Are Popular Classes More Defect Prone?

Alberto Bacchelli *and* Marco D'Ambros *and* Michele Lanza

REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

Abstract. Traces of the evolution of software systems are left in a number of different repositories: such as configuration management systems, bug tracking systems, mailing lists. Developers use e-mails to discuss issues ranging from low-level concerns (bug fixes, refactorings) to high-level resolutions (future planning, design decisions). Thus, e-mail archives constitute a valuable asset for understanding the evolutionary dynamics of a system.
We introduce metrics that measure the "popularity" of source code artifacts, *i.e.,* the amount of discussion they generate in e-mail archives, and investigate whether the information contained in e-mail archives is correlated to the defects found in the system. Our hypothesis is that developers discuss problematic entities more than unproblematic ones. We also study whether the precision of existing techniques for defect prediction can be improved using our popularity metrics.

1 Introduction

Knowing the locations of future software defects allows project managers to optimize the resources available for the maintenance of a software project by focusing on the most problematic components. However, performing defect prediction with enough precision to produce useful results is a challenging problem. Researchers have proposed a number of approaches to predict software defects, exploiting a variety of sources of information, such as source code metrics [1,2,3,4,5], code churn [6], process metrics extracted from versioning system repositories [7,8], past defects [9,10]. A possible source of information for defect prediction that was not exploited so far is development mailing lists.

Due to the increasing extent and complexity of software systems, it is common to see large teams, or even communities, of developers working on the same project in a collaborative fashion. In such cases e-mails are the favorite media for the coordination between all the participants. Mailing lists, which are preferred over person-to-person e-mails, store the history of inter-developers, inter-users, and developers-to-users discussions: Issues range from low-level decisions (*e.g.,* bug fixing, implementation issues) up to high-level considerations (*e.g.,* design rationales, future planning).

Development mailing lists of open source projects are easily accessible and they contain information that can be exploited to support a number of activities. For example, the understanding of software systems can be improved by adding sparse explanations enclosed in e-mails [11]; the rationale behind the system design can be extracted from the discussions that took place before the actual implementation [12]; the impact of changes done on the source code can be assessed by analyzing the effect on the mailing list [13]; the behavior of developers can be analyzed to verify if changes follow

discussion, or vice-versa; hidden coupling of entities that are not related at code level can be discovered if often mentioned together in discussions.

One of the biggest challenges when dealing with mailing lists as a source of information is correctly linking each e-mail to any source code entity it discusses. In previous work we specifically tackled this issue [14]. Using a benchmark of a statistically significant size, we showed that lightweight grep-based techniques reach an acceptable level of precision in the linking task. Such techniques allow us to use mailing lists as a source of information about the source code.

Why would one want to use e-mails for defect prediction? The source code of software systems is only written by developers, who must follow a rigid and terse syntax to define abstractions they want to include. On the other hand of the spectrum, mailing lists, even those specifically devoted to development, archive e-mails written by both programmers and users. Thus, the entities discussed are not only the most relevant from a development point of view, but also the most exploited during the use of a software system. In addition, the content of e-mail is expressed using natural language, which does not require the writer to carefully explain all the abstractions using the same level of importance, but easily permits to generalize some concepts and focus on others. For this reason, we expect information we extract from mailing lists to be independent from those provided by the source code analysis. Thus, they can add valuable information to software analysis.

We present different "popularity" metrics that express the importance of each source code entity in discussions taking place in development mailing lists. Our hypothesis is that such metrics are an indicator of possible flaws in software components, thus being correlated with the number of defects. We aim at answering the following research questions:

- *Q1: Does the popularity of software components in discussions correlate with software defects?*
- *Q2: Is a regression model based on the popularity metrics a good predictor for software defects?*
- *Q3: Does the addition of popularity metrics improve the prediction performance of existing defect prediction techniques?*

We provide the answers to these questions by validating our approach on four different open source software systems.

