Master Thesis

January 28, 2014

MiningHub

A Social Mining and Data Sharing Platform

Silvan Troxler

of Hildisrieden, Switzerland (08-708-646)

supervised by

Prof. Dr. Harald C. Gall

Martin Brandtner

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

MiningHub

A Social Mining and Data Sharing Platform

Silvan Troxler

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Silvan Troxler, silvan.troxler@uzh.ch

Project period: August 15, 2013 - January 30, 2014

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank Prof. Dr. Harald C. Gall for giving me the opportunity to write a thesis in the software evolution and architecture lab at the University of Zurich and Martin Brandtner for his valuable feedback.

Abstract

The popularity of social coding websites has led to an increasing number of publicly available software repositories over the last years. Researchers extract data sets from such repositories to conduct studies based on them for finding novel insights in the software engineering field. Mining software repositories is often time-consuming and error-prone. In spite of that, published papers usually do not make the used data sets available, which complicates evaluation and replication of those studies. We developed MiningHub, a web-based and extensible mining platform to overcome these limitations. Researchers can add support for arbitrary repositories to MiningHub and use the platform to obtain data from these. In addition, we provided plugins for the social coding website GitHub, bug tracking system JIRA, and discussion board Stack Overflow. These plugins and mined data sets can also be shared among others. The results of the conducted evaluation indicate that by using MiningHub, the time needed to mine software project data from several types of repositories can be reduced for selected studies.

Zusammenfassung

Die Popularität von sozialen Programmier-Webseiten hat in den letzten Jahren dazu geführt, dass eine immer grösser werdende Anzahl an Software-Repositories öffentlich zugänglich ist. Forscher extrahieren Daten aus diesen Repositories und führen damit Studien durch, welche häufig neue Erkenntnisse im Bereich der Software-Entwicklung liefern. Das Extrahieren von Daten aus Software-Repositories ist allerdings oft aufwändig und fehleranfällig. Und trotzdem veröffentlichen publizierte Studien nur selten die verwendeten Daten, was sowohl Evaluation als auch Replizierung dieser Studien erschwert. Um diese Probleme zu umgehen haben wir die webbasierte und erweiterbare MiningHub-Plattform entwickelt. Forscher haben die Möglichkeit, MiningHub um die Unterstützung von beliebigen Repositories zu erweitern und anschliessend Daten von diesen zu beziehen. Wir haben bereits Plugins für die soziale Programmier-Plattform GitHub, das Fehlerverwaltungssystem JIRA sowie die Diskussionsforen von Stack Overflow hinzugefügt. Diese Plugins und abgeholte Datensätze können jeweils auch mit anderen geteilt werden. Die durchgeführte Evaluation zeigt, dass durch die Verwendung von MiningHub die benötigte Zeit zur Extrahierung von Daten aus Software-Repositories für einige Studien reduziert werden kann.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

The advent of social coding websites like GitHub[1] has led to an increasing number of publicly available software repositories. These constitute large amounts of data in the software engineering field and are gold mines for researchers. As a result, many recent studies use data from source code version control, issue trackers, discussion board, or other software project repositories. These data sets have to be extracted from the corresponding repository prior to any use. The process of such a data extraction is called *software repository mining* and there is even a dedicated conference named *Working Conference on Mining Software Repositories* (MSR)[2]. Studies published in the context of that conference use software repository data for purposes like bug prediction, analyzing code ownership, or investigating effects of social interactions. Mining software repositories may thus yield novel and useful information about productivity, quality, maintainability, and other aspects in software development. As such, many researchers in the MSR field mine software repositories and conduct studies based thereof. The mining step oftentimes takes a large part of the time used for the whole study, though. This is caused by the non-trivial extraction process of data from external repositories (e.g. [GS12]). It is tedious and error-prone as there are many aspects to keep in mind when mining software repositories. Furthermore, when publishing a study paper in the field, the used data sets are usually not made available to the public (e.g. [Rob10]). The missing data often complicates replication and evaluation of studies although these are essential properties of empirical research [RGBICH11]. It also causes repetitive work in the research communities to create mining tools as well as extract the same data many times. These are often stated problems in the mining software repositories field (e.g. [HNB+13]).

1.2 Goal

In this thesis, we try to solve two limitations in the mining software repositories field. We want to simplify the mining process as well as make retrieved data sets available for reuse and replication. Therefore, we present our approach to mining software repositories called MiningHub. MiningHub is a web-based and extensible platform which provides the basic features that are necessary to retrieve data from software repositories. By using mining scripts, which are text files containing information about repositories, researchers can extend the functionality of MiningHub and add support for further data sources. In addition, all data that is mined by MiningHub can be shared among other researchers to achieve replication possibilities. Based on the MiningHub

[1] `https://github.com`
[2] `http://msrconf.org`

platform, we try to answer the following research question: **How can a social mining platform be an enabler for data analysis across multiple software repositories**? For this, we conduct an evaluation of our approach by means of a study replication involving three different types of software repositories.

This thesis is structured as follows. After the introduction, we go through related projects and studies. Then in Chapter 3, we describe the current situation in the mining software repository field and present the proposed solution. In Chapter 4 we describe the approach and implementation we take with MiningHub as well as mining scripts. Chapter 5 evaluates our mining platform based on a replication study and answers the research question. The conclusion of our approach and future work can be found in Chapter 6.

Chapter 2

Related Work

In this chapter we present related work to the MiningHub project. The first section shows platforms that have the same or a similar goal as MiningHub. That section also includes some tools used for mining software repositories. In addition, we present recommendations from researchers to avoid common pitfalls when mining repositories. The last section explains different solutions of unified software project data schemas for data exchange and reuse.

2.1 Data Collection

The idea of developing an application that collects software project data from various sources and then allows to publish it for open use is not new. There are even applications that directly analyze the data as a service. For example, Boa by Dyer et al. [DNRN13] is a language and infrastructure for analyzing large-scale software repositories. They developed a domain-specific language (DSL) that enables researchers to mine and directly analyze software repositories with only some lines of code. The data sets are automatically retrieved from SourceForge and cached as monthly snapshots. They also made use of massive parallelism that allows Boa to calculate the churn rates (i.e. the average number of changed files per revision) of 620,000 projects in only 62 seconds using six lines of code. A Java program with the same task running serially would take about 174 *minutes*, which is 168 times slower [DNRN13]. While their approach is very useful for large-scale analyses, it is not easily extendable by researchers themselves; for example, they cannot just add a new kind of repository. A similar application to Boa is SOFAS (SOFtware Analysis Services) as described by Ghezzi and Gall [GG11]. SOFAS provides a web service to create analyses and retrieve their results. It employs a catalog of software analysis processes (e.g. metrics calculation or change coupling) of which a researcher can pick from and combine several processes to a workflow. Using version history services (CVS, SVN, and Git) and issue tracking history services (Bugzilla, Google Code, Trac, and SourceForge), SOFAS retrieves the needed data and executes the workflow based on it. As soon as a workflow completes, the results of the analyses are made available to the researcher. Another similar example is MetricMiner as described by Sokol et al. [SAG13]. It is a web application which allows researchers to calculate metrics, extract data, or compute statistical inference. While researchers can write code to calculate their own metrics, MetricMiner already calculates some metrics on its own as soon as a new project is added. And so far, all Apache open source projects were added to their index. Based on these metrics, researchers then can formulate SQL queries and download the result directly from the MetricMiner webpage. While this approach is uncomplicated for gaining metrics or other statistics, it seems limited in other areas where, for example, more complex calculations have to be done or data from other sources than the code itself is needed. Also for extracting code metrics in software projects is Sourcerer as described by Ossher et al. [OBL+09] and Linstead et al. [LBN+08]. After

downloading arbitrary repositories, Sourcerer statically analyzes the code and also tries to find the topics embedded in the code. Another goal is to calculate developer similarity and source file similarity as well as cross-linking between different repositories. While Sourcerer performs a lot of different analysis on the code, the same limitations as to MetricMiner apply. Another example of a software project analysis application is Ohloh[1] which aims to be a social network for software developers and projects while exposing several kinds of statistics about them. The statistics that Ohloh calculates are not numerous, but they have an extensive catalogue of software projects that can be requested by using an API.

As a more low-level approach, Gousios and Spinellis [GS12] and Gousios [Gou13] presented GHTorrent[2], a system to obtain and mirror project data from GitHub repositories. They created a model of entities and belonging attributes used on software projects, such as *User*, *Repository*, *Commit*, and *name*, *url* (uniform resource locator), *date*, to appropriately store this information. Their approach delivers a distributed data source but currently, they only support data from GitHub. Similarly, the GitHub Archive[3] by Ilya Grigorik aims to act as a mirror for the event data (e.g. *CreateEvent*, or *ForkEvent*) that is created day after day on GitHub. It does so by regularly requesting the public GitHub event timeline and storing the event data for every single hour. It then allows to download dumps of the events which were tracked and exposed by GitHub. The limitation of this approach is that the data is not linked between different data sets, it is merely a list of events. And the GitHub Archive is—as the name implies—only available for GitHub. A further data platform was presented by Howison et al. [HCC06] with their FLOSSmole project which is a collaborative data collection over several different software project services and other data sources. They also host donated data sets and scripts for data retrieval from further research teams. Their data is separated over many thousand, mostly unrelated files but not very well structured, though. Merobase by Janjic et al. [JHSA13] is yet another data repository. It crawls SourceForge, JavaForge as well as the web to index code repositories using CVS or SVN. The indexed code can then be searched by using their code search engine. Additionally, they offer a heavy download of all the stored data. Based on this data set, the authors also showed that producing statistics (e.g. find the largest code file or the types of open source licenses used) on a large amount of projects can be done relatively easily.

In contrast to the mostly hosted platforms described before, there are other tools that have to be downloaded by their users and have to be run on their own computers where all data is stored locally. An example of this is Mozkito[4] that consists of a collection of modules to mine software repositories. These modules include *versions*, *issues*, *persons*, as well as a *persistence* module to store data in a database. Combined they build a framework to retrieve data by using a command line tool; only some arguments (e.g. the repository URL) and a configuration file are required. Mozkito supports many different repositories including bug trackers from Bugzilla, JIRA, Mantis, GoogleCode, and version control from Git, Mercurial, Subversion. Another example is MetricsGrimoire[5] which consists of several different tools to retrieve data from code version control systems, issue trackers, or mailing lists. While the manual setup on one's own computer might be a hurdle, there is FLOSSMetrics, as described by Herraiz et al. [HICRH+09], that uses the tools from MetricsGrimoire to get the data of various projects and stores them online for researchers to download. However, it appears that FLOSSMetrics has not been updated anymore since 2009. Similar to MetricsGrimoire, Alitheia Core by Gousios and Spinellis [GS09] is another example of a local data collection tool. It has to be downloaded and repository data can be added so that metrics can be calculated. To extend the already included metrics, the researcher has the

[1] `https://www.ohloh.net`

[2] In [Gou13], Gousious consequently wrote about *GHTorent* but we believe the official name of that tool is *GHTorrent* as [GS12] and the corresponding web address suggest.

[3] `http://www.githubarchive.org`

[4] `https://mozkito.org`

[5] `http://metricsgrimoire.github.io`

possibility to add his own metric calculations. In addition to the presented applications, there is a range of further tools to support mining software repositories. Some of these tools were included in the comparative analysis by Olatunji et al. [OIAGAG10]. The authors compared six different applications that enable researchers to simplify the data retrieval step. Thereby, they found that the tools are mostly focused on one type of repository only. Another limitation of some of those was that they are language dependent. Bosu [Bos12] presented a research plan to find community structures for a certain project. To retrieve data he proposed different tools for different type of repositories. CVSAnalY to mine version control, Bicho for retrieving issues from bug trackers, and Mailing List Stats to obtain discussions from mailing lists. There are also many more tools available.

2.2 Repository Mining Recommendations

As mining software repositories is a non-trivial task, there are many papers giving recommendations on what to keep in mind to avoid common problems. These often apply regardless of whether using an existing tool or developing a new one. Bird et al. [BRB+09] conducted a study on the promises and perils when mining the distributed Git source code management system (DSCM). They showed that compared to other SCMs, Git has differing semantics of *commit* and *branch*. This is caused by Git being a distributed SCM where a commit not necessarily means that all developers may directly see the change. Furthermore, commits may be reordered, deleted, or edited. As a result, mining Git is different from mining other SCMs as concepts differ in some aspects. Robles et al. in [RGBICH09] and [RGBICH11] also gave advice for potential problems that can occur when retrieving software repository data. They studied source code management systems, mailing lists, and bug tracking systems to find pitfalls. For each of these types of repositories, they also presented several tools that are capable of mining these. Furthermore, they mentioned that in version control systems, it is important to separate the developers of code and the actual committers. They also described challenges when merging data from different repositories. In [RGBICH11], the authors also emphasized how important it is to reuse data sets to allow validation and replication of studies as these are essential properties in empirical research. With their MSR Cookbook, Hemmati et al. [HNB+13] created a survey on all the MSR proceedings between 2004 and 2012 to come up with a list of recommendations. These recommendations were then clustered in four categories of which *Data Acquisition and Preparation* and *Sharing and Replication* are of our interest in the context of this thesis. For example, the authors found as well that when mining bug tracking systems it is important to distinguish between committers and submitters of a patch. Another recommendation was to only consider issues whose resolution is either *fixed* or *closed* to avoid including issues which are work in progress or duplicates. On the other hand, researchers should explicitly include such issues when they are trying to, for example, find duplicate issues. Regarding sharing and replication, the authors noted that most MSR research consisted of studies applying certain techniques on certain data sets. However, a single study is not able to provide convincing evidence regarding their findings. As a result, many researchers recommend making replicating studies more common in the MSR field. In their study, Hemmati et al. even suggested making a website to collect and make available the used data and tools of each research paper.

2.3 Data Exchange

Making the used data sets available provides a way to replicate studies. Establishing common file formats to store the different aspects of software development would help even further. Kim et

al. [KZK+06] presented TA-RE, their exchange language for mining software repositories. They defined a folder structure where a transaction (i.e. a commit) is a directory and contains XML files for information about the transaction, the changed content, and the original file content. To specify this information, they defined XML schemas for the different file types. A limitation of their approach is that they only supported file revision control systems and no other types of software repositories. A more recent paper was presented by Würsch et al. [WGH+12]. Their solution, called SEON (Software Evolution ONtology), aimed to establish a shared taxonomy of software evolution. Therefore, they layered different concepts into several levels of abstraction. At the bottom of the so-called SEON pyramid, there are system specific concepts such as Bugzilla, Java, and Git. On the next higher level, they put domain specific concepts like issue tracking, source code, and version history. On top of that, they defined domain spanning concepts that specify code analyzations (e.g. code clones), and general domain-independent concepts of the software project (e.g. activity, stakeholder, or file). Each of these layers were modeled by OWL classes so that each item has got a fixed schema. And as ontologies are meant to be linked together, these items also have properties like `dependsOn` that can be used for relations. That approach allows to build domain-independent tools to support many different facets of the software development domain as explained by the authors. The data sets ideally can be imported in and exported from different tools which allows sharing of mined software data. SEON is also used in SOFAS by Ghezzi and Gall [GG11]. Furthermore, with WEON (SoftWare Ecosystem ONtology) by Gutierrez and Robbes [GR13], there is also a planned extension of SEON. The authors of WEON described their roadmap to specify software project ecosystems in addition to just the project itself. That study also shows that the approach of modeling software development artifacts in semantic web technologies enables extension of existing taxonomies.

Chapter 3

Preliminaries

This chapter starts with describing the current situation and problems in the mining software repositories field. After that, we present a review of the papers published in MSR 2013 and the respective data sources. Then we describe the proposed solution with which we aim to overcome the limitations of repository mining and also conduct a technology evaluation.

3.1 Background

Due to the advent of open source development on social coding platforms (e.g. GitHub), the number of publicly available software repositories is steadily increasing. These repositories include source code version control, issue tracking systems, and other platforms that are often used in software development. All of these constitute large amounts of data in the software engineering domain, and the project repositories are growing every day. Those data sets are gold mines for research. As a result, many recent studies are based on data obtained from software repositories. Before researchers can work on data they want to analyze, they usually have to retrieve it from a corresponding platform. The process of such a data retrieval is called *software repository mining*. And the interest in that domain has even led to a dedicated conference called *Working Conference on Mining Software Repositories* (MSR). In that field, there are many interesting studies that use mined software data. For example, Naguib et al. [NNBH13] proposed an approach to automatically recommend an assignee to a new bug report. They based their studies on bug report data as well as changes thereof (e.g. when was the resolution of an issue changed to *fixed*) from three different software projects. Another example is finding difficult aspects of programming as presented by Allamanis and Sutton [AS13b]. They analyzed questions and answers on the help forum Stack Overflow[1]. Based on these data sets, they then extracted insights about what aspects of programming languages are most difficult to understand for its users. Such studies may uncover findings in the field and might eventually bring their results into the industry. There, research findings can help save time and money, and make software better.

Mining software repositories is the first step of many studies that rely on data in software engineering research. Obtaining data from external repositories is a non-trivial task, though (e.g. [GS12, RGBICH11, KZK+06]). Interfaces from different platforms have to be requested and the data then needs to be extracted from the response, it also has to be properly stored. The whole process also requires error handling to avoid lost data or other problems. As a result, mining software repositories is an error-prone and time-consuming step of many research studies. To avoid common problems and pitfalls, there are studies that give recommendations on how to properly mine software repositories (e.g. [HNB+13, RGBICH11, RGBICH09, BRB+09]). Despite this less

[1] `http://stackoverflow.com`

than ideal situation, there are many research groups that use their own tools to retrieve data. This leads to repetitive work as other teams might already have created a similar tool or retrieved the same data. To overcome these limitations, there is a broad variety of tool sets available that can be used to obtain software data. As an example, Olatunji et al. [OIAGAG10] conducted a comparative analysis of various tools used to mine software repositories. Most of these tools can only be used for a single type of repository or are even programming language dependent, though. Using different applications also requires to get used to all of them and handle different data formats for the integration. Another problem in the MSR field is that the replication of studies is only rarely directly possible as the used data sets are usually not published [Rob10]. However, a single study is not able to provide convincing evidence regarding their findings, and verification by replicating the studies is required as stated by Hemmati et al. [HNB+13]. As a result, Hemmati et al. suggested to create a website to collect and make available the data sets used in the papers. Conclusively, it can be said that mining software repositories often is a time-consuming task and publishing the retrieved data for studies has not yet been established in the field.

3.2 MSR Paper Review

Mining Software Repositories (MSR) Conferences are being organized since 2004 and in 2013 the tenth edition was held[2]. In these years, the researchers in this field conducted many studies that were based on software repository data. In order to better understand which data sources were often used by researchers in the recent years, we reviewed all papers published in the context of the Mining Software Repositories Conference 2013. During that systematic review, we looked through a total of 63 papers (not including the front matter and keynote). This review also includes a total of twelve papers of the yearly category *Mining Challenge* which conducted studies on a predefined data set. In MSR 2013, the Mining Challenge data set was a simplified version of the official Stack Overflow dump from August 2012 [Bac13].

Of all the 63 papers from MSR 2013, we extracted the data sources that were used by the respective authors of these papers. A grouped list of the results can be found in Table 3.1 that shows the number of papers that used these data sources. There are several papers that not only used one data source, but took data sets from two or more repositories or other origins. That caused the total number of used data sources reflected in Table 3.1 to be higher than 63. When looking at the table, it is striking that *version history and source code*, *discussion boards*, and *issue trackers* are used the most often. The number of data of *discussion boards* might be high because of the Mining Challenge 2013 which included data sets of Stack Overflow. However, we assume that this number also would have been high without the Mining Challenge as Stack Overflow is a highly successful question and answer platform as for example stated by Mamykina et al. [MMM+11]. As such, it is an interesting data repository anyway. That the top three repositories in the table are the most used in the reviewed papers is probably no coincidence. Version history and issue trackers are used by (almost) all larger software projects and constitute a large number of data to study. Furthermore, the data from these platforms is often accessible using an application programming interface (API) or similar mechanisms. As such, data sets can be extracted from the platform itself and downloaded by the researcher in an accessible file format. In addition, it is also interesting to see what other data sources are used by researchers for their studies. For example, notifications and Twitter might not be obvious sources for research in the software engineering field.

An overview of software projects whose data was used in the MSR 2013 papers can be found in Table 3.2. There are several paper authors that conducted studies on diverse projects without mentioning all of them. Such papers are not included in that table. Overall, it is striking that many studies were conducted on the same or similar software projects. For example, there are nine

[2] http://2014.msrconf.org/history.php

Data sources	#Papers
Version history and source code	23
Discussion boards	13
Issue trackers	13
MSR papers	4
Mailing lists	3
Maven repositories	2
Mobile app stores	2
Code documentations	1
Code reviews	1
Notifications	1
Stacktraces	1
System logs	1
Twitter	1
Others	6

Table 3.1: Data sources and the number of MSR 2013 papers that used those in their studies.

Software projects	#Papers
Eclipse and Eclipse projects	9
Apache projects	7
Android	5
ArgoUML	5
Mozilla and Mozilla projects	5
Google Chrome	3
Gnu projects	2
Hadoop	2
IBM projects	2
JabRef	2
Linux	2
Maven	2
Others	20
No specific project	24

Table 3.2: Software projects and the number of MSR 2013 papers that used those in their studies.

paper authors that used data from Eclipse or other projects from the Eclipse Foundation. Seven studies relied on projects of the Apache Software Foundation. And five papers were written based on Android, ArgoUML, and the Mozilla Foundation projects each. We can only speculate about the reasons for that distribution. In any case, all of these projects have a long development history and many thousand lines of code what makes them good resources for research. There may also be existing data dumps available or tools that explicitly mine these software repositories. Maybe the data of other projects is less accessible than, for example, the top five from the table. Either way, the choice of a software project for a study seems to be biased according to the MSR 2013 papers. Table 3.2 also shows that there were studies that did not use a specific software project as can be seen in the last row. The high number of papers is, in this case, influenced by the MSR Mining Challenge where most study authors did not focus on a certain software project.

In addition to our review of the MSR 2013 papers in this section, Demeyer et al. [DMWL13] conducted a text mining study on the MSR papers published from 2004 to 2012 in order to find trends in this research area. One of their findings was targeted to popular and emerging mining infrastructures. They uncovered that CVS and Bugzilla are the most frequently cited version control system and bug tracking system. However, they also found that recently SVN, GIT, and JIRA[3] have steadily gained more interest in the field. Demeyer et al. also noted that researchers should pay close attention to the more recent platforms and repositories which may provide features helping to answer questions that could not be answered back in 2004. Although they did not explicitly mention it, we think that GitHub is an example of such a more recent platform. GitHub was just founded in 2008[4] and after rapid growth they have millions of people collaborat-

[3] https://www.atlassian.com/software/jira/
[4] https://github.com/about

ing on over ten million repositories (see Section 4.2.2). Another recent platform is Stack Overflow which was also founded in 2008[5] and today is part of the Stack Exchange question and answer site network. The large user base of Stack Exchange has asked several million questions over the last years (see Section 4.2.4). Both, GitHub and Stack Exchange, constitute huge data sets for researchers in the mining software repositories field to work with.

3.3 Proposed Solution

The MSR paper review in Section 3.2 shows that the tools and repository platforms that researchers use for their studies change over time and are also generally diverse. Furthermore, the used software projects are numerous while there are some projects or software foundations whose software is used more often than others. As a result, a mining and data sharing platform must be flexible enough to easily extend its functionality to further types of repositories and must not be focused on certain projects only. Instead, it should be a flexible, general-purpose mining solution that can be upgraded without much effort. Based on these requirements, we present our approach to mining software repositories.

In order to help researchers in retrieving data from different software repository types as simply as possible, we create a web-based repository miner application called MiningHub. The main focus of MiningHub is simplicity; researchers should be able to obtain data for their studies with as little effort as possible. To achieve this, we need to hide the API requests and the quirks that come with them from the user. At the same time, MiningHub needs to be extensible to add support for further repositories and also to be able to mine diverse projects without any modifications. We ourselves cannot add support for each and every API or repository, so we have to give the users the possibility to add new services on their own. We do so by means of a novel mining description format that we call mining script. Such scripts contain simple instructions on what parameters the user has to provide, how to call a specific API, and how to store the retrieved data. These commands can be added for each service that answers queries to its data by means of an API. Researchers should therewith be able to write their own mining scripts and upload them to the MiningHub script catalog. We also define a specification that mining scripts have to comply with before they can be executed on MiningHub. This ensures that MiningHub can accurately parse them into a a valid set of instructions and use them to mine software repositories. When researchers would like to retrieve data from a repository, they just fill in the corresponding URL (e.g. `https://github.com/groovy/gmaven/`) to a MiningHub import dialog. Based on URL patterns, appropriate mining scripts are selected from the script catalog employed by MiningHub and the researchers can choose a suitable one for their studies. Then the required parameters (e.g. an API key, or a time frame) as defined in the chosen mining script have to be entered. After that, the data import can be started and runs completely autonomously as long as it needs to retrieve all requested data. As soon as the mining process has successfully completed, the researchers can download the imported data sets to work on them. Furthermore, MiningHub also allows to search the data that was already mined using the platform.

