Master-Thesis in Computer Science

Towards a Representative and Diverse Analysis of Issue-Tracker Related Code and Process Metrics

Maximilian Capraro

December 16, 2013

Supervised by Prof. Dr. Dirk Riehle, MBA

Professorship of Open Source Software

Friedrich-Alexander-University Erlangen-Nürnberg

Abstract

Issue Tracking Systems of Open Source Software projects deliver manifold insights to the regarding projects. Due to the huge diversity in the format and structure of the different Issue Tracker product's data, performing a large-scale empirical analysis to derive these insights is not a trivial task. We present a uniform integrated data model that is capable of storing data from various Issue Tracker products. Based on that, we developed a toolkit for crawling and incrementally updating real world Issue Tracking data into this model. We identified 8 Key Information Needs stakeholders of Open Source Software projects have regarding the data from Issue Tracking Systems. Most of these information needs can be satisfied by a set of Visualizations and Metrics presented within the thesis. Based on data we collected from 13228 projects, we perform an empirical analysis on the structure of the user base. Also, we demonstrate that release dates have no significant and immediate effect on the activity within Issue Tracking Systems. We discuss multiple correlations between Trends in the Issue Tracking Systems' data.

Acknowledgments

First and foremost, I would like to thank my supervisor Prof. Dirk Riehle for his valuable guidance and advice during the course of this thesis project. Also, his encouragement and support to write my master thesis in Boston enabled me to broaden my academic and personal horizon in manifold ways.

I have greatly benefited from the inspiring environment at Black Duck Software. I am particularly grateful to the enormous contribution of Dr. Sushil K. Bajracharya and Abhay Mujumdar. Their constructive comments, innovative ideas and continuous feedback were invaluable. I am thankful to Andreas Zink for his support and for the opportunity to execute my thesis project at Black Duck Software. All of my colleagues at Black Duck Software - especially Peter Degen-Portnoy, Jared Barboza, Benjamin Dubrovsky and James Berets - were never hesitating to share their experience with me. Thank you.

Finally, I wish to thank my parents and friends for their support, understanding and encouragement throughout my study.

Contents

List of Abbreviations

GQM	Goal Question Metric approach
ITS	Issue Tracking System
JSON	JavaScript Object Notation
NOI	Number of Issues
NOIL	Number of Issues linked to this Issue
NOIsT	Number of Issue's Transactions
NOU	Number of Users
NOUsIT	Number of User's Issue Transactions
OSS	Open Source Software
SCM	Source Code Management
SE	Software Engineering
UML	Unified Modeling Language

List of Figures

List of Tables

Part I.

Preliminary Considerations

1. Introduction

1.1. Preamble to the Topic

Issue Tracking Systems (ITS) contain a diverse set of information regarding software development projects, the people that work within these projects and the outcomes that are produced. Depending on the specific project and its culture just technical discussions on the resolution of bugs but also strategic and abstract debates about new features take place within ITS.

Open Source Software (OSS) projects as well as Closed Software projects and firms make heavy use of ITS. As manifold publications show (see section 1.3), the task of triaging, understanding and resolving issues and bug binds a huge portion of each project's resources.

The data of Issue Tracking Systems is highly relevant for various forms of research also in the empirical Software Engineering (SE). For OSS projects an imposing amount of this data is available publicly and with nearly unrestricted access. Carefully analyzed, this data can teach a lot and give new insights about an Open Source project - for example regarding the project's community or the its evolution. The aggregated data from a huge amount of ITS can allow conclusions about Open Source Software development practices in general.

Because of this it is an important challenge to make the data accessible and unleash its potential for empirical research. This thesis will introduce an implementation that enables to normalize and use the data of a huge amount of Issue Tracking Systems. Also derived insights will be presented.

1.2. Context of the Thesis Project

This thesis was performed within a joint project of Black Duck Software Inc., Burlington, MA and the Open Source Research Group at the Friedrich-Alexander-University Erlangen-Nürnberg under supervision of Prof. Dr. Dirk Riehle, M.B.A.

The work was carried out within Black Duck Software's team that is responsible for the development and maintenance of Ohloh.net. Ohloh.net is a online community platform maintained by Black Duck Software. It displays various visualizations and metrics based on measures from mainly Source Code Management systems. As of November 2013 Ohloh.net contains data about 658.775 projects of which 350.744 are estimated to be active.

A part of the findings presented in this thesis are candidates for future features of Ohloh.net or other Black Duck Software products.

1.3. Relevant Literature in the Field

An overwhelming amount of publications regarding the analysis of Issue Tracking Systems exists. As we do not aim to provide a complete literature survey this section will present only those publication that were most influencal for our work or needed to understand the context of this thesis. Most of the publications aim to solve practical problems Open Source projects experience using their ITS.

Triaging of Issues Multiple publications present solutions for triaging the stack of issues a project team might be confronted with. While Jalbert & Westley [29] describe a semi-automated approach for filtering duplicates, Cavalcanti et al. [10] and Bortis [7] present tools that support project teams in triaging and filtering huge amounts of issues. Robbes [41] and Anvik [4] present approaches for determining experts within ITS instance. This can support the manual triaging of issues. Joeng et al. [30] discuss the reassignments of inadequately triaged issues.

Quality of Issues Various publications deliver a broader understanding on the quality of issue reports within an ITS. Hooimeijer & Weimer [27] and Bettenburg et al. [6] postulate a definition for the quality of issue reports. The quality model of Hooimeijer & Weimer is simple, quality can be estimated with few resources. The quality definition of Bettenburg et al. [6] is more precise. Their automated approach for determining the quality of reports is more complex. Lotufo & Czarnecki [33] present general methods for improving the comprehensibility of bug and issue reports.

Process Quality and Efficiency A set of publications utilizes the data from ITS to estimate a project teams efficiency. Francalanci & Merlot [20] present a visualization and a metric for the efficiency in resolving issues. Kim & Whitehead [31], Panjer [38] and Weiss et al.[53] analyse the time needed to resolve issues. While Kim & Whitehead [31] utilize Source Code Management Systems (SCM), Weiss et al. [53] and Panjer [38] utilize ITS data.

Extracting Semantics An assortment of publications makes use of the semantics of the data stored in ITS. Martie et al. [35] present an approach to extract topics that are currently discussed within an ITS. Tran et al. [50] discusses search methods within ITS data. Fischer et al. [16] describe the extraction and tracking of features based from ITS.

Issues as a Measure for Software Quality The opinion of authors diverge whether ITS data can be used to estimate the quality of software artifacts. Glott et al. [22], Hassan et al. [24] and Yu et al. [56] argue that ITS data is valid for estimating the regarding software quality. Raja & Tretter [40] argue that ITS data can not be utilized directly to estimate software quality. They discuss that an increasing amount of issue reports, primarily indicats an increasing interest in an OSS project.

Utilizing ITS for Emprical Research The scientific community has perceived the importance of ITS data for understanding OSS projects better. E.g. Open Source Quality models as described by Soto & Ciolkowski [47] or as discussed by Glott et al. [22] also utilize ITS metrics.

However no large-scale empirical analysis of ITS data was identified. Most publications (e.g. [43], [27], [38], [42], [10], [3], [11]) only discuss one project or a small set of projects for performing an analysis. Tran et al. [50] briefly discusses the required normalizations that are needed to perform an empirical study with data from diverse ITS products. Their model is simpler and less expressive than the model we will present.

1.4. Contributions of this Thesis

With this thesis we contribute software artifacts and features but also give answer to some specific empirical research questions.

As part of the *engineering work* to process data from Issue Tracking Systems (ITS) for empirical research we make the following contributions:

1. Presentation of a uniform model capable of expressing data from various Issue Tracking Systems.

2. Development of a toolkit for populating and incrementally updating data within this model on a large scale.

Also we make the following contributions to deliver a *broader insight* to specifics of ITS in Open Source environments:

3. Identification of 8 Key Information Needs regarding Issue Tracking Systems.

4. Development and Implementation of 7 Visualizations that satisfy Key Information Needs.

Further we answer a set of research questions. A reduced set of projects from Ohloh.net - those hosted on Github.com - is used within this thesis. We first needed to analyze the data set's capabilities. We will proof the following hypotheses:

- **H0.1** A sample of active Github.com projects is not representative in regards to nor does it cover the diversity of the universe of Open Source Software projects.

We want to achieve a broader understanding of the community structure within OSS projects. This results in the research question:

- **RQ1** What is the structure of the communities within Issue Tracking Systems of Open Source projects?

We will show that the following answers are valid:

- **H1.1** Within most of the ITS (91.1%) 40% to 70% of the users report only one issue.

- **H1.2** The community of most ITS (94.6%) consists of 5% to 30% team members.

We are interested in what influences the trends and activities within ITS. We therefore formulate the following research question:

- **RQ2** How do trends within the ITS depend on each other?

We will provide the following answers:

- **H2.1** The Closing Trend and the Commit Activity are significantly correlated.

- **H2.2** The Trend of Active Users correlates with the Issue Opening Trend.

- **H2.3** The amount of Reopened Issues and the Efficiency Trend are not correlated.

Release dates are an important day in every project team's calendar. We want to explore the influence of release dates further:

- **RQ3** How do release dates influence the activity within ITS?

We will proof the following hypotheses:

- **H3.1** Release Dates do not induce an immediate peak in the Opening or Closing Trend of an ITS instance.

The next section will give an overview on the structure of the thesis.

1.5. Structure of the Thesis

The thesis consists of 9 chapters. The content of each chapter will be described briefly in the following paragraphs.

Chapter 1 introduces the user to the topic. The meaning of ITS within a SE context is explained. Meta information about the thesis is provided. A list of relevant publications this thesis' contributions is presented.

Chapter 2 delivers basic definitions and concepts regarding metrics within an SE context. The chapter introduces the basic concepts of Measure, Metric and Indication. Classification approaches for metrics are discussed. It is described that metrics can be validated qualitatively and quantitatively.

Chapter 3 describes an integrated data model. This model is capable of representing the data of various ITS instances and products. The model expresses normalizations of different values with similar semantics. It is capable of tracking the historic changes within ITS data.

Chapter 4 introduces an implemented toolkit for crawling and incrementally updating the data. The updating mechanisms are described in detail. It is discussed how the toolkit properly reacts to errors without falsifying the stored data. The toolkit is also capable of pre-computing analyses within the data.

Chapter 5 discusses information needs of contributors and users of OSS regarding the data gathered from ITS instances. The information needs are extracted from various research publications.

Chapter 6 introduces to a set of visualization and metrics to further understand the ITS data of various projects. The described visualizations will be implemented as features for the Ohloh.net platform.

Chapter 7 gives an overview on the projects that are currently stored within the database. An evaluation of their diversity is given. The diversity is analyzed in regards of project size, used programming language and age of the project.

Chapter 8 describes some findings in the data. Cross-correlations between Trends in the ITS data are presented. The structure of the user community and the influence of release dates to its activity are discussed.

Chapter 9 closes the thesis with a summary. Potential threads to the validity of the results are discussed. The last chapter also suggests a set of ideas for future research based on the findings within this thesis.

2. Metrics and Measures

The introduction in the first chapter, gave an idea of the potentials of ITS data and the usage of Measurements and Metrics this thesis. Before later chapters of this thesis will describe the engineering work for gathering the data and the results of the data analysis; this chapter will give an overview of basic concepts and definitions regarding Measurements in the Software Engineering context.

2.1. Metrics - Adding Semantics to a Measure

2.1.1. Concepts of Measure, Metric and Indicator

For the scope of this thesis we utilize the three basic concepts Measure, Metric and Indicator as suggested by Pressmann [39].

A Measure is the basic concept: It provides an expression "of the extent, amount, dimension, or size of some attribute of a (software) product or process" [39]. It expresses the attribute quantitatively and is comparable. Also, a Measure, in contrast to a Metric, does not aggregate other Measures and is atomic. Because of this we extend the definition of Pressmann [39]:

> A Measure x_i provides a quantitative expression of the extent, amount, dimension, or size of an attribute i of a software product or process. A Measure is atomic, and not aggregated or composed of other Measures.

The specific scale or data type of a Measure can vary. While some Measures may result in natural or rational numbers; others may deliver categorical or logical results. Also an event stream, or a part of it, can be seen as a Measure resulting in a time series. For our domain a time series of issues transitioned to state open or closed as well as the number of currently open issues are examples for a Measurement.

A Metric adds semantics and a meaning to one or more Measures. It aggregates Measures or performs other calculations with the measured values. We therefor define a Metric as follows:

A Metric M is a function $M(x_1, \ldots, x_n)$ with $n > 0$. It aggregates n Measures and performs calculations. The resulting value – as well as the Measures x_i given as parameters – can be of various data types.

It is important to mention that this definition of metrics diverts from the definition given by IEEE 1061 [28] which does not distinguish the concepts of Measure, Metric and Indication explicitly.

A Metric or a set of Metrics is selected to enable an observer to draw conclusions from the data. While the Metric delivers an interpretable number, an Indicator delivers a deeper understanding and is base for an informed decision. We define an Indicator as follows:

An Indicator I provides insights that enable individuals looking at the Metric results to adjust the environment, the processes or products.

The next subsection will give an example of Measures, Metrics and Indicators from the domain of analyzing ITS data.

2.1.2. Examples for Measures, Metrics and Indicators

Various measures can be performed within an Issue Tracking System (ITS). Two fruitful measures can be the measurements of opened and closed issues resulting in time series $x_{resolved}$ and x_{opened}. While $x_{resolved}$ consists of a set of time stamps that express the points in time an arbitrary issue of the ITS got resolved, the set of timestamps in x_{opened} expresses all the points in time an issue was opened.

The metrics $M_{monthly-opened}$ and $M_{monthly-resolved}$ can be applied. These metrics aggregate the measures by counting the resolved ($x_{resolved}$) respectively opened (x_{opened}) issues per month. Assuming the example ITS had a lifetime of k months, the result will be a tuple of months $T = (t_1, ..., t_k)$ with $\forall t_i \epsilon T \exists t_i \rightarrow a_i$. Each month t in T has a mapped amount of opened respectively resolved issues a_i.

Applied for our example ITS the metrics delivered the following results ($k = 5$ with a ITS lifetime from January to May 2012):

$$M_{opened} = ((Jan12 \rightarrow 10), (Feb12 \rightarrow 15), (Mar12 \rightarrow 20), (Apr12 \rightarrow 25)) \tag{2.1}$$

$$M_{resolved} = ((Jan12 \rightarrow 10), (Feb12 \rightarrow 15), (Mar12 \rightarrow 15), (Apr12 \rightarrow 15)) \tag{2.2}$$

A simple Indicator I could estimate the efficiency of the project team: Whenever $M_{resolved} - M_{opened}$ for a month t_i is negative, this can be seen as an indicator for a project team that is not efficient enough to resolve issues in a timely manner. The environment could be adjusted by trying to establish a broader contributor base, increasing the triaging efforts or other actions.

The last section defined and explained the concepts of Measures, Metrics and Indicators. The next section will give an overview in how Metrics can be classified according to different approaches.

2.2. Classification of Metrics

Metrics in the field of Software Engineering can be classified in manifold ways. A frequently used approach is the approach by De Marco [13] that classifies the metrics based on the underlying measures in Source-Code- and Process-Metrics. This classification is lacking an important part of the information regarding a software project. For producing or maintaining software more than good processes and a high-quality code base is needed. Also the available resources play a key role. Because of that, other authors, as Fenton and Neil [15] or Caldiera et al. [9], use a classification approach that takes this aspect into account by adding a third class of metrics - namely Resource-Metrics.

Both classification approaches were designed for the context of a Closed-Source development environment. In Open-Source individuals and organizations, influenced by own schedules and/or own interests, perform a huge part of the work. Some of the individuals may follow a specific process or be driven by processes of the organizations they belong to. We therefore believe that a strict distinction between Process- and Resource-Metrics for Open-Source projects is not beneficial. We rather want to understand the community and its actors as a whole.

We therefore classify Metrics into the classes Source-Code- or Community Metrics:

- A Source-Code-Metric delivers an insight about the technical software artifacts and is primarily based on measures within the Source Code of a Software Product
- A Community-Metric delivers an insight about an Open-Source community and covers processes as well as human resources and other specifics that are not covered by purely looking at the source code.

Community-Metrics can be classified further. A straightforward approach, utilized by the Ohloh.net team, is to classify the Community-Metrics by the system that is source for most of the measures. Community-Metrics could be classified in the following groups. (This list is not intended to be exhaustive.)

- Issue-Tracker-Metrics

- Mailing-List-Metrics
- Bulletin-Board-Metrics
- Wiki- / Documentation-Metrics

While this approach is easy and simple to use in practice, it might be misleading. A relevant metric could rely on measurements from a variety of systems. This would weaken a classification using this approach. The weakness does not have much relevance for our considerations: Following the presented approach, most of the metrics relevant for this thesis can be classified as Community-Metrics from the sub-class Issue-Tracker-Metrics.

The next section will give an overview, on how metrics should be selected in real world projects. It will be discussed where the drawbacks of such a selection approach for a large scale analyses as in this thesis are.

2.3. Selection of Metrics

The last sections described different concepts regarding metrics and how metrics should be classified. This section will discuss how relevant metrics can be selected.

The frequent performing of measures, the calculation and surveillance of metric results, and finally the appropriate reaction to indications of a metric result are expensive. Because of that it is important to carefully select metrics that are relevant for a project.

A frequently used approach for selecting metrics is the Goal-Question-Metric approach (GQM) presented by Calidera et al. [9].

The most distinct attribute of GQM is the selection of metrics with a top-down approach. Using GQM project teams do not search for things they can measure, but identify information needs first. To find metrics using GQM a project team needs to go through 3 levels:

1. Conceptual Level - Goal: The Organization needs to define an abstract goal that should be accomplished by tracking a metric. An example of a goal is the improvement of the efficiency in a specific process.
2. Operational Level - Question: By utilizing the knowledge about the environment and context of the project and the organization that hosts it, a set of specific questions must be elaborated. The questions are based on the defined goals.
3. Quantitative Level - Metrics: A set of measures, metrics and indicators needs to be defined. A metric can be associated to one or more questions; also one question can be associated to one or more metrics.

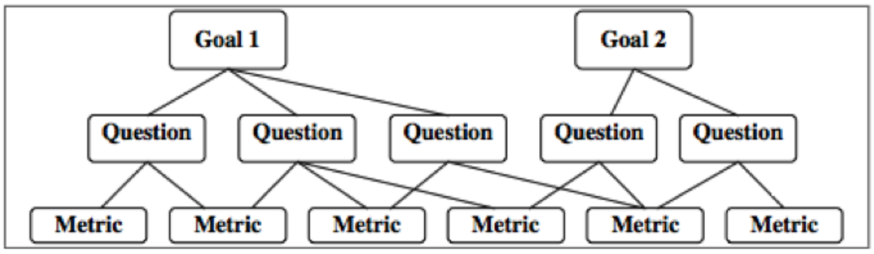


Figure 2.1.: Relation of Goals, Questions and Metrics [9]

Figure 2.1 illustrates the relation of conceptual, operational and quantitative level.

In this thesis we do not need to select metrics for one project or a small set of projects. We aim for the whole universe of Open Source projects - at least those projects on Ohloh.net. This creates two major problems in selecting interesting metrics:

1. We cannot utilize all the context information regarding each project while selecting the questions that will lead to the metrics.

2. We cannot formulate a specific goal or specific goals for each project.

These problems will be addressed as follows: As we cannot utilize context information for defining the questions and metrics for each project, we will aim for taking specifics into account that characterize Open Source projects in general. The information will be presented objectively and individuals looking at it can interpret the displayed information based on their own experiences and background knowledge regarding the specific projects.

Instead of formulating specific business goals we will identify a set of general information needs (See chapter 5). While this strategy will decrease the precision of the showcased metrics regarding specific projects it will allow us to deliver valid results for a larger set of projects.

The next section will discuss quality attributes of metrics and introduce to approaches for validating a metric's correctness.

2.4. Quality of a Metric

2.4.1. Quality Attributes

A simple approach for validating and determining a metric's quality are the quality attributes defined by Balzert [5]. According to Balzert a metric of high quality needs to fulfill the following attributes:

1. Objectivity: The results of the metric need to be independent of the individual performing measures and calculations.

2. Validity: The metric delivers results about what it is expected to deliver. No unknown or not communicated artifacts or measurements interfere with the results.

3. Repeatability: The measures and calculations can be repeated at any point in time. The result will stay the same.

4. Usefulness: The result has a use and meaning to users.

5. Comparability: The outcomes of the metrics are comparable - internally and externally.

6. Economically: Using the metric justifies the cost of performing the measures and calculations.

Objectivity can easily be created though a formal definition or automated implementation of the measure processes and metrics. Also repeatability can be achieved in similar ways. To ensure repeatability it is also important to not introduce non-deterministic specifics into the process of calculation the metric results (for example by performing calculations on a randomly selected subset of the measures for performance reasons).

To achieve external and internal comparability is not an easy task. Internal comparability may be achieved by using appropriate scales. Achieving external comparability is difficult due to specifics and context sensitivity in every project.

