most papers report on a new solution that they propose, which makes it harder for practitioners to evaluate how to apply the solution in practice (i.e. validation research is missing).

Evaluation Method

Table 5.5 presents the evaluation method used in each of the 16 papers. Six papers miss an evaluation and four papers only contain a preliminary evaluation. We double-checked to see if newer papers of the same authors have been published, but this was not the case. There were not many (only 3) quantitative evaluations. In combination with our analysis in the previous section this shows that most papers report on an existing practice or new solution, without a full-blown validation. Note that for our own work [HZ14] we have executed a more extensive quantitative evaluation which is under submission (i.e. Chapter 4 of this thesis). A good thing is that 70% of the evaluations (7 out of 10) involved industry projects or companies. This confirms our earlier remark that both industry and academia are working on this subject.

5.4 Results: Quality Criteria Used in Literature

We made an inventory of quality criteria for agile requirements specifications that are mentioned in each of the 16 selected papers. The method we used to create this inventory is described in Section 5.2.

In Figure 5.2 we present the classification of quality criteria and mention between brackets for each quality criterion which paper(s) mention them. In Table 5.8 in the appendix of this chapter we briefly discuss the contribution of each of the papers to this inventory. Together, the 16 selected papers mention 28 different quality criteria for agile requirements specifications, of which most criteria are confirmed by more than one paper. Note that out of these 28 criteria, 11 criteria had not been mentioned in our previous work [HZ14] and four of them are only mentioned in [HZ14].

In what follows we will highlight criteria that are mentioned by three or more papers.

Priority Five papers mention that requirements should have a priority defined. [SS05] mentions 'Requirements Prioritization' as an important technique for agile

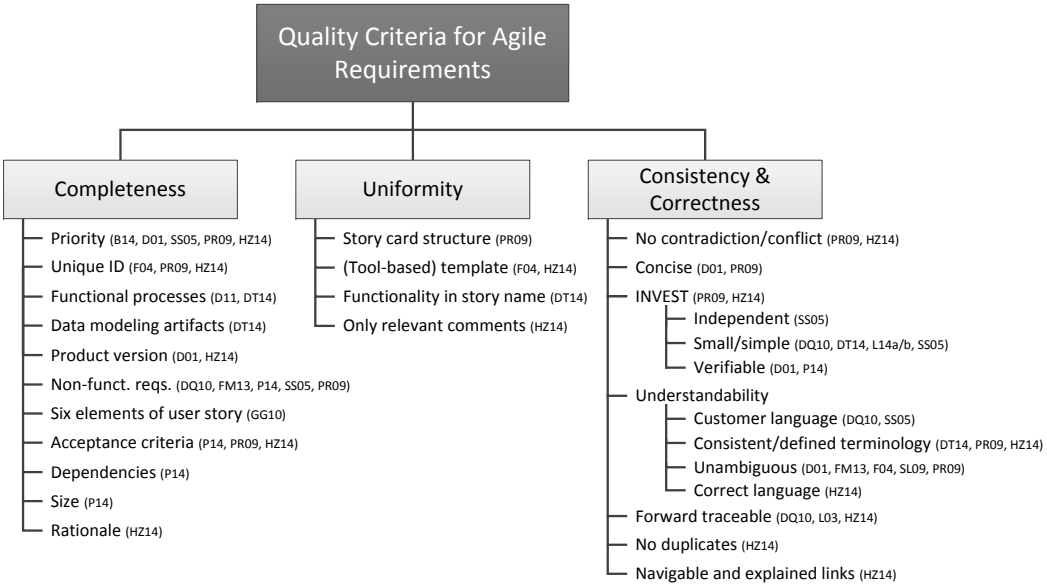

Figure 5.2: Quality criteria for agile requirements (next to each quality criterion the papers that mention it).

methods. [HZ14] mentions priority as an example of 'relative importance'. [B14] calls this 'a grade for the importance of each requirement' and describes a clustering algorithm based on this. [D01] calls this 'Annotated by Relative Importance', based on (Davis et al. 1993). [PR09] suggests to prioritize story cards 'based on agile software development values'.

In agile development the priority of a requirement is very important to know, because the priority is used to plan iterations. In each iteration the open requirements with the highest priority are detailed and subsequently developed. Priority is allowed to change as long as the requirement is open. In this way agile development ensures that the customer receives what he needs most at any given moment. This also allows for the customer to change his/her mind and upgrade or downgrade requirements during the project by adjusting the priority attribute.

Unique Identifier Three papers mention that requirements need a unique identifier. [F04] recommends a template for agile requirements where 'ID' is one of the columns. Since [PR09] is about story cards, they call this unique identifier a 'story card number'. [HZ14] assumes that requirements are stored in an electronic tool and thus automatically assigned a unique ID.

Note that an unique identifier is also useful in oral communication of agile requirements. In oral communication we need an easy and unambiguous way to refer to requirements we are discussing, whereas in written communication we could use other means such as hyper-links or paragraph numbers.

Non-Functional Requirements Five papers ([DQ10], [FM13], [P14], [SS05], [PR09]) indicate that non-functional requirements should not be overlooked in agile development. [P14] states that 'architecture criteria (performance, security, etc.)' should be identified for a User Story to be considered 'Ready'.

Recommendations are to arrange meetings to discuss the non-functional requirements as early as possible ([DQ10] and [SS05]), to include them as part of the story card [PR09], use quality metrics and a risk-driven algorithm to plan them [FM13].

Acceptance Criteria [P14], [PR09] and [HZ14] all mention acceptance criteria or acceptance tests as an important part of user stories.

In traditional requirements engineering acceptance tests are often developed together with the upfront requirements specification. Agile development does not recommend writing such elaborate test documentation upfront since there is a good chance that certain requirements do not get implemented or will change and thus the test cases will be obsolete. To replace comprehensive upfront test documentation, agile requirements should be elaborated with acceptance criteria (in Behaviour-Driven Development these acceptance criteria are even formalized according to a template).

INVEST An acronym introduced specifically for agile requirements quality is INVEST: user stories should be Independent, Negotiable, Valuable, Estimable, Small and Testable (Wake 2003). For readability of Figure 5.2 we have included INVEST as 1 sub-item, when in fact it is a collection of six criteria. INVEST is mentioned as-is in [HZ14]. [PR09] does not mention the acronym, but does mention each of the six criteria separately.

Seven other papers mention that agile requirements should be independent [SS05], should be kept small or simple [DQ10, DT14, L14a/b, SS05] and that the requirements should be verifiable (= testable) [D01, P14]. [DT14] states that user stories should be kept small in the sense that "It is expected that each user story be mapped to only one functional process.", [L14a] and [L14b] state that the 'Expected Implementation Duration' should be kept low. [DQ10] suggests to split requirements that are considered too complex by the team. [GG10] dedicated their whole paper to how to perform what they call "story slicing". [P14] phrases verifiability as 'team knows what it will mean to demo the User Story'.

Practitioners working with agile requirements should also keep in mind that INVEST/SMART could also be applied to other types of agile requirements.

Understandability We have used the term 'Understandability' to group several criteria related to the choice of wording for the requirements specification. Defining

clear requirements can save a lot of time in discussion and question answering during the implementation.

Two papers [DQ10, SS05] mention that this can be achieved with requirements *written in the language of the customer*. [PR09] states that story cards should "use language simply, consistently and concisely". Two more papers deem it important to use a *consistent and defined terminology* throughout all requirements: [DT14] and [HZ14]. [HZ14] advocates the use of a 'glossary' for this purpose and also recommends the use of *correct language* (i.e. spelling- and grammar-wise). In total five papers mention that it is important for the requirements to be *unambiguous* in general [D01, FM13, F04, SL09, PR09].

Forward Traceable Three papers mention that agile requirements should be forward-traceable [DQ10, L03, HZ14].

It is important to know to which source code and test cases the requirements trace (forward traceability) to be aware where things must be changed when requirements change, since agile development embraces change as a given factor (Beck et al. 2001).

5.5 Results: Recommendations for Practitioners

In this section we discuss how practitioners should use the quality criteria in practice. We support our discussion with references to the relevant papers.

Use a list of quality criteria We have found 16 papers that contain quality criteria for agile requirements specifications. We advise practitioners to consider the full list of criteria summarized in Section 5.4 and establish a list of quality criteria appropriate for their own project, team or environment.

[HZ14] describes how a given list of quality criteria can be customized by a team: 1) decide which criteria are not relevant for the team; 2) add missing criteria by interviewing team members, by re-evaluating old requirements or by applying the criteria in practice and improve them on-the-fly.

The quality criteria should be applied to the agile requirements specifications, but do not have to hold from the beginning. According to [DQ10] "any documents produced in the early stages can quickly become irrelevant because the agile principles encourage requirements change". This is confirmed by e.g. Ernst and Murphy (2012) who coined the term "just-in-time requirements" for this. The recommendation is to only document what is relevant at a given moment, and postpone all other requirements documentation to as late as possible. Analogue to this we can also say that the quality criteria should hold just-in-time: for each criterion it should be decided at which point in time it should hold (e.g. at creation time of the requirement or just before development starts).

Checklist or Definition of Ready [HZ14] calls the resulting list of quality criteria a checklist. [PR09] also promotes the use of a so-called 'validation checklist' to assess correctness as part of a high maturity level for agile requirements engineering. [P14] advises to include such criteria in a so-called 'Definition of Ready'. A Definition of Ready (DoR) is a sort of checklist that an agile team uses to determine when a user story is ready for the developers to start implementing it. According to [P14] "not having a definition of ready can seriously impede the flow of work through your system". A DoR is something that can also be implemented for other types of agile requirements.

Use of a tool To simplify the process of applying checks the requirements could be stored in a tool. According to [B14] "tools that allow collaboration and provide tracing of changes and allow recording of requirements in standardized formats, along with a proper plan for team coordination, are considered as essential". This same finding is supported by [HZ14] that already assumes agile requirements are stored in a tool, resulting in quality criteria that are inherent to the tool (e.g. unique identifiers). [DQ10] recommends the use of tools not only for storing requirements but also for storing traceability information between requirements, tests and code.

5.6 Results: Research Agenda

In this section we discuss the 16 papers with respect to open points for researchers to work on in the area of quality of agile requirements:

1. *More than user stories*. Most selected papers investigate or mention user stories (see Table 5.5). However, we think that the characteristics of a good user story also hold for other types of agile requirements. An example of this is given in [HZ14] for feature requests in open source projects. We would like to see more research papers on the topic of suitable requirements formats for agile environments with of course a focus on the quality aspects of each of those formats. This would help practitioners to select the proper requirements format for their environment and to be aware of the caveats for each of the possible formats.

2. *Validation research*. There are few papers with a thorough validation of the proposed solution (see Table 5.5). The selected papers are one-off publications, without a continuation in future work. Existing and new frameworks or methodologies for creating good quality requirements should be validated more extensively by the research community. This makes it easier for practitioners to see if the method could work in their daily practice.

3. *Case studies*. The generalizability of results is a threat to validity for most of the selected papers. Almost all of them mention that more case studies should be done in future work to validate the results in other situations. For

practitioners it is extremely valuable to know to what extent the described results can be applied in other situations. Therefore, as a research community we need to publish more case studies in the area of agile requirements engineering with real-life data sets or industrial setting. This need for more case studies is also confirmed by other publications on research agendas for agile development (Dingsoyr et al. 2008; Dybå and Dingsøyr 2008). In addition, we would also want to advocate the need for *longitudinal studies* (e.g., see Runeson et al. (2012)) in agile requirements engineering to study how agile requirements engineering quality practices change or vary over time.

4. *Tooling*. [DQ10], [HZ14], [L03] and [PR09] mention that tools could be useful to automate some of the checks for quality criteria. We think this is true. Researchers could develop prototypes of such tools, test them and make them available as e.g. plug-ins for requirements tools. In this way practitioners can use the guidelines for good quality agile requirements without the extra burden of checking things manually.

