Delft University of Technology
Software Engineering Research Group
Technical Report Series

Can we Predict Types of Code Changes? An Empirical Analysis

Emanuel Giger, Martin Pinzger, and Harald C. Gall

Report TUD-SERG-2012-018

TUD-SERG-2012-018

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of the Working Conference on Mining Software Repositories, 2012, ACM, IEEE CS Press.

Can We Predict Types of Code Changes? An Empirical Analysis

Emanuel Giger
University of Zurich
giger@ifi.uzh.ch

Martin Pinzger
Delft University of Technology
m.pinzger@tudelft.nl

Harald C. Gall
University of Zurich
gall@ifi.uzh.ch

***Abstract*—There exist many approaches that help in pointing developers to the change-prone parts of a software system. Although beneficial, they mostly fall short in providing details of these changes. *Fine-grained source code changes* (*SCC*) capture such detailed code changes and their semantics on the statement level. These *SCC* can be condition changes, interface modifications, inserts or deletions of methods and attributes, or other kinds of statement changes. In this paper, we explore prediction models for whether a source file will be affected by a certain type of *SCC*. These predictions are computed on the static source code dependency graph and use social network centrality measures and object-oriented metrics. For that, we use change data of the Eclipse platform and the Azureus 3 project. The results show that Neural Network models can predict categories of *SCC* types. Furthermore, our models can output a list of the potentially *change-prone* files ranked according to their change-proneness, overall and per change type category.**

***Keywords*-Software maintenance; Machine Learning; Software quality**

I. Introduction

Researchers have developed methods and tools to better cope with software maintenance and evolution. Some approaches, *e.g.*, [1], [2], [3], use source code metrics to train prediction models, which can guide developers towards the change-prone parts of a software system. The main motivation for these approaches is that developers can better focus on these change-prone parts in order to take appropriate counter measures to minimize the number of future changes [4]. Other approaches, *e.g.*, [5], [6], [7], support developers in modification tasks that affect different source code locations by automatically eliciting past changes and change couplings between these source code entities. Moreover, the sensitivity to which the design of a system reacts to changes can be an indicator for its quality [8].

While the results of existing approaches are promising they fall short in providing insights into the details of changes. In particular, most of the current prediction models are based on coarse-grained change measures, such as code churn (lines added/deleted) or number of file revisions, *e.g.*, [9], [3]. These measures, however, do not capture the details about the semantics of changes. For instance, they do not provide detailed information whether a condition expression has changed or the declaration of a method was modified.

We explore in this paper to which extent data-mining models can predict if a source file will be affected by a certain category of source code change types, *e.g.*, declaration changes. For that, we leverage the (semantic) change information of *fine-grained source code changes* (*SCC*) [10]. To compute the prediction models we focuse on object-oriented metrics (*OOM*) [11] and centrality measures from social network analysis (*SNA*) [12] computed on the static source code dependency graph since they showed explicitly well predicting performance and in some cases achieved better performance than traditional metrics—both for change [1] and bug prediction [13].

Being able to predict not only if a file will most likely be affected by changes but additionally by what types of changes has practical benefits. For example, if a developer is made aware that there will be API changes she can plan accordingly and allocate resources for systemwide integration tests with dependent modules and, furthermore, she might account for additional time to update the API and design documents. In contrast, if only small statement changes are predicted localized unit tests will be sufficient and no further change impact can be expected.

In particular, we formulate two hypotheses:

H 1: *OOM* and *SNA* measures correlate positively with fine-grained source code changes.
H 2: *OOM* and *SNA* measures can predict categories of source code changes.

We investigate these hypotheses by a quantitative and manual analysis of 19 Eclipse plug-in projects as well as the Azureus 3 project. The results of our studies show that *OOM* and *SNA* metrics can be used to compute models to predict the likelihood that a source file will be affected by a certain category of source code changes. For instance, the models for predicting changes in the method declarations of Java classes obtained a median precision of 0.82 and a recall of 0.77. In all our models, the complexity of classes as well as the number of outgoing method invocations show the highest correlation and predictive power for our change type categories.

The remainder of the paper is organized as follows: Section II describes the process of data collection. Section III contains the empirical study with respect to our hypotheses and the manual analysis. Section IV provides a discussion of the findings. We describe related work in Section VI. Section VII points out possible future work.

978-1-4673-1761-0/12/$31.00 © 2012 IEEE

II. Data Collection

In this section we describe our approach, and the methods and tools we used in this paper. We need four kinds of information to prepare the dataset for our experiments in Section III: (1) Source code dependency graph; (2) Centrality measures from social network analysis [12] based on the dependency graph; (3) object-oriented source code metrics [11]; (4) and fine-grained source code changes [10].

Dependency Graph. The dependency graph of a software system depicts the relational structure between individual source code entities. We use the EVOLIZER suite [10] to extract the dependency information based on the version of the source code checked out from the trunk at the end of the timeframes listed in Table III for each project. We consider the following set of dependencies for our study: Method invocation, field access, inheritance, type cast, and instance-of check. We aggregated all dependency information on file level granularity. Hence, the nodes in the graph represent files and the edges indicate the existence of a dependency between two files. Similar to Zimmermann *et al.* [13], we distinguish between in- and outgoing connections and allow for self-connections, *i.e.*, a file can have dependencies to itself. In contrast to [13], however, we include weighted connections defined by the number of dependencies between two files. We choose weighted edges since centrality measures computed from an unweighted dependency graph showed lower correlation with *SCC* in our dataset. The result is a file-based, directed dependency graph in which edges are labeled with the number of dependencies.

Centrality Measures (*SNA*) stem from social network analysis and characterize the concept of *centrality* that identifies nodes in "*favored positions*" with more power [12]. Therefore, files having more ties to other files are "*more central*" and can be interpreted as more important. In practice, several approaches exist to measure the concept of centrality, see Table I. Centrality measures computed on the static dependency graph performed explicitly well for (bug) prediction purposes, *e.g.*, [14]. Moreover, in some cases they achieved better results than traditional metrics, *e.g.*, LOC, [13]. Hence, regarding **H 1** and **H 2**, we hypothesize that files which are more central and have more connections to others files are more *change-prone*.

For an overview of social network analysis we refer to [12]. We computed the centrality measures on the afore extracted file dependency graph with UCINET [15]. All measures were obtained at file level.

