Characterization of Source Code Defects by Data Mining Conducted on GitHub

Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Rudolf Ferenc

Department of Software Engineering, University of Szeged, Hungary
Gyimesi.Peter@stud.u-szeged.hu, Gyimesi.Gabor@stud.u-szeged.hu,
zizo@inf.u-szeged.hu, ferenc@inf.u-szeged.hu

Abstract. In software systems the coding errors are unavoidable due to the frequent source changes, the tight deadlines and the inaccurate specifications. Therefore, it is important to have tools that help us in finding these errors. One way of supporting bug prediction is to analyze the characteristics of the previous errors and identify the unknown ones based on these characteristics. This paper aims to characterize the known coding errors.
Nowadays, the popularity of the source code hosting services like GitHub are increasing rapidly. They provide a variety of services, among which the most important ones are the version and bug tracking systems. Version control systems store all versions of the source code, and bug tracking systems provide a unified interface for reporting errors. Bug reports can be used to identify the wrong and the previously fixed source code parts, thus the bugs can be characterized by static source code metrics or by other quantitatively measured properties using the gathered data. We chose GitHub for the base of data collection and we selected 13 Java projects for analysis. As a result, a database was constructed, which characterizes the bugs of the examined projects, thus can be used, inter alia, to improve the automatic detection of software defects.

Keywords: Bug Database, GitHub, Data Mining

1 Introduction

The characterization of source code defects is a popular research area these days. Programmers tend to make mistakes despite the assistance provided by the development environments, and also errors may occur due to the frequent changes and not appropriate specifications, therefore, it is important to get more tools to help the automatic detection of errors. For automatic recognition of defects, it is required to characterize the known ones.

One possible way of characterization is to try retrieving useful information from defective code parts. This requires the knowledge whether a given source code contains bugs or not. Defective sections of code can be characterized in different aspects after locating them.

During the software development cycle, programmers use a wide variety of tools, including bug tracking, task management, and version control systems.

There are numerous commercial and open source software systems available for these purposes. Furthermore, different web services are built to meet these needs. The most popular ones like SourceForge, Bitbucket, Google Code and GitHub fulfill the above mentioned functionalities. They usually provide more services, such as source code hosting and user management. Different APIs make it possible to retrieve various data properties, thus can be used as data sources. For example, they can be used to examine the behavior or the co-operation of users or even to analyze the source code itself. Since most of these services include bug tracking, it raises the idea to use this information in the characterization of source code defects [16]. To do this, the bug reports managed by these source code hosting providers must be connected to the appropriate source code parts [14]. A common practice in version control systems is to describe the changes in a comment belonging to a commit and often provide an ID for the associated bug report which the commit is supposed to fix [10]. This can be used to identify the faulty versions of the source code. Processing diff files can help us to obtain the code sections affected by the bug [15]. We can use textual similarity between faulty code parts [2], if we have such a database. We can also use static source code metrics [5], for which we only need one tool that is able to produce them. For the sake of completeness, other information extracted from the services can be involved in a database, such as user statistics.

To build a database containing useful bug characterization information, we have chosen GitHub since it has several regularly maintained projects and also a well defined API that makes it possible to integrate an automatic data retrieval mechanism in our own project. We selected 13 Java projects, which are suitable for such examination and serve as a base for creating a bug database with different characterizations. We have taken into consideration all reported bugs stored in the bug tracking system. With attached diff files we located the affected source parts. For the characterization we used static source code metrics and some other ones we defined based on the set of data retrieved from GitHub. The set of these metrics describe the projects from many aspects that can be a good starting point to execute different bug prediction techniques (e.g building models for prediction).

2 Related Work

Many approaches have been presented dealing with bug characterization and localization. Zhou et al. published a study describing BugLocator [16], a tool that detects the relevant source code files that need to be changed in order to fix a bug. BugLocator uses textual similarities (between initial bug report and the source code) in order to rank potential fault-prone files. Prior information about former bug reports is stored in a bug database. Ranking is based on the idea that descriptions with high similarity assume that the related files are highly similar too. A similar ranking is done by Rebug-Detector [12] a tool made by Wang et al. for detecting related bugs from source code using bug information. The tool focuses on overridden and overloaded method similarities.

