MASTER'S THESIS | LUND UNIVERSITY 2015

Automating Traceability in Agile Software Development

Richard Simko

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-29

Automating Traceability in Agile Software Development

Richard Simko
ada09rsi@student.lu.se

RefinedWiki AB

Advisors
Lars Bendix (Lund University)
Emil Sjödin (RefinedWiki AB)

Examiner
Ulf Asklund

June 17, 2015

Contact Information
Author
Richard Simko
ada09rsi@student.lu.se

External Advisor
Emil Sjödin
RefinedWiki AB
emil@refinedwiki.com

University Advisor
Lars Bendix
Department of Computer Science
bendix@cs.lth.se

Examiner
Ulf Asklund
Department of Computer Science
ulf.asklund@cs.lth.se

Printed in Sweden
E-huset, Lund, 2015

Abstract

In order for Software Configuration Management (SCM) tools to provide good traceability links there is usually a great deal of manual effort required. The result is that data becomes unreliable, inconsistent and in many cases nonexistent. The main focus in previous research in this area has been to make general tools which can provide traceability between any type of configuration artifact in any project. The focus of this report is instead to go back to the core of software development, i.e. the code and the developers, and produce a prototype of a tool providing traceability related to the code. Instead of making a one-size-fits-all focus is on agile projects using version control tools and the idea is to reduce developer workload by automating traceability link generation and maintenance based on state analysis of issues and parsing of previous commit messages. In this report the current state of traceability in a few companies is analyzed, a prototype is developed and then reviewed by several industry experts and finally a set of related traceability links which could be automatically generated and maintained is presented as a part of a framework to define traceability links that can be automated. The prototype covers traceability links between code and tickets and the framework is then used to expand on automated traceability generation to find new areas for improvement.

Keywords: software configuration management, traceability, traceability automation, version control

Contents

List of Figures

Preface

Target Audience

The target audience of the report is anyone with an interest in configuration management and software development in general or traceability in particular. While it helps to have at least some background knowledge of how configuration management works an attempt is made to explain all important aspects and terms. The goal is that the report should be legible by anyone with an interest in software development.

Acknowledgments

The author would like to thank Lars Bendix for his guidance and support as advisor as well as Emil Sjödin for his input as a company advisor. In addition the author thanks Christian Pendleton for the very fruitful discussions, his interest and detailed feedback on the project. Finally a thanks goes out to the author's colleagues at RefinedWiki for providing input on the project and a great workplace.

Chapter 1
Introduction

Keeping track of the original reason behind a code change is becoming a more and more difficult task each day as project sizes increase and code complexity becomes greater and greater. Add to this that distributed teams are more and more common and put it in an agile context where change happens often and quick and the task becomes even less manageable. [1] In spite of this the data is often needed, many developers want to be able to go back and trace changes between different configuration artifacts and knowing why changes are made unlocks a great deal of knowledge about the change itself as well as enables the prediction of future changes. This thesis aims to explore how to simplify the generation of these traceability links, turning them into a piece of data that is recorded automatically in the same way as the commit's author or date. In order to do this a proof-of-concept will be presented, showing that implementing this type of automation is something that can be done. In addition several new scenarios in which traceability data can be automatically generated and maintained will be presented and discussed.

1.1 Background

Providing good traceability today generally requires a great deal of manual effort from both developers and configuration managers. The traceability links are generated as a result of strict processes which are to be followed by each member of the team and should anyone make a human mistake the links will become faulty or not be created at all. Especially in agile projects traceability is often frowned upon as being too much effort for too little gain. [2]

Different companies have approached this issue in different ways, some use tools which require linking of a changeset to a story or similar (Such as Team Foundation Server) while others simply ask their developers to provide IDs of tickets or requirements in their commits. However traceability is rarely an integrated part of development and configuration management tools but rather something of an afterthought, implemented as best possible with the tools available. Agile development processes add yet another layer of complexity since there very little literature on the subject, such as the Agile Manifesto [3], or configuration management at all for that matter. [4] This presents a unique challenge where those in charge of configuration management want the result of traceability but the development

team does not want to put in the effort to provide it. Agile development processes tend to side with the developers, stating that anything seen as a hindrance to development should be removed. [5]

The idea behind having good traceability is to provide better context for everyone involved in a project. This context helps when it comes to understanding relationships between different configuration artifacts including code, requirements and issues. Much research has been made on the importance of traceability and it's value to software development, specifically when it comes to change management. [6] Although these studies mainly focused on the traceability between requirements some of the same conclusions can be applied to code as well. For instance one study concluded that poor traceability greatly affected the project's cost and schedule negatively. In addition the quality went down and a higher number of bad decisions were made which of course affected both the end result and the participating team members negatively. [7] As a result one might easily conclude that there is a clear need for good traceability and an excellent next step in doing this is to automate the generation and maintenance of traceability links.

1.2 Purpose

The purpose of this report is to provide some insight into how traceability could be automated. It was written as part of the author's Mather's Thesis work at Lund University, Faculty of Science. The main goal of the project, as outlined in Section 1.4, is to present how generation and maintenance different forms of traceability links can be automated. This is done in two parts with two different goals.

The first is a proof-of-concept providing a way of showing that it is indeed possible to automate the creation of traceability links for certain types of traceability using only the data already available in most software development teams. The purpose of the tool is to provide as high-quality links as possible between an issue tracking system and the source code in a repository while requiring a minimal amount of user input.

The second part is to construct a framework using which the automation of traceability can be evaluated. This framework is then used to construct a list of different traceability types which the author believes could be automated within a reasonable time frame. This list should provide future researchers with material in order to start investigating new forms of automated traceability.

1.3 Limitations

Since the project was constrained both with regards to time and manpower (As it was performed by a single student over a 20 week period) some limits had to be set up. First of all, the scope of the report is focused on agile projects. The reasoning for this being that, based on previous literature, traditional software development teams usually have traceability integrated as part of the process. Agile teams on the other hand tend to struggle with doing traceability in an agile form. In addition to this the company where the project was conducted, RefinedWiki, use an agile

development method an were interested in finding solutions to this problem that would work in their specific context.

Furthermore, the initial research and theory part is mainly based on previous research and only a limited amount of original research is performed. While a few interviews with key persons were made as well as a relatively large survey the sample size is not large enough for there to be any conclusive results. However, combining this limited analysis with the previous research that was studied hopefully gives a clear enough picture of the current state. It is also not the purpose of this report to present conclusive evidence about the current state but rather to learn a sufficient amount about it in order to begin development of a prototype.

1.4 Research Questions

The research presented in this thesis will attempt to address these two main questions.

- How difficult is it to automate traceability between code and tickets?
- What additional traceability can be automated?

The reasoning behind these two questions is that they both sum up what needs to be done in order to improve the automation of traceability as described in the previous sections. Initially a small proof-of-concept can answer the specific question of how difficult it is to automate traceability between code and tickets. In addition to this it will also help answer the much more generalized question of how difficult it is to automate traceability in a more general sense. While this is a limited subset of the problem it will still shed some light on whether or not it is possible at all.

Using the answer to the first question as a stepping stone the research will then expand to finding new ways of automating traceability between other software development artifacts. This will provide the reader with some ideas about what areas automated traceability can expand into in the near future and which ones of these should be focused on. This part of the research will mention not only potential improvements to the prototype developed but also propose a generalized way of analyzing and presenting traceability links which can be automatically generated and maintained.

It is the author's belief that developers want to spend as little as time as possible dealing with configuration management and other so-called supporting tasks and instead focus on development. However they are still interested in the benefits of configuration management and traceability links. In other words they want the cake but they don't want to bake it. With this in mind automating the generation and maintenance of traceability links as much as possible is important since it will allow developers to employ the advantages of traceability with a minimal amount of effort in order to provide the data.

1.5 Case Description

While the focus of this research is wider than the company at which it was performed it is still good for the reader to be aware of the context in which the development and studies was performed. RefinedWiki is a small but fast-growing company developing plugins for Atlassian's products. The fast growth combined with the use of agile methods has lead to a rising need for good traceability that preferably handles itself. As a result automated traceability is a given solution to this problem. However in addition to developing a tool for in-house use there is also interest in making the product viable for other companies. Most notably making it a reasonable product for larger companies where the need for traceability is greater.

1.6 Research Method

As described in Section 1.1 there is a need for a tool which can provide maximum traceability with minimum effort. This tool should preferably integrate with existing version control tools and issue tracking systems. This thesis will describe an attempt to implement such an automated tool for adding traceability between code and user stories with the intention of reducing the developers' workload and improving the quality of traceability data. The specific features of the tool will be described later in the report.

In order to elicit the required features of the tool, as well as confirm the hypothesis that there is indeed a needed for such a tool, a series of interviews as well as a survey was performed. The interviews served the purpose of providing in-depth ideas and feedback regarding the implementation, and since they were done with a variety of professionals with different roles the views provided covered a relatively wide audience. In addition to the interviews the survey provided quantitative data on how different people use or want to use traceability in their software development process. It also served as a way to determine how much extra effort engineers were prepared to put in to the process of gaining extra traceability.

After having completed the analysis of the current situation the development of the prototype started. This was the main part of the project, spanning a total of ten weeks of development and two weeks of review. The review consisted of interviews and group discussion with colleagues at RefinedWiki as well as external interviewees including some of those who were interviewed as part of the initial analysis.

Prior to starting the project it was thought that a simple way to extract traceability data would be to use feature branches were each developer creates a branch for each feature being worked on. This workflow is similar to the Git Flow workflow [8] and should provide a simple way to add traceability since each commit is related to a branch which has a direct connection to a ticket in the issue tracking system.

1.7 Outline of the Report

This first chapter introduces the context of the report. It provides some background to the issues, goes over the research purpose and limitations and describes the scenarios in which the report was constructed.

The second chapter covers some theoretical knowledge which is useful to the reader. This ranges from some definitions of different important terms as well as an analysis of related work. This covers some previous research and also provides the reader with detail regarding systems related to the project. This is mainly for the reader who is unfamiliar with these systems.

Following this the initial analysis covers the theoretical analysis made prior to starting development. This includes the interviews and surveys performed as well as an analysis of systems providing similar services.

After this comes the Design chapter which covers how the solution was thought of and what the idea was prior to development. The Implementation chapter then covers some implementation details and provides the reader with insight into what was implemented. Finally the Evaluation chapter covers how the developed prototype was evaluated and how the evaluation went.

At the end comes the theoretical part of the thesis which covers the results from testing the prototype as well as the framework for traceability data, both in the Results chapter. Lastly followed by a Discussion and Conclusion chapter which discusses the feedback given and how traceability could be automated in the future as well as summarizes the report.

Chapter 2
Theoretical Background

This chapter covers background information which is relevant to the reader. It also limits the domain of the thesis through definitions of the different critical areas which will be covered.

Initially traceability is defined in the context of this report. A few different accompanying definitions are given and the tool which was produced is categorized based on this. This is followed by a presentation of what the author considers agile software development and finally a formal definition of configuration management to help readers unfamiliar with the term.