The usefulness and economics of a metric are relevant for us even though our economic goals are different from project teams that may implement metrics: We do not want to measure and understand specific projects but want to deliver general information about a huge set of projects that may be of interest for the users of Ohloh.net. An economical metric from our perspective is primarily a metric that justifies its implementation costs by providing a value to the Ohloh.net users.

Assess the validity of a metric is also not a trivial task. Within the next two sections we will present approaches for validating metrics quantitatively and qualitatively.

2.4.2. Quantitative Validation

As metrics aim to deliver quantitative and objective insights into software processes and artifacts it makes sense to also evaluate a metric's validity using a quantitative approach. Schneidewind [44] presented a quantitative validation approach for a set of Metrics $\mu = \{M_1, ..., M_n\}$. The approach follows the following algorithm (We strongly simplified the approach for brevity):

1. Calculate the metrics in μ for a set project of A in a project phase p_i and store the results.

2. Select a set of quality factors F. The results of the metrics in phase p_i are suspected to predict quality factors F in the next project phase p_{i+1}. These factors are collected by questioning users of project A.

3. Examine whether a correlation among the metrics in μ in phase p_i and the quality factors in F in phase p_{i+1} exists.

4. Repeat for multiple project phases and projects.

The correlation among the metrics and the quality factors is the validity criterion of this approach. For the following reasons we assume this approach be not fit for usage within the scope of this thesis:

1. The approach aims for validating metrics for usage in one project or a set of similar projects. We cannot collect quality factors F for a large-scale analysis of projects by questioning users.

2. We cannot assume to have all projects using a plan-driven roadmap with defined project phases p_i. Further, we cannot assume that, if defined project phases exist, these are externally comparable from project to project.

3. The approach assumes that a metric M out of μ is an indirect expression for a solid quality factor F. While this may be the case for most Closed Sources projects, we do not believe that a set of simple but solid and valid quality factors for Open Source projects exist. Taking the high dynamics of an open source project into account, the quality factor itself would need intensive validation.

The problem mentioned in (1) can be resolved by using more generic quality factors instead of those gathered by questioning the users for each project. Also the drawback described in (2) can be minimized by introducing arbitrary time slices for the assumed time delay.

Point (3) describes a problem that cannot be solved easily within the scope of this thesis. Even though general Open Source Quality models exist ([22], [47]), we are unaware of any set of quality factors F that could be utilized for a validation as described in the last section.

We argue, that the presented approach is not fit for validating metrics in a large-scale analysis of multiple Open Source projects. Next section will describe how qualitative validation approaches could be performed. It will be demonstrated that qualitative approaches are reducing the objectivity and precision of a validation but are a reasonable and effective approach.

2.4.3. Qualitative Validation

In contrast to the quantitative validation presented in the last subsection, the qualitative validation is not based on numbers but on subjective opinions of a selected set of representative individuals.

Such a validation can be performed using a survey method [49]: Individuals are presented a survey e.g. in form of a questionnaire. The individuals are presented with metrics regarding specific project they are familiar with. Then they answer how helpful these metrics were in regards to specific information needs (or questions and goals as described in section 2.3).

A qualitative approach is less precise than a quantitative approach. Humans are always a factor to that possibly introduces threads to validity. A larger and/or more representative set of individuals can enhance the quality of a qualitative validation using the surveying method.

In section 2.3 we described that the interpretation of the metrics can be very project specific and is not easy to capture for a huge amount of projects as in our large-scale analysis. The selected individuals can use their own experiences and context knowledge about the presented projects to decide how valid the presented metrics are.

We argue that for the selection of metrics in a large-scale project as this, a qualitative validation approach is more beneficial than a quantitative approach.

This chapter introduced to basic concepts regarding measurements and metrics in OSS projects. The three concepts of measure, metrics and indicator were introduced. We described classification approaches for metrics and discussed why the classification in Community- and Code-Metrics is the most beneficial for this thesis project.

The Goal-Question-Metric approach (GQM) was briefly presented and it was described that a top-down approach inspired by GQM can be used to select relevant metrics for this project.

A set of 6 quality attributes for metrics within the context of Software Engineering was presented. It was discussed why a qualitative validation approach is superior to a quantitative validation within the context of this project.

While this chapter focused on a preliminary theoretical discussion of metrics in Software Engineering, the chapters of the following part of the thsis will describe the engineering work performed to enable measurements and the calculation of metrics for Issue Tracking Systems.

Part II.

Engineering Aspects

3. Uniform and Integrated Data Model

3.1. Demand for a Uniform Integrated Data Model

Past chapters described the motivation to perform Measurements in ITS and the theories behind it. We discussed how measurements and metrics quantify and describe different facts and circumstances of concrete ITS instances; their application within a project can satisfy the various information demands.

Algorithms that implement analyses and Measurements in ITS require input data in a unified format that contains all the needed attributes and information. Due to the huge amount of ITS products, each with a specific underlying data model, such a uniform data pool is not preexisting. Instead each ITS product, even each ITS instance, can vary in how the data is stored, and in which information is tracked at all.

For example, the Jira ITS on the one hand tracks a huge selection of data. A typical Jira instance contains information about an issue's type, its priority, a history of states, a textual issue description, users' comments on the issue, and other attributes. Simpler ITS like the GitHub.com ITS on the other hand may track only a subset of this data. The GitHub.com ITS does not store the types or priorities of issues. Also data present in both, the Jira and GitHub.com ITS, is of different format and structure.

Most existing work performed on the analysis of open and closed source software project's ITS does not address the issue of the heterogeneous data pool: While a lot of publications (e.g. [43], [27], [38]]) only discuss one project or a small set of projects for validating their hypotheses or performing an analysis, other research projects ([3], [11], [42], [10]) avoid problems in normalizing the data by focusing on one specific ITS product. Open Source Quality Models ([22], [47]) use such a small subset of available ITS data attributes, that further efforts for data normalization are trivial or not necessary.

Tran et al. [50] present a uniform model for ITS data. It does consider priorities, issue types, resolution, states and other attributes. It does not track historic changes of issues. Our model is intendet to enable empirical research within ITS data. The model of Tran et al. is designed to enable semantic searches within ITS data. We will not discuss their model any further.

For performing a broad empirical analysis as required for this thesis, data of multiple ITS instances and products needs to be accessed. Therefore we developed a uniform integrated data model which is capable of representing data integrated from various ITS products and instances.

The following sections of this chapter will explain key decisions made in designing and implementing the data model. Section 3.2 will compare the data used by a selection of ITS products. The resulting uniform integrated data model for ITS data will be described in section 3.5.

3.2. Comparing Different ITS Products' Data

3.2.1. Bugzilla, Jira and Github.com as Reference ITS

For the comparison of the data in different ITS products, the following products will be analyzed:

1. Bugzilla (Version 4.2.6)
2. Jira (Version 6.0)
3. Github.com ITS (Version of August 1st, 2013)

The standard configurations of these 3 ITS products will serve as representatives.

The Bugzilla ITS was chosen due to its high prevalence within the Open Source community. The spread of Bugzilla was determined by using the Ohloh.net database. On May 21, 2013 a selection of 20 projects was performed, which the most Ohloh.net users were affirming to use. Of these projects, 5 were using a Bugzilla instance as ITS. 15 projects were using other ITS products or mailing lists to communicate about bugs.

The Jira ITS was selected because it is believed to have a high spread within enterprise scale software projects. Jira is also used within Black Duck Software Inc., the company that hosts this thesis project. An understanding of the characteristics of Jira's data will ensure that the methodologies and implementations, derived from this thesis, can also be used outside the Open Source context, for example at Black Duck Software or its customers. Jira also has a huge relevance for Open Source communities. For example most Apache Foundation projects (like Hadoop, Jena, Lucene, Struts) use Jira ITS.

The Github.com ITS will be considered as a representative of ITS products with a smaller functionality. A relevant amount of active Open Source projects is hosted on free software forges like Github.com (26.3% of Ohloh.net's projects). Some of these projects have no need for a complex ITS, like Jira or Bugzilla, and use the built-in ITS products. For this reason it is important to also take the specifics of smaller ITS products into account while performing the analysis of data features. We estimate Github.com to be of high relevance for the Open Source community in general. The "Mining Challenge" of the conference "Mining Software Repositories" of year 2014 is to work with Github.com's data, as provided by Gousios [23].

3.2.2. Diversity of Data Attributes

Extraction of Data Attributes

We extracted the data attributes used by each of the ITS tracker. For extracting the attributes, the following algorithm was used:

1. For each ITS product, a sample instance was selected. (For Jira and Bugzilla we used the demonstration instances [1] the vendors provide. To explore the Github.com ITS we used the instance of the JUnit project).
2. For each ITS product, data attributes were manually extracted from the sample instances using the part of the graphical user interface that displays the data regarding a single issue.
3. The resulting lists of data attributes were consolidated. Duplicates were eliminated and attribute names were normalized. For example the title of an issue is referred to as "alias" in Bugzilla and "summary" in Jira.

In total 23 data attributes were extracted. Table 3.1 gives a summarized overview on attributes implemented by the 3 sample ITS products. Each row represents one attribute. The columns represent the names each attribute has in the specific ITS product. Different names of attributes with a similar meaning (for example Alias, Title and Summary) were given one unique name under the column "name".

Values for some attributes can be changed over the lifetime of an issue. Whenever an attribute can be changed, but the issue tracker does not store the chronicle of the historic values, the attribute was marked with a star (*) in table 3.1. The issue of storing historic values will be addressed separately in section 3.4.

We grouped the identified attributes into 7 categories:

- General: General data about an issue
- Environment: Data about the environment the issue occurred in

[1]The demonstration instance of Jira can be found at *https://www.atlassian.com/de/software/jira/demo*. The demonstration instance of Bugzilla can be found at *http://landfill.bugzilla.org/*. Both links were checked last in November 2013.

	Name	**Bugzilla**	**Jira**	**Github.com**
General	Title †	Alias	Summary	Title *
	Description †	Description	Description	Description *
	Type †		Type	
	Attachment	Attachment	Attachment	Attachment
Environment	Version †	Version	Affected Version	
	Hardware	Hardware		
	Operation System	Operation System		
	Environment D.		Environment	
Network	Issue Relations	(depend., links)	(comp., links)	(links)
	SCM Relations		(existing)	(existing)
Grouping	Project	Product	Project	
	Component †	Component	Component	
	Labels †	Keywords	Labels	Labels *
Prioritization	Priority †	Priority	Priority	
	Severity	Severity		
	Milestone †	Target Milestone	Fix Version	Milestone *
Status	State †	Status	Status	Status
	Resolution †	Resolution	Resolution	
Community	Comments †	Comments	Comments *	Comments *
	Reporter †	Reporter	Reporter	Reporter
	Assignee †	Assignee	Assignee	Assignee
	QA Contact	QA Contact		
	Watcher	CC-List	Watchers *	

Table 3.1.: Attributes of various Issue Tracking Systems

- Network: Information about an issues relation to other artifacts
- Grouping: Data enabling to further group or classify an issue
- Prioritization: Information about the priority of an issue
- Status: Information about an issues status
- Community: Data regarding the people involved into the issue

Comparison of Results

Bugzilla provides the most (22) attributes to describe an issue. Jira (16) and Github.com (11) store less attributes regarding an issue. While comparing the data attributes we made the following observations:

- Github.com's data model of an issue is less expressive than Jira's and Bugzilla's in describing an issue.
- 8 Attributes are common in all analyzed ITS products.

- Github.com does not track changes in most of the attributes (5). Jira does not track changes in some it its attributes (2).
- Github.com does not track one single attribute about the environment of an issue.
- Bugzilla and Github.com do not track the type of the issue. Github.com however encourages its users to use labels to distinguish between different issue types.
- Bugzilla has a more structured approach to store data about the environment of an issue than Jira. Jira allows users to enter a free text to describe the issue's environment.
- Each issue tracker is capable of expressing relations among issues. While a Jira issue can be composed out of other issues, a Bugzilla issue can carry information about an issue it depends on. All analyzed issue trackers allow the usages of links to other issues in an issue's comments.
- Bugzilla does not support linking between an issue and source code management artifacts.
- Github.com does not allow the user to express sub-projects or components. However such a clustering can be implemented using the label attribute.
- Github.com's state model is much less complex than the state models of Jira and Bugzilla.

These findings clearly show a huge diversity in the expressiveness of the different ITS product's data.

Resolving the Diversity of Attributes

For designing an integrated and uniform model, the fact, that each issue tracker has a different set of attributes needs to be addressed. This can be done in the following two ways:

A first strategy is to just track an intersection of the attributes of all the relevant issue trackers. The model would lose a big part of the ITS products' expressiveness: For the selected 3 sample ITS products this would result in tracking 8 out of 23 attributes. A model created with such a reduced set of attributes is simpler. Less normalization work needs to be performed.

A second strategy is to implement a super set of all the attributes. Even though the model is capable of storing all the attributes, an attribute which, was not implemented by the ITS product we load data from, can be left out. For example an issue from the Github.com ITS can be expressed using a super set of Github.com's attributes. The values of the attributes we have no information about - like severity, priority, resolution and other attributes in Github.com - can be simply set to a ”null” value.

Our goal is it to develop a data model that is as expressive as possible. Losing of information must be avoided whenever possible. Because of this it is more beneficial to design a model capable of expressing a super set of the identified attributes instead of just being focused on the small intersection set.

3.2.3. Identification of Relevant Attributes for the Data Model

Gathering data to populate the model with, binds a certain amount of calculation and bandwidth resources. For the first iteration we therefore aim to reduce the set of attributes. We only consider those attributes marked with a cross (†) in table 3.1. 9 attributes (Attachment, Hardware, Operation System, Severity, Environment Description, SCM Relations, Sub-Project, QA-Contact and Watcher) will not be discussed further nor be part of the integrated model.

In appendix A a list of 49 ITS Metrics and Measurements is presented. By ignoring the 10 afore mentioned attributes one of these Metrics (NOIL - Number of Linked Issues) cannot be calculated based on the model. Two other Metrics can still be calculated but their results may be influenced by omitting the attributes (NOIsT - Number of Issues Transactions, NOUsIT - Number of User’s Issue Transactions). The limitation of the expressiveness does not hinder the analyses described in later parts of this thesis.

The last paragraphs described how diversity of the different issue tracker’s attributes affects their expressiveness. A model using a super set of all relevant attributes was shown to be a beneficial approach to tackle this diversity. The next section will give an overview on the diversity of the values certain attributes can hold. We will present a strategy to normalize the values of some attributes to a set of values with well-known semantics.

3.3. Normalizing Values using Naive Mapping

The attributes described in the last section mostly hold values that are very specific regarding certain projects or issues. A milestone or an affected version, exists in one specific project, and has no semantic when observed out of the project’s scope. Summary or Description texts even lose their semantic meaning when observed outside the context of one specific issue. By nature these values are very divers. It is difficult or impossible to design an algorithm for an automated normalization of these attribute’s values.

For a small subset of the attributes it is possible to automatically extract semantic information. Whenever the value does not lose its semantics when observed outside the context of the issue or project and it is a categorical attributes, it is fairly easy to normalize the values of the attribute: Each possible value is mapped to one element of a reduced set of values with well-known semantics. We call this approach Naive Mapping. Such normalization enables the user of the data to easily implement analyses of the data without making assumptions about the underlying ITS product.

The following 4 attributes were identified to be categorical and not project specific. Therefore they can be normalized using naive mapping:

- Type: The type of an issue, for example whether an issue is a bug or a feature request, has a similar meaning across projects.
- Priority: A higher priority denotes a higher importance of an issue, regardless of a specific project's context.
- Resolution: Some resolutions for an issue, are estimated to occur in a huge amount of ITS products and instances.
- State: The variability of state models in different ITS products and instances

The next paragraphs will describe how the mapping to known values is performed for the 4 identified attributes.

3.3.1. Normalization of Issue Types

Each issue type in the model can be assigned to one of the following normalized issue types:

- Bug: The issue is describing a fault or bug
- Feature: The issue is a proposal for a new feature

Table 3.2 gives an overview on how the mappings are performed. The columns show the normalized types. The rows display the 3 sample ITS products. The cells of the table display which of the normalized issue types the issue types are assigned to.

Bugzilla does not track any information regarding the type of an issue. As shown in table 3.2, all Bugzilla issues will not hold any issue type in our model. Bugzilla issues carry a severity attribute, which can hold the value "Enhancement". Issues that are flagged to be an enhancement are not necessarily requests or proposals for new features. A number of relevant projects store code quality, refactoring issues as well as feature requests as enhancements or bugs. [2],[16]. This leads to the conclusion that the presence of this value cannot be used as an indication for the issue type.

	Bug	**Feature**	**No Issue Type**
Bugzilla			All Issues.
Jira	Issues Type "Bug" / "Problem"	Issues Type "New Feature	Other Issues.
Github	Issues with a label "Bug"	Issues with a label "Feature"	Other Issues.

Table 3.2.: Naive Mapping of Issue Types

Github.com encourages its users to use labels for marking an issues type. The standard configuration of each Github.com ITS instance is configured to know the labels "bug" and "feature". However a lot of projects do not use these labels. For a Github.com project not using one of these labels, no normalized issue type will be assigned.

Jira stores rich information about the type of each issue. Mapping the type of a Jira issue to a normalized issue type is trivial.

3.3.2. Normalization of Priorities

Priorities have an ordinal scale. The priorities available in different ITS products can only differ in the granularity. Only 3 normalized priorities are defined, to avoid problems while integrating data from ITS instances that only contain coarse-grained priority information:

- Increased Priority: Issues that have a higher than normal priority.
- Normal Priority: The standard value of the issue tracker; an average neither increased nor decreased priority.
- Decreased Priority: Issues that are of lower priority than the normal priority.

Table 3.3 gives an overview on how the priorities of the different ITS products will be mapped to normalized priorities. The columns show the normalized priorities. The rows display the 3 sample ITS products. The cells of the table display which of the normalized priorities the native priorities are assigned to.

The Bugzilla Documentation [17] states issues of priority P1 and P2 to be likely to be implemented while issues of priority P4 and P5 are not likely to be implemented in future. The definition of P3 does not give any indication for a decreased or increased priority:

> "[P3 means] this isn't a bad idea, and maybe we'll want to implement it at some point in the future [...] Some core Bugzilla developer may work on it." [17]

	Increased P.	**Normal P.**	**Decreased P.**	**No Priority**
Bugzilla	P1, P2	P3	P4, P5	
Jira	Blocker, Critical, Major	Minor	Trivial	
Github				All Issues.

Table 3.3.: Naive Mapping of Issue Priorities

Because of this we map P3 to be the "normal Priority". P1 and P2 are mapped to be "increased Priorities" while P4 and P5 express a decreased priority.

Jira uses "minor" as the standard priority for newly created issues: We map this priority to the normalized priority "Normal Priority". Whenever a Jira user chooses to assign another priority, he actively decides to increase or decrease the priority. We therefore identified "Blocker", "Critical" and "Major" to express an increased priority, while "Trivial" expresses a decreased priority. Github.com ITS does not track priorities for its issues. Because of that we assume all Github.com issues to have no normalized priority.

3.3.3. Normalization of Resolutions

Resolutions further specify how an issue was resolved. We identified the following normalized resolutions:

- Fixed: Work was performed to resolve the issue.
- Duplicate: The issue will be discarded because it is a duplicated of another issue.
- Invalid: The issue is invalid but its resolution is not "Duplicate" or "Will-Not-Fix".
- Will-Not-Fix: For an unknown reason the project team decided to perform no work on the issue.

Table 3.4 gives an overview on how the resolutions of the different ITS products will be mapped to normalized resolutions. The columns show the normalized resolutions. The rows display the 3 sample ITS products. The cells of the table display which of the normalized resolutions the found resolutions are assigned to.

For both Jira and Bugzilla it is a trivial task to normalize resolutions into the normalized resolutions "Fixed", "Duplicate" and "Will-Not-Fix". All issues whose resolution indicates that they were rejected are mapped into the category "Invalid". Github.com ITS does not

	Fixed	Duplicate	Invalid	Will-Not-Fix	No Resolution
Bugzilla	FIXED	DUPLICATE	INVALID, WORKS-FORME, INCOM-PLETE	WONTFIX	
Jira	Fixed	Duplicate	Invalid, Incomplete, Works as Designed, Cannot Reproduce	Won't fix	
Github					All Issues.

Table 3.4.: Naive Mapping of Issue Resolutions

provide resolutions. Because of that all Github.com issues do not have a resolution stored in the database.

3.3.4. Normalization of States

The state models of the different ITS products are very divers. While Github.com only knows opened, closed and reopened issues, a Bugzilla issue can be in various states. To enable basic analyses we defined the following 4 normalized states:

- Opened: The issue was opened. No work was performed yet.
- Active: Work is currently performed on the issue.
- Closed: The issue has been resolved. Within this thesis we use resolved and closed as synonyms.
- Reopened: The issue was resolved but opened again.

Table 3.5 gives an overview of how the concrete states of the different ITS products are mapped into normalized states. The columns show the normalized states. The rows display the 3 sample ITS products. The cells of the table display which of the normalized states the found states are assigned to.

The mapping for Jira is trivial: The normalized states are nearly equivalent to the states found at Jira ITS instances.