Object-Oriented Metrics (*OOM*) are a set of well established metrics measuring the size and complexity of object-oriented systems [11], see Table II. Prior work demonstrated their usefulness for building prediction models: For defect prediction, *e.g.*, [16], as well as change prediction models, *e.g.*, [1], [9]. The underlying rationale for our work is that more complex parts of a system are more likely to face changes. Again, the object-oriented metrics were computed on the version of the source code checked out from the trunk at the end of the timeframes listed in Table III for each project using UNDERSTAND (http://www.scitools.com/). We aggregated all metrics on file level.

Fine-Grained Source Code Changes (*SCC*). Version Control Systems (VCS) such as CVS, SVN, or GIT handle source files as pure text files ignoring their implicit code structure. Therefore, change measures such as lines added/deleted are rather imprecise since they can indicate code changes although no source code entities were changed, *e.g.*, in case of text formatting. Furthermore, they can not distinguish between different types of changes; changing the name of a class or the parameter list of a method declaration will both likely result in "+1 line changed". Another problem is that recording changes solely on file level, *i.e.*, revisions, can be too coarse grained: In our dataset around 8 distinct source code entities of the same file were changed per revision. Fluri

Table I
DESCRIPTION OF NETWORK CENTRALITY MEASURES (*SNA*)

Network Centrality Measures	
Approach	Measure (n=normalized [0-1])
Degree Centrality measures the concept of centrality based on the immediate ties of a file, *i.e.*, the more ties the more central a file is.	**Freeman Degree** counts the number of immediate ties a file has with other files. We distinguish between outgoing (*nOutDegree*) and incoming (*nInDegree*) ties [17].
	Bonacich's Power (*nPower*) computes centrality based on the number of immediate ties of a file, and on the number of immediate ties of its neighbors [18].
Closeness Centrality includes the indirect ties and focuses on the distance of an individual file to all other files in the dependency graph.	**Freeman shortest path closeness** is the reciprocal of the sum of the lengths of all shortest paths from a file, *i.e.*, *outCloseness* (or to a file, *i.e.*, *inCloseness*) to all other files in the graph [17].
	Reachability is the number of files reachable from a file (*nOutReach*) or which can reach a file (*nInReach*) within 1..n hops [15].
Freeman Betweenness (*nBetweeness*) determines how often a file is part of the shortest path between two other files [17].	

Table II
DESCRIPTION OF THE OBJECT-ORIENTED METRICS (*OOM*)

Object-Oriented Metrics [11]
Weighted methods per class (WMC) is the sum of the cyclomatic complexity of all methods of a class.
Coupling between object classes (CBO) counts the coupling to other classes.
Lack of cohesion in methods (LCOM) counts the number of pairwise methods without any shared instance variables, minus the number of pairwise methods that share at least one instance variable.
Depth of inheritance tree (DIT) denotes the maximum depth of the inheritance tree of a class.
Number of children (NOC) is the number of direct subclasses of a class.
Response for class (RFC) counts the number of local methods (including inherited methods) of a class.

et al. developed a tree differencing algorithm to extract *fine-grained source code changes* (*SCC*) [19]. They leverage the implicit structure of source code by comparing two different versions of the abstract syntax tree (AST) of a program and can track source code changes down to statement level, *e.g.*, method invocation statements.

Their algorithm matches nodes between two AST versions using string similarity measures for leave nodes and tree similarity measures for subtrees. Finding such node matches between two versions of an AST is necessary to determine wether a node was inserted, deleted, update, moved, or did not experience any change at all. Inserting, deleting, updating, and moving nodes constitute the basic tree edit operations that can possibly alter the structure of an AST (or any tree like structure in general). Next, a (minimal) set of such basic tree edit operations transforming one version of the AST into the other is generated. Each tree edit operation for a given node is then combined with the semantic information about the particular source code entity that the node represents within the AST. This allows to classify a basic tree edit operation using the taxonomy of source code changes presented in [10]. For instance, consider two AST nodes, A and B, that were inserted. A represents a method invocation statement within the AST structure and B an else-part. Accordingly, these two basic insert operations are classified as statement insert and else-part insert respectively.

The algorithm is implemented in CHANGEDISTILLER [10]. This tool compares the ASTs of each pair of subsequent revisions of all files of a system provided by its VCS. We applied CHANGEDISTILLER to the VCSs of all projects and extracted all *SCC* that occurred during the timeframes listed in Table III for each file.

Table III
DATASET USED IN THIS STUDY (DB=DEBUG)

Project	#Files	#Rev	*#LM*	*#SCC*	Time[M, Y]
Compare	154	2'953	111'749	17'263	May01-Sep10
jFace	378	5'809	304'744	22'203	Sep02-Sep10
JDT DB Jdi	144	1'936	63'602	6'121	May01-July10
JDT DB Eval	105	1'610	27'337	6'091	May01-July10
JDT DB Model	98	2'546	78'225	12'566	May01-July10
Resource	274	6'558	260'298	28'948	May01-Sep10
Team Core	169	1'995	38'317	4'607	Nov01-Aug10
CVS Core	188	5'448	157'176	23'301	Nov01-Aug10
DB Core	187	3'033	76'594	12'342	May01-Sep10
jFace Text	312	4'980	107'461	23'633	Sep02-Oct10
Update Core	275	6'379	151'823	27'465	Oct01-Jun10
DB UI	788	10'909	281'485	57'075	May01-Oct10
JDT DB UI	381	5'395	108'920	28'956	Nov01-Sep10
Help	110	999	20'661	5'919	May01-May10
JDT Compiler	322	19'466	1'099K	171'915	Jun01-Sep10
JDT Dom	157	6'608	233'105	32'699	Jun01-Sep10
JDT Model	420	16'892	596'320	90'128	Jun01-Sep10
JDT Search	115	5'475	201'876	44'372	Jun01-Sep10
OSGI	395	6'455	239'430	38'203	Nov03-Oct10
Azureus 3	368	6'327	187'869	46'232	Dec06-Apr10

Table IV
CATEGORIES OF CHANGE TYPES USED IN THIS STUDY.

