Degree project

Mining Git Repositories
An introduction to repository mining

Author: Emil Carlsson
Date: 2013-08-02
Subject: Computer Science
Level: Bachelor
Course code: 2DV00E
Abstract
When performing an analysis of the evolution of software quality and software metrics, there is a need to get access to as many versions of the source code as possible. There is a lack of research on how data or source code can be extracted from the source control management system Git. This thesis explores different possibilities to resolve this problem.

Lately, there has been a boom in usage of the version control system Git. Github alone hosts about 6,100,000 projects. Some well known projects and organizations that use Git are Linux, WordPress, and Facebook. Even with these figures and clients, there are very few tools able to perform data extraction from Git repositories. A pre-study showed that there is a lack of standardization on how to share mining results, and the methods used to obtain them.

There are several tools available for older version control systems, such as concurrent versions system (CVS), but few for Git. The examined repository mining applications for Git are either poorly documented; or were built to be very purpose-specific to the project for which they were designed.

This thesis compiles a list of general issues encountered when using repository mining as a tool for data gathering. A selection of existing repository mining tools were evaluated towards a set of prerequisite criteria. The end result of this evaluation is the creation of a new repository mining tool called Doris. This tool also includes a small code metrics analysis library to show how it can be extended.

Keywords: repository mining, msr, git, quality analysis, version control system, vcs, source control management, scm, data mining, data extraction
Acknowledgements

First and foremost I would like to thank my supervisor Daniel Toll for all the time and energy he has put into supervising me. He has gone far and beyond what could be expected of him for me with this thesis and the entire school year. He has been a mentor and an inspiration for me to perform at my best.

I would also like to thank Caroline Millgårdh, Fredrik Forsmo, and Josef Ottosson for reading the first drafts, giving me feedback, submitting to being guinea pigs, and testing code examples every now and then.

Last but not least, a big thank you and acknowledgment to D. W. McCray for helping me proofread this thesis.
# Table of content

Abstract
Acknowledgements
1. Introduction
2. Problem
   2.1 Introduction to repository mining
   2.2 Problem background
   2.3 Problem definition
3. Method ontology
4. Current state-of-the-art and practice
   4.1 Selection process
   4.2 Conduction
   4.3 Desired result
   4.4 Threats to validity
   4.5 Study results
5. Evaluation of existing programs
   5.1 Initial testing environment
   5.2 Program selection
   5.3 Limitation
   5.4 Threats to validity
   5.5 Initial testing
   5.6 Runtime testing
   5.7 Results
   5.8 Problems
6. Conclusion of pre-study
7. Mining Git with Doris
   7.1 Program specification
   7.2 Implementation
   7.3 Testing, problems encountered and solutions
   7.4 Practical usage test
8. General discussion
   8.1 Hard to find previous results and software
   8.2 Selecting a tool
   8.3 Ethical issues
   8.4 Git is growing
9. Conclusion
   9.1 Common problems encountered with repository mining
   9.2 Profits of mining Git repositories
   9.3 Successful data extraction for software quality analysis
   9.4 Best practice
9.5 Hypotheses
9.6 Future work

Sources
Appendices

A. Articles included in literature study 1
B. Usage documentation of Doris 2
C. Source code metrics measurement 5
1. Introduction
Repository mining is a useful technique when performing software engineering research. With the help of successful repository mining you can, for example, make links between bug fixes [1], propose other classes that also have been edited when changing a function [2], or measure productivity in software projects [3]. All of these techniques have code metrics and repository mining in common.

To be able to perform an analysis of source code evolution, you need access to at least two states of that source code. The more states you have available to analyze, the finer the granularity of the analysis will be. Repository mining helps to gather many different snapshots of the state of the code. This can be done manually through programs such as Tortoise [4]. But in a research situation, automation can be of great benefit.

When performing repository mining, a researcher is faced with a variety of problems. Planning for these problems is essential to ensure useful results. This paper will identify the problems that a programmer or researcher might come across while performing repository mining. Problems that were discovered in the process of writing this thesis were investigated to see if solutions already existed elsewhere. If no other solution had been found, an attempt to find an implementation of a solution to a similar problem was made; failing that, a theoretical solution was proposed.

This paper is intended to further knowledge of what problems that can be encountered when mining Git repositories. Git was chosen because there is little research made on that version control system. It recently has had a large increment of usage. The result of this thesis is a list of commonly found problems, a test of existing repository mining tools for Git, and a repository mining tool written to match the specifications (section 2.3.1) to mine Git repositories.

When studying articles and papers about the subject it was discovered that repository mining is very common. It is noted by S. Kim et al. [5], and H. Nakamura et al. [6] that there is a lack of sharing how this is done.

The thesis starts with an introduction and a description of the problem. Following that, a short description of the method of research. This description is then further elaborated in three sections starting with a literature study, testing of some tools created to perform repository mining, and an exploration of how to create a repository mining tool for Git. Finally, it is summarized by a discussion chapter and a clarification on what the results of the research are.
2. Problem
In this section there will be a brief introduction to repository mining, problem background and also the problem definition and a hypothesis.

2.1 Introduction to repository mining
A version control system (VCS) is also known as Source Control Management (SCM) is a method to keep track of different milestones. It is sometimes used as a remote back-up of the source code in development of software. How they are used in practice differs from organization to organization [1]. A version control system can host repositories for different purposes. Each repository contains a snapshot of a state of the source code, or other logged data, in what in this thesis is referred to as a commit. Some well-known VCS are Subversion [7], Concurrent Version System (CVS) [8], Mercurial [9] and Git [10].

In this paper, repository will refer to a source code repository stored in a version control system (VCS), where developers can co-operate and back-trace changes in the source code. Repositories contain a lot of information on the progression and metrics of the program in question [11], [6], [12].

Repository mining is used in Computer Science fields highly coupled with the research of software metrics and software development (e.g., [13], [5], [14]). It is used to extract data from a VCS. This data can later be used to perform different kinds of analysis with the help of various tools to see details of architecture, testing, and code duplication, among others.

Git is a decentralized VCS. This means that all information about a repository is stored locally and not at a remote server [15]. Often a central main version of the software is stored where developers merge local branches to and/or pull from that repository.

To gain access to Git’s functionality programmatically, an Application Programming Interface (API) [16] can be used. This is common to use when there is a need to access functionality without using the functions from the existing program’s source or forking the existing program and scraping the output. This simplifies the interaction between applications and makes it easier to use software written in a different programming language.

Software metric analysis is performed to measure different parts of a program’s source code [17]. One example of a metric is source lines of code (SLOC), which simply measures how many lines of source code there are. Another software metric, described by T. J. McCabe [18], is cyclomatic complexity that counts the different paths through the logic of a program. There are various metrics to measure different theoretical and practical information in an application.

2.2 Problem background
Git has, since its release, become an increasingly common version control system used by software projects. The service provider Github hosts approximately 6,100,000 open source repositories. Gaining access to these repositories would increase the code basis on which researchers can perform software analysis.

Repository mining has been used, with great success, with previous version control systems (e.g., Subversion, Concurrent Versions System etc.). The research community has largely not explored this approach with Git repositories. Because of the extended
usage of Git within different open source communities, there is a need to get a working tool to extract data from Git repositories.

The big challenge is to create a tool that can perform automated repository mining with Git. Previous research has mostly used centralized version control systems such as CVS in their research. How to perform repository mining on a decentralized repository differs from a centralized one.

To ensure that the work had not already been made and to get knowledge of the current state-of-the-art, a pre-study consisting of reading previous research and short evaluations of existing tools was performed. This pre-study (section 4 and 5) resulted in a list of common problems and also showed that there was no tool that fulfilled the requirements (section 2.3.1).