With that approach, some limitations in the mining software repositories field can be overcome. For a start, we write several mining scripts for important software repositories as a proof-of-concept. However, the MiningHub platform is extensible so that researchers can write and execute their own mining scripts as well. Furthermore, they can share their scripts with other researchers for reuse. And also the mined software repository data sets can be made publicly available so that others can work on exactly the same data. This enables a way to replicate studies and compare them to similar approaches by other authors. MiningHub aims to simplify the mining software repositories step of many studies as well as make it possible for other researchers to

[5] `http://stackexchange.com/about`

work on exactly the same data sets as were used in former experiments. We believe that by using the described approach, MiningHub is able to attain both of these goals.

3.4 Technology Evaluation

MiningHub is a web-based platform to retrieve data sets from software repositories. Before starting with the implementation phase, we conducted an evaluation of different technologies and programming languages that we could use. In conclusion, we use Groovy[6] with Grails[7], MongoDB[8], Bootstrap[9], and jQuery[10]. For the specification of mining scripts, we use YAML[11]. The rationals behind these decisions can be found in the following paragraphs.

The client-side, which represents the user interface (HTML, CSS) and JavaScript, must ensure compatibility for mobile and desktop browsers. It also must enable effortless use of asynchronous requests, for example to load new content or store changed data. To fulfill these requirements, we decided on using Bootstrap and jQuery. Bootstrap is a popular front-end framework which simplifies creating appealing interfaces and is also well-suited for mobile websites. jQuery is one of the most used JavaScript frameworks available. It makes adding new functionality and handling asynchronous (AJAX) requests comfortable. Also, due to its popularity, there is a broad range of resources and example code available.

Regarding databases, there are numerous products to choose from. To narrow down the number of solutions to look at, we preselected the following popular systems: MySQL, Oracle Database, Microsoft SQL Server, MongoDB, and CouchDB. To decide for one of them, we first separated these database platforms into the two main categories *NoSQL databases* (MongoDB, CouchDB) and (mainly) *relational databases* (all others). While NoSQL is a very broad term lacking an agreed-on definition (for example [Tiw11]), both MongoDB and CouchDB are similar document-based databases without the need of a fixed schema. In our project, we use mining scripts created by users to retrieve and accurately store data mined from diverse APIs. Therefore, we have absolutely no chance to create one single database schema beforehand that will fit all kinds of data sources. We would have to dynamically change it based on different mining scripts which would be cumbersome and error-prone. So the not required schema alone was already enough for us to prefer a NoSQL database (DB) for this project. However, there is also another advantage in using such a document-oriented DB instead of a purely relational one. MongoDB and CouchDB both support nesting of data sets similar to the nesting that is used in the Extensible Markup Language (XML) or JavaScript Object Notation (JSON). As APIs mostly use one of these formats, the exposed data is often nested. As a result, using a document-oriented DB simplifies storing the retrieved data as it does not have to be transformed or separated into several relations. Also, exporting the data is simpler as it does not have to be assembled and nested again but instead can be exported directly, maybe even without any modification at all. So we preferred either MongoDB or CouchDB over a relational database for MiningHub. Choosing between the two is not straightforward, though. They are similar, and both have their individual advantages and drawbacks which makes deciding for one of them based on technical arguments a difficult task. Looking at the non-technical differences, we decided in favor of MongoDB as its community appears to be larger than the one of CouchDB. A larger community often means it has more help resources as well as more up-to-date plugins and drivers for programming languages or frameworks. It is furthermore important to point out that MongoDB uses a special terminology. Tables

[6] http://groovy.codehaus.org
[7] http://grails.org
[8] http://www.mongodb.org
[9] http://getbootstrap.com
[10] http://jquery.com
[11] http://www.yaml.org

are called *collections* and *documents* are the content of these collections. In relational databases, documents would be rows in a table.

	Java	**Scala**	**Groovy**	**JRuby**
Paradigm or specialty	plain, old Java; object oriented programming	functional programming	"shorthand Java", compact code	similarities to Java and other languages, object oriented, procedural
Type system	static	static	static & dynamic	dynamic
Size of the community	huge community	large community	large community	rather small community
IDE support	great IDE support	good IDE support	good IDE support	IDE support needs external plugins
Build tool & dependency manager	Maven, Ant	sbt	Gmaven, Gradle, Gant	RubyGems
Web frameworks	Spring, Play 2	Lift, Play 2	Grails	Ruby on Rails
Support for JSON	possible to create & read JSON but laborious	possible to create & read JSON but laborious	very easy syntax to create & read JSON	very easy plugin(s) to create & read JSON

Table 3.3: Comparing Java and three Java-based programming languages. A green background shows a good property in the corresponding language whereas a red background represents a bad property with respect to our project. All other cells represent a neutral rating or are not very decisive for our goal.

We narrowed down the choice of a programming language to the ones that are Java-based, the reason being that there are many third-party libraries available for these languages. In order to confine that preselection even more, we selected Java, Scala[12], Groovy, and JRuby[13] as candidates as they appear to be among the most used Java-based languages available. In order to evaluate them, we compared several features and characteristics as can be seen in Table 3.3. One of the most important requirement is good support for JSON as we have to handle different types of APIs which often expose data in the JSON format. As MongoDB also answers queries in the same data format, we have to deal with it a lot and the programming language should make it as effortlessly as possible. An aspect which might also directly influence the JSON support is the type system of the programming language. While dynamic typing makes parsing and creating JSON data easier, static typing can help speeding up the development, debugging as well as refactoring. As can be seen in Table 3.3, Groovy supports both static and dynamic types, and offers good support for JSON which gives it an advantage. Regarding community size and IDE support,

[12] `http://scala-lang.org`
[13] `http://jruby.org`

Java is the clear winner as it has a very large community, a lot of help resources, and also the most mature IDE support. On the contrary, JRuby has got a rather small community (while Ruby itself does have a large community, JRuby does not). As a broad adoption of a programming language most often means it has got a lot of plugins (e.g. database drivers or API wrappers), popularity is an advantage here. As for tool support and web frameworks, all of the four languages have widely-used build tools as well as full-featured and well-known frameworks. So after evaluating each language, we decided on using Groovy and Grails. Especially the simple JSON handling as well as the broad adoption of Groovy brought about this decision. And as Grails is the main framework of Groovy, it is obvious to embrace it.

Regarding mining scripts, we first planned to use a domain specific language to implement these. However, we found that mining scripts do not need to contain functionality but only information. A text-based file format is therefore easier to use for the user and also uncomplicated to parse by an application. Obvious candidates would be JSON and also the lesser-known YAML format (the markup language, not the CSS framework with the same name). Compared to JSON, YAML has the advantages of no required braces, commas, or quotes, and support for comments. These differences make YAML much easier to read and write by hand. As a result, we chose YAML as format for our mining scripts.

Chapter 4

Implementation

This chapter explains the exact implementation approach we took with MiningHub. First, we describe its software architecture. Then we show what mining scripts are and how they can be written. The third section is about the database as well as the general data structure of stored content. Then we go through the MiningHub software repository mining approach including its error handling. We also show how MiningHub provides search and download capabilities. The last section depicts the user interface as well as social features and mobile support of our platform.

4.1 Software Architecture

MiningHub is a platform for mining software repositories. As described in the solution proposal (Section 3.3), it should allow the creation of import jobs that retrieve software repository data in the background. We enable that functionality with a software architecture based on three components as follows. MiningHub consists of the website where researchers can create import jobs, the importer that retrieves data from repositories, and a component containing all shared functionality. The reason for this setting is that the importer might be ran from several autonomous instances (i.e. import workers) at the same time. This allows MiningHub to scale with the amount of work it has to do and helps to overcome API limitations, for example when a repository limits the amount of requests from a certain IP address. The MiningHub workers are scheduled to regularly check for new import jobs. If there is any work to do, they handle the specified imports as described in Section 4.4.3. As there may be only one or instead many different workers retrieving data at the same, they use the database for synchronization. That database runs on a server and can be remotely accessed by the workers. This architecture of workers and the database also allows to use MongoDB sharding to increase the DB scaling in the future. The MiningHub website is implemented using the model-view-controller (MVC) pattern with Grails. The model defines the data structure using an object-relational mapping between object-oriented domain models and the database. It also provides functions to read and update data from the DB. The views specify how the MiningHub webpages look like and what content they contain. Functionality is provided by controllers, these handle requests, user sessions, and send content to the views. The functionality of controllers are encapsulated in service classes that are called from controllers based on the requests. This simplifies reusability and testing capabilities of the code. The website and worker components also share some domain classes and utility functions. To avoid code duplication, shared functionality is implemented by means of a Grails plugin that is inserted in both other components. By using the Grails dependency resolution, the shared code then can be used as if it was implemented within the website or worker component themselves.

4.2 Mining Scripts

MiningHub is an extensible platform for retrieving data from different types of repositories. It allows to be controlled by mining scripts, these are sets of commands written in YAML that specify how MiningHub can query data from repositories. MiningHub employs a catalog of different mining scripts, each with a title and a description so that researchers can look through them. All mining scripts are meant to extract information from one (or even several) software repositories. For example, they define the URLs of the API endpoints and parameters that are needed from the user. They also define which components from a repository are retrieved and stored by MiningHub. Mining scripts also specify whether a component has to be retrieved from an API using a paging mechanism or whether it depends on another component. MiningHub then uses these scripts to request the repositories and retrieve data accordingly. Overall, the MiningHub approach aims to encourage researchers to write their own mining scripts so that they can upload and use them to mine software repositories using MiningHub. Mining scripts can also be shared and reused by other researchers. For that purpose, the script catalog also shows the complete specification of the mining scripts and researchers can either reuse them directly or modify them according to their needs. For example, an existing mining script to mine an issue tracker can be configured to only retrieve issues of type *defect* in the date period from 2010 to 2013. Mining scripts allow to change or add completely new functionality to MiningHub and are thus an essential part of its overall approach. This section explains the essentials of the YAML format and describes the mining script approach on three examples of popular repositories. After that, we describe the mining script validation mechanism.

4.2.1 YAML

Mining scripts are written in the human-friendly text serialization format YAML (a recursive acronym for *YAML Ain't Markup Language*). Its feature set is similar to the one of JSON. However, as YAML is lesser-known than JSON, we think it is important to go through the essentials of its syntax. YAML is a simple, nested file format that was created with the aim to be friendly for both humans *and* computers. A basic example can be seen in Listing 4.1, which shows the notation of key/value, maps, comments, and lists. For humans, YAML looks very clean and intuitive. Its structure and nesting is similar to JSON. But in contrast to that, YAML does not require curly braces, square brackets, or commas to format the content. Nesting is simply done by indenting. Listing 4.1 also shows that values do not need to be enclosed by quotation marks, and they are still recognized as strings. For computers, a YAML document can be read and, thanks to the fixed semantic and the nesting, parsed into either a list of values or a list of key/value pairs. After parsing the YAML example, the values can be accessed by using for example `key`, `map.key2`, `list[0]`, or `listOfMaps[1].key1`. These data structures can then be iterated through and used in the program logic to influence the behavior of the software as we do for mining software repositories. So the main reason why we chose YAML for defining mining scripts in the first place was that it makes creating them very straightforward. The users do not have to think about (or close) braces or commas. In addition, most quotation marks can be omitted and there is support for comments. But YAML still has the same expressiveness as JSON and is easily parsable. YAML even supports implicit and explicit value types as can be seen in Listing 4.2. Some identifiers like `true` (boolean), `n` (boolean), and `1` (integer) imply a type and are handled as such. However, when these values should be interpreted as string types, they have to be enclosed by quotes. In general, quotes for strings are only needed when a YAML parser infers a value to be of another type than a string or when the parsed content could be ambiguous. So they can be omitted most of the time.

```yaml
key: value
map:
    key1: value1
    key2: value2
list:
  - element1
  - element2
# This is a comment
listOfMaps:
  - key1: value1a
    key2: value1b
  - key1: value2a
    key2: value2b
```

Listing 4.1: Basic YAML example.

```yaml
boolean:
  - false
  - y
integer:
  - 0
string:
  - text
  - "false"
  - "y"
  - "0"
  - " "
null:
  -
```

Listing 4.2: Types in YAML.

4.2.2 GitHub

GitHub is a popular social coding website that offers public and private project repositories with code version control and issue tracking systems. As of January 7, 2014, their 4.9 million users have created more than 10.2 million repositories. This constitutes a very large source of data for software engineering research. And GitHub is indeed sometimes used or at least mentioned in studies of the MSR and others, for example to calculate code complexity metrics [AS13a]. A study by Demeyer et al. [DMWL13] in 2013 created a trend analysis of past MSR papers. They found that while newer platforms like GitHub were referenced but only slowly picked up in studies, researchers should pay close attention to them. These platforms might provide new features that others do not and thus allow to answer questions that kept being unanswered so far. As a result of emerging platforms, the MSR 2014 Mining Challenge uses data from GitHub [Gou13]. So GitHub appears to be an important repository for research and is getting more and more attention in the field. Therefore, we created a mining script that allows to retrieve data from this repository using MiningHub. In this section, we go through that script and explain the essentials of the mining script syntax. As a result, this section contains an extensive description of the mining script concept and structure on the example of GitHub. In contrast to that, the scripts for JIRA (Section 4.2.3) and Stack Overflow (Section 4.2.4) are less broadly explained as most things apply to all three of them. We also indicate that all the keys that are printed in bold in mining script listings are part of the schema definition (Section 4.2.5) and have a special meaning. Keys that are not in bold are defined by the writer of the mining script and name components, parameters or arguments.

The YAML code in Listing 4.3 shows the outline of a mining script for GitHub. Lines marked with `...` mean that we left out some content of the full script for brevity. The complete mining script for GitHub can be found in Listing A.2 in the appendix. The first two lines of Listing 4.3 define the name and description of a mining script, this information is used in the MiningHub script catalog. On line 3, the format of the API that GitHub provides is defined; in this case it is *json*. JSON is the default value and is used if the `apiFormat` definition is missing from a mining script. Currently, JSON is also the only supported format. We plan to include more formats, like XML, in the future, though. Lines 4 to 8 show the `urls` list that contains one or several URL patterns that this mining script supports. The `pattern/parameters` block defines a pattern for a given URL, for example `https://github.com/troxler/test-repository/`, and causes MiningHub to extract some contents of the URL into the parameters. In this case, *troxler* and *test-*

```
1  name: GitHub.com
2  description: Powerful collaboration, code review, and code management for open
       source and private projects.
3  apiFormat: json
4  urls:
5    - pattern: "^http[s]?://github.com/([^/]+)/([^/]+)[/]?"
6      parameters:
7        - user
8        - repo
9    ...
10 parameters:
11     branch:
12         type: string
13         description: The name of the branch to mine (often master or trunk).
14         default: master
15     accessToken:
16         type: password
17         description: The Personal Access Token obtained from GitHub.
18 globalArguments:
19     user:
20     repo:
21     accessToken:
22 components:
23     ...
```

Listing 4.3: Outline of the GitHub mining script.

repository would be stored in `user` and `repo`, respectively. These variables can then be used to request data from an API URL. Some more URL patterns of GitHub repositories were left away in the listing. It should be noted that the `parameters` list is optional and has not to be defined if the URL does not contain any identifiers that need to be used when requesting the API. So a valid mining script url pattern could also just look like `"^http[s]?://github.com"`, which matches all GitHub URLs. Such a mining script might, for example, extract data from the GitHub event timeline and thus does neither need a user nor repository name to request the API. Line 10 to 17 define parameters that the user has to fill in prior to the start of the import process. If defined, each sub-map of `parameters` causes the import creation dialog to show a parameter input. Each parameter has a name that is defined by the key (e.g. *branch* on line 11), a type, and a description. The type is optional or one of *string* (default type), *password*, *date*, *datetime*, or *int*. For API keys and other private parameters, the type must be defined as *password* (line 16) because only parameters of that type are completely hidden in the MiningHub user interface. Values of all other parameter types are visible instead. The description of a parameter must be clear enough so that a user knows what he has to fill in to retrieve the desired data sets. In addition, it is also possible to set a `default` value for a parameter. This value is then pre-filled in the corresponding input field in the import creation dialog but might still be changed if needed. In the case of GitHub, a default value for the branch name is defined as this is most often, but not always, *master*. Furthermore, this parameter also allows to mine another branch (e.g. a feature branch) in a repository. On line 18 to 21 of Listing 4.3, global arguments are listed. These arguments are meant to replace the corresponding placeholders in API URLs (see line 5 of Listing 4.4 for an example).

They are called global because they are part of every single URL that is requested from the API. In addition, it also possible to declare arguments on a component-based approach instead as can be seen in the description of Listing 4.4. The global arguments are just a shortcut for that to avoid repetitive argument lists. In the case of GitHub, on every API address that is called, MiningHub first has to insert the arguments `user`, `repo`, and `accessToken`. These arguments can either be extracted from the URL given to the import dialog or filled in from the user. So it is important to ensure that these two types of arguments do not include any duplicate identifiers. That means that `urls[*].parameters` and `parameters` must not contain the same names. Line 22 of Listing 4.3 shows the `components` map. This is a container for all the components that this mining script retrieves from an API. Each component has to be specified with a key and a declaration of various settings as can be seen in Listing 4.4. Currently, the mining script for GitHub retrieves the following components: `commits`, `commitComments`, `issues`, `milestones`, `issueComments`, `branches`, `tags`, `releases`, and `forks`.

```
1  commits:
2      description: Code commits
3      primaryKey: sha
4      call:
5        - url: "https://api.github.com/repos/{{user}}/{{repo}}/commits?sha={{
             branch}}&last_sha={{lastSha}}&access_token={{accessToken}}"
6          arguments:
7              lastSha:
8              branch:
9          store: .
10         index:
11           - commit.message
12           - commit.author
13         argumentRotation:
14             name: lastSha
15             start: ""
16             type: text
17             stop:
18                 source: http
19                 key: Link
20                 isset: false
21             continue:
22                 valueFromRegex: 'last_sha=([^&]+)[^>]+>; rel=\"next\"'
```

Listing 4.4: Component for commits of the GitHub mining script.

Listing 4.4 shows how a component in a mining script is defined. It specifies the complete definition of the `commits` component that is a sub-map of the `components` container. Each component must have at least a description and the definition of a primary key. The description defines what exactly the corresponding component retrieves from the API and is an important information that can be viewed in the MiningHub script catalog. The `primaryKey` specifies the key name of a unique value returned in the retrieved component. Most APIs return their data in a representation similar to Listing 4.5, which contains a list of commits. MiningHub has to store this result in a database. And to avoid any duplicate entries, the value of the key name defined in

`primaryKey` is copied to a MongoDB `_id` key that always has a unique constraint. Furthermore, MiningHub has to be able to reference certain items in each components. To ensure this reference, MiningHub uses the unique identifier in `_id` for each of them. In the case of GitHub commits, the primary key is the `sha` value that uniquely identifies each commit. So it could be used to retrieve more information about a certain item of a component from the API (e.g. the commit with the `sha` value of *91064d282b...b*).

```
1  [
2    {
3      "sha": "91064d282b1ddd4b7f1620b0ba4211e3c1d33c3b",
4      "commit": {
5        "author": { "name": "troxler" },
6        "message": "Create README.md"
7      },
8      ...
9    },
10   {
11     "sha": "5e9bd60dfdf1b2778381aaf056526ea3f93b324a",
12     "commit": {
13       "author": { "name": "..." },
14       "message": "..."
15     },
16     ...
17   }
18 ]
```

Listing 4.5: Minimized JSON response from a GitHub API request.

Line 4 of Listing 4.4 starts the most important part of a component. It defines how and where data has to be retrieved from an API. There are actually two different types of specifications, namely `call` and `each`. The fundamental difference between the two is that `call` is used to retrieve lists of items whereas `each` is used for requesting more information about a specific item. The latter is explained in Section 4.2.4 on the example of Stack Overflow and we focus on the `call` requests in this section for GitHub. Line 5 defines the API URL to request commits with some placeholders that are denoted by `{{placeholderName}}`. The static arguments `user`, `repo`, and `accessToken` where already listed in the `globalArguments` section of Listing 4.3. On the contrary, arguments that are only used in the commits component are `lastSha` and `branch` as can be seen on line 6 to 8 of Listing 4.4. `branch` has to be defined by the user in the import dialog and `lastSha` is assumed to be empty for the moment. With all these arguments listed in the mining script, MiningHub knows that it has to replace the placeholders in the `url` key with the actual values. After that has been done, MiningHub issues a hypertext transfer protocol (HTTP) request to the complete URL and handles the response. The most important step in doing so is storing the returned data. For this, mining scripts have to define what exactly needs to be stored from the response using the `store` key. Line 9 of Listing 4.4 states `store:.`, which is also the default when nothing was defined. A dot (.) value indicates that a list of items is expected and MiningHub has to store each element of that list in a MongoDB document. On other APIs, the result of requests might look differently. An example response could be `{"results":[...], "otherKey":"otherValue"}`, which encapsulates the results in a sub-list. To store each el-

ement in the `results` list, `store:results` has to be used. When the nesting is deeper, the definition could also look like `store:results.commits.items` for example. It is important to point out that the `primaryKey` definition is relative to the value in `store`. Line 10 to 12 define which values of the response have to be indexed. This is specified so that MiningHub can search the retrieved items as is described in Section 4.5. When a mining script does not define any values to index, retrieved items cannot be found by the search engine. However, the indexing or search capabilities have no effect on the data import or the download thereof. As a result, indexing is optional.

```
1 HTTP/1.1 200 OK
2 Server: GitHub.com
3 Status: 200 OK
4 Content-Type: application/json; charset=utf-8
5 Link: <https://api.github.com/repositories/927442/commits?sha=master&last_sha=2
      f0f7b4bce...6>; rel="next", <https://api.github.com/repositories/927442/
      commits?sha=c979a1fefa...d>; rel="first"
```

Listing 4.6: Minimized HTTP response from a GitHub API request.

Most `call` requests are either only called once in an import process and then never again, or they use a paging mechanism to retrieve more data. Paging in APIs is very similar to paging in, for example, a discussion board. For a certain question, there might be ten answers on the first page. To read the next ten answers, the next page has to be requested. It is exactly the same for API requests. Usually, the URLs are thus just extended by a `page=2` argument to retrieve the second page. For GitHub commits, this is slightly different. The page number is not an increasing integer starting from one, but instead it uses the `sha` values that are returned in the API responses. The paging mechanism in mining scripts is achieved with the `argumentRotation` block starting on line 13 of Listing 4.4. It defines the `name` of the argument that changes between consecutive requests and also its `start` value. That `start` value can be either a string (in this case it is an empty string) or an integer, the default value is 1. Similarly, the `type` defines the value type of the changing argument. It is either *text* or *int*, whereas the latter is the default. As for GitHub commits a string identifier is needed for the paging, MiningHub cannot just increase it by 1. So for the first request, it just uses the `start` value which is an empty string in this case. After each response of the API, MiningHub then analyzes the result and also the `stop` condition on line 17. This condition defines whether all the items for a component were retrieved and no more requests are needed. So when MiningHub examines the condition and it resolves to *true*, the corresponding component request is marked as completely retrieved and no further requests are made on it. Such a condition always involves a value out of the response. Currently, there are two possible `source`s for that value, it is either *result* (default) or *http*. The former selects a value of the response content, which is mostly JSON. The latter reads the HTTP headers of the response instead. For both of them, a `key` is necessary to be defined so that MiningHub knows which value it has to extract from the response or the HTTP header. Then either `isset` or `value` have to be defined to compare the extracted value. `isset` is either *true* or *false* and checks whether the specified `key` is existing or not. `value` specifies a value of an arbitrary type (e.g. null, boolean, string, integer) and causes MiningHub to compare the defined and the extracted values. For the GitHub commits, the `source` of the comparison value is *http*. So MiningHub examines the HTTP response of the request and tries to find the key name defined in `key`. If the key is not found, the `stop` condition resolves to *true* and no more requests are done on the `call` block of this

component. But if the condition does not resolve to *true*, the `continue` block is examined. Line 22 defines a `valueFromRegex` that is currently only available on HTTP responses and extracts—as the name implies—a value using a regular expression. GitHub uses such HTTP responses for many different aspects, for example to return the URL of the next page. A minimized example of such a response can be found in Listing 4.6. As the mining script defines the comparing `key` to be *Link*, the important part of that listing is line 5, which defines full URLs to the next and the first page of commits. The regular expression that is specified in `valueFromRegex` then extracts the `last_sha` value of the next commit page from the HTTP header. After that extraction, all required arguments are given and the placeholders in the `url` can be replaced by their proper values. For the paging mechanism, the `last_sha={{lastSha}}` is then replaced with a proper *sha* value and the second page of commits can be requested. These steps can be repeated until the `stop` condition eventually evaluates to *true*. The `continue` block also provides an `increment` key which is set to *1* by default for simpler paging approaches. Such paging examples can be found in the description of Listing 4.7 below and in Section 4.2.3 on the example of the issue tracking system JIRA.

```
1  commitComments:
2      description: Comments to Commits
3      primaryKey: id
4      call:
5        - url: "https://api.github.com/repos/{{user}}/{{repo}}/comments?page={{
             commentPage}}&access_token={{accessToken}}"
6          arguments:
7              commentPage:
8          store: .
9          argumentRotation:
10             name: commentPage
11             start: 1
12             stop:
13                 key: .
14                 value: []
```

Listing 4.7: Component for commit comments of the GitHub mining script.

