2011 27th IEEE International Conference on Software Maintenance (ICSM)

Using Source Code Metrics to Predict Change-Prone Java Interfaces

Daniele Romano
Software Engineering Research Group
Delft University of Technology
The Netherlands
Email: daniele.romano@tudelft.nl

Martin Pinzger
Software Engineering Research Group
Delft University of Technology
The Netherlands
Email: m.pinzger@tudelft.nl

***Abstract*—Recent empirical studies have investigated the use of source code metrics to predict the change- and defect-proneness of source code files and classes. While results showed strong correlations and good predictive power of these metrics, they do not distinguish between interface, abstract or concrete classes. In particular, interfaces declare contracts that are meant to remain stable during the evolution of a software system while the implementation in concrete classes is more likely to change.**

This paper aims at investigating to which extent the existing source code metrics can be used for predicting change-prone Java interfaces. We empirically investigate the correlation between metrics and the number of fine-grained source code changes in interfaces of ten Java open-source systems. Then, we evaluate the metrics to calculate models for predicting change-prone Java interfaces. Our results show that the external interface cohesion metric exhibits the strongest correlation with the number of source code changes. This metric also improves the performance of prediction models to classify Java interfaces into change-prone and not change-prone.

I. Introduction

Software systems are continuously subjected to changes. Those changes are necessary to add new features, to adapt to a new environment, to fix bugs or to refactor the source code. However, the maintenance of software systems is also risky and costly. For instance, Brooks states that software maintenance accounts for 90% of the total costs of a software system [1].

Several approaches have been developed to optimize the maintenance activities and reduce the costs. They range from automated reverse engineering techniques to ease program comprehension to prediction models that can help identifying the change- and defect-prone parts in the source code. Developers should focus on understanding these change- and defect prone parts in order to take appropriate counter measures to minimize the number of future changes [2].

Many of these prediction models have been developed using source code metrics, such as by Briand *et al.* [3], Subramanyam *et al.* [4], and Menzies *et al.* [5]. While those prediction models showed good performance, they work on file and class level. None of them takes the kind of class into account, whether it is a concrete class, abstract class, or interface that is change- or defect-prone. We believe that changes in interfaces can have a stronger impact than changes in concrete and abstract classes, and should therefore be treated separately. Interfaces are meant to represent contracts among modules and logic units in a software system. For this reason, they are supposed to be more stable to avoid contract violations and to reduce the effort to maintain a software system.

In this paper, we focus on *Java interfaces* and investigate the predictive power of various source code metrics to classify Java interfaces into *change-prone* and *not change-prone*. Concerning the source code metrics, we take into account (1) the set of metrics defined by Chidamber and Kemerer [6]; (2) a set of metrics to measure the complexity and the usage of interfaces; and (3) two metrics to measure the external cohesion of Java interfaces. The number of fine-grained source code changes (*#SCC*), as introduced by Fluri *et al.* [7], is used to distinguish between change-prone and not change-prone interfaces.

We selected the Chidamber and Kemerer (C&K) metrics suite because it is widely used and it has been validated by several approaches, such as [8], [9], [10]. The two external cohesion metrics are *Interface Usage Cohesion (IUC)* and a clustering metric. These metrics are meant as heuristics to indicate violations of the Interface Segregation Principle (*ISP*) as described by Martin [11]. We believe that the violation of the *ISP* can impact the maintenance of interfaces and the software system as a whole. The complexity and usage metrics for interfaces have been added to provide a broader set of interface metrics for our study.

To investigate our claim, we perform an empirical study with the source code and versioning data of ten Java open source systems, namely: eight plug-in projects from the Eclipse platform, Hibernate2 and Hibernate3. In the study, we address the following two research hypotheses:

- **H1:** ***IUC*** **has a stronger correlation with the** ***#SCC*** **of interfaces than the C&K metrics**
- **H2:** ***IUC*** **can improve the performance of prediction models to classify Java interfaces into** ***change-*** **and** ***not change-prone***

The results show that most of the C&K perform well for predicting change-prone concrete and abstract classes but are limited in predicting change-prone Java interfaces, therefore confirming our claim that interfaces need to be treated separately. The *IUC* metric exhibits the strongest correlation with

978-1-4577-0664-6/11/$26.00 ©2011 IEEE

#SCC of Java interfaces and proves to be an adequate metric to compute prediction models for classifying Java interfaces.

The remainder of this paper is organized as follows. Section II discusses the C&K metrics and their effectiveness when used for measuring the size and complexity of interfaces. We furthermore introduce the *IUC* metric and several other interface complexity and usage metrics. Section III describes the approach used to measure the metrics and to mine the fine-grained source code changes from versioning repositories. The empirical study and results are presented in Section IV. Section V discusses the results and threats to validity. Related work is presented in Section VI. We draw our conclusions and outline directions for future work in Section VII.

II. Interface Metrics

In this section, we present the set of source code metrics used in our empirical study. We furthermore discuss their applicability to measure the size, complexity, and cohesion of Java interfaces. We then present the *IUC* metric and motivate its application to predict change-prone interfaces. At the end of the section, we list additional metrics to measure the complexity and the usage of interfaces. Those metrics are meant to provide further validation of the predictive power of the *IUC* metric.

A. Object-Oriented Metrics & Interfaces

Among the existing product metrics [12], we focus on the object-oriented metrics introduced by Chidamber and Kemerer [6]. They have been widely used as quality indicators of object-oriented software systems. These metrics are:

- *Coupling Between Objects (CBO)*
- *Lack of Cohesion Of Methods (LCOM)*
- *Number Of Children (NOC)*
- *Depth of Inheritance Tree (DIT)*
- *Response For Classes (RFC)*
- *Weighted Methods per Class (WMC)*

We selected the C&K metrics mainly because prior work demonstrated their usefulness for building models for change prediction, *e.g.*, [9] [13], as well as defect prediction, *e.g.*, [10]. In the following, we briefly describe each metric and discuss its application to interfaces.

1) Coupling Between Objects (CBO): The *CBO* metric represents the number of data types a class is coupled with. More specifically, it counts the unique number of reference types that occur through method calls, method parameters, return types, exceptions, and field accesses. If applied to interfaces, this metric is limited to method parameters, return types and exceptions leaving out method calls and field accesses.

