# Too trivial to test? An inverse view on defect prediction to identify methods with low fault risk

**Rainer Niedermayr** Corresp., [1, 2] , **Tobias Röhm** [1] , **Stefan Wagner** [2]

[1] CQSE GmbH, München, Germany
[2] Institute of Software Technology, University of Stuttgart, Stuttgart, Germany

Corresponding Author: Rainer Niedermayr
Email address: niedermayr@cqse.eu

**Background.** Test resources are usually limited and therefore it is often not possible to completely test an application before a release. To cope with the problem of scarce resources, development teams can apply defect prediction to identify fault-prone code regions. However, defect prediction tends to low precision in cross-project prediction scenarios.

**Aims.** We take an inverse view on defect prediction and aim to identify methods that can be deferred when testing because they contain hardly any faults due to their code being "trivial". We expect that characteristics of such methods might be project-independent, so that our approach could improve cross-project predictions.

**Method.** We compute code metrics and apply association rule mining to create rules for identifying methods with low fault risk. We conduct an empirical study to assess our approach with six Java open-source projects containing precise fault data at the method level.

**Results.** Our results show that inverse defect prediction can identify approx. 32-44% of the methods of a project to have a low fault risk; on average, they are about six times less likely to contain a fault than other methods. In cross-project predictions with larger, more diversified training sets, identified methods are even eleven times less likely to contain a fault.

**Conclusions.** Inverse defect prediction supports the efficient allocation of test resources by identifying methods that can be treated with less priority in testing activities and is well applicable in cross-project prediction scenarios.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

# Too Trivial To Test? An Inverse View on Defect Prediction to Identify Methods with Low Fault Risk

**Rainer Niedermayr[1], Tobias Röhm[2], and Stefan Wagner[3]**

[1,2]**CQSE GmbH, Garching b. München, Germany**
[1,3]**University of Stuttgart, Stuttgart, Germany**

Corresponding author:
Rainer Niedermayr[1]

Email address: niedermayr@cqse.eu

## ABSTRACT

***Background.*** Test resources are usually limited and therefore it is often not possible to completely test an application before a release. To cope with the problem of scarce resources, development teams can apply defect prediction to identify fault-prone code regions. However, defect prediction tends to low precision in cross-project prediction scenarios. ***Aims.*** We take an inverse view on defect prediction and aim to identify methods that can be deferred when testing because they contain hardly any faults due to their code being "trivial". We expect that characteristics of such methods might be project-independent, so that our approach could improve cross-project predictions. ***Method.*** We compute code metrics and apply association rule mining to create rules for identifying methods with low fault risk. We conduct an empirical study to assess our approach with six Java open-source projects containing precise fault data at the method level. ***Results.*** Our results show that inverse defect prediction can identify approx. 32–44% of the methods of a project to have a low fault risk; on average, they are about six times less likely to contain a fault than other methods. In cross-project predictions with larger, more diversified training sets, identified methods are even eleven times less likely to contain a fault. ***Conclusions.*** Inverse defect prediction supports the efficient allocation of test resources by identifying methods that can be treated with less priority in testing activities and is well applicable in cross-project prediction scenarios.

## 1 INTRODUCTION

In a perfect world, it would be possible to *completely* test every new version of a software application before it was deployed into production. In practice, however, software development teams often face a problem of scarce test resources. Developers are busy implementing features and bug fixes, and may lack time to develop enough automated unit tests to comprehensively test new code [Ostrand et al. (2005); Menzies and Di Stefano (2004)]. Furthermore, testing is costly and, depending on the criticality of a system, it may not be cost-effective to expend equal test effort to all components [Zhang et al. (2007)]. Hence, development teams need to prioritize and limit their testing scope by restricting the code regions to be tested [Menzies et al. (2003); Bertolino (2007)]. To cope with the problem of scarce test resources, development teams aim to test code regions that have the best cost-benefit ratio regarding fault detection. To support development teams in this activity, defect prediction has been developed and studied extensively in the last decades [Hall et al. (2012); D'Ambros et al. (2012); Catal (2011)]. Defect prediction identifies code regions that are likely to contain a fault and should therefore be tested [Menzies et al. (2007); Weyuker and Ostrand (2008)].

This paper suggests, implements, and evaluates another view on defect prediction: inverse defect prediction (IDP). The idea behind IDP is to identify code artifacts (e.g., methods) that are so *trivial* that they contain hardly any faults and thus can be deferred or ignored in testing. Like traditional defect prediction, IDP also uses a set of metrics that characterize artifacts, applies transformations to pre-process metrics, and uses a machine-learning classifier to build a prediction model. The difference rather lies in the predicted classes. While defect prediction classifies an artifact either as *buggy or non-buggy*, IDP identifies methods that exhibit a *low fault risk* (LFR) with high certainty and does not make an assumption about the remaining methods, for which the fault risk is at least medium or cannot be reliably determined. As a consequence, the objective of the prediction also differs. Defect prediction aims to achieve a high recall, such that as many faults as possible can be detected, and a high precision, such that only few false positives occur. In contrast, IDP aims to achieve high precision to ensure that low-fault-risk methods contain indeed hardly any faults, but it does not necessarily seek to predict all non-faulty methods. Still, IDP needs to achieve a certain recall such that a reasonable reduction potential arises when treating LFR methods with a lower priority in QA activities.

**Research goal:** We want to study whether IDP can reliably identify code regions that exhibit only a low fault risk, whether ignoring such code regions—as done silently in defect prediction—is a good idea, and whether IDP can be used in cross-project predictions.

To implement IDP, we calculated code metrics for each method of a code base and trained a classifier for methods with low fault risk using association rule mining. To evaluate IDP, we performed an empirical study with the Defects4J dataset [Just et al. (2014)] consisting of real faults from six open-source projects. We applied static code analysis and classifier learning on these code bases and evaluated the results. We hypothesize that IDP can be used to pragmatically address the problem of scarce test resources. More specifically, we hypothesize that a generalized IDP model can be used to identify code regions that can be deferred when writing automated tests if none yet exist, as is the situation for many legacy code bases.

**Contributions:** 1) The idea of an inverse view on defect prediction: While defect prediction has been studied extensively in the last decades, it has always been employed to identify code regions with *high* fault risk. To the best of our knowledge, the present paper is the first to study the identification of code regions with *low* fault risk explicitly. 2) An empirical study about the performance of IDP on real open-source code bases. 3) An extension to the Defects4J dataset [Just et al. (2014)]: To improve data quality and enable further research—reproduction in particular—we provide code metrics for all methods in the code bases and an indication whether they were changed in a bug-fix patch, a list of methods that changed in bug fixes only to preserve API compatibility, and association rules to identify low-fault-risk methods.

The remainder of this paper is organized as follows. Section 2 provides background information about association rule mining. Section 3 discusses related work. Section 4 describes the IDP approach, i.e., the computation of the metrics for each method, the data pre-processing, and the association rule mining to identify methods with low fault risk. Afterwards, Section 5 summarizes the design and results of the IDP study with the Defects4J dataset. Then, Section 6 discusses the study's results, implications, and threats to validity. Finally, Section 7 summarizes the main findings and sketches future work.

## 2 ASSOCIATION RULE MINING

Association rule mining is a technique for identifying relations between variables in a large dataset and was introduced by Agrawal et al. in 1993 [Agrawal et al. (1993)]. A dataset contains *transactions* consisting of a set of *items* that are binary attributes. An *association rule* represents a logical implication of the form $\{\ antecedent\ \} \rightarrow \{\ consequent\ \}$ and expresses that the *consequent* is likely to apply if the *antecedent* applies. Antecedent and consequent both consist of a set of items and are disjoint. The *support* of a rule expresses the proportion of the transactions that contain both antecedent and consequent out of all transactions. It is related to the significance of the itemset [Simon et al. (2011)]. The *confidence* of a rule expresses the proportion of the transactions that contain both antecedent and consequent out of all transactions that contain the antecedent. It can be considered as the precision [Simon et al. (2011)]. A rule is *redundant* if a more general rule with the same or a higher confidence value exists [Bayardo et al. (1999)].

Association Rule Mining has been successfully applied in defect prediction studies [Song et al. (2006); Czibula et al. (2014); Ma et al. (2010); Zafar et al. (2012)]. A major advantage of association rule mining is the natural comprehensibility of the rules [Simon et al. (2011)]. Other commonly used machine-learning algorithms for defect prediction, such as support vector machines (SVM) or Naive Bayes classifiers, generate black-box models, which lack interpretability. Even decision trees can be difficult to interpret due to the subtree-replication problem [Simon et al. (2011)]. Another advantage of association rule mining is that the gained rules implicitly extract high-order interactions among the predictors.



