On the Role of Developer's Scattered Changes in Bug Prediction

Dario Di Nucci*, Fabio Palomba*, Sandro Siravo†, Gabriele Bavota‡, Rocco Oliveto†, Andrea De Lucia*
*University of Salerno, Fisciano (SA), Italy — †University of Molise, Pesche (IS), Italy
‡Free University of Bozen-Bolzano, Bolzano (BZ), Italy

***Abstract*—The importance of human-related factors in the introduction of bugs has recently been the subject of a number of empirical studies. However, these observations have not been captured yet in bug prediction models which simply exploit product metrics or process metrics based on the number and type of changes or on the number of developers working on a software component. Some previous studies have demonstrated that focused developers are less prone to introduce defects than non focused developers. According to this observation, software components changed by focused developers should also be less error prone than software components changed by less focused developers. In this paper we capture this observation by measuring the structural and semantic scattering of changes performed by the developers working on a software component and use these two measures to build a bug prediction model. Such a model has been evaluated on five open source systems and compared with two competitive prediction models: the first exploits the number of developers working on a code component in a given time period as predictor, while the second is based on the concept of code change entropy. The achieved results show the superiority of our model with respect to the two competitive approaches, and the complementarity of the defined scattering measures with respect to standard predictors commonly used in the literature.**

I. Introduction

Empirical studies have been carried out to assess under which circumstances and during which coding activities developers tend to introduce bugs (see e.g., [1], [2], [3], [4], [5], [6]). Also, bug prediction techniques built on top of *process metrics* (i.e., metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) have been proposed [7], [8], [9], [10], [11], [12], [13]. Several of these techniques have demonstrated their superiority [14] with respect to approaches only exploiting *product metrics* (i.e., metrics capturing intrinsic characteristics of the code components, like their size and complexity) [15], [16], [17], [18], [19].

However, the role of developer-related factors in the bug prediction field is only partially explored. Indeed, our knowledge on this topic is mainly amenable to few empirical studies performed in the last years. The first is the one by Eyolfson *et al.* [2], who showed that more experienced developers tend to introduce less faults in software systems. The second has been performed by Rahman and Devanbu [3], and contradicts in part the study by Eyolfson *et al.* [2] by showing that the experience of a developer has no clear association with the introduction of buggy code. Bird *et al.* [20] found that high levels of ownership are associated with fewer bugs. Posnett *et al.* [6] showed that focused developers (i.e., developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers. This observation has been confirmed by Tufano *et al.* [21], who reported as commits (i.e., code changes) impacting unrelated code components (i.e., files belonging to different subsystems or, more in general, implementing unrelated responsibilities) are more likely to introduce bugs with respect to commits performed on highly related code components. These works have pioneered the investigation of human-related factors in the context of bug introduction.

Although such studies showed the importance of human-related factors in bug prediction, these observations have not been captured yet in bug prediction models based on process metrics extracted from version history. Indeed, previous work have proposed the use of predictors based (i) on the number of developers working on a code component [10] [11]; (ii) on the analysis of change-proneness [14] [12] [13]; and (iii) on the entropy of changes [9]. None of them consider how focused are the developers performing changes and how scattered are these changes. With this work we aim at making a further step ahead, by studying the role played by *scattering changes* in bug prediction. We firstly define two measures, namely the developer's *structural* and *semantic scattering*. The first aims at assessing how "structurally far" in the software project are the code components modified in a given time period by a developer. The "structural distance" between two code components is measured as the number of subsystems one needs to cross in order to reach one component from the other. The second measure (i.e., the *semantic scattering*) is instead meant to capture how much spread in terms of implemented responsibilities are the code components modified in a given time period by a developer. We expect that high levels of *structural* and *semantic scattering* make the developer more error-prone. In order to verify our conjecture we build two predictors exploiting the proposed measures, and then we use them in a prediction model, comparing its performances with two baseline techniques based on the number of developers working on a code component and the entropy of changes, respectively [10], [9].

The context of our empirical investigation is represented by five large Java open-source systems. The results achieved show the superiority of our model, achieving a prediction accuracy ranging between 68% and 94%, as compared to the 43%-74% achieved by the change entropy model [9] and the 19%-49% obtained by exploiting the number of developers working on a code component as predictor.

Most importantly, the two scattering measures show a high degree of complementarity with the measures exploited by the baseline prediction models, paving the way to more sophisticated and performing bug prediction models to be developed in the future.

II. Related Work

In the last decade a lot of effort has been devoted to the definition of approaches aimed at predicting bug-prone code components. Such approaches mainly differ for the underlying algorithmic solution they exploit and for the predictors they use, with the main distinction between *product metrics* (e.g., lines of code, code complexity, etc) and *process metrics* (e.g., past changes and bug fixes performed on a code component).

Basili *et al.* [15] proposed the use of the Chidamber and Kemerer (CK) metrics [22] to identify buggy classes, showing that five of the experimented metrics are actually useful in characterizing the bug-proneness of a class. The same set of metrics has been successfully exploited in the context of bug prediction by El Emam *et al.* [23] and by Subramanyam *et al.* [24]. Both works concur on reporting the ability of CK metrics in predicting buggy code components. A deeper investigation on the relationship between CK metrics and code bug-proneness has been performed later on by Gyimothy *et al.* [16] by exploiting issues data present in the Bugzilla database. Their results report that the Coupling Between Object metric is the best in predicting the bug-proneness of classes, while other CK metrics—such as the Dept of Inheritance Tree—are untrustworthy. Ohisson *et al.* [17] focused the attention on the use of design metrics to identify bug-prone modules. The investigated metrics include the *number of nodes*, and the *fan-in* and *fan-out* of modules. Their model has been experimented on a system developed at Ericsson, reporting the practical applicability of design metrics for the identification of buggy modules. Nagappan and Ball [18] exploited two static analysis tools to predict the pre-release bug density for Windows Server. Their results show that it is possible to perform a coarse grained classification between high and low quality components with an accuracy of 83%. Nagappan *et al.* [25] also investigated the use of metrics in the prediction of buggy components across five Microsoft projects. Their main finding highlights that while it is possible to successfully exploit complexity metrics in bug prediction, there is no single metric that could act as a universally best bug predictor (i.e., the best predictor is project-dependent). Complexity metrics in the context of bug prediction is also the focus of the work by Zimmerman *et al.* [19]. Their study reports a positive correlation between code complexity and bugs. Still in terms of product metrics, Nikora *et al.* [26] showed that measurements of a system's structural evolution (e.g., number of executable statements, number of nodes in the control flow graph, etc) can serve as predictors of the number of bugs inserted into a system during its development.

