Developer-Related Factors in Change Prediction
An Empirical Assessment

Catolino, Gemma; Palomba, Fabio; De Lucia, Andrea; Ferrucci, Filomena; Zaidman, Andy

DOI
10.1109/ICPC.2017.19

Publication date
2017

Document Version
Accepted author manuscript

Published in
Proceedings - 2017 IEEE 25th International Conference on Program Comprehension, ICPC 2017

Developer-Related Factors in Change Prediction: An Empirical Assessment

Gemma Catolino[1], Fabio Palomba[2], Andrea De Lucia[1], Filomena Ferrucci[1], Andy Zaidman[2]
[1]University of Salerno — [2]Delft University of Technology
gcatolino@unisa.it, f.palomba@tudelft.nl, adelucia@unisa.it, fferrucci@unisa.it, a.e.zaidman@tudelft.nl

***Abstract*—Predicting the areas of the source code having a higher likelihood to change in the future is a crucial activity to allow developers to plan preventive maintenance operations such as refactoring or peer-code reviews. In the past the research community was active in devising change prediction models based on structural metrics extracted from the source code. More recently, Elish *et al.* showed how evolution metrics can be more efficient for predicting change-prone classes. In this paper, we aim at making a further step ahead by investigating the role of different developer-related factors, which are able to capture the complexity of the development process under different perspectives, in the context of change prediction. We also compared such models with existing change-prediction models based on evolution and code metrics. Our findings reveal the capabilities of developer-based metrics in identifying classes of a software system more likely to be changed in the future. Moreover, we observed interesting complementarities among the experimented prediction models, that may possibly lead to the definition of new combined models exploiting developer-related factors as well as product and evolution metrics.**

***Keywords*-Change prediction; Empirical Studies; Mining Software Repositories;**

I. Introduction

During software maintenance and evolution, *change is the rule rather than the exception* [1]. Classes undergo frequent modifications due to continuous change requests, lack of a deep understanding of the requirements or lack of communication with the stakeholders [1]. In such a scenario, and because the need of meeting strict deadlines, software developers often perform maintenance activities in an undisciplined manner, leading to the erosion of the original design and, thus, reducing the quality of a software system [2].

Knowing in advance the code elements potentially exhibiting a higher change-proneness is vital for developers for two main reasons: on the one hand, change-proneness can be considered as a quality indicator that can be used to warn developers when touching code that should be refactored [3]; on the other hand, developers can plan preventive maintenance operations, such as refactoring [4], peer-code reviews [5], and testing [6], aimed at increasing the quality of the code and reducing future maintenance effort and costs [4].

Change prediction is widely recognized as an effective technique to identify the classes more prone to be modified in the future, being able to help developers in both planning preventive maintenance actions and understanding the complexity of source code [7]. For this reason, researchers devoted a lot of attention to the problem by (i) analyzing the factors influencing the change-proneness of classes [6], [8], [9], [10], [11], and (ii) devising prediction models able to alert developers about the classes on which preventive actions should be focused on [12], [13], [14], [15].

Most of the previous work relied on product metrics (*e.g.,* the Chidamber and Kemerer metrics [16]) as indicators of the change-proneness of classes. The underlying assumption is that code elements having low quality are more prone to be subject of changes in the future. For example, Zhou *et al.* [3] investigated which cohesion, coupling, and inheritance metrics are more suitable for predicting change-prone classes, finding a subset of them that should be used in the context of change prediction models. At the same time, they also showed that the number of lines of code is not a good predictor [3].

More recently, Elish *et al.* [17] started investigating the role of process metrics as predictors of change-prone classes. To this aim, they theoretically and empirically evaluated a new set of metrics (called "evolution metrics") that characterized the history of a class in order to delineate its future change-proneness. For instance, they considered the number of previous modifications a class underwent during a given time period. The application of a prediction model based on such new metrics produces more accurate predictions than the ones provided when using the traditional code metrics suggested by Zhou *et al.* because of the direct relationship existing between previous and future modifications of a class [17].

Although Elish *et al.* exploited some process metrics, they did not take into account developer-related factors that considering how developers apply changes in the source code could be able to capture the **complexity of the development process**. For instance, it is still unclear whether non-focused developers that apply scattered changes over the entire system tend to introduce maintainability pitfalls that lead to increase the change-proneness of the modified classes. Our conjecture is that such aspects can be a useful source of information to predict classes more likely to be changed in the future. In this paper, we aim at verifying our conjecture by studying *the role of the metrics measuring the complexity of the development process in change prediction*. In our study we investigated three prediction models previously defined in literature each one based on metrics that capture the complexity of the development process under different perspectives, *i.e.,* (i) the Basic Code Change Model (BCCM) proposed by Hassan [18] which relies on the entropy of changes applied by developers, (ii) the Developer Changes Based Model (DCBM) devised by

Di Nucci *et al.* [19] that considers to what extent developers apply scattered changes in the system, and (iii) the Developer Model (DM) proposed by Bell *et al.* [20] which analyzes how many developers touched a code element over time.

Even though the models that we investigate have originally been proposed for fault prediction, we conjecture they can be adopted in the change prediction context since they are based on metrics able to influence the change-proneness of classes as well. For instance, the lack of coordination between multiple developers working on the same code element may lead to the introduction of design pitfalls that negatively influence the maintainability of source code [21], possibly making it more change-prone. In order to assess the performance of the three prediction models we employed ten open source software systems with different size scope.

Moreover, to have a comprehensive view of the usefulness of the experimented models, we compared their performance with the ones achieved by the state-of-the-art change prediction models proposed by Elish *et al.* [17] and Zhou *et al.* [3].

The results of our study highlight the good prediction capabilities of the experimented prediction models, which range between 60% and 78% in terms of accuracy. In particular, we observed that the best performance is achieved by the model defined by Di Nucci *et al.* [19]. When compared to the model exploiting the evolution metrics devised by Elish *et al.*, DCBM still produces better performances. This result highlights how previous changes of a class are not enough for adequately predicting its future change-proneness, while measuring the complexity of the development process can give more accurate predictions. Furthermore, the change prediction model relying on code metrics achieved the worst performance (accuracy=57%), indicating that structural analysis is not sufficiently suitable for predicting change-prone classes.

