Acta Cybernetica **23** (2017) 537–559.

Automatic Calculation of Process Metrics and their Bug Prediction Capabilities

Péter Gyimesi[a]

Abstract

Identifying fault-prone code parts is useful for the developers to help reduce the time required for locating bugs. It is usually done by characterizing the already known bugs with certain kinds of metrics and building a predictive model from the data. For the characterization of bugs, software product and process metrics are the most popular ones. The calculation of product metrics is supported by many free and commercial software products. However, tools that are capable of computing process metrics are quite rare. In this study, we present a method of computing software process metrics in a graph database. We describe the schema of the database created and we present a way to readily get the process metrics from it. With this technique, process metrics can be calculated at the file, class and method levels. We used GitHub as the source of the change history and we selected 5 open-source Java projects for processing. To retrieve positional information about the classes and methods, we used SourceMeter, a static source code analyzer tool. We used Neo4j as the graph database engine, and its query language - *cypher* - to get the process metrics. We published the tools we created as open-source projects on GitHub.

To demonstrate the utility of our tools, we selected 25 release versions of the 5 Java projects and calculated the process metrics for all of the source code elements (files, classes and methods) in these versions. Using our previous published bug database, we built bug databases for the selected projects that contain the computed process metrics and the corresponding bug numbers for files and classes. (We published these databases as an online appendix.) Then we applied 13 machine learning algorithms on the database we created to find out if it is feasible for bug prediction purposes. We achieved F-measure values on average of around 0.7 at the class level, and slightly better values of between 0.7 and 0.75 at the file level. The best performing algorithm was the *RandomForest* method for both cases.

Keywords: process metrics, graph database, bug prediction

[a] Department of Software Engineering, University of Szeged, Hungary,
E-mail: pgyimesi@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.7

1 Introduction

Nowadays, companies tend to spend a large amount of resources on debugging and fixing software faults. Predicting these bugs can greatly help to reduce the costs. For this reason, bug prediction has become a popular research area. Recognizing bug-prone source code parts requires that one characterize them in some way.

There are many good studies on bug characterization [6, 22, 3, 4]. It can be carried out with classic product metrics, with software process metrics or with some metrics of a different nature like textual similarity. Product metrics are extracted from the structure of the source code. Some examples are lines of code, cyclomatic complexity and number of methods. There are many tools – some of them are free – which can produce these metrics for projects of different programming languages. These metrics are frequently used for bug characterization [19], because it is easy to compute them. Product metrics depend on a single state of the software and no project history is required; thus no temporal characteristics are used. These metrics are usually computed for files or classes, but an increasing number of tools support methods too.

Software process metrics are computed from developer activities. Most of them include some kind of temporal information. The most common ones are based on the number of previous modifications, number of different contributors, number of modified lines and the time of the modifications. These metrics can of course be used for a variety of purposes. Since the computation is based on the developers' activities, these values are perfect for examining the developers' behavior. Furthermore, locating key source code parts that are modified often or recently is another possible utilization of these metrics.

Previous studies [20, 16, 10, 11] have shown that while software process metrics are generally better bug predictors than product metrics, tools that can compute these metrics are still quite rare. The studies focus mainly on the definition of process metrics and the results. The method of computing these metrics is not always described, so reproducing these results may be a challenging task. It may be due to the difficulties of storing and processing the historical information. An important criterion here is to have available project history. Versioning systems (like Git or Subversion) are commonly used in software development, thus the history of a project is quite often accessible through an API. Source code hosting services like GitHub or Bitbucket are becoming evermore popular and contain open-source projects of various programming languages. Another criterion is that the developers have to use this system correctly, otherwise this information is not useful and it may be misleading.

The first problem we run into is the size of this data set. A project may contain hundreds of thousands lines of source code and also thousands or tens of thousands of commits. To compute the process metrics for one software version, the whole history has to be processed, hence an efficient method is required for this task.

Another aspect of this problem is the granularity of process metrics. Calculations can be made at different levels: file, class or method. At the file level, it is fairly simple, because versioning systems work with files, so no additional analysis is needed. However, at the class level and the method level, a thorough source code analysis is required to extract the source code elements and their position. For more accurate results, other information (empty lines, comments, etc.) may be gathered. This task can be carried out with a static source code analysis tool[1].

The next issue is how to store the gathered data in an easily accessible form. In the past few years, the popularity of graph databases has increased due to the improving technologies behind them. A graph database can handle a large amount of data and it is suitable for storing weakly structured data. The historical data of a software package can be represented as a graph, so studies [2] have started to examine the application of these graph databases to assess software quality, especially in the calculation of software process metrics, hence graph databases seem to be a good choice for this task.

We chose GitHub as a data source because it contains more than 38 million repositories[2] and has a readily usable API[3] to access these projects. Furthermore, these repositories are accessible via Git[4]. We chose SourceMeter[5] as a source code analysis tool, because it is capable of processing five programming languages (Java, C++, C#, Python, RPG) and it can extract detailed information about the source code elements, including methods. The results of this analysis is of course a graph. It contains the source code elements (files, classes, methods) as nodes and the corresponding relationships between these elements as edges. Also, it has a Java API for the graph. These features make it an ideal choice for this task. Due to the amount and structure of data we are dealing with, we decided to use Neo4j[6] for data storage. It is currently the most popular open-source graph database. It also has a powerful query language called *cypher* that can be utilized for computing process metrics.

Our motivation is to provide a way of computing software process metrics quickly and easily. The main contributions of this study are the following:

- A method for automatically calculating process metrics in a graph database for files, classes and methods;
- An open-source implementation of the presented method;
- Assessment of the bug prediction capabilities of the calculated process metrics; and
- A publicly available bug database with process metrics.

The remainder of the paper is organized as follows. Section 2 presents some related work and Section 2.1 summarizes the process metrics used in other studies. In Section 3, we present the database schema that we designed, the steps of its

[1] `https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`
[2] `https://github.com/about`
[3] `https://developer.github.com/v3/`
[4] `https://git-scm.com/`
[5] `https://www.sourcemeter.com/`
[6] `https://neo4j.com/`

construction and the calculation of process metrics in this database. Section 4 lists the processed software systems with some statistics concerning them. In Section 5, we present the results of machine learning algorithms applied to this database. Lastly, in Section 6 we draw some pertinent conclusions and suggest some plans for future study.