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

## 3 RELATED WORK

Defect prediction is an important research area that has been extensively studied [Hall et al. (2012); Catal and Diri (2009)]. Defect prediction models use code metrics [Menzies et al. (2007); Nagappan et al. (2006); D'Ambros et al. (2012); Zimmermann et al. (2007)], change metrics [Nagappan and Ball (2005); Hassan (2009); Kim et al. (2007)], or a variety of further metrics (such as code ownership [Bird et al. (2011); Rahman and Devanbu (2011)], developer interactions [Meneely et al. (2008); Lee et al. (2011)], dependencies to binaries [Zimmermann and Nagappan (2008)], mutants [Bowes et al. (2016)], code smells [Palomba et al. (2016)]) to predict code areas that are especially defect-prone. Such models allow software engineers to focus quality-assurance efforts on these areas and thereby support a more efficient resource allocation [Menzies et al. (2007); Weyuker and Ostrand (2008)].

Defect prediction is usually performed at the component, package or file level [Nagappan and Ball (2005); Nagappan et al. (2006); Bacchelli et al. (2010); Scanniello et al. (2013)]. Recently, more fine-grained prediction models have been proposed to narrow down the scope for quality-assurance activities. Kim et al. presented a model to classify software changes [Kim et al. (2008)]. Hata et al. applied defect prediction at the method level and showed that fine-grained prediction outperforms coarse-grained prediction at the file or package level if efforts to find the faults are considered [Hata et al. (2012)]. Giger et al. also investigated prediction models at the method level [Giger et al. (2012)] and concluded that a Random Forest model operating on change metrics can achieve good performance. More recently, Pascarella et al. replicated this study and confirmed the results [Pascarella et al. (2018)]. However, they reported that a more realistic inter-release evaluation of the models shows a dramatic drop in performance with results close to that of a random classifier and concluded that method-level bug prediction is still an open challenge [Pascarella et al. (2018)]. It is considered difficult to achieve sufficiently good data quality at the method level [Hata et al. (2012); Shippey et al. (2016)]; publicly available datasets have been provided in [Shippey et al. (2016)], [Just et al. (2014)], and [Giger et al. (2012)].

Cross-project defect prediction predicts defects in projects for which no historical data exists by using models trained on data of other projects [Zimmermann et al. (2009); Xia et al. (2016)]. He et al. investigated the usability of cross-project defect prediction [He et al. (2012)]. They reported that cross-project defect prediction works only in few cases and requires careful selection of training data. Zimmermann et al. also provided empirical evidence that cross-project prediction is a serious problem [Zimmermann et al. (2009)]. They stated that projects in the same domain cannot be used to build accurate prediction models without quantifying, understanding, and evaluating process, data and domain. Similar findings were obtained by Turhan et al., who investigated the use of cross-company data for building prediction models [Turhan et al. (2009)]. They found that models using cross-company data can only be "useful in extreme cases such as mission-critical projects, where the cost of false alarms can be afforded" and suggested using within-company data if available. While some recent studies reported advances in cross-project defect prediction [Xia et al. (2016); Zhang et al. (2016); Xu et al. (2018)], it is still considered as a challenging task.

Our work differs from the above-mentioned work in the target setting: we do not predict artifacts that are fault-prone, but instead identify artifacts (methods) that are very unlikely to contain any faults. While defect prediction aims to detect as many faults as possible (without too many false positives), and thus strives for a high recall [Mende and Koschke (2009)], our IDP approach strives to identify those methods that are not fault-prone to a high certainty. Therefore, we optimized our approach towards the precision in detecting low-fault-risk methods and considered the recall as less important. To the best of our knowledge, this is the first work to study low-fault-risk methods. Moreover, as far as we know, cross-project prediction has not yet been applied at the method level. To perform the classification, we applied association rule mining. Association rule mining has previously been applied with success in defect prediction [Song et al. (2006); Morisaki et al. (2007); Czibula et al. (2014); Ma et al. (2010); Karthik and Manikandan (2010); Zafar et al. (2012)].

## 4 IDP APPROACH

This section describes the inverse defect prediction approach, which identifies low-fault-risk (LFR) methods. The approach comprises the computation of source-code metrics for each method, the data pre-processing before the mining, and the association rule mining. Figure 1 illustrates the steps.



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

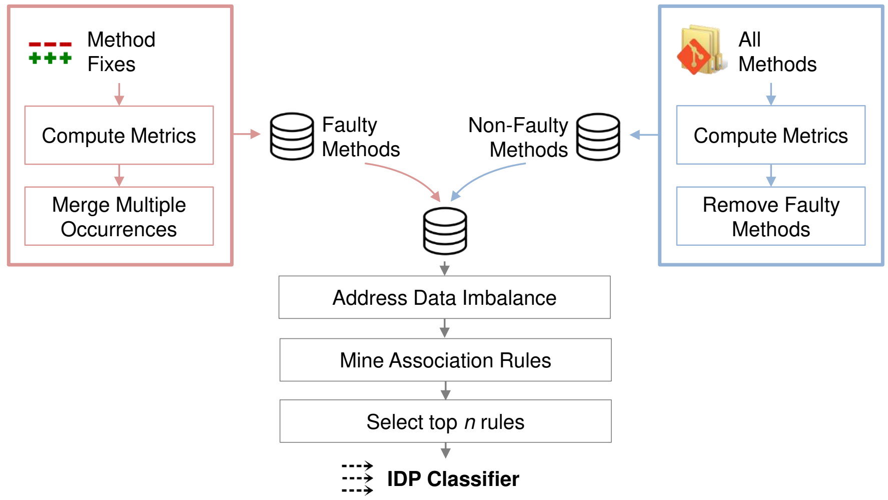


**Figure 1.** Overview of the approach. Metrics for faulty methods are computed at the faulty state; metrics for non-faulty methods are computed at the state of the last bug-fix commit.

### 4.1 Metric Computation

Like defect prediction models, IDP uses metrics to train a classifier for identifying low-fault-risk methods. For each method, we compute the source-code metrics listed in Table 1 that we considered relevant to judge whether a method is trivial. They comprise established length and complexity metrics used in defect prediction, metrics regarding occurrences of programming-language constructs, and categories describing the purpose of a method.

*SLOC* is the number of source lines of code, i.e., LOC without empty lines and comments. *Cyclomatic Complexity (CC)* corresponds to the metric proposed by McCabe [McCabe (1976)]. Despite this metric being controversial [Shepperd (1988); Hummel (2014)]—due to the fact that it is not actionable, difficult to interpret, and high values do not necessarily translate to low readability—it is commonly used as variable in defect prediction [Menzies et al. (2004); Zimmermann et al. (2007); Menzies et al. (2002)]. Furthermore, a low number of paths through a method could be relevant for identifying low-fault-risk methods. *Maximum nesting depth* corresponds to the "maximum number of encapsulated scopes inside the body of the method" [ndepend (2017)]. Deeply nested code is more difficult to understand, therefore, it could be more fault-prone. *Maximum method chaining* expresses the maximum number of chain elements of a method invocation. We consider a method call to be chained if it is directly invoked on the result from the previous method invocation. The value for a method is zero if it does not contain any method invocations, one if no method invocation is chained, or otherwise the maximum number of chain elements (e.g., two for `getId().toString()`, three for `getId().toString().subString(1)`). *Unique variable identifiers* counts the distinct names of variables that are used within the method. The following metrics, metrics M6 to M31, count the occurrences of the respective Java language construct [Gosling et al. (2013)].

Next, we derive further metrics from the existing ones. They are redundant, but correlated metrics do not have any negative effects on association rule mining (except on the computation time) and may improve the results for the following reason: if an item generated from a metric is not frequent, rules with this item will be discarded because they cannot achieve the minimum support; however, an item for a more general metric may be more frequent and survive. The derived metrics are:

- *All Conditions*, which sums up *If Conditions*, *Switch-Case Blocks*, and *Ternary Operations* (M16 + M27 + M29)
- *All Arithmetic Operations*, which sums up *Incrementations*, *Decrementations*, and *Arithmetic Infix Operations* (M7 + M8)

Furthermore, we compute to which of the following categories a method belongs (a method can belong to zero, one, or more categories):