2) Lack of Cohesion Of Methods (LCOM): The *LCOM* metric counts the number of pairwise methods without any shared instance variable, minus the number of pairwise methods that share at least one instance variable. More precisely, *LCOM* is defined as:

$$LCOM = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu\left(A_j\right)\right) - m}{1 - m}$$

where a represents the number of attributes of a class, m the number of methods, and $\mu(A_j)$ the number of methods which access each attribute A_j of a class. Perfect cohesion is defined as all methods accessing all variables, in that case the value of *LCOM* is 0. In contrast, if all methods do not share any instance variable, the value of *LCOM* is 1.

The *LCOM* metric is not applicable to interfaces since interfaces do not contain logic and consequently attribute accesses. For instance, the commercial metric tool Understand[1] outputs either 0 or 1 as values for *LCOM* for an interface. The value 1 denotes that the interface also contains the definition of constant attributes, otherwise the value for *LCOM* is 0. This limits the use of *LCOM* for computing prediction models.

3) Weighted Methods per Class (WMC): *WMC* is the sum of the cyclomatic complexities of all methods declared by a class. Formally, the metric is defined as:

$$WMC = \sum_{i=1}^{n} c_i$$

where c_i is the cyclomatic complexity of the ith method of a class. In case of Understand, this metric corresponds to the Number Of Methods (*NOM*), since the complexity of each method declared in an interface is 1. In case of the Metrics tool[2] this metric is always 0 for interfaces. This limits the predictive power of this metric for predicting change-prone interfaces.

4) Number Of Children (NOC): The *NOC* metric counts the number of directly derived classes of a class or interface. Even though this metric is sound for interfaces, we argue that its application for predicting change prone interfaces is limited. The main reason being that interfaces inherit only the type definition (*i.e.*, sub-typing) while abstract classes and concrete classes also inherit the business logic.

5) Depth of Inheritance Tree (DIT): The *DIT* metric denotes the length of the longest path from a sub-class to its base class in an inheritance structure. The idea behind the usage of *DIT* as change-proneness indicator is that classes contained in a deep inheritance structure are more likely to change (*e.g.*, changes in a super-class cause changes in its sub-classes). Similar to *NOC*, we believe that this metric is more useful for abstract and concrete classes than for interfaces.

6) Response For Classes (RFC): The *RFC* metric counts the number of local methods (including inherited methods) of a class. This metric remains valid for interfaces, but it is close to the *WMC* metric since the only added information is the count of the inherited method.

In summary, while most of the C&K metrics are adequate metrics for abstract and concrete classes they are not as powerful for interfaces. Moreover, these metrics fall short in expressing the cohesion of interfaces, therefore we introduce the two external cohesion metrics as presented in the following section.

[1] http://www.scitools.com/

[2] http://metrics.sourceforge.net/

B. External Cohesion Metrics of Interfaces

The main problem a developer can face in designing interfaces is coping with *fat* interfaces. This problem has been formalized in the *Interface Segregation Principle* (*ISP*) described by Martin in [11]. The *ISP* principle states that *fat* interfaces need to be split into smaller interfaces according to the clients of an interface. Any client should only know about the set of methods provided by an interface that are used by the client. In literature the lack of conformance to the *ISP* principle is mainly associated to a higher risk for clients to change when an interface is changed. To the best of our knowledge there exists no empirical evidence that underlines this association.

In order to measure the violation of the *ISP* principle, we use two cohesion metrics: the external cohesion metric for services called *Service Interface Usage Cohesion (SIUC)* taken from Perepletchikov *et al.* [14], [15] and a clustering metric.

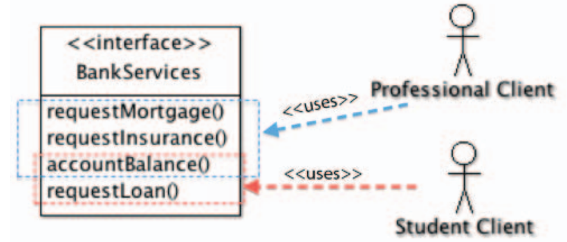
In the following, we refer to the SIUC metric as *Interface Usage Cohesion (IUC)* because we apply it in the context of object-oriented systems. The metric is defined as:

$$IUC(i) = \frac{\sum_{j=1}^{n} \frac{used_methods(j,i)}{num_methods(i)}}{n}$$

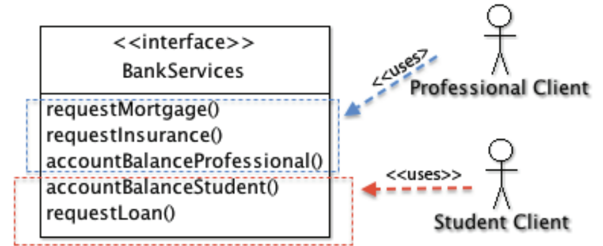
where *j* denotes a client of the interface *i*; *used_methods (j,i)* is the function which computes the number of methods defined in *i* and used by the client *j*; *num _methods(i)* returns the total number of methods defined in *i*; and *n* denotes the number of clients of the interface *i*. The external cohesion defined by Perepletchikov *et al.*, and hence the *IUC* metric, states that there is a strong external cohesion if every client uses all methods of an interface. We argue that interfaces with strong external cohesion (the value of *IUC* is close to one) are less likely to change. On the other hand, when there is a high lack of external cohesion (the value of *IUC* is close to zero) the interface is more likely to change due to the larger number of clients.

Consider the example in Figure 1a that shows an interface for providing bank services. The service is used by two different clients, namely the *Professional Client* and the *Student Client*. The two clients share only one interface method, namely the method *accountBalance()*. Since this method is shared by two different clients, it is more likely to change to satisfy the requirements of the different clients. The design of the *BankServices* interface does not conform to *ISP*. The value of IUC for this interface is $\frac{(3/4+2/4)}{2} = 5/8$.

Consider another example depicted in Figure 1b. It shows the same interface, except the shared method *accountBalance()* has been split into two different methods to serve the two different clients. The design of the interface still violates the *ISP* and changes in the clients can lead to changes in the interface. In fact, the clients depend upon methods that are not used, and the implementing classes implement methods that are not needed. The IUC of this interface is $\frac{3/5+2/5}{2} = 1/2$ which denotes a lower cohesion compared to the interface in Figure 1a. The lower cohesion is mainly due to the higher number of methods, namely 5.