For Bugzilla it is a questionable point whether the state "READY" should be mapped into the normalized state "Active" or "Opened". No developer work is performed at this point

	Opened	**Active**	**Closed**	**Reopened**
Bugzilla	UNCONFIR-MED, NEW	READY, ASSIGNED	RESOLVED, VERIFIED, CLOSED	REOPENED
Jira	Open	In Progress	Resolved, Closed	Reopened
Github	Opened		Closed	Reopened

Table 3.5.: Naive Mapping of Issue States

which is an argument for mapping it into "Opened". However transitioning an issue into the state "READY" requires the project team to evaluate the readiness. The following step that needs to be performed is resolving or assigning the issue; the issue was already extracted from the stack of freshly opened issues. Because of that we assume an issue in state "READY" to be mapped best to the normalized state "active".

3.3.5. Limitations of the Naive Mapping Approach

For the Github.com ITS the naive approach described in the last sections is not capable of finding normalizations for most of the relevant attributes (Resolutions and Priorities; no "active" state). Also it is not capable of extracting type information regarding Bugzilla issues. For future iterations we suggest the following strategies to overcome these limitations:

1. Issue types for Bugzilla and Github.com issues may be identified by a textual analysis of issue titles and descriptions as in Antoniol et al. [2]. The time frame of this thesis project did not allow us to implement such a rich analysis satisfying the performance needs of Ohloh.net with its more than 500,000 projects.

2. Resolutions and Priorities could also be extracted from assigned labels: A manual analysis of labels could be performed to identify labels that indicate certain priorities or resolutions.

3. Github.com only knows closed, opened and reopened issues. Issues that satisfy our definition of being active cannot be marked. Future data mining could focus on identifying certain activities in an issue's lifetime that denote a transition from the Opened to the Active state. Examples could be the first comments by team members or simply assignment of a user.

The mappings - especially for priorities - are subjective and cannot be validated easily. Over the course of a projects' lifetime specific usage patterns can emerge that are clear to

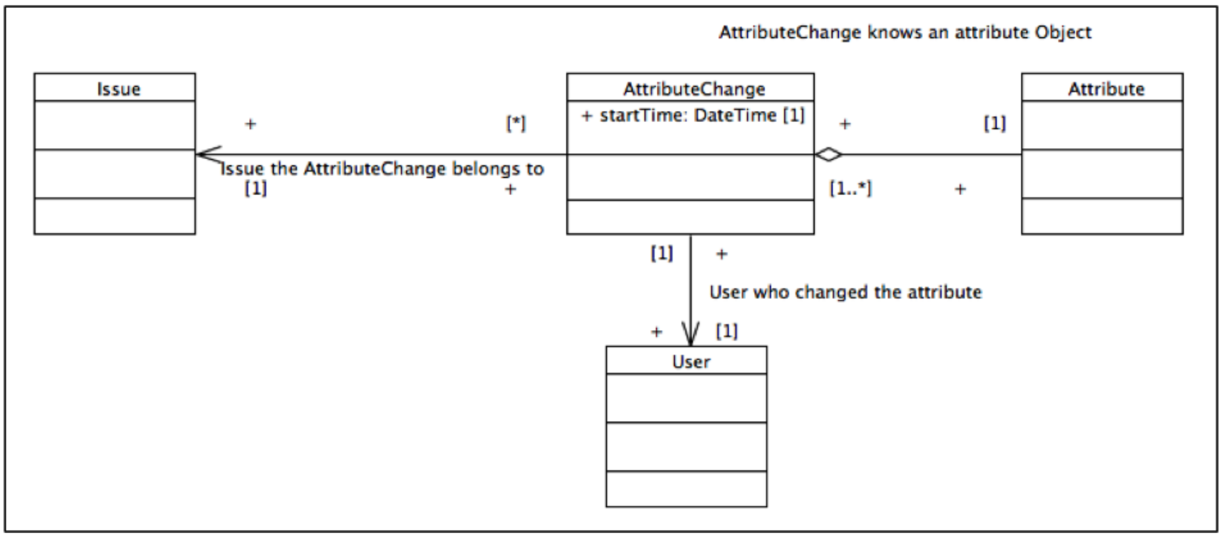

AttributeChange knows an attribute Object
Issue
AttributeChange
+ startTime: DateTime [1]
Attribute
+
[*]
Issue the AttributeChange belongs to
[1]
+
+
[1]
[1..*]
+
[1]
+
User who changed the attribute
+
[1]
User

Figure 3.1.: UML Class Diagram showing Temporal Changes

project members but not easy to understand from the outside. Also the defined mappings do not properly work, if the project teams explicitly define new resolutions, types, priorities or states. To decrease the risk of falsifying analyses we track both the original and the normalized values. A researcher performing analyses based on the models data must be made aware of the fact, that the normalization can interfere with the quality of the data and must take this into account when performing a validation of the specific claims worked on.

3.4. Storing a Chronicle of Changes

An issue that is stored by an issue tracker is not immutable. The values many attributes can be changed. A uniform data model must enable its users to store these changes.

We implement the temporal dimension of the model using Fowler's "Temporal Property" pattern [19] and a terminology similar to the one suggested by Snodgrass [46]. Figure 3.1 shows a generic example of our implementation.

The figure shows, that for those attribute we want to track historical changes, the issue is not related to its attributes directly. Instead, for each pair of attribute and issue a set of "Change" objects exists. These "Change" objects each carry the time a change became effective (startTime) as well a link to the user who performed the change. The set of change objects regarding a specific issue, allows determining the effective values for each attribute at each point in time. For all attributes the "Change" objects are of the same structure.

An exemplary set of objects of the type MilestoneChange regarding one issue is given in table 3.6. The milestone of the issue was changed 3 times. At the creation of the issue, the milestone with the ID 42 was assigned to the issue. The initial value of the issue has a

issueId	startDate	storedBy	milestoneId
10023	null	23023	42
10023	2013-06-23 15:00:00	54839	44
10023	2013-06-24 17:30:00	54839	42

Table 3.6.: Instances of a MilestoneChange

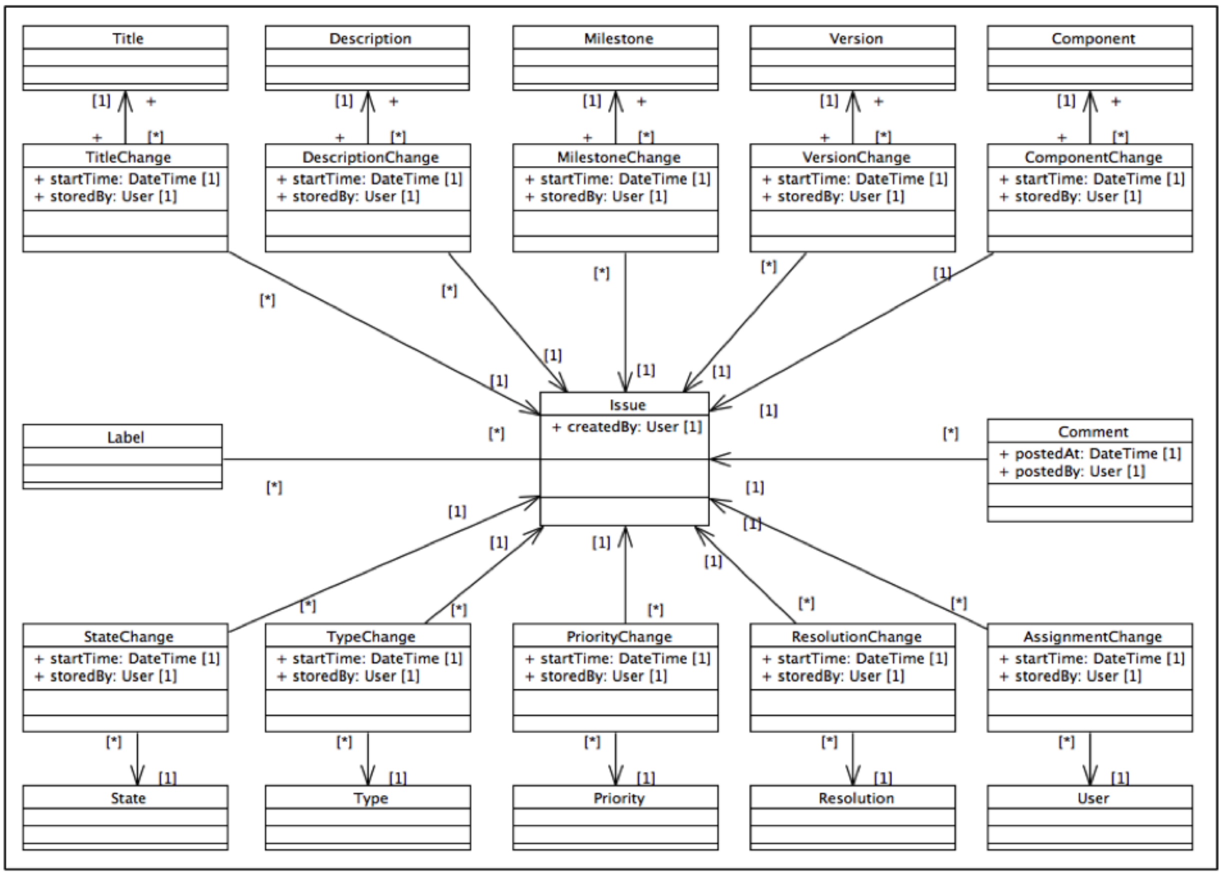


Figure 3.2.: UML Class Diagram giving an Overview of a Simplified Version Model

”startTime” set to ”null”. At June 23, 2012 the user with the ID 54839 changed the milestone of the issue to the milestone with ID 44. At June 24, 2013 the user with the ID 54839 noticed he made a mistake and assigned the issue again to the milestone with the ID 42. The other attributes and their ”Change” objects are shown in figure 3.2 and follow the same mechanism.

3.5. Overview on the Uniform Data Model

The last sections described the thoughts that guided the design of the uniform integrated data model for issue tracker data. This section will give a brief summary describing the model. Figure 3.2 gives an overview of the implemented model.

The UML Class Diagram shows, that for Labels and Comments no historical data changes are tracked. All other attributes (Description, Title, Milestone, Version, State, Component, Type, Priority, Resolution) are expressed using the ”Change” objects as described in the

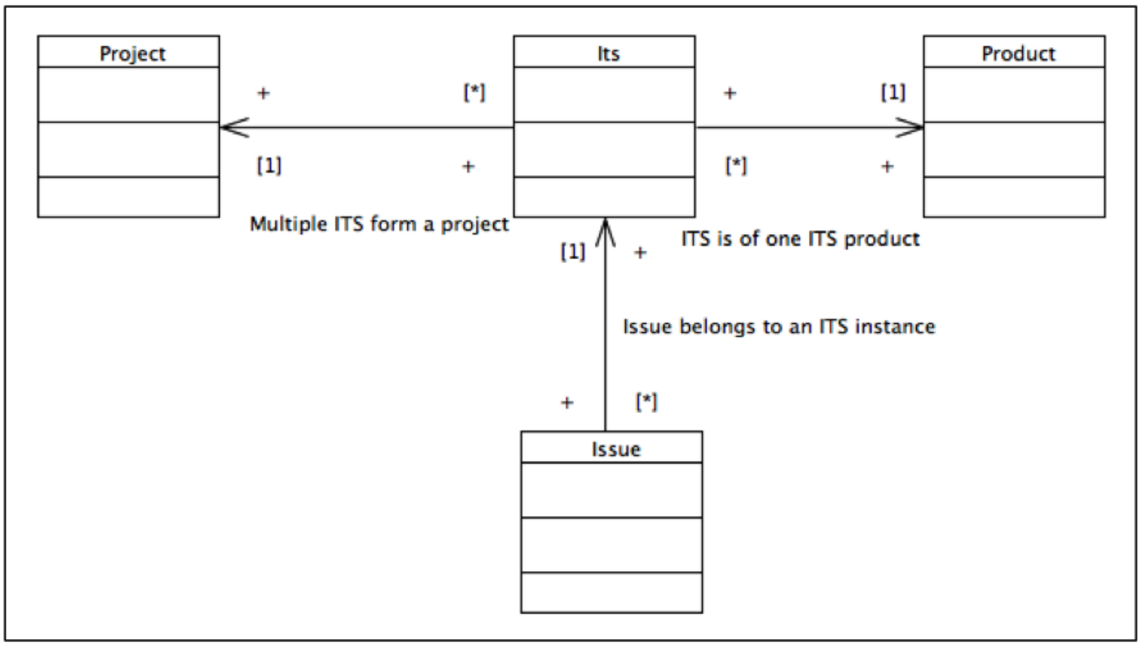

Project
Its
Product
Issue
+
[*]
[1]
+
Multiple ITS form a project
+
[1]
[*]
+
ITS is of one ITS product
[1]
+
Issue belongs to an ITS instance
+
[*]

Figure 3.3.: UML Class Diagram describing the Context of an Issue

previous section 3.4. The normalized attributes described in section 3.3 are omitted in this overview to keep the diagram comprehensible.

The attributes are segregated from the issue class; each attribute has its own group of classes namely the attribute class and the regarding change class. This gives the model a very modular structure, which enables to add new attributes or delete data about attributes with minimized implementation effort.

The issues are associated with an ITS object representing an ITS instance. Figure 3.3 visualizes that each ITS instance is of a product type. Multiple ITS instances belong to one project.

In this chapter we presented the attributes used by various ITS products. We explained how to address the diversity in attributes and their values by normalization using a naive mapping approach. Based on these outcomes we developed an Integrated Uniform Data Model for ITS data. We utilized Fowler's [19] "Temporal Attribute" pattern to express the changes of issue's attributes over time.

The model is capable of expressing a super set of the data for each of the selected ITS products (not taking into account the omitted attributes). We assume that the model is also capable of storing data regarding many other ITS products due to its modular structure which was described in the last sections. The next section will describe the toolkit we implemented for populating and managing ITS data based on the presented model.

4. Toolkit for Managing ITS Data

4.1. Architectural Overview

4.1.1. Functional Overview

The last chapter introduced a data model which is capable of representing the data of diverse ITS. This chapter will give an overview on a developed software toolkit for managing, manipulating and using the data stored in that data model.

The toolkit is designed to fulfill a variety of functionalities for gathering data from ITS instances and reasoning about this data. The following 5 functionalities are the most important capabilities of the software toolkit:

1. Populating of the Model: The toolkit is capable of fetching issues stored in an ITS instance, and saving them into a local database which is formatted according to the model described in chapter 3.

2. Incremental Update for Changes: The toolkit periodically checks for changes in the issues stored by an ITS instance. It is capable of updating the data incrementally without loading all issues again.

3. Normalizing of Values: Besides storing the issues in the defined format, the toolkit normalizes values for certain attributes as described in section 3.3.

4. Analyzing of the Data: The toolkit performs metric calculations on the stored data and saves the results into relational database.

5. Publishing of the analyses results: The toolkit enables the users to consume the results of the analyses with a JSON API.

Some of these functionalities can be mapped to the concepts of Measures, Metrics and Indicators described in section 2. The populating of the model, the incrementally updating of the data as well as the normalizing are equivalents of performing a measure. Analyzing the data is the equivalent of the metric calculation. Publishing of the data via a Webservice allows users of the toolkit to compare the results with indicators they want to monitor.

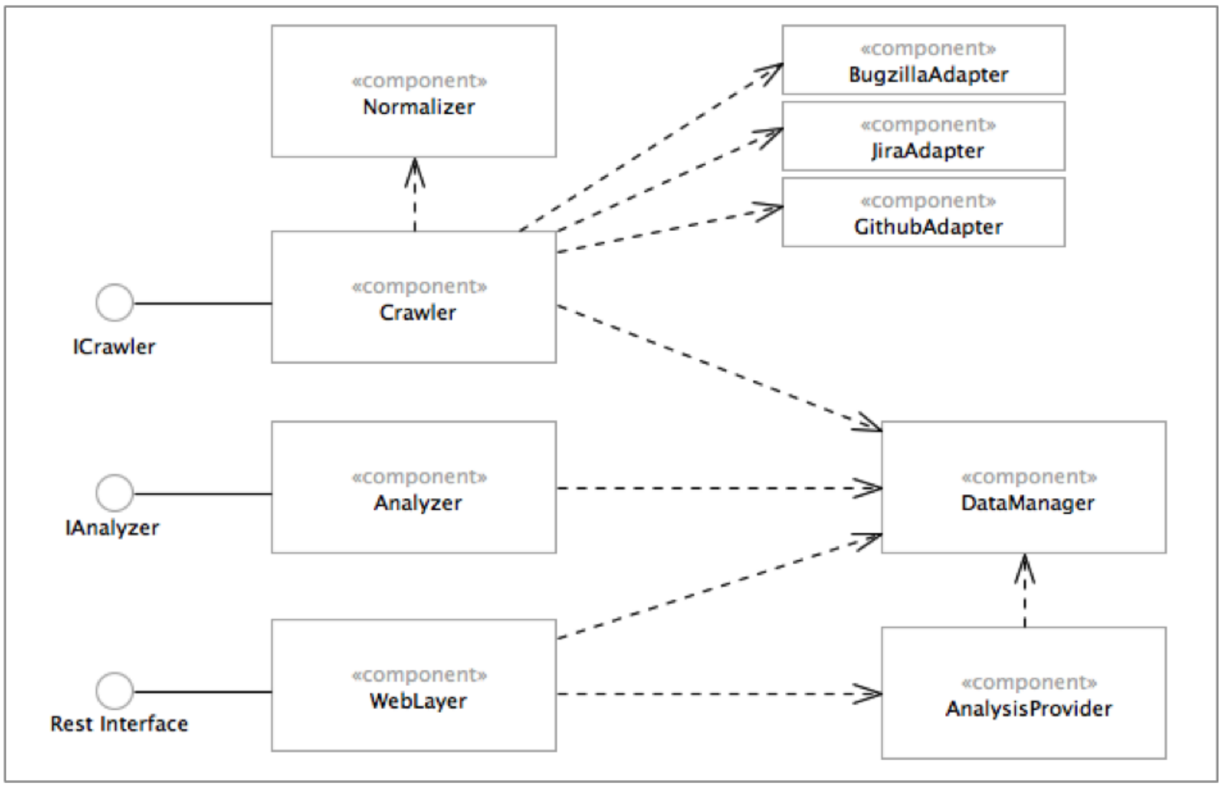


Figure 4.1.: UML Component Diagramm showing an Overview of the Crawler & Analyzer Toolkit

4.1.2. Structural Overview

Figure 4.1 shows a UML diagram that illustrates the components the toolkit is composed of. The components Crawler, Analyzer and WebLayer constitute the frontend of the toolkit. These components expose interfaces that can be used by the users of the toolkit: The Crawler and Analyzer components expose a Java interface while the WebLayer can be accessed using a JSON web service.

The diagram shows how the Crawler component orchestrates a set of ITS adapters for fetching the issues from the ITS instances. It utilizes the Normalizer component to normalize the values of issue states, types, resolutions and priorities. It depends on the DataManager component for storing the fetched issues into a relational database. The Crawler and Adapter components will be described further in the following section 4.2.

The Analyzer component precomputes the metric calculations and stores them into the database utilizing the DataManager component. It will be described further in section 4.3.

The WebLayer component publishes the metric results as a JSON API. It depends on the DataManager and the AnalysisProvider component. The AnalysisProvider component aggregates metric results for multiple ITS instances that belong to one Open Source project. The WebLayer component will be described briefly in section 4.4.

4.2. Measure - The Crawler Functionality

4.2.1. Adapters to the diverse ITS products

The adapters are important components of the Crawler and Analyzer toolkit. For each supported ITS product an adapter component needs to be implemented. Each adapter implements two functionalities:

1. An adapter implementation is capable of reading the data of the regarding ITS products. For example it is capable of using the API of an ITS product, accessing an ITS product's data store or utilizes screen scraping mechanisms.
2. An adapter understands the syntax and semantics of the ITS product's data. It is capable of transforming the data from the preexisting format into the presented integrated data model.

The functionality of the ITS adapters is accessed by the simple interface IAdapter. The interface is given in the following listing:

```
public interface IAdapter {
        public void setIts(Its _its);

        public Iterator<Issue>  fetchIssuesByModificationInterval(
                Date _startDate,
                Date _endDate
        );

        public Issue fetchIssue(Integer _id);
}
```

The method *fetchIssue* fetches one issue and is intended purely for testing. The method *setIts* expects a pointer to the ITS the data shall be gathered from at the moment.

The method *fetchIssuesByModificationInterval* is the most important method of the interface. It returns an iterator of all issues that were modified between the start and the end date. Using the Java element Iterator, instead of an ArrayList or a similar data structure, enables programmers implementing adapters to use lazy loading mechanisms. The results need to be sorted chronologically by the date of the last modification.

The next subsection will describe how the toolkit manages the known ITS instances to ensure to deliver pointers to the ITS.

4.2.2. Managing the Known ITS instances

One of the main goals of the implemented toolkit is the capability to copy and incrementally update information stored in various ITS instances. The huge set of ITS instances needs to be managed properly: To perform the crawling correctly, we not only need to track data on where to find and how to address the ITS instance, but also meta data on when and how the various ITS instances were parsed last time.