ReLink[14] is developed to explore missing links between changes committed in version control systems and fixed bugs. This tool could be helpful for software engineering research that are based on the linkage data, such as software defect prediction. ReLink mines and analyzes information like bug reporter, description, comments, date from bug database and then try to pair the bug with the appropriate source code files based on the set of source code information extracted from a version control system.

The history of version control systems shows us the concerned files and their changed lines only, but software engineers are also interested in which source code elements (e.g. classes or methods) are affected by a change or a bug [13]. Tóth et al. presented a method for tracking low level source code elements' (class, method) positions in files by processing version control system log information [15]. This method helps to keep source code positions up-to-date.

Kalliamvakou et al. mined GitHub repositories to investigate their characteristics and their qualities [10]. They presented a detailed study discussing different project characteristics, such as (in)activity. Further research questions were involved – whether a project is standalone or a part of a more massive system. Results have shown that the extracted data set can serve as a good input for various investigations, however one must use them with mistrust and always verify the usefulness and reliability of the mined data. It is a good practice to choose projects with many developers and commits, moreover should keep in mind that the most important point is to choose projects that fit well for your own purpose. In our case we have tried to create a database that is reliable (some manual validation is performed) and general enough for testing different bug prediction techniques.

Bird et al. presented a study on distributed version control systems, thus the paper focuses mainly on Git [3]. They examined the usage of version control systems and the available set of data (such as whether the commits are removable, modifiable, movable) gathered by the use of them (with respect of differentiate central and distributed systems). The main purpose of this paper was to draw attention on pitfalls and help researchers to avoid such pitfalls during the processing and analysis of mined Git information set.

Many research papers have shown that using a bug tracking system improves the quality of the developed software system. Bangcharoensap et al. introduced a method to locate the buggy files in a software system very quickly using the bug reports managed by the bug tracking system [2]. The presented method contains three different approaches to rank the fault-prone files, namely:

- Text mining: ranks files based on the textual similarity between a bug report and the source code itself.
- Code mining: ranks files based on prediction of the potential buggy module using source code product metrics.
- Change history: ranks files based on prediction of the fault-prone module using change process metrics.

They used the gathered project data collected on Eclipse platform to investigate the efficiency of the proposed approaches. Finally, they showed that these

three ways are suitable to locate buggy files. Furthermore, bug reports with short description and many specific words greatly increase the effectiveness of finding the weak points (the files) of the system.

Not only the above presented method can be used to predict the occurrence of a new bug, but a significant change in source code metrics can be also a clue that the relevant source code files contain a potential bug or bugs [9]. Couto et al. presented a paper that shows the possible relationship between changed source metrics (used as predictors) and bugs [5]. They described an experiment to discover more robust evidences towards causality between software metrics and the occurrence of bugs.

Previously mentioned approaches use a self-made database for their own purpose as we could seen this advice in the work of Kalliamvakou et al. too [10]. Bug prediction techniques and approaches can be presented and compared in different ways; however, there are some basic points that can serve as common components. One common element can be a database used for evaluation of the various approaches. PROMISE [11] is a database that contains many bugs gathered from open source and also from industrial software systems. The main purpose of PROMISE is to support prediction methods and summarize a bunch of bugs and their characterization extracted from various projects. A similar database for bug prediction was presented, and commonly known as *Bug prediction dataset* [8]. The reason for creating this data set was mainly inspired by the idea of measuring the performance of the different prediction models and also comparing them to each other. This database handles the bugs and the relevant source code parts at class level, in other words the bugs are assigned to classes located in the source code.

At last but not least, the iBUGS database is presented [7] that contains a large amount of information describing projects from the aspect of testing different automatic defect localization methods. Bug describing information comes from version control systems and from bug tracking systems too. iBUGS used the following open source projects to extract the bugs from (in parentheses the number of extracted bugs are shown):

- AspectJ – an extension for the Java programming language to support aspect oriented programming (223).
- Rhino – a JavaSript interpreter written in Java (32).
- Joda-Time – provides a quality replacement (extension) for the Java date and time classes.

An attempt was performed on generating the iBUGS database in an automatic way and the generated set was compared with the manually validated set of bugs [6]. iBUGS is a very promising database since the set of validated bugs is considerable (263), although three projects only cannot guarantee the generality sufficiently. Our database includes several various projects from GitHub which are available from a public API. The given dataset extends the previously shown databases by including more metrics and storing more entries. The shown works successfully made use of their narrowed datasets, thus an extended database can serve as a base for further investigations. Besides 52 static source code metrics,

our dataset contains additional metrics extracted from version control and user management systems. We used the diff files from the version control system to automatically identify the faulty source elements (classes).

