Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MANUEL FREDERIC ZERPIES
MASTER THESIS

MEASURING PATCH FLOW ON GITHUB

Submitted on 2 June 2015

Supervisors: Prof. Dr. Dirk Riehle, M.B.A., M.Sc. Maximilian Capraro
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 2 June 2015

License

Erlangen, 2 June 2015

Contents

Acronyms

API Application Program Interface. 4, 9, 10, 29, 32–34, 36, 37, 39, 41

GB Gigabyte. 9

IS Inner Source Software. 1, 5, 7, 14–16, 30, 32

KB Kilobyte. 9

OS Open Source Software. vii, 1, 3–7, 14–16, 19, 25, 30, 32

SCM Source Code Management. vii, 1, 14, 30

SLOC Source Lines of Code. 5, 10, 12, 14, 16, 18–20, 25, 26

VCS Version Control System. 5, 6

List of Figures

List of Tables

Abstract

For Open Source Software (OS) projects, collaboration is a key to success, as less collaboration between projects leads to projects with less progress. Patches from other OS projects provide the projects with a higher code quality or functionality. In literature, several papers examine the extent of collaboration on OS projects. Yet, most of these studies do not cover the collaboration between different projects.

To understand the collaboration between OS projects, Source Code Management (SCM) repositories are an essential source. Between repositories exists a connection by patches, which can be obtained by data mining the projects repositories. The measurement of the connection by patches is very difficult, because the information about where the patches go and where they come from is not stored within a repository. Collaboration between OS projects can be expressed as so called *Patch Flow*. As an example for the OS world, I use GitHub.com as data source.

I present to which extent *Patch Flow* exists between repositories and what circumstances influence *Patch Flow*. Further, I introduce a model which represents the *Patch Flow* in detail. Based on this model, I developed a crawler to collect data from the GitHub.com repositories. The analysis of the gathered data shows, that *Patch Flow* between OS projects exists. Numbers suggest, that collaboration among projects is common in OS projects.

1 Introduction

This thesis aims to crawl SCM repositories to understand and measure collaboration among repositories represented as *Patch Flow*. Collaboration among repositories can take place either on code-level or on social networks such as mailing lists or forums. This thesis concentrates on code-level collaboration.

The main goal of this thesis was to develop a model, that describes *Patch Flow*. Based on this model, a crawler is implemented to mine SCM repositories in both, OS and Inner Source Software (IS), to measure *Patch Flow*. The crawler is tested on OS repositories, hosted on GitHub.com.

The collected data is filtered by specific criteria to show the existence and extent of *Patch Flow*.

The remainder of this thesis is structured as follows. Chapter 2 describes the research:

- section 2.1 introduces the field of research,
- the related work is reviewed in section 2.2,
- I present the research questions I answer in this thesis in section 2.3,
- in section 2.4 I describe my research methods,
- section 2.5 introduces the crawler, GitHubble,
- the model and definition of *Patch Flow* is presented in section 2.6,
- the results are presented in section 2.7 and discussed in section 2.8,
- section 2.9 discusses the threats to validity this thesis faces and
- section 2.10 concludes and presents future work.

In chapter 3, I elaborate the research, present the hurdles this thesis faced and how I solved them. Further I show how GitHubble works in detail.

2 Research

2.1 Introduction

In the OS world, collaboration is a key to success. Less collaboration between projects can lead to less progress or quality. Projects can only survive because many people are willing to share their knowledge and to contribute. Countless developers contribute to OS projects, like Linux, but statistics that measure the interaction between projects do barely exist. How collaboration among projects itself can be measured is not researched deeply.

Understanding collaboration between projects is essential to understand the dynamics of OS. With this information it is possible to show, how OS repositories are collaborating and interacting on code-level and what factors influence collaboration among projects.

Collaboration between projects can be understood as transfer of code. The code is passed via so called patches. I call this connection between repositories by patches *Patch Flow*. Some patches are just of structural or textual nature, like deleting trailing whitespaces or refactoring code. Other patches fix bugs, introduce new features or add documentation. Many more types of content are possible.

I gather data from GitHub.com. The free Git hosting website hosts more than 6 million repositories, which are connected to each other through collaboration. The data is freely available and therefore ideal for research. Automated processing is the key for research on large data sets. For this purpose I present GitHubble, a Java based tool to gather data from GitHub.com.

My contributions are:

- The definition of *Patch Flow*.
- Showing to which extent *Patch Flow* exists between projects.
- Discussing the extent of *Patch Flow* between projects.
- The presentation of the automated tool GitHubble to crawl GitHub.com.

In the following chapters I will mark out the field of my thesis. In section 2.2 I show the related work. In section 2.3, I present the research questions that this thesis answers. After that, in section 2.4, I explain my scientific method, followed by the presentation of my tool, GitHubble, that I wrote to collect the data that is necessary to answer the research questions in section 2.5. Later, in section 2.6, I define *Patch Flow* and present my results in section 2.7. In section 2.8, I discuss the results. Finally, I show the limitations of my approach in section 2.9.

2.2 Related Work

From the OS research field, I identified two important groups for this thesis: One uses GitHub.com as a research source and the other concentrates on the collaboration between OS projects. We do not use the whole universe of open source projects, but the subset of projects hosted on GitHub.com.

2.2.1 Using GitHub for Research

In the following, I present a selection of the work concentrating on GitHub.com as the source of data and the differences between these papers and my thesis.

As Gousios and Spinellis (2012) I crawl the GitHub.com repositories, but I do not intend to make the GitHub.com Application Program Interface (API) publicly available on a higher bandwidth, but understand the connections between GitHub.com repositories.

Kalliamvakou et al. (2014) show how users on GitHub.com interact and use repositories on GitHub.com in general, but not the way the projects hosted on GitHub.com are connected. In this thesis I focus on how the repositories are connected and how one can measure these connections.

Like Tymchuk et al. (2014), I am interested in collaboration between OS repositories, but unlike them, I do not concentrate much on the users locations, but on the collaborators' connection to the repositories.

Heller et al. (2011) are using visualization techniques to show information about users' geographic location to reveal possible connections between developers. I want to know more about collaboration between repositories based on the code, e. g. the distribution of code among repositories, no matter where contributors are located.

2.2.2 Collaboration between Open Source Projects

Collaboration between OS projects can be divided into two subgroups: One that concentrates on code-level collaboration and takes into account the source code, and a second one that sources the repository forges social networks for events, user locations or activities beyond the code.

Code-Level Collaboration

Research on code-level collaboration concentrates on changed code and the impact it has on the repository. Collaboration between repositories based on code is well researched. In the following paragraphs I present a selection of the related work based on code-level collaboration and where my thesis differs from the work in the field.

The idea of Robles et al. (2004) is that large scale retrieval and analysis of data of software repositories is only possible with automation. Automating the data collection is also important for crawling repositories and is one contribution of this thesis. Further they mostly provide data about the number of Source Lines of Codes (SLOCs) a project has over time, but not for where the patch comes from and how the projects are connected. When implementing GitHubble, I aimed also for IS projects, which is a not yet well covered field of research. To test it on OS projects is a first step.

Gousios et al. (2008) are presenting a model for contribution measurement. They not only take SLOCs into account, but also the activities and events within the repository. Their model weights the contributions of a developer with the impact it has on the repository. They cut the problem of collaboration between repositories by looking at events and changed SLOCs, but do not point out the importance of collaboration. Further they do not provide numbers on, e. g. how many repositories they actually examined or how the repositories are collaborating with each other. On the contrary, I will show that *Patch Flow* exists in the OS world and provide numbers on the extent.

Conklin et al. (2005) is collecting data about OS projects and puts it in an Open Source database. This approach is practical for OS, but not for IS, OS techniques used within corporations. I see the need for a program that even works on repositories not hosted on OS forges. With GitHubble I developed a program which is highly modular and can be used on IS projects in the future.

Jermakovics et al. (2013) developed a collaboration graph for the users of a project by mining the Version Control System (VCS) repository of a project and visualize similarities between the patches. They did not measure the collaboration between the projects. It could be extended to also measure the collaboration among projects by taking into account other projects, but this would only work for OS projects. My thesis also works for IS projects.

Collaboration based on Social Networks

In the following works, researchers describe the collaboration among projects based on the social networks. I use another approach to show collaboration

among OS projects. Thus, I explain how my work differs from the work in the field.

Ohira et al. (2005) tried to set up a database on "who should I ask?" and "what can I ask?" for OS projects on SourceForge and gathered data from their social network. They also store information about the knowledge network among repositories. They aim to help developers to collaborate and share knowledge. In contrast, I compile data on the collaboration between repositories and examine the dependencies on code level, not on knowledge level.

Lopez-Fernandez et al. (2004) mines VCS repositories and tries to get an understanding of the social structure of the repositories by extracting committer information, building a graph and analyzing the graph using social network analysis. I do not rely on the social network for my measurement and analysis, but on the repositories themselves.

German and Mockus (2003) presented their software with multiple modules to crawl VCS logs, ChangeLog files, etc. They can extract all contributors to a software change, but have to crawl mailing lists, ChangeLogs, VCS logs and so forth. In contrast, I just extract data from the commits and connect the repositories by their committers and authors. This allows me to also use the crawler in the field of Inner Source, where, e. g. mailing lists are not common.

2.3 Research Questions

In IS, collaboration between projects is a not yet well researched field. Therefore models for the collaboration between projects are not common. In order to understand the participation of users in different projects, I present a model to describe it in detail. For that reason, I answer the following research questions.

1. To which extent does patch flow exist between OS projects?
2. Which attributes benefit or hinder collaboration among OS projects?