2 Methodology

Our goal is first to inspect whether popularity metrics correlate with software defects, and to study whther existing bug prediction approaches can be improved using such metrics. To do so, we follow the methodology depicted in Figure 1:

- We extract e-mail data, link it with source code entities and compute popularity metrics. We extract and evaluate source code and change metrics.
- We extract defect data from issue repositories and we quantify the correlation of popularity metrics with software defects, using as baseline the correlation between source code metrics and software defects.

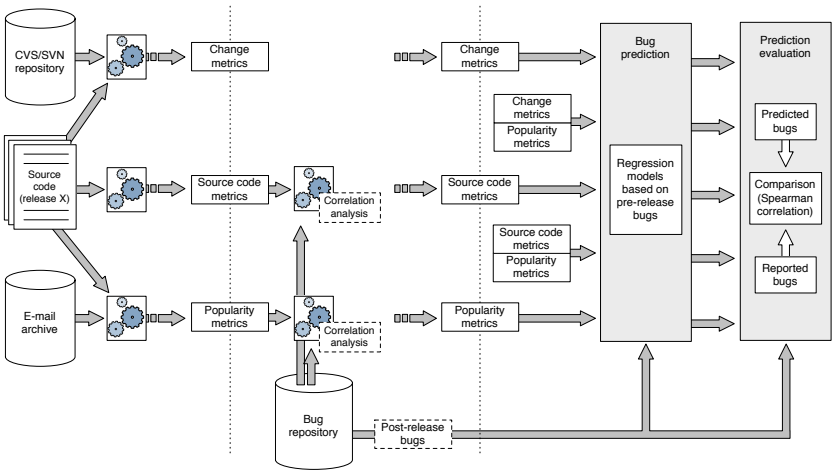

Fig. 1. Overall schema of our approach.

– We build regression models with popularity metrics as independent variables and the number of post-release defects as the dependent variable. We evaluate the performance of the models using the Spearman's correlation between the predicted and the reported bugs. We create regression models based on source code metrics [3,4,5,10] and change metrics [7,8] alone, and later enrich these sets of metrics with popularity metrics, to measure the improvement given by the popularity metrics.

In the experiments, we focus on object-oriented Java software systems using classes as target entities.

Modeling. The first step is to model the source code of the software systems we analyze. Using the tool inFusion[1], we extract the object-oriented model of the source code according to FAMIX, a language independent meta-model of object oriented code [15].

Computing Source Code Metrics. Once we obtain the FAMIX model of a software system, we compute a catalog of object oriented metrics, listed in Table 1. The catalog includes the Chidamber and Kemerer (CK) metrics suite [16], which was already used for bug prediction [1,17,3,4], and additional object oriented metrics.

Computing Change Metrics. Change metrics are "process metrics" extracted from versioning system log files (CVS and SVN in our experiments). Differently from source code metrics which measure several aspects of the source code, change metrics are measures of how the code was developed over time. We use the set of change metrics listed in Table 2, which is a subset of the ones used in [7].

[1] `http://www.intooitus.com/inFusion.html`

CK metrics		Other OO Metrics	
WMC	Weighted Method Count	FanIn	Number of other classes that reference the class
DIT	Depth of Inheritance Tree	FanOut	Number of other classes referenced by the class
RFC	Response For Class	NOA	Number of attributes
NOC	Number Of Children	NOPA	Number of public attributes
CBO	Coupling Between Objects	NOPRA	Number of private attributes
LCOM	Lack of Cohesion in Methods	NOAI	Number of attributes inherited
		LOC	Number of lines of code
		NOM	Number of methods
		NOPM	Number of public methods
		NOPRM	Number of private methods
		NOMI	Number of methods inherited

Table 1. Class level source code metrics.

Change Metrics			
NR	Number of revisions	NREF	Number of times file has been refactored
NFIX	Number of times file was involved in bug-fixing	NAUTH	Number of authors who committed the file
CHGSET	Change set size (maximum and average)	AGE	Age of a file

Table 2. Class level change metrics.