5. *Cooperation*. The types of venues that have published papers on the topic of quality of agile requirements are very heteregenuous (see Table 5.4). Both the agile community and the requirements engineering community have published papers. Next to that, a whole scala of different smaller and bigger venues welcome papers on the topic (from the measurement community to the web intelligence community to the human aspects of software engineering community). A valuable contribution can be made if researchers working on the topic would collaborate more across communities. In 2015 we see two good examples of this: the Just-in-Time Requirements Engineering workshop (JITRE)[2] in conjunction with the Requirements Engineering conference (RE'15), and the Workshop on Requirements Engineering in Agile Development (READ)[3] in conjunction with the Conference on Agile Software Development (XP2015). Also here it could be remarked that it would have been even better if both communities would have joined in one single workshop. Although in this case at least Neil Ernst and Maya Daneva are involved in both workshops (organizing the one and serving as a Program Committee member for the other).

6. *Also start from the problems.* Next to investigating sources that mention quality criteria for agile requirements specifications, we should also start from the other side. What are quality problems with software that has been developed in an agile way and which of those problems link back to quality problems in the agile requirements specifications? When we investigate the quality problems in practice, we can subsequently define quality criteria for

[2] https://jitre.wordpress.com/

[3] http://www.xp2015.org/1st-international-workshop-on-requirements-engineering-in-agile-development/

agile requirements specifications that will help us to avoid or prevent those problems in practice. In that way we would let problems in agile practice guide us towards the most important quality criteria for agile requirements specifications. This could also lead to newly discovered quality criteria.

7. *Investigate necessity and impact of agile requirements correctness.* In our introduction we have stated that correctness of agile requirements specifications is important. We have gathered further support for this claim with the publications in this survey. However a lot of questions around this topic still remain open. Do written requirements play an important role in agile? What is the difference between distributed and non-distributed teams for the importance of written requirements correctness? What is the impact of non-correct requirements on final product quality? We would like to see more research in this area to answer these important questions and further confirm our claims.

5.7 Discussion

In this discussion we will revisit our research question and we will discuss the threats to validity.

[RQ5] Which are the known quality criteria for agile requirements specifications?

Based on the selected papers of our literature survey, we identified 28 quality criteria that have been mentioned for agile requirements specifications. These quality criteria are listed in Figure 5.2. In this section we analyze which criteria come from traditional requirements engineering and which new quality criteria have been presented.

Quality criteria for traditional requirements specifications When investigating the 28 quality criteria, we find that most of them are also applicable to traditional up-front requirements specifications. Criteria specific for agile requirements specifications are: six elements of a user story [GG10], story card structure [PR09], functionality in story name [DT14] and INVEST [PR09, HZ14]. Of course the interpretation of traditional criteria might be slightly different for agile requirements. For example, acceptance criteria should not be full-blown test scenarios but short statements indicating when the implementation of a requirement can be accepted; priority is paramount for agile requirements as it is the basis for all planning activities and determines when to spend time on detailing the requirement; understandability might be less important for agile requirements specifications in environments where conversation with the customer can be used to clarify ambiguities.

The papers that we selected do not provide enough evidence to completely answer the question which traditional criteria still apply to agile requirements specifications. However, [D01] contains an analysis where it is shown which traditional

quality criteria are not or less applicable to eXtreme Programming. A similar analysis is made in [HZ14] for feature requests in open source systems.

New quality criteria for agile requirements specifications The INVEST model seems to be the only 'new' quality criterion that is mentioned in the context of agile requirements specifications, since the other three new ones (six elements of a user story [GG10], story card structure [PR09], functionality in story name [DT14]) can be seen as translations of existing criteria to the agile requirements format. The goal of INVEST is to divide the system to be developed in small deliverables that can be delivered independently, one of the key principles of agile development. INVEST is currently tied to user stories, but we think it is also valuable for other types of (agile) requirements.

Just-in-time quality In our own continued research we decided to add a timing dimension to quality criteria for agile requirements specifications. Since the requirements are specified or detailed just-in-time, some quality attributes also do not have to hold from the creation of the requirement, but should hold just-in-time. For example an initial specification might be ambiguous as long as any unclear terms or wordings have been resolved just before development starts.

Threats to Validity

Threats to the validity of the systematic review are analyzed according to the following taxonomy: construct validity, reliability, internal validity and external validity.

Construct Validity

Construct validity reflects to what extent the phenomenon under study really represents what the researchers have in mind and what is investigated according to the research questions.

Requirements engineering in general and requirements engineering in an agile context are broad subjects. We explicitly confined our survey to agile requirements engineering research with a special focus on *quality criteria* for agile requirements specifications. In order to be as objective as possible on which papers to include or exclude in our survey, we followed the advice of Kitchenham (2004) and Brereton et al. (2007) to use predefined selection criteria that clearly define the scope of the survey.

Our process of collecting relevant articles was based on keyword searches in well-established scientific digital libraries (IEEE Xplore, ACM Digital Library, Scopus, DBLP and ScienceDirect). However, as Brereton et al. (2007) point out, "current software engineering search engines are not designed to support systematic literature reviews". This observation is confirmed by Staples and Niazi (2007). In order to mitigate this threat we also manually processed relevant venues in a certain

period of time, in particular the Agile Alliance Agile Conference (AGILE), International Conference on Agile Software Development (XP), IEEE Requirements Engineering Conference (RE), International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), the Requirements Engineering journal (REJ) and the Agile Requirements Engineering Workshop (AREW'11). In an additional step, we also involved Google Scholar[4]. Our query string returned more than 98 000 results with Google Scholar. Of course it is not feasible to manually check this amount of items, so we decided to include the top-100 of Google Scholar as a double-check that we do not miss any important publications not indexed by the main databases mentioned above. No new articles were found with this double-check.

We used a single query string when querying the aforementioned digital libraries. It might be that we missed papers, because different terms are used by different authors. We tried to mitigate this issue by performing the aforementioned manual search, but also through citation snowballing (Wohlin 2014). We performed the snowballing recursively until we could no longer add relevant literature to the set of papers under consideration.

In Section 5.2 we have discussed and justified the inclusion and exclusion criteria for the selection strategy of publications. However, it is still possible to miss some relevant literature. One such instance is the existence of "grey literature" (Auger 1994) such as technical reports and PhD theses (Kitchenham and Charters 2007). In this literature review, we did not include such information. On the other hand, we did include book chapters for which we know that they underwent an external review process ([SS05]).

This work is centred around agile requirements *specifications*, thus leaving out face-to-face communication or prototyping, both elements which are very important in an agile setting (Beck et al. 2001). While we acknowledge this importance, Inayat et al. (2014) explicitly mention situations (e.g., distributed teams, large teams, complex projects) that require documented agile requirements that are fully elaborated in writing. As soon as agile teams cannot rely on face- to-face communication the "correctness" of written documentation becomes more and more important.

Inconsistent terminology or use of different terminology with respect to the exercised search string (see Section 5.2) may have biased the identification of primary studies. The manual search and recursive snowballing approach we followed should mitigate the risk that we have missed important articles.

Agile requirements engineering in general and quality criteria for these agile requirements are young research areas. This shows in the papers that we surveyed as many are preliminary in nature and have the purpose to launch and discuss ideas. These papers typically do not contain a full-blown evaluation. In fact, by looking

[4] `http://scholar.google.com`, last visited July 18th, 2015

at Table 5.5 we observe that 6 out of 16 papers contain no evaluation at all, while another 3 have a preliminary evaluation. There is no immediate mitigation for this risk, but it is our opinion that each of these papers has an important contribution.

Another threat to validity is the strength of evidence with regard to the quality criteria as mentioned in the papers that we surveyed. We acknowledge that not all papers have quality criteria for agile requirements as their primary goal of investigation. While there is no immediate mitigation for this risk, it is our view that each of the insights generated in these papers is an important piece of information for the overall area.

Reliability Validity

Reliability Validity focuses on whether the data is collected and the analysis is conducted in a way that it can be repeated by other researchers with the same results.

In this chapter we have presented a series of findings based on the papers that we selected during our systematic literature review. Since conducting a survey is a largely manual task, most threats to validity relate to the possibility of researcher bias, and thus to the concern that other researchers might come to different results and conclusions. One remedy we adopted is to follow, where possible, guidelines on conducting systematic reviews as suggested by, e.g. Kitchenham (2004) and Brereton et al. (2007). In particular, we documented and reviewed all steps we made in advance (per pass), including selection criteria and attribute definitions.

The first author of this chapter did the actual paper selection and refinement. The application of the inclusion and exclusion criteria might however be subjective. Therefore, the second author was involved in a series of checks in which the second author re-applied the inclusion and exclusion criteria. The results showed some discrepancies, which were discussed among the authors and resulted in a common understanding of why a paper should be included or excluded from the set.

We acknowledge that the classification of the articles in Table 5.5 is probably the most subjective step in our approach. This step is also subject to researcher bias as the interpretation may seek for results that the reviewers were looking for. Our countermeasure has been a systematic approach towards the analysis, including the use of an established framework for research types as proposed by Wieringa et al. (2005).

Internal Validity

Internal validity is concerned with the analysis of the data. Since no statistical analysis was done considering the small sample size, the analysis is mostly qualitative and thus subject to researcher bias (also see Section 5.7).

External Validity

External validity is about generalizations of the findings derived from the primary studies.

As the field of quality criteria for agile requirements is flourishing, the observations we have made in this chapter are valid at the time of writing, but might not generalize into the future. More specifically, it might be that future work extends upon the 28 quality criteria that we have currently listed.

Similarly, while we have paid a lot of effort on finding all relevant literature (also see Section 5.7), it might still be that the list of 28 quality criteria is incomplete.

We are aware of the fact that agile requirements engineering in general is a very actively debated topic among practitioners through for example blog posts. We excluded such non-reviewed material in our study. The insights that are thus generated might not have been reported upon in an academic paper. Future work might want to investigate this particular aspect, in a similar way to the structured mapping study by Kabbedijk et al. (2015) who combined the academic and industrial perspective in their particular area.

5.8 Conclusion

Context For traditional up-front requirements specifications quality is considered important and standards define what quality of requirements specifications means. Our main research question is which quality criteria apply to agile requirements specifications, since no standards have been defined for them.

Summary In this chapter we report on a systematic literature review on quality criteria for agile requirements specifications. Through a structured process we selected 16 articles. From these articles we devised an overview of existing quality attributes that can be used in informal verification. We also derived (1) recommendations for practitioners that want to perform informal verification of requirements specifications in an agile setting and (2) a research agenda for academics that highlights the need for further research in this area. In addition, the resulting systematic overview is useful as a reference work for researchers in the field of informal verification and helps them identify both related work and new research opportunities in this field.

Quality Criteria Figure 5.2 shows the list of 28 quality criteria that we collected. Most of these criteria are not new for agile requirements specifications, although the way to apply them to the specification might be slightly different from a traditional setting. The only new criterion we encountered is INVEST (Independent, Negotiable, Valuable, Estimable, Small and Testable).

Practical recommendations Practitioners working with agile requirements can take the recommendations in Section 5.5 as a starting point. Our main recommendation is to follow the list of quality criteria in this chapter. When a team includes the applicable criteria in the Definition of Ready for their agile requirements, the

assessment of quality criteria on those requirements will be an undeniable part of the daily agile development process.

Research Agenda In Section 5.6 we present our research agenda for the area of quality criteria for agile requirements. Our overall observation is that only few papers focus on quality criteria for agile requirements. However, the topic is deemed important by both researchers (Inayat et al. 2014) and practitioners that we encountered during our own investigations. Our most important recommendation is thus for academia to put more effort in this area in general.

Contributions In short, this chapter makes the following contributions:

- A classification of existing quality criteria that can be used for the assessment of agile requirements specifications;
- Recommendations for practitioners for quality assessment of agile requirements specifications;
- A research agenda in the area of quality of agile requirements containing 7 important avenues for future research.

Future Work This literature review shows that the number of publications on informal verification of agile requirements specifications has increased over the last few years. This indicates that there should be coming more in the next few years. We plan to keep track of new publications in this area with the goal of updating this literature review. Furthermore we plan to further evolve and validate our own framework with quality criteria and we hope that others will also build on our work. Additionally, another avenue of future work is to check which quality criteria for agile requirements specifications are described in non-academic blogs, white papers, etc.