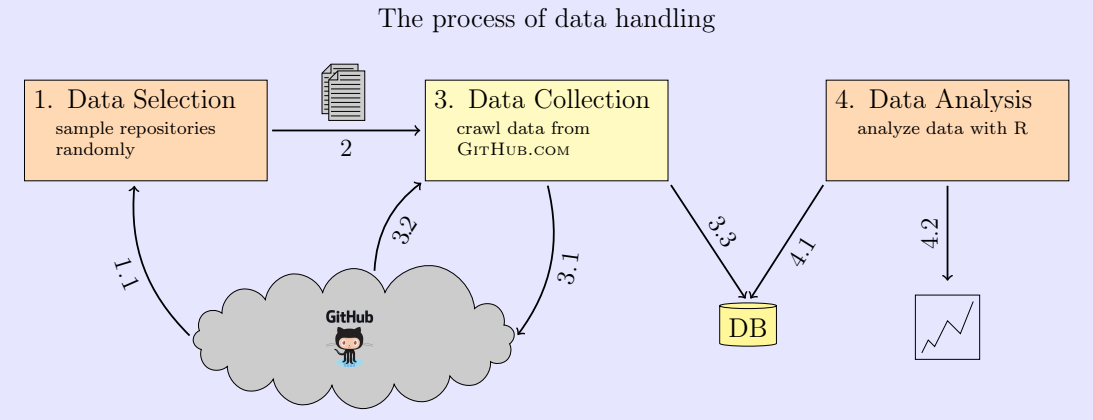

Figure 2.1: The process of data handling: First I sample the repositories I want to crawl, next I crawl the repositories and finally I run statistic evaluations with R.

2.4 Research Method

In order to show how patch flow works and to proof that it exists, I gather data from GitHub.com. In this chapter I describe how I select and collect the data and how I process it to get my results.

2.4.1 Three Steps to the Data

Figure 2.1 visualizes the process of data handling. First I randomly sample the repositories I want to crawl (1). After that, I hand the samples over to the crawler (2). Next the crawler (3) crawls the sampled repositories and stores the resulting data in a database. In the end, I run statistic evaluations written in R (4), a programming language for statistic analysis. In the following paragraphs I will describe how my process works in detail.

Data Selection

My data source, GitHub.com, has a large database, so I have to decide what data I want to examine. I will have a closer look on the last three years (2012 to 2014). To get a representative data set for the forges activity during these years, I sample the repositories randomly.

The number of randomly sampled repositories can be chosen freely. I decide to sample 100 random repositories per month in one sampling run. This is sufficient for this thesis to proof that *Patch Flow* exists. For a larger study this number can

easily be increased. For a broad overview the sampling process can be repeated multiple times, which results in more than 100 repositories per sampled month. The repositories that I sample must fit one criterion: They need to be created in the month that I sample.

The advanced GitHub.com search API has different options to search for repositories. The general options include searching for items from specific users, creation date of items, specific repositories or the programming language of the code. With the repository specific options it is possible to search for the repositories with a certain amount of stars or a certain amount of forks. The search API also makes searching for the size in Kilobyte (KB), the last push date and taking forks into account possible.

With these search options, I can search for repositories, which has its own difficulties. Searching for a specific language would rule out too many repositories, as the language in some cases determines the use cases of the software, e. g. C is common for hardware based programming. Entries of a special user would also set me on a too individual area of software.

The repository specific search for a certain amount of stars or forks serves the most or least popular repositories, which would be reasonable, but I want an equal distribution of all repositories within a month. Even though the size of a repository can be measured in KB, this is not suitable for the search for repositories, because a repository containing only binaries with Gigabytes (GBs) of data might not be as important as the Linux kernel, that has a very high level of collaboration. The last push date is only interesting, when looking for recently active repositories. The most reasonable search criterion is the creation date, which gives me a random collection of repositories within a month.

The sample size is reasonable in regards to performance and memory cost. Also GitHub.com limits the API query replies to a maximum of 100 entries per API request. In each sample run, I generate a random time span between the beginning and the middle and the middle and the end of a month to get a random sample of 100 repositories created within the month. I run the program several times to get a large number of repositories within the examined time span of three years, which gives me a very diverse set of repositories.

Data Collection

To collect data from web based repository forges, I considered three different options: web spidering, database access and access to the data via a standardized web API.

The web spidering approach is not recommended, as web page designs can change

over time and keeping track of all forges I want to crawl is hard. Also database access is hard to obtain from the forges, if not completely impossible, as some forges save critical personal data within their databases and anonymizing the data is a huge effort, which not every forge is willing to make.

Some forges, such as GitHub.com, offer a publicly available API that allows to gather data from their database. In case of GitHub.com, the documentation for the API is well structured. I can easily access the data needed and store them in a suitable format for my research.

Writing a crawler, which complies with the API regulations of each forge and obtains only the data needed is the easiest way for me to maintain. My crawler GitHubble uses the GitHub.com API, processes the given data and stores the repository data in a database. It is modular so that extending it to other forges, e. g. BitBucket, only depends on the quality of their APIs.

Data Analysis

I used the statistical programming language R to analyze the data. After crawling a forge, I filter the data by specific aspects, using SQL statements.

First, I create a mapping, that maps a user to a repository as his/her base project. The reasons are discussed in section 2.6.2. Therefore I select the users by the maximum of changed SLOCs. If a user has multiple projects with the same maximum amount of changed SLOCs, I select the first project in the list as his/her base project.

Then, I filter all commits that are contributed to the own project. For this, I select every commit, where the author to a base project equals the author in a patch. For *Patch Flow*, I am looking for the exact opposite: All patches, that are contributed to a foreign project.

The next step is to select all repositories which contributed to and received patches from other repositories. Therefore I select every target repository with its source repository that is involved in *Patch Flow*. After that I first select the source repositories, count the number of repositories they contribute to and store the data in a table. Then I do the same for the target repositories. After that, I combine the resulting tables into a table, sorted by repository, and add the total amount of changed SLOCs, contributors and commits per repository.

With this information I can then select specific criteria, e. g. if the amount of users affects the number of repositories users from the own repository contribute to.

2.4.2 Extent of the Collected Data

The selection of the data has an impact on its utility, since collecting different data sets leads to different possibilities in interpreting the data. To measure the *Patch Flow* among projects, I select the collected data carefully.

My database contains all projects I crawled. As GitHub.com does not distinguish between repositories and projects, I assume a repository and a project are the same. I discuss the reasons in section 2.6.1 and the consequences in section 2.9.

The data also contains the users who contribute to or are the owner of a repository. I store the real name, username and email address of a contributor. I consider the three tuple of real name, username and email unique, because contributors can have either no username, real name or email address, or only one of them. At the moment I have 48 141 users in my database of which 4 864 have no username. These users all own a repository or have contributed to a repository in the database. Table 2.1 shows a set of data from the database.

id	real_name	username	e_mail
93	Benjamin *****	ben*****	********@gmail.com
1	Ko****	ko***	
3		nw**	
11	De**	de**	github@*******
12		excilys	
35	Ma** ******ps	Ph********ps	**********@gmail.com
42297	root		root@ip-10-179-54-5.ec2.internal
42571	Ol** ****as		ok@**********
52550	cvs		

Table 2.1: Example data from the user table. The names are pseudo anonymized for data protection.

To retain a mapping which user owns which repository, I store the ID of the owner the database gave the user and the name of the repository. Along with that, I store the creation date on GitHub.com and the date the repository was last updated within the database to only update when needed. My database contains 4 366 repositories. An example how the data is stored, can be seen in table 2.2

I also store the data of the patches of the repositories that I inspect. To have a good understanding of whether patch flow exists and if so to which extent, I store the patches with additional data. It contains the commit ID, the SHA-1 hash from Git, its author and committer, represented by the ID of the user in the database, the commit date, lines added, lines deleted, lines changed and the repository the patch belongs to. Table 3.2 on page 45 presents the format the

id	owner	repo_name	creation_date	last_updated_db
11906	20037	DroppyScrollView	2014-10-16 23:16:50+02	2015-05-03 10:52:03.13+02
11907	27199	lenscap	2014-10-16 22:35:04+02	2015-05-03 10:52:07.817+02
11908	13320	manifestreplace-plugin	2014-10-19 01:47:29+02	2015-05-03 10:52:10.533+02
10	10	ECSlidingViewController	2012-01-25 23:54:29+01	2015-05-26 11:24:38.71+02
83	82	orbit.js	2013-07-11 22:52:50+02	2015-05-26 11:24:42.56+02
59	59	cheat	2013-07-30 03:32:12+02	2015-05-26 11:24:45.097+02
89	88	the-little-redis-book	2012-01-23 17:43:02+01	2015-05-26 11:24:47.095+02
21	21	FileAPI	2012-01-17 07:47:39+01	2015-05-26 11:24:55.594+02
9	9	c3	2013-07-18 08:51:40+02	2015-05-26 11:25:07.898+02

Table 2.2: Example data from the repository table.

data is stored in the database. By the end of this thesis the database contained 2 224 396 patches in total.

With the data I collect from the patches, I can derive information about the patches. I can detect where a patch originates by looking at the authors' ID. The changed SLOCs help me to retain this mapping. With the commit date, I can determine if the repository existed before its appearance on GitHub.com.

2.5 The Application GitHubble

I collect my data directly from GitHub.com with my crawling application GitHubble. In this chapter I present GitHubble, describe how it is structured and how the algorithm works.

The following graphic shows an overview over the components of GitHubble and how they are connected.

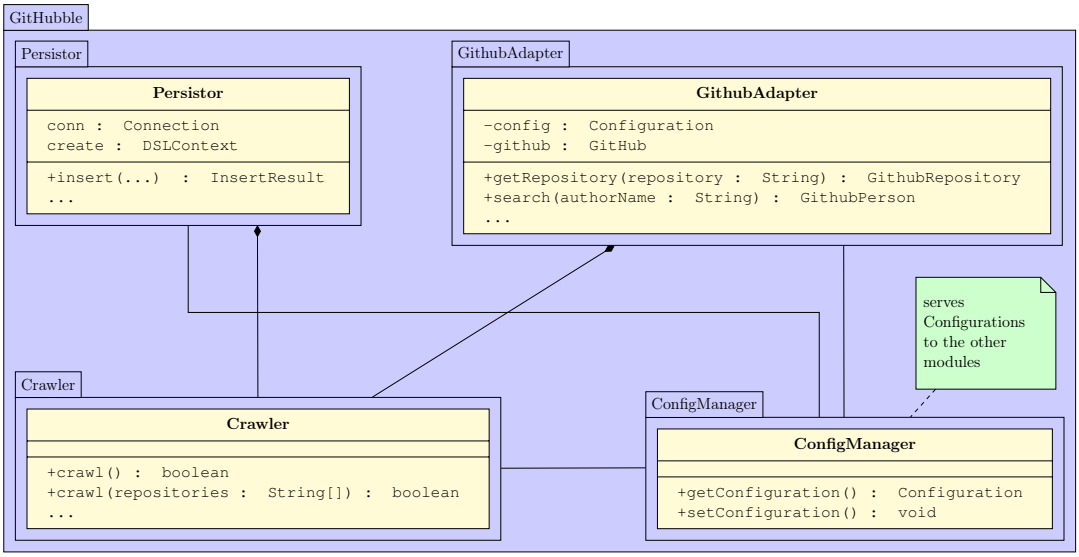

Figure 2.2: GitHubble has four components: a ConfigManager which manages the Configuration for the other components, a GithubAdapter to abstract the GitHub.com layer, a Persistor to model the database and a Crawler which uses the GithubAdapter to crawl GitHub.com and the Persistor to store the gathered data in the database.