**Table 1.** Computed metrics for each method.

| | Metric Name | Type |
|---|---|---|
| M1 | Source Lines of Code (SLOC) | length |
| M2 | Cyclomatic Complexity (CC) | complexity |
| M3 | Max. Nesting Depth | max. value |
| M4 | Max. Method Chaining | max. value |
| M5 | Unique Variable Identifiers | unique count |
| M6 | Anonymous Class Declarations | count |
| M7 | Arithmetic In- or Decrementations | count |
| M8 | Arithmetic Infix Operations | count |
| M9 | Array Accesses | count |
| M10 | Array Creations | count |
| M11 | Assignments | count |
| M12 | Boolean Operators | count |
| M13 | Cast Expressions | count |
| M14 | Catch Clauses | count |
| M15 | Comparison Operators | count |
| M16 | If Conditions | count |
| M17 | Inner Method Declarations | count |
| M18 | Instance-of Checks | count |
| M19 | Instantiations | count |
| M20 | Loops | count |
| M21 | Method Invocations | count |
| M22 | Null Checks | count |
| M23 | Null Literals | count |
| M24 | Return Statements | count |
| M25 | String Literals | count |
| M26 | Super-Method Invocations | count |
| M27 | Switch-Case Blocks | count |
| M28 | Synchronized Blocks | count |
| M29 | Ternary Operations | count |
| M30 | Throw Statements | count |
| M31 | Try Blocks | count |
| M32 | All Conditions | count |
| M33 | All Arithmetic Operations | count |
| M34 | Is Constructor | boolean |
| M35 | Is Setter | boolean |
| M36 | Is Getter | boolean |
| M37 | Is Empty Method | boolean |
| M38 | Is Delegation Method | boolean |
| M39 | Is ToString Method | boolean |



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

- *Constructors:* Special methods that create and initialize an instance of a class. They might be less fault-prone because they often only set class variables or delegate to another constructor.
- *Getters:* Methods that return a class variable. They usually consist of a single statement and can be generated by the IDE.
- *Setters:* Methods that set the value of a class variable. They usually consist of a single statement and can be generated by the IDE.
- *Empty Methods:* Non-abstract methods without any statements. They often exist to meet an implemented interface, or because the default logic is to do nothing and is supposed to be overridden in certain sub-classes.
- *Delegation Methods:* Methods that delegate the call to another method with the same name and further parameters. They often do not contain any logic besides the delegation.
- *ToString Methods:* Implementations of Java's `toString` method. They are often used only for debugging purposes and can be generated by the IDE.

Note that we only use source-code metrics and do not consider process metrics. This is because we want to identify methods that exhibit a low fault risk due to their *code.*

Association rule mining computes frequent itemsets from categorical attributes; therefore, our next step is to discretize the numerical metrics. (In defect prediction, discretization is also applied to the metrics: Shivaji et al. [Shivaji et al. (2013)] and McCallum et al. [McCallum and Nigam (1998)] reported that binary values can yield better results than using counts when the number of features is low.) We discretize as follows:

- For each of the metrics M1 to M5, we inspect their distribution and create three classes. The first class is for metric values until the first tertile, the second class for values until the second tertile, and the third class for the remaining values.
- For all count metrics (including the derived ones), we create a binary "has-no"-metric, which is true if the value is zero, e.g., $\mathit{CountLoops} = 0 \quad \Rightarrow \quad \mathit{NoLoops} = \texttt{true}$.
- For the method categories (setter, getter, ...), no transformation is necessary as they are already binary.

### 4.2 Data Pre-Processing

At this point, we assume that we have a list of faulty methods with their metrics at the faulty state (the list may contain a method multiple times if it was fixed multiple times) and a list of all methods. Faulty methods can be obtained by identifying methods that were changed in bug-fix commits [Zimmermann et al. (2007); Giger et al. (2012); Shippey et al. (2016)]; we describe in Section 5.3 how we extracted faulty methods from the Defects4J dataset.

Prior to applying the mining algorithm, we have 1) to address faulty methods with multiple occurrences, 2) to create a unified list of faulty and non-faulty methods, and 3) to tackle dataset imbalance.

1) A method may be fixed multiple times; in this case, a method appears multiple times in the list of the faulty methods. However, each method should have the same weight and should therefore be considered only once. Consequently, we consolidate multiple occurrences of the same method: we replace all occurrences by a new instance and apply majority voting to aggregate the binary metric values. It is common practice in defect prediction to have a single instance of every method with a flag that indicates whether a method was faulty at least once [Menzies et al. (2010); Giger et al. (2012); Shippey et al. (2016); Mende and Koschke (2009)].

2) To create a unified dataset, we take the list of all methods, remove those methods that exist in the set of the faulty methods, and add the set of the faulty methods with the metrics computed *at the faulty state.* After doing that, we end up with a list containing each method exactly once and a flag indicating whether a method was faulty or not.

3) Defect datasets are often highly imbalanced [Khoshgoftaar et al. (2010)], with faulty methods being underrepresented. Therefore, we apply *SMOTE*[1], a well-known algorithm for over- and under-sampling, to address imbalance in the dataset used for training [Longadge et al. (2013); Chawla et al. (2002)]. It artificially generates new entries of the minority class using the nearest neighbors of these cases and reduces entries from the majority class [Torgo (2010)]. If we do not apply *SMOTE* to highly imbalanced

[1] Synthetic Minority Over-sampling Technique



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

datasets, many non-expressive rules will be generated when most methods are not faulty. For example, if 95% of the methods are not faulty and 90% of them contain a method invocation, rules with high support will be generated that use this association to identify non-faulty methods. Balancing avoids those nonsense rules.

### 4.3 IDP Classifier

To identify low-fault-risk methods, we compute association rules of the type $\{Metric1, Metric2, Metric3, \ldots\} \rightarrow \{NotFaulty\}$. Examples for the metrics are *SlocLowestThird*, *NoNullChecks*, *IsSetter*. A method that satisfies all metric predicates of a rule is not faulty to the certainty expressed by the confidence of the rule. The support of the rule expresses how many methods with these characteristics exist, and thus, it shows how generalizable the rule is.

After computing the rules on a training set, we remove redundant ones (see Section 2) and order the remaining rules first descending by their confidence and then by their support. To build the low-fault-risk classifier, we combine the top $n$ association rules with the highest confidence values using the logical-or operator. Hence, we consider a method to have a low fault risk if at least one of the top $n$ rules matches. To determine $n$, we compute the maximum number of rules until the faulty methods in the low-fault-risk methods exceed a certain threshold in the training set.

Of course, IDP can also be used with other machine-learning algorithms. We decided to use association rule mining because of the natural comprehensibility of the rules (see Section 2) and because we achieved a better performance compared to models we trained using Random Forest.

## 5 EMPIRICAL STUDY

This section reports on the empirical study that we conducted to evaluate the inverse defect prediction approach.

### 5.1 Research Questions

We investigate the following questions to research how well methods that contain hardly any faults can be identified and to study whether IDP is applicable in cross-project scenarios.

**RQ 1: How many faults do methods classified as "low fault risk" contain?** To evaluate the precision of the classifier, we investigate how many methods that are classified as "low-fault-risk" (due to the triviality of their code) are faulty. If we want to use the low-fault-risk classifier for determining methods that require less focus during quality assurance (QA) activities, such as testing and code reviews, we need to be sure that these methods contain hardly any faults.

**RQ 2: How large is the fraction of the code base consisting of methods classified as "low fault risk"?** We study how common low-fault-risk methods are in code bases to find out how much code is of lower importance for quality-assurance activities. We want to determine which savings potential can arise if these methods are excluded from QA.

**RQ 3: Is a trained classifier for methods with low fault risk generalizable to other projects?** Cross-project defect prediction is used to predict faults in (new) projects, for which no historical fault data exists, by using models trained on other projects. It is considered a challenging task in defect prediction [He et al. (2012); Zimmermann et al. (2009); Turhan et al. (2009)]. As we expect that the characteristics of low-fault-risk methods might be project-independent, IDP could be applicable in a cross-project scenario. Therefore, we investigate how generalizable our IDP classifier is for cross-project use.