Differently from the previous discussed techniques, other approaches try to predict bugs by exploiting process metrics. Khoshgoftaar *et al.* [7] studied two subsequent releases of a large legacy system to assess the role played by debug churns (i.e., the number of lines of code changed to fix bugs) in the identification of bug-prone modules. In particular, modules showing a debug churns exceeding a defined threshold are marked as bug-prone. The reported results show a misclassification rate of 21%. Graves *et al.* [8] experimented both product and process metrics for bug prediction. Their findings contradict in part what observed by other authors, showing that product metrics are poor predictors of bugs. Instead, they found process metrics and in particular the module's age and its change-proneness to be the best predictors. D'Ambros *et al.* [27] performed an extensive comparison of bug prediction approaches relying on process and product metrics, showing that there is not a technique that works better in all contexts. Hassan and Holt [28] introduced the concept of entropy of changes as a measure of the complexity of the development process. They also presented some years later "The Top Ten List" [29], a methodology to highlight to managers the top ten subsystems more likely to present bugs. The set of heuristics behind their approach includes a number of process metrics, such as considering the *most recently modified*, the *most recently fixed* and the *most frequently fixed* subsystems. Moser *et al.* [14] performed a comparative study between the predictive power of product and process metrics. Their study, performed on Eclipse, highlights the superiority of process metrics in predicting buggy code components. Moser *et al.* [12] also performed a deeper study on the bug prediction accuracy of process metrics, reporting that the *past number of bug-fixes performed on a file* (i.e., bug-proneness), the *maximum changeset size occurred in a given period*, and the *number of changes involving a file in a given period* (i.e., change-proneness) are the process metrics ensuring the best performances in bug prediction. Also Bell *et al.* [13] pointed to the code components' change-proneness as the best bug predictor. The complexity of the development process, and in particular the entropy of changes, has been exploited by Hassan [9] to build two bug prediction models, namely Basic Code Change Model (BCCM) and Extended Code Change Model (ECCM). These two models mainly differ for the choice of the temporal interval where the bug proneness of components is studied. The results of a reported case study indicate that the proposed techniques have a prediction accuracy of a prediction model based purely on code components changes. All of the predictors above do not consider how many developers apply changes to a component, neither how many components they changed at the same time. Ostrand *et al.* [11], [10] propose the use of the *number of developers who modified a code component in a give time period* as a bug-proneness predictor. Their results show that the detection accuracy of prediction model exploiting products and process metrics is poorly (positively) impacted by also considering the developers' information. Our work does not use a simple count information of developers who worked on a file, but also takes into consideration the change activities they carry out. Finally, among the studies investigating the influence of developers'

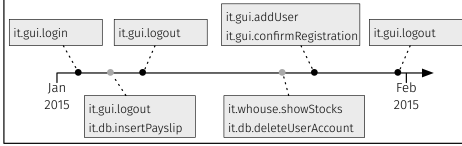

Fig. 1. Example of two developers having different levels of "scattering"

related factors on the introduction of bugs (see Section I), worthy deepening in this section is the one by Posnett *et al.* [6]. The authors investigate factors related to the one we aim at capturing in this paper, i.e., the developer's scattering. In particular, the "focus" metric presented by Posnett *et al.* [6] is based on the idea that a developer performing most of her activities on a single module (a module could be a method, a class, etc) has a higher focus on the activities she is performing and is less likely to introduce bugs. As it will be clearer later, our scattering measures not only take into account the frequency of changes made by developers over the different system's modules, but also considers the "distance" between the modified modules. This means that, for example, the contribution of a developer working on a high number of files all closely related to a specific responsibility might not be as much "scattered" as the contribution of a developer working on few *unrelated* files.

III. Computing Developer's Scattering Changes

We conjecture that the developer's effort in performing maintenance and evolution tasks is proportional to the number of involved components and their spread across different subsystems. In other words, we believe that a developer working on different components scatters her attention due to continuous changes of context. This might lead to an increase of the developer's *"scattering"* with a consequent higher chance of introducing bugs.

To get a better idea of our conjecture, consider the situation depicted in Figure 1, where two developers, d_1 (black point) and d_2 (grey point) are working on the same system, during the same time period, but on different code components. The tasks performed by d_1 are very focused on a specific part of the system (she mainly works on the systems' GUI) and on a very targeted topic (she is mainly in charge of working on GUIs related to the users' registration and login features). On the contrary, d_2 performs tasks scattered across different parts of the system (from GUIs to database management) and on different topics (users' accounts, payslips, warehouse stocks).

Our conjecture is that during the time period shown in Figure 1, the contribution of d_2 might have been more "scattered" than the contribution of d_1, and has a higher likelihood of introducing bugs in the system during her change activities. To verify our conjecture we define two measures, named the *structural* and the *semantic scattering* measures, aimed at assessing the scattering of a developer d in a given time period p. Note that both measures are meant to work in object oriented systems at the class level granularity. In other words, we measure how scattered are the changes performed by developer d during the time period p across the the different classes of the system. However, our measures can be easily adapted to work at other granularity levels.