To use change metrics in our experiments, we need to link them with source code entities, *i.e.,* classes. We do that by comparing the versioning system filename, including the directory path, with the full class name, including the class path. Due to the file-based nature of SVN and CVS and to the fact that Java inner classes are defined in the same file as their containing class, several classes might point to the same CVS/SVN file. For this reason, we do not consider inner Java classes.

Computing Popularity Metrics. The extraction of popularity metrics, given a software system and its mailing lists, is done in two steps: First it is necessary to link each class with all the e-mails discussing it, then the metrics must be computed using the links obtained. In the following, we briefly present the technique to link e-mails to classes, then we discuss the popularity metrics we propose to answer our research questions. Figure 2 shows the process used to prepare the data for evaluating the popularity metrics.

First, we parse the target e-mail archive to build a model according to an e-mail meta-model we previously defined [14]. We model body and headers, plus additional data about the inter messages relationships, *i.e.,* thread details. Then, we analyze the FAMIX object-oriented model of the target source code release to obtain the representation of all the classes. Subsequently, we link each class with any e-mail referring it, using lightweight linking techniques based on regular expressions, whose effectiveness was validated in a previous work [14]. We obtain an object-oriented FAMIX model enriched with all the connections and information about classes stored in the e-mail archive. Through this model we can extract the popularity metrics listed in Table 3.

For each popularity metrics that we propose, we also provide the rationale behind their creation and a high-level description of their implementation, using our enriched object-oriented model.

POP-NOM: To associate the popularity of a class with discussions in mailing lists, we count the number of mails that are mentioning it. Since we are considering develop-

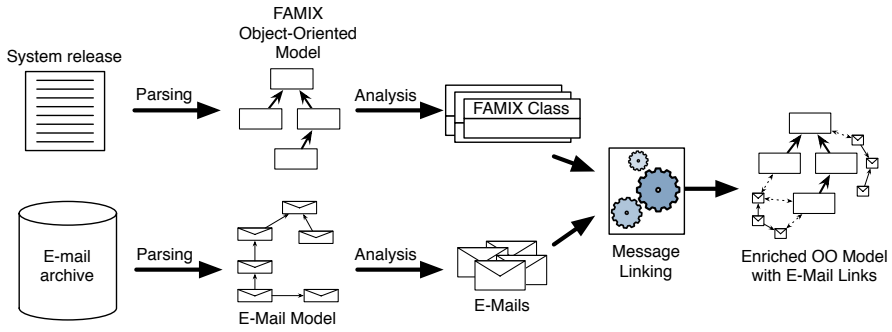

Fig. 2. Linking mails and classes.

Popularity Metrics			
POP-NOM	Number of Mails	POP-NOCM	Number of Characters in Mails
POP-NOT	Number of Threads	POP-NOMT	Number of Mails in Threads
POP-NOA	Number of Authors		

Table 3. Class-level popularity metrics.

ment mailing lists, we presume that classes are mainly mentioned in discussions about failure reporting, bug fixing and feature enhancements, thus they can be related to defects. Thanks to the enriched FAMIX model we generate, it is simple to compute this metric. Once the mapping from classes to e-mails is completed, and the model contains the links, we count the number of links of each class.

POP-NOCM: Development mailing lists can also contain other topics than technical discussions. For example, while manually inspecting part of our dataset, we noticed that voting about whether and when to release a new version occurs quite frequently in Lucene, Maven and Jackrabbit mailing lists. Equally, announcements take place with a certain frequency. Usually this kind of messages are characterized by a short content (*e.g.,* "yes" or "no" for voting, "congratulations" for announcements). The intuition is that e-mails discussing flaws in the source code could present a longer amount of text than mails about other topics. We consider the length of messages taking into account the number of characters in the text of mails: We evaluate the POP-NOCM metric by adding the number of characters in all the e-mails related to the chosen class.