A. Details of Selected Papers on Quality Criteria for Agile Requirements

This appendix contains detailed information on each of the 16 selected papers. We repeat author, title and venue and include number of pages. We provide a short summary of the paper based on the paper abstract. We briefly explain the relevance of the paper for quality criteria for agile requirements (in italic) and we sum up each of the quality criteria mentioned in the paper. For an overview of all quality criteria from all papers, see Figure 5.2.

[ID] Reference	Title and Venue	#pp.	Summary and Relation to Quality Criteria
[B14] Belsis et al. (2014)	PBURC: A Patterns-Based, Unsupervised Requirements Clustering Framework for Distributed Agile Software Development (Requirements Engineering Journal)	13	This paper presents a patterns-based, unsupervised requirements clustering framework, which makes use of machine-learning methods for requirements validation, being able to overcome data inconsistencies and determine appropriate requirements clusters for the definition of software development sprints. *[B14] is of low relevance for quality criteria as it focuses on a clustering algorithm for requirements.* *Quality criteria: priority.*
[D01] Duncan (2001)	The Quality of Requirements in Extreme Programming (Journal of Defence Software Engineering)	4	This paper describes and evaluates the quality of requirements generated by using XP and discusses how the XP process can assist or hinder proper requirements engineering. *[D01] compares eXtreme Programming to a list of quality criteria for traditional requirements from Davis et al. (1993).* *Quality criteria: priority, product version, verifiable, concise, unambiguous.*
[D11] Desharnais et al. (2011)	Using the COSMIC Method to Evaluate the Quality of the Documentation of Agile User Stories (IWSM-MENSURA)	4	In the research reported here, the COSMIC method is used to analyze and report on the quality of the documentation of user stories. The functional size of evolving requirements can be measured with COSMIC measurement method. To support this measurement activity, the quality of the documentation is important to be interpreted correctly. *See [DT14].*

[ID] Reference	**Title and Venue**	**#pp.**	**Summary and Relation to Quality Criteria**
[DT14] Dumas-Monette and Trudel (2014)	Requirements Engineering Quality Revealed through Functional Size Measurement: An Empirical Study in an Agile Context (IWSM-MENSURA)	11	This paper reports preliminary results related to a software development organization. The functional size of the software was measured and compared with development and measurement effort, taking into account the quality rating of requirements. The results led to recommendations for the organization and recommendations for planning any software measurement project. *Both [D11] and [DT14] focus on size measurement of agile requirements (using COSMIC). They claim that quality of those requirements is a necessary condition for accurate estimation. Quality in this sense mainly means completeness: descriptions of functional processes and data modeling artifacts are needed to base the COSMIC size measurements on. [DT14] contains a case study that also highlights other quality issues related to size measurement: user story names should at least contain the name of the functional process it maps to in order to be correctly measured; it is expected that each user story be mapped to only one functional process; consistent terminology should be used for data groups/objects.* *Quality criteria: functional processes, data modeling artifacts, functionality in story name, small/simple, consistent/defined terminology.*
[DQ10] De Lucia and Qusef (2010)	Requirements Engineering in Agile Software Development (Journal of Emerging Technologies in Web Intelligence)	9	This paper discusses problems concerned with requirements engineering in agile software development processes and suggests some improvements to solve some challenges caused by large projects. The paper also discusses the requirements traceability problem in agile software development.

[ID] Reference	Title and Venue	#pp.	Summary and Relation to Quality Criteria
			[DQ10] is a general overview paper on requirements engineering for agile methods. The focus is not specifically on quality, but [DQ10] mentions some criteria: non-functional requirements should not be forgotten, requirements should be in the language of the customer, requirements should not be too complex, requirements should be forward-traceable (i.e. to source code and tests). *Quality criteria: non-functional requirements, small/simple, customer language, forward traceable.*
[F04] Firesmith (2004)	Generating Complete, Unambiguous, and Verifiable Requirements from Stories, Scenarios, and Use Cases (Journal of Object Technology)	13	This paper shows how to transform incomplete and vague stories, scenarios, and use cases into a proper set of textual requirements. *[F04] promotes a template for writing textual requirements: ID, trigger, precondition, action, postcondition.* *Quality criteria: unique ID, template, unambiguous.*
[FM13] Farid and Mitropoulos (2013)	NORPLAN: Non-functional Requirements Planning for Agile Processes (IEEE Southeastcon)	8	This research proposes project management and requirements quality metrics. NORPLAN proposes two prioritization schemes, Riskiest-Requirements-First and Riskiest-Requirements-Last, for planning release and sprint cycles using a risk-driven approach. The approach is validated through visual simulation and a case study. *[FM13] focuses on non-functional requirements planning for agile processes. It contains a table with agile requirements quality metrics. However, these metrics are mainly process-related, except for ambiguity*. *Quality criteria: non-functional requirements, unambiguous.*
[GG10] Gottesdiener and Gorman (2010)	Slicing Requirements for Agile Success	8	This paper presents different options for slicing user stories.

[ID] Reference	Title and Venue	#pp.	Summary and Relation to Quality Criteria
	(Better Software Magazine)		*[GG10] does not specifically focus on quality, but presents a way of slicing (splitting) user stories. By explaining the ways of slicing, they also mention six important elements of a user story: user roles, actions, data objects, business rules, interfaces, quality attributes (i.e. non-functional requirements).* *Quality criteria: six elements of a user story.*
[HZ14] Heck and Zaidman (2014c)	A Quality Framework for Agile Requirements: A Practitioner's Perspective (TU Delft Computer Science Report)	11	This paper presents a quality framework for informal verification of agile requirements and instantiates it for feature requests in open source projects. The framework was derived based on an existing framework for traditional requirements specifications, literature about agile and open source requirements and the authors' experience with agile and open source requirements. *[HZ14] presents a framework for agile requirements quality criteria, which is instantiated for feature requests in open source projects and user stories. It presents a large number of quality criteria. The use of a template is indirectly also promoted because they claim that a tool should be used to store the requirements. This tool, e.g. an issue tracker, would have predefined fields for each requirement, thereby functioning like a template.* *Quality criteria: priority, unique ID, product version, acceptance criteria, rationale, template, only relevant comments, no contradiction/conflict, INVEST, consistent/defined terminology, correct language, forward traceable, no duplicates, navigable and explained links.*
[L03] Lee et al. (2003)	An Agile Approach to Capturing Requirements and Traceability	7	This paper presents a tool-based approach that provides for the implicit recording and management of relationships between conversations about requirements, specifications, and design decisions.

[ID] Reference	Title and Venue	#pp.	Summary and Relation to Quality Criteria
	(Workshop on Traceability in Emerging Forms of Software Engineering)		*[L03] is of low relevance for our research question as it focuses on a tool for traceability. However, it mentions forward traceability (e.g., to tests and source code) as an important quality aspects of agile requirements.* *Quality criteria: forward traceable.*
[L14a] Liskin et al. (2014a)	Why We Need a Granularity Concept for User Stories (XP)	16	This paper investigates Expected Implementation Duration of a user story as a characteristic of granularity by conducting a study with software engineering practitioners. Many developers have experienced certain problems to occur more often with coarse user stories. The findings emphasize the importance to reflect on granularity when working with user stories. *[L14a] and [L14b] both focus on the usefulness of Expected Implementation Duration (EID) as a quality criterion for user stories. They observe that user stories should be kept small in terms of this EID.* *Quality criteria: small/simple.*
[L14b] Liskin et al. (2014b)	Understanding the Role of Requirements Artifacts in Kanban (Workshop on Cooperative and Human Aspects of Software Engineering)	8	This paper explores how to utilize requirements artifacts effectively, what their benefits and challenges are, and how their scope granularity affects their utility. It studies a software project carried out in the University of Helsinki. The requirements artifacts were investigated and then developers and the customer were interviewed about their experiences. *See [L14a].*
[P14] Power (2014)	Definition of Ready: An Experience Report from Teams at Cisco (XP)	8	This paper presents an example of definition of ready used by agile teams in Cisco. These teams have developed three levels of ready that apply for user stories, sprints and releases. The paper describes how definition of ready provides a focus for backlog refinement, and some consequences of not meeting definition of ready. The paper finishes with perspectives from different roles in the organization.

[ID] Reference	Title and Venue	#pp.	Summary and Relation to Quality Criteria
			[P14] gives an example of the definition of ready (when are user stories ready to start implementing them?) for Cisco teams. This list of criteria mainly contains process-related items, but also some product-related: non-functional requirements should be identified, acceptance criteria should be identified, dependencies should be identified, size should have been estimated, team knows what it will mean to demo the user story (i.e. verifiable). *Quality criteria: non-functional requirements, acceptance criteria, dependencies, size, verifiable.*
[PR09] Patel and Ramachandran (2009)	Story Card Maturity Model (SMM): A Process Improvement Framework for Agile Requirements Engineering Practices (Journal of Software)	14	This paper describes an ongoing process to define a suitable process improvement model for story cards based requirement engineering at agile software development environments: the SMM (based on CMM). It also presents how the identified areas of improvement from assessment can be mapped with best knowledge based story cards practices for agile software development environments. *[PR09] focuses on story cards. The appendix contains a list of guidelines for story cards, unfortunately without a detailed description of each guideline. The guidelines are placed in a maturity framework.* *Quality criteria: priority, unique ID, non-functional requirements, acceptance criteria, story card structure, no contradiction/conflict, concise, INVEST, consistent/defined terminology, unambiguous.*
[SL09] Srinivasan and Lundqvist (2009)	Using Agile Methods in Software Product Development: A Case Study (ITNG)	6	This paper presents an in-depth case study of agile methods adoption in a software product development firm. Using a mix of interviews, observation and archival data, the evolution of agile adoption within the firm is reconstructed.

[ID] Reference	Title and Venue	#pp.	Summary and Relation to Quality Criteria
			[SL09] is a short case study of the use of agile software development in a small company. They do not specifically mention requirements quality criteria, but they observe some problems that occurred because of the ambiguity of the requirements. *Quality criteria: unambiguous.*
[SS05] Sillitti and Succi (2005)	Requirements Engineering for Agile Methods (Engineering and Managing Software Requirements)	18	This paper introduces Agile Methods as the implementation of the principles of the lean production in software development. It discusses several agile practices that deal with requirements. These practices focus on a continuous interaction with the customer to address the requirements evolution over time, prioritize them, and deliver the most valuable functionalities first. *[SS05] is a general overview paper on requirements engineering for agile methods. The focus is not specifically on quality, but [SS05] mentions some criteria*. *Quality criteria: priority, non-functional requirements, independent, small/simple, customer language.*

6 Conclusion

In this thesis we have explored the quality of just-in-time (JIT) requirements. As our main topic of interest we analyzed open source feature requests. We started out by a case study in open source projects. This made us see that many duplicate requests exist. Our assumption was that it must be difficult for users to get an overview of all existing feature requests when they enter a new feature request. A first step towards generating or visualizing such an overview would be to detect related feature requests. We presented a possible solution for detecting these so-called horizontal traceability links between feature requests by using a Vector Space Model.

Our next step was to explore more generally which quality criteria ('no duplicates' is one of them) apply to feature requests. We developed a framework for this and instantiated the framework for open source feature requests. We also provided guidelines for customizing the framework for other types of JIT requirements. we concluded this thesis with a systematic literature review on quality criteria for agile requirements as a way to confirm that our framework is not missing any important quality criteria.

Below we present our contributions, answers to research questions and suggestions for future work. We also discuss our research in light of the evaluation criteria from Wieringa et al. (2005).

6.1 Summary of Contributions

The main contributions of this dissertation are:

- A categorization of duplicate feature requests in issue trackers in open source projects (Chapter 2).
- A method to identify horizontal traceability links for feature requests (Chapter 3)
- A framework for quality criteria for just-in-time requirements (Chapter 4)
- A quality score for requirements in open source projects (Chapter 4)
- An overview of literature on quality of agile requirements specifications (Chapter 5)

- Recommendations for practitioners working on quality of agile requirements (Chapter 5)
- A research agenda on quality of agile requirements (Chapter 5)

6.2 The Research Questions Revisited

The goal of this thesis was to obtain a deeper understanding of the notion of quality for just-in-time requirements. In this section we will summarize the results obtained in this thesis by re-discussing each of the main research questions and the main threats to validity before we present our final conclusion.