For the ITS instances stored in our database we track the following data:

- A relation to the project the ITS instance belongs to
- Information about the ITS product and its version
- Data on where to find the issue tracker and how to access it (url of the ITS instance, used credentials)
- Information about past errors that occurred while crawling
- The time $t_{last-crawl}$ which indicates when the ITS instance was last crawled
- The time $t_{last-modification}$. All modifications before $t_{last-modification}$ are assumed to be already crawled
- The time $t_{last-analysis}$ expressing when the last analysis was performed

For brevity only the most important data stored for each ITS instance is listed here. The left out data is assumed to be not relevant for our considerations.

The 3 time values $t_{last-crawl}$, $t_{last-modification}$ and $t_{last-analysis}$ are crucial for the crawling and the incremental update of issues. This process will be discussed in more detail in the next section.

4.2.3. Processes while Crawling

The last section described which information and meta data is tracked regarding the ITS instances known to the system. The 3 time values $t_{last-crawl}$, $t_{last-modification}$ and $t_{last-analysis}$ were introduced. These time values form the vector T:

$$T = (t_{last-crawl}, t_{last-modification}, t_{last-analysis}); \tag{4.1}$$

Currently Ohloh.net's analyses are mainly based on data from CSM. While parsing data from CMS we can always utilize some kind of version information for loading and performing incremental updates on artifacts. Issues are usually not versioned and not immutable. We

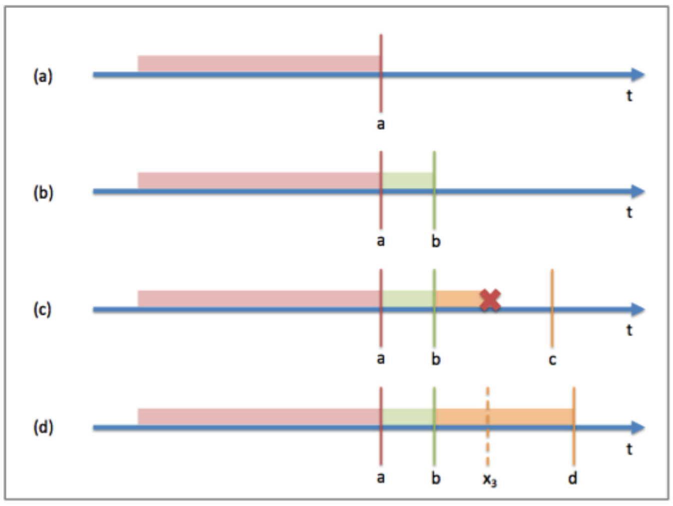


Figure 4.2.: Visualization of the Crawling process. (a) The Initial Crawl (b) The Incremental Update (c) An Error Forced to Stop the Crawl Run (d) Resumption after an Error

therefore need to utilize the data of vector T, especially $t_{last-modification}$ and $t_{last-crawl}$, for determining whether the data stored in our database is identical to the data in the ITS instance.

Using the approach we present in this section we can parse and incrementally update issues from all issue trackers that fulfill the following two constraints:

- The ITS product allows us to identify one specific issue using a unique identifier.
- The ITS product publishes the point in time each issue was last modified.

It is assumed that these constraints are fulfilled for a majority of ITS products.

Within the next paragraphs we will describe the approach using an issue tracker Q. This ITS instance was totally unknown to our toolkit - no issues have been crawled yet.

Initial Crawling

For an initial and not crawled ITS instance the vector T is given with $(null, null, null)$. The point in time we start crawling is point a. Figure 4.2 row (a) illustrates the initial crawling. The initial crawling follows the following approach:

1. Vector $T = (null, null, null)$
2. Choose the ITS adapter based on the ITS product of Q
3. Call adapter's method fetchIssuesByModificationInterval($null$, a). The first parameter is given by $t_{last-modification}$ which equals $null$. The adapter will interpret the parameter $null$ as the beginning of time

4. We iterate over issues:
    a) Current issue I is stored into the database by the DataManager
    b) Vector $T = (null, t_*, null)$. The time t_* denotes the last modification of I
5. $T = (a, a, null)$;

As the last action step (5) sets the $t_{last-crawl}$ to be time point a. It also sets $t_{last-modification}$ to a because it is save to assume that all modifications before a are known to the system now.

Incremental Update

At time point a we performed an initial crawl. The database is now filled with all issues that were known up to this point. At later points in time new issues can be reported. Also, as issues are not immutable, it is possible that some of the issues that are known to the system have changed.

Because of that an incremental update of the known issues can be performed. Figure 4.2 row (b) illustrates such an incremental update at a time point b. The incremental update is performed as follows:

1. Vector $T = (a, a, null)$
2. Choose the ITS adapter based on the ITS product of Q
3. Call adapter's method fetchIssuesByModificationInterval(a, b). The first parameter is given by $t_{last-modification}$ which equals a.
4. We again iterate over the issues:
    a) Current issue I is stored or updated by the DataManager
    b) Vector $T = (null, t_*, null)$. The time t_* denotes the last modification of I
5. $T = (b, b, null)$;

The data manager is capable of determining whether an issue is already known and needs to be updated or is completely new to the system and needs to be inserted (See step (4a)). For this determination the data manager uses the unique identified of the issue. The algorithm itself is identical to the algorithm of the initial crawl presented in an earlier paragraph. This incremental updating can be repeated frequently - for the currently implemented demo application we perform crawl runs once a day.

The crawler toolkit has to deal with a lot of remote systems - every ITS instance is hosted on another server in the Internet. This denotes a potential for faults and errors while crawling the ITS instances. The next section will describe how the system reacts to such error situations.

4.2.4. Error Handling

Self Recovery after an Error

The toolkit is designed to recover the state of the data automatically after an error is occurred. The following listing describes the system's reaction to a not further described error. Figure 4.2 row (c) and (d) illustrate the sequence further.

At time point c a new crawl run is started. The algorithm is performed as known from the last sections:

1. Vector $T = (b, b, null)$;
2. Choose the ITS adapter based on the ITS product of Q
3. Call Adapter's method fetchIssuesByModificationInterval(b,c).
4. ...

The issue iterator contains a tuple of issues with the modification times $t_{last-modification}$ given by $x = (x_1, ...x_n)$ with $x_1 <= x_2 <= ... <= x_n$. While iterating through the issues the vector T is subsequently changed from $(b, x_1, null)$ to $(b, x_2, null)$ to $(b, x_3, null)$ et cetera.

Lets assume the following scenario: After the issue with $t_{last-modification} = x_3$ was inserted to the database an error occurred and forced the toolkit to stop the crawl run. The vector T remains as $(b, x_3, null)$. It is possible that multiple issues have $t_{last-modification} = x_3$. Because of that we can only safely assume that we stored all issues with $t_{last-modification} < x_3$. (Keep in mind, that the iterator is ordered by $t_{last-modification}$.) Figure 4.2 row (c) illustrates this circumstance.

At point in time d the crawl run is resumed. The resumption can be performed seamlessly:

1. Vector $T = (b, x_3, null)$
2. Choose the ITS adapter based on the ITS product of Q
3. Call Adapter's method fetchIssuesByModificationInterval(x_3,d).
4. ...

On condition that no second error forces the toolkit to interrupt the new issues and all updates until point d can be parsed. No data loss or loss of data integrity occurred. Figure 4.2 row (d) illustrates the final state.

Exception Handling

Not always does it make sense to directly resume the crawling after an error occurred. In the last section we assumed that it is safe to resume the crawling after $d - c$ time units. For some errors it may even be the case that we cannot resume crawling until a human performed manual work on the error.

We identified the following most frequent causes of errors during the crawling:

1. Temporarily Network Faults lead to timeouts and errors
2. API Limits leads to refusal of requests
3. The huge amount of real world data is very divers. Unforeseen data formats can lead to problems while storing or transforming the data
4. Locations (or for forges project names) changed

Three exceptions that are part of the adapters' interface help to distinguish among errors that demand different reaction from the toolkit. The adapters are encouraged to use these exceptions; the toolkit is implemented to react properly to them. The following 3 exceptions are defined:

- *AdapterWarning*: This exception is used if an error was detected that is assumed to not falsify the data. The data integrity is kept and it is safe to resume crawling as described in this section. No wait time is needed before he resumption.
- *AdapterFailure*: An AdapterFailure exception is thrown when an error was detected that affects all ITS using the same product or at least all issues on the current ITS instance. After a wait time it is safe to resume the crawling. The data integrity is not endangered.
- *AdapterException*: This exception is thrown when an unknown error occurs. The component catching this exception should log the error properly. It is not safe to resume the crawling for this ITS. A human should investigate the cause of the error further.

A network or timeout error as in error (1) could cause an AdapterWarning to be thrown. An AdapterFailure can be thrown when the API limit is hit as in error (2). Error (3) and (4) both require human interaction and are good examples for errors that cause an AdapterException to be thrown.

This section discussed how the data regarding a huge set of issues from various ITS instances can be efficiently loaded into the database. The next section will introduce to an analysis mechanism that allows to make maximized use of the data.

4.3. Metric Calculation - Pre-Computing the Analyses

4.3.1. Motivation behind Pre-Computing

Last section's Crawler component enables the user of the toolkit to crawl the various ITS instances - measures are performed.

A portion of metric calculations that need to be performed using the data are very simple and can be done on the fly. Other metrics are calculated by more complex algorithms, need more calculation time and cannot be calculated in real time. Some of the visualizations and metrics we will describe in chapter 6 - for example the lag time overview or the collaboration network - even have quadratic or exponential complexities.

We will pre-compute these metrics. This allows us to deliver the results to the users fast. Also pre-computing the metrics will help us to not waste calculation resources by doing the same work multiple times.

Pre-computed analyses help to logically separate the concepts of measure and metric in the software toolkit. After an analysis is performed, new crawl runs can be done without worrying about versioning. As long as the crawler is running and no new analysis is performed, the stored analyses will remain in their old valid state. After the crawling is finished a new analysis can be started - the user will never face invalid data.

The next sections will describe the Analyzer component, which performs the analyses of the stored data. We will describe how the analyses are performed.

4.3.2. Storing Query Results as Analyses

The analyses that need to be performed are made known to the Analyzer component in the form of SQL queries. These queries are executed on the database and directly inserted into the regarding table for the analysis. The data is not transported through the JDBC layers and processed within the Java application - the analyses are solely performed by the underlying database.

For performing the analyses the toolkit first selects the set of ITS instances that need to be analyzed. Looking at vector T from the last section it selects all ITS instances with $T = (t_{last-crawl}, t_{last-modification}, t_{last-analysis})$; and $t_{last-analysis} < t_{last-modification}$. The toolkit then iterates over the resulting ITS instances. For each ITS instance the set of analysis queries is performed.

The are two major types of analysis queries:

- Analyses for a flat metric or a set of connected flat metrics: The analysis returns one single row.

- Analyses for monthly chronicles: The analysis returns multiple rows. One row for each month.

Examples for analyses that result in flat numbers are the number of open issues or the total number of issues known to an issue tracker. The metrics for monthly chronicles can have different gestalts: It can be a chronicle about the monthly changes of a metric or more specific things like rankings of users within the project. Also analyses of monthly dispersions of values will be performed using a monthly analysis. Within the currently implemented demo application we perform one analysis run per day for all ITS instances with changes.

4.3.3. Faking Transactions

As discussed some of these analysis require a long calculation time. This creates predicament:

1. Every Database transaction binds resources and locks parts of the data. We do not want transactions that have a runtime of a minute or more.
2. We want the analyses to be transaction safe: Whenever a user requests to see analysis data the toolkit must return valid data.

The toolkit does not use native database transactions for keeping the data integrity. Application logic of the toolkit manages the data integrity on its own. We use the following approach:

Every analysis result holds an is_valid flag that denotes whether this result can be shown to users (true) or not (false). Whenever an analysis is performed and the results are stored they are stored to be not valid. After the analysis is performed the toolkit starts a native database transaction. Within this transaction it deletes everything that currently has the is_valid flag to be true. After that, all analysis results regarding the ITS will be set to is_valid equals true. The transaction then will be committed.

This small transaction is running faster than a long transaction as described in point (1). The analysis results delivered to the user are always valid as demanded by point (2).

4.4. Publishing the Metric Results

The metric results will be published using a web service. The data format of the web service will be based on the JavaScript Object Notation (JSON). JSON values are a common format for the data exchange in web services and enable us to use the data within our client JavaScript application without performing massive transformations. Data expressed in the JSON format carries fewer overheads than XML data [54]. A full documentation of the JSON format is given at JSON.org [52].

A JSON key value pair will simply express the results of the flat metrics. The results of the monthly analysis will be expressed by a multidimensional JSON array. The following listing presents the data format on the example of the issues that were opened per month:

```
{"opened":
        [
                [1298937600000,51],
                [1301616000000,108],
                [1304208000000,1080],
                ...
                [1383264000000,223]
        ]
}
```

The array for the values consists out of pairs of two integers. While the second number denotes the measured metric - in this case the number of issues opened in this month; The first values is a epoch time that indicates the month the value was measured in. The data can be parsed with a JavaScript parser directly. The API can be consumed by multiple target applications. This API decouples the toolkit and potential consumers of the analysis data.

The last chapter introduced the toolkit for crawling and analyzing the data. We presented how the data is be fetched from the ITS instances and how incremental updates of the data are performed. It was described how the system autonomously recovers errors and keeps the data integrity. It was briefly described how analyses are performed and how the results are published to the users using a JSON based web service.

We discussed the performed engineering work in the last chapters; the next part of the thesis will describe the features we developed based on the crawled data. These features need to be useful for the users of Ohloh.net. Because of that, we identified a set of key information needs that will be described in the next chapter.

Part III.

Development of Features

5. Key Information Needs

5.1. Different Stakeholders with diverse Needs

As described in section 2.3 it is important to first identify information needs (or abstract goals) before defining specific metrics. Because of that, this chapter will describe key information needs regarding the data and processes within ITS instances. The next chapter 6 will introduce to the features and analytics we developed for Ohloh.net based on these information needs. The following paragraphs will describe how different audiences have different information needs.

An ITS is a tool that is broadly used within Software Engineering projects. A lot of Software Engineers are assumed to be familiar with it. Many stakeholders in diverse environments and contexts have interests in understanding the data. The amount of publications (see section 1.3) regarding the utilizing, understanding, predicting and enhancing the data within issue trackers proof this fact.

Every group of stakeholders to a project can be assumed to have different information needs. Also the information needs within Open Source and Closed Source contexts are assumed to be different.

We will not further discuss the stakeholders and information needs in Closed Source projects. The interested reader can find information about Closed Source projects' stakeholders in the SWEBOK [1]. For this thesis we estimate Open Source projects to be more interesting: Ohloh.net is a community for Open Source projects; the data we use for the analysis will be purely crawled from Open Source projects. It is a potential field for future work to investigate how the findings can be extrapolated to Closed Source projects. For Open Source projects we identified the following stakeholders:

- Contributors: Individuals contributing code to the project
- Committers: Individuals with the right to commit code
- Users: Individuals using the outcomes of the project

- Potentials Users: Individuals that have a requirement which could be satisfied by the outcomes of the project
- Potential Contributors: Programmers that are interested in engaging their selves in an Open Source project of a specific field / domain / structure
- Various Commercial stakeholders with different interests

It is highly interesting to identify the information needs for all of the identified 6 stakeholder groups. However as this thesis project needs to be performed within a limited time frame we will reduce this stakeholder groups to two basic groups:

- Individuals that are involved in the project (e.g. contributors and committers)
- Individuals that are using or planning to use an the outcomes of the project

We assume that most of the things users are interested in, are also interesting for people that are involved with the project team. Some of the information needs may be only relevant for people involved with the project. If an information need is assumed to be relevant for only a part of the stakeholders we will mention it in its description.

The next section will describe the approach we used for identifying the key information needs. It will also discuss the identified key information needs.

5.2. Extraction of Key Information Needs

5.2.1. Description of the Used Method

For identifying the key information needs we used the following approach:

1. We investigated literature about ITS in general and also measurements (mainly process) metrics for ITS and Software Engineering in general
2. We extracted questions that these papers discuss or aim to answer
3. We iterated over these questions and binned them into abstract groups.

These abstract groups are identical with the extracted information needs.

It is important to mention that this approach cannot replace a complete literature survey. Much more we just focused on specific papers that we found to be of interest. The huge body of publications regarding ITS and metrics within ITS did not allow us to take all papers into account. By ignoring publications we introduced a potential bias: It is possible that

	Question	**Source**
Q1	Which are the most discussed and relevant topics within the ITS?	[35]
Q2	How comprehensible are issue reports?	[33]
Q3	How is the quality of the issue reports?	[6] [27]
Q4	Who are the experts within an ITS instance?	[41] [4]
Q5	Are there duplicate issues within the ITS instance?	[29]
Q6	How long did it take to close past issues?	[31]
Q7	How long will it take to resolve future issues?	[53] [38]
Q8	Which are the features the project team worked on?	[16]
Q9	When did the project teams worked on specific features?	[16]
Q10	What is the quality of the OSS projects outcome?	[22] [24] [56]
Q11	What is the topology of the communities within the ITS instance?	[48]
Q12	Why did the software change?	[36]
Q13	How maintainable is the developed software?	[57]
Q14	Is the classification among bugs and other issue type correct?	[2]
Q15	How many issues are bugs?	[2]
Q16	How efficient are project teams in handling and closing issues?	[34]
Q17	How can we search for specific topics in the ITS instances?	[50]
Q18	Are there frequently repeated spikes in the activity within the issue tracker?	[25]
Q19	Does the release train mechanism affect the activity within ITS instances?	[20]

Table 5.1.: Questions Extracted from Literature

some information needs were not covered our investigations. Future research can focus on delivering further information needs by surveying more publications.

The next section will describe the questions extracted from the ITS publications.

5.2.2. Questions from Issue Tracker Literature

We iterated over a total of 22 publications regarding ITS and the usage of metrics and measures in ITS. The questions listed in table 5.1 were extracted from the publications. These questions were discussed by the publications listed in the publication column of the table.

We extracted 19 questions from the selected publications. Some questions were discussed by more than one of the selected publications: Question Q10 (quality of project outcomes) was discussed by three of the selected 22 publications. Publications about Open Source quality models [22], [47] discuss similar questions - we did not consider them for the extraction of questions because they cover a wide field of measures and metrics and do only focus partially on ITS. Questions Q3 (quality of issue reports), Q4 (experts within the ITS) and Q7 (prediction of time to resolve) are discussed in two of the 22 selected publications.

5.2.3. Extraction of Seven Plus One Information Needs

The 19 questions from the last section were identified to belong to 8 basic information needs. These information needs are presented in the next paragraphs. While some of the identified needs are very different, others (e.g. Collaboration and Key Individuals) are similar and differentiate only in fine nuances.

Activity

Based on question Q18 (frequently repeated spikes) and Q19 (effects of release planning on ITS activity) we assume that individuals are interested in developing an understanding about the general activity within an ITS instances. Very basic statistics that indicate a "heart beat" of the project can enable the user to get a first overview on the community behind the ITS.

Collaboration

Question Q11 (topologies within the community) expresses the need of understanding how the community is structured, to which degree individuals collaborate with each other and who the key social hubs in the community are. Understanding social collaboration and social communities is not only interesting in the context of ITS.

Efficiency

The questions Q6 (analysis of time to resolve), Q7 (prediction of time to resolve) and Q16 (closing efficiency) denote the need to estimate and predict the efficiency of the project teams. Analytics regarding this information should present how the project team deals with its workload and how it reacts to changing activity.

Important Topics

The questions Q1 (most discussed topics), Q8 (identification of features), Q9 (chronicle of features), Q12 (reasons for changes) and Q17 (search for specific topics) indicate an interest in knowing which topics are or were important within the issue tracker or in finding issues regarding already identified topics.

User Base

Question Q4 (experts within the ITS) expresses the need of understanding the user base. For example there is an information need to know which individuals play a key role within the issue tracker. These can be persons that perform a specific task very frequently or persons that are otherwise expected to have a specific expertise of play a key role for the project.

Quality of Issues

Questions Q2 (comprehensibility of issues), Q3 (quality of issues), Q5 (duplicate issues) and Q14 (correctness of issue types) indicate an interest in the quality of issue reports. It is rational that the project teams have an interest in having the issues reported in as high as possible quality. Knowing the quality of the stored issue reports is the first step to enhancing the quality and with that maximizing the productivity of the project team.

Quality of Software

The questions Q10 (quality of project outcomes), Q13 (maintainability of the software), Q15 (distinction of issue types) express an interest in the quality of software. We will discuss in section 6.4.2 that it is not trivial to utilize ITS data for the estimation of an Open Source software's quality.

Workload

No question was indicating an interest in how the workload of a project develops. The issues stored within an ITS instance give an overview of a part of the project team's workload. We assume that individuals are interested in getting a high level understanding of this workload. We therefore will also consider this information need - even though we did not find a regarding question within the set of 22 selected publications.