POP-NOT: It is a long tradition in mailing lists to divide discussions in *threads*. Our hypothesis is that all the messages that form thread discuss the same topic: If an author wants to start talking about a different subject she can create a new thread. We suppose that if developers are talking about one defect in a class they will continue talking about it in the same thread. If they want to discuss about an unrelated or new defect (even in the same classes) they would open a new thread. The number of threads, then, could be a popularity metric whose value is related to the number of defects. After extracting e-mails from mailing lists, our e-mail model also contains the information about threads. Once the related mails are available in the object-oriented model, we retrieve this thread

information from the messages related to each class and count the number of different threads. If two, or more, e-mails related to the same class are part of the same thread, they are counted as one.

POP-NOMT: Inspecting sample e-mails from the mailing lists which form our experiment, we noticed that short threads are often characteristic of "announcements" e-mails, simple e-mails about technical issues experimented by new users of the systems, or updates about the status of developers. We hypothesize that longer threads could be symptom of discussions about questions that raise the interest of the developers, such as those about defects, bugs or changes in the code. For each class in the source code, we consider the thread of all the referring mails, and we count the total number of mails in each thread. If a thread is composed by more than one e-mail, but only one is referring the class, we still count all the e-mails inside the thread, since it is possible that following e-mails reference the same class implicitly.

POP-NOA: A high number of authors talking about the same class suggests that it is subject to broad discussions. For example, a class frequently mentioned by different users can hide design flaws or stability problems. Also, a class discussed by many developers might be not well-defined, comprehensible, or correct, thus more defect prone. For each class, we count the number of authors that wrote in referring mails (*i.e.,* if the same author wrote two, or more, e-mails, we count only one).

Extracting Bug Information. To measure the correlation of metrics with software defects, and to perform defect prediction, we need defect information and we need to link it with source code entities, *i.e.,* we need to map each problem report to classes of the system that it affects. We link FAMIX classes with versioning system files, as we did to compute change metrics, and the files with bugs retrieved from a Bugzilla[2] or a Jira[3] repository. Figure 3 shows the bug linking process.

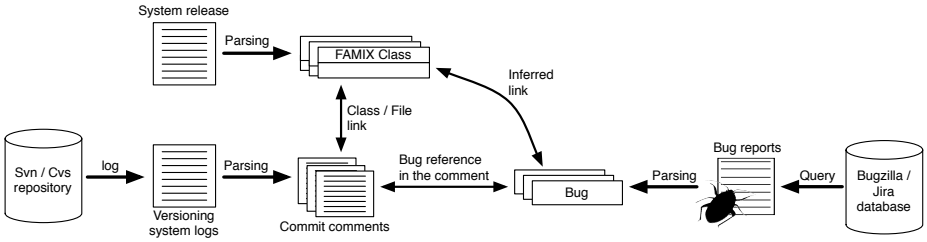

Fig. 3. Linking bugs, SCM files and classes.

A file version in the versioning system contains a developer comment written at commit time, which often includes a reference to a problem report (*e.g.,* "fixed bug 123"). Such references allow us to link problem reports with files in the versioning system, and therefore with source code artifacts, *e.g.,* classes. However, the link between

[2] `http://www.bugzilla.org`

[3] `http://www.atlassian.com/software/jira/`

a CVS/SVN file and a Bugzilla/Jira problem report is not formally defined, and to find a reference to the problem report id we use pattern matching techniques on the developer comments, a widely adopted technique [18,10]. Once we have established the link between a problem report and a file version, we verify that the bug has been reported before the commit time of the file version.

To measure the correlation between metrics and defects we consider all the defects, while for bug prediction only post-release defects, *i.e.,* the ones reported within a six months time interval after the considered release of the software system[4].

The output of the bug linking process is, for each class of the considered release, the total number of defects and the number of post-release defects.

3 Experiments

We conducted our experiment on the software systems depicted in Table 4.