2.3 Problem definition
This thesis will tackle two different hypotheses. They are:

1. An existing repository mining tool exists that can extract data from Git repositories.
2. Repository mining can be conducted decentralized repositories in the same way as centralized repositories.

To be able to test these hypotheses six requirements have been created. The requirements are based on the fact that a limitation on what kind of tools is to be investigated.

2.3.1 Hypothesis requirements
The following list is a set of requirements that the final repository mining tool found or created in the line of this thesis have to fulfill.

- Source code should be easily compared between different commits.
- No external dependencies except for the programming language interpreter.
- Full automation in the mining process after being started.
- Verbose reporting of errors when performing a mining session.
- Must handle the version control system Git.
- Work on both Windows and Unix-like.
3. Method ontology

A pre-study was performed (section 4 and 5) to answer how data can be extracted from Git repositories. The main focus of the pre-study was to see if tools existed that fulfilled the requirements (section 2.3.1). The secondary focus was to gain knowledge of repository mining practices and the current state of the art.

This pre-study gave insight on how repository mining is used and also what can be expected of a tool created for this purpose. It also gave better knowledge of what kind of problems could be anticipated (section 4.6). The study also showed that there was no extant tool that fulfilled the requirements. It was also discovered that there few repository mining tools that actually perform extraction of data and store it in a manner so that software quality measurements can be performed easily. Most current available tools work more with the log-system of Git than with the actual source code. The software that was found was hard to modify and use due to a lack of documentation (section 5.7).

The results of the pre-study showed that it was necessary to create a new repository mining tool for Git (section 7). When creating this tool, some hardware related problems were discovered. The problems found are not restricted to repository mining. They are however relevant to performing repository mining.

These problems had to be solved without compromising the requirements previously stated. To ensure that the tool actually would work in a realistic situation where repository mining could be useful a practical usage test implementation (section 7.4) was performed. The development of this program also centered on making the program easy to modify and to create a clear source code documentation and user guide.

The results and experiences gained from the pre-study is then combined with the results and experience of the creation of a repository mining tool to be able to generate an answer (section 8 and 9) to the problem posed: How can data be extracted from Git repositories to perform a software metrics and software quality analysis?
4. Current state-of-the-art and practice
For this thesis, previous work was of great importance. This resulted in a literature study of what researchers have done prior, to get an insight in the field and see what can be expected to come in the future. This will also provide a benchmark for what the current state-of-the-art and the current practice is within the field of mining software repositories.

The papers and articles that have been used for this study can be found in appendix Appendix A: Articles included in literature study.

4.1 Selection process
The articles read for this study were selected through inspecting the abstracts of previous works. Articles which referenced the mining of software repositories qualified for inclusion. To get a wide spectrum the date of publication was not considered.

The papers were found via the Association for Computing Machinery (ACM) digital library [19], and IEEE Xplore Digital Library [20].

Selected papers come from both universities and research departments of private companies; the type of source was not considered relevant to weigh or inclusion a particular mining tool or a particular VCS. The literature study was limited to include no less than 10 and no more than 15 articles.

4.2 Conduction
The articles were read with a focus on theories of how repository mining can be performed. A critical view was held as to the writer’s preferred VCS. Problems were scrutinized: article-specific problems were given lower priority than those which applied in multiple situations. Substantial attention was given to the actual usage of repository mining and the intended results of the paper in question.

4.3 Desired result
The desired result of this literature study was to find problems related to repository mining and solutions for these problems, if available. Another desired result was finding a working repository mining tool using the VCS Git which would fetch source code for use in establishing a metrics pipeline.

4.4 Threats to validity
Setting an upper limit on the number of articles to be included in the literature study introduced the possibility of missing useful findings presented in other articles. It was decided to use findings from a wide range of scenarios, as opposed to focusing too tightly on a specific tool or environment.

4.5 Study results
This pre-study gave some vital results to be able to see if the problem should be further investigated. These results are listed in what could be extracted as the different main difficulties when performing repository mining.

4.5.1 Project knowledge
Knowledge about the project being mined is one important part of achieving a good result. There is often hidden information that can be found with participants of the
project with commit access [5], [13], [12]. This information might be needed to get a correlation between new functions and bug-reports, or refactoring to project size.

Careful selection of a project to study is necessary to overcome these issues. A researcher’s pre-existing relationship with a project can result in biased findings. Project size and current use of metric data are also important points to consider: many smaller projects to not make use of metrics, and a project large enough to do so might quickly generate enough data to overwhelm a researcher. A researcher should also consider his or her communication skills as well as those of the project contact; if a contact does not know that the researcher is interested in certain information, he or she might not pass it on.

One way of handling this could be to use a project that has been used by previously by another researcher. There are some projects that are more popular than others. Most of these are well known open-source project e.g., Apache HTTPD [21], GNOME [22], Eclipse [23], and NetBeans [24]. These are also frequently reoccurring projects used in the papers read for this study. D. M. German [25] also makes a valid point that using a close source project will generally restrict what data you are able to gather, and what data you are able to publish. These problems generally do not apply to an open source project. That said, there are some ethical issues that should be considered, some of which are discussed in section 7.2.

4.5.2 Lack of standard
There are currently no real standard to perform the repository mining. A need for standards and suggestions of such can be found in articles by Kim et al. [5], Matsumoto et al. [12], Nakamura et al. [6], and Kiefer et al. [14]. If some sort of standard were to be imposed on repository mining, researchers could use data mined by other researchers in related work. This is not possible today as the result, and in some cases even the tool used to get said result, is not always accessible to the public.

There is no real solution for this problem, but there is a need for it. This topic could probably be the basis for a thesis on its own.

4.5.3 Knowledge sharing
There is a lot of research based on repository mining. But information on how the practical work has been performed was sparse in the papers used for this study. Programs used to perform the mining can be hard to find. These are either highly customized, difficult to get to start, or out of data (section 5). Another problem with this is that the kind of analysis tool or the mining procedure itself can create different results [12].

When the program no longer is available to the public it will result in a problem, making it impossible to reproduce experiments made by the researcher using that program. This is further discussed in section 7.1.

4.5.4 Differences between version control systems
There is a difference between the version control systems in use. CVS and Subversion differ on file units, numbering, and log formats [12]. The main difference between Git, CVS, and Subversion is that Git is decentralized [15]. There are of course more differences than that, but the other differences are more conventions and program specific implementations. This in turn, sets up the need for a different analytics tool [12] which can work with the specific VCS used. These differences make it difficult to create an “all in one” tool to mine the different version control systems.
To create a repository mining tool that can handle every VCS on the market would be a large project. There are many different VCS on the market. It could be narrowed down by just using the most popular ones, but details of market share among VCS implementations are currently unavailable and not likely to become available. The only figure found was statistic from GitHub [26]. These statistics does include unmaintained projects and non-programming repositories.

4.5.5 Detailed knowledge of version control system
To create a solution for one VCS you need to have good knowledge of the system in question [5]. One thing that is crucial to know is how the log system works. Does each log store information about previous commits or just the current commit viewed? Are there separate logs for individual committer, commentary, and source changes? Will the logs be changed in some form when merges or rollbacks occur? In some cases this information might be unclear or even missing in the documentation [25]. This might also require an effort in reverse engineering that particular repository system.

This is a problem that cannot be prevented. The best one can do here is to either increase the timespan for the work need to be finished or lower the standard for the extractions. To handle this problem, a solution might be to limit the extraction to one single form of version handler, e.g., only use projects that are hosted via Git.

