

# A Novel Bug Prediction Dataset, Process Metrics, and a Public Dataset of JavaScript Bugs

**Péter Gyimesi**

Department of Software Engineering  
University of Szeged

Szeged, 2023

Supervisor:

Dr. habil. Rudolf Ferenc

SUMMARY OF THE PH.D. THESIS



University of Szeged  
Ph.D. School in Computer Science



# Introduction

Managing software bugs is an essential part of software development and companies tend to spend a large amount of resources on it. Programmers tend to make mistakes despite the assistance provided by different integrated development environments, and errors may also occur due to frequent changes in the code and inappropriate specifications; therefore, it is important to get more and/or better tools to help the automatic detection of errors [24]. Dealing with software bugs consists of tasks like preventing, finding, and fixing bugs.

Finding software bugs is usually done by checking the source code manually and looking for the root of the problem based on bug reports. It is a time- and resource-consuming activity, and minimizing the required effort would help to reduce the cost of the development. Unit tests can also help us detect and localize software faults, but it requires writing test cases in parallel with development, and it is also a resource-intensive task. Another way of assisting bug localization is to characterize the known ones with some appropriate metrics and try to predict which source code elements have the highest probability of containing a bug. The most important step in facilitating error detection is to analyze already known errors to identify patterns or trends.

Analyzing known bugs requires a source code change history and a bug tracking system. Nowadays, many developers use a versioning system - like Subversion or Git -, hence the source code history is often available. The use of bug tracking systems is also quite common in software development. There are numerous commercial and open-source software systems available for these purposes. The bug reports are recorded within these systems and all changes related to the bugs are also tracked, including the source code fixes. Furthermore, different web services are built to meet these needs. The most popular ones, like SourceForge, Bitbucket, and GitHub, fulfill the above-mentioned functionalities. They usually provide several services, such as source code hosting and user management. Their APIs make it possible to retrieve various kinds of data, e.g., they provide support for the examination of the behavior or the cooperation of users, or even for the analysis of the source code itself. Since most of these services include bug tracking, the idea of using this information in the characterization of buggy source code parts is raised [38]. To do so, the bug reports managed by these source code hosting providers must be connected to the appropriate source code parts. A common practice in version control systems is to describe the changes in a comment belonging to a commit (log message) and often to provide the identifier of the associated bug report that the commit is supposed to fix [26]. This can be used to identify the faulty versions of the source code. GitHub contains more than 330 million repositories and has a readily usable API to access these projects, which are accessible via Git; hence it is a convenient choice as a data source for the studies.

In terms of programming languages, some of the most popular languages in use today include Java, Python, C++, JavaScript, and PHP. According to the TIOBE Index, the most popular programming language in 2023 was Python, followed by C, C++, and Java. Java has been a popular programming language for many years and is widely used in enterprise software development due to its scalability, reliability, and portability. JavaScript (JS) is the de-facto web programming language globally, and the most adopted language on GitHub. JavaScript is massively used in the client side of web applications to achieve high responsiveness and user-friendliness. In recent years, due to its flexibility and effectiveness, it has also been increasingly adopted for server-side development, leading to full-stack web applications [11]. Platforms such as Node.js allow developers to conveniently develop both the front- and back-end of the applications entirely in JavaScript. Despite its popularity, the intrinsic characteristics of JavaScript—such as weak typing, prototypal inheritance, and run-time evaluation—make it one of the most error-prone programming languages. As such, a large body of software engineering research has focused on

the analysis and testing of JavaScript web applications [14, 35, 36, 29, 20, 9, 28, 10]. For these reasons, although there are many different programming languages, we made a conscious decision to focus our efforts on analyzing bugs reported in Java or JavaScript projects.

Although a vast amount of raw data is available regarding software bugs, collecting, filtering, and processing it can be a time-consuming and resource-intensive task. Many papers have dealt with bug databases using many kinds of approaches, such as bug prediction, fault localization, or testing techniques [17, 32, 30, 27, 39]. Researchers often use a database created for their own purposes, but these datasets are rarely published for the community. Despite the abundance of research on software bugs, the availability of bug databases that are accessible to the public is notably inadequate and overlooked. Additionally, subject programs or accompanying experimental data are rarely made available in a detailed, descriptive, curated, and coherent manner. This not only hampers the reproducibility of the studies themselves but also makes it difficult for researchers to assess the state-of-the-art of related research and compare existing solutions. Specifically, testing techniques are typically evaluated with respect to their effectiveness at detecting faults in existing programs, however, real bugs are hard to isolate, reproduce, and characterize. Therefore, the common practice relies on manually seeded faults or mutation testing [23]. Each of these solutions has limitations. Manually injected faults can be biased toward researchers' expectations, undermining the representativeness of the studies that use them. Mutation techniques, on the other hand, allow for generating a large number of "artificial" faults. Although research has shown that mutants are quite representative of real bugs [21, 25, 12], mutation testing is computationally expensive to use in practice. For these reasons, publicly available benchmarks of bugs are of paramount importance for devising novel debugging, bug prediction, fault localization, or program repair approaches.