Finally, all the experimented prediction models showed interesting complementarities in the set of change-prone classes correctly predicted. Indeed, different models capture different change-prone instances, possibly indicating that better prediction abilities can be obtained by combining the predictors used by the experimented models.

Structure of the paper. Section II discusses the related literature in the context of change prediction. In Section III the design of the empirical study is described, while Section IV reports the results achieved when evaluating the performances of the experimented change prediction models. Section V discusses the threats that could affect the validity of our study. Finally, Section VI concludes the paper.

II. Related Work

The analysis of the change-proneness of classes has been explored by the research community from two main perspectives. A consistent body of research analyzed the factors influencing the phenomenon [8], [9], [10], [11], [6], while others focused on understanding the role of product and evolution metrics to predict the future change-proneness of classes [22], [20], [19], [23], [17]. Since this paper is about change prediction models, in the following we summarize the related literature on previous research in this branch.

Product metrics have been widely exploited in the context of change prediction [24]. Lindvall [25] found that larger classes are statistically more change-prone than classes having a small size, and that developers tend to apply more changes to such classes during maintenance and evolution [26]. Further studies showed that coupling metrics are relevant measures to estimate the changeability of source code [27], [28], [29], while Chaumun *et al.* [30] and Tsantalis *et al.* [23] generalized the usefulness of CK metrics [16] for change prediction. The statistical analyses conducted by Lu *et al.* [31] and Malhotra *et al.* [32] clarified which Object Oriented metrics are better suited for change prediction, reporting a set of cohesion, coupling, and inheritance metrics that should be used in this context. On the basis of these results, several prediction models based on product metrics have been devised.

Romano *et al.* [33] relied on code metrics for predicting change-prone fat interfaces, while Eski *et al.* [34] proposed a model based on both CK and QMOOD metrics [35] to estimate change-prone classes and to determine parts which should be tested first and more deeply.

Other previous research tried to estimate the change-proneness of classes using alternative methodologies. For instance, the combination between dependencies mined from UML diagrams [36] and code metrics has been proposed [12], [13], [14], [15]. Also genetic and learning algorithms have been proposed in this context [37] [38] [39]. Specifically, Malhotra *et al.* [37] validated the CK metrics suite for building an efficient software quality model which predict change prone classes with the help of Gene Expression Programming. Marinescu [38] reported the goodness of GAs for both change- and fault-prediction. Finally, Peer *et al.* [39] devised the use of adaptive neuro-fuzzy inference system (ANFIS) to estimate the change-proneness of classes.

Later on, Zhou *et al.* [3] showed that size metrics may lead to multi-collinearity [40] when mixed together with other cohesion and coupling metrics. As a result, they suggested to avoid using the LOC metric in product-based change prediction models [3].

The closest works to the one proposed in this paper are the studies by Elish *et al.* [17] and Girba *et al.* [41]. Elish *et al.* [17] reported the potential usefulness of evolution metrics for change prediction. In particular, they defined a set of historical metrics such as (i) the birth date of a class, (ii) the total amount of changes applied in the past, and (iii) the date of the first and the last modification applied on a class. Their findings showed how such evolution metrics may be useful for predicting change-prone classes. Girba *et al.* [41] defined a tool that suggests change-prone code elements by summarizing previous changes. In a small-scale empirical study involving two systems, they observed that previous changes can effectively predict future modifications.

Besides the evolution metrics defined by Elish *et al.* [17] and Girba *et al.* [41], in this paper we also analyzed the role of developer-related factors that have been shown to be relevant

for prediction purpose in other contexts [19].

III. Empirical Study Definition and Design

The *goal* of the empirical study is to evaluate to what extent metrics capturing the complexity of the development process are useful when discovering change-prone source code classes, with the *purpose* of improving the allocation of resources in preventive maintenance activities (*e.g.,* refactoring, code inspections *etc.*) focusing on classes having a higher change-proneness. The *quality focus* is on the prediction performance and complementarity between the investigated approaches, while the *perspective* is of researchers who want to evaluate the effectiveness of using developer-related factors when identifying change-prone classes.

The *context* of the study consists of ten open source software systems having different size scope. Table I reports the characteristics of the considered systems, and in particular (i) the software history that we investigated, (ii) the percentage of change-prone classes identified (as explained later), and (iii) the size in terms of number of commits, developers, classes, methods, and KLOC.

The specific research questions formulated in this study are the following:

- **RQ$_1$:** *To what extent are developer-based prediction models able to correctly estimate the change-proneness of classes?*
- **RQ$_2$:** *How does the performance of developer-based prediction models differ from the ones of existing change prediction models?*
- **RQ$_3$:** *To what extent are developer-based change prediction models complementary to existing change prediction models?*

To answer **RQ$_1$** and understand the predictive power of developer-related factors in change prediction, we decided to test the performance of three prediction models (we refer to them as *developer-based* model since they rely on developer related factors):

1) The Basic Code Change Model (BCCM) defined by Hassan [18], which relies on the entropy of changes applied by developers in a time window of size α.
2) The Developer Changes Based Model (DCBM) proposed by Di Nucci *et al.* [19]. It employs the structural and semantic scattering of the developers that worked on a code element in a time window of size α as predictors. The structural scattering measures the distance between every pair of classes modified by the developer, while the semantic scattering computes the degree of textual similarity between every pair of classes modified by the developer.
3) The Developer Model (DM) devised by Bell *et al.* [20] that takes into account the number of developers that worked on a specific component of source code in a time period of size α.

While such models have originally been defined in the context of fault prediction, the choice of using them for change prediction was guided by the will of exploring the role of different aspects of the development process on the change-proneness of classes. For instance, having a high entropy of changes might indicate the presence of a complex development process where developers apply changes in an undisciplined manner that lead to source code that is less maintainable and possibly more change-prone in the future.

Once chosen the baseline prediction models *i.e.,* BCCM, DCBM, DM, the subsequent step regarded the identification of the machine learning technique to use for building the change prediction models. The related literature proposed several alternatives (*e.g.,* Tsantalis *et al.* [23] relied on Logistic Regression [42], while Romano and Pinzger [33] suggested the use of Support Vector Machine [43]), however it is still unclear which classifier is able to give the best overall performance.