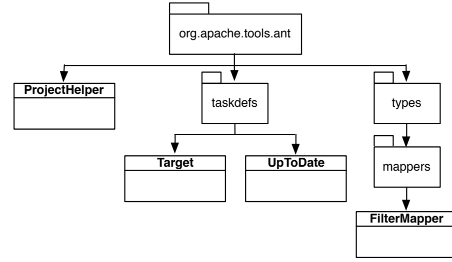

Fig. 2. Example of structural scattering

A. Structural scattering

Let $CH_{d,p}$ be the set of classes changed by a developer d during a time period p. We define the *structural scattering* measure as:

$$\text{StrScat}_{d,p} = \frac{|CH_{d,p}|}{\binom{|CH_{d,p}|}{2}} \times \sum_{\forall c_i, c_j \in CH_{d,p}} [dist(c_i, c_j)] \quad (1)$$

where $dist$ is the number of packages to traverse in order to go from class c_i to class c_j; $dist$ is computed by applying the shortest path algorithm on the graph representing the system's package structure. For example, the $dist$ between two classes `it.user.gui.c`$_1$ and `it.user.business.db.c`$_1$ is three, since in order to reach `c`$_1$ from `c`$_2$ we need to traverse `it.user.business.db`, `it.user.business`, and `it.user.gui`. The multiplication factor at the beginning of the formula has a two-fold objective: (i) normalizing the distances between the code components by the number of pairs of code components modified by the developer during the time period p—see the denominator $\binom{|CH_{d,p}|}{2}$—and (ii) assigning a higher scattering to developers working on a higher number of code components in the given time period—see the numerator $|CH_{d,p}|$.

To better understand how the *structural scattering* measure is computed and how it is possible to use it in order to estimate the developer's scattering in a time period, Figure 2 provides a running example based on a real scenario we found in the open source project *Apache Ant*[1], a tool to automate the building of software projects.

The tree shown in Figure 2 depicts the activity of a single developer in the time period between 2012-03-01 and 2012-04-30. In particular, the leafs of the tree represent the classes modified by the developer in the considered time period, while the internal nodes (as well as the root node) illustrate the package structure of *Apache Ant*. In this example, the developer worked on the classes `Target` and `UpToDate`, both contained in the package `org.apache.tools.ant.taskdefs` grouping together classes managing the definition of new commands that the *Ant*'s user can create for customizing her own building process. In addition, the developer also modified `FilterMapper`, a class containing utility methods (e.g., map a java String into

[1] http://ant.apache.org/

TABLE I

EXAMPLE OF STRUCTURAL SCATTERING COMPUTATION

Changed components		Distance
org.apache.tools.ant.ProjectHelper	org.apache.tools.ant.taskdefs.Target	1
org.apache.tools.ant.ProjectHelper	org.apache.tools.ant.taskdefs.UpToDate	1
org.apache.tools.ant.ProjectHelper	org.apache.tools.ant.types.mappers.FilterMapper	2
org.apache.tools.ant.taskdefs.Target	org.apache.tools.ant.taskdefs.UpToDate	0
org.apache.tools.ant.taskdefs.Target	org.apache.tools.ant.types.mappers.FilterMapper	3
org.apache.tools.ant.taskdefs.UpToDate	org.apache.tools.ant.types.mappers.FilterMapper	3
Structural Developer scattering		**6.67**

an array), and the class `ProjectHelper` responsible for parsing the build file and creating java instances representing the build workflow. To compute the *structural scattering* we calculate the distances between every pair of classes modified by the developer. If two classes are in the same package, as in the case of the classes `Target` and `UpToDate`, then the distance between them will be 0. Instead, if they are in different packages, like in the case of `ProjectHelper` and `Target`, their distance is the minimum number of packages one needs to traverse to reach one class from the other. For example, the distance is one between `ProjectHelper` and `Target` (we need to traverse the package `taskdefs`), and three between `UpToDate` and `FilterMapper` (we need to traverse the packages `taskdefs`, `types` and `mappers`).

After computing the distance between every pair of classes, it is possible to calculate the *structural scattering* measure. Table I shows a summary of the computation of the distances between every pair of classes involved in our example and the value for the final measure. Note that, if the developer had modified only the `Target` and `UpToDate` classes in the considered time period, then her *structural scattering* would have been zero (the lowest possible), since her changes were focused on just one package. By adding the change performed to `ProjectHelper`, the *structural scattering* raises to 3.00, and reaches the reported value on 6.67 when also considering the change to the `FilterMapper` class. Thus, the *structural scattering* is a direct scattering measure (i.e., the higher the measure, the higher the *estimated* developer's scattering).

B. Semantic scattering

Considering the package structure might not be an effective way of assessing the classes similarity (i.e., how much the modified classes implement the same responsibilities). Because of the software "aging" or wrong design decisions, classes grouped in the same package may have completely different responsibilities [30]. In such cases, the *structural scattering* measure might provide a wrong assessment of the level of developer's scattering, by considering classes implementing different responsibilities as similar only because contained inside the same package. For this reason, we propose the *semantic scattering* measure, based on the textual similarity of the changed software components. Textual similarity between documents is computed using the *Vector Space Model* (VSM) [31] technique. In our application of VSM we (i) used *tf-idf* weighting scheme [31], (ii) normalized text by performing the splitting of identifiers (we also have maintained the original identifiers), (iii) applied a stop word removal, and (iv) made stemming (using the well known Porter stemmer).

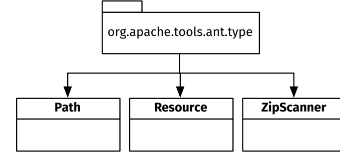

Fig. 3. Example of semantic scattering measure

TABLE II

EXAMPLE OF SEMANTIC SCATTERING COMPUTATION

Changed components		Text. sim.
org.apache.tools.ant.type.Path	org.apache.tools.ant.type.Resource	0.22
org.apache.tools.ant.type.Path	org.apache.tools.ant.type.ZipScanner	0.05
org.apache.tools.ant.type.Resource	org.apache.tools.ant.type.ZipScanner	0.10
Semantic Developer scattering		**36.39**

The semantic scattering measure is computed as:

$$\text{SemScat}_{d,p} = \frac{|CH_{d,p}|}{\binom{|CH_{d,p}|}{2}} \times \frac{1}{\sum_{\forall c_i, c_j \in CH_{d,p}} [sim(c_i, c_j)]} \quad (2)$$

where the sim function returns the textual similarity between the classes c_i and c_j as a value between zero (no textual similarity) and one (the textual content of the two classes is identical). Note that, as for the *structural scattering*, we also adopt a normalization factor taking into account the number of code components modified by developer d during the time period p.