The characterization of buggy source code elements through various methods is still a popular research area. For automatic recognition of unknown faulty code elements, it is a prerequisite to characterize the already known ones. There are many good studies on bug characterization [22, 34, 16, 19]. Processing the diff files of a commit can help us obtain the exact code sections affected by the bug. The most commonly used methods for bug characterization include textual similarities with faulty code parts [13], source code analysis, product metrics [18, 33], or process metrics. There are numerous tools, some of which are free, that are capable of analyzing source code written in different programming languages and producing product metrics for the code elements. During our studies, we used the OpenStaticAnalyzer tool for source code analysis, because it is able to process source code written in either Java or JavaScript programming languages and it can extract detailed information about the source code elements.

Our work is comprised of three thesis points. The objectives of these thesis points are as follows:

- I. Present a novel method for constructing a bug database and evaluate its usefulness for bug prediction while also comparing it with a database made using the traditional approach.**
- II. Present a method for computing software process metrics using a graph database, assess the metrics' predictive power for bugs, and compare them with product metrics.**
- III. Present a benchmark of real, manually verified JavaScript bugs, and discuss the results of our quantitative and qualitative analyses of these bugs.**

# I. A Novel Bug Prediction Dataset and its Validation

Previously published datasets have followed a conventional approach to creating benchmark datasets for testing bug prediction techniques. These datasets typically include all code elements, both faulty and non-faulty, from one or more versions of the analyzed system. In this thesis point, we introduce a new approach that focuses on collecting snapshots of source code elements affected by bugs, along with their characteristics, before and after the bugs were fixed. This approach excludes code elements that were not impacted by bugs.

By utilizing this kind of dataset, we can effectively capture the changes in software product metrics during bug fixes. This enables us to examine the differences in source code metrics between faulty and non-faulty code elements and gain valuable insights. To conduct our analysis, we selected 15 open-source projects from GitHub and considered all reported bugs from their bug tracking systems. We constructed databases at three source code levels: file, class, and method. This new dataset is called the *BugHunter Dataset*.

Using 11 machine learning algorithms, we built prediction models and demonstrated the dataset’s potential for bug prediction and further investigations. An important aspect to investigate is how the bug prediction models built from the novel dataset compare to the ones that used the traditional datasets as corpus. To conduct this comparison we utilized our traditional dataset, which was generated from the same 15 projects and referred to as the *GitHub Bug Dataset* [7].

We investigated the following research questions:

- **Research Question 1:** Is the *BugHunter Dataset* usable for bug prediction purposes?
- **Research Question 2:** Are the method-level metrics projected to the class level better predictors than the class-level metrics themselves?
- **Research Question 3:** Is the *BugHunter Dataset* more powerful and expressive than the *GitHub Bug Dataset*?

To answer the *first research question*, we analyzed the results obtained by different machine learning algorithms at the method, class, and file levels. Considering the results we obtained, we can state that creating bug prediction models at the method level is more successful than at file and class levels if we consider the full dataset. We also observed variations in F-measure values across different projects, supporting our hypothesis that not all projects provide suitable training sets. We achieved promising F-measure values for individual projects, reaching up to 0.7573 at the method level, 0.7400 at the class level, and 0.7741 at the file level. We can answer this research question in a positive manner and say that the constructed dataset is usable for bug prediction.

The dataset contains the bug information on both method and class levels, and we also know the containing relationships between classes and methods. However, since classes have a different set of source code metrics than methods, a question arose: can we (and more importantly, should we) use method-level metrics to predict faulty classes? To answer our *second research question*, we carried out an experiment where we projected the results of the method-level learning to the class level. During the cross-validation of the method-level learning, we used the containing classes of the methods to calculate the confusion matrix from the number of classes classified as buggy and non-buggy. Classes containing at least one buggy method were considered buggy.

We compared this result with the result of the class-level prediction. The results in Table 1 show that the projection method performs much better than the prediction with class-level metrics.

Table 1: The results of projected learning

Algorithm	Precision		Recall		F-Measure	
	Projected	Class	Projected	Class	Projected	Class
trees.RandomForest	0.7471	0.5336	0.7370	0.5336	0.7405	0.5334
trees.RandomTree	0.7421	0.5381	0.7273	0.5380	0.7330	0.5376
functions.SGD	0.7441	0.5718	0.7288	0.5676	0.7322	0.5626
rules.DecisionTable	0.7425	0.5703	0.7404	0.5705	0.7309	0.5637
trees.J48	0.7390	0.5531	0.7250	0.5530	0.7290	0.5520

Using method-level metrics for class-level bug prediction performed the best in our study; the F-measure values reached 0.74. We suspect that this is due to the generality of class-level metrics, which are therefore not powerful enough to effectively distinguish source code bugs. As an extension of the answer to the first research question, we can provide the above-described mechanism to locate class-level bugs with higher accuracy in a software system.

Comparing the expressive power of different datasets is a harsh task since, as the various datasets were created with different purposes, they often have only a few independent variables in common. To answer our *third research question*, we provide an objective comparison between our traditional bug dataset, the *GitHub Bug Dataset*, and the *BugHunter Dataset*. The datasets include exactly the same 15 projects, and their sets of independent variables are common and also calculated in the same way with the same tool. We used the same machine learning algorithms to build prediction models. This way, it is quite straightforward to compare the expressiveness and compactness of these datasets.