2 Related Work

Bug prediction is an intensively studied research area [12, 14, 11] and there are publicly available databases that can be used for bug prediction purposes. The biggest of these datasets is the tera-PROMISE [13] repository. It is up-to-date and it is regularly maintained. It contains, among other things, bug databases with various metrics, like rule violations, object-oriented metrics and complexity metrics. Actually, some studies utilize this dataset, but there are many researchers who build their own databases and never publish them.

D'Ambros et al. [4] made an extensive study on bug prediction. They compared the well-known bug prediction approaches. As part of their research, they created a benchmark database from several open-source projects (Eclipse, Mylyn, Lucene). This database contains bug numbers at the class level with 15 change metrics and 17 product metrics. The bug information was extracted from the commit messages and bug tracking systems by using pattern matching, as others did in earlier studies [24, 5]. They describe the whole process of building such a database, but the links to the tools used do not work anymore. They computed change metrics and bug information for files due to the file-based version control systems. In the case of Java inner classes, the same information was linked to multiple classes. Since they did not have a solution to this problem, they filtered these inner classes from the process. They found that the Weighted Churn and Linearly Decayed Entropy metrics perform the best (around 90%) for bug prediction, but the computation of these metrics is quite complex. Furthermore, they concluded that multiple metrics should be used for this purpose in order to achieve good results across multiple systems.

The Eclipse project is used quite often for studies on bug prediction. Bernstein et al. [1] used this project to examine whether temporal features are suitable for bug prediction. They gathered change information from CVS and bugs from Bugzilla and they computed several temporal features. They built non-linear models for the bug database they created and they achieved a high accuracy score (99%) on predicting defects. They concluded that temporal features (process metrics) and non-linear models are suitable for bug prediction. Moser et al. [15] also used the Eclipse project to investigate the characteristics of change metrics in bug prediction. They calculated 18 change metrics at the file level. They achieved better results with these metrics than with product metrics [16] and they showed that 3 out of 18 change metrics can achieve good results, and they are as stable as the model with all the metrics. These three metrics are the following: number of revisions, number of bug fixes, maximum size of all of its change sets.

Shihab et al. [22] also examined whether the number of the predictors can be reduced. As a data source, they used the Eclipse data set [24]. They showed that the 34 product and process metrics can effectively be reduced to 4 with very little difference in the overall prediction accuracy. They found that the most stable independent metrics were: total prior changes, number of pre-release defects and TLOC.

The study made by Krishnan et al. [10] sought to answer the research question of whether the process metrics are good bug predictors for the family of products in the evolving Eclipse product line. They replicated the results previously achieved by Moser et al. [16] and extended them with their observations. They concluded that process metrics are good bug predictors for the Eclipse product line. Furthermore, they found that a small subset of these metrics are stable and consistent across multiple projects. They are called maximum changeset, number of revisions and number of authors.

Graves et al. [6] also made a study on the bug prediction capabilities of process metrics. They computed the metrics at the module level and analyzed systems written in C. Their observation was that the best model used the weighted time damp metric and the best linear models used number of changes and age metrics. They found that the number of developers and the changeset metrics did not influence the accuracy of the fault prediction.

In an earlier paper [7], we presented a method for characterizing software bugs with product metrics. In this method, we include temporal information by building the bug database from the buggy source code elements before and after the fix, but of course more sophisticated temporal characteristics should be included.

Studies have shown that process metrics usually perform better in bug prediction than product metrics do. Rahman et al. [20] analyzed the properties of process metrics from the perspective of performance, stability, portability and stasis. They found that product metrics have a higher stasis - which means they do not change much compared to the process metrics -, thus the same elements were predicted as defective over and over. Also, product metrics are less stable and less portable across projects.

Hassan [8] went further. In his paper, he proposed complexity metrics that are based on process metrics. He analyzed 6 projects written in C and C++ and computed process metrics at the file level, but he did not give a detailed description of the method of processing and how to compute these metrics. He concluded that the proposed change complexity metrics are better fault predictors than the well-known process metrics. He also said that we should consider using these metrics instead of the simple metrics like number of prior modifications and number of prior faults.

Buginfo[7] is a tool that is used for collecting bug information from source code repositories [9]. It uses regular expressions on the commit messages to count the number of bugs in classes, as other studies did [24, 5]. It is also capable of computing process metrics, but unfortunately the tool is not maintained.

[7] `https://kenai.com/projects/buginfo`

In this study, we present a method to automatically compute the process metrics for GitHub projects. We use Neo4j, a graph database engine to store the information collected, and we utilize the cypher query language to readily compute these metrics. None of the previous studies used GitHub as a data source. Also, none of them defined a graph schema for the data they collected nor did they use a graph database. Furthermore, there are no studies to date that calculate software process metrics at the method level.

2.1 Process Metrics

In the literature, there are many software process-related metrics [17, 18, 8, 1] and they were mainly used in studies concerning bug prediction. In these studies, the authors evaluated the predictive capability of these metrics and they often compared this capability with that of product metrics [16]. They found that the age of a file and the size of the change metrics are usually better predictors than the others [15, 10]. In this study, we did not rank these metrics in any way. Here, we enumerate the definitions of the most common metrics:

- **Number of Modifications**: The number of previous modifications of the source element.
- **Number of Bug Fixes**: The number of previous modifications of the source element that reflect an intention to fix a bug.
- **Number of Versions**: The number of software versions (revisions) since the source element is created. In other words, the number of commits on the whole project since the creation of the element.
- **Number of Re-factorings**: The number of previous modifications of the source element that were committed in order to perform re-factoring.
- **Age**: The age of the source element in days, weeks or months.
- **Weighted Age**: The weighted age is calculated using the age and size of the previous modifications. [16] It may be expressed in days, weeks, or months. The formal definition is the following:

  $$WeightedAge(e) = \frac{\sum_v Age(v) \times NumberOfAddedLines(e,v)}{\sum_v NumberOfAddedLines(e,v)} \qquad (1)$$

  In this formula, we would like to compute the metric for the source element e. The *Age* is the age of the software version v (days, weeks, or months), where v is earlier than the version for which we want to calculate. The *NumberOfAddedLines* represents the number of lines added for source element e in version v.
- **Number of Contributors**: How many different developers contributed to the source element.