### 5.2 Study Objects

For our analysis, we used data from Defects4J, which was created by Just et al. [Just et al. (2014)]. Defects4J is a database and analysis framework that provides real faults for six real-world open-source projects written in Java. For each fault, the original commit before the bug fix (faulty version), the original commit after the bug fix (fixed version), and a minimal patch of the bug fix are provided. The patch is minimal such that it contains only code changes that 1) fix the fault and 2) are necessary to keep the code compilable (e.g., when a bug fix involves method-signature changes). It does not contain changes that do not influence the semantics (e.g., changes in comments, local renamings), and changes that were



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

**Table 2.** Study objects.

| Name | SLOC | #Methods | #Faulty Meth. |
|---|---|---|---|
| JFreeChart *(Chart)* | 81.6k | 6.8k | 39 |
| Google *Closure* Compiler | 166.7k | 13.0k | 148 |
| Apache Commons *Lang* | 16.6k | 2.0k | 73 |
| Apache Commons *Math* | 9.5k | 1.2k | 132 |
| *Mockito* | 28.3k | 2.5k | 64 |
| Joda *Time* | 89.0k | 10.1k | 45 |

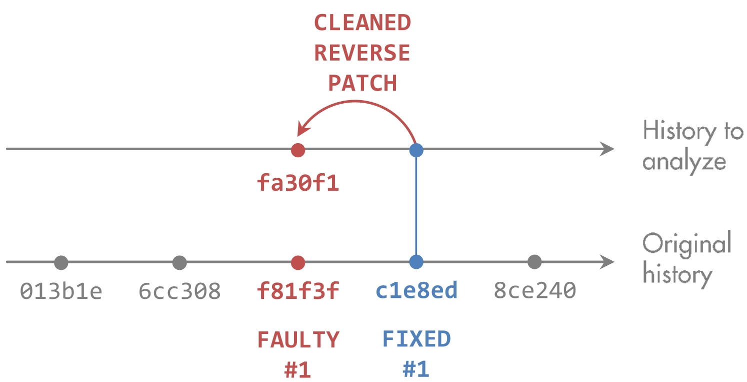


**Figure 2.** Derivation of faulty methods. The original bug-fix commit `c1e8ed` to fix the faulty version `f81f3f` may contain unrelated changes. Defect4J provides a reverse patch, which contains only the actual fix. We applied it to the fixed version `c1e8ed` to get to `fa30f1`. We then identified methods that were touched by the patch and computed their metrics at state `fa30f1`.

included in the bug-fix commit but are not related to the actual fault (e.g., refactorings). Due to the manual analysis, this dataset at the method level is much more precise than other datasets at the same level, such as [Shippey et al. (2016)] and [Giger et al. (2012)], which were generated from version control systems and issue trackers without further manual filtering. The authors of [Just et al. (2014)] confirmed that they considered every bug fix within a given time span.

Table 2 presents the study objects and their characteristics. We computed the metrics *SLOC* and *#Methods* for the code revision at the last bug-fix commit of each project; the numbers do not comprise sample and test code. *#Faulty methods* corresponds to the number of faulty methods derived from the dataset.

### 5.3 Fault Data Extraction

Defects4J provides for each project a set of reverse patches[2], which represent bug fixes. To obtain the list of methods that were at least once faulty, we conducted the following steps for each patch. First, we checked out the source code from the project repository at the original bug-fix commit and stored it as *fixed version*. Second, we applied the reverse patch to the fixed version to get to the code before the bug fix and stored the resulting *faulty version*.

Next, we analyzed the two versions created for every patch. For each file that was changed between the faulty and the fixed version, we parsed the source code to identify the methods. We then mapped the code changes to the methods to determine which methods were touched in the bug fix. After that, we had the list of faulty methods. Figure 2 summarizes these steps.

We inspected all 395 bug-fix patches and found that 10 method changes in the patches do not represent bug fixes. While the patches are minimal, such that they contain only bug-related changes (see Section 5.2), these ten method changes are semantic-preserving, only necessary because of changed signatures of other methods in the patch, and therefore included in Defects4J to keep the code compilable. Figure 3 presents

[2] A reverse patch reverts previous changes.



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

```
   */
  public static String escapeJavaScript(String str) {
+     return escapeJavaStyleString(str, true, true);
-     return escapeJavaStyleString(str, true);
  }

@@ -152,12 +152,12 @@ public class StringEscapeUtils {
   */
  private static String escapeJavaStyleString(String str,
+         boolean escapeSingleQuotes, boolean escapeForwardSlash) {
-         boolean escapeSingleQuotes) {
      if (str == null) {
```

**Figure 3.** Example of method change without behavior modification to preserve API compatibility. The method `escapeJavaScript(String)` invokes `escapeJavaStyleString(String, boolean, boolean)`. A further parameter was added to the invoked method; therefore, it was necessary to adjust the invocation in `escapeJavaScript(String)`. For invocations with the parameter value `true`, the behavior does not change [*Lang*, patch 46, simplified].

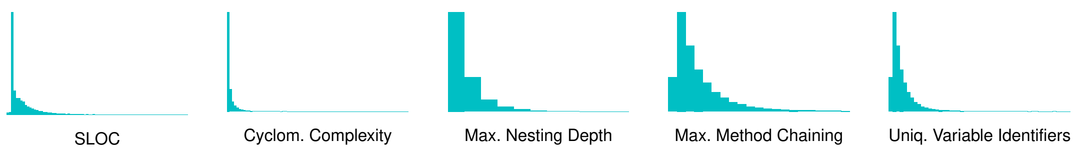


**Figure 4.** Metrics M1 to M5 are not normally distributed.

an example. Although these methods are part of the bug fix, they were not changed semantically and do not represent faulty methods. Therefore, we decided to remove them from the faulty methods in our analysis. The names of these ten methods are provided in the dataset to this paper [Niedermayr et al. (2018)].

### 5.4 Procedure

After extracting the faulty methods from the dataset, we computed the metrics listed in Section 4. We computed them for all faulty methods at their faulty version and for all methods of the application code[3] at the state of the fixed version of the last patch. We used Eclipse JDT AST[4] to create an AST visitor for computing the metrics. For all further processing, we used the statistical computing software R[5].

To discretize the metrics M1 to M5, we first computed their value distribution. Figure 4 shows that their values are not normally distributed (most values are very small). To create three classes for each of these metrics,[6] we sorted the metric values, and computed the values at the end of the first and at the end of the second third. We then put all methods until the last occurrence of the value at the end of the first third into class 1, all methods until the last occurrence of the value at the end of the second third into class 2, and all other methods into class 3. Table 3 presents the value ranges of the resulting classes. The classes are the same for all six projects.

We then aggregated multiple faulty occurrences of the same method (this occurs if a method is changed in more than one bug-fix patch) and created a unified dataset of faulty and non-faulty methods (see Section 4.2).

Next, we split the dataset into a training and a test set. For RQ 1 and RQ 2, we used 10-fold cross-validation [(Witten et al., 2016, Chapter 5)]. Using the *caret* package [from Jed Wing et al. (2017)], we randomly sampled the dataset of each project into ten stratified partitions of equal sizes. Each partition is used once for testing the classifier, which is trained on the remaining nine partitions. To compute the

[3]code without sample and test code

[4]`http://www.eclipse.org/jdt/`

[5]`https://cran.r-project.org/`

[6]We did not use the `ntile` function to create classes, because it always generates classes of the same size, such that instances with the same value may end up in different classes (e.g., if 50% of the methods have the complexity value 1, the first 33.3% will end up in class 1, and the remaining 16.7% with the same value will end up in class 2).

**Table 3.** Generated classes and their value ranges.

| Metric | Class 1 | Class 2 | Class 3 |
|---|---|---|---|
| SLOC | $[0;3]$ | $[4;8]$ | $[9;\infty)$ |
| Cyclomatic Complexity | $[1;1]$ | $[2;2]$ | $[3;\infty)$ |
| Max. Nesting Depth | $[0;0]$ | $[1;1]$ | $[2;\infty)$ |
| Max. Method Chaining | $[0;1]$ | $[2;2]$ | $[3;\infty)$ |
| Uniq. Variable Identifiers | $[0;1]$ | $[2;3]$ | $[4;\infty)$ |

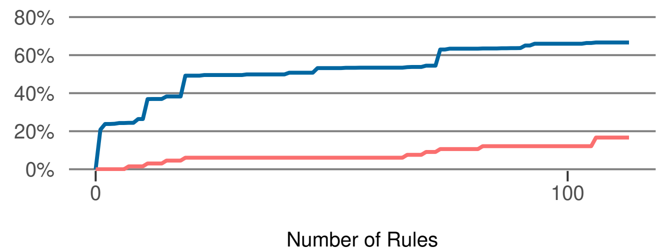


**Figure 5.** Influence of the number of selected rules (*Lang*). The number of rules influences the — proportion of low-fault-risk (LFR) methods and the — share of faulty methods in LFR out of all faulty methods.

association rules for RQ 3—in which we study how generalizable the classifier is—for each project, we used the methods of the other five projects as training set for the classifier.

Before computing association rules, we applied the SMOTE algorithm from the *DMwR* package [Torgo (2010)] with a 100% over-sampling and a 200% under-sampling rate to each training set. After that, each training set was equally balanced (50% faulty methods, 50% non-faulty methods).[7]

We then used the implementation of the *Apriori* algorithm [Agrawal et al. (1994)] in the *arules* package [Hahsler et al. (2017, 2005)] to compute association rules with *NotFaulty* as target item (rule consequent). We set the threshold for the minimum support to 10% and the threshold for the minimum confidence to 90% (support and confidence are explained in Section 2). We experimented with different thresholds and these values produced good results (results for other configurations are in the dataset provided with this paper [Niedermayr et al. (2018)]). The minimum support avoids overly infrequent (i.e., non-generalizable) rules from being created, and the minimum confidence prevents the creation of imprecise rules. Note that no rule (with *NotFaulty* as rule consequent) can reach a higher support than 50% after the SMOTE pre-processing. After computing the rules, we removed redundant ones using the corresponding function from the *apriori* package. We then sorted the remaining rules descending by their confidence.