Table 2: Predictive capabilities and sizes of the datasets

Dataset	Avg.	Std.dev.	Min	Max	Size
METHOD LEVEL					
BugHunter	0.6319	0.0836	0.3376	0.7573	109,244
Traditional	0.7348	0.0789	0.4019	0.8339	167,708
CLASS LEVEL					
BugHunter	0.5685	0.0704	0.3572	0.7400	66,092
Traditional	0.7710	0.0869	0.3446	0.8331	27,216
FILE LEVEL					
BugHunter	0.5147	0.0749	0.3328	0.7741	49,868
Traditional	0.6058	0.1076	0.2882	0.8247	16,235
PROJECTED					
BugHunter	0.7405	0.0914	0.3178	0.8386	-
Traditional	0.7831	0.0716	0.4399	0.8825	-

Firstly, we compare the size of the datasets expressed with the number of entries located in

the datasets (see Table 2). A rate of  $1.54$  is achieved at the method level,  $0.41$  at the class level, and  $0.33$  at the file level. The obtained rate is higher than 1.0 for the method level, which shows that the new approach contains fewer entries at this level. One would expect that the new approach will contain fewer entries than the traditional one since the *BugHunter Dataset* only contains the entries that were affected by a closed bug. However, the traditional *GitHub Bug Dataset* only depends on the size (number of files, classes, and methods) of the projects included. In contrast, the *BugHunter Dataset* highly depends on the number of closed bugs in the system (a large project can have a small number of reported bugs). To sum up, we cannot clearly decide whether the novel dataset is more compact, however, it is clearly visible that BugHunter could compress the bug-related information at the method level. We achieved an F-measure value of 0.6319 at the method level and the composed dataset contains 58,464 fewer entries than the traditional one. In both datasets, the number of entries is sufficient to build a predictive model from, however, we should investigate the predictive capabilities first to conclude our findings related to expressive power and compactness.

Table 2 presents machine learning results for method, class, and file levels, and also the F-measure values for the projected method-level predictors, respectively.

On the traditional *GitHub Bug Dataset*, the machine learning algorithms performed better, achieving higher F-measure values in every case. The two kinds of datasets differ fundamentally because they are constructed with two different methods. The method of building a traditional dataset leads to some uncertainty in the dataset because it could happen that the bug is not yet present in the assigned release version. Table 3 shows some characteristics of this uncertainty.

Table 3: Uncertainty in the traditional dataset

<b>Project</b>	<b>Average days</b>	<b>Average commits before reported</b>	<b>Average commits before fixed</b>
ANDROID U. I. L.	78.78	179.04	22.82
ANTLR v4	83.73	94.83	66.21
BROADLEAF COMM.	96.40	524.88	116.74
ECLIPSE CEYLON	136.05	442.00	20.22
ELASTICSEARCH	93.85	1,004.60	382.79
HAZELCAST	84.61	1,905.88	143.54
JUNIT	91.94	76.71	171.09
MAPDB	102.09	150.47	25.06
MCMMO	108.71	289.83	41.72
MISSION CONTROL T.	64.00	203.00	55.93
NEO4J	39.53	535.77	189.30
NETTY	83.65	411.60	48.96
ORIENTDB	99.21	568.76	179.30
ORYX	63.00	104.42	3.40
TITAN	51.35	65.91	59.85

The second column is the average number of days elapsed between the date of the release and the date of the bugs reported. We can see that these averages are quite high; the overall

average is 85 days. The third column is the average number of commits contributed to the project between the release commit and the date of the bug report. These values vary for each project because they depend on the developers' level of activity. For some projects (ELASTICSEARCH, HAZELCAST) it could mean thousands of modifications before the bug was reported. The more commits are performed, the higher the probability that the source code element became buggy after the release. The fourth column shows the average number of commits performed between the time when the bug was reported and when the fix was applied. These numbers are much smaller, which also demonstrates that bugs are fixed relatively fast. This fast corrective behavior causes before and after fix states to be less different for the BugHunter approach. Consequently, less difference in metric values makes building a precise prediction model more difficult.

The new BugHunter approach, however, is free from the uncertainty mentioned above because it only uses the buggy and the fully fixed states of the bug-related source code elements. This way, the produced bug dataset is more precise, hence it is more appropriate for machine learning. Therefore, we cannot clearly state that the traditional dataset is better, even despite the higher F-measure values. The difference between the values of the two datasets is around 0.10 at the method level, 0.21 at the class level, and 0.09 at the file level. Projecting method-level metrics to the class level achieved almost as high an F-measure value (0.7405) as in the traditional case (0.7831). The difference is only 0.04, yet it is on a much more precise dataset.

### ***The Author's Contributions***

The author has made several significant contributions to this research. Firstly, he designed and implemented the novel method presented for constructing bug databases. Additionally, he actively participated in the literature review and the process of defining criteria for the inclusion of projects in the BugHunter dataset. The author was responsible for executing the method, constructing the dataset, and gathering all the statistics related to the projects. Lastly, the author took a leading role in producing and analyzing the results obtained from the machine learning techniques utilized in this study. The publications related to this thesis point are:

- ◆ **Péter Gyimesi**, Gábor Gyimesi, Zoltán Tóth, and Rudolf Ferenc. Characterization of Source Code Defects by Data Mining Conducted on GitHub. In *15<sup>th</sup> International Conference on Computational Science and Its Applications (ICCSA 2015)*, Banff, AB, Canada, June 22–25, pages 47–62, LNCS, Volume 9159. Springer International Publishing, 2015.
- ◆ Zoltán Tóth, **Péter Gyimesi**, and Rudolf Ferenc. A Public Bug Database of GitHub Projects and its Application in Bug Prediction. In *16<sup>th</sup> International Conference on Computational Science and Its Applications (ICCSA 2016)*, Beijing, China, July 4–7, pages 625–638, LNCS, Volume 9789. Springer International Publishing, 2016.
- ◆ Rudolf Ferenc, **Péter Gyimesi**, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An Automatically Created Novel Bug Dataset and its Validation in Bug Prediction. *Journal of Systems and Software*, 2020, 169: 110691.



