Free University of Bolzano
Faculty of Computer Science

OSSQuery

A Data Mining System for Investigation on Cause - Effect Relations in OSS

Author
Manuel Piubelli

Supervisor
Prof. Barbara Russo

Co - Supervisor
Dipl.-Inf. Maximilian Steff

Thesis submitted for the Bachelor of Science in Applied Computer Science
July 2011

Acknowledgements

Many people have contributed to this work and to these last three years of study. It is now my great pleasure to thank them.
Foremost, I thank Prof. Barbara Russo for introducing me to the world of research and guiding me through this project.
Moreover, I thank Maximilian Steff, who assisted me whenever I was stuck or needed help during the last six months.
I would also like to express my gratitude to my family, that is my parents Maurizio and Beatrix for their motivation and support during my studies and my life, and my sister Jasmin for making excellent cake and coffee.
Finally, I thank my friends Corneliu, Marco, Patrick, Raphael and in particular my girlfriend Sara for making the time of my studies that much more enjoyable.

Manuel

Abstract

The availability of data on open source projects increased dramatically throughout the last decade. This tendency attracts the attention of many researchers who focus their attention on different software attributes and on how these affect each other. Commonly, such research involves the analysis of individual or small groups of projects and thus provides results that are strongly biased by the choice of these projects. To exploit the full value of the available information, it is necessary to consider at least a representative proportion of the universe of open source software. Such a universal set would allow us to draw generalizable conclusions about software development and the development process. These can both serve as a benchmark to re-evaluate current beliefs as well as build new knowledge valuable to both industry and academia.

A quantitative investigation of open source software requires the automated collection and examination of large amounts of data. Such automation is fraught with problems, though. Project data such as source code, versioning history, bug data, and descriptive information is scattered around heterogeneous sources and thus differs in form and content. Moreover, much information originating from abandoned, incomplete, or small projects is inconsistent and can thus lead to unreliable results. Finally, it is difficult to support investigations concerning general software attributes as *Size* or *Quality* since such analyses require a high level of abstraction.

In this thesis, we present OSSQuery, a data mining system that indexes, collects, cleans, updates, and analyzes publicly available project data to allow quantitative research on open source software. An efficient, extensible, and reliable design allows our system to construct a representative data set and to analyze a variety of questions on open source software on demand. In spite of primarily addressing cause – effect relations, the modular architecture can readily accommodate several types of analysis.

OSSQuery indexes over 143,000 open source projects to date, it stores both historical and recent information on more than 2,500 projects. We performed a number of investigations to test our system. The outcomes confirm our primary hypothesis, namely that research on a limited set of projects might not be valid for all open source software. While our results commonly agree with outcomes of other studies, they also show that a significant proportion of projects do not fit these findings, yet.

Contents

1 Introduction **1**
1.1 Background and Motivation . . . 1
1.2 Major Problems . . . 1
1.3 OSSQuery . . . 2

2 Related Work **4**
2.1 Measuring Software . . . 4
2.2 Data Collection . . . 5
2.3 Data Mining . . . 6
2.4 Background to Queries . . . 8

3 Method and Design **9**
3.1 Method and Design Goals . . . 9
3.2 System Architecture . . . 9
3.3 Data Collection . . . 10
3.4 Data Mining . . . 14

4 Data Collection, Realization **17**
4.1 Implementation . . . 17
4.2 Practical Problems and Solutions . . . 19
4.3 Results and Remarks . . . 21

5 Data Mining, Realization **23**
5.1 Implementation . . . 23
5.2 Practical Problems and Solutions . . . 25
5.3 Results . . . 26

6 Discussion **28**
6.1 Limitations . . . 28
6.2 Results . . . 28
6.3 Future Work . . . 34

7 Conclusion **36**

References **38**

Appendix A : Data Collector Design **41**

Appendix B : Further Statistics on Data Collection **42**

Appendix C : Concept Hierarchies **43**

Appendix D : OSSQuery Screenshots **45**

List Of Figures **46**

1 Introduction

1.1 Background and Motivation

Open Source Software (OSS) has become a phenomenon of real interest to a large range of people and organizations [18], and is still rapidly evolving. The collaborative nature of OSS endorses its fast diffusion and the development of high quality products [41]. Consequently, the availability of information regarding OSS on the web increased dramatically. Huge so-called forges, e.g. SourceForge, GitHub, and Google Code, provide hosting facilities for hundreds of thousands of projects each and act as a centralized location for developers to control, manage, and release their software. Besides storing the source code, such forges commonly provide an infrastructure for publishing information on bugs, collaborators, versioning history, releases, and descriptive data.
While literature has investigated characteristics of proprietary software since decades, the rapid growth in the availability of information attracts many researchers to study open source software. Thus, in recent years, experts extensively investigated individual projects or small groups of projects on different attributes, e.g. [1] [2] [3] [6]. To improve the quality of software and of software processes, researchers are particularly interested in studying and revealing relations among different software attributes.
Such research and studies produced useful outcomes and models concerning particular OSS projects and phenomena. Producing results that are project independent and thus of general validity however, is only possible by considering and measuring at least a significant proportion of the universe of available open source data. We present two major considerations on this behalf. First, analyzing a large proportion of OSS prevents results to be affected and distorted by the choice of projects. Second, conducting certain kinds of analyses is not feasible on a constrained data set, because important elements may be outside the boundaries of a project. For example, it is only possible to measure the isolation of a developer community while considering all other projects, to which the community under analysis might have contributed. A quantitative analysis on open source data requires an automated identification, collection, cleaning, and analysis of open source projects from the web.
The development of a tool to automate this complex research process is thus motivated by the availability of an invaluable amount of OSS data and by the lack of quantitative research on these data. Such a tool allows the discovery of results with general validity, and the description of the OSS universe through a representative data set.

1.2 Major Problems

Although online forges provide abundant information on open source projects, the automation of both retrieving and analyzing data is fraught with conceptual complexities and practical problems [5], some of these are listed in the following.

1. Project information has to be identified and located in each forge.

2. Such information differs both in form and in content from project to project and must for instance be extracted from a web interface, i.e. HTML site, or through an analysis of the source code.

3. Forges commonly contain a large number of abandoned, incomplete, and unmaintained projects which provide inconsistent information.

4. The automated research process should be able to answer to a variety of different and

possibly complex queries regarding OSS. These queries may regard abstract project attributes and relations, e.g. *Size* or *Quality*.

5. Both large-scale data retrieval and analysis are resource–consuming processes. We aim for an efficient implementation to achieve the task of automating such processes with with limited resources.

Thus, the availability of OSS project information allows collecting and analyzing a significant set of open source projects. The automation of this task is challenging since it includes a number of problems from the fields of information retrieval and data mining, though.

1.3 OSSQuery

Through this work, we present our approach to the problem of automating the identification, collection, cleaning, and analysis of open source projects from the web: OSSQuery, a data mining system for investigation on cause–effect relations in OSS. OSSQuery is a system that automates such a research process by building and updating a large and consistent set of project information and by performing different analyses to answer research queries related to OSS. More precisely, our contribution is a system that:

- Identifies and locates a large amount of projects on different project hosting facilities, i.e. forges
- Retrieves different, descriptive project information regarding the projects it located
- Downloads and analyzes the source- and byte code of an open source project to retrieve different metrics
- Analyzes the versioning system and the bug tracker to retrieve information on the development history and the defects of a project
- Builds a centralized, large-scale, and consistent data set containing information on projects with different characteristics
- Iteratively updates the data set to store both historical and current information on the projects
- Performs a wide range of analyses and data mining tasks on the collected data
- Provides a simple structured web interface that allows performing analysis with few clicks for an easy interaction

Since providing trustworthy results on such a large-scale is on the one side a complex, resource consuming and on the other side a delicate task, OSSQuery has the following qualities.

Extensibility, to embrace the continuous change in the world of open source and to permit a variety of different kinds of analyses on a diverse data set. The modular architecture of our system can readily accommodate new types of projects and analyses (Chapter 3).

Abstraction, to support and analyze heterogeneous data from different sources. OSSQuery transforms and converges data to a unified, abstract, data model that fits the data of different forges.

Performance/Efficiency, to allow the collection of a large data set in reasonable time. Through testing and measuring the processes of data collection and data mining in OSSQuery, we show that the system is (1) performing because collecting and analyzing >100,000 of projects and (2) efficient because processing a project within 2 minutes and a data mining task within 40 seconds on average (Chapter 4,5).

Reliability, to provide trustworthy results. The validation of our methods through manual testing demonstrates soundness of our approach. Only a negligible amount of projects is e.g. not being collected due to exceptional circumstances.

Precision, to provide exact and comparable results. OSSQuery delegates delicate tasks, e.g. the source code analysis to commonly used and approved tools. Moreover, the collected data set is validated through comparison with findings of other collection tools [22] [23].

Although there exist tools that collect data from online forges [4] [22], we found no other system that allows performing automated analyses on a consistent, unified data set for quantitative investigations on OSS.
This thesis is structured as follows: In Chapter 2 we investigate on related research to outline major contributions in the field of automated data collection and analysis on OSS. We present our approach, method, and thus the design of OSSQuery in Chapter 3. Chapter 4 and 5 are dedicated to the realization, and thus to implementation details, major problems, and results regarding the data collection and data mining, respectively. Chapter 6 discusses limitations, results, and directions for future work.

2 Related Work

Our aim is to create a system that automates the open source research process to facilitate quantitative analyses on public OSS data. Commonly, this research process [4] involves two major steps: the collection and the interpretation of that data. This Section starts by presenting how software and software projects can be measured to give a clear picture of what data we are processing. Then we investigate different methods and techniques concerning the automation of both steps. Finally, we explore the background of some queries our system can address.

2.1 Measuring Software

There is no precise or unique picture of what information a practitioner needs to describe and measure a project [24]. Literature proposes a variety of different methods and approaches to measure software, depending on which software attribute is examined.

- Code is measured in terms of size and structure [7]. There exist four concepts of size: length, functionality (function points), computational complexity, and reuse. Structure, instead, is measured with respect to control flow structure e.g. cyclomatic complexity [7], to information flow e.g. CK metrics [8], and to data structure e.g. database size.
- Defects are an important indicator for software quality and can be tracked on most OSS repositories. Defects might be faults, bugs, failures, or anomalies in the functionality of software.
- Communities are analyzed extensively in research. Literature [6] [25] is particularly interested in how properties of developer communities such as size, connectivity, or evolution affect the software produced. A developer community is defined as the set of all active collaborators or maintainers of a project.
- Project activity and maturity is an abstract concept studied in different ways [2] [6]. Forges commonly provide historical information of software development through versioning systems.

We will briefly introduce the CK metrics suite [8], since it is of particular relevance to understand the outcomes of our work (Section 6). The suite proposes six measures for the structural quality of object oriented software.

1. WMC: *Weighted Methods per Class*, the sum of the number of methods of a class weighted with respect to the cyclomatic complexity
2. DIT *Depth of Inheritance Tree*, the longest path from a class to the most remote ancestor
3. NOC: *Number of Children*, the count of all the direct children of a class
4. CBO: *Coupling Between Objects*, a measure of the dependencies than an object has with other objects
5. RFC: *Response For a Class*, the number of methods of a class than can be invoked in response of a call to a method of a class
6. LCOM: *Lack of Cohesion in Methods*, the absence of cohesion among the methods of a given class

2.2 Data Collection

There exists a huge amount of sources providing OSS data over the Internet. These data differ in type and form and are partially "raw", i.e. they have to be analyzed and combined to be useful [5]. In the following, we will report major techniques for gathering data from the Web.
Research proposes various solutions to how information can be located and retrieved from the Internet. Web crawling is the automated process of discovering and indexing web pages on the World Wide Web. Most of the problems in retrieving OSS project information are similar to the standard problems in web – crawling. Those are scale, content selection, social obligations, and adversaries, i.e. misleading or useless information [9]. However, a number of major differences make a general web-crawler [10] not suitable for retrieving OSS project information.

1. Limited search domain, i.e. only very specific sites are interesting
2. There is no updated and reliable public index of all project web pages in a forge.
3. Many links are self-referential or redundant, i.e. they point to a page that contains the same structured information. This may lead a general crawler to index the same information several times.

Thus, a general web-crawler indexing all reachable links is too expensive, crawling for OSS information needs a more precise and focused approach.
Previous research has also proposed various solutions for the problem of updating retrieved information. In theory, information should be updated only when it changed. In practice, a web-crawler cannot know when a referenced resource is changed. Clustering methods can address this problem by grouping resources according to their recent update frequency. The crawler can then decide to re-crawl an entire cluster by looking whether a few samples from that cluster where updated [11]. This technique however, requires deeper investigation and is out of the scope of this work.
OSS project forges are inevitably full of missing, noisy, and inconsistent data due to their huge size and their multiple heterogeneous sources of data. Since low data quality data leads to low quality results, data mining systems preprocess the data before analyzing it [12].
First, data cleaning is applied to remove noise and inconsistency. Literature proposes various methods to handle inconsistent data e.g. ignore tuple, use the attribute mean (by category), or use a Bayesian formalism to calculate the most probable value [13]. Applying such a method to OSS information would make not be appropriate since we are talking about source code, byte code, and project details which cannot be omitted or inferred. The problem of noisy data is task-specific and therefore addressed in the following Section.
Second, data from different, heterogeneous sources is integrated and combined to form a homogenous data store, as in data warehousing. This task requires a general, unified data model fitting the data models of the sources. Moreover, it can be tricky to correctly match the same information from different sources. This problem is called the entity identification problem; literature [14] suggests general solutions which are less suitable for domain specific data.
Recently, Howison, Conklin, and Crowston (2006) introduced "FLOSSMole, a collaborative project designed to gather, share and store comparable data and analyses of free and open source software development for academic research" [4]. Their idea is to gather huge amounts of data from several forges through web crawling or database dumps. FLOSSMole made significant contributions in the field of collecting OSS data; therefore, some tasks are similar

to the tasks of OSSQuery. The purpose however, is different: FLOSSMole aims to provide comparable and clean data to researchers, whereas OSSQuery aims to provide a data mining system which, based on a comparable and clean data set, answers research questions to cause – effect relations between OSS attributes. In spite of collecting a huge amount of data, FLOSSMole is not suitable as data source for OSSQuery.