4.5.6 Reinvention of the wheel
When reading papers and articles it can be noted that most researchers reinvent the wheel by writing a new mining program. This is done even if they are after similar information from the repositories. The software is highly specialized to one version handler, e.g., Git, Subversion, and Concurrent Version System (CVS)). There is also a lack of documentation [12] of the applications so they cannot be maintained by others without first studying the source code.

This results in a lot of repository mining programs being available. But most of them are either out of date or do not come with a clear documentation on how they work and how to adapt them to fit your needs. This could be avoided if documentation of current tools and availability of them would increase. Then a more wide spread reuse could occur instead of reinvention.

4.5.7 Non-standardized usage of commits
One problem that was found by Bachmann and Bernstein [1] is that in some projects the repository is treated as a back-up device as well as a version handling system. But in other projects the repository is only used for committing final and working program code.

When doing a commit with the repository as a back-up device there will be source code committed that is non-functional. When compiling or analyzing a commit made for back-up purposes, the source might be incomplete and cause errors. This will require that manual inspection of commits to make sure that it is not a “back-up” commit and not a “functionality” commit.
5. Evaluation of existing programs
The practical experiments are divided into testing of existing tools and creation of a program in Java to create a mining tool. The reasons for testing existing tools were both to find a benchmark and to see if a new tool would have to be created. The second part is to get some knowledge of how much effort is required to create a repository mining tool.

5.1 Initial testing environment
The environment used to test existing solutions was in a virtual computer. The specification of that computer was 2 GB of RAM, 50 GB of hard drive, 2 core AMD Phenom II 965 3.4 GHz processor.

This was emulated with the help of VM Ware workstation on a computer with an AMD Phenom II X4 965 with 8 GB of DDR3 RAM. The underlying operating system was a Windows 7 Professional 64bit.

On this an installation of Mint Linux [27] and a Windows 2008 [28] Server was used as operating system to perform tests. Main tests were done in the Linux installation to get the basic information.

The selection of Mint Linux was based on the fact that it has program repositories containing both GNU is not Unix (GNU) [29] and non-GNU programs. The tester’s previous experience with different Linux distributions’ was also taken into account.

Windows 2008 server was chosen for installation reasons. When trying to install Windows 2008 Server R2 the installation malfunctioned. To minimize the risk for any erroneous behavior because of installation errors Windows 2008 server was chosen instead.

5.2 Program selection
The selection process for programs was based on programs mentioned and used in articles read. This process was to get some guarantee that they could perform repository mining from a research point of view. This cannot always be guaranteed when the program is not written with research as its main purpose.

The actual search for source code or binaries of the program was performed via Google, article texts and references, and also, if needed, the creator of the program was contacted. This time was not included in the testing time (section 5.5). The reason for this not being included in the testing time was that this time is highly individual for the person performing the search, also it does not affect the time that is needed to run a program.

5.3 Limitation
Different initial testing stages were chosen to minimize the time consumption as far as possible. Focus was on making the programs run in a short period of time to be able to spend more time in extensive testing to see how RAM and time consumption in detail for the programs.

5.4 Threats to validity
Programs that were not found in articles used for this thesis were not included. There are however more tools available. Some of these were included if there was a direct connection to Git, and they were found in the progress of searching for mentioned tools.
5.5 Initial testing
For a program to be considered useable there were some main criteria that needed to be fulfilled. These were:

1. Existence of public webpage with download.
2. Date of last update.
3. Handling of Git.
4. Used in articles read for this paper.

If these four criteria could be matched on the Linux installation they were later also tested on the Windows installation.

All programs were set up in accordance to the documentation for that program and results that are expected from it. No changes to the source code were made in this step.

A time limit of one hour was given to each program to go from complete download to runnable. This time also included setup time of dependencies of that particular program that were not reusable over more than one program. These were MySQL database, language interpreter/compiler, etc. The time limit was based on a tight time schedule and to give all programs tested an equal playing field.

The reason that an initial testing phase was used was to save time when finding out the basic functionality of the program. The reason for using both Linux and Windows was to make sure that the program could be used in both environments. A short summary of the outcome of the testing can be found in table 5.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Accessible</th>
<th>Last update</th>
<th>Active webpage</th>
<th>Handles Git</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kenyon</td>
<td>No</td>
<td>Unknown</td>
<td>No</td>
<td>Unknown</td>
</tr>
<tr>
<td>APFEL</td>
<td>No</td>
<td>Unknown</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Evolizer</td>
<td>Yes</td>
<td>Unknown</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>git_mining_tools</td>
<td>Yes</td>
<td>2009-03-26</td>
<td>Yes – via Github</td>
<td>Yes</td>
</tr>
<tr>
<td>Shrimp*</td>
<td>Yes</td>
<td>2012-09-27</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Gitdm**</td>
<td>Yes</td>
<td>2013-04-11</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

| Table 1 Programs tested for repository mining. * = Not only a mining tool. ** = Not mentioned in a paper. |

5.6 Runtime testing
If a program passed all tests in initial testing, it was then used for a longer time period to extract a large project on a virtual server with an Intel Xeon X5570 processor running at 2.93 GHz, 2 GB of ram and 200 GB hard drive. The operating system was Windows Server 2008 R2 64-bit.

This server was used to emulate a more realistic repository mining situation where a server is used with more limited hardware rather than a computer with more powerful hardware for programming. In contrast to the server, a development machine is likely to have more RAM for a computer used to compile programs. Also the fact that it is a virtual server and not a physical server make the access to hardware different than on a non-virtual system.

5.7 Results
This section contains the results and a written evaluation of how the different programs performed during the initial test phase.

5.7.1 Kenyon
Website: http://dforge.cse.ucsc.edu/projects/kenyon/
After looking for Kenyon, neither the source code or executable binaries were found. The link to the software at the University of California Santa Cruz was a dead link.

5.7.2 APFEL
Website: http://www.st.cs.uni-saarland.de/softevo/apfel/

The search for APFEL resulted in finding a webpage stating that APFEL was not supported anymore and the source code had been removed. An e-mail was sent to the person listed as contact person for APFEL (Thomas Zimmermann) who passed the information that the source was not any longer public. The reason for this was lack of time from the programmers to maintain the source code.

5.7.3 Evolizer
Website: http://www.evolizer.org/

The source of Evolizer was investigated and no libraries were found that supported Git. This meant that if Evolizer is going to be used it have to be extended to support Git. This is a possible option for future research. This was not considered a valid option at this time due to the time constraints of this project and the involved process of re-engineering Evolizer to work with Git.

5.7.4 git_mining_tools
Website: https://github.com/cabird/git_mining_tools/

This is by the bat the most promising tool found. It is written in ruby and mines Git repositories. Sadly the documentation was scarce and it is hard to understand how the tool is started. Ruby is an unfamiliar language for the tester and the code could not be reverse engineered in the timespan to clarify how to make it work. Dependency issues also made getting this tool working too time-intensive to be considered.

The fact that the tool uses a database to store the result in also causes the need for some sort of extraction tool from the local database. As a result the outcome of a full mine is not available as compilable source code without a second tool to extract the files.

5.7.5 Shrimp
Website: http://sourceforge.net/projects/chiselgroup/

When reading about shrimp it was unclear if it was a visualization library, mining library or an all in one tool. Further investigation showed that it is a visualization tool that depends on other tools. No further research on Shrimp was made.

5.7.6 Gitdm
Website: https://github.com/markmc/openstack-gitdm

Git data mining is a tool written in Python. It is a plugin to the Git client which is accessed through the logging functionality built in to Git. The tool does not mine source code but mine the Git log files. This can be a very useful tool when performing analysis of the Git log and commit actions. It is however not a tool used for mining Git repositories for source code.