3 Approach

In this section we will introduce some prerequisites and the process of creating a database containing bug characterization. At first, we will show some collected information dealing with GiHub, then define different metrics to characterize the reported bugs. Later in this section we will present the data mining process including data collection, processing raw data, analysis of source code versions, and extracting the characteristics of reported bugs.

3.1 GitHub

GitHub is one of today's most popular source code hosting services. It is used by several major open source projects for managing their project, among others Node.js, Ruby on Rails, Spring Framework, Zend Framework, and Jenkins. GitHub offers public and private Git repositories for its users, with some collaborative services, for example built-in bug and issue tracking systems. Since this set of abilities are supported by GitHub, we decided to use this source code hosting service (the well-defined API supports extracting these characteristics). This system can be used for bug reporting, since any GitHub user can add an issue. Issues can be labeled by the collaborators. The system provide some basic labels, such as "bug", "duplicate" and "enhancement", but anybody can customize these tags if required. In an optimal case, the collaborators review these reports and label them with the proper labels, for instance, the bug reports with "bug" label. For us, the most important feature of bug tracking is that we can refer to an issue from the comment of the commit, thereby we can identify a connection between the source code and the reported bug. GitHub has an API[1] that can be used for managing repositories from other systems, or query information about them. This information include events, feeds, notifications, gists, issues, commits, statistics, and user data.

With the GitHub Archive[2] project that also uses this API, we can get up-to-date statistics about the public repositories. For instance, Table 1 presents the number of created repositories in 2014 using the top 10 languages.

As can be seen, this is a large amount of information, and since this is public, it can be useful for mining different properties of the projects stored in GitHub. It means we can obtain the list of commits related to bug reports.

3.2 Metrics

The characterization of developed software systems by certain aspects is a difficult task, because a lot of subjective factors also play roles in them. With

[1] https://developer.github.com/v3/
[2] http://www.githubarchive.org/

Table 1: The number of created repositories in 2014 by the top 10 languages

Language	**Number of repositories**
JavaScript	792 613
Java	562 142
Ruby	480 181
CSS	354 845
PHP	347 113
Python	317 525
C	290 113
C++	164 936
C#	123 707
Objective-C	119 454

metrics we can measure the properties of a project objectively. These properties can describe the whole system itself from various points of view. Metrics can be obtained from the source code, from the project management data or from the execution traces of the source code. There are several different ways to measure them. From software metrics we can deduce higher-level metrics, such as the quality of source code or the distribution of defects, but they can be used to build a cost estimation model, apply performance optimization or to improve activities supporting software quality. In our case, the static source code metrics and the metrics obtained from GitHub are taken into consideration. These can be used to characterize the defective code sections on file level or even on source code element (class) level.

Source Code Metrics The area of object-oriented source code metrics has been researched for many years [4], so no wonder that several tools have been developed for measuring them. These tools are suitable for detailed examination of systems written in various programming languages.

The static object-oriented source code metrics can be divided into several types or groups: size, inheritance, coupling, cohesion and complexity. Calculated product and process metrics can be used for different quality assurance methods as well. One such example can be a development of a quality rating model [1] or another application can be the determination of the correlation between the distribution of the bugs and the calculated metrics [9]. For such purposes a database containing readily extracted software metrics and located bugs provides a great opportunity. The list of used software metrics in the characterization is shown in Table 2.

