Predicting Bugs Using Antipatterns

Seyyed Ehsan Salamati Taba[1], Foutse Khomh[2], Ying Zou[3], Ahmed E. Hassan[1], and Meiyappan Nagappan[1]
[1] *School of Computing, Queen's University, Canada*
[2] *SWAT, École Polytechnique de Montréal, Québec, Canada*
[3] *Department of Electrical and Computer Engineering, Queen's University, Canada*
taba@cs.queensu.ca, foutse.khomh@polymtl.ca, ying.zou@queensu.ca, ahmed@cs.queensu.ca, mei@cs.queensu.ca

***Abstract*—Bug prediction models are often used to help allocate software quality assurance efforts. Software metrics (*e.g.*, process metrics and product metrics) are at the heart of bug prediction models. However, some of these metrics like churn are not actionable; on the contrary, antipatterns which refer to specific design and implementation styles can tell the developers whether a design choice is "poor" or not. Poor designs can be fixed by refactoring. Therefore in this paper, we explore the use of antipatterns for bug prediction, and strive to improve the accuracy of bug prediction models by proposing various metrics based on antipatterns. An additional feature to our proposed metrics is that they take into account the history of antipatterns in files from their inception into the system. Through a case study on multiple versions of Eclipse and ArgoUML, we observe that (i) files participating in antipatterns have higher bug density than other files; (ii) our proposed antipattern based metrics can provide additional explanatory power over traditional metrics, and (iii) improve the *F-measure* of cross-system bug prediction models by 12.5% in average. Managers and quality assurance personnel can use our proposed metrics to better improve their bug prediction models and better focus testing activities and the allocation of support resources.**

***Keywords*-bug prediction, antipattern, software quality**

I. Introduction

Software systems are pervasive in our society and play a vital role in our daily lives. We depend on software systems for our transportation, communication, finance, and even for our health. Therefore, correct functioning of software systems is essential. However, identifying and fixing errors in software systems is very costly. It is estimated that 80% of the total cost of a software system is spent on fixing bugs [1]. To reduce this cost, many bug prediction models [2], [3], [4] have been proposed by the research community to identify areas in software systems where bugs are likely to occur. The vast majority of these bug prediction models are built using product (*e.g.*, code complexity [5]) and process (*e.g.*, code churn [3]) metrics, most of which are not actionable. For example, Nagappan and Ball [3] have used code churns to predict bugs in software systems. Yet, it is unclear how developers should act on the churn values of a class to reduce the risk of future bugs occurring.

Different from metrics, antipatterns [6] which identify "poor" solutions to recurring design problems can tell developers whether the design of a class is "poor" or not, and how to improve it using refactorings [7]. Antipatterns are usually introduced in software systems by developers lack of knowledge or experience to solve a particular problem. Although antipatterns do not usually prevent a program from functioning, they indicate weaknesses in the design that may increase the risk for bugs in the future. In other words, antipatterns indicate a deeper problem in a software system. Previous work by Khomh et al. [8] have found that classes with antipatterns are more prone to bugs than other classes. Antipatterns can be removed from systems using refactoring. If we can predict bugs using antipatterns information, development teams will be able to use refactorings to reduce the risk for bugs in systems. In this paper, we explore the possibility of predicting bugs using antipatterns and strive to improve the accuracy of state-of-the-art bug prediction models by proposing various metrics based on antipatterns. We use statistical modeling to establish and inspect dependencies between our proposed metrics and bugs counts. Using antipatterns and bug information from multiple versions of two open source software systems of Eclipse[1] and ArgoUML[2], we address the following three research questions:

RQ1) Do antipatterns affect the density of bugs in files?
We find that files with antipatterns tend to have higher bug density than the others.

RQ2) Do the proposed antipattern based metrics provide additional explanatory power over traditional metrics?
We find that our proposed antipattern based metrics (ANA, ACM, and ARL) can provide additional explanatory power over the traditional metrics LOC, PRE and Churn. Among these metrics, ARL shows significant improvement in terms of AIC and D^2.

RQ3) Can we improve traditional bug prediction models with antipatterns information?
We find that ARL can also improve bug prediction models across systems. It has a low collinearity with most process and product metrics from the literature and can improve cross-systems bug prediction models by an average of 12.5% in terms of *F-measure*.

The remainder of this paper is organized as follows. First, we summarize the related literature on antipatterns and bug prediction models in Section II. Next, we describe the experimental setup of our study in Section III and report our findings in Section IV. In Section V, we discuss threats to

[1]http://www.eclipse.org/
[2]http://argouml.tigris.org/

the validity of our work. Section VI concludes our work and outlines avenues for future works.

II. Related Work

In this section, we discuss the related literature on antipatterns and bug prediction models.

A. Antipatterns and Code Smells

The first book on antipatterns in object-oriented development was written in 1995 by Webster [9]. Fowler et al. [7] defined 22 code smells that are bad structures in source code. They mentioned that these smells indicate implementation issues that can be solved using refactoring. Moreover, They claim that code smells have detrimental effects on software. However, little empirical evidence was provided to support this claim.

The literature related to antipatterns and code smells generally fall into two categories. The first one focuses on detecting antipatterns and code smells (*e.g.*, [10]). The second category concentrates on investigating the relation between antipatterns and software quality (*e.g.*, [11]). Our work in this paper has the same aim as these studies (*i.e.*, the improvement of software quality). Li and Shatnawi [11] investigate relationships between 6 code smells and class error probability in three different versions of Eclipse. They report that classes with antipatterns, such as: God Class, God Method and Shotgun Surgery are positively associated with higher error probability. Moreover, Khomh et al. [8] show that there is a relation between antipatterns and the bug-proneness of a file. These studies provide empirical evidences on the relation between antipatterns and bugs. In this paper, we build up on these previous works to investigate the possibility of predicting bugs in software systems using antipattern information.

Olbrich *et al.* [12] study the evolution of two different code smells (*i.e.*, Shotgun surgery and God class) over time in the development process of two software systems. They conclude that the relative number of components having code smells does not decrease over time meaning that not a lot of refactoring activities are performed on the systems. We also observe this behavior (as shown in Figure 2) on our studied systems; the density of antipatterns does not increase significantly overtime. Peters *et al.* [13] studied the lifespan of 5 different code smells over different releases, and the refactoring behaviour of developers in seven open source systems. They conclude that given the low number of refactorings performed by developers, the number of long-living code smell instances increases over time. These studies show that code smells and antipatterns mostly remain in systems. In this study, we investigate the link between the persistent antipatterns and post release bugs in software systems.