2.1 Traceability

In order to better understand the domain of this thesis one must first define traceability. A few different definitions exist, Jacobsson defines traceability as

> Traceability is practices and routines used to trace requirements and changes throughout development. The tracing information should be useful for everyone involved in the project and not just the developers. In order to have traceability it needs to be possible to trace in both direction[s]. [5]

For the purposes of this thesis Jacobsson's definition is a bit too focused on requirements, however the fact that the information should be useful to everyone as well as bi-directional are very good points. In this case the interesting type of traceability on which the research focuses is that between requirements, often expressed in stories or tickets in agile projects, and code. As Jacobsson writes, traceability should be two-way, i.e. it should be possible to find why a piece of code was written, with the answer in the form of a story, as well as what code was written as a result of a story, with the answer in the form of files.

Gotel and Finkelstein [9] make the separation Pre-requirements specification and Post-requirements specification and in this case the focus will be on post-RS. While that paper was written in 1994 long before agile methods came in to style (The agile manifesto wasn't published until 2001 for example) the definitions still hold true, although one must be a little bit flexible with the term requirements specification.

The third definition is that of Appleton et al. who present the *Five Orders of Traceability*. [10] This is a tiered model where each step up on the order scale represents additional traceability, ranging from no traceability (Zeroth order) to traceability about the Five Orders of Traceability (Fourth order). The tool described in this thesis would end up on the second or third order. The second enforces metadata in addition to the links, "Not only are there links, but there is also contextual information", (Which is provided by the version control tool in this case). The third order involves having links between metadata, i.e. "effectively capturing important decisions, criteria, constraints, and rationale at the various points in the knowledge creation lifecycle". This context can to some extent be provided by the issue tracking system if the team uses for example the comments available on a ticket. Because of this, whether this tool falls under the second or third order will very much depend on how the team uses both this tool and other tools which it will interact with.

The proof-of-concept tool is focused on traceability between tickets in an issue tracking system (Although it could easily be applied to any form of requirements) and code. This limitation was made in order to provide a manageable domain for which to implement a tool which is as complete as possible given the limited available time frame and. Due to the focus on agile methods it was chosen to focus on the part closest to the developers, i.e. traceability involving code, since this is where a large amount of time can be saved. With regards to the other part of the report other types of traceability are of course covered as well.

2.2 Software Configuration Management

While there are many definitions of Software Configuration Management (SCM) out there, one formal definition is the one set up by the IEEE. The definition of Configuration Management (CM) according to IEEE is:

> A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements. [11, p. 20]

One can easily deduct that it's a very broad term spanning all types of activities related to managing the change of some type of artifact, in this case software. An important thing to note however is that CM for CM's sake serves no purpose but is rather a tool to support other parts of the organization. Depending on how a specific team works this can lean more towards the practical side of implementing tools or against the more abstract side of creating processes. Another formal definition is the one in CMMI who define it as including both technical and administrative practices to "establish and maintain the integrity of work products using configuration identification, configuration control, configuration status accounting and configuration audits". [12, p. 114]

2.3 Agile Methods

Since this thesis will focus on making traceability compatible with agile teams it is also important to define what an agile team is and what sets it apart from the traditional software development. With a vast majority of software development companies using agile methods one might consider it a norm and something which need not be defined. [13] However since people's view of agile might differ it is reasonable to state formally what is meant in the context of this thesis.

As mentioned in the State of Agile survey, Scrum is by far the most popular agile methodology and hence it would make sense to focus on it. [13, p. 4] However in the context of this thesis there is little difference between the different agile methodologies. The important takeaway is that the team works with constantly changing requirements and artifacts and does not use explicit baselines preventing such change. It should also be noted that scrum does not explicitly direct how development should take place but rather the managerial structure around the development team. [14]

An important difference, which was mentioned by one of the interviewees during the initial analysis is how *Configuration Status Accounting* (Which one might call a product of traceability) differs between agile and traditional software development. It used to be that this information was designed to be consumed by management in an attempt to judge if the project would be on time, if it's going the right way and if would be within budget. In contrary to agile teams where the consumer of this information is the team, mainly the developers, in order for the team to be able to make decisions and estimations related to their sprints or iterations. This vastly changes what is required from the traceability, not only is different information needed, due to the fast-moving nature of agile development it must also be much more current. This of course spawns the need for automated tools to provide the information.

2.4 Definitions

Ticket Refers to an entry in an issue tracking system such as JIRA. This covers a wide range of types from bug reports to stories although in the context of this report the type of ticket is not relevant.

Issue Tracking System While this term is somewhat ambiguous in this context, it refers to the system in which tickets are logged. As mentioned tickets doesn't necessarily refer to issues but can also refer to new functionality in the form of stories. Regardless of type an issue tracking system in the context of this report is a system where tickets are stored.

2.5 Related Systems

This section will describe the systems related to the developed tool in order to give the reader an idea of what they are. This section is mainly aimed at those not familiar with Atlassian's products.

2.5.1 Bitbucket & Stash

Bitbucket is Atlassian's cloud hosting platform for code. Not unlike Github it has support for the most basic source control activities such as permissions and pull requests. In addition it features some rudimentary issue tracking and wiki pages. While this project could had been made using any type of Git hosting platform with support for a `post-receive-hook` [15] Bitbucket was close at hand due to the fact that it's being used at RefinedWiki. In addition to this it has a REST API for querying the contents of repositories as well as receiving notifications in JSON format when commits are pushed. Both of these greatly simplified the development of the prototype.

While no direct development was made using Stash it's still worth mentioning since it's more widely used among corporations than Bitbucket. The idea behind Stash is that it provides the same features as Bitbucket, combined with a few additional ones and the ability to host the software on your own server. This means that it can be placed behind a firewall and that your code is being hosted by you. Stash has the same API support as Bitbucket, making it easy to expand the prototype to support Stash in the future.

2.5.2 JIRA

JIRA is Atlassian's issue tracking system used for keeping tickets about features, problems, ideas, stories bug reports etc. JIRA is made to be configurable which means that each company's implementation may differ greatly. As such it is hard to give a more general description since the use cases and applications differ. However one important note is that it's a web-based tool that can be run either in the cloud or locally, with extended plugin support. In the context of the research made here JIRA was ran on a server managed by the author in order for the plugin developed to work properly. [16]

2.6 Previous Research

This study is greatly based on the Master's thesis written by Jacobsson in 2009 called "Implementing Traceability in Agile Software Development". [5] This is an analysis of how traceability is implemented in agile projects and how it could be improved in the future. Jacobsson presents a list of problems which teams face today as the reason for why traceability is not widely adopted in agile teams. After this he introduces several traceability practices and rates them with regards to the costs and benefits. Jacobsson also raises the specific point that automated tools are required writing that "*In order to make heavy traceability more agile tools and automation will be needed. Tools and automation will result in an initial cost from implementing the tools but will reduce the work load for the team*".

This theory is confirmed by Antonino et al. who, in their paper "A Non-Invasive Approach to Trace Architecture Design, Requirements Specification, and Agile Artifacts" claim that Jacobsson's observations are not isolated but a general mindset of agile teams. They also write that it's important that developers, among other things, know why a certain decision was taken and the impacts of the

decision. They write: "*All this is achieved when a proper traceability mechanism - i.e., minimally invasive, automatic, implicit - is used in the development process*". [2]

Kowalczykiewicz and Weiss also discuss the need for automated generation of traceability links between different project artifacts in their proceedings paper "Traceability: Taming uncontrolled change in software development". They discuss a similar observation as is made in this study, namely that there is a lack of tools in general and automated tools in particular when it comes to traceability. Instead many teams focus on the development process and try to provide traceability through a good process, e.g. the daily Scrum meetings which is traceability in a way since it spreads knowledge about what is being done throughout the team. While this is not bad it does not eliminate the need for a tool. The result of this study is a tool which will be discussed more in Section 3.3. [17]

Farah writes about the importance of traceability and the need to trace whatever can be traced in his article "Traceability and Auditability: Satisfying Your Customers". However he also explicitly points out that incomplete or invalid traceability data is no improvement over not having traceability at all. "*Here's where tools are important*", Farah writes and goes on to mention several different scenarios where tools could easily be integrated to provide better traceability data. [18]

Mohan et al. conducted research similar in method to what has been presented in this report. [19] As they express it their research consists of an attempt to integrate SCM and traceability, providing a tool for traceability which cooperates with the SCM tool being used. They developed a model for traceability, implemented a tool supporting this model and then evaluated it at the same company during a long-running study with several interviews. The tool integrated with Microsoft Visual SourceSafe, a version control tool which has since been discontinued, and spanned traceability between all types of configuration management artifacts. The model they define initially is were comprehensive, spanning all types of artifacts and most possible links between these. When it comes to automation the developed tool, called Tracer, does little more than check out files which are related to a certain change when someone manually confirms that he is working on that change. However while the tool mainly requires manual work it's a great integration of traceability and version control tools. In addition, one should also keep in mind that the research is 10 years old meaning that other tools have evolved and capabilities have improved.

Chapter 3
Initial Analysis

This chapter will cover the initial analysis of the situation which was made prior to implementation. One might see this part of the research as the elicitation of requirements from potential future stakeholders. The need for traceability has already been established in previous research, as described in Section 2.6.

In order to establish what types of traceability could be implemented a series of interviews as well as a survey was performed. The purpose of this was to uncover how a traceability tool could be implemented. For this reason it was important to establish what the situation looks like today in most companies, it is also important do define what data is already available in order to answer the first question set up in Section 1.4. The definition chosen is that any data companies already require from their developers can be seen as existing.

3.1 Interviews

Before starting with the implementation work a series of interviews were made to confirm the results from previous research. The interviews were conducted with a variety of professionals working in the software industry. A high-level configuration manager at a major telecommunications company, referenced to as Person A at Company A, as well as two engineers at a large software development company, Persons B and C at Company B, were interviewed, each giving their unique perspective on the issue. In addition to this several less formal discussions with colleagues at RefinedWiki took place during this part of the project.

The interviews were conducted as semi-structured interviews, based on the questions in Appendix A. The questions were used more as a basis for discussion rather than direct questions, meaning that the interviews themselves diverged quite heavily from the asked questions. This did align well with the purpose however, which was to provide ideas for how traceability could be achieved in an agile environment.

The main purpose of the interviews were to provide a creative discussion which could ultimately lead to the elicitation of a few requirements for the tool. In addition to this, the interviews were performed prior to the survey which meant that the answers from the interviews were used as a basis for the survey questions. In order to confirm that some of the claims made by the interviewees indeed represented a wider view on traceability similar questions were asked in the survey.

3.1.1 Results

A clear conclusion that could be drawn was that none of the companies used branches to implement traceability as had previously been hypothesized. (As suggested in Section 1.6) Instead all teams used specific syntax in their commit messages. An example would be referencing ticket IDs, thereby tying each changeset to a specific ticket in an issue tracking system. While they all claimed this worked well, it did come with some drawbacks. First of all is the risk of a developer simply typing the wrong ticket number. In the system used by Company A ticket numbers are a 6 digit series of numbers and a typo can easily slip in to the commit message. Company B uses JIRA meaning that ticket numbers are represented by a combination of a project key and a unique number (Eg. `PROJ-435`). While this might seem like an easier approach mistakes were common at Company B as well, according to the interviewees. This led to a shift in focus away from branching for traceability (As described in Section 1.6) to implementing a tool which could support this model of work instead, providing what was needed for developers to more effectively tag their changesets using commit messages.