For this reason, we experimented with several classifiers previously used for prediction purposes from the research community, *i.e.,* ADTree [44], Decision Table Majority [45], Logistic Regression [42], Multilayer Perceptron [46], Support Vector Machine [43], and Naive Bayes [47]. We empirically compared the results achieved when applying each classifier on each experimented baseline model on the software systems in our study (more details on the adopted procedure later in this section), finding that Logistic Regression [42] provided the best performances for all the tested prediction models. Thus, in this paper we report the results of the models built with this classifier. A comprehensive report of the analysis conducted in order to identify the machine learning technique to use is reported in online appendix [48].

To assess the performance of the three prediction models, we split the evolution history of the subject systems into *three-month* time periods and we adopted a three-month sliding window to train and test the change prediction models. Specifically, starting from the first time window TW_1 (*i.e.,* the one starting from the first commit), we train each model on it, and test its performances on the time window TW_2 (*i.e.,* the subsequent three-month period). Then, we moved three months forward to the next time window, training the classifier using the data available in TW_2 and testing the model on TW_3. This process has been repeated until the end of the evolution history of the subject systems.

The choice of the validation methodology was based on two aspects. Firstly, all the models refer to a specific time window of size α in which their own predictors have to be computed. Therefore, this validation technique better fits the characteristics of the experimented models. Secondly, this methodology has been widely used in recent years to test the performance of prediction models [18], [19]. Moreover, the choice of considering three-month periods is based on (i) the results of previous work, such as the one by Hassan [18], and (ii) the findings of the empirical assessment we performed on such a parameter, which showed that the best results for all experimented techniques are achieved when using three-month periods. In particular, we tested time windows of size $\alpha = 1, 2, 3, 6$ months. A report of the results is available in replication package [48].

TABLE I: Characteristics of the Software Projects in Our Dataset

System	Period	% Change-prone Classes	#Releases	#Commits	#Dev.	#Classes	#Methods	KLOCs
ArgoUML	Oct 2002-Dec 2012	28%	16	19,961	31	777-1,415	6,618-10,450	147-249
Apache Ant	Jan 2000-Jul 2014	35%	22	13,054	55	83-813	769-8,540	20-204
Apache Cassandra	Mar 2007-Jan 2012	22%	13	20,026	128	305-586	1,857-5,730	70-111
Apache Xerces	Nov 1999-Feb 2014	19%	16	5,471	34	162-736	1,790-7,342	62-201
aTunes	Aug 2005-Apr 2010	31%	31	6,276	21	141-655	1,175-5,109	20-106
FreeMind	Jun 2000-Feb 2012	28%	16	722	13	25-509	341-4,499	4-103
JEdit	Jan 2005-Jun 2012	24%	29	24,340	18	228-520	1,073-5,411	39-166
JFreeChart	Feb 1999-Jul 2013	33%	23	14,099	15	86-775	703-8,746	15-231
JHotDraw	Jan 2001-Dec 2012	23%	16	1,121	27	159-679	1,473-6,687	18-135
JVLT	Jan 2007-Dec 2012	29%	15	623	16	164-221	1,358-1,714	18-29
Overall	**-**	**-**	**197**	**105,693**	**358**	**25-1,415**	**341-10,450**	**4-249**

To measure the ability of the change prediction models in correctly predicting change-prone classes, we needed an oracle reporting the actual change-prone classes present in each of the time windows analyzed. To the best of our knowledge, a public oracle reporting the *ground-truth* for the phenomenon taken into account is not available in literature. Thus, we needed to build our own oracle. To this aim, we followed the guidelines provided by Romano *et al.* [33], which considered a class change-prone if, in a given time period *TW*, it underwent a number of changes higher than the median of the distribution of the number of changes experienced by all the classes of the system. We made the oracle reporting the change-prone classes of all the ten considered systems publicly available in online appendix [48].

Once we defined the oracle and ran the prediction models on every three-month window, we answered $\mathbf{RQ}_1$ by using three widely-adopted Information Retrieval metrics, namely *accuracy*, *precision* and *recall* [49]. As an aggregate indicator of precision and recall, we also reported the *F-measure*, a metric defined as the harmonic mean of precision and recall [49]. In addition, we reported the *Area Under the ROC Curve* (AUC-ROC) obtained by the experimented prediction models. This metric quantifies the overall ability of a prediction model to discriminate between change-prone and non-change-prone classes. The closer the AUC-ROC to 1, the higher the ability of the classifier to discriminate classes that will change less and more in the future. On the other hand, the closer the AUC-ROC to 0.5, the lower the accuracy of the classifier.

Finally, we also statistically compared the F-measure achieved by the experimented prediction models. To this aim, we exploited the Mann-Whitney test [50] (results are intended as statistically significant at α=0.05). Furthermore, we estimated the magnitude of the measured differences by using Cliff's Delta (or d), a non-parametric effect size measure [51] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $|d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [51].

As for $\mathbf{RQ}_2$, we are interested in understanding to what extent the performance achieved by developer-based prediction models is different from those achievable using existing prediction models. To this aim, we firstly investigated the Evolution Model (EM) proposed by Elish *et al.* [17], which relies on the set of metrics they defined, *i.e.,* (i) birth date of a class, (ii) total amount of changes applied on a class in a time window of size α, and (iii) the date of the first and the last modification applied on a class. Note that this model directly uses the number of previous changes of a class to predict its future change-proneness. It is based on the concept of *"change-caching"*, *i.e.,* classes that underwent more changes in the past will have more changes in the future since they encapsulate most of the complexity of the system.

We also tested the performance of a product-based prediction model. As reported in Section II, the research community has been active in the definition of change prediction models relying on code metrics as predictors [24]. In the context of this paper, we used as baseline the model by Zhou *et al.* [3] which relies on a set of cohesion (*i.e.,* the Lack of Cohesion of Method — LCOM), coupling (*i.e.,* the Coupling Between Objects — CBO — and the Response for a Class — RFC), and inheritance metrics (*i.e.,* the Depth of Inheritance Tree — DIT). The choice of the baseline has been driven by the findings reported by Zhou *et al.*, which showed how such metrics are effective in detecting the change-proneness of classes [3]. In the following, we refer to this model as CM, *i.e.,* Code Metrics Model.

Note that for sake of readability, in Table II we report the (i) abbreviations used over all the paper, (ii) the names, and (iii) a brief description of the investigated models.