The component to retrieve commit comments in Listing 4.7 is similar to the commit component but shows two important differences. First, it uses a different paging mechanism, and second, it shows another `stop` approach. The paging for commit comments works with increasing page numbers. In the listing, this is reflected on line 11 where the `start` value is defined as 1. The `type` can be left away this time as *int* is the default. Furthermore, the whole `continue` block is omitted as the default behavior for numeric page values is to increase by 1 on each iteration. Evidently it is possible to write shorter mining scripts by using these default specifications. The `stop` block only defines a `key` and a `value`. That causes MiningHub to compare the whole response (defined by `key: .` on line 13) with an empty list as specified by the `value` key on line 14. The rational behind this is that GitHub returns an empty JSON list when a client requests items from a too large page number. This is not ideal as it wastes a request of the limited API quota. However, it is a needed step as the GitHub API is inconsistent on that matter. While some components do not include the *Link* header in the HTTP response on the last page, other components always transmit that header. As result, we use the response header to determine

whether there is a further page and if that header is always set, we use the approach with the empty list. Either way, both approaches work well to see whether a `call` request of a component has completed. The further components retrieved by the GitHub mining scripts work similarly to Listing 4.4 and Listing 4.7 and are thus omitted for brevity.

4.2.3 JIRA

JIRA is a popular issue and project tracking software by Atlassian. It can either be used as a service or downloaded and installed on a server. There are many large software projects and companies that use JIRA for their project planning. Examples are the Apache Software Foundation[1], Spring by Pivotal Software[2], and projects at Codehaus[3]. As a result, there are large data sets available for research. Thus, JIRA is often used in mining software repository studies as can also be seen in the MSR 2013 paper review in Section 3.2. A very often used JIRA instance is the one of Apache which includes many different projects of that foundation and is a huge source of data. The popularity among software projects and also researchers has led us to create a mining script for JIRA, too. In this section, we describe some parts of it. However, as most aspects of the JIRA mining script are similar to the one used for GitHub (Section 4.2.2), in this section we only go through the differences between the two. The complete mining script for JIRA can be found in Listing A.3 in the appendix.

```
1  name: "Jira: Public Issue Trackers"
2  description: This mining script will collect data about a single project and
       all its issues from public issue trackers. ...
3  urls:
4    - pattern: "(^.*)/rest/api/2[/]?"
5      parameters:
6        - url
7  parameters:
8      projectKey:
9          type: string
10         description: The mandatory key name of a project, it must be UPPERCASE.
11 globalArguments:
12     url:
13     projectKey:
```

Listing 4.8: Header of the mining script for JIRA.

Listing 4.8 shows the head of the mining script for JIRA. It, again, starts with a `name` and `description`. In this case, the `name` value must be enclosed with quotes as it contains a double colon and YAML requires quotes to avoid ambiguity. The `urls` container starting on line 3 does not define a domain name as JIRA may be hosted on any domain. As a result, it is required to extract and store the domain name from the given URL as can be seen on line 4. Furthermore, the mining script only matches URLs that include */rest/api/2*, which is part of the API address of JIRA. An example of such a URL is `https://issues.apache.org/jira/rest/api/2/` that obviously points to the Apache JIRA instance. The `parameter` block causes MiningHub to

[1] `https://issues.apache.org/jira/`
[2] `https://jira.springsource.org`
[3] `http://jira.codehaus.org`

ask for a project name, which must be written in uppercase or JIRA does not recognize it. An example project for Apache could be *SOLR*. The `globalArguments` then define the `url` of the JIRA instance as well as the given `projectKey` to be used in the API requests.

```
1 projects:
2     description: Project information
3     primaryKey: id
4     call:
5       - url: "{{url}}/rest/api/2/project/{{projectKey}}"
6         # only store one single element, not each element in a collection
7         store: ..
```

Listing 4.9: Component for projects of the mining script for JIRA.

The `projects` component in Listing 4.9 retrieves information about the project that was defined in the `projectKey` parameter. That includes, for example, a description, the project lead, its components, issue types, versions, and so on. Line 5 shows the URL that does not specify a domain but instead has a placeholder. As the domain is extracted from the URL given in the import creation dialog, the `url` contains just the placeholder which is replaced by the proper domain before any request to the API is made. The `projects` component contains a `call` request that is done once per import and directly retrieves all available information in one row. Thus, there is neither a `stop` condition nor an `argumentRotation` block needed. One important aspect of this component is also the definition of `store`. The JIRA API returns a map of key/value pairs in the response for project data. If instead `store:.` was used, MiningHub would iterate through the map and store each single value in an individual MongoDB document. To avoid that behavior and cause MiningHub to store the whole result of the response in a single document including all the keys, this component uses the double dot (`..`) notation. Its semantic is similar to the handling of paths, for example in a command line. A double dot causes the current working directory to be changed by going up one level. In mining scripts, the semantic is the same. Listing 4.10 shows the second component of the JIRA mining script, namely the `issues` component. It is similar to other components that use a paging mechanism with increasing numbers. However, the paging on the JIRA API works slightly different. It is zero-index, which means that the first issues can be found on page 0 and not 1. This is reflected by a proper `start` value on line 11. In addition, the page number increases not by 1, but instead by the number of issues per page. So as the `url` on line 5 defines `maxResults` per page to 20, the second issue page has the number 20. As a result, the mining script has to explicitly define the `increment` value to be 20 (line 16) instead of the default behavior to just increment by 1.

The mining script for JIRA is relatively small and Listings 4.8, 4.9, and 4.10 cover almost its whole content (we only omitted `index` blocks). But although that mining script only defines two components on a total of 53 lines of YAML, it retrieves many different data sets regarding the issues. Reason being that the API of JIRA returns all the information related to issues when iterating through them. For example, all the comments to an issue extended by their user details are directly included in the issue response. Other aspects that are included are votes, resolution, priority, watches, labels, reporter including user details, and many more. As a result, API responses for issues on JIRA are extensive regarding their size and we limited the `maxResults` per page to a low value of only 20.

```
1  issues:
2      description: Issues
3      primaryKey: id
4      call:
5        - url: "{{url}}/rest/api/2/search?jql=project={{projectKey}}&startAt={{
             issuePage}}&maxResults=20&fields=*all"
6          arguments:
7              issuePage:
8          store: issues
9          argumentRotation:
10             name: issuePage
11             start: 0
12             stop:
13                 key: issues
14                 value: []
15             continue:
16                 increment: 20
```

Listing 4.10: Component for issues of the mining script for JIRA.

4.2.4 Stack Overflow

Stack Overflow is a popular question and answer board for professional and enthusiast programmers by Stack Exchange. Stack Exchange operates a whole network of numerous different answer sites on diverse topics. As of January 7, 2014, their 4.6 million users asked 8 million questions and wrote 14.2 million answers. Stack Overflow is the largest of their sites by far as it alone had 6.4 million questions. The Stack Exchange network is often used in research as, for example, can be seen in the MSR 2013 paper review in Section 3.2. However, it should be noted that the mentioned number of papers that used data from Stack Overflow is influenced by the fact that the MSR 2013 Mining Challenge provided data sets from that Stack Exchange site [Bac13]. Nonetheless, Stack Overflow is a rich collection of data and an interesting platform for research. As a result, we created a mining script for Stack Overflow that retrieves questions, answers, comments to either of them, and also user data for a certain question tag. Its complete specification can be found in Listing A.4 in the appendix.

The mining script for Stack Overflow is similar to the ones of GitHub and JIRA. Therefore we omit the header part this time and only describe it briefly. To mine questions from Stack Overflow, as always, an import with a given URL has to be created on MiningHub. The mining script we wrote accepts URLs with the pattern `^http[s]?://stackoverflow.com/questions/tagged/([^/+]+)`. This matches all Stack Overflow addresses that show the questions of a certain tag. That tag is automatically extracted from the URL and stored so that it can be used for the API requests. The mining script also specifies parameters that the user has to fill in, namely the API `key` as well as `fromdate` and `todate`. The last two are both of `type` *date* and specify a date period from within questions should be considered during the retrieval phase. The date type is used in the HTML code of the parameter input fields of the import dialog. Some browsers support the *date* type directly and show a user interface to insert dates. For browsers that do not yet support that input type, we show the pattern *YYYY-MM-DD* of a valid date in the text input. Listing 4.11 defines how to retrieve questions from Stack Overflow according the the given parameters (`argumentRotation` was omitted for brevity). On line 5 it specifies a URL

```
1  questions:
2      description: questions
3      primaryKey: question_id
4      call:
5        - url: "https://api.stackexchange.com/2.1/questions?page={{page}}&
             pagesize=100&fromdate={{fromdate}}&todate={{todate}}&sort=creation&
             tagged={{tag}}&site=stackoverflow&filter=withBody&key={{key}}"
6          arguments:
7              todate:
8                  type: date
9                  format: unix
10             fromdate:
11                 type: date
12                 format: unix
13             tag:
14             page:
15         store: items
```

Listing 4.11: Component for questions of the mining script for Stack Overflow.

that includes parameters for paging, sorting, filtering, and also to narrow down the questions by date and tag. In contrast to others, this mining script uses the `type` and `format` specifications in the `arguments` container. The Stack Exchange API allows to retrieve questions by date using the `fromdate` and `todate` parameters. These expect as a value a unix timestamp, which is the number of seconds since January 1, 1970, at 00:00:00 UTC. As an example, the date of January 1, 2014, 00:00:00 UTC would be *1388534400* in unix time. Having to insert such timestamps as parameters would be annoying for the user. So MiningHub translates arguments of `type` *date* and `format` *unix* to a unix timestamp before inserting it into the API URL.

Listing 4.12 shows the `answers` component that causes MiningHub to retrieve answers for every question that was stored in the database. This is specified using the `each` request that looks similar to `call` requests but always depends on another component. In this case, `answers` depends on the component *questions* as is defined on line 5. On the Stack Exchange API, answers are retrieved by requesting `/questions/{{questionID}}/answers`. The `{{questionID}}` can thereby either be replaced by a single question ID or by several IDs concatenated with the semicolon (*;*) as separator. MiningHub is able to deal with both approaches and for this mining script we use the latter as that uses less API requests. First, the page argument start `value` is set to 1. Then, the `argumentEach` block is examined. It defines the `name` of the argument that is used to retrieve answers on. We defined it as *ids* and also put the corresponding placeholder into the `url` on line 6. The `key` specifies the name of the identifier that is used in the `component` defined on line 5. In the case of answers, it is the *question_id* that has to be used to retrieve answers for that corresponding question. The `merge` key defines how many question IDs are merged together and the `separator` defines what character is used as a separator. To sum up, MiningHub reads `question_id`s from the retrieved questions, merges 10 of them together with the separator *;* and replaces `{{ids}}` in the `url` value with the concatenated IDs. Then MiningHub can request the answers. After a response was retrieved and the content of `items` (as defined on line 15) was stored, MiningHub examines the `repeat` block. Similar to a `stop` container, `repeat` compares a part of the response with a defined value and then either continues or aborts the iteration. Again, the optional `source` key defines the response source and is either *result* (default) or *http*. The

```
1  answers:
2      description: Answers to the Questions
3      primaryKey: answer_id
4      each:
5        - component: questions
6          url: "https://api.stackexchange.com/2.1/questions/{{ids}}/answers?page
               ={{page}}&pagesize=100&order=asc&sort=creation&site=stackoverflow&
               filter=withBody&key={{key}}"
7          arguments:
8              page:
9                  value: 1
10         argumentEach:
11             name: ids
12             key: question_id
13             separator: ;
14             merge: 10
15         store: items
16         repeat:
17             key: has_more
18             value: true
19             argument: page
```

Listing 4.12: Component for answers of the mining script for Stack Overflow.

`key` specifies which key of the response has to be compared with either a `value` or a boolean `isset`. With the definition on line 17 and 18, MiningHub reads the *has_more* `key` from the JSON response and compares it to the boolean *true* value. If that comparison evaluates to *true*, the `argument` is increased by 1 and inserted into the `url`. Then MiningHub retrieves the next page of answers for the given question IDs. Thus, the `repeat` block works similar to a `do {...} while (condition)` loop known from some programming languages. An `each` request block can also be simplified in certain cases. If no merging of different identifiers has to be done, the `separator` and `merge` keys in the `argumentEach` block can be omitted. In addition, the `repeat` block is optional if it is clear that only one request has to be made in each iteration. The notation of `argumentEach` and `repeat` is slightly different to the concept of `argumentRotation` and `continue`. In the case of `call` requests, there is almost always a paging mechanism needed and MiningHub performs numerous iterations to retrieve all data. For `each` requests, on the other hand, it is rather unusual that more than one iteration is done. The functionality for it is available if it has to be used, nonetheless.

In the Stack Overflow mining script, `each` requests are done to retrieve answers and comments to questions, then they cause the retrieval of comments to answers, and then they are used to retrieve user data for questioners, answerers, and commenters to questions and answers. This means that the `users` component retrieves data based on four other components. For such cases, mining scripts allow both `call` and `each` blocks to request more than just one API address. Both of these blocks contain a list rather than a map of elements and thus can be extended by several different types of requests. An example of such a multi request component is `users` in Listing A.4 in the appendix. The `component`, `url`, `argumentEach`, and `store` keys are thereby repeated for every request as this allows for flexibility in cases where these must be handled differently. The use of a proper `primaryKey` value for components that employ more than one `call` or `each` re-

quest is especially important. Overlapping data sets, for example a question and an answer with the same author, could otherwise lead to duplicate entries in the database.