All the interviewees agreed that developers would most likely be prepared to spend quite a lot of time providing information for traceability. This would especially be true if the value of the achieved traceability could be adequately explained. As Person A put it, pedagogically explaining the purpose of different CM-practices is the largest challenge of working in configuration management. This follows closely with the development that Configuration Status Accounting has shifted from being an activity designed to provide information to managers towards being about providing information to developers. Since the developers are the consumers of the information they are also much more inclined to spend extra effort providing raw data for that information. This does to some extent contradict the initial hypothesis that developers want to do as little work as possible with regards to configuration management which was mentioned in Section 1.4. On the contrary, as confirmed by the survey, most both developers and configuration management persons are prepared to spend quite a lot of extra time in order to gain traceability (See Figure C.6).

Regarding what type of traceability the persons interviewed were interested in, this varied greatly. Unsurprisingly the configuration manager was interested in providing traceability for everything. He claimed that the type of traceability discussed in this thesis is the easy type. An interesting artifact of traceability would be the link between issues and requirements. This could provide detail regarding whether or not a defect will actually affect a stakeholder in a scenario where multiple stakeholders are involved in a project. If the issue does not relate to a requirement posed by that stakeholder one can safely assume that he will not be affected.

3.1.2 Analysis

The conclusion that can be drawn from these interviews is that developers are prepared to spend more time than previously thought to gain good traceability. Another important conclusion is that the work should probably focus more on parsing commit messages than using branches to provide traceability. This defined

much of the work performed going forward, as the project shifted focus from analyzing branch names and supporting branches to handling commit messages in a clever way instead.

3.1.3 Limitations

One should be aware that the sample size is small and that the conclusions drawn from these interviews may very well be an exception. The main purpose of the interviews was not to draw any general conclusions about how traceability is used. Rather the purpose was to sift out some ideas and thoughts about traceability. The reason for this being that interviews are great at providing focused input on a specific topic from a limited number of individuals, due to the effort they require from both the interviewer and the interviewee. The wider conclusions were left to the survey (See Section 3.2).

3.2 Survey

In addition to the interviews a series of surveys were performed. These had the purpose gaining more knowledge about what the overall situation looks like and what could be improved. In contrast to the interview questions the survey questions (Appendix B) were a lot less open, providing a basis for some statistics about a few companies. They were sent out to various industry professionals with different areas of expertise.

In total there were 72 replies to the survey, which of course isn't that much but thanks to the distribution between different companies it still gave quite a wide sample. This can be seen since there was a small overlap between companies, at most 3-4 persons from the same company replied and in the cases with several employees from the same company they were often part of different teams. All in all one can conclude that while the data does not represent 72 different software development teams is does represent relatively many. It must also be said that several of the questions are subjective and receiving views from several team members from the same team can be very valuable.

The purpose of the survey was to confirm that the hypothesis as well as some of the views expressed in the interviews indeed represent a wider view of traceability within the software development community. A survey was the easiest way to quickly get a larger number of replies, covering an audience as wide as possible in the shortest amount of time. While one could had done additional interviews that would had been too time consuming and it must be remembered that this part is only meant as a brief initial analysis in preparation for development.

3.2.1 Results and Analysis

A selection of answers to the most important questions can be found in Appendix C.

The survey clearly shows that JIRA is a very common platform for issue tracking, making it a great choice for developing a prototype such as this one. The fact

that most companies seem to have guidelines for commit messages was very interesting and provided a great starting point for the development. Since as many as 71% (See Figure C.8) of respondents already require their developers to provide this data, it can be considered a part of the existing data set. In contrast, only 25% branched by feature using a specific pattern (38% branched by feature and 65% of those used a pattern for the branch name), making that a much less common approach to traceability. As such, the main focus should lie on providing traceability through commit messages and assisting developers in doing this.

The results regarding how much extra time people were prepared to spend in order to gain better traceability (See Figure C.6) mirrored the hypothesis set up by the interview quite well. Everyone was prepared to spend a relatively large amount of time with a median around 5-10% (37% were prepared to spend 5% more time and 22% of respondents were prepared to spend 10% extra time). While one must consider that not all respondents were developers, they were the single biggest group (46%, see Figure C.2).

Another clear result of the survey is that everyone seem to consider traceability an important thing. When respondents were asked to rate the importance on a scale from 1 to 5 a majority answered 4 or 5 (23 and 21 responses respectively, see Figure C.5). No one answered 1, meaning that everyone seems to realize that it holds some value. This explains the answer to the amount of time that developers and other are prepared to spend in order to gain traceability, if you feel something is important then you are quire likely to be prepared to spend a lot of time achieving it.

3.2.2 Limitations

The survey does suffer from a heavy bias towards companies which are already working with CM in a good way. The survey was distributed mainly though the SNESCM[1] mailing list, consisting mainly of CM experts. These were also asked to distribute the survey among their colleagues, making sure that not all answers came from people working with CM. While the question about roles (See Figure C.2) shows a clear distribution between developers and CM professionals one must remember that the fact that a company simply has someone responsible for CM (Who reads the SNESCM mailing list) means that this company takes CM seriously. This in turn means that they are likely to already be working with traceability and more likely to see the importance of it.

While the sample size might look limiting one should consider that the 72 respondents were well distributed between different companies. The result is that the answers came from approximately 30 to 40 different companies, giving a good spread. If one takes into account that these companies are most likely different in many regards the result is that the survey covers a wide range of companies of different sizes, maturities and with different development processes.

[1]Scandinavian Network of Excellence in Software Configuration Management

3.3 Similar Implementations

This section brings up some similar implementations, details what their likenesses are to the prototype described in the report and what lessons were learned by studying them.

3.3.1 OPHELIA

One of the implementations was mentioned in Section 2.6, called OPHELIA (Open Platform and methodologies for devELopment tools IntegrAtion). The idea of this project was to create a set of interfaces which could be used in order to have different software communicate to provide traceability. The project's effort consisted of developing the interfaces and creating an implementation of it that they call Orpheus. The tool consisted of several different ways to add relations between objects and viewing these relations. It allowed users to subscribe to changes in certain objects, including if something changed which was linked to the object. The major feature of OPHELIA is that it integrates all parts of the project in to a single tool for traceability, which is great. It is also, supposedly, not tied to a single vendor or tool but rather a general purpose interface which can be integrated with almost any other tool. [17] However, despite how great this sounds and the project being funded by the European Union it seems to have been shut down. The project website[2] mentioned in the paper no longer exists and any references to the project were written 10 or more years ago.

While the basic concept is similar to the prototype described here there are some key differences. The OPHELIA project had some great ideas such as integrating many tools into one and providing an enormous set of features (The traceability is just one of several modules). However the project most likely didn't succeed for this very reason, it was too large and there was no financial gain for tool developers to implement the interface. The authors of the article to bring up that point and claim that the Orpheus implementation would be the solution to that. The lesson learned is to not try to support every single tool and use case but instead focus on a few for a first implementation. This is part of the reason why the focus of the project described in this report was so narrow when it comes to tools and platforms (For more see Section 4.1).

3.3.2 Atlassian's JIRA integration

Since the proof-of-concept is made using some of Atlassian's products (JIRA and Bitbucket) it is relevant to cover what Atlassian themselves provide. Atlassian's products integrate in several ways providing, among other things, traceability between entries in JIRA and commits in Stash or Bitbucket. JIRA is Atlassian's issue tracking system with support for both agile stories as well as issues relating to bugs and problems in the software. This can integrate to with Stash, which is Atlassian's repository management system that the customer can host themselves, or Bitbucket which is a cloud hosted repository (Described closer in Section 4.1). In terms of traceability links the integration provides a display of branches, commits,

[2] `http://www.opheliadev.org`

pull requests and builds related to a certain ticket and in addition it automatically creates links to tickets mentioned by their key (For example `STASHDEV-4785`) on Stash or Bitbucket. [20] A similar integration is provided for Bitbucket and several other code hosting platforms such as Github in the form of the JIRA DVCS Connector Plugin which has a similar feature set. [21]

These tools provide an excellent companion to the tool developed as a part of this project. Atlassian's integration tools show the traceability links in a way which is easy to understand for the user and provides the context which is the end purpose of traceability links. As a result of this, there is no need for this project to focus on the display of traceability links, since that area is already covered. Instead the focus is on generating these links, as Atlassian's products still requires the users to manually input the data and create the links. Combining this with a tool which automatically generates the data, leveraging the data already available, should provide an excellent software suite for traceability data.

3.3.3 Initial Requirements

As a result from the interviews and the survey a few domain-level requirements were set up. These requirements were vague enough as to not jeopardize the experimental nature of the project while in the mean time being clear enough to keep it on track. [22] The following requirements were elicited (In order of importance):

1. The system should support entering ticket-IDs in the commit message.
2. The system should support branching by ticket
3. The system should provide support for validating such ticket-IDs
4. The system should provide a suggestion of potential ticket-IDs
5. There should be a way to find a commit related to a ticket directly from that ticket's page in JIRA
6. The system should suggest ticket-IDs to the user

The basic features, involving only verifying the users' input, is the attempt at answering the first research question. Due to the overwhelming amount of teams already requiring developers to enter a valid ticket-ID, as mentioned in Section 3.2 this can be seen as existing data. The biggest issue with this data is that it is not validated in any way. The result is that the product of this data (The traceability) can not be trusted without manual validation and as a result can easily go unused in favor of, for example, asking a fellow developer. If this data can be validated its accuracy can be greatly improved and as a result proper traceability can be ensured. Since the development was divided into two stages (More on that in Chapter 5) a natural division was formed between the requirements. The more basic features involving only verifying the user's input were scheduled for the first stage and the more advanced features were scheduled for the second.

The other variant of doing roughly the same thing is branching using unique branch names which relate to a ticket's ID. In the teams which use this approach

(RefinedWiki is among them) this data can of course also be seen as present from the start. In this scenario an additional problem arises where branches generally aren't kept around after completion of a ticket. Once a branch has been merged there is no way to recover the data that was captured in the branch name unless additional precautions are taken. While it it true that named branches in Mercurial stay even after closing this is not a convenient way to provide traceability. [23] Once again providing verification of the users' input improves the quality of the traceability, as well as integrating the traceability data into the issue tracking system.

In addition to providing verification the system should also suggest an issue and prepare the commit message with this issue. This is an advanced feature which was implemented later in the development cycle and the purpose was to further simplify the generation of good commit messages.

Chapter 4
Design

This chapter will outline how the implementation of the prototype was made. It will cover the idea for a solution to the problems presented in Section 1.4, how this was implemented and the motivation for the tools which were used to do so. In the end the evaluation of the tool will be described.

4.1 Solution Idea

The implementation was focused around Atlassian's products (Mainly JIRA[1], Bitbucket[2], and Stash[3]). A reason for this being that RefinedWiki is focused on developing products for the Atlassian ecosystem, making it an obvious choice. They also use both JIRA and Bitbucket themselves and as such tests could easily be performed in a live environment with a real team. The other reason was the survey results which showed that JIRA was very popular as a bug tracking system with about 45% of respondents using it (See Figure C.4) which makes it a good platform for reaching a wide audience. Bitbucket and Stash both integrate well with JIRA and for that reason were easy to base the prototype development on. A real product should however preferably work regardless of where the repository is hosted.