Category	Description
cDecl	Combines all changes that modify the declaration of a class, *e.g.*, changing the class name.
func	Combines the insertion and deletion of functionality, *i.e.*, adding or removing methods.
oState	Combines the insertion and deletion of object states, *i.e.*, adding or removing class attributes.
mDecl	Combines all changes that modify the declaration of a method, *e.g.*, changing the method name.
stmt	Combines all changes that modify executable statements, *e.g.*, changing the name of local variable.
cond	Combines all changes that modify the condition expression in control structures.
else	Combines the insertion and deletion of *else*-parts.

III. EMPIRICAL STUDY

This section presents the empirical study we carried out to investigate our hypotheses formulated in Section I. We describe the dataset, the statistical methods we applied, and report on the results and findings.

A. Dataset

We conducted our study with 19 plugin projects of the Eclipse platform and the Azureus 3[1] project. They are well established in their domains, have a maintenance history of several years, and were often subject to prior research, *e.g.*, [20], [21], [22], [14], [23].

Table III gives an overview of our dataset: #Files denotes the number of unique **.java* files we obtained when checking out the source code version at the end of the timeframe (Time) from the trunk of the version control system. #Rev denotes the total number of revisions of the given source files within the timeframe, #LM denotes the total number of lines added and deleted, and *#SCC* is the total number of fine-grained source code changes.

An initial investigation of the dataset revealed large differences in how often certain *SCC* types occurred. In order to have higher frequencies for our experiments we combined several change types into one change type category according to their semantics (see Table IV). Some change types such as *adding attribute modifiability* (removing the keyword `final` from an attribute declaration) account—even if combined—for less than one percent of all changes and are left out in our analysis (see [10] for a list of all change types).

B. Correlation Analysis

We use the Spearman rank correlation to analyze the correlation of *SNA* and *OOM* metrics with *#SCC* (**H 1**). We choose the Spearman over Pearson correlation because it does not make any assumptions about the distribution of the data and measures the strength of a monotonic relation (rather than linear) between two variables [24]. The Spearman correlation obtains values between +1 and -1: +1 represents a high positive and -1 a high negative correlation between two variables. We consider values below -0.5 and above +0.5

[1] http://www.vuze.com, CVS-Path: azureus.cvs.sourceforge.net:/cvsroot/azureus Module: azureus3

as substantial correlations [25], and values below -0.7 and above 0.7 as strong correlations [2], [25]. We first examine the correlation between *#SCC* and centrality measures (*SNA*) and then analyze the correlation between *#SCC* and object-oriented metrics (*OOM*).

Centrality Measures: The left columns of Table V list the results of the correlation analysis per project. With a median correlation of 0.66 and exhibiting the largest value for 16 projects *nOutDegree* shows the strongest correlation of all centrality measures. For 8 projects we can observe strong correlations; 2 out of them have values of 0.8 and above. Only 3 projects are below 0.5. *nPower* is close to *nOutDegree* with the second highest median (0.65). 8 projects have strong correlations and 4 are below the level 0.5. The third highest median (0.61) has *nInDegree*. With a median of 0.53 *nOutReach* has a substantial correlation with *#SCC* on average.

outCloseness and *nBetweenness* both do not have a substantial correlation on average, but their values above 0.4 indicate an existing positive correlation that might have discriminatory power when building prediction models [13]. With an average of 0.02 and 0.19 *inCloseness* and *nInReach* do not exhibit any correlation and will be excluded from the prediction experiments.

The values in Table V indicate that there are differences regarding the correlations of certain centrality measures and *#SCC*. To investigate these differences, we first performed the *Related Samples Friedman Test* that was significant at $\alpha = 0.05$. We then used pair-wise post-hoc tests to statistically investigate the differences between individual centrality measures. We adjusted the α-level using the *Bonferroni Procedure* [24] for all post-hoc tests. Although the decisions are borderline, the post-hoc tests confirmed that measures based on outgoing connections, *i.e.*, *nOutDegree*, *nOutReach*, and *outCloseness* have significantly higher correlations than their corresponding measures based on incoming connections, *i.e.*, *nInDegree*, *nInReach*, and *inCloseness*.

Object-Oriented Metrics: The columns on the right hand side of Table V show the Spearman rank correlation values between object-oriented metrics and *#SCC* for each project. With a median of 0.73 WMC shows the highest correlation of all metrics for each project. For more than half of the projects it shows values of 0.7 and above. CBO has the second strongest correlation with a median of 0.64. It still has a substantial correlation on average, however, it is significantly lower than WMC. LCOM and RFC have a median correlation below 0.5. DIT shows a weak correlation with a median of 0.3. NOC shows no correlation with *#SCC* and will be excluded from the prediction experiments.

When comparing the object-oriented metrics with the network centralities, we observe that the median values of WMC and *nOutDregree*—both have the highest median correlation in their respective metric set—differ by 0.07 towards WMC. A *Wilcoxon Signed Rank Test* showed that this difference is significant. Not surprisingly, CBO showed no significant difference with *nOutDegree*, *nInDegree*, and *nPower*; according to their definitions in Table I and II, these metrics measure the immediate relation to other source files.

To summarize our results: The degree centrality measures *nOutDegree*, *nInDegree*, and *nPower*, and the object oriented metrics WMC and CBO showed substantial to strong correlation with *#SCC* measured for source files. WMC showed the strongest correlation with a median of 0.73. Furthermore, centrality measures based on outgoing connections are significantly stronger correlated than measures based on incoming connections. This is similar to the findings in [13] where outgoing connections of the dependency graph were more related to bugs than incoming connections. In both metric sets we observed measures that show very weak or no correlations at all. Based on these findings we accept **H 1**—*OOM* and *SNA* correlate positively with *#SCC*.

C. Predicting Change Type Categories

We first performed a series of classification experiments to investigate if network centrality measures and object-oriented metrics can be used for computing models to predict *change-prone* files. We then analyze whether those classification models can be refined towards our change type categories (**H 2**).

Experimental set-up: Prior to classification, we binned the files of each project into *change-prone* or *not change-prone* using the *median* of the total number of *SCC* per file (*#SCC*). These bins represent the *observed classes* when assessing the performance of the classification models later on. The median is a more robust measure, especially for highly skewed values as in our case.