RQ2: How can we assist the users as the main actors in the requirements evolution process in open source projects?

In Chapter 2 we have investigated requirements evolution in 20 open source community web sites and saw that in most cases an issue tracker is used to evolve requirements (which we call feature requests). Within those web sites that use Bugzilla as a requirements management tool we have observed a high number of duplicate feature requests (up to 36%) in the open source projects we analyzed. To explain those duplicate requests we manually analyzed the duplicate requests of six projects to determine why these duplicates were created. This resulted in seven categories of duplicates. Using this classification we have suggested improvements for manual search and creation of feature requests ([R1] till [R8]) and seven opportunities for tool support to avoid duplicates, e.g. improved linking mechanisms, visualization of those links, clustering or advanced search.

The case study we did to answer this first question made us realize that there are quality problems (e.g. too many duplicates) with feature requests and inspired us to further investigate the possibility to visualize so-called feature request networks. As said, having those networks helps users to avoid adding duplicate requests. For visualizing the networks we need, amongst others, horizontal traceability links between feature requests. This is the subject of our next research question.

RQ3: Can TF-IDF help to detect horizontal traceability links for feature requests?

In Chapter 3 we applied a Vector Space Model (VSM) with 'Term Frequency - Inverse Document Frequency' (TF-IDF) to three open source projects (Mylyn Tasks, ArgoUML and NetBeans). We showed that a VSM with TF-IDF can be beneficial to detect related feature requests. In our experiment, on the one hand we retrieved already known relations between feature requests, while on the other hand we retrieved feature requests pairs that were previously not marked as related, but that are confirmed to be related by project members after we confront them with our results. We explain our results for one example of a feature request network. We re-

peated our analysis on the Netbeans project (the larger of the three projects) with Latent Semantic Analysis (LSA) instead of TF-IDF. The results of our experiment with LSA show us that our initial assumption (that LSA would improve results as it takes into account e.g. synonyms) does not hold for the Netbeans case.

We see that TF-IDF can help to detect horizontal traceability links and thereby serve as a first step in visualizing feature request networks. As said these networks could help to reduce the number of duplicate feature requests, thereby improving feature request quality. After this analysis we decided to move away from the feature request networks to answer the more general question of what would be applicable quality criteria for feature requests specifically and JIT requirements in general.

RQ4: Which criteria should be used for the quality assessment of just-in-time requirements?

In Chapter 4 we have developed a framework for quality criteria for JIT requirements based on earlier work on traditional upfront requirements, our experience with feature requests in open source projects and analysis of literature on just-in-time requirements. The framework was positively evaluated by practitioners. We have instantiated the framework for feature requests in open source projects and indicated how it can be customized for other types of JIT requirements. When comparing our framework to known quality criteria for traditional requirements we conclude that we have defined 8 instances of 'additional criteria' or 'new' interpretations of existing criteria. And of course we have added the time dimension to each of the criteria by specifying if it should hold at creation-time (*C) or just-in-time (*J).

We applied our framework to three open source projects (Mylyn Tasks, ArgoUML and Netbeans). From this we have distilled a set of recommendations that makes this research actionable for practitioners.

RQ5: Which are the known quality criteria for agile requirements specifications?

In Chapter 5 we report on a systematic literature review on informal verification of agile requirements specifications. Through a structured process we selected 16 papers. From these papers we devised an overview of existing quality attributes (using the framework from Chapter 4) that can be used in informal verification. We also derived (1) recommendations for practitioners that want to perform informal verification of requirements specifications in an agile setting and (2) a research agenda for academics that highlights the need for further research in this area. In addition, the resulting systematic overview is useful as a reference work for researchers in the field of informal verification and helps them identify both related work and new research opportunities in this field.

The systematic literature review also serves to validate the framework (and its instantiations) in Chapter 4. The results of the literature review did not make us change anything on our framework.

Threats to Validity

This thesis investigates the quality of JIT requirements largely by investigating open source feature requests. Feedback from industry practitioners and findings in literature indicate that other types of JIT requirements have a similar notion of quality. That is why we present our findings in the form of a framework. The framework itself consists of three categories and a timing notion, for which we assume that it is usable for all types of JIT requirements. We present an instantiation of the framework for open source feature requests and indicate how it could be customized for e.g. user stories. However, future work remains to validate this in case studies in agile or closed source environments.

Another threat to validity is related to the fact that we assume (as in traditional requirements engineering) that the quality of JIT requirements is important for the quality, cost or duration of the developed software product. This in general is confirmed by practitioners we interviewed and literature (e.g. Fitzgerald et al. (2012)). We can however only assume that the application of our framework has the same positive effects on the developed software product. Future work remains to apply the framework in practice to get quantitative feedback about the impact of it.

A final point is that in this thesis we only consider quality of *written* JIT requirements. We feel that this is a valid choice to make, because in our experience we encountered many cases in which oral communication of JIT requirements was not possible. This is also confirmed by literature (e.g. Inayat et al. (2014)). The framework (three categories and timing notion) we present in our opinion also holds for oral JIT requirements but it remains yet to be investigated which specific quality criteria would apply to these oral JIT requirements.

Conclusion

To summarize the quality of JIT requirements can be defined by our framework with three main categories (Completeness, Uniformity and Correctness/Consistency), for ease of reference repeated in Figure 6.1. The criteria in Figure 6.1 are the ones we derived for feature requests and user stories. The framework consist of the three categories and a timing dimension, but the criteria in each of the categories should be selected and adjusted based on the specific JIT environment (i.e. “just-enough”). Also for each selected criterion it should be specified at which point in time the criterion should hold (e.g. at creation time or just before implementation starts, i.e. “just-in-time”). In a few words we would summarize quality of JIT requirements as “Just-Enough and Just-in-Time”.

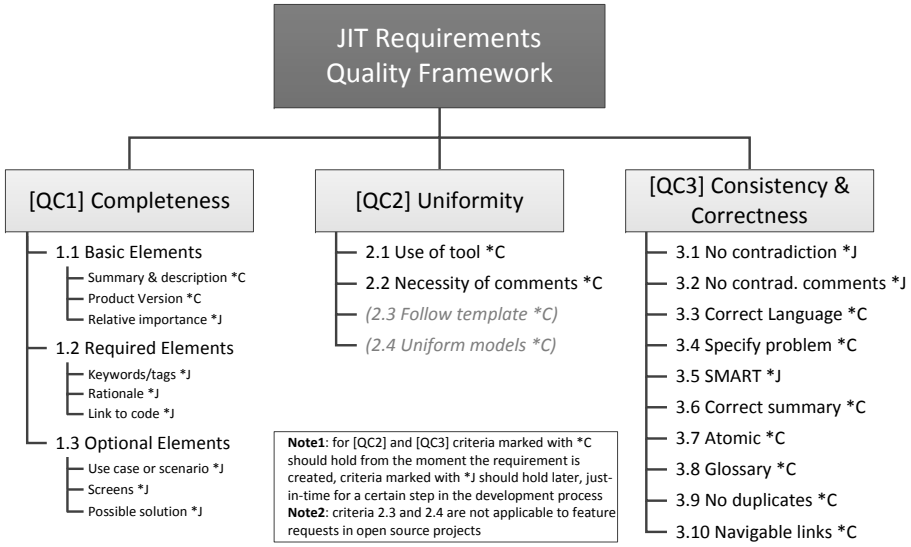

Figure 6.1: Just-in-time requirements quality framework (instantiated for feature requests in open source projects and user stories)

6.3 Requirements Engineering Research Evaluation Criteria

In this section we will discuss our research in light of the criteria proposed by Wieringa et al. (2005). In their paper Wieringa et al. present an engineering cycle and a paper classification. The engineering cycle contains six steps: 1) Problem investigation; 2) Solution design; 3) Solution validation; 4) Solution selection; 5) Solution implementation; 6) Implementation evaluation. For both lines of research we only conducted the first three steps. We did not evaluate or compare different solutions for the same problem and unfortunately we did not go into implementing our solutions in agile or JIT environments to be used in daily practice. However, we have gone to great extent to validate our proposed solutions in a case study (open source projects) setting and to get confirmation from practitioners. We have also gathered recommendations for practitioners based on our experiences in the case studies. As a result this thesis should be classified as a combination of "Proposal of solution" and "Validation research" (Wieringa et al. 2005).

In their paper, Wieringa et al. mention a number of evaluation criteria for these two paper classes, see Table 6.1. We now briefly discuss our thesis in light of each of these evaluation criteria.

As we noted in the introduction we can distinguish two lines of research in our thesis: 1) duplication and horizontal traceability of feature requests; 2) a quality framework for JIT requirements (Chapter 4). For those two lines of research Chapter 3 and 4 each describe both the "Proposal of solution" as well as the "Validation research". This is why we focus on those two chapters here. In what follows each

Table 6.1: Paper evaluation criteria, taken from Wieringa et al. (2005)

	"Proposal of Solution" Papers		**"Validation Research" Papers**
P1.	Is the problem to be solved clearly explained?	V1.	Is the technique to be validated clearly described?
P2.	Is the technique or its application to this kind of problem novel?	V2.	Are the causal or logical properties of the technique clearly stated?
P3.	Is the technique sufficiently well described to be validated in later research?	V3.	Is the knowledge claim validated?
P4.	Is the technique sound?	V4.	Is the research method sound?
P5.	Is the broader relevance of the technique argued?	V5.	Is it clear under which circumstances the technique has the stated properties?
P6.	Is there sufficient discussion of related work?	V6.	Is there sufficient discussion of related work?
		V7.	Is this a significant increase in knowledge about this technique?

of the chapters is discussed and where applicable the criterion from Table 6.1 is mentioned between square brackets.

General Points

Each chapter in this thesis starts with a description of the context of the problem to be solved [P1], after which the problem itself is defined in one main research question with (three) subsidiary questions. The same is true for the main thesis, which sketches a context and defines the main goal for the thesis in the Introduction chapter.

For each of the chapters and the main thesis we also build a clear path from research questions to conclusion: 1) we introduce the questions; 2) we describe the setup of our research; 3) we describe the detailed results per research question; 3) we discuss each of the research questions briefly; 4) we present limitations or threats to validity of our research; 5) we conclude by summarizing results and contributions; 6) we present future work. Through this uniform setup we ensure that any claims we make in the conclusion are supported by the detailed results we gathered while analyzing the research questions [V3].

We limited our scope to feature requests in open source projects. This is due to circumstances (availability of material for analysis) and in our opinion is one of the main limitations of this thesis [V5]. To mitigate this limitation, where applicable, we indicate if and how our results could be translated to other types of JIT requirements. Furthermore we have included opportunities for future work related to this

limitation in Section 6.4.

For all chapters and the main thesis we included a section about 'Related Work' [P6, V6]. The last chapter with the SLR could be seen as one big 'Related Work' chapter for the entire thesis. The problem of this thesis is, however, that the body of related work is quite small. We indicate this in each of the related work sections and we cite some work from related areas like 'Quality Assurance in Agile' or 'Agile Requirements Engineering'.

Chapter Three: Horizontal Traceability of Feature Requests

The application of TF-IDF to horizontal traceability for feature requests is the application of an existing technique, but the application to this kind of problem is novel [P2, P4]. We specifically discuss the relevance of the technique for this application [P5]. We have described the main algorithms in the tool and indicated that source code is available [V1]. We summarized the main points from previous work about the techniques we are using [V2]. The chapter has a section about the setup of the experiment and we made our dataset available on-line on Figshare (`figshare.com`): (Heck and Zaidman 2014a) [P3]. We discussed limitations of our experiment at the end of the chapter [V4]. As we mention, a next step would be to get measures on thresholds, recall and precision for the retrieval of the traceability links, by using (industrial) case studies where we are able to identify all those links beforehand. Chapter 3 teaches us that [V7]: 1) a simple technique like TF-IDF can deliver good results for detecting horizontal traceability links; 2) the results for using TF-IDF on just feature requests differ from using it on bug reports in general. As said before, we did not explore this path any further.