The ConfigManager provides a configuration for every module, so every module can work properly. The Persistor is the interface for the database. It provides all needed operations to perform on a database in a simple to use interface. The GithubAdapter provides the interface to GitHub.com and makes it easy and straight forward to crawl GitHub.com.

The Crawler gets instances of all of those modules and crawls the repositories given by the user. The algorithm is very easy to understand: for every repository the Crawler gets the commits and stores the meta information in the database.

2.6 Patch Flow between Projects

The *Patch Flow* is an abstract model of how collaboration between OS projects takes place. In this section I will explain what *Patch Flow* is, how I define it and how I measure it.

2.6.1 Terminology

Within this thesis, some assumptions for the terminology are made for practicality reasons.

Projects are assumed to be the same as repositories, because my research object, GitHub.com, does not distinguish between them. Each repository on GitHub.com can contain other projects, such as *eclipse/jgit* does. *eclipse/jgit* is the repository, but it contains *org.eclipse.jgit.ant*, *org.eclipse.jgit.junit*, etc. In the broader OS world this simplification does not hold true anymore. Bigger projects, e. g. Mozilla, can have multiple repositories, e. g. Firefox, Thunderbird and Firefox OS. Also, in terms of IS, this assumption is no longer maintainable, as in big companies the subdivisions have multiple projects with possibly multiple repositories to maintain.

When talking about SCM repositories, a few other terms are important. There are two groups of people: authors and committers. I choose to call both contributor. Both contribute code and distinguishing between them is only necessary, if the committer is not the author. In the cases, where differentiating between the two groups is necessary, I do so.

2.6.2 The Concept of *Patch Flow*

Every repository consists of files and commits saving the contributions made to the repository over time. These commits contain patches, which are the actual contributions to the files stored in the repository. People who contribute to more than one project are considered to have one project, which I define as their base project. These are the projects, the users identify with. Every other project is a side project. People contributing to only one project got this project as their base project.

To decide which projects are the base projects, I count the SLOCs of a user per repository and define the one with the most changed SLOCs as their base project. It is very likely, that many changed SLOCs are an expression of commitment to the repository. This is important, as the direction of the *Patch Flow* and where a user originates can be determined. The effort to define these terms is only needed

for OS projects. IS projects have all those properties defined by design. Every user belongs to one specific subdivision. Projects and repositories belong to, e. g. another specific subdivision. It is very clear where a patch originates and where it goes to.

Code can flow from one *unit of concern* to the other by a patch. If, e. g. a user *Bob* from *unit of concern I* has some changes for *unit of concern II*, the patch *Bob* contributed will be coming from *unit of concern I*, because *Bob* originates from there.

2.6.3 Definition of *Patch Flow*

With this concept in mind, I can now define what *Patch Flow* is.

> *Patch Flow* is a model to show and measure *code-level collaboration* among different *units of concern*.

In the following parts I will explain the terms *unit of concern* and *code-level collaboration*.

Unit of Concern

A unit of concern can be one out of the following:

- an individual
- an OS project
- an organizational unit

An individual is a project member of a project. The member will have one project he/she originates from. This is the members' base project.

An OS project can be everything from a repository containing the abstract implementation of a website up to the concrete project with the project specific code. Every project may or may not be divided into repositories, which contain code and data.

An organizational unit can be a department of a company, of which one or more members contribute to one or more repositories. For example the Firefox repository belongs to the Mozilla corporation and one of their employees on the Thunderbird team commits to the Firefox repository.

Code-Level Collaboration

It is important to distinguish between code-level collaboration, which is just based on contributions of code, and collaboration based on social networks, which includes co-ordination by a forum or discussing features on mailing lists. Those cover two different fields. One concentrates on social interactions and the other focuses on the patches, the code that is being transferred between repositories.

The focus of this thesis lies on the code-level collaboration.

Every transfer of code between units of concern appears as a patch within the repository. The patch has a few properties, which are interesting for the model of *Patch Flow*, e. g. the author, the committer and the lines of code changed.

To determine in which direction the patch flows, a mapping of user to project is needed. As pointed out in section 2.6.2, in IS this mapping is present by design.

In OS, the mapping has to be created by specific criteria. I can sort by four criteria: most changed SLOCs per user in a repository, owner of a repository, user has commit rights to the repository and sum of commits per user.

As only a few users own a repository, this criterion is not adequate. Even though the owner might be committed to his/her repository, this can not be assumed the default case. I myself own eight repositories on GitHub.com and feel committed to only one. Having commit rights to a repository is a better measure, as users who have commit rights to repositories are often committed to those repositories. But this rules out the users, who only author commits for the repositories I crawl. This happens, when a user only contributes to repositories, where he/she has no commit rights.

Both, amount of changed SLOCs and sum of commits, are good measures, but only the sum of changed SLOCs is telling, because the same amount of changed SLOCs can be achieved by a different number of commits. Also the SLOCs is an expression of work and commitment to the repository.

Knowing from which project to what project a patch flows, a mapping of the source project to the target project is needed. In IS it is easy to do this mapping, as it is clear who works in what subdivision. Again, in OS this needs to be defined. I know which user authored the patch and where the user originates, so I can map the target repository to the repository the author comes from.

2.7 Research Results

In this section I present my results. In section 2.8, I interpret and explain the data and put it into context.

2.7.1 General Data

The database contains 4366 repositories, 48141 users who contributed to those repositories, of which 3624 own at least one repository, and 2224396 patches. Of all users, 10.1% have no username.

2.7.2 Extent of *Patch Flow*

In the following paragraphs I will present the results to the first research question.

Contribution to Repositories

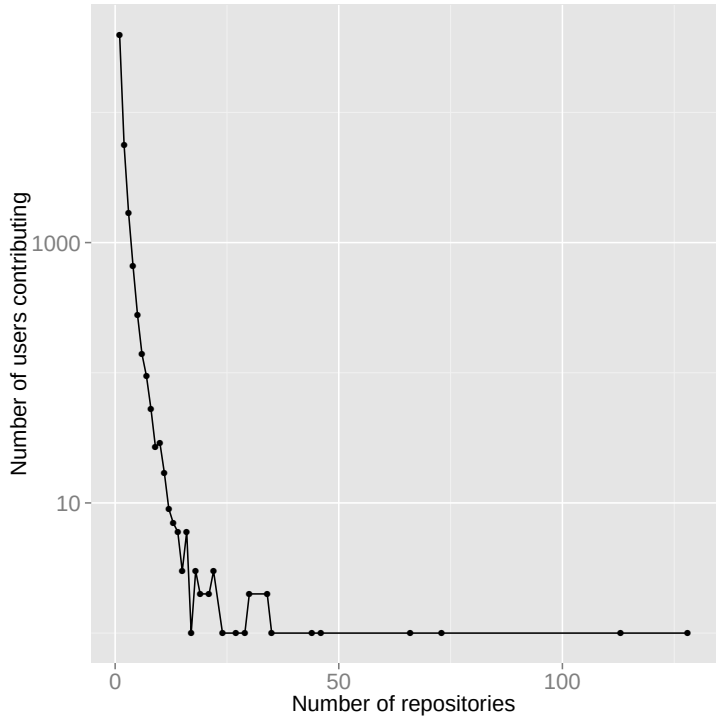

Figure 2.3: How many contributors contribute to how many repositories.

My data shows, that 81.98% of the contributors only contribute to one repository, as figure 2.3 shows. The rest contributes to multiple repositories, but has one project the users identify with, their base project, as elaborated in section 2.6.2.

The data suggests, that 18.02% of all contributors to one repository also contribute to other repositories. The repositories these people contribute to, make up for 74.62% of all repositories.

The patches contributed by *Patch Flow* just change 2.71% of the SLOCs in the projects. As figure 2.4 shows, people contributing to only one repository tend to have less commits per user than users contributing to more repositories.

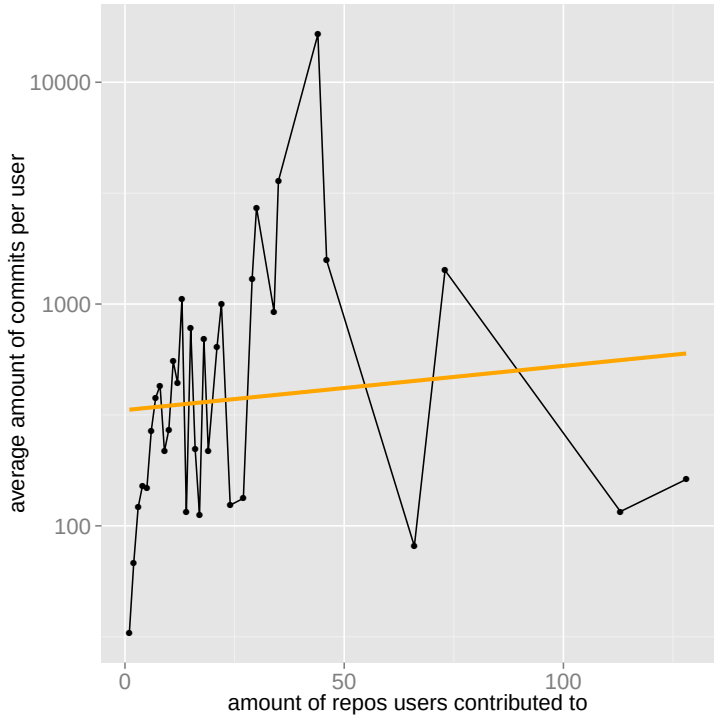

Figure 2.4: Average commits per user, when contributing to how many repositories. The line in orange symbolizes the trend.

According to the data, people are very bound to their base repositories. They commit 43 036 SLOCs per user on average when contributing to their base project. Just a sixth of it, 6 655.9 SLOCs per user, is contributed to the side projects by the users.

Possession of Repositories

Also, 96.47% of the contributors to only one repository are not its owner. Looking at my data, only 7.53% of all contributors own a repository, that they contribute to.

