A Public Unified Bug Dataset for Java*

Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi,
István Siket
Department of Software Engineering,
University of Szeged, Hungary
{ferenc,zizo,lgergely,siket}@inf.u-szeged.hu

Tibor Gyimóthy
MTA-SZTE Research Group on
Artificial Intelligence, Szeged, Hungary
gyimothy@inf.u-szeged.hu

ABSTRACT

Background: Bug datasets have been created and used by many researchers to build bug prediction models.
Aims: In this work we collected existing public bug datasets and unified their contents.
Method: We considered 5 public datasets which adhered to all of our criteria. We also downloaded the corresponding source code for each system in the datasets and performed their source code analysis to obtain a common set of source code metrics. This way we produced a unified bug dataset at class and file level that is suitable for further research (e.g. to be used in the building of new bug prediction models). Furthermore, we compared the metric definitions and values of the different bug datasets.
Results: We found that (i) the same metric abbreviation can have different definitions or metrics calculated in the same way can have different names, (ii) in some cases different tools give different values even if the metric definitions coincide because (iii) one tool works on source code while the other calculates metrics on bytecode, or (iv) in several cases the downloaded source code contained more files which influenced the afferent metric values significantly.
Conclusions: Apart from all these imprecisions, we think that having a common metric set can help in building better bug prediction models and deducing more general conclusions. We made the unified dataset publicly available for everyone. By using a public dataset as an input for different bug prediction related investigations, researchers can make their studies reproducible, thus able to be validated and verified.

CCS CONCEPTS

• **Software and its engineering** → *Software reliability*;

KEYWORDS

Bug dataset, code metrics, static code analysis

ACM Reference Format:
Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket and Tibor Gyimóthy. 2018. A Public Unified Bug Dataset for Java. In *The 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'18), October 10, 2018, Oulu, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3273934.3273936

1 INTRODUCTION

Finding and eliminating bugs in software systems has always been one of the most important issues in software engineering. Software testing is often limited because of the given resources, thus a more focused resource allocation should be applied. Bug localization is a process when we want to find the exact locations of the occurring bugs. Bug localization is a crucial and very expensive part of software engineering, therefore many researches have examined this topic and several different approaches were proposed which tried to reduce costs and to create more powerful methods [38].

Bug or defect prediction is a process by which we try to learn from mistakes committed in the past and build a prediction model to leverage the location and amount of future bugs. Many research papers were published on bug prediction, which introduced new approaches that aimed to achieve better precision values [15, 36, 39, 41]. Unfortunately, a reported bug is rarely associated with the source code lines which caused it or with the corresponding source code elements (e.g. classes, methods) containing it. Therefore, to carry out such experiments, bugs have to be associated with source code (or with classes or methods) which itself is a difficult task (this is where bug localization steps in). It is necessary to use a version control system and a bug tracking system properly during the development process and even in this case it is still challenging to associate bugs with the problematic source code locations.

Although several algorithms were published on how to associate a reported bug with the relevant, corresponding defective source code [8, 9, 37], only few such bug association experiments were carried out. Furthermore, not all of these studies published the bug dataset or even if they did, closed source systems were used which limits the verifiability and reusability of the bug dataset. In spite of these facts, several bug datasets (containing information about open-source software systems) were published and made publicly available for further investigations or to replicate previous approaches [30, 35]. They are very popular and for example the NASA and the Eclipse Bug Dataset [42] were used in numerous experiments [12, 19, 31].

The main advantage of these bug datasets is that if someone wants to create a new bug prediction model or validate an existing one, it is enough to use a previously created bug dataset instead of building a new one, what is very resource consuming. It is common in these bug datasets that all of them store some specific information about the bugs such as the containing source code element(s) with their source code metrics or any additional bug related information. Since different bug prediction approaches try to use various sources

*Produces the permission block, and copyright information

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE'18, October 10, 2018, Oulu, Finland
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6593-2/18/10…$15.00
https://doi.org/10.1145/3273934.3273936

of information as predictors (independent variables), different bug datasets are constructed. Defect prediction approaches and hereby bug datasets can be categorized into larger groups based on the captured characteristics [11]:

- Datasets using process metrics [25, 26].
- Datasets using source code metrics [5, 7, 13, 33].
- Datasets using previous defects [20, 27].

Different bug prediction approaches use various public or private bug datasets. Although these datasets seem very similar, they are often very different in some aspects that is also true within the categories mentioned above. In this study, we gather datasets that can be found, but we will focus on datasets that use static source code metrics. Since this category itself has grown so immense, it is worth studying it as a separate unit. This category has also many dissimilarities between the existing datasets including the granularity of the data (source code elements can be files, classes, or methods, depending on the purpose of the given research or on the capabilities of the tool used to extract data) and the representation of element names (different tools may use different notations). For the same reason, the set of metrics can be different as well. Even if the name or the abbreviation of a metric calculated by different tools is the same it can have different meanings because it can be defined or calculated in a slightly different way. The bug related information given for a source code element can also be contrasting. An element can be labeled whether it contains a bug, but it can also show how many bugs are related to that given source code element. From the information content point of view it is less important but not negligible that the format of the files containing the data can be CSV (Comma Separated Values), XML, or ARFF (which is the input format of Weka [14]) and these datasets can be found on different places on the Internet.

The constructed dataset can represent a good input for machine learning algorithms to build a prediction model [4, 22, 24]. Some researchers argued that the used dataset is not as important as the applied machine learning algorithm [23]. However, the selection of software metrics to build a prediction model from can severely influence the accuracy and the complexity of the model [28].

Finally, there is usually a lack of information about the reliability and no specification is given on how a given dataset should be used. On the other hand, it would be a difficult task and would require lots of efforts to validate the metric values and the number of bugs, especially for systems where the source code is not available for the public. In spite of all these drawbacks, researchers should consider using these bug datasets first and not creating new, specialized ones if it is possible. Necessarily, they can build new ones if needed, but first they should be attentive and try to use public datasets and further improve them. Our contributions can be listed as follows:

- Collection of the bug datasets and source code.
- Unification of the collected bug datasets.
- Extension of the metrics suites.
- Assessment of the datasets.
- Making the results publicly available.

2 DATA COLLECTION

In this section, we give a detailed overview about how we collected the datasets that includes a literature review, how we analyzed the datasets, and investigated what characteristics they have.

Data collection can be divided into two parts, the first part is the collection and the evaluation of the literature review papers in the subject area. In the second part we use the literature review papers and the case studies presented in them to collect the bug datasets themselves with the corresponding source code.

Collecting literature review papers. Starting from the early 70's [18, 29] a large number of studies was introduced in connection with software faults. According to Yu et al. [40] 729 studies were published until 2005 and 1564 until 2015 on bug prediction (the number of studies has been doubled in 10 years). From time to time the enormous number of new publications in the topic of software faults made it unavoidable to collect the most important advances in literature review papers. By using the existing literature review papers, we were able to focus on the empirical aspects of the collected datasets.

Collecting bug datasets. We went through the union of the references used in the review studies and filtered out the relevant papers based on keywords, title, abstract and the introduction. Then we collected all available information about the used bug datasets located in the remained set of scientific papers. We took into consideration the following properties:

- Basic information (authors, title, date, publisher).
- Accessibility of the bug dataset (public, non public, partially public).
- Availability of the source code.

The latter two are extremely important when investigating the datasets since we cannot construct a unified dataset without obtaining the appropriate underlying data.