(a) Different clients share a method



(b) Different clients do not share any methods

Fig. 1: An example of lack of external cohesion

Another heuristic to measure the external cohesion is the *ClusterClients(i)* metric. This metric counts the number of clients of an interface *i* that do not share any method. Higher values for this metric indicate lower cohesion. For the interface in Figure 1a the value of *ClusterClients* is 1 and for the interface in Figure 1b the value is 2. We use this metric to investigate whether the contribution of the shared methods, as computed by the *IUC* metric, is relevant to predict change-prone interfaces.

C. Complexity and Usage Metrics for Interfaces

In addition to the object-oriented metrics we validate the *IUC* metric against several other metrics defined to measure the complexity and usage of an interface. The complexity metrics are:

- *NOM(i)*: counts the number of methods declared in the interface *i*;
- *Arguments(i)*: counts the total number of arguments of the declared methods in the interface *i*;
- *APP(i)*: measures the mean size of method declarations of an interface *i* and is equal to *Arguments(i)* divided by *NOM(i)*, as defined by Boxall *et al.* [16];

The usage metrics are:

- *Clients(i)*: counts the number of distinct classes that invoke the interface *i*;
- *Invocations(i)*: counts the number of static invocations of the methods declared in the interface *i*;
- *Implementing_Classes(i)*: counts the number of direct classes that implement the interface *i*.

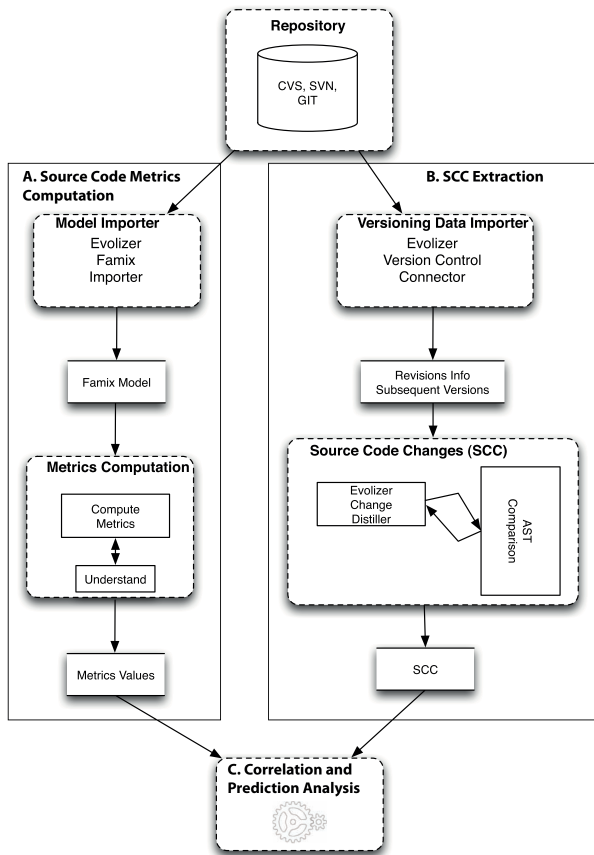


Fig. 2: Overview of the data extraction and measurement process

III. The Approach

In this section, we illustrate the approach used to extract the fine-grained source code changes, to measure the metrics and to perform the experiments aimed at addressing our research hypotheses. Figure 2 shows an overview of our approach that consists of three stages: (A) in the first stage we checkout the source code of the projects from their versioning repositories and we measure the source code metrics; (B) we then compute the number of *SCC* from the versioning data for each class and interface; (C) finally we use the metrics and the number of *SCC* to perform our experiments with the PASWStatistics[3] and RapidMiner[4] toolkits.

A. Source Code Metrics Computation

The first step of the process consists of checking out the source code of each project from the versioning repositories. The source code of each project then is parsed with the Evolizer Famix Importer, belonging to the Evolizer[5] tool set. The parser extracts a FAMIX model that represents the source code entities and their relationships [17]. Figure 3 shows the core of the FAMIX meta model. The model represents inheritance relationships among classes, the methods belonging to a class, the attribute accessed by a method and the invocations among methods. For more details we refer the reader to [17].

After obtaining the FAMIX model, the next step consists of measuring the source code metrics of classes and interfaces. We use the Understand tool to measure the C&K metrics. We decided to use the Understand tool because in our view it provides the most precise measurement of these metrics for interfaces. We use the FAMIX model to measure the external cohesion, complexity and usage metrics of interfaces. For example, to measure the *Invocations(i)* metric we count the number of *invocation* objects in the FAMIX model that point to a method of the interface *i*.

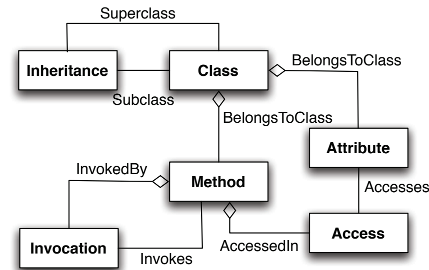


Fig. 3: Core of the FAMIX meta model [17]

B. SCC Extraction

The first step of the *SCC* extraction stage consists of retrieving the versioning data from the repositories (*e.g.*, CVS, SVN, or GIT) for which we use the Evolizer Version Control Connector [18]. The versioning repositories provide log entries that contain information about revisions of files that belong to the system under analysis. For each log entry, it extracts the revision number, the revision timestamp, the name of the developer who checked-in the revision, the commit message, the total number of lines modified (*LM*), and the source code.

In the second step, we use ChangeDistiller [19] to extract the *fine-grained source code changes (SCC)* from the various source code revisions of each file. ChangeDistiller implements a tree differencing algorithm, that compares the Abstract Syntax Trees (*ASTs*) between all direct subsequent revisions of a file. Each change represents a tree edit operation that is required to transform one version of the *AST* into the other. In this way we can track fine-grained source changes down to the statement level. Based on this information we count the number of fine-grained source code changes (*#SCC*) for each class and interface over the selected observation period.