2.7.3 Attributes Influencing *Patch Flow*

The following numbers are the results of the second research question, which attributes benefit or hinder collaboration among OS projects.

When comparing the influencing factors of *Patch Flow*, the fraction of source repositories that transfer patches to other repositories is 87.57%.

I examined four main factors with three minor factors each, which potentially influence *Patch Flow*. The four main factors are: the number of repositories a repository contributes to and receives patches from and the number of patches a repository contributes to and receives from other repositories. Each of these four factors can be divided into three categories, which influence *Patch Flow*: number of patches, number of users and number of changed SLOCs per repository.

Number of Repositories

The graphs in figure 2.5 on page 20 show the influence of the number of repositories a repository contributes to and receives patches from. Graphs (a) and (b) in figure 2.5 show the influence of the number of users per repository on the number of repositories a repository contributed to or received patches from. This minor factor has a strong influence on the collaboration between projects, as shown in the graphs.

The number of commits per repository only has a moderate influence on the collaboration. The trends suggest, that the amount of commits in a repository have a mildly stronger effect on the tendency of contributing to other repositories (graph (c)), than on the tendency of people outside a repository to send patches (graph (d)).

I found, that the number of changed SLOCs per repository has a minimal measurable influence on the number of repositories a repository receives patches from, as graph (f) shows. The number of changed SLOCs, however, has a stronger influence on the number of repositories people from the own repository contribute to, as graph (e) illustrates.

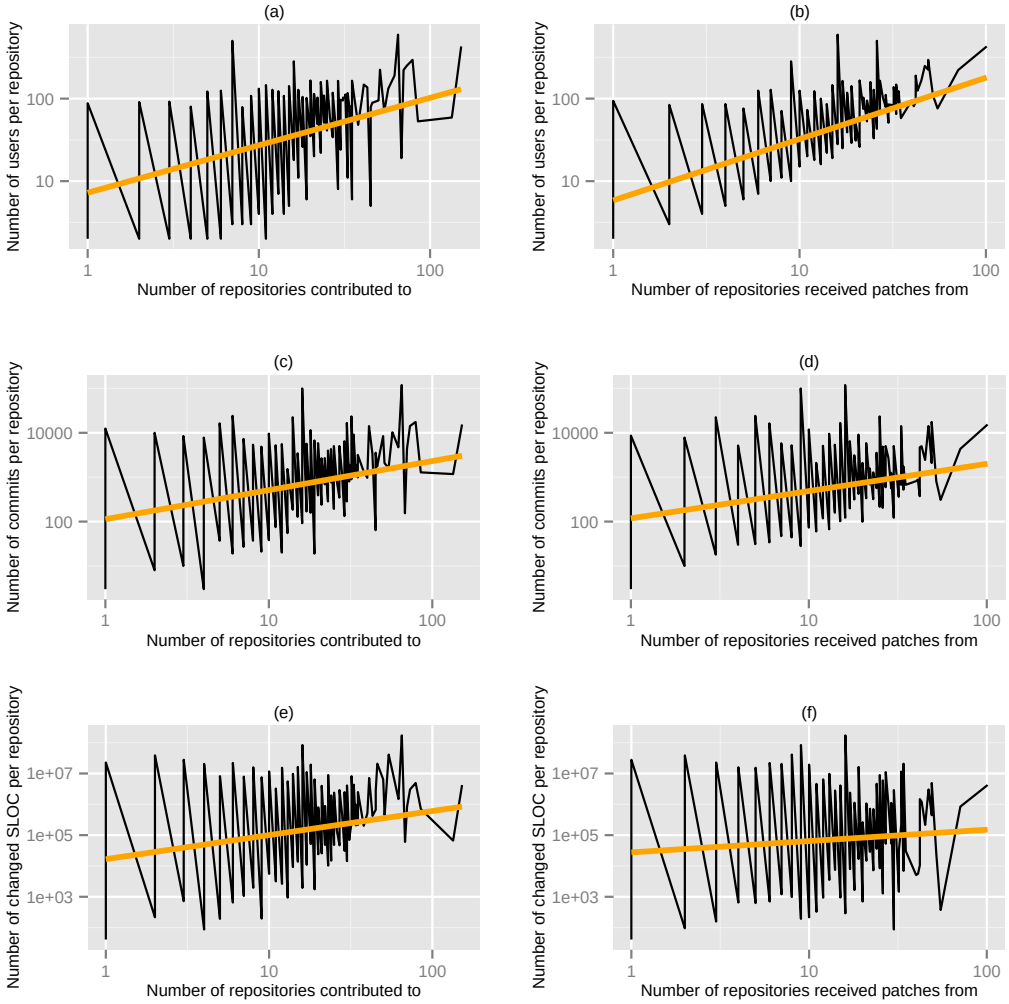

Figure 2.5: Influence of different parameters on the number of repositories people from the own repository contribute to and the number of repositories contributing to the own repository. The line in orange symbolizes the trend.

Number of Patches

As figure 2.6 on page 21 shows, the results are different, when comparing the minor factors to the number of patches a repository contributed to and received from other repositories. Graphs (a) and (b) show, that the number of users per repository nearly have an equally strong influence on the number of patches contributed to and received from other repositories.

While the changed SLOCs did not show a significant effect on the number of commits a repository receives from other repositories (graph (f)), it has a valid

impact on how many patches a repository contributes to other repositories (graph (e)).

Also the number of patches contributed to other repositories is more telling. The number of commits per repository has an equal effect on the number of patches a repository receives from other repositories and the number of patches it contributes to other repositories. The graphs (c) and (d) in figure 2.6 illustrate this.

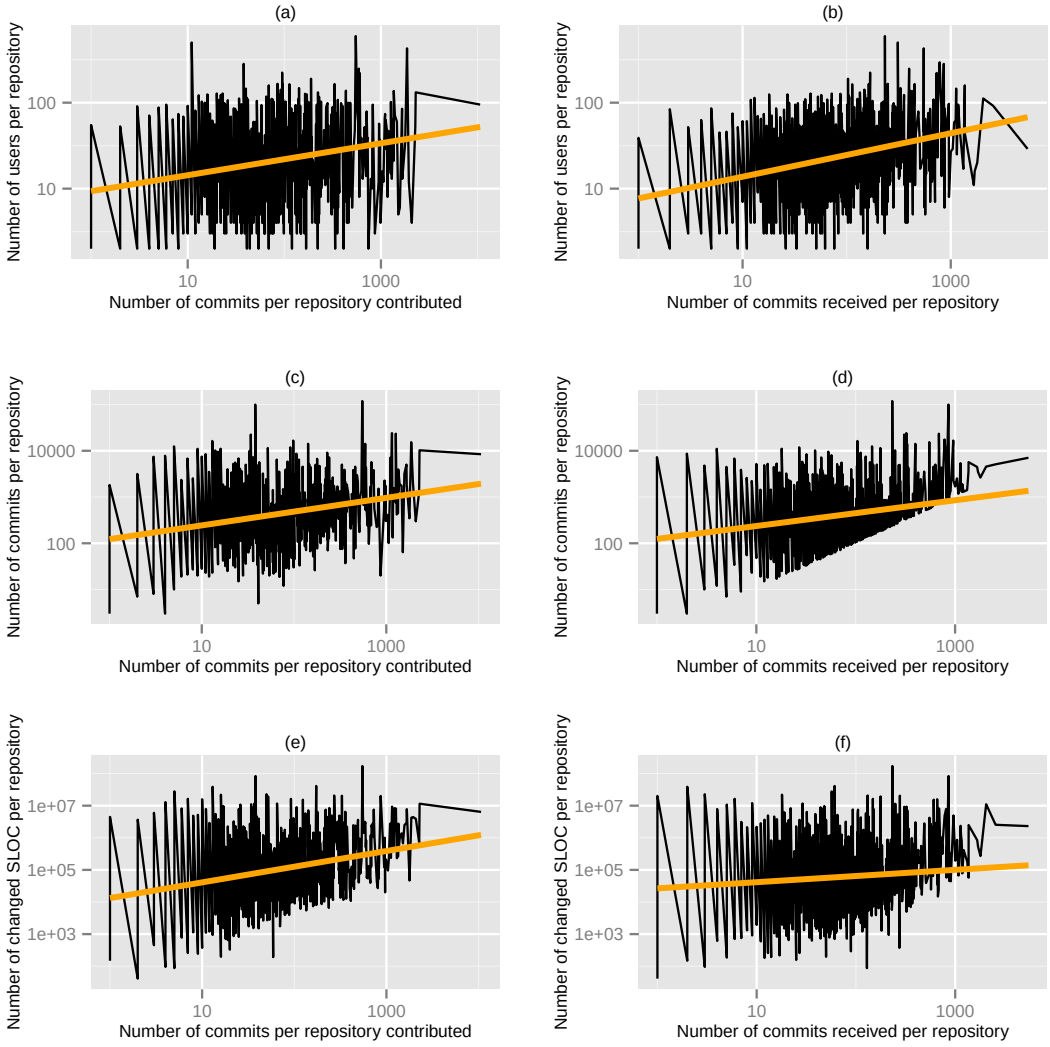

Figure 2.6: Influence of different parameters on the number of patches contributed to and the number of patches received from other repositories. The line in orange symbolizes the trend.

The repositories contributing or receiving the most patches are rarely the same, as

figure 2.7 on page 23 illustrates. The following table shows the top five repositories receiving and contributing the most patches to and from other repositories.

Place	Receiving from (amount)	Contributing to (amount)
1	docker/docker (101)	docker/docker (152)
2	facebook/react (71)	postcss/postcss (134)
3	sindresorhus/awesome-nodejs (55)	eBay/restcommander (112)
4	gulpjs/gulp (52)	coreos/rkt (85)
5	django/django (49)	django/django (79)

Table 2.3: Top five repositories receiving patches from and contributing patches to other repositories

The sample containing the top five repositories with the most received patches shows, that these are the repositories with the most stars on GitHub.com in my database. The repositories with the most contributions to other repositories is a mixed field.

Comparing all influencing factors on contribution among repositories I examined, the number of users contributing to a repository is the one with most influence on the collaboration among repositories.