Regarding the calculation of the classification models, we followed the advice by Lessmann *et al.* who found that more sophisticated classifiers might outperform others [26]. We therefore selected the 8 different classifiers. So far we could not consistently observe significant differences between all those classifiers in our experiments. However, Neural Network (NN) and Bayesian Network (BNet) achieved slightly better results. Hence, we only report on the classification performance of these two learners.

Concerning the evaluation of the classification models, we use the area under the receiver operating characteristic curve statistic (AUC), and also report the median precision (P) and recall (R) values. AUC represents the probability, that, when choosing randomly a *change-prone* and a *not change-prone* file the trained model then assigns a higher score to the change-prone file [26]. AUC is a robust measure to assess and compare the performance of classifiers since it is independent of prior probabilities and is also the recommended performance measure in [26], [27]. All models were trained using 10 fold cross-validation and AUC, precision, and recall were computed by evaluating each model on the same dataset it was computed from. We consider AUC

Table V
SPEARMAN RANK CORRELATION BETWEEN DIRECTED NETWORK CENTRALITY MEASURES, OBJECT-ORIENTED METRICS, AND *#SCC* AT THE LEVEL OF SOURCE FILES. * MARKS SIGNIFICANT CORRELATIONS AT $\alpha = 0.01$. THE LARGEST VALUE PER METRIC SET IS PRINTED IN **BOLD**.

Project	nOutDegree	nInDegree	nPower	outCloseness	inCloseness	nOutReach	nInReach	nBetweenness	WMC	CBO	LCOM	DIT	NOC	RFC
Compare	**0.77***	0.67*	0.74*	0.49 *	-0.20	0.68*	0.06	0.68*	**0.7***	0.67*	0.67*	0.57*	-0.19	0.66*
jFace	**0.75***	0.64*	0.74*	0.51*	-0.12	0.59*	0.02	0.42*	**0.77***	0.61*	0.65*	0.55*	0.02	0.61*
JDT DB Jdi	0.72*	**0.77***	0.72*	0.08	0.31*	0.17	0.26*	0.4*	**0.75***	0.44*	0.37*	-0.21	0.17	0.02
JDT DB Eval	**0.45***	0.41*	0.44*	-0.02	0.12	0.11	0.12	0.33	**0.65***	**0.65***	0.49*	-0.15	0.11	0.06
JDT DB Model	0.6*	**0.71***	0.57*	0.42*	0.07	0.52*	0.44*	0.56*	**0.7***	0.65*	0.52*	0.32	0.1	0.45*
Resource	**0.68***	0.64*	0.65*	0.45*	-0.08	0.56*	0.35*	0.55*	**0.75***	0.63*	0.46*	0.33*	-0.18*	0.65*
Team Core	0.45*	**0.50***	0.35*	0.23*	0.21*	0.29*	0.25*	0.35*	**0.51** *	0.46*	0.32*	0.28*	0.15	0.45*
CVS Core	**0.76***	0.62*	0.75*	0.26*	0.13	0.64*	0.28*	0.55*	**0.76***	0.66*	0.51*	0.31*	0.01	0.48*
DB Core	0.45*	**0.49***	0.42*	0.21*	0.03	0.37*	0.25*	0.35*	**0.62***	0.58*	0.49*	0.33*	-0.11	0.53*
JFace Text	**0.66***	0.64*	0.65*	0.57*	-0.22*	0.58*	-0.18*	0.41*	**0.75***	0.66*	0.6*	0.56*	-0.16*	0.71*
Update Core	**0.62***	0.60*	0.61*	0.42*	0.02	0.49*	0.13	0.37*	**0.72***	0.64*	0.41*	0.23*	-0.06	0.48*
DB UI	**0.65***	0.54*	0.40*	0.21*	0.14*	0.47*	0.19*	0.46*	**0.61***	0.56*	0.49*	0.29*	-0.08	0.35*
JDT DB UI	**0.56***	0.52*	**0.56***	0.31*	0.15*	0.34*	0.17*	0.31*	**0.54***	0.48*	0.37*	0.32*	-0.13*	0.31*
Help	**0.52***	0.45*	0.5*	0.42*	-0.20	0.52*	0.09	0.42*	**0.47***	**0.47***	0.47*	0.29*	-0.27*	0.3*
JDT Compiler	**0.85***	0.63*	**0.85***	0.63*	0.06	0.70*	0.40*	0.63 *	**0.84***	0.75*	0.52*	0.25*	0.17*	0.51*
JDT Dom	**0.76***	0.67*	0.75 *	0.51*	0.01	0.58*	0.13	0.36*	**0.85***	0.73*	0.39*	0.28*	-0.23*	0.46*
JDT Model	**0.66***	0.54*	**0.66***	0.52*	-0.25*	0.53*	0.32*	0.44*	**0.73***	0.68*	0.51*	0.23*	-0.05	0.49*
JDT Search	**0.8***	0.71*	0.76*	0.55*	-0.01	0.56*	0.25*	0.65*	**0.76***	0.61*	0.61*	0.35*	-0.08	0.42*
OSGI	**0.52***	0.48*	**0.52***	0.34*	-0.09	0.38*	0.18*	0.41*	**0.56***	0.52*	0.41*	0.29*	-0.15*	0.49*
Azureus 3	**0.71***	0.59*	**0.71**	0.55*	-0.01	0.58*	0.06	0.49*	**0.77***	0.67*	0.52*	0.55*	-0.17*	0.67*
Median	**0.66**	0.61	0.65	0.42	0.02	0.53	0.19	0.42	**0.73**	0.64	0.49	0.3	-0.08	0.48

Table VI
MEDIAN CLASSIFICATION RESULTS OVER ALL PROJECTS PER CLASSIFIER AND PER MODEL

	SNA			OOM			SNA&OOM		
	AUC	P	R	AUC	P	R	AUC	P	R
BNet	0.86	0.84	0.78	0.86	0.86	0.74	0.88	0.87	0.84
NN	0.87	0.88	0.82	0.86	0.89	0.81	0.86	0.87	0.83

Table VII
MEDIAN AUC, PRECISION, AND RECALL OF ACROSS ALL PROJECTS AND PER CATEGORY BASED ON NEURAL NETWORKS (NN)

Project	cDecl	func	oState	mDecl	stmt	cond	else
Median AUC	0.69	0.81	0.84	0.78	0.89	0.86	0.87
Median Precision	0.71	0.82	0.77	0.82	0.9	0.72	0.71
Median Recall	0.73	0.77	0.86	0.77	0.87	0.89	0.88

values above 0.7 to have adequate classification power [26]. We used Rapid Miner for all prediction experiments.[2]

Discriminating change-prone and not change-prone files: The first set of experiments is concerned with calculating models that are able to classify source files into *change-prone* and *not change-prone*. Table VI lists the median AUC, precision, and recall values across all projects when discriminating files into *change-prone* and *not change-prone*. *SNA* and *OOM* refer to the models that were trained when using centrality measures and object-oriented metrics separately as independent variables. *SNA&OOM* refers to the combined model using both metric sets in combination.