## II. Calculation of Process Metrics and their Bug Prediction Capabilities

Studies have shown that process metrics outperform product metrics in bug prediction. However, the use of process metrics remains limited, and there is a scarcity of research in this area. Therefore, in this thesis point, we aim to address these gaps by presenting an effective method for computing a variety of software process metrics.

We implemented the calculation of 22 process metrics for files, classes, and methods, along with the corresponding bug counts. To evaluate the effectiveness of process metrics, we selected five open-source Java projects from GitHub and generated databases for 5 release versions of each project. Finally, we evaluated their ability to predict bugs and compared them with the product metrics.

We investigated the following research questions:

- **Research Question 1:** Is the dataset containing process metrics usable for bug prediction purposes?
- **Research Question 2:** Which metrics are more effective for predicting software bugs: process metrics or product metrics?
- **Research Question 3:** What is the relation between product metrics and process metrics?

To answer the *first research question*, we evaluated the 11 algorithms on all 25 release versions and only used process metrics as predictors. The results obtained suggest that databases with process metrics are suitable for bug prediction purposes, with the RandomForest and DecisionTable methods performing the best. Specifically, we achieved F-measure values of 0.7997 (TITAN) at the class level, 0.8180 (MAPDB) at the file level, and 0.8185 (MAPDB) at the method level using the RandomForest algorithm. Furthermore, it is important to note that not all projects are equally capable of providing an appropriate training set for bug prediction. Additionally, on average, process metrics provide stronger indications of bug-proneness at higher levels of source code, such as classes or files.

Table 4: Comparison of the average F-measure values achieved with product metrics and process metrics

	File		Class		Method	
	<i>Product</i>	<i>Process</i>	<i>Product</i>	<i>Process</i>	<i>Product</i>	<i>Process</i>
ANTLR4	<b>0.7035</b>	0.6940	<b>0.7129</b>	0.6748	<b>0.7198</b>	0.5142
BROADLEAF C.	<b>0.6926</b>	0.6766	<b>0.7470</b>	0.6772	<b>0.7830</b>	0.6116
MAPDB	0.6420	<b>0.7860</b>	<b>0.6939</b>	0.6531	0.6709	<b>0.7328</b>
JUNIT	0.5647	<b>0.6805</b>	<b>0.7164</b>	0.6143	<b>0.5824</b>	0.5350
TITAN	0.5759	<b>0.6968</b>	0.6971	<b>0.7386</b>	0.6252	<b>0.7004</b>

To compare the predictive power of product and process metrics and to answer the *second research question*, we evaluated the dataset using the two different sets of metrics on the release version with the highest number of bug entries for each project. The achieved F-measure values are listed in Table 4. Looking at the results it can be concluded that creating bug prediction models based on process metrics at the file level generally yields better performance compared to

those using product metrics. However, at lower levels of source code, such as the class or method level, product metrics often outperform process metrics. An interesting observation is that the standard deviation of the resulting F-measure values is consistently lower for process metrics in almost all cases, with an average difference of 0.0375. Furthermore, the overall lowest achieved F-measure values are higher for process metrics in almost all cases, with an average difference of 0.0809. These findings suggest that bug prediction results obtained from process metrics are more robust and reliable, indicating their suitability for predicting bugs in source code.

Finally, to answer the *third research question*, we computed the Pearson correlation coefficient values between product and process metrics. The results indicate that process metrics correlate more with each other than with product metrics. There are higher correlation values ( $\sim 0.45$ ) between a few size-based product metrics and process metrics, but in general, there are no dominant correlation values between product and process metrics.