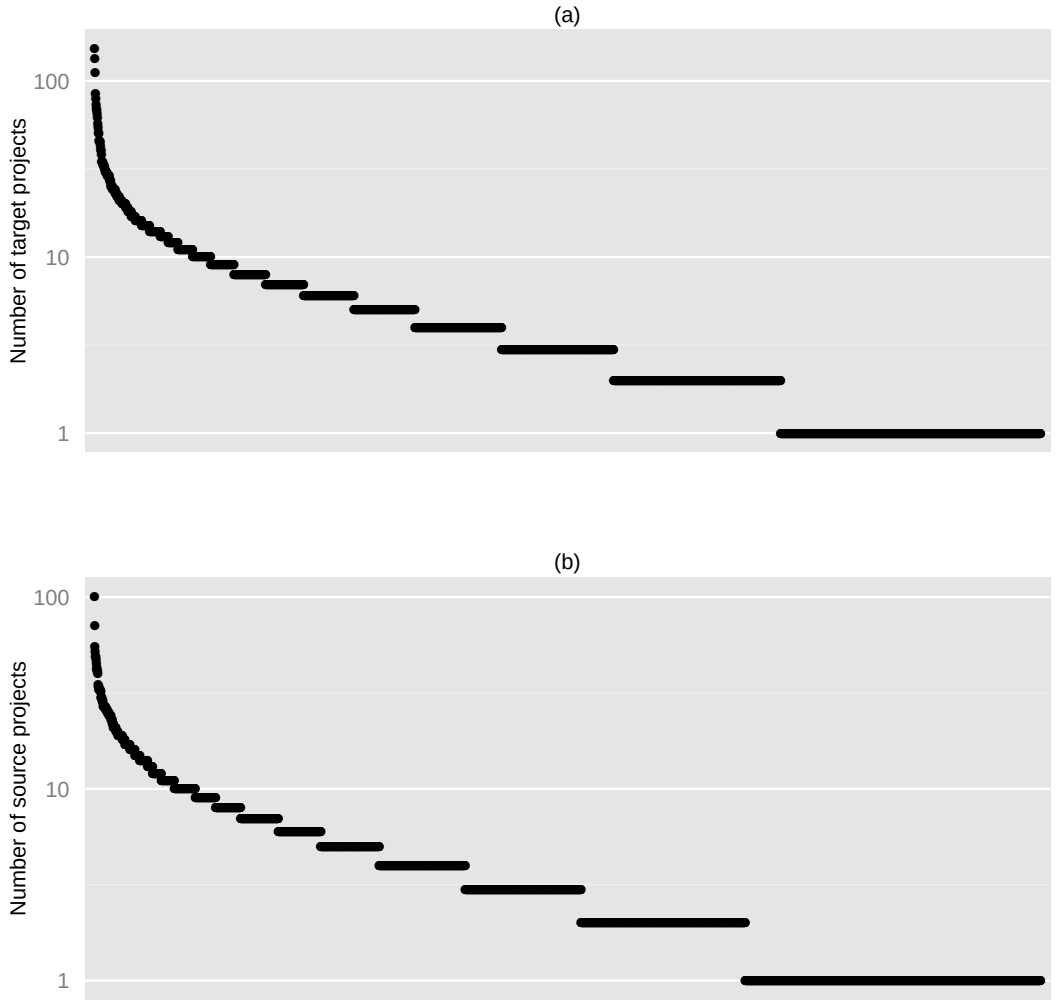

Figure 2.7: Number of repositories contributing patches to and receiving patches from other repositories in my database, sorted by the maximum of repositories contributing patches to or receiving patches from other repositories.

2.8 Results Discussion

The data presented in section 2.7 will be interpreted and discussed in this section.

2.8.1 General Data

Every user in the database can be matched to one person. As pointed out in section 2.7.1, 10.1% of the users in my database have no username. A user can have no username in my database for three reasons:

1. the user does not exist on GitHub.com,
2. the user does exist, but the email used for the commit is not connected to the profile on GitHub.com and there are too many other users with the same real name,
3. the user existed on GitHub.com, but does not anymore.

The first reason is obvious. When a user doesn't exist on GitHub.com I can't provide a username to the database.

In the second case I cannot unambiguously say which of the many people is the right one, so I choose to not select any of them.

In the third case it is difficult. Even though GitHub.com can provide a profile picture for the user, GitHub.com can not provide a profile to the user. Because I rely on GitHub.com to provide me with a profile, I can not get a username for the specific user, that used to have a GitHub.com profile.

Even though the users with no username are only 10.1%, this number is not totally surprising. Not every GitHub.com repository did start as a GitHub.com repository. Some projects, e. g. herbstluftwm/herbstluftwm, had their repository on a private server and migrated the repository to GitHub.com or still use a private Git server and use GitHub.com as a mirror. So the chance, that someone contributed to the repository who does not have a GitHub.com account is given. After the projects migrating to GitHub.com, these contributors may have chosen to use some other software and are not interested in those projects anymore. Getting a username from those people is hardly possible.

The data gathered contains commits from 2001 through 2015, even though I only sampled repositories with a creation date on GitHub.com from 2012 to 2014. That suggests that some repositories were migrated to GitHub.com.

2.8.2 Extent of *Patch Flow*

Contribution to Repositories

These numbers are not surprising. In the OS community the users mostly use the software. When someone stumbles over a bug, he/she files a bug report and a developer deals with the issue. A few people though, apart from the core developers, download the code and fix the bug themselves and send a patch. After that a small part of the people stay on the project to develop it further. The vast majority of those people stick to this single project and identify with it very strongly.

In OS, it appears that few people do very much work and many people do less work. This can be seen in the data, as only 18.02% of all contributors contribute to 74.62% of all repositories.

Even though the amount of SLOCs changed by *Patch Flow* is - with 2.71% - very small, this is expected. *Patch Flow* mostly fixes a bug or implements a small feature. These changes are not very large. Contrary a major feature implementation or the port to a new version of the used programming language can cause major changes and large amounts of changed code. These tasks are normally done by project members. This correlates with my data, as the average amount of SLOCs per user is much higher in base projects (43 036 SLOCs per user) whereas the side projects just get 6 655.9 SLOCs per user on average.

Possession of Repositories

The people contributing to repositories mostly do not own the repository for various reasons. The people use software and, e. g. found a bug and fixed it, implemented a feature or did some refactoring and sent in a patch. This is a very common phenomenon in the field of OS and does not require the ownership of a project.

2.8.3 Attributes Influencing *Patch Flow*

When comparing the influencing factors of *Patch Flow*, the fraction of source repositories that transfer patches to other repositories involved in *Patch Flow* is 87.57%. This is a surprisingly high number. It means, that the community is very active and *Patch Flow* is a very important issue regarding collaboration.

Number of Repositories

The graphs (a) and (b) in figure 2.5 on page 20 show the influence of the number of repositories a repository contributes to and receives patches from. The strong influence of the number of users per repository on the contributions is not surprising: The more people a repository has, the more people will contribute to other repositories and their own repository, because it has a certain importance. Also more people can do more development jobs.

Even though one might think that the number of commits in a repository is a strong indicator if a repository is important or not, and therefore attracts more users to contribute, this is only a moderate factor. As the graphs (c) and (d) show. The number of commits are often seen as the main factor to determine, if a repository is important, which is not maintainable with these results.

The graphs (e) and (f) suggest, that the amount of changed SLOCs per repository has the least significance for the tendency of people from outside contributing to the own repository or people contributing to other repositories. This is surprising, as, along with the amount of commits, the changed SLOCs within a repository usually suggest the importance of a repository. With the data gained, I can disprove that. The reason is simple: important repositories can have small amounts of commits and changed SLOCs and large, unimportant repositories can have many commits with many changed SLOCs. This indicator is not defensible with my data.

Number of Patches

Figure 2.6 on page 21 shows, the results are different, when using the commits a repository contributes to or receives from other repositories as main factors.

The influence of the amount of users per repository also has a strong influence on the number of patches a repository receives and contributes, as graphs (a) and (b) illustrate. This is, as for the number of repositories, not surprising, as more people can do more and different work and can contribute to other repositories.

While no significant influence on the number of received patches per repository by the changed SLOCs can be explained by the fact, that also small repositories can have many changed SLOCs, the measurably stronger influence on the contributed commits per repository is the effect of many people in a repository with many commits and changed code. This phenomenon can be seen in graphs (e) and (f).

Equally strong influence on the number of commits received from and contributed to other repositories has the number of commits per repository, as the graphs (c) and (d) show. This is not surprising, as the number of patches a repository

receives or contributes is dependent on the attention people pay to the repository. Repositories with many commits tend to attract more attention than repositories with less commits.

That the repositories contributing or receiving the most patches are rarely the same, is also less surprising, because the repositories contributing many patches to other repositories hold much knowledge and share it. The repositories with less knowledge receive patches from the repositories with more knowledge. This explains just one part of the phenomenon. The other part of repositories receiving patches are simply the popular ones as table 2.3 shows.

2.9 Threats to Validity

In this section, I will explain the limits of my work and the consequences I have to take into account.

Forks

On GitHub.com many repositories have the same name and mainly the same content, even patches with the same commit IDs, because people fork repositories. This creates a personal branch of a repository, which results in a repository for the person who forked a repository. In this thesis I do not distinguish between forks and normal repositories. This might lead to duplicate data and, more importantly, false assumptions. But looking at forks, these repositories express the will to contribute to a project even without commit rights or to customize a project. So the distinction between a repository and a fork is not necessary.

Direction of *Patch Flow*

With my mapping of users to repositories I can tell from which repository to which other *Patch Flow* exists. One problem remains: The direction can not always be determined clearly, because sometimes a patch is transferred from another repository that a user contributed to, than his base project. These cases I can not represent when fitting the data into my model. I also could determine the direction of *Patch Flow*, when not using a base project, but I lose the ability to tell, where a user originates. The direction of *Patch Flow* is not as important as the fact that *Patch Flow* actually happened. So this is negligible.

Mapping Repository to Project

On GitHub.com, every repository represents a project. This fact is represented in my mapping of user to repository, when I assign a user to a repository as his base project. Multiple users can be assigned to one repository as their base project. This simplification can be made, as the GitHub.com organizations represent organizations and not projects or organizational units. I have no doubt the GitHub.com organizations are misused to also represent projects. To have a proper mapping of user to project, the GitHub.com organizations are worthless. The reason is very simple: not every project uses an organization to organize their members.