- **Number of Contributor Changes**: The number of developer changes in the code history. A developer change occurs when the next sequential modification on the same source element was performed by a different developer.
- **Sum of Added Lines**: The total sum of the lines of code added to the source element.
- **Maximum Number of Added Lines**: The maximum number of lines of code added with one commit to the source element.
- **Average Number of Added Lines**: The average number of lines of code added to the source element.
- **Number of Additions**: The number of previous commits in which new lines were added to the source element.
- **Sum of Deleted Lines**: The total sum of the lines of code deleted from the source element.
- **Maximum Number of Deleted Lines**: The maximum number of lines of code deleted with one commit from the source element.
- **Average Number of Deleted Lines**: The average number of lines of code deleted from the source element.
- **Number of Deletions**: The number of previous commits containing lines that were deleted from the source element.
- **Code Churn**: The sum of lines added minus lines deleted from the source element [17].
- **Relative Code Churn**: The normalized Code Churn metric. Normalization can be achieved with, for example, lines of code, file count or time period [18].
- **Maximum Number of Elements Modified Together**: The maximum number of distinct elements that were modified with one commit.
- **Average Number of Elements Modified Together**: The average number of distinct elements that were modified together with the source element.
- **Average Time Between Changes**: The average number of days, weeks or months that passed between consecutive modifications of the source element.
- **Author**: The identity of the original author of the source element. It may include other information about the developer, such as the total number of commits of the author and number of projects.
- **Number of Referenced Issues**: The number of distinct issues referenced in the comments of commits that contain modifications of the source element.

- **Number of Commits Without Message**: The number of previous modifications without any comment message.

Most of these metrics are based on the versioning information and the issue tracking data, but there are others, for instance, that include software management data. We did not process such data source so we will not describe them here. Furthermore, more specific characteristics may be taken into account like *Number Of Referenced High Priority Issues* if issue priority is available. Since these details may vary from system to system, we omitted these variations from the study and we concentrated on the most common ones.

Other metrics can be formed like *Change Activity Rate* [21], which is defined as the overall number of modifications relative to the age of the source element in months. This metric can be computed by a simple division. The calculation of these combined metrics is straightforward, so we will not discuss them.

Most of the metrics listed above are computed for a given version (revision), except the fixed characteristic like *Author*. In the literature, these metrics are defined for files or modules (collection of files). We defined them for source elements which may be a file, class or even a method. Furthermore, a time period can be specified for most of these metrics. For example, we can limit the interval of the calculation for the last six months. This way, we can produce metrics like *Number Of Modification In The Last Six Months*. To save space, we did not list every variation.

3 Methodology

After analyzing the available data sets (project history, static source code analysis results), we designed a graph database schema and it is shown in Figure 1. Our goal was to construct a graph with a structure that supports the computation of process metrics, so the change information should be easy to obtain. Actually, it contains seven types of nodes. The *Project* node represents the repository of a project. Since we used GitHub, it has two attributes: the GitHub user and repository identifiers. With this node, one database can be used for multiple projects. It may be useful if we are dealing with cross-repository issue referencing, which is also one of the GitHub features. In bigger companies or on GitHub, developers usually contribute to multiple repositories. If we put these repositories into this database, then the developers are connected to multiple projects, hence we can compute with this property as well. The *User* node simply represents a developer. It has one attribute, namely the number of commits. This property is provided by the GitHub API. The *Issue* node represents a bug report from the issue tracker. It has two attributes, namely *opened* and *closed*. The former is the date of the bug report, while the latter is the date of closing the bug report or it is null if it is still open.

The *Commit* node represents a software version. It has three attributes, these being *hash*, *created* and *fix*. The first one is the unique hash of the commit. The second is the time stamp of the commit's creation. *Fix* is a Boolean property and it tells us whether this commit is a bug fix or not. A commit is treated as a fix if the commit message references a bug report. This connection is provided by the GitHub API and in the schema, it is represented as a *Referenced* edge between *Commit* and *Issue* nodes. A commit is made by one user, thus we connect the *Commit* nodes to the *User* nodes with an *Author* edge. Sometimes the commits do not have such an edge because the developer is removed from GitHub. The *Parent* edges of *Commit* nodes represent the relationship between consecutive commits. Two commits are connected if one of them is directly followed by the other. One commit may have multiple parents in the case of merge commits.