B. Bug Prediction Models

Researchers have tried to uncover the possible reasons for software bugs using different classes of software metrics, such as process and product metrics [14], [15] or entropy of changes [16]. However, their primary goal has been established on improving the accuracy of bug prediction (localization) models. Zimmermann *et al.* [14] conducted an empirical study on three different versions of Eclipse to show that a combination of complexity metrics can predict bugs. They conclude that large files (*i.e.*, high LOC values) are more prone to bugs than others. Another case study performed using 85 versions of 12 releases of Apache projects [17] show how and why process metrics are better indicators of bugs with respect to performance, portability and the stability of the model. Moreover, Kamei *et al.* [18] and Chen *et al.* [19] introduce metrics based on the effort and topics in software systems to improve bug prediction models.

Following the same line of work, in this paper, we propose antipatterns as another factor to enhance the accuracy of bug prediction models. More specifically, we propose four new metrics based on the history of antipatterns in files, and perform a case study to verify whether the proposed metrics provide additional explanatory power to bug prediction models built using traditional product and process metrics.

III. Study Design

This section presents the design of our case study, which aims to address the following three research questions:

1) Do antipatterns affect the density of bugs in files?
2) Do the proposed antipattern based metrics provide additional explanatory power over traditional metrics?
3) Can we improve traditional bug prediction models with antipatterns information?

A. Data Collection

Our work studies bug prediction using 12 versions of Eclipse and 9 versions of ArgoUML. Eclipse is a popular IDE used both in open-source communities and in industry. It has an extensive plugin architecture. ArgoUML is an open source UML-based system design tool. These systems encompass different domains and have different sizes. Eclipse is close to the size of real industrial systems (*e.g.*, release 3.3.1 is larger than 3.5 MLOCs), while ArgoUML is a smaller project. Table I shows descriptive statistics of the systems.

B. Data Processing

Figure 1 shows an overview of our data processing steps. First, we mine the source code repositories of Eclipse and ArgoUML to compute product and process metrics. Next, we detect antipatterns in the two software systems. Then, we mine the bug repositories of the systems to extract information about bugs. Finally, we use statistical models to analyze the collected data and answer our three research questions. The remainder of this section elaborates on each of these steps.

1) Mining Source Code Repositories: We download 12 versions of Eclipse and 9 versions of ArgoUML from their respective CVS repositories. We use the Ptidej tool [20] to compute metrics on the source code of each downloaded version. We also use a perl script developed for the purpose of this study to calculate code churn metric values.

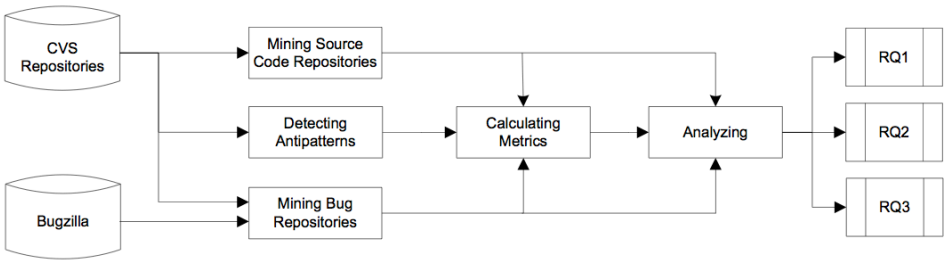

Figure 1. Overview of our data collection process.

Table I
SUMMARY OF THE CHARACTERISTICS OF THE ANALYSED SYSTEMS

Systems	Releases(#)	Total Number of Antipatterns	Churn	Total Number of Post Bugs	Total Number of Pre Bugs	LOCs
Eclipse	2.0 – 3.3.1(12)	273,766	148,454	27,406	23,554	26,209,669
ArgoUML	0.12 – 0.26.2(9)	15,100	21,427	2,549	2,569	2,025,730

2) Detecting Antipatterns: We use the DECOR method proposed by Moha *et al.* [10] to specify and detect antipatterns in our subject systems. DECOR is based on a thorough domain analysis of code smells and antipatterns in the literature, and provides a domain-specific language to specify code smells and antipatterns and methods to detect their occurrences automatically. Moha *et al.* [10] reported that DECOR's antipatterns detection algorithms achieve 100% recall and an average precision greater than 60%. In this study, we focus on the 13 antipatterns described in Table II. We choose only these antipatterns due to the following reasons: (i) they are well-described by Brown *et al.* [6] and Fowler [7]; and (ii) we could find enough of their occurrences in several releases of our subject systems.

Figure 2 shows the density of antipatterns over the different releases of our subject systems. We define density of antipatterns for a version as the total number of antipatterns over the total number of files in that version. As shown in Figure 2, the density of the antipatterns is quite stable during the evolution of the systems. Our premise in this work is that acting on these antipatterns can help reduce the risk for bugs in the systems.

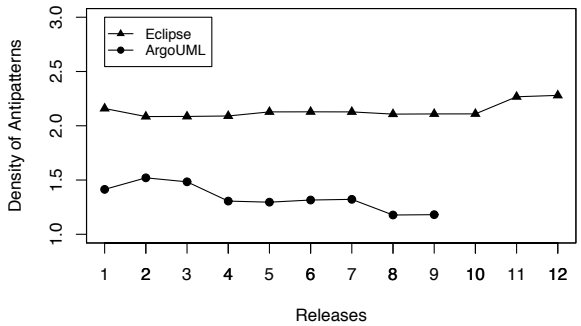

Figure 2. Density of Antipatterns over ArgoUML and Eclipse projects.

3) Mining Bug Repositories: For each version of our studied systems, we extract the change logs of all commits performed after the version is released and download bug reports from the bug tracking system (*i.e.*, Bugzilla). We parse the change logs and apply the heuristics proposed by Fisher *et al.* [21] to identify bug fixes locations. We retain only bugs for which a "bug ID" is found in CVS commits and the Resolution field is set to "FIXED" or the Status field set to "CLOSED". We refer to the CVS commits as bug fixing commits and extract the list of files that are changed to fix the bug.