2.1 Public Datasets

As we collected the literature review papers we created a list of the found datasets and repositories. Furthermore, we included a few additional papers which were published recently, so they were not included in any previous literature review. We considered the following list to check whether a dataset meets our requirements:

- the dataset is publicly available,
- source code is accessible for the included systems,
- bug information is provided,
- bugs are associated with the relevant source code elements,
- included projects were written in Java,
- the dataset provides bug information at file/class level, and
- the source code element names are provided and unambiguous (the referenced source code is clearly identifiable).

If any condition is missing then we had to exclude the subject system or the whole dataset from the study because they cannot be included in the unified bug dataset. The list of found public datasets we could use for our purposes is the following:

- PROMISE [3]
- Eclipse Bug Dataset [42]
- Bug Prediction Dataset [10]
- Bugcatchers Bug Dataset [16]
- GitHub Bug Dataset [34]

In the following we will describe these datasets in more details and investigate each dataset's peculiarities and we will also look for common characteristics.

2.1.1 PROMISE. PROMISE (a.k.a. tera-PROMISE) [3] is one of the largest research data repositories in software engineering. It is a collection of many different datasets, including the NASA dataset which was used by numerous studies in the past. Besides many other categories like code analysis, testing, software maintenance, it also provides a category for defects.

Although some datasets were excluded, since they did not match all the requirements, we kept the remained set and used it in our investigations. We will use the name 'PROMISE' in the remainder of the paper to reference the subset of datasets under the PROMISE repository that fulfilled all the prerequisites.

2.1.2 Eclipse Bug Dataset. Zimmerman et al. [42] mapped defects from the bug database of Eclipse 2.0, 2.1, and 3.0. The resulting data set lists the number of pre- and post-release defects on the granularity of files and packages that were collected from the BUGZILLA bug tracking system. They collected static code features using the built-in Java parser of Eclipse. They calculated some features at a finer granularity, these were aggregated taking average, total, and maximum values of the metrics. Data is publicly available and was used in many studies since then.

2.1.3 Bug Prediction Dataset. D'Ambros et al. [10] presented a publicly available dataset consisting of five open-source software systems. They also extended the source code metrics with change metrics, which according to their findings, could improve the performance of the fault prediction methods.

2.1.4 Bugcatchers Bug Dataset. Hall et al. [16] introduced the Bugcatchers Bug Dataset which contains bug information for Eclipse, ArgoUML, and some Apache software. They investigated the relationship between faults and code smells. Their findings suggest that some smells do indicate fault-prone code in some circumstances but the effect that these smells have on faults is small.

2.1.5 GitHub Bug Dataset. Tóth et al. selected 15 Java systems from GitHub and constructed a bug dataset at class and file level [34]. This dataset was employed as an input for 13 different machine learning algorithms to investigate which algorithm family performs the best in bug prediction. They included many static source code metrics in the dataset and used these measurements as independent variables in the machine learning process.

Table 1 summarizes some basic size statistics about the aforementioned datasets. We used the cloc (https://www.npmjs.com/package/cloc) program to measure the Lines of Code. We only considered Java files and we did not consider blank lines.

Table 1: Basic statistics about the public bug datasets

Dataset	Systems	Versions	Lines of Code	System Name
PROMISE	14	45	2,805,253	Ant, Camel, Ckjm, Forrest, Ivy, JEdit, Log4J, Lucene, PBeans, Poi, Synapse, Velocity, Xalan, Xerces
Eclipse Bug Dataset	1	3	3,087,826	Eclipse
Bug Prediction Dataset	5	5	1,171,220	Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, Lucene, Mylyn
Bugcatchers Bug Dataset	3	3	1,833,876	Apache Commons, ArgoUML, Eclipse JDT Core
GitHub Bug Dataset	15	105	1,707,446	Android Universal Image Loader, Antlr 4, Broadleaf Commerce, Ceylon IDE Eclipse Plugin, Elasticsearch, Hazelcast, JUnit, MapDB, mcMMO, MCT, Neo4J, Netty, OrientDB, Oryx, Titan

3 DATA PROCESSING

Although the found public datasets have similarities (e.g. containing source code metrics and bug information), they are very inhomogeneous. For example, they contain different metrics which were calculated with different tools and for different kinds of code elements, and the file formats are different as well, therefore it is very difficult to use these datasets together. Consequently, our aim was to transform them into a unified format and to extend them with such source code metrics that are calculated with the same tool for each system. In this section we will describe the steps we performed to produce the unified bug dataset.

First, we transformed the existing datasets to a common format. This means that if a bug dataset for a system consists of separate files we conflated them into one file. Next, we changed the *CSV* separator in each file to *comma (,)* and renamed the number of bug column in each dataset to *bug* and the source code element column name to *filepath* or *classname* depending on the granularity of the dataset. Finally, we transformed the source code element identifier to the standard form (e.g. *org.apache.tools.ant.AntClassLoader*).

3.1 Metrics Calculation

The bug datasets contain different kinds of metric sets which were calculated with different tools, therefore even if the same metric name appears in two or more different datasets we cannot be sure they mean exactly the same metric. To eliminate this deficiency we analyzed all the systems with the same tool. For this purpose we used the free and open-source *OpenStaticAnalyzer* [1] static source code analyzer tool that is able to analyze Java systems (among other languages). It calculates more than 50 different kinds (size, complexity, coupling, cohesion, inheritance, and documentation) of source code metrics for packages and class-level elements, about 30 metrics for methods, and a few ones for files. OpenStaticAnalyzer detects code duplications (Type-2 clones) as well, and calculates code duplication metrics for packages, classes, and methods. OpenStaticAnalyzer has two different kinds of textual outputs: the first one is an XML file that contains among others the whole structure of the source code (files, packages, classes, methods), their relationships and the metric values for each element (e.g. file, class, method). The other output format is CSV. Since different elements have different metrics, therefore there is one CSV file for each kind of elements.

For calculating the new metric values we needed the source code itself. Since all datasets belonged to a release version of a given software, therefore if the software was open-source and the given release version was still available we could manage to download and analyze it. This way, for each system we obtained two results: one from the downloaded bug datasets and one from the OpenStaticAnalyzer analysis.

3.2 Dataset Unification

We merged these two results into one by using the "unique identifiers" of the elements (Java standard names at class level and paths at file level). More precisely, the basis of the unified dataset was our source code analysis result and it was extended with the data of the given bug dataset. This means that we went through all elements of the bug dataset and if the "unique identifier" of an element was found in our analysis result then these two elements were conjugated (paired the original dataset entry with the one found in the result of *OSA*), otherwise it was left out of the unified dataset. Table 2 shows the results of this merging process: column *OSA* shows how many elements OpenStaticAnalyzer found in the analyzed systems, column *Bug dataset* presents the number of elements originally in the datasets and column *Dropped* tells us how many elements of the bug datasets could not be conjugated and so they were left out from the unified dataset. Although these numbers are very good we had to "modify" a few systems to achieve this but there were cases we simply could not solve the problem.

The details of the source code modifications and main reasons of the dropped elements were the following:

Camel 1.2 In the *org.apache.commons.logging* there were 13 classes in the original dataset that we did not find in the source code. Besides, there were 5 *package-info* classes (in different packages) which were not real Java classes because their name would not be a valid Java class name. There were 5 *package-info.java* files in the system but these files did not contain any Java class therefore OpenStaticAnalyzer did not find such classes, what is correct.

Camel 1.4 Besides the 7 *package-info.java* files, the original dataset contained information about 24 Scala files which were not Java source files, therefore OpenStaticAnalyzer did not analyze them.

Camel 1.6 There were 8 *package-info.java* and 30 Scala files.

Ckjm 1.8 There was a class in the original dataset that did not exist in version 1.8.

Forrest-0.8 There were two different classes which appeared twice in the source code, therefore we deleted the 2 copies from the *etc/test-whitespace* subdirectory.