The last section described 8 key information needs, which we assume to be relevant for people involved in a project or depending on a project. The next section will describe how a validation of correctness and completeness of the identified information needs can be performed.

5.3. Validating the Key Information Needs

The last section presented the information needs we assume people involved in an Open Source project and people depending on the project have.

A careful validation of the identified information needs for correctness but also completeness should be performed. The following questions need to be answered by the validation:

1. Are individuals in the defined target groups really interested in getting a high level view on the described topics? (*correctness*)

2. Is the set of defined information needs complete? (*completeness*)

We identified specific artifacts that need to be validated on a larger scale: A surveying method [49] should be utilized to validate this information needs. However, from Black Duck Software's industry perspective it is not relevant to perform a tight and scientific validation of the information needs. Therefore we will not perform a survey in the scope of this thesis project. However we suggest future research based on the information needs to validate them properly.

This chapter introduced to a basic set of information needs regarding ITS data. The next chapter will introduce to a set of metrics and visualizations that satisfy the identified information needs.

6. Features for Satisfying Information Needs

6.1. Overview of the Developed Features

The last section introduced to 8 key information needs. Each of them represent an artifact or attribute that is interesting for individuals - project members and other persons with an interest in an Open Source project - in regards to ITS.

Based on these information needs we developed to kind of features:

1. Rich Visualizations that satisfy one or more information needs by presenting multidimensional data (mainly the development of specific measures or metrics over time)
2. Flat Metrics that satisfy mainly one information need and present it in a one-dimensional way enabling an individual to easily compare specific ITS instances / projects

The next section will introduce to the implemented multidimensional visualizations. The flat and one-dimensional metrics will be described in the later section 6.3.

6.2. Multidimensional Visualizations

6.2.1. Overview of Developed Visualizations

We developed a set of 8 graphs and charts that visualize divers aspects of the issue tracker data. The reference implementations of the charts were performed using HTML5 and JavaScript. For visualizing charts we used the HighCharts toolkit. The Collaboration Overview is created using Scalable Vector Graphics (SVG) and the D3.js toolkit.

Table 6.1 visualizes which information needs are satisfied by or connected to each of the developed visualizations. The need of the stakeholders to learn about the project's software quality cannot be satisfied completely by the visualizations we deliver - section 6.4.2 will discuss this issue further. Also the information needs about important topics within the ITS instance and the quality of issue reports cannot be tackled by the presented visualizations. These information needs will be addressed by follow-up projects.

Information Need	Chart
Activity	Activity Chart, User Activity Chart
Collaboration	Collaboration Overview
Efficiency	Efficiency Chart, Backlog Chart, Activity Chart
Important Topics	
User Base	User Activity Chart, User Rankings, Collaboration Overview
Quality of Issues	
Quality of Software	
Workload	Backlog Chart

Table 6.1.: Mapping of Information Needs and their regarding Charts

6.2.2. Activity Chart

The Activity Chart's main purpose is visualizing the activity within a project's ITS instances. An exemplary Activity Chart is given in Appendix B by figure 9.1. The red graph visualizes the amount of issues that were opened for each month, while the green graph visualizes the amount of opened issues. These are called Opening Trend and Closing Trend. While the y-Axis expresses the amount of issues, the x-Axes shows the time in months. The graph only displays the issues that were opened and closed within each regarding period of time - it does not accumulate all open or closed issues.

The over time presentation of opened and closed issues gives the user a quick overview of the activity within the ITS instance. A user looking at the graph can visually correlate the Opening and Closing Trends and derive specific findings. The presentation of Opening and Closing trend with the same y-Axis allows a basic estimation of the efficiency.

An individual knowing the specific project can connect its own context knowledge about the project to the findings in the graph. Specific minima and maxima within the chart may be connected to events in the project history known to users with specific knowledge about the project. .

The orange chart displays the amount of issues that were reopened within each time month. The amount of reopened issues - issues that were already closed and later opened again - is a representation of how precise the communication and information within the issues and comments is. Also it shows how well the specific solutions presented by the contributors fit the demands of the project team and user base.

6.2.3. User Activity Chart and User Rankings

The User Activity Chart displays the number of unique users within the project. An exemplary User Activity Chart is given in Appendix B by figure 9.2. The purple line indicates the number of active unique users in each month of the ITS lifetime. The light green and light red lines show the Closing and Opening Trends described in the last section. We display the

User Activity Chart together with the Closing and Opening Trend in order to enable the users to compare and visually normalize the charts.

Next to the graph we display an assortment of User Rankings. The User Rankings are illustrated by Figure 9.3 in Appendix B. Whenever the user uses the scroll bars of the User Activity Chart to change the considered time interval, the User Rankings are refreshed and show the most active ITS users for the new interval. Currently three User Rankings are implemented: Those of users that open and close the most issues, as well as those showing the users performing most comments.

While the User Activity chart satisfies the need for knowing and understanding the activity of the project, the User Rankings show key individuals within the ITS instance.

6.2.4. Lag Time Overview

The Lag Time Overview visualizes the dispersion of lag times within an ITS instance. Lag time is the time interval between the opening and closing of an issue. A box plot is used to visualize this dispersion of lag times. An exemplary Lag Time Overview is given in Appendix B by figure 9.4.

The y-Axis of the Lag Time Overview shows the Lag Time in hours. The box plot we utilized follows the following conventions: The upper and lower end of the box represents the lowest 25% (1st quartile) and the lowest 75% (3rd quartile) of the lag times. The line in the middle of the box displays the median of lag times. The antennas, so called whiskers, represent the lowest and highest 5% of the data.

To enable an individual looking at the chart to detect changes over time, we designed the Lag Time Overview to consist of a total of two box plots. The x-Axes show the months in the lifetime of the ITS instance: The first boxplot visualizes the dispersion of lag time for issues that were opened within a specific month, the second for issues that were closed in a specific month.

The dispersion of lag times within a project informs about the efficiency and the project team's capability to react to changes within the workload. Also it presents information that can be used to draw conclusions about the projects culture in regards to administrating the issues. It is suggested to display the Lag Time Overview together with visualizations satisfying the information need for activity: In this way users can visually correlate whether specific maxima and minima within the Lag Time Overview are connected to specific peaks within the project activity.

6.2.5. Efficiency Chart

The Efficiency Chart displays, how efficient the project team closes opened issues. By doing so it satisfies the identified need for information regarding a project team's efficiency. An exemplary Efficiency Chart is given by figure 9.5 in Appendix B. The y-Axis shows the amount and the x-Axis shows the time in months.

The efficiency chart shows the area between the Opening and Closing Trend. If the Closing Trend is higher, the area is colored green. If the Opening Trend is higher than the Closing Trend, the area is colored red. As a result, months with low efficiency are marked red and months with high efficiency are marked green. The bigger an area is, the more drastic is the positive or negative efficiency.

6.2.6. Backlog Chart

The Backlog Chart visualizes the work that still needs to be performed by the project team. It satisfies the information need for the team's workload. A stable or decreasing backlog is also a sign of an efficient issue resolution process. Figure 9.6 in Appendix B gives an example of a Backlog Chart. The y-Axis of the chart shows the amount of issues while the x-Axis shows time in months.

The blue line expresses the development of the backlog: For each month the number of currently open issues is displayed. (This is a different metric than the amount of issues that have been opened within a month.) The increase or decrease in the backlog is given by the gray bars. The gray bars visualize the net backlog increase, which is the first level derivate of the blue line indicating the workload after every month.

The chart also contains the Opening and Closing Trends in light red and right green. The Closing trend is negated. These two trends allow individuals looking at the graph to correlate areas of high activity with specific maxima and minima within the Backlog Chart.

6.2.7. Collaboration Overview

The Collaboration Overview visualizes the collaborations among users in the ITS instance. It satisfies the identified information need to gather a deeper understanding about the collaboration within an ITS instance.

Our first naive approach was to display the collaboration with a network graph: Each node in the graph expresses one user. Each edge between two nodes expresses that the users connected by the node collaborated with each other. Collaboration is defined as changing the state or commenting on the same issue. Each of the edges within the graph has an annotated weight. Based on the weights we planned to position each node within the view port using the

Fruchterman-Reingold algorithm [21]. This algorithm is a so-called force-directed algorithm. Each edge is seen as a force that pulls the two nodes together. The sum of the forces acting on the nodes determines their position on the display.

Such a visualization of graph networks is very intuitive. For the collaboration between users in ITS instances we cannot apply it: The most important users collaborate with such a huge amount of other users that the edges are very dense and do not allow. For the most active users the amount of edges can usually approximated with $e = n(n-1)/2$ edges e for n nodes (Also known as the handshake formula). This results in a very dense graph that does not allow the user to derive much information.

We therefore display the collaboration within an adjacency matrix. The same list of users denotes the rows and columns of this matrix. Depending on how frequently two users collaborate with each other we colored the cell of these to users in strong or lighter colors. Figure 9.7 in Appendix B illustrates an example Collaboration Overview based on an adjacency matrix. As the users are ordered by their importance within the project, it is not surprising to sea the opacity of the colors decreasing when moving to the bottom and to the right.

Future iterations of the project will utilize social network clustering algorithms for identifying groups with heavy collaborations within the user base. These groups can be marked in different colors in the adjacency matrix.

6.3. Flat Metrics for Project Comparison

Besides the rich visualizations presented in the last sections also a set of flat metrics needs to be displayed. This metrics allow the users to perform a quick comparison of projects regarding the identified information needs.

Table 6.2 gives an overview on the simple and one-dimensional metrics we implemented to satisfy the needs for information about the activity, efficiency, user base and workload of the project. Collaboration, Quality of Issues, Quality of Software and Important Topics within the ITS instance are not expresses by any of these metrics. Follow-Up projects will enhance this list of one-dimensional flat metrics. Some of the metric candidates can be seen in the list of ITS metrics, which is given in Appendix A.

Information Need	Chart
Activity	Number of Issues Opened in the last 7 days, Number of Issues Opened in the last 30 days, Number of Issues Opened in the last 90 days, Number of Issues Closed in the last 7 days, Number of Issues Closed in the last 30 days, Number of Issues Closed in the last 90 days
Collaboration	
Efficiency	Number of Open Issues Older than 30 days, Number of Open Issues Older than 90 days, Number of Open Issues Older than 1 year, Number of Open Issues Older than 3 years
Important Topics	
User Base	Number of Users (NOU), Number of Users Active in the Last 30 days, Number of Users Active in the Last 90 days
Quality of Issues	
Quality of Software	
Workload	Number of Open Issues (NOOI)

Table 6.2.: Flat Metrics for Project Comparison

6.4. No Visualizations for Software Quality

6.4.1. Definitions of Software Quality

The last sections described a variety of visualizations and one-dimensional metrics based on ITS data that satisfy various information needs. We were not able to implement a visualization or metric that presents valid information about the quality of the software that is delivered by the regarding project.

The CISQ Quality model defines quality to be composed of the efficiency, maintainability, reliability, security, and size of a software system [51]. None of these quality attributes can be directly extracted from ITS data.

Kitchenham et al. [32] the technical side of software quality. According to Kitchenham et al. Quality can be seen from 5 perspectives:

- Transcendental Perspective: Software Quality is something that cannot be defined but recognized by informed individuals
- User Perspective: Software Quality is the fitness for use from the users' perspective
- Manufacturing Perspective: Software Quality is the conformance to norms, standards and the specification
- Product Perspective: The Quality of a Software Product is determined by the characteristics of the software components, code and artifacts.

- Value Based View: Software Quality is based on the value it creates.

Only the User Perspective, maybe also the Transcendental Perspective, could be estimated based on findings within ITS data. The previously discussed metrics and visualizations can support individuals in gathering a transcendental view on the quality aspects of each project.

The next section will discuss why we estimate this data to be a poor choice for measuring and understanding the software quality of a project.

6.4.2. ITS Data's Validity for Estimating Software Quality

Most authors believe that data from ITS is valid for assessing a software project's software quality [22], [24]. For example Yu et al. [56] suggest that the number of issues with the type "bug" are a direct indicator for the software quality. In the next paragraphs we will discuss, why ITS data is not fit for estimating a software project's software quality at all. The data is influenced by to many factors for estimating software quality with a high precision. (See also Raja & Tretter [40])

Looking at a Closed Source environment a company may have approximate data about the amount of people using their software. Estimations can be based on the volume of sold licenses for client applications or on the amount of registered users for Internet applications. The amount of users can be used to normalize the amount of opened issues or bugs.

But, this normalized amount of opened issues over time does not express the quality of the software itself. Because every organization and individual may have slightly different requirements to a software system, this metric is not valid for estimating software's quality. The only thing that can be expressed with such an approach is the satisfaction of the users (regarding their specific definition of quality attributes for a specific software artifact). It is still possible to argue that the satisfaction of users may be a good indirect measurement for the software quality.

For Open Source software however we are not aware of any studies that allow an estimation of the magnitude of the user base. One option would be to utilize the amount of users within mailing groups or just within the ITS instances. These users could be filtered in order to not consider users under a certain threshold of activity. A broad range of factors on the other hand, influences the amount of opened issues within a certain time interval.

We do not aim to completely solve this dilemma. However the data analysis that will be presented within the next chapters, will also discusses possible relations between the activity within the ITS and other factors.

6.5. Validation of Metrics and Visualizations

For the same reasons as discussed in section 5.3 we do not perform a tight validation of the presented metrics and visualizations.

In theory, we need to validate the metrics regarding some of the discussed quality attributes from section 2.4.1.

Objectivity and Repeatability are ensured by the automated implementation of measuring and the metric calculation. The Comparability of the metrics is partially given: The metrics visualizations use the same scale and allow naive comparison of various projects. However these scales do not adapt to project specifics. The user still needs a portion of context knowledge to compare projects based on the visualizations and metrics. The metrics are economically according to our situation if they are bringing a benefit to the users and entertain and inform them while using Ohloh.net.

Therefore a validation needs to be focused on the quality attributes Validity and Usefulness. While the general usefulness can be shown by validating the key information needs, it needs to be validated separately whether the visualizations deliver the results we expect.

As discussed in section 2.4 a qualitative validation approach should be used. We suggest using an experimental setup where individuals are asked to compare and rate projects based on the visualizations. Their rating is matched to what we expected the graphs to show.

Part IV.

Research Part

7. Exploring the Currently Parsed Projects

7.1. Current Approach for Loading Projects

At the time of writing this thesis, the Adapters for the ITS products Bugzilla and Jira were not implemented completely. The currently discussed data set only consists from Projects hosted at Github.com. Projects with an Ohloh.net activity level index bigger than 30 were excluded from the selection.

This activity level index is determined by a heuristic. It takes the monthly contributors as well as the commits into account. Older activities are decayed exponentially. An activity score of 30 or more denotes a project with moderate or higher activity.

This extraction mechanism resulted in 13228 projects with their regarding ITS instances. These ITS instances store a total of 1.5M Issues and 4.2M Comments. In average 46 users are involved per ITS instance.

In the next sections we will demonstrate that considering only Github.com projects minimizes the representativeness and diversity of the currently parsed data. The next section will give an explorative overview on the parsed data.

7.2. Comparative Exploration of the Projects

7.2.1. Age and Creation Time of the Projects

In this section we will explore the age and creation time of the projects in our sample data set in comparison to the Ohloh.net projects.

Figure 7.1 part (a) shows the amount of projects that were created within each month. The blue line shows all the Ohloh.net projects while the red line represents the projects from the sample. For better presentation Ohloh's projects were limited to these created after year 1990. As Github.com was founded in 2008 [8], the red graph does not show any project creations before 2008.

Figure 7.1 part (b) isolates the red graph and displays it with a linear y-Axis. A clear spike of project creations in year 2012 can be seen. After that the project creations decrease.

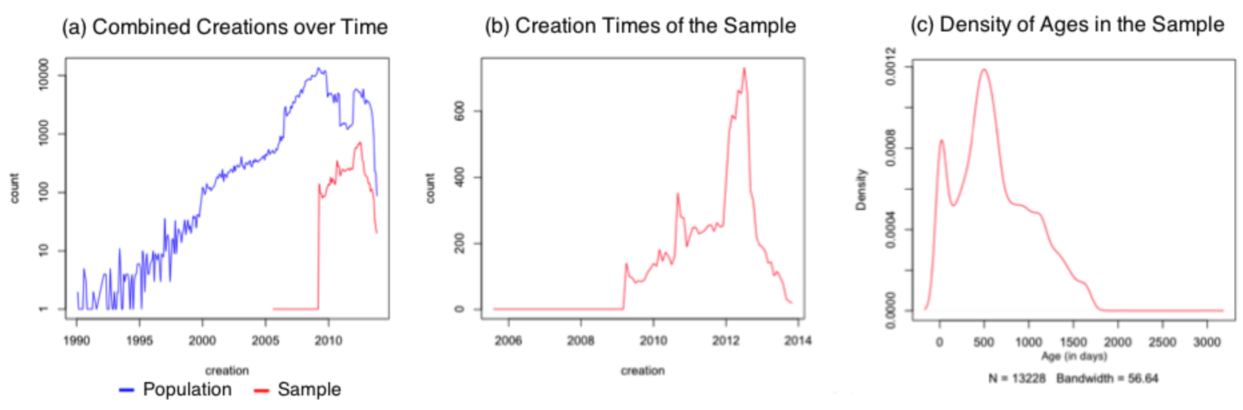


Figure 7.1.: Creation Times and Ages of Projects in the Population and the Sample

We assume changes in Ohloh.net's parsing algorithm to be responsible for the changes in the spike as well as the following decrease in creations. Also, in year 2012 an amount of 100M USD was invested into Github.com [55]. The forge was integrating more development tools. We assume Github.com's popularity increased due to these investments. This could explain smaller spikes or increases in project creations. We assume it to be not the reason of the spikes and the following decrease in year 2012.

Figure 7.1 part (c) shows the distribution of the project ages in a kernel density plot. The x-Axis - the project age - is given in days. As we filtered for projects above a certain activity threshold it is a trivial observation, that the creation times in part (b) are symmetric to the ages visualized in part (c) if the x-Axis is assumed to be the symmetry axis. We determined the age as the difference of the last minus the first action within the ITS instance. The peak at an age of 0 is caused by projects that never used their Github.com ITS instance or only used it once.

The data shows clearly, that the selected sample of Github.com projects is not representative for the population of all Ohloh.net projects in regards of project age and creation time.

7.2.2. Used Programming Languages

The programming languages used by all Ohloh.net projects slightly diverge from the subset of Github.com projects. Table 7.1 shows the top 10 programming languages in Ohloh.net and the subset. The second column shows the total amount of projects and the proportional amount in percent in regards to the subset or the population.

The table shows that the selected Github.com projects and all Ohloh.net project share nine of the top ten languages. Only Objective-C respective XAML cannot be found in both rankings. The last places in the ranking with less than 3% difference in usage are assumed

	Population		**Subset**	
	Language	Amount	Language	Amount
1	Java	61231 (12%)	JavaScript	2514 (19%)
2	JavaScript	60284 (12%)	Python	1708 (12%)
3	Python	36024 (7%)	Ruby	1612 (12%)
4	Ruby	31347 (7%)	Java	1396 (10%)
5	PHP	30289 (6%)	PHP	1265 (9%)
6	C#	27869 (5%)	C	943 (7%)
7	C++	27407 (5%)	C++	786 (5%)
8	C	25160 (5%)	Perl	529 (3%)
9	*XAML*	9311 (1%)	C#	379 (2%)
10	Perl	9302 (1%)	*Objective-C*	361 (2%)

Table 7.1.: Top 10 Languages in Ohloh.net and the Sample

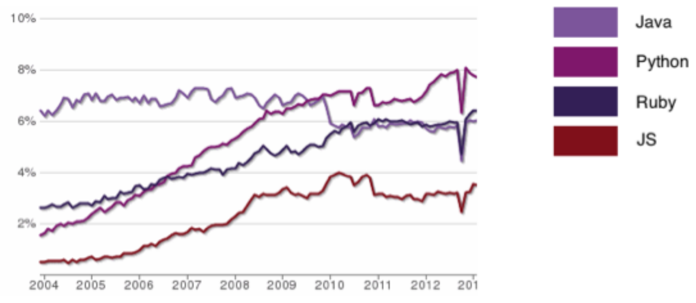


Figure 7.2.: Ratio of Projects with at least one Commit per Language over Time

to be not significant.

Significant is that 19% of the selected Github.com projects use JavaScript as the main language, while for all projects on Ohloh.net only 12% use JavaScript. Python and Ruby seem to be more prevalent within the Github.com projects.

Figure 7.2 shows the percentage of projects with at least one commit per language over time. While Java shows a stagnating or slightly decreasing amount of projects; The amount of projects that involve JavaScript, Python and Ruby increase.

Because of that, the significant differences in the usage of Java, JavaScript, Ruby and Python in sample and population can be explained by the fact that younger projects are overrepresented in our sample with only Github.com projects (projects created before 2008 are not existing).

7.2.3. Project Size in Regards of Issue Count and Users

Another way to characterize projects is by their size. In regards to ITS, we can measure the size of a project for example based on the amount of stored issues or users. Figure 7.3 gives a histogram describing the distribution of project sizes by number of issues (NOI); while figure 7.4 describes the distribution by number of users NOU).