Table II
ANTIPATTERN DEFINITION

Antipatterns	Description
AntiSingleton	A class that provides mutable class variables, which consequently could be used as global variables.
Blob	A class that is too large and not cohesive enough. It monopolises most of the processing, and takes most of the decisions.
ClassDataShould-BePrivate (CDSBP)	A class that exposes its fields, thus violating the principle of encapsulation.
ComplexClass	A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
LargeClass	A class that has grown too large in term of LOCs.
LazyClass	A class that has few fields and methods.
LongParameter-List (LPL)	A class that has (at least) one method with a too long list of parameters in comparison to the average number of parameters per methods in the system.
LongMethod	A class that has (at least) a method that is very long, in term of LOCs.
MessageChain	A class that uses a long chain of method invocations to realise (at least) one of its functionality.
RefusedParent-Bequest (RPB)	A class that redefines inherited method using empty bodies, thus breaking polymorphism.
SpaghettiCode	A class declaring long methods with no parameters and using global variables.
SwissArmyKnife	A class that has excessive number of method definitions, thus providing many different unrelated functionality.
Speculative-Generality	A class that is defined as abstract but that has very few children, which do not make use of its methods.

4) Analysis Methods: We investigate the possibility of using antipatterns to predict bugs in software systems.

a) Analyzing the relation between the occurrences of antipatterns and the density of future bugs: We use the Wilcoxon rank sum test [22] to compare the density of

future bugs of classes with and without antipatterns. We define density of future bugs in a file as the total number of bugs over the total LOCs in the file. The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions have equally large values. Non-parametric statistical methods do not make assumptions about the distributions of assessed variables.

b) Exploring bug prediction using antipatterns information: As mentioned before, state of the art metrics can be classified into product metrics (*e.g.*, Lines of Code (LOC)[23]) which are static, and process metrics (*e.g.*, Code Churn [3]) which require historical information on a system. To investigate the use of antipatterns in bug prediction models, we propose new metrics that capture antipatterns information in a system. Then, we build logistic regression models to compare each new antipattern based metric to respectively LOC, PRE, Code Churn and the combination of them. We select LOC, PRE and Code Churn as our baseline metrics since previous studies have found them to be good predictors of bugs in software systems [3], [15], [24], [25]. A similar decision is made in studies by Bird *et al.* [26] and Chen *et al.* [19].

We create the models following a hierarchical modelling approach: we start with our baseline metrics and then build subsequent models by adding step by step, our proposed antipatterns metrics (*i.e.*, APMetric). We chose to follow a hierarchical modelling approach because contrary to a step-wise modelling approach, the hierarchical approach has the advantage of minimizing the artificial inflation of errors and therefore the overfitting [27]. For each model, we compute the variance inflation factors (VIF) [28] of each metric to examine multi-collinearity between the variables of the model. We remove all variables with VIF > 2.5.

We report for each statistical model the percentage of deviance explained D^2 [29] and the Akaike information criterion (AIC)[30] of the model. The deviance of a model M is defined as $D(M) = -2 * LL(M)$, where $LL(M)$ is the log-likelihood of the model M. The deviance explained (*i.e.*, D^2) is the ratio between $D(Bugs \sim Intercept)$ and $D(M)$. A higher D^2 value generally indicates a better model fit. AIC is used to compare the fitness of different models. A lower AIC score is better. For each subsequent model $M_{Base+APMetric}$ derived from a model M_{Base}, we also test the statistical significance of the difference between $M_{Base+APMetric}$ and M_{Base}. We report the corresponding p-values.

IV. Study Results

This section presents and discusses the results of our three research questions.

RQ1: Do antipatterns affect the density of bugs in files?

Motivation. Previous work by Khomh et al. [8] have shown that files participating in antipatterns are more likely to have bugs than other files. Moreover, in this research question, we examine the density of bugs in files with antipatterns. We want to know when bugs occur in files with antipatterns, they occur in larger number compared to other files or not.

Approach. We apply DECOR [10] to specify and detect antipatterns in all the versions of our subject systems as described in Section III-B2. For each version, we classify the files in two groups: a group of files with at least one antipattern, and a group of files without antipatterns. For each file from the two groups, we compute the number of post release bugs in the file as described in Section III-B3. Since previous studies (*e.g.*, [14], [15], [24]) have found that the size of code is related to the number of bugs in a file. To control for the confounding effect of size, we divide the number of future bugs of each file by the size of the file. We obtain the density of future bugs for each file. We test the following null hypothesis:

H_{01}^1: *there is no difference between the density of future bugs of the files with antipatterns and the other files without antipatterns*.

Hypothesis H_{01}^1 is two-tailed since it investigates whether antipatterns are related to a higher or a lower density of bugs. We perform a Wilcoxon rank sum test [22] to accept or refute H_{01}^1, using the 5% level (*i.e.*, p-value < 0.05). We also compute and report the difference between the average bug densities in the two groups of files with and without antipatterns (*i.e.*, $D_A - D_{NA}$).

Table III
Wilcoxon rank sum test results for the bug density in files with and without antipatterns

Eclipse			ArgoUML		
Version	$D_A - D_{NA}$%	***p*-value**	**Version**	$D_A - D_{NA}$%	***p*-value**
2.0	-5.78	<0.05	0.12	-10.75	0.58
2.1.1	-4.36	<0.05	0.14	63.26	<0.05
2.1.2	3.43	<0.05	0.16	8.09	<0.05
2.1.3	19.74	<0.05	0.18.1	19.58	<0.05
3.0	11.60	<0.05	0.20	36.78	<0.05
3.0.1	3.01	<0.05	0.22	72.93	<0.05
3.0.2	13.60	<0.05	0.24	-22.33	0.07
3.2	3.98	<0.05	0.26	-18.71	0.71
3.2.1	-1.82	<0.05	0.26.2	50.75	<0.05
3.2.2	4.23	<0.05			
3.3	19.81	<0.05			
3.3.1	-13.22	0.10			

Findings. In general, the density of bugs in a file with antipatterns is higher than the density of bugs in a file without antipatterns. The Wilcoxon rank sum test was statistically significant for 11 out of 12 versions of Eclipse and 6 out of 9 versions of ArgoUML (see Table III). In 8 versions of Eclipse and 6 versions of ArgoUML (highlighted in Table III), the density of bugs in files with antipatterns is significantly higher than the density of bugs in other files.