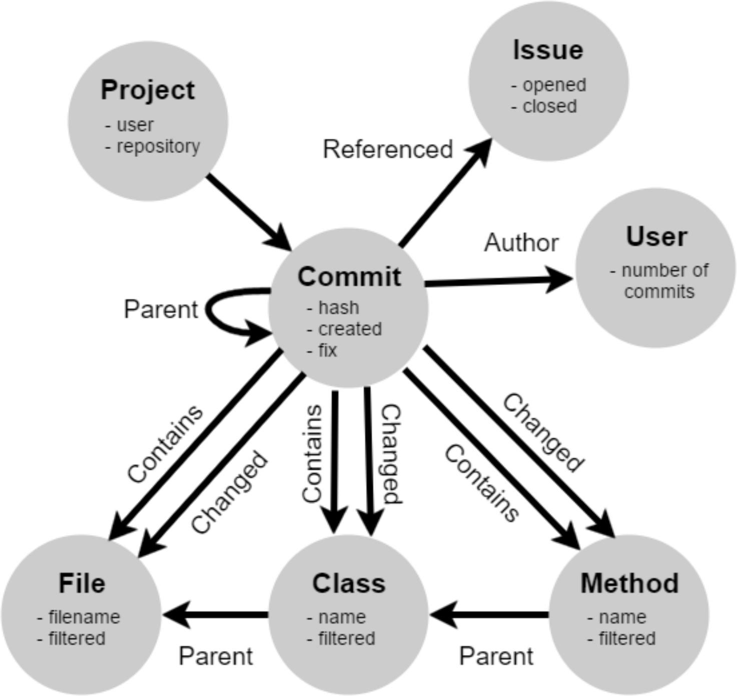

Figure 1: The Graph Database Schema

The bottom three nodes - *File*, *Class*, *Method* - represent the source code elements. The *Parent* edges between them is the containment relationship. Since we focused on the Java programming language, other containment relations are not possible. *Method* and *Class* nodes have a *name* attribute, which is the fully qualified name of the elements. The *File* nodes always have a *filename* attribute. The *filename* contains the full path of the file. In our previous study [7, 23], in order to avoid marking non-buggy test code as buggy, we filtered the test-related source code elements during the collection of bug information. This filtering is based on the file name and the qualified name. The *filtered* attribute of *File*, *Class*, and *Method* nodes indicates whether a certain file, class or method was filtered or not. The values of *name* and *filename* attributes are unique, and this means only one node is created for a given source element that lives across multiple software versions. This way, it is easy to get the changes of a file, a class or a method. This is a crucial feature of the database in terms of creating an efficient method. The *Contains* edge between commits and source elements is responsible for showing whether a given commit actually contains the specific element.

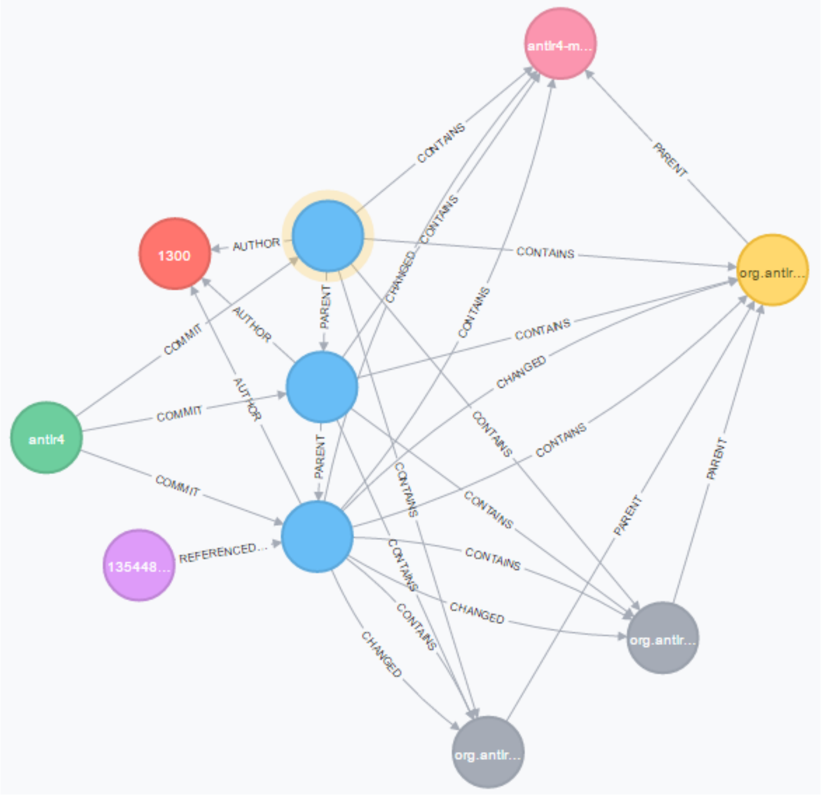

Figure 2: Example Graph (green: project, purple: issue, red: user, blue: commit, pink: file, yellow: class, grey: method)

The *Changed* edges between commits and the source elements indicate whether an element is changed during a commit. These edges have the following attributes: *added* - number of added lines, *deleted* - number of deleted lines and *modified* - number of modified lines. These values are computed from the commit patch file, which can also be got via the GitHub API. The patch file is based on files, hence for classes and methods additional mapping is required. For this task, we used the source position available from the static source code analysis. A patch file contains sections (deltas). A delta has a begin line number and an end line number. The mapping is carried out by checking whether a delta intersects the position of a source code element. From the patch file, we can get the begin and end line numbers of a change (delta) and from the SourceMeter output, we can get the source position of a source element in a form of row numbers. Now, let us consider a delta with line values 31-46 and a method with position 24-37. The first and last three lines of a delta are unchanged, so we can subtract them from the section. After this step, we get 34-43 as line values for the delta. The intersecting range is 34-37. This means that 4 lines of the method have changed. If we look at the original version of a delta, we can extract information about the type of the change. If the size of the original is zero, then the change is an addition. Conversely, if the size of the new part is zero, then it is a deletion. Otherwise it is a modification.

Now that we have a schema definition, we can proceed to the metric calculation. Figure 2 shows a small part of the graph database created for the ANTLR4 project (more details in Section 4). In this graph, we have at least 1 of each type of nodes and relationships. As an example, let us show how to compute a simple process metric called the *Number of Modifications* in this graph. The basis of the calculation is the highlighted commit (uppermost) and the source element is the upper right method. We have to look for *Commit* nodes that are created before the subject commit and have a *Changed* relationship with the selected *Method* node. We can use the *Parent* edge between commits, or the *created* attribute for selecting the past commits. Adding the *Changed* edge to the match condition leads to the desired commits, which in this example is the lowermost *Commit* node. With a simple aggregation (counting) we get the value for the computed process metric. As we mentioned earlier, Neo4j has a query language called *cypher*. With this language, it is easy to formulate these process metrics. For example,

```
match (n:METHOD{name:'...'})<-[:CONTAINS]-(c1:COMMIT{hash:'...'}),
(n)<-[:CHANGED]-(c2:COMMIT)
where c1.created >= c2.created
return n.name as name, count(c2) as ‘Number of Modifications‘
```

We will not go into details about the syntax of this query language. A detailed description is available on the official Neo4j website[8]. The other metrics can be formulated into a single query too, hence it is an easy way to compute them. Table 1 lists the implemented process metrics. Switching to the class level is simple, because all we need to do is change the node type in the query. To produce these metrics automatically for the selected project's selected version, we created a framework. Below, we will describe the overall picture of the framework.