- There is no easy access to the data; one must either download the whole database manually or request an account to have limited access to their database.
- FLOSSMole provides no efficient update facility. Thus, an update requires re-downloading the whole database.
- FLOSSMole does not provide any source code and only few software metrics, it is therefore not adequate for our purpose.
- Due to the high number of forges and information types FLOSSMole addresses, it uses a different data model for each forge. This fact leads to difficulties in data interpretation and to inconsistency.

Another, research project [22], collects data from SourceForge only. Similar to FLOSSMole, this project is limited to the bulk-download of general project information. In conclusion, research has shown that data collection includes the steps of data identification, cleaning, retrieval, transformation, and storage. Thus, literature proposes a variety of solutions that we analyzed in this Section.

2.3 Data Mining

Han and Kamber state that "data mining refers to extracting or mining knowledge from large amounts of data" [12]. Commonly, data mining includes the automatic discovery of relations among data. Our goal, however, is not the discovery but proving the existence or non-existence of assumed relations. To achieve this goal, we apply the steps of the process of knowledge discovery (Figure 1) and perform guided data mining [12]. This Section touches data mining techniques most relevant to the purpose of OSSQuery.

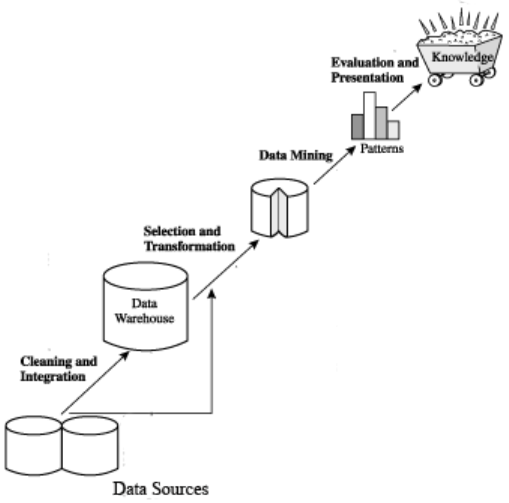


Figure 1: The Process of Knowledge Discovery

A data set must be clean and homogeneous before it can be mined. For this purpose, we first need to remove inconsistent and erroneous data. Literature presents different approaches for data cleansing [12], among others techniques to identify or remove outliers. Since we are interested in cause-effect relations among OSS data though, outlier projects are a potential source of knowledge (Section 6). Thus, we limit ourselves to setting a number of empirical thresholds [21] to filter out small, abandoned, and inconsistent projects. In the following, we present a number of data mining techniques that are relevant to our purposes.

Correlation Analysis
Correlation measures the dependence and thus describes a relationship between two variables X and Y. Although dependence is neither directional nor sufficient to infer causality, it is a valid instrument to reveal potential cause-effect relations. A hypothesis test can then indicate whether the calculated correlation corresponds to the true correlation with a significant confidence. The most common correlation measure for large, normally distributed samples is Pearson's correlation coefficient (1).

$$r_{XY} = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2 \sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{1}$$

Other correlation measures are Spearman's rho and Kendall's tau. Given our data, Pearson's rho is sufficient for our purposes.

Regression Analysis
While correlation measures the dependence and thus the strength of a relationship between two variables X and Y, the regression models this dependency through a function. Regression can thus define the nature of a cause – effect relation by depicting how the typical value of the dependent (response) variable changes when any of the independent (predictor) variables vary. Moreover, regression allows the prediction of a value for the dependent attribute, given one or more independent attributes. Linear regression is the simplest form of regression (2).

$$y = \alpha x + \beta \tag{2}$$

Where y is the response attribute, x is the predictor attribute, and α and β are regression coefficients which depend on the data set. In the case of linear regression, the method of least squares can solve these coefficients

Clustering Analysis
Clustering groups a collection of data items into clusters of similar items. The objects of one cluster are then dissimilar to the objects of another cluster. Clustering does not only classify items, but it gives a measure for similarity between objects. Such analyses are especially interesting for OSSQuery, since they can deal with different kinds of attributes and with noisy data. Most applications use heuristic partitioning methods, where clusters are constructed around the cluster mean or median. The k-Means algorithm, for example, randomly selects k item values as means for k clusters, assigns the remaining items to the nearest k value, and calculates the new cluster mean. This process is repeated until the distance of the item values of a cluster to their mean is minimal.
Other data mining techniques include the classification of items, and association mining [12].

While the former might be relevant for further analysis on OSS projects and thus for future work, the latter is somehow related to our main purpose. Association rule mining is the automated discovery of recurrent patterns and thus relations among data, whereas we aim to facilitate the investigation on these relations. A very specific and much-studied domain, i.e. open source software, allows us to perform directed analyses instead of identifying all possible relations.
To be effective, a data mining tool must allow the user to define a data mining task precisely through a query. Han et. al. [21] propose five general considerations to this problem.

- The set of data relevant to a data mining task should be specified in a data mining request
- The kinds of knowledge to be discovered should be specified in a data mining request
- Background knowledge could be generally available for a data mining process
- Data mining results should be able to be expressed in terms of generalized or multilevel concepts.
- Various kinds of thresholds should be able to be specified flexibly to filter out less interesting knowledge.

2.4 Background to Queries

The idea behind OSSQuery is to address a variety of different research questions on relations between attributes of OSS. As a proof-of-concept, we will investigate four commonly studied research questions; this Section presents the background of these questions.

1. Do non-isolated communities produce quality code? Isolated communities are developer communities that are not connected to other communities, i.e. where no member contributes to other projects. Research has investigated both on developer communities [6] and on code quality [2]. Although the relation of the two attributes is very interesting for researchers and managers, not much related research has been done up to now [6]. This question seeks for a prediction model that describes the relationship between isolation and quality.

2. To what extent does isolation within the developer community affect code activity? Previous literature investigates both on community and on code evolution patterns [6]. Instead of giving models, we aim to give a precise measure based on a quantitative analysis.

3. Does size affect code quality? According to Lehman's Laws of Software Evolution 1,6, and 7 [19] software is continuously growing in size and complexity, and quality is decreasing if not rigorously maintained.

4. Does activity indicate project maturity? OSS software arguably suffers from the "permanently beta" syndrome and low maturity. We aim to discover whether OSS project maturity depends on activity.

To conclude, we have seen that there exist many techniques for selecting data, cleaning data, and for mining knowledge from that data. Moreover, we have presented a number of considerations regarding the specification of data mining queries and we have examined different examples of such queries.

3 Method and Design

Retrieving and mining data from several large online forges is a complex task and full of practical problems and conceptual dilemmas [5]. It is important to notice that every architectural and methodological solution comprises a number of decisions that influence the outcomes of the analyses. This Section presents the method and decisions in our approach.

3.1 Method and Design Goals

OSSQuery's primary task is to identify, filter, transform, store, and evaluate a large set of comparable data. On the one hand, different forges provide data that is radically different both in form and in content. On the other hand, only a diverse and large set of forges can provide a large and statistically relevant dataset. Moreover, researchers and practitioners could use the data and analyses OSSQuery produces to draw conclusions on OSS. The combination of these premises leads to five major design goals.

1. Extensibility The system must be extensible to support the collection of projects of several programming languages from different forges, versioning systems, and bug trackers. Similarly, it must be extensible to facilitate and support different types of investigations on the data.

2. Abstraction: Through a high level of abstraction, the system must integrate and convert heterogeneous and raw data into an abstract and unified data model.

3. Reliability: The system must be functional and reliable to provide trustworthy outcomes. High scalability and large data sets increase OSSQuery's overall complexity. Such a complex system must be designed in a modular way to isolate points of failure and to not bias project data during the collection or analysis.

4. Efficiency and Performance: A data mining system must be performing to collect and analyze a large and representative set of OSS data. Moreover, it must be efficient to accomplish this task in reasonable time.

5. Usability: Since the output of OSSQuery is primarily used by researchers, a clear, simple, and functional representation is necessary to provide unambiguous results.

These goals call for a highly modular, extensible, and multi-threaded architecture and for a simple and functional front-end.

3.2 System Architecture

OSSQuery has a two-component architecture: The Data Collector (Section 4) identifies, retrieves, analyzes, cleans, and stores data from several forges. This component iteratively takes snapshots of the forges and thus runs as a service, i.e. it is a daemon. The Data Miner (Section 5) interprets a user query, retrieves related data from the database, and analyzes the data according to the query. This component is request – driven, i.e. the user starts a mining task by requesting a certain analysis. The database is iteratively updated and expanded, thus it contains both historical and current data, and it grows indefinitely.

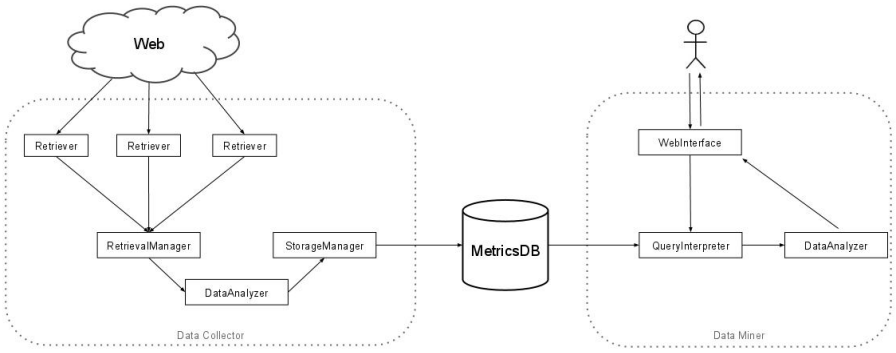


Figure 2: Logical Representation of Information Flow among OSSQuery Modules

3.3 Data Collection

Designing a reliable and modular piece of software that builds a large data set of clean project information is a complex task. What is, at first glance, a simple sequential process is not feasible by indexing, downloading and analyzing data impetuously for a number of reasons.

1. The webpage of projects differs both in content and in form from forge to forge, thus we need to implement support for each type of webpage.

2. The versioning system may differ in structure (centralized, distributed) or in granularity (project versioning, file versioning). The most used systems are Subversion, CVS, Git, and Mercurial.

3. The programming language differs from project to project. To analyze different languages and extract metrics, we need to interact with different tools.

4. The bug tracker varies from forge to forge, to perform exhaustive analysis on software defects we thus require support for each tracker.

Hence, to be scalable and to analyze data from a diverse and large set of projects, we would have an exponential complexity of data collection modules in our system. More precisely, we needed to support and thus to create a new module for any possible combination of characteristics listed above (3).

$$S_m = F_0 \times F_1 \times \ldots \times F_n \tag{3}$$

Where S_m is the set of data collection modules, and F_0 is the set of modules needed for one specific feature (analogously for $F_1 \ldots F_n$). Supporting a new characteristic, e.g. programming language, leads to a linear growth, while adding a new supported feature, e.g. mailing list, leads to an exponential growth in the number of modules. Such a design is neither modular nor maintainable and thus unreliable, error prone, and not extensible.
Fortunately, there exists a fitting solution for this problem [32]. The Abstract Factory is an architectural design pattern that allows "creating families of related objects without specifying their concrete classes" (Gamma et. al, 2002). The idea is that an Abstract Factory can chose and combine a set of concrete classes to create a general object/a family of objects, analysis modules, in our case, that are defined through abstract interfaces.
In our case, an Abstract Factory can compose complete analysis modules according to the general project information and its link. Through this pattern, we can easily assemble an

ad–hoc module meeting the characteristics of a specific project by using its descriptive information. The Abstract Factory pattern reduces our complexity significantly (4).

$$S_m = F_0 \cup F \cup \ldots \cup F_n \tag{4}$$

Supporting a new feature, e.g. programming language, leads to a constant linear growth of one module, and supporting a new artifact, e.g. mailing lists, leads to a constant linear growth of |Fi| modules. Moreover, the points of failure are well isolated and identifiable, since there is only one implementation, e.g. for analyzing Java code. Reducing the architectural complexity increases maintainability, modularity, reliability, and extensibility and thus satisfies design goals 1,2, and 3.

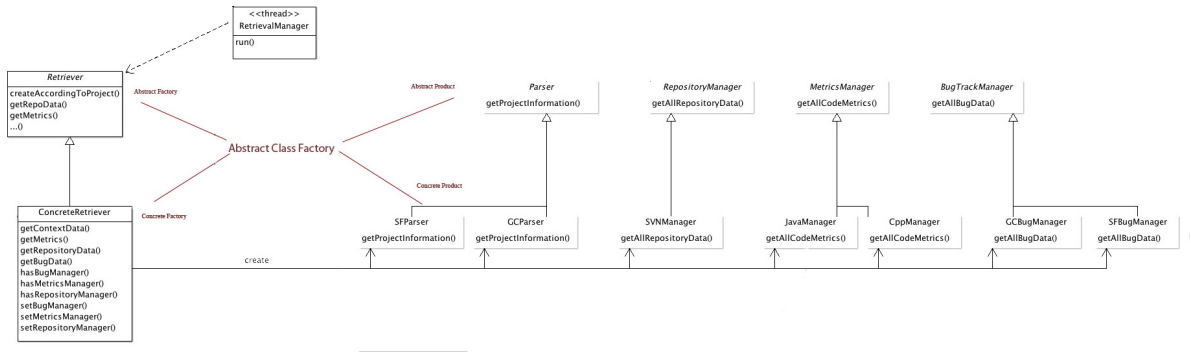


Figure 3: Abstract Factory Pattern in the Data Collector

The design of the Data Collector in OSSQuery is based on these considerations and makes extensive use of the Abstract Factory. Figure 3 depicts the implementation of this pattern in the Data Collection component, a complete class diagram is presented in Appendix A. Up to this point, we have seen how an elegant design can support a variety of different features. In the following, we will examine how to actually identify, retrieve, extract, clean, and store project data.

Link Retrieval

To collect data on OSS projects it is necessary to locate, identify, and index project information in the first place. Web spiders are systems which browse the World Wide Web and index web pages in an automated manner. They are predominantly used to build a corpus of web pages for search engines and commonly face problems of scale, efficiency, and of making too many HTTP requests [9]. In the case of OSSQuery, we have a limited set of forges and thus a limited set of different site-types to inspect. We can use this knowledge to restrict the set of links that have to be inspected significantly. In information retrieval, we refer to the set of links that have been found, but still need to be inspected as spider frontier. OSSQuery addresses the problems of scale, efficiency, and numerous requests by adapting a basic web crawling algorithm [9].