4.2.5 Specification and Validation

Mining scripts are flexible specifications for the behavior of MiningHub. Each user can write his own scripts and upload them to our platform to mine software repositories. As a result, it is very important to check each new mining script prior to any use to avoid illegal states or other problems on the MiningHub platform. In addition, such a check helps researchers who write mining scripts to spot obvious errors. We enable such a foregoing examination by validating mining scripts before they can be stored on MiningHub. For YAML, this is a challenge as there does not seem to be an established and well-documented YAML schema definition. Instead, we use the standard JSON schema and utilize the similarity of YAML to JSON; they are almost fully compatible in the data structures they can store. So we wrote an extensive JSON schema definition for mining scripts (see Listing A.1 in the appendix) as if they were actually written in JSON. Given a mining script written in YAML, we parse it and translate it to JSON. Then we are able to validate the mining script with the JSON schema definition. This translation approach works very well and has the advantage that we can use a standard schema definition with proven tools available. A pure YAML schema would have given us neither.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Schema to define the contents of a Mining Script for MiningHub.",
    "type": "object",
    "properties": {
        "name": { "type": "string" },
        "description": { "type": "string" },
        "apiFormat": { ... },
        "urls": { ... },
        "parameters": { ... },
        "globalArguments": { ... },
        "backoffWhen": { ... },
        "components": { ... }
    },
    "additionalProperties": false,
    "required": [
        "name",
        "description",
        "urls",
        "components"
    ]
}
```

Listing 4.13: Outline of the mining script schema written in JSON.

The outline of the mining script schema written in JSON can be found in Listing 4.13. It starts with a schema version definition and a title, as well as the type of the root element. The

`properties` map then specifies all the properties that may be contained in the root of the script. Apart from `name` and `description`, these have all special type constraints and mostly several subelements. The `...` again denote that the further definition was omitted for brevity. Line 15 disallows the addition of further properties as MiningHub does not use them anyway and it also might be an error. Lines 16 to 21 show a list of required properties, without these a mining script is not valid and cannot be executed. Generally, when adding a mining script to MiningHub which violates any of the rules in the schema definition, the platform prints an error message that exactly tells what error occurred. This helps to spot mistakes and missing definitions. The complete schema that we use for validating mining scripts can be found in Listing A.1 in the appendix.

4.3 Database and Data Structure

MiningHub uses MongoDB to store its data as already noted in the technology evaluation in Section 3.4. MongoDB is a document-oriented database engine that in principle stores its data in JSON. While the internals of MongoDB are different from JSON, most content it exposes is JSON-like and we do not further differentiate the formats in this thesis. MongoDB has the advantage that it can store data with dynamic schemas without having to define a data structure beforehand. This means that arbitrary lists, maps, values, and also nested structures can be stored in MongoDB collections. As a result, MiningHub can request APIs and store the response without having any information about its content. This approach is very flexible and allows mining scripts to request APIs without giving MiningHub insights about what the data actually contains. For requesting or changing data, MongoDB provides a language similar to JavaScript. That approach allows not only to retrieve data but also to process, change, extend, or even store it in different collections. Additionally, there is support for functions, loops and variables. Example uses of these flexible commands can be found in the evaluation (Chapter 5).

MiningHub uses several different MongoDB collections to store its data. The most important ones contain information about stored mining scripts and import jobs that use these scripts. Furthermore, there are also collections to store users, starred items (see Section 4.7), and the search index (see Section 4.5). The collection for mining scripts stores the user ID of the script creator as well as the creation date. It also has fields to define whether a mining script is enabled or disabled and whether it is shared, private, or public. The mining script definition itself is stored in a parsed map with nested key/value pairs instead of a string representation. This means that a mining script only has to be parsed once and is then stored in MongoDB in a nested JSON object. The import collection contains information about data retrieval jobs, such as the mining script to use, the arguments given by the user, visibility settings, and also some more fields that are needed to achieve a proper import as described in Section 4.4.3. Storing the data from mined repositories is done in a dynamic approach as follows. Each import job has got an ID and retrieves several components as defined in the used mining script. To store data for a certain component, MiningHub uses collections named like `import_ID_COMPONENT`. For example, the import with ID *1* stores the component *issues* in a collection called `import_1_issues`. Such collections each contain one item (e.g. one issue) per document. That approach has the advantage that MiningHub can use the MongoDB `_id` key that has a unique constraint. By using the `primaryKey` value defined in mining scripts as `_id`, it ensures that no duplicates are stored. Furthermore, it allows for simple download mechanisms as described Section 4.6. As a result of this approach, each `import_ID_COMPONENT` collection contains a list of items that expose the exact same data structure and nesting as provided by the used API service.

For programmatically querying the database, we use two different libraries. Grails, the framework we used for developing MiningHub, provides a flexible approach of handling data and databases called Grails object-relational mapping (GORM). GORM allows to store and retrieve

data from databases without touching the internals of it in a very simple manner. It is also deeply integrated in Grails and offers many convenience functions as well as domain objects. The latter can be used to define tables and their schema so that GORM can create and manage these tables on its own. The user interface of the MiningHub website uses GORM for showing imports, mining scripts and all other database related content. GORM also has some limitations regarding the handling of MongoDB, though. The biggest problem we faced with GORM was the inability to store data in collections without a predefined name. It appears to be impossible to use tables without a domain object in place. However, as MiningHub stores the retrieved data in collections with dynamic names, GORM cannot be used for the import worker code. Instead, we use the GMongo[4] framework that is a wrapper library around the MongoDB Java driver. It was created for Groovy and resembles the syntax from the MongoDB command line interface as close as possible. It therefore allows to access and store data in arbitrarily named collections using `db.COLLECTION.COMMAND()`. In addition, it makes updating nested structures in MongoDB much easier than the GORM approach of reading, changing, and updating. GMongo instead supports updating certain fields only, for example, using a syntax like `map.subkey1.subkey2`. It also allows to use special key names prefixed with `$` that MongoDB uses to enforce special behavior when updating collections. For example, functions like `$inc` to increment an integer, `$addToSet` to add a new item to a set, or just `$set` to update a value are very practical when dealing with MongoDB. Furthermore, GMongo has the advantage that it does not use sessions like GORM and when a change is stored, it is instantly persisted in the database. This is especially important as MiningHub uses the database to synchronize different import workers as described in Section 4.1.

Using GMongo has given us many advantages over GORM but also caused several problems. A development version of MiningHub running on a server repeatedly crashed with an *OutOfMemoryException: Java heap space* error after running for about two days. Later we found out that GMongo exposes a thread leak when database connections are not explicitly closed after their use. When we changed the code to always close the connections after some time of inactivity, the problem was not gone, though. By more debugging, we found that GMongo also exposes a memory leak when creating and closing connections for each request. Instead, it is necessary to reuse the same connection over all requests and never close that connection. After having implemented these two changes, we never ran into a thread or memory leak again. But it is an important aspect to keep in mind when dealing with GMongo.

4.4 Mining Software Repositories

The core functionality of MiningHub is retrieving data from software repositories. By doing so, it relies on the specifications contained in mining scripts and hides the complexity from the users. But MiningHub itself still has to react properly in case of any unexpected results. This section describes error handling as well as the whole mining process executed by MiningHub.

4.4.1 Error Handling

When mining software repositories with MiningHub or any other tool, a process retrieves data from an external platform over the Internet. By doing so, there are many problems and errors that can occur. The requested platform might be temporarily unavailable, the given URL parameters could be invalid, the returned response might be incomplete, an invalid API key could have been transmitted, or the API limits could have been reached. These are just some examples of many different types of problems that could occur. As we want MiningHub to run repository mining jobs

[4] `https://github.com/poiati/gmongo`

autonomously, it has to be able to handle these and other unexpected behaviors and errors properly. As a result, there are many conditions in which MiningHub does either temporarily pause or completely stop an import job. When a temporary problem occurs while MiningHub tries to retrieve data from a repository, it marks the corresponding import job to back off further requests for a certain amount of time. This causes the import to be paused and MiningHub does not retrieve any data for it anymore. After the back off period has elapsed, the import is activated again and MiningHub continues to retrieve data. Should there be a temporary problem once more, the import is backed off again. This process repeats until the import is able to continue retrieving data from the repository. Reasons causing MiningHub to back off requests for an import job are numerous. It starts at connection problems like no Internet connection on the worker, timeouts, or API providers that are temporarily unavailable. Further reasons include invalid or non-standard server answers (e.g. HTTP status codes, see Section 4.4.2) or when API quota limits are reached. And it is also possible for a mining script creator to manually set back off conditions, for example when a certain key is or is not defined or when a numeric value is lower than 1. The latter could be used when an API returns the number of remaining API quota requests. In most of these cases, the problems are only temporary and after a certain waiting time they do not occur again. Permanent errors have to be handled differently as they will not just go away. Examples of such situations include when the data provider notifies an invalid API key or an uncaught exception occurs. The latter may be caused by problems in MiningHub where, for example, the handling of retrieved data does not work as intended. Another cause for a permanent error is when MiningHub detects an improper mining script. That may occur as although a mining script is valid (see Section 4.2.5), it does not necessarily mean that all its content is meaningful and MiningHub may abort its execution. When a permanent error occurs, MiningHub marks the corresponding import as stopped and no further data is retrieved anymore. In addition, it is possible to manually stop and continue the execution of an import process. The latter can even be done when the import was stopped automatically so that erroneously stopped imports can be continued by the user. Every time a temporary or permanent error occurs, a log message is emitted by MiningHub. These messages are stored separately for every single import job and are only accessible for its creator. Based on these log messages, the user can be informed about any type of problems that occurred and may slow down the data retrieval process. It also tells when a problem regarding the API key occurred so that the import creator realizes that he might have used an invalid key.

4.4.2 Handling HTTP Status Codes

When a client requests a result over the hypertext transfer protocol from a server, an HTTP status code is part of the response. These three digit numbers tell the client whether the request has been answered successfully, or if not, what the problem was. There are APIs that return such status information in the response content itself (e.g. encoded using JSON), but other APIs instead use these HTTP status codes to answer requests or extend their response (e.g. the GitHub API Version 3). Therefore, MiningHub needs to be able to understand the meaning of HTTP status codes. The codes and their meanings have been defined by the World Wide Web Consortium in RFC 2616[5]. While developing MiningHub, we have seen several strange occurrences of status codes, however. For example, we regularly received a `400 Bad Request` that indicates that the request was malformed and has to be modified to work. But minutes later the exact same request worked without any modification whatsoever. Because of such incidences, most error codes cause MiningHub to just back off further requests to the server for some time. We also presume the creator of mining scripts to have tested them thoroughly enough so that the most common errors, such as a wrong API address, would not occur in the first place. So when a problem is indicated by a status code while importing, we assume the remote server to have

[5] `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`

HTTP status code	MiningHub behavior
100 Continue	Back off
101 Switching Protocols	Back off
200 OK	Continue
201 Created	Continue *
202 Accepted	Continue *
203 Non-Authoritative Information	Continue *
204 No Content	Continue *
205 Reset Content	Continue *
206 Partial Content	Continue *
300 Multiple Choices	Back off
301 Moved Permanently	Redirect and continue
302 Found	Redirect and continue
303 See Other	Redirect and continue
304 Not Modified	Back off
305 Use Proxy	Back off
306 (Unused)	Back off
307 Temporary Redirect	Redirect and continue
400 Bad Request	Back off
401 Unauthorized	Stop
402 Payment Required	Back off
403 Forbidden	Stop
404 Not Found	Back off
405 Method Not Allowed	Back off
406 Not Acceptable	Back off
407 Proxy Authentication Required	Back off *
408 Request Timeout	Back off
409 Conflict	Back off
410 Gone	Complete component
411 Length Required	Back off
412 Precondition Failed	Back off
413 Request Entity Too Large	Back off *
414 Request-URI Too Long	Back off
415 Unsupported Media Type	Back off
416 Requested Range Not Satisfiable	Back off *
417 Expectation Failed	Back off
...	
429 Too Many Requests	Back off
500 Internal Server Error	Back off
501 Not Implemented	Back off
502 Bad Gateway	Back off
503 Service Unavailable	Back off
504 Gateway Timeout	Back off
505 HTTP Version Not Supported	Back off

Table 4.1: HTTP status codes as defined in RFC 2616 extended by selected codes as defined in RFC 6585.

temporary problems. As a result, MiningHub backs off further data requests for some time to give the provider a chance to recover. Later, it will retry the request. In what follows, we describe the behavior of MiningHub based on each category of status codes. A complete list of the behavior can be found in Table 4.1. In addition, the behavior on some status codes is marked with a star (*). This indicates that in theory according to code description, there might be APIs that use that code to require special behavior. As an example, `206 Partial Content` could be used to implement a paging mechanism to import a lot of data over more than one request. We have not come across any APIs that use the marked codes in such a way, though. As a result, MiningHub handles these codes only as described in the table. But in the future, that behavior might change should there be APIs that require such a handling. Status codes that were not listed in Table 4.1 cause the MiningHub importer to temporarily back off any new requests.

The informational `1xx` status codes should not be emitted by an API as these make no sense in that context. Should such a status code occur nevertheless, we assume a temporary problem as described above and back off further calls for some time. The `2xx` codes represent a successful request and response; MiningHub continues normally. This category also contains some codes (marked with the *) that could be interpreted differently if required by the API. But so far MiningHub does not handle them differently and just continues processing data. Status codes `3xx` describe redirections. In general, we think that an API should have a fixed specification for a given API version. Some parts of it might be experimental, but the stable behavior should never change. So in theory, redirections should not occur. However, not all API providers agree with that. For example, the specification of GitHub API version 3 explicitly states that HTTP redirections might occur[6]. To avoid problems with redirections, MiningHub follows and handles them as if no redirections occurred. For other codes in this category that do not directly relate to redirections (e.g. `300 Multiple Choices`) MiningHub backs off further requests. `4xx` status codes represent client errors and are either an authorization, API quota, or temporary problem. A further reason for such a code are errors within a mining script. As, again, we presume mining scripts to be tested, most codes cause MiningHub to back off and try again later. There are two exceptions: `401 Unauthorized` are sometimes returned when no or a wrong API key was used; MiningHub stops the import in this case. The second exception is `410 Gone`, which indicates that a resource does not even exist. In such a case, the requested component is directly marked as completely retrieved. Table 4.1 also includes `429 Too Many Requests` as defined in RFC 6585[7] that is sometimes used when the API rate limits are maxed out. MiningHub backs off in these cases. Status codes `5xx` indicate remote server errors and MiningHub backs off to give them time to recover.

When MiningHub is backing off further requests, it always emits a log message so that the user knows what is happening. It also tries to read the `Retry-After` header (if defined) and respects the amount of time that the remote server asked to wait until further requests are made. MiningHub also logs any occurrences of status codes on which it continues but there might be some special behavior needed, as might be the case in `201` to `206`. Status code `410 Gone` are logged as well.

4.4.3 Data Retrieval Process

MiningHub has the two main application components *website* and *worker* as described in Section 4.1. On the website, researchers can create import jobs to retrieve data from repositories and the workers run in the background and do the work of actually retrieving it. There may be only one or several workers that mine repositories concurrently and use the database to synchronize their behavior. Each worker instance regularly checks the database for active import jobs on a

[6] `http://developer.github.com/v3/#http-redirects`
[7] `http://tools.ietf.org/html/rfc6585`

configurable schedule, currently this is set to once every 60 seconds. Considered as active jobs are all imports that are not stopped, not completed, not backing off, and are not currently or were too recently handled by another worker. From the list of active imports, the worker selects one by random and stores the current date to the `lastIterationStarted` key in the chosen import to show other workers that they must not consider it as active anymore. The worker then reads the mining script definition and loops through all its components. For each component, the worker reads the `call` and `each` requests defined in the script and executes them. Each of these requests returns whether or not it already has completed. It is important to know that the two types of requests are handled differently and are described in detail in the paragraphs below. When the execution of all components—that is, an iteration—has completed, the worker updates the database by setting `lastIterationStarted` to *null* and `lastIterationEnded` to the current date. The latter ensures that other workers do not pick up the same import immediately after finishing an iteration to avoid flooding an API provider with requests. After a worker has executed a full iteration with at least one active import, it reschedules its check for active imports to a shorter time frame (currently, this is set to 10 seconds). This increases the speed of the data retrieval from a repository. That whole process is repeated until all imports are either stopped, completed, or otherwise inactive. Imports are stopped or backed off when in one of the components a corresponding condition occurs (see Section 4.4.1). Import jobs are marked as completed when all `call` and `each` requests notify that they retrieved all necessary data. And completed imports are marked as finished and the user can download the mined software repository data. In the next two paragraphs, both `call` and `each` requests are described in more detail.

`call` Requests. Each `call` request individually stores whether or not it has completed already. That is detected, for example, when the paging mechanism has reached the last page or an empty page was returned. In an iteration, all completed requests are directly skipped. When a request has not yet completed, it is handled as follows. First, the API URL with placeholders is retrieved and the actual values (e.g. the API key) are inserted from arguments stored in the database. The proper URL is then requested from a repository and the HTTP status codes are handled according to Section 4.4.2. The HTTP response is then parsed from a string to a nested map or list representation and the part of the response that has to be stored is extracted according to the `store` definition in the mining script. Each retrieved item is then extended by a unique `_id` key according to `primaryKey` and stored in the corresponding collection. After that, the `argumentRotation` block is examined, if given. The `stop.key` value is extracted from the defined `stop.source` and compared to either `stop.value` or `stop.isset`. When the comparison resolves to *true*, the request is marked as complete and skipped in the next iterations. If it is not yet complete, the `continue` block gets examined. Depending on its content, the variable whose value has to be changed for the next iteration is either incremented if it is numeric or it is set to a new string value. Either way, the new argument value is stored in the database and is used to replace the corresponding placeholder in the next iteration. That whole process repeats in each iteration until the import has finished.

`each` Requests. `each` requests differ from `call` requests in that they always depend on a certain other component. For example, they are used to retrieve all the answers for a certain question in the Stack Overflow mining script (see Section 4.2.4). So when executing an `each` request, MiningHub first has to read items from another component from the database. It does so by reading the keys defined in `argumentEach.key` from the collection that stores the `component` the currently executing component is depending on. To avoid repetitive calls for the answers to the same question for example, the items that the current component depends on individually store whether their depending components were already updated. MiningHub only retrieves depending items when they were not yet requested. If `argumentEach.merge` is defined and larger

than 1, MiningHub reads several items and merges their `argumentEach.key` values together separated by `argumentEach.separator`. This is useful when the API allows to retrieve, for example, answers to more than one question in a single request. So MiningHub can merge several question IDs together like *1;2;3;4;5* to retrieve all of their answers in one iteration. Based on the merged (or just a single) `argumentEach.key` values retrieved from the database and other arguments, MiningHub creates the proper URL and requests the specified API. From the result, the HTTP status codes are handled in the same way as `call` requests. Furthermore, the response is parsed and the part that has to be stored is extracted according to `store`. Again, each item is extended by a unique `_id` key and stored in the database. If the current `each` request also defines a `repeat` block, it has to be examined as well. Similar to the behavior in `call` requests, the `repeat.key` value is extracted from the defined `repeat.source` and compared to either `repeat.value` or `repeat.isset`. When the comparison resolves to *true*, the `repeat.argument` is increased by 1, inserted into the API URL and the request repeats. After that, it again checks whether there is another request required and repeats this as long as needed. When all depending items (e.g. answers) of another item (e.g. a question) are retrieved, the latter is marked as complete and skipped in the next iterations. This whole process repeats in every iteration. An `each` request has completed when there are no items from the depending components left to extend. However, as the depending components may retrieve more items in each iteration, `each` requests have to check whether there are more items to retrieve in every iteration as well.

4.5 Searching Retrieved Data

When MiningHub is used to mine software repositories, data can not only be retrieved and stored, but also searched. A precondition for that functionality to be available is that mining scripts explicitly specify which parts of the retrieved data sets have to be searchable. This can be done by creating an `index` list of key names in each component of a mining script. An example of that can be seen in the mining script for GitHub in Listing A.2 in the appendix. The `commits` component indexes the *title* and *body* while for `commitComments` only the *body* is indexed. MiningHub examines the used mining script when importing data and checks whether any keys have to be indexed. When there is an `index` list, MiningHub extracts their values from the API response and uses Apache Lucene[8] functionalities to tokenize the content. The tokenizer performs several optimizations on the words including removal of punctation marks and stop words, stemming, and further simplifications. The resulting tokens are then indexed in the database and linked with the corresponding imports. After importing some data, it is possible to search it on the MiningHub website. The search query is tokenized the same way as before and then compared to the indexed imports. The search page lists all matched items from import components that contain all of the words from the query. Matches are ordered by the number of words, this means that matches with less words are ranked higher than others with more words. Due to a limitation of MongoDB, the search page currently cannot show the total amount of found results. The calculation of `query.count()` takes very long on a larger (i.e. starting from about 100,000 index items) indexes and slows down the whole search experience. Based on the search, a researcher might find a suitable data set for his studies and download the corresponding import.

4.6 Exporting Retrieved Data

When an import job has successfully completed or it was stopped, the retrieved data sets can be downloaded from the MiningHub platform. Each component can be downloaded on its own or

[8] `http://lucene.apache.org`

in a compressed ZIP file containing all the components. The download mechanism reads the data directly from the database and streams it to the user's browser. So there is no need to create a dump file on the server first. Furthermore, there are no modifications made to the data structure that is read from the database. The download format thus is JSON as can be seen in Listing 4.14 that shows an example of a commit component. When downloading, each component collection is exported into its own JSON file that contains a list of items. So the content of the file is exactly the same as in the database collection on the server. The first and last line of each file contain square brackets and in between the items are stored one item per line. There are two reasons for that format: (1) It is valid JSON and (2) after removal of the first and last line (i.e. the square brackets) the file can be imported into a MongoDB collection without file size limits. MongoDB provides command line tools to import and export JSON files to and from collections. To import data, the content of the file is either a JSON array as in Listing 4.14 or, when the total file size is over 16MB, a list of one item per line. The latter is not valid JSON as a whole file, each line itself must be valid, though. To define the behavior in these two cases, the `--jsonArray` switch is used. An example of importing a complete JSON file can be seen on line 1 in Listing 4.15. However, the import capability for MongoDB is only one advantage of the JSON export format from MiningHub. JSON itself is easily parsable and there are many frameworks to handle it in all major programming languages. Therefore, it is possible to parse an exported component into an arbitrary file format with only some lines of code. As a result, the data sets downloaded from MiningHub can be consumed by many applications that work with data. If an application does not support JSON itself, it might be possible to convert the files to the needed data format by using a programming language or an already existing mapper application.

```
1 [
2 { "_id": 123, "commit": { "text": "message1", "user": "name1" } },
3 { "_id": 456, "commit": { "text": "message2", "user": "name2" } }
4 ]
```

Listing 4.14: File format of data sets downloaded from MiningHub.

The MongoDB command line tools allow to import and export JSON. In addition, they also allow to handle files that contain comma separated (CSV) or tabulator separated values (TSV). That means that JSON files downloaded from MiningHub can be converted to CSV and TSV with just two short command line functions. An example that shows how to do this for a file containing the code in Listing 4.14 can be seen in Listing 4.15. Line 1 imports a file called *commits.json* into the collection *commits* within a database called *mininghub*. It is then already available in MongoDB and can be queried or updated. However, to export it to CSV we use the `mongoexport` command with the `--csv` switch that converts the data to CSV and exports all the keys defined in `-fields`. It is mandatory to list all the key names that should be exported because MongoDB is schema-less and each document might have a different structure. So the user has to define which keys have to be included in the resulting CSV file. The output of that command can be seen on line 4 to 6. There are two columns as defined in the command, they contain the `_id` and a string representation of the nested `commit` key. CSV does not support nesting and this is the only way of showing the sub-keys. To properly export the sub-keys as well, the `mongoexport` command has to be written as in line 8 where the fields are defined with the dot syntax (e.g. `commit.text`). The resulting output (line 9 to 11) indeed shows the content properly. However, the first approach can still be useful to export lists to CSV as this would not be possible without the JSON formatting in a CSV

cell. The commands in Listing 4.15 do not export the CSV data to a file but just write it to the console. That is because we omitted the `-o filename.csv` argument for this demonstration.

```
$ mongoimport -d mininghub -c commits --jsonArray commits.json

$ mongoexport -d mininghub -c commits --csv -fields _id,commit
_id,commit
123,"{ ""text"" : ""message1"", ""user"" : ""name1"" }"
456,"{ ""text"" : ""message2"", ""user"" : ""name2"" }"