Chapter Four: Quality Framework for JIT Requirements

Our framework as described in Chapter 4 can be seen as a novel technique, but we argue that this is the application of existing quality criteria to a new area (thus that only the application to this kind of problem is novel) [P2,P4]. We have described the complete framework and indicated how to customize it [V1]. We summarized the main points from the previous work we are building upon [V2]. We describe the setup of the case study and the survey and we made our dataset available on-line on Figshare (`figshare.com`): (Heck 2014) [P3]. We used a three-step process to get a feeling for the viability of our framework [V4]: 1) we studied literature on agile and JIT requirements; 2) we interviewed practitioners to obtain their opinion; 3) we asked final-year students to apply the framework. This constitutes an initial evaluation of our framework. We argued that the framework should be usable in all JIT environments and we even included a special section on the customization of the framework [P5]. We applied the framework in real open source projects and we got positive feedback from practitioners about the framework and its relevance [P5, V7]. Without this validation we would only have related work to base our analysis on, which makes for much less of a strong argument. We would also like to point

out that the SLR in Chapter 5 is a second way of validating our framework [V7]. At the same time Chapter 5 presents gathered knowledge about recommendations for practitioners and researchers working in the area of JIT requirements quality [V7].

6.4 Recommendations for Future Work

There are a number of interesting open issues for future work related to the topic of this thesis. In the following we briefly discuss some aspects of our thesis and suggest several recommendations:

1. Incorporate Visualization of Feature Request Networks into Issue Trackers

In Chapter 3 we designed a tool that can detect horizontal traceability links between feature requests. In this Chapter we also made use of links between feature requests that are present in the issue tracker. The issue trackers we know can only visualize parts of this so-called feature request networks (e.g. just the children of the current feature request). This makes it infeasible for the user to explore the network (as it would require clicking through children of children and then parents of parents to get back to the starting point). We have indicated the development of a complete graph-based visualization tool/component for feature request networks as an important recommendation in both Chapter 2 and Chapter 3. However, in this thesis we did not follow up on this recommendation, in favor of developing a framework for quality criteria for JIT requirements. We would like to emphasize this point here because we still think that such a visualization tool could be of great use to people involved in projects with a large database of individual requirements. This is also confirmed by Baysal et al. (2014) and by Fitzgerald et al. (2012) who state that "the large number of feature requests and poor structuring of information make the analysis and tracking of feature requests extremely difficult for project managers; this affects the quality of communication between project managers and stakeholders and makes it hard for project managers to identify stakeholders' real needs. Consequently, various problems may arise later in the features development life-cycle".

2. Replicate Case Studies in a Closed Source Environment

This thesis investigates quality of JIT requirements. Out of practical considerations we have used open source feature requests as our main source of study. According to Alspaugh and Scacchi (2013) "closed source software bug reports and feature requests and the process for managing them look much like those for open source software", so we expect our results to hold in both cases. In Chapter 4 we have interviewed 8 practitioners from a closed source environment. These interviews, our own experience in industry projects and the other contacts we had with industry indicate to us that in fact there is a practical use for the framework we developed in industry or closed source projects. Our analysis in Chapter 5 also includes literature describing case studies in closed source projects, showing that there is no specific

difference between the two. However, we still deem it interesting to repeat our case studies from Chapter 3 (text-based discovery of requirements networks) and Chapter 4 (quality score for JIT requirements) in closed source environments to see if similar results can be achieved.

3. Replicate Case Studies in an Agile Environment

Related to the previous recommendation is the fact that in this thesis we have mostly considered feature requests. In agile environments a more common format for requirements is the so-called 'user story', see also Chapter 5. Section 4.3 describes the characteristics of user stories and indicates quality criteria that should hold for them. In this section we argue that most criteria that we defined for feature requests are also valid for user stories. This opinion was also shared by the 8 practitioners from the Dutch agile community that we interviewed in Chapter 4. Also in Chapter 5 no specific quality criteria for user stories stand out. According to Koch (2004) open source projects and agile projects have many similarities. The main difference he sees is the "team co-location and personal contact demanded by agile development, which is not seen as a precondition in open source development". This actually is the main reason we studied open source projects as the absence of co-location results in a large body of on-line requirements documentation. Corbucci and Goldman (2010) conducted a survey amongst developers working in both agile and open source communities and found that "the communities themselves are not that close to each other even if the mindset is quite similar". These somewhat contradicting conclusions would make it interesting to replicate especially the case study from Chapter 4 (quality score for JIT requirements) in agile environments with user stories to see if there are differences with open source feature requests.

4. Investigate Further the Impact of Good/Bad JIT Requirements Quality

In this thesis we have built upon the generally accepted idea that quality of requirements is an important factor for successful software development (see e.g. (Denger and Olsson 2005)). In our introduction we quoted Cockburn (2000): "the better the internal communications between usage experts and developers, the lower the cost of omitting parts of the use case template". One could indeed argue that for agile projects the quality problems in the requirements are being compensated by extensive face-to-face contact or prototyping (customers see what they will get and developers can just ask the customer for clarifications), so quality of written JIT requirements might be a less important aspect than in traditional environments. However Cockburn (2000), Inayat et al. (2014) and Nawrocki et al. (2014) describe that also in agile environments a lot of situations exist where face-to-face contact is not feasible, weakening this argument. Rubin and Rubin (2011) confirm this view and explicitly state that "due to the general trend toward globalization, documentation becomes increasingly important since in physically separated development teams, communication becomes even more difficult, especially in agile development processes". The 8 practitioners that we interviewed in Chapter 4 also

confirm that quality of JIT requirements (of what is written) is an important aspect in their daily work.

Radliński (2012) conducted a preliminary investigation into the impact of requirements engineering on software quality. One of his findings was that some factors like having a requirement specification positively influence some quality aspects like 'Speed of Designing/Providing'. In our experience the same is true for agile or JIT environments. Although these environments allow for easier correction of requirements mistakes, doing it right the first time can save time and money. Eberlein and Leite (2002) confirm this by stating: "The agile community claims that they do tackle requirements, but we think that this is poorly performed requiring more validation cycles than necessary and relying too much on individuals. This, in the long run, may bring severe problems to the software organization responsible for software built following an agile method". For example, Fitzgerald et al. (2012) conducted a study of feature requests failure in seven large projects. They state that defects in the description of a feature request may cause faults in the decision to either reject or accept a feature request. According to Fitzgerald et al., these faults may, in turn, cause 5 different types of failures in the product or development process: product failure, abandoned development, rejection reversal, stalled development, removed feature.

We deem it necessary to conduct JIT-specific experiments to gather more data on the impact of good/bad JIT requirements quality on the rest of the software development process and the final product quality. In this way we would have more quantitative data to back up our assumptions.

5. Tool Support for Checking JIT Requirements Quality

In Chapter 4 we have conducted a case study with 86 software engineers that each manually inspected the quality of 20 open source feature requests. Our feeling is that a number of the checks they had to do could be automated or at least supported with tools. E.g. the presence of words like 'and, or, besides' could indicate a requirement that is not atomic. Some tools in this direction have already been developed for traditional requirements (e.g. (Gnesi et al. 2005), (Kiyavitskaya et al. 2008), (Verma and Kass 2008)) and can easily be adjusted to use in a JIT environment. Some criteria in our framework are subjective (see Section 4.6), but it is an interesting open question if we could develop automated support to reduce the subjectivity of the check. Having a tool or automated support for checking JIT requirements quality would make it even more feasible to include the quality checks into the daily working practice of all agile/JIT development teams. As Eberlein and Leite (2002) state "simple tools to check early requirements descriptions associated with effective management practices for applying inspections, of course upon availability of check-lists, can improve the quality of agile processes, which usually only rely on validation.".

Appendix: Tables for Chapter 2

Table A1: Analysis of Apache HTTPD duplicate feature requests (part 1 of 2)

Id	ChildSummary	Id	ParentSummary	Search	D	P	F	Type	Comment
8516	ScriptAction to invoke a file instead of a URL path?	7483	Add FileAction directive to assign a cgi interpreter other than using registry or shebang?	“script action file"	N	N	Y	Duplic. solution	Marker wrowe discusses child first and spots duplicate three months later only
9792	Introduce DESTDIR in Makefiles	7803	make install root= from apache 1.3 doesn’t work in 2.0.35	“DESTDIR"	Y	Y	Y	Patch	Parent is defect
10145	ShebangAlias config directive - to keep to make CGI scripts more portable	7483	Add FileAction directive to assign a cgi interpreter other than using registry or shebang?	“shebang cgi"	Y	N	Y	Wording	Last vote on parent in 2003 but still ASSIGNED status
12241	adding svg and ico mime-types	10993	Missing MIME type image/x-icon	“svg ico mime"	Y	Y	Y	Patch	N/A
13608	Catch-all vhost	13607	Catch-all enhancement for vhost_alias?	N/A	Y	N	Y	Author	Marked by two persons at the same time (but different direction)
14335	AddOutputFilterByType doesn’t work with proxy requests	31226	AddOutputFilterByType deflate not active with mod_proxy	“AddOutputFilterByType"	Y	N	N	Defect	Parent is a defect that is reported later than child
15965	make uninstall for non default source installations	11259	‘make install’ should provide a deinstaller	N/A	Y	N	Y	Wording	Uninstall vs. Deinstall
16391	compressing logs using rotatelogs	28815	ErrorLog piped command doesn’t handle redirections	N/A	N	N	N	Duplic. solution	Same patch for two different problems; parent is a defect
17800	use of tags in index generation makes invalid HTML	9307	mod_autoindex generates bad HTML	“<pre>"	Y	N	Y	Wording	Parent is defect WONTFIX
21907	suggest that one can set ExpiresByType while other files do not set expires header	23748	“internal error: bad expires code" if expires criteria aren’t met	“ExpiresByType"	Y	Y	N	Patch	Patch with parent; parent was created later
25444	ProxyPassReverse always rewrites redirects to same scheme as request	21272	Support for full fixed reverse mappings	“ProxyPassReverse"	Y	N	Y	Wording	N/A
28561	DOCUMENT_ROOT is not using VirtualDocumentRoot	26052	DOCUMENT_ROOT environment variable set incorrectly with VirtualDocumentRoot	“DOCUMENT_ROOT VirtualDocumentRoot"	Y	N	Y	No check done	Parent has more duplicates (defects); much discussion on defect or enhancement
29260	new v-hosting features desired	40441	intelligently handling dynamic subdomains	N/A	N	N	N	Partial match	Part of request is answered in parent 3 years later which has totally different main question
29511	Adding virtualroot for AuthUserFile Directive	25469	create AuthRoot for defining paths to auth files	“AuthUserFile"	Y	N	Y	Wording	N/A
30173	Extending FakeBasicAuth	20957	New option SSLUserName	“FakeBasic Auth"	Y	Y	Y	Patch	N/A
31383	OCSP support	41123	Support of OCSP in mod_ssl (rewritten patch from bug #31383)	N/A	Y	Y	N	Version	Created on purpose with link to store patch for new version (see title)

D = Duplicate?; P = Patch included by author?; F = Parent created before child?