To compare the performance of the CM model with the developer- and evolution-based models, we used the same procedures and metrics used in the context of $\mathbf{RQ}_1$, *i.e.,* accuracy, precision, recall, F-measure, and AUC-ROC. Moreover, we have statistically compared the F-Measure achieved by such models. It is worth noting that also in this case we investigated different machine learning techniques, finding again that Logistic Regression [42] provided the best performance. Thus, the comparison between the investigated models is fair.

Finally, to answer $\mathbf{RQ}_3$ and analyze the complementarity between the considered predictors, we investigated to what extent different models correctly classify the change-proneness of different classes. To this aim, we exploited the overlap metrics. Specifically, for each pair m_i and m_j of the experimented prediction models, we computed the overlap between the sets of true positives correctly identified by both models (denoted by $corr_{m_i \cap m_j}$) and the percentage of change-prone classes correctly classified by m_i only and missed by m_j (denoted by $corr_{m_i \setminus m_j}$) defined as follows:

TABLE II: Summary of the five investigated change prediction models

Abbreviation	Name	Description
BCCM [18]	Basic Code Change Model	It is based on the entropy of changes applied by developers in a given time period.
DCBM [19]	Developer Changes Based Model	It takes into account the developers structural and semantic scattering. The first measures how "structurally" far the code components modified by a developer in a given time period are. The second capture how much spread in terms of implemented responsibilities the code components modified by a developer in a given time period are.
DM [20]	Developer Model	It relies on the number of developers who modified a code component in a give time period.
EM [17]	Evolution Model	Based on a set of historical metrics such as the birth date of a class, the total amount of changes applied in the past, and date of the first and the last modification applied on a class.
CM [3]	Code Metrics Model	It relies on a set of cohesion (*i.e.,* LCOM), coupling (*i.e.,* CBO and RFC), and inheritance metrics (*i.e.,* DIT).

$$corr_{m_i \cap m_j} = \frac{|corr_{m_i} \cap corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \quad (1)$$

$$corr_{m_i \setminus m_j} = \frac{|corr_{m_i} \setminus corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \quad (2)$$

where $corr_{m_i}$ represents the set of change-prone classes correctly classified by the prediction model m_i.

IV. Analysis of the Results

In this section we report the results achieved in the study, discussing the performance of the investigated models, and the complementarity between them.

A. RQ_1: The Performances of Developer-based Models

Table III reports the performance achieved by the five investigated change prediction models over the ten considered subject systems. Looking at the table, we can immediately provide quantitative answers to our first research question. In the first place, while developer-based models tend to perform well, it is worth noting that none of them achieves an overall accuracy higher than 78%. Even if this value is still quite positive, it is also important to highlight that a notable percentage of classes (at least 22%) is not correctly classified by using the models independently. Thus, the problem of identifying the change-proneness of classes seems to be not easily addressable by employing models counting single aspects of the development process.

Among the three developer-based models investigated, DCBM [19] tends to perform better than the others, achieving the best scores in term of all the quality metrics computed, *i.e.,* accuracy=78%, precision=61%, recall=70%, F-Measure=66%, AUC-ROC=69%. Based on these results, we can claim that the way developers apply changes in the system has an influence of the likelihood to make the touched classes more change-prone. The superiority of DCBM is particularly evident in the comparison with the DM model (*i.e.,* the model based on the number of developers), where the F-Measure is 8% higher. The result clearly highlights that it is not the simple number of developers working on a class that influences the change-proneness, but rather the way developers apply (scattered) changes in the system. Our findings confirm, in the context of change prediction, previous findings achieved by Di Nucci *et al.* [19], which showed the superiority of the DCBM model in predicting bugs. For instance, consider the case of the class `org.gjt.sp.BufferHistory` of the JEdit system. Between August and October 2009 (*i.e.,* one of the three-month periods considered in our study) the class was modified 19 times by one developer. The DM model predicted the class as non-change-prone. However, in the same time period such developer performed 36 modifications over five different packages, thus accumulating a high level of both semantic and structural scattering. The scattered changes applied by the developer led to a decreasing of the cohesion of the modified classes (*i.e.,* overall, the LCOM[1] increases of 16% in such classes): interestingly, the LCOM of the class `org.gjt.sp.BufferHistory` is the one increasing more (from 3 to 12). This made it more prone to be changed since they encapsulated different responsibilities. Due to the high scattering of the developer, DCBM correctly predicts the change-proneness of the class. Thus, the results seem to delineate that the scattered changes applied by developers can produce some forms of software degradation that have effects on the change-proneness of classes. The statistical analyses conducted (see Table IV) confirm the superiority of DCBM with respect to DM ($\alpha < 0.01$, $d = 0.81$).

A similar discussion can be made when comparing the DCBM and BCCM models. From Table III we can observe that DCBM is able to obtain an F-Measure almost 8% higher than the alternative model. Once again, the improvement is statistically significant ($\alpha < 0.01$) with a large effect size ($d = 0.73$). The gain provided by DCBM is also visible when considering the other evaluation metrics: for instance, the accuracy is about 3% higher, while the recall 7%. Interestingly, both models obtain the same level of AUC-ROC (69%). From a practical point of view, this result indicates that DCBM and BCCM have a comparable overall ability in distinguishing those classes having a high change-proneness with respect to those characterized by a low change-proneness. However, the scattering metrics can capture the phenomenon with a higher accuracy. This is due to the fact that DCBM works at a higher abstraction level than BCCM [18]. Specifically, it considers the way developers apply changes rather than the changes themselves, allowing the model to be more efficient when the change process is not chaotic, but developers continuously perform modifications over different parts of the system. To better understand the reasons behind the different performances of these models, let us consider the case of the class `chartMeter.Legend` belonging to the JFreeChart system. Between April and June 2005, the class underwent 10 of the total 16 changes applied in that time window. In this case, the entropy of changes involving this class is low (*i.e.,* -0.13), since most of the effort has been devoted to maintain it. However, the two developers performing mod-

[1]Note that the lower the LCOM the higher the cohesiveness of a class.