Using these rules, we created two classifiers to identify low-fault-risk (LFR) methods. They differ in the number of comprised rules. The strict classifier uses the top $n$ rules until the share of faulty methods in all methods (of the training set) exceeds 2.5% in the LFR methods (of the training set). The more lenient classifier uses the top $n$ rules until the share exceeds 5% in the LFR methods. (Example: We applied the top one rule to the training set, then applied the next rule, . . . , until the matched methods in the training set contained 2.5% out of all faults.) Figure 5 presents how an increase in the number of selected rules affects the proportion of LFR methods and the share of faulty methods that they contain. For RQ 1 and RQ 2, the classifiers were computed for each fold of each project. For RQ 3, the classifiers were computed once for each project.

To answer **RQ 1,** we used 10-fold cross-validation to evaluate the classifiers separately for each project. We computed the number and proportion of methods that were classified as "low-fault-risk" but contained a fault ($\approx$ false positives). For the sake of completeness, we also computed precision and recall; although, we believe that the recall is of lesser importance for our purpose. This is because we do not

[7] We computed the results for the empirical study once with and once without addressing the data imbalance in the training set. The prediction performance was better when applying SMOTE, therefore, we decided to use it.



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

**Table 4.** RQ 1, RQ 2: Evaluation of within-project IDP to identify low-fault-risk (LFR) methods.

| Project | Faults in LFR | | LFR methods | | LFR methods | | LFR SLOC | | LFR methods contain ... % | fault-density reduction | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | # | % | Prec. | Rec. | # | % | # | % | of all faults | (methods) | (SLOC) |
| ***Within-project IDP, 10-fold:*** *min. support = 10%, min. confidence = 90%, rules until fault share in training set =* ***2.5%*** | | | | | | | | | | | |
| Chart | 4 | 0.1% | 99.9% | 44.1% | 2,995 | 43.9% | 11,228 | 15.8% | 10.3% | 4.3 | 1.5 |
| Closure | 6 | 0.2% | 99.8% | 29.2% | 3,759 | 28.9% | 15,497 | 10.5% | 4.1% | 7.1 | 2.6 |
| Lang | 3 | 0.5% | 99.5% | 29.6% | 576 | 28.6% | 2,242 | 13.8% | 4.1% | 7.0 | 3.4 |
| Math | 2 | 1.1% | 98.9% | 18.4% | 190 | 16.5% | 570 | 4.8% | 1.5% | 10.9 | 3.1 |
| Mockito | 5 | 0.6% | 99.4% | 35.1% | 875 | 34.4% | 6,128 | 25.1% | 7.8% | 4.4 | 3.2 |
| Time | 8 | 0.1% | 99.9% | 80.4% | 8,063 | 80.2% | 62,063 | 78.1% | 17.8% | 4.5 | 4.4 |
| *Median* | | *0.3%* | *99.7%* | *32.3%* | | *31.7%* | | *14.8%* | *6.0%* | *5.7* | *3.2* |
| ***Within-project IDP, 10-fold:*** *min. support = 10%, min. confidence = 90%, rules until fault share in training set =* ***5%*** | | | | | | | | | | | |
| Chart | 4 | 0.1% | 99.9% | 44.8% | 3,040 | 44.6% | 11,563 | 16.3% | 10.3% | 4.3 | 1.6 |
| Closure | 15 | 0.3% | 99.7% | 41.8% | 5,385 | 41.5% | 25,981 | 17.6% | 10.1% | 4.1 | 1.7 |
| Lang | 6 | 0.7% | 99.3% | 45.0% | 879 | 43.7% | 3,630 | 22.3% | 8.2% | 5.3 | 2.7 |
| Math | 7 | 2.7% | 97.3% | 24.3% | 255 | 22.1% | 878 | 7.3% | 5.3% | 4.2 | 1.4 |
| Mockito | 6 | 0.5% | 99.5% | 47.8% | 1,189 | 46.8% | 8,260 | 33.8% | 9.4% | 5.0 | 3.6 |
| Time | 9 | 0.1% | 99.9% | 82.8% | 8,298 | 82.5% | 63,333 | 79.7% | 20.0% | 4.1 | 4.0 |
| *Median* | | *0.4%* | *99.6%* | *44.9%* | | *44.1%* | | *20.0%* | *9.8%* | *4.3* | *2.2* |

want to predict *all* methods that do not contain any faults in the dataset; we only want to identify those methods that we can say, *with high certainty*, contain hardly any faults.

As the dataset is imbalanced with faulty methods in the minority, the proportion of faults in low-fault-risk methods might not be sufficient to assess the classifiers (SMOTE was applied only to the training set). Therefore, we further computed the *fault-density reduction*, which describes how much less likely the LFR methods contain a fault. For example, if 40% of all methods are classified as "low fault risk" and contain 10% of all faults, the factor is 4. It can also be read as: 40% of all methods contain only one fourth of the expected faults. We mathematically define the fault-density reduction factor based on methods as

$$\frac{\text{proportion of LFR methods out of all methods}}{\text{proportion of faulty LFR methods out of all faulty methods}}$$

and based on SLOC as

$$\frac{\text{proportion of SLOC in LFR methods out of all SLOC}}{\text{proportion of faulty LFR methods out of all faulty methods}}.$$

For both classifiers (strict variant with 2.5%, lenient variant with 5%), we present the metrics for each project and the resulting median.

To answer **RQ 2,** we assessed how common methods classified as "low fault risk" are. For each project, we computed the absolute number of low-fault-risk methods, their proportion out of all methods, and their extent by considering their SLOC. *LFR SLOC* corresponds to the sum of SLOC of all low-fault-risk methods. The proportion of LFR SLOC is computed out of all SLOC of the project.

To answer **RQ 3,** we computed the association rules for each project with the methods of the other five projects as training data. Like in RQ 1 and RQ 2, we determined the number of used top $n$ rules with the same thresholds (2.5% and 5%). To allow a comparison with the within-project classifiers, we computed the same metrics like in RQ 1 and RQ 2.

### 5.5 Results

This section presents the results to the research questions. The data to reproduce the results is available at [Niedermayr et al. (2018)].

**RQ 1: How many faults do methods classified as "low fault risk" contain?** Table 4 presents the results. The methods classified to have low fault risk (LFR) by the stricter classifier, which allows a maximum fault share of 2.5% in the LFR methods in the (balanced) training data, contain between 2 and 8 faulty methods per project. The more lenient classifier, which allows a maximum fault share of 5%,



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

**Table 5.** Top three association rules for *Lang* (within-project, fold 1).

| # | Rule | Support | Confidence |
|---|---|---|---|
| 1 | { *UniqueVariableIdentifiersLessThan2*, *NoMethodInvocations* } ⇒ { *NotFaulty* } | 10.98% | 100.00% |
| 2 | { *SlocLessThan4*, *NoMethodInvocations*, *NoArithmeticOperations* } ⇒ { *NotFaulty* } | 10.98% | 100.00% |
| 3 | { *SlocLessThan4*, *NoMethodInvocations*, *NoCastExpressions* } ⇒ { *NotFaulty* } | 10.60% | 100.00% |

classified between 4 and 15 faulty methods as LFR. The median proportion of faulty methods in LFR methods is 0.3% resp. 0.4%.

The fault-density reduction factor for the stricter classifier ranges between 4.3 and 10.9 (median: 5.7) when considering methods and between 1.5 and 4.4 (median: 3.2) when considering SLOC. In the project *Lang*, 28.6% of all methods with 13.8% of the SLOC are classified as LFR and contain 4.1% of all faults, thus, the factor is 7.0 (SLOC-based: 3.4). The factor never falls below 1 for both classifiers.

> IDP can identify methods with low fault risk. On average, only 0.3% of the methods classified as "low fault risk" by the strict classifier are faulty. The identified LFR methods are, on average, 5.7 times less likely to contain a fault than an arbitrary method in the dataset.

Table 5 exemplarily presents the top three rules for *Lang*. Methods that work with fewer than two variables and do not invoke any methods as well as short methods without arithmetic operations, cast expressions, and method invocations are highly unlikely to contain a fault.

**RQ 2: How large is the fraction of the code base consisting of methods classified as "low fault risk"?** Table 4 presents the results. The stricter classifier classified between 16.5% and 80.2% of the methods as LFR (median: 31.7%, mean: 38.8%), the more lenient classifier matched between 22.1% and 82.5% of the methods (median: 44.1%, mean: 46.9%). The median of the comprised SLOC in LFR methods is 14.8% (mean: 24.7%) respectively 20.0% (mean: 29.5%).