Algorithm 1 CPL (Crawling for Project Links)

```
Input: interestingURLRegex, projectURLRegex, seedURL
1. Put seedURL into frontier
2. While(frontier not empty)
        2.1 take URL from frontier
        2.2 make http request and fetch html code
        2.3 for(each link in html code)
                2.3.1 if(link matches interestingURLRegex)
                                add link to frontier
                2.3.2 if(link matches projectURLRegex)
                                add link to projectLinks
3. return projectLinks
```

CPL (Algorithm 1) takes two regular expressions, which (1) match links that potentially lead to project links and (2) match project links, as input. Since all major forges use a standard URL naming convention and there is a very limited set of large forges, it is easy to manually define the regular expressions. CPL uses two data structures, the frontier queue, which contains interesting links to be inspected and the projectLinks list, which contains the project URLs and is the return value. Both queues do not allow duplicates to avoid redundant spidering. In OSSQuery, the frontier uses a FIFO prioritizations which, with respect to a LIFO or more advanced URL ordering algorithms [17], guarantees a simple implementation and a reliable coverage of all indexed pages [26].

Project Data Retrieval

Once the URLs of the OSS projects have been collected, OSSQuery identifies, filters, and downloads project information to build the data set. To answer a broad range of research questions, we need the following information:

- Descriptive Information such as project name, description, labels, programming language, and activity.
- Collaborator Information such as id, alternative id, email address, or name to identify the collaborator on other projects/forges.
- The link to the Versioning System for the analysis of the source code and the commit history.
- The link to the current release for the analysis on the byte code.
- The link to a working bug tracker for the analysis of bugs and bug reporting.

There are different ways to retrieve this information from a forge. OSSQuery, depending on their availability, uses one or a combination of the following methods.

- Web services: major forges usually provide an XML/RSS interface to access project information. Web services, with respect to other methods, are commonly reliable and stable, i.e. robust towards change. However, they are not always present and complete (e.g. they might not provide bug information).
- HTML parsing: accessing project information by parsing the presentational web interface of a project commonly provides all available information on a project, and is always

available. This interface is may change over time, though. Such a variation may lead a parser to misbehave or even fail.

- Java-API: is a usually a reliable, simple, and stable interface to the project database of forges. Unfortunately, such an API is rarely available and provided only for some specific information (e.g. for bug information only).

An alternative to collecting OSS information via spidering is asking a forge for its database dumps, thus having direct access to data. Although this method is easier in practice, it involves a number of methodological problems. First, dumps containing OSS project information are rarely available since they are the central value of such forges and their distribution involves additional costs. Second, they are static snapshots of the database and do not allow efficient updating. Repositories face a growing number of requests for access to their databases, as a consequence they reject deny or provide access to accredited academics only. This leads to non-comparable and non-transparent research.
Once the necessary information has been retrieved, a first cleaning phase is necessary. Metrics obtainable by source and byte code, collaborator information, and versioning information are crucial for answering interesting research questions. Projects which fail to provide any of this information, must be filtered to maintain a consistent dataset.

Project Data Analysis
OSSQuery is not limited to the retrieval of general project information and code, but actively analyzes the versioning system, the source code, the byte code, and the bug tracker. While versioning systems and bug trackers provide an apposite API in many cases [27] [28], the analysis of source code or byte code requires the integration of external tools to get comparable results. Since it is known that different tools provide different results [29], it is important to select accredited and popular tools to produce reliable and comparable results. Section 4.1 presents further considerations on the choice of APIs and tools for the analysis of versioning systems, bug trackers, and code

Data Storage and Updates
All OSS project information and metrics within OSSQuery are stored in the Unified Project Data Model (Figure 4). This data model must accommodate general project information, collaborator information, code and versioning analysis outcomes, and bug information. Moreover, it must be abstract, i.e. project and forge independent, fit the data models proposed by the forges, and include historical and current project information. We want to accumulate data over time to be able to not only analyze the current state of projects, but also their history and development. For that purpose, the Data Collector fetches project information periodically. To minimize HTTP requests and resource consumption, previous literature [11] proposes to use clustering algorithms to guess the change frequency of a certain web page. Such an approach might be adapted for predict changes in project, but it would require a significant training set and is beyond the scope of this work. The web spiders thus follow a simple strategy: a project is updated if it was not updated for a certain time span (e.g. 2 weeks). The update consists of retrieving the general project information and re-analyzing artifacts only if they haven't been analyzed before.

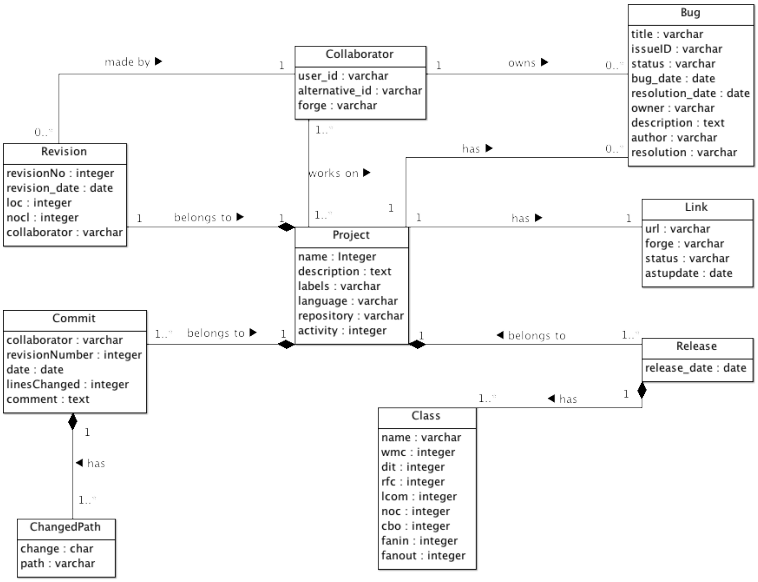


Figure 4: Unified Project Data Model

3.4 Data Mining

The idea behind OSSQuery is to allow researchers to perform a wide range of mining tasks based on a large and representative sample of OSS attributes automatically. The design of the Data Mining component provides the same level of extensibility as the Data Collector by using the concept of Abstract Factory to allow different types of analysis on the collected data. Moreover, it is necessary to fit this architecture to a standard, abstract process of analysis.

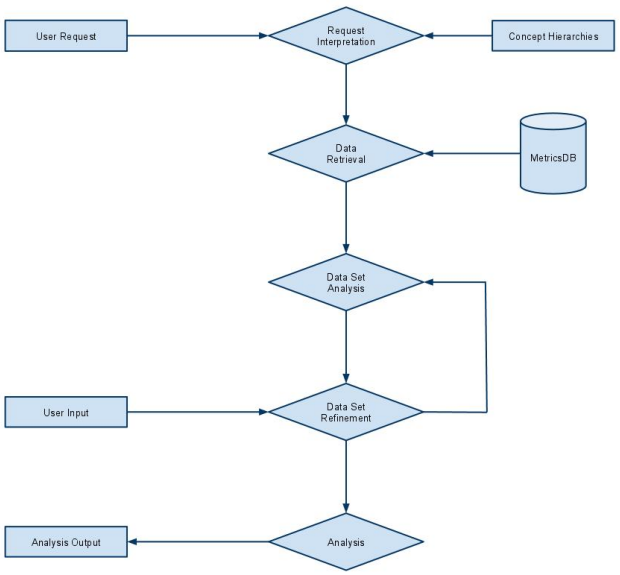


Figure 5: Data Mining Process

Figure 5 depicts an abstract process that follows the five principles to design a data mining query language (Section 2.2) [21] in five steps:

1. The user defines and launches the Data Mining task. Such tasks commonly contain general software attributes, as quality or size, thus it is necessary to define these attributes we want to analyze. In OSSQuery, attributes are organized on multilevel concept hierarchies [21]. This way, we can map high-level software attributes to low-level project information and thus handle the requests with a certain precision. These taxonomies have a strong impact on the outcomes of the analysis, and should therefore be transparent to the user.

2. Upon a request, the system automatically generates and launches queries to retrieve all task relevant information from the database.

3. The system analyzes the data set on the statistical distribution of each attribute. The corresponding results are presented to the user.

4. The user can refine the data set to focus on specific projects as well as consider different data sets for a certain investigation. Subsequently, he may either proceed to the results of the mining task, or re-check the new distribution of the data set. The user can carry out this step of refinement for an indefinite number of times.

5. Finally, the system performs the data mining/analysis through different statistical tools presented in Chapter 2 on the refined data set. The results are presented to the user.

The diverse data set and the extensible architecture of the Data miner allow a variety of analyses. Since the realization of different types of mining tasks is beyond the scope of this work though, we focus on one type of analysis as a proof of concept and provide an extensible architecture for future work (Section 6). Research in OSS is particularly interested in dependencies among software attributes; consequently, we decided to focus on cause-effect relations. Such relations describe a connection between two facts, where one fact is the consequence of the other fact.
Notably the software measure are approximately normally distributed around an ideal value, thus we decided to use Pearson's correlation to measure dependence. As a measure of support for the correlation, we give the p-value for the null-hypothesis H_0 "The two variables are zero correlated", thus a low p-value signals a high reliability on the fact that a correlation truly exists. Pearson's correlation is a measure for the dependence and thus the relation between attributes, but gives no information on how the attributes are related. For this purpose, we perform a regression analysis. A proper mathematical relation defines a cause–effect relation.
Software metrics exists on various levels of granularity, e.g. there exist project level, class level, and method level metrics. Blindly comparing metrics of different granularity leads to inconsistent results. It is possible to compare the metrics values on the same level by calculating the mean value. This is prone to biased for two reasons, though. First, the mean value is often not representative since it reduces the variance a distribution has. The deviation from the mean and subtle trends that could be significant are neglected. Second, the operation of calculating the mean is not additive in general, thus giving misleading results when used for further computations (such as correlation or regression).
In case two or more metrics have different granularity, OSSQuery implements the following mehod. First, we calculate minimum, maximum, and average value of the class-level data by project. Second, we cluster both attributes into three major clusters (small, medium, and large numbers). Finally, we give a percentage for the correspondence between the clusters at

project level and at class level.

loc\cbo	small	medium	large
small	85%	13%	2%
medium	90%	9%	1%
large	97%	3%	0%

Table 1: Cluster Analysis on LOC and CBO metrics

Table 1, for example, gives rough information on how the *LOC* (*Lines Of Code*) metric relates to the *CBO* (*Coupling Between Objects*). Notably, larger projects tend to have a lower *CBO* value and therefore a modular, less complex, and less faulty structure. Considering all the values on the table, a researcher in OSS can easily get a rough image of how the *CBO* metric behaves for projects with of different sizes.
In this Chapter, we have presented how structural patterns can facilitate an extensible design. We also propose a number of methods to address the challenges of both collecting and analyzing a large set of project information. In the following Chapters we will validate our approach through different tests. Moreover, to assess our overall method and thus the reliability of our results, we have chosen four OSS research questions on cause-effect relations from literature (Section 2.3). Comparison of our results to previous research on the same questions can give us clues on whether our methodology is sound.

4 Data Collection, Realization

While Chapter 3 introduces the design and method of our approach, this Chapter presents the realization of the Data Collection component. In particular, we will focus on the implementation, major practical problems, solutions, and specific results of the data collection in OSSQuery.

4.1 Implementation

Open Source Software is scattered around a large number of forges, in different programming languages, using different versioning systems and different bug trackers. We aim to collect a large and representative set focusing on projects with certain characteristics. In this Section, we first state which types of projects OSSQuery currently supports, and subsequently give implementation details.
Fortunately, most of the projects share a small number of common properties. Therefore, it is possible to analyze a large set of projects supporting a limited set of characteristics. We have focused on the projects with the following properties.

Forges
Most of the OSS projects reside on the most popular project hosting sites [4], such as SourceForge, GitHub, GoogleCode, Launchpad, CodePlex, or GNU Savannah. Unfortunately it is also true that some well-known and much-studied Apache and Linux projects prefer to use their own repositories. We decided to retrieve our projects from SourceForge (300,000 projects [1]) and GoogleCode (>250,000 projects [2]), because they offer a wide range of information on a huge number of projects. GitHub is not relevant for our purpose since it provides no byte code releases and it exclusively features Git as versioning system.

Programming Languages
Different pragmatic approaches for measuring the popularity of programming languages [38] [39] agree on the fact that Java, C, and C++ are prevalently used in OSS (disregarding languages such as RegEx, HTML, and XML). For the selection of the supported programming languages, we also considered the availability of reliable metrics. Chidamber and Kemerer (1994) propose a metrics suite [8] that gives an extensive and precise analysis of object-oriented code [37]. For the popularity and the availability of adequate metrics, we focus on Java (approx. 18% of OSS projects) and C++ (approx. 10% of OSS projects) .

Versioning Systems
Large-scale investigation [39] shows that 58 % of OSS projects use Subversion as Versioning Control System. The remaining part predominantly uses GitRepository (25%) or CVS (13%). Considering these numbers and the availability of APIs to the repositories, we have chosen to focus on projects using Subversion.

Bug Trackers
We have decided to use the bug trackers provided by the forges, i.e. Sourceforge and Google Code. Although there exist a number of large, independent bug trackers, we noticed that people tend to report bugs where they downloaded the software.

[1] www.sourceforge.net
[2] http://googleblog.blogspot.com/2009/12/meaning-of-open.html

Further Limitatons
Since we want to gather a consistent and reliable set of project information, we decided to exclude small, incomplete, or not measurable projects. Therefore, we apply the following thresholds.

1. The project must provide a release with at least 10 compiled modules.
2. The project must provide source code with at least 10 modules.
3. The project must have at least 10 commits in the last year.
4. We collect the bugs of a project only if there exist more than 10 reported bugs.
5. The project must have at least one active collaborator.

These criteria were theoretically chosen to match projects with consistent information, while still allowing a quantitative analysis on a large and diverse data set. Moreover, the modular and extensible design of the Data Collector permits an easy extension of the supported features (Section 6.3). A modular, extensible, and reliable architecture must be implemented with an adequate programming language. The Data Collector of OSSQuery is entirely written in the Java programming language (v. 1.6), whose strengths are modularity for extensible and reusable code, a large standard library, and the availability of many third-party libraries. Collecting, storing and analyzing OSS requires interaction with the WWW, the local file system, a database, and with external software. In the following, we present external tools and libraries we integrated to gather and store project information.