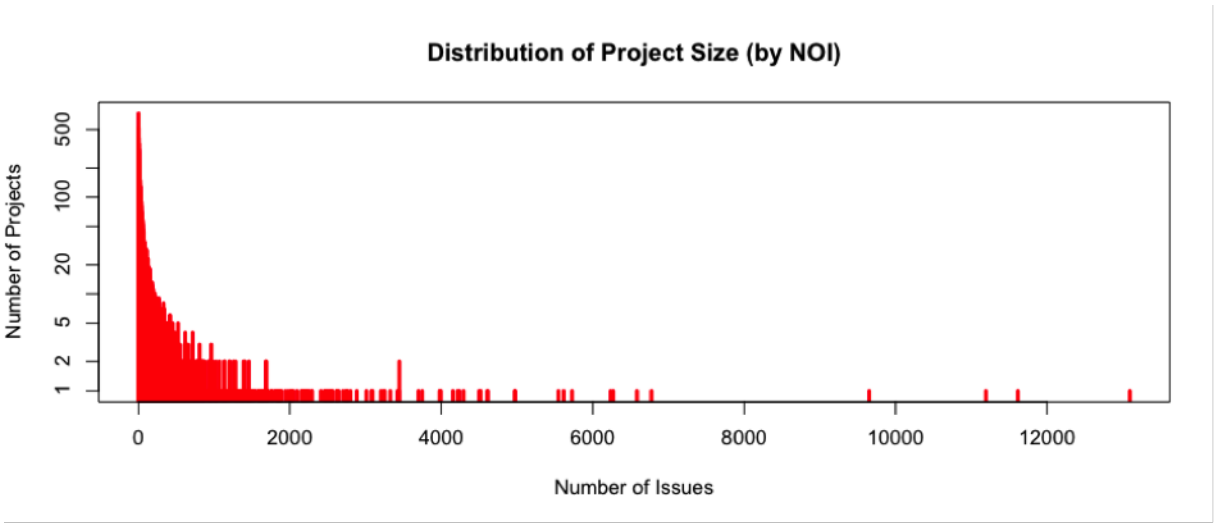


Figure 7.3.: Distribution of Project Size by Number of Issues

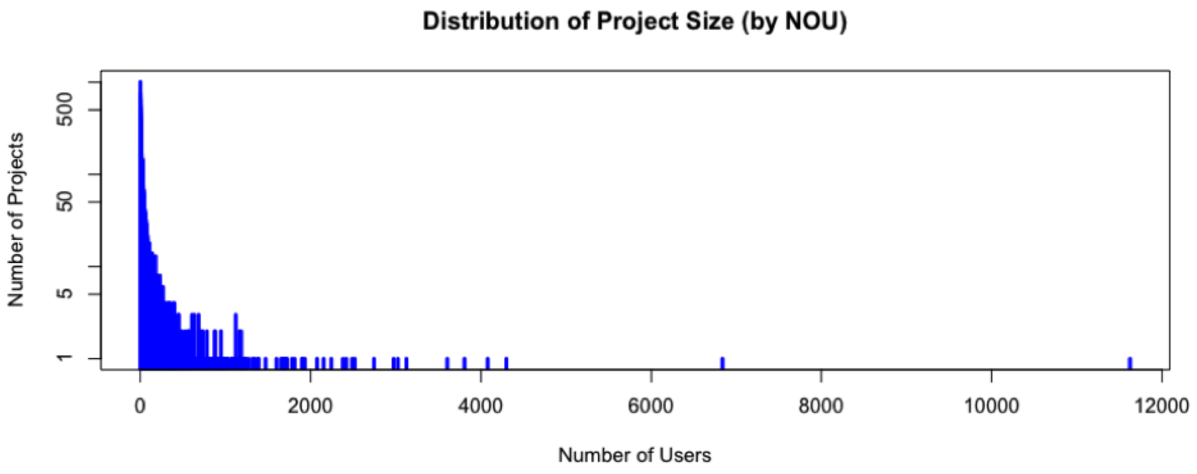


Figure 7.4.: Distribution of Project Size by Number of Users

Figure 7.3 shows that the majority of projects has less than 2000 issues. While 12105 projects have under 2000 issues; only 71 projects have between 2000 and 6000 issues, only 5 between 6000 and 10000 issues and only 3 more than 10000 issues - namely Twitter's Bootstrap, Ruby on Rails and the trinity core. (The lower boarders are including.)

Figure 7.4 shows a similar distribution based on the number of users (NOU): The majority of projects has less than 1000 users. While 12116 projects have less than 2000 users, only 15 projects have between 2000 and 6000 users. Only two projects have more than 6000 users: Ruby on Rails with 6840 and Twitter's Bootstrap with 11626 users.

Projects not hosted a Github.com may even have by far more than 10000 issues per project. While the Linux Kernel had around 22,000 issues in November 2013, the Mozilla Firefox project had 131865 and the Eclipse IDE 17231 issues. ITS instances of this size are not properly represented by the subset we selected based on Github.com projects.

7.3. Evaluation of the Sample's Validity

7.3.1. Concepts of Diversity and Representativeness

The last section explored the data available in the database. This section will evaluate the representativeness and diversity of the currently parsed data sample in regards to Ohloh.net.

Nagappan et. al define representativeness as follows:

> "In a representative sample the size of each subgroup in the sample is proportional to the size of that subgroup in the population. (Let us assume a population of 400 subjects of type X and 100 subjects of type Y,) a perfectly representative sample would be $4 * X$ and $1 * Y$" [37]

In accordance with Nagappan et al. diversity is defined as follows:

> "A diverse sample contains members of every subgroup in the population and within the sample the subgroups have roughly equal size. (Let us) assume a population of 400 subjects of type X and 100 subjects of type Y. In this case, a perfectly diverse sample would be $1 * X$ and $1 * Y$. " [37]

Some may argue that the condition on equally sized groups may be decreasing the representativeness of a sample. Per definition that is the truth. It however does not necessarily affect the validity of claims made based on the data.

For example when asking individuals about their political opinion before a governmental election we do not want to have equally sized groups. In such a case we want our groups to be of the same proportion in the sample and in the population - representative.

Within this thesis, we prepare a large-scale empirical analysis of data within an engineering context. For this goal it is important, to have a solid set of data for every identified subgroup of OSS projects. Only such a set will allow us to carefully analyze which implication from the data can be applied to which kind of OSS project. For sampling the projects - we want to achieve highest diversity. Also, Nagappan et al. [37] suggest aiming for diversity instead of representativeness.

7.3.2. Neither Diverse nor Representative Sample

For evaluating the diversity and representativeness of a sample it is important to identify the dimensions that characterize each object - in our case the OSS projects - and help to formulate the subgroups.

The selection of dimensions to characterize subgroups is not a trivial task. In the last sections we presented the three attributes project size, project age and the used programming language as possible dimensions. Also the total lines of code, number of contributors, recent activity, a classification of the project domain and other dimensions can be relevant [37]. The three dimensions discussed in detail were sufficient for demonstrating the representativeness and diversity of our sample in regards to the Ohloh.net population.

As we showed in section 7.2.1 the ITS instances in the sample are young. Older projects found on Ohloh.net are not represented. As shown in section 7.2.3 also the ITS instances are small compared to other ITS instances of projects on Ohloh.net. The selected OSS projects in the sample are neither representative nor to they cover the diversity of Ohloh.net.

The following hypotheses H0.1 is therefore proven to be correct:

> H0.1: A sample of active Github.com projects is not representative in regards to nor does it cover the diversity of the universe of OPen Source Software projects.

Primarily the last paragraphs showed that the representativeness and diversity does not cover the Ohloh.net population. Ohloh.net is delivering the biggest subset of OSS project data known to the author of this thesis; and researchers draw conclusions based on empirical studies that utilize the Ohloh.net data (e.g. [26], [37], [14]). It is assumed to be representative for the whole universe of OSS projects. Because of that, the selected sample projects also do not cover the diversity or are representative in regards to the universe of OSS projects.

The last sections explored the data. We showed why the data set is not representative regarding the universe of OSS projects. The next chapter will discuss some findings that were more specific. This chapter's discussion of the diversity and representativeness of the data will help the reader to evaluate the limits of the findings.

8. Findings in our the Parsed Data

8.1. Structure of the Community

8.1.1. Identification of User Groups

Within this chapter we will introduce to some of the findings within the currently parsed ITS data. Section 8.2 will discuss correlations between different Trends within the ITS and the Source Code Management Systems (SCM). The later section 8.3 will discuss the influence of release dates on the activity within the ITS. This section will analyze the structure of the user base.

We identified 3 types of users that are mainly participating within an ITS instance:

1. One-Time user: This individual only reports one issue in the whole project lifetime.
2. Project Team Member: This individual is a part of the project's team.
3. Community Member: This individual is not part of the project's team but actively uses the issue tracker.

While the identification of One-Time users (1) is fairly trivial, the identification of team members (2) is not: Due to privacy reasons data about the user base is the most difficult to crawl from the ITS instances. Most ITS instances do not explicitly mark team member. We therfor identify team members as those users that at least closed or assigned one issue or had at least one issue assigned to themselves.

The users that neither belong to group (2) nor to group (1) are treated as Community members. This is a possible thread to validity: It is arguable that also a user reporting two, three or n issues might not be an active user within the community.

We assume that a magnitude of the users can be found within the group of the One-Time users. Davies [12] described that this is the case for the Debian project. Further it is assumed that both group (1) and (2) grow with the user based. The group of project members is assumed to never sink below a certain ratio.

8.1.2. Presence of User Groups

Figure 8.1 part (a) shows the percentage of One-Time users in a scatter plot. Each project is represented by one point. The y-Axis shows the percentage of One-Time users within the project, the x-Axis shows the project size in NOU. Figure 8.1 part (c) shows a similar scatter plot for those users that were identified to be members of the project team.

Figure 8.1 part (b) and (d) show the distributions of the ratio of One-Time users respective team members within the different projects as a histogram. Projects with less than 250 users were excluded. The frequency is given by number of projects.

Figure 8.1 part (a) indicates that a magnitude of projects have between 40% and 60% of One-Time users in their user base. Most of the projects have less than 2000 users (This issue is discussed in section 7.2.1). Also the histogram in part (b) shows the highest frequency for projects with between 45% and 55% of One-Time users.

Our analyses of the data show, that for projects bigger than 250 users, 92.90% of the projects have more than 40% users that only report one issue. 91.1% of these projects have between 40% and 70% One-Time users. This proves the following hypothesis:

> H1.1: Within most of the ITS (91.1%) 40% to 70% of the users report only one issue.

This shows, that the project team members have to deal with a lot of users that are not familiar with the projects, the reporting or communication culture. This can be seen as one of the causes for the immense resources that need to be invested for triaging issues within ITS. ([30], [7])

On the other hand, this clearly shows that users that are not active within the projects, do use ITS instances of the regarding projects to express their needs and tribulations with the software. This numbers show ITS instances are tools that are not exclusively used by members of the project team to organize the bug fixing and feature discussion processes. Also external individuals use ITS instances as a tool to communicate with the developer base.

The scatter plot in figure 8.1 part (c) as well as the histogram in part (d) visualize that most of the projects have between 5% and 30% project team members within their user base. Only 5.4% of the projects lay outside this range. This proves the following hypothesis:

> H1.2: The community of most ITS (94.6% of projects) consistsof 5% to 30% team members.

No correlation of the ratio of One-Time users or team members with the size of the project has been detected. The user bases of the projects grow homogenously. Future research will

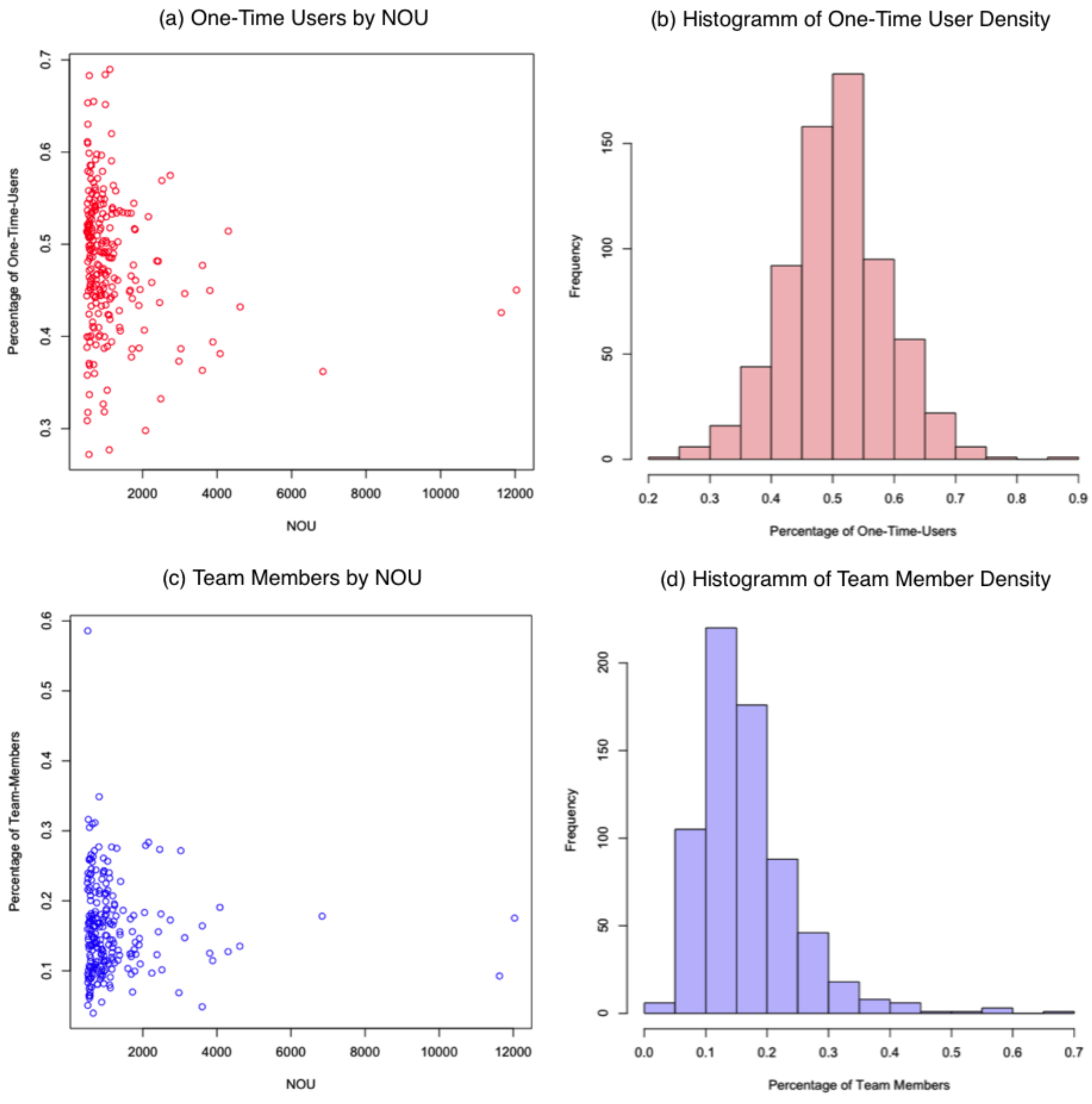


Figure 8.1.: (a) Scatter Plot showing the Ratio of One-Time users in regards to the Project size (b) Histogram of Ratios of One-Time users (c) Scatter Plot showing the Ratio of Team Members in regards to the Project Size (d) Histogram of the ratio of Team Members

have to show whether this homogenous growth is also the case for bigger projects that are not represented properly in our sample. Also we assume that for bigger projects the user group (3), which is very small for the presented projects, might be growing.

Generally, the user group (3) is interesting for future research. These users are neither team members nor One-Time users. We assume they can be utilized for empirical research about the demands and the growth of the user base.

8.2. Correlations among Time Series within the ITS

8.2.1. Suspected Correlations

As discussed in the previous chapters of this thesis ITS instances contain a rich variety of data about a project and its community. Also other tools in an OSS environment may contain manifold information. For example mailing list, wikis and the Source Code Management Systems (SCM). The data from the SCM is especially interesting because it allows to close the gap between the code and community based metrics.

In the following subsections we will show that the Closing Trend is correlated to the Commit Activity (Subsection 8.2.3). We will explain that the amount of active users correlates with the Issue Opening Trend (Subsection 8.2.4). We will introduce our findings about the correlations between Reopening Trend and Efficiency (Subsection 8.2.5). We will discuss why the not measurable correlation between Efficiency Trend and Reopening Trend can be an indication for well functioning triaging processes.

The next subsection will introduce the method used for calculating the correlations among the various time series.

8.2.2. Method for Finding Correlations

We see each time series as a function $f(t) = y$ that returns a value y for each defined point in time t. The points in time t have the precision of months.

The calculation of the correlation of two time series $f(t)$ and $g(t)$ is performed using the cross-correlation approach as described by Shumway and Stoffer [45]. We use a lag $\tau = 0$. The values are already aggregated by month. We do not assume lags significantly bigger than one month.

For the two time series $f(t)$ and $g(t)$ we consider each pair of outputs that returns valid and known data for both time series. If for a specific month t one or both time series do not have known values this month will not be considered for calculating the correlation.

We will perform the calculation for a set of preselected projects. We selected all 1413

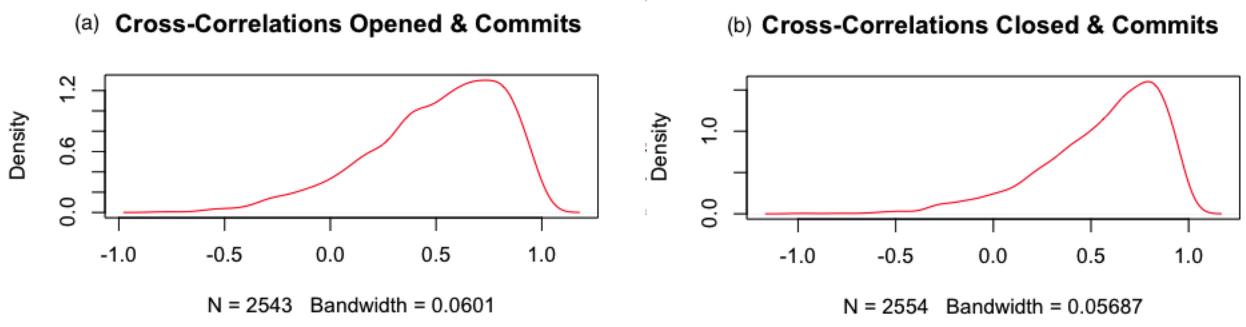


Figure 8.2.: (a) Density of Cross-Correlations Between Opening Trend & Commit Activity
(b) Density of Correlations between Closing Trend and Commit Activity

projects in our database with a size of more than 200 issues. The results are displayed for all projects using a density chart that visualizes the density of correlation results on a scale from -1 (negatively correlated) to 1 (positively correlated).

8.2.3. Correlation between Closing Trend and Commit Activity

The Opening and Closing Trends as well as the Commit Activity are important measures of the activity within an OSS project. Compared to the code churn the commit activity does not deliver information about the size but only about the amount of encapsulated work units performed. Also the Issue Opening and Closing Trends do not tell about the complexity or size of the regarding issues. The detected correlations imply that the Commit Activity has a similar expressiveness in regards to activity as the Opening and Closing Trends.

Graph 8.2 part (a) shows the density of calculated correlations between the Opening Trends and the Commit Activity for the selected projects. For most of the projects a significant correlation was determined. Graph 8.2 part (b) shows the density of correlations between Closing Trend and Commit Activity. The density for projects that show a correlation between 0.7 and 0.9 is higher than on the left hand graph. This leads to the conclusion that the correlation between the Closing Trend and the Commit Activity is more significant than the correlation between the Opening Trend and the Commit Activity. The following hypotheses is proven:

H2.1 The Closing Trend and the Commit Activity are significantly correlated.

The correlation between the Closing Trend and the Commit Activity expresses a causal relation: Whenever an issue is closed that is not invalid (for example a duplicate) and induces a change in the software, the developer resolving the issue needs to perform a commit.

The fact that such a significant correlation occurs for these two trends shows, that ITS instances are one important driver for the change and evolution within OSS projects.

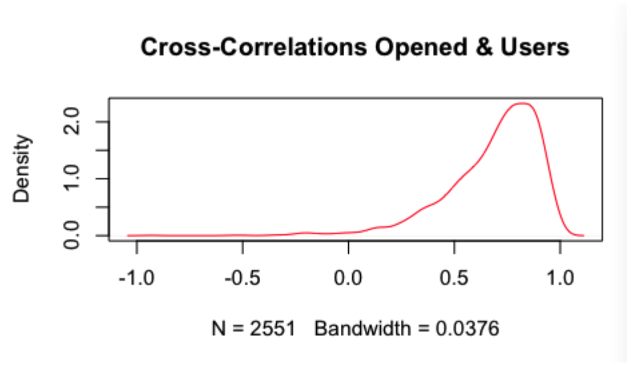


Figure 8.3.: Density of Cross-Correlations between Opening Trend and Active Users

The result may be biased by the fact that Github.com ITS implementation encourages reporters to deliver patches with their issue reports (so called "pull requests"). This property of the Github.com software forge may lead to a reduced lag time between the Closing Trend and the Commit Activity. This does not affect the validity of the results itself, but primarily the possibility to compare the results to those from projects that are not hosted on Github.com.