Git was chosen as the version control tool on which the prototype should be based as it was by far the most popular one in the survey (See Figure C.3). It is also a flexible and modern tool which proved quite easy to work with, its distributed nature and the support for it on both Bitbucket and Stash made it easy to set up a test environment. A secondary feature was to implement support for Mercurial since this is the version control tool currently used at RefinedWiki. However it proved impossible to implement due to Mercurial's lack of certain hooks required for the automation to work. (More on that in Chapter 5)

Initially the idea was to design a tool which could be set up to parse an existing repository for information about traceability. This would be done through a combination of using branch names to which the commit had been made as well as old commit messages. However it was quickly discovered that Git does not save branch names after merges, nor is there a way to retroactively find which branch

[1] `https://www.atlassian.com/software/jira`
[2] `https://bitbucket.org`
[3] `https://www.atlassian.com/software/stash`

a specific commit was originally made to due to Git's distributed nature. This technical limitation, along with the results from the survey and interviews (See Chapter 3), made the project shifted focus towards parsing commit messages and providing logic around this instead. However, in order to not drop support for the roughly 40% of surveyed companies which actually did branch by feature according to specific patterns (See Figure C.7) the idea for parsing branch names was kept but in a reduced form. Instead of parsing old commits this type of traceability would only be available in commits made after the installation of the tool.

4.2 Views

This section will briefly cover what the tool looks like in order to give the reader a rudimentary impression of what has been implemented and how it will look for the end-user.

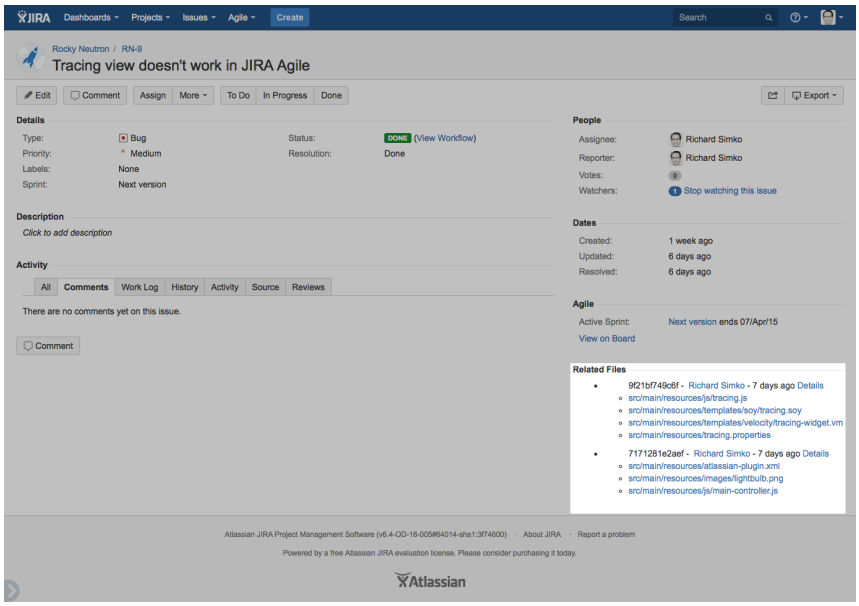

Figure 4.1: The files related to a specific ticket

During an early brainstorming session at RefinedWiki it was decided to split the product into two major pillars. The first one would be the view in JIRA in which the user sees the ticket as well as the commits related to that ticket (See Figure 4.1). The second is the hook integrated with the version control system which will provide the developer with feedback regarding the traceability data which he provided (See Figure 4.2). The reasoning behind this was that the tool probably has two main audiences. The first is the developers who are providing the traceability data and the second consists of both developers and other team members who will read the data. These two groups (Even though developers can belong to both groups) have quite different use cases for the tool and as such it

makes sense to divide it into two parts depending on the use case. Since the focus of the project wasn't on providing a way to present traceability data but rather a way to automate its generation most of the effort was spent on the second pillar.

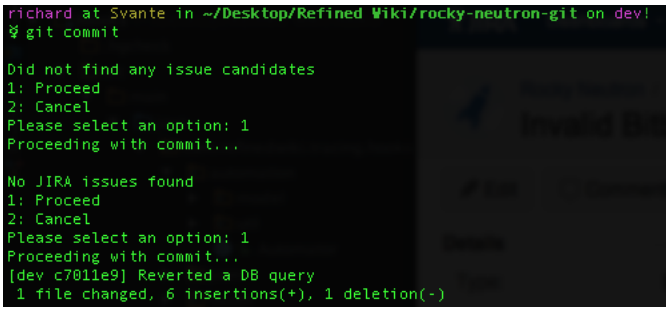

Figure 4.2: The command line UI for the hook after a commit without a valid ID has been made

The hook view is where the traceability data is created, here the focus is on *writing* traceability data. While the data isn't submitted directly through the hook it provides a suggestion and validation mechanism to make sure that the data provided is correct and consistent. The developers are also given the opportunity to fix certain problems with the ticket they reference. For instance if the issue is unassigned the developer is given the option to assign the issue to himself which performs the change in JIRA and also leaves a note informing that the assignment was made via the hook. This view should mainly be used by developers since they will most likely be the ones submitting commits in the version control tool.

The JIRA view is the view used for *reading* the data. This view shows all commits that have been connected to a ticket. It presents the changed files grouped by each commit, together with the author, a date and an expandable commit message. It was understood early on that this view would be used by several members of the team, including developers, project managers, configuration managers and most likely testers and those responsible for builds. The wide audience meant that this view had to be made to fit each role and as such compromises had to be made. As the project progressed it was also discovered that this part of the prototype has been implemented in several similar tools and because of this there was a shift in focus away from making this view more usable and towards making the generation of the commit message easier instead.

Chapter 5

Implementation

This section will cover the technical details regarding how the software works. It will cover how it collects the data as well as how it parses and matches it and produces the traceability data.

5.1 Data Flow

The flow of the data is presented in Figure 5.1. Commits are made containing traceability data in the form of issue keys which are then pushed to Bitbucket. In the context of this prototype it is restricted to only this service, however expanding it to include a generalized Git hosting platform is not difficult as future work (How this could be done is described in Section 7.5). The push triggers a hook on the Git server which sends the data via HTTP POST to the JIRA instance. JIRA parses the data and generates the links between the commits pushed and the mentioned issues. In addition to this JIRA can fetch the data directly from Bitbucket, for instance if the system is deployed on an existing repository. The reason for using the commit message is that this was the most readily available place to input data which is to be shared with the issue tracking system. Since the only thing needed to reference a ticket is the ticket's ID, using a generalized field for textual input was the easiest solution. Another potential approach could be to use `git-notes` which allows a hook to modify a commit without affecting its hash. [24] However this data is never presented to the user when performing a commit, rather it's something that can be appended after a commit has been completed. This did not suit how this tool is to be used since the data accuracy isn't high enough to be completely automatically generated but instead should have manual verification. For this reason the ticket ID is included in the commit message prior to showing the message to the commit author, who can then verify the auto-generated message.

Finally the generated data is used to generate new data. This is done through a series of steps in the client side hook which will be described in more detail later. The client fetches this data as required using a HTTP REST interface provided by JIRA.

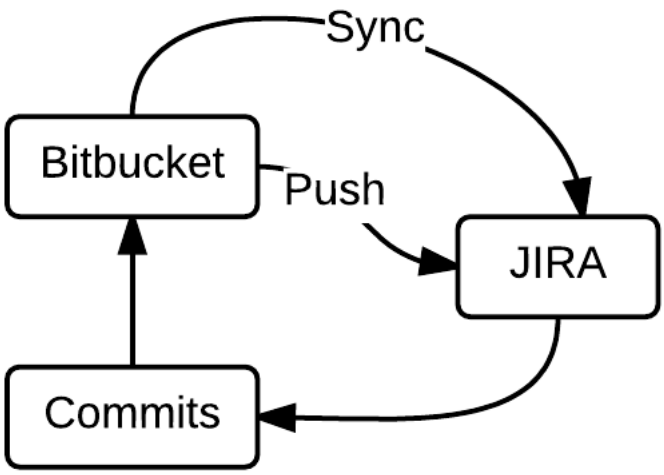

Figure 5.1: An overview of how the data flows between the different systems

5.2 Automation of Traceability

The main focus of the project is how to automate traceability and as such this is where the most in-depth technical description will be.

The automation consists of three parts. The first is a step prior to committing which parses your previous commits and attempts to determine if you're still working on one of the features you've worked in your previous commits. The system analyzes a fixed set of commits backwards on your current branch, if there are no commits on the current branch it assumes you're working on something new and moves on to step two. Using the user's average number of commits per ticket (Which is calculated in JIRA) the system compiles a list of reasonable candidates for what you're currently working on. These are then compared by their Euclidean distance based on the recency and frequency of a certain ticket:

$$\sqrt{r^2 \cdot w_r + f^2 \cdot w_f} \tag{5.1}$$

Where r is the inverse (I.e. higher is better to simplify comparisons) of how recently a ticket was mentioned , w_r is a weight constant, f is how frequently a ticket has been mentioned and w_f is a weight constant for the frequency. The weight constants are used to scale the two parameters based on which one is considered most important and can be tweaked by the user based on their preferences. This was the simplest method to compare and rate the commits in order to determine

a likeliness that the developer is currently working on that issue. This could of course be done in a variety of different mathematical ways but the focus of the project isn't data analysis but rather simply providing a proof-of-concept this way works well enough. When it comes to the weight constants they were both set to 1 meaning that they had no impact on the calculations, however they will most likely be needed if a system like this is to be used. They should be tweaked later through empirical testing.

The idea behind how the automation is implemented is that it should support each task that the developer has to do anyway and provide help along the way. For instance if a version control tool is used then commits have to be made. This is an excellent time to present the developer with a suggestion regarding what traceability data is supposed to go with the commit. Even if the data is faulty it will remind the developer that the data should be supplied.

If these results are conclusive, i.e. if the highest rated ticket outweighs the other by more than a specified threshold, that ticket is selected and inserted into the commit message. What this threshold should be is not finally determined but must be discovered through some empirical testing of the tool with a wider array of developers. Perhaps some self-learning could be integrated with this as well so that it adapts to a specific developer's usage pattern. If it does not pass this threshold the system moves on to the next step which consists of fetching data from JIRA. Using a REST endpoint the user's assigned, in-progress, issues are fetched and ordered by how recently they were updated. If there is more than one issue that matches this criteria the user is presented with a choice (See Figure 5.2) of what ticket to enter in the commit message. These search criteria can of course be expanded to include more parameters or to be more flexible and customizable as per user requirements. For instance it was mentioned during review that some companies don't assign tickets to individual developers but rather to teams. This would of course reduce accuracy but the system can easily be tweaked to allow this type of workflow. Once the choice has been made a commit message is generated with the ticket ID like `[RN-1]` at the start.