Figure 3 shows an example of computation for the *semantic scattering* measure. Also in this case the figure depicts a real scenario we identified in *Apache Ant* of a single developer in the time period between 2004-04-01 and 2004-06-30. In the example shown, the developer worked on the classes `Path`, `Resource` and `ZipScanner`, all contained in the package `org.apache.tools.ant.types`. `Path` and `Resource` are two data types and have some code in common, while `ZipScanner` is an archive scanner. To compute the *semantic scattering* we calculate the textual similarities between every pair of classes modified by the developer, as reported in Table II. While the *structural scattering* is zero for the example depicted in Figure 3 (all classes are from the same package), the *semantic scattering* is quite high (36.39) due to the low textual similarity between the pairs of classes contained in the package (see Table II).

C. Applications of scattering Measures

The scattering measures defined above could be adopted in different areas related to the monitoring of maintenance and evolution activities. As an example, a project manager could use the scattering measures to estimate the workload of a developer, as well as to the re-allocate resources. In the context of this paper, we propose the use of the defined measures for bug prediction. The basic conjecture is that *developers having a high scattering are more likely to introduce bugs during code change activities*. To exploit the defined scattering measures in the context of bug prediction, we built a new prediction model called *Developer Changes Based Model* (DCBM) that analyzes the components modified by developers in a given time period. The model exploits a machine learning algorithm built on top of two predictors. The first, called *structural scattering predictor*, is defined starting from the structural scattering measure, while the second one, called *semantic scattering*

TABLE III
CHARACTERISTICS OF THE SOFTWARE SYSTEMS USED IN THE STUDY

Project	Period	#Commits	#Dev.	#Classes	KLOC	% buggy classes
Ant	Jan 2000-Jul 2014	13,054	55	1,215	266	72
JMeter	Sep 1998-Apr 2014	10,440	34	1,054	192	37
Log4j	Nov 2000-Feb 2014	3,274	21	309	59	58
Poi	Jan 2002-Aug 2014	5,742	35	2,854	542	62
Xerces-J	Nov 1999-Feb 2014	5,471	34	833	260	6
Overall	**-**	**37,981**	**179**	**6,265**	**1,319**	**52**

predictor, is based on the semantic scattering measure. The predictors are defined as follow:

$$\text{StrScatPred}_{c,p} = \sum_{d \in developers_c} \text{StrScat}_{d,p} \quad (3)$$

$$\text{SemScatPred}_{c,p} = \sum_{d \in developers_c} \text{SemScat}_{d,p} \quad (4)$$

where the $developers_c$ is the set of developers that worked on the component c during the time period p.

IV. Empirical Study Definition and Design

The *goal* of the study is to evaluate the ability of the developer's scattering measures in the prediction of bug-prone components, with the *purpose* of improving the allocation of resources in the verification & validation activities focusing on components having an higher bug-proneness. The *quality focus* is on the detection accuracy and completeness as compared to state-of-the-art approaches, while the *perspective* is of researchers, who want to evaluate the effectiveness of using information about developer scattered changes in identifying bug-prone components.

The *context* of the study consists of five software projects with different size and scope, namely `Apache Ant`[2], `Apache JMeter`[3], `Apache Log4j`[4], `Apache Poi`[5], and `Apache Xerces Java`[6]. Table III reports the characteristics of the analyzed systems, and in particular (i) the software history that we investigated, (ii) the mined number of commits, (iii) the size of the active developers base (those who performed at least one commit in the analyzed time period), (iv) the system's size in terms of KLOC and number of classes, and (v) the percentage of buggy files identified (as explained later) during the entire change history. All data used in our study are publicly available [32].

A. Research Questions and Oracle Definition

In the context of the study, we formulated the following research questions:

- **RQ$_1$:** *Which is the accuracy of a predictor based on developer's scattering measures in detecting bug-prone components?* This research question aims at quantifying the accuracy of a prediction model based on developer's scattering measures (DCBM).
- **RQ$_2$:** *How does a predictor based on developer's scattering measures compare with state-of-the-art techniques?* This research question compares the accuracy of DCBM in detecting bug-prone components with two baseline prediction models. The first is the prediction model based on the work by Ostrand *et al.* [11], [10] and exploiting the number of developers that work on a code component in a specific time period as predictor variable (from now on, we refer to this model as DM). The second is the Basic Code Change Model (BCCM) proposed by Hassan and using code change entropy information [9]. The results of this comparison will provide insights on the usefulness of developer's scattering measures for detecting bug-prone components.

[2]http://ant.apache.org/
[3]http://jmeter.apache.org/
[4]http://logging.apache.org/log4j/
[5]http://poi.apache.org/
[6]http://xerces.apache.org/xerces-j/

Our choices of the baselines with which to compare is mainly focused on techniques exploiting similar information. In particular, the prediction model proposed by Ostrand *et al.* [11], [10] is based on the developers working on a software component, while the prediction model proposed by Hassan [9] is based on the changes to a software component. Our approach combine these two types of information by considering the way the changes made by the different developers on a software components are scattered within the system. In addition our choice of BCCM is also justified by its superiority with respect to other techniques exploiting change-proneness information [14], [12], [13]. While such a superiority has been already demonstrated by Hassan [9], we also compared these techniques before choosing BCCM as one of the baselines for evaluating our approach. In particular, it is worth noting that the BCCM works better with respect to a model that simply counts the number of changes because it filters the changes that differ from the code change process (i.e., fault repairing and general maintenance modifications) considering only the *Feature Introduction modifications* (FI), namely the changes related to the adding or enhancing features. However, we observed a high overlap between the BCCM and the model that use the number of changes as predictor (almost 84%) on the dataset used for the comparison, probably due to the fact that the nature of the information exploited by the two models is the same. The interested reader can find the comparison between these two models in our online appendix [32].