8.2.4. Correlation between Opening Trend and User Activity

In section 8.1 we discussed that a huge portion of users only reports one issue and never participates in other community activities within the ITS. This finding implied the question whether this group also significantly influences the Opening Trend.

Figure 8.3 shows the density of Cross-Correlation between Opening Trend and the amount of unique active users within the ITS instances. A clear spike in the density can be seen for correlation between 0.7 and 0.9. For most of the projects the amount of active users correlated with the Opening Trend. Nearly no projects show a negative or insignificantly positive correlation. The following hypothesis if proven:

H2.2 The Trend of Active Users correlates with the Issue Opening Trend.

The fact that the amount of uniquely active users per month correlates with the Opening Trend indicates, that the so-called One-time users (see section @) have a strong influence on the Opening trends. It shows that the One-time users are inducing a big portion of the Opening activity.

8.2.5. No Correlation between Efficiency and Reopening Trend

We also investigated the correlation between the Reopening Trend and the Efficiency. The Reopening Trend lists the amount of issues reopened within one month. The Efficiency trend

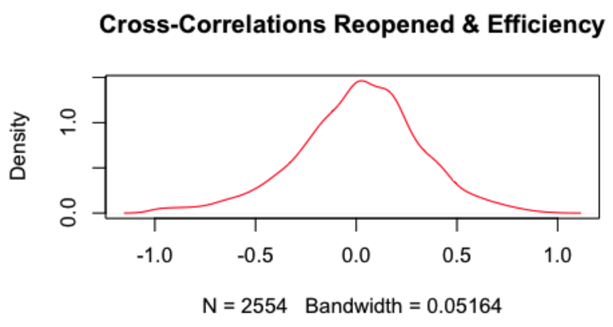


Figure 8.4.: Correlation between Reopening Trend and Efficiency

is given as the difference of the Closing Trend minus the Opening Trend.

Figure 8.4 displays the density of correlations for the sampled projects. The graph has the highest density for correlations between 0.0 and 0.2. For these projects no significant correlation was detected.

H2.3 The amount of Reopened Issues and the Efficiency Trend are not correlated.

Initially we were assuming a positive correlation: We thought projects might loose effectiveness of the triaging processes with increased efficiency. The data indicates the effectiveness of the triaging and resolution processes is not harmed by a more efficiency.

However this does not per se imply a high effectiveness of the triaging process for a majority of the projects: It is possible that a not considered time lag occurred or the mistakenly closed issue was reported again after the initial issue was closed. Also the correlations are falsified by the relatively small values for Reopening Trends compared to Closing or Opening Trends.

Within the last sections, we investigated a set of correlations between time series data gathered from ITS and SCM to further understand the behaviors of the OSS communities. It was shown how the user base influences the Opening Trends and how Opening and Closing trends correlation with the Commit Activity. The Closing of Issues was identified to be an important driver of the Commit Activity.

The next section will discuss the influence of Release Dates on the Activity within the ITS instances.

8.3. Influence of Releases Dates on ITS Activity

8.3.1. Release Dates and Activity Peaks

Release Dates are important within every software project's calendar. These dates are assumed to affect the activity within the issue tracker. For example Francalanci and Merlo [20]

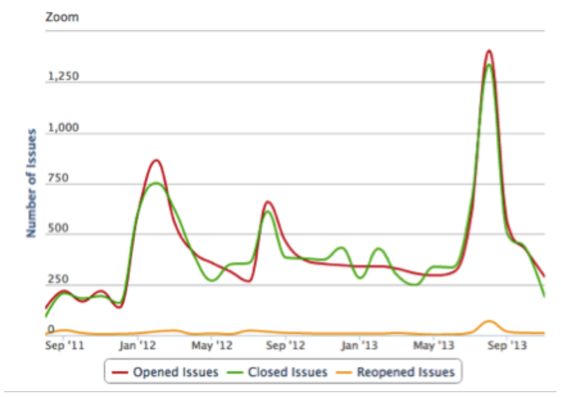


Figure 8.5.: Activity Chart of the Bootstrap Project

showed that release dates influence the issue fixing processes within 9 exemplary sampled OSS projects: Around release dates the most issues are resolved.

A case study of the Bootstrap project supports this claim: Figure 8.5 shows the Closing Trend of the Bootstrap project in green and the Opening Trend in red. In January and August 2012 as well as in August 2013 maxima in the trends can be seen. These maxima correlate with the releases of the Bootstrap versions 2.0.0 in January 2012, 2.1.0 in August 2012 and 3.0.0 in August 2013.

Despite the findings in the Bootstrap project and those of Francalanci and Merlo [20], we do not assume that release dates immediately induce a higher activity within the ITS instances. Release dates are assumed to be one out of many important factors that interfere with the activity of an OSS community within the ITS instance.

We will therefore show in the next sections, that release dates do not immediately or significantly induce maxima in the activity within an ITS instance. The next section will discuss the selection of 10 sample projects. Based on this sample the analysis will be performed.

8.3.2. Selection of Sample Projects

For validation of the claim that release dates do not significantly influence the activity within ITS we selected a sample of 10 projects. The projects were selected to be diverse regarding to the size of their ITS instance. We randomly selected two projects with 500 to 1000 issues, 1000 to 2000, 2000 to 4000, 4000 to 8000, and 8000 to 16000 issues. (The lower boundaries are including, the higher boundaries excluding). Table 8.1 shows the identified projects in each group.

For each project we identified all major release dates. We do not consider the release dates that are not within the ITS instances life span. The release dates were chronologically ordered and name $a, b, c....$

Group	Project	Language
500-100	Amber Smalltalk	A Smalltalk Implementation that runs on top of a JavaScript runtime.
	JBoss Tools	Software Tools regarding JBoss implementations.
1000-2000	Paperclip	A Library for file attachment management in Fowler's [18] Active Record pattern in Ruby.
	Akka	Framework for building concurrent and distributed aplications.
2000-4000	Devise	An authentication toolkit for Ruby on Rails.
	Font-Awesome	A font containing a vector-based icon set for web applications.
4000-8000	Gitlab	A project management and code hosting application.
	dotCMS	An enterprise content management system.
8000-16000	ownCloud	A client-server system for storing data within a self hosted cloud
	Ruby on Rails	A framework for ruby web applications.

Table 8.1.: Sample Projects for Studying the Influence of Release Dates on the ITS Activity

The next section will describe the method we applied for each of the projects to determine the correlation of activity and release dates.

8.3.3. Presentation of the Used Method

For each of the projects we derived the Closing and Opening trend with a precision of one month. For each month and each trend we investigated whether the month is a local minimum, maximum or has a increasing or decreasing elevation. We also determined a normalized rank r^* for each month of each trend.

This rank r^* is calculated by $r^* = (r - 1)/(m - 1)$. The symbol r represents the not normalized rank: The position from 1 to m the month has in a list that is ordered decreasing by amount of Opened respectively Closed issues. The symbol m denotes the number of months in the Opening or Closing trend. If for example a month q has the Closing Trend rank value of $r^* = 0$ it means, that the amount of Closed issues in month q is higher than in all other months. If the rank is $r^* = 1$ the month has the fewest closed issues. A rank $r^* = 0.5$ denotes the month with the median amount.

For each project we identified the release dates. For the month of the release dates we identified the normalized rank r^* and whether the month denotes an extreme value in the trends. The influence of a release date on the Closing or Opening Trend is said to be significant and immediate if the rank is within the first quartile ($r^* < 0.25$) and the month is a local maximum of the regarding trend.

Project	**Release**	**Date**	**Closing**		**Opening**	
			Char.	Rank	Char.	Rank
Amber Smalltalk	0.10	10.03.2013	MAX	0.275	MAX	0.172
	0.11	22.06.2013	MAX	0.413	MAX	0.379
	0.12	11.11.2013	MIN	0.724	MIN	0.655
JBoss Tools	4.0.x	16.02.2013	MAX	0.285	MAX	0.357
Paperclip	v3.0.0	26.03.2012	INC	0.2580	INC	0.032
Akka	release 2.0	01.03.2012	MIN	0.7666	MIN	0.766
Devise	v1.0.0	08.02.2012	MIN	0.250	MIN	0.1562
	v2.0.0	26.01.2013	INC	0.281	INC	0.406
	v3.0.0	14.07.2013	MAX	0.435	MAX	0.500
Font-Awesome	v3.0.0	01.01.2013	MAX	0.421	MAX	0.368
	v4.0.0	22.10.2013	MAX	0.526	MAX	1.000
Gitlab	v2.0.0	20.12.2011	DEC	0.120	DEC	0.080
	v3.0.0	21.10.2012	MAX	0.240	MAX	0.160
	v4.0.0	22.12.2012	MAX	0.440	MAX	0.240
	v5.0.0	19.03.2013	MIN	0.480	MIN	0.280
	v6.0.0	20.08.2013	INC	0.920	INC	0.960
dotCMS	2.0	17.04.2012	INC	0.750	INC	0.7
ownCloud	v5.0.0	11.03.2013	DEC	0.133	MAX	0.133
Ruby on Rails	v4.0.0	25.06.2013	INC	0.406	MIN	0.5

Table 8.2.: Charecteristics and Normalized Ranks of Release Dates within their regarding Closing and Opening Trends.

8.3.4. Results of the Analysis

For the 10 identified projects we found a total of 21 release dates within the life span of the ITS instance. Table 8.2 displays the normalized rank r^* and whether it is an extreme value for Closing and Opening trend of each release dates within each project.

Eight of the identified release dates correlated with maximum values in the Closing Trend. Five releases correlated with increasing values, four with minimum values and two with decreasing values. The Opening Trend shows the same distribution: Only 42.1% of the release dates do appear with local maxima in the Opening respectively Closing Trend.

For the Closing Trend 1 out of 8 maxima have a rank $r^* < 0.25$. The remainder of the maxima have a rank between 0.275 and 0.526. For the Opening Trend 3 out of 8 maxima have a rank $r^* < 0.25$. The remainder of the ranks for maxima are between 0.357 and 1.000.

The data shows that a the discussed naive theory about the influence of release dates is not correct:

> H3.1: Release Dates do not induce an immediate peak in the Opening or Closing Trend of an ITS instance.

The discussed algorithm was suitable for gathering enough proofs to validate this hypothesis. Due to its simplicity and superficiality it is not suitable for exploring the influence of release dates on ITS instances' activity any further. The next section will give brief ideas on future research methods.

8.3.5. Future Research on the Affect of Release Dates

The simple approach performed in the last subsections was sufficient for showing that release dates do not immediately elevate the activity within an ITS instance. For performing a more expressive analysis we suggest the following approach:

1. An explorative investigation of patterns within various ITS trends before, after and around release dates should be performed.
2. The identified patterns are codified as functions $f(t)$ where t is the regarding time step.
3. Validation of the patterns by calculating Cross-correlations for the trends of various ITS instances and the codified patterns or combinations of those patterns are calculated. Following variations are possible:
   a) Changing the time lag of the cross correlation
   b) Manipulating the expansion of the pattern by introducing a factor u in $f(t * u)$. Cross-correlations are performed for a variety of factors u.
4. Clustering of the results regarding identified attributes of the projects and the calculated cross-correlation values.

We believe that not only release dates but also other events affect the activity within ITS instances. These events need to be identified and patterns in the ITS Trends need to be extracted. The validity of these patterns could be validated with a similar approach.

Within the last chapter findings within the data were described. The structure of the user base was analyzed. It was shown that release dates do not immediately affect the activity within an ITS. Correlations between various time series were discussed. The next chapter will close the thesis. It will give a summary, describe suggestions for future research and discussed the threads to validity of our results.

9. Concluding Thoughts

9.1. Threats to Validity

External Validity The data set used for the analyses of correlations, the structure of community and the influence of release set does not cover the diversity of the OSS universe. The findings only have validity for projects with similar attributes.

Internal Validity The presented Key Information needs (Chapter 5) are not validated qualitatively by questioning individual people active within OSS communities. The assumed threat validity is small, as the Information Needs were mainly extracted from other publications.

It is currently not validated qualitatively whether the Visualizations (Chapter 6) are comprehensible for users and therefore satisfy their information needs. However due to the tight collaboration with User Experience designers at Black Duck Software we assume the charts to be comprehensible for users.

Validity of Statistical Conclusions Correlations between variables of interest (section 8.2) do not imply causal relation. Due to the size of the sample we feel confident in assuming other variables to be of low influence to the shown correlations.

Drawbacks in Engineering Work We decided to omit a set of attributes when implementing the model (Chapter 3). This potentially reduces the usefulness of the model. It is not assume to be a significant drawback as the model can be extended easily due to its modular structure.

9.2. Summary

Within this thesis we developed tools and methods for analyzing ITS data on a large scale, and utilized this tool to enhance our understanding about the role of ITS within OSS projects.

We delivered a definition of measures, metrics and indications. We presented approaches to classify metrics within the context of OSS projects and presented quality attributes. We described how metrics can be validated quantitatively and qualitatively.

We developed and implemented an integrated uniform model for data from diverse ITS products and instances. As issues are not immutable, the model is also capable of expressing the historical changes within the ITS data. A toolkit for populating and incrementally updating the model with data from real world ITS instances was developed.

Further, we extracted 8 Key Information Needs regarding ITS data from 22 publications. We implemented metrics and visualizations to satisfy a majority of these information needs.

We utilized the fetched data to perform various analyses. We showed that the user base within an ITS instance consists of team members, users that were only active once and users that are actively participating within the community. We showed that release dates do not have an immediate and significant effect on the activity within ITS. Specific trends within the ITS were identified to be significantly correlated.

9.3. Suggestions for Future Research

The defined data model and the tool for populating it with data, enable empirical research with ITS data in multiple directions. For the concrete work presented within this thesis we see the following possibilities for follow-up research and implementation work:

- Some of the identified data attributes of ITS products were not implemented in the model presented in chapter 3. Especially the possibility to express relations between issues and between issues and other tools (e.g. SCM) should be added to the model.
- We identified Key Information Needs (Chapter 5) of stakeholders within OSS communities. Further research could focus on extrapolating these information needs to closed source environments. Where are differences in the information needs of OSS and Closed Source projects?
- A more intensive validation of the Key Information Needs using the survey method should be performed.
- A more intensive validation of the visualizations (Chapter 6) using controlled experiments should be performed.
- Some of the Key Information Needs are currently not satisfied by metrics or visualizations. Further metrics and visualisations should be implemented.
- The adjacency matrix informing individuals about the collaboration within OSS projects' ITS instances coud be clustered using algorithms of cluster detection of social network analysis.

- The sample data set used in chapter 7 and 8 does not cover the diversity of the OSS universe. More projects from other sources should be added to the data base in order to repeat the studies with a more diverse dataset.
- The naive mapping approach for normalizing diverse values of different ITS products is limited. Section 3.3.2 gave ideas about other possible approaches that could be implemented in future iterations of the project.
- Further possibilities to explore the influence of release dates to ITS activities were presented in section 8.3.5

Appendix

Appendix A - List of Identified Metrics

User Centric Metrics

Acronym	**Name**	**Description**
NOU	Number of Users	
NOUbC	Number of Users by Characteristics	
NOUbT	Number of Users by Type	
NOUsIA	Number of User's Issues Assigned	
NOUsIAT	Number of User's Issues Assigned To	
NOUsIC	Number of User's Issues Created	
NOUsIR	Number of User's Issues Resolved	
NOUsIT	Number of User's Issue Transactions	
NOUsWI	Number of User's Worked on Issues	

Metrics regarding a Single Issue

Acronym	**Name**	**Description**
NOIsA	Number of Issues Assigments	
NOIsRC	Number of Issue's Resolution Changes	
NOIsRO	Number of Issue's Reopenings	
NOIsT	Number of Issue's Transactions	Every change of an issue is called a transaction.
NOIsU	Number of Issue's Users	
ILTI	Issue Life Time Information	
ILTIE	Issue Life Time Information Estimate	A presentation of the amount of issues for several lag times.
IA	Issue Age	A description of an issues age or the age of a group of issues.

Metrics regarding a Set of Issues

Acronym	Name	Description
NOB	Number of Bugs	
NOI	Number of Issues	
NOI-O	Number of Issues that were opened (in time frame)	
NOI-C	Number of Issues that were closed (in time frame)	
NOI-RO	Number of Issues that were reopened (in time frame)	
NOIbAV	Number of Issues by Affected Version	
NOIbAS	Number of Issues by Abstract State	
NOIbC	Number of Issues by Component	
NOIbL	Number of Issues by Label	
NOIbM	Number of Issues by Milestone	
NOIbP	Number of Issues by Priority	
NOIbR	Number of Issues by Resolution	
NOIbST	Number of Issues by Submitter Type	
NOIbAT	Number of Issues by Assignment Type	
NOIbT	Number of Issues by Type	

Efficiency Indicators

Acronym	Name	Description
BMI	Backlog Management Index	Similar to scrum's Backlog Management Index.
IURLT	Unresolved Issue Last Touched	The last times pot. old unresolved issues have been touched.
ITC	Issue Trend Continuity	Can be applied for various Trends. Can be calculated using skewness values of trends.
ITE	Issue Trend Efficiency	Can be applied for various Trends. Can be calculated using kurtosis values of trends.
NOIUR	Number of Unresolved Issues	
NOIURRESP	Number of Unresolved Issues with Response	
NOIURWRESP	Number of Unresolved Issues without Response	
NOIURWM	Number of Issues Unresolved without Milestone	

Process Quality Metrics

Acronym	Name	Description
NOIL	Number of Linked Issues	
NOIRO	Number of Reopened Issues	
NOIRWM	Number of Issues Resolved without Milestone	
NOIWV	Number of Issues without Version Information	
NOID	Number of Duplicated Issues	
NOIRNT	Number of Issues Resolved By Not Team Members	It is not clear whether this can be really gathered from the data. Might be possible for Github ITS.
ITriP	Issue Triage Precision	
ITriR	Issue Triage Recall	