C. Correlation and Prediction Analysis

We use the collection of metric values and *#SCC* of each class and interface as input to our experiments. First, we use the PASWStatistics tool to perform a correlation analysis between the source code metrics and the *#SCC*. Then, we use the RapidMiner tool to analyze the predictive power of

[3]http://www.spss.com/software/statistics/
[4]http://rapid-i.com/content/view/181/196/
[5]http://www.evolizer.org/

TABLE I: Dataset used in the empirical study

Project	#Files	#Interfaces	#Rev	#SCC	Time[M,Y]
Hibernate3	970	165(17%)	30774	34960	Jun04-Mar11
Hibernate2	494	69(14%)	13584	22960	Jan03-Mar11
eclipse.debug.core	188	97(52%)	8295	11670	May01-Mar11
eclipse.debug.ui	793	129(16%)	41860	55259	May01-Mar11
eclipse.jface	381	105(28%)	22136	27041	Sep02-Mar11
eclipse.jdt.debug	469	140(30%)	11711	33895	Jun01-Mar11
eclipse.team.core	172	44(26%)	3726	4551	Nov01-Mar11
eclipse.team.cvs.core	189	25(13%)	12343	23311	Nov01-Mar11
eclipse.team.ui	293	45(15%)	20183	32267	Nov01- Mar11
eclipse.update.core	274	71(26%)	7425	25617	Oct01-Mar11

the source code metrics to discriminate between *change-* and *not change-prone* interfaces. We perform a series of classification experiments with different machine learning algorithms, namely: *Support Vector Machine*, *Naive Bayes Network* and *Neural Nets*. The next section details the empirical study.

IV. Empirical Study

The *goal* of this empirical study is to evaluate the possibility of using the *IUC* metric for predicting the change-prone interfaces and to highlight the limited predictive power of the C&K metrics. The *perspective* is that of a researcher, interested in investigating whether the traditional object-oriented metrics are useful to predict change-prone interfaces. The results of our study are also interesting for quality engineers who want to monitor the quality of their software systems, using an external cohesion metric for interfaces.

The *context* of this study consists of ten open-source systems, widely used in both, the academic and industrial community. These systems are eight plugins from the *Eclipse*[6] platform and the *Hibernate2* and *Hibernate3* systems.[7] *Eclipse* is a popular open source system that has been studied extensively by the research community (*e.g.*, [20], [21], [22], and [23]). *Hibernate* is an object-relational mapping (ORM) library for the Java language.

Table I shows an overview of the dataset used in our empirical study. The *#Files* is the number of unique Java files, *#Interfaces* is the number of unique Java interfaces, *#Rev* is the total number of Java file revisions, *#SCC* is the number of fine-grained source code changes performed within the given time period (*Time*).

In this study, we address the following two research hypotheses:

- **H1:** ***IUC*** **has a stronger correlation with the** ***#SCC*** **of interfaces than the C&K metrics**
- **H2:** ***IUC*** **can improve the performance of prediction models to classify Java interfaces into** ***change-*** **and** ***not change-prone***

We first perform an initial analysis of the extracted information, in terms of number of changes and in terms of metric values. Figure 4 shows the box plots of the *#SCC* of Java classes and interfaces mined from the versioning repositories of each project.

[6]http://www.eclipse.org/
[7]http://www.hibernate.org/

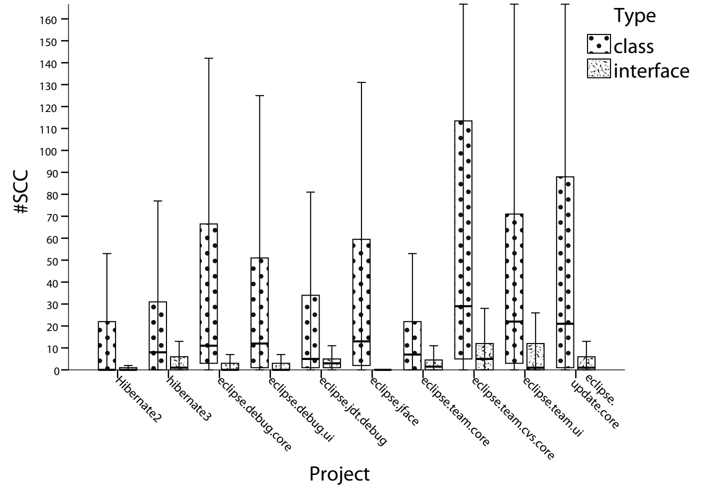


Fig. 4: Box plots of the *#SCC* of interfaces and classes per project

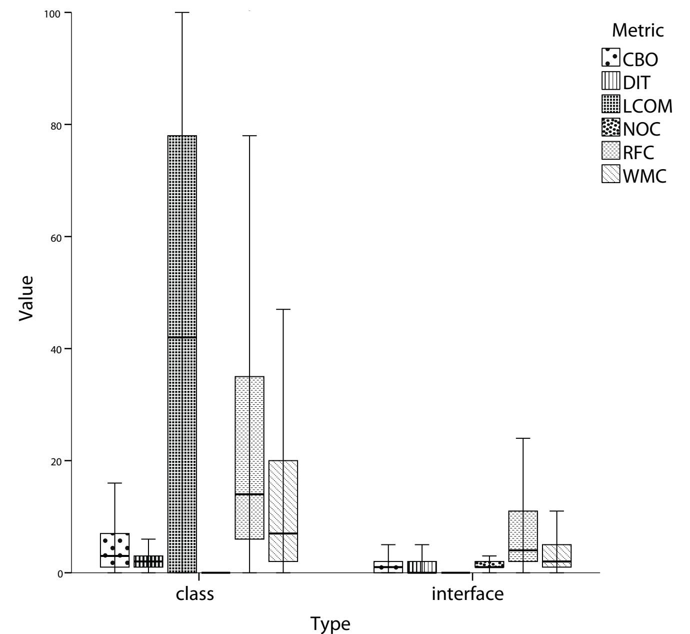


Fig. 5: Box plots of the C&K metric values for classes and interfaces measured over all selected projects

The results show that on average the number of changes involving Java classes are at least one order of magnitude higher than the ones involving Java interfaces. This result is not surprising since interfaces can be considered *contracts* among modules, and in general among logic units of a system.