Table 2: Used metrics for characterization

Abbreviation	Full name
LCOM5	Lack of Cohesion in Methods 5
NOA	Number of Ancestors
NOC	Number of Children
NOD	Number of Descendants
NOP	Number of Parents
NOI	Number of Outgoing Invocations
NOS	Number of Statements
CBOI	Coupling Between Object classes Inverse
NPA	Number of Public Attributes
TCLOC	Total Comment Lines of Code
TNLM	Total Number of Local Methods
TNLG	Total Number of Local Getters
TNLA	Total Number of Local Attributes
NPM	Number of Public Methods
CLOC	Comment Lines of Code
NLPM	Number of Local Public Methods
AD	API Documentation
TNLS	Total Number of Local Setters
NLPA	Number of Local Public Attributes
TNPM	Total Number of Public Methods
TNPA	Total Number of Public Attributes
NLG	Number of Local Getters
NLM	Number of Local Methods
DIT	Depth of Inheritance Tree
NLA	Number of Local Attributes
NLE	Nesting Level Else-If
TNOS	Total Number of Statements
CD	Comment Density
NLS	Number of Local Setters
LOC	Lines of Code
LLOC	Logical Lines of Code
TCD	Total Comment Density
RFC	Response set For Class
NG	Number of Getters
NL	Nesting Level
NM	Number of Methods
NA	Number of Attributes
NS	Number of Setters
TNLPM	Total Number of Local Public Methods
DLOC	Documentation Lines of Code
TNLPA	Total Number of Local Public Attributes
NII	Number of Incoming Invocations
WMC	Weighted Methods per Class
TNG	Total Number of Getters
TLLOC	Total Logical Lines of Code
TNA	Total Number of Attributes
PUA	Public Undocumented API
TLOC	Total Lines of Code
TNS	Total Number of Setters
TNM	Total Number of Methods
PDA	Public Documented API
CBO	Coupling Between Object classes

Metrics Extracted from Version Control System In addition to the static source code metrics we gathered also other metrics from the available data. From the version control system the number of modifications and fixes on a file can be

easily determined; moreover, committer identity can be mapped to the changed files. Furthermore, GitHub provides statistics about the users, that includes the number of commits per user on a project. From these data we determined to create the following metrics on file level:

- Number of modifications
- Number of fixes
- Number of opened issues
- Number of modifications the committer performed on the project

3.3 Data Mining

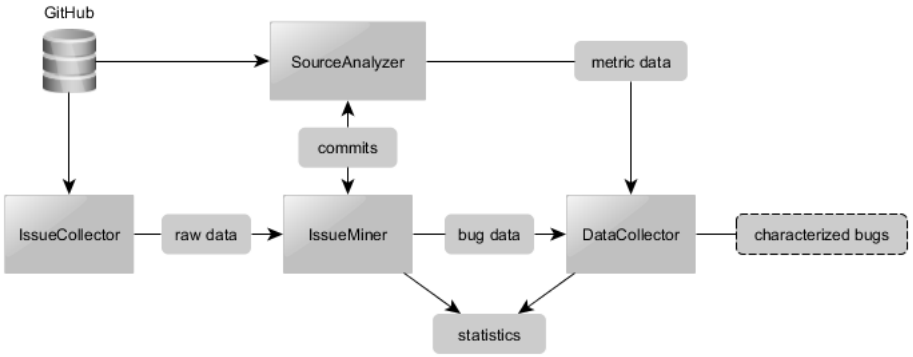

Fig. 1: The components of the process

We carried out the data processing in multiple steps. First we collected the data from GitHub by our IssueCollector program. Then we processed the raw data and created statistics, for which we have developed a program component called IssueMiner. For the next step we applied the SourceAnalyzer component, which downloads the necessary source code versions from GitHub and analyzes them. After this, we connected the results of the analysis with the data downloaded from GitHub, and located the defective sections of code and characterized them with the calculated metrics. For this purpose, the DataCollector tool was developed. The process and components are illustrated in Figure 1.

The Criteria for Choosing Projects We considered a number of criteria when searching for appropriate projects on GitHub. First of all, we searched for Java language projects, especially larger ones, because these are more suitable for this kind of analysis. It was also important to have an adequate number of commits which use bug labels in reports to separate them clearly from other reports, moreover to refer to the appropriate bug report from the description of

the commits. In addition, we preferred the currently active projects. We found many projects during the search, which would have fulfilled most aspects but in many cases developers used an external bug tracker system, so it would have been difficult to process them.

The List of Selected Projects We have selected 13 projects based on the previously described aspects. Some properties of the projects will be presented, but first we introduce the set of selected software systems. The following projects were considered adequate for selection:

- **JUnit**[3]: A Java framework for writing unit tests.
- **Mission Control Technologies**[4]: Originally developed by NASA for the space flight operations. It is a real-time monitoring and visualization platform that can be used for monitoring any other data as well.
- **OrientDB**[5]: A popular document-based NoSQL graph database. Mainly famous for its speed and scalability.
- **Neo4j**[6]: The world's leading graph database with high performance.
- **MapDB**[7]: A versatile, fast and easy to use database engine in Java.
- **mcMMO**[8]: An RPG game based on Minecraft.
- **Titan**[9]: A high-performance, highly scalable graph database.
- **Oryx**[10]: It is an open source software with machine learning algorithms that allows the processing of huge data sets.
- **jHispter**[11]: A versatile software for generating Java Web applications.
- **Universal Image Loader**[12]: An Android library that assists the loading of images.
- **Netty**[13]: It is an asynchronous event-driven networking framework.
- **ANTLR v4**[14]: A popular software in the field of language processing. It is a powerful parser generator for reading, processing, executing, or translating structured text or binary files.
- **Elasticsearch**[15]: A popular RESTful search engine.