Log4j In log4j, there was a *contribs* directory which contains the source code of different contributors. These files were put into the appropriate sub-directories as well (where they belonged according to their packages), what means that they occurred twice in the analysis and this prevented their merging. Therefore, in these cases we analyzed only those files that were in their appropriate subdirectories and excluded the files found in the *contribs* directory.

Lucene In all three versions there was an *org.apache.lucene.search .RemoteSearchable_Stub* class in the original dataset that did not exist in the source code.

Velocity In versions 1.5 and 1.6 there were two *org.apache.velocity. app.event.implement.EscapeReference* classes in the source code, therefore it was impossible to conjugate them by using their "unique identifiers" only.

Xerces 1.4.4 Although the name of the original dataset and the corresponding publication state that this is the result of Xerces 1.4.4 analysis we found that 256 out of the 588 elements did not exist in that version. We examined a few previous and following versions as well and it turned out that the dataset is much closer to 2.0.0 than to 1.4.4 because only 42 elements cannot be conjugated with the analysis result of 2.0.0. Although version 2.0.0 was still not matched perfectly, we did not find a "closer version" therefore in this case we used Xerces 2.0.0.

Eclipse JDT Core 3.4 There were lots of classes which appeared twice in the source code: once in the "code" and once in the "test" directory, therefore we deleted the test directory.

Eclipse PDE UI 3.4.1 The missing 6 classes were not found in its source code.

Table 2: Merging results

Dataset	Name	Granularity	OSA	Bug dataset	Dropped
PROMISE	Ant 1.3	class	530	125	0
	Ant 1.4	class	602	178	0
	Ant 1.5	class	945	293	0
	Ant 1.6	class	1,262	351	0
	Ant 1.7	class	1,576	745	0
	Camel 1.0	class	734	339	0
	Camel 1.2	class	1,348	608	13 (+5)
	Camel 1.4	class	2,339	872	0 (+31)
	Camel 1.6	class	3,174	965	0 (+38)
	Ckjm 1.8	class	9	10	1
	Forrest 0.6	class	159	6	0
	Forrest 0.7	class	76	29	0
	Forrest 0.8	class	53	32	0
	Ivy 1.4	class	421	241	0
	Ivy 2.0	class	637	352	0
	JEdit 3.2	class	552	272	0
	JEdit 4.0	class	647	306	0
	JEdit 4.1	class	722	312	0
	JEdit 4.2	class	888	367	0
	JEdit 4.3	class	1,181	492	0
	Log4J 1.0	class	180	135	0
	Log4J 1.1	class	217	109	0
	Log4J 1.2	class	410	205	0
	Lucene 2.0	class	758	195	1
	Lucene 2.2	class	1,394	247	1
	Lucene 2.4	class	1,522	340	1
	Pbeans 1	class	38	26	0
	Pbeans 2	class	77	51	0
	Poi 1.5	class	472	237	0
	Poi 2.0	class	667	314	0
	Poi 2.5	class	780	385	0
	Poi 3.0	class	1,508	442	0
	Synapse 1.0	class	319	157	0
	Synapse 1.1	class	491	222	0
	Synapse 1.2	class	618	256	0
	Velocity 1.4	class	275	196	0
	Velocity 1.5	class	377	214	1
	Velocity 1.6	class	458	229	1
	Xalan 2.4	class	906	723	0
	Xalan 2.5	class	992	803	0
	Xalan 2.6	class	1,217	885	0
	Xalan 2.7	class	1,249	909	0
	Xerces 1.2	class	564	440	0
	Xerces 1.3	class	596	453	0
	Xerces 1.4	class	782	588	42
Eclipse Bug Dataset	Eclipse 2.0	file	6,751	6,729	0
	Eclipse 2.1	file	7,909	7,888	0
	Eclipse 3.0	file	10,635	10,593	0
Bug Prediction Dataset	Eclipse JDT Core 3.4	class	2,486	997	0
	Eclipse PDE UI 3.4.1	class	3,382	1,497	6
	Equinox 3.4	class	742	324	5
	Lucene 2.4	class	1,522	691	21
	Mylyn 3.1	class	3,238	1,862	457
Bugcatchers Bug Dataset	Apache Commons	file	491	191	0
	ArgoUML 0.26 Beta	file	1,752	1,582	3
	Eclipse JDT Core 3.1	file	12,300	560	25
GitHub Bug Dataset	Android U. I. L. 1.7.0	class	84	73	0
	ANTLR v4 4.2	class	525	479	0
	Elasticsearch 0.90.11	class	6,480	5,908	0
	jUnit 4.9	class	770	731	0
	MapDB 0.9.6	class	348	331	0
	mcMMO 1.4.06	class	329	301	0
	MCT 1.7b1	class	2,050	1,887	0
	Neo4j 1.9.7	class	6,705	5,899	0
	Netty 3.6.3	class	1,300	1,143	0
	OrientDB 1.6.2	class	2,098	1,847	0
	Oryx	class	562	533	0
	Titan 0.5.1	class	1,770	1,468	0
	Eclipse p. for Ceylon 1.1.0	class	1,651	1,610	0
	Hazelcast 3.3	class	3,765	3,412	0
	Broadleaf C. 3.0.10	class	2,094	1,593	0
	Android U. I. L. 1.7.0	file	63	63	0
	ANTLR v4 4.2	file	411	411	0
	Elasticsearch 0.90.11	file	3,540	3,035	0
	jUnit 4.9	file	308	308	0
	MapDB 0.9.6	file	137	137	0
	mcMMO 1.4.06	file	267	267	0
	MCT 1.7b1	file	1,064	413	0
	Neo4j 1.9.7	file	3,291	3,278	0
	Netty 3.6.3	file	914	913	0
	OrientDB 1.6.2	file	1,503	1,503	0
	Oryx	file	443	280	0
	Titan 0.5.1	file	981	975	0
	Ceylon for Eclipse 1.1.0	file	699	699	0
	Hazelcast 3.3	file	2,228	2,228	0
	Broadleaf C. 3.0.10	file	1,843	1,719	0
Sum	All	class	76,623	48,242	624
	All	file	57,530	43,772	28
	All	class+file	134,153	92,014	652

Equinox 3.4 Three classes could not be conjugated because they did not have a unique name (there are more classes with the same name in the system) while two classes were not found in the system.

Lucene 2.4 (BPD) 21 classes from the original dataset were not present in the source code of the analyzed system.

Mylyn 3.1 457 classes were missing from our analysis that were in the original dataset, therefore we downloaded different versions of Mylyn but still could not find the matching source code. We could not achieve better result without knowing the proper version.
ArgoUML 0.26 Beta There were 3 classes in the original dataset that did not exist in the source code.
Eclipse JDT Core 3.1 There were 25 classes that did not exist in the analyzed system.
GitHub Bug Dataset The class level bug datasets contained elements having the same "unique identifier" so this information was not enough to conjugate them. Luckily, the paths of the elements were also available and we used them as well, therefore all elements could be conjugated. The GitHub Bug Dataset consists of 15 systems and the authors constructed multiple datasets for release versions that ends up in 210 different data files, which is way too much to present in the table. They performed a machine learning step on the versions that contain the most bugs, so we also decided to select these release versions and present the characteristics of these release versions for each system they used in their study. We will use these versions of the systems in the remaining of the paper.

As a result of this process we obtained a unified bug dataset which contains all of the public datasets in a unified format, furthermore they were extended with the same set of metrics provided by the OpenStaticAnalyzer tool. The last three lines of Table 2 show that only 1.29% (624 out of 48,242) of the classes and 0.06% (28 out of 43,772) of the files could not be conjugated, which means that only 0.71% (652 out of 92,014) of the elements was left out from the unified dataset.