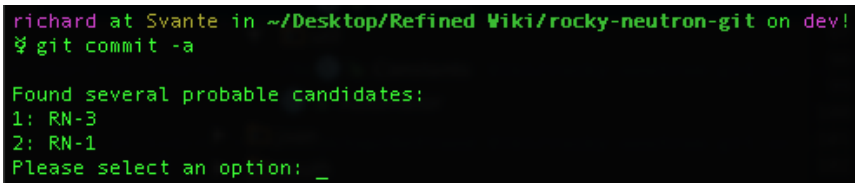

Figure 5.2: The issue selection screen

The third and final part of the automation system is the generation of pull request. If one works using a workflow similar to Git Flow [8] pull requests are a great way to integrate code review and merging into a single process. However in order for reviewers to better understand what has been implemented in the current pull request a good practice is to enter the ticket IDs in the PR description. This is automated by parsing the previous commit messages and extracting references to tickets. The user simply enters the title of the pull request and then it's created automatically and the Bitbucket page with the pull request is launched in a

browser. Here the user can see which tickets were included, make final edits and add reviewers as required.

The automated generation of pull request messages can be expanded using Bitbucket's JIRA integration which provides users with links from valid ticket IDs on Bitbucket's site directly to JIRA. This provides even more context beyond the scope of the tool developed as a part of this project.

Chapter 6
Evaluation

This chapter will cover how the tool was evaluated using external evaluation. It will give some details about who was contacted and in what form the reviews were done.

6.1 Evaluation Method

Two evaluations were made during the development, an initial after approximately five weeks of development, and a second one near the end of the development cycle. The purpose of these were quite different, the first one being meant for inspiration and discovering new ideas for the future development. The second one was intended to review the work done and conclude whether or not the prototype had achieved what it set out to do.

The first round of evaluation was made through a combination of internal review (At RefinedWiki) and interviews with external parties with the focus being on external parties. Additional interviews were performed with the same persons as the initial round of interviews (See Section 3.1) in order to continue the same discussion with a rudimentary prototype as a basis. This provided many great ideas for how to continue development as well as providing details on other types of traceability with which the system could be expanded. These interviews were performed in a very informal way compared to the first round, leaning more towards open discussions rather than strict interviews. The purpose of this was to achieve a more creative discussion where the interviewees would feel comfortable to bring up ideas either indirectly or directly. None of the questions focused explicitly on the automation of traceability link generation but rather on what types of traceability links the interviewee would be interested in, using the prototype as a basis for discussion about what could be achieved.

In addition to this the product was presented at a Configuration Management Coffee Meeting (CMCM) which was arranged by Dr. Bendix and attended by several industry professionals from different companies and with different backgrounds as well as a student. The meeting consisted of a round-table discussion which provided a wide range of different views and opinions on traceability.

Finally the prototype was tested by the author during his continued work and tested in shorter sessions by other team members at RefinedWiki. This provided some hands-on experience with actually using the prototype and lead to some

rapid improvements in terms of user experience.

At the end of the second round of development an additional evaluation was carried out. This focused on reviewing the work in the context of the requirements set up in Section 4.1. Because of this it was mainly carried out internally at RefinedWiki however some discussions with developers at other companies was made.

While this approach to reviewing gave some great feedback and was a good contribution towards making the project as successful as it was it could had been done a bit differently. Instead of having two longer review periods it would probably had been better to divide the project in to shorter iterations with reviews after each iteration. This would had been ideal as it would provide faster feedback and allow the project to change direction faster if required. However it would not have been realistic since the feedback required interaction with industry professionals in longer sessions which required a lot of time. It was not realistic to demand this amount of time from these interviewees.

6.2 Implementation Feedback

This section will cover the feedback given about the prototype. It will cover how feedback was gathered as well as what was said about the prototype and what ideas were gathered from the feedback interviews.

6.2.1 Interviews

Interviews were performed with Person A from the interviews prior to development as well as a person working with configuration management in a large company referenced as Person B. Both the interviewees proved to be great sources of inspiration and insight, coming up with several improvements to the software as well as other potential areas where traceability could be automated. Person A had several ideas about how the tool could be improved and how it could be extended to new areas. Several of the traceability types mentioned in Section 7.4 are based, either directly or indirectly, on suggestions provided by him during the interview.

Person B's interview also yielded some great input. He confirmed the hypothesis that developers have a hard time providing good traceability data, either because they actively don't want to participate or because they make mistakes leading to errors in the data. The company where Person B works has implemented several different processes in different areas, for example strict rules regarding commit messages. However most of these rules do not work out since developers don't want complex processes getting in the way of actual development. Instead Person B brought up the need for automated tools which support the developers rather than restricts them. Another area he mentioned was the possibility to trace commits to code reviews. In the process used by Company B the code reviews are made in a system separate from the version control tool. This means that there is no real verification that it's the commits that have been reviewed that actually are the ones being merged into the development branch. This is of course a very specific use case, since if the merging and code reviews are integrated this problem

is eliminated automatically. However it does show the need to include traceability in the code review process.

6.2.2 Configuration Management Coffee Meeting

The round-table discussions of the CMCM (Configuration Management Coffee Meeting) provided a wide array of different views on the subject because of the varied background of the attendees. An interesting point which was raised was whether or not there was actually any need for traceability what so ever, a question which clearly divided the meeting's attendees into two groups. Those who did not see the need for the traceability links were also further divided into two groups, some did not see the need at all, others could not see how it was a problem to generate it. Those who didn't see the problem generally had a good process in place which was respected by all developers. This means that the traceability is in place "automatically", provided that all new employees are schooled in the process employed.

The other category, those who didn't see the need at all, generally worked in smaller teams with less critical products that are easily updated. The smaller organizations generally means that communication is easier and if the products are non-critical there is never any real need to provide the traceability data to any external parties (Such as government investigations or law enforcement). Unlike some attendees working for a company which provided security systems for, among other things, nuclear power plants where external parties require a high level of traceability. As an example, the US Department of Defense prescribes that requirements should be traceable through design and tests to the end-product, the code in this case. [25, p. 19]

Chapter 7
Results

This chapter will detail the result of the development. This means detailing what traceability could be presented using the existing data as well as what additional traceability can be presented using new data and processes.

7.1 Evaluation

Evaluating the tool proved hard because of technical constraints. As mentioned earlier the version control tool used by RefinedWiki is Mercurial, which proved hard to implement support for. This meant that the original pool of testers was lost and in the end tests had to be carried out through demonstrations rather than actually using the tool. The result was that no real data could be gathered regarding how well the tool worked, however when used by the author and following the previously mentioned workflow the accuracy was near 100%. This is of course greatly biased and further tests would have been useful.

7.2 Gathered Data

The tool's focus was to gather traceability data between code and tickets. This was achieved through tagging each commit with a ticket ID, referencing the work being done. This process was automated, meaning that the traceability between code and tickets comes more or less for free. While the prototype for this tool does present the data in a way which is satisfactory for determining the function of the tool, there are other plugins to JIRA available which do a better job (See Section 3.3).

Instead, the main focus of the tool was to provide an automated and user-friendly way to establish the traceability link with minimal effort from the developer. This means that the tool developed can gather the data and establish the links but leave it to other implementations to present the data.

In addition to gather traceability data between code and tickets the tool was expanded to provide an additional layer of traceability, namely to code reviews in the form of pull requests. This means that the persons reviewing the code get additional context when it comes to determining what has been implemented by the code being reviewed, making it easier to review.

7.3 Required Workflow

While it was the goal to create a tool that would be as wokflow independent as possible some constraints on the input are of course required in order for a tool to work properly. In this case an attempt has been made to ensure that as much of the user provided data as possible is given to the system as part of tasks that still have to be performed. For example the system uses the ticket's current JIRA state and assignee to generate the commit message. Providing this data is done though JIRA by simply assigning the ticket and changing its state, something which most companies already do. The workflow presented here is the minimal recommendation in order for the tool to work properly. If one desires to have a more complex workflow that builds upon this then the tool should work fine provided that the following steps make up a part of it.

In addition to this the automatic commit message generation relies on a good branch model since it only searches the current branch for mentions of tickets. While this does work somewhat in a single branch it greatly reduces accuracy since there is no clear divider between tasks. The ideal workflow would look like this:

1. Begin work on ticket
   (a) Mark ticket as "In progress"
   (b) Assign ticket to developer
2. Create a branch for the ticket
3. Make an initial commit (This will show the screen in Figure 5.2)
4. Make any additional commits on the same branch (The work can be on different tickets as long as they are assigned to the developer)
5. Finish work
   (a) Merge the branch when the ticket is done
   (b) Mark the ticket as done

Finally the user's environment has to be configured with the provided hooks for each repository. The system also uses the developer's Git email address (The one set using `git config -global user.email`) to match developers against users in JIRA. Because of this it's important that these two are the same.

7.4 Types of Traceability

In order to answer the second research question mentioned in Section 1.4 a framework was constructed in order to evaluate the automation of traceability in different aspects of software development. This section will describe each type, motivate the purpose as well as rate them based on the difficulty to implement that feature. The difficulties are rated *Low*, *Medium* and *High* based on how easy they are to automate. Low means that it's practically free with current tools, Medium means

that there are tools readily available although some configuration may be required and High means that no automated solution exists. In addition to this the need for this type of automated traceability is presented on the same scale, ranging from *Low* to *High*.

Tracing Tickets to Code

This is an example of a traceability type which was implemented as a part of the prototype. The purpose is to provide the reader with an idea of how this framework relates to the actual implemented product. In this case the example is briefly presented and the reader can interpret how the framework is to be read by comparing it to the previously documented prototype.

Description

Tracing from tickets to code provide a way to see what code has implemented what tickets as well as what parts of the code base have been affected by a certain ticket.

Motivation

The main reasoning behind this is to provide context for developers and other members of the team. Reading of code is greatly simplified if it can be linked to a tangible ticket where comments, reporters and a description of the issue can be seen. This means that a developer viewing the code can easily get a clear picture of why a certain implementation has been made. In the reverse direction it gives viewers of a ticket a better picture of the impact that ticket had on the code base, what files changed and how they changed.

Difficulty *Low*

Since this form of automation was implemented as a working proof-of-concept as a part of this thesis it should not be difficult to expand it to a fully working product.

Need *High*

This was found to be the single most important form of traceability link for developers when analyzed prior to the development of the prototype. As a result the prototype was focused on this type of link.

7.4.1 Tracing Issues to Tests

During the interviews one thing that came up several times was the ability to trace issues to tests and vice-versa. This would enable developers to react quicker to failed builds and as a result reduce the time it takes to fix errors. It would also be much easier for a company to create a standard regarding how failed tests cases should be reported since if the reporting is automated it can be performed according to a template. The opposite direction, from issues to test cases, could mean that only individual tests could be run in order to determine if an issue has been fixed. Eg. if an issue has been found as a result of a certain failed test case, when a commit is made that references that issue the tests referenced by the issue runs in order to determine if the issue still exists. If the tests succeed the build system could continue according to the company's standards, for example running integration tests on the entire system to make sure that the fix works with all parts of the code.