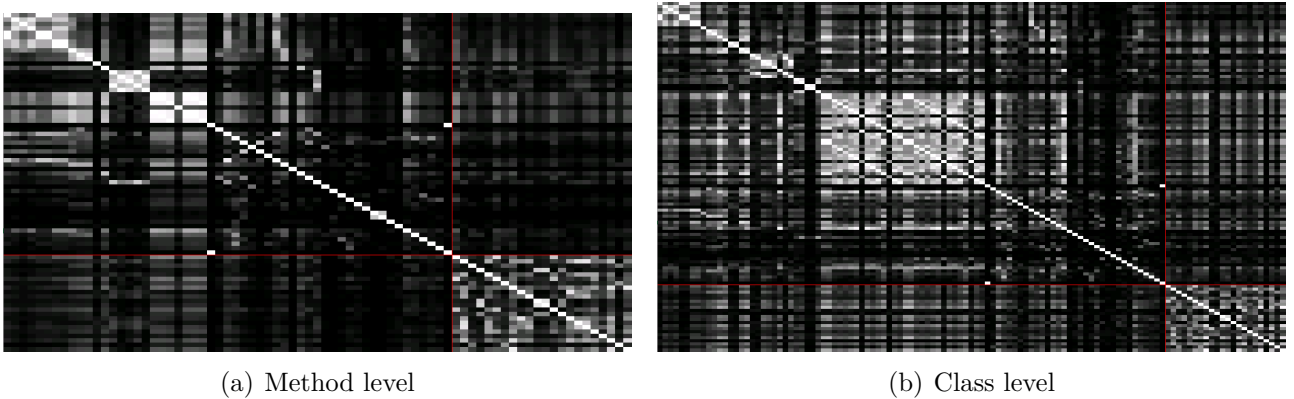


Figure 1: Correlation of product and process metrics

We illustrate the correlation matrices in Figure 1. 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). The process metrics are in the bottom right quarter. We can state that process metrics provide a distinct perspective in characterizing the source code elements compared to product metrics, as there are no strong correlations between the two types of metrics.

### ***The Author's Contributions***

This study is the author's independent work. He reviewed the literature, designed the methodology, and implemented the necessary tools. Subsequently, he generated the bug databases containing process metrics, conducted the evaluations, and drew conclusions. The publications related to this thesis point are:

- ◆ **Péter Gyimesi.** Automatic Calculation of Process Metrics and their Bug Prediction Capabilities. In *Acta Cybernetica*, pages 537–559, Volume 23, No 2, 2017.
- ◆ **Péter Gyimesi.** An open-source solution for automatic bug database creation. In *Proceedings of the 10<sup>th</sup> International Conference on Applied Informatics (ICAI 2017)*, Eger, Hungary, January 30–February 1, pages 111–119, 2017.

### III. A Public Dataset of JavaScript Bugs

Despite the extensive research on JavaScript (JS), a well-organized repository of labeled JS bugs was still missing. The presence of numerous JS implementations further complicated the task of creating a cohesive bugs benchmark. To address this gap, in this thesis point, we introduced a benchmark, called BUGSJS, comprising a total of 453 manually selected and validated JS-related bugs from 10 open-source JS projects. Additionally, we have developed a framework to automate research processes utilizing our benchmark. We conducted an investigation to determine whether the bug-fixing patterns for JS bugs align with existing classification schemes. By conducting both quantitative and qualitative analyses on the bugs, we constructed a taxonomy of JS bugs present in the benchmark, which, to our knowledge, is the first of its kind. We also explored the relationship between this categorization and bug-fixing patterns.

We investigated the following research questions:

- **Research Question 1:** Do the bug-fixing patterns for JavaScript bugs in BUGSJS match existing classification schemes?
- **Research Question 2:** How do the bug-fixing patterns in BUGSJS relate to our taxonomy of bugs?

To answer the *first research question*, we investigated the connection between the bug-fixing patterns for JS and existing classification schemes. Similar studies [31, 37, 15] in the past have explored patterns in bug-fixing changes within Java programs, suggesting that the existence of such patterns reveals certain code constructs that could indicate weak points in the source code, where developers are consistently more likely to introduce bugs. We used the categories suggested by Pan et al. [31] to classify bug-fixing patterns, which were originally related to Java bug fixes. The aim was to assess whether these categories generalize to JS or whether there are specific bug-fix patterns that emerge in JS. Our findings are listed in Table 5.

Table 5: Bug-fixing change types found in BUGSJS

	Category	Example	#
EXISTING	if-related	Changing <code>if</code> conditions	291
	Assignments	Modifying the RHS of an assignment	166
	Function calls	Adding or modifying an argument	152
	Class fields	Adding/removing class fields	22
	Function declarations	Modifying a function’s signature	94
	Sequences	Adding a function call to a sequence of calls, all with the same receiver	42
	Loops	Changing a loop’s predicate	5
	<code>switch</code> blocks	Adding/removing a switch branch	6
	<code>try</code> blocks	Introducing a new <code>try-catch</code> block	1
	NEW	<code>return</code> statements	Changing a <code>return</code> statement
Variable declaration		Declaring an existing variable	2
Initialization		Initializing a variable with empty object literal/array	3

Our analysis revealed that 88% of the bugs in our benchmark had fixes falling into one of the proposed categories. The most common fix patterns involved modifying an `if` statement. Interestingly, the same three categories were also found to be the most frequent in Java code. Furthermore, we identified three JS-specific recurring patterns.

By conducting both quantitative and qualitative analyses on the bugs, we constructed a taxonomy of JS bugs present in the benchmark. To answer the *second research question*, we compared the taxonomy with the bug-fixing patterns used to fix the bugs. We focused our analysis on the first three main bug categories of our taxonomy: incomplete feature implementation, incorrect feature implementation, and generic.