Figure 5 shows the values of the C&K metrics for classes and interfaces over all ten projects. The values of the *CBO* metric are in general lower for interfaces, since it counts only the number of reference types in the parameters, return types and thrown exceptions in the method signatures. The values of the *RFC* metric are higher for classes than for interfaces. Also the values of the *DIT* metric are in general higher for

TABLE II: Spearman rank correlation between the C&K metrics and the #*SCC* computed for Java classes and Java interfaces (** marks significant correlations at α= 0.01, * marks significant correlations at α= 0.05, values in bold mark a significant correlation)

Project	CBO_c	CBO_i	NOC_c	NOC_i	RFC_c	RFC_i	DIT_c	DIT_i	$LCOM_c$	$LCOM_i$	WMC_c	WMC_i
Hibernate3	**0.590****	**0.535****	0.109**	0.029	0.338**	**0.592****	-0.098**	0.058	0.367**	0.103	0.617**	**0.657****
Hibernate2	0.352**	0.373**	0.134**	0.065	0.273**	0.325**	-0.156**	-0.010	0.269**	0.006	0.455**	**0.522****
eclipse.debug.core	**0.560****	0.484**	-0.025	0.105	0.431**	0.486**	0.065	0.232*	0.564	0.337	**0.600****	**0.597****
eclipse.debug.ui	**0.566****	0.216*	0.087*	0.033	0.291**	0.152	0.473**	0.324**	**0.626****	0.214*	-0.048	0.131
eclipse.jface	**0.570****	0.239*	0.257**	0.012	**0.516****	0.174**	0.173**	0.103	**0.563****	0.320**	**0.754****	0.137
eclipse.jdt.debug	**0.502****	**0.512****	0.154**	0.256**	0.132	0.349**	-0.089	-0.049	0.237**	0.238**	**0.668****	0.489**
eclipse.team.core	0.453**	0.367*	0.180*	0.102	0.435**	0.497**	0.060	0.243	0.335**	0.400	**0.561****	0.451**
eclipse.team.cvs.core	**0.655****	**0.688****	0.347**	-0.013	0.407**	**0.738****	-0.145	**0.618****	0.477**	**0.610****	**0.753****	**0.744****
eclipse.team.ui	**0.532****	0.301*	0.152**	-0.003	0.382**	0.299*	0.039	-0.103*	0.493**	0.395**	**0.595****	0.299*
eclipse.update.core	**0.649****	0.499**	0.026	-0.007	0.364**	0.381**	0.007	0.146	0.326**	0.482**	**0.735****	**0.729****
Median	**0.563**	0.428	0.143	0.031	0.373	0.365	0.023	0.124	0.422	0.328	**0.608**	**0.505**

classes than for interfaces.

Analyzing the *LCOM* we can notice that Java classes have a low median *LCOM* and hence a high cohesion. On the other hand, interpreting the *LCOM* of interfaces we can state that most of them do not expose any attributes in their body. In fact, the Understand tool registers a 0 *LCOM* when there are no attribute declarations, and 1 if there are some. The values of *WMC* confirm the assumptions made in Section III about the loss of meaning of this metric when applied to interfaces. In fact, the values of *WMC* correspond exactly to the value of the *NOM (Number of Methods)*. As expected, we registered higher values of the *NOC* for interfaces than for classes. This is due to the number of implementing classes that are counted as children by Understand.

Concerning the distribution of *IUC* values for interfaces, the median value is equal to 0.444 and the variance and the standard deviation are respectively equal to 0.165 and 0.406. Moreover, the skewness value is equal to 0.138, meaning that the distribution can be considered approximately symmetric [24].

A. Correlation between metrics and #SCC

The next step in our study aims at investigating the correlation between the metrics and the *#SCC* mined from the versioning repositories. We used the Spearman rank correlation analysis to identify highly-correlated metrics. Spearman compares the ordered ranks of the variables to measure a monotonic relationship. Differently to the Pearson correlation, the Spearman correlation does not make assumptions about the distribution, variances and the type of the relationship [25]. A Spearman value of +1 and -1 indicates high positive or high negative correlation, whereas 0 indicates that the variables under analysis do not correlate at all. Values greater than +0.5 and lower than -0.5 are considered to be substantial; values greater than +0.7 and lower than -0.7 are considered to be strong correlations.

To test the hypothesis **H1**, we performed two correlation analyses: (1) we analyze the correlation among the C&K metrics and the #*SCC* in Java classes and Java interfaces. An insignificant correlation of the C&K metrics for interfaces is a precondition for any further analysis of the interface complexity and usage metrics. (2) We explore the extent to which the interface cohesion, complexity and usage metrics correlate with *#SCC*.

Table II lists the results of the correlation analysis between the C&K metrics and *#SCC* for classes and interfaces in each project. The heading X_c indicates the correlation of the metric *X* with the *#SCC* of classes, and X_i the correlation with the #*SCC* of interfaces.

The first important result is that only the metrics CBO_c and WMC_c have a substantial correlation with the #*SCC* of Java classes, since their median correlation is greater than 0.5. In five projects out of ten WMC_c exhibits a substantial correlation and in three cases the correlation is strong. Similarly, the CBO_c metric shows a substantial correlation in eight cases but no strong correlations. The other metrics do not show a significant correlation with the *#SCC*.

The median correlation values of the C&K metrics applied to interfaces are significantly lower. Among the six metrics WMC_i exhibits the strongest correlation with *#SCC*. It shows three substantial and two strong correlations. CBO_i shows a substantial correlation for three projects.

We applied the same correlation analysis to the interface complexity and usage metrics defined in Section III. We report the result in Table III. IUC_i is the only metric that exposes a substantial correlation with the #*SCC* of interfaces. This metric shows a median correlation value of -0.605, having a substantial correlation in six projects and a strong correlation in one project. The negative correlation is due to the nature of the metric and it means that the IUC_i value is inversely proportional to the #*SCC*. More precisely, the stronger the external cohesion is (values of IUC_i close to one) the less frequently an interface changes.

Concerning the other metrics, the NOM_i shows the strongest correlation with the #*SCC*. This result is not surprising since the more methods are declared in the interface the more likely the interface changes. Surprisingly, neither the number of clients nor the number of invocations result in a substantial correlation with the #*SCC*. The $Arguments_i$ metric correlates only in three projects out of ten, while the APP_i shows a correlation only for one project. The $ClustersClients_i$ metric shows a substantial correlation only in one project.