In many cases the analysis results of OpenStaticAnalyzer contained more elements than the original datasets. Since we did not know how the bug datasets were produced we could not give an exact explanation for the differences but we list some possible causes:

- In some cases we could not find the proper source code for the given system (e.g. Xerces 1.4.4 or Mylyn) so two different but close versions of the same system might be conjugated.
- OpenStaticAnalyzer takes into account nested, local, and anonymous classes while some datasets simply associated Java classes with files.

4 ORIGINAL AND EXTENDED METRICS SUITES

In this section we present the metrics proposed by each dataset. Additionally, we will show a metrics suite that is used by the unified dataset we have constructed.

4.1 PROMISE

The authors [3] calculated the metrics of the PROMISE dataset with the tool called *ckjm.* All metrics, except McCabe's Cyclomatic Complexity (CC), are *class* level metrics. Besides the CK metrics they also calculated some additional metrics shown in Table 3.

Table 3: Metrics used in PROMISE dataset

Name	Abbr.
Weighted methods per class	WMC
Depth of Inheritance Tree	DIT
Number of Children	NOC
Coupling between object classes	CBO
Response for a Class	RFC
Lack of cohesion in methods	LCOM
Afferent couplings	Ca
Efferent couplings	Ce
Number of Public Methods	NPM
Lack of cohesion in methods (by Henderson-Sellers)	LCOM3
Lines of Code	LOC
Data Access Metric	DAM
Measure of Aggregation	MOA
Measure of Functional Abstraction	MFA
Cohesion Among Methods of Class	CAM
Inheritance Coupling	IC
Coupling Between Methods	CBM
Average Method Complexity	AMC
McCabe's cyclomatic complexity	CC
Maximum McCabe's cyclomatic complexity	MAX_CC
Average McCabe's cyclomatic complexity	AVG_CC
Number of files (compilation units)	NOCU

4.2 Eclipse Bug Dataset

In the Eclipse Bug Dataset there are two types of predictors. By parsing the structure of the obtained abstract syntax tree they [42] calculated the number of nodes for each type in a package and in a *file* (e.g. the number of return statements in a file). By implementing visitors to the Java parser of Eclipse they also calculated various complexity metrics at method, class, file, and package level. Then they used avg, max, total avg, total max aggregation techniques to accumulate to file and package level. The complexity metrics used in the Eclipse dataset are listed in Table 4.

Table 4: Metrics used in Eclipse Bug Dataset

Name	Abbr.
Number of method calls	FOUT
Method lines of code	MLOC
Nested block depth	NBD
Number of parameters	PAR
McCabe cyclomatic complexity	VG
Number of field	NOF
Number of method	NOM
Number of static fields	NSF
Number of static methods	NSM
Number of anonymous type declarations	ACD
Number of interfaces	NOI
Number of classes	NOT
Total lines of code	TLOC
Number of files (compilation units)	NOCU

4.3 Bug Prediction Dataset

The Bug Prediction Dataset collects product and change (process) metrics. The authors [10] produced the corresponding product and process metrics at *class* level. Besides the classic CK metrics, they calculated some additional object-oriented metrics that are listed in Table 5.

Table 5: Product metrics used in Bug Prediction Dataset

Name	Abbr.
Number of other classes that reference the class	FanIn
Number of other classes referenced by the class	FanOut
Number of attributes	NOA
Number of public attributes	NOPA
Number of private attributes	NOPRA
Number of attributes inherited	NOAI
Number of lines of code	LOC
Number of methods	NOM
Number of public methods	NOPM
Number of private methods	NOPRM
Number of methods inherited	NOMI

4.4 Bugcatchers Bug Dataset

The Bugcatchers Bug Dataset is a bit different from the previous datasets, since it does not contain traditional software metrics, but the number of bad smells for files. They [16] used five bad smells which are the following: Data Clumps, Message Chains, Middle Man, Speculative Generality, and Switch Statements. Besides, in the CSV file there are four source code metrics (blank, comment, code, codeLines) which are not explained in the corresponding publication [16].

4.5 GitHub Bug Dataset

The GitHub Bug Dataset [34] used the free version of SourceMeter [2] static analysis tool to calculate the static source code metrics including software product metrics, code clone metrics, and rule violation metrics. The rule violation metrics were not used in our research, therefore Table 6 shows only the list of the software product and code clone metrics.

Table 6: Metrics used in GitHub Bug Dataset

Name	Abbr.	Name	Abbr.
API Documentation	AD	Number of Local Public Methods	NLPM
Clone Classes	CCL	Number of Local Setters	NLS
Clone Complexity	CCO	Number of Methods	NM
Clone Coverage	CC	Number of Outgoing Invocations	NOI
Clone Instances	CI	Number of Parents	NOP
Clone Line Coverage	CLC	Number of Public Attributes	NPA
Clone Logical Line Coverage	CLLC	Number of Public Methods	NPM
Comment Density	CD	Number of Setters	NS
Comment Lines of Code	CLOC	Number of Statements	NOS
Coupling Between Object classes	CBO	Public Documented API	PDA
Coupling Between Obj. classes Inv.	CBOI	Public Undocumented API	PUA
Depth of Inheritance Tree	DIT	Response set For Class	RFC
Documentation Lines of Code	DLOC	Total Comment Density	TCD
Lack of Cohesion in Methods 5	LCOM5	Total Comment Lines of Code	TCLOC
Lines of Code	LOC	Total Lines of Code	TLOC
Lines of Duplicated Code	LDC	Total Logical Lines of Code	TLLOC
Logical Lines of Code	LLOC	Total Number of Attributes	TNA
Logical Lines of Duplicated Code	LLDC	Total Number of Getters	TNG
Nesting Level	NL	Total Number of Local Attributes	TNLA
Nesting Level Else-If	NLE	Total Number of Local Getters	TNLG
Number of Ancestors	NOA	Total Number of Local Methods	TNLM
Number of Attributes	NA	Total Number of Local Public Attr.	TNLPA
Number of Children	NOC	Total Number of Local Public Meth.	TNLPM
Number of Descendants	NOD	Total Number of Local Setters	TNLS
Number of Getters	NG	Total Number of Methods	TNM
Number of Incoming Invocations	NII	Total Number of Public Attributes	TNPA
Number of Local Attributes	NLA	Total Number of Public Methods	TNPM
Number of Local Getters	NLG	Total Number of Setters	TNS
Number of Local Methods	NLM	Total Number of Statements	TNOS
Number of Local Public Attributes	NLPA	Weighted Methods per Class	WMC

4.6 Unified Bug Dataset

The unified dataset contains all the datasets with their original metrics and with further metrics that we calculated with OpenStaticAnalyzer. The set of metrics calculated by OpenStaticAnalyzer concurs with the metric set of the GitHub Bug Dataset because SourceMeter is a product based on the free and open-source OpenStaticAnalyzer tool. Therefore, all datasets in the Unified Bug Dataset are extended with the metrics listed in Table 6 except the GitHub Bug Dataset, because it contains the same metrics originally.

In spite of the fact that several of the original metrics can be matched with the metrics calculated by OpenStaticAnalyzer, we decided to keep all the original metrics for every system included in the unified dataset because they can differ in their definitions or in the ways of their calculation. One can simply use the unified dataset and discard the metrics that were calculated by OpenStaticAnalyzer if s/he wants to work only with the original metrics. Furthermore, this provides an opportunity to confront the original and the OpenStaticAnalyzer metrics.

Instead of presenting all the definitions of metrics here, we give an external resource to show metric definitions because of lack of space. All the metrics and their definitions can be found in the Unified Bug Dataset file reachable as an online appendix (see Section 7).