> Using within-project IDP, on average, 32–44% of the methods, comprising about 15–20% of the SLOC, can be assigned a lower importance during testing.

> In the best case, when ignoring 16.5% of the methods (4.8% of the SLOC), it is still possible to catch 98.5% of the faults (*Math*).

**RQ 3: Is a trained classifier for methods with low fault risk generalizable to other projects?** Table 6 presents the results for the cross-project prediction with training data from the respective other projects. Compared to the results of the within-project prediction, except for *Math*, the number of faults in LFR methods decreased or stayed the same in all projects for both classifier variants. While the median proportion of faults in LFR methods slightly decreased, the proportion of LFR methods also decreased in all projects except *Math*. The median proportion of LFR methods is 23.3% (SLOC: 8.1%) for the stricter classifier and 26.3% (SLOC: 12.6%) for the more lenient classifier.

The fault-density reduction improved compared to the within-project prediction for both the method and SLOC level in both classifier variants: For the stricter classifier, the median of the method-based factor is 10.9 (+5.2); the median of the SLOC-based factor is 3.9 (+0.7). Figures 6 illustrates the fault-density reduction for both within-project (RQ 1, RQ 2) and cross-project (RQ 3) prediction.

> Using cross-project IDP, on average, 23–26% of the methods, comprising about 8–13% of the SLOC, can be classified as "low fault risk". The methods classified by the stricter classifier contain, on average, less than one eleventh of the expected faults.

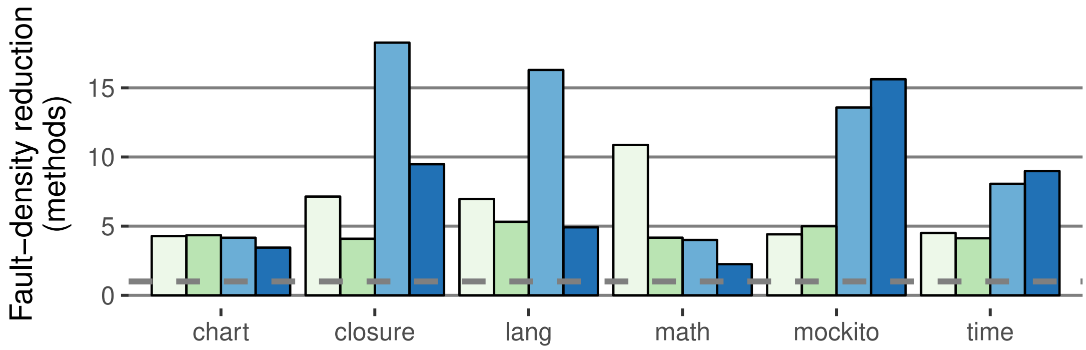


**Figure 6.** Comparison of the IDP within-project (2.5%, 5.0%) with the IDP cross-project (2.5%, 5.0%) classifiers (method-based). The fault-density reduction expresses how much less likely a LFR method contains a fault (definition in 5.4). Higher values are better. (Example: If 40% of the methods are LFR and contain 5% of all faults, the factor is 8.) The dashed line is at one; no value falls below.

**Table 6.** RQ 3: Evaluation of cross-project IDP.

| Project | Faults in LFR | | LFR methods | | LFR methods | | LFR SLOC | | LFR methods contain . . . % | fault-density reduction | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | # | % | Prec. | Rec. | # | % | # | % | of all faults | (methods) | (SLOC) |
| ***Cross-project IDP:** min. support = 10%, min. confidence = 90%, rules until fault share in training set = **2.5%*** | | | | | | | | | | | |
| Chart | 3 | 0.1% | 99.9% | 32.1% | 2,182 | 32.0% | 7,434 | 10.5% | 7.7% | 4.2 | 1.4 |
| Closure | 2 | 0.1% | 99.9% | 25.0% | 3,207 | 24.7% | 11,584 | 7.9% | 1.4% | 18.3 | 5.8 |
| Lang | 1 | 0.2% | 99.8% | 23.1% | 449 | 22.3% | 1,357 | 8.3% | 1.4% | 16.3 | 6.1 |
| Math | 8 | 2.9% | 97.1% | 26.6% | 280 | 24.3% | 1,129 | 9.4% | 6.1% | 4.0 | 1.6 |
| Mockito | 1 | 0.2% | 99.8% | 21.7% | 539 | 21.2% | 1,698 | 6.9% | 1.6% | 13.6 | 4.4 |
| Time | 1 | 0.1% | 99.9% | 18.4% | 1,845 | 18.3% | 5,807 | 7.3% | 2.2% | 8.3 | 3.3 |
| *Median* | | *0.2%* | *99.8%* | *24.0%* | | *23.3%* | | *8.1%* | *1.9%* | *10.9* | *3.9* |
| ***Cross-project IDP:** min. support = 10%, min. confidence = 90%, rules until fault share in training set = **5%*** | | | | | | | | | | | |
| Chart | 4 | 0.2% | 99.8% | 35.5% | 2,411 | 35.4% | 9,363 | 13.2% | 10.3% | 3.4 | 1.3 |
| Closure | 4 | 0.1% | 99.9% | 25.9% | 3,327 | 25.6% | 15,583 | 10.6% | 2.7% | 9.5 | 3.9 |
| Lang | 4 | 0.7% | 99.3% | 27.7% | 542 | 26.9% | 1,959 | 12.0% | 5.5% | 4.9 | 2.2 |
| Math | 18 | 5.1% | 94.9% | 32.9% | 354 | 30.7% | 1,634 | 13.7% | 13.6% | 2.2 | 1.0 |
| Mockito | 1 | 0.2% | 99.8% | 25.0% | 620 | 24.4% | 3,495 | 14.3% | 1.6% | 15.6 | 9.1 |
| Time | 1 | 0.0% | 100.0% | 20.0% | 2,007 | 20.0% | 7,552 | 9.5% | 2.2% | 9.0 | 4.3 |
| *Median* | | *0.2%* | *99.8%* | *26.8%* | | *26.3%* | | *12.6%* | *4.1%* | *6.9* | *3.1* |



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

## 6 DISCUSSION

The results of our empirical study show that only very few low-fault-risk methods actually contain a fault, and thus, they indicate that IDP can successfully identify methods that are not fault-prone. On average, 31.7% of the methods (14.8% of the SLOC) matched by the strict classifier contain only 6.0% of all faults, resulting in a considerable fault-density reduction for the matched methods. In any case, low-fault-risk methods are less fault-prone than other methods, (fault-density reduction is higher than one in all projects); based on methods, LFR methods are at least twice less likely to contain a fault. For the stricter classifier, the extent of the matched methods, which could be deferred in testing, is between 5% and 78% of the SLOC of the respective project. The more lenient classifier matches more methods and SLOC at the cost of a higher fault proportion, but still achieves satisfactory fault-density reduction values. This shows that the balance between fault risk and matched extent can be influenced by the number of considered rules to reflect the priorities of a software project.

Interestingly, the cross-project IDP classifier, which is trained on data from the respective other five projects, exhibits a higher precision than the within-project IDP classifier. Except for the *Math* project, the LFR methods contain fewer faulty methods in the cross-project prediction scenario. This is in line with the method-based fault-density reduction factor of the strict classifier, which is in four of six cases better in the cross-project scenario (SLOC-based: three of six cases). However, the proportion of matched methods decreased compared to the within-project prediction in most projects. Accordingly, the cross-project results suggest that a larger, more diversified training set identifies LFR methods more conservatively, resulting in a higher precision and lower matching extent.

*Math* is the only project in which IDP within-project prediction outperformed IDP cross-project prediction. This project contains many methods with mathematical computations expressed by arithmetic operations, which are often wrapped in loops or conditions; most of the faults are located in these methods. Therefore, the within-project classifiers used few, very precise rules for the identification of LFR methods.

To sum up, our results show that the IDP approach can be used to identify methods that are, due to the “triviality” of their code, less likely to contain any faults. Hence, these methods require less focus during quality-assurance activities. Depending on the criticality of the system and the risk one is willing to take, the development of tests for these methods can be deferred or even omitted in case of insufficient available test resources. The results suggest that IDP is also applicable in cross-project prediction scenarios, indicating that characteristics of low-fault-risk methods differ less between projects than characteristics of faulty methods do. Therefore, IDP can be used in (new) projects with no (precise) historical fault data to prioritize the code to be tested.

### 6.1 Limitations

A limitation of IDP is that even low-fault-risk methods can contain faults. An inspection of faulty methods incorrectly classified to have a low fault risk showed that some faults were fixed by only adding further statements (e.g., to handle special cases). This means that a method can be faulty even if the existing code as such is not faulty (due to missing code). Further imaginable examples for faulty low-fault-risk methods are simple getters that return the wrong variable, or empty methods that are unintentionally empty. Therefore, while these methods are much less fault-prone, it cannot be assumed that they never contain any fault. Consequently, excluding low-fault-risk methods from testing and other QA activities carries a risk that needs to be kept in mind.