Table A2: Analysis of Apache HTTPD duplicate feature requests (part 2 of 2)

Id	ChildSummary	Id	ParentSummary	Search for	D	P	F	Type	Comment
35805	New require attribute for mod_auth_ldap	31352	RFE, Bind to LDAP server with browser supplier user/pass	N/A	N	Y	Y	Duplic. solution	May be found when looking into all enhancements for component; patch can be modified and applied; author has marked as duplicate
37287	Optionally make mod_auth return HTTP_FORBIDEEN for failed login attempts	40721	401 vs 403 in httpd	“401 instead of 403"	Y	N	N	Wording	Marking is 6 years after creation
38153	Add Filename To A Few Errors In dav_method_put	38149	Add Filename To A Few Errors In dav_method_put	N/A	Y	N	Y	Author	N/A
42341	chroot patch directly after child creation	43596	Chroot patch	“chroot"	Y	Y	N	Patch	Both parent and child have a patch and authors comment on each other
42557	Missing parameter to control LDAP referral chasing	40268	Credentials are not supplied when connecting to LDAP referrals	“LDAP referral"	Y	Y	Y	Patch	Component differs mod_ldap vs. mod_authz_ldap; parent is defect
47051	Subject Alternative Name not used while checking certificate	32652	mod_ssl: match hostnames against subjectAltName DNS names too	"X509" returns long list, alternative: search in mod_ssl for previous enhancements	Y	Y	Y	Patch	Subject Alternative Name vs. subjectAltName
48841	mod_proxy: Allow loadfactor of 0.	51247	Enhance mod_proxy and _balancer with worker status flag to only accept sticky session routes	N/A	N	Y	N	Duplic. solution / Patch	Both describe a patch with a different solution for the same problem (remove server from load balancer); duplicate was detected by kmashint (parentReporter)
49076	Skip document root check during start up (-T option)	41887	-T option unavailable for Apache 2.0.x/2.3.0 ->available	N/A	Y	Y	Y	Patch / Wrong	Wrong behavior to create new request for patch for new version; author refers to parent
50555	Allow mod_status to show HTTP Host header instead of vhost name	45148	The actual host of the request will be more helpful in mod_status, instead of the server name	“Host header" with component mod_status	Y	Y	Y	Patch	Search needed in both summarry and comments
50714	Enhance AllowCONNECT directive in mod_proxy to allow wildcard	23673	AllowCONNECT cannot be configured to allow connections to all ports	“AllowConnect"	Y	N	Y	Wording	N/A
54002	[PATCH] FallbackResource cannot be disabled when it was enabled in a parent directory	54003	[PATCH] FallbackResource cannot be disabled when it was enabled in a parent directory	N/A	Y	N	Y	Author	N/A

D = Duplicate?; P = Patch included by author?; F = Parent created before child?

Bibliography

Aberdour, M. (2007). Achieving quality in open-source software. *Software, IEEE*, 24(1):58–64.

Alspaugh, T. and Scacchi, W. (2013). Ongoing software development without classical requirements. In *Int'l Req. Engineering Conference (RE)*, pages 165–174.

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. (2008). Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304–23:318, New York, NY, USA. ACM.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. (2002). Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983.

Antoniol, G., Canfora, G., Casazza, G., and Lucia, A. D. (2000). Identifying the starting impact set of a maintenance request: A case study. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 227–230. IEEE.

Auger, C. P. (1994). *Information Sources in Grey Literature*. Bowker-Saur.

Baysal, O., Holmes, R., and Godfrey, M. W. (2014). No issue left behind: Reducing information overload in issue tracking. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 666–677. ACM.

Beck (2002). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development. Available from `http://agilemanifesto.org`. Last visited: April 16th, 2015.

Belsis, P., Koutoumanos, A., and Sgouropoulou, C. (2014). PBURC: a patterns-based, unsupervised requirements clustering framework for distributed agile software development. *Requirements Engineering*, 19(2):213–225.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2008a). What makes a good bug report? In *Int'l Symp. on Foundations of Software Engineering (FSE)*, pages 308–318. ACM.

Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. (2008b). Duplicate bug reports considered harmful ... really? In *Proc. Int'l Conf. on Software Maintenance (ICSM)*, pages 337–345. IEEE.

Bhasin, S. (2012). Quality assurance in agile: A study towards achieving excellence. In *AGILE India (AGILE INDIA), 2012*, pages 64–67.

Binkley, D. and Lawrie, D. (2011). Maintenance and evolution: Information retrieval applications. In *Encyclopedia of Software Engineering*, chapter 49, pages 454–463. Taylor & Francis LLC.

Bjarnason, E., Unterkalmsteiner, M., Engström, E., and Borg, M. (2015). An industrial case study on test cases as requirements. In Lassenius, C., Dingsøyr, T., and Paasivaara, M., editors, *Agile Processes, in Software Engineering, and Extreme Programming*, volume 212 of *Lecture Notes in Business Information Processing*, pages 27–39. Springer International Publishing.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice-Hall.

Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., and Khalil, M. (2007). Lessons from applying the systematic literature review process within the software engineering domain. *J. Syst. Software*, 80(4):571–583.

Cao, L. and Ramesh, B. (2008). Agile requirements engineering practices: An empirical study. *IEEE Software*, 25(1):60–67.

Cavalcanti, Y. C., da Cunha, C. E. A., de Almeida, E. S., and de Lemos Meira, S. R. (2009). BAST - a tool for bug report analysis and search. In *XXIII Simpósio Brasileiro de Engenharia de Software (SBES)*, Fortaleza, Brazil.

Cavalcanti, Y. C., da Mota Silveira Neto, P. A., Lucrédio, D., Vale, T., de Almeida, E. S., and de Lemos Meira, S. R. (2013a). The bug report duplication problem: an exploratory study. *Software Quality Journal*, 21(1):39–66.

Cavalcanti, Y. C., da Mota Silveira Neto, P. A., Machado, I. d. C., Vale, T. F., de Almeida, E. S., and de Lemos Meira, S. R. (2013b). Challenges and opportunities for software change request repositories: a systematic mapping study. *Journal of Software: Evolution and Process*, 26(7):620–653.

Cleland-Huang, J. (2012). Traceability in agile projects. In *Software and Systems Traceability*, pages 265–275. Springer London.

Cleland-Huang, J., Czauderna, A., Gibiec, M., and Emenecker, J. (2010). A machine learning approach for tracing regulatory codes to product specific requirements. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 155–164. ACM.

Cleland-Huang, J., Dumitru, H., Duan, C., and Castro-Herrera, C. (2009). Automated support for managing feature requests in open forums. *Commun. ACM*, 52(10):68–74.

Cleland-Huang, J., Gotel, O., and Zisman, A., editors (2012). *Software and Systems Traceability*. Springer.

Cockburn, A. (2000). *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Corbucci, H. and Goldman, A. (2010). Open source and agile methods: Two worlds closer than it seems. In Sillitti, A., Martin, A., Wang, X., and Whitworth, E., editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 48 of *Lecture Notes in Business Information Processing*, pages 383–384. Springer Berlin Heidelberg.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702.

Creswell, J. W. (2013). *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.

Dalle, J.-M. and den Besten, M. (2010). Voting for bugs in Firefox: A voice for mom and dad? In *OSS*, volume 319 of *IFIP Advances in Information and Communication Technology*, pages 73–84. Springer.

Davis, A. and Hickey, A. (2009). A quantitative assessment of requirements engineering publications – 1963-2008. In Glinz, M. and Heymans, P., editors, *Requirements Engineering: Foundation for Software Quality*, volume 5512 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin Heidelberg.

Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., and Theofanos, M. (1993). Identifying and measuring quality in a software requirements specification. In *Int'l Software Metrics Symposium*, pages 141–152.

De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., and Panichella, S. (2013). Applying a smoothing filter to improve IR-based traceability recovery processes: An empirical investigation. *Inf. Softw. Technol.*, 55(4):741–754.

De Lucia, A., Fasano, F., and Oliveto, R. (2008). Traceability management for impact analysis. In *Frontiers of Software Maintenance (FoSM)*, pages 21–30. IEEE.

De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G. (2006). Can information retrieval techniques effectively support traceability link recovery? In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 307–316.

De Lucia, A. and Qusef, A. (2010). Requirements engineering in agile software development. *Journal of Emerging Technologies in Web Intelligence*, 2(3).

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407.

Denger, C. and Olsson, T. (2005). Quality assurance in requirements engineering. In Aurum, A. and Wohlin, C., editors, *Engineering and Managing Software Requirements*, pages 163–185. Springer Berlin Heidelberg.

Desharnais, J.-M., Kocaturk, B., and Abran, A. (2011). Using the COSMIC method to evaluate the quality of the documentation of agile user stories. In *Software Measurement, 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 269–272.

Dietze, S. (2005). Agile requirements definition for software improvement and maintenance in open source software development. In *Proc. Int'l Workshop on Situational Requirements Engineering Processes*, pages 176–187.

Dingsoyr, T., Dyba, T., and Abrahamsson, P. (2008). A preliminary roadmap for empirical research on agile software development. In *Agile, 2008. AGILE '08. Conference*, pages 83–94.

Doran, G. T. (1981). There's a SMART way to write management's goals and objectives. *Management Review*, 70(11):35–36.

Dumais, S. T. (2004). Latent semantic analysis. *Annual Review of Information Science and Technology*, 38(1):188–230.

Dumas-Monette, J.-F. and Trudel, S. (2014). Requirements engineering quality revealed through functional size measurement: An empirical study in an agile context. In *Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2014 Joint Conference of the International Workshop on*, pages 222–232.

Duncan, R. (2001). The quality of requirements in extreme programming. *CrossTalk, June*, 19:22–31.

Dybå, T. and Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Inf. Softw. Technol.*, 50(9-10):833–859.

Eberlein, A. and Leite, J. (2002). Agile requirements definition: A view from requirements engineering. In *Proceedings of the International Workshop on Time-Constrained Requirements Engineering (TCRE'02)*, pages 4–8.

Ernst, N. A., Borgida, A., Jureta, I. J., and Mylopoulos, J. (2014a). Agile requirements engineering via paraconsistent reasoning. *Information Systems*, 43:100–116.

Ernst, N. A., Borgida, A., Jureta, I. J., and Mylopoulos, J. (2014b). An overview of requirements evolution. In *Evolving Software Systems*, pages 3–32. Springer.

Ernst, N. A. and Murphy, G. (2012). Case studies in just-in-time requirements analysis. In *Int'l Workshop on Empirical Requirements Engineering*, pages 25–32. IEEE.

Ernst, N. A., Mylopoulos, J., and Wang, Y. (2009). Requirements evolution and what (research) to do about it. In *Design Requirements Engineering: A Ten-Year Perspective*, volume 14 of *LNBIP*, pages 186–214. Springer.

Farid, W. and Mitropoulos, F. (2013). NORPLAN: Non-functional requirements planning for agile processes. In *Southeastcon, 2013 Proceedings of IEEE*, pages 1–8.

Firesmith, D. (2004). Generating complete, unambiguous, and verifiable requirements from stories, scenarios, and use cases. *Journal of Object Technology*, 3(10):27–40.

Fitzgerald, C., Letier, E., and Finkelstein, A. (2012). Early failure prediction in feature request management systems: an extended study. *Requirements Engineering*, 17(2):117–132.

Génova, G., Fuentes, J., Llorens, J., Hurtado, O., and Moreno, V. (2013). A framework to measure and improve the quality of textual requirements. *Requirements Engineering*, 18(1):25–41.

Gnesi, S., Fabbrini, F., Fusani, M., and Trentanni, G. (2005). An automatic tool for the analysis of natural language requirements. *CRL Publishing: Leicester*, 20:53–62.

Gotel, O., Cleland-Huang, J., Hayes, J. H., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J. I., and Mäder, P. (2012). Traceability fundamentals. In Cleland-Huang et al. (2012), pages 3–22.

Gotel, O. C. Z. and Finkelstein, A. (1994). An analysis of the requirements traceability problem. In *Proceedings of the First IEEE International Conference on Requirements Engineering (ICRE)*, pages 94–101. IEEE.

Gottesdiener, E. and Gorman, M. (2010). Slicing requirements for agile success. *Better Software*, 2010(04).

Grau, R., Lauenroth, K., Bereza, B., van Veenendaal, E., and van der Zee, S. (2014). Requirements engineering and agile development-collaborative, just enough, just in time, sustainable.

Gu, H., Zhao, L., and Shu, C. (2011). Analysis of duplicate issue reports for issue tracking system. In *Int'l Conf on Data Mining and Intelligent Information Technology Applications (ICMiA)*, pages 86–91.

Hayes, J. H., Dekhtyar, A., and Osborne, J. (2003). Improving requirements tracing via information retrieval. *2003 11th IEEE International Requirements Engineering Conference (RE)*, 0:138.

Heck, P. (2014). JIT requirements quality framework (Figshare). `http://dx.doi.org/10.6084/m9.figshare.938214`.

Heck, P., Klabbers, M., and van Eekelen, M. C. J. D. (2010). A software product certification model. *Software Quality Journal*, 18(1):37–55.

Heck, P. and Parviainen, P. (2008). Experiences on analysis of requirements quality. In *Int'l Conf. on Softw. Eng. Advances (ICSEA)*, pages 367–372. IEEE.