Appendix B - Multidimensional Visualizations

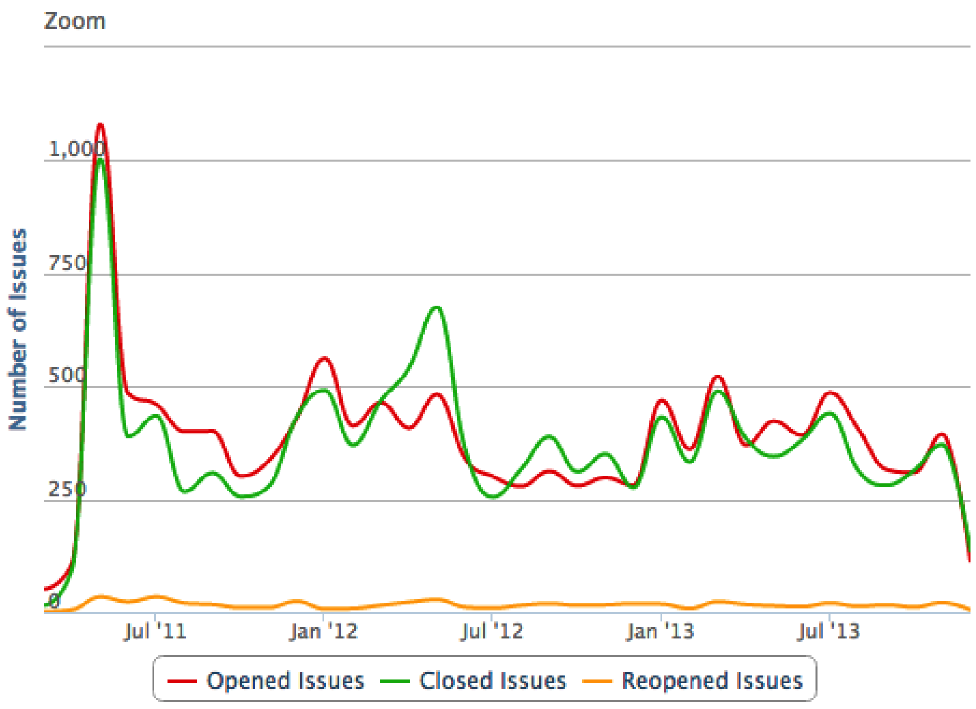


Figure 9.1.: Activity Chart of the Ruby on Rails project

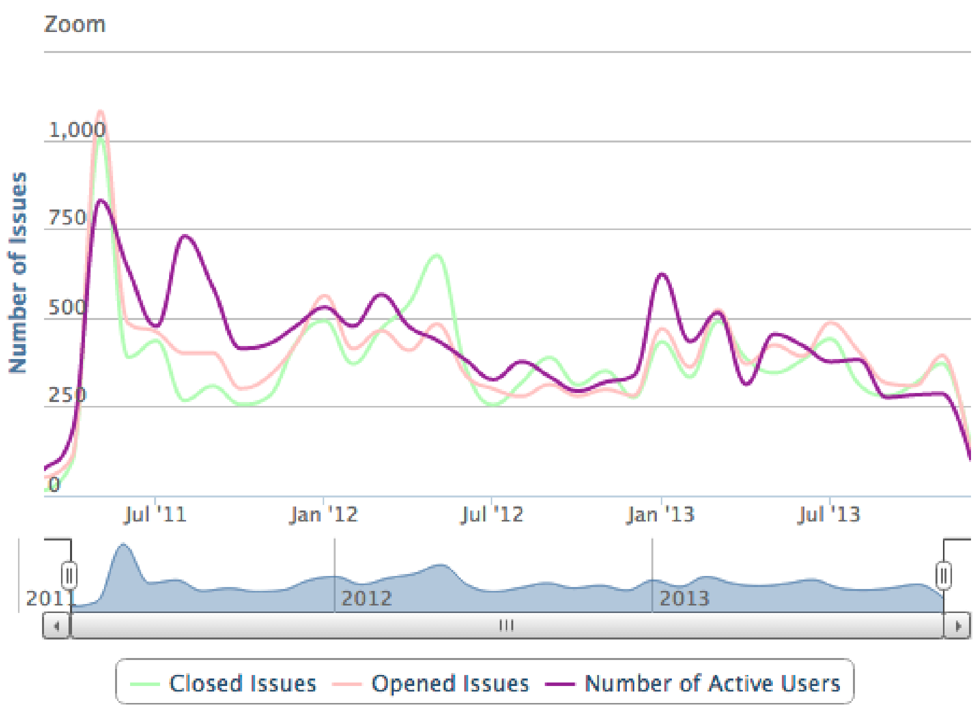
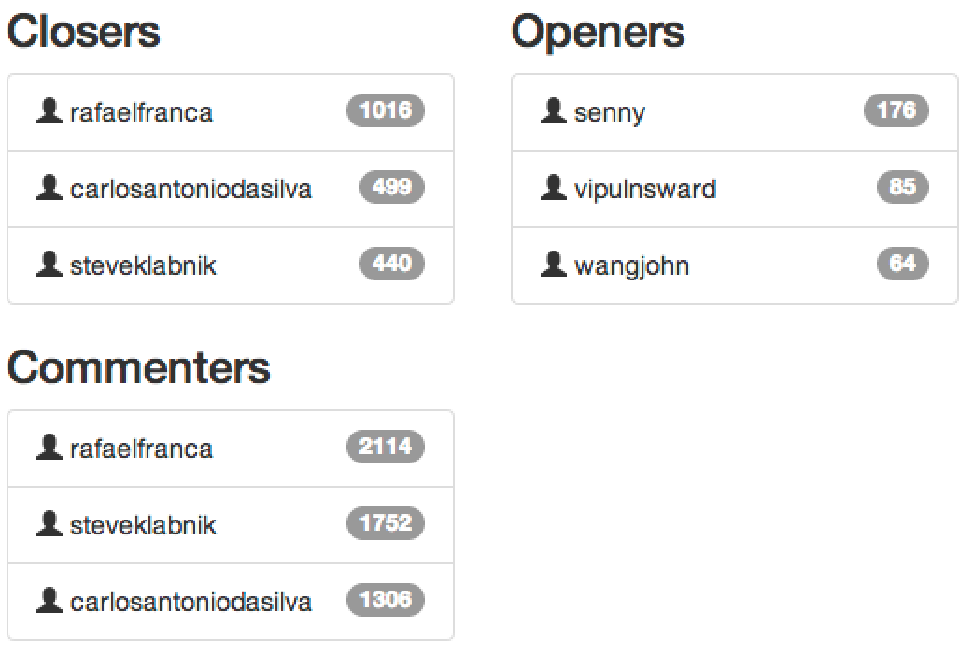


Figure 9.2.: User Activity Chart of the Ruby on Rails project



Closers

rafaelfranca	1016
carlosantoniodasilva	499
steveklabnik	440

Openers

senny	176
vipulnsward	85
wangjohn	64

Commenters

rafaelfranca	2114
steveklabnik	1752
carlosantoniodasilva	1306

Figure 9.3.: User Rankings of the Ruby on Rails project

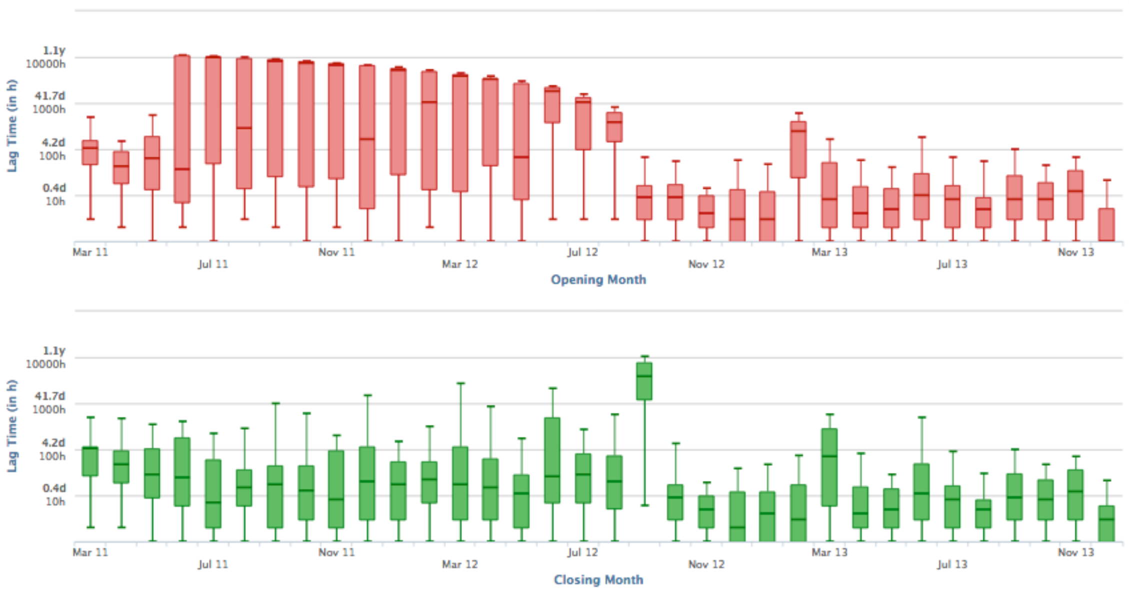


Figure 9.4.: Lag Time Overview of the jquery-file-upload Plugin

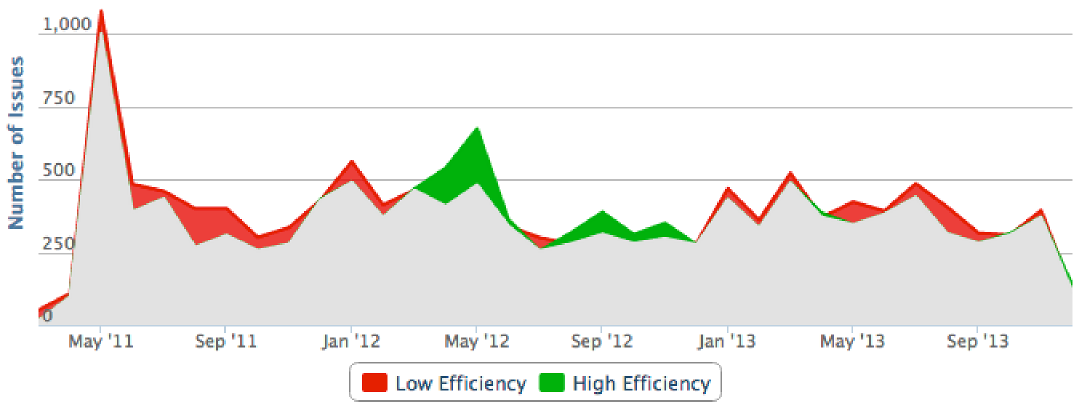


Figure 9.5.: Efficiency Chart of the Ruby on Rails Project

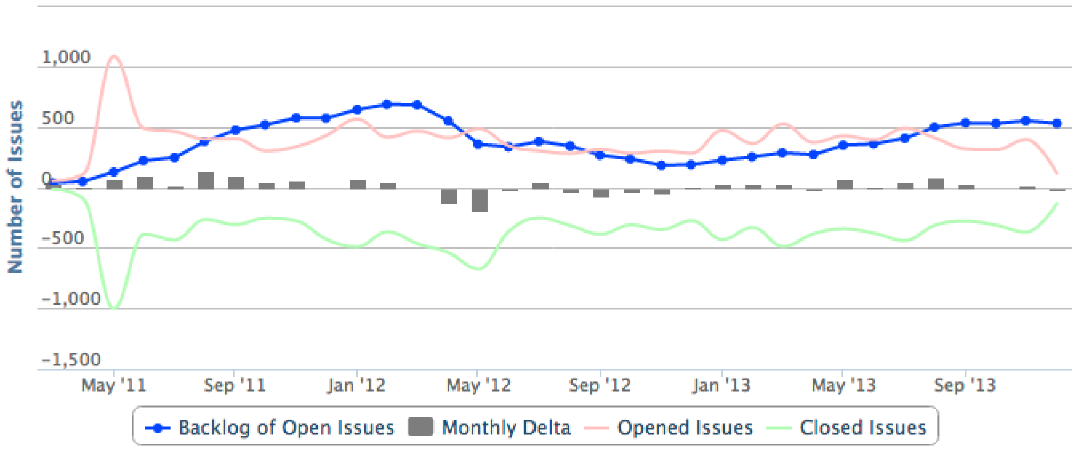


Figure 9.6.: Backlog Chart of the Ruby on Rails Project

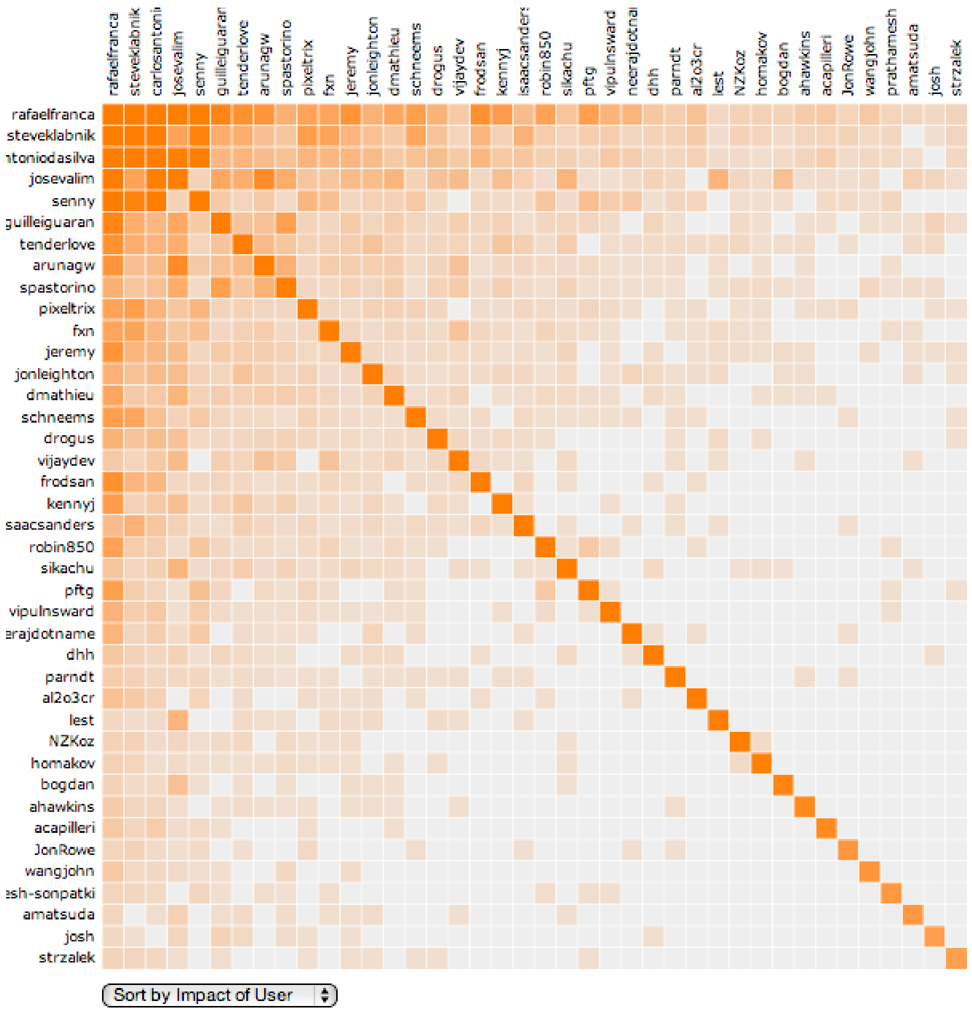


Figure 9.7.: Example Collaboration Overview of the Ruby on Rails Project

Bibliography

[1] Abran, A., and Bourque, P. *SWEBOK: Guide to the software engineering Body of Knowledge.* IEEE Computer Society, 2004.

[2] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds* (2008), ACM, p. 23.

[3] Anvik, J., Hiew, L., and Murphy, G. C. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* (2005), ACM, pp. 35–39.

[4] Anvik, J., and Murphy, G. C. Determining implementation expertise from bug reports. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on* (2007), IEEE, pp. 2–2.

[5] Balzert, H. *Lehrbuch der Softwaretechnik: Softwaremanagement (German Edition).* Spektrum Akademischer Verlag, 2008.

[6] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (2008), ACM, pp. 308–318.

[7] Bortis, G., and Hoek, A. v. d. Porchlight: a tag-based approach to bug triaging. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 342–351.

[8] Butler, B. Github growth points to open source's enterprise acceptance, Dec. 2012. http://www.networkworld.com/news/2012/122112-github-265318.html.

[9] Caldiera, V. R. B. G., and Rombach, H. D. The goal question metric approach. *Encyclopedia of software engineering 2*, 1994 (1994), 528–532.

[10] Cavalcanti, Y. C., da Cunha, C. E. A., de Almeida, E. S., and de Lemos Meira, S. R. Bast-a tool for bug report analysis and search. *XXIII Simpósio Brasileiro de Engenharia de Software (SBES'2009), Fortaleza, Brazil* (2009).

[11] Dalle, J.-M., and den Besten, M. Different bug fixing regimes? a preliminary case for superbugs. In *Open Source Development, Adoption and Innovation*. Springer, 2007, pp. 247–252.

[12] Davies, J., Zhang, H., Nussbaum, L., and German, D. M. Perspectives on bugs in the debian bug tracking system. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on* (2010), IEEE, pp. 86–89.

[13] DeMarco, T. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall, 1986.

[14] Deshpande, A., and Riehle, D. The total growth of open source. In *Proceedings of the Fourth Conference on Open Source Systems* (2008), Springer, p. 197–209.

[15] Fenton, N. E., and Neil, M. Software metrics: roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (2000), ACM, pp. 357–370.

[16] Fischer, M., Pinzger, M., and Gall, H. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering* (2003), IEEE Computer Society, p. 90.

[17] Foundation, M. Bugzilla documentation, Nov. 2013. http://www.bugzilla.org/docs/.

[18] Fowler, M. *Patterns of Enterprise Application Architecture (Addison Wesley Signature Series) Patterns of Enter.* Prentice Hall, 2002.

[19] Fowler, M. Temporal property pattern, Mar. 2004. http://martinfowler.com/eaaDev/TemporalProperty.html.

[20] Francalanci, C., and Merlo, F. Empirical analysis of the bug fixing process in open source projects. In *Open Source Development, Communities and Quality*. Springer, 2008, pp. 187–196.

[21] Fruchterman, T. M., and Reingold, E. M. Graph drawing by force-directed placement. *Software: Practice and experience 21*, 11 (1991), 1129–1164.

[22] Glott, R., Groven, A.-K., Haaland, K., and Tannenberg, A. Quality models for free/libre open source software towards the "silver bullet"? In *Software Engineering*

and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on (2010), IEEE, pp. 439–446.

[23] Gousios, G. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (Piscataway, NJ, USA, 2013), MSR '13, IEEE Press, pp. 233–236.

[24] Hassan, A. E., and Holt, R. C. The top ten list: Dynamic fault prediction. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on* (2005), IEEE, pp. 263–272.

[25] Hindle, A., Godfrey, M. W., and Holt, R. C. Mining recurrent activities: Fourier analysis of change events. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on* (2009), IEEE, pp. 295–298.

[26] Hofmann, G., Riehle, D., Kolassa, C., and Mauerer, W. A dual model of open source license growth. In *Open Source Software: Quality Verification.* Springer, 2013, pp. 245–256.

[27] Hooimeijer, P., and Weimer, W. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (2007), ACM, pp. 34–43.

[28] IEEE. Ieee std 1061-1998. ieee standard for a software quality metrics methodology, 1998.

[29] Jalbert, N., and Weimer, W. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on* (2008), IEEE, pp. 52–61.

[30] Jeong, G., Kim, S., and Zimmermann, T. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2009), ACM, pp. 111–120.

[31] Kim, S., and Whitehead Jr, E. J. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories* (2006), ACM, pp. 173–174.

[32] Kitchenham, B., and Pfleeger, S. L. Software quality: the elusive target [special issues section]. *Software, IEEE 13*, 1 (1996), 12–21.

[33] Lotufo, R., and Czarnecki, K. Improving bug report comprehension.

[34] Luijten, B., Visser, J., and Zaidman, A. Assessment of issue handling efficiency. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on* (2010), IEEE, pp. 94–97.

[35] Martie, L., Palepu, V. K., Sajnani, H., and Lopes, C. Trendy bugs: Topic trends in the android bug reports. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on* (2012), IEEE, pp. 120–123.

[36] Mockus, A., and Votta, L. G. Identifying reasons for software changes using historic databases. In *Software Maintenance, 2000. Proceedings. International Conference on* (2000), IEEE, pp. 120–130.

[37] Nagappan, M., Zimmermann, T., and Bird, C. Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ACM, pp. 466–476.

[38] Panjer, L. D. Predicting eclipse bug lifetimes. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on* (2007), IEEE, pp. 29–29.

[39] Pressman, R. S. *Software engineering: a practitioner's approach*, vol. 5. McGraw-hill New York, 1992.

[40] Raja, U., and Tretter, M. J. Defining and evaluating a measure of open source project survivability. *Software Engineering, IEEE Transactions on 38*, 1 (2012), 163–174.

[41] Robbes, R., and Röthlisberger, D. Using developer interaction data to compare expertise metrics. In *Mining Software Repositories (MSR), 10th IEEE Working Conference on* (2013). Preprint.

[42] Schackmann, H., and Lichter, H. Comparison of process quality characteristics based on change request data. In *Software Process and Product Measurement*. Springer, 2008, pp. 127–140.

[43] Schackmann, H., and Lichter, H. Evaluating process quality in gnome based on change request data. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on* (2009), IEEE, pp. 95–98.

[44] Schneidewind, N. F. Methodology for validating software metrics. *Software Engineering, IEEE Transactions on 18*, 5 (1992), 410–422.

[45] Shumway, R. H., and Stoffer, D. S. *Time Series Analysis and Its Applications: With R Examples (Springer Texts in Statistics).* Springer, 2010.

[46] Snodgrass, R. T., Gray, J., and Melton, J. *Developing time-oriented database applications in SQL*, vol. 42. Morgan Kaufmann Publishers San Francisco, 2000.

[47] Soto, M., and Ciolkowski, M. The qualoss open source assessment model. In *3rd Internation Symposiumm on Empirical Software Engineering and Measurement* (2009).

[48] Sowe, S., Ghosh, R., and Haaland, K. A multi-repository approach to study the topology of open source bugs communities: Implications for software and code quality. In *3rd IEEE International Conference on Information Management and Engineering, IEEE ICIME* (2011).

[49] Stol, K.-J., Babar, M. A., Russo, B., and Fitzgerald, B. The use of empirical methods in open source software research: Facts, trends and future directions. In *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development* (2009), IEEE Computer Society, pp. 19–24.

[50] Tran, H. M., Chulkov, G., and Schönwälder, J. Crawling bug tracker for semantic bug search. In *Managing Large-Scale Service Deployment.* Springer, 2008, pp. 55–68.

[51] unkown. Cisq-tr-2012-01 - cisq specifications for automated quality characteristic measures, 2012.

[52] Unkown. Json specification, Nov. 2013. http://www.bugzilla.org/docs/.

[53] Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories* (2007), IEEE Computer Society, p. 1.

[54] Wenz, C. *JavaScript und AJAX: Das umfassende Handbuch.* Galileo Computing, 2006.

[55] Werner, T.-P. Investing in github, 2012. https://github.com/blog/1189-investing-in-github.

[56] Yu, L., Ramaswamy, S., and Nail, A. Using bug reports as a software quality measure. In *Proceedings of the 16th International Conference on Information Quality (ICIQ'11)* (2011), pp. 277–286.

[57] Yu, L., Schach, S. R., and Chen, K. Measuring the maintainability of open-source software. In *Empirical Software Engineering, 2005. 2005 International Symposium on* (2005), IEEE, pp. 7–pp.

Erklärung

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

Nürnberg - December 16, 2013