5 FINDINGS

5.1 Datasets and Bug Distribution

Table 7 shows the basic properties about each dataset. In the *SCE* column the number of source code elements are presented, based on granularity it means the number of classes or files in the system. There are systems in the datasets with a wide variety of size from 2,636 *Logical Lines of Code (LLOC)*[1] up to 1,594,471.

SCEwBug means the number of source code elements which have at least one bug, *SCEwBug%* is the percentage of the source code elements with bugs in the dataset. There are datasets with very high (Xalan 2.7: 98.79%) and very low percentages (MCT: 0.48%) of buggy source code elements. The *SCEwBug%* as the percentage of buggy classes or files describes how well-balanced the datasets are.

5.2 Summary Meta Data

Table 8 lists some properties of the datasets, which show the circumstances of the dataset, rather than the data content. Our focus is on how the datasets were created, how reliable the used tools and the applied methods were. Since most of the information in the table was already described in previous sections (Analyzer, Granularity, Metrics, and Release), in this section we will describe only the Bug information row.

The Bug Prediction Dataset used the commit logs of SVN and the modification time of each file in CVS to collect co-changed files, authors and comments. Then they [10] linked the files with bugs from Bugzilla and Jira using the bug id from the commit messages. Finally, they verified the consistency of timestamps. They filtered out inner classes and test classes.

The PROMISE dataset used Buginfo to collect whether an SVN or CVS commit is a bugfix or not. Buginfo uses regular expressions to detect commit messages which contain bug information.

The bug information of the Eclipse Bug Dataset was extracted from the CVS repository and Bugzilla. In the first step they identified the corrections or fixes in the version history by looking for patterns which are possible references to bug entries in Bugzilla. In the second step they mapped the bug reports to versions using the version field of the bug report. Since the version of a bug report can change during the life cycle of a bug they used the first version.

The Bugcatchers Bug Dataset followed the methodology of Zimmermann et al. (Eclipse Bug Dataset). They developed an Ant script using the SVN and CVS plugins of Ant to checkout the source code and associate each fault with a file.

The authors of the GitHub bug dataset gathered the relevant versions to be analyzed from GitHub. Since GitHub can handle references between commits and issues, it was quite handy to use this information to match commits with bugs. They collected the number of bugs located in each file/class for the selected release versions (about 6 month long time intervals).

5.3 Comparison of the Metrics

In the unified dataset each element has lots of metrics but these values were calculated by different tools, therefore we assessed

[1] Lines of code not counting comments and whitespace.

Table 7: Basic properties of each dataset

Name	Granu-larity	SCE	SCE-wBug	SCE-wBug%	#Bug	kLLOC	Bug/kLine
Ant 1.3	class	125	20	16.00%	33	33	1.00
Ant 1.4	class	178	40	22.47%	47	43	1.09
Ant 1.5	class	293	32	10.92%	35	72	0.49
Ant 1.6	class	351	92	26.21%	184	98	1.88
Ant 1.7	class	745	166	22.28%	338	116	2.91
Camel 1.0	class	339	13	3.83%	14	26	0.54
Camel 1.2	class	590	216	36.61%	522	47	11.11
Camel 1.4	class	841	145	17.24%	335	76	4.41
Camel 1.6	class	928	188	20.26%	500	99	5.05
Ckjm 1.8	class	9	5	55.56%	23	8	2.88
Forrest 0.6	class	6	1	16.67%	1	19	0.05
Forrest 0.7	class	29	5	17.24%	15	4	3.75
Forrest 0.8	class	32	2	6.25%	6	3	2.00
Ivy 1.4	class	241	16	6.64%	18	31	0.58
Ivy 2.0	class	352	40	11.36%	56	54	1.04
JEdit 3.2	class	272	90	33.09%	382	55	6.95
JEdit 4.0	class	306	75	24.51%	226	63	3.59
JEdit 4.1	class	312	79	25.32%	217	72	3.01
JEdit 4.2	class	367	48	13.08%	106	88	1.20
JEdit 4.3	class	492	11	2.24%	12	109	0.11
Log4J 1.0	class	135	34	25.19%	61	10	6.10
Log4J 1.1	class	109	37	33.94%	86	14	6.14
Log4J 1.2	class	205	189	92.20%	498	23	21.65
Lucene 2.0	class	194	91	46.91%	268	68	3.94
Lucene 2.2	class	246	144	58.54%	414	111	3.73
Lucene 2.4	class	340	203	59.71%	632	126	5.02
Pbeans 1	class	26	20	76.92%	36	3	12.00
Pbeans 2	class	51	10	19.61%	19	6	3.17
Poi 1.5	class	237	141	59.49%	342	63	5.43
Poi 2.0	class	314	37	11.78%	39	82	0.48
Poi 2.5	class	385	248	64.42%	496	94	5.28
Poi 3.0	class	442	281	63.57%	500	140	3.57
Synapse 1.0	class	157	16	10.19%	21	20	1.05
Synapse 1.1	class	222	60	27.03%	99	33	3.00
Synapse 1.2	class	256	86	33.59%	145	46	3.15
Velocity 1.4	class	196	147	75.00%	210	26	8.08
Velocity 1.5	class	213	141	66.20%	330	33	10.00
Velocity 1.6	class	228	78	34.21%	190	37	5.14
Xalan 2.4	class	723	110	15.21%	156	104	1.50
Xalan 2.5	class	803	387	48.19%	531	126	4.21
Xalan 2.6	class	885	411	46.44%	625	154	4.06
Xalan 2.7	class	909	898	98.79%	1,213	160	7.58
Xerces 1.2	class	440	71	16.14%	115	65	1.77
Xerces 1.3	class	453	69	15.23%	193	69	2.80
Xerces 1.4	class	547	396	72.39%	1,554	74	21.00
Lucene 2.4	class	670	63	9.40%	96	126	0.76
Mylyn 3.1	class	1,405	209	14.88%	296	166	1.78
PDE UI 3.4.1	class	1,492	208	13.94%	340	186	1.83
Equinox 3.4	class	319	126	39.50%	239	64	3.73
Eclipse JDT Core 3.4	class	997	206	20.66%	374	630	0.59
Apache Commons	file	191	84	43.98%	223	53	4.21
ArgoUML 0.26 Beta	file	1,579	192	12.16%	285	186	1.53
Eclipse JDT Core 3.1	file	535	310	57.94%	567	1,594	0.36
Eclipse 2.0	file	6,729	2,610	38.79%	7,635	792	9.64
Eclipse 2.1	file	7,888	2,139	27.12%	4,975	988	5.04
Eclipse 3.0	file	10,593	2,913	27.50%	7,422	1,307	5.68
Android U. I. L. 1.7.0	class	73	20	27.40%	35	4	8.75
ANTLR v4 4.2	class	479	21	4.38%	27	40	0.68
Broadleaf C. 3.0.10	class	1,593	292	18.33%	353	125	2.82
Eclipse for Ceylon 1.1.0	class	1,610	68	4.22%	104	112	0.93
Elasticsearch 0.90.11	class	5,908	678	11.48%	1,182	362	3.27
Hazelcast 3.3	class	3,412	380	11.14%	1,053	179	5.88
JUnit 4.9	class	731	35	4.79%	41	16	2.56
MapDB 0.9.6	class	331	40	12.08%	96	47	2.04
mcMMO 1.4.06	class	301	57	18.94%	114	23	4.96
MCT 1.7b1	class	1,887	9	0.48%	9	104	0.09
Neo4j 1.9.7	class	5,899	58	0.98%	60	328	0.18
Netty 3.6.3	class	1,143	271	23.71%	423	76	5.57
OrientDB 1.6.2	class	1,847	280	15.16%	494	184	2.68
Oryx	class	533	74	13.88%	80	29	2.76
Titan 0.5.1	class	1,468	96	6.54%	106	80	1.33
Android U. I. L. 1.7.0	file	63	18	28.57%	33	4	8.25
ANTLR v4 4.2	file	411	41	9.98%	59	40	1.48
Broadleaf C. 3.0.10	file	1,719	286	16.64%	350	125	2.80
Ceylon for Eclipse 1.1.0	file	699	57	8.15%	94	112	0.84
Elasticsearch 0.90.11	file	3,035	487	16.05%	899	362	2.48
Hazelcast 3.3	file	2,228	317	14.23%	911	179	5.09
JUnit 4.9	file	308	42	13.64%	50	16	3.13
MapDB 0.9.6	file	137	22	16.06%	59	47	1.26
mcMMO 1.4.06	file	267	57	21.35%	114	23	4.96
MCT 1.7b1	file	413	6	1.45%	6	104	0.06
Neo4j 1.9.7	file	3,278	32	0.98%	33	328	0.10
Netty 3.6.3	file	913	243	26.62%	381	76	5.01
OrientDB 1.6.2	file	1,503	270	17.96%	493	184	2.68
Oryx	file	280	44	15.71%	48	29	1.66
Titan 0.5.1	file	975	70	7.18%	80	80	1.00