Table 6: Taxonomy and bug-fixing types

	AS	CF	IF	JS	LP	MC	MD	None	SQ	SW	TY
INCOMPLETE FEATURE IMPLEMENTATION											
configuration processing	1	0	9	1	0	2	1	1	1	0	0
missing type check	1	0	0	0	0	1	0	0	0	0	0
error handling	1	0	12	1	0	3	4	3	1	0	1
callbacks	1	0	4	0	0	2	0	0	1	0	0
incomplete data processing	4	2	9	1	0	11	7	3	1	1	0
incomplete output message	0	0	1	0	0	3	0	0	0	0	0
missing input validation	29	8	53	4	1	11	22	2	2	2	0
empty input parameters	1	0	3	0	0	1	0	0	1	0	0
missing handling of spaces	2	0	2	1	0	1	2	0	0	0	0
missing handling of special characters	6	0	5	1	0	1	0	0	0	0	0
missing null check	0	0	5	2	0	0	0	0	0	0	0
missing type check	18	2	39	2	1	10	10	2	1	1	0
INCORRECT FEATURE IMPLEMENTATION											
configuration processing	5	0	3	1	1	2	1	3	0	0	0
incorrect data processing	24	2	39	5	1	27	13	3	9	1	0
incorrect initialization	5	0	0	0	0	4	0	3	5	0	0
incorrect type comparison	0	0	1	0	0	0	0	0	0	0	0
incorrect filepath	4	0	4	2	0	4	1	0	2	0	0
incorrect handling of regex expressions	13	0	4	0	0	6	3	0	0	0	0
incorrect input validation	25	1	56	12	0	27	15	2	4	1	0
empty input parameters	0	0	1	0	0	0	0	1	0	0	0
incorrect handling of special characters	9	0	7	0	1	12	9	1	3	0	0
unnecessary type check	5	0	7	0	0	0	2	0	0	0	0
incorrect output	1	0	0	3	0	5	0	1	0	0	0
incorrect output message	2	2	8	1	0	10	2	3	0	0	0
performance	1	0	5	0	0	1	0	0	7	0	0
GENERIC											
data processing	1	0	1	2	0	2	0	1	1	0	0
loop statement											
incorrect loop statement	0	0	0	0	0	0	0	1	0	0	0
missing type conversion	2	0	1	0	0	0	0	0	1	0	0
return statement											
incorrect return statement	0	0	0	0	0	1	0	0	0	0	0
missing return statement	0	0	1	0	0	0	0	2	0	0	0
typo	1	0	2	0	0	2	1	0	0	0	0
variable initialization											
incorrect variable initialization	3	1	0	0	0	0	0	0	0	0	0
missing variable initialization	0	4	0	5	0	2	0	0	2	0	0
PERFECTIVE MAINTENANCE	1	0	9	1	0	1	1	0	0	0	0

Table 6 provides statistics about the occurrence of each bug-fix type corresponding to the bug types. The table can serve to analyze the emergence of correlations between bug-fix types and bug types. Overall, the most common bug-fixes in BUGSJS are `if`-related (291), the second most common are assignment-related (166), and the third most common are method-call-related (152) bug-fixes. These bug-fix types are mostly related to the most prominent bug categories, namely missing input validation, incorrect input validation, and incorrect data processing. Another correlation is that assignment-related fixes are also the preferred way to fix regexes. These are perhaps the only correlations between bug-fix types and bug types that are observable in our benchmark.

This comprehensive analysis demonstrates that the dataset encompasses a wide range of bugs and can serve as a reliable benchmark for conducting reproducible research in software analysis and testing.

### ***The Author’s Contributions***

During this research, the author actively participated in designing the research plan and implementing the framework, which included the benchmark. He was responsible for collecting and analyzing JavaScript projects, selecting suitable bugs, and extracting relevant bug fixes. Additionally, he took part in manually validating the bugs. Throughout the analysis of the bugs, the author actively contributed to examining bug-fixing patterns, as well as creating and validating the bug taxonomy. Finally, the author took a leading role in analyzing the correlation between the bug taxonomy and the bug-fixing patterns. The publications related to this thesis point are:

- ◆ **Péter Gyimesi**, Béla Vancsics, Andrea Stocco, Davood Mazinianian, Arpád Beszédes, Rudolf Ferenc and Ali Mesbah. BugsJS: A Benchmark of JavaScript Bugs. In *12<sup>th</sup> IEEE Conference on Software Testing, Validation and Verification (IEEE ICST 2019), Xi’an, China, April 22–27*, pages 90–101, IEEE, Volume 1. IEEE Computer Society Press, 2019.
- ◆ **Péter Gyimesi**, Béla Vancsics, Andrea Stocco, Davood Mazinianian, Arpád Beszédes, Rudolf Ferenc and Ali Mesbah. BugsJS: A Benchmark and Taxonomy of JavaScript Bugs. *Journal of Software Testing, Verification and Reliability (STVR 2021)*, John Wiley & Sons Publishing. 38 pages.
- ◆ Béla Vancsics, **Péter Gyimesi**, Andrea Stocco, Davood Mazinianian, Arpád Beszédes, Rudolf Ferenc and Ali Mesbah. Poster: Supporting JavaScript Experimentation with BugsJS. In *12<sup>th</sup> IEEE Conference on Software Testing, Validation and Verification (IEEE ICST 2019), Poster Track, Xi’an, China, April 22–27*, pages 375–378, IEEE, Volume 1. IEEE Computer Society Press, 2019.

# Summary

We showed that previous datasets created using traditional approaches contain uncertainties. Therefore, we developed a novel method to capture the before-fix and after-fix states of buggy source code elements, resulting in a dataset with reduced uncertainties. We carried out empirical evaluations and found that this dataset can be useful for bug prediction. We also conducted an experiment to compare the bug prediction capabilities of method-level metrics projected to the class level with those of class-level metrics. Our findings indicate that projecting method-level metrics to the class level enhances their predictive power for bug prediction.