Tools for Code Analysis
To collect a set of correct and comparable code metrics, it is necessary to use reliable and popular tools [34]. While there exist many tools that collect exclusively simple metrics, such as *LOC* (*Lines Of Code*) or *NOM* (*Number Of Modules*), software for a more advanced analysis of code is scarce [33]. We pointed out free academic tools that are commonly used to calculate the CK metrics [8]. The first, CKJM [30], analyzes Java byte-code and is integrated as a library. The second, CCCC [31], analyzes C++ source code, but is written in C and offers no interface to Java. Thus, OSSQuery executes CCCC as an external process, parsing the results in XML format. To analyze simple code metrics such as *LOC* or *NOM*, we adapt JLOC [42], an implementable and reliable tool that analyzes both C++ and Java source code.

Database
OSSQuery gathers a large data set of projects and stores a considerable set of information for each project (Chapter 6). Moreover, to perform exhaustive analysis, we must query the data continuously. Thus, a reliable and fast database management system (DBMS) is crucial to our task. We chose PostgreSQL because of its speed (especially in recent versions), its robustness, and its ACID compliance. To interact with the DBMS, we use JDBC, a popular API that provides methods to launch queries directly from Java.

Additional Libraries
Java provides a Standard Library of remarkable size, which supports many features of OSSQuery. In particular, all HTTP requests and file system operations are implemented using standard libraries (Java Net and Java IO). For other, more specific needs, it was necessary to use external libraries. Since we need both the changelog and the source code of a project, it is necessary to have an API for the versioning system, i.e. Subversion. SVNKit [27] provides an exhaustive and reliable interface to Subversion. To analyze code it is necessary to extract

code from a number of archives, we use the *TAR* and *JAR* Java libraries to support zip, g- and bzip(1 and 2), tar, and jar archives. Moreover, CKJM [30] requires different modules of the BCEL library. Finally, to download project information from online forges (Section 3.3), we use the Google Data API 1 for bug information and the XML–DOAP [44] API for project information from SourceForge.
We developed and tested the Data Collector component under Mac OS X and Ubuntu 10.04 LTS.

4.2 Practical Problems and Solutions

Developing a large-scale, extensible tool for data collection is fraught with a number of spikes and pitfalls [5]. This Section reports major practical problems encountered during the development of the Data Collector. Moreover, we present possible solutions by showing how we address these problems in OSSQuery.

Efficiency
Web Spidering as such is a resource and time consuming task [9], mining for project information is even more expensive, though. In spite of the cost of parsing web pages, the Data Collector must download byte- and source-code, and inspect bug tracker and versioning system without making too many HTTP requests. To address this problem, the data Collector uses multiple threads which concurrently process several projects. Differing project size makes the threads asynchronous, thus facilitating an uninterrupted employment of critical resources. In our case, the major bottleneck is limited bandwidth for downloading project data. A large amount of threads (e.g. 20) make this approach even more effective. To decrease the number of subsequent HTTP requests and to exploit the full download bandwidth, the threads alternate projects from different forges. Moreover, each page is requested once only, i.e. all relevant data is immediately extracted and stored. Outcomes and results related to efficiency in OSSQuery are presented in Section 4.3.

Identification of Project Release
Many forges do not exclusively provide the current release of a project on the web site. Beside the release, they provide documentation, screenshots, executable files, or older releases which we are not interested in at this point. There commonly exists no tag or flag indicating the current project release. The web site provides the name, description, keywords, date, and size of the file, though. While such identification is generally easy for a human, we found no general, implementable pattern to detect the correct file. Thus, we define a pattern, combining different characteristics commonly observed in the release files we are looking for. More precisely, we associate a weight to each characteristic and thus rank the files. The file with the highest ranking is expected to be the archive containing the current release of the project under analysis. If that should not be the case, the file ranked second will be chosen. We compute the rank of the link as follows:

- Increase of 4 points if the keywords of the link contain “featured”, decrease of 4 points if the keywords of the link contain “deprecated”
- Increase of 3 points if the filename contains the project name
- Increase of 2 points if the filename or the description contains one of the following keywords: “bin”, “release”, “dist”, “built”, decrease of 2 points if the filename or the description contains one of the following keywords: “doc”,”pdf”,”html”,”screen”
- Increase of 1 point if the filename or the description contains one of the following keywords: “stable”, “unix”, “linux”

- Assign up to 6 points for the relative file size and 6 points for the relative number of downloads (relative to other files in the list); large and more frequently downloaded files usually contain the byte code.
- Assign up to 4 points for the relative age of the file, older files are usually deprecated

To obtain this ranking policy, we iteratively set and tested different weights for a certain characteristic, while keeping other weights constant. This process was repeated until a satisfactory percentage of correct matches on a random test was met. Although the combination of optimal weights is not necessarily the overall best weighting policy, our approach provided positive results. The algorithm was tested on a sample size of 50 randomly selected projects providing a release. A manual inspection of the files/archives chosen by our algorithm revealed that 98% (i.e. 49) of the files actually contained the current release of the byte code, no file contained a former release of the project, and 1 file didn't contain a release at all. In this case, we ignore the project.

Matching and Identifying Users

Many forges identify the collaborators of a project in different ways throughout their facilities. Both in SourceForge and in Google Code, the ID given to a certain collaborator in the bug tracker does usually not coincide with the ID of the same collaborator in the versioning system or on the project website, e.g. Table 2.

Forge	ID on Web-Site	ID on Bug-Tracker	ID on VCS
Google Code	@UBBUSldZBBVFXgR7FQ%3D%3D	ma. . . @gmail.com	manuel.piubelli@gmail.com
SourceForge	mpiubelli89	Manuel Piubelli	manuel.piubelli@gmail.com

Table 2: User Identification on SourceForge and Google Code

Fortunately, each registered user has a dedicated webpage which matches user_id and bugowner ID with an abbreviated version of the email address. To match the abbreviated version to a complete version of an email address, we first match the domain of the address. Then we check whether the local part of the abbreviated email address (e.g. "*ma*") is a substring of the local part of complete email address (e.g. *"manuel.piubelli*"). Finally, if more than one abbreviated address remains, we assign a rank to the abbreviated email address according to the length of the local part (such that "*ma*" is preferred to "*m*"). This algorithm has a computational complexity of $O(n) = \frac{n}{2}^2$ and was tested on 90 collaborators of three different projects. Manual verification resulted that the algorithm worked in 100% of the test cases (i.e. on all 90 collaborators).

Parsing HTML Code

To gather consistent and reliable data, and to remain operational also in the future, we prefer to use APIs instead of parsing web-interfaces. Some information, however, is provided on HTML sites only, making scripted parsing the only solution. Large forges commonly generate their websites from templates which give them a rather consistent structure suitable for parsing [5]. A limited domain of forges makes precise knowledge of these templates possible and thus allows a purpose-specific parser to be more suitable than general solutions [35]. To identify important information, our parser relies on regular expressions matching basic, structural tags of the templates. Since the templates assure a clear-cut division between content and presentation, this strategy is robust against likely modifications of the website. Thus, a major restructuring of the SourceForge interface in April 2011 did not affect our parsing activity in any way.

Besides these major problems, we have encountered a number of minor spikes which we will not treat in detail. Some of these are file management, support of different (nested) archives, efficient memory handling, and efficient query launching.

4.3 Results and Remarks

In this Chapter, we will present Data Collection – specific results, the general outcomes of the OSSQuery project will be presented in Chapter 6. These results predominantly describe the population of projects and thus the forges that we inspect. Moreover, we will present a few general remarks on collecting data from forges that might be of interest for further research, e.g. [36]. We developed an extensible, reliable, and efficient Data Collector following the method presented in Section 3 with the limitations of Section 4.1. The Data Collector collected a large dataset indexing a significant proportion of publicly available open source software. Hence, we registered the following distribution of projects.

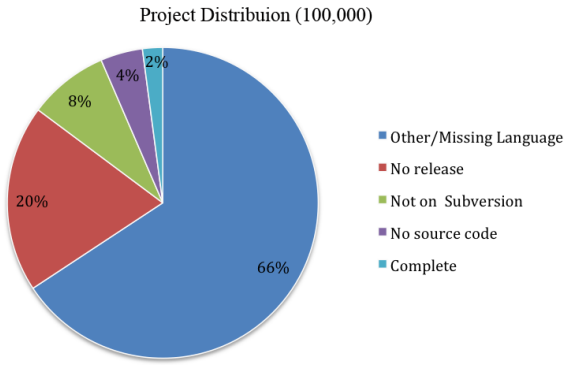


Figure 6: OSS Distribution over 100,000 Projects

This statistic considers 100,000 projects that were analyzed and filtered by OSSQuery in June 2011. Such an amount of projects is remarkable, since similar research [22] has indexed 60,000 projects, while FLOSSMole [4], a EU funded project running since 2006 indexes 500,000 projects. Notably, only 2% of the collected projects satisfy our threshold. This value is relatively high with respect to findings of other measurements, though:

- Other census [38] [39] claim that the percentage of Java and C++ projects amounts to approximately 28%. We notice, that in our statistic 34% of projects are either written in Java or in C++
- We agree with other research [5] [36] on the fact that the majority of projects are incomplete, small, incomplete, or abandoned projects. Thus, we consider only 9% of projects that use Subversion and are written in Java or C++, which amounts to 2% of all indexed projects.
- According to other census [39] 58% of all projects are versioned on Subversion, our statistics claims 92%. This discrepancy heavily depends on the fact that we do not consider GitHub, a forge which provides exclusively GitRepository. Thus, our statistic refers to project from SourceForge and Google Code only. Moreover, versioning systems are not mutually exclusive, i.e. one project may be versioned on more than one versioning system.

Thus, the above shows that only few projects are complete in terms of code, versioning and bugs, while most of them are too abandoned, small, or provide inconsistent data.
In addition to population statistics, we logged the time the Data Collector would take to analyze a project both to assess the efficiency of our tool and to get a rough picture of the project size. The average time of a complete analysis with a download bandwidth of approximately 2Mbps is 124s. The Data Collector takes, on average, 0.4s to identify a project on a forge, 109s to download all necessary data, 15s to analyze and extract information from this data, and 1.4s to store the data (approx. 1,000 tuples per project) on the database. If we include all currently indexed projects, the average time to process a project amounts to 4.2s per project. More information to this statistic is given in Appendix B.
Finally, we share some practical remarks regarding the automation of collecting data from online forges.

1. Java APIs suitable for data collection are rarely available and commonly unmaintained. Both XML and Java APIs must be integrated with HTML parsing since they do not provide all the data which is provided by the presentational interface.

2. A FIFO queue in the spidering deamon leads to a breadth first traversal of a forge (Section 3.3) and thus, if starting from the homepage of the forge, most relevant and featured projects are detected first. Appendix B shows the distribution of complete and analyzed projects with respect to the number of projects analyzed.

3. It is broadly believed that both SourceForge and Google Code implement defenses to prevent spidering [5]. Alternating requests according to forges, specifying a User-Agent header in the requests and implementing a delay in the crawling algorithm may avoid banning. We were never banned from any forge to date.

4. We have noticed that bugs are commonly tracked all in one place, i.e. bug reports of one project are commonly not scattered over different trackers. Approximately 53% of the complete, and stored project provide bug tracking on the forge where they are hosted.

This Section presented the implementation, problems, and results of collecting data from online forges. We have shown that this task involves a number of practical problems and decisions. Moreover, while analyzing the sample of indexed projects we noticed that only a small proportion of OSS projects provide consistent and complete data. General outcomes and results of the OSSQuery project are presented in Chapter 6.

5 Data Mining, Realization

This Chapter presents the realization of the Data Mining component according to the method and design in Chapter 3. In particular, we will focus on the implementation, major practical problems, solutions, results, and give some remarks regarding the realization of the Data Miner in OSSQuery. General project results and outcomes are presented in Chapter 6.

5.1 Implementation

This Section provides a technical insight into the Data Miner of OSSQuery by presenting the implementation details, tools and techniques we use to complete the five-step process of Data Mining presented in Chapter 3 (Figure 5). The extensive data set and the extensible architecture of the Data Miner permit the implementation of a number of different analysis and mining tasks. As a proof-of-concept, we focus on the investigation of cause–effect relations in OSS, thus we implemented correlation, regression, and clustering analyses on different types of software attributes.
To define a number of abstract and general software attributes which can be queried, we use concept hierarchies [20]. Concept hierarchies are multilevel taxonomies that map abstract concepts to concrete information, in our case software attributes to the collected OSS data. We can thus encode domain specific knowledge (Section 2.3) such as the composition of abstract attributes into a concise formalism. To remain extensible and modular, we define concept hierarchies in an xml file. Hence, we can easily add and therefore support new attributes by modifying the corresponding file.

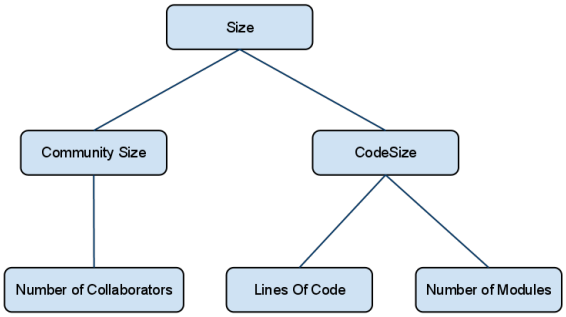


Figure 7: Concept Hierarchy for Size

Figure 7 depicts a simple concept hierarchy defining the attribute of *Size* in OSS projects, other hierarchies are presented in Appendix C. Although each of these hierarchies defines a different concept, a "*leaf*" node may appear more than once in different hierarchies. For example, *Number of Collaborators* appears in both the concept of *Size* and of *Community Isolation*. Besides defining software attributes, these hierarchies link the concrete software metrics, i.e. their "*leaves*", to a query which retrieves that metric from the database, such as the following.

SELECT project_id, count(distinct collaborator_id) AS NumberOfCollaborators FROM ProjectCollaborator GROUP BY project_id;

OSSQuery can thus identify task relevant data by expanding all the respective concept hier-

archies. Moreover, using the queries, the system is able to retrieve the identified information from the database as we will show in the following Section.
The second, computational part of the data mining process involves three major steps:

1. The Data Miner analyzes the collected data regarding their distribution and presents a histogram, the mean value, standard deviation, minimum, and maximum value to the user.