The Dataset

An exact model is not feasible when using the complete dataset. I chose to crawl a reduced dataset, that has a small footprint, i. e. uses a minimum amount of GitHub.com API requests and stores only relevant data. With my dataset, I am able to draw many conclusions, although I have very little data for each project or patch. Therefore my dataset has a reasonable format. The dataset I chose is also very informative, as no other data is necessary for the conclusions drawn in section 2.8.

Mapping User to Project

When mapping a user to a project, this is errorprone by design. A user can identify with a few projects, as he/she can contribute nearly equally to each of them. This is not represented in my mapping of the users to projects. The reason I need this mapping is the determination of the direction of *Patch Flow* and where a user originates. But as mentioned before, this is not necessarily needed to actually show that *Patch Flow* is happening.

2.10 Conclusions and Future Work

2.10.1 Conclusions

With the presented model of *Patch Flow* it is possible to show collaboration among OS SCM repositories. The crawler GitHubble collects data from GitHub.com and stores the data into a database. From the database it is easy to analyze the data based on the model of *Patch Flow*. This thesis shows, that *Patch Flow* exists in the OS world and is a common phenomenon.

2.10.2 Future Work

In the future, some more modules will be implemented to also use the crawler on IS repositories and other SCM tools than Git. Also the application will be separated into a server and a client application.

3 Elaboration of Research

3.1 The Application GitHubble

Automating the process of data collection is essential when crawling big amounts of data. Fortunately, GitHub.com offers a great and well documented API, that allows to collect exactly the data a researcher is interested in. This section further explains how GitHubble, my application to crawl GitHub.com, works, what the challenges in implementing such a program are and what the limits of GitHubble are.

3.1.1 The Design

When designing GitHubble, I made sure to make it as modular as possible, as using the Crawler for OS projects was only to proof the concept of *Patch Flow*. The future goal is to analyze IS projects. This requires some other modules, e. g. for SVN.

In section 2.5, I present an overview over the modules of GitHubble. The modules are the ConfigManager for configurations, e. g. information for authentication and path variables, GithubAdapter for everything regarding GitHub.com and the Persistor for everything database related. The Crawler uses the three modules to crawl the given repositories. The application calling the Crawler is GitHubble, which initializes the Crawler and calls the respective functions from the Crawler to begin the crawling.

ConfigManager

In figure 3.1 I show the ConfigManager that manages the configuration for all the modules. Every module receives information from the configuration. The Persistor, e. g. gets information about what username, server and password the database has, the GithubAdapter where to clone the repositories to, etc. The configuration is a simple text file containing the configuration options line by line.

GithubAdapter

Within the GithubAdapter I encapsulated all functionality related to GitHub.com and its repositories. In the following paragraphs I will describe the parts of this module in detail.

The GithubAdapter class is the interface of the functionality of GitHub.com to the other modules, in this case the Crawler. It serves the repositories,

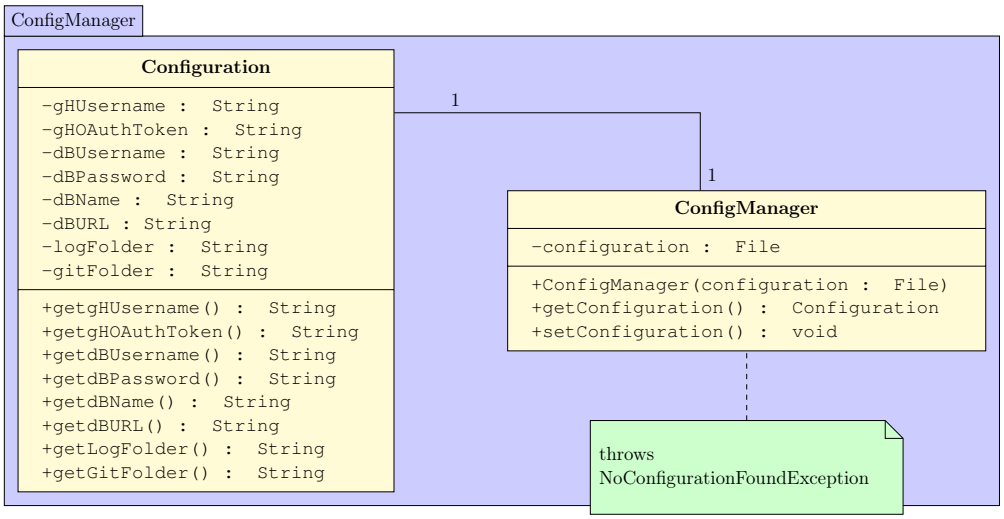

Figure 3.1: The ConfigManager manages the authentication and path variables.

provides the ability to search for users, holds the amount of remaining API requests and so on.

GitHubble works on the *GithubRepository* class. It clones the repository from GitHub.com to the local file system on initialization and deletes it on destruction of the object. Furthermore, it serves the commits from GitHub.com when I need them to extract data. A *GithubRepository* also contains the *LocalCommit*s from the repository cloned to the local hard drive. The *LocalCommit*s offer the possibility to extract the committer, the author and the lines added, changed and deleted much faster as if I used the GitHub.com API for each commit, as the API requests are limited. I explain the model of rate limiting of GitHub.com in section 3.1.2.

A *GithubCommit* contains everything that a commit from GitHub.com contains. From the username and real name of each, the author and the committer, up to the deleted and added lines of code.

The *GithubCommitRange* is the range of commits I want to get as a batch. This saves memory and works as fast as just looping over all the commits themselves.

With a *GithubPerson* the actual user on GitHub.com is represented in my class design. It serves the username, real name and email address. When crawling, I insert every new user into a database. In the Crawler I use a fuzzy matching algorithm to insert only unique users into the database, described in section 3.1.4.

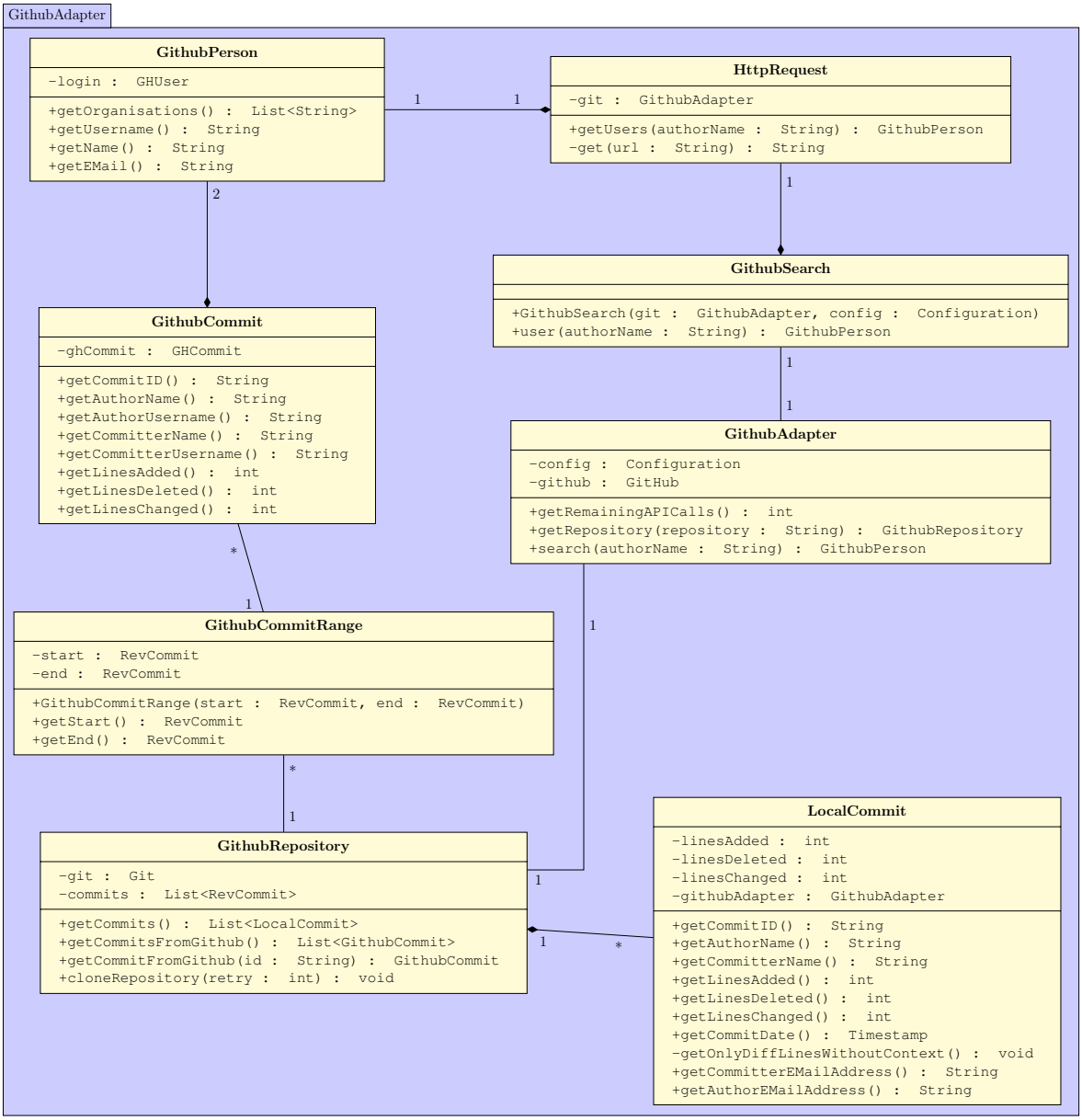

Figure 3.2: The GithubAdapter represents the functionality needed to access the data from GitHub.com.

To get as many complete data sets of the users as possible, I implemented the *GithubSearch* to search for a users username on GitHub.com. This is necessary, if the real name and email used in the commit are different from the GitHub.com profile.

To perform the API GET requests in the *GithubSearch*, I set up an *HttpRequest* class, which performs the requests and delivers the results of the API request.

Persistor

On the Persistor, that I show in figure 3.3 I can perform every necessary database operation needed by the Crawler. Most components of the *GithubAdapter* have a corresponding class within the Persistor, to have a proper representation of the entities on GitHub.com.