5.8 Problems
When doing research of the current available tools for mining Git repositories, some major problems/shortcomings were found with every tested tool. Most of them cannot be bound to a specific program. Also the factors of some language specific drawbacks
are not considered a problem, as in this case of a program written in C#\(^1\), which will not have the drawback of being platform dependent. This is because the program, or the programmer, can work around the C# limitations.

5.8.1 Hidden tools
As a rule of thumb it was very difficult to find tools to perform repository mining of those that were mentioned in the articles read. They were most often only mentioned by name but never referred to via a web link, etc. Not even when looking for them through the department who evolved them or via Google helped. Several hours were spent trying to find particular tools. Some tools were removed from the original list of tools to be investigated because the information trail ended after reading the paper where it was used.

5.8.2 Unclear documentation
In many cases where the tool was found, there was a problem understanding how to get the program running. Tools using version control systems other than Git were investigated in the documentation. By the documentation alone, very few of them would have been able to be started. Many tools needed to have insight in both the VCS it was supposed to mine and the language in which the program was written.

At best, a readme file was included, with little to no information on how the tool should be started. Scarce information on what different functions were inside the program made it virtually impossible to make changes to it.

\(^1\) Using .NET and not Mono.
6. Conclusion of pre-study

There are many existing tools that can be used when performing repository mining. But there are few that are specialized for Git. Most current tools are created for CVS. The few tools created for Git are poorly documented and knowledge about the language they are created in is needed to use them.

Because of the timeframe in which this thesis was written not all possible tools could be tested. The one that was tested stored information of commits in a database which would require a database to be installed that the host system could access. It would also require a second program to extract the information for any measurements. With this in mind the need to create new tool for the purpose of mining Git repositories became apparent. This tool should have no need of external storage systems other than a regular file system.

There are many problems associated with repository mining. They vary from being bound to the version control system mined to how the version control system is being used. It also depends on what kind of research is being performed.

In some fields of research there is the problem of how bug reports are made and what the hidden heuristics look like [11], [13]. In others, there might be a problem that the version control system has been swapped between for example, Subversion to Git. To get a complete set of problems is virtually impossible, and it would also become depreciated information as the version control systems continue to be developed.

However, some general problems were found (see section 4.5) and how to handle these problems could be suggested. These problems should be kept in mind when starting research involving repository mining. The primary concern when performing repository mining is to remember what information from the repository is needed.
7. Mining Git with Doris

Shortcomings were found in every investigated program for mining Git repositories. These included being outdated, and a lack of documentation. This lead to the development of a new program.

The goal of this development can be divided into two parts:

- Gain knowledge of problems when creating a repository mining tool.
- See how the found shortcomings can be managed or eliminated.

After reading articles and testing existing tools, some flaws and problems were discovered. But the biggest problem being there are very few Git repository mining tools. Out of all the repository mining tools found with the limitations set up (see 5.2 Program selection) only one was found that incorporated Git. This program failed to even start because of a Ruby dependency which would not install correctly.

This made it clear that some software needed to be developed that was easy to start out of the box, supported most providers of Git repositories, and would work on as many operating systems as possible. The program was named Data Oriented Repository Information System (Doris).

7.1 Program specification

The program specifications were broken down into the following main elements.

7.1.1 Clear documentation

The documentation should be easy to understand. No knowledge of the programming language the tool is written in should be necessary to run the tool. It should also be well documented enough for other programmers to develop it further.

7.1.2 Easy to find

The program should be published under a GNU is Not Unix (GNU) General Public License (GPL) (hence forth referred to as GPL) and be available from a publicly available location. Researchers and other persons interested in repository mining should have easy access to the program. The source was placed on Github for free and public access [30]. This was an easy way to get both documentation and source code visible for as many as possible.

7.1.3 Configurable automated mining

The tool should be fully automatic and configurable, meaning the user should be able to specify the number of commits to mine. The user also needs to be able to limit the data gathered. If nothing but a .git-file or an address to such a file is provided all the commits are to be retrieved.

7.1.4 Directory structured

The mined source should be easy to browse manually and to open in an integrated development environment (IDE). The source should be contained in a directory structure with each commit clearly labeled in a directory. The structure shall be in ascending order with the initial commit first. It should also be easy to create automated analysis of the mined source code.
7.1.5 Metadata
The metadata for the repository should be stored in such a format that it is easy to extract information for particular commits.

Metadata is in this case used as a reference to information about a particular commit. It can be information such as committer, commit time, commit name etc.

7.1.6 Platform independent
The program should support at least the operating systems Windows and Linux. A plus is to be able to run the program on UNIX and Mac OS too. If this can be achieved the four largest operating systems are covered.

7.1.7 No external dependencies
The program should not depend on other software such as database engines. With no external dependencies except for a potential interpreter for the programming language, the setup and running of the program will be easier.

7.2 Implementation
The program was implemented using Java to achieve platform independence. This will result in ease of installation for researchers wanting to use this tool. Also with Java, external dependencies will be kept to a minimum. Drivers for databases can be kept internal and therefore of little concern to the user.

The JGit application programming interface (API) was chosen based on its extensive Git support and thorough documentation.

To achieve configurability, a system with flags was developed (full documentation can be found in appendix Appendix B Usage documentation of Doris).

The supported formats to retrieve .git-files are the hypertext transfer protocol (http) and Git protocol. The local file:// notation can also be used to pass a link to a .git-file. The decision to leave out secure shell (SSH) was based on the fact that extra internal functionality would be needed to be included in the program. If the only way to access a .git-file is through ssh the user has to clone a bare repository of the head revision and then pass the file:// link to that bare repository.

As a meta-data log format, extensible markup language (xml) was selected due to the ability to customize the structure and still maintain a standardized format that does not require a custom built parser.

To retrieve the commits, multithreading is used. If there are less than two cores, there will automatically be multiple threads-per-core created; otherwise the thread count will be adapted to the number of cores on the host system. Through tests it was shown that even on a single core processor multiple threads were faster than a single thread. This is most likely due to IO-wait time and the possibility to perform computations while this IO-wait occurred.

7.2.1 JGit
JGit [31] is an API for using the Git VCS through Java. It is an open source library hosted by the Eclipse foundation. JGit has few external dependencies which makes it favorable for embedding into software. Few external dependencies keep the general size of the entire software smaller.
7.3 Testing, problems encountered and solutions
During the creation of Doris there were some problems were encountered. Some of the problems found were expected thanks to the results of the pre-study, while others were discovered only through experimentation. Some problems expected due to the pre-study were not encountered, but do remain a theoretical possibility. The results of these tests can be found from section 7.3.4 to 7.3.9. Table 7.1 also gives a short comparison between the different results.

7.3.1 Knowledge of Git
As found by Kim et al. [5] detailed knowledge of the VCS to be mined is essential. The documentation of JGit [32] presupposed that a knowledge of Git terminology. Such knowledge was obtained by reading Pro Git [15]. After an understanding of Git specific commands and syntax was acquired, the development continued rapidly without any major problems. This also confirms the problem described in section 4.5.5.

7.3.2 Metadata
The problem with metadata logs was to decide what to include, as the use of metadata and its importance differs depending on what kind of research is being performed. This means that there is no “cheat sheet” that can be used to find what information is important to the general public. An educated guess had to be made for the “out-of-the-box” logging (see appendix B section Log file). Making the log creation class easy to modify was given precedence over trying to optimizing what metadata to include. Git stores every bit of information locally the metadata can be extracted through the .git-file [15]. This assures that it is not as crucial to store all details when performing the mining operation.