An overview of the process is shown in Figure 3. The shape in the top left corner represents our data source, GitHub. The two connected elements are the first steps. These were partially described in our previous study [23]. Stated briefly, the project data is exported from the GitHub API and the source code versions are analyzed with SourceMeter during these steps. Next, the graph nodes and edges - according to the previously presented schema - are exported into CSV files. These files can be directly imported into a Neo4j graph database. The next task

[8] https://neo4j.com/developer/cypher-query-language/

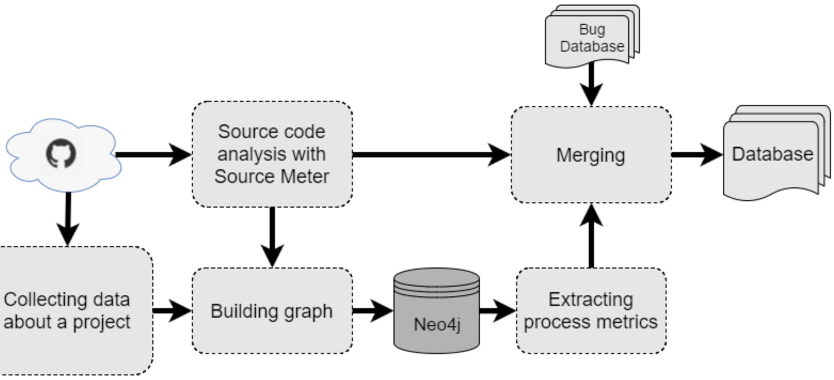
Bug Database
Source code analysis with Source Meter
Merging
Database
Collecting data about a project
Building graph
Neo4j
Extracting process metrics

Figure 3: Overview of processes involved

is to produce the required process metrics listed in Table 1. The selection of the implemented metrics is based on the data available to us. The *cypher* queries are executed and the results are saved in separate CSV files for each metric. The next and final step of the process is to merge the available CSV files so as to produce the desired database. SourceMeter also exports the source elements into CSV files with some essential properties, like qualified name, source position and static source code metrics. In our previous article [23], we published a bug database[9] that contains bug information for files and classes. In this database, bug numbers are collected from the known and previously fixed source code defects. For a given release version, the database contains the number of reported, but not yet fixed bugs for each element. The source code elements affected by bugs are determined from bug fixing modifications. From these data sets, we created databases that contain the source code elements (files, classes, methods), the static source code metrics and the process metrics. Also, the file and class level databases include the actual number of bugs taken from our earlier studies. Here, these databases are in CSV form. The first line contains the header information - as in the previously published data sets - extended with the process metric names. The final header element is the number of bugs. The rest of the lines are the source elements taken from the original database along with the associated process metrics. The method level databases have the same structure, but since we did not produce method level bug databases previously, the number of bugs column is missing.

The tools created are published as open-source projects in the following GitHub repository:
`https://github.com/sed-szeged/BugHunterToolchain`

[9] `http://www.inf.u-szeged.hu/~ferenc/papers/GitHubBugDataSet/`

Table 1: The list of implemented process metrics.

Age
Average Number of Added Lines
Average Number of Deleted Lines
Average Number of Elements Modified Together
Average Time Between Changes
Last Contributor Commits
Maximum Number of Added Lines
Maximum Number of Deleted
Maximum Number of Elements Modified Together
Number of Additions
Number of Contributor Changes
Number of Contributors
Number of Deletions
Number of Fixes
Number of Fixed in the Last Six Months
Number of Modifications
Number of Modifications in the Last Six Months
Number of Versions
Sum of Added Lines
Sum of Deleted Lines
Time Passed Since the Last Change
Weighted Age

4 Experimental Set-up

To demonstrate our method, we processed 5 of the Java projects we used in our previous study. Table 2 lists the selected projects with some basic statistics. The first column is the name of the systems, while the second column is the general domain of the projects. The next two columns contain the number of commits and the thousand lines of code. This statistics tells us that these projects are dissimilar regarding domain and size too. During the process, we had to analyze every single version of the systems to extract the change information. Although the analysis of individual versions was quick, the overall run time was quite high. For the Broadleaf Commerce project, the analysis of nearly ten thousand versions took around 200 hours. From a process perspective, it was just an initial step, and only needed to be executed once.

The next step was to build the graph databases. The total run time for all of the projects was around 6 hours. For a single version this time is negligible, thus the database can be extended efficiently with the new version. Table 3 gives statistics on the size of the graph databases. The first column shows the name of the project.

Table 2: The chosen Java projects.

Project	**Domain**	**Commits**	**kLOC**
ANTLR4	Language processing	3276	85
Broadleaf Commerce	E-commerce framework	9292	282
jUnit	Test framework	2053	36
MapDB	Database engine	1345	83
Titan	Database engine	3830	119

The next two columns are the number of nodes and the number of relationships (edges) in the graph, expressed in thousands. The next column contains the disk space occupied by the graph databases in Megabytes. The final column is the size of the results of source code analysis in Gigabytes. We can see that the graph is a compact way of storing the information about the project history and process metrics can be efficiently derived from it.

Table 3: The graph databases that we created.

Project	**kNodes**	**kEdges**	**Size of Graph (MB)**	**Size of Raw Data (GB)**
ANTLR4	24	13 069	484	18
Broadleaf Commerce	91	145 966	4 828	220
jUnit	14	7 457	300	9
MapDB	13	4 629	208	7
Titan	219	21 471	804	30

After setting up the database, we computed the process metrics for 25 release versions of the systems (5 each). The release versions were selected just like those in our previous study [23], namely at 6-monthly intervals. Due to smaller inactive periods in project development, it may happen that the bug numbers are zero in a given release version. In such cases, we dropped this release version, then the time interval between some of the versions was larger than six months.

Lastly, we constructed bug databases (at the file and class levels) for the selected 5 projects' 25 release versions with the computed process metrics. What is more, we created method-level databases - without bug information - which also include both product and process metrics.