The need for this traceability type was mentioned by several of the interviewees. Person A mentioned the need for traceability from issues to tests since the company's code base is large, builds are slow and there are plenty of tests. Waiting for hours in order to get feedback does not suit the agile methods very well, so only re-running affected test cases could prove very useful when it comes to reducing feedback cycle times. The other feedback given during the interviews was regarding the ability to trace from tests to issues so that issues would be opened automatically on failed builds if certain criteria is met. Having well-defined criteria is important since for instance opening an issue on a failed build in the development branch would probably generate many unnecessary issues. automatically testing the software against newer versions of dependencies and generating issues for failed builds could give the team a heads-up about potential integration issues with other software.

Description

Tracing issues to tests means providing a way to see which failed test cases generated a specific issue.

Motivation

In large systems with long test times, tracing failed tests to issues can greatly improve feedback times for issue solving. This would mean that a build could be made and the failed test cases could be ran first, providing an initial indication of whether or not the issue was solved. If the test cases don't pass then the issue is of course not solved and the issue tracking system would prevent closing of the issue. Both interviewees in Section 6.2 mentioned the need for this type of traceability but in different contexts.

Difficulty *Medium*

A naive solution would be to simply automate the opening of an issue once a test case fails. Depending on how the specific product and development organization looks this could be an effective way. However it disregards the fact that failed test cases could have a common cause. An example of an automated tool which takes this into account is one developed during a Master's Thesis project at Lund University called NIOCAT. The tool groups similar test failures together and could most likely easily be expanded to, with minimal user input, link those groups to issues. [26]

Need *High*

Considering how it was mentioned by several persons independently there is a definite need for this type of traceability. If it can be well implemented there is a good chance that it can simplify maintenance for teams and improve the change of discovering integration difficulties early.

7.4.2 Tracing Requirements to Tickets

Certain companies, including some whose representatives where interviewed, use a combination of agile and waterfall-like development methods. These companies use waterfall style requirements for the overarching product and then divide these down into something more friendly for agile projects, such as stories. However this can mean that a large amount of data is lost in this transformation, for example why a specific story exists. If one can establish traceability links between these requirements and the tickets (Which can represent stories) it would mean that the lost data can be recovered.

An example scenario where this could be useful is if the company has a number of customers, each with their own set of requirements. The company then turns these requirements into stories and divides them up in sprints. When a sprint is done a release is made, however the customer may not want to receive the release unless one of their requirements have been implemented. If traceability is properly established one can clearly see which customers should receive the current release since there should be a trace back to these customer's requirements.

Description

If an organization uses regular requirements this type of tracing establishes a link between those and tickets (I.e. stories and issues) in an issue tracking system. This means that one can know what requirements are affected by a certain bug and also what requirements get implemented as part of a story.

Motivation

In an organization where both requirements and agile methodologies are used, establishing a link between the requirements and tickets provides a clearer picture of what is actually implemented. An example is if two customers have two different set of requirements which have some overlap, it can be determined if a customer is affected by a certain issue and hence if an update needs to be sent to that specific customer after the issue has been resolved.

Difficulty *High*

As far as the research in this thesis goes no clear way to provide this traceability link has been found. One potential solution would be to use a requirements engineering tool which can trace requirements to a customer. Through this one can trace the customer to tickets and as such establish a link back to the requirements.

Need *Medium*

This is a type of traceability that only relates to a specific subset of companies. As such the need is not as high as some of the other types mentioned previously.

7.4.3 Tracing Code Dependencies

Code dependencies can be seen as one form of traceability, tracing what code requires what other code in order to function. Traceability between code artifacts is something that's normally handled by a compiler or some form of language syntax such as Java's `import` statement. How these dependencies function differs between languages, in Java's case it's an explicit dependency between classes which implies a certain amount of traceability between files since a class is normally contained in a class file. In C and its derivatives however, dependencies are set explicitly between files and the direct traceability between classes (In for instance C++) is only optional. In addition to this there are languages where explicitly stating dependencies isn't required, such as JavaScript. In languages like this all dependencies are implied and established at run time and without the help of external tools it's virtually impossible to keep track of traceability links during development.

The need for this type of traceability has become apparent at RefinedWiki due to the fact that a large part of development takes place in JavaScript. When a project consists of a large amount of JavaScript files it can be hard to keep track of which files that depend on what. In addition to this, dependencies aren't always simply direct method calls but can be implied through an observer-like pattern which complicates matters even more. There is a clear need for a tool which can analyze the code and produce a readable form of traceability between different code artifacts.

Description

Tracing code dependencies means taking care of traceability links between code artifacts. In addition to the support provided by syntax and compilers in certain languages, other languages might require the manual creation of these links. These languages don't have support for explicit dependencies between different parts of the code but instead rely on a system of implied dependencies.

Motivation

Keeping track of these implied dependencies manually is tedious and not a viable strategy as companies and code bases grow. More and more of web focused development is leaning towards client-heavy applications which means that JavaScript has risen in popularity over recent years. The result is that the need for this type of tool increases. In addition, this type of traceability link is one which rapidly becomes impossible to document manually and instead developers keep it in their heads. This means that the quality of the traceability data will depend on the communication within the team and the individual team member's ability to memorize complex dependencies.

Difficulty *High* and *Medium*

The reason this item has two different difficulties is that it depends on how the solution is implemented. A generalized solution would most likely be extremely difficult to implement considering the number of cases that need

to be taken into account. On the other hand, a solution specialized towards a specific language could probably be implemented with significantly less trouble. An example of this is Require.js which strives to create an `import/include`-like environment for JavaScript. [27]

Need *High*

As mentioned on the Require.js page many web sites are transitioning into becoming web applications instead. This means relying more and more on front-end code which, in the case of web development, is JavaScript. In addition to this, several other languages don't completely support the amount of traceability that is satisfactory for a large project. These two combined mean that the need for automating this type of traceability is high in spite of many languages having partial support for it built-in.

7.5 Future Improvements

This section will list some potential improvements of the tool. The idea is to create a list of requirements which were brought up either during the initial research or in one of the feedback interviews. This means that these ideas have all been analyzed as a part of this report but due to the limited time available for development did not make it into the prototype.

7.5.1 Push Hook

In its current form the system uses a REST API using which the client-side hook communicates with the server. While the API does use OAuth 2.0 for authentication, exposing data externally naturally increases the risk of vulnerabilities. In addition each client needs to authenticate which creates a risk that one of the client's data will leak if a device is stolen or compromised. Furthermore, JIRA does not provide any granularity when it comes to OAuth authentication meaning that a user authenticated via OAuth is equivalent to a logged in user. As a result any admin users can perform admin-only actions via the REST API if an attacker was to steal the credentials, which are stored in plain text in a properties file next to the hook.

Instead of putting this much logic in the client a transition to a sever-side hook would be a great improvement. The result of this would be that, if the JIRA and Git servers are run on the same machine, no REST points would be needed. If they are on different machines a separate account could be created for the hook to run on, making it easier to keep secure. In addition to this a server-side hook can not be skipped by developers who don't want to adhere to the rules. This of course can also be drawback since sometimes skipping is needed so a recommendation is to still create some possibility of skipping the verification using certain keywords. However the use of keywords in the message would still create a permanent record that the verification has been skipped. A final positive thing about this change is that it supports even more workflows. Some developers like to make commits to their local repositories which they then tidy up prior to a push, squashing commits and rebasing as needed. This is not quite supported with the current system as it

does not deal well with rebase and does a check against JIRA for every commit. Performing this check on push instead would make much more sense as that is the time when the commit is finalized.

The reason this feature was never implemented is that it would require more advanced support for hooks on the server side. Bitbucket only supports implementing notifications about incoming pushes but there is no possibility to deny these pushes or perform actions with results on the Git server. Instead a Stash implementation or similar server software would be needed and it was not available during this project.

7.5.2 Improve History Searching

As discussed in Section 7.3 the prototype requires a specific workflow as it is designed now. Among other things it does not support using a single branch for all developers and features. Improving this in some way is almost required in order for this to become a viable product in the end. It would also align well with the idea to support multiple workflows and not lock developers in with a process enforced by tools.

An easy way to make it more robust is to improve the searching of commit history. This could be done by searching only for commits made by the current user. In a single shared branch the system would then parse a greater number of commits in the history (It currently parses twice the average number of commits per ticket for the current user or until the current branch ends) and gather only commits by the current user. This would improve flexibility and allow for the use of a single branch for all developers or using a shared branch per team which is perhaps a more common approach. If one looks at the results from the survey regarding branching strategies (See Figure C.7) one can see that 6% of companies branch by team, 13% don't use branches at all and 41% branch only by release. All of these scenarios means that branches will be shared among users without a clear division between different features.

This feature was never implemented due to lack of time. The prototype was prioritized based on the results seen in Figure C.7 and since branching by feature was one of the larger strategies which also simplified development by a significant margin it was chosen to focus on this strategy.

7.5.3 Expand Trace Generation

While the research for this thesis was made similar research has been conducted elsewhere. For example Tien-Duy et al. similarly concluded that there is a need for automating traceability links between code and issue reports. Their approach revolves around analyzing the contents of the commits and their context. As a result it works very well on existing repositories and does not seem to add any requirements to the development process. They take a number of factors into account, not just the contents of the commit message, in order to generate the trace between the commits and tickets. [28]

Expanding the tool that has been presented in this thesis to use similar methods in order to generate the traceability data could greatly improve the utility

and accuracy. This is the most important improvement when it comes to making the tool more versatile and getting away from the required workflow mentioned Section 7.3 and allowing the tool to support any type of development process. In addition to this, Tien-Duy et al. mentions the future possibilities that open up when using more advanced classification techniques for commits and issues. Among other things it's possible to perform statistical analysis on the code in order to determine what parts of the code cause the most bugs or the most severe issues. This gives can give developers pointers regarding what parts of the code need refactoring and where the effort is best spent.

Chapter 8
Discussion

This chapter will contain some reflections about the results. It mainly focuses on the work process, describing the experiences gained from the prototype development, as well as the reviews. It will also attempt to draw some conclusions from the feedback provided by different companies regarding where the need for traceability is the largest.

8.1 Prototype Feedback

A clear conclusion that can be drawn from the feedback is that the need for traceability, and as a result the need for automation of traceability, varies greatly in different organizations. While a few common traits can be seen among the organizations not wanting traceability the connection is not particularly obvious. Smaller team size tended to reduce the need for traceability however certain teams (Such as the one at RefinedWiki) still see the need for this. The same reasoning can be seen with products that aren't critical, such as apps or other types of consumer products. This is rational since that type of product is generally easy to update and any issues which might arise from lacking traceability generally don't have a large impact. Compared to business products where update cycles may be several years and damages caused could result in major monetary losses or even deaths. An interesting discussion about this arose during the CMCM where different persons from different companies had widely differing views regarding the need for traceability. While not all can be explained with purely technical reasoning, some of it is most likely related to company culture and the preference of employees or more likely the persons in charge.