$ mongoexport -d mininghub -c commits --csv -fields _id,commit.text,commit.user
_id,commit.text,commit.user
123,"message1","name1"
456,"message2","name2"
```

Listing 4.15: Export a JSON file to CSV using MongoDB.

4.7 User Interface and Social Aspects

MiningHub is a web-based platform to help researchers mine software repositories. During its development, we focused on a good usability and easy understanding of all the functionalities that MiningHub provides. We think this is an important attribute to be an attractive social mining and data sharing platform as is our goal. In this section we describe user interfaces of certain pages of MiningHub and describe their functionality. Figure 4.1 shows the MiningHub start page. The dark row on the top is the navigation bar with links to the start page, the import and mining script sections, the projects list as well as to the documentation and user profile. In all further screenshots of the MiningHub user interface, we omit the navigation as it is the same for all of them. The start page gives a quick introduction on what MiningHub is and how it works. And the three boxes on the bottom of the screen guide through the main steps, namely creating or reusing a mining script, run an import, and then download the retrieved data. In addition, there is also the documentation section that explains how to use MiningHub in more detail.

The import section of MiningHub employs a tabbed interface with different pages. The first tab links the section where researchers can create new import jobs to retrieve data from repositories. The *Running Imports* and *Finished Imports* tabs each list a number of imports jobs. The former only contains imports that are active (i.e. not completed or stopped), and the latter lists all finished import jobs. This distinction helps researchers to quickly see whether a data retrieval step is still running or has already finished. Both of these lists show the title as well as the state—either *running*, *stopped*, or *completed*—of an import. Further shown information are the used mining script as well as parameters. The last tab contains search functionality to find items retrieved from software repositories that match a certain query. To retrieve data from a repository, first an import job has to be created in the corresponding section on MiningHub. This is a multi-step process; a repository URL has to be added, then a suitable mining script has to be chosen and required parameters need to be filled in. Figure 4.2 shows the step after selecting a repository URL. MiningHub lists all the mining scripts that can handle the given address and the user has to select one of them. In this case, there is only one mining script available that can handle repositories from GitHub. After a mining script is selected, MiningHub asks to fill in required parameters as can be

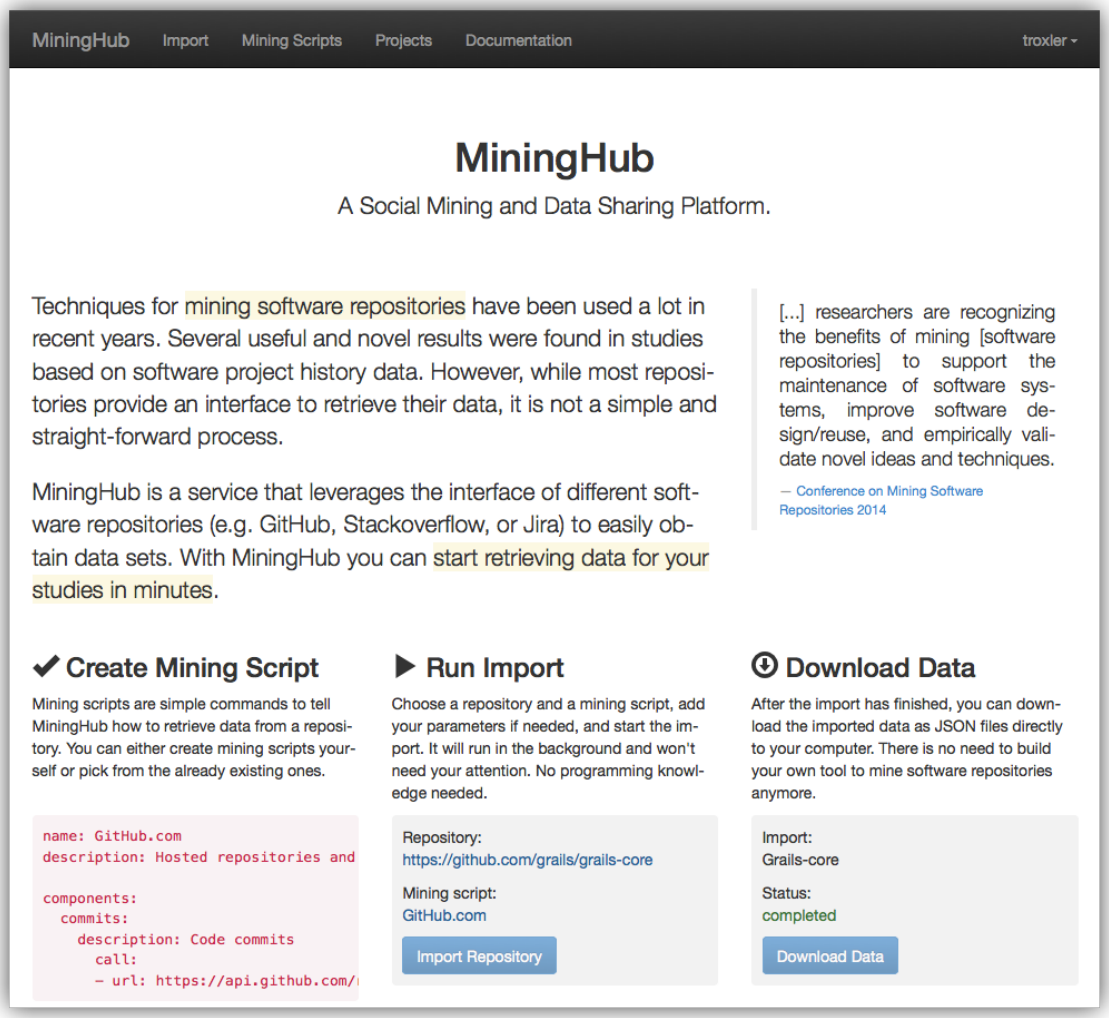

Figure 4.1: The MiningHub start page showing the navigation and a short introduction to MiningHub.

seen in Figure 4.3. An import name is always necessary as it helps to distinguish between different import processes in the views. The mining script for GitHub (see Section 4.2.2) also requires an *accessToken* which authorizes a certain user on the GitHub API. As can be seen in the figure, there is also a note saying that *This entry will be hidden for any user*. API keys should always be defined as `type:password` so that they cannot be viewed by anyone and are kept private. There are also similar notes for other parameter types. After filling in the parameters and starting the import, MiningHub shows a detail view of the created import; an example can be seen in Figure 4.4. That page lists information about an import such as the user that created it, the used mining script, the dates of its start and completion as well as its status. The latter is one of *running* when the import is still retrieving data, *stopped* when it was manually or automatically stopped, or *completed* when the import was successful. A running import can be stopped and continued by its creator if needed. For stopped and completed imports, the retrieved data sets can be downloaded as JSON files (see Section 4.6) and the total duration of the import is being showed. The import page also shows a list of the components that the chosen mining script retrieves from the repository as well

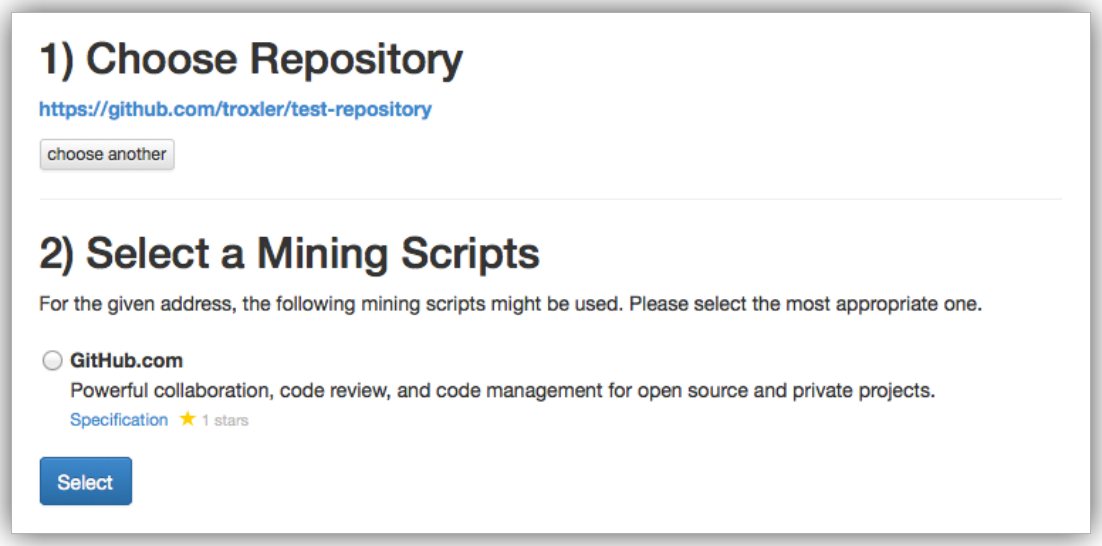

Figure 4.2: Creating an import job on MiningHub: Selecting a Mining Script.

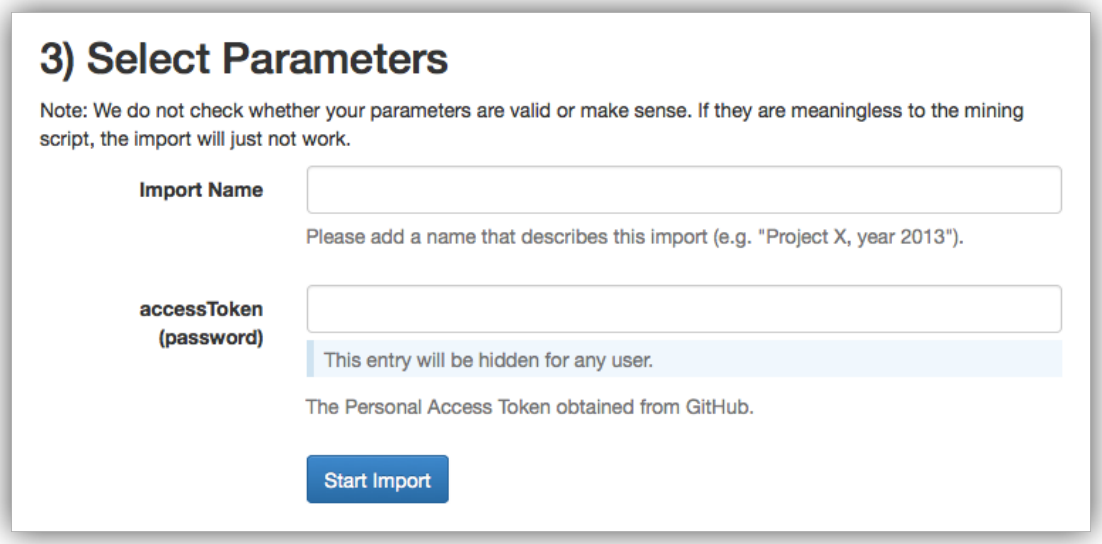

Figure 4.3: Creating an import job on MiningHub: Selecting Parameters.

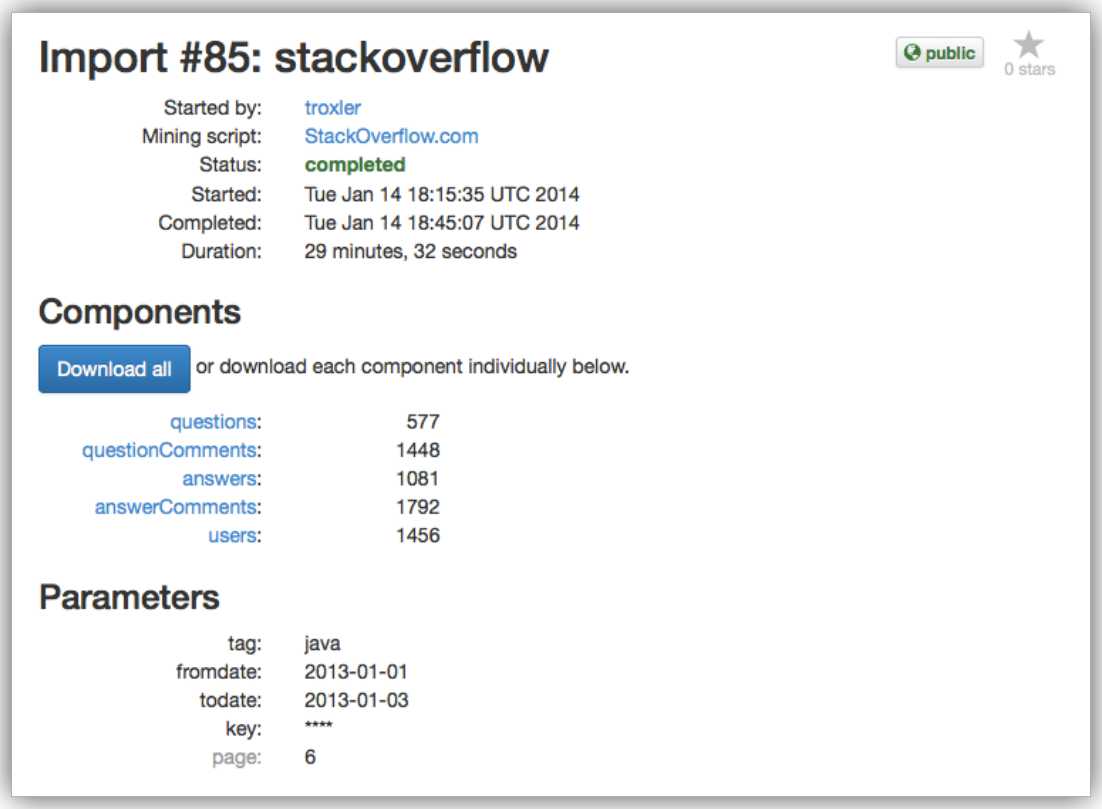

Figure 4.4: Import detail view of MiningHub showing the state of the import, its components and parameters.

as the number of stored items for each of them. Furthermore, the import parameters, the logging system and the comment section are shown (the last two are missing in Figure 4.4). The latter ensures that users can comment on imports and exchange further information. For that purpose, we use Disqus[9] which is a web service that allows to include commenting functionalities on any website. In addition, it is possible to change the visibility settings of imports. Their creators can decide whether they want them to be private so that only they alone can see them, shared so that they and the defined users can see them, or public so that everybody can see them. All users that are allowed to see the import can also star it and the total number of stars is displayed to anybody. That can, for example, be used to bookmark import jobs. There is also a Twitter[10] button so that the users can publish the imported data by means of a link in a tweet.

The mining script section uses a similar tabbed interface as the import section. The first tab shows a page to add new mining scripts. In the second and third tab, there are lists of active and disabled scripts, respectively. When creating a new import, only active scripts can be used because disabled ones do probably not work anymore or are superseded by a new one. The page to add a new mining script just shows an empty text input where scripts can be copied into. Another approach to upload a new script is to drag a local mining script file into the input field and the file gets uploaded instantly. Before being able to store the new mining script, it has to be validated according to the schema (see Section 4.2.5). Invalid script inputs cause MiningHub

[9] http://disqus.com
[10] https://twitter.com

to show an appropriate error message that exactly explains what is wrong. Valid mining scripts are added to the script catalog and can be viewed in the active list. The detail view of mining scripts shows information about its content. This includes the title and description of the script, information about its creator, the supported URL patterns, the components, and the required parameters. The view also shows the complete specification written in YAML. In addition, the view has similar social functions as the import section. The creator of a mining script can change its visibility from private (default) to shared with some others or make it available to the public. Users who can view the script can also tweet about it (only when its visibility is *public*), comment on it, or star it. The comment system allows to discuss potential problems with a mining script or suggest ways to extend it. The starred scripts are highlighted in every view (e.g. when creating an import and having to choose a mining script) so that they can be found among others in a more user-friendly way. The creator of a mining script can also disable it when it was superseded by a newer version, for example.

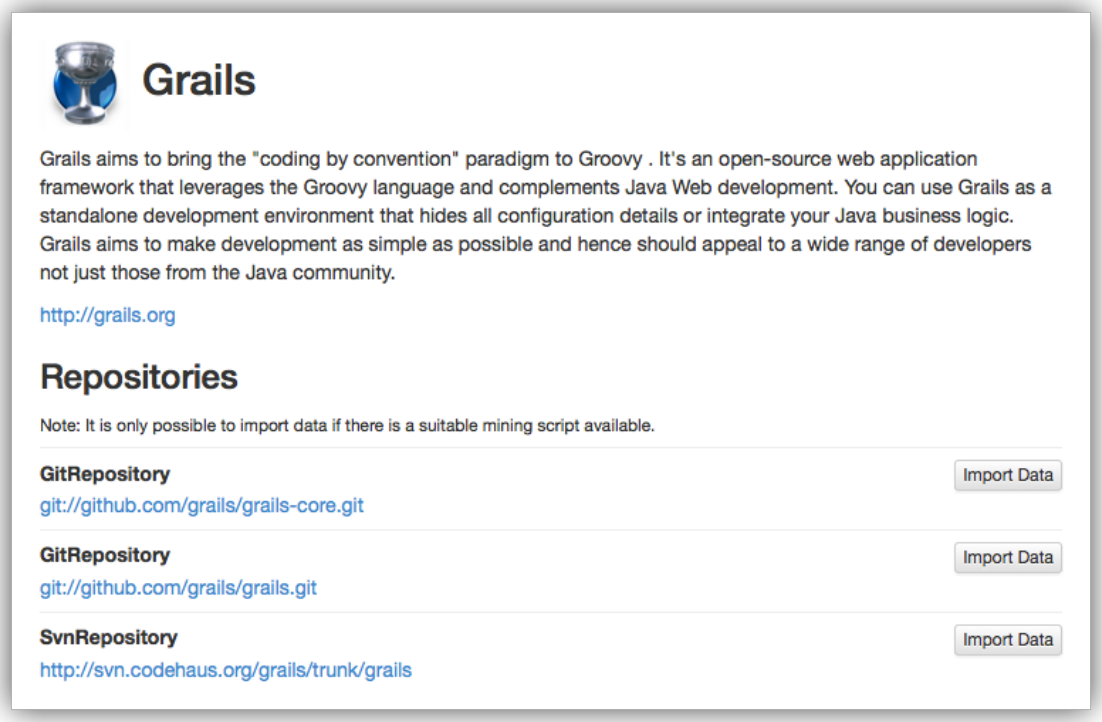

Figure 4.5: The Grails project in the MiningHub software project section.

MiningHub aims to simplify data retrieval from software repositories. As a result, we integrated a software project catalog directly into the MiningHub website. In that project section it is possible to search for a certain project and start an import process on it. To enable that function, we use the project API from Ohloh which provides a large number of software projects including the types of repositories they use. An example of the project view can be seen in Figure 4.5. That page shows a description of the Grails project and lists different repositories that are used. Each repository has a button to start a data import from the corresponding repository URL. But it is important to note that this also requires a mining script that can handle the corresponding URL.

If there is, the data retrieval step can be started for the repository of an arbitrary software project without leaving the MiningHub website.

On the top right of Figure 4.1, there is the user section of MiningHub. Most views are available for any user without logging in but there are some functions that require authentication or authorization. For the login mechanism, we use the GitHub OAuth API[11] so that each user of GitHub can log in on the MiningHub website. We only use the GitHub user name and therefore do not request any private data. As a result, the MiningHub application is only allowed to access a user's followers, public repositories and public gists. There is no way to be granted access to the user name only. After logging in, a user has access to all functionality of MiningHub. He can create new import jobs, add mining scripts, and star imports as well as scripts. There is also a user view where the created or starred imports and scripts of the logged in user or of an arbitrary user can be seen. Commenting and sharing on Twitter is always possible and does not require being logged in. In addition, it is also possible to view public imports, mining scripts as well as users without any authentication.

Figure 4.6: The MiningHub start page on an Apple iPhone 4.

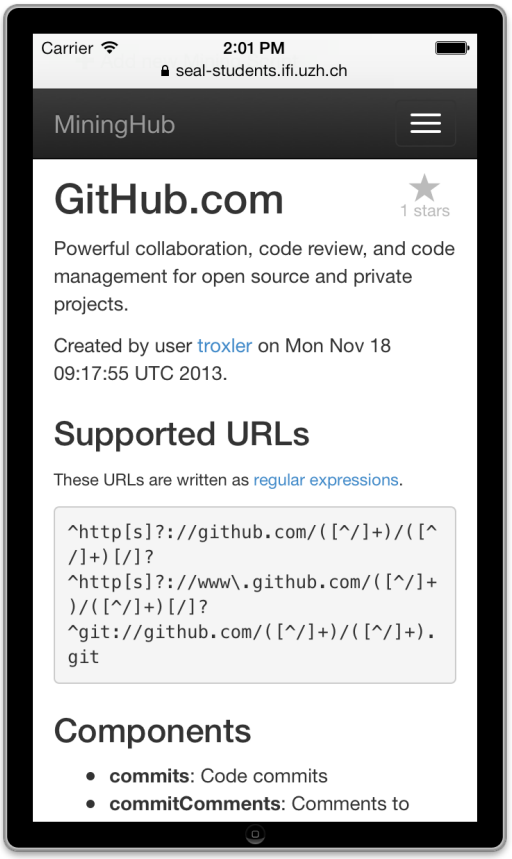

Figure 4.7: The MiningHub mining script view on an Apple iPhone 4.

[11] http://developer.github.com/v3/oauth/

4.8 Mobile Support

During the development of MiningHub, we also focused on a good usability for mobile device users. A large part of that was already achieved just by using Bootstrap (see Section 3.4) which uses a mobile-first approach. Web pages are responsive to different screen sizes by default and optimize the layout of the content depending on the users' devices. In addition, we also further optimized some views and functionalities for mobile devices. As a result, mobile users can use the exact same functionalities of MiningHub as is possible on complete web browsers. An example screenshot of the MiningHub start page on an Apple iPhone 4 can be seen in Figure 4.6. The navigation is automatically collapsed on smaller screens and can be shown by tapping on the button on the top right. That way, the complete navigation is available without covering a large part of the screen. Figure 4.7 depicts another screenshot on an iPhone 4 showing the mining script view. As can be seen, the content breaks at the edge of the screen and no horizontal scrolling is needed. Both Figure 4.6 and Figure 4.7 are screenshots taken from the iOS simulator on Mac OS X 10.8. In addition to the simulator, we also tested MiningHub on different real iOS and Android mobile devices, though.

Chapter 5

Evaluation

In this chapter, we evaluate the MiningHub approach as follows. In the first section we replicate a paper by Bosu [Bos12] in which he described how to reveal the community structure of a software project based on version history, issue tracking, and mailing lists. We used his research plan and performed a basic community structure study on the Groovy programming language project. For the data retrieval we used MiningHub. In the second section, we examine four additional papers and analyze the data that the respective authors used for their studies. For each paper, we then show whether and how MiningHub would be able to retrieve the used data sets. The third section is a roundup of the evaluation chapter.

5.1 Community Structures

In order to show that MiningHub can be used to retrieve data needed for studies, we conducted a replication of a research plan by Bosu [Bos12]. In that paper, he explained an approach to measure the community structure of software projects by calculating an activity score for each member. With that score, the community structure can then be analyzed. Bosu therefore mentioned the hierarchical *onion model* as for example described by Crowston and Howison [CH05]. In the center of the hierarchy there are the core developers who contribute the most part of the code and are responsible for the project itself. The next outer ring is employed by the co-developers who provide patches that are reviewed by core developers. In the next outer level are the active users who do not provide code but file issues and test new releases. All other users are passive, they do not contribute to neither the code nor discussion boards. The exact layers of the onion model are not fixed, for example, Bosu [Bos12] defined them as core members, active developers, peripheral developers, patch submitters, active users, and passive users. This differentiation is obviously more exact than the one of Crowston and Howison [CH05]. Bosu argued that the activity of the developers and users may vary considerably in each type of repository. For example, a developer might provide a lot of code but is rarely active on discussion boards. So when we want to reveal the overall community structure of a project, different aspects of activity have to be considered. Bosu extended the onion model to include such variations and also more types of users. However, he also mentioned that it is still uncertain how to determine the number of users in each hierarchical level. He introduced the activity score to overcome this limitation. That score is calculated for each member of an open source community as defined in (5.1). The total score is summed up from a user's relative activity in version control, bug tracking system, and mailing lists for a single project. That formula also includes weights for each of the three types of repository so that, for example, the importance of code changes could be emphasized by increasing its weight. Bosu, though, noted that the values of the three weighting variables need to be determined empirically. Our replication study did not try to find good weights but instead aimed

to show that research in the field can be simplified with an automatic data retrieval approach. As a result, for our study we simplified the equation for our needs as can be seen in (5.2). First, we defined all the weights to be 1 so that they can be omitted. Secondly, we did not mine a mailing list but instead used the questions and answers on Stack Overflow. So the equation to calculate the activity score for people involved in a certain project is just the sum of the relative activity of the users on GitHub, JIRA, and Stack Overflow.

$$\begin{aligned} \text{Activity Score by Bosu} = {} & \%\text{ of code commits} * \text{code\_commit\_weight} \\ & + \%\text{ of mailing list posts} * \text{mailing\_post\_weight} \\ & + \%\text{ of interaction in bug repository} * \text{bug\_interaction\_weight} \end{aligned} \tag{5.1}$$

$$\begin{aligned} \text{Simplified Activity Score} = {} & \%\text{ of code commits} \\ & + \%\text{ of discussion board posts} \\ & + \%\text{ of interaction in bug repository} \end{aligned} \tag{5.2}$$

For the evaluation of MiningHub, we replicated the research plan of Bosu [Bos12]. In doing so, our main goal was to show that the software repository mining step can be simplified considerably by using MiningHub. This simplification allows to conduct studies based on the mined data sets with less effort, and in the case of Bosu's research plan, with just a few queries. So to find the activity scores of developers and other users, we first had to select an open source project. As in the context of this thesis, we already wrote extensive mining scripts for repositories on GitHub, JIRA bug tracking, and questions on Stack Overflow (see Section 4.2), we wanted to use all three of these mining scripts for this evaluation. Because of that, we used a project that is hosted on GitHub, but does not use the included issue tracker. MiningHub itself was developed in the JVM-based programming language Groovy. That project has large enough dimensions regarding, for example, number of commits, developers, issues, and questions. And their users are active on GitHub, JIRA, and Stack Overflow. Thus, we calculated the overall activity score for all the users involved in the Groovy project.

The activity score (5.2) depends on the observed date range. For example, a core developer of a project who was active up until five years ago could today still have a very large activity score. Therefore, we limited the data that we used to analyze the community structure of the Groovy project to the date range from December 1, 2012 to November 30, 2013. With a full year of data, we calculated a very recent activity score of the users involved in the Groovy community. Furthermore, each of the involved software repositories exposes data about several components as described in Section 4.2. To calculate the activity score we did not need all of them, however. As a result, we only included the most important repository components into our study. For GitHub, we only used the commits as these represent work on the code, which is the main activity on GitHub. Other components were ignored in this replication as they are either not important in this context or are very small regarding the number of data sets they contain. For the JIRA issue tracker, we used both, the issues and issue comments as these are very important contributions to a software project. Regarding Stack Overflow, we took the contributions from questions and answers into account. Comments to either questions or answers were ignored as they are often only very short and as such do not represent a large contribution in a community. In addition, we did not use the user component as both, questions and answers, already include enough information about their respective authors.

Based on these data sets, we executed the research plan as described by Bosu [Bos12]. As a first step, we used MiningHub and the three mining scripts for each GitHub, JIRA, and Stack Overflow to retrieve repository data of the Groovy project. While mining scripts are very flexible and we could have written new scripts to just retrieve the data sets that we actually needed and

only from the specified date interval, we used general-purpose mining scripts instead of updating them. Reason being that we want to show that mining scripts that fetch almost all available data can also be used for more basic tasks. So general-purpose mining scripts can also be used for very specific data needs. The next step in this replication study was the matching of the individual users between all three repositories. This step is needed as the users employ different accounts on the different platforms and they have to be combined first. Based on that matching, we were then able to merge and count the contributions of each user in the different repositories. The notion of *contribution* was used for any activity of a user in one of the components that we looked at in this study. As an example, a contribution might be one commit, one reported issue, or answering one question. After matching different user accounts, we calculated the overall activity score of each user involved in the Groovy project. With that score, we were then able to identify the most active users on the whole project.

5.1.1 Data Retrieval

The Groovy project uses several different repositories for their development. Their version history is hosted on GitHub[1], their bug tracking system is a JIRA installation at Codehaus[2], and a lot of questions about Groovy are asked on Stack Overflow[3]. For these three repositories, we have already created mining scripts (see Section 4.2) that we thus used for the data retrieval step. Starting the imports on MiningHub is straightforward: As described in Section 4.4, we only had to paste the URL of a repository in the MiningHub *Choose Repository* form and select the proposed mining script. In addition to an import name, the GitHub and Stack Overflow mining scripts also require an API key to be filled in. For the Stack Overflow questions, we also added the specified date range. After the parameter dialog, the imports could be started. In total, the creation of the three import jobs on MiningHub, which occurred on Friday, December 13, 2013, took only about five minutes. After creating them consecutively, the imports then ran in parallel on a server using two workers (see Section 4.1 for the notion of *workers*). As the mined software repositories were not that large, all import processes were successfully finished in less than one and a half hours, as can be seen in Table 5.1. Retrieving Groovy related questions from Stack Overflow only took 69 minutes. Data imports from GitHub and JIRA both completed with almost the same running time after approximately 80 minutes. As soon as an import has completed on MiningHub, the retrieved data sets can be download in the JSON file format. Table 5.1 also shows the *file size* which is the uncompressed total size of all component files for each repository. MiningHub retrieved a total of 112.7 MegaBytes (MB) from three repositories. The last row of the table shows the number of components that the used mining scripts obtain from the different repositories (see Section 4.2). And with that, the data retrieval step for the study replication has already been done.

Repository	Mining duration	File size	#Components
GitHub	1 hour, 20 minutes, 10 seconds	35.4 MB	9
JIRA	1 hour, 19 minutes, 45 seconds	61.7 MB	2
Stack Overflow	1 hour, 08 minutes, 59 seconds	15.6 MB	5

Table 5.1: Duration of the mining process for each of the three Groovy repositories as well as the total file size and the number of obtained components.

[1] `https://github.com/groovy/groovy-core`
[2] `https://jira.codehaus.org/browse/GROOVY`
[3] `http://stackoverflow.com/questions/tagged/groovy`

In comparison to the MiningHub approach, Bosu mentioned three different tools he intended to use, each only capable of mining one of the three involved types of repository (version control, bug tracking, mailing lists). For each of them, Bosu also noted some disadvantages like, for example, being limited to a single service provider. He then mentioned that for each of the three tools, to overcome their limitations, he planned to write some scripts. That shows that there is a heterogeneous environment of applications available to mine software repositories. However, researchers might still be forced to add new or extend existing functionality.

5.1.2 Data Preparation

To work with the imported data, we had to import it into a database engine and prepare it so that we could work on it. We also removed some data items that we did not use or need.

Import. After downloading and extracting the compressed JSON files, we had to import them into a database engine. As we have worked with MongoDB during developing MiningHub, and the data sets are already in JSON, it is very straightforward to use MongoDB again. To import JSON files into collections, there is already a command line utility called `mongoimport` that comes with MongoDB. Its interface is called like `mongoimport [parameters] -d DATABASE -c COLLECTION JSON_FILE` and Listing 5.1 shows how we used it to import the downloaded files. As can seen by the `--jsonArray` parameter, we used two different import modes. The reason is a limitation in `mongoimport`. When downloading JSON files from MiningHub, they look like Listing 4.14 (page 36). The first line is an open square bracket (`[`) and the last line is a closing bracket (`]`), everything in between is a list of objects (one object per line). Using the `--jsonArray` parameter, it is possible to directly import a whole file as is into a collection. However, there is a file size limit of 16 MB when using that parameter. As the commits.json and issues.json files were sized 33.1 MB and 61.7 MB, respectively, this did not work for them. To import them anyway, we used the default behavior of `mongoimport`, which reads each line of a file separately and imports that line as a single entry into the specified collection. We had to remove the first and the last line of the two mentioned files and imported the data with the commands on line 5 and 8 of Listing 5.1. After the import step, we then had to prepare the data sets for our use case.

```
1 mongoimport --jsonArray -d evaluation -c so_answers ID_answers.json
2 mongoimport --jsonArray -d evaluation -c so_questions ID_questions.json
3
4 # delete first and last line before this step
5 mongoimport -d evaluation -c gh_commits ID_commits.json
6
7 # delete first and last line before this step
8 mongoimport -d evaluation -c jr_issues ID_issues.json
```

Listing 5.1: Import JSON files into MongoDB collections.

Date Format. We were only working on data from a twelve month time frame. While the mining script for Stack Overflow already allows to specify a time frame, those for GitHub and JIRA do not. Because of that, we had to remove items from the mined data which had a date either before our date range, or after that range. After importing the data sets into MongoDB, the dates

are stored as plain strings with which we cannot make comparisons. So first, we had to transform the string dates to real date objects so that they were comparable. Listing 5.2 shows the approach that we used to change the date format for all the involved collections. The listing actually shows JavaScript code that runs on the MongoDB shell. So to adjust the date format, we let MongoDB loop through a collection, changed the string dates to date objects, and wrote them back to the database. This step was then repeated for all the collections that contained entries which lay outside of our date range.

```
1 db.COLLECTIONNAME.find().forEach(function(doc) {
2     doc.PATH.TO.DATE = new Date(doc.PATH.TO.DATE);
3     db.COLLECTIONNAME.save(doc);
4 });
```

Listing 5.2: Change string dates to real date objects.

Remove Unneeded Data. We have limited the data that we used to the twelve month range from December 1, 2012 to November 30, 2013. To also reflect that on our data set, we had to remove commits and issues which were created before or after that period. The components of Stack Overflow in contrast were already in that date range because we specified it accordingly when creating the import job on MiningHub. So Listing 5.3 only removes data mined from GitHub and JIRA. To understand said listing properly, it is important to know that `new Date()` using three parameters as we did has a special notion of *month*. Namely, months are zero-indexed as opposed to the one-index we are used to. That means that `new Date(2013,10,30)` is November 30, 2013, and not October 30. In addition to the removal of entries that are not in the date range, we also deleted a total number of four commits which did not have a `committer` and/or not have an `author` value, but just a `null` value. Such `null` values are most probably caused by GitHub users that deleted their account.