Table 3 provides a more accurate picture of the projects. This table shows the number of bug reports and commits of the projects. Explanation of used abbreviations are described in the following:

[3] https://github.com/junit-team/junit
[4] https://github.com/nasa/mct
[5] https://github.com/orientechnologies/orientdb
[6] https://github.com/neo4j/neo4j
[7] https://github.com/jankotek/MapDB
[8] https://github.com/mcMMO-Dev/mcMMO
[9] https://github.com/thinkaurelius/titan
[10] https://github.com/cloudera/oryx
[11] https://github.com/jhipster/generator-jhipster
[12] https://github.com/nostra13/Android-Universal-Image-Loader
[13] https://github.com/netty/netty
[14] https://github.com/antlr/antlr4
[15] https://github.com/elasticsearch/elasticsearch

Table 3: Statistics about the selected projects

	NC	NCBR	NBR	NOBR	NCLBR	ANCBR
Android Universal I. L.	914	52	80	5	75	0,69
ANTLR v4	2 941	109	146	16	130	0,84
Elasticsearch	9 764	979	1 331	91	1 240	0,79
jHipster	1 436	52	68	0	68	0,76
jUnit	1 942	66	75	4	71	0,93
MapDB	1 052	80	109	18	91	0,88
mcMMO	4 476	251	635	11	624	0,40
Mission Control T.	975	15	37	9	28	0,54
Neo4j	29 208	76	268	112	156	0,49
Netty	6 254	567	747	28	719	0,79
OrientDB	8 404	362	710	212	498	0,73
Oryx	295	29	27	0	27	1,07
Titan	1 690	50	88	6	82	0,61

NC Number of Commits
NCBR Number of Commits per Bug Reports
NBR Number of Bug Reports
NOBR Number of Open Bug Reports
NCLBR Number of CLosed Bug Reports
ANCBR Average Number of Commits per Bug Reports

Figure 2 shows the number of commits for closed bug reports. This shows that there is a relatively large number of cases without a single commit. There are possible causes, for example, bug report is not referred from the commit description, the error has already been fixed, or a commit was not made with the purpose to fix the problem.

Figure 3 shows the ratio of the number of commits per projects, illustrating the activity and the size of the projects. Neo4J is dominant if we only consider the number of commits, however bug report related activities are slight.

3.4 Data Collection

At the beginning we saved the data for the selected projects via the GitHub API. It was necessary, because the data is continuously changing on GitHub due to the activity of the projects and we need a consistent data source for the analysis.

The saved data set includes the users assigned to the repository (Contributors), the open and the closed bug reports (Issues), and all of the commits. About users we stored the user id and the number of commits on the repository they formerly have applied to. From open issues we only stored the date of their creation. For closed issues we stored the creation date, closing date and the commit identifiers with their creation dates. The data we stored for the commits includes the id of the contributor, the id of the development branch, the parent(s) of the commit and the affected files with the diff files.