Despite previous studies showing that process metrics outperform product metrics in bug prediction, the use of process metrics is not widespread, and there is a scarcity of research on process metrics. To address this, we developed a method to efficiently compute process metrics for files, classes, and methods using a graph database. We confirmed that bug databases with process metrics are suitable for bug prediction. We also compared process and product metrics and found that the use of process metrics in bug prediction yields more stable results. Moreover, process metrics provide a different perspective on characterizing source code elements compared to product metrics.

Lastly, we created a benchmark of real, manually-verified JavaScript bugs, along with a framework to automate research processes. We analyzed the bugs included in the benchmark and found that most have fixes that fall into existing bug-fixing patterns. We also created a bug taxonomy and investigated whether this categorization relates to bug-fixing patterns. This benchmark serves as a reliable source for conducting reproducible research in software analysis and testing.

Table 7 summarizes the main publications and how they relate to our thesis points.

<b>№</b>	[4]	[7]	[1]	[2]	[3]	[5]	[6]	[8]
I.	◆	◆	◆					
II.				◆	◆			
III.						◆	◆	◆

Table 7: Thesis contributions and supporting publications

# Acknowledgements

I could not have reached this point without assistance, and I want to express my gratitude for the support I received. I am immensely thankful to my supervisor, Dr. Rudolf Ferenc, for his invaluable guidance and profound insights. He has shown me that with dedication and perseverance, I can overcome any challenge and achieve my goals. His mentorship has been extremely valuable, and I am truly grateful for his belief in my potential and the opportunities he has opened for me. I would also like to extend my sincere thanks to Dr. Tibor Gyimóthy, the former head of the Department of Software Engineering, for his support in my journey to become

a PhD student. A special acknowledgment goes to my coworkers, Dr. Zoltán Tóth, Béla Vancsics, and Gábor Gyimesi, whose contributions and unwavering support have played a significant role in my progress. I would also like to express my deep appreciation to all of my co-authors for their valuable contributions, as well as to Edit Szűcs for her assistance in providing stylistic and grammatical comments on this thesis.

*Péter Gyimesi, 2023*

# References

## Corresponding Publications of the Author

- [1] Rudolf Ferenc, Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software*, 169:110691, 2020.
- [2] Péter Gyimesi. Automatic calculation of process metrics and their bug prediction capabilities. *Acta Cybernetica*, 23(2):537–559, 2017.
- [3] Péter Gyimesi. An open-source solution for automatic bug database creation. In *Proceedings of the 10th International Conference on Applied Informatics*, page 111–119, 2017.
- [4] Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Rudolf Ferenc. 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.
- [5] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: A benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.
- [6] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark and taxonomy of javascript bugs. *Software Testing, Verification And Reliability*, 31(4):e1751, 2021.
- [7] 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.
- [8] Béla Vancsics, Péter Gyimesi, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Poster: Supporting javascript experimentation with bugsjs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 375–378. IEEE, 2019.



## Other Publications

- [9] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. Repairing event race errors by controlling nondeterminism. In *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017.
- [10] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 2015.
- [11] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016.
- [12] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of International Conference on Software Engineering*, 2005.
- [13] P. Bangcharoensap, A. Ihara, Y. Kamei, and K. Matsumoto. Locating source code to be fixed based on initial bug reports - a case study on the eclipse project. In *Empirical Software Engineering in Practice (IWESEP), 2012 Fourth International Workshop on*, pages 10–15, Oct 2012.
- [14] Marina Billes, Anders Møller, and Michael Pradel. Systematic black-box analysis of collaborative web applications. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.
- [15] E. C. Campos and M. d. A. Maia. Common bug-fix patterns: A large-scale observational study. In *Proc. of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017.
- [16] Cagatay Catal. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.
- [17] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.
- [18] C. Couto, C. Silva, M.T. Valente, R. Bigonha, and N. Anquetil. Uncovering causal relationships between software metrics and bugs. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 223–232, March 2012.
- [19] Marco D’Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 31 – 41, 2010.
- [20] Markus Ermuth and Michael Pradel. Monkey see, monkey do: Effective generation of GUI tests with inferred macro events. In *Proc. of 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016.
- [21] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Proc. of International Symposium on Software Reliability Engineering*, 2014.

- [22] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.
- [23] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5), 2011.
- [24] Bryant Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.
- [25] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proc. of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.
- [26] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. *MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101, 2014.
- [27] Yan Ma, Lan Guo, and Bojan Cukic. A statistical framework for the prediction of fault-proneness. *Advances in Machine Learning Application in Software Engineering*, Idea Group Inc, pages 237–265, 2006.
- [28] Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. Feedback-directed instrumentation for deployed JavaScript applications. In *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016.
- [29] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. A Study of Causes and Consequences of Client-Side JavaScript Bugs. *IEEE Transactions on Software Engineering*, 43(2):128–144, February 2017.
- [30] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340–355, 2005.
- [31] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, June 2009.
- [32] Adam Porter, Richard W Selby, et al. Empirically guided software development using metric-based classification trees. *Software, IEEE*, 7(2):46–54, 1990.
- [33] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.
- [34] Emad Shihab, Zhen Ming Jiang, Walid M Ibrahim, Bram Adams, and Ahmed E Hassan. 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.
- [35] J. Wang, W. Dou, C. Gao, Y. Gao, and J. Wei. Context-based event trace reduction in client-side JavaScript applications. In *Proc. of International Conference on Software Testing, Verification and Validation (ICST)*, 2018.