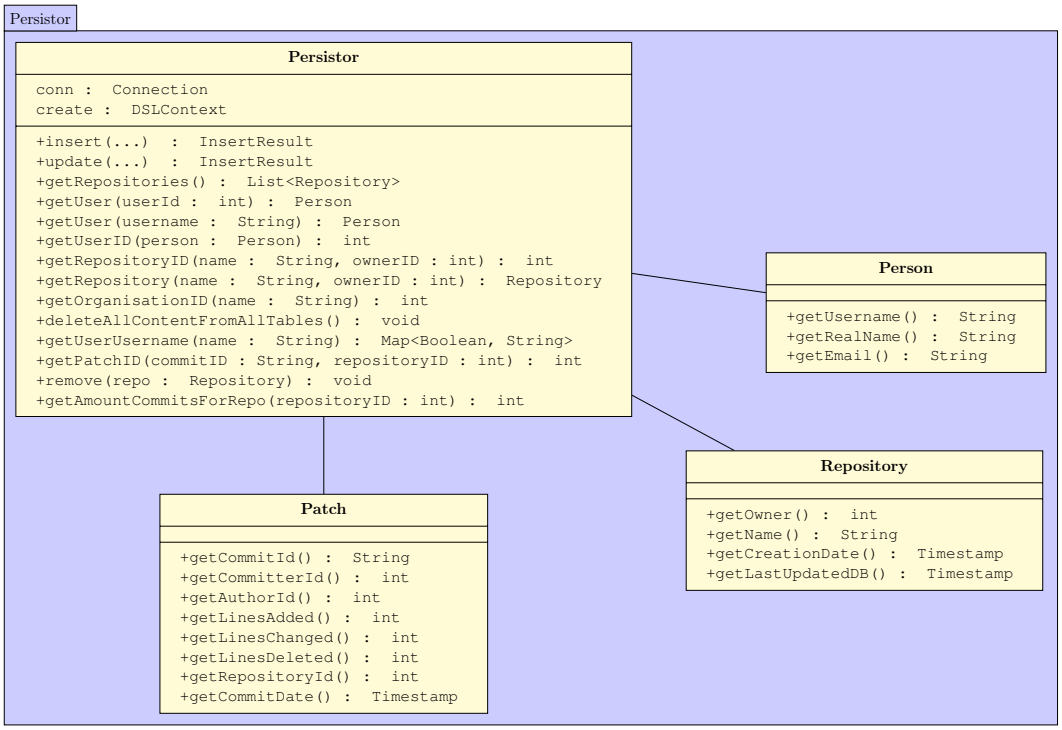

Figure 3.3: The Persistor serves and stores data in the database.

Crawler

The Crawler combines the functionality of the Persistor and the *GithubAdapter*. Here I insert users and repositories into the database, crawl the repositories for patches, put the patches into the database and use my algorithm described in section 3.1.3 to efficiently extract the needed data from the repositories using the configuration from the ConfigManager and the functionality from the *GithubAdapter*. I sotre the extracted data in the database with the Persistor.

I have two functions inside my Crawler to crawl repositories: One to crawl repositories already in the database to update the repositories and a second to crawl repositories, that are not in the database, yet. Both methods use one procedure for the crawling mechanism, but use different approaches to get the repositories to crawl.

In order to crawl the repositories already in the database, I get a list of all repositories in the database and crawl them sequentially. The method, that crawls repositories not in the database yet, gets an array of Strings, each representing the full repository name of a repository on GitHub.com. The Strings are taken and the represented repositories from GitHub.com are inserted into the database. After that, the repositories are appended to a list, which is given to the method to crawl the repositories.

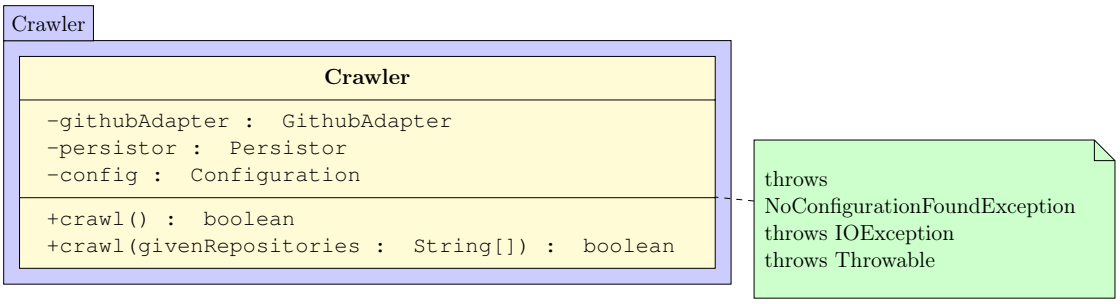

Figure 3.4: The Crawler uses the GithubAdapter and the Persistor to crawl GitHub.com and store the gained data in the database.

GitHubble

GitHubble is the command line application, which either takes user input, processes it and forwards it to the Crawler to crawl, or instructs the Crawler to get the repositories from the database and crawl those.

3.1.2 GitHub.com API RateLimit

With GitHub.com being one of the most popular developer platforms of the present, the operators face an ever growing amount of API requests. To properly handle all those requests, GitHub.com limits the amount of requests, depending on what API functionality is used and whether authentication is used or not.

Using unauthenticated access, 60 requests per hour for normal API requests are the maximum a developer can submit. Using authenticated access, the limit is 5000 requests per hour. The search has its own rate limit with 20 requests per minute for the authenticated user, and 10 requests per minute for the unauthenticated user.

Exceeding the rate limit multiple times causes the application to be "temporarily blocked"[1]. Exceeding the rate limit, causes the application to wait until the hour is passed to send the next request. That is one of the reasons why I decided to clone the repositories to the local hard drive and to crawl the repositories locally. The other reason is, that I am able to crawl more repositories within the 5000 API calls. The amount of API calls I need per repository dramatically drops from several thousand to way under one hundred for normally sized repositories to several hundred in exceptional cases. The latter cases contain, e. g. repositories where many people with unconnected profiles commit code. In those circumstances I need more API calls to get all information I need, than for normal repositories. The reduction of API calls allows me to crawl constantly, without waiting too long between the repositories.

3.1.3 The Algorithm

To crawl GitHub.com with a minimum amount of API requests per repository, which allows me to crawl more repositories within the limited amount of API requests, using a simple but efficient algorithm is necessary.

Figure 3.5 on page 38 visualizes the algorithm. First I initialize the Crawler with all necessary configuration data. After that, I get the repositories I want to crawl either from the database or from the parameters the user provided on the command line. If I get the repositories from the command line, at this point I insert the repositories into the database. The following steps are executed for every repository in the database.

The next step is to clone the repositories from GitHub.com to the local hard drive. This saves me most API requests, which are used to collect the users' or the commits' data. After that I get a list of all the commits of the current repository. To save memory I get the commits in batches of 100, process these 100 and get the next 100 repeatedly. Afterwards I process the commits by extracting all necessary information from the repositories and inserting it into the database. After that I completed one repository and delete it from the hard drive to save space.

[1] GitHub.com does not tell the user how long the application is blocked, but from my experience it is the remaining time until the hour or minute has passed and the RateLimit is reset.

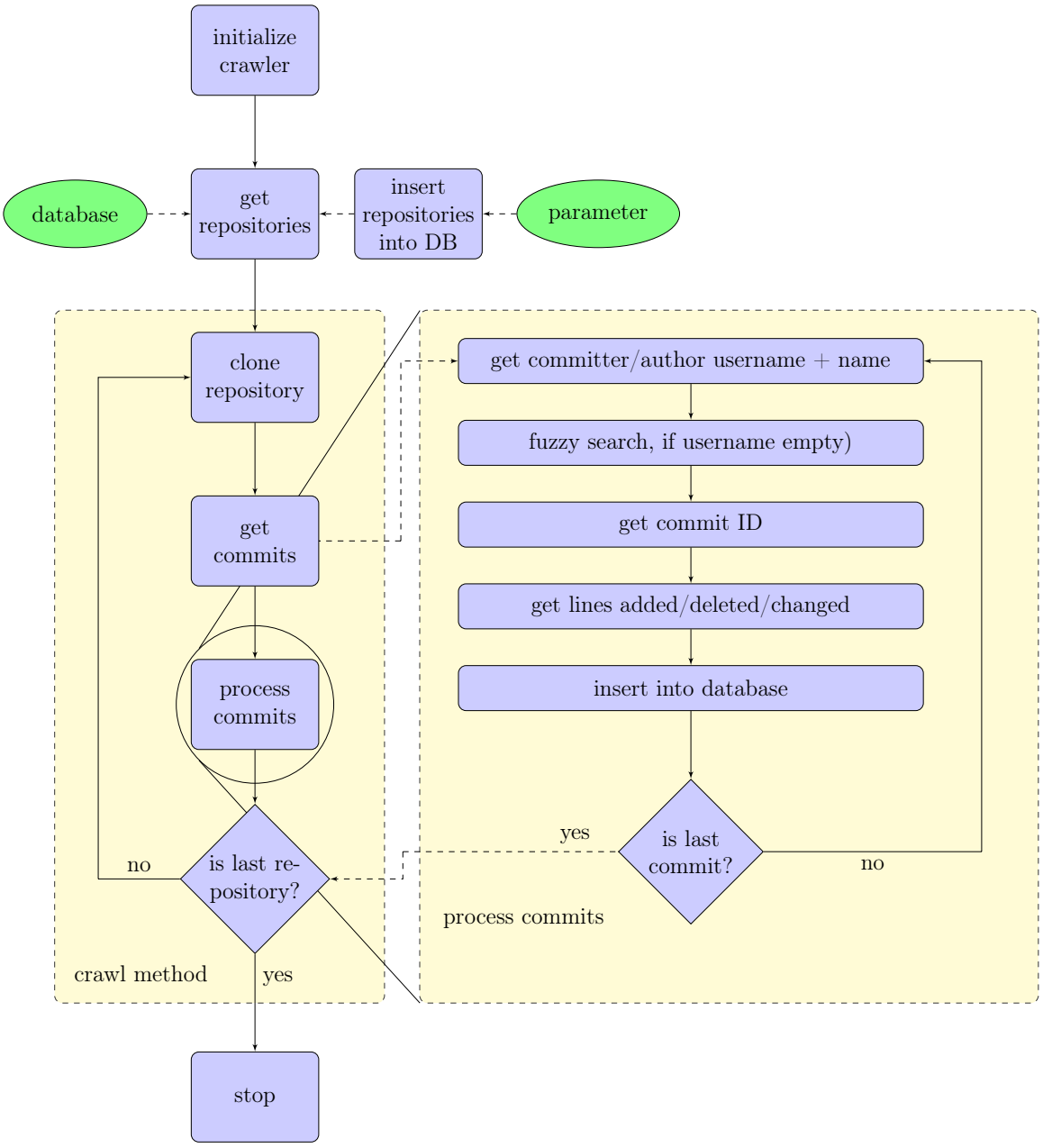

Figure 3.5: First I initialize the crawler. After that, I get the repositories I want to be crawled either from the database or the parameters the user provided on the command line. The next steps are to clone the repository to the local harddrive and get a list of all the commits. Afterwards I process the commits, get all necessary information from the repositories and restart the cycle.