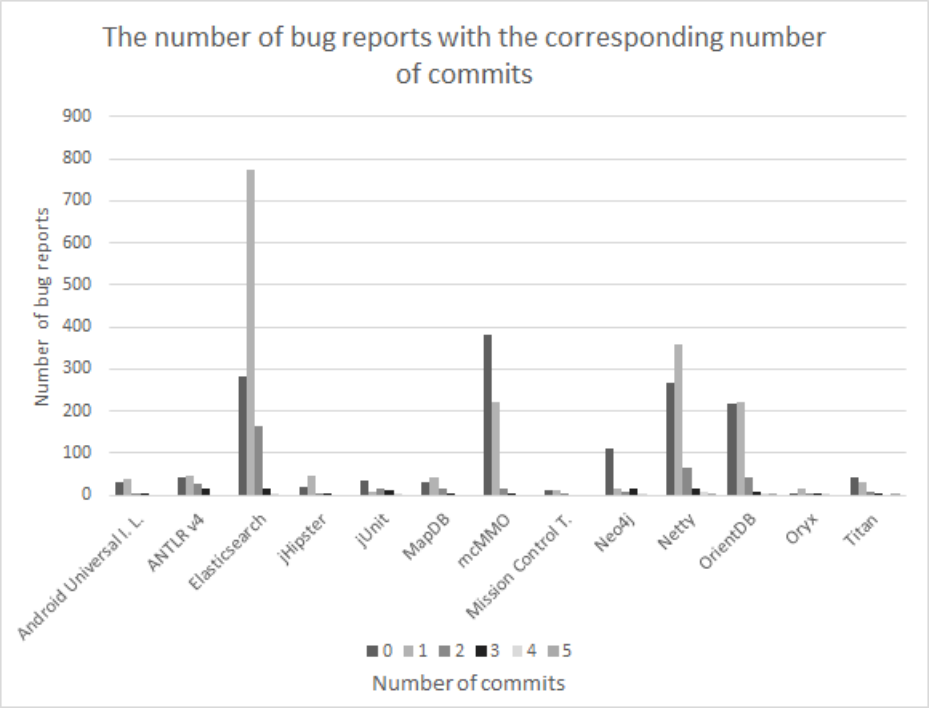

Fig. 2: The number of bug reports with the corresponding number of commits

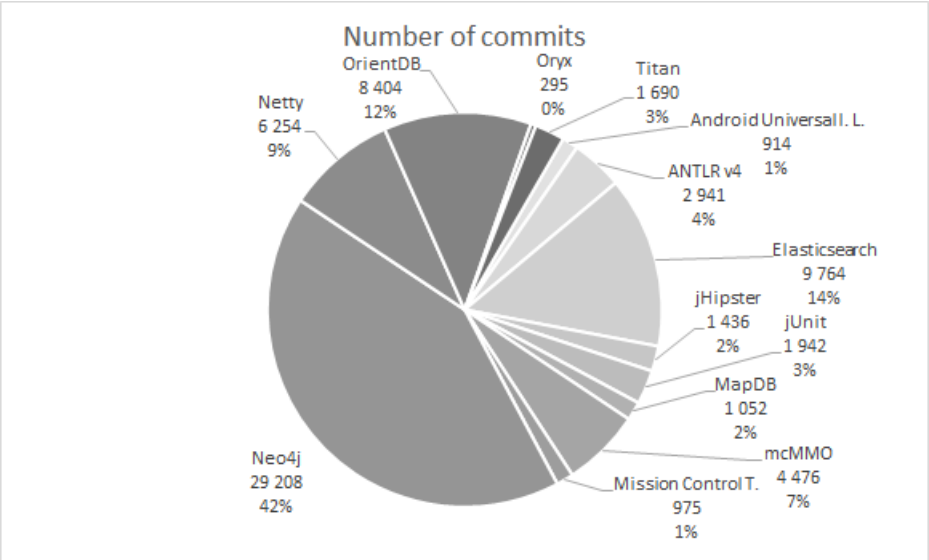

Fig. 3: The number of commits per projects

3.5 Processing of Raw Data

Data saved from GitHub is only a raw data set that includes all commits. We only need the ones that are relevant to the bug reports. These are the commits with a reference to a bug issue (fix) and the commits applied after the submission of the report but before the commit referenced the issue. Between these commits some further ones can occur that need to be removed because they are no longer available through Git (deleted, merged). During data processing, we performed such filtering and we made some more statistics about the projects including the number of issues closed from commits. Figure 4 shows that not all of the projects use this feature. In other words, we must deal with both options.