Heck, P. and Zaidman, A. (2013). An analysis of requirements evolution in open source projects: Recommendations for issue trackers. In *Int'l Workshop on Principles of Software Evolution (IWPSE)*, pages 43–52. ACM.

Heck, P. and Zaidman, A. (2014a). Horizontal traceability for JIT requirements (Figshare). `http://dx.doi.org/10.6084/m9.figshare.1030568`.

Heck, P. and Zaidman, A. (2014b). Horizontal traceability for just-in-time requirements: the case for open source feature requests. *Journal of Software: Evolution and Process*, 26(12):1280–1296.

Heck, P. and Zaidman, A. (2014c). A quality framework for agile requirements: A practitioner's perspective. Technical Report TUD-SERG-2014-006, Software Engineering Research Group, Delft University of Technology.

Heck, P. and Zaidman, A. (2015a). A framework for quality assessment of just-in-time requirements. the case of open source feature requests. *Requirements Engineering Journal, Under submission*.

Heck, P. and Zaidman, A. (2015b). Quality criteria for just-in-time requirements: Just enough, just-in-time? In *Proc. JITRE*, pages 1–4. IEEE.

Heck, P. and Zaidman, A. (2016). A systematic literature review on quality criteria for agile requirements specifications. *Software Quality Journal, Under submission*.

Herzig, K., Just, S., Rau, A., and Zeller, A. (2013). Classifying code changes and predicting defects using change genealogies. Technical report, Software Engineering Chair, Saarland University, Dept. of Informatics. `https://www.st.cs.uni-saarland.de/publications/details/herzig-genealogytechreport-2011/`.

Herzig, K., Just, S., and Zeller, A. (2012). It's not a bug, it's a feature: How misclassification impacts bug prediction. Technical report, Universitaet des Saarlandes, Saarbruecken, Germany.

Huo, M., Verner, J., Zhu, L., and Babar, M. (2004). Software quality and agile methods. In *Int'l Computer Software and Applications Conference*, pages 520–525, vol.1.

IEEE (1990). IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.

IEEE-830 (1998). IEEE recommended practice for software requirements specifications. *IEEE Std 830-1998*.

IIBA (2009). A guide to the business analysis body of knowledge (BABOK Guide). *International Institute of Business Analysis (IIBA)*.

Inayat, I., Salim, S. S., Marczak, S., Daneva, M., and Shamshirband, S. (2014). A systematic literature review on agile requirements engineering practices and challenges. *Computers in Human Behavior*.

Jalbert, N. and Weimer, W. (2008). Automated duplicate detection for bug tracking systems. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 52–61.

Kabbedijk, J., Bezemer, C.-P., Jansen, S., and Zaidman, A. (2015). Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software*, 100:139–148.

Kamata, M. and Tamai, T. (2007). How does requirements quality relate to project success or failure? In *Int'l Req. Engineering Conference (RE)*, pages 69–78.

Kassab, M. (2014). An empirical study on the requirements engineering practices for agile software development. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 254–261.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, EBSE Technical Report EBSE-2007-01.

Kitchenham, B. A. (2004). Procedures for performing systematic reviews. Technical report, Technical Report TR/SE-0401, Keele University, and Technical Report 0400011T.1, National ICT Australia.

Kiyavitskaya, N., Zeni, N., Mich, L., and Berry, D. (2008). Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. *Requirements Engineering*, 13(3):207–239.

Knauss, E. and El Boustani, C. (2008). Assessing the quality of software requirements specifications. In *Int'l Req. Engineering Conference (RE)*, pages 341–342.

Ko, A. J., Myers, B. A., and Chau, D. H. (2006). A linguistic analysis of how people describe software problems. In *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC)*, pages 127–134. IEEE Computer Society.

Koch, S. (2004). Agile principles and open source software development: A theoretical and empirical discussion. In *Extreme Programming and Agile Processes in Software Engineering*, pages 85–93. Springer.

Kulshreshtha, V., Boardman, J. T., and Verma, D. (2012). The emergence of requirements networks: the case for requirements inter-dependencies. *IJCAT*, 45(1):28–41.

Lee, C., Guadagno, L., and Jia, X. (2003). An agile approach to capturing requirements and traceability. In *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 17–23.

Lee, M. (2002). Just-in-time requirements analysis—the engine that drives the planning game. In *Proceedings 3rd International Conference Extreme Programming and Agile Processes in Software Engineering*, pages 138–141.

Leffingwell, D. (2011). *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley Professional, 1st edition.

Lehman, M. M. (1984). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221.

Li, J., Zhang, H., Zhu, L., Jeffery, R., Wang, Q., and Li, M. (2012). Preliminary results of a systematic review on requirements evolution. In *Proc. Int'l Conf. on Evaluation Assessment in Software Engineering (EASE)*, pages 12–21. IEEE.

Liskin, O., Pham, R., Kiesling, S., and Schneider, K. (2014a). Why we need a granularity concept for user stories. In Cantone, G. and Marchesi, M., editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 179 of *Lecture Notes in Business Information Processing*, pages 110–125. Springer International Publishing.

Liskin, O., Schneider, K., Fagerholm, F., and Münch, J. (2014b). Understanding the role of requirements artifacts in kanban. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE 2014, pages 56–63, New York, NY, USA. ACM.

Lormans, M., van Deursen, A., and Groß, H.-G. (2008). An industrial case study in reconstructing requirements views. *Empirical Software Engineering*, 13(6):727–760.

Lucassen, G., Dalpiaz, F., Brinkkemper, S., and van der Werf, J. (2015). Forging high-quality user stories: Towards a discipline for agile requirements. *Proceedings of the IEEE International Requirements Engineering Conference*.

Lucia, A. D., Fasano, F., Oliveto, R., and Tortora, G. (2007). Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4).

Maarek, Y. S., Berry, D. M., and Kaiser, G. E. (1991). An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Software Eng.*, 17(8):800–813.

Maletic, J. I. and Marcus, A. (2001). Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 103–112. IEEE Computer Society.

Marcus, A. and Maletic, J. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135.

Marcus, A. and Maletic, J. I. (2001). Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, pages 107–114. IEEE Computer Society.

Marcus, A., Maletic, J. I., and Sergeyev, A. (2005). Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(5):811–836.

Martakis, A. and Daneva, M. (2013). Handling requirements dependencies in agile projects: A focus group with agile software development practitioners. In *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*, pages 1–11.

Matharu, G. S., Mishra, A., Singh, H., and Upadhyay, P. (2015). Empirical study of agile software development methodologies: A comparative analysis. *SIGSOFT Softw. Eng. Notes*, 40(1):1–6.

McCall, J. A., Richards, P. K., and Walters, G. F. (1977). Factors in software quality. In *Nat'l Tech.Information Service, no. Vol. 1, 2 and 3*. General Electric Company.

Meade, A. W. and Craig, S. B. (2012). Identifying careless responses in survey data. *Psychological Methods*, 17(3):437–455.

Melnik, G., Maurer, F., and Chiasson, M. (2006). Executable acceptance tests for communicating business requirements: customer perspective. In *Agile Conference, 2006*, pages 12 pp.–46.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346.

Moreno, L., Bandara, W., Haiduc, S., and Marcus, A. (2013). On the relationship between the vocabulary of bug reports and source code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 452–455. IEEE.

Natt och Dag, J., Gervasi, V., Brinkkemper, S., and Regnell, B. (2004). Speeding up requirements management in a product software company: Linking customer wishes to product requirements through linguistic engineering. *2012 20th IEEE International Requirements Engineering Conference (RE)*, 0:283–294.

Nawrocki, J., Ochodek, M., Jurkiewicz, J., Kopczyńska, S., and Alchimowicz, B. (2014). Agile requirements engineering: A research perspective. In Geffert, V., Preneel, B., Rovan, B., Štuller, J., and Tjoa, A., editors, *SOFSEM 2014: Theory and Practice of Computer Science*, volume 8327 of *Lecture Notes in Computer Science*, pages 40–51. Springer International Publishing.

Noll, J. (2008). Requirements acquisition in open source development: Firefox 2.0. In *OSS*, volume 275 of *IFIP*, pages 69–79. Springer.

Noll, J. and Liu, W.-M. (2010). Requirements elicitation in open source software development: a case study. In *Proc. Int'l Workshop on Emerging Trends in FLOSS Software Research and Development*, pages 35–40. ACM.

North, D. (2006). Behavior modification. *Better Software*, 2006.

Oliveto, R., Gethers, M., Poshyvanyk, D., and Lucia, A. D. (2010). On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the IEEE International Conference on Program Comprehension (ICPC)*, pages 68–71. IEEE Computer Society.

Paetsch, F., Eberlein, A., and Maurer, F. (2003). Requirements engineering and agile software development. In *Int'l Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 308–308. IEEE.

Patel, C. and Ramachandran, M. (2009). Story card maturity model (SMM): A process improvement framework for agile requirements engineering practices. *JSW*, 4(5):422–435.

Philippo, E. J., Heijstek, W., Kruiswijk, B., Chaudron, M. R., and Berry, D. M. (2013). Requirement ambiguity not as important as expected —– results of an empirical evaluation. In *Requirements Engineering: Foundation for Software Quality*, pages 65–79. Springer.

Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, 14(3):130–137.

Poshyvanyk, D., Guéhéneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432.

Power, K. (2014). Definition of ready: An experience report from teams at Cisco. In Cantone, G. and Marchesi, M., editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 179 of *Lecture Notes in Business Information Processing*, pages 312–319. Springer International Publishing.

Qusef, A., Bavota, G., Oliveto, R., Lucia, A. D., and Binkley, D. (2014). Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software*, 88:147–168.

Radliński, L. (2012). Empirical analysis of the impact of requirements engineering on software quality. In Regnell, B. and Damian, D., editors, *Requirements Engineering: Foundation for Software Quality*, volume 7195 of *Lecture Notes in Computer Science*, pages 232–238. Springer Berlin Heidelberg.

Ramesh, B., Cao, L., and Baskerville, R. (2010). Agile requirements engineering practices and challenges: an empirical study. *Information Systems Journal*, 20(5):449–480.

Robertson, J. and Robertson, S. (2000). Volere: Requirements specification template. Technical report, Technical Report Edition 6.1, Atlantic Systems Guild.

Rubin, E. and Rubin, H. (2011). Supporting agile software development through active documentation. *Requir. Eng.*, 16(2):117–132.

Runeson, P., Alexandersson, M., and Nyholm, O. (2007). Detection of duplicate defect reports using natural language processing. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 499–510. IEEE.

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engineering*, 14(2):131–164.

Runeson, P., Host, M., Rainer, A., and Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley.

Salman, I., Misirli, A. T., and Juristo, N. (2015). Are students representatives of professionals in software engineering experiments? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 666–676, Piscataway, NJ, USA. IEEE Press.

Sandusky, R. J., Gasser, L., and Ripoche, G. (2004). Bug report networks: Varieties, strategies, and impacts in a F/OSS development community. In *Proc. Int'l Workshop on Mining Software Repositories (MSR)*, pages 80–84.

Santhiappan, S. and Gopalan, V. P. (2010). Finding optimal rank for LSI models. In *Proceedings of ICAET*.

Scacchi, W. (2001). Understanding the requirements for developing open source software systems. In *IEE Proceedings - Software*, pages 24–39.

Scacchi, W. (2009). *Understanding requirements for open source software*. Springer.

Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Sfetsos, P. and Stamelos, I. (2010). Empirical studies on quality in agile practices: A systematic literature review. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 44–53.

Sillitti, A. and Succi, G. (2005). Requirements engineering for agile methods. In Aurum, A. and Wohlin, C., editors, *Engineering and Managing Software Requirements*, pages 309–326. Springer Berlin Heidelberg.

Srinivasan, J. and Lundqvist, K. (2009). Using agile methods in software product development: A case study. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pages 1415–1420.

Staples, M. and Niazi, M. (2007). Experiences using systematic review guidelines. *J. Syst. Software*, 80(9):1425–1437.

Stapleton, P. (2013). *Agile Extension to the BABOK Guide*. International Institute of Business Analysis.