System URL	Description	Classes	Mailing lists			Bug Tracking Systems	
			Creation	E-Mails Total	E-Mails Linked	Time period	Number of defects
Equinox eclipse.org/equinox	Plugin system for the Eclipse project	439	Feb 2003	5,575	2,383	Feb 2003 - Jun 2008	1,554
Jackrabbit jackrabbit.apache.org	Implementation of the Content Repository for Java Technology API (JCR)	1,913	Sep 2004	11,901	3,358	Sep 2004 - Dec 2008	674
						Sep 2004 - Aug 2009	975
Lucene lucene.apache.org	Text search engine library	1,279	Sep 2001	17,537	8,800	Oct 2001 - May 2008	751
						Oct 2001 - Sep 2009	1,274
Maven maven.apache.org	Tool for build automation and management of Java projects	301	Nov 2002	65,601	4,616	Apr 2004 - Sep 2008	507
						Apr 2004 - Aug 2009	616

Table 4. Dataset

We considered systems that deal with different domains and have distinct characteristics (*e.g.,* popularity, number of classes, e-mails, and defects) to mitigate some of the threats to external validity. These systems are stable projects, under active development, and have a history with several major releases. All are written in Java to ensure that all the code metrics are defined identically for each system. By using the same parser, we can avoid issues due to behavior differences in parsing, a known issue for reverse engineering tools [19].

Public development mailing lists used to discuss technical issues are available for all the systems, and are separated from lists specifically thought for system user issues. We consider e-mails starting from the creation of each mailing list until September 2009. Messages automatically generated by bug tracking and revision control systems are filtered out, and we report the resulting number of e-mails and the number of those referring to classes according to our linking techniques. All systems have public bug tracking systems, that were usually created along with the mailing lists.

[4] Six months for post release defects was also used by Zimmermann *et al.* [10].

3.1 Correlations Analysis

To answer the research question Q1 "*Does the popularity of software components correlate with software defects?*", we compute the correlation between class level popularity metrics and the number of defects per class. We compute the correlation in terms of both the *Pearson's* and the *Spearman's* correlation coefficient (r_{prs} and r_{spm}, respectively). The Spearman's rank correlation test is a non-parametric test that uses ranks of sample data consisting of matched pairs. The correlation coefficient varies from 1, *i.e.,* ranks are identical, to -1, *i.e.,* ranks are the opposite, where 0 indicates no correlation. Contrarily to Pearson's correlation, Spearman's one is less sensitive to bias due to outliers and does not require data to be metrically scaled or of normality assumptions [20]. Including the Pearson's correlation coefficient augment the understanding about the results: If r_{spm} is higher than r_{prs}, we might conclude that the variables are consistently correlated, but not in a linear fashion. If the two coefficients are very similar and different from zero, there is indication of a linear relationship. Finally, if the r_{prs} value is significantly higher than r_{spm}, we can deduce that there are outliers inside the dataset. This information first helps us to discover threats to construct validity, then put in evidence single elements that are heavily related. For example, a high r_{prs} can indicate that, among the classes with the highest number of bugs, we can find also the classes with the highest number of related e-mails.

We compute the correlation between class level source code metrics and number of defects per class, in order to compare the correlation to a broadly used baseline. For lack of space we only show the correlation for the source code metric LOC, as previous research showed that it is one of the best metrics for defect prediction [4,21,22,23]. Table 5 shows the correlation coefficients between the different popularity metrics and the number of bugs of each system.

System	POP-NOM		POP-NOCM		POP-NOT		POP-NOTM		POP-NOA		LOC	
	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}
Equinox	.52	.51	.52	.42	**.53**	**.54**	.52	.48	.53	.50	.73	.80
Jackrabbit	.23	.35	.22	.36	**.24**	**.36**	.23	-.02	.23	.34	.27	.54
Lucene	.41	.63	.38	.57	.41	.57	**.42**	**.68**	.41	.54	.17	.38
Maven	.44	.81	.39	.78	**.46**	.78	.44	**.81**	.45	.78	.55	.78

Table 5. Correlation coefficients.