TABLE III: Performances (in percentage) achieved by the investigated change prediction models. A=Accuracy; P=Precision; R=Recall; F-M=F-Measure; AR=AUC-ROC

Project	BCCM					DCBM					DM					EM					CM				
	A	P	R	F-M	AR	A	P	R	F-M	AR	A	P	R	F-M	AR	A	P	R	F-M	AR	A	P	R	F-M	AR
ArgoUML	89	87	88	87	93	87	93	81	86	93	62	57	61	58	52	87	93	82	87	93	73	73	45	55	76
Apache Ant	72	65	79	72	82	71	63	71	67	66	48	51	57	55	51	67	58	63	61	55	56	55	61	58	52
Apache Cassandra	88	79	85	82	91	77	84	90	87	84	65	67	68	68	64	69	62	60	61	58	61	61	65	63	59
Apache Xerces	76	69	73	72	65	87	69	71	70	62	69	66	75	71	52	75	59	66	62	57	81	69	71	70	61
aTunes	62	66	50	58	50	63	58	62	60	67	61	52	55	54	53	63	50	50	50	57	56	42	48	45	52
FreeMind	35	35	28	31	42	68	42	45	44	70	62	62	64	63	63	68	36	44	40	83	52	55	36	44	75
JEdit	75	48	53	51	63	78	42	74	58	62	55	48	52	50	52	64	29	35	32	50	54	42	50	45	50
JFreeChart	71	45	67	56	55	73	42	62	52	55	64	45	61	53	57	73	36	51	44	50	58	48	55	52	67
JHotDraw	97	77	59	67	79	97	66	72	69	75	62	61	69	65	61	98	79	78	79	86	42	46	27	34	51
JVLT	80	50	50	50	50	81	51	76	62	59	49	44	48	46	52	61	51	75	61	59	41	37	43	40	50
Overall	**75**	**56**	**63**	**58**	**69**	**78**	**61**	**70**	**66**	**69**	**60**	**55**	**61**	**58**	**56**	**72**	**55**	**60**	**58**	**65**	**57**	**53**	**50**	**51**	**59**

TABLE IV: Wilcoxon's t-test p-values of the hypothesis F-Measure achieved by a model is $>$ than the compared model. Statistically significant results are reported in bold face. Cliff Delta d values are also shown.

Compared models	p-value	Cliff Delta	Magnitude
DCBM - BCCM	$<$ **0.01**	0.73	large
DCBM - DM	$<$ **0.01**	0.81	large
DCBM - EM	$<$ **0.01**	0.82	large
DCBM - CM	$<$ **0.01**	0.84	large
BCCM - DM	0.07	0.35	medium
BCCM - EM	**0.04**	0.21	small
BCCM - CM	$<$ **0.01**	0.74	large
DM - EM	0.94	0.09	negligible
DM - CM	**0.03**	0.44	medium
EM - CM	**0.03**	0.48	large

ifications in the time window not only apply changes to the `chartMeter.Legend` class, but also to other classes involving 3 different packages. All these modifications were related to the visualization of chart legends, and indeed different other classes related to visualization components (*e.g.,* the `chart.VerticalBarRenderer` class) were modified. However, the changes applied by developers had the effect of reducing the overall quality of such classes, making them more prone to be changed in the future. For instance, the CBO of `chartMeter.Legend` reached 8 (+3 with respect to the previous version). This example seems to confirm the hypothesis behind the good performance of the DCBM, namely the negative effect that scattering changes have on the maintainability of classes.

To broaden the scope of the discussion, we can generally observe that models previously used in the context of fault prediction achieve good performance also when employed in the identification of change-prone classes. This is somehow unexpected and seems to delineate a direct relationship between the complexity of the development process and several maintainability issues, including the change- and fault-proneness of classes. We plan to perform an extensive analysis of the impact of developer-related factors on a wider range of maintainability problems, as well as of the interplay between change- and fault-proneness of classes in our future research agenda.

Summary for RQ$_1$. The investigated developer-based models achieve quite positive results. Among them, the prediction model relying on scattering metrics obtains the highest performances, having an overall F-Measure equals to 66% and an accuracy equals to 78%. The superiority of DCBM is statistically significant and has a large effect size when compared to all the other models.

B. RQ$_2$: The Comparison between Developer-based and State-of-the-art Models

The results achieved by the baseline change prediction models investigated in this study (*i.e.,* EM and CM) are reported in Table III. As it is possible to see, the EM model achieves the same overall F-Measure as the DM and BCCM models (*i.e.,* 58%), while it is always outperformed by the DCBM model (-8% in terms of F-Measure). From Table IV we can observe how the differences between EM and the other developer-based models are often small or negligible, even if mostly statistically significant. The only exception regards the comparison between DCBM and EM, where the differences are statistically significant ($\alpha < 0.01$) and the magnitude is large ($d = 0.82$). When considering the CM model, we can see that EM is generally a better predictor (the overall F-Measure is 58% vs 51% for EM and CM, respectively) and, indeed, the results are statistically confirmed ($\alpha = 0.03$, $d = 0.48$).

Generally, it is important to remark that EM is the only model that directly measures the previous number of changes of a class to predict its future change-proneness: our results indicate that this feature is not able to characterize the future change-proneness of classes better than other predictors. This confirms previous findings by Ekanayake *et al.* [52] on the variability of the change-proneness of classes during different stages of software evolution. As a consequence, the previous knowledge about the number of changes a class underwent is not always suitable to correctly identify change-prone classes in future versions of a software system. Further analyzing the predictions provided by EM, we discovered that it is generally effective when a class has a central role in the architecture of a system and, as such, usually undergoes a high number of changes. For example, in the JHotDraw system, the class `svg.io.SVGFigureFactory` is responsible for performing the main functionality of the entire project, *i.e.,* it manages the graph creation. This class is present in the

system since its first commit and it was frequently modified by developers among all the time windows analyzed. In this case, the predictors used by the EM model (*e.g.,* previous changes and birth date) are particularly effective since they characterize well the change-proneness of the class. On the other hand, the performance decreases in cases where a significant restructuring of the system's architecture is applied, since the responsibilities of several code artifacts are modified and, therefore, predictors such as the birth date or the previous changes are less meaningful. For instance, in the time window ranging between December and February 2006 the APACHE ANT developers performed an entire restructuring of the system, which led to the removal of some old classes as well as the re-distribution of the responsibilities of several code artifacts[2]. As a consequence, the data considered by the EM model was not sufficient to correctly predict the change-proneness of classes: in fact, the accuracy achieved by the model in that time window was 43%. Noticeably, in the same time period the DCBM and DM models reached an accuracy equal to 87% and 83%, respectively. As excepted, in the considered period the developers was busy modifying the source code and, thus, models relying on such information were performing better.