Overall, we reject H_{01}^1 and conclude that the occurrence of an antipattern in a file is not only related to a higher risk for bugs in the file (as reported by Khomh et al. *[8]), but also to a higher density of bugs.*

RQ2: Do the proposed antipattern based metrics provide additional explanatory power over traditional metrics?

Motivation. Antipatterns presented in Table II are detected in software systems using thresholds defined over source code metrics and other lexical information [8]. Since antipatterns refer to specific design and implementation problems in software systems, they are likely to be better indicators than metrics for developers. Indeed, an antipattern can tell developers whether a design implementation is "poor" or not, by means of thresholds defined over metrics and other lexical information; while without antipatterns knowledge, developers would have to judge by themselves which metric values are problematic. Results of **RQ1** show that antipatterns are related to higher numbers of bugs. By acting on antipatterns (*e.g.*, using refactorings), it may be possible to reduce post release bugs in a system. In this question, we investigate this hypothesis in details. We want to understand which proportion of the deviance (*i.e.*, model fitness) in post release bugs can be explained by antipatterns information.

Approach. We introduce the following four metrics to capture antipatterns information in a software system.

Let's $S^i, i \in \{1 \dots n\}$ be the list of consecutive versions of a system S, *i.e.*, S^1 being the first released version and $S = S^n$. For each file $f \in S$, we denote by f^i the version of f in S^i, *i.e.*, $f^i \in S^i$, $i \in \{1 \dots n\}$ and $f = f^n$.

Let's χ^i be the indicator function defined on S^i by:

$$\chi^i(f^i) = \begin{cases} 1, & \text{if } f^i \text{ contains one or more bugs.} \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Let's $n_{AP}(f^i)$ be the total number of antipatterns in f^i, $i \in \{1 \dots n\}$. To capture the distribution of antipatterns in past buggy versions of a system, we introduce the function N_{AP} defined on $(\bigcup S^i)_{i \in \{1 \dots n\}}$ as follows:

$$N_{AP}(f^i) = \begin{cases} \chi^i(f^i) * n_{AP}(f^i), & \text{if } 1 \leq i < n \\ n_{AP}(f^i) & \text{if } i = n \ . \end{cases} \quad (2)$$

Using N_{AP}, we define the Average Number of Antipatterns (ANA) metric to capture the distribution of antipatterns in previous buggy versions of a file following Equation (3). For each file $f \in S$,

$$ANA(f) = \frac{1}{n} * \sum_{i=1}^{n} N_{AP}(f^i), \quad (3)$$

Where n is the total number of versions in the history of f and $f = f^n$.

To capture the distribution of antipatterns across the files of a specific version $i \in \{1 \dots n\}$, we compute the Shannon entropy [31] of antipatterns in S^i following Equation (4).

$$H^i = -\sum_{k=1}^{m} p(f_k^i) * \log_2 p(f_k^i), \quad (4)$$

Where $p(f_k^i) \geq 0, \forall k \in 1,\dots m$; m is the total number of files in S^i and $p(f_k^i)$ is the probability of having antipatterns in file f_k^i; $p(f_k^i)$ is computed following Equation (5).

$$p(f_k^i) = \frac{n_{AP}(f_k^i)}{\sum_{l=1}^{m} n_{AP}(f_l^i)}, \quad (5)$$

Using the entropy of antipatterns in S^i, we introduce the Antipattern Complexity Metric (ACM) following Equation (6). This metric is similar to the HCM metric proposed by Hassan *et al.* [16] to capture the complexity of source code changes.

$$ACM(f) = \sum_{i=1}^{n} p(f^i) * H^i, \quad (6)$$

Where n is the total number of versions in the history of f and $f = f^n$.

To capture the consecutive occurrence of antipatterns in a file, we introduce the Antipattern Recurrence Length (ARL) metric following Equation (7).

$$ARL(f) = rle(f) * e^{\frac{1}{n} * (c(f) + b(f))}, \quad (7)$$

Where n is total number of versions in the history of f, $c(f)$ is the number of buggy versions in the history of f in which f has at least one antipattern, $b(f) < n$ is the ending index of the longest consecutive stream of antipatterns in buggy versions of f, and $rle(f)$ is the maximum length of the longest consecutive stream of antipatterns in the history of f (see [32] for a detailed definition of rle).

To illustrate this metric let's consider a file f that has 5 previous versions in its history. Let's assume that the distribution of $N_{AP}(f^i)$ among these 6 versions (*i.e.*, 5 previous versions and the current version) is as follows: {3,4,0,2,1,3}. The value of $ARL(f)$ is 18.76. Where the value of $rle(f)$ is 3, n is 6, the number of buggy versions having antipatterns (*i.e.*, $c(f)$) is 5, and the ending index of the longest consecutive stream of antipatterns (*i.e.*, $b(f)$) is 6.

Our last metric is the Antipattern Cumulative Pairwise Differences (ACPD) metric, which aim is to capture the growth tendency of the antipatterns in a file over time. ACPD is computed following Equation (8).

$$ACPD(f) = \sum_{i=1}^{n} [N_{AP}(f^{i-1}) - N_{AP}(f^i)], \quad (8)$$

Where n is the total number of versions in the history of f and $f = f^n$. Positive values of ACPD reflect a decreasing tendency of the number of antipatterns over the history of a file.

Using these four metrics (*i.e.*, ANA, ACM, ARL, and ACPD) we build four sets of logistic regression models for every version of our subject systems, following the method described in Section III-B4.