2. The user can then refine the data set, remove outliers, and refresh the distribution analysis until he/she is satisfied with the data and decides to submit the refinement.

3. The Data Miner performs either correlation and regression, in case the concepts are of the same granularity, or clustering analysis in case they have different granularity. In either case, we compare each property, i.e. *leaf node* of the concept hierarchy, of the first attribute with each property of the second attribute. The results are presented both in graphical and textual form.

In the following, we will analyze these steps in more detail, it is important to notice though, that this process refers to the current state of implementation and can be extended to a variety of other types of analyses by simply adding or altering an analysis module (class). For example, we could support nonlinear regression by adding a class that defines the respective R Code.
First of all, we must prepare the data for different kinds of analyses. For step 1, i.e. the distribution analysis, data is simply arranged into unordered vectors. For step 3, instead, we create property pairs, i.e. pairs of ordered vectors containing the values of two measures (one of the first attribute and one of the second attribute) for all projects of the dataset. Such vector pairs are needed to analyze the relation between two properties, i.e. to calculate correlation and regression.
Although the computations, such as correlation and regression may also be directly implemented in Java, we decided to delegate this task to an external tool, hence keeping the system modular and extensible to other types of analyses. R [45] is a programming language that offers a complete software environment for statistical computing. A valid, but more specialized alternative to R is WEKA [46]. We have chosen R because of its concise syntax and the availability of a large set of packages and features. Hence, the R scripts we wrote to perform distribution, regression, correlation, and clustering analysis are simple and concise. Finally, the results are presented to the user.
The layer diagram of Figure 8 presents the technologies in the various layers of the Data Mining component in OSSQuery. To interact with the user, we created a presentational web interface using Java Server Pages (JSP) and Java Servlets. The business logic of the Data Miner application is written in Java, with exception of the analysis scripts (R) and the concept hierarchies (XML). We host the complete application with interface and business logic on an Ubuntu 10.04 LTS webserver kindly provided by the Free University of Bolzano.

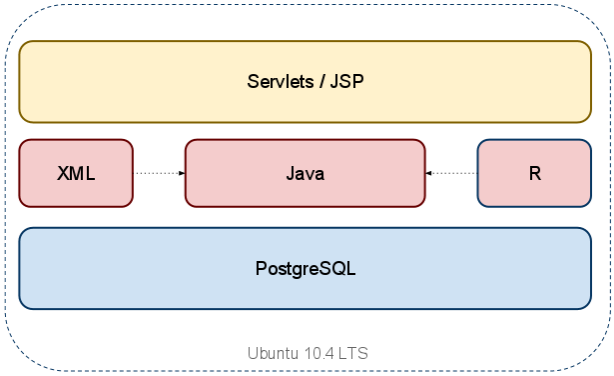


Figure 8: Data Miner Layers

5.2 Practical Problems and Solutions

Analyzing large data sets in quasi-real-time is full of practical problems. In this Section we present major problems and possible solutions to the problems for the execution of user-issued mining tasks.

Efficiency
As in collecting data, efficiency is a crucial problem in performing large-scale data mining. On the one side, the Data Miner is affected by high resource consumption due to complex computations. On the other side, users perform analyses in real-time, i.e. they launch a query and expect an immediate result, thus the Data Miner must have a low response time. These two premises force a data mining system to be efficient in retrieving, arranging, and analyzing data. To optimize data retrieval, we first need to join data of different software properties, i.e. associate and order them with respect to the project they belong to. It is definitely more efficient if such an expensive task is performed by an optimized DBMS using indices rather then by a custom implementation. Hence, we combine the queries from the concept hierarchies into one single query, thereby optimizing the joins and avoiding duplicate, redundant retrieval of some metrics. We will treat the problem of generating such a query in the following paragraph. Once the Data Miner retrieved the data from the database, we create property pairs (Section 5.1) and launch the analysis scripts in R. Refining the dataset is the only task we perform in Java. The cost of this task is linear proportional to the number of projects in the worst case ($O(n)$). Thus, except for the data refinement, all expensive operations are performed by specialized and optimized tools, i.e. PostgreSQL and R.

Query Generation
To retrieve and arrange data efficiently, we decided to generate large queries from the queries defined in the concept hierarchies (Section 5.1) automatically. Thus, we avoid the cost of joining the results of each single query in Java, and delegate this operation to the optimized DBMS. Since pre-defined queries may contain more internal "JOIN" operators in the "FROM" clause, or aggregate functions in the "SELECT" clause, such an automatic generation is not trivial. In OSSQuery, we combine the subqueries clause–by–clause, i.e. we merge all "SELECT" clauses, all "FROM" clauses, all "WHERE" clauses, etc. This approach requires an automatic labeling of all "FROM" and a relabeling of the rows in the "SELECT" clause to avoid ambiguities. Finally, we join and group the "FROM" clauses with respect to the project ID.

Interaction with R
There exist several interfaces from Java to R, the most commonly used is JRI [47], which provides an extensive low level Java API to R. For the scripts we are currently using, a more restrictive but simpler approach is more convenient. RCaller [43] provides a way to launch R scripts (in string format) directly from Java code, and it transforms result–objects (e.g. vectors of double values) from R into Java.

Besides the problems presented in this Section, we faced a number of issues in developing the Data Miner which will not be treated in detail. Some of these are representing multiple types of analyses and plots clearly, allowing an efficient, iterative refinement of the data set, and providing an understandable user interaction.

5.3 Results

This Section presents results regarding the realization of the Data Miner, more general outcomes, such as actual mining task results are given in Section 6. Moreover, we assess the solutions we proposed in Section 5.2 by giving measurements and test results.
We designed and developed an extensible, efficient, and usable Data Miner as presented previously. The application is currently active on a web-server of the Free University of Bolzano and features a web-interface to launch data mining tasks (Appendix D).
Our major concern is efficiency both in querying the database and in analyzing data. To test our efficiency, we register the time of querying data, of analyzing data, and of loading the result page. We perform tests with a connection bandwidth of 1Mbps, and we clean the caches before each test to not affect the outcomes. The following results represent the average response time for a mining task that involves 600 projects and approximately 4 mln tuples.

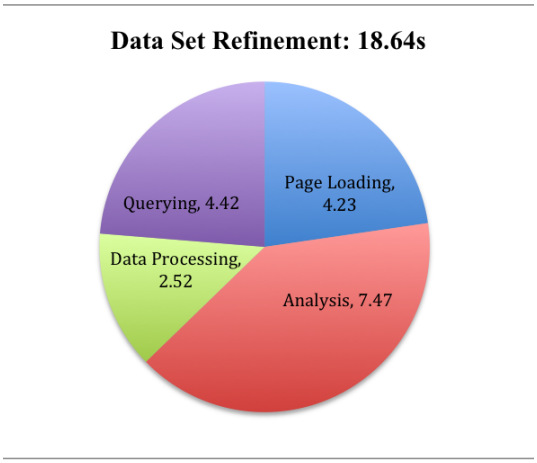


Figure 9: Data Set Refinement Time

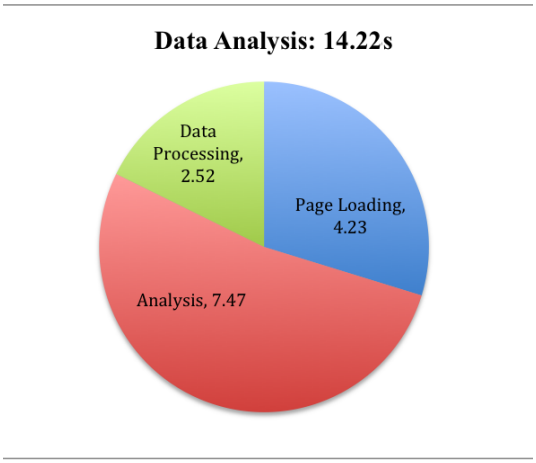


Figure 10: Data Analysis Time

Figure 9 refers to the time to assess the distribution and to present the data refinement site, whereas Figure 10 depicts the time needed to perform the investigation and to present the results. *Querying* refers to the time taken to launch a query from Java, to retrieve the data from the database, and to get the result-set in Java. *Analysis* refers to the computations performed in R, *Data Processing* refers to the time taken for all processing performed in Java, including the generation of queries and the data arrangement. *Page Loading*, instead,

is the time the results-page is loaded and thus includes the transferal of textual and graphical data, and depends on the loading time of the browser. Considering the low bandwidth, the amount of tuples to be queried, the computational complexity of the performed analyses, a response time of 18.60s and 14.22s for refining the data set and completing the analysis, respectively, seems reasonable. Other projects featuring large-scale data analyses, hosted on much faster systems [39] have response times of five to six seconds. We will present the outcomes of distribution analyses, data mining tasks, and the queries we described in Section 2.3 in Chapter 6.

This Section presented the implementation, problems, and results of realizing an efficient and functional Data Mining module to perform analyses on project information. We have explained our approach to solve major practical problems. Finally, we have presented the resulting Data Miner application and assessed its efficiency.

6 Discussion

In this Chapter, we will discuss the outcomes of the OSSQuery project. We first present the limitations of the project to define the scope of our work, then we give major results regarding the data set and the data analysis. Finally, since OSSQuery is a very broad and extensible project, we outline directions for further research beyond what was possible in the timeframe of this thesis.

6.1 Limitations

This Section delineates the scope of our work by defining which functionalities we developed and which problems we solved in the current state of OSSQuery. Ideas and suggestions for future research are presented in Section 6.3. In spite of addressing a broad range of projects through a careful selection of which features, i.e. programming languages, versioning systems, bug tracking systems, and forges to support, there are still many projects that our Data Collector does not index. In particular, we are currently limited to publicly available projects that are

- written in Java and C++ programming languages
- hosted on SourceFroge and Google Code forges
- versioned on Subversion versioning system
- using either SourceForge and Google Code bug tracker

Moreover, we limit ourselves to the collection of explicitly available information (descriptive information, logs, bug tracking, etc.) and do not aim to include other, implicit information such as communications over mailing lists, forums, or feature requests in our analyses. Nevertheless, we were able to retrieve and store a good percentage of the available data, thus building a diverse and representative data set. Neither do we intend to perform more advanced analysis on the reliability of software, e.g. testing the software to measure Mean Time To Failure. Notably, such analyses are difficult to automate and introduce a new dimension of variables that can influence the outcomes (such as the context in which software is tested). The collected data offers a remarkable range of possible analyses. In our current work, we investigate cause–effect relations as a proof of concept. Accordingly, we focused on correlation and linear regression for attributes of the same granularity, and cluster values for attributes of different granularity. Both types of analysis can indicate a strong dependence and thus a potential cause-effect relation between attributes. Notably, linear regression is not always the best measure, additional support of polynomial or exponential regression would enhance the analysis (Section 6.3). Finally, it is not yet possible to define a more complex, concise, or restrictive data mining task, e.g. to specify which kind of projects should be considered and which type of analysis should be performed.
This Section presented the major limitations to define the scope of our work and thus to better explain and support the results (Section 6.2) and future work (Section 6.3) of the OSSQuery project.

6.2 Results

The general aim of this work is to automatize the OSS research process of data collection and data investigation to allow studies on a data set representative for the population of OSS projects on the Internet. In this Section, we present results and outcomes of our solution. In

particular, we will first discuss the data set we accumulate, then we compare the results of OSSQuery to results of other, manual investigations on four research questions explained in Chapter 2. Finally, we discuss and interpret the results.
To perform analysis on a significant and representative set of information on OSS projects, we have arbitrarily chosen to take a database snapshot after indexing 100,000 and collecting 2,076 projects from SourceForge and Google Code to answer our research questions. The data collection has been running since and as of 05/07/11 we have collected information on more than 2,500 projects while indexing over 143,000 projects. In the following, we briefly analyze the data set we consider for the research questions. Agreeing with other sources [39], we found that Java projects outweigh C++ projects on the considered OSS forges (Figure 11).

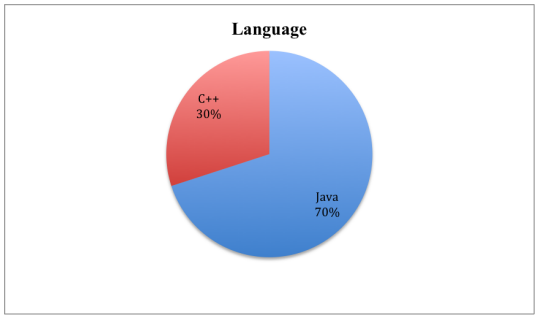


Figure 11: Distribution wrt Language

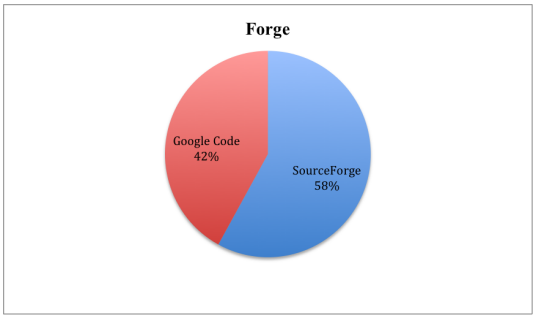


Figure 12: Distribution wrt Forge

We have also found that, indexing the same number of projects from both forges, we could collect more projects from SourceForge than from Google Code (Figure 12). The cause for this observation might be a different distribution of programming languages, versioning systems, but also a different percentage of clean and consistent projects.