TABLE III: Spearman rank correlation between the interface complexity and usage metrics and #*SCC* (** marks significant correlations at α= 0.01, * marks significant correlations at α= 0.05, values in bold mark a significant correlation)

Project	IUC_i	$Clients_i$	$Invocations_i$	$ClustersClients_i$	$ImplementingClasses_i$	$Arguments_i$	APP_i	NOM_i
Hibernate3	**-0.601****	0.433**	**0.544****	0.302**	0.021	**0.668****	0.450**	**0.657****
Hibernate2	-0.373**	0.104	0.165	0.016	0.054	0.531**	0.288**	**0.522****
eclipse.debug.core	**-0.682****	0.327**	0.317**	0.273**	0.070	0.298**	0.125	**0.597****
eclipse.debug.ui	**-0.508****	0.498**	0.497**	0.418**	0.139	0.128	-0.022	0.131
eclipse.jface	-0.363**	0.099	0.205*	0.106**	0.063	0.207*	0.110	0.137
eclipse.jdt.debug	**-0.605****	0.471	0.495**	0.474**	0.223	0.474**	0.361**	0.489**
eclipse.team.core	-0.475**	0.278	0.261	0.328*	0.102	0.241	0.138	0.451**
eclipse.team.cvs.core	**-0.819****	**0.608****	**0.557****	0.369	-0.037	**0.614****	0.383	**0.744****
eclipse.team.ui	**-0.618****	0.270	0.290	0.056	-0.003	0.144	-0.107*	0.299*
eclipse.update.core	**-0.656****	**0.656****	**0.677****	**0.606****	-0.095	0.433**	0.278	**0.729****
Median	**-0.605**	0.327	0.317	0.328	0.063	0.365	0.208	**0.505**

Therefore we conclude that the contribution of the number of methods shared among different clients is relevant for the correlation analysis. The weakest correlation is by the $ImpementingClasses_i$ metric.

Based on this result we can accept H1. Among the selected metrics, the IUC_i metric exhibits the strongest correlation with #*SCC* of interfaces. This result confirms our belief that the violation of the *Interface Segregation Principle* can impact the robustness of interfaces.

B. Prediction analysis

To test the research hypothesis **H2**, we analyzed whether the *IUC* metric can improve prediction models to classify interfaces into *change-prone* and *not change-prone*. We performed a series of classification experiments with three different machine learning algorithms. Prior work [26] showed that some machine learning techniques perform better than others, even though they state that performance differences among classifiers are marginal and not necessarily significant. For that reason we used the following classifiers: *Support Vector Machine (LibSVM)*, *Naive Bayes Network (NBayes)* and *Neural Nets (NN)* provided by the RapidMiner toolkit.

For each project, we binned the interfaces into *change-prone* and *not change-prone* using the median of the *#SCC* per project:

$$interface = \begin{cases} change\text{-}prone & if \quad \# \, SCC > median \\ not \; change\text{-}prone & otherwise \end{cases}$$

First, we trained the machine learning algorithms using the following object oriented metrics: *CBO*, *RFC*, *LCOM*, *WMC*. We selected these metrics because they showed the strongest correlation with the #*SCC*. We refer to this set of metrics as *OO*. Next, the training is performed using the *OO* metrics plus the *IUC* metric. We refer to this set of metrics as *IUC*.

In order to evaluate the classifications models, we use the area under the curve statistic (*AUC*). In addition we report the precision (*P*) and recall (*R*) of each model. *AUC* represents the probability, that, when choosing randomly a *change-prone* and a *not change-prone* interface, the trained model assigns a higher score to the *change-prone* interface [27]. We trained the models using 10 fold cross-validation and we considered models with an *AUC* value greater than 0.7 to have adequate classification performance [26].

Table IV reports the results obtained with the *NBayes* learner. The results show that the median *AUC* is higher when we include the *IUC* metric. Moreover, for each project we obtained an adequate performance (*AUC*>0.7) with the *IUC*. Only for two projects (*JDT Debug* and *Team UI*) out of ten we registered a better performance for the *OO* metrics. Using the *LibSVN* (see Table V) and the *NN* (see Table VI) classifiers we obtained similar results. With *LibSVN*, in eight projects the *IUC* metrics outperformed the *OO* metrics. Using *NN*, in seven projects out of ten the *IUC* metrics outperformed the *OO* metrics.

The median values of the *Precision* and *Recall* show similar results for most of the projects. In several projects, however, the Precision and Recall is affected by the lack of information about interfaces (*i.e.*, a high percentage of interfaces did not change during the observed time period). For instance, in the *eclipse.jface* project the number of interfaces that did not change is 81% (85 out of 105). The result is that the prediction model computed with the NN learner showed a *Precision* and *Recall* of 0.

TABLE IV: AUC, Precision and Recall using Naive Bayes Network (NBayes) with *OO* and *IUC* to classify interfaces into *change-prone* and *not change-prone*

Project	AUC_{OO}	P_{OO}	R_{OO}	AUC_{IUC}	P_{IUC}	R_{IUC}
eclipse.team.cvs.core	0.55	90	75	**0.75**	92.6	83.33
eclipse.debug.core	0.75	93	38	**0.79**	94.1	55.23
eclipse.debug.ui	0.66	63.81	40.33	**0.72**	69	41
hibernate2	0.745	78.62	32.02	**0.807**	84.22	85.33
hibernate3	0.835	88.61	57.92	**0.862**	82.8	56.31
eclipse.jdt.debug	**0.79**	69.67	47.67	0.738	77.71	45.38
eclipse.jface	0.639	50	28.33	**0.734**	53.85	48.33
eclipse.team.core	0.708	68.75	48.13	**0.792**	58.33	43.33
eclipse.team.ui	**0.88**	85	70	0.8	78.95	75
eclipse.update.core	0.782	67.49	46.5	**0.811**	81.19	61.67
Median	0.747	74.14	47.08	**0.791**	80.07	55.77

To investigate whether the difference between the AUC values of *OO* and *IUC* metrics are significant we performed the *Related Samples Wilcoxon Signed-Ranks Test*. The results of the test show a significant difference at α= 0.05 for the median AUC obtained with *LibSVN*. The difference between the medians obtained with *NBayes* and *NN* was not significant.

Based on these results we can partially accept the hypothesis

H2. The additional information provided by the *IUC* metric can improve the median performance of the prediction models by up to 9.2%. The Wilcoxon test confirmed this improvement for the *LibSVM* learner, however not for *NBayes* and *NN* learners. This result highlights the need to analyze a wider dataset in order to provide a more precise validation.