Findings. Among ANA, ACM, ARL, and ACPD metrics, ARL has the most significant improvement over traditional metrics LOC, PRE, Code Churn. Figure 3 and Figure 4 show the percentage of deviance explained (D^2)

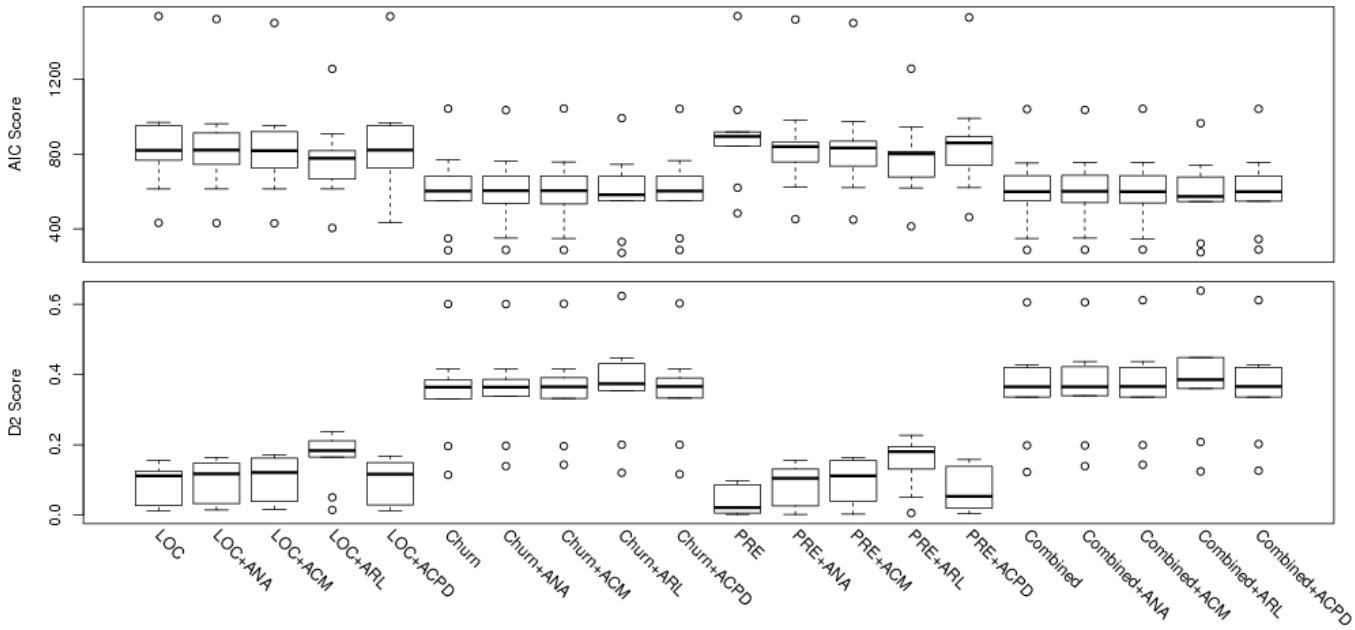

Figure 3. D^2 and AIC scores over 9 versions of ArgoUML by adding proposed metrics to historical software metrics. Each boxplot shows the distribution of D^2 and AIC scores over the different versions of ArgoUML.

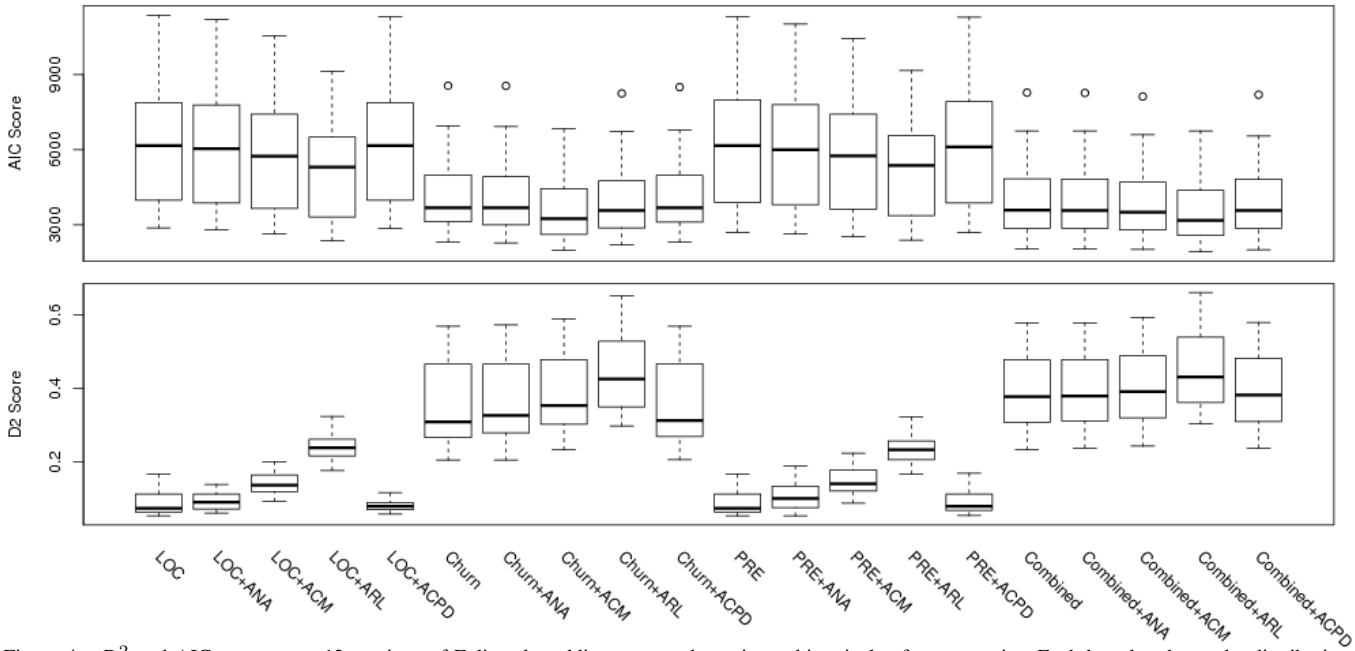

Figure 4. D^2 and AIC scores over 12 versions of Eclipse by adding proposed metrics to historical software metrics. Each boxplot shows the distribution of D^2 and AIC scores over the different versions of Eclipse.

and the Akaike information criterion (AIC) of the model built for each version of ArgoUML and Eclipse from our data set. For each model, a high D^2 score and a low AIC score is desirable. A high D^2 score (respectively a low AIC score) indicates a better model fit. For both ArgoUML and Eclipse versions, we observe that including ANA, ACM, and ARL provides additional explanatory power about the bug-proneness of files over existing traditional product and process metrics (LOC, PRE, Code Churn). The biggest improvement is obtained with the ARL metric, both in terms of AIC and D^2 (*i.e.*, 20% decrease of AIC and 300% increase of D^2). Therefore we answer our research question positively. Since ARL captures the proportion of antipatterns with past bugs in streams of consecutive persistent occurrences of antipatterns, we recommend that development teams take the necessary steps to refactor persistent antipatterns that exhibited bugs in the past. Files with long streams of consecutive occurrences of antipatterns should also be refactored. ACM shows the second biggest improvement of explanatory power after ARL. We also recommend that development teams refactor complex

distributions of antipatterns across files since they are likely to be related to a higher risk for bug. Since we observed no improvement with the metric ACPD, it seems that a temporary increase of the number of antipatterns in a system does not necessarily translates into a high risk for bugs. The persistence of antipatterns and the complexity of their distribution across classes in a software system seem to be the main factors behind the increased risk for future bugs observed in systems with antipatterns.