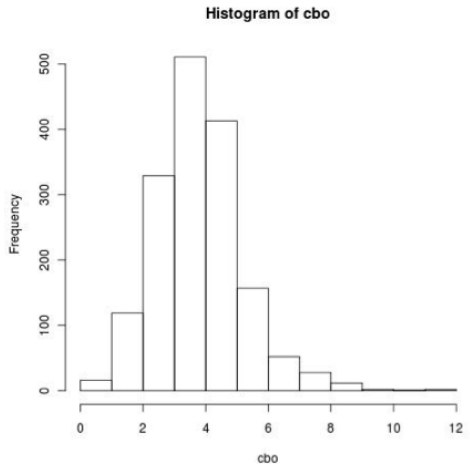


Figure 13: *CBO* Historgram

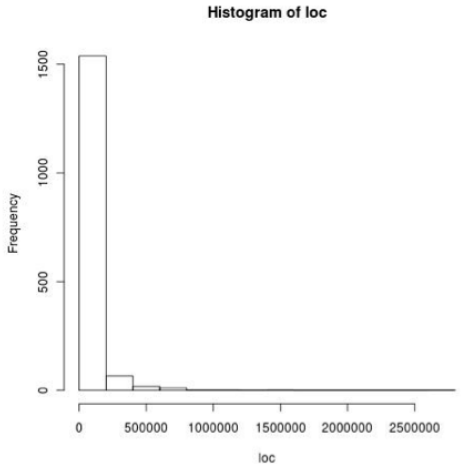


Figure 14: *LOC* Histogram

The distribution of size–related metrics is strongly skewed to the right (Figure 14), while quality–related metrics as the CK metrics suite [8] commonly follow a normal distribution around their ideal value, e.g. the *Coupling Between Objects* (*CBO*) metric (Figure 13). The distribution of some CK metrics, as *Depth of Inheritance Tree* (*DIT*) and *Weighted Methods per Class* (*WMC*) show a significant amount of outlier projects in the data set, though. From the *DIT* distribution in Figure 15, we hypothesize that many projects do not use inheritance, which leads to higher structural and computational complexity of the code. The distribution of the *WMC* metric (Figure 16) reflects this effect: a consistent amount of projects has an average *WMC* value exceeding 10.

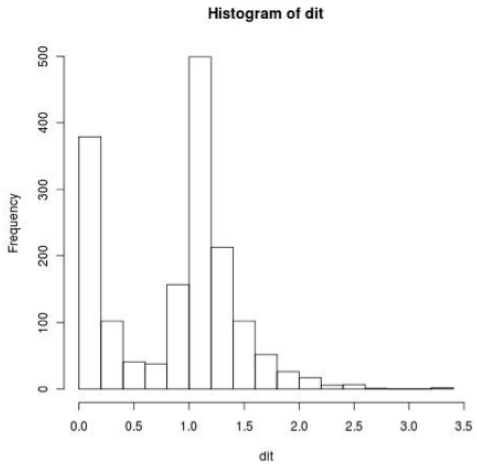


Figure 15: *DIT* Historgram

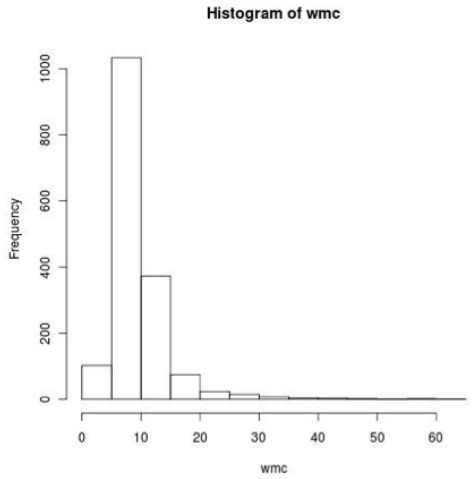


Figure 16: *WMC* Histogram

Using the data set of 2,076 projects described in the above, we will now discuss the results to four well studied research queries we explained in Chapter 2 and, where possible, compare these to results of previous, related research. To remain neutral and present representative results, we have not refined the data while performing the following analyses.

Q1. Do non-isolated communities produce quality code?
To investigate this research question, we analyze the relation between community isolation and code quality. We consider the number of collaborators, and the number of links, i.e. the weighted number of shared collaborators, to other projects as measure of isolation. Thus, a collaborator contributing in two further projects represents two links. For code quality, instead, we examine the CK metrics suite [8] and the bug density. Since the CK metrics (Section 2) are defined on class level, we perform a clustering analysis on relations involving such metrics, whereas we perform a correlation and regression analysis on relations involving bug density. While the CK metrics suite correlates the structural quality of software, bug density is a valid measure for reliability of software. The outcomes of the analyses on different relations between properties of isolation and properties of quality coincide and indicate a slight negative correlation. For example, the relation between the number of shared collaborators and the bug density shows a negative correlation of -0.08, and, correspondingly a negatively sloped regression line (Figure 17).

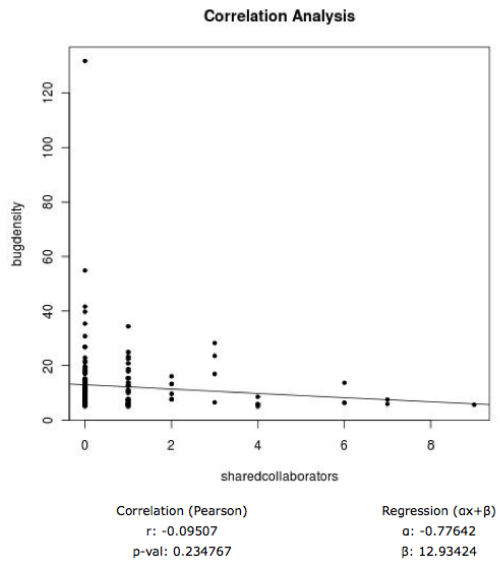


Figure 17: Correlation between *SharedCollaborators* and *BugDensity*

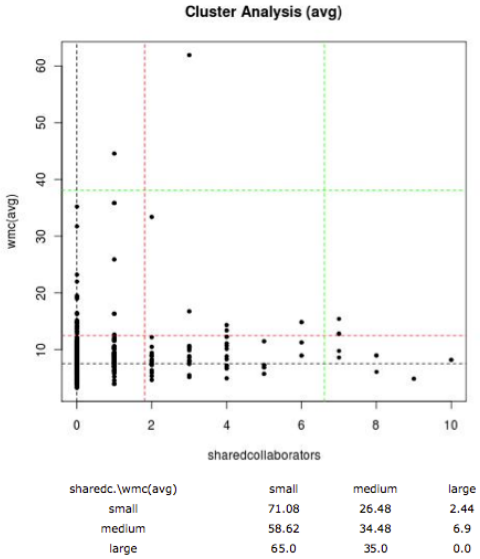


Figure 18: Clustering on *SharedCollaborators* and *WMC*

The clustering analyses agree on the fact that highly connected communities produce quality code. The relation between shared collaborators and Weighted Methods per Class (WMC) in Figure 18, for instance, shows that while isolated communities produce code with a variable WMC value, interconnected communities produce less complex code (0% of projects with non-isolated communities have a high WMC). This numbers indicate that isolated projects produce slightly more complex and thus less maintainable code. In conclusion, we agree with previous research which claims that non-isolated communities tend to produce quality code [6]. The current data set, however, does not permit the rejection or the acceptance of the hypothesis that there exists no correlation between isolation and quality.

Q2. To what extend does isolation within communities affect code activity?
Previous literature [6] claims that the developer community and especially the collaboration with other communities influence the activity of a project. We considered churn, commit, and code-rate as measures for activity and the weighted number of shared collaborators as measure for community isolation. The churn rate measures the average change in the code between two commits, the commit rate indicates the frequency of commits, and the code rate measures the growth rate of the code. While the commit rate is positively correlated with the number of shared collaborators, the churn rate, which is a strong indicator for software evolution, is negatively correlated with isolation (Figure 19). This trend indicates a more stable, controlled use of the versioning system by non-isolated communities and may be the effect of a more organized and systematic development process. However, we cannot make any inference on the true correlation at any reasonable confidence level due to a too high p-value.

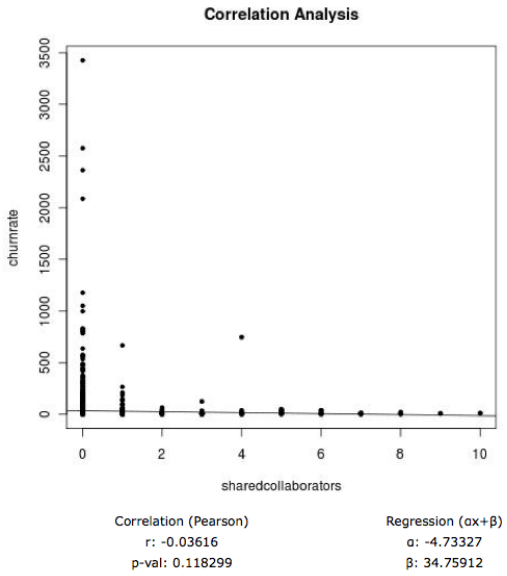


Figure 19: Correlation between *SharedCollaborators* and *ChurnRate*

Q3. Does size affect code quality?
According to Lehman's Laws of Software Evolution 1,6, and 7 [19] software is continuously growing in size and complexity, and quality is decreasing if the software is not rigorously maintained. Thus, this research query investigates the quality of different sized projects but, following the above laws, also the state of maintenance of OSS. Our results indicate a negative correlation between size and quality.

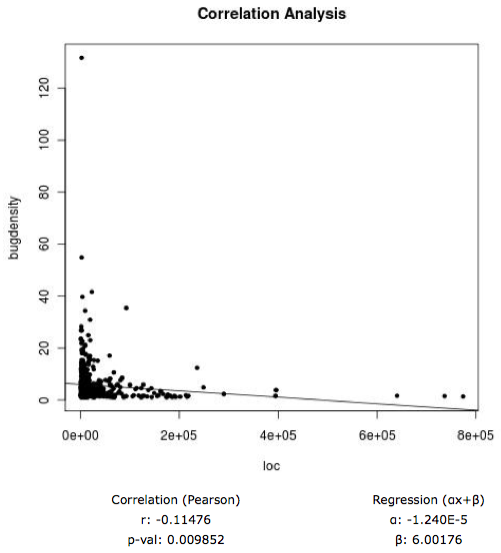


Figure 20: Correlation between *LOC* and *BugDensity*

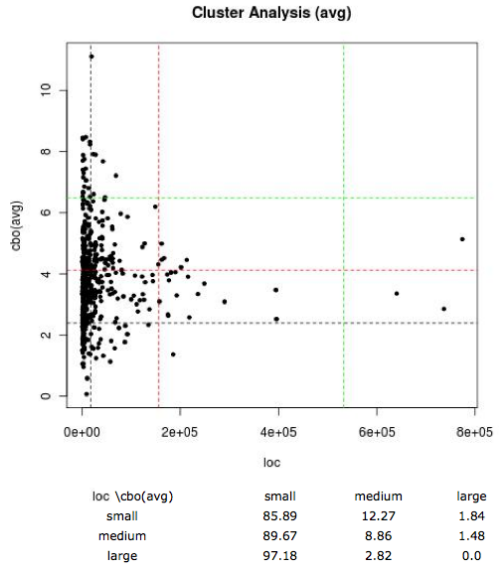


Figure 21: Clustering on *LOC* and *CBO*

The relations of both Bug Density and *CBO* to *LOC* indicate that larger projects tend to be less faulty and less complex than smaller project. The p-value permits a rejection of the H_0 hypothesis with a confidence level of 99% and thus shows a strong validity of this correlation. This result is not in contrast to Lehman's law though, it shows that larger projects are generally more rigorously maintained and therefore provide high quality code. Our results agree with the results of previous research [40] which claims that software quality is positively correlated to size, and show that online forges generally contain a large amount of small, low quality projects.

Q4. Does activity affect project maturity?
Open Source Software (OSS) generally suffers from the permanent beta syndrome. We aim to discover whether a high activity generally leads to higher project maturity and thus can prevent projects from this syndrome. Our analysis indicates a negative correlation between activity and maturity.

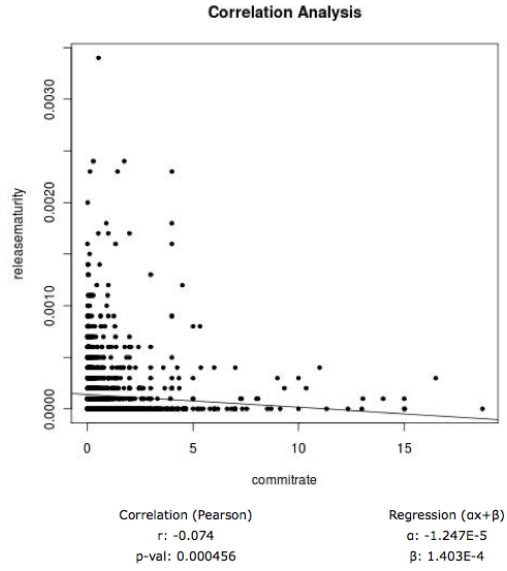


Figure 22: Correlation between *CommitRate* and *ReleaseMaturity*

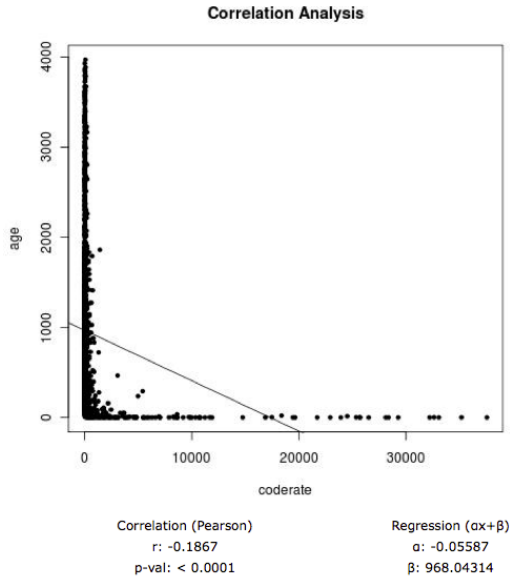


Figure 23: Correlation on *Coderate* and *Age*

Maturity is measured in terms of age, the release maturity i.e. releases per line of code, and, analogously, revision maturity, measures for activity where presented in Q2. The relations of different measures for activity and maturity coincide and indicate that projects with high process activity are commonly less mature. As stated in Chapter 2, correlation cannot detect the direction of causality, though. Therefore it is probable that low project maturity causes high activity, i.e. that projects experience high development activity in early stages and reach a stable activity at some point. The negative correlation is supported by a low p-value, permitting the rejection of the hypothesis, that correlation would be zero, with a confidence level of 99 %.
We conclude this Section by outlining and interpreting the results presented above. Our major contribution through this work is an extensible data mining system capable of indexing a huge number of OSS projects (>143,000), collecting a large data set (>2,500) of projects with different characteristics, and performing investigation on cause-effect relations through

different analysis types. The collected dataset corresponds to the findings of other tools [39] and to the population of OSS projects on forges. The tests on different queries (Q1 – Q4) show general agreement with other research on smaller samples [1] [2] [6] [34]. The analyses on the data set and on the queries, however, present some divergences with respect to other literature. The analyses on the Queries demonstrate, that especially small projects do not show a common trend, as reported in previous research. These facts lead to weakly supported correlations, e.g. in Q1 and Q2, and make out possible dissimilarities to results of other research.