Table 8: Summary Meta Data

Dataset	Bug Prediction Dataset	PROMISE	Eclipse Bug Dataset	Bugcatchers Bug Dataset	GitHub Bug Dataset
Analyzer	inFusion Moose	ckjm	Visitors written for Java parser of Eclipse	Bad Smell Detector	SourceMeter
Granularity	Class	Class	File	File	Class, File
Bug information	CVS, SVN, Bugzilla, Jira	SVN, CVS	CVS, Bugzilla	CVS, SVN, Bugzilla, Jira	GitHub
Metrics	CK, process metrics	CK	Complexity, Structure of abstract syntax tree	Bad Smell	CK, Complexity, Clone, Rule violation
Release	post	pre	pre & post	pre	pre & post

them in more detail to get answers to questions like the following ones: Do two metrics with the same name have the same meaning?

Table 9: Number of elements in the merged systems

Name	Merged	Remained elements
Bug Prediction Dataset	11,370	4,167
Eclipse Bug Dataset	25,295	25,210
Bugcatchers bug Dataset	14,543	2,305

Do metrics with different names have the same definition? Can two metrics with the same definition be different? What are the causes of the differences if the metrics have the same definition?

Three out of the five datasets contain class level elements, but unfortunately, for each dataset a different analyzer tool was used to calculate the metrics (see Table 8). To be able to compare class level metrics calculated by all the used tools, we needed at least one dataset for which all metrics of all three tools are available. We were already familiar with the usage of the ckjm tool, so we choose to calculate the ckjm metrics for the Bug Prediction dataset. This way we could assess all metrics of all tools, because the Bug Prediction dataset was originally created with Moose, we extended it with OpenStaticAnalyzer metrics, and – for the sake of this comparison – also with ckjm metrics.

In case of the three file level datasets, the other analyzer tools were not available, therefore we could only compare the file level metrics of OpenStaticAnalyzer with the results of the other two tools separately on Eclipse and Bugcatchers Bug datasets.

In each comparison, we merged the different result files of each dataset into one, which contained the results of all systems in the given dataset and deleted those elements that did not contain all metric values. The resulting spreadsheet files can be found in the online appendix. Table 9 shows how many classes or files were in the given dataset and how many of them remained.[2] We calculated the basic statistics (minimum, maximum, average, median, and standard deviation) of the examined metrics and compared them. Besides, we calculated the pairwise differences of the metrics for each element and examined its basic statistics as well. Even though we cannot provide deep findings from these statistics, they give a first impression.

5.3.1 Class level metrics. The unified bug dataset contained the class level metrics of OpenStaticAnalyzer and Moose on Bug Prediction dataset. We downloaded the Java binaries of the systems in this dataset and used ckjm version 2.2 to calculate the metrics. The first difference is that while OpenStaticAnalyzer and Moose calculate metrics on source code, ckjm uses Java bytecode and takes into account "external dependences" therefore we excepted differences in, e.g., coupling metrics.

We compared the metric sets of the three tools and found that, for example, CBO or WMC have different definitions. On the other hand, efferent coupling metric is a good example for the metric which is calculated by all three tools but with different names (see Table 10, CBO row). In the following, we examine only those metrics whose definitions coincide in all three tools even if their names differ. Table 10 shows these metrics where the *Metric* column contains the abbreviation of the most widely used name of the metric. The *Tool* column presents the analyzer tools, in the *Metric name* column, the metric names are given using the notations of the different datasets. The "$\text{tool}_1 - \text{tool}_2$" means the pairwise difference where for each element we extracted the value of tool_2 from the value of tool_1 and the name of this "new metric" is diff. The rest of

[2] The big differences between the number of merged and remained elements is explained in Section 3.2, see Table 2.

the columns present the basic statistics of the metrics and in the following we will analyze their values one metric at a time.

Table 10: Comparison of the common class level metrics on the extended Bug Prediction dataset

Metric	Tool	Metric name	Min	Max	Avg	Med	Dev
WMC	OSA	NLM	0	426	11.04	7	18.12
	Moose	Methods	0	403	9.96	6	14.38
	ckjm	WMC	1	426	11.96	7	18.49
	OSA−Moose	diff	-4	420	1.08	0	9.40
	OSA−ckjm	diff	-48	0	-0.91	0	1.94
	Moose−ckjm	diff	-421	4	-1.99	-1	9.57
CBO	OSA	CBO	0	214	8.86	5	12.25
	Moose	fanOut	0	93	6.22	4	7.79
	ckjm	Ce	0	213	13.78	8	16.88
	OSA−Moose	diff	-32	161	2.65	2	7.61
	OSA−ckjm	diff	-120	83	-4.91	-1	9.72
	Moose−ckjm	diff	-160	32	-7.56	-4	11.84
CBOI	OSA	CBOI	0	607	9.38	3	26.14
	Moose	fanIn	0	355	4.69	1	14.30
	ckjm	Ca	0	611	7.64	2	22.13
	OSA−Moose	diff	-18	607	4.69	1	16.55
	OSA−ckjm	diff	-100	189	1.74	0	11.02
	Moose−ckjm	diff	-611	146	-2.95	-1	15.30
RFC	OSA	RFC	0	600	22.82	12	34.53
	Moose	rfc	0	2,603	50.62	23	108.06
	ckjm	RFC	2	684	38.93	23	49.72
	OSA−Moose	diff	-2,095	600	-27.80	-8	83.70
	OSA−ckjm	diff	-327	12	-16.11	-9	22.72
	Moose−ckjm	diff	-673	2,049	11.69	-1	75.42
DIT	OSA	DIT	0	8	1.31	1	1.63
	Moose	dit	1	9	2.08	2	1.44
	ckjm	DIT	0	5	0.38	0	0.60
	OSA−Moose	diff	-3	0	-0.76	-1	0.43
	OSA−ckjm	diff	-5	8	0.94	1	1.96
	Moose−ckjm	diff	-4	9	1.70	2	1.79
NOC	OSA	NOC	0	107	0.73	0	3.27
	Moose	noc	0	49	0.64	0	2.55
	ckjm	NOC	0	107	0.64	0	2.95
	OSA−Moose	diff	-3	97	0.08	0	1.68
	OSA−ckjm	diff	0	42	0.09	0	1.15
	Moose−ckjm	diff	-97	34	0.01	0	1.81
LOC	OSA	LLOC	2	8,746	131.99	56	357.39
	Moose	LinesOfCode	0	7,341	124.01	51	306.54
	ckjm	LOC	4	26,576	399.42	147	1142.60
	OSA−Moose	diff	-1,068	7,824	7.98	3	157.69
	OSA−ckjm	diff	-19,150	112	-267.43	-91	791.30
	Moose−ckjm	diff	-26,541	198	-275.41	-93	879.89
NPM	OSA	NLPM	0	404	7.23	4	13.67
	Moose	PublicMethods	0	387	6.42	4	11.28
	ckjm	NPM	0	404	7.48	5	13.64
	OSA−Moose	diff	-4	236	0.81	0	6.55
	OSA−ckjm	diff	-8	0	-0.25	0	0.45
	Moose−ckjm	diff	-237	3	-1.06	0	6.55