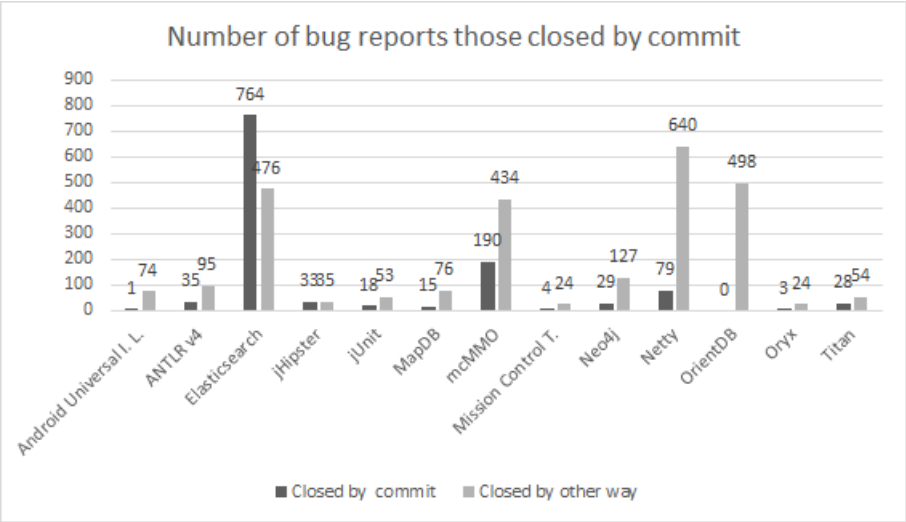

Fig. 4: Number of bug reports closed from commit compared with closed from web interface

3.6 Analysis of Source Code Versions

We downloaded the proper versions of the source code based on the previous step. In this step we filtered out the ones that cannot be downloaded because they have been deleted or are created by merging branches. In such cases, there is an alternative commit with the same content, and it can be downloaded. At this point we had the source code versions which we want to analyze. For code analysis purposes we used our SourceAnalyzer component. It wraps the results of the SourceMeter[16] tool that computes the static source code metrics and determines the source code positions of the code elements. We got the results in

[16] https://www.sourcemeter.com/

the form of graphs which contains the packages and classes with the computed data.

3.7 Extracting the Characteristics of Bugs

The next step is to link the two data sets – the graphs of analysis and the data gathered from GitHub – and extract the characteristics of the bugs. In this step we determine the followings:

- the source code elements affected by the commits
- the static source code metrics of the affected source code elements
- the number of modifications and fixes of the files in each commit
- the last user modified a file in relevant commits
- the number of open bug reports in relevant commits

This was carried out by our DataCollector program. To determine the affected source code parts, we used diff files. These files contain the differences between two source code versions in a unified diff format. In the following, a unified diff file snippet is shown.

```
--- /path/to/original ''timestamp''
+++ /path/to/new ''timestamp''
@@ -1,4 +1,4 @@
+Added line
-Deleted line
 This part of the
 document has stayed the
 same
```

Each difference contains a header information specifying the starting line number and the number of affected lines. Using this prior information, we can get the range of the modification. To obtain a more accurate result, we subtracted the unmodified code lines from this range. The diff files generated by GitHub contain additional information about which method is affected. For us, it does not carry any extra information because the difference can affect multiple source code elements. Thus, there is no further task to do but to examine the source code elements in every modified file and identify which ones of them are affected by the changes (method uses the source code element positions). We identified the source code elements by their fully qualified names. With this algorithm we got the affected source elements as a result set.

Now we have enough information to calculate some additional metrics. The program calculates the number of open bug reports for each commit. This is done by counting the issues whose creation date is earlier than the creation of the commit, and the closing time is later. Furthermore, it calculates the number of modifications of each file. At first, it arranges the commits by the order of creation time. Starting with the earliest one, it increases the counter on a file if it is affected by a commit. During this process it is also counting the number

of fixes. A modification is considered as a fix if it is in the last commit for a closed issue. Lastly, it determines the user for each file who most recently has modified it. It is used to connect the user statistics with the modifications. The user statistics means the number of modifications on a project applied by the user. The number of modifications is collected at the time of downloading the data from GitHub and not at the time the commit was made.