From the median performance values in Table VI we see that the AUC values of the trained models are all above the limit of 0.7, hence show adequate classification power. BNet based on *SNA&OOM* shows the best performance with a median AUC of 0.88, a median precision of 0.87, and a median recall of 0.84. Similarly good results are obtained by NN.

For SNA and OOM separately and for the joint model all AUC values are close. This means that centrality measures as well as object-oriented metrics can predict changes in files equally well. The combination of both metric sets improves the prediction performance slightly but not significantly. A comparison of the values at project level showed that in cases where centrality measures have a comparably lower correlation with changes, prediction models can gain more from using object-orient metrics. For example, JDT DB Eval and DB Core show the largest correlation difference between *SNA* and *OOM* (see Table V).

Discriminating change-prone and not change-prone files according to change type categories: Based on the promising results from the first set of experiments, we next refine our models in order to classify source files into *change-prone*$_C$ and *not change-prone*$_C$ given a change type category C defined in Table IV. Analogously to the previous experiment, we binned the files of each project into the observed classes *change-prone*$_C$ and *not change-prone*$_C$ using the median of the total number of changes for a given category C in a file: *cDecl*, *func*, *oState*, *mDecl*, *stmt*, *cond*, or *else*.

We trained Bayesian Networks (BNet) and Neural Networks (NN) in this experiment using *SNA* and *OOM* in combination as input variables. We could not observe a consistently significant performance difference across all categories between both classifiers. However, NN showed a better performance in the case of *cDecl*. We therefore only report on the median results of NN in Table VII and skip the results of BNet for readability and space reasons. Except for *cDecl* all other categories have an average AUC value above 0.7. A *One Sample Wilcoxon Signed-Ranks Test* showed that

[2] http://rapid-i.com/

except for *cDecl* all other categories have significantly higher median AUC values than 0.7. *oState*, *stmt*, *cond*, and *else* perform significantly higher than 0.8.

The median AUC indicates that prediction performance might vary based on how changes affect source code entities: Categories that aggregate changes within method bodies (MB categories), *i.e.*, *stmt*, *cond*, and *else* are above 0.8 and rank as the top three regarding the median AUC values; class body changes (CB categories), *i.e.*, *func* and *oState* have median AUC values close to 0.8; and declaration changes (D categories), *i.e.*, *cDecl* and *mDecl* show the lowest median AUC values around 0.7. We used a *Related Samples Friedman Test* and post-hoc tests with the *Bonferroni Procedure* to statistically investigate the AUC values of the different categories. The tests confirmed aforementioned observations: Within MB, CB, and D the performance differences of the respective change type categories are not significant. However, there are significant differences across MB, CB, and D.

To summarize our results: The first set of experiments showed that SNA and OOM metrics can be used to train models to accurately identify *change-prone* source files. With a median AUC of 0.88, BNet with *SNA* and *OOM* metrics as predictor variables showed the best performance. Second, we refined our models to identify *change-prone* source files according to a given change type category. For each change type category, except *cDecl*, the NN learner obtained models with good predictive power using the *SNA* and *OOM* metrics as independent variables. An analysis of the AUC values among these categories revealed that the performance of the models differs between the types of changes that affect the declaration or body of a class or method. Based on these findings we accept **H 2**—*SNA&OOM* measures can predict categories of source code changes.

D. Manual Analysis of Changes

The correlation analysis and our prediction models showed that coupling to other classes is strongly related with changes. Method invocation statements are part of the *stmt* category and typically account for most of the coupling. To further investigate this potential relationship between coupling and changes, we carried out an initial analysis in which we manually analyzed a sample set of methods and their changes in our dataset. The goal is to find evidence that outgoing dependencies (*i.e.*, method invocations) are indicators for the change-proneness of a method and class, respectively. In particular, we searched for specific instances of invocation statements that changed multiple times in a series of revisions. Such invocations caused maintenance effort over an extended time period rather than only once.

We selected different methods from each project according to the following two criteria: (1) The method was changed frequently in the past (relative to the other methods in the project). (2) A relatively large portion of the changes in a method affected method invocations. After having selected the candidate methods, we manually inspected all subsequent revision-pairs of each method (in total over 350 revisions).

Table VIII
nOutDegree, CBO, THEIR MEDIANS AT PROJECT LEVEL, AND THE PROBABILITY BY WHICH BNET MODELS USING *SNA* AND *OOM* AS PREDICTORS CORRECTLY CLASSIFIED A FILE AS *change-prone*.

File	nOutDegree	Median	CBO	Median	BNet Prob
LaunchConfiguration	0.3	0.003	19	2	1.0
ComparePreferencePage	0.1	0.007	25	2	1.0
EclipseSynchronizer	0.9	0.04	32	6	1.0

In the following, we report on three representative method examples that we found during the manual inspection. These examples include methods of the classes LaunchConfiguration, ComparePreferencePage, and EclipseSynchronizer of the projects DB Core, Compare and CVS Core, respectively. All these classes exhibit large values of the metrics *nOutDegree* and CBO compared to the project specific median values. Moreover, our BNet model classified those classes as *change-prone* with a probability of 1.0 (see Table VIII).

In the remainder, $R[r_t - r_{t+1}]$ denotes the subsequent (file) revision pair in which a method changed, *e.g.*, R[1.6-1.7]. For each change we state the change type category in brackets.