WMC: This metric counts the complexity of a class as the sum of the complexity of its methods. In its original definition the method complexity is deliberately not defined exactly and usually the uniform weight of 1 is used and this metrics expresses the number of methods. In this case, this variant of WMC is calculated by all three tools. Its basic statistics are more or less the same and the pairwise values of OpenStaticAnalyzer and ckjm are also close to each other (see OSA−ckjm row) but they differ very much from Moose. Among the Moose results there were several very low values where the other tools found lots of methods and that caused the extreme difference (e.g. the max. value of OSA−Moose is 420).

CBO: In this definition CBO counts the number of classes the given class depends on. Although it is a coupling metric, it counts efferent (outer) couplings, therefore the metric values should have been similar. On the other hand, based on the statistical values and the pairwise comparison we can say that these metrics differ a lot. The reasons can be, for example, that ckjm takes into account "external" dependencies (e.g. classes from *java.util*) or it counts coupling based on generated elements (e.g. generated default constructor) but further investigation would be required to determine all causes.

CBOI: It counts those classes that use the given class. Although the basic statistics of *OSA* and *ckjm* are close to each other, its pairwise comparison suggests that they are different. Moreover, the metrics values of *Moose* are very different. The main reason can be, for example, that *OSA* found two times more classes, therefore it is explicable that more classes use the given class or *ckjm* takes into account the generated classes and connections as well that exist in the bytecode but not in the source code.

RFC: all three tools defined this metric in the same way but the comparison shows that the metric values are very different. The reasons for this are mainly the same as in case of the CBO metric.

DIT: Although the statistical values "hardly" differ compared to the previous ones, these values are usually small (as the max. values show) therefore these differences are quite large. From the minimal values we can see that Moose probably counts Object too as the base class of all Java classes, while the other two tools neglect this.

NOC: The values of OpenStaticAnalyzer and ckjm are close to each other, while Moose gave very different values. Since it counts efferent coupling, the cause of its variation is similar to CBOI.

LOC: Lines of code seem the most unambiguous metric but it also differs a lot. Although this metric has several variants and it is not defined exactly how Moose and ckjm counts it, we used the closest one from OpenStaticAnalyzer based on the metric values. The very large value of ckjm is surprising but it counts this value from the bytecode, therefore it is not easy to validate it. Besides, OpenStaticAnalyzer and Moose have different values in spite of the fact that both of them calculate LOC from source code. The 0 minimal value of Moose is also interesting and suggest that either it used different definition or the algorithm was not good enough.

NPM: The number of public methods metrics of OpenStaticAnalyzer and ckjm are really close to each other while Moose has different results in this case as well.

The comparison of the three tools revealed that even though they calculate the same metrics, the results are very divergent. A few of its reasons can be that *ckjm* calculates metrics from bytecode while the other two tools work on source code, or *ckjm* takes into account external code as well while *OSA* does not. Besides, we could not compare the detailed and precise definitions of the metrics to be sure that they are really calculated in the same way, therefore it is possible that they differ slightly which causes the differences.

5.3.2 *File level metrics.* Bugcatchers, Eclipse, and GitHub Bug Dataset are the ones which operate at file level (GitHub Bug Dataset contains class level too). Unfortunately, we could make only pairwise comparisons between file level metrics since we could not replicate the measurements used in the Eclipse Bug Dataset (custom Eclipse JDT visitors were used) and in the Bugcatchers Bug Dataset (unknown bad smell detector was used).

In case of Bugcatchers Bug Dataset we compared the results of OpenStaticAnalyzer and the original metrics which were produced by a code smell detector. Since OpenStaticAnalyzer only calculates a narrow set of file level metrics, Logical Lines of Code (LLOC) is the only metric we could use in this comparison. Table 11 presents the result of this comparison. Min, max, and median values are likely to be the same. Moreover, the average difference between LLOC values is less than 1 with a standard deviation of 6.05 which can be considered as insignificant in case of LLOC at file level.

There is an additional common metric (CLOC) which is not listed in Table 11 since OpenStaticAnalyzer returned 0 values for all the files. This possible error in OpenStaticAnalyzer makes it superfluous to examine CLOC in further detail.

In case of the Eclipse Bug Dataset LLOC values are the same in most of the cases (see Table 12). OpenStaticAnalyzer counted one

Table 11: Comparison of file level LLOC on Bugcatchers

Metric	Tool	Met. name	Min	Max	Avg	Med	Dev
LLOC	OSA	LLOC	3	5,774	93.33	41	221.16
	Smell Detector	code	3	5,774	92.34	40	219.06
	OSA−Smell Detector	diff	-11	130	0.98	0	6.05

extra line in 10 cases out of 25,210 which is a negligible difference. Unfortunately, there is a serious sway in case of the McCabe's Cyclomatic Complexity. There is a case when the difference is 299 in the calculated values which is extremely high in case of this metric. We investigated these cases and found that OpenStaticAnalyzer does not include the number of methods in the final value. There are many cases when OpenStaticAnalyzer gives 1 as a result while the Eclipse Visitor calculates 0 as complexity. Probably OpenStaticAnalyzer counts class definitions but not method definitions which could be specious. There are cases when OpenStaticAnalyzer has higher complexity values. It turned out that OpenStaticAnalyzer took into consideration the ternary operator (?:) what is correct, since these statements also form a condition. Both calculation techniques seem to have some minor issues or at least we have to say that the metric definitions of cyclomatic complexity differ.

Table 12: Comparison of file level metrics on Eclipse

Metric	Tool	Metric name	Min	Max	Avg	Med	Dev
LLOC	OSA	LLOC	3	5,228	122.59	52	230.02
	Visitor	TLOC	3	5,228	122.59	52	230.02
	OSA−Visitor	diff	-7	1	0.0001	0	**0.048**
McCC	OSA	McCC	1	1,198	19.55	5	48.27
	Visitor	VG_sum	0	1,479	28.06	10	60.35
	OSA−Visitor	diff	-299	123	-8.50	-4	15.83

6 THREATS TO VALIDITY

We accepted the collected datasets "as is", which means that we did not validate the data, we just used them to create the unified dataset and examined how similar the metrics of the different datasets are. Since the bug datasets did not contain the source code, neither a step-by-step instruction how to reproduce the bug datasets, we had to accept them even if there are a few notable anomalies in them. For example, Camel 1.4 contains classes with LOC metrics of 0, or in the Bugcatchers dataset there are two MessageChains metrics, and in several cases the two metric values are different.