### 6.2 Relation to Defect Prediction

As discussed in detail in Section 1, IDP presents another view on defect prediction. The focus of IDP on low-fault-risk methods allows optimizing towards precision, while recall is less important. Therefore, a precision-and-recall comparison of our study results with method-level defect prediction studies from other papers, such as [Giger et al. (2012)] or [Hata et al. (2012)], would lead to a performance comparison of the used metrics or classifiers, which is not what differentiates IDP from traditional defect prediction.

### 6.3 Threats to Validity

Next, we discuss the threats to internal and external validity.

#### *6.3.1 Threats to Internal Validity*

The learning and evaluation was performed on information extracted from Defects4J [Just et al. (2014)]. Therefore, the quality of our data depends on the quality of Defects4J. Common problems for defect



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

datasets created by analyzing changes in commits that reference a bug ticket in an issue tracking system are as follows. First, commits that fix a fault but do not reference a ticket in the commit message cannot be detected [Bachmann et al. (2010)]. Consequently, the set of commits that reference a bug fix may not be a fair representation of all faults [Bird et al. (2009); D'Ambros et al. (2012); Giger et al. (2012)]. Second, bug tickets in the issue tracker may not always represent faults and vice versa. Herzig et al. pointed out that a significant amount of tickets in the issue trackers of open-source projects is misclassified [Herzig et al. (2013)]. Therefore, it is possible that not all bug-fix commits were spotted. Third, faults may not have been detected or fixed yet. In general, it is not possible to prove that a method does not contain any faults. Fourth, a commit may contain changes (such as refactorings) that are not related to the bug fix, but this problem does not affect the Defects4J dataset due to the authors' manual inspection. These threats are present in nearly all defect prediction studies, especially in those operating at the method level. Defect prediction models were found to be resistant to such kind of noise to a certain extent [Kim et al. (2011)].

Defects4J contains only faults that are reproducible and can be precisely mapped to methods; therefore, faulty methods may be under-approximated. In contrast, other datasets created without manual post-processing tend to over-approximate faults. To mitigate this threat, we replicated our IDP evaluation with two study objects used in [Giger et al. (2012)] by Giger et al. The observed results were similar to our study.

#### *6.3.2 Threats to External Validity*

The empirical study was performed with six mature open-source projects written in Java. The projects are libraries and their results may not be applicable to other application types, e.g., large industrial systems with user interfaces. The results may also not be transferable to projects of other languages, for the following reasons: First, Java is a strongly typed language that provides type safety. It is unclear if the IDP approach works for languages without type safety, because it could be that even simple methods in such languages exhibit a considerable amount of faults. Second, in case the approach as such is applicable to other languages, the collected metrics and the low-fault-risk classifier need to be validated and adjusted. Other languages may use language constructs in a different way or use constructs that do not exist in Java. For example, a classifier for the C language should take constructs such as `GOTO`s and the use of pointer arithmetic into consideration. Furthermore, the projects in the dataset (published in 2014) did not contain code with lambda expressions introduced in Java 8.[8] Therefore, in newer projects that make use of lambda expressions, the presence of lambdas should be taken into consideration when classifying methods. Consequently, further studies are necessary to determine whether the results are generalizable.

As done in most defect prediction studies, we treated all faults as equal and did not consider their importance. In reality, not all faults have the same importance, because some cause higher failure follow-up costs than others.

## 7 CONCLUSION

Developer teams often face the problem scarce test resources and need therefore to prioritize their testing efforts (e.g., when writing new automated unit tests). Defect prediction can support developers in this activity. In this paper, we propose an inverse view on defect prediction (IDP) to identify methods that are so "trivial" that they contain hardly any faults. We study how unerringly such low-fault-risk methods can be identified, how common they are, and whether the proposed approach is applicable for cross-project predictions.

We show that IDP using association rule mining on code metrics can successfully identify low-fault-risk methods. The identified methods contain considerably fewer faults than the average code and can provide a savings potential for QA activities. Depending on the parameters, a lower priority for QA can be assigned on average to 31.7% resp. 44.1% of the methods, amounting to 14.8% resp. 20.0% of the SLOC. While cross-project defect prediction is a challenging task [He et al. (2012); Zimmermann et al. (2009)], our results suggest that the IDP approach can be applied in a cross-project prediction scenario at the method level. In other words, an IDP classifier trained on one or more (Java open-source) projects can successfully identify low-fault-risk methods in other Java projects for which no—or no precise—fault data exists.

[8] http://www.oracle.com/technetwork/articles/java/architect-lambdas-part1-2080972.html



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

For future work, we want to replicate this study with closed-source projects, projects of other application types, and projects in other programming languages. It is also of interest to investigate which metrics and classifiers are most effective for the IDP purpose and whether they differ from the ones used in traditional defect prediction. Moreover, we plan to study whether code coverage of low-fault-risk methods differs from code coverage of other methods. If guidelines to meet a certain code coverage level are set by the management, unmotivated testers may add tests for low-fault-risk methods first because it might be easier to write tests for those methods. Consequently, more complex methods with a higher fault risk may remain untested once the target coverage is achieved. Therefore, we want to investigate whether this is a problem in industry and whether it can be addressed with an adjusted code-coverage computation, which takes low-fault-risk methods into account.

## ACKNOWLEDGMENT

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant “SOFIE, 01IS18012A”. The responsibility for this article lies with the authors. We thank Nils Göde and Florian Deißenböck for their valuable feedback.

## REFERENCES

Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM.

Agrawal, R., Srikant, R., et al. (1994). Fast algorithms for mining association rules. In *Proc. 20th International Conference on Very Large Data Bases (VLDB’94)*, volume 1215, pages 487–499.

Bacchelli, A., D’Ambros, M., and Lanza, M. (2010). Are popular classes more defect prone? In *Proc. 13th International Conference on Fundamental Approaches to Software Engineering (FASE’10)*, pages 59–73. Springer.

Bachmann, A., Bird, C., Rahman, F., Devanbu, P., and Bernstein, A. (2010). The missing links: Bugs and bug-fix commits. In *Proc. 18th International Symposium on Foundations of Software Engineering (FSE’10)*, pages 97–106. ACM.

Bayardo, R. J., Agrawal, R., and Gunopulos, D. (1999). Constraint-based rule mining in large, dense databases. In *Proc. 15th International Conference on Data Engineering (ICDE’99)*, pages 188–197. IEEE.

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *Proc. Future of Software Engineering (FOSE’07)*, pages 85–103. IEEE Computer Society.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., and Devanbu, P. (2009). Fair and balanced? bias in bug-fix datasets. In *Proc. 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE’09)*, pages 121–130. ACM.

Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. (2011). Don’t touch my code! examining the effects of ownership on software quality. In *Proc. 8th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE’11)*, pages 4–14. ACM.

Bowes, D., Hall, T., Harman, M., Jia, Y., Sarro, F., and Wu, F. (2016). Mutation-aware fault prediction. In *Proc. 25th International Symposium on Software Testing and Analysis (ISSTA’16)*, pages 330–341. ACM.

Catal, C. (2011). Software fault prediction: A literature review and current trends. *Expert Systems with Applications*, 38(4):4626–4636.

Catal, C. and Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346–7354.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research (JAIR)*, 16:321–357.

Czibula, G., Marian, Z., and Czibula, I. G. (2014). Software defect prediction using relational association rule mining. *Information Sciences*, 264:260–278.

D’Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577.

from Jed Wing, M. K. C., Weston, S., Williams, A., Keefer, C., Engelhardt, A., Cooper, T., Mayer, Z., Kenkel, B., the R Core Team, Benesty, M., Lescarbeau, R., Ziem, A., Scrucca, L., Tang, Y., Candan, C., and Hunt., T. (2017). *caret: Classification and Regression Training*. R package version 6.0-76.

Giger, E., D'Ambros, M., Pinzger, M., and Gall, H. C. (2012). Method-level bug prediction. In *Proc. 6th International Symposium on Empirical Software Engineering and Measurement (ESEM'12)*, pages 171–180. ACM.

Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. (2013). The java language specification, java se 7 edition, february 2012. `http://docs.oracle.com/javase/specs/jls/se7/html/index.html`. [Online; accessed 08-August-2017].

Hahsler, M., Buchta, C., Gruen, B., and Hornik, K. (2017). *arules: Mining Association Rules and Frequent Itemsets*. R package version 1.5-2.

Hahsler, M., Gruen, B., and Hornik, K. (2005). arules – A computational environment for mining association rules and frequent item sets. *Journal of Statistical Software*, 14(15):1–25.

Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering (TSE)*, 38(6):1276–1304.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proc. 31st International Conference on Software Engineering (ICSE'09)*, pages 78–88. IEEE Computer Society.

Hata, H., Mizuno, O., and Kikuno, T. (2012). Bug prediction based on fine-grained module histories. In *Proc. 34th International Conference on Software Engineering (ICSE'12)*, pages 200–210. IEEE.

He, Z., Shu, F., Yang, Y., Li, M., and Wang, Q. (2012). An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199.

Herzig, K., Just, S., and Zeller, A. (2013). It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proc. 35th International Conference on Software Engineering (ICSE'13)*, pages 392–401. IEEE.

Hummel, B. (2014). McCabe's Cyclomatic Complexity and Why We Don't Use It. `https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/`. [Online; accessed 08-August-2017].

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *Proc. 23rd International Symposium on Software Testing and Analysis (ISSTA'14)*, pages 437–440, San Jose, CA, USA. Tool demo.

Karthik, R. and Manikandan, N. (2010). Defect association and complexity prediction by mining association and clustering rules. In *Proc. 2nd International Conference on Computer Engineering and Technology (ICCET'10)*, volume 7, pages V7–569. IEEE.

Khoshgoftaar, T. M., Gao, K., and Seliya, N. (2010). Attribute selection and imbalanced data: Problems in software defect prediction. In *Proc. 22nd International Conference on Tools with Artificial Intelligence (ICTAI'10)*, volume 1, pages 137–144. IEEE.

Kim, S., Whitehead Jr, E. J., and Zhang, Y. (2008). Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering (TSE)*, 34(2):181–196.

Kim, S., Zhang, H., Wu, R., and Gong, L. (2011). Dealing with noise in defect prediction. In *Proc. 33rd International Conference on Software Engineering (ICSE'11)*, pages 481–490. IEEE.

Kim, S., Zimmermann, T., Whitehead Jr, E. J., and Zeller, A. (2007). Predicting faults from cached history. In *Proc. 29th International Conference on Software Engineering (ICSE'07)*, pages 489–498. IEEE Computer Society.

Lee, T., Nam, J., Han, D., Kim, S., and In, H. P. (2011). Micro interaction metrics for defect prediction. In *Proc. 8th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, pages 311–321. ACM.

Longadge, R., Dongre, S., and Malik, L. (2013). Class imbalance problem in data mining: Review. *International Journal of Computer Science and Network (IJCSN)*, 2(1):83–87.

Ma, B., Dejaeger, K., Vanthienen, J., and Baesens, B. (2010). Software defect prediction based on association rule classification. In *Proc. 1st International Conference on E-Business Intelligence (ICEBI'10)*. Atlantis Press.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, (4):308–320.

McCallum, A. and Nigam, K. (1998). A comparison of event models for naive bayes text classification. In *Proc. Workshop on Learning for Text Categorization (AAAI-98-W7)*, volume 752, pages 41–48.

Madison, WI.
Mende, T. and Koschke, R. (2009). Revisiting the evaluation of defect prediction models. In *Proc. 5th International Conference on Predictor Models in Software Engineering (PROMISE'09)*, page 7. ACM.
Meneely, A., Williams, L., Snipes, W., and Osborne, J. (2008). Predicting failures with developer networks and social network analysis. In *Proc. 16th International Symposium on Foundations of Software Engineering (FSE'08)*, pages 13–23. ACM.
Menzies, T. and Di Stefano, J. S. (2004). How good is your blind spot sampling policy? In *Proc. 8th International Symposium on High Assurance Systems Engineering*, pages 129–138. IEEE.
Menzies, T., Di Stefano, J. S., Chapman, M., and McGill, K. (2002). Metrics that matter. In *Proc. 27th Annual NASA Goddard Software Engineering Workshop*, pages 51–57. IEEE, IEEE/NASA.
Menzies, T., DiStefano, J., Orrego, A., and Chapman, R. (2004). Assessing predictors of software defects. In *Proc. Workshop Predictive Software Models (PROMISE'04)*.
Menzies, T., Greenwald, J., and Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering (TSE)*, 33(1):2–13.
Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., and Bener, A. (2010). Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407.
Menzies, T., Stefano, J., Ammar, K., McGill, K., Callis, P., Davis, J., and Chapman, R. (2003). When can we test less? In *Proc. 9th International Symposium on Software Metrics (SMS'03)*, pages 98–110. IEEE.
Morisaki, S., Monden, A., Matsumura, T., Tamada, H., and Matsumoto, K.-i. (2007). Defect data analysis based on extended association rule mining. In *Proc. 4th International Workshop on Mining Software Repositories (MSR'07)*, page 3. IEEE Computer Society.
Nagappan, N. and Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proc. 27th International Conference on Software Engineering (ICSE'15)*, pages 284–292. IEEE.
Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In *Proc. 28th International Conference on Software Engineering (ICSE'06)*, pages 452–461. ACM.
ndepend (2017). Code Metrics Definitions. `http://www.ndepend.com/docs/code-metrics\#ILNestingDepth`. [Online; accessed 08-August-2017].
Niedermayr, R., Röhm, T., and Wagner, S. (2018). Dataset: Too trivial to test?
Ostrand, T. J., Weyuker, E. J., and Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering (TSE)*, 31(4):340–355.
Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., and Oliveto, R. (2016). Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In *Proc. 32nd International Conference on Software Maintenance and Evolution (ICSME'16)*, pages 244–255. IEEE.
Pascarella, L., Palomba, F., and Bacchelli, A. (2018). Re-evaluating method-level bug prediction. In *Proc. 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*, pages 592–601. IEEE.
Rahman, F. and Devanbu, P. (2011). Ownership, experience and eefects: A fine-grained study of authorship. In *Proc. 33rd International Conference on Software Engineering (ICSE'11)*, pages 491–500. ACM.
Scanniello, G., Gravino, C., Marcus, A., and Menzies, T. (2013). Class level fault prediction using software clustering. In *Proc. 28th International Conference on Automated Software Engineering (ASE'13)*, pages 640–645. IEEE Press.
Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36.
Shippey, T., Hall, T., Counsell, S., and Bowes, D. (2016). So you need more method level datasets for your software defect prediction?: Voilà! In *Proc. 10th International Symposium on Empirical Software Engineering and Measurement (ESEM'16)*. ACM.
Shivaji, S., Whitehead, E. J., Akella, R., and Kim, S. (2013). Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering (TSE)*, 39(4):552–569.
Simon, G. J., Kumar, V., and Li, P. W. (2011). A simple statistical model and association rule filtering for classification. In *Proc. 17th International Conference on Knowledge Discovery and Data Mining (SIGKDD'11)*, pages 823–831. ACM.
Song, Q., Shepperd, M., Cartwright, M., and Mair, C. (2006). Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering (TSE)*, 32(2):69–82.



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018

Torgo, L. (2010). *Data Mining with R, learning with case studies*. Chapman and Hall/CRC.

Turhan, B., Menzies, T., Bener, A. B., and Di Stefano, J. (2009). On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578.

Weyuker, E. J. and Ostrand, T. J. (2008). What can fault prediction do for you? In *Proc. 2nd International Conference on Tests and Proofs (TAP'08)*, pages 1–10. Springer.

Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. (2016). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.

Xia, X., Lo, D., Pan, S. J., Nagappan, N., and Wang, X. (2016). Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on Software Engineering (TSE)*, 42(10):977–998.

Xu, Z., Liu, J., Luo, X., and Zhang, T. (2018). Cross-version defect prediction via hybrid active learning with kernel principal component analysis. In *Proc. 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*, pages 209–220. IEEE.

Zafar, H., Rana, Z., Shamail, S., and Awais, M. M. (2012). Finding focused itemsets from software defect data. In *Proc. 15th International Multitopic Conference (INMIC'12)*, pages 418–423. IEEE.

Zhang, F., Zheng, Q., Zou, Y., and Hassan, A. E. (2016). Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proc. 38th International Conference on Software Engineering (ICSE'18)*, pages 309–320. ACM.

Zhang, H., Zhang, X., and Gu, M. (2007). Predicting defective software components from code complexity measures. In *Proc. 13th Pacific Rim International Symposium on Dependable Computing (PRDC'07)*, pages 93–96. IEEE.

Zimmermann, T. and Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. In *Proc. 30th International Conference on Software Engineering (ICSE'08)*, pages 531–540. IEEE.

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., and Murphy, B. (2009). Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proc. 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pages 91–100. ACM.

Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. In *Proc. 3rd International Workshop on Predictor Models in Software Engineering (PROMISE'07)*, page 9. IEEE Computer Society.




PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27304v1 | CC BY 4.0 Open Access | rec: 30 Oct 2018, publ: 30 Oct 2018