LaunchConfiguration.launch(...):[3] 26% of all changes over 20 revisions were method invocation changes. Between revisions R[1.18-1.19] the method call `initializeSourceLocator(...)` was added to the method body (1 x *stmt*). In the following, this method invocation changed in 7 revisions. Between R[1.20-1.21], the invocation statement was moved to an if-statement that performs null-checks of its parameters (1 x *stmt*). From R[1.22-1.23] the invocation statement was moved to the else-part of the parent if-statement (1 x *stmt*, 1 x *else*). From R[1.41-1.42] the condition of the if-statement was changed (1 x *stmt*, 1 x *cond*). From R[1.45-1.46] the invocation was moved to a different location within the method body and its else-part was deleted (1 x *stmt*, 1 x *else*). For a better exception handling it was then moved into a try-catch block between R[1.46-1.47] (1 x *stmt*). From R[1.51-1.52] the method invocation was removed and then re-inserted at the same source location in the subsequent revision R[1.52-1.53] (2 x *stmt*). After R1.53, it was not changed anymore.

ComparePreferencePage.createGeneralPage(...):[4] 62% of all changes over 18 revisions were method invocation changes. In this example, a group of similar invocation statements experienced multiple changes over several revisions. From R[1.11-1.12] a new instance of the method invocation `addCheckBox(...)` was added next to two existing ones (1 x *stmt*). It was the beginning of a series of insertions and deletions of instances of this particular invocation spanning

[3] org.eclipse.debug.internal.core.LaunchConfiguration.launch (String, IProgressMonitor)

[4] org.eclipse.compare.internal.ComparePreferencePage.createGeneralPage(Composite)

over 13 out of 18 revisions of the method. In particular, from R[1.51-1.52] a "commented" instance of the method invocation was inserted (1 x *stmt*); the comment was removed from R[1.52-1.53] (1 x *stmt*).

EclipseSynchronizer.endOperation(...):[5] 30% of all changes over 14 revisions were method invocation changes. This is an example where even after refactoring changes occurred. From R[1.12-1.13] existing source code was refactored into the new method `commitCache()` and replaced by a corresponding delegate invocation and an if-statement (*1 x stmt*, 1 x *func*). Between R[1.16-1.17] status handling was added to the invocation statement (1 x *stmt*). It was then moved into a try-finally block in R[1.20-1.21] (1 x *stmt*). The condition expression of the if-statement around the invocation and the finally-part were changed from R[1.21-1.22] (1 x *cond*, 4 x *stmt*). Again, the condition expression was changed from R[1.59-1.60] (1 x *cond*, 1 x *stmt*). The finally-part was changed again one revision later, *i.e.*, R[1.60-1.61] (1 x *stmt*).

To summarize our results: All three files of the above described methods exhibit a large number of method invocations in our dataset, *i.e.*, they exhibit *nOutDegree* and *CBO* values significantly above the median of their respective project and were all correctly classified as *change-prone* by our models. These findings support the meaningfulness of our models for predicting *change-prone* files based on their (outgoing) coupling properties. However, a more in depth analysis over time is necessary to validate if an early awareness of the upcoming changes in a particular file raised by our models could have prevented the above observed changes. For instance, in case of the second example by redesigning the initial UI design and behavior.

In this manual analysis we focused on method invocations for two reasons: (1) *nOutDegree* showed the strongest correlation out of all centrality measures. (2) Using CHANGEDISTILLER we can map changes directly to invocation statements as they are part of the AST. Complexity (WMC) on the other hand is based on the control flow graph. Hence, relating fine-grained changes to the concept of *complexity* is less clear.

IV. DISCUSSION

In the following we discuss the possible scenarios and practical implications that emerge from being able to predict the type of changes (rather than changes in general) in the context of the software development process, testing, and release planning.

Software development process. The additional information provided by our models about change type categories can help developers in systematically classifying the predicted change-prone files according to the expected change impact and development effort. For example, statement changes (*stmt*) are locally limited to their proximate context and typically do not induce changes at other locations in the source code. In contrast, changes in the API, *e.g.*, denoted by the method declaration (*mDecl*) and functionality (*func*) change type categories, typically have a higher impact and induce more work. For instance, changes in the API also require developers to update the API and design documentation, and to synchronize with other developers using the API. Hence, the ability to predict the "type" of changes that will occur in source files helps to estimate and anticipate in advance the kind and dimension of effort related to that change type.

Testing. Predicting the type of changes is of practical interest for software testing as different changes require different tests: While for small changes (*stmt*) unit-tests are mostly sufficient, API changes, such as indicated by the categories *mDecl* and *func*, might require integration-tests which are more expensive. Adding an else-part (*else*) or changing conditional expressions (*cond*) require new branch-testing procedures that cover the modified structure of the control flow graph. Hence, knowing which types of changes will most likely occur in a source file can help to optimally plan and develop tests, and (in case of limited resources) prioritize among different types of testing.

Release planning. Early awareness regarding the type of changes can help to plan development tasks when facing upcoming release dates. Typically, statement changes are more frequently integrated through small patches, whereas API changes are only released in major steps or not until the final agreement of quality assurance or senior developers.

Furthermore, by applying our models to the source code of a legacy system, they can warn developers if a certain critical threshold for *nOutDegree* or *WMC* is reached. For example, in our dataset the median of these two metrics is roughly such a threshold regarding the change type category *cond*. Hence, exceeding it will significantly raise the probability of a file to be affected by that category. Based on our models we can provide such simple rules of thumb to developers. This explicit empirical quantification between a particular category of changes and the coupling and complexity structure of a file enables systematic re-factorings to prevent specifically such control flow changes in the future rather than (textual) changes in general.

The basic tools, such as CHANGEDISTILLER and EVOLIZER, exist. As future work we plan to integrate them together with our models into Eclipse or a continuous integration environment, *e.g.*, Hudson, allowing automated model building, *e.g.*, during work, nightly builds, or before committing a new version.

V. THREATS TO VALIDITY

Despite the promising results a careful discussion of the validity of our work is needed.