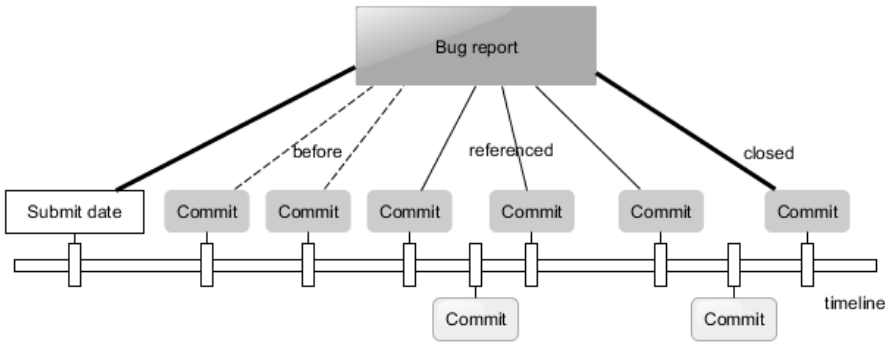

Fig. 5: The relationship between the bug reports and commits

Next, our program determines the commits that were performed after the creation time of an issue and before the first commit for fixing that issue. These are essential because these versions presumably contained the buggy source code parts. The relation between the bug report and the commits is shown in Figure 5. To mark the code sections affected by the bug in these commits, the program accumulates the modifications on issue level. It is done by collecting the fully qualified name of these elements. Then these metrics are exported into a CSV document. This is done for each bug report due to resource saving purposes because the graphs can be very large in size. The metrics for files and classes are exported to different files. One file specifies whether a source code was buggy or not, the other one contains assignment of source code elements and the number of bugs related to them. Thus, four types of output is generated in this manner. Finally, it concatenates these files resulting in a large set of data. The first line of this CSV file contains the header with the metric names.

Once our dataset is created it can serve as an input for building a model for fault prediction that one can use to forecast fault-prone spots in any developed software system. The database can be easily updated since only a filtered analysis should be performed (from a given date) that can extend the previous version of the dataset.

4 Conclusion and Future Work

In this study, we developed a method and performed its implementation, which generates a bug related database mined from GitHub project hosting service using static source code metrics of the relevant code parts. It identifies the faulty source code elements from the past automatically by using diff files from the version control system. This way it allows the simultaneous processing of several publicly available projects located on GitHub, thereby resulting in the production of a large database. Previous studies have dealed with only few larger data sets created under strict management, as opposed to our way. Additionally, our dataset contains new static source metrics compared to the other databases, allowing the examination of the relationship between these metrics and software bugs. Furthermore, in result of examining the projects on GitHub, we selected 13 suitable Java projects, which we used to build the database.

We are planning to expand the database with additional projects and additional data sources, such as SourceForge and Bitbucket. We also plan to refine the metrics and define new features. The calculated file level metrics – with further analysis – can be determined for lower level source elements, i.e. for classes and methods, and data gathered from GitHub also can be used do define more metrics. Our ultimate goal is to use the data to examine the correlation between the bugs and the source code metrics, and to apply the results to facilitate the automatic recognition of source code defects.

Acknowledgment

The publication is partially supported by the European Union FP7 project "REPARA – Reengineering and Enabling Performance And poweR of Applications", project number: 609666.

We also thank Zsuzsanna Fehér and László Szoboszlai for their valuable assistance.

References

1. T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. A probabilistic software quality model. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 243 –252, sept. 2011.
2. P. Bangcharoensap, A. Ihara, Y. Kamei, and K. Matsumoto. Locating source code to be fixed based on initial bug reports - a case study on the eclipse project. In *Empirical Software Engineering in Practice (IWESEP), 2012 Fourth International Workshop on*, pages 10–15, Oct 2012.
3. C. Bird, P.C. Rigby, E.T. Barr, D.J. Hamilton, D.M. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 1–10, May 2009.
4. Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

5. C. Couto, C. Silva, M.T. Valente, R. Bigonha, and N. Anquetil. Uncovering causal relationships between software metrics and bugs. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 223–232, March 2012.
6. Valentin Dallmeier and Thomas Zimmermann. Automatic extraction of bug localization benchmarks from history. Technical report, Universitat des Saarlandes and Saarbrücken and Germany, 2007.
7. Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436. ACM, 2007.
8. Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 31 – 41, 2010.
9. Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, 2005.
10. Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. *MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101, 2014.
11. Tim Menzies, Bora Caglayan, Zhimin He, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, June 2012.
12. Deqing Wang, Mengxiang Lin, Hui Zhang, and Hongping Hu. Detect related bugs from source code using bug information. *Computer Software and Applications Conference (COMPSAC)*, 2010.
13. Chadd C Williams and Jeffrey K Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *Software Engineering, IEEE Transactions on*, 31(6):466–480, 2005.
14. Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM, 2011.
15. Toth Z., Novak G., Ferenc R., and Siket I. Using version control history to follow the changes of source code elements. *Software Maintenance and Reengineering (CSMR)*, 2013.
16. Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. *Software Engineering (ICSE), 2012 34th International Conference on*, 2012.