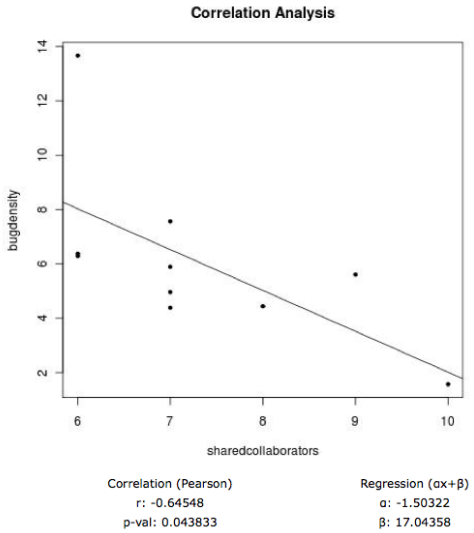


Figure 24: Correlation betw. *SharedCollaborators* and *BugDensity*

Figure 24 depicts the result of Q1 "Do non isolated communities produce high quality code?" if we filter out smaller projects. The correlation is now strongly supported by a much lower p-value and a clear tendency is visible. These results sustain the hypothesis, that the weak support, especially in Q1 and Q2, the apparently random distribution of metrics for some projects, and differences to other research are caused by a high number of small, though consistent projects. We deduce that many of these projects are developed with no specific process, strategy, or organization. Considering that the projects of our data set are already filtered against small and inconsistent projects, we can infer that such projects constitute a significant proportion of all OSS. Our criticism, however, is not directed to this part of projects, which does not fit the models and results of previous research in OSS. We rather claim that investigating on a small set of projects can lead to outlier analyses and results. Such research results [1] [2] [6] [34] do not represent the general situation of OSS, on the contrary it may mislead to the idea, that OSS fits precise models and results.

6.3 Future Work

The process of knowledge extraction (Figure 1) combines a wide range of concepts and techniques, but also different fields of computer science. Accordingly, the automation of extracting knowledge from large online forges encompasses this complexity, but opens a range of possibilities as well. Neither do we claim to collect all available OSS project information (Section 4), nor do we claim to perform all possible analyses on the collected data (Section 5). However, we designed and developed an extensible and modular architecture that follows a transparent

method and can readily accommodate any type of collection from online forges and any type of data mining tasks (Section 3). Thus, both the Data Collector and the Data Miner can easily be enhanced to analyze an even larger proportion of OSS with different types of mining tasks.

The Data Collector currently supports a limited set of project characteristics. Although we have focused and implemented support for the most popular and widespread characteristics, a number of projects are still neglected. The support of additional programming languages, e.g. of C# or Python, versioning systems, e.g. CVS or GitRepository, and forges, e.g. Launchpad or GNU Savannah, would expand the proportion of indexed projects.

The Data Miner currently supports the analysis of cause–effect relations through correlation, regression, and clustering analyses. Both the collected data and the extensible structure of the Data Miner permit an easy extension in types of analyses and attributes to investigate. Other data mining tasks, as classification, different clustering algorithms, or even unguided association rule mining may provide significant results in the field of OSS.

The support of different types of analysis and an even larger data set call for a powerful query language to better define advanced data mining tasks. OSSQuery, in its current state, additionally collects descriptions and labels which could be used to better define the data set by allowing keywords in the query. Previous research [21] proposes a valid and applicable solution to this problem.

In conclusion, OSSQuery is an extensible and performant data mining system that collects large data sets and provides plausible results. Comparison with other, manual research has shown that an automatic, quantitative investigation may provide generally valid, representative results and trends. Moreover, we presented a number of possible enhancements for our system, and its possible contributions to research. The combination of a large data set, a variety of analyses, and a strong formalism to launch queries would provide a complete and extensive data mining system for the general analysis of OSS.

7 Conclusion

Our major contribution through this work is a system that automates the process of identifying, collecting, cleaning, and analyzing open source projects from the web to allow quantitative and comparable research on OSS. Such automation involves numerous conceptual complexities and practical issues both in collecting and in analyzing data. While widely available, information on OSS is scattered across places in heterogeneous forms. Moreover, to extract all information from the available data, it is necessary to analyze source code, versioning system, and bug tracker of projects with radically different properties. Finally, to perform useful analyses and investigations, we must support a variety of queries on abstract and general software attributes, e.g. code quality or project size.
We have designed and developed OSSQuery, a data mining system for investigation on cause–effect relations. It applies methods of information retrieval, as web crawling, and of data mining, as correlation and clustering analyses, on the universe of publicly available OSS data to extract valuable knowledge on e.g. relations between software attributes. Hence, OSSQuery aims to facilitate investigation and discovery of valid and representative results on different research questions on cause-effect relations in OSS.
The method and design of our approach seek for an efficient, extensible, and reliable system that can accommodate the collection of projects with radically different properties, e.g. programming languages, and a variety of different mining tasks through elegant architectural patterns. In its current state, OSSQuery collects, stores, and updates C++ and Java projects from SourceForge and GoogleCode iteratively. Moreover, the system is able to investigate on cause–effect relations among different software attributes through correlation, regression, and clustering analyses on demand.
As of July 2011, OSSQuery identified, indexed, and filtered over 143,000 projects constructing a data set of more than 2,500 projects and we expect this data set yet to grow. Thus, the database stores a comprehensive amount of information of the projects in approximately 18 mln tuples. The composition and distribution of this data set correspond to the findings of other tools, thus we can deduce that it is representative for the universe of OSS.
As a proof-of-concept, we performed different analyses on much studied research questions using OSSQuery. In the following, we will briefly present the results and a corresponding interpretation of such investigations.

1. Do non-isolated communities produce quality code? Our results show that non-isolated communities generally produce high quality code, the support for this claim is rather low, though.

2. To what extent does isolation within the developer community affect code activity? Our investigation confirms that non-isolated communities follow a more constant, regular development activity.

3. Does size affect code quality? The results clearly show that larger projects tend to be of superior quality.

4. Does activity indicate project maturity? Our investigation demonstrates that mature projects usually have a constant, lower activity than new projects.

Our results are generally similar to those of other, manually conducted analyses on the same problems. The findings of such manual research, do only fit the characteristics of a minor proportion of projects, though. Thus, we claim that research on small groups of projects may not provide results that are valid or representative for all OSS. Our system is designed

and built to be extended. We currently collect project information from Google Code and SourceForge and perform investigations on cause-effect relations. The support of further forges and analyses would increase the value of OSSQuery and allow researchers to perform any type of quantitative investigation on open source.
To our knowledge, while other approaches simply collect and store general project data [4], our system is the only tool that actively performs quantitative analyses on OSS attributes. We are convinced that OSSQuery can be useful to conduct large-scale investigations and allows a re-evaluation of current beliefs as well as a discovery of new results in the field of OSS.

References

[1] M. Zhao, C. Wohlin, N. Ohlsson, and M. Xie, "A Comparison between Software Design and Code Metrics for the Prediction of Software Fault Content", *Information and Software Technology*, 1998.

[2] A. Mockus, R.T. Fielding, and J.D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2002

[3] A.G. Koru, D. Zhang, and H. Liu, "Modeling the Effect of Size on Defect Proneness for Open-Source Software", *In Proceedings of the Third International Workshop on Predictor Models in Software Engineering,* 2007

[4] J. Howison, M. Conklin, and K. Crowston, "FLOSSmole: A collaborative repository for FLOSS research data and analyses", *International Journal of Information Technology and Web Engineering,* 2006

[5] J. Howison and K. Crowston, "The perils and pitfalls of mining SourceForge", *In Proceedings of the International Workshop on Mining Software Repositories*, 2004

[6] K. Nakakoji, Y. Yamamoto, Y Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities", *In Proceedings of the International Workshop on Principles of Software Evolution*, 2002

[7] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach, 2nd,* PWS Publishing Co. Boston, MA, USA, 1998

[8] S.R. Chidamber and C.F. Kemerer, "A metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, 1994

[9] C. Olston and M. Najork, *Foundations and Trends in Information Retrieval, 4th,* Now Publishers Inc., Hanover MA, USA, 2010

[10] A. Heydon and M. Najork, "Mercator: A scalable, extensible Web crawler", *World Wide Web Vol. 2*, 1999

[11] Q. Tan and P. Mitra, "Clustering based incremental web crawling", *ACM Transactions on Information Systems (TOIS),* 2010

[12] J. Han and M. Kamber, *Data Mining, Concepts and Techniques, 2nd,* Morgan Kaufmann Publishers, Elsevier, 2006

[13] G. F. Cooper and E. Herskovits, "A Bayesian Method for the Induction of Probabilistic Networks from Data", *Machine Learning Vol. 9*, 1992

[14] V.T. Chakaravarthy, V Pandit, S Roy, P. Awasthi, and M. Mohania, "Decision trees for entity identification: approximation algorithms and hardness results" *In Proceedings of the 26th symposium on Principles of database systems,* 2007

[15] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases" *In Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[16] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation" *Data Mining and Knowledge Discovery*, 2004

[17] J. Cho, H. Garcia-Molina, and L. Page, "Efficient crawling through URL ordering", *Computer Networks and ISDN Systems*, 1998

[18] J. Lerner and J. Tirole, "Some simple economics of open source" *The journal of industrial economics*, 2002

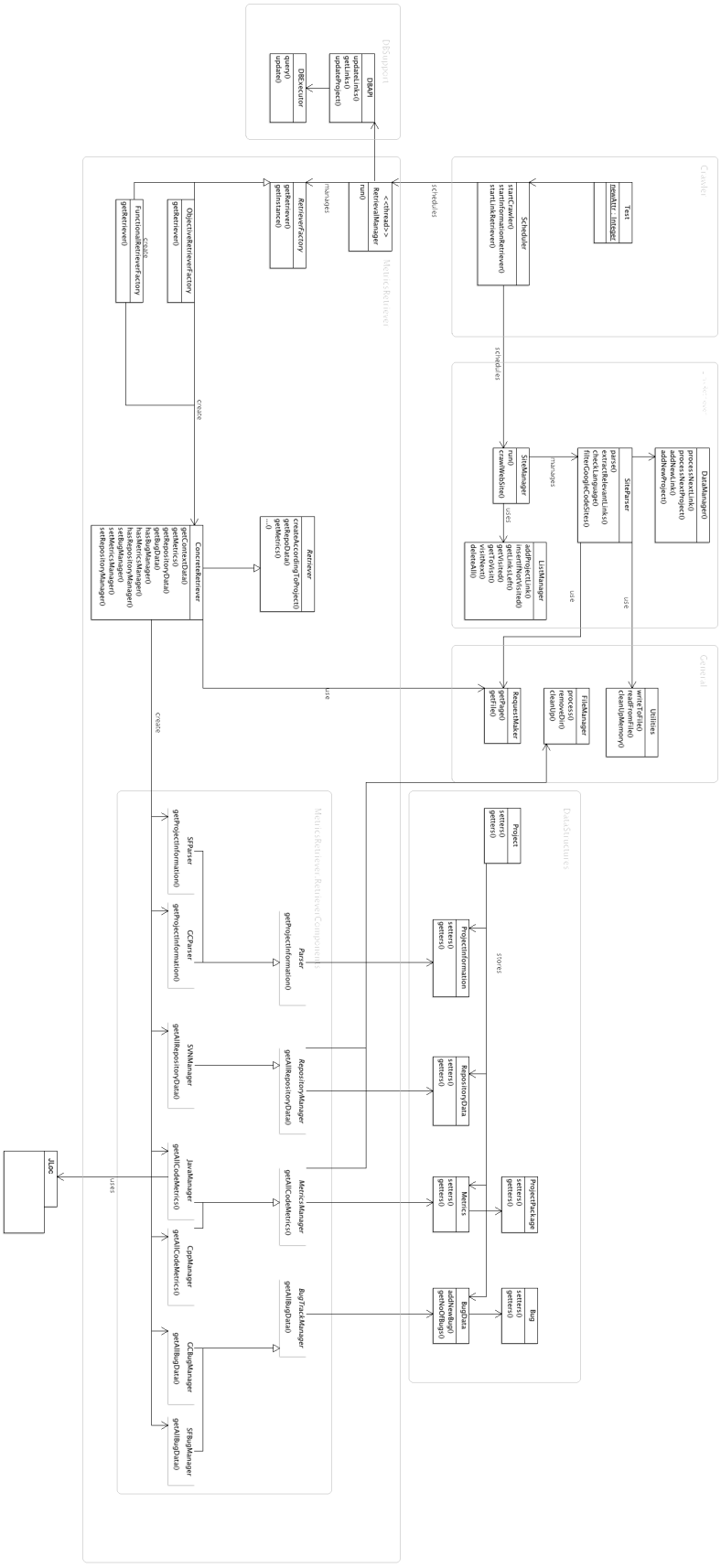
[19] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski "Metrics and laws of software evolution", *In 4th International Software Metrics Symposium*, 1997

[20] J. Han and Y. Fu, "Dynamic generation and refinement of concept hierarchies for knowledge discovery in databases", *AAAI Technical Report*, 1994