Although for each system the version information was available, in some cases there were notable differences between the result of OpenStaticAnalyzer and the original result in the corresponding bug dataset. Even if the analyzers would parse the classes in different ways, the number of files should have been equal. If the analysis result of OpenStaticAnalyzer contains the same or more elements, and (almost) all elements from the corresponding bug dataset could be conjugated, we can say that the unification is acceptable, because all elements of the bug dataset were put into the unified dataset. On the other hand, for a few systems we could not find the proper source code version and we had to leave out a notable number of elements from the unified dataset.

Many systems are more than 10 years old when the actual Java version was 1.4 and these systems were analyzed according to that standard. The Java language has evolved a lot since then and we analyzed all systems according to the latest standard, which might cause minor mistakes in the results, but we think they are negligible.

In Section 5.3 we used ckjm 2.2 to analyze the projects included in the Bug Prediction Dataset. We chose version *2.2* since the original paper did not mark the exact version of ckjm [10], consequently we experimented with different ckjm versions (1.9, 2.0, 2.1, 2.2) and we experienced version 2.2 to be the best candidate since it produced the smallest differences in metric values compared to the original metric values in the Bug Prediction Dataset.

We used a heuristic based on name matching to conjugate the elements of the datasets. Although there were cases when the conjugation was unsuccessful, we examined those cases manually and it turned out that the heuristic worked well and the cause of the problem originated from the differences of the two datasets (in Section 3 all cases are listed). We examined the successful conjugations as well and all of them were correct. Even though the heuristic could not handle elements having the same name during the conjugation, only a negligible amount of such cases happened.

Even when the matching heuristic worked well, the same class name can have different meanings in different datasets. For example, *OSA* handles nested, local, and anonymous classes as different elements, while other datasets did not take into account such elements. Even more, the whole file was associated with its public class. This way, a bug found in a nested or inner class is associated with the public class in the bug datasets but during the matching this bug will be associated with the wrong element of the more detailed analysis result of *OSA*.

7 CONCLUSION AND FUTURE WORK

There are several public bug datasets available in the literature. Our aim was to create one public unified bug dataset which contains the publicly available ones in a unified format. This dataset can provide researchers real value by offering a readily available rich bug dataset for their new bug prediction experiments.

To find the available public bug datasets we performed a literature review. We found five different public bug datasets that adhered to our expectations: the PROMISE, the Eclipse, the Bug Prediction, the Bugcatchers, and the GitHub Bug datasets. We gave detailed information about each dataset which contains among others their size, enumeration of the included software systems, used version control and bug tracking systems.

We developed and executed a method on how to create the unified set of bug data which encapsulates all the information that is available in the datasets. Different datasets use different metrics suites, hence we collected the source code for all software systems of each dataset and analyzed them with one particular static source code analyzer (*OSA*) in order to have a common and uniform set of code metrics for every system. We constructed the unified bug dataset from the gathered public datasets at file and class level and made this dataset publicly available to anyone for future use.

We evaluated the datasets according to known meta data, including the investigation of the used static code analyzer, granularity, bug tracking and version control system, the set of used metrics, etc. We also compared the values of the metrics calculated by different tools, and found that there are quite large differences, which strengthens the need for a uniform bug dataset.

We encourage researchers to use this large and public unified bug dataset in their experiments and we also welcome new public bug datasets. As a future work we would like to keep this unified dataset up-to-date and extend it with public datasets which were published recently (i.e. Had-oops dataset [17], Mutation bug dataset [6], ELFF dataset [32]) and others, which will be published in the future.

ONLINE APPENDIX

The Unified Bug Dataset 1.0 which was created during this work is available as an online appendix at: http://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet

The *UnifiedBugDataset-1.0.zip* file contains (1) the original bug datasets in their original form, (2) the source code of the systems that was used to develop the datasets, (3) the unified dataset in CSV format at file/class level, (4) description of the OpenStaticAnalyzer metrics, and (5) metrics comparisons in spreadsheet format of the PROMISE [3], Eclipse [42], Bug Prediction [10], Bugcatchers [16], and GitHub [34] bug datasets.

ACKNOWLEDGMENTS

This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things".

REFERENCES

[1] OpenStaticAnalyzer static code analyzer. https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer.

[2] SourceMeter static code analyzer. https://www.sourcemeter.com.

[3] The promise repository of empirical software engineering data, 2015.

[4] Vera Barstad, Morten Goodwin, and Terje Gjøsæter. Predicting source code quality with static analysis and machine learning. In *NIK*, 2014.

[5] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[6] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 330–341. ACM, 2016.

[7] Lionel C. Briand, John W. Daly, and Jurgen K Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1):91–121, 1999.

[8] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *SEKE*, pages 238–243, 2011.

[9] Valentin Dallmeier and Thomas Zimmermann. Automatic extraction of bug localization benchmarks from history. In *Proc. IntâĂŹl Conf. on Automated Software Eng*, pages 433–436. Citeseer, 2007.

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

[11] Marco DâĂŹAmbros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[12] David Gray, David Bowes, Neil Davey, Y Sun, and Bruce Christianson. Reflections on the nasa mdp data sets. *IET software*, 6(6):549–558, 2012.

[13] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

[14] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[15] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[16] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):33, 2014.

[17] Mark Harman, Syed Islam, Yue Jia, Leandro L. Minku, Federica Sarro, and Komsan Srivisut. Less is more: Temporal fault predictive performance over multiple hadoop releases. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering*, pages 240–246, Cham, 2014. Springer International Publishing.

[18] J Horning, H Lauer, P Melliar-Smith, and Brian Randell. A program structure for error detection and recovery. *Operating Systems*, pages 171–187, 1974.

[19] Marian Jureczko and Lech Madeyski. 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.

[20] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[21] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 131–142. ACM, 2008.

[22] Ruchika Malhotra, Shivani Shukla, and Geet Sawhney. Assessment of defect prediction models using machine learning techniques for object-oriented systems. In *Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO), 2016 5th International Conference on*, pages 577–583. IEEE, 2016.

[23] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1), 2007.

[24] Ayse Tosun Misirli, Ayse Bener, and Resat Kale. Ai-based software defect predictors: Applications and benefits in a case study. *AI Magazine*, 32(2):57–68, 2011.

[25] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190. ACM, 2008.

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

[27] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[28] Shruthi Puranik, Pranav Deshpande, and K Chandrasekaran. A novel machine learning approach for bug prediction. *Procedia Computer Science*, 93:924–930, 2016.

[29] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 10(2):220–232, 1975.

[30] Gregorio Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 171–180. IEEE, 2010.

[31] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering*, 39(9):1208–1215, 2013.

[32] Thomas Shippey, Tracy Hall, Steve Counsell, and David Bowes. So you need more method level datasets for your software defect prediction?: Voilà! In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 12. ACM, 2016.

[33] Ramanath Subramanyam and Mayuram S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering*, 29(4):297–310, 2003.

[34] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. 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.

[35] Elaine J Weyuker, Robert M Bell, and Thomas J Ostrand. Replicate, replicate, replicate. In *Replication in Empirical Software Engineering Research (RESER), 2011 Second International Workshop on*, pages 71–77. IEEE, 2011.

[36] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15(3):277–295, 2010.

[37] W Eric Wong, Vidroha Debroy, and Dianxiang Xu. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(3):378–396, 2012.

[38] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[39] Zhiwei Xu, Taghi M Khoshgoftaar, and Edward B Allen. Prediction of software faults using fuzzy nonlinear regression modeling. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 281–290. IEEE, 2000.

[40] Zhe Yu, Nicholas A Kraft, and Tim Menzies. How to read less: Better machine assisted reading methods for systematic literature reviews. *arXiv preprint arXiv:1612.03224*, 2016.

[41] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. 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*, pages 91–100. ACM, 2009.

[42] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the third international workshop on predictor models in software engineering*, page 9. IEEE Computer Society, 2007.