To answer our research questions, we need an oracle reporting the presence of bugs in the source code. Although the PROMISE repository collects a large dataset of bugs in open source systems [33], it provides oracles at release-level. Since the proposed measures work at time period-level, we had to build our own oracle. Firstly, we identified bug fixing commits happened during the change history of each object system by mining regular expressions containing issue IDs in the change log of the versioning system, e.g., *"fixed issue #ID"* or *"issue ID"*. After that, for each identified issue ID, we downloaded the corresponding issue reports from their issue tracking system and extracted the following information from them: *product name*; *issue's type*, i.e., whether an issue is a bug, enhancement request, etc; *issue's status*, i.e., whether an issue was closed or not; *issue's resolution*, i.e., whether an issue was resolved by fixing it, or whether it was a duplicate

bug report, or a "works for me" case; *issue's opening date*; *issue's closing date*, if available.

Then, we checked each issue's report to be correctly downloaded (e.g., the issue's ID identified from the versioning system commit note could be a false positive). After that, we used the issue type field to classify the issue and distinguish bug fixes from other issue types (e.g., enhancements). Finally, we only considered bugs having *Closed* status and *Fixed* resolution. Basically, we restricted our attention to (i) issues that were related to bugs as we used them as a measure of fault-proneness, and (ii) issues that were neither duplicate reports nor false alarms.

Once collected the set of bugs fixed in the change history of each system, we used the SZZ algorithm [34] to identify when each fixed bug was introduced. The SZZ algorithm relies on the annotation/blame feature of versioning systems. In essence, given a bug-fix identified by the bug ID, k, the approach works as follows:

1) For each file f_i, $i = 1 \dots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k), and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).
2) starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of *Git* is used to identify the file revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island grammar parser [35]. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions *rel-bug*$_{i,j,k}$, $j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for the inducing of a fix.

By adopting the process described above we are able to approximate the periods of time where each class of the object systems was affected by one or more bugs (i.e., was a buggy class). In particular, given a bug-fix BF_k performed on a class c_i, we consider c_i buggy from the date in which the bug fixed in BF_k was introduced (as indicated by the SZZ algorithm) to the date in which BF_k (i.e., the patch) was committed in the repository.

B. Data Analysis

To evaluate the performances of the experimented bug prediction techniques (i.e., DBCM, BCCM, and DM), firstly we need to define the machine learning classifier to use. For each prediction technique, we experimented several classifiers, namely ADTree [36], Decision Table Majority [37], Logistic Regression [38], Multilayer Perceptron [39] and Naive Bayes [40]. We empirically compared the results achieved by the three different models on the software systems used in our study (more details on the adopted procedure later in this section). For all the prediction models the best results were obtained using the Majority Decision Table (the comparison among the classifiers can be found in our online appendix [32]). Thus, we exploit it in the implementation of the three models. This classifier can be viewed as an extension of one-valued decision trees [37]. It is a rectangular table where the columns are labeled with predictors and rows are sets of decision rules. Each decision rule of a decision table is composed of (i) a pool of conditions, linked through and/or logical operators which are used to reflect the structure of the if-then rules; and (ii) an outcome which mirrors the classification of a software entity respecting the corresponding rule as bug-prone or non bug-prone. Majority Decision Table uses an attribute reduction algorithm to find a good subset of predictors with the goal of eliminating equivalent rules and reducing the likelihood of over-fitting the data.

To assess the performance of the three models, we split the change-history of the object systems into three-month time periods and we adopt a three-month sliding window to *train* and *test* the bug prediction models. In particular, starting from the first time period TP_1 (i.e., the one starting at the first commit), we train each model on it, and test its ability in predicting buggy classes on TP_2 (i.e., the subsequent three-month time period). Then, we move three months forward the sliding window, training the classifiers on TP_2 and testing their accuracy on TP_3. This process is repeated until the end of the analyzed change history (see Table III) is reached. Note that our choice of considering three-month periods is not random, but based on: (i) choices made in previous work, like the one by Hassan *et al.* [9]; and (ii) the results of an empirical assessment we performed on such a parameter showing that the best results for all experimented techniques are achieved by using three-months length periods. In particular, we experimented with time windows of one, two, three, and six months. The complete results are available in our replication package [32].

Once defined the oracle and obtained the predicted buggy classes by DCBM for every three-month period, we answer **RQ**$_1$ by using three widely-adopted metrics, namely accuracy, precision and recall [31]:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \quad (5)$$

$$precision = \frac{TP}{TP + FN} \quad (6)$$

$$recall = \frac{TP}{TP + TN} \quad (7)$$

where TP is the number of classes containing bugs that are correctly classified as bug-prone; TN denotes the number of bug-free classes classified as non bug-prone classes; FP and FN measure the number of classes for which a prediction model fails to identify bug-prone classes by declaring bug-free classes as bug-prone (FP) or identifying actually buggy classes as non buggy ones (FN). As an aggregate indicator of precision and recall, we also report the F-measure, defined as the harmonic mean of precision and recall:

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall} \quad (8)$$

To answer $\mathbf{RQ}_2$, we performed the bug prediction using BCCM and DM on the same systems and the same periods on which we previously run DCBM in the context of $\mathbf{RQ}_1$.

After that, we analyzed the equivalence of the different measures used by the three experimented bug prediction models using Principal Component Analysis (PCA). PCA is a statistical technique able to identify various orthogonal dimensions (principal components) captured by the data (bug-proneness of classes in our case) which measure contributes to the identified dimensions. Through the analysis of the principal components and the contributions (scores) of each predictor to such components, it is possible to understand whether different predictors contribute to the same principal components. Two models are complementary if the predictors they exploit contribute to capture different principal components. Hence, the analysis of the principal components provides insights on the complementarity between models.