The actual logging is stored as xml. The advantage of this is that most programming languages have a native xml parsing library, such as XPath [33], which makes the output of doris easier to analyze through automated software.

A drawback to using xml for this is that xml takes quite a bit of memory, which is an issue with a large number of commits to process. To solve this problem either another format of storing meta-data would have to be created, or the use of another xml parser for creating the file would need to be used. Another solution for this can also be to decrease the meta-data extracted while mining a repository.

As a temporary solution, a flag to turn meta-data storage off was included for larger repositories. This would not pose any real problem. If the meta-data information is of essence all of the information can, as mentioned earlier, be extracted from the bare clone of the repository.

One other problem encountered when mining Git repositories that were not controlled for testing was that some characters were translated into invalid XML. This resulted in an exception being thrown by the XML parser used when the file was loaded into memory.

After an investigation of what special characters that are not available for XML/HTML replacements [34] an array with the ASCII character numeral representation was created and all messages was scanned for these prior to adding them as node content. During that scan all these characters was removed. Since the characters that lacked representation were special characters (such as escape and substitute etc.) this could be done without interfering with the meaning being conveyed by the message.
7.3.3 Documentation
To tell if the documentation is good or bad is very subjective. Hence a guarantee of this project leading to exemplary documentation cannot be given. Good documentation will, in my opinion, only come from a community that is involved with development of it.

As a short test, some people were asked to run Doris by just using the usage guide. The test subjects had no previous experience of repository mining but hold programming knowledge. With the help of the documentation they were able to make Doris mine repositories and also were able to find bugs.

Also the JavaDocs of Doris were provided through the Github repository through a functionality called gh-pages [35]. This was to simplify modifications that might be made by other users of Doris. This also forces the programmer to create useful comments in the source code.

7.3.4 Multiple providers
To support multiple providers can be a problem depending on how they give access to the repositories. In this paper Github was the main provider that was used. Repositories hosted at Bitbucket were also used to test with more than one provider. But to eliminate issues with any other providers the file:// protocol can be used to pass a .git-file to the program was included.

Also all work is performed on the local computer with just one connection made to an external server when making a bare clone of a repository. This was to work around any limitations in requests to the server or login information.

Another problem this solves is that there is no real need for an internet connection to mine a Git repository. This is because Git stores all information locally instead of at a centralized server.

7.3.5 Multiple version control systems
To support multiple version control systems (VCS) there is a need to understand how all of them work. To get the detailed knowledge on how logs are kept by the system in question can be a time consuming prospect. In this project almost a week was spent to get the knowledge needed to mine Git. For support for more VCS there would be a lot of time spent understanding all VCS.

7.3.6 Storage space
Repository mining can require a fair amount of hard drive space. The required amount is most often unknown before starting the mining. This can pose a problem when a larger repository is mined. It also creates the need for a repository mining tool to hold a “start point” from where it should start performing a mining sequence. This give the person performing the mining the option to move already mined material to a different place to free up disk space.

To minimize the disk usage Doris removes all the internal .git directories after a mining session is completed. This could not be done while an active mining session was taking place due to read/write collisions.

This deletion became a larger problem than anticipated since the JVM locked one particular file in the .git directory. Even after the object using the file had been nullified and manual garbage collection had been requested. To solve this problem all files had to have their properties manually changed and closed within the mining class.

To compare disk space usage between keeping internal .git log files and deleting them the repository of joda-time [36] was chosen.
Without removing internal .git files the entire mined repository required, excluding log file, 17.1 gigabytes of disk space on Windows, compared to 9.48 gigabytes of storage space used after deleting .git directories in the individual commit folder.

Since the head .git file contains all information of previous commits and the entire log, the internal ones for each commit can be deleted without any information loss. During the test with automatic cleanup, there were nine, out of approximately 791 000, files that encountered failure in deletion. In a consecutive test five files of the same repository could not be deleted. Doris informed the user of what nine files that were not deleted and Doris reported the file paths as expected.

7.3.7 Time consumption
For this test joda-time was also selected. The reason for this was that joda-time was a large enough project to take more than a few minutes to download and contains over 1000 commits. The joda-time repository was also small enough to not cause storage space concerns. This was important as the worst case scenario for time consumption is both a lot of IO work parallel to computations. This meant a fair amount of stress was put on both calculations and writing to disk and a mix between the two kinds of operations.

When all commits were downloaded using a single thread and in consecutive order the download of 1626 commits took 5 hours and 11 minutes. After this the layout was changed to using multi-threading.

Five commits were downloaded using a single thread each. The choice of five threads was arbitrary. This version of Doris downloaded 1626 commits in 2 hours and 42 minutes excluding automatic cleanup. Including the automatic cleanup it took 3 hours and 29 minutes.

7.3.8 RAM Consumption
One theoretical problem is RAM Consumption. This is due to the current library’s need to load the entire XML file into memory.

During the mining of even the largest repository used, this problem was not encountered. But it should be recognized as a potential problem. This can be prevented by changing the library used to create the XML.

This could also be a factor that slows down the mining in practice as each time a repository has to be mined it writes the log. To get rid of this factor the log creation could be postponed until all mining is complete or the log could be generated before mining begins from the initial bare clone of the repository.

As a quick fix to this problem, a flag to disable log creation was included. This is however a problem that should be further investigated. This issue appears to depend upon the ability of the hardware used to run the application.

7.3.9 Large repository test
To see what would happen if the computer runs out of disk space, the Git project’s own repository [37] was mined via a windows server. Downloading this repository took 38 hours and 42 minutes, and 3957 commits consisting of 160 gigabytes was downloaded. When the disk space was filled, Doris gave the expected error message of “out of disk space” and the commit that failed was reported, along with full SHA-1 name so the mining could be continued from the failing repository after more disk space had been made available.
It was during this test many of the bugs appeared that were hard to predict. One example of such a bug is the XML character problem.

<table>
<thead>
<tr>
<th>Type of mining</th>
<th>Repository</th>
<th>Size (GB)</th>
<th>No of Files</th>
<th>Time consumption</th>
<th>Number of commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>None deletion of .git files</td>
<td>Joda-Time</td>
<td>17.1</td>
<td>790 855</td>
<td>5h 11m</td>
<td>1626</td>
</tr>
<tr>
<td>Multithread none deletion of .git files</td>
<td>Joda-Time</td>
<td>17.1</td>
<td>790 855</td>
<td>2h 42m</td>
<td>1626</td>
</tr>
<tr>
<td>Multithread deletion of .git files</td>
<td>Joda-Time</td>
<td>9.47</td>
<td>699 806</td>
<td>3h 29m</td>
<td>1626</td>
</tr>
<tr>
<td>Provoke space shortage (single thread)</td>
<td>Git</td>
<td>160</td>
<td>n/a</td>
<td>38h 42m</td>
<td>3957</td>
</tr>
</tbody>
</table>

Table 7.1 Comparison of time consumption, number of files and required space between different sort of mining.

7.3.10 Threats to validity
The time consuming tests were only performed a limited number of times. After a test had been finished, Doris was tweaked to improve performance. The main goal of the tests was to find room for improvement, not as a benchmark of execution time. Also the fact that a virtual server was used for these tests may have had an impact on the runtime, depending on the work load of the host server at that moment.

7.4 Practical usage test
To test that Doris is actually useful, a simple analysis of software metrics was performed on repositories mined by Doris. This was done to show that the directory structure that the different commits are stored is easy to use in an automated analysis.

This metrics add-on is included in the source code of Doris and can be invoked via the --metrics flag contained in the package se.lnu.cs.doris.metrics (appendix Appendix C Source code metrics measurement). The class is not needed to run Doris itself, it was added as a flag to simplify the running of this analysis.