We put in bold the highest values achived for both r_{spm} and r_{prs}, by system. Results provides evidence that the two metrics are rank correlated, and correlations over 0.4 are considered to be strong in fault prediction studies [24]. The Spearman correlation coefficients in our study exceed this value for three systems, *i.e.,* Equinox, Lucene, and Maven. In the case of Jackrabbit, the maximum coefficient is 0.24, which is similar to value reached using LOC. The best performing popularity metric depends on the software system: for example in Lucene, POP-NOTM, which counts the length of threads containing e-mails about the classes, is the best choice, while POP-NOT, number of threads containing at least one e-mail about the classes, is the best performing for other systems.

3.2 Defect Prediction

To answer the research question Q2 "*Is a regression model based on the popularity metrics a good predictor for software defects?*" , we create and evaluate regression models in which the independent variables are the class level popularity metrics, while the dependent variable is the number of post-release defects per class. We create regression models based on source code metrics and change metrics alone, as well as models in which these metrics are enriched with popularity metrics, where the dependent variable is always the number of post-release defects per class. We then compare the prediction performances of such models to answer research question Q3 "*Does the addition of popularity metrics improve the prediction performance of existing defect prediction techniques?*" We follow the methodology proposed by Nagappan *et al.* [5] and also used by Zimmermann *et al.* [24], consisting of: Principal component analysis, building regression models, evaluating explanative power and evaluating prediction power.

Principal Component Analysis is a standard statistical technique to avoid the problem of multicollinearity among the independent variables. This problem comes from intercorrelations amongst these variables and can lead to an inflated variance in the estimation of the dependent variable. We do not build the regression models using the actual variables as independent variables, but instead we use sets of principal components (PC). PC are independent and do not suffer from multicollinearity, while at the same time they account for as much sample variance as possible. We select sets of PC that account for a cumulative sample variance of at least 95%.

Building Regression Models. To evaluate the predictive power of the regression models we do cross-validation: We use 90% of the dataset, *i.e.,* 90% of the classes (training set), to build the prediction model, and the remaining 10% of the dataset (validation set) to evaluate the efficacy of the built model. For each model we perform 50 "folds", *i.e.,* we create 50 random 90%-10% splits of the data.

Evaluating Explanative Power. To evaluate the explanative power of the regression models we use the adjusted R^2 coefficient. The (non-adjusted) R^2 is the ratio of the regression sum of squares to the total sum of squares. R^2 ranges from 0 to 1, and the higher the value is, the more variability is explained by the model, *i.e.,* the better the explanative power of the model is. The adjusted R^2, takes into account the degrees of freedom of the independent variables and the sample population. As a consequence, it is consistenly lower than R^2. When reporting results, we only mention the adjusted R^2. We test the statistical significance of the regression models using the F-test. All our regression models are significant at the 99% level ($p < 0.01$).

Evaluating Prediction Power. To evaluate the predictive power of the regression models, we compute Spearman's correlation between the predicted number of post-release defects and the actual number. Such evaluation approach has been broadly used to assess the predictive power of a number of predictors [21,22,23]. In the cross-validation, for each random split, we use the training set (90% of the dataset) to build the regression model, and then we apply the obtained model on the validation set (10% of the dataset), producing for each class the predicted number of post-release defects. Then, to evaluate the performance of the performed prediction, we compute Spearman's correlation, on the validation set, between the lists of classes ranked according to the predicted and actual number of post-release defects. Since we perform 50 folds cross-

validation, the final values of the Spearman's correlation and adjusted R^2 are averages over 50 folds.

Results. Table 6 displays the results we obtained for the defect prediction, considering both R^2 adjusted values and Spearman's correlation coefficients.