Such an analysis was necessary to assess whether the exploited predictors (e.g., the number of developers for DM as compared to structural and semantic scattering for DCBM) assign the same bug-proneness to the same set of classes. However, PCA does not tell the whole story. Indeed, using PCA it is not possible to identify to what extent a prediction model complements another and *vice versa*. This is the reason why we complemented the PCA by analyzing the overlap of the three prediction models. Specifically, given two prediction models m_i and m_j, we computed:

$$corr_{m_i \cap m_j} = \frac{|corr_{m_i} \cap corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \quad (9)$$

$$corr_{m_i \setminus m_j} = \frac{|corr_{m_i} \setminus corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \quad (10)$$

where $corr_{m_i}$ represents the set of bug-prone classes correctly classified by the prediction model m_i, $corr_{m_i \cap m_j}$ measures the overlap between the set of true positive correctly identified by both models m_i and m_j, $corr_{m_i \setminus m_j}$ measures bug-prone classes correctly classified by m_i only and missed by m_j. Clearly, the overlap metrics are computed by considering each combination of the three experimented detection techniques (e.g., we compute $corr_{BCCM \cap DM}$, $corr_{BCCM \cap DCBM}$, and $corr_{DM \cap DCBM}$). In addition, given the three experimented prediction models m_i, m_j and m_k, we computed:

$$corr_{m_i \setminus (m_j \cup m_k)} = \frac{|corr_{m_i} \setminus (corr_{m_j} \cup corr_{m_k})|}{|corr_{m_i} \cup corr_{m_j} \cup corr_{m_k}|}\% \quad (11)$$

that represents the bug-prone classes correctly identified only by the prediction model m_i.

V. Empirical Study Results

In this section we discuss the results achieved aiming at answering the research questions formulated in Section IV. To avoid redundancies, we report the results for both research questions together.

Table IV reports the results—in terms of accuracy, precision, recall, and F-measure—achieved by the three experimented bug prediction models, i.e., our model, exploiting the developer's scattering metrics (DCBM), the BCCM proposed by Hassan [9], and a prediction model that uses as predictor the number of developers that work on a code component (DM) [11], [10]. The achieved results indicate that the proposed prediction model (i.e., DCBM) ensures better prediction accuracy as compared to the competitive techniques. Indeed, the accuracy of DCBM ranges between 68% and 94%, outperforming the competitive models. Also, in terms of precision and recall (and, consequently, the value of F-measure) DCBM achieves better results. In particular, across all the five object systems, DCBM achieves a higher F-measure with respect to both DM (mean=+46.2%, median=+45%) and BCCM (mean=+11.4%, median=+13.5%). The higher values achieved for precision and recall means that DCBM provides less false positive (i.e., non-buggy classes indicated as buggy ones) while also being able to discover more classes actually affected by a bug as compared to the competitive models.

Interesting is the case of `Xerces-J` where DCBM is able to identify buggy classes with 94% of accuracy (see Table IV), as compared to the 74% achieved by BCCM and the 49% of DM. We looked inside this project to understand the reasons behind such a strong result. We found that the `Xerces-J`'s buggy classes are often modified by few developers that, on average, perform a small number of changes on them. As an example, the class `XSSimpleTypeDecl` of the package `org.apache.xerces.impl.dv.xs` has been modified only two times between May 2008 and July 2008 (i.e., one of the three-month periods considered in our study), by two developers. However, the sum of their structural and semantic scattering in that period was very high (*161* and *1,932*, respectively). Thus, while a model based on the change entropy (BCCM) or on the number of developers modifying a class (DM) experiences difficulties in identifying this class as buggy due to the low number of changes it underwent and to the low number of involved developers, respectively, our model does not suffer of such a limitation thanks to the exploited developers' scattering information.

TABLE V
Results achieved applying the Principal Component Analysis

	PC1	PC2	PC3	PC4
Proportion of Variance	0.76	0.15	0.08	0.01
Cumulative Variance	0.76	0.91	0.99	1.00
Structural scattering predictor	**1.00**	-	-	0.11
Semantic scattering predictor	-	**0.53**	0.45	0.27
Change entropy	-	0.47	**0.54**	**0.42**
Number of Developers	-	-	0.01	0.29

Table V reports the results of the Principal Component Analysis (PCA), aimed at investigating the complementarity between the predictors exploited by the different models. The different columns (PC1 to PC4) represent the components identified by the PCA as those describing the phenomenon of interest (in our case, bug-proneness). The first row (i.e., the proportion of variance) indicates on a scale between zero and one how much each component contributes to the phenomenon description (the higher the proportion of variance, the higher the component's contribution). The identified components are ordered on the basis of their "importance" in describing the

TABLE IV
ACCURACY, PRECISION, RECALL, AND F-MEASURE OF THE THREE BUG PREDICTION MODELS

System	DCBM				DM				BCCM			
	Accuracy	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure
Ant	69%	66%	72%	69%	26%	28%	37%	31%	63%	67%	68%	68%
JMeter	77%	72%	68%	70%	29%	24%	53%	33%	65%	65%	63%	64%
Log4j	71%	62%	66%	64%	19%	13%	26%	17%	43%	36%	78%	49%
Poi	68%	88%	59%	71%	25%	34%	16%	22%	60%	74%	49%	59%
Xerces-J	94%	94%	88%	91%	49%	28%	35%	31%	74%	59%	80%	68%

TABLE VI
OVERLAP ANALYSIS BETWEEN DCBM AND DM

System	DCBM ∩ DM%	DCBM \ DM%	DM \ DCBM%
Ant	9	74	17
JMeter	8	89	3
Log4j	13	75	12
Poi	11	72	17
Xerces	32	55	13
Overall	**16**	**70**	**14**