```
1 db.gh_commits.remove({"commit.author.date": {$gt: new Date(2013,10,30)}});
2 db.gh_commits.remove({"commit.author.date": {$lt: new Date(2012,11,1)}});
3 db.gh_commits.remove({"commit.committer.date": {$gt: new Date(2013,10,30)}});
4 db.gh_commits.remove({"commit.committer.date": {$lt: new Date(2012,11,1)}});
5 db.jr_issues.remove({"fields.created": {$gt: new Date(2013,10,30)}});
6 db.jr_issues.remove({"fields.created": {$lt: new Date(2012,11,1)}});
```

Listing 5.3: Remove entries from collections that are not in the specified date range.

Statistics. After having performed the cleaning process as described above, we present some statistics about the mined data; in each case before and after the data cleaning. The MiningHub import overview shows the number of mined items for every component so that in most cases, we could just take these numbers. For components that contained more entries than specified by the date range and thus were reduced, we used the MongoDB function `db.COLLECTIONNAME.count()` to count the number of documents in a collection. For each of the three tables mentioned in this paragraph, the columns are the same: (1) the names of the components, (2) the

number of items per component before removing any data, and (3) the number of items per component after cleaning the data set.

Component	Number of items	After cleaning
Commits	9,553	975
Commit comments	41	-
Issue comments	652	-
Branches	13	-
Tags	117	-
Forks	185	-

Table 5.2: Number of items per component of the Groovy project on GitHub.

For GitHub, Table 5.2 shows the number of items that were retrieved by MiningHub. The number of issues, milestones, and releases are all zero and are therefore omitted. The component *issue comments* can be non-zero even when the project does not use issue tracking system of GitHub. Reason being that comments to pull requests are also counted as issue comments. By removing all commits that were pushed before December, 2012 or after November, 2013, the number was decreased to 975 commits. All other components mined by the used mining script are omitted for this study which is denoted by a dash (-) in the third column of the table.

```
1 db.jr_issues.aggregate([{
2     $group: {
3         _id: null,
4         total: {$sum: "$fields.comment.total"}
5     }
6 }]);
```

Listing 5.4: Count total number of nested issue comments.

For JIRA, the way of counting the number of issue comments was different. The corresponding mining scripts retrieves only the components *project* and *issues*. Comments themselves are not in their own component but nested in the issues because the JIRA API returns issues including all of their comments. So to count the nested comments, we used the MongoDB query in Listing 5.4. This query reads the number of comments stored in the field `fields.comment.total` of each issue and then aggregates them by summing up the individual values. The total number of issues and issue comments can be found in Table 5.3. In total, we worked with 622 issues and 1,314 issue comments in our study.

The data we mined from Stack Overflow did not need any removals as the date range was already obeyed in the retrieval process. The total number of items can be found in Table 5.4 that also depicts that we used only questions and answers for our study. It must be pointed out that while the user component itself was omitted, we still used the user data for the matching process. However, the questions and answers themselves already included all the user information that

we relied on. In total, we used 2,196 questions and 2,683 answers for calculating the activity score of each individual user.

Component	Number of items	After cleaning
Issues	6,220	622
Issue comments	18,891	1,314

Table 5.3: Number of items per component of the Groovy project on the JIRA instance.

Component	Number of items	After cleaning
Questions	2,196	2,196
Question comments	2,778	-
Answers	2,683	2,683
Answer comments	3,687	-
Users	2,322	-

Table 5.4: Number of items per component for questions tagged with *Groovy* on Stack Overflow.

5.1.3 Community Structure per Repository

Before we calculated the activity score over all three repositories that are used by the Groovy project, we examined each of them on its own. This way, we found the top contributors of each of the repositories, namely version control at GitHub, issue tracking with JIRA, and help board on Stack Overflow.

GitHub. To change code on GitHub, the developers commit their changes to the version control system named Git[4]. In the observed date range, there are a total of 975 such commits as can be found in Table 5.2. However, there is one important distinction regarding commits: A single commit can be issued by one or *two* different developers. The intuition behind this is that the author (i.e. the person who changed the code) is not necessarily the committer (i.e. the person who added the change to the project). An example of such a situation are pull requests. When a developer who does not have write access to a project wants to change the code anyway, he can fork the project, make his changes and issue a pull request. As a result, that developer is the author of the commit. But before the change is actually added to the project, someone with write privileges has to review the change and commit it. So the reviewer is the committer. We reflected this distinction when calculating the activity score. A developer committing his own changes was counted as one contribution. On the contrary, a commit that has different author and committer was counted as one contribution per developer as both coding and reviewing changes takes a certain amount of time. So to count the number of contributions for each developer, we looped

[4] `http://git-scm.com`

through all the commits and added the users to a new collection as can be seen at line 1 to 15 in Listing 5.5. Thereby we stored a new document for each contribution and made the distinction between one and two person commits. We did that by simply storing a `type` key whose value is either *author*, *committer*, or *both*. Incorporating this distinction led to a total number of 996 contributions compared to 975 commits to the version history of Groovy.

```
1  db.gh_commits.find().forEach(function(doc) {
2      var user = {}
3      if (doc.committer.login == doc.author.login) {
4          user.user = doc.committer.login
5          user.type = "both"
6          db.gh_commits_users.insert(user);
7      } else {
8          user.user = doc.author.login
9          user.type = "author"
10         db.gh_commits_users.insert(user);
11         user.user = doc.committer.login
12         user.type = "committer"
13         db.gh_commits_users.insert(user);
14     }
15 });
16
17 db.gh_commits_users.aggregate({
18     $group: {
19         _id: "$user",
20         sum: {$sum: 1}
21     }
22 }, {
23     $sort: {sum: -1}
24 });
```

Listing 5.5: Create a new collection for all users involved in commits, and then perform an aggregation to count the number of contributions per user.

After having created the new commit contributors collection, we wanted to aggregate its data. We did so by use of the `aggregate` function in MongoDB as can be seen at line 17 to 24 in Listing 5.5. This query calculates the total number of contributions for each developer and sorts it accordingly. To also calculate the contributions where a person was committer or author only, the query can be extended by adding the match parameter `$match:{type:"author"}` to the `aggregate()` function. The result of this aggregation can be found in Table 5.5 that shows the top ten code contributors of the Groovy project. It is obvious that, at least in the top ten, commits with different author and committer are not numerous. However, we still think that this distinction was and is important and should be done especially when a larger date range is taken into account for the study. Irrespective of that, the table shows that in the specified date range, there are not more than about eight developers (depending on the definition) that repeatedly make changes to the code. One can also see that about five committers appear to be core developers (again, depending on the definition).

User	Author and committer	Author only	Committer only	Total
paulk-asert	217	-	8	225
melix	206	1	10	217
PascalSchumacher	196	3	1	200
blackdrag	105	-	1	106
tkruse	93	-	-	93
andresteingress	38	6	-	44
glaforge	30	-	-	30
timyates	23	-	-	23
aalmiray	2	5	-	7
bura	6	1	-	7

Table 5.5: Top ten committers to the Groovy GitHub repository.

JIRA. When a bug description or a feature request is filed in an issue tracking system, many times there are also written comments with questions, more information, or attachments to these issues. As the comments often help resolving issues, we counted both the creation of issues and the writing of comments as one contribution each. Therefore, the total contribution by a user is the sum of his created issues and his written comments. To calculate the top contributors to the Groovy issue tracking system, we used queries similar to the ones in Listing 5.5. Again, we created a new collection that for each contribution contained the user name and the type which is either *reporter* or *commenter*. To do that, we looped through all issues, added the reporters, and then iterated through the nested issue comments and also added the commenters. After creating the new collection, finding the top ten contributors was straightforward and very similar to the second query in Listing 5.5. The results of this aggregation can be found in Table 5.6. As before, the number of contributions in the top ten were diverging, from lowest of 29 to the highest of 295. That is a large difference, especially when also considering the total number of contributors which was 243 people. When comparing Table 5.5 and Table 5.6, we could already see by looking at the user names that at least six developers were in the top ten of code committers and also in the top ten of issue tracking activity. In addition, the top four users were the same in both repositories, even when they were not in the same order. So we expected all of them to also be at least in the top ten of the overall activity score.

Stack Overflow. For Stack Overflow we only took questions and answers into consideration for simplicity. We omitted comments to questions or answers. When observing the top contributors to questions and answers tagged as *Groovy*, we found that there is no overlap between the top ten questioners and top ten answerers. The reason for this is that Groovy beginners are asking a lot of questions at the beginning of the learning process. On the other side, experienced Groovy developers can answer many questions based on their knowledge but themselves do not ask many questions as they know how to help themselves. The data aggregation in this case was very straightforward as we could directly use the `so_questions` collection we already had and group by the key `owner.display_name`. Same thing worked for the `so_answers` collection. We list the top five questioners in Table 5.7 and the top five answerers in Table 5.8. It is remarkable how large the difference between the top contributor (user *tim_yates*) and the second most active

User	Reporter	Commenter	Total
paulk	37	258	295
melix	54	166	220
blackdrag	16	191	207
pschumacher	32	140	172
asteingress	6	47	53
lhotari	11	33	44
guillaume	6	32	38
tim_yates	11	22	33
mxm	17	13	30
pniederw	11	18	29

Table 5.6: Top ten contributors to the Groovy bug tracking system (JIRA).

User	Questions
Anthony	16
Himanshu Yadav	16
Clinton	15
user1170330	15
birdy	13

Table 5.7: Top five questioners of questions tagged with *Groovy* on Stack Overflow.

User	Answers
tim_yates	418
dmahapatro	173
Will P	104
Peter Niederwieser	75
ataylor	61

Table 5.8: Top five answerers of questions tagged with *Groovy* on Stack Overflow.

contributor (user *dmahapatro*) was. Furthermore, it is interesting to note that just based on the user names, *tim_yates* seems to be the only contributor that was in the top ten of each GitHub and JIRA as well as in the top five on Stack Overflow. As a result, said developer is definitely in the top ten users ranked according to their activity score in all three repositories.

5.1.4 Matching Users

The activity score by Bosu included the activity of developers on each of code version control, issue tracking, and discussion boards. To properly calculate that score, the users had to be matched between the three different platforms first. This is obviously a requirement for the activity score calculation, but a non-trivial task as the people often use several varying user names which might be similar or completely different. To still being able to match at least some of the users, we used their email addresses to uniquely identify them. This is possible as both GitHub and JIRA include their user's email addresses in the API responses, Stack Overflow at least provides the message digest 5 (MD5) hash of the addresses. As a result, we could match the users even when their names are completely different but they at least used the same email address on two or more platforms.

In addition to the email address, we also matched users that have the exact same user name on different platforms or a name which is very similar. For the latter, we used the Levenshtein distance measurement [Lev66]. Before we go ahead with the actual matching description, we show some basic statistics that can be found in Table 5.9. This table gives a résumé over the number of contributions and contributors for all three of the mined software repositories. The number of contributions was calculated by using a `count()` query on a collection. The number of contributors in contrast, was calculated on the temporary collections we created in Section 5.1.3 by a query using `distinct("name").length`, which returns each user name only once and then counts them. The number of contributions was later used to calculate the relative activity for a user on each repository. But also the numbers of contributors are interesting. They show that few people are working on the code of Groovy, more people are involved in reporting and commenting bugs or feature requests, and a lot of people are asking or answering questions regarding the use of Groovy. We believe that this kind of distribution between different types of repositories are similar for many open source software projects.

Repository	Number of contributions	Number of contributors
GitHub	996	32
JIRA	1,936	243
Stack Overflow	4,879	2,000

Table 5.9: Total number of contributions and contributors for each repository.

In order to match different user accounts, we first created a collection of accounts that contributed to Groovy on Stack Overflow as the number of users in this community was the largest of the analyzed Groovy repositories. The query we used for that can be found in Listing 5.6. As can be seen in that query, we tried to extract parts of the `profile_image` field. This is because most users of Stack Overflow that are represented in our data set have an avatar that is linked to Gravatar[5] with a URL that contains the hashed email address (e.g. `https://gravatar.com/avatar/HASH`). However, some users have their avatars linked to Imgur[6] with a URL that only contains a short ID instead of a hash (e.g. `http://i.stack.imgur.com/ID.jpg`). We could not compare email address hashes to this ID. There were also some users that did not have any profile image at all. So to prepare user matching, we tried to extract the hash that is used for Gravatar URLs. Therefore, we had to check whether the `profile_image` was defined at all. If it was, we tried to extract the hashed email address from the image URL using a regular expression. Then we inserted the `display_name` into the collection as document key so that no duplicate values occured. We also directly added the number of contributions of a user on Stack Overflow. This was done with the MongoDB `upsert` functionality that either inserted a new entry with a contribution of one or updated a matching entry and increased the number of contributions by one. As the query in Listing 5.6 only ran on Stack Overflow questions, we had to run the same query against the answers collection as well. After the execution of these two queries, we had a new collection containing a list of unique contributing Stack Overflow users including their display name (`_id`), an MD5 hash of their email address (`so_hash`, if given), and the number of their help board contributions (`so_contributions`).

After having created a user collection based on Stack Overflow accounts, we also added users from the JIRA data set. But first, we created another new collection which contained all unique

[5] Avatar hosting service: `http://gravatar.com`
[6] Image hosting service: `http://imgur.com`

```
1  db.so_questions.find().forEach(function(doc) {
2      var match = doc.owner.profile_image
3          ? doc.owner.profile_image.match(/.*avatar\/([^\?]*)/)
4          : null;
5      db.users.update(
6          {_id: doc.owner.display_name},
7          {
8              $setOnInsert: {
9                  so_hash: match ? match[1] : null,
10             },
11             $inc: {so_contributions: 1}
12         },
13         {upsert: true}
14     );
15 });
```

Listing 5.6: Query to create a new collection of Stack Overflow users.

accounts that contributed to issues. The corresponding query can be found in Listing 5.7. It iterated through all issues and added the reporter to the collection, then it did the same for every issue comment. To avoid duplicate users, we again used the MongoDB `_id` which must be unique in every collection. By iterating through the users, we also added their email addresses and summed up their contributions to the Groovy project. We did so by again using the `upsert` functionality provided by MongoDB. After executing that query to generate a list of unique JIRA users, we tried to match them with the collection of Stack Overflow accounts. Listing 5.8 shows the query that we were using for that. Users matched when they have got the same nick name or the email address from the JIRA account matches the MD5 hash (see `hex_md5()` on line 4) of the address stored at Stack Overflow. Using this approach, we were able to combine 39 user accounts. When we tried to match more accounts using the Levenshtein distance measurement [Lev66], we were noticing that there were already a lot of false positives using the smallest possible distance of one. By manually looking through the proposed duplicate users, we were also unable to find true obvious matches. Consequentially, we did not use the Levenshtein distance measurement anymore in further matching processes. That approach might help to find possible matches, but the rate of false positives can be very large. In our opinion, that measurement is not accurate enough to match user names which are often short and therefore also very similar. So for this study we continued matching users by identical display name or email address only. The matches that we found, were then merged with their respective account and extended by name, email address, and number of contributions on JIRA. After that, we added all other JIRA users that did not match anything to the `users` collection as well. The query for this can be found in Listing 5.9. It simply looped through all user names from JIRA and checked whether the `users` collection already contains such an account. If it did not, it was inserted with all the information we needed later for calculating the activity score.

The matching process for users from GitHub was very similar. There was one very important difference, though: When a user commits a change using Git, his account details are attached to the commit. As these account details are taken from a local configuration file, they do not necessarily reflect the details stored on GitHub about that user. That also means that the local configuration can be changed effortlessly. As a result, there were commits from the same user but with different email addresses. Consequentially, we had more user data available when a

```
db.jr_issues.find().forEach(function(doc) {
    db.jr_users.update(
        {_id: doc.fields.reporter.name},
        {
            $setOnInsert:{
                displayname:doc.fields.reporter.displayName,
                email:doc.fields.reporter.emailAddress
            },
            $inc: {contributions:1}
        },
        {upsert: true}
    );
    doc.fields.comment.comments.forEach(function(comment) {
        db.jr_users.update(
            {_id: comment.author.name},
            {
                $setOnInsert:{
                    displayname:comment.author.displayName,
                    email:comment.author.emailAddress
                },
                $inc: {contributions:1}
            },
            {upsert: true}
        );
    });
});
```

Listing 5.7: Create collection of unique JIRA accounts.

developer changed his local email address. So to utilize that information, we had to collect all the email addresses used in commits for each user. We again created a new collection of unique users similar to Listing 5.7. However, instead of `$setOnInsert` we used `$addToSet` for the email addresses. This way we created a list of unique addresses for each user account. That procedure was repeated on every commit for every author and committer. Thereby we also summed up the contributions where we counted a commit with identical author and committer as one contribution only. The exact query we used for creating that collection can be found in Listing B.1 in the appendix. As after that we had a list of unique accounts from GitHub users involved in the Groovy project, we could match them to the `users` collection. As before, this was done by matching email addresses and user names as in Listing 5.8. The difference that time was that we also looped through all the email addresses of each users and tried to match them. Found matches were then extended by the user account from GitHub including the number of contributions. The complete query that performed this process can be found in Listing B.2 in the appendix. By comparing the users' names and their email addresses, a total of 29 GitHub accounts were merged with entries in the `users` collection. As final step, we repeated the query in Listing 5.9 to add the remaining users. But that time on the `gh_users` collection to ensure that there were all unmatched accounts stored in `users` as well.

After this procedure, we had finished matching the user accounts of the three different repositories GitHub, JIRA, and Stack Overflow. In that matching process, a total of 2,275 user accounts

```
db.jr_users.find().forEach(function(doc) {
    db.users.update({$or: [
        {_id:doc._id},
        {so_hash: hex_md5(doc.email)}
    ]},
    {
        $set:{
            jr_name:doc._id,
            jr_email:doc.email,
            jr_contributions:doc.contributions
        }
    });
});
```

Listing 5.8: Match JIRA accounts to Stack Overflow users.

```
db.jr_users.find().forEach(function(doc) {
    if (db.users.find({jr_name:doc._id}).count() == 0) {
        db.users.insert({
            jr_name:doc._id,
            jr_email:doc.emails,
            jr_contributions:doc.contributions
        });
    }
});
```

Listing 5.9: Insert all JIRA accounts that were not matched into the user collection.

from three different sources were merged to 2,218 accounts. While this is not a large reduction, keep in mind that our matching process was very basic and only merged obvious duplicate accounts. And still, almost all accounts from GitHub could be matched with the ones from JIRA and/or Stack Overflow. The process as is could definitely be improved for more advanced studies, though. More elaborate matching algorithms as well as manual work would certainly increase the matching rate if needed.

5.1.5 Overall Community Structure

After having performed the account matching, we had a collection containing a list of 2,218 unique users including their user names and number of contributions from three different repositories. When looking at the top contributors by the total number of activities, we noticed that there were two obvious duplicates which were not found by our automatic user matching approach, though. We combined them manually. We also found a user that apparently had two user accounts on JIRA. We combined that user and summed up his contributions manually. After these changes, the total number of users was 2,215. Overall, we were able to merge 31 of 32 GitHub users to other accounts. So for the GitHub data sets we used, even our naive matching approach worked very good. After having cleaned parts of the user data, we wanted to calculate the overall activity score for each individual account. For that, we used the equation defined in (5.2), which

calculates the sum of the relative activity on each repository on its own. As we already had the absolute number of contributions for each user, we just had to compute the relative activity and then sum them up. We used the query in Listing 5.10 for this calculation. First, the relative activity for each user on all the repositories was computed and stored in the `activity` map. The numerical divisors on line 3-5 represent the total number of contributions in the corresponding repository as shown in Table 5.9. Then, the individual scores were summed up to the overall activity score and that value was stored to the database.