TABLE V: AUC, Precision and Recall using Support Vector Machine (LibSVN) with *OO* and *IUC* to classify interfaces into *change-prone* and *not change-prone*

Project	AUC_{OO}	P_{OO}	R_{OO}	AUC_{IUC}	P_{IUC}	R_{IUC}
eclipse.team.cvs.core	0.692	55.61	54.2	**0.811**	90.91	83.33
eclipse.debug.core	0.806	82.61	46	**0.828**	89.47	52.5
eclipse.debug.ui	0.71	75	21.33	**0.742**	80.83	26.8
hibernate2	**0.735**	70	40	0.708	66.76	45
hibernate3	0.64	52	33.45	**0.856**	82.4	73.36
eclipse.jdt.debug	0.741	67.17	56.24	**0.82**	68.56	58.33
eclipse.jface	0.607	66.67	45	**0.778**	72	62
eclipse.team.core	**0.617**	66.67	45	0.608	58.33	45
eclipse.team.ui	0.74	73.33	70	**0.883**	83.33	75
eclipse.update.core	0.794	86.67	56.83	**0.817**	81	64.17
Median	0.722	68.58	45.5	**0.814**	80.91	60.16

TABLE VI: AUC, Precision and Recall using Neural Nets (NN) with *OO* and *IUC* to classify interfaces into *change-prone* and *not change-prone*

Project	AUC_{OO}	P_{OO}	R_{OO}	AUC_{IUC}	P_{IUC}	R_{IUC}
eclipse.team.cvs.core	0.8	71.43	71.43	0.8	87.5	100
eclipse.debug.core	0.85	80	80	**0.875**	91.67	70
eclipse.debug.ui	0.748	79.33	44.67	**0.766**	78.05	58.5
hibernate2	0.702	53.85	50	**0.747**	50	45
hibernate3	**0.874**	83.17	69.52	0.843	78.49	69.05
eclipse.jdt.debug	**0.77**	73.39	63.24	0.762	80.5	58.05
eclipse.jface	**0.553**	0	0	0.542	0	0
eclipse.team.core	0.725	53.33	50	**0.85**	61.11	63.33
eclipse.team.ui	0.65	83.33	75	**0.75**	78.95	75
eclipse.update.core	0.675	70	58.33	**0.744**	78.33	56.67
Median	0.736	72.41	60.78	**0.764**	78.41	60.69

C. Summary of Results

The results of our empirical study can be summarized as follows:

The *IUC* metric shows a stronger correlation with the #*SCC* of interfaces than the C&K metrics. With a median Spearman rank correlation of -0.605, the *IUC* shows a stronger correlation with the #*SCC* on Java interfaces than the C&K metrics. Only the *WMC* metric shows a substantial correlation in five projects out of ten, with a median value of 0.505, hence we accepted H1.

The *IUC* metric improves the performance of prediction models to classify *change-* and *not change-prone* interfaces. The models trained with the *LibSVN* and *NBayes* using the IUC metric set outperformed the models computed with the OO metric set in eight out of ten projects. Using the *NN* learner, the models of seven projects showed better performance with the IUC metric set. This improvement in performance is significant for the models trained with *LibSVN*, however not for the other two learners. Therefore, we partially accepted H2.

V. Discussion

This section discusses the implications of our results and the threats to validity.

A. Implications of Results

The implications of the results of our study are interesting for researchers, quality engineers and, in general, for developers and software architects.

The results of our study can be used by researchers interested in investigating software systems through the analysis of source code metrics. Studies based on source code metrics should take into account the nature of the entities that are measured. This can help to obtain more accurate results.

Quality engineers should consider the possibility to enlarge their metric suite. In particular, the set of metrics should include specific metrics for measuring the cohesion of interfaces, such as the *IUC* metric. The C&K metrics are limited in measuring this cohesion of interfaces.

Finally, developers and software architects should use the *IUC* metric to measure the conformance to the *ISP*. Our results showed that low *IUC* values, indicating a violation of the *ISP*, can increase the effort needed to maintain software systems.

B. Threats to Validity

We consider the following threats to validity: *construct*, *internal*, *conclusion*, *external* and *reliability validity*. Threats to *construct validity* concern the relationship between theory and observation. In our study, this threat can be due to the fact that we measured the metrics on the last version of the source code. Previous studies in literature also used metrics collected from a single release (*e.g.*, [28] [29]). We mitigated this threat by collecting the metrics from the *last* release, since this release reflects the history of a system. Nevertheless, we believe that further validation with metrics measured over time (*i.e.*, from different releases) is desirable.

Threats to *internal validity* concern factors that may affect an independent variable. In our study, the independent variables (values of the metrics and #*SCC*) are computed using deterministic algorithms (provided by the Understand and Evolizer tools) that always deliver the same results.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. Wherever possible, we used proper statistical tests to support our conclusions for the two research questions. We used the Spearman correlation, which does not make any assumption on the underlying data distribution to test H1. To address H2 we selected a set of three machine learning techniques. Further techniques can be applied to build predictive models, even though previous work [26] states that performance differences among classifiers are not significant.

Threats to *external validity* concern the generalization of our findings. In our study, this threat can be due to the fact that eight out of ten projects stem from the Eclipse platform. Therefore, the generalizability of our findings and conclusions should be verified for other projects. Nevertheless, we considered systems of different size and different roles

in the Eclipse platform. Eclipse has been widely used by the scientific community and we can compare our findings with previous work. Moreover, we added two projects from Hibernate. As a matter of fact, any result from empirical work is in general threatened by the bias of their datasets [5].

Threats to *reliability validity* concern the possibility of replicating our study and obtaining consistent results. The analyzed systems are open source systems and hence publicly available; the tools used to implement our approach (Evolizer and ChangeDistiller) are available from the reported web sites.

VI. Related Work

In this section, we discuss previous work related to the usage of change prediction models to guide and understand maintenance of software systems.

Rombach was among the first researchers to investigate the impact of software structure on maintainability aspects [8], [30]. He focused on comprehensibility, locality, modifiability, and reusability in a distributed system environment, highligthing the impact of the interconnectivity between components.

In literature several approaches used source code metrics to predict the change-prone classes. Khoshgoftaar and Szabo presented an approach to predict maintenance measured as lines changed [31]. They trained a regression model and a neural network using size and complexity metrics. Li and Henry used the C&K metrics to predict maintenance in terms of lines changed [9]. The results show that these metrics can significantly improve a prediction model compared to traditional metrics. In 2009, Mauczka *et al.* measured the relationship of code changes with source-level software metrics [28]. This work focuses on evaluating the C&K metrics suite against failure data. Zhou *et al.* [32] used three size metrics to examine the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. A further validation of the object-oriented metrics was provided by Alshayeb [29]. This work highlights the capability of those metrics in two different iterative processes. The results show that the object-oriented metrics are effective in predicting design efforts and source lines modified in the short-cycled agile process. On the other hand they are ineffective in predicting the same aspects in the long-cycled framework process.

Object-oriented metrics were not only successfully applied for maintenance but als for defect prediction. Basili *et al.* [10] empirically investigated the suite of object-oriented design metrics as predictors of fault-prone classes. Subramanyam *et al.* [4] validated the C&K metrics suite in determining software defects. Their findings show that the effects of those metrics on defects vary across the data set from two different programming languages, C++ and Java.