TABLE VII
OVERLAP ANALYSIS BETWEEN DCBM AND BCCM

System	DCBM ∩ BCCM %	DCBM \ BCCM %	BCCM \ DCBM %
Ant	39	37	24
JMeter	28	45	27
Log4j	16	67	17
Poi	37	33	30
Xerces	22	43	35
Overall	**32**	**45**	**23**

TABLE VIII
OVERLAP ANALYSIS BETWEEN DM AND BCCM

System	DM ∩ BCCM %	BCCM \ DM %	DM \ BCCM %
Ant	10	71	19
JMeter	11	83	6
Log4j	15	69	16
Poi	15	68	17
Xerces	30	59	11
Overall	**13**	**69**	**18**

TABLE IX
OVERLAP ANALYSIS CONSIDERING EACH MODEL INDEPENDENTLY

System	DCBM \ (BCCM ∪ DM)%	BCCM \ (DCBM ∪ DM)%	DM \ (DCBM ∪ BCCM)%
Ant	51	31	18
JMeter	65	26	9
Log4j	61	28	11
Poi	48	36	16
Xerces	53	33	16
Overall	**59**	**32**	**9**

phenomenon (e.g., the PC1 in Table V is the most important, capturing almost 76% of the phenomenon as compared to 1% of PC4). Finally, the real values reported at row i and column j indicate how much the predictor i contributes in capturing the PCj (e.g., structural scattering captures 100% of PC1). Interestingly, the *structural scattering* predictor is the only predictor fully orthogonal with respect to the other three, since it is the only capturing PC1. As for the other predictors, the semantic scattering and the change entropy information seem to be quite related by capturing the same components (i.e., PC2 and PC3). Finally, the number of developers is the one better capturing the less important component (PC4).

As a next step toward understanding the complementarity of the three prediction models, Tables VI, VII, and VIII report the overlap metrics computed between DCBM-DM, DCBM-BCCM, and DM-BCCM, respectively. In addition, Table IX shows the percentage of buggy classes correctly identified only by each of the single bug prediction models (e.g., identified by DCBM and not by DM and BCCM).

Regarding the overlap between our predictor (DCBM) and the one built using the number of developers (DM), it is interesting to observe that there are some complementarity between the two models, with an overall 70% of correctly classified buggy classes only identified by our model, 14% only by DM, and 16% of instances correctly classified by both models. This result is consistent on all the object systems (see Table VI). A similar trend is shown in Table VII, when analyzing the overlap between our model and BCCM. In this case, our model correctly classified 45% of buggy classes that are not identified by BCCM that is, however, able to capture a 23% of buggy classes missed by our approach. Finally, 32% of buggy classes are correctly identified by both models.

Thus, our prediction model showed a high degree of complementarity with the two competitive models. On the same line, when looking at the overlap metrics between the DM and the BCCM (see Table VIII), we can see that the models are highly complementary too, with the BCCM providing much better performances and identifying 69% of correct buggy classes missed by DM.

Finally, looking at Table IX, we can see that our approach identifies 59% of buggy classes missed by the other two techniques, as compared to 32% of BCCM and 9% of DM. This confirms that (i) our model captures something missed by the competitive models, and (ii) by combining our model with BCCM/DM we could further improve the detection accuracy of our technique.

An example of a buggy class detected only by DCBM can be found in the *Apache Ant* system. The class `Exit` belonging to the package `org.apache.tools.ant.taskdefs` has been modified just one time by a single developer in the time period going from January 2004 to April 2004. However, the sum of the structural and semantic scattering in that period was very high for the involved developer (*461.61* and *5,603.19*, respectively), who modified a total of 38 classes spread over 6 subsystems. In the considered time period the DM does not identify `Exit` as buggy given the single developer who worked on it, and the BCCM fails too due to the single change `Exit` underwent between January and April 2004. Overall, the results achieved in our study allow us to answer our research questions:

- **RQ**$_1$: Our approach showed quite high accuracy in detecting buggy classes. Among the five object systems its accuracy ranged between 66% and 94%, while the F-measure between 64% and 91%.
- **RQ**$_2$: The comparison between our approach and the

two competitive techniques showed (i) its superiority in detecting buggy classes providing a lower number of false positives, and (ii) its high complementarity, paving the way to future developments of our model aimed at further increasing its performances.

VI. Threats to Validity

This section describes the threats that can affect the validity our study. Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is the most important kind of threat for our study, and is related to:

- *Missing or wrong links between bug tracking systems and versioning systems [41]*: although not much can be done for missing links, as explained in the design we verified that links between commit notes and issues are correct;
- *Imprecision due to tangled code changes [42]*. We cannot exclude that some commits we identified as bug-fixes grouped together tangled code changes, of which just a subset represented the committed patch.
- *Imprecision in issue classification made by issue-tracking systems [4]*: while we cannot exclude misclassification of issues (e.g., an enhancement classified as a bug), at least all the systems considered in our study used Bugzilla as issue tracking system, explicitly pointing to bugs in the issue type field;
- *Undocumented bugs present in the system*: while we relied on the issue tracker to identify the bugs fixed during the change history of the object systems, it is possible that undocumented bugs were present in some classes, leading to wrong classifications of buggy classes as "clean" ones.
- *Approximations due to identifying fix-inducing changes using the SZZ algorithm [34]*: at least we used heuristics to limit the number of false positives, for example excluding blank and comment lines from the set of fix-inducing changes.

Threats to *internal validity* concern external factors we did not consider that could affect the variables being investigated. We computed the developer's scattering measures by analyzing the developers' activity on a single software system. However, it is well known that, especially in open source communities and ecosystems, developers contribute to multiple projects in parallel [43]. This might negatively influence the "developer's scattering" assessment made by our metrics. Still, the results of our approach can only improve by considering more sophisticated ways of computing our metrics.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. The metrics used in order to evaluate our defect prediction approach, i.e., accuracy, precision, recall, and F-Measure, are widely used in the evaluation of the performances of defect prediction techniques [27]. Moreover, we used appropriate statistical procedures, i.e., PCA [44], and the computation of overlap metrics to study the orthogonality between our model and the competitive ones.