5 Evaluation

We applied machine learning algorithms to our bug database in order to check whether it was suitable for bug prediction. In the preliminary step, similar to our previous study, we grouped the source elements into two classes based on the bug numbers. Source elements with zero bug numbers formed a non-defective class, while the others formed a defective class. The structure of the learning tables was the following: it contained a unique id for every instance; next, it contained the predictors (22 software process metrics); lastly it contained the label of the class as Boolean values (true - defective, false - non-defective). Separate learning tables were constructed for files and classes in each release version, hence we got 50 learning tables in total. The number of instances in a defective class was much smaller than the number of instances in a non-defective class. To avoid any distortion in the results, we applied random under sampling to the databases. This method helps to balance the number of positive and negative instances in the training set. To achieve more reliable results, we applied this method ten times to the data sets and computed an average.

We used Weka[10], the popular machine learning library to perform the training part. For the evaluation part, we used the same set of algorithms as in our earlier study [23]. Namely,

- NaiveBayes
- NaiveBayesMultinomial
- Logistic
- SGD
- SimpleLogistic
- SMO
- VotedPerceptron
- DecisionTable
- OneR
- PART
- J48 (C4.5)
- RandomForest
- RandomTree

[10] http://www.cs.waikato.ac.nz/~ml/weka/

We used 10-fold cross validation and we measured the performance with F-measure metrics that are defined by the following:

$precision = \frac{TP}{TP+FP}$

$recall = \frac{TP}{TP+FN}$

$F - measure = 2 \cdot \frac{precision \cdot recall}{precision + recall}$,

where TP(True Positive) is the number of instances that were predicted as defective and observed as defective, FP(False Positive) is the number of instances that were predicted as defective but observed as non-defective, FN(False Negative) is the number of instances that were predicted as non-defective but observed as defective.

Table 4: Average F-measure values at the class level.

Project	**#1**	**#2**	**#3**	**#4**	**#5**
ANTLR4	0.7235	0.5502	0.6771	0.7101	0.7765
Broadleaf C.	0.6772	0.6736	0.6812	0.6729	0.6923
MapDB	0.5626	0.6560	0.7179	0.7261	0.7426
jUnit	0.6989	0.6522	0.5949	0.6156	0.8127
Titan	0.5712	0.6231	0.6640	0.6543	0.7423

Table 5: Average F-measure values at the file level.

Project	**#1**	**#2**	**#3**	**#4**	**#5**
ANTLR4	0.7252	0.7548	0.6961	0.7334	0.6872
Broadleaf C.	0.6402	0.6759	0.6799	0.6969	0.6869
MapDB	0.5652	0.7879	0.6606	0.6930	0.8362
jUnit	0.7279	0.6102	0.7000	0.6792	0.5384
Titan	0.6082	0.7147	0.7108	0.7303	0.6924

The goal of our first investigation was to find out whether the created bug databases with the process metrics were suitable for bug prediction. We evaluated the 13 algorithms on all 25 release versions and only used process metrics as predictors. Our first observation was that the F-measure values at the class level generally improved with time except for the jUnit project. Table 4 shows the average F-measure values at the class level for all 25 versions. The first column contains the project names, while the next columns are the average results for each version in chronological order. The first is the earliest in time, the following is the next, and so on. From this table, we can see that the values increase slightly with time. One possible explanation for this is the nature of the metrics used. Most of the process metrics are based on temporal characteristics, hence these values may be more reliable with bigger time intervals. At the file level, things are not so straightforward. In the case of the ANTLR4 and jUnit projects, the values do not follow this trend, as shown in Table 5. This may be due to the varying size of the training sets, but as we cannot generalize this observation, more investigation is needed to learn the reason for it.

Table 6: Comparison of F-measure values at the class level.

Project	**Product metrics**	**Process metrics**
ANTLR v4	0.7179	0.6771
Broadleaf Commerce	0.7544	0.6812
MapDB	0.6999	0.6560
jUnit	0.7233	0.6156
Titan	0.7058	0.7423

Table 7: Comparison of F-measure values at the file level.

Project	**Product metrics**	**Process metrics**
ANTLR v4	0.7061	0.6961
Broadleaf Commerce	0.6955	0.6799
MapDB	0.6306	0.7879
jUnit	0.5600	0.6792
Titan	0.5730	0.6924

In our previous study [23], as we reported F-measure values from release versions that have the most bug entries, we will now compare the results for these versions. Tables 6 and 7 allow us to compare the average F-measure values got with the two different sets of predictors for these release versions. Our previous database at the file level contains some process metrics, hence we once again applied the machine learning algorithms without these metrics. Also, we repeated the learning process at the class level. From these values, we can see that the process metrics performed worse than the product metrics at the class level in 4 out of 5 cases. At the file level, process metrics performed better than the product metrics in 3 out of 5 cases. Despite process metrics not performing well in the same cases as product metrics, in other cases the F-measure values we got indicate that the former perform more robustly than the latter. As we mentioned earlier, process metrics perform better in the later versions than in the earlier ones. At the file level, the F-measure values vary more with product metrics and the best performing algorithms are different for each version. With the process metrics, more or less the same set of algorithms are at the top for all of the cases studies. We did not include every result, due to the large amount of algorithms and learning tables. The detailed F-measure tables can be found in the online appendix.

Table 8: F-measure values at the class level.

Algorithm	**ANTLR4**	**Broadleaf**	**MapDB**	**jUnit**	**Titan**	**AVG**
RandomForest	0.7328	0.7638	0.7483	0.6702	0.7997	0.7430
DecisionTable	0.6704	0.7147	0.6719	0.6712	0.7742	0.7005
SMO	0.7059	0.7015	0.6682	0.6232	0.7788	0.6955
OneR	0.7101	0.6982	0.6065	0.6770	0.7597	0.6903
SimpleLogistic	0.6970	0.7059	0.6163	0.6345	0.7878	0.6883
PART	0.6730	0.7056	0.6755	0.6221	0.7468	0.6846
J48	0.6971	0.6953	0.6618	0.6077	0.7567	0.6837
SGD	0.6196	0.7039	0.6838	0.6125	0.7832	0.6806
RandomTree	0.6530	0.6861	0.6547	0.6365	0.7540	0.6769
Logistic	0.5827	0.6885	0.6459	0.6006	0.7441	0.6523
NaiveBayes	0.7106	0.5846	0.6868	0.6457	0.5672	0.6390
NaiveBayesMultinomial	0.7065	0.5920	0.6042	0.5349	0.6689	0.6213
VotedPerceptron	0.6432	0.6159	0.6040	0.4663	0.7290	0.6117

In Table 8, we list the F-measure values for the versions with the most bug entries. The first column contains the machine learning algorithm names that we used. The subsequent columns contain the resulting F-measure values for each project. The last column is the average F-measure value over projects. The table is ordered by the average value in decreasing order. Here, we notice that tree-, rule-, and function-based algorithms performed the best. The highest average F-measure value is 0.7430, while the lowest is 0.6117. The overall highest F-measure value in these versions is 0.7997 and it was achieved by the Titan project. The best performing algorithm is the *RandomForest* method. We should add that process metrics did not preform the best in these release versions.