In conclusion, we found that including the antipattern based metrics ANA, ACM, and ARL provides additional explanatory power about the bug-proneness of files over the following existing traditional product and process metrics LOC, PRE, Code Churn.

Table IV
MEASURED PROCESS AND PRODUCT METRICS

	Metric names	Description
Product metrics	LOC	Source lines of codes
	MLOC	Executable lines of codes
	PAR	Number of parameters
	NOF	Number of attributes
	NOM	Number of methods
	NOC	Number of children
	VG	Cyclomatic complexity
	DIT	Depth of inheritance tree
	LCOM	Lack of cohesion of methods
	NOT	Number of classes
	WMC	Number of weighted methods per class
Process metrics	PRE	Number of pre-released bugs
	Churn	Number of lines of code added modified or deleted

RQ3: Can we improve traditional bug prediction models with antipatterns information?

Motivation. In **RQ2** we observe that including antipattern information provides additional explanatory power to bug prediction models built using the traditional product and process metrics, LOC, PRE, and Code Churn. The proposed antipattern based metrics ANA, ACM, and ARL are able to increase the deviance explained of models by up to 300%. However, in practice, bug prediction models are not built using only LOC, PRE, and Code Churn. They take into account a variety of other metrics, including those presented in Table IV. Hence, it is interesting to further investigate how our proposed metrics perform in comparison to a more larger collection of metrics.

Another important aspect of bug prediction models is the possibility to apply them across systems. This aspect is particularly important because training data is often not available for software systems from small companies or software systems in their first release (*i.e.*, for which no past data exists). In such situations, development teams attempt to predict bugs using models built and trained on systems from other companies. In this research question, we investigate to what extent one can use cross-system antipattern information to predict bugs. More specifically, we examine whether our proposed antipattern based metrics can improve traditional bug prediction models within and across systems.

Approach. To answer this research question we build two different sets of models: intra-system models and cross-system models.

A. Intra-system Models

Intra-system models are built using different versions of the same system. Logistic regression models are generally used for this purpose. In our case, the independent variables are our proposed metrics and a collection of code and process metrics that have been used in previous studies from the literature [33], [15], [14]. Table IV describes the metrics. The dependent variable of our models is a two-value variable that represents whether or not a file has one or more bugs. We broke the process of our analysis into two parts following the approach from Shihab et al. [33]. In the first part, we perform a step-wise analysis to remove statistically insignificant independent variables for each version. This process is repeated until we reach to a model that contains only statistically significant independent variables. Second, we remove highly collinear independent variables from the logistic regression model by controlling the levels of variance inflation. At the end, the remaining independent variables in the model will be statistically significant and minimally collinear. The following sections elaborate more on these steps.

1) Removing Independent Variables: In this part we build a multivariate logistic regression model based on our dependent and independent variables, and then in a iterative process we remove the independent variables that are statistically insignificant. We do this process until all the variables are statistically significant. To this end, we use the threshold p-value < 0.1 to determine whether an independent variable is statistically significant or not.

2) Collinearity Analysis: Multicollinearity exists whenever two or more independent variables in a regression model are moderately or highly correlated. Multicollinearity does not reduce the predictive power or reliability of the model as a whole, at least within the sample data themselves, but the problem with multicollinearity is that as the independent variables become highly correlated, it becomes more difficult to determine which independent variable is actually producing the effect on the dependent variable. Tolerance and Variance Inflation Factor (*VIF*) is often used to measure the level of multicollinearity of models. A variance inflation factor (*VIF*) quantifies how much the variance is inflated. A tolerance value close to 1 means that there is no correlation among the independent variables of the model, and hence the variance of our model is not inflated at all. In this paper, we set the maximum *VIF* value to be 2.5, as suggested in [27]. A *VIF* value that exceeds 2.5 requires further investigation, while *VIF*s exceeding 10 are signs of serious multicollinearity that require correction [34]. In this work we use the *VIF* command in the *car* package of R toolkit [35] to examine the *VIF* values of all independent variables used to build the multivariate logistic regression model.

Table V
P-VALUES OF STATISTICALLY SIGNIFICANT AND MINIMALLY COLLINEAR INDEPENDENT VARIABLES FOR ARGOUML MODELS.

	0.12.0	**0.14.0**	**0.16.0**	**0.18.1**	**0.20.0**	**0.22.0**	**0.24.0**	**0.26.0**	**0.26.2**
Churn	2.49e-13***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***
PRE		0.00352**		0.01439*		0.01418*	0.012635*		
LOC					9.17e-07***			0.00553**	
MLOC	0.0596.					7.58e-08***	0.014052*	0.02727*	
NOT							0.068654.		0.0087**
NOF	0.0230*						0.048091*		
NOM						0.00840**			0.0234*
ACM			7.26e-12***						6.77e-06***
ACPD			0.000768***			0.00122**			
ARL		4.68e-05***	<2e-16***	6.6e-05***	3.03e-07***	5.74e-15***	0.000577***		1.63e-10***
AIC	528.73	567.95	911.04	577.12	721.53	546.99	669.29	251.64	288.73
D^2	0.16	0.42	0.42	0.39	0.37	0.47	0.22	0.50	0.68

Table VI
P-VALUES OF STATISTICALLY SIGNIFICANT AND MINIMALLY COLLINEAR INDEPENDENT VARIABLES FOR ECLIPSE MODELS.