```
1 db.users.find().forEach(function(doc) {
2     doc.activity = {
3         gh: (doc.gh_contributions ? doc.gh_contributions /  996 : 0),
4         jr: (doc.jr_contributions ? doc.jr_contributions / 1936 : 0),
5         so: (doc.so_contributions ? doc.so_contributions / 4879 : 0)
6     };
7     doc.activity.overall = doc.activity.gh + doc.activity.jr + doc.activity.so;
8     db.users.save(doc);
9 });
```

Listing 5.10: Calculate the overall activity score for each user account.

After that processing, all data that we needed was stored in the `users` collection and we could analyze it. As already mentioned, we observed the overall top contributors to the Groovy project on GitHub, JIRA, and Stack Overflow. The only thing left to do was to query users from the collection and sort them after the overall activity score using `sort("activity.overall":-1)`. The top 15 contributors found by that query can be seen in Table 5.10 that, in its main columns, shows the results for GitHub, JIRA, Stack Overflow and then the overall activity score. The rows each represent a single user that was merged from different accounts (if a match was found). Accounts that were not matched between the different repositories (i.e. we did not find the matching account or it does not exist) are denoted by a question mark (*?*). The columns below repository types are separated between user name as well as absolute (#) and relative (%) contribution on that repository. The fourth column, the overall activity score, is the sum of the individual relative contribution scores. On JIRA, the accounts *asteingress* and *andre.steingress* appear to belong to the same person and as such were consolidated to a single user. As a result, we have found the overall most active users involved in the Groovy project.

5.1.6 Discussion

After we calculated the top contributors to the Groovy project, we present a short discussion of the results and threats to their validity. And more importantly, we also compare the data retrieval step to the approach proposed by Bosu [Bos12] in more depth.

Results. Table 5.10 shows the overall top contributors of the Groovy project. The largest activity score of about 0.378 was achieved by user *paulk-asert*. He had large scores of 0.226 and 0.152 on GitHub and JIRA, respectively. On Stack Overflow he was not very active in the observed date range, though. Overall, a large portion of the top contributing people on the individual repositories (see Tables 5.5, 5.6, and 5.8) were also represented in the top 15 of the summed activity scores. Namely, nine of the top ten contributors on GitHub, all of the top ten from JIRA, and four of the top five users from Stack Overflow. Furthermore, it is important to note that while we did

GitHub			JIRA			Stack Overflow			Overall
User	Contributions		User	Contributions		User	Contributions		Activity score
	#	%		#	%		#	%	
paulk-asert	225	0.22590	paulk	295	0.15238	Paul King	1	0.00020	0.37848
melix	217	0.21787	melix	220	0.11364	melix	5	0.00102	0.33253
PascalSchumacher	200	0.20080	pschumacher	172	0.08884	?			0.28965
blackdrag	106	0.10643	blackdrag	207	0.10692	blackdrag	17	0.00348	0.21683
timyates	23	0.02309	tim_yates	33	0.01705	tim_yates	418	0.08567	0.12581
tkruse	93	0.09337	tkruse	24	0.01240	?			0.10577
andresteingress	44	0.04418	asteingress andre.steingress	68	0.03512	Andre Steingress	2	0.00041	0.07971
glaforge	30	0.03012	guillaume	38	0.01963	?			0.04975
?			?			dmahapatro	173	0.03546	0.03546
pniederw	2	0.00201	pniederw	29	0.01498	Peter Niederwieser	75	0.01537	0.03236
?			lhotari	44	0.02273	FlareCoder	1	0.00020	0.02293
?			?			Will P	104	0.02132	0.02132
?			mxm	30	0.01550	?			0.01550
aalmiray	7	0.00703	aalmiray	10	0.00517	aalmiray	12	0.00246	0.01465
?			russel	28	0.01446	?			0.01446

Table 5.10: Top 15 contributors to the Groovy project on GitHub, JIRA, and Stack Overflow.

not use any weighting mechanism as in (5.1), contributions to the code were worth more to the overall score than activity on the other repositories. In our case, this was caused by the different absolute numbers of contributions (see Table 5.9). For example, one commit on GitHub added a larger contribution to the overall activity score then a Stack Overflow answer. Reason being that the relative activity is higher when the total number of contributions is lower. As a conclusion of Table 5.10 and the calculated activity scores, one can say that in the case of the Groovy project most users that were very active on one repository were also active on the others. This is especially true for the code repository and issue tracking, but not that much for the help board at Stack Overflow. Our study using a naive user matching approach indicates that especially the lead developers of Groovy are not very active on Stack Overflow.

Mining Software Repositories. When calculating the overall activity scores of members of the Groovy community, we followed the research plan of Bosu [Bos12]. One important difference, though, was the way we retrieved data from the different repositories. In our study, we used our novel approach with mining scripts that were executed on the MiningHub platform as described in Section 5.1.1 (page 47). On the other hand, Bosu explained his approach of mining software repositories in his research plan relatively extensively. He started by presenting CVSAnalY[7] that is a tool to mine source code repository logs and store the extracted information into a database. In spite of its name, CVSAnalY is also capable of mining version control systems other than just CVS. For mining bug trackers, Bosu recommended using Bicho[8]. Bicho is a command line tool that parses the issues in a corresponding repository and stores them locally. Back when Bosu wrote his research plan, Bicho was able to handle three different types of bug tracking systems. For mining mailing lists, Bosu planned to use a tool called Mailing List Stats[9] (or short, MLStats). Based on a given mailing list, MLStats can download the emails and then create a database by extracting the information contained in these discussions. Bosu also noted that each of these tools has limitations, for example they can only mine one or a very limited number of repositories. To overcome these limitations and to extend the functionality of the tools, he planned to write scripts for each of them to be able to retrieve the data that he wanted. While in the meantime all of these tools might have been improved gradually, some limitations still apply. Before being able to start to mine software repositories, each of these tools has to be downloaded and installed. For each, the documentation (if available) has to be read and it is required to understand how the tool actually works. They also have to be configured according to one's needs and it is necessary to make sure that they run properly on the used computer. In the case of Bosu's research plan, all the mentioned tools are at least capable of writing their data into a relational database so the data format is probably similar. However, when using other tools, there might also be varying data formats a researcher has to deal with. As a result, before actually being able to start with the study work, researchers have to handle (and maybe even extend) different tools that might be complicated and produce data sets in different file formats. In contrast to that, getting started with MiningHub is very straightforward. There is no need to install any tool as MiningHub works as a web service in the browser. There are also no configuration settings, as the complete behavior is defined in mining scripts. So as long as there is a suitable mining script available for a given task, it is possible to start mining repositories right away. A researcher just has to fill in some parameters and then the process runs autonomously in the background. The already existing mining scripts cover popular types of repositories, like GitHub, JIRA, and Stack Overflow. And when a researcher wants to mine some other repository, it is possible to write further mining scripts, which is relatively easy and well documented in this thesis. It is important to note that mining scripts have no logic, they are just key/value pairs that store the information that MiningHub needs to know how to

[7] http://metricsgrimoire.github.io/CVSAnalY/
[8] http://metricsgrimoire.github.io/Bicho/
[9] http://metricsgrimoire.github.io/MailingListStats/

retrieve data from an API (e.g. the URLs to request the specified components). They do not have to handle HTTP requests, timeouts, errors, logging, databases and so on. Additionally, new mining scripts can be reused and even extended by other researchers. In this study, we have also shown that the starting process of data retrieval jobs on MiningHub is straightforward and only takes few minutes, as can be seen in Section 5.1.1. And as soon as a mining process completed, all the retrieved data sets can be downloaded in the JSON file format. The files can be imported in a database that supports nesting, such as MongoDB. Using MongoDB commands, it is also very simple to export to other file formats, for example to CSV. The downloaded files are structured in the exact same way as the utilized API exposes its data. So if somebody is already familiar with that API, the result of MiningHub are especially user-friendly to work with. MiningHub, thus, can simplify the whole mining software repository process. As a result, a researcher can focus on his study instead of setting up different mining tools on his computer. Creating mining jobs takes only some minutes and everything else is done by MiningHub. The researchers then just have to download the requested data sets and start doing the real work.

Threats to Validity. We conducted a basic replication study on the activity scores of people involved in the Groovy community. For that purpose, we only took commits, issues, issue comments, questions, and answers into account. There would be more information available that we could have used, for example question comments, comments on pull requests and so on. Regarding the observed date range, we only used one year of data, namely activity from December, 2012 to November, 2013. Depending on what exactly a study wants to show, this period might be too short or too long. Furthermore, our user matching process was very basic. We only merged accounts when their nick names or email addresses were identical. There would have been more data available to use, for example the name part of the email address or full names if given. Also, more manual work would have improved the matching. As a result, our merging procedure has definitely not found all existing account matches. Another threat could be the missing consideration of weights. We omitted them for simplicity and because of the need to estimate them empirically first. By including weights, the results could look differently, though. Weights could even be used to differentiate between the activities on a single repository, for example between issue reporting and issue commenting. It should also be noted that there are no reference results we could use to compare with our outcome.

5.2 Repository Mining

In Section 5.1, we have shown that MiningHub can be used to simplify the error prone first step of some repository mining processes. We did that by completely replicating a research plan by Bosu [Bos12]. In this section, we show other use cases for which MiningHub might be an advantage over other repository mining approaches. We do so by looking at four different research papers and elaborate on their data need and, if explained, also their mining process. If possible or needed, we explain how mining scripts have to be changed or extended to replicate a corresponding data retrieval step. However, in this section we only describe the approach that we would take to mine software repositories using MiningHub. We did not actually perform that step, though. Additionally, we also show some limitations in the use of MiningHub for repository mining.

5.2.1 GHTorrent

In the paper about GHTorrent [GS12], Gousios and Spinellis described their approach of a mirror of data produced on and with GitHub. To create such a mirror, GHTorrent relies on the GitHub event API, which is a publicly available list of recent events that occurred on GitHub and can be

found on `https://api.github.com/events`. That list contains the latest 30 events, including the creation of commits, issues, comments, wiki pages, releases, forks, downloads, and so on. Using a paging mechanism, it is possible to retrieve the 300 latest events in total. As there are dozens of events occurring in a short time frame, that API has to be read very regularly to avoid missing data. Gousios and Spinellis found that a rate of about 450 requests per hour is adequate to keep up with the event stream of GitHub. Based on the retrieved events, GHTorrent can then crawl the repositories that were represented in the event stream more deeply. When that paper was published, the authors did that for commits and watch events. In the meantime, there are probably more event types that are followed. After the data has been mined from the software repositories, it is made available to other researchers as monthly snapshots. As a result, GHTorrent offers evolutionary information about software projects.

Contrary to GHTorrent, MiningHub is not meant for large-scale repository mining. MiningHub works well for single projects, of which researchers only need a snapshot of data to work with. Furthermore, MiningHub currently does not support temporal constraints in a mining script. So if one wants to replicate at least parts of the GHTorrent functionality, a slight reconfiguration and some tricks with mining scripts are needed. To avoid missing data of the event stream, MiningHub would have to be configured with a lower waiting time between consecutive requests. Currently, this configuration can only be done globally and only by the administrator of the MiningHub installation. A mining script that retrieves events from the stream could look like Listing 5.11. It retrieves events from the stream URL and stores them to the database with the event `id` as primary key so that no duplicates will be stored. Using the `argumentRotation` functionality, we ensure that the stream is requested multiple times. For that, we store an argument called `incrementor` with a starting value of 1 but it is not used in the URL as an argument. It just ensures that MiningHub thinks it requests several different API results. The event stream then is regularly retrieved until the stop condition (starting on line 9) is *true*. We set that condition to be *true* if the `key` of the result is *false*. However, as the `events` API returns a list, we cannot extract a key from it so that the complete list is used as value and, as a result, the condition is never *true*. With these additions to a mining script, we allow MiningHub to retrieve events forever. As a result, it is—at least in theory—possible to replicate the event stream mirroring of GHTorrent. With our approach several limitations apply of which the data download capabilities are the most severe. Normally, data can only be downloaded from an import if it has completed or it was stopped. As a never-stopping import does not allow us to download any data, we would have to stop it temporarily. But by doing so we risk the loss of events from the GitHub stream as MiningHub does not request them while being stopped. Furthermore, every time we wanted to download the latest events, we would have to download the whole data set again. There is no way to download only weekly or monthly dumps instead. A further limitation is the back-off mechanism used by MiningHub to avoid flooding of an API with requests when there is a temporary problem. Would a back-off condition be met, further requests to the GitHub API would stop for a certain amount of time. During that back-off period new events cannot be tracked and would be lost, though. So as a conclusion, there is currently no practical way of completely replicating the functionality of GHTorrent with MiningHub.

5.2.2 Innovation Diffusion Through Link Sharing

On the questions and answers platform Stack Overflow, there are a lot of links shared every day. Gomez et al. [GCS13] wanted to analyze and categorize the types of these links. To do that, they used the data set provided by the MSR 2013 Mining Challenge [Bac13], which is based on the official Stack Exchange dump from August 2012. Gomez et al. then extracted all the clickable URLs from the questions and answers in that data set. In total, they found 1,999,026 unique links and 4,197,085 total link citations. As the authors wanted to manually observe linked websites,

```
components:
    events:
        primaryKey: id
        call:
          - url: "https://api.github.com/events?page=1"
            argumentRotation:
                name:  incrementor
                start: 1
                stop:
                    key: key
                    isset: false
```

Listing 5.11: Component for events of a mining script to retrieve GitHub events.

they extracted a random sample of 1000 links and categorized them by type. They found that of the sample, a total of 185 (18%) links just redirected to a *404 page not found* error. Apart from that, the most linked website types were official documentations with a total of 153 links. Blog posts (135 links) and product or project websites (129 links) were second and third most shared website types, respectively. Besides further statistics, they also extracted the number of shares of each domain on the full data set. The top five domains of most shared links on Stack Overflow were `stackoverflow.com` itself, `microsoft.com`, `wikipedia.org`, `google.com`, and `github.com`.

When trying to replicate the data retrieval step of the paper by Gomez et al. using MiningHub, we can use the mining script for Stack Overflow (Section 4.2.4). There are only two changes needed. First, the existing mining script collects the user data about questioners and answerers on Stack Overflow. However, the study by Gomez et al. does not use these data and thus, it does not have to be mined. Therefore, we can remove the complete `users` component block of the existing mining script to save time and network traffic. And second, we have to remove the `tag` parameter from the existing mining script as we want to retrieve questions regardless of their assigned tags. This can be done by just deleting the `tag` from the `urls.parameters` block in the mining script head, removing it from the `arguments` block of each request, and replacing `&tagged={{tag}}` of each request URL. For convenience, we can also change the `urls.pattern` to something like `"^http[s]?://stackoverflow.com"` to match all the Stack Overflow URLs. With these changes done, we can add the updated mining script to MiningHub and start a new import with parameters *from date* of something like *2000-01-01* (this does not really matter as long as it is a date before the inception of Stack Overflow) and a *to date* of *2012-08-31*. Using that approach, it would be possible to mine the same data as was used in this research paper. However, the used MSR Mining Challenge data set [Bac13] contains a huge amount of information; the XML dump weights about 7.1 GigaBytes (GB) and the PostgreSQL dump is sized about 3.1 GB. It would certainly take several days—or rather weeks—to retrieve all data with our approach. Furthermore, MiningHub was not yet tested with data sets sized several GigaBytes. We ran different imports that resulted in data of more than a GigaByte, but these were still smaller than three or even seven GigaBytes. While there are no hard limits related to maximum supported file sizes, we still have to investigate the limits of MiningHub in that matter.

5.2.3 Analyzing Questions by Topic, Type, and Code

Another study based on data from Stack Overflow was conducted by Allamanis and Sutton [AS13b]. For certain categories, for example Java, Python, or Android, they wanted to find insights into certain concepts that are the most confusing (e.g. *how to perform encoding*). They did so by analyzing the questions for a defined set of question tags. The data set they used for that was, again, from the MSR 2013 Mining Challenge [Bac13]. However, they only used a very small part of it: Only questions and no answers or comments from a total of 16 tags were used. On these questions, the authors then applied different text analyzing techniques, such as word stemming, to extract concepts and the types of questions. Examples of such question categories are *concepts that have been coded but do not work*, *not knowing how to implement something*, or *the way of using some piece of code or API*. Based on these concepts, the authors also found that, at least between Python and Java, the types of questions are similar and independent of the programming language. Furthermore, they found that within tag categories (e.g. version control systems or programming languages), the used words are often similar and might be used for code-related information retrieval.

To replicate the repository mining step of the paper by Allamanis and Sutton using MiningHub, the mining scripts can be reused. The script for Stack Overflow can even be reduced by removing all the components except `questions` as this is the only needed component. That means, we can remove `questionComments`, `answers`, `answerComments`, and `users` from the `components` block. After having made these changes to the mining script, the only thing left to do is creating import jobs on MiningHub. For each of the 16 tags, a job has to be created with a date range of for example *2000-01-01* to *2012-08-31*. That range reflects the one of the MSR 2013 Mining Challenge data set [Bac13]. These imports would certainly take some time to retrieve all the requested questions. As an example, Allamanis and Sutton used a total of 281,508 Java related questions in their study [AS13b]. As soon as all the imports have completed, the data can be downloaded. Before actually being able to perform the word analysis, duplicate questions have to be removed. Such duplicates may occur as the list of tags contains overlapping categories, for example *java* and *java-ee*, and there might be questions which were tagged with more than just one of these. As a result, the questions tagged with *java* might contain some questions that are also tagged with *java-ee*. To remove such duplicates, we could for example create a MongoDB collection and import all the mined questions using their question `id` as `_id` in the collection. That way, duplicates are ignored and the result is a collection just containing unique questions. This exactly replicates the data set that was used by Allamanis and Sutton.

5.2.4 Predicting Defect Numbers

When a certain defect of a software project is added to its issue tracker, it normally has a state of *new* or is maybe already *assigned* to somebody. After a developer has fixed the problem, the issue state will then be changed to *resolved* and it will be *closed*. Additionally, there might be intermediate issue states. These defect state transitions are the focus of the paper by Wang and Zhang [WZ12]. They presented BugStates, a method to predict the number of defects at each state based on defect state transition models built on historical issue data. The authors argued that such a prediction can record the defect-fixing performance of a project team or community. Furthermore, it allows to better estimate the software maintenance effort and allocate developer resources for future defect fixing. To evaluate their approach, the authors used six software projects relying on two different bug tracking systems. The projects they used were PDE.Build, JDT.Text, JDT.UI and Platform.Debug by the Eclipse foundation whose bug tracking system is Bugzilla, as well as Apache Lucene and Spring.NET that use JIRA. For each of these projects, they extracted the state transitions of defects from the HTML pages of the issues. On these data sets, they then applied

statistical models to predict the next states of a defect. The authors found that using BugStates, they were able to outperform other related methods in terms of errors and thus achieve a more accurate prediction of defect state numbers.

Mining issue tracking systems is an important and often performed process in the MSR field. For MiningHub, there is already a mining script available for JIRA that can be extended to replicate parts of the data retrieval step of Wang and Zhang's study [WZ12]. Both, SpringSource (creator of Spring.NET) and Apache (creator of Lucene) have publicly open JIRA APIs which can be used to retrieve their issues. Before the existing mining script for JIRA can work for us, we need to change two aspects for this special use case. First, only issues of type *defect* are used for the study. So we have to change the query that we use for selecting the issues of a certain project in the mining script. The corresponding URL has to be extended by an `issuetype=Bug` filter. For the JIRA API, this can be done by using the JIRA query language (JQL) which leads to a URL-escaped query like `search?jql=issuetype%3DBug`. Now, MiningHub retrieves only issues from JIRA which are defects. The second change we have to do is related to the defect transition information as the current mining script does not retrieve it. To explicitly include the transitions in the mining process, we have to create an `each` request which iterates through all the issues and retrieves their defect transition information. This can be collected from the API using a URL like `issue/<issueIdOrKey>?expand=changelog`. That API request essentially retrieves the issue information again but this time it includes all the issue state transitions because of the `expand` argument. So retrieving information from the JIRA installations is relatively effortless. Regarding Bugzilla it is different. Although Bugzilla provides a representational state transfer (REST) API, the instance operated by the Eclipse foundation has not enabled that feature. Instead, they provide an XML remote procedure call (XML-RPC) API, which MiningHub does not yet support. As an alternative, it is also possible to retrieve individual issues as XML by just adding a `ctype=xml` parameter to the issue address. We did not find any way of retrieving a list of issues for a certain project with that method, though. So it appears that retrieving data from the Eclipse Bugzilla with MiningHub is not possible at the moment. As a conclusion, MiningHub works well for issue trackers of single projects as long as they offer a REST API. Currently, there is no other way of mining repositories with MiningHub.

5.3 Roundup

Mining software repositories is an important process in software engineering research. We argue that this process should not be a step that takes a lot of effort and is error-prone. By using the right tools for the right job, repository mining can be simplified considerably. In this chapter, we evaluated MiningHub, our approach to enable software repository mining using the flexible methodology of user-contributed mining scripts. As such, MiningHub has its advantages and limitations. As we have shown in Section 5.1 about finding community structures in a software project community, MiningHub can considerably simplify the data retrieval process at least for some use cases. We replicated a research plan by Bosu [Bos12] and analyzed the activity of community members on GitHub, JIRA, and Stack Overflow for the Groovy project. Each of these platforms are very popular in the software development field and often used in research. We have shown that, in contrast to Bosu's proposed tools, the repository mining step on these platforms using MiningHub is only a very small fraction of the total time needed for the whole study. Furthermore, we argue that the study itself is easier to conduct using MiningHub as we only have to deal with one single file format instead of transforming the data from three different tools. In Section 5.2.1, we tried to replicate the functionality of GHTorrent by Gousios and Spinellis [GS12]. We found that it is not doable with the current version MiningHub. It is only possible to retrieve the GitHub event timeline after a reconfiguration of the MiningHub API request delays. This il-

lustrates a limitation in the current version of MiningHub; there is no way of changing the timing constraints in a mining script. Some APIs might only offer a low amount of requests per time range, others could allow many requests per second or even no limits at all. MiningHub currently cannot support such cases with mining scripts. Furthermore, we found that the lacking download capabilities are the main reason why MiningHub is not suitable for large-scale mining. Currently, there is no way to download, for example, weekly dumps of the retrieved data; only complete dumps are supported. For studies that need large data sets and thus a long retrieval period, (re-)downloading complete data dumps might not be a viable solution. We also analyzed a paper about link sharing on Stack Overflow by Gomez et al. [GCS13] in Section 5.2.2. We found that the retrieval step of that study can be replicated by slightly changing the existing mining script that only retrieves questions related to a single tag. The flexibility of mining scripts allows to remove that constraint and retrieve all questions regardless of their tag instead. Then an import job has to be created on MiningHub and ran for several days (or rather, weeks). While we already let MiningHub import data weighting more than one GigaByte several times, we do not yet know where its limits are. MiningHub works very well for small and medium data sets, for large imports we do not yet have enough experience, though. In Section 5.2.3, we investigated in the data retrieval replicability of a study by Allamanis and Sutton [AS13b] that categorized Stack Overflow questions regarding their content. We found that the provided mining script for Stack Overflow can be reused. It can even be reduced to just retrieve the content of the questions without the need to request all the other components. With that change, retrieving the needed data can be done in less time. One downside of the MiningHub approach in this case is the repetitive task of creating 16 individual import jobs with different question tags and identical parameters (date range and API key). MiningHub currently does not offer an API itself or any other way that could be used to automate the creation of such imports. So while MiningHub can be used for use cases where several different tags or different projects have to be mined, it is not very comfortable to create numerous jobs with almost the same parameters each time. A study by Wang and Zhang [WZ12] about predicting defect numbers in each issue state (e.g. *new*, *assigned*, or *resolved*) was analyzed in Section 5.2.4. Our findings show that retrieving data from public JIRA instances is unproblematic using mining scripts. The one for JIRA can be changed to only retrieve the issues of type *defect* and to include the transitions between different states. These changes can be achieved by appending more arguments to the already specified API URLs. For Bugzilla, on the other hand, there is not yet a mining script available that can be used. But as MiningHub does not support XML-RPC, there is currently no way to extract the issues from the Eclipse projects using MiningHub. This exposes another limitation of MiningHub; currently it is impossible to retrieve data from platforms that either do not provide an API at all or provide an API that does not allow requests by adding parameters to a URL. XML-RPC on the contrary, requires requests to be done to a single URL by sending data as XML. As a result, the data retrieval step of this study can only partly be replicated by MiningHub.

In general, MiningHub works fine on small to medium-sized data sets, such as the version history and issues of a single software project. The repository mining step of such studies can be considerably reduced by using MiningHub in some cases. Using mining scripts, researchers are also able to exactly specify what data they want to retrieve and what parameters are required to be given. Such scripts allow a high flexibility and can—but do not have to—be fine-tuned individually for each study. Mining scripts even allow to extend MiningHub to support new repositories. MiningHub also allows to create several individual jobs to retrieve more than just one project. But this is currently laborious, especially for cases when a large number of imports would be required. When broader data sets are needed, researchers can write mining scripts that retrieve a lot of data of projects in the same job. This is not always possible, though. An interesting feature for MiningHub would thus be the addition of an API which could allow easier import creation for repetitive parameters. Furthermore, MiningHub has limited capabilities when

it comes to large-scale repository mining. For example, there is no way to download intermediate data for long-running imports. Only complete dumps can be (re-)downloaded. Support for daily or weekly dumps would be useful in such cases. MiningHub currently only works on APIs that allow to request data with URL parameters but do not require any further content to be sent. As a result, some API types like XML-RPC are not supported; extending the capabilities in that matter is left to future work. While mining scripts allow flexible modifications, there is currently no way of changing temporal constraints. MiningHub might thus be unable to exploit the number of allowed API requests for certain providers while it may flood others that have low request limits. This is an area where mining scripts could still be improved. And finally, we do not yet know where the limits of MiningHub are. We found that it works very well for some small to medium-sized imports and can also be used for larger data sets. But it is an open issue on how large these data sets can be.

Chapter 6

Conclusion and Future Work

The increasing interest in social coding sites has led to more publicly available software repositories which are frequently used to conduct research. Retrieving data from these repositories is often time-consuming and error-prone, and yet many published MSR studies do not make the used data sets available. This complicates replication of these studies and causes research communities to perform repetitive work on creating tools and retrieving data. In this thesis, we presented our software repository mining approach called MiningHub to overcome limitations in the MSR community. MiningHub is a web-based platform that can be extended by writing mining scripts. Such scripts are textual specifications describing how certain repositories need to be mined and thus allow researchers to add support for these on the MiningHub platform. Mining scripts can be shared among other research communities so that repetitive work can be avoided. Mined data can also be published for other researchers to enable reuse and replication of studies. In our evaluation, we replicated a research plan by Bosu [Bos12] to analyze community structures over three different project repositories. We have shown that by using MiningHub, retrieving data from these repositories takes only a small amount of time. Based on the mined data we were able to completely replicate that study. Furthermore, we have analyzed the repository mining steps of four other studies [AS13b,GCS13,GS12,WZ12] and found certain evidence that MiningHub could be used for three of them. MiningHub was technically evaluated and we found it to be a viable approach for some small and medium-sized data requirements. We believe that MiningHub has the potential to simplify the mining step of many papers published in the context of the MSR conference and also others. The extensible platform supports an arbitrary number of repositories and, ideally, MiningHub is the only tool needed for mining software projects. Data of each of the supported repositories can be retrieved and downloaded in the same data formats, which simplifies the integration of data obtained from multiple software repositories. The MiningHub platform also enables sharing and reusing of both mining scripts and mined data sets. The social features and sharing capabilities still have to prove themselves in practical use. With more mining scripts, we expect that MiningHub can become an attractive platform for mining software repositories as it is capable of simplifying the mining process and enables study replications.

Future Work. The MiningHub platform has reached its first milestone and enables mining of most software repositories with its mining script mechanism. Both the scripts and the mined data sets can be shared among certain researchers only or with the public. In addition, we plan to further extend and improve the functionality of the MiningHub platform. As a first step, more public mining scripts that add support for additional repositories should be added. Candidates

are for example Bugzilla[1], Google Groups[2], BitBucket[3], Jenkins[4], and many more. We also think about adding an API so that the creation and handling of repository mining jobs on MiningHub can be automated using corresponding applications. This is useful when it is needed to retrieve data from a list of projects instead of just a single one, for example. Currently, only manually started import jobs are possible. Support for scheduled retrieval processes has to be added so that MiningHub regularly checks for and retrieves new data. We also plan to improve support for large-scale repository mining. Therefore, timed snapshots of data are needed so that partial data sets can be downloaded instead of only one complete dump. A further step of extension should be the support for more download formats like CSV, XML, SEON, or the resource description framework (RDF). In addition to technical improvements, MiningHub has to be evaluated regarding the social features in practice.

[1] http://www.bugzilla.org
[2] https://groups.google.com
[3] https://bitbucket.org
[4] http://jenkins-ci.org

Appendix A

Mining Scripts

These sections show the complete mining script schema as well as the mining scripts for GitHub, JIRA, and Stack Overflow.

A.1 Mining Script Schema

Listing A.1: JSON schema definition of a valid mining script.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Schema to define the contents of a Mining Script for MiningHub.",
    "type": "object",
    "properties": {
        "name": { "type": "string" },
        "description": { "type": "string" },
        "apiFormat": {
            "type": "string",
            "enum": [ "json" ]
        },
        "urls": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "pattern": { "type": "string" },
                    "parameters": {
                        "type": "array",
                        "items": { "type": "string" }
                    }
                },
                "additionalProperties": false,
                "required": [ "pattern" ]
            }
        },
        "parameters": {
            "type": "object",
            "patternProperties": {
                "[a-z0-9]": {
                    "type": "object",
                    "properties": {
                        "type": {
                            "type": "string",
                            "enum": [
                                "string",
```

```
                                "password",
                                "date",
                                "datetime",
                                "int"
                            ]
                        },
                        "description": { "type": "string" }
                    },
                    "additionalProperties": false,
                    "required": [ "description" ]
                }
            },
            "additionalProperties": false
        },
        "globalArguments": {
            "type": "object",
            "id": "#arguments",
            "patternProperties": {
                "[a-z0-9]": {
                    "type": [
                        "object",
                        "null"
                    ],
                    "properties": {
                        "type": {
                            "type": "string",
                            "enum": [
                                "date",
                                "datetime",
                                "int",
                                "string"
                            ]
                        },
                        "format": {
                            "type": "string",
                            "enum": [
                                "unix"
                            ]
                        },
                        "value": {
                            "type": [
                                "string",
                                "integer"
                            ]
                        }
                    },
                    "additionalProperties": false
                }
            },
            "additionalProperties": false
        },
        "backoffWhen": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "key": { "type": "string" },
                    "source": {
                        "type": "string",
                        "enum": [
                            "http",
                            "result"
                        ]
                    },
```

```
                    "type": {
                        "type": "string",
                        "enum": [
                            "int",
                            "string"
                        ]
                    },
                    "lowerThan": { "type": "integer" }
                },
                "additionalProperties": false,
                "required": [ "key" ]
            }
        },
        "components": {
            "type": "object",
            "patternProperties": {
                "[a-z0-9]": {
                    "type": "object",
                    "properties": {
                        "description": { "type": "string" },
                        "primaryKey": { "type": "string" },
                        "call": {
                            "type": "array",
                            "items": {
                                "type": "object",
                                "properties": {
                                    "url": { "type": "string" },
                                    "arguments": { "$ref": "#/properties/globalArguments" },
                                    "store": { "type": "string" },
                                    "argumentRotation": {
                                        "type": "object",
                                        "properties": {
                                            "name": { "type": "string" },
                                            "start": {
                                                "type": [
                                                    "string",
                                                    "integer"
                                                ]
                                            },
                                            "type": {
                                                "type": "string",
                                                "enum": [
                                                    "text",
                                                    "int"
                                                ]
                                            },
                                            "stop": {
                                                "type": "object",
                                                "properties": {
                                                    "key": { "type": "string" },
                                                    "source": {
                                                        "type": "string",
                                                        "enum": [
                                                            "result",
                                                            "http"
                                                        ]
                                                    },
                                                    "value": { },
                                                    "isset": { "type": "boolean" }
                                                },
                                                "additionalProperties": false,
                                                "required": [
                                                    "key"
                                                ]
```

```
                                        },
                                        "continue": {
                                            "type": "object",
                                            "properties": {
                                                "increment": { "type": "integer" },
                                                "valueFromRegex": { "type": "string" }
                                            },
                                            "additionalProperties": false
                                        }
                                    },
                                    "additionalProperties": false,
                                    "required": [
                                        "name",
                                        "stop"
                                    ]
                                }
                            },
                            "additionalProperties": false,
                            "required": [ "url" ]
                        }
                    },
                    "each": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "component": { "type": "string" },
                                "url": { "type": "string" },
                                "arguments": { "$ref": "#/properties/globalArguments" },
                                "argumentEach": {
                                    "type": "object",
                                    "properties": {
                                        "name": { "type": "string" },
                                        "key": { "type": "string" },
                                        "separator": { "type": "string" },
                                        "merge": { "type": "integer" }
                                    },
                                    "additionalProperties": false,
                                    "required": [
                                        "name",
                                        "key"
                                    ]
                                },
                                "store": { "type": "string" },
                                "repeat": {
                                    "type": "object",
                                    "properties": {
                                        "key": { "type": "string" },
                                        "source": {
                                            "type": "string",
                                            "enum": [
                                                "result",
                                                "http"
                                            ]
                                        },
                                        "value": { },
                                        "isset": { "type": "boolean" },
                                        "argument": { "type": "string" }
                                    },
                                    "additionalProperties": false,
                                    "required": [
                                        "key",
                                        "argument"
                                    ]
```

```
                                    }
                                },
                                "additionalProperties": false,
                                "required": [
                                    "component",
                                    "url"
                                ]
                            }
                        }
                    },
                    "additionalProperties": false,
                    "required": [
                        "description",
                        "primaryKey"
                    ]
                }
            },
            "additionalProperties": false
        }
    },
    "additionalProperties": false,
    "required": [
        "name",
        "description",
        "urls",
        "components"
    ]
}
```