- [36] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in Node.js. In *Proc. of International Conference on Automated Software Engineering*, 2017.
- [37] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proc. of 37th International Conference on Software Engineering (ICSE)*, pages 913–923, 2015.
- [38] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. *Software Engineering (ICSE), 2012 34th International Conference on*, 2012.
- [39] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *Software Engineering, IEEE Transactions on*, 32(10):771–789, 2006.

# Declaration

In the PhD dissertation of **Péter Gyimesi** entitled *A Novel Bug Prediction Dataset, Process Metrics and A Public Dataset of JavaScript Bugs*, **Péter Gyimesi's** contribution was decisive in the following results:

## 1. A Novel Bug Prediction Dataset and its Validation

- Designing the novel bug prediction dataset [3]
- Implementing the designed tools [1], [2], [3]
- Constructing the dataset and gathering statistics [1], [2], [3]
- Analyzing the result of the study [3]

## 2. Calculation of Process Metrics and their Bug Prediction Capabilities

- Reviewing the existing related studies [4], [5]
- Designing the method to calculate process metrics [4], [5]
- Implementing the designed tools [4], [5]
- Producing the result of the machine learning techniques [4], [5]
- Analyzing the result of the study [4], [5]

## 3. A Public Dataset of JavaScript Bugs

- Collecting and analyzing JavaScript projects for the benchmark [6], [7], [8]
- Selecting suitable bugs and extracting related bug-fixes [6], [7], [8]
- Analyzing the relationship between the bug taxonomy and the bug-fixing patterns [7]

These results cannot be used to obtain an academic research degree, other than the submitted PhD thesis of **Péter Gyimesi**.

In the PhD dissertation of **Péter Gyimesi** entitled *A Novel Bug Prediction Dataset, Process Metrics and A Public Dataset of JavaScript Bugs*, **Péter Gyimesi** and the corresponding coauthors share the following joint and undividable contributions:

## 1. A Novel Bug Prediction Dataset and its Validation

- Reviewing the existing related studies [1], [2], [3]
- Defining criteria for the inclusion of projects [1], [2], [3]
- Producing the result of the machine learning techniques [1], [2], [3]


## 3. A Public Dataset of JavaScript Bugs

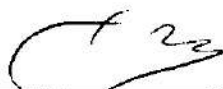
- Designing and implementing the framework and the benchmark [6], [7], [8]
- Manually validating the bugs selected for the benchmark [6], [7], [8]
- Extracting and analyzing the bug-fixing patterns [6], [7]
- Creating and validating the bug taxonomy [7]

Referenced publications:

- [1] **Péter Gyimesi**, Gábor Gyimesi, Zoltán Tóth, and Rudolf Ferenc. Characterization of Source Code Defects by Data Mining Conducted on GitHub. In 15th International Conference on Computational Science and Its Applications (ICCSA 2015), Banff, AB, Canada, June 22–25, pages 47–62, LNCS, Volume 9159. Springer International Publishing, 2015.
- [2] Zoltán Tóth, **Péter Gyimesi**, and Rudolf Ferenc. A Public Bug Database of GitHub Projects and its Application in Bug Prediction. In 16th International Conference on Computational Science and Its Applications (ICCSA 2016), Beijing, China, July 4–7, pages 625–638, LNCS, Volume 9789. Springer International Publishing, 2016.
- [3] Rudolf Ferenc, **Péter Gyimesi**, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An Automatically Created Novel Bug Dataset and its Validation in Bug Prediction. *Journal of Systems and Software*, 2020, 169: 110691.
- [4] **Péter Gyimesi**. Automatic Calculation of Process Metrics and their Bug Prediction Capabilities. In *Acta Cybernetica*, pages 537–559, Volume 23, No 2, 2017
- [5] **Péter Gyimesi**. An open-source solution for automatic bug database creation. In Proceedings of the 10th International Conference on Applied Informatics (ICAI 2017), Eger, Hungary, January 30–February 1, pages 111–119, 2017.
- [6] **Péter Gyimesi**, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. BugsJS: A Benchmark of JavaScript Bugs. In 12th IEEE Conference on Software Testing, Validation and Verification (IEEE ICST 2019), Xi'an, China, April 22–27, pages 90–101, IEEE, Volume 1. IEEE Computer Society Press, 2019.
- [7] **Péter Gyimesi**, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. BugsJS: A Benchmark and Taxonomy of JavaScript Bugs. *Journal of Software Testing, Verification and Reliability (STVR 2021)*, John Wiley & Sons Publishing
- [8] Béla Vancsics, **Péter Gyimesi**, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Poster: Supporting JavaScript Experimentation with BugsJS. In 12th IEEE Conference on Software Testing, Validation and Verification (IEEE ICST 2019), Poster Track, Xi'an, China, April 22–27, pages 375–378, IEEE, Volume 1. IEEE Computer Society Press, 2019.


Szeged, 20 November 2023

  
Péter Gyimesi

  
Dr. habil. Rudolf Ferenc

The head of the Doctoral School of Computer Science declares that the declaration above was sent to all of the coauthors and none of them raised any objections against it.

Szeged, 20 November 2023

  
  
Dr. Márk Jelasity