8.2 Traceability Automation

It's clear from the results presented that there is room for improvement when it comes to automating traceability. There are several areas in which the traceability data that exists today can be automatically generated in order to improve the accuracy of the data or the convenience of its generation. Furthermore, several of these traceability types can be automatically generated and maintained without requiring major efforts. Some tools are already available for performing tasks like this within specific areas meaning that the implementation or extension of such

tools should be possible. In the areas where no tools are known to be available automation is naturally more difficult but even those tasks seem far from impossible based on the analysis in this thesis. Instead the area seems like it's simply been overlooked

8.3 Limitations

8.3.1 Limited Sample Size

An obvious problem with the studies performed as part of this thesis is the sample size. While great care has been taken to ensure that representatives of a varied group of companies gave their input the sample size is still small. Interviews were made with less than ten persons meaning that the major part of the views upon which this work is based represents only very few companies and professionals. In addition to this the survey, which was made to verify the views expressed in the interviews with a larger pool of persons only reached approximately 70 persons. In the context of the number of software development companies in the world this sample size is quite small. However considering that the research was made by a single person over the course of 20 weeks the conclusion is that the sample size is good enough although further research is required in order to draw any wider conclusions.

The same issue is present when it comes to the review. Since no survey was made to review the prototype the sample size is even smaller. However this is somewhat compensated by the fact that the interviewees were to a great extent persons with great amounts of experience meaning that they were hopefully able to share insights based not only on their own situation right now but rather on a more general level.

8.3.2 Workflow

If the developer follows a simple process (As described in Section 7.3 the success rate for the automated commit message generation is near 100%. However if one starts to deviate from this process the success rate does of course go down. Although even when not following this process the success rate is quite high but the number of close cases increases, especially if the team uses a common branch since this makes it harder to determine the number of commits which should be searched.

Part of the goal with this project was to develop a tool which doesn't require any additional manual effort from the team and because of this and requirements on the process must be seen as something of a failure, however small it may be. In this case the author has worked hard throughout the project to make sure that the required extra labour required was kept to a minimal and one can conclude that the resulting process is one that is already followed by many companies. In addition several parts are not required for the tool to function but simply easy ways to improve its function, furthermore they are all great practices which companies most likely will follow anyway thanks to their many other benefits.

8.4 Future Work

Since the development of the tool was simply made as a proof-of-concept there is much room for improvement there if it is to become a usable product. The future potential developments that have been analyzed to a greater extent are covered in Section 7.5. These included moving some of the functionality to the server side in order to be able to prevent commits from coming in. Currently the check can be skipped if the developer would like to do so. In addition it was discussed how history searching could be improved in order to minimize the required workflow as well as expanding the tool to look at other factors than commit message in order to determine which ticket it belongs to.

Firstly the algorithm mentioned in Equation (5.1) for determining what ticket is being worked on is very basic. This could be expanded with something more advanced that not only examines the current branch but uses a larger data set and a better, more accurate, mathematical approach.

Secondly, there is currently no support for automatically referencing multiple tickets in a single commits. This could prove a useful feature to add as the situation will surely arise where a developer sees it fit to work on several tickets in a commit.

Chapter 9
Conclusion

Going back to the original research questions, trying to judge how difficult it would be to automate traceability between code and tickets as well as finding additional types of traceability which can be automated. First it was established that a pressing need for automated traceability exists since the manual practices aren't sustainable in the long run. Industry professionals from several companies were interviewed and additional were surveyed in order to establish the current state of the industry.

The first question was answered in the form of a prototype designed to provide a proof-of-concept that such a tool could in fact be implemented. Over the course of the project this tool was implemented and reviewed both by colleagues at RefinedWiki as well as several external professionals. In order to automate this traceability the tool analyzed the user's previous commits as well as the state of tickets in an issue tracker in order to determine what item the user is working on. As soon as the tool was in a working state it was also used during the developer's continued development of the prototype. This provided some hands-on experience with how it worked and its success rate. This proved to be high provided that a few basic constraints were put on the workflow.

In addition to this several other traceability types were analyzed and categorized with regards to how they could be automated. This was done by setting up a framework using which the traceability types could be analyzed. The idea was to cover a wide array of traceability types in order to answer the second research question regarding what other types of traceability can be automated.

References

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

[2] P. Antonino *et al.*, "A Non-invasive Approach to Trace Architecture Design, Requirements Specification and Agile Artifacts," in *Software Engineering Conference (ASWEC), 2014 23rd Australian*, April 2014, pp. 220–229.

[3] K. Beck *et al.*, *Manifesto for Agile Software Development*, 2001.

[4] U. Asklund, L. Bendix, and T. Ekman, "Software Configuration Management Practices for eXtreme Programming Teams," in *11th Nordic Workshop on Programming and Software Development Tools and Techniques - NWPER'2004*, August 2004.

[5] M. Jacobsson, "Implementing traceability in agile software development," Master's thesis, Lund University, Faculty of Science, Lund, Sweden, 2009.

[6] B. Ramesh and M. Jarke, "Towards Reference Models For Requirements Traceability," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, 2001.

[7] R. Dömges and K. Pohl, "Adapting Traceability Environments to Project-specific Needs," *Commun. ACM*, vol. 41, no. 12, pp. 54–62, Dec. 1998.

[8] *Gitflow Workflow | Atlassian Git Tutorials*. [Accessed: 2015-02-23]. [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[9] O. Gotel and A. Finkelstein, "An analysis of the requirements traceability problem," in *Requirements Engineering, 1994., Proceedings of the First International Conference on*, April 1994, pp. 94–101.

[10] S. Berkczuk, B. Appleton, and R. Cowham, "The trouble with tracing: Traceability dissected," *CMCrossroads*, November 2005.

[11] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Standard 610, 1990.

[12] *CMMI for Development, Version 1.2*. Software Engineering Institute, 2006.

[13] VersionOne. (2013) *8th Anual State of Agile Survey.* [Accessed: 2015-02-24]. [Online]. Available: http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf

[14] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[15] *Git - Git Hooks.* [Accessed 2015-04-09]. [Online]. Available: http://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks#Server-Side-Hooks

[16] Atlassian. *JIRA - Issue & Project Tracking Software.* [Accessed 2015-04-09]. [Online]. Available: https://www.atlassian.com/software/jira

[17] K. Kowalczykiewicz and D. Weiss, "Traceability: Taming uncontrolled change in software development," in *proceedings of the IV National Software Engineering Conference*, Tarnowo Podgorne, Poland, 2002.

[18] J. Farah, "Traceability and Auditability: Satisfying Your Customers," *CM-Crossroads*, August 2007.

[19] K. Mohan *et al.*, "Improving Change Management in Software Development: Integrating Traceability and Software Configuration Management," *Decis. Support Syst.*, vol. 45, no. 4, pp. 922–936, Nov. 2008.

[20] Atlassian. *JIRA integration.* [Accessed: 2015-04-02]. [Online]. Available: https://confluence.atlassian.com/display/STASH/JIRA+integration

[21] ——. *JIRA DVCS Connector Plugin.* [Accessed: 2015-04-02]. [Online]. Available: https://marketplace.atlassian.com/plugins/com.atlassian.jira.plugins.jira-bitbucket-connector-plugin

[22] S. Lauesen, *Software Requirements: Styles and Techniques*, 1st ed. Addison-Wesley, 2002.

[23] Will. (2010) *Working With Named Branches in Mercurial.* [Accessed: 2015-02-24]. [Online]. Available: http://humblecoder.co.uk/blog/2010/02/24/working-with-named-branches-in-mercurial/

[24] *Git - git-notes Documentation.* [Accessed 2015-04-21]. [Online]. Available: http://git-scm.com/docs/git-notes

[25] *Department of Defense Handbook - System Requirements Document Guidance.* MIL-HDBK-520A, 2011.

[26] N. Erman and V. Tufvesson, "Navigating Information Overload Caused by Automated Testing," Master's thesis, Lund University, Faculty of Science, Lund, Sweden, June 2014.

[27] *RequireJS.* [Accessed 2015-04-30]. [Online]. Available: http://requirejs.org

[28] T.-D. B. Le *et al.*, "RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information," in *proceedings of the International Conference on Program Comprehension*, May 2015.

Appendix A

Interview Questions

The interviews were performed in Swedish and the questions have been translated to English.

1. What is your current role?
2. How large is the team you work with?
3. What configuration management related tools do you use? This includes VCS and issue trackers for example.
4. What development method do you follow? Agile?
5. Do you recognize the need for traceability?
   (a) From what situation?
   (b) How did you solve it?
6. What type of traceability would you be interested in?
7. Do you work with traceability today?
   (a) How? What information is required from your developers?
   (b) If not, how come? What obstacles do you see?
8. How much time do you think people are prepared to spend to achieve better traceability?
9. How do you work with branches?
10. Do you have any specific branching strategy?

Appendix B

Survey Questions

1. What size is the team you work in?
    - 1-10 people
    - 10-50 people
    - 50-100 people
    - More than 100 people
2. What version control system are you using today?
    - Git
    - Subversion
    - Mercurial
    - Team Foundation Server
3. What bug tracking system do you use?
    - Free-text answer
4. What is your main area of responsibility?
    - Development
    - CM (Configuration manager or similar)
    - Management (Also includes Scrum Master or similar)
    - Requirement engineer
5. What type of traceability are you interested in?
    - Requirements > Code (Also includes tests)
    - Code (Also includes tests) > Requirements
    - Requirements > Requirements
    - None
6. How important do you feel traceability is?
    - Scale 1-5

7. How much extra effort would you be prepared to spend in order to gain traceability?
    - Less than 1%
    - 1%
    - 5%
    - 10%
    - More than 10%
8. What branching strategy does your team mainly use?
    - No branches
    - Branch by feature, story, ticket etc
    - Branch by developer
    - Branch by release
    - Branch by team

    If the answer was Branch by feature:
9. Do you use any specific pattern?
    - Yes
    - No
10. Do you employ any particular guidelines for commit messages?
    - Yes
    - No

    If the answer was Yes:
11. What type of guidelines?
    - Free-text answer
12. Email Address (Optional)

Appendix C

Survey Results

This section highlights some of the most important results from the survey.

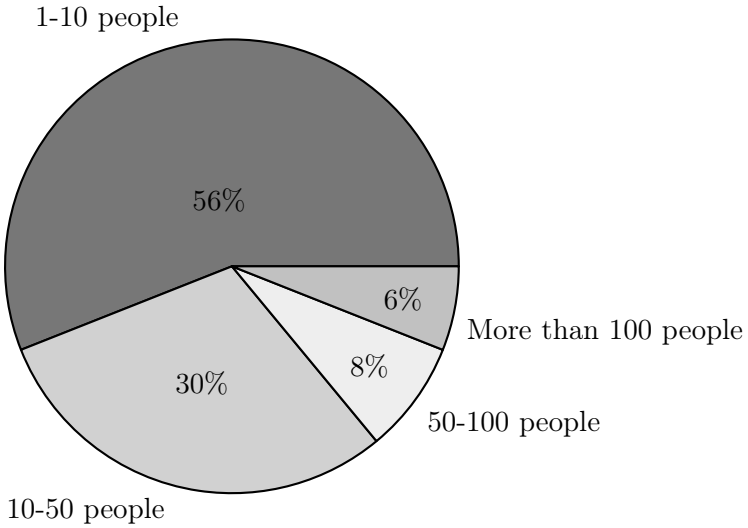

Figure C.1: What size is the team you work with?