A.2 GitHub

Listing A.2: Mining script for GitHub.

```
name:        GitHub.com
description: Powerful collaboration, code review, and code management for open source and private
     projects.
apiFormat:   json

urls:
  - pattern: "^http[s]?://github.com/([^/]+)/([^/]+)[/]?"
    parameters:
      - user
      - repo
  - pattern: "^http[s]?://www\\.github.com/([^/]+)/([^/]+)[/]?"
    parameters:
      - user
      - repo
  - pattern: "^git://github.com/([^/]+)/([^/]+).git"
    parameters:
      - user
      - repo

parameters:
    branch:
        type: string
        description: The name of the branch to mine (most often master or trunk).
        default: master
    accessToken:
        type: password
        description: The Personal Access Token obtained from GitHub.
```

```
27
28
29 globalArguments:
30     user:
31     repo:
32     accessToken:
33
34
35 components:
36
37     commits:
38         description: Code commits
39         primaryKey: sha
40         call:
41           - url: "https://api.github.com/repos/{{user}}/{{repo}}/commits?sha={{branch}}&last_sha={{
                 lastSha}}&access_token={{accessToken}}"
42             arguments:
43                 lastSha:
44                 branch:
45             store: .
46             index:
47               - commit.message
48               - commit.author
49             argumentRotation:
50                 name:  lastSha
51                 start: ""
52                 type: text
53                 stop:
54                     source: http
55                     key: Link
56                     isset: false
57                 continue:
58                     valueFromRegex: 'last_sha=([^&]+)[^>]+>; rel=\"next\"'
59
60     commitComments:
61         description: Comments to Commits
62         primaryKey: id
63         call:
64           - url: "https://api.github.com/repos/{{user}}/{{repo}}/comments?page={{commentPage}}&
                 access_token={{accessToken}}"
65             arguments:
66                 commentPage:
67             store: .
68             index:
69               - body
70             argumentRotation:
71                 name: commentPage
72                 start: 1
73                 stop:
74                     key: .
75                     value: []
76
77     issues:
78         description: Project Issues
79         primaryKey: number
80         call:
81           # open issues
82           - url: "https://api.github.com/repos/{{user}}/{{repo}}/issues?page={{openIssuePage}}&state=
                 open&access_token={{accessToken}}"
83             arguments:
84                 openIssuePage:
85             store: .
86             index:
87               - title
```

```
            - body
          argumentRotation:
              name:  openIssuePage
              start: 1
              stop:
                  source: http
                  key: Link
                  isset: false
        # closed issues
        - url: "https://api.github.com/repos/{{user}}/{{repo}}/issues?page={{closedIssuePage}}&
            state=closed&access_token={{accessToken}}"
          arguments:
              closedIssuePage:
          store: .
          index:
            - title
            - body
          argumentRotation:
              name:  closedIssuePage
              start: 1
              stop:
                  source: http
                  key: Link
                  isset: false

    milestones:
        description: Project Milestones
        primaryKey: number
        call:
          # open milestones
          - url: "https://api.github.com/repos/{{user}}/{{repo}}/milestones?page={{openMilestonePage
              }}&state=open&access_token={{accessToken}}"
            arguments:
                openMilestonePage:
            store: .
            index:
              - title
              - description
            argumentRotation:
                name:  openMilestonePage
                start: 1
                stop:
                    source: http
                    key: Link
                    isset: false
          # closed milestones
          - url: "https://api.github.com/repos/{{user}}/{{repo}}/milestones?page={{
              closedMilestonePage}}&state=closed&access_token={{accessToken}}"
            arguments:
                closedMilestonePage:
            store: .
            index:
              - title
              - description
            argumentRotation:
                name:  closedMilestonePage
                start: 1
                stop:
                    source: http
                    key: Link
                    isset: false

    issueComments:
        description: Comments to project issues
```

```
149         primaryKey: id
150         call:
151           - url: "https://api.github.com/repos/{{user}}/{{repo}}/issues/comments?page={{
                issueCommentPage}}&access_token={{accessToken}}"
152             arguments:
153                 issueCommentPage:
154             index:
155               - body
156             argumentRotation:
157                 name: issueCommentPage
158                 start: 1
159                 stop:
160                     key: .
161                     value: []
162
163     branches:
164         description: Development branches
165         primaryKey: commit.sha
166         call:
167           - url: "https://api.github.com/repos/{{user}}/{{repo}}/branches?page={{branchesPage}}&
                access_token={{accessToken}}"
168             arguments:
169                 branchesPage:
170             store: .
171             argumentRotation:
172                 name: branchesPage
173                 start: 1
174                 stop:
175                     source: http
176                     key: Link
177                     isset: false
178
179     tags:
180         description: Development snapshots
181         primaryKey: name
182         call:
183           - url: "https://api.github.com/repos/{{user}}/{{repo}}/tags?access_token={{accessToken}}"
184
185     releases:
186         description: Project releases
187         primaryKey: id
188         call:
189           - url: "https://api.github.com/repos/{{user}}/{{repo}}/releases?access_token={{accessToken
                }}"
190             index:
191               - body
192
193     forks:
194         description: Project forks
195         primaryKey: id
196         call:
197           - url: "https://api.github.com/repos/{{user}}/{{repo}}/forks?sort=oldest&page={{forksPage
                }}&access_token={{accessToken}}"
198             arguments:
199                 forksPage:
200             argumentRotation:
201                 name: forksPage
202                 start: 1
203                 stop:
204                     key: .
205                     value: []
```

A.3 JIRA

Listing A.3: Mining script for JIRA.

```
name:        "Jira: Public Issue Trackers"
description: This mining script will collect data about a single project and all its issues from
     public issue trackers. As Jira can be installed on any domain, this script will match all urls
     that contain "/rest/api/2", the default Jira API accessor.
apiFormat:   json

urls:
  - pattern: "(^.*)/rest/api/2[/]?"
    parameters:
      - url

parameters:
    projectKey:
        type: string
        description: The mandatory key name of a project, it *must* be UPPERCASE.

globalArguments:
    url:
    projectKey:


components:

    projects:
        description: Project information
        primaryKey: id
        call:
          - url: "{{url}}/rest/api/2/project/{{projectKey}}"
            store: ..
            # only store one single element, not each element in a collection
            index:
              - name
              - description
              - lead.name
              - lead.displayName

    issues:
        description: Issues
        primaryKey: id
        call:
          - url: "{{url}}/rest/api/2/search?jql=project={{projectKey}}&startAt={{issuePage}}&
               maxResults=20&fields=*all"
            arguments:
                issuePage:
            store: issues
            index:
              - fields.summary
              - fields.description
            argumentRotation:
                name:  issuePage
                start: 0
                stop:
                    key: issues
                    value: []
                continue:
                    increment: 20
```

A.4 Stack Overflow

Listing A.4: Mining script for Stack Overflow.

```
name:        StackOverflow.com
description: StackOverflow.com is a help board for programmers.
apiFormat:   json

urls:
  - pattern: "^http[s]?://stackoverflow.com/questions/tagged/([^/+]+)"
    parameters:
      - tag

parameters:
    fromdate:
        type: date
        description: The date on which to start getting data.
    todate:
        type: date
        description: The date on which to end getting data.
    key:
        type: password
        description: The API key you received from StackExchange.

globalArguments:
    key:


components:

    questions:
        description: questions
        primaryKey:  question_id
        call:
          - url: "https://api.stackexchange.com/2.1/questions?page={{page}}&pagesize=100&fromdate={{
               fromdate}}&todate={{todate}}&order=asc&sort=creation&tagged={{tag}}&site=stackoverflow
               &filter=withBody&key={{key}}"
            arguments:
                todate:
                    type: date
                    format: unix
                fromdate:
                    type: date
                    format: unix
                tag:
                page:
            store: items
            index:
              - body
              - title
              - tags
            argumentRotation:
                name:  page
                start: 1
                stop:
                    key: has_more
                    value: false


    questionComments:
        description: Comments to Questions
        primaryKey:  comment_id
```

```
57          each:
58            - component: questions
59              url: "https://api.stackexchange.com/2.1/questions/{{ids}}/comments?page={{page}}&pagesize
                    =100&order=asc&sort=creation&site=stackoverflow&filter=withBody&key={{key}}"
60              arguments:
61                  page:
62                      value: 1
63              argumentEach:
64                  name: ids
65                  key: question_id
66                  separator: ;
67                  merge: 10
68              store: items
69              index:
70                - body
71              repeat:
72                  key: has_more
73                  value: true
74                  argument: page
75
76
77      answers:
78          description: Answers to the Questions
79          primaryKey:  answer_id
80          each:
81            - component: questions
82              url: "https://api.stackexchange.com/2.1/questions/{{ids}}/answers?page={{page}}&pagesize
                    =100&order=asc&sort=creation&site=stackoverflow&filter=withBody&key={{key}}"
83              arguments:
84                  page:
85                      value: 1
86              argumentEach:
87                  name: ids
88                  key: question_id
89                  separator: ;
90                  merge: 10
91              store: items
92              index:
93                - body
94              repeat:
95                  key: has_more
96                  value: true
97                  argument: page
98
99
100     answerComments:
101         description: Comments to Answers
102         primaryKey:  comment_id
103         each:
104           - component: answers
105             url: "https://api.stackexchange.com/2.1/answers/{{ids}}/comments?page={{page}}&pagesize
                    =100&order=asc&sort=creation&site=stackoverflow&filter=withBody&key={{key}}"
106             arguments:
107                 page:
108                     value: 1
109             argumentEach:
110                 name: ids
111                 key: answer_id
112                 separator: ;
113                 merge: 10
114             store: items
115             index:
116               - body
117             repeat:
```

```
118                 key: has_more
119                 value: true
120                 argument: page
121
122
123     users:
124         description: Users
125         primaryKey:  user_id
126         each:
127           - component: questions
128             url: "https://api.stackexchange.com/2.1/users/{{ids}}?pagesize=100&site=stackoverflow&key
                      ={{key}}"
129             argumentEach:
130                 name: ids
131                 key: owner.user_id
132                 separator: ;
133                 merge: 100
134             store: items
135             index:
136               - display_name
137           - component: questionComments
138             url: "https://api.stackexchange.com/2.1/users/{{ids}}?pagesize=100&site=stackoverflow&key
                      ={{key}}"
139             argumentEach:
140                 name: ids
141                 key: owner.user_id
142                 separator: ;
143                 merge: 100
144             store: items
145             index:
146               - display_name
147           - component: answers
148             url: "https://api.stackexchange.com/2.1/users/{{ids}}?pagesize=100&site=stackoverflow&key
                      ={{key}}"
149             argumentEach:
150                 name: ids
151                 key: owner.user_id
152                 separator: ;
153                 merge: 100
154             store: items
155             index:
156               - display_name
157           - component: answerComments
158             url: "https://api.stackexchange.com/2.1/users/{{ids}}?pagesize=100&site=stackoverflow&key
                      ={{key}}"
159             argumentEach:
160                 name: ids
161                 key: owner.user_id
162                 separator: ;
163                 merge: 100
164             store: items
165             index:
166               - display_name
```

Appendix B

Evaluation Queries

This section contains queries that were used in the MiningHub evaluation (Chapter 5).

```
db.gh_commits.find().forEach(function(doc) {
    db.gh_users.update(
        {_id: doc.committer.login},
        {
            $addToSet: {
                emails: doc.commit.committer.email
            },
            $inc: {contributions: 1}
        },
        {upsert: true}
    );
    if (doc.committer.login != doc.author.login) {
        db.gh_users.update(
            {_id: doc.author.login},
            {
                $addToSet: {
                    emails: doc.commit.author.email
                },
                $inc: {contributions: 1}
            },
            {upsert: true}
        );
    }
});
```

Listing B.1: Create collection of unique GitHub users and consolidate their email addresses.

```
1  db.gh_users.find().forEach(function(doc) {
2      doc.emails.forEach(function(mail) {
3          db.users.update(
4              {
5                  $or: [
6                      {_id: doc._id},
7                      {jr_name: doc._id},
8                      {so_hash: hex_md5(mail)},
9                      {jr_email: mail}
10                 ]
11             },
12             {
13                 $set: {
14                     gh_name:doc._id,
15                     gh_email:mail,
16                     gh_contributions:doc.contributions
17                 }
18             }
19         );
20     });
21 });
```

Listing B.2: Match GitHub accounts to all other users by looping through each email address.

Appendix C

Contents of the CD

- Written thesis (*Masterarbeit.pdf*)
  - Abstract (*Abstract.txt*)
  - German abstract (*Zusfsg.txt*)
- Mining script schema (*mining-script-schema/schema.json*)
- Mining scripts
  - GitHub (*mining-scripts/github.yaml*)
  - JIRA (*mining-scripts/jira.yaml*)
  - Stack Overflow (*mining-scripts/stackoverflow.yaml*)
- Source code (*code/*)
- Evaluation data (*evaluation-data/*)

Bibliography

[AS13a] Miltiadis Allamanis and Charles Sutton. Mining Source Code Repositories at Massive Scale using Language Modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216, San Francisco, CA, USA, 2013. IEEE Press.

[AS13b] Miltiadis Allamanis and Charles Sutton. Why, When, and What: Analyzing Stack Overflow Questions by Topic, Type, and Code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 53–56, San Francisco, CA, USA, 2013. IEEE Press.

[Bac13] Alberto Bacchelli. Mining Challenge 2013: Stack Overflow. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, page to appear, San Francisco, CA, USA, 2013.

[Bos12] Amiangshu Bosu. Mining repositories to reveal the community structures of Open Source Software projects. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 397–398, Tuscaloosa, AL, USA, 2012. ACM.

[BRB+09] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The Promises and Perils of Mining Git. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, Vancouver, Canada, 2009.

[CH05] Kevin Crowston and James Howison. The social structure of free and open source software development. *First Monday*, 10(2):1–21, 2005.

[DMWL13] Serge Demeyer, Alessandro Murgia, Kevin Wyckmans, and Ahmed Lamkanfi. Happy Birthday! A Trend Analysis on Past MSR Papers. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 353–362, San Francisco, CA, USA, 2013. IEEE Press.

[DNRN13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431, San Francisco, CA, USA, 2013. IEEE Press.

[GCS13] Carlos Gómez, Brendan Cleary, and Leif Singer. A Study of Innovation Diffusion through Link Sharing on Stack Overflow. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 81–84, San Francisco, CA, USA, 2013. IEEE Press.

[GG11] Giacomo Ghezzi and Harald C. Gall. SOFAS: A Lightweight Architecture for Software Analysis as a Service. In *9th Working IEEE/IFIP Conference on Software Architecture*, pages 93–102, Boulder, CO, USA, 2011. IEEE Computer Society.

[Gou13] Georgios Gousios. The GHTorent Dataset and Tool Suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 233–236, San Francisco, CA, USA, 2013. IEEE Press.

[GR13] Claudio Gutierrez and Romain Robbes. WEON: Towards a softWare Ecosystem ONtology. In *Proceedings of the 2013 International Workshop on Ecosystem Architectures*, pages 16–20, Saint Petersburg, Russia, 2013. ACM.

[GS09] Georgios Gousios and Diomidis Spinellis. Alitheia Core: An extensible software quality monitoring platform. In *Proceedings of the 31st International Conference on Software Engineering*, pages 579–582, Washington, DC, USA, 2009. IEEE Computer Society.

[GS12] Georgios Gousios and Diomidis Spinellis. GHTorrent: Github's Data from a Firehose. In *9th IEEE Working Conference of Mining Software Repositories*, pages 12–21, Zurich, Switzerland, 2012. IEEE.

[HCC06] James Howison, Megan Conklin, and Kevin Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[HICRH+09] Israel Herraiz, Daniel Izquierdo-Cortazar, Francisco Rivas-Hernandez, Jesús Gonzalez-Barahona, Gregorio Robles, Santiago Dueñas Domínguez, Carlos Garcia-Campos, Juan Francisco Gato, and Liliana Tovar. FLOSSMetrics: Free / Libre / Open Source Software Metrics. In *13th European Conference on Software Maintenance and Reengineering*, pages 281–284, Kaiserslautern, Germany, 2009.

[HNB+13] Hadi Hemmati, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W. Godfrey. The MSR Cookbook: Mining a Decade of Research. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 343–352, San Francisco, CA, USA, 2013. IEEE Press.

[JHSA13] Werner Janjic, Oliver Hummel, Marcus Schumacher, and Colin Atkinson. An Unabridged Source Code Dataset for Research in Software Reuse. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 339–342, San Francisco, CA, USA, 2013. IEEE Press.

[KZK+06] Sunghun Kim, Thomas Zimmermann, Miryung Kim, Ahmed Hassan, Audris Mockus, Tudor Girba, Martin Pinzger, E. James Whitehead Jr., and Andreas Zeller. TA-RE : An Exchange Language for Mining Software Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 1–4, Shanghai, China, 2006. ACM.

[LBN+08] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: Mining and Searching Internet-Scale Software Repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, October 2008.

[Lev66] Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[MMM+11] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design Lessons from the Fastest Q&A Site in the West. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2857–2866, New York, NY, USA, 2011. ACM.

[NNBH13] Hoda Naguib, Nitesh Narayan, Bernd Brügge, and Dina Helal. Bug Report Assignee Recommendation using Activity Profiles. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 22–30, San Francisco, CA, USA, 2013. IEEE Press.

[OBL+09] Joel Ossher, Sushil Bajracharya, Erik Linstead, Pierre Baldi, and Cristina Lopes. SourcererDB: An Aggregated Repository of Statically Analyzed and Cross-Linked Open Source Java Projects. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, pages 183–186, Vancouver, Canada, 2009.

[OIAGAG10] Sunday O. Olatunji, Syed U. Idrees, Yasser S. Al-Ghamdi, and Jarallah Saleh Ali Al-Ghamdi. Mining Software Repositories – A Comparative Analysis. *International Journal of Computer Science and Network Security*, 10(8):161–174, 2010.

[RGBICH09] Gregorio Robles, Jesús M. González-Barahona, Daniel Izquierdo-Cortazar, and Israel Herraiz. Tools for the Study of the Usual Data Sources found in Libre Software Projects. *International Journal of Open Source Software and Processes*, 1(1):24–45, 2009.

[RGBICH11] Gregorio Robles, Jesús M. González-Barahona, Daniel Izquierdo-Cortazar, and Israel Herraiz. Tools and datasets for mining libre software repositories. *Multi-Disciplinary Advancement in Open Source Software and Processes*, 1:24–42, 2011.

[Rob10] Gregorio Robles. Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories Proceedings. In *Proceedings of the 7th International Working Conference on Mining Software Repositories*, pages 171–180, Cape Town, South Africa, 2010. IEEE.

[SAG13] Francisco Zigmund Sokol, Mauricio Finavaro Aniche, and Marco Aurélio Gerosa. MetricMiner: Supporting Researchers in Mining Software Repositories. In *13th International Working Conference on Source Code Analysis and Manipulation*, pages 142–146, Eindhoven, Netherlands, 2013. IEEE.

[Tiw11] Shashank Tiwari. *Professional NoSQL*. Programmmer to programmer. Wiley, 1 edition, 2011.

[WGH+12] Michael Würsch, Giacomo Ghezzi, Matthias Hert, Gerald Reif, and Harald C. Gall. SEON: a pyramid of ontologies for software evolution and its applications. *Computing*, 94(11):857–885, July 2012.

[WZ12] Jue Wang and Hongyu Zhang. Predicting Defect Numbers Based on Defect State Transition Models. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 191–200, New York, NY, USA, 2012. ACM Press.