[5] org.eclipse.team.internal.ccvs.core.resources.EclipseSynchronizer.endOperation(IProgressMonitor)

Our correlation analysis and prediction models indicate the potential existence of relations of *OOM* and *SNA* with *#SCC*, but do not provide a solid causal proof and explanation for it. There might be other reasons that impose changes to a software system as indicated, for instance, by commit messages, bug reports, or patches. Additional work is needed that includes these potential change indicators as well. Moreover, we treated this relation as directional in our work, *i.e.*, using *SNA* and *OOM* to predict *#SCC*. However, the examples in Section III-D indicate that it might be mutual. For instance, when inserting an invocation statement (*stmt*), the coupling structure is altered by changes themselves. This possibly threatens the content validity. We chose OOM and SNA since they proofed their usefulness in prior work for various prediction tasks. We are aware that other features, *e.g.*, LOC, and feature selection methods, *e.g.*, *Information Gain* [27], exist in literature. For that, further experiments are needed to guarantee optimal performance and models with the most predictive (sub-)set of features. An extensive comparison study with a richer set of features is part of our future work. A threat to external validity stems from the fact that our work is dominated by data from the Eclipse platform. This imposes a certain bias caused by characteristics typical to the Eclipse maintenance process, *e.g.*, specific commit behavior. Therefore, the generalizability of the findings and conclusions of this paper might be influenced and have to be verified for other projects. As a matter of fact, any result from empirical work is threatened by the bias of its datasets [27]. Software development is influenced by a variety of factors, both internally and externally, that differ across domains and projects. However, Eclipse and Azureus are relatively large and established systems; both projects have been subject of numerous studies, *e.g.*, [20], [21], [22], [14], [23], [28], before. As such we can benefit from and continue upon prior findings. Replication is central to empirical research and can—even if not carried out identically—enhance existing knowledge [29]. Therefore, we are convinced that our findings can add to existing work, sustain current hypotheses and raise new results.

A threat to construct validity might be caused by how we measured *SNA* and *OOM* compared to *SCC*. *SCC* reflect the maintenance activities during a given period. In contrast, *SNA* and *OOM* are based on the source code and represent the dependency and complexity structure at a specific point in time. For this study, the dataset was composed of all *SCC* regarding the timeframes in Table III. On the other hand, *SNA* and *OOM* were measured on the latest source code version available at the end of those timeframes. Our method does not take into account this time gap, *i.e.*, that the relation between *SNA* and *OOM* with *SCC* can change over time.

VI. Related Work

This section discusses related work about social network techniques in software engineering and change prediction.

Social Network Analysis. Work on this subject stems from the emerging perception that todays software projects are complex networks in which people, technologies, and artifacts show manifold interactions. The idea is to shift to the socio-technial aspects of a project.

Bird *et al.* found out that sub-communities can emerge among the members of open-source (OS) projects [30]. Social network analysis was applied to CVS information to investigate the structure and evolution of OS projects [31]. Huang *et al.* used a *Legitimate Peripheral Participation* model to describe the interactions between developers in OS projects and divide them into core and peripheral teams [32]. OS teams often consist of a small number of developers who seek knowledge beyond their own. Ohira *et al.* used collaborative filtering and social network analysis to locate expertise and knowledge across different projects [33].

Duchenaut investigates the process of newcomers becoming a core member in the Python project [34]. The success of this socialization process is determined by two factors: (1) An individual learning process where newcomers acquire technical skills and project related knowledge. (2) A political process where newcomers have to gain reputation among the senior developers by demonstrating their skills and following certain established rites within the project. A study about the process of people joining OS projects was carried out in [35].

Our approach focuses solely on source code and its dependency structure rather than the relation of people participating in a certain project. Closer to our work are studies that relate social network analysis to the quality of a software system. In [13], social network measures based on the static dependency graph of Windows Server 2003 binaries turned out to be good indicators for defect-prone binaries. Recently, this work was replicated with data from the Eclipse platform [14]. Pinzger *et al.* related centrality measures computed on the developer contribution network of Windows Vista to post-release failures [25]: More central binaries tend to be more failure-prone. In contrast, our goal is to predict categories of changes rather than defects.

Change Prediction builds models to guide and understand maintenance. Rombach was among the first to study the relation between the structure of a system and maintenance [36]. In particular, they showed that architectural design measures capturing the interconnectivity between components influence maintainability. The same object-oriented metrics as in our work were used to predict maintenance in terms of lines changed [1]. The results show that these metrics can significantly improve prediction model compared to traditional metrics, *e.g.*, number of semicolons. Their dataset was re-applied in [9] with the focus on comparing the performance of several learning models. Object-oriented metrics and *SCC* were not only successfully applied for maintenance but for defect prediction as well, *e.g.*, [16], [37], [23]. In [38] an approach is presented to predict maintenance

measured as lines changed using a regression model and a neural network based on size and complexity metrics. A study to predict different maintenance types, *e.g.*, preventive maintenance, is given in [2]. Several regression models were build based on object-oriented metrics and an extended set of coupling metrics. Similar to our observations, the study states that (immediate) coupling metrics are good predictors while inheritance related metrics are not. To introduce an analogy to meteorology the term *Yesterdays Weather* was coined [4]: Classes that changed recently will likely change again in the future. Dynamic instead of static coupling information was used—collected at runtime and from dynamic UML models—to predict changes in terms of lines changed [3]. In [39] and [40], two approaches were presented that focus on the probability that a source code change will introduce a failure.

A complementary branch of change prediction is the detection of *change-couplings* between code entities. Such dependencies are often logical and implicit and can not be detected by static analysis alone. Shirabad *et al.* used a decision tree to identify files that are change-coupled [41]. They showed that models built on text features, *i.e.*, words extracted from source code comments and problem reports, performed the best. Tsantalis *et al.* calculated the change-proneness of a class by determining the probability by which it is affected when features in a system change [8]. The idea is to quantify the quality of an object-oriented design that ideally should not be sensitive to changes. Ying *et al.* used association rule mining to recommend additional classes that are potentially relevant for modification tasks [6]. The ROSE tool suggests change-coupled source code entities to developers at a fine-grained level, *e.g.*, instance variables [5]. Robbes *et al.* used fine-grained source changes to detect several kinds of distinct logical couplings between files [7]. CHANGEDISTILLER was used to detect changes that are irrelevant (non-essential) to change tasks [28]. Our work is complementary in the way that we explored the feasibility to predict categories of changes.