7.4.1 Projects used
The projects used for this experiment are:
- Facebook iOS SDK (https://github.com/facebook/facebook-ios-sdk)
- Hosebird Client (https://github.com/twitter/hbc)
- Twitter Async (https://github.com/jmathai/twitter-async)

The projects were selected based upon how many commits they had. In addition two projects from private creators and one project from a large organization were included in order to have enough commits to see changes and to contrast a large organization to a single programmer. The programming language of the project was only considered to the extent necessary to understand how comments were marked in that language.
7.4.2 Manner of execution

The measurements were made in a very simple manner. The source code file of a particular sort (e.g., .java, .c, .js) was read line by line. If the line started with the characters // it was considered a comment line. Also if a line of /* was introduced, all lines following until an */ was encountered was also considered a comment line. A line consisting of only white space was not included in the calculation. After this was done for all files with the requested type the sum of all comment lines and all source code lines were summed to the total number of lines.

The values from the initial commit were stored as a base value (chain index [38][20]) and every commit after this was compared to that value. The comparison was made by dividing the value for that repository with the base value and multiplying it by 100. Since no compensation was made for initial commits that were empty only projects with a non-empty initial commit could be included to prevent calculation errors when dividing by zero.

7.4.3 Results

The Facebook iOS SDK (Figure 7.1) had a higher change of lines of comments than lines of source code. Total lines almost mirror the comments values except the change being larger. After approximately half the time-line the comments changed a bit less. The lines of source code changed less than the total lines in the project and the comments.

![Figure 7.1 Measurement results of Facebook iOS SDK.](image)
The Hosebird client (Figure 7.2) had a large increase of lines of comments and total lines very early. This later settles down and then spikes again at the 24\textsuperscript{th} commit. Lines of source code are fairly stable until the 88\textsuperscript{th} commit where a large increase is made that is mirrored in the total lines graph.

![Figure 7.2 Measurement results of Hosebird client.](image)

With the J Pacman (Figure 7.3) project there was a larger change of the lines of comments than lines of source code and total lines. There was not as much change in this as with Facebook iOS SDK. The changes follow each other better and the change in total lines and lines of source code are almost identical, differing at the most with 10%.

![Figure 7.3 Measurement results of JPacman project.](image)
The twitter async (Figure 7.4) project showed some more interesting development. Here the dominant change in the lines of source code at the start was fairly unique. At around commit 42 there is a peak in the comment changes that fast goes back down again at commit number 44. Then between commit 55 and 88 the changes in comments and source code are almost identical. At about 90 the comment peaks and after that it is fairly stable.

![Graph showing lines of source code and comments](image)

Figure 7.4 Measurement results of twitter-async.

### 7.4.4 Conclusion

The projects showed no similarities in how the change between lines of comments and lines of source code. But only four projects were included in the study and the tool used to perform the measurements is not the best method to use. Since an analysis of these results is out of the scope for this thesis further study of this is needed to come to a valid conclusion.

However, the main purpose of the study was to show that Doris could be useful in a situation where repository mining is needed to perform the measurement. This could be done with no problem. The measurement class used could simply enter each directory with a name consisting of an integer and automatically compare them to each other. This proved that Doris can be used to perform repository mining and the result can later be used to compare different commits to gain insight of changes between them.

Also by adding another namespace to Doris it showed that Doris could easily be modified to perform an analysis automatically paired with the mining.
8. General discussion
In this section a general discussion of different problems that were found during the work of this thesis and the pre-study will be held. It will focus on a few larger points found.

8.1 Hard to find previous results and software
Every mining tool discussed in this thesis was sourced from an academic study. The tools themselves, however, were in several cases unlocatable, or at least made to seem that way, and were therefore unavailable for close inspection or study.

This is a violation of the scientific principals of access, reproducibility, and testability. Were a chemistry paper to not include the specific chemicals used, it would not be fit for publishing on those grounds. The current system of including an external reference to the current-as-of-publication URL at which the tool can be found is insufficient to the purpose is supposes to address: university accounts get closed, web services go out of existence, and companies completely re-arrange their sites with no regard for their own documentation, much less third party references.

There is also the issue of who bears the responsibility for keeping these tools available, the researcher or the institution which funded the research. An individual researcher or research team may not have the resources to keep a perpetual archive available. A corporate entity may not have the institutional memory to know why something should be kept, or may simply decide not to keep something available in the service of the bottom line.

Additionally, there is a strong tendency for the person who writes code to believe continual maintenance and support is necessary if said code is to remain public. In at least one case, the original author of a tool referenced in this study was contacted in order to get the code for the tool, and said author refused on the grounds that the code was not currently being maintained and that the author did not have the time to devote to maintaining and supporting the tool. While this is a completely valid reasoning for withholding production-oriented code, it clearly shows a lack of consideration for academic usage: a modified version of the code used in the paper can very well distort the results obtained, which makes reproduction of the original experiment impossible.

As we have not yet established stable publicly-accessible long-term storage for the digital resources of academic projects, the best option found during the writing of this thesis was to use a public source storage provider such as Sourceforge, Google code, or Bitbucket. The source code of the tool constructed for this paper is now available via Github. Any change in hosting will maintain availability of the actual downloadable source code. I have also included the link to the current repository in the list of sources (source [30]) of this thesis so that it can be easily found. If more researchers and the organizations for which they work would do something similar, it would be much easier to continue based upon previous findings, and scientific integrity can be maintained.

8.2 Selecting a tool
As with most things, different repository mining tools have different strong suits. It can be wasteful and disruptive to find out that a program does not fit your needs during the course of an active project. To avoid this, a thorough comparison between tools should be made beforehand. In this research, no academically conducted comparisons were found between different repository mining tools which handled Git. This is somewhat
surprising, as Git is now a widely used version control system. Git is used by the Linux foundation, Facebook, WordPress, and jQuery. These are fairly large organizations where research that requires repository mining could be done to great benefit.

Knowing the goal of your repository mining is paramount when choosing a mining tool. If the information needed can be gathered from the repository-provided metadata, downloading the complete source code archive would be wasteful; if the needed information can only be gotten from the source code, a system which relies on metadata, log messages, and difference information will require more processing to gather the needed data.

8.3 Ethical issues
When performing analysis of material mined from repositories, there are ethical issues which need to be confronted. How much information about each developer’s work should be included in the report?

“Would the developers of an open source project consider their software trails open too? What are the implications of publishing aggregated data about a project? For example, would it be ethical to claim (in a research paper for example) that code from a certain developer tends to have more defects than any other developer’s code in the same project?” —Daniel M. German [25]

In some repositories, only certain persons are allowed access to commit changes, and those persons function as gatekeepers to code inclusion; all commits would be attributed to this group, and the actual authorship would not be shown simply by examining the repository-provided data. Such authorship obfuscation can also take place in a pair-programming environment. Relying strictly on given metadata can result in credit not being given where it is due. When made public, such misattribution can have repercussions in the long term, which are outside the scope of this thesis.

8.4 Git is growing
Github alone hosts over 6,100,000 projects [26]. Besides Github, other providers (e.g., Bitbucket [39], Google code [40]) provide Git as version control system for open and/or closed source projects. However, Git seems to be under-represented as an object of study, particularly in the area of repository mining. In the course of research for this thesis, only one paper was found that dealt exclusively with Git. A majority of the papers studied made use of CVS as their primary version control system. Though a great deal of the research was completed prior to the release of Git, this does not mitigate the lack of focus upon it since.