On the one hand, our results confirm previous findings on the potential usefulness of the evolution metrics in the context of change prediction [17]. On the other hand, we also found how the *"change-caching"* concept exploited by this model is valid on classes having a central role in the system, while it has less effect in other cases. At the same time, we showed that (i) other metrics based on developers can be effectively used for prediction purpose, and (ii) they seem to capture information orthogonal with respect to the EM model.

Switching the attention to the results obtained by the model relying on code metrics, we can observe that developer-based prediction models generally obtain higher performances than the product-based baseline. Indeed, all these models have an overall F-Measure always higher than the CM model. For instance, DCBM achieves, overall, an F-Measure 15% higher than the model based on code metrics (66% vs 51%). The superiority of DCBM is also confirmed when considering all the other evaluation metrics, *i.e.,* accuracy=+18%, precision=+8%, recall=+20%, AUC-ROC=+10%. This result contradicts previous findings [3], [31], demonstrating that the use of code metrics is not enough to efficiently predict change-prone classes. A clear example is represented by the class `xerces.dom.ElementImpl` of the APACHE XERCES project. During the time window between May and July 2007, the class experienced only three changes (*i.e.,* it is non-change-prone) applied by two different developers, who focused all their activities on the maintenance of classes belonging to the `xerces.dom` package. As a consequence, the value of their scattering metrics is zero, since they never performed modifications outside the scope of the package [19]. Thus, the DCBM model correctly marked this class as non-change-prone. At the same time, the class has an LCOM=28 and a CBO=7. Both the metrics are higher than the average metric values of the other classes composing the system, and for this reason the CM model wrongly marked the class as change-prone. This example highlights an important aspect related to the maintainability of source code: indeed, even if the code may be considered poorly maintainable looking at the values of code metrics, this seems to be not always a real issue since developers performing focused maintenance activities (thus, being more expert on the modified code) can keep its complexity under control.

[2]As indicated in the release notes of the version 1.7.1, which correspond to that time period: http://tinyurl.com/hqwazgg

Summary for RQ$_2$. Developer-based prediction models generally perform better than the existing models. This is particularly true when considering the DCBM model, which has an overall F-Measure 15% higher than the CM model and 8% higher than EM.

C. RQ$_3$: The Complementarity of the Investigated Models

Table V reports the complementarity between each pair of prediction models. Note that for sake of space limitations, the results on the complementarity have been aggregated by considering the overall overlap between the models. A complete report of the findings on each system is available in online appendix [48].

As it is possible to observe from the table, all the investigated prediction models are complementary to each other, thus being able to correctly point out different sets of change-prone classes. To better understand the reasons behind such complementarities, we deeper analyzed the predictions provided by different models. Firstly, it is worth discussing the complementarity between DCBM and the other models. When considering the relationship between scattering and code metrics, we observed a consistent set of change-prone classes (*i.e.,* 43%) classified by both the prediction models, but at the same time in almost 40% of the cases the only model able to correctly predict the change-proneness is the DCBM model. Finally, 27% of change-prone classes have been identified only using code metrics. This result highlights the high complementarity between the two models, showing that different predictors work well on different sets of classes.

As for the comparison between DCBM and DM, we observe that 51% of the predicted change-prone classes are in the intersection, while 39% of change-prone classes are detected correctly by only the DCBM model. Finally, the change-proneness of a smaller percentage of classes (10%) can be solely detected using the DM model. Thus, the two models partially complement each other, making prediction improvements conceivable. An interesting case explaining when the DM model is able to outperform the DCBM model can be found in the FREEMIND project (the smallest one of our dataset). Here the seven developers of the system often perform changes to a few classes located in the two core packages. Due to the small structure of the system, the scattering metrics

TABLE V: Overlap among the experimented change prediction models.

	A=BCCM			A=DCBM			A=DM			A=EM			A=CM		
	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A
B=BCCM	-	-	-	63	14	23	59	22	19	55	26	19	53	28	19
B=DCBM	63	14	23	-	-	-	51	39	10	53	26	21	43	40	27
B=DM	59	22	19	51	39	10	-	-	-	59	20	21	54	27	19
B=EM	55	26	19	53	26	21	59	20	21	-	-	-	47	31	22
B=CM	53	28	19	43	40	27	54	27	19	47	31	22	-	-	-

cannot correctly capture the developers' activities and, thus, they always have low values. In such case, the DM model produces more reliable predictions: indeed, it is worth noting that this project is the only one where the DM model performs better than the DCBM one (see Table III).

The discussion is similar when comparing the DCBM and BCCM models. Even if the model based on scattering metrics generally achieved better performance than the BCCM model (Table III), we observed an interesting complementarity that may lead to an additional improvement in the prediction through a combination. In fact, Table V shows that the change-proneness of almost 37% of classes can be correctly detected by only one of the two models (*i.e.,* 23% of correct prediction have been made only by DCBM, 14% only by BCCM). Moreover, it is worth noting that the complementarity between BCCM and the other models is high as well. For instance, when compared to the CM model, we found 28% of correct predictions performed by the BCCM only and a further 19% of classes for which the change-proneness has been identified using code metrics. An interesting example is represented by the class `thrift.CassandraServer` which had a value of LCOM=44 and an RFC=23 in the time window between March and May 2010. In that period, this class has been changed 13 times, being classified as an actual change-prone class. However, the BCCM model was not able to correctly mark its change-proneness because the class always changed together with a few other classes of the system (on average, 2 classes). As a consequence, the entropy of changes is low. On the other hand, the poor quality of the class was a relevant indicator of the change-proneness. Furthermore, it is important to note that also the evolution metrics have nice complementarities with the other models. For instance, when comparing EM and BCCM, we observed that in 26% of the cases the change-proneness of classes can be correctly identified by the EM model only. At the same time, the contribution provided by the EM model is still more valuable in comparison to the CM model, where 31% of the change-prone classes are identified by using only the evolution metrics. An interesting example of a change-prone class correctly classified by EM and missed by CM is present in the ARGOUML project. During the time period between October and December 2006, the class `ui.ProjectBrowser` underwent 19 changes, while it has been introduced at the beginning of the project. Even though the structural metrics do not indicate issues in the maintainability of the class (*i.e.,* LCOM=6, CBO=2, DIT=2, RFC=4), it tends to change frequently, being an actual class to keep under control. In this case, the CM model does not recognize the change-proneness of the class, while the evolution metrics are able to better characterize its future maintainability. Conversely, an example of a class identified by CM and missed by EM in the same ARGOUML project is `generator.GeneratorJava`. This class has been introduced during the time window between March and May 2006 (*i.e.,* in the middle of the observed history), where it underwent 10 changes. Since the class has not been introduced in the early stages of software development, the EM model was not able to correctly mark this class as change-prone. On the other hand, the class contains a well-known design issues, *i.e.,* it is affected by a *Complex Class* code smell. Thus, the code metrics are particularly high (*e.g.,* LCOM=49) and effective in capturing the change-proneness of the class.