Sun, C., Lo, D., Khoo, S.-C., and Jiang, J. (2011). Towards more accurate retrieval of duplicate bug reports. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, pages 253–262. IEEE.

Sun, C., Lo, D., Wang, X., Jiang, J., and Khoo, S.-C. (2010). A discriminative model approach for accurate duplicate bug report retrieval. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 45–54.

Tian, Y., Sun, C., and Lo, D. (2012). Improved duplicate bug report identification. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 385–390. IEEE.

Verma, K. and Kass, A. (2008). Requirements analysis tool: A tool for automatically analyzing software requirements documents. In Sheth, A., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., and Thirunarayan, K., editors, *The Semantic Web - ISWC 2008*, volume 5318 of *Lecture Notes in Computer Science*, pages 751–763. Springer Berlin Heidelberg.

Vlas, R. and Robinson, W. (2011). A rule-based natural language technique for requirements discovery and classification in open-source software development projects. In *44th Hawaii International Conference on System Sciences (HICSS)*, pages 1–10.

Wake, B. (2003). INVEST in good stories, and SMART tasks. `http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/`. [Accessed Nov-2013].

Wang, X., Zhang, L., Xie, T., Anvik, J., and Sun, J. (2008). An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 461–470. ACM.

Warsta, J. and Abrahamsson, P. (2003). Is open source software development essentially an agile method? In *Workshop on Open Source Softw. Eng.*, pages 143–147.

Wieringa, R., Maiden, N., Mead, N., and Rolland, C. (2005). Requirements engineering paper classification and evaluation criteria: A proposal and a discussion. *Requir. Eng.*, 11(1):102–107.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 38:1–38:10. ACM.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA.

Yin, R. K. (2013). *Case Study Research: Design and Methods, 5th edition*. Sage Publications.

Zaidman, A., Pinzger, M., and van Deursen, A. (2010). Software evolution. In Laplante, P. A., editor, *Encyclopedia of Software Engineering*, pages 1127–1137. Taylor & Francis.

Summary

QUALITY OF JUST-IN-TIME REQUIREMENTS: JUST-ENOUGH AND JUST-IN-TIME

– Petra Heck –

The goal of this thesis was to obtain a deeper understanding of the notion of quality for Just-in-Time (JIT) Requirements. JIT requirements are the opposite of up-front requirements. JIT requirements are not analyzed or defined until they are needed meaning that development is allowed to begin with incomplete requirements.

We started our analysis by investigating one specific format of JIT requirements: open source feature requests. We discovered that in open source projects there is the problem of many duplicate feature requests. We found seven different categories for those duplicates. Analyzing the duplicates has also led to recommendations for manual search and creation of feature requests. Furthermore we have indicated possible tool support to avoid duplicate feature requests.

One possibility for tool support is to visualize so-called feature request networks. For this one needs to have the links between feature requests. We show that it is possible to detect horizontal traceability links between feature requests by using a Vector Space Model with TF-IDF as a weighing scheme. We have determined the optimal preprocessing steps for the feature requests to be used for our text-based analysis. Using a more advanced technique like Latent Semantic Analysis takes significantly more processing time without yielding better results in the three open source projects that we have included in our experiment.

Then we took a step back to look at quality criteria for JIT requirements in general. We developed a framework for those quality criteria and constructed a specific list of quality criteria for open source feature requests. We used agile user stories to indicate how the framework could be customized for other types of JIT

requirements. We conducted interviews with agile practitioners to evaluate our framework. After their positive feedback we conducted a case study in three open source projects in which we used our framework to score the quality of feature requests. This case study also led to recommendations for practitioners working with feature requests.

We conclude this thesis with a broader perspective on JIT requirements quality by presenting the results of a systematic literature review on quality criteria for agile requirements. This review resulted in a list of 28 quality criteria for JIT requirements, recommendations for practitioners working on quality assessment of agile requirements and a research agenda on quality of agile requirements.

To conclude we claim that the quality of Just-in-Time Requirements can be characterized as 'Just-Enough and Just-in-Time Quality'. Our framework can be used to define what Just-Enough and Just-in-Time mean for the specific JIT environment.

Samenvatting

Kwaliteit van *Just-in-Time Requirements*: net voldoende en net op tijd

– Petra Heck –

Het doel van dit proefschrift was om een diepgaander begrip te krijgen van de kwaliteitsdefinitie voor *Just-in-Time Requirements*. *Just-in-time (JIT) Requirements* zijn eisen voor een softwaresysteem die gaandeweg (i.e. net op tijd, *just-in-time*) de ontwikkeling van het systeem worden vastgesteld. Dit in tegenstelling tot eisen die volledig worden vastgelegd voordat de ontwikkeling van het systeem begint. *JIT*-eisen worden niet geanalyseerd of gedefinieerd totdat ze nodig zijn. Dit betekent dat de softwareont-wikkeling mag starten met incomplete eisen.

We zijn onze analyse begonnen met het onderzoeken van één specifiek type *JIT*-eisen: *open source feature requests*. We ontdekten dat in *open source* projecten het probleem bestaat dat er veel dubbele *feature requests* zijn. We hebben zeven verschillende categorieën gevonden voor die dubbelen. Het analyseren van de dubbelen heeft ook aanbevelingen opgeleverd voor handmatig zoeken naar *feature requests* en voor het creëren van nieuwe *feature requests*. Bovendien hebben we aangegeven welke tools ondersteuning kunnen bieden bij het voorkomen van dubbele *feature requests*.

Een van de mogelijkheden voor ondersteuning door tools is om netwerken van *feature requests* te visualiseren. Daarvoor is het nodig om de relaties tussen *feature requests* te kennen. We laten zien dat het mogelijk is om horizontale traceerbaarheidsrelaties tussen *feature requests* te detecteren, gebruikmakend van een vectorruimtemodel met *TF-IDF* als weegfactor. We hebben de optimale voorbewerking van de feature requests bepaald om deze tekstgebaseerde analyse toe te kunnen passen. Het gebruik van een meer geavanceerde techniek zoals Latente Semantische Analyse kost significant meer processortijd zonder tot betere resultaten te leiden voor de drie *open source* projecten in ons experiment.

Daarna hebben we een stap teruggezet om de kwaliteitscriteria voor *JIT*-eisen in zijn algemeenheid te bekijken. We hebben een raamwerk ontwikkeld voor die kwaliteitscriteria en een specifieke lijst met kwaliteitscriteria voor *open source feature requests* opgesteld. We hebben *agile user stories* gebruikt om aan te geven hoe het raamwerk toegepast kan worden voor andere typen van *JIT*-eisen. We hebben interviews gehouden met mensen uit de *agile*-praktijk om ons raamwerk te evalueren. Na hun positieve feedback hebben we een studie uitgevoerd met drie *open source* projecten waarin we ons raamwerk hebben gebruikt om de kwaliteit van de *feature requests* te scoren. Deze studie heeft ook geleid tot aanbevelingen voor beroepsbeoefenaars die met *feature requests* werken.

We eindigen dit proefschrift met een breder perspectief op kwaliteit van *JIT*-eisen door de resultaten van een systematisch literatuuronderzoek op het gebied van kwaliteitscriteria voor *agile requirements* te presenteren. Dit onderzoek resulteerde in een lijst van 28 kwaliteitscriteria voor *JIT*-eisen, aanbevelingen voor beroepsbeoefenaars die werken met kwaliteitsbeoordeling van *agile requirements* en een onderzoeksagenda voor kwaliteit van *agile requirements*.

Tot slot kunnen we zeggen dat de kwaliteit van *JIT*-eisen kan worden gekarakteriseerd als 'net-voldoende en net-op-tijd kwaliteit'. Ons raamwerk kan gebruikt worden om te definiëren wat net-voldoende en net-op-tijd betekenen voor een specifieke *JIT*-omgeving.

Curriculum Vitae

Ir. Petra Marianne Heck
Born: June 1^{st}, 1979
in Sittard, The Netherlands.

Education

2012 – 2016: Ph.D., Computer Science
Delft University of Technology, Delft, The Netherlands. Under the supervision of Prof.dr. Arie van Deursen.

1996 – 2002: M.Sc., Computer Science, Cum Laude
Eindhoven University of Technology, Eindhoven, The Netherlands.

1996 – 1997: First-Year Diploma, Mathematics
Eindhoven University of Technology, Eindhoven, The Netherlands.

Work Experience

September 2012 – March 2016: Part-Time Assistant in Opleiding (AIO). *Research Trainee*
Software Technology Department, Delft University of Technology. Mekelweg 4, 2628CD Delft, The Netherlands.

September 2012 – now: Lecturer Software Engineering
Fontys Hogeschool ICT, Fontys Applied University. Rachelsmolen 1, 5612MA Eindhoven, The Netherlands.

2008 – 2011: Senior Consultant in the area of Quality Management, Testing and Requirements
SQS Geneva, SQS Switzerland. Rue du Vieux-Collège 10, 1204 Genève, Switzerland.

2004 – 2008: Consultant and Researcher in the area of Software Verification & Validation
Laboratory for Quality Software (LaQuSo), Eindhoven University of Technology. Den Dolech 2, 5612AZ Eindhoven, The Netherlands.

2002 – 2004: Software Engineer
Web Development Group, Organon NV. Kloosterstraat 6, 5349AB Oss, The Netherlands.

Review Experience

- Served as PC member for the First International Workshop on Just-in-Time Requirements Engineering (JITRE'15)
- Served as PC member for the Industry track of the International Conference on Software Maintenance (ICSM'13)

Publications

- P. Heck, A. Zaidman. A systematic literature review on quality criteria for agile requirements. In *Software Quality Journal*, Under submission.
- P. Heck, A. Zaidman. A framework for quality assessment of just-in-time requirements. The case of open source feature requests. In *Requirements Engineering Journal*, Under submission.
- P. Heck, A. Zaidman. Quality criteria for just-in-time requirements. Just enough, just-in-time?. In *International Workshop on Just-in-Time Requirements Engineering (JITRE'15)*, 2015
- P. Heck, A. Zaidman. Horizontal traceability for just-in-time requirements: the case for open source feature requests. In *Journal of Software: Evolution and Process*, 2014
- P. Heck, A. Zaidman. An analysis of requirements evolution in open source projects: Recommendations for issue trackers. In *Proceedings of the Internationall Workshop on Principles of Software Evolution (IWPSE'13)*, 2013
- M. Arendsen, J. Cannegieter, A. Grund, P. Heck, S. de Klerk, J. Zandhuis. Succes met de requirements!, Book in Dutch published by Academic Service, 1st print 2008, 2nd print 2010, 3rd print 2012
- P. Heck, M. Klabbers, M. van Eekelen. A software product certification model. In *Software Quality Journal*, 2010

- P. Heck, P. Parviainen. Experiences on Analysis of Requirements Quality. In *Proceedings of the Third International Conference on Software Engineering Advances (ICSEA'08)*, 2008
- P. Heck. Quality Business Process Descriptions for Information System Development. In *Proceedings of the 10th International Conference on Quality Engineering in Software Technology (CONQUEST'07)*, 2007
- P. Heck. Constructing Consistent User Requirements. Lessons Learnt from Requirements Verification. In *Proceedings of the Ninth International Conference on Enterprise Information Systems (ICEIS'07)*, 2007
- P. Heck. Verification of SOA Orchestration. In *Proceedings of the 9th International Conference on Quality Engineering in Software Technology (CONQUEST'06)*, 2006
- P. Heck. A Maturity Model for Software Product Certification. In *Proceedings of the International Workshop on Software Certification (CERTSOFT'06)*, 2006
- P. Heck, A. Serebrenik, M. van Eekelen. Tools en Technieken voor Kwaliteitsbepaling van Productsoftware. In *Ondernemen met Productsoftware* (Proceedings of the Nationaal Productsoftware Congres), 2006
- A. Serebrenik, P. Heck. Software verification and validation. In *Handbook EDP*, 2006
- S. Roubtsov, P. Heck. Use Case-Based Acceptance Testing of a Large Industrial System: Approach and Experience Report. In *Proceedings of the Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'06)*, 2006

Titles in the IPA Dissertation Series since 2013

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05