Table 9 shows the resulting F-measures for the versions with the most bug entries. The structure of the table is the same as that for Table 8. The same set of algorithms perform the best in these cases as well. The performance of the Bayesian methods varies for each version, hence we cannot say that using these metrics, they are the best to predict bugs. The best results are over 0.8 (MapDB project) and the worst result is 0.5278; however, on average the F-measure values are around 0.7. The highest average F-measure value is 0.7448, while the lowest is 0.6622. In summary, the results obtained appear to indicate that the databases with the computed process metrics are suitable for bug prediction purposes and the best performing algorithms are the *RandomForest* and *DecisionTable* methods.

Next, we examined whether there were any relationships among the metrics. Since our databases contain both product and process metrics for classes and methods, we computed correlations among these values. As the results are similar to each other between the versions and listing the correlations for all 25 versions would take up too much space, we will only present the results for a single version. Instead

Table 9: F-measure values at the file level.

Algorithm	**ANTLR4**	**Broadleaf**	**MapDB**	**jUnit**	**Titan**	**AVG**
RandomForest	0.7804	0.7328	0.8180	0.6659	0.7271	0.7448
DecisionTable	0.7299	0.7152	0.8143	0.7061	0.7103	0.7352
SimpleLogistic	0.7234	0.6910	0.7869	0.7071	0.7094	0.7236
SGD	0.7033	0.7025	0.8036	0.6857	0.7159	0.7222
SMO	0.7192	0.7055	0.8085	0.6893	0.6796	0.7205
NaiveBayesMultinomial	0.6920	0.6342	0.8152	0.6732	0.7533	0.7136
OneR	0.6911	0.6642	0.7786	0.7371	0.6781	0.7098
J48	0.6553	0.6994	0.7983	0.6614	0.7304	0.7090
PART	0.6957	0.6907	0.7879	0.6552	0.6575	0.6974
RandomTree	0.6826	0.6740	0.7261	0.6933	0.6896	0.6931
VotedPerceptron	0.6267	0.6340	0.8019	0.6149	0.7406	0.6836
Logistic	0.6765	0.6950	0.6699	0.6649	0.6821	0.6777
NaiveBayes	0.6732	0.6008	0.8334	0.6757	0.5278	0.6622

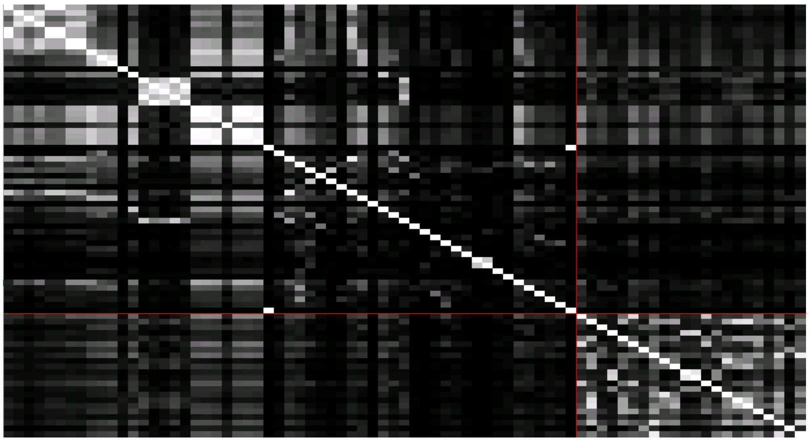

Figure 4: Correlation of method metrics

of including the correlation matrices that have over a hundred rows and columns, we illustrated these with colored tables (Figures 4, 5 and 6). The black cells denote the low absolute value of the correlation (close to zero), while the white cells denote the high absolute value of the correlation (near one or minus one). Figure 4 shows the correlation of method metrics tested. The product metrics (including rule violations) are separated from the process metrics by a red line. From this image (bottom right quarter), we can see that the process metrics correlate more with each other than with product metrics. There are some correlation values around 0.4-0.5 between a few size-based product metrics (Lines of Code, Number of State-