3.1.4 The Challenges

When implementing a crawler like this, the programmer faces multiple challenges. Some are easy to solve, some are very subtle and some will cause the program to fail. The titles of this section are the assumptions I made, while implementing that prooved to be false. The challenges are user identification, API request limit and external libraries.

The username on GitHub.com is unique

Signing up to GitHub.com, everyone has to have a unique username to contribute to projects on GitHub.com. This holds true for all users on GitHub.com. Representing this fact in a database, a developer would first try to make the username in the database unique. An entry for a person could look like this:

username	**real name**	**email**
johndoe	John Doe	john.doe@example.com
siegfried	Roy	siegfried.roy@thetiger.com

The challenge begins, when crawling a repository, which was e. g. migrated from a private Git server to GitHub.com. In this case, it is likely to get a contributor who is not member of GitHub.com, thus has no username on GitHub.com.

At this point, there still is an easy solution: insert the user with the real name and email provided by the Git repository. After inserting the new user without a username our table looks like this.

username	**real name**	**email**
johndoe	John Doe	john.doe@example.com
siegfried	Roy	siegfried.roy@thetiger.com
	Ronald McChutney	ron.mcchutney@peril.de

Continuing, I must assume that the repository may contain other users without usernames. Bearing in mind the database has a unique condition for the username, all such inserts must fail, as there already is a user with no username. I assigned the empty, unique username to the first ever commit without a username, erroneously blocking all other inserts of users without a username.

Thus a commit, which has a user with no username as author and/or committer gets the first, and only, user with no username as author and/or committer.

To prevent this, the primary key must be a three tuple of username, real name and email, where only real name and email are unique. This prevents inserting the exact same user with the same credentials over and over again. This, however, does not prevent inserting the same person with different credentials multiple

times. The developer has to think of a sensible way to check if the person exists in the database and merge his identity.

Users use a consistent profile

Contributing to repositories on GitHub.com, users need to connect their profile with an email address, so GitHub.com can connect the commit with the profile. This email address can be chosen to be invisible to the public. On GitHub.com users can submit multiple email addresses to connect with their profile, so commits from different computers, e. g. business computer with the business email and the private computer with the private email, are connected to the same profile.

In many cases the users just forget to add the email address they use on one of their computers to their profile on GitHub.com. A few users however just refuse to do so, because the system allows it. GitHub.com can not map those users to the profile they have. I developed a heuristic to map those users to their profiles.

When a user has no username in the first run, I try to get a user, with the same email and real name from the database first. If that fails, I try to get the user by its real name from GitHub.com. If that also fails, I resign and insert the user to the database without a username.

The email field in a Git repository contains a valid email

To contact a committer or author, a Git repository also contains a field for an email address, which should contain a valid email address.

The idea of compensating the non-uniqueness of the username with the tuple of email and username is only a partially valid assumption. Most of the users contributing to a Git repository submit a valid email address. Unfortunately Git has no requirements on how the email address has to look like. So the field containing the alleged email can contain whatever string comes to mind. Thus, using this field for any valuable information is not possible.

The name field contains the real name of a user

Every Git repository also contains a field for the real name of the committer and the author. But here also every string that comes to mind can be inserted. So everything from an email address or a username up to nothing or even a real name can be found in this field. Using this field for a first orientation who the

person might be, can be useful, but challenging as the name can be nonsense. GitHubble just inserts it into the database.

Reaching the API-Rate-Limit

As pointed out in section 3.1.2, GitHub.com limits the amount of requests a user can submit. On the first glance it seems to be hard to reach the API rate limit, that GitHub.com grants with OAuth authentication. But actually it is very easy to exceed it. Using only GitHub.com and the API for information collection, the rate limit can be exceeded within a few minutes, which causes waiting until the hour has passed to send the next requests.

A valid solution to this problem is to clone the repository to the local computer, crawl the repository locally and just look up the parts in question, i. e. user information.

Using external libraries

To have a standardized way of implementing requests to the GitHub.com API or a local Git repository, using external libraries is the standard approach. When using libraries, a developer usually has to deal only with the errors the library throws or the values it returns. Sometimes, though, it is a bit more complicated.

A developer would assume, that a library for handling Git repositories is capable of cloning repositories reproducibly. In case of JGit this does not hold true. Even though it is one of the best Java implementations for Git repository handling, it is not capable of reproducibly clone a repository. This just partially is JGit's fault. The other part plays Java itself.

Java seems to have a problem when allocating large objects that need data from the internet. When cloning a repository with JGit, the developer has to pay attention that the repository gets cloned correctly. Usually, when offered such a functionality by a library, the library takes care of the correct handling of errors and corresponding side effects.

The problem is though, that a minimal example always works, but in the field with any kind of data the so called "inflater" for objects retrieved from the network gets closed every now and then. The workaround is, to repeat the process of cloning for a specified amount of times and to give up after that. Looking at most network protocols this is good practice in the field. I filed a bug report against JGit[2] and Java[3] itself.

[2] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=463007`
[3] `http://bugs.java.com/bugdatabase/view_bug.do?bug_id=8080363`

3.2 Acknowledgement

At first, I want to thank Prof. Dirk Riehle and M.Sc. Maximilian Capraro for the possibility to write a thesis with a strong focus on implementation. It was a great experience to write code in a language (Java) I normally don't use.

Further I am very thankful for all the people, who proof-read this thesis multiple times and had a great impact on comprehensibility and linguistic expression. Namely, in random order, Sybille Hötzel for linguistic expression, Christoffer Löffler for both, linguistic and technical expression, and comprehensibility and Conrad Meier for a technical view from outside the field of Computer Science. Thank you!

3.3 About the Author

Manuel Zerpies was born on January 11th, 1987 in Nuremberg, Germany. He graduated the Leibniz-Gymnasium in Altdorf bei Nürnberg, Germany in 2006 with advanced cources in chemistry and history.

From October 2006 to September 2009 he studied Material Science at the Friedrich-Alexander Universität Erlangen-Nürnberg. He then changed to Computer Science in fall 2009. In late 2011 he began working as a tutor for "Grundlagen der Informatik", a course for all non CS students at the Technische Fakultät. In January 2013 he finished his Bachelor of Science in Computer Science and enrolled for Master of Science.

In April 2012 he joined the VAMOS research team an wrote his Bachelor Thesis in the field. He continued his work in 2013 as a research student until the end of the year. In October 2014 he began working on his Master Thesis at the chair for Open Source Software at Friedrich-Alexander-Universität Erlangen-Nürnberg.

Appendix A Tables

Number of users contributing	Number of repositories
39468	1
5622	2
1698	3
660	4
278	5
139	6
94	7
53	8
27	9
29	10
17	11
9	12
7	13
6	14
3	15
6	16
1	17
3	18
2	19
2	21
3	22
1	24
1	27
1	29
2	30
2	34
1	35
1	44
1	46
1	66
1	73
1	113
1	128

Table 3.1: How many people contribute to how many repositories.

id	commit_id	committer_id	author_id	lines_added	lines_deleted	lines_changed	repository_id	commit_date
1	9e167c5ed9c24 11723a444ff28b 2ccb23817de88	699	699	3012	0	3015	1	2013-08-17 09:15:57+02
2	bbbbfcc37a9a2 423d541a13f85c 6a9481bba177c	699	699	3	6	9	1	2013-08-17 09:33:28+02
835	0c3d1e84bb59f 04ec8a855fa7d5 fd96c4202dcf9	870	870	66	4	70	3	2012-09-08 08:28:48+02
836	b462151860160 691b315abf754f 66075c07a650b	870	870	6	18	24	3	2012-09-08 10:26:02+02
837	147af8a438b9f c367413bcc2349 6065c7ecb50f5	870	870	17	17	34	3	2012-09-08 13:18:28+02
6990	bcbc3866b4084 7e7e14fee7b71d 2c201fbcf0cec	1269	8	2	4	6	8	2012-02-27 21:56:41+01

Table 3.2: Example data from the patch table.

References

Conklin, M., Howison, J. & Crowston, K. (2005). Collaboration using ossmole: a repository of floss data and analyses. In *Acm sigsoft software engineering notes* (Vol. 30, pp. 1–5).

German, D. & Mockus, A. (2003). Automating the measurement of open source projects. *Proceedings of the 3rd workshop on open source software engineering*, 63–67.

Gousios, G., Kalliamvakou, E. & Spinellis, D. (2008). Measuring developer contribution from software repository data. In *Proceedings of the 2008 international working conference on mining software repositories* (pp. 129–132).

Gousios, G. & Spinellis, D. (2012). Ghtorrent: Github's data from a firehose. In *Mining software repositories (msr), 2012 9th ieee working conference on* (pp. 12–21).

Heller, B., Marschner, E., Rosenfeld, E. & Heer, J. (2011). Visualizing collaboration and influence in the open-source software community. In *Proceedings of the 8th working conference on mining software repositories* (pp. 223–226).

Jermakovics, A., Sillitti, A. & Succi, G. (2013). Exploring collaboration networks in open-source projects. *Open Source Software: Quality Verification*, 97–108.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M. & Damian, D. (2014). The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories* (pp. 92–101).

Lopez-Fernandez, L., Robles, G. & Gonzalez-Barahona, J. M. (2004). Applying social network analysis to the information in cvs repositories. In *International workshop on mining software repositories* (pp. 101–105).

Ohira, M., Ohsugi, N., Ohoka, T. & Matsumoto, K.-i. (2005). Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. *ACM SIGSOFT Software Engineering Notes*, *30*(4), 1–5.

Robles, G., González-Barahona, J. M. & Ghosh, R. A. (2004). Gluetheos: Automating the retrieval and analysis of data from publicly available software repositories. In *Proceedings of the international workshop on mining software repositories* (pp. 28–31).

Tymchuk, Y., Mocci, A. & Lanza, M. (2014). Collaboration in open-source projects: myth or reality? In *Proceedings of the 11th working conference on mining software repositories* (pp. 304–307).