	2.0	**2.1.1**	**2.1.2**	**2.1.3**	**3.0**	**3.0.1**	**3.0.2**	**3.2**	**3.2.1**	**3.2.2**	**3.3**	**3.3.1**
Churn	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***
PRE	<2e-16***	1.58e-05***	0.000108***		1.17e-09***	2.26e-06***			1.12e-12***		0.00036***	6.65e-07***
LOC									0.00081***			
MLOC								6.46e-11***	2.63e-05***			
NOT	7.25e-05***								0.094773.	0.003649**	1.08e-05***	0.01944*
NOF	0.00054***	0.00980**		0.029776*	0.009207**		0.0417*				0.003904**	
NOM	0.032206*		0.030065*	0.001938**		0.00320**	0.0422*	0.0275*	0.014777*			0.04749*
ACM								8.99e-06***	6.05e-08***			
ACPD						0.00139**		1.42e-06***	6.18e-15***	<2e-16***	0.011604*	
ARL	<2e-16***			<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***	<2e-16***		
AIC	3235.8	2111.7	1788.7	2332	2752	4321	6169	2797	4110	3445	2561	5020
D^2	0.35	0.40	0.45	0.46	0.38	0.36	0.47	0.56	0.54	0.57	0.68	0.58

We narrow down our list of independent variables to consider only those that are statistically significant and minimally collinear with each other (*i.e.*, *VIF* = 2.5). We use these variables to build the final logistic regression model.

Findings. Among our proposed metrics, ARL remained statistically significant and had a low collinearity with other metrics in almost all the versions of two studied systems. Table V and Table VI present the results for different versions of ArgoUML and Eclipse. Each column, represents a version of the studied systems. For each version, we report the p-value of the metrics that remained significant after the first aforementioned iterative process and that have a low collinearity with other independent variables (*i.e.*, *VIF* < 2.5). As one can see, ARL is statistically significant and have a low collinearity with other independent variables for 7 out of 9 versions of ArgoUML, and for 8 out of 12 versions of Eclipse. However, the frequency of occurrence of the other three antipattern based metrics ANA, ACM and ACPD is not considerable. This result shows that ARL captures a different aspect of bug-proneness than the metrics from Table IV. Therefore, it is helpful to include ARL in a model for predicting bugs. Among the metrics from Table IV, *Churn* and *PRE* also have a high impact in predicting bugs. The product metrics NOF and NOM also contribute significantly but their occurrence in the different models is not persistent over different versions and systems. In conclusion, we recommend that software development teams make use of ARL to improve their bug-prediction models.

B. Cross-system Models

Cross-system bug prediction is defined as the process of building a model from one system and applying that model to another system in order to successfully predict bugs [4]. To investigate the extent to which cross-system antipattern information can be used to predict bugs, we analyze cross-system bug prediction models built on 12 different versions of Eclipse and 9 versions of ArgoUML, using our antipattern based metrics and metrics from Table IV. We built models for all possible combinations across the systems. Each model was trained on one version of a system and tested on one version of the other system and vice versa. In total we obtained 216 different models. For each pair, we built a logistic regression model using the process and product metrics from Table IV as independent variables, and a two value variable which indicates whether a file has at least one bug or not, as our dependent variable. We calculate the *F-measure* of the model by training the model on its corresponding training data (*e.g.*, Eclispe 2.0) and testing it on its corresponding testing data (*e.g.*, ArgoUML 0.12). Next, we add the ARL metric to the previous independent variables and compute the *F-measure* of the new model using the same training and testing data. This enables us to measure the potential benefit of ARL for cross-system prediction. The *F-measure* is the harmonic mean of *precision* and *recall*. The *precision* of a model is the proportion of bugs that are predicted correctly by the model, while the *recall* is the proportion of real bugs that are predicted successfully by the model.

Findings. ARL can improve cross-system bug prediction on the two studied systems by an average of 12.5% in terms of *F-measure*. Figure 5 shows distributions of *F-measure* for the 108 x 2 = 216 models that were built. On this figure, AE (respectively EA) means that the models were trained on ArgoUML (respectively Eclipse) and tested on Eclipse (respectively ArgoUML). AE_base (respectively AE_base + ARL) refers to models built using the metrics from Table IV (respectively the metrics from Table IV and ARL).

The average improvement observed on the *F-measure* of models trained on ArgoUML (respectively Eclipse) and tested on Eclipse (respectively ArgoUML), when ARL was added to the models is 10.71% (respectively 12.5%). This result reinforce our previous finding that ARL captures a different aspect of bug-proneness than the metrics from Table IV. Additionally, this result show that the information captured by ARL is transferable across systems.

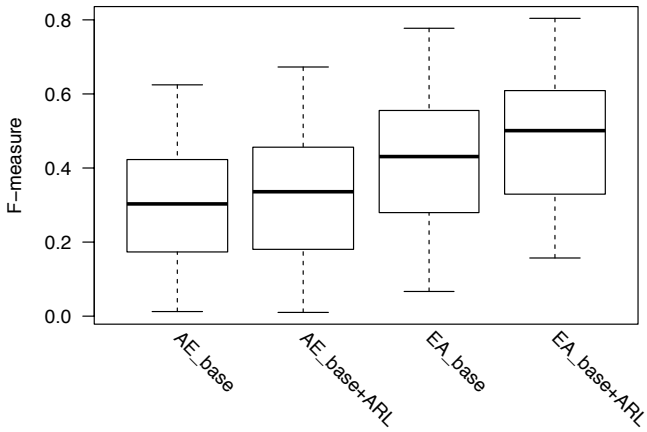

Figure 5. Performance comparison of the models for cross-project prediction when adding ARL to the traditional metrics (Table IV).

In summary, we observed that our proposed antipattern based metric (ARL) can improve bug prediction models both within and across systems. ARL has a low collinearity with most process and product metrics from the literature and can improve cross-systems bug prediction models by an average of 12.5% in terms of F-measure.

V. Threats to Validity

We now discuss the threats to validity of our study following common guidelines for empirical studies [36].

Construct validity threats concern the relation between theory and observation. In this study, they are mainly due to measurement errors. For ArgoUML, issues dealing with fixing bugs are marked as "DEFECT" in the issue tracking system. For Eclipse, we mitigated the use of possibly erroneous bugs by discarding issues explicitly labeled as "Enhancements" and focusing on issues marked as "FIXED" or "CLOSED" because they required some changes. It is unlikely, in Eclipse, that hard-to-fix issues would stay longer "OPENED" than others, because Eclipse is being backed up by IBM, which strives to offer a stable product. To identify bug fix locations, we mine CVS logs and apply the heuristics by Fisher *et al.* [21]. Although this technique may not be a hundred percent accurate, it has been used satisfactorily in many previous studies, *e.g.*, [8], [21]. For the sake of simplicity, we assumed to have one class per file. This assumption could introduce an error in case of non-public top-level classes and inner classes. We did not find any inner class participating in any antipattern in the analysed versions of the systems. Non-public top-level classes are rare and did not participate in any antipattern.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. The accuracy of DECOR impacts our results since the number of antipatterns computed with DECOR is used to compute our proposed metrics. Other antipattern detection techniques and tools should be used to confirm our findings.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models; in particular we used non-parametric tests that do not require any assumption on the underlying data distribution.