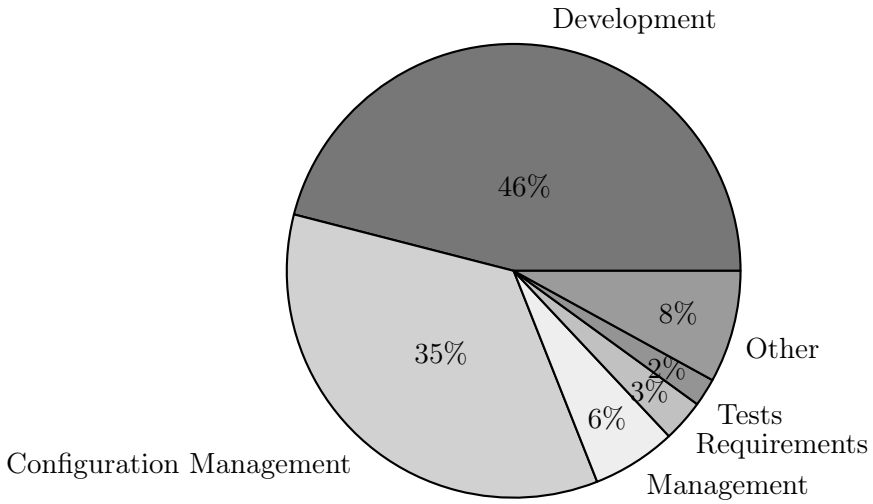

Figure C.2: What is your main area of responsibility?

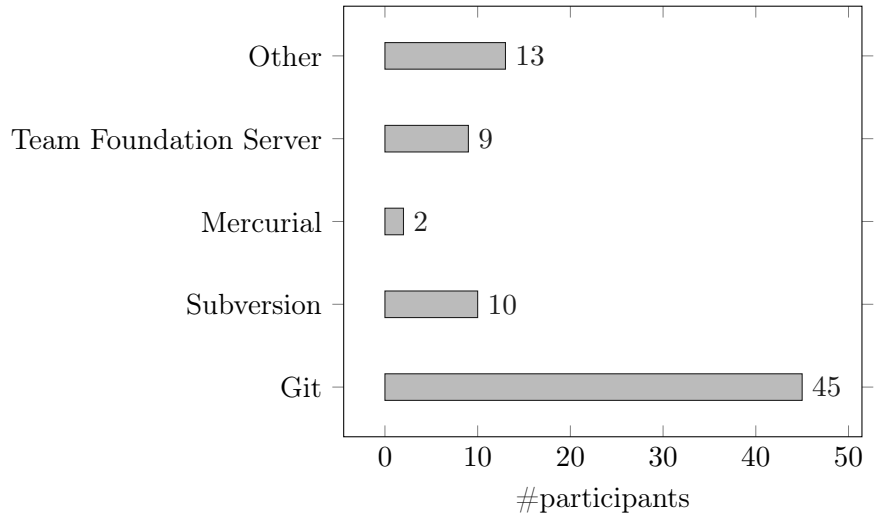

Figure C.3: What version control system do you use?

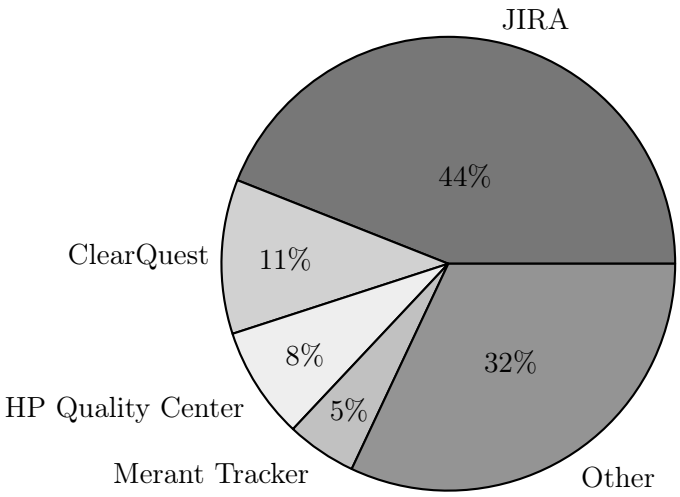

Figure C.4: What issue tracking system do you use?

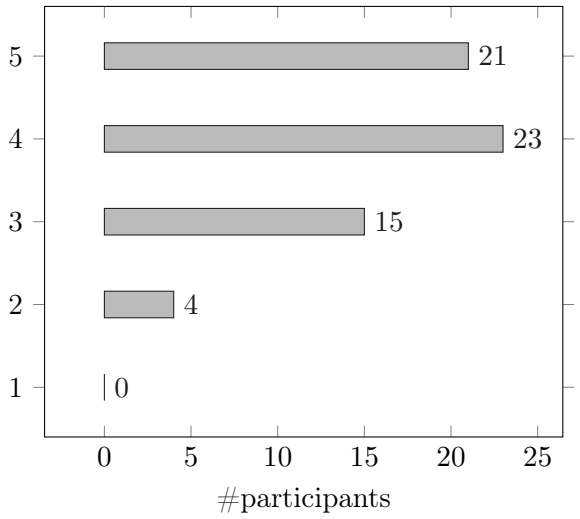

Figure C.5: How important do you feel traceability is?

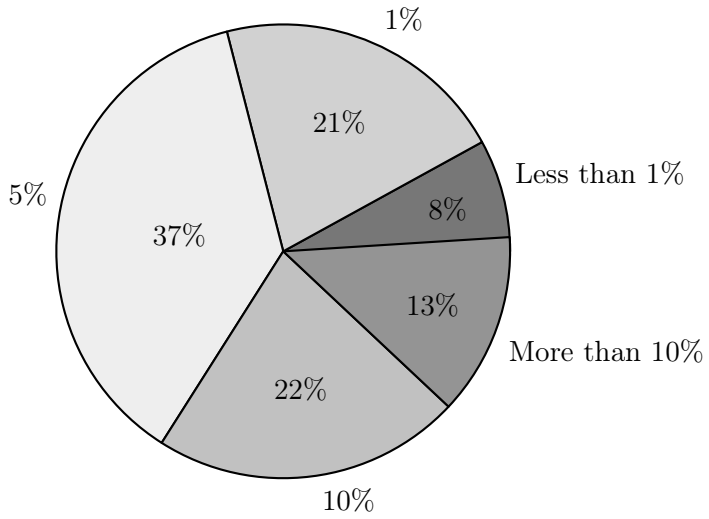

Figure C.6: How much extra effort would you be prepared to spend in order to gain traceability?

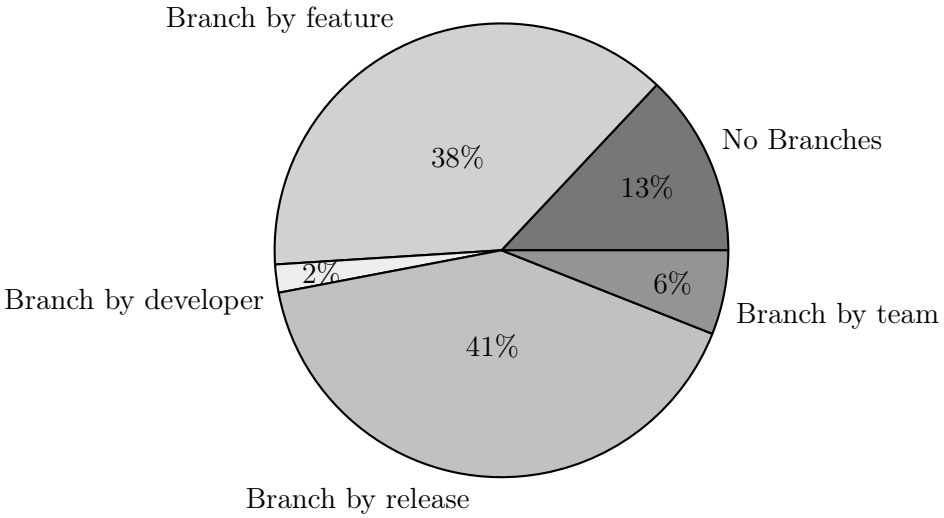

Figure C.7: What branching strategy does your team mainly use?

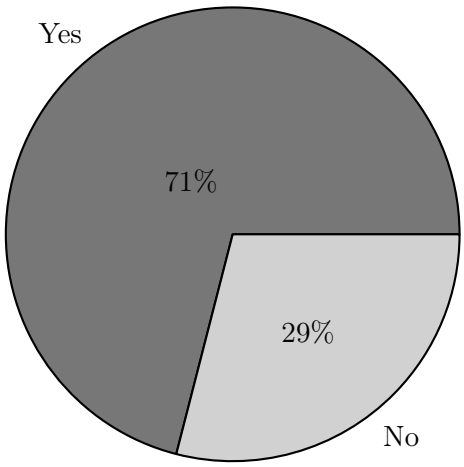

Figure C.8: Do you employ any particular guidelines for commit messages?

Automating Traceability in Agile Software Development

THESIS SUMMARY
RICHARD SIMKO

Supervizor: Lars Bendix (LTH), Emil Sjödin (RefinedWiki)
Examiner: Ulf Asklund (LTH)

Keeping track of why changes are made in software development projects becomes harder and harder as sofware grows more complex, projects become larger and less centralized and more and more companies shift to agile. This leads to developers wasting a lot of time manually documenting why their changes are made. This Master's thesis explores the ability to automate these tasks.

The Current State of Traceability

When developing software today most functions of traceability are manual. For example when tracing changes in code to requirements, change requests, bug reports or similar, this is done through the developer writing the ID of the work item a particular commit connects to in the commit message. One can easily guess that this is prone to errors since IDs can grow long and complex which increases the risk of a developer inputing the wrong data. This then leads to data which can not be trusted which in many cases can be worse than not having any data at all.

Previous research shows that this is the case as well as documents the need for a more automated approach to generating traceability links to code. In addition to that a survey and a series of interviews are performed as part of the thesis, further strengthening the view that there is a need for automation in this area.

Solution

Instead of implementing a traceability solution which should fit all types of projects and work with every tool, as has been tried previously, this thesis was quickly narrowed down. The focus was to develop a prototype of a tool that can present the developer with suggestions regarding what work items are most likely to have been worked on in his current commit.

This is done by analyzing previous commits by the same developer as well as the state of tickets in the project/issue tracking system. This data is then aggregated and a few best guesses are generated for the developer to choose from. This completely eliminates the process of manually entering numbers and gets rid of the risk that a developer should enter incorrect numbers.

Results

The prototype proved hard to test due to technical constraints. As such no good data regarding its performance could be provided. Instead analysis of its performance was made on a more abstract level, through demonstrations and interviews with developers and configuration management experts. Overall the response was positive and it was commonly said that people wanted a tool like this. However it was interesting to note that there were several, especially CM professionals, persons who could not at all see the need for a tool like this. This proved to be related to how well traceability generation worked in their organization today, since they had good guidelines in place which were followed by everyone the need for a tool was non-existant.

Traceability Automation Framework

In addition to developing a prototype the thesis also creates a framework for analyzing other types of traceability and the possibility for automation. This was purely theoretical and the idea was to uncover which areas would be most interesting to focus on for future researchers. One thing which stood out was traceability links between failed test cases and issue reports, eg. being able to run a smoke test when an issue has been fixed greatly speeds up development. However in order to be able to do this today someone has to manually tie issue reports to failed test cases.