All in all, the analyses conducted show that the problem of change prediction cannot be solved by only relying on a subset of metrics considered. More importantly, different models are able to capture different change-prone classes: from a practical point of view, this means that the investigated developer-based metrics can nicely complement evolution metrics, possibly providing additional performance improvements when combined. At the same time, the CM model can provide further insights, being able to correctly recognize the change-proneness of a good portion of classes missed by other models (*e.g.,* CM identified 22% of classes that the EM model was not able to identify).

Summary for RQ$_3$. All the investigated models show nice complementarities, being able to correctly capture the change-proneness of the different classes. As a consequence, our findings reveal the possibility to achieve better performance when considering a combination of the predictors considered in this study.

V. THREATS OF VALIDITY

This section describes the threats that can affect the validity of our study.

Construct Validity. Threats to *construct validity* concern the relationship between theory and observation. We exploited the guidelines provided by Romano *et al.* [33] in order to build a *golden set* reporting the actual change-prone classes present in each of the analyzed time windows. This strategy has been widely used in the past to assess the change-proneness of classes [3], [17], [34], and it is recognized as a efficient way to distinguish change and non-change prone classes [33].

Internal Validity. A factor that possibly could have affected the variables investigated regards the evaluation procedure we

exploited to test the different prediction models. In particular, since we had the need to exploit change history information to compute the metrics composing the experimented developer-based models, the evaluation design adopted in our study is different from the ten-fold validation [53] generally exploited in the context of change prediction. In particular, we split the change history of the object systems into three-month time periods and we adopted a three-month sliding window to train and test the experimented fault prediction models. This type of validation is typically adopted when using process metrics as predictors [18], although it might be penalizing when using code metrics, which are typically assessed using a ten-fold cross validation.

Another threat is related to the use of developer-based and evolution metrics as predictors of the change-proneness of classes. Indeed, they somehow encapsulate the concept of change, possibly producing an "interplay" between independent and dependent variables of a prediction model. While the model proposed by Elish *et al.* [17] directly uses the number of changes a class underwent by a class in a previous time window as predictor of the future change-proneness of that class, we carefully verified whether this possible interplay produced unreliable results, finding that the usefulness of the model is limited to the cases where a class has a central role in the system. As for the BCCM, DCBM, and DM models, it is important to note that all of them rely on metrics able to capture the complexity of the development process under different perspectives (*e.g.,* the number of developers who worked on a code component). Thus, they provide a higher abstraction level and do not directly measure the change-proneness of a class.

Conclusion Validity. Threats to *conclusion validity* refer to the relation between treatment and outcome. In order to evaluate the change prediction models we used metrics such as accuracy, precision, recall, F-Measure, and AUC-ROC, which are widely used in the evaluation of the performances of prediction models. Moreover, we also applied appropriate statistical procedures, *i.e.,* the Wilcoxon [50] and the Cliff's tests [51], to understand whether the differences in the performance of the experimented models were significant.

External Validity. As for the generalizability of the results, we analyzed ten different systems from different application domains and having different characteristics (size, number of classes, *etc.*). However, we are aware that our study is based on systems developed in Java only, and therefore future investigations aimed at corroborating our findings on a different set of systems would be worthwhile.

VI. Conclusion

Predicting the classes more likely to change in the future is an effective way to focus preventive maintenance activities on specific parts of a software system. While several researchers relied on code or evolution metrics to build change prediction models, little knowledge is available on the actual usefulness of developer-related factors in this context. This paper aimed at bridging this gap, by providing an empirical analysis of the performance achieved by three developer-based change prediction models on a set of ten software systems. Specifically, the contributions made by this paper are:

1) **An empirical investigation into the role of developer-related factors in change prediction.** To this aim, we analyzed the performance attained by three prediction models relying on metrics able to capture the complexity of the development process under different perspectives [19], [18], [20].
2) **A comparison between developer-based and state-of-the-art change prediction models.** We compared the prediction capabilities of developer-based models with two baseline approaches, *i.e.,* the Evolution Model [17] and the Code Metric model [3].
3) **An analysis of the complementarity between the investigated models.** We evaluated the orthogonality of the different experimented models by computing overlap metrics and providing qualitative examples to understand under which situations a given model performs better than others.

The achieved results provide several findings:

- Developer-based change prediction models generally show good performance. Among them, the DCBM proposed by Di Nucci *et al.* [19] shows the best performance, reaching an overall F-Measure of 66% and an accuracy equals to 78%.
- Developer-based change prediction models work better than a model built using code metrics. In particular, when developers apply focused modifications in a given time period they are able to keep the complexity of the source code under control even in the cases where the code metrics highlight design issues.
- The studied models show interesting complementarities, indicating that different metrics are suitable for predicting the change-proneness of different classes.

Our observation of complementarity of models using different sources of information is our main input for future research in this field. Indeed, we plan to define a change prediction model that efficiently combines different sources of information. We also plan to corroborate our results on a larger set of software systems. Finally, a very important next step that we envision is to perform an extensive analysis of a wide range of maintainability problems and how they are impacted by developer-related factors. Part of this analysis is to study the relationship between these developer-related factors and the interplay between change-proneness and fault-proneness.

References

[1] M. M. Lehman and L. A. Belady, Eds., *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., 1985.

[2] D. L. Parnas, "Software aging," in *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE, 1994, pp. 279–287.

[3] Y. Zhou, H. Leung, and B. Xu, "Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 607–623, 2009.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.