Reliability validity threats concern the possibility of replicating this study. Every result obtained through empirical studies is threatened by potential bias from data sets [37]. To mitigate these threats we tested our hypotheses over 12 versions of Eclipse and 9 versions of ArgoUML. Two systems from different size and from different domains. Also, we attempt to provide all the necessary details to replicate our study. The source code repositories and issue-tracking systems of Eclipse and ArgoUML are publicly available to obtain the same data.

Threats to external validity concern the possibility to generalize our results. We have studied multiple versions of two systems having different sizes and belonging to different domains. Nevertheless, further validation on a larger set of software systems is desirable, considering systems from different domains, as well as several systems from the same domain. In this study, we used a particular yet representative subset of antipatterns. Future work using different antipatterns are desirable.

VI. Conclusion

In this paper, we provided empirical evidence that antipatterns can help predict bugs. To begin with, we show that a file that has antipatterns tends to have a higher density of bugs than other files. Then, we proposed four metrics based on the history of antipatterns in a file to capture antipatterns information in software systems.

We performed a detailed case study using two large real-world software systems (*i.e.*, Eclipse and ArgoUML), to investigate the possibility to predict bugs using the four proposed metrics. The highlights of our analysis include:

- Files that have antipatterns tend to have higher density of bugs than the others (**RQ1**).
- Our proposed metrics can provide additional explanatory power over traditional metrics such as LOC, PRE, Churn.

Among the four proposed metrics, ARL shows the biggest improvement both in terms of AIC and D^2, *i.e.*, 20% decrease of AIC and 300% increase of D^2 (**RQ2**).

- ARL can also improve bug prediction models both within and across systems. It has a low collinearity with most process and product metrics from the literature and can improve cross-systems bug prediction models by an average of 12.5% in terms of *F-measure* (**RQ3**).

In future work, we plan to replicate this study on other systems than Eclipse and ArgoUML to assess the generality of our results. We also plan to investigate more metrics based on antipatterns and clone genealogies.

REFERENCES

[1] N. I. of Standards & Technology, "The economic impacts of inadequate infrastructure for software testing," May 2002, uS Dept of Commerce.

[2] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software defect association mining and defect correction effort prediction," *IEEE Trans. Softw. Eng.*, vol. 32, no. 2, pp. 69–82, Feb. 2006.

[3] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292.

[4] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09, 2009, pp. 91–100.

[5] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec.

[6] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998. [Online]. Available: www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase_theanti patterngr/103-4749445-6141457

[7] *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[8] F. Khomh, M. D. 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, Jun. 2012.

[9] B. F. Webster, "Pitfalls of object-oriented development." 1995, pp. I–X, 1–256.

[10] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[11] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. Syst. Softw.*, vol. 80, no. 7, pp. 1120–1128, Jul. 2007.

[12] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09, 2009, pp. 390–400.

[13] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, ser. CSMR '12, 2012, pp. 411–416.

[14] 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, 2007, pp. 9–.

[15] M. DAmbros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proceeding of the 7th Conference on Mining Software Repositories*, 2010, pp. 31–41.

[16] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 78–88.

[17] F. Rahman and P. Devanbu., "How, and why, process metrics are better."

[18] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10, 2010, pp. 1–10.

[19] T.-H. Chen, S. Thomas, M. Nagappan, and A. Hassan, "Explaining software defects using topic models," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, june 2012, pp. 189 –198.

[20] Y.-G. Gueheneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *Software Engineering, IEEE Transactions on*, vol. 34, no. 5, pp. 667–684, Sept.-Oct.

[21] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, Sept., pp. 23–32.

[22] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007.

[23] S. G. Crawford, A. A. McIntosh, and D. Pregibon, "An analysis of static metrics and faults in c software," *J. Syst. Softw.*, vol. 5, no. 1, pp. 37–48, Feb. 1985. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(85)90005-6

[24] J. Rosenberg, "Some misconceptions about lines of code," in *Software Metrics Symposium, 1997. Proceedings., Fourth International*, Nov, pp. 137–142.

[25] S. Biyani and P. Santhanam, "Exploring defect data from development and customer usage on software modules over multiple releases," in *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, Nov, pp. 316–320.

[26] 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. New York, NY, USA: ACM, 2011, pp. 4–14. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025119

[27] M. Cataldo, A. Mockus, J. Roberts, and J. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *Software Engineering, IEEE Transactions on*, vol. 35, no. 6, pp. 864–878, Nov.-Dec.

[28] M. Kutner, C. Nachtsheim, and J. Neter, *Applied Linear Regression Models*. 4^{th} International Edition, McGraw-Hill/Irwin., September 2004.

[29] J. Nelder and R. Wedderburn, "Generalized linear models," *Journal of the Royal Statistical Society. Series A (General)*, vol. 135, no. 3, p. 370384, 1972.

[30] H. Akaike, "A new look at the statistical model identification," *Automatic Control, IEEE Transactions on*, vol. 19, no. 6, pp. 716–723, 1974.

[31] C. E. Shannon, "A mathematical theory of communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, no. 1, pp. 3–55, Jan. 2001. [Online]. Available: http://doi.acm.org/10.1145/584091.584093

[32] G. Held and T. R. Marshall, *Data compression: techniques and applications, hardware and software considerations (2nd ed.)*. New York, NY, USA: John Wiley & Sons, Inc., 1987.

[33] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan, "Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10, 2010, pp. 4:1–4:10.

[34] N. Bettenburg and A. E. Hassan, "Studying the impact of social structures on software quality," in *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ser. ICPC '10, 2010, pp. 124–133.

[35] "R toolkit," 19-Dec-2012. [Online]. Available: http://www.r-project.org

[36] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[37] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.