Threats to *external validity* concern the generalization of results. We analyzed five different systems from different application domains and with different characteristics (number of developers, size, number of classes, etc). However, other systems should be analyzed to corroborate our findings.

VII. Conclusion and Future Work

A lot of effort in the last decade has been devoted to analyze the influence of the development process in the likelihood of introducing bugs. Several empirical studies have been carried out to assess under which circumstances and during which coding activities developers tend to introduce bugs. In addition, bug prediction techniques built on top of process metrics have been proposed. However, changes in source code are made by developers that often work under stressing conditions due to the need of delivering their work as soon as possible.

The role of developer-related factors in the bug prediction field is still a partially explored area. This paper makes a further step ahead, by studying the role played by the *developer's scattering* in bug prediction. Specifically, we defined two measures that consider the amount of code components a developer modifies in a given time period and how these components are spread structurally (*structural scattering*) and in terms of the responsibilities they implement (*semantic scattering*). The defined measures have been evaluated as bug predictors in an empirical study performed on five open source systems. In particular, we build a prediction model exploiting our measures and compared its prediction accuracy with two state-of-the-art techniques exploiting process metrics as predictors. The achieved results showed the superiority of our model and its high level of complementarity with respect to the considered competitive techniques.

Our future research agenda includes:

1) the definition of more sophisticated bug prediction techniques, flanking the defined measures with complementary predictors, like those exploited by Hassan [9].
2) a deeper investigation of the factors causing scattering to developers, and negatively impacting their ability of dealing with code change tasks. We plan to reach such an objective by performing a large survey with industrial and open source developers.
3) the replication of our study on a larger set of systems, implemented in different programming languages.

References

[1] J. Sliwerski, T. Zimmermann, and A. Zeller, "Don't program on fridays! how to locate fix-inducing changes," in *Proceedings of the 7th Workshop Software Reengineering*, May 2005.

[2] L. T. Jon Eyolfso and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 153–162.

[3] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 491–500.

[4] E. J. W. J. Sunghun Kim and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 2, pp. 181–196, 2008.

[5] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '12, 2012, pp. 104–113.

[6] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 452–461.

[7] A. N. Taghi M. Khoshgoftaar, Nishith Goel and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Software Reliability Engineering*. IEEE, 1996, pp. 364–371.

[8] J. S. M. Todd L. Graves, Alan F. Karr and H. P. Siy, "Predicting fault incidence using software change history," *Software Engineering, IEEE Transactions on*, vol. 26, no. 7, pp. 653–661, 2000.

[9] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE*. Vancouver, Canada: IEEE Press, 2009, pp. 78–88.

[10] R. Bell, T. Ostrand, and E. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9178-4

[11] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Programmer-based fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 19:1–19:10. [Online]. Available: http://doi.acm.org/10.1145/1868328.1868357

[12] R. Moser, W. Pedrycz, and G. Succi, "Analysis of the reliability of a subset of change metrics for defect prediction," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: ACM, 2008, pp. 309–311. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414063

[13] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does measuring code change improve fault prediction?" in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, ser. Promise '11. New York, NY, USA: ACM, 2011, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2020390.2020392

[14] W. P. Raimund Moser and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *International Conference on Software Engineering (ICSE)*, ser. ICSE '08, 2008, pp. 181–190.

[15] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, Oct 1996.

[16] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 31, no. 10, pp. 897–910, 2005.

[17] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switchess," *Software Engineering, IEEE Transactions on*, vol. 22, no. 12, p. 886894, 1996.

[18] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 580–586. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062558

[19] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 9–. [Online]. Available: http://dx.doi.org/10.1109/PROMISE.2007.10

[20] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 4–14.

[21] M. Tufano, "An empirical study on factors impacting the commits bug-proneness - technical report," 2015. [Online]. Available: http://www.cs.wm.edu/semeru/data/ICSME15-Fix-Inducing-Commits/files/technicalReport.pdf

[22] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, no. 6, pp. 476–493, June 1994.

[23] W. M. Khaled El Emam and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, p. 6375, 2001.

[24] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, p. 297310, 2003.

[25] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 452–461. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349

[26] A. P. Nikora and J. C. Munson, "Developing fault predictors for evolving software systems," in *Proceedings of the 9th IEEE International Symposium on Software Metrics*. IEEE CS Press, 2003, pp. 338–349.

[27] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, p. 531577, 2012.

[28] A. E. Hassan and R. C. Holt, "Studying the chaos of code development," in *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003.

[29] ——, "The top ten list: dynamic fault prediction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005*, ser. ICSM '05. IEEE Computer Society, 2005, pp. 263–272.

[30] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[31] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[32] D. D. Nucci, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, and A. D. Lucia, "Don't multitask! the role of developer's scattering in defect prediction - replication package - https://dibt.unimol.it/fpalomba/reports/devconfusion/," 2015.

[33] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012, June) The promise repository of empirical software engineering data. [Online]. Available: http://promisedata.googlecode.com

[34] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*. ACM, 2005.

[35] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, 2001, pp. 13–22.

[36] L. M. Y. Freund, "The alternating decision tree learning algorithm," in *Proceeding of the Sixteenth International Conference on Machine Learning*, 1999, pp. 124–133.

[37] R. Kohavi, "The power of decision tables," in *8th European Conference on Machine Learning*. Springer, 1995, pp. 174–189.

[38] S. le Cessie and J. van Houwelingen, "Ridge estimators in logistic regression," *Applied Statistics*, vol. 41, no. 1, pp. 191–201, 1992.

[39] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1961.

[40] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Eleventh Conference on Uncertainty in Artificial Intelligence*. San Mateo: Morgan Kaufmann, 1995, pp. 338–345.

[41] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595716

[42] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 121–130.

[43] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 280–289.

[44] I. Jolliffe, *Principal Component Analysis*. John Wiley & Sons, Ltd, 2005. [Online]. Available: http://dx.doi.org/10.1002/0470013192.bsa501