Considering that many large groups and organizations use Git as their main version control system, I believe that there would be a great benefit in trying to shift more energy toward investigating Git and how it can be used.
9. Conclusion
The conclusion of the thesis and a short summary discussion of the conclusions.

9.1 Common problems encountered with repository mining
There are many problems associated with repository mining. They vary from being bound to the version control system mined to how the version control system is being used. It also depends on what kind of research is being performed.

In some fields of research there is the problem of how bug reports are made and what the hidden heuristics looks like [13], [14] and others where there might be a problem that the version control system have been swapped from e.g., Subversion to Git. So to get a complete set of problems is virtually impossible and it would also become depreciated information as the version control systems are being further developed.

However, some general problems were found (see section 4.5) and solutions how to handle these problems could be suggested. These problems should be kept in mind when starting research involving repository mining. But the main thing to keep in mind when performing repository mining is to remember what the important information needed is.

The problem that was most prominently confirmed was that there is a need of detailed knowledge of how the version control system works. There is a great need of understanding the terminology and how the thought behind the VCS is. Also the goal of the mining tool needs to be clear. It is hard to cover all aspects in a single tool and there is a risk of the tool becoming bloated. Software metric analytical tools or libraries should also be created by programmers that have a high degree of knowledge of it. If this is later used within the repository mining tool or ran afterwards is irrelevant.

For the purpose of this thesis the problem was that there are few repository mining tools aimed towards Git. Since this realization came fairly early in the process the possibility of creating such a tool with, basic functionality, was feasible within the time frame. The tool created was called Doris (Data Oriented Repository Information System).

Creating this program confirmed some general problems and also opened up a new field of problems concerning hardware limitations and time consumption. One example of this is the RAM usage problem with Doris (section 7.3.8).

9.2 Profits of mining Git repositories
The study performed in section 4 concluded that Git repositories are not discussed much in academic works. In the course of this thesis only a few papers were found that dealt with Git. Out of them only one was focused on repository mining and using Git.

This came as a surprise to me since there are many open source projects maintained through Git and hosted publicly via Github, Bitbucket, Google code or similar services. These projects would be able to contribute to research regarding software quality and metrics research. This is a tremendous source of information for researchers.

Also since Git is decentralized there is only a snapshot of the current state needed to be able to reproduce experiments without keeping track of what commits were used and when; all that is needed to use is the same .git file.
9.3 Successful data extraction for software quality analysis

The main purpose for this thesis was to see how extraction of data for software quality analysis could be performed. To see if Doris is successful with this would require an entire thesis on its own. But with the help of a basic practical usage test (section 7.4) I believe it can be used.

When performing repository mining for the purpose of software quality analysis, there are some things that are more important than others. In this case the possibility of comparing different commits to each other and gaining access to full source code (e.g. compiling and running each commit) were two high priorities. In that case a mining tool which would store information in a directory structure is preferred. Metadata is also of little interest except for potential commit messages to be able to weed out potential back-up commits.

The analysis should be able to be either hooked onto the mining tool or be run as a batch job when the mining is done. This requires that the mining tool is either verbose or easy to modify. Preferably it should be both. That the mined commits be logically structured is also a requirement. If there is metadata, this should be easily connected to a certain commit.

Metadata should be stored in a format that can be easily represented through automated software. This representation should also have a clear connection to the source code it belongs to. This rules out elaborate models and promotes solutions such as json or xml. In these formats objects can be represented without any external dependencies, because there is a standardized notation and most programming languages have libraries to parse them. If the metadata log file is created correctly then an automated analysis where certain commits are ignored can be performed. These commits might have a commit message with the line “back-up” in them. It can also be reversed and commits tagged with “refactoring” might trigger a comparison between that and the prior commit.

I believe that with the extensive documentation and verbosity of Doris makes it easy to modify to add analysis tools both through source code and through scripts that read output. The general idea of the log file makes it easy to connect metadata to a particular commit by using a program.

9.4 Best practice

There is a lack of best practices and standards in the field of repository and data mining, and how to share results and tools created [12], [5], [6], [14]. Either researchers do not see the importance of their program for their paper, or they focus too much on maintaining the source code when making them publicly available. But considering reproducibility, it would be better to have the software public and make it clear it is no longer maintained instead of removing it. This would also make it easier for future researchers to be able to benefit from experiences gathered while creating the software.

When creating software for research purposes, commenting and documenting the code is important for others who might want to use or learn from the software. This can be achieved with the help of JavaDocs or similar comment systems. This information should then be made public through the publication that uses the software. There are multiple free providers for source management that can be used for this and then a link to that software can be added somewhere in the publication.
9.5 Hypotheses

To answer the first hypothesis “An existing repository mining tool exists that can extract data from Git repositories.” The conclusion would be “yes”. But there are many problems found with this yes. First off there are several problems in documentation and to simply get the tools that were found to start without having a detailed knowledge about both the tool and the language that it is created in. They also have a tendency to be very purpose specific. This makes it prone to have new tools created for each study.

The second hypothesis “Repository mining can be conducted decentralized repositories in the same way as centralized repositories” also gave a positive outcome. With the help of Doris decentralized repositories was mined in a similar fashion as centralized repositories. The main difference was that all mining can be performed at the local computer. This also eliminates the need for a connection to the central server when performing a repository mining.

One benefit of this is that researchers with a poor network connection can mine decentralized repositories as efficient as researchers with a better network connection. In the end this can open up for research in Computer Science that requires repository mining to universities with the lack of a stable internet connection by having large repositories sent to them on a Universal Serial Bus (USB) flash drive by someone with a more stable internet connection.

9.6 Future work

A full comparison between different repository mining tools should be made. In this comparison a more thorough investigation of how the tools work, what their drawbacks and strong sides are.

A measurement of to what extent different version control systems are used. At the moment much of the research seems to be made on CVS and little is done on the newer version control systems. Also, a more generic way of performing repository mining could be useful.

An analysis should be conducted on the use of languages in smaller open source projects which are not funded by large organizations. This might show that there are more efficient ways of solving algorithm problems that go missed outside academic work and larger organizations.
Sources


Appendix A. Articles included in literature study
Articles are listed in chronological ascending order.

D. M. German, “Mining CVS repositories, the softChange experience” (2004)


I. Hammouda, K. Koskimies, “Concern-Based Mining of Heterogeneous Software Repositories” (2006)

L. Voinea, A. Telea, “Mining Software Repositories with CVSgrab” (2006)


C. Kiefer, A. Bernstein, J. Tappolet,”Mining Software Repositories with iSPARQL and a Software Ontology” (2007)

A. Bachmann, A. Bernstein,”Software Process Data Quality and Characteristics – A historical View on Open and Closed Source Projects” (2009)


Appendix B. Usage documentation of Doris

Doris

Table of contents

- License
- About
- Dependencies
- Usage guide
  - Help
  - URI
  - Target
  - Start point
  - End point
  - Limit
  - No log
  - Metrics
  - Important
- Log file

License
Doris is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. Doris is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with Doris. If not, see <http://www.gnu.org/licenses/>.

About
Doris was created by Emil Carlsson as part of a bachelor thesis about problems encountered when mining software repositories. The main goal of the thesis was to find a mining tool that could handle git, work with as few dependencies as possible, and also provide automated reproducible extraction and measurement pipeline.

Dependencies
Doris is written in Java and requires Java (JRE 1.7 or newer) to be installed on the computer running it.