[21] J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane, "DMQL: A data mining query language for relational databases", *In Workshop for research issues on Data Mining and Knowledge Discovery*, 1996
[22] D. Weiss, "A Large Crawl and Quantitative Analysis of Open Source Projects Hosted on SourceForge" *Research report ra-001/05*, 2005
[23] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history" *In IEEE International Working Conference on Software repositories*, 2009
[24] B. Kitchener, "What's up with software metrics? - A preliminary mapping study" *Journal of Systems and Software*, 2010
[25] M. Weiss, G. Moroiu, and P. Zhao, "Evolution of Open Source Communities", *IFIP International Federation for Information Processing*, 2006
[26] J. Cho and U. Schonfeld, "RankMass crawler: a crawler with high personalized PageRank coverage guarantee" *In Proceedings of the 33rd international conference on Very large data bases*, 2007
[27] SVNKit 1.3.5 , *https://svnkit.com* TMate software,
[28] Google Data API, *http://code.google.com/apis/gdata/,* Google
[29] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools", *In Proceedings of the international symposium on Software testing and analysis*, 2008
[30] D. D. Spinellis, "CKJM - A Tool for Calculating Chidamber and Kemerer Java Metrics", *http://www.spinellis.gr/sw/ckjm/,* (Retrieved in March 2011)
[31] T. Littlefair "CCCC - C and C++ Code Counter", *http://cccc.sourceforge.net/ ,* (Retrieved in March 2011)
[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994
[33] D. D. Spinellis "Tool writing: A forgotten art?" *IEEE Software,* 2005
[34] H. M. Olague, L. H. Etzkorn, and G. W. Cox, "An entropybased approach to assessing object-oriented software maintainability and degradation - a method and case study", *In proceedings of the International Conference on Software Engineering Research and Practice*, 2006
[35] V. Crescenzi, G. Mecca, and P. Merialdom, "RoadRunner: Towards automatic data extraction from largeWeb sites", *In Proceedings of the 26th International Conference on Very Large Data Bases*, 2001
[36] S. Chengalur-Smith and A. Sidorova, "Survival of Open-Source Projects: A Population Ecology Perspective", *In ICIS 2003 Proceedings*, 2003
[37] H. M. Olague, L.H. Etzkorn, S. Gholston, and Stephen Quattlebaum, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes", *IEEE Transactions on Software Engineering*, 2007
[38] Tiobe Software, *http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html* , (Retrieved in May 2011)
[39] Black Duck Software, *www.ohloh.net*, (Retrieved in May 2011)
[40] D. E. Harter, M. S. Krishnan, and S. A. Slaughter, "Effects of Process Maturity on Quality, Cycle Time, and Effort in Software Product Development", *Management Science*, 2000
[41] S. Weber, *The success of open source*, Harvard University Press, USA, 2004
[42] JLoc, *http://jloc.sourceforge.net/*, (Retrieved in March 2011)
[43] M. H. Satman, "RCaller - Another way of Calling R from Java", *http://www.mhsatman.com/rcaller.php,* 2011
[44] E. Dubmill, "DOAP – description of a project", *http://trac.usefulinc.com/doap* (Retrieved in March 2011)

[45] R Development Core Team, “R: A Language and Environment for Statistical Computing”, *R Foundation for Statistical Computing,* 2009
[46] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I. H. Witten “The WEKA Data Mining Software: An Update”, *SIGKDD Explorations*, 2009
[47] RJava Development Team “JRI - Java/R Interface”, *http://www.rforge.net/rJava* (Retrieved in May 2011)

Appendix A : Data Collector Design



Appendix B : Further Statistics on Data Collection

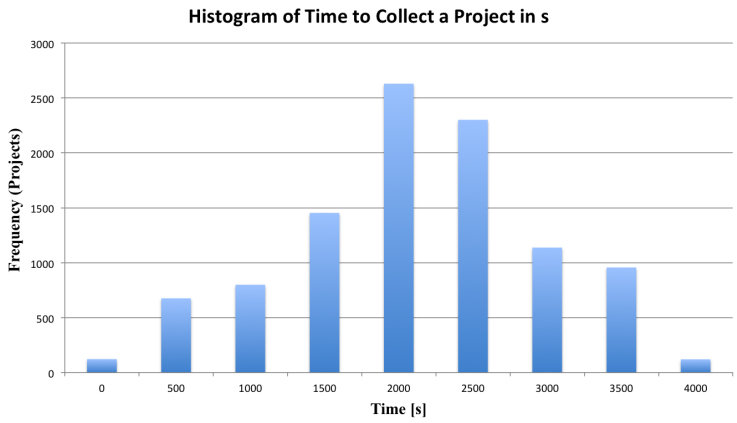


Figure 25: Histogram of Time to Collect a Project's Information [s]

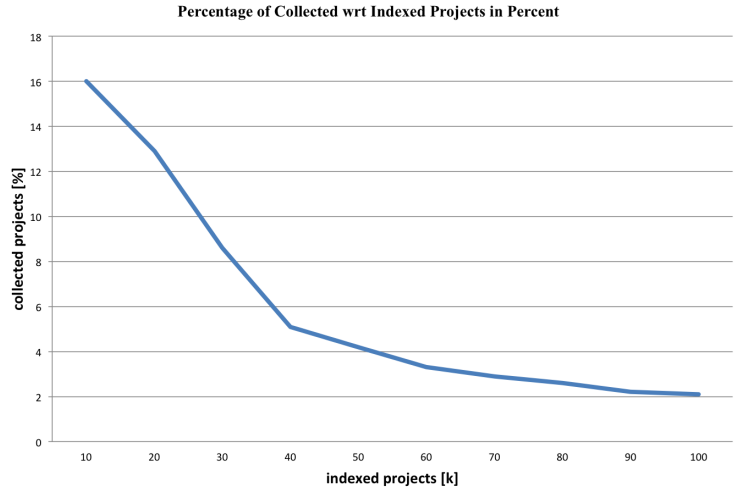


Figure 26: Number of Good Projects per Indexed Projects [%]

Appendix C : Concept Hierarchies

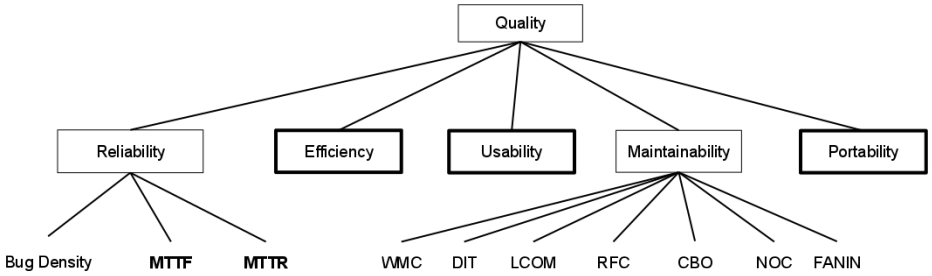


Figure 27: Concept Hierarchy of *Quality*

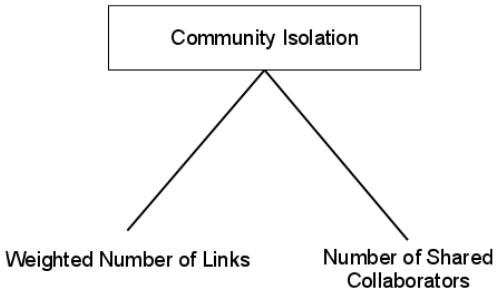


Figure 28: Concept Hierarchy of *Community Isolation*

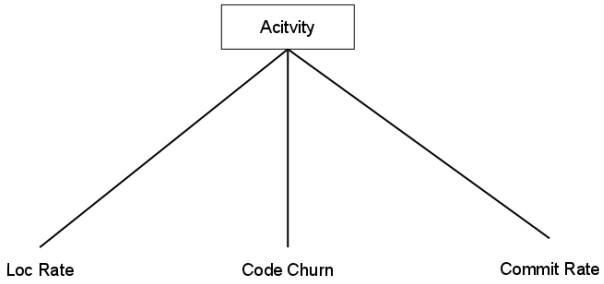


Figure 29: Concept Hierarchy of *Activity*

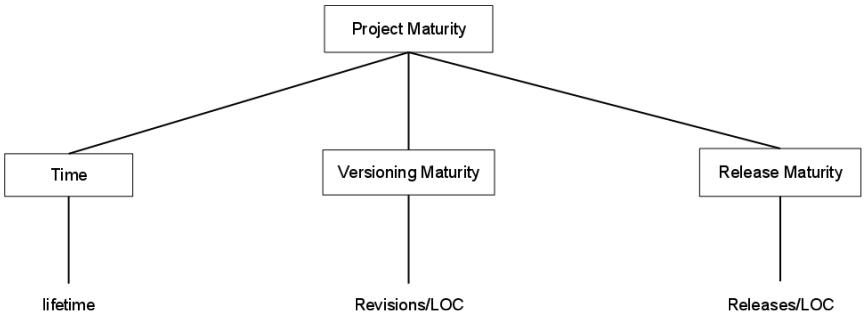


Figure 30: Concept Hierarchy of *Maturity*

Appendix D : OSSQuery Screenshots

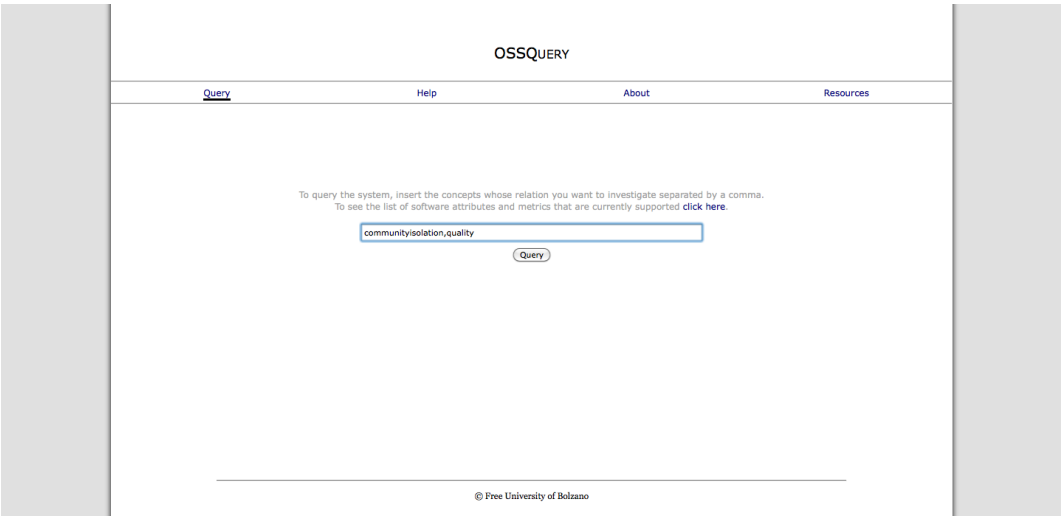


Figure 31: Screenshot of OSSQuery Homepage

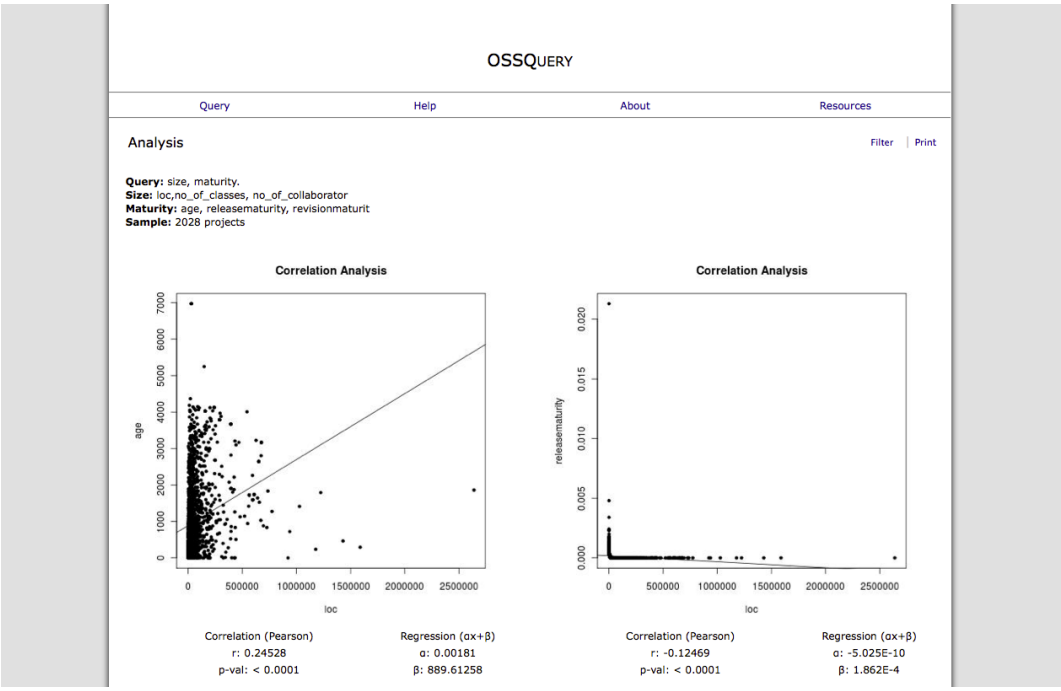


Figure 32: Screenshot of a Analysis Results

List of Figures

1 The Process of Knowledge Discovery . . . 6
2 Logical Representation of Information Flow among OSSQuery Modules . . . 10
3 Abstract Factory Pattern in the Data Collector . . . 11
4 Unified Project Data Model . . . 14
5 Data Mining Process . . . 14
6 OSS Distribution over 100,000 Projects . . . 21
7 Concept Hierarchy for Size . . . 23
8 Data Miner Layers . . . 25
9 Data Set Refinement Time . . . 26
10 Data Analysis Time . . . 26
11 Distribution wrt Language . . . 29
12 Distribution wrt Forge . . . 29
13 *CBO* Historgram . . . 29
14 *LOC* Histogram . . . 29
15 *DIT* Historgram . . . 30
16 *WMC* Histogram . . . 30
17 Correlation between *SharedCollaborators* and *BugDensity* . . . 31
18 Clustering on *SharedCollaborators* and *WMC* . . . 31
19 Correlation between *SharedCollaborators* and *ChurnRate* . . . 32
20 Correlation between *LOC* and *BugDensity* . . . 32
21 Clustering on *LOC* and *CBO* . . . 32
22 Correlation between *CommitRate* and *ReleaseMaturity* . . . 33
23 Correlation on *Coderate* and *Age* . . . 33
24 Correlation betw. *SharedCollaborators* and *BugDensity* . . . 34
25 Histogram of Time to Collect a Project's Information [s] . . . 42
26 Number of Good Projects per Indexed Projects [%] . . . 42
27 Concept Hierarchy of *Quality* . . . 43
28 Concept Hierarchy of *Community Isolation* . . . 43
29 Concept Hierarchy of *Activity* . . . 44
30 Concept Hierarchy of *Maturity* . . . 44
31 Screenshot of OSSQuery Homepage . . . 45
32 Screenshot of a Analysis Results . . . 45