VII. CONCLUSIONS & FUTURE WORK

We showed that centrality measures from social network analysis (*SNA*) computed on the static source code dependency graph and object-oriented metrics (*OOM*) positively correlate with fine-grained source code changes (*SCC*). Our models can output a list of the potentially *change-prone* files ranked according to their change-proneness, overall and per change type category. In summary, the results of our work are:

- *SNA* and *OOM* positively correlate with *#SCC*. Moreover, *Degree Centrality* measures and complexity (WMC) show particularly strong correlations (accepted **H 1**).
- Neural Networks based on *SNA* and *OOM* can predict categories of code change types. For instance, the model for predicting changes in method declarations of classes obtained a median precision of 0.82 and a recall of 0.77 (accepted **H 2**).
- A manual analysis of a subset of changes confirmed the empirical findings regarding the relation between coupling and changes.

Re-running our experiments with different timeframes, *e.g.*, release based, could give insights into how the relation between the position in the dependency graph of a file, its complexity and fine-grained changes evolves over time. Currently our dataset is Eclipse dominated. We plan to replicate our study with other systems including other labeling values in addition to the median. Our models were successful by means of quantitative criteria, *e.g.*, AUC. As future work we will conduct user studies to validate their usefulness for software maintenance, *e.g.*, preventing changes.

REFERENCES

[1] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, 1993.

[2] M. Dagpinar and J. Jahnke, "Predicting maintainability with object-oriented metrics - an empirical comparison," in *Proc. Working Conf. on Reverse Eng.*, 2003, pp. 155–164.

[3] E. Arisholm, L. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Trans. Softw. Eng.*, vol. 30, no. 8, pp. 491–506, 2004.

[4] 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*, 2004, pp. 40–49.

[5] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proc. Int'l Conf. on Softw. Eng.*, 2004, pp. 563–572.

[6] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, 2004.

[7] R. Robbes, D. Pollet, and M. Lanza, "Logical coupling based on fine-grained change information," in *Proc. Working Conf. on Reverse Eng.*, 2008, pp. 42–46.

[8] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the probability of change in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 7, pp. 601–614, 2005.

[9] Y. Zhou and H. Leung, "Predicting object-oriented software maintainability using multivariate adaptive regression splines," *Journal of Systems and Software*, vol. 80, no. 8, pp. 1349–1361, 2007.

[10] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

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

[12] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*, 1st ed. Camb. Univ. Press, 1994.

[13] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. Int'l Conf. on Softw. Eng.*, 2008, pp. 531–540.

[14] T. Nguyen, B. Adams, and A. Hassan, "Studying the impact of dependency network measures on software quality," in *Proc. Int'l Conf. on Softw. Maint.*, 2010, pp. 1–10.

[15] S. Borgatti, M. Everett, and L. Freeman, *UCINET 5.0 Version 1.00*, Natick: Analytic Technologies., 1999.

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

[17] L. C. Freeman, "Centrality in social networks conceptual clarification," *Social Networks*, vol. 1, no. 3, pp. 215 – 239, 1979.

[18] P. Bonacich, "Power and centrality: A family of measures," *The American Journal of Sociology*, vol. 92, no. 5, pp. 1170–1182, 1987.

[19] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Trans. on Softw. Eng.*, vol. 33, no. 11, pp. 725–743, 2007.

[20] A. Schroeter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proc. Int'l Symp. on Empirical Softw. Eng.*, 2006, pp. 18–27.

[21] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. Int'l Workshop on Predictor Models in Softw. Eng.*, 2007, pp. 9–15.

[22] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. Int'l Conf. on Softw. Eng.*, 2008, pp. 181–190.

[23] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proc. Int'l Workshop on Mining Softw. Repositories*, 2011, pp. 83–92.

[24] S. Dowdy, S. Weardon, and D. Chilko, *Statistics for Research*, 3rd ed., ser. Probability and Statistics. Hoboken, New Jersey: John Wiley and Sons, 2004.

[25] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proc. Symp. on the Found. of Softw. Eng.*, 2008, pp. 2–12.

[26] S. Lessmann, B. Baesens, C. M. Swantje, and Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. on Softw. Eng.*, vol. 34, no. 4, pp. 485–496, 2008.

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

[28] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proc. Int'l Conf. on Softw. Eng.*, 2011, pp. 351–360.

[29] N. Juristo and S. Vegas, "Using differences among replications of software engineering experiments to gain knowledge," in *Proc. Int'l Symp. on Empirical Softw. Eng. and Measurement*, 2009, pp. 356–366.

[30] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proc. Int'l Symp. on Found. of Softw. Eng.*, 2008, pp. 24–35.

[31] L. González, G. Barahona, and G. Robles, "Applying social network analysis to the information in cvs repositories," in *Proc. Int'l Workshop on Mining Softw. Repositories*, 2004, pp. 101–105.

[32] S.-K. Huang and K. min Liu, "Mining version histories to verify the learning process of legitimate peripheral participants," in *Proc. Int'l Workshop on Mining Softw. Repositories*, 2005, pp. 1–5.

[33] M. Ohira, N. Ohsugi, T. Ohoka, and K. Matsumoto, "Accelerating cross-project knowledge collaboration using collaborative filtering and social networks," in *Proc. Int'l Workshop on Mining Softw. Repositories*, 2005, pp. 1–5.

[34] N. Ducheneaut, "Socialization in an open source software community: A socio-technical analysis," *Comput. Supported Coop. Work*, vol. 14, pp. 323–368, 2005.

[35] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, "Open borders? immigration in open source projects," in *Proc. Int'l Workshop on Mining Softw. Repositories*, 2007, p. 6.

[36] D. Rombach, "A controlled experiment on the impact of software structure on maintainability," *IEEE Trans. on Softw. Eng.*, vol. 13, no. 3, pp. 344–354, 1987.

[37] R. Subramanyam and M. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, April 2003.

[38] T. Khoshgoftaar and R. Szabo, "Improving code churn predictions during the system test and maintenance phases," in *Proc. Int'l Conf. on Softw. Maint.*, 1994, pp. 58–67.

[39] A. Mockus and D. Weiss, "Predicting risk of software changes," *Bell Labs Technical J.*, vol. 5, pp. 169–180, 2000.

[40] S. Kim, J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, March 2008.

[41] S. Shirabad, T. Lethbridge, and S. Matwin, "Mining the maintenance history of a legacy software system," in *Proc. Int'l Conf. on Softw. Maint.*, 2003, pp. 95–104.

TUD-SERG-2012-018
ISSN 1872-5392