Usage guide
When using parameters and not specifying target directory, Doris will automatically create a directory with the same name as the .git file used for mining. If no parameters are passed to Doris, Doris will prompt for URI to .git file and the target to store the
results from the mining. All flags are to be appended after the command to initialize
Doris. When using flags the URI flag must be included as a minimum.
Run Doris on *nix:

emil@linux-computer ~ $ ./dors.jar

Run Doris on Windows:

C:\> java -jar c:\path o\doris.jar

Help

-h, --help [flag]
Shows help information. If a flag is appended it will show help information of that
particular flag.

URI

-u, --uri <link to .git-file>
Specifies the URI where the .git file can be found. The protocols that Doris can handle
is http(s)://, git:// and file://. Example of
formatting: git://github.com/GingerSwede/doris.git.

Target

-t, --target <path to target directory>
Specifies the target where the different commits should be stored. When omitted Doris
will use the current working directory and set up a folder named after the .git-file used
in the URI.

Start point

-s, --startpoint <commit sha-1>
Set a starting point for Doris to start mining the repository from. Full sha-1 is needed. If
the sha-1 value is incorrect the mining will never be started.

End point

-e, --endpoint <commit sha-1>
Set a commit where Doris should stop mining. Full sha-1 is needed. If the sha-1 value
is incorrect the mining will not stop. The given sha-1 commit will not be included in
the mining results.

Limit

-l, --limit <max number of commits>
Set a maximum number of commits Doris should mine. Amount is to be given as an
integer (e.g., 6, 10, and 600).

No log

-n, --nolog
When this flag is passed the logging option in Doris is turned off. This is recommended
when mining larger repositories that will generate many commits. All information that
is logged by Doris can manually be obtained through the .git-file copied to local access. It can be found in the same directory as the mining results.

**Metrics**

`-m, --metrics <file ending[,file ending[,file ending]]>`

Creates a simple software metrics analysis where the amount of source code is compared with the amount of comments in percent. Multiple file endings separated with a comma and no spaces.

**Important**

If the `-e` and the `-l` flag is used in combination Doris will end on the flags criteria that is reached first.

**Log file**

Unless the `-n` flag is used Doris will automatically log basic information about the different commits in an xml-file. The log contains information about parent commit, author, committer, commit message and commit time (given in UNIX time). Example:

```xml
<project project_name="ExampleRepository">
  <commit commit_name="08046e7b57f772f270619601d1a9420f76320066" commit_number="0" commit_time="1358168496">
    <author e_mail="john.doe@example.com" name="John Doe"/>
    <committer e_mail="john.doe@example.com" name="John Doe"/>
    <commit_message>
      Initial commit
    </commit_message>
  </commit>
</project>
```
Appendix C. Source code metrics measurement

package com.gingerswede.source.metrics;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import se.lnu.cs.doris.global.Utilities;

/**
 * This file is a part of Doris
 * Doris is free software: you can redistribute it and/or modify it under
 * the
 * terms of the GNU General Public License as published by the Free
 * Software
 * Foundation, either version 3 of the License, or (at your option) any
 * later
 * version. Doris is distributed in the hope that it will be useful, but
 * WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 * You should have received a copy of the GNU General Public License along
 * with
 * Doris. If not, see <http://www.gnu.org/licenses/>.
 * @author Emil Carlsson
 */

public class SLOC {

    private File m_mainDir;
    private final String m_avoid = ".git";
    private int m_baseValueTotal = -1;
    private int m_baseValueCode = -1;
    private int m_baseValueComments = -1;
    private String m_projectName;
    private String[] m_fileEndings;

    public SLOC(String path, String[] fileEndings, String projectName) {
        this(new File(path), fileEndings, projectName);
    }

    public SLOC(File dir, String[] fileEndings, String projectName) {
        this.m_mainDir = dir;
        this.m_projectName = projectName;
        this.m_fileEndings = fileEndings;
    }
}
public void generateCSV() throws Exception {
    if (this.m_mainDir == null) {
        throw new Exception("Base directory not set.");
    }

    File csvFile = new File(this.m_mainDir, this.m_projectName + ".csv");
    for (File f : this.m_mainDir.listFiles()) {
        if (f.isDirectory() && !f.getName().contains(this.m_avoid)) {
            int commitNumber = Utilities.parseInt(f.getName());
            int slocd = 0;
            int slocmt = 0;
            int sloct = 0;

            for (File sd : f.listFiles()) {
                if (!sd.getName().toLowerCase().contains(this.m_avoid)) {
                    slocd += this.countLines(sd, false);
                    slocmt += this.countLines(sd, true);
                    sloct += slocd + slocmt;
                }
            }

            if (this.m_baseValueTotal < 0) {
                this.m_baseValueTotal = sloct;
                this.m_baseValueComments = slocmt;
                this.m_baseValueCode = slocd;
                sloct = 100;
                slocmt = 100;
                slocd = 100;
            } else {
                sloct = (int) ((double) sloct
                        / (double) this.m_baseValueTotal * 100);
                slocmt = (int) ((double) slocmt
                        / (double) this.m_baseValueComments * 100);
                slocd = (int) ((double) slocd
                        / (double) this.m_baseValueCode * 100);
            }

            String appendString = String.format("%d;%d;%s;%s;%s\n",
                    commitNumber, sloct, slocd, slocmt);
            this.appendString(appendString, csvFile);
        }
    }
}

private void appendString(String appendString, File csvFile) {
    if (!csvFile.exists()) {
        this.createCSVFile(csvFile);
    }
}

Writer writer = null;
try {


writer = new BufferedWriter(new FileWriter(csvFile.getAbsolutePath(), true));
    writer.append(appendString);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} finally {
    try {
        writer.close();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } // TODO: Find decent api for getting sloc.
}

private void createCSVFile(File csvFile) {
    Writer writer = null;
    try {
        csvFile.createNewFile();
        writer = new BufferedWriter(new FileWriter(csvFile.getAbsolutePath(), true));
        writer.append("Commit number;Base value;Total lines;Lines of source code;Lines of comments\n");
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            writer.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    int sloc = 0;
    if (file.isDirectory()) {
        for (File f : file.listFiles()) {
            sloc += this.countLines(f, countComments);
        }
    } else {
        Boolean readFile = true;
        if (this.m_fileEndings == null) {
            readFile = true;
        } else {
            for (String s : this.m_fileEndings) {
                if (file.getName().endsWith(s)) {
                    readFile = true;
                    break;
                } else {
                    readFile = true;
                }
            }
        }
    }
    // TODO Auto-generated catch block
    e.printStackTrace();
}
readFile = false;
}
}
if (readFile) {
    BufferedReader br = new BufferedReader(new FileReader(file));
    Boolean isEOF = true;
    Boolean isComment;
    Boolean isBlankLine;
    Boolean inMultiLineComment = false;
    Boolean prevMultiLineComment = inMultiLineComment;
    do {
        String t = br.readLine();
        if (t != null) {
            isComment = this.lineIsComment(t);
            isBlankLine = t.trim().equals(""');
            prevMultiLineComment = inMultiLineComment;
            inMultiLineComment = this.resolveMultiLineComment(t,
                  inMultiLineComment);
            isEOF = false;
            if (!isBlankLine
                 && (!countComments ? !isComment
                                  && !prevMultiLineComment : isComment
                                  && prevMultiLineComment)) {
                sloc++;
            } else {
                isEOF = true;
            }
        } while (!isEOF);
    } while (!isEOF);
    br.close();
}
return sloc;
}
private Boolean lineIsComment(String line) {
    return (line.trim().startsWith("//") || (line.trim().startsWith("/*")
    && line
        .contains("*/"));
}
private Boolean resolveMultiLineComment(String line, Boolean
    inCommentBlock) {
    return (line.trim().startsWith("/*") || !line.contains("*/"));
}