Besides the correlation between metrics and change proneness, other design practices have been investigated in correlation with the number of changes. Khomh *et al.* [33] investigated the impact of classes with code smells on *change-proneness*. They showed that classes with code smells are more change-prone than classes without and that specific smells are more correlated than others. In 2008, Di Penta *et al.* [34] developed an exploratory study to analyze the change-proneness of design patterns and the kinds of changes occurring to classes involved in design patterns.

A complementary branch of change prediction is the detection of change couplings. Shirabad *et al.* [35] used a decision tree to identify files that are change coupled. Zimmermann *et al.* [36] developed the ROSE tool that suggests change coupled source code entities to developers. They are able to detect coupled entities on a fine-grained level. Robbes *et al.* [37] used fine-grained source changes to detect several kinds of distinct logical couplings between files. Canfora *et al.* [38] use the multivariate time series analysis and forecasting to determine whether a change occurred on a software artifact was consequentially related to changes on other artifacts.

Our work is complementary to the existing work since (1) we explore limitations of the C&K metrics in predicting the change-prone Java interfaces; (2) we investigate the impact of the *ISP* violation as measured by the *IUC* metric on the change-proneness of interfaces.

VII. Conclusions and Future Work

Interfaces declare contracts that are meant to remain stable during the evolution of a software system while the implementation in concrete classes is more likely to change. This leads to a different evolutionary behavior of interfaces compared to concrete classes.

In this paper, we empirically investigated this behavior with the C&K metrics that are widely used to evaluate the quality of the implementation of classes and interfaces. The results of our study with eight Eclipse plug-in and two Hibernate projects showed that:

- The *IUC* metric shows a stronger correlation with *#SCC* than the C&K metrics when applied to interfaces (we accepted H1)
- The *IUC* metric can improve the performance of prediction models in classifying Java interfaces into change-prone and not change-prone (we partially accepted H2)

Our findings provide a starting point for studying the quality of interfaces and the impact of design violations, such as the ISP, on the maintenance of software systems. In particular, the acceptance of the hypothesis H1 implicates engineers should measure the quality of interfaces with specific interface cohesion metrics. Software designers and architects should follow the interface design principles, in particular the ISP. Furthermore, researchers should consider distinguishing between classes and interfaces when investigating models to estimate and predict the change-prone interfaces.

In future work, we plan to evaluate the *IUC* metric with more open source and also commercial software systems. Furthermore, we plan to analyze the performance of our models taking into account releases (train the model with a previous release to predict the change-prone interfaces of the next release). Another direction of future work is to apply our models to other types of systems, such as Component Based

Systems (*CBS*) and Service Oriented Systems (*SOS*), in which interfaces play a fundamental role.

ACKNOWLEDGMENT

This work has been partially funded by the NWO-Jacquard program within the ReSOS project.

REFERENCES

[1] F. P. Brooks, Jr., "The mythical man-month," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, pp. 193–.

[2] 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. on Softw. Maintenance*. IEEE Press, 2004, pp. 40–49.

[3] L. Briand, W. Melo, and J. Wuest, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 706–720, July 2002.

[4] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 297–310, April 2003.

[5] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 2–13, January 2007.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 476–493, June 1994.

[7] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 35–45.

[8] H. D. Rombach, "A controlled expeniment on the impact of software structure on maintainability," *IEEE Trans. Softw. Eng.*, vol. 13, pp. 344–354, March 1987.

[9] W. Li and S. M. Henry, "Object-oriented metrics which predict maintainability," Blacksburg, VA, USA, Tech. Rep., 1993.

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

[11] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Prentice-Hall, Inc, 2002.

[12] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[13] Y. Zhou and H. Leung, "Predicting object-oriented software maintainability using multivariate adaptive regression splines," *J. Syst. Softw.*, vol. 80, pp. 1349–1361, August 2007.

[14] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *QSIC*, 2007, pp. 328–335.

[15] M. Perepletchikov, C. Ryan, and Z. Tari, "The impact of service cohesion on the analyzability of service-oriented software," *IEEE T. Services Computing*, vol. 3, no. 2, pp. 89–103, 2010.

[16] M. A. S. Boxall and S. Araban, "Interface metrics for reusability analysis of components," in *Proceedings of the 2004 Australian Software Engineering Conference*, ser. ASWEC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 40–.

[17] S. Tichelaar, S. Ducasse, and S. Demeyer, "Famix and xmi," in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, ser. WCRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 296–.

[18] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Softw.*, vol. 26, pp. 26–33, January 2009.

[19] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 725–743, November 2007.

[20] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, ser. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 11–18.

[21] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *ISSRE*, 2010, pp. 309–318.

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

[23] 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. New York, NY, USA: ACM, 2009, pp. 91–100.

[24] M. Bulmer, *Principles of statistics*. Dover Publications, 1979.

[25] S. D. S.Weardon and D. Chilko, *Statistics for Research. Probability ans Statistics*. John Wiley and Sons, 2004.

[26] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, pp. 485–496, July 2008.

[27] D. M. Green and J. A. Swets, *Signal detection theory and psychophysics*. Wiley, 1966, vol. 1.

[28] B. M. Mauczka A., Grechenig T., "Predicting code change by using static metrics," in *Software Engineering Research, Management and Applications*, 2009, pp. 64–71.

[29] M. Alshayeb and W. Li, "An empirical validation of object-oriented metrics in two different iterative software processes," *IEEE Trans. Softw. Eng.*, pp. 1043–1049, November 2003.

[30] H. D. Rombach, "Design measurement: Some lessons learned," *IEEE Softw.*, vol. 7, pp. 17–25, March 1990.

[31] T. M. Khoshgoftaar and R. M. Szabo, "Improving code churn predictions during the system test and maintenance phases," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 58–67.

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

[33] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 75–84.

[34] M. D. Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the relationships between design pattern roles and class change proneness," in *ICSM*, 2008, pp. 217–226.

[35] J. S. Shirabad, T. C. Lethbridge, and S. Matwin, "Mining the maintenance history of a legacy software system," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 95–.

[36] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572.

[37] R. Robbes, D. Pollet, and M. Lanza, "Logical coupling based on fine-grained change information," in *Proceedings of the 2008 15th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 42–46.

[38] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "Using multivariate time series and association rules to detect logical change coupling: An empirical study," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.