[6] Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez, "Change-based test selection: An empirical evaluation," *Empirical Softw. Engg.*, vol. 21, no. 5, pp. 1990–2032, 2016.

[7] A. G. Koru and H. Liu, "Identifying and characterizing change-prone classes in two large-scale open-source products," *Journal of Systems and Software*, vol. 80, no. 1, pp. 63 – 73, 2007.

[8] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, 2012.

[9] N. N. Miryung Kim, Tom Zimmermann, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Transactions on Software Engineering*, vol. 40, July 2014.

[10] M. Di Penta, L. Cerulo, Y. G. Gueheneuc, and G. Antoniol, "An empirical study of the relationships between design pattern roles and class change proneness," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 2008, pp. 217–226.

[11] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, "Design patterns and change proneness: an examination of five evolving systems," in *Proc. Int'l Workshop on Enterprise Networking and Computing in Healthcare Industry*, 2003, pp. 40–49.

[12] A. R. Sharafat and L. Tahvildari, "A probabilistic approach to predict changes in object-oriented software systems," in *Proc. Conf. on Softw. Maintenance and Reengineering (CSMR)*. IEEE, 2007, pp. 27–38.

[13] A.-R. Han, S.-U. Jeon, D.-H. Bae, and J.-E. Hong, "Behavioral dependency measurement for change-proneness prediction in uml 2.0 design models," in *32nd Annual IEEE International Computer Software and Applications Conference*. IEEE, 2008, pp. 76–83.

[14] A. R. Sharafat and L. Tahvildari, "Change prediction in object-oriented software systems: A probabilistic approach," *Journal of Software*, vol. 3, no. 5, pp. 26–39, 2008.

[15] A.-R. Han, S.-U. Jeon, D.-H. Bae, and J.-E. Hong, "Measuring behavioral dependency for improving change-proneness prediction in uml-based design models," *Journal of Systems and Software*, vol. 83, no. 2, pp. 222–234, 2010.

[16] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Softw. Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[17] M. O. Elish and M. Al-Rahman Al-Khiaty, "A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software," *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 407–437, 2013.

[18] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Int'l Conf. Software Engineering (ICSE)*. IEEE, 2009, pp. 78–88.

[19] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Trans. on Softw. Engineering*, p. to appear, 2017.

[20] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Softw. Engg.*, vol. 18, no. 3, pp. 478–505, 2013.

[21] R. E. Kraut and L. A. Streeter, "Coordination in software development," *Commun. ACM*, vol. 38, no. 3, pp. 69–81, Mar. 1995.

[22] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[23] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the probability of change in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 601–614, 2005.

[24] R. Malhotra and A. Bansal, "Predicting change using software metrics: A review," in *Int'l Conf. on Reliability, Infocom Technologies and Optimization (ICRITO)*. IEEE, 2015, pp. 1–6.

[25] M. Lindvall, "Are large C++ classes change-prone? An empirical investigation," *Software-Practice and Experience*, vol. 28, no. 15, pp. 1551–1558, 1998.

[26] ——, "Measurement of change: stable and change-prone constructs in a commercial c++ system," in *Proc. Int'l Software Metrics Symposium*. IEEE, 1999, pp. 40–49.

[27] L. C. Briand, J. Wust, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 1999, pp. 475–482.

[28] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 491–506, 2004.

[29] M. Abdi, H. Lounis, and H. Sahraoui, "Analyzing change impact in object-oriented systems," in *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*. IEEE, 2006, pp. 310–319.

[30] M. A. Chaumun, H. Kabaili, R. K. Keller, and F. Lustman, "A change impact model for changeability assessment in object-oriented software systems," in *Proc. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 1999, pp. 130–138.

[31] H. Lu, Y. Zhou, B. Xu, H. Leung, and L. Chen, "The ability of object-oriented metrics to predict change-proneness: a meta-analysis," *Empirical software engineering*, vol. 17, no. 3, pp. 200–242, 2012.

[32] R. Malhotra and M. Khanna, "Investigation of relationship between object-oriented metrics and change proneness," *International Journal of Machine Learning and Cybernetics*, vol. 4, no. 4, pp. 273–286, 2013.

[33] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE, 2011, pp. 303–312.

[34] S. Eski and F. Buzluca, "An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes," in *Proc. Int'l Conf Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 566–571.

[35] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[36] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[37] R. Malhotra and M. Khanna, "A new metric for predicting software change using gene expression programming," in *Proc. Int'l Workshop on Emerging Trends in Software Metrics*. ACM, 2014, pp. 8–14.

[38] C. Marinescu, "How good is genetic programming at predicting changes and defects?" in *Int'l Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2014, pp. 544–548.

[39] A. Peer and R. Malhotra, "Application of adaptive neuro-fuzzy inference system for predicting software change proneness," in *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on*. IEEE, 2013, pp. 2026–2031.

[40] R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & Quantity*, vol. 41, no. 5, pp. 673–690, 2007.

[41] T. Girba, S. Ducasse, and M. Lanza, "Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes," in *Proc. Int'l Conf. Softw. Maintenance (ICSM)*. IEEE, 2004, pp. 40–49.

[42] S. Le Cessie and J. C. Van Houwelingen, "Ridge estimators in logistic regression," *Applied statistics*, pp. 191–201, 1992.

[43] L. Bottou and V. Vapnik, "Local learning algorithms," *Neural Comput.*, vol. 4, no. 6, pp. 888–900, Nov. 1992.

[44] Y. Freund and L. Mason, "The alternating decision tree learning algorithm," in *icml*, vol. 99, 1999, pp. 124–133.

[45] R. Kohavi, "The power of decision tables," in *European conference on machine learning*. Springer, 1995, pp. 174–189.

[46] F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," DTIC Document, Tech. Rep., 1961.

[47] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Proc. Conf. on Uncertainty in artificial intelligence*. Morgan Kaufmann, 1995, pp. 338–345.

[48] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman. (2017) Developer-related factors in change prediction: An empirical assessment - replication package - https://www.mediafire.com/folder/stknd94rilid3/ICPC17.

[49] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[50] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[51] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[52] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Time variance and defect prediction in software projects," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 348–389, 2012.

[53] P. A. Devijver and J. Kittler, *Pattern Recognition: A Statistical Approach*, 1982.