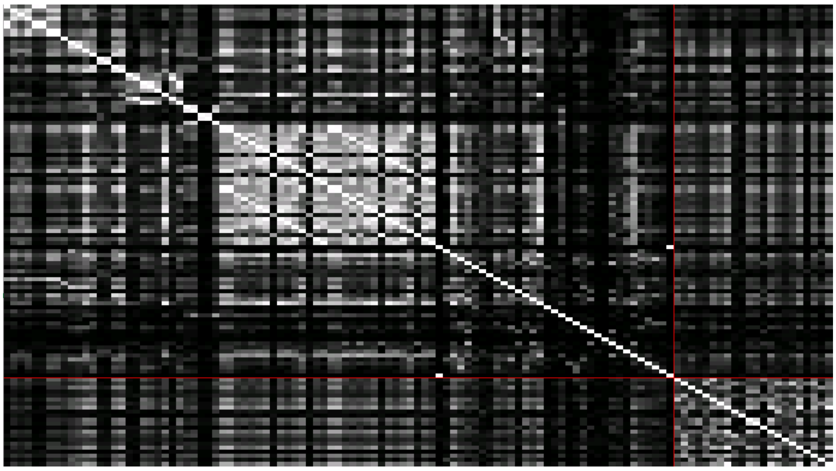

Figure 5: Correlation of class metrics

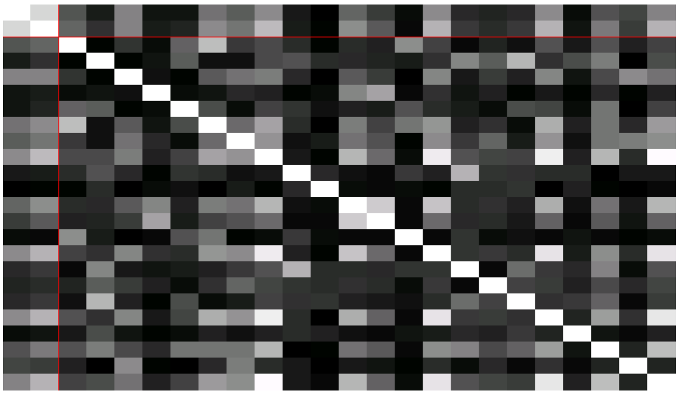

Figure 6: Correlation of file metrics

ments) and process metrics (Number of Added Lines, Number of Modifications), but in general there are no high correlation values.

If we look at the correlation results between any two class metrics in Figure 5, we notice that the relation between process and product metrics is a little clearer. Nevertheless, the correlation is still noticeably higher between the same type of metrics. At the file level (Figure 6), as the database contains only two product metrics, we cannot draw any conclusions from it.

From the correlation results presented above, we can say that process metrics are of a different nature from product metrics. These metrics characterize the source code elements from a different perspective than product metrics do. The full correlation matrices, the results of the evaluation and the databases created

are accessible as an online appendix at the following URL:
`http://www.inf.u-szeged.hu/~pgyimesi/papers/ActaCybernetica2016/`

6 Conclusions and Future Work

In this study, we presented a method that efficiently computes software process metrics in a graph database. Also, we made an implementation available on GitHub as an open-source project that is capable of computing 22 process metrics. We selected 5 Java projects and with our implementation, we processed these systems and produced databases for 25 release versions that were selected from an earlier study. The databases created contain the implemented process metrics for files, classes and methods. Afterwards, we employed our previously published bug databases (at the file and class levels) and extended them with process metrics; then we applied 13 machine learning algorithms on them to investigate whether the database was suitable for bug prediction purposes and we achieved promising results. Based on the F-measure values, we found that tree- and rule-based methods perform the best and, in particular, the *RandomForest* method performed well in every case.

In the future, we intend to implement more process metrics and experiment with new ones. We also plan to extend the list of processed systems. Furthermore, we would like to set up a method-level bug database and to evaluate the bug prediction capability of method-level process metrics.

7 Acknowledgements

I would like to express my gratitude to my supervisor Dr. Rudolf Ferenc for his useful comments and remarks. He also helped clarify certain points and issues during the study.

References

[1] Bernstein, Abraham, Ekanayake, Jayalath, and Pinzger, Martin. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18. ACM, 2007.

[2] Bhattacharya, Pamela, Iliofotou, Marios, Neamtiu, Iulian, and Faloutsos, Michalis. Graph-based analysis and prediction for software evolution. In *Proceedings of the 34th International Conference on Software Engineering*, pages 419–429. IEEE Press, 2012.

[3] Catal, Cagatay. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.

[4] D'Ambros, Marco, Lanza, Michele, and Robbes, Romain. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41. IEEE, 2010.

[5] Fischer, Michael, Pinzger, Martin, and Gall, Harald. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.

[6] Graves, Todd L, Karr, Alan F, Marron, James S, and Siy, Harvey. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[7] Gyimesi, Péter, Gyimesi, Gábor, Tóth, Zoltán, and Ferenc, Rudolf. Characterization of source code defects by data mining conducted on GitHub. In *International Conference on Computational Science and Its Applications*, pages 47–62. Springer, 2015.

[8] Hassan, Ahmed E. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[9] Jureczko, Marian and Madeyski, Lech. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 9. ACM, 2010.

[10] Krishnan, Sandeep, Strasburg, Chris, Lutz, Robyn R, and Goševa-Popstojanova, Katerina. Are change metrics good predictors for an evolving software product line? In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, page 7. ACM, 2011.

[11] Madeyski, Lech and Jureczko, Marian. Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal*, 23(3):393–422, 2015.

[12] Malhotra, Ruchika. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.

[13] Menzies, T., Krishna, R., and Pryor, D. The Promise Repository of Empirical Software Engineering Data, 2015. http://openscience.us/repo. North Carolina State University, Department of Computer Science.

[14] Menzies, Tim, Milton, Zach, Turhan, Burak, Cukic, Bojan, Jiang, Yue, and Bener, Ayşe. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[15] Moser, Raimund, Pedrycz, Witold, and Succi, Giancarlo. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 309–311. ACM, 2008.

[16] Moser, Raimund, Pedrycz, Witold, and Succi, Giancarlo. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190. IEEE, 2008.

[17] Munson, John C and Elbaum, Sebastian G. Code churn: A measure for estimating the impact of code change. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 24–31. IEEE, 1998.

[18] Nagappan, Nachiappan and Ball, Thomas. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292. IEEE, 2005.

[19] Radjenović, Danijel, Heričko, Marjan, Torkar, Richard, and Živkovič, Aleš. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[20] Rahman, Foyzur and Devanbu, Premkumar. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[21] Ratzinger, Jacek, Pinzger, Martin, and Gall, Harald. EQ-Mine: Predicting short-term defects for software evolution. In *International Conference on Fundamental Approaches to Software Engineering*, pages 12–26. Springer, 2007.

[22] Shihab, Emad, Jiang, Zhen Ming, Ibrahim, Walid M, Adams, Bram, and Hassan, Ahmed E. Understanding the impact of code and process metrics on post-release defects: a case study on the Eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 4. ACM, 2010.

[23] Tóth, Zoltán, Gyimesi, Péter, and Ferenc, Rudolf. A Public Bug Database of GitHub Projects and Its Application in Bug Prediction. In *International Conference on Computational Science and Its Applications*, pages 625–638. Springer, 2016.

[24] Zimmermann, Thomas, Premraj, Rahul, and Zeller, Andreas. Predicting defects for Eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE, 2007.