Metrics	R^2adj					$R_{spearman}$				
	Equinox	**Jackrabbit**	**Lucene**	**Maven**	***Avg***	**Equinox**	**Jackrabbit**	**Lucene**	**Maven**	***Avg***
All Popularity Metrics	.23	.00	.31	.55	*.27*	.43	.04	.27	.52	*.32*
All Change Metrics	.55	.06	.43	.71	*.44*	.54	.30	.36	.62	*.45*
All C.M. + POP-NOM	.56	.06	.43	.71	*.44*	.53	**.32**	.38	**.69**	***.48***
All C.M. + POP-NOCM	.58	.06	.43	.70	*.44*	.57	.31	**.43**	.60	*.48*
All C.M. + POP-NOT	.56	.06	.43	.71	*.44*	.54	.31	.39	.59	*.46*
All C.M. + POP-NOMT	.56	.06	.43	.70	*.44*	.53	.29	.41	.60	*.46*
All C.M. + POP-NOA	.56	.06	.43	.70	*.44*	**.58**	.29	.37	.43	*.42*
All C.M. + All POP	**.61**	**.06**	**.45**	**.71**	***.46***	.52	.30	.38	.43	*.41*
Improvement	*11%*	*0%*	*+5%*	*0%*	*+4%*	*+7%*	*+7%*	*+19%*	*+11%*	*+11%*
Source Code Metrics	.61	.03	.27	.42	*.33*	.51	.17	.31	.52	*.38*
S.C.M. + POP-NOM	.62	.03	.33	.59	*.39*	.53	.14	.35	.52	*.38*
S.C.M. + POP-NOCM	.62	**.04**	.32	.56	*.38*	.51	.15	.36	.60	*.41*
S.C.M. + POP-NOT	.61	.03	.31	.57	*.38*	.49	.15	**.38**	.52	*.38*
S.C.M. + POP-NOMT	.62	.03	.35	.60	*.40*	.55	.14	.33	.43	*.36*
S.C.M. + POP-NOA	.61	.04	.30	.56	*.38*	.53	.12	.38	**.70**	***.43***
S.C.M. + All POP	**.62**	.03	**.37**	**.61**	***.41***	**.58**	.14	.32	.52	*.39*
Improvement	*+2%*	*+25%*	*+37%*	*+45%*	*+27%*	*+14%*	*-12%*	*+23%*	*+35%*	*+15%*
CK Metrics	.54	.01	.39	.28	*.31*	.51	.13	.36	.60	*.40*
CK + POP-NOM	.56	.02	.40	.54	*.38*	.48	.13	.35	**.69**	***.41***
CK + POP-NOCM	.57	.02	.40	.50	*.37*	.50	**.17**	.33	.42	*.35*
CK + POP-NOT	.56	.01	.40	.51	*.37*	**.53**	.13	.34	.52	*.38*
CK + POP-NOMT	.57	.01	.40	.56	*.39*	.52	.14	.25	.49	*.35*
CK + POP-NOA	.56	.02	.40	.51	*.37*	.52	.14	**.41**	.53	*.40*
CK + All POP	**.57**	**.02**	**.42**	**.58**	***.40***	.51	.16	.30	.52	*.37*
Improvement	*+6%*	*+50%*	*+8%*	*+107%*	*+43%*	*+4%*	*+31%*	*+14%*	*+15%*	*+16%*
All Source Code Metrics	.66	.04	.44	.45	*.40*	.48	.15	.35	.36	*.33*
All S.C.M. + POP-NOM	.67	.04	.45	.60	*.44*	**.59**	.15	.34	**.62**	***.43***
All S.C.M. + POP-NOCM	.66	.04	.45	.56	*.43*	.51	.16	.30	.31	*.32*
All S.C.M. + POP-NOT	.66	.04	.44	.57	*.43*	.50	.14	**.35**	.52	*.38*
All S.C.M. + POP-NOMT	.67	.04	.44	.62	*.44*	.53	.14	.35	.34	*.34*
All S.C.M. + POP-NOA	.66	.04	.44	.57	*.43*	.51	.15	.34	.43	*.36*
All S.C.M. + All POP	**.67**	.04	**.46**	**.63**	***.45***	.51	**.16**	.33	.52	*.38*
Improvement	*+2%*	*0%*	*+5%*	*+40%*	*+12%*	*+23%*	*+7%*	*+0%*	*+72%*	*+26%*

Table 6. Defect prediction results

The first row shows the results achieved using all the popularity metrics defined in Section 2. In the following four blocks, we report the prediction results obtained through the source code and change metrics, first alone, then by incorporating each single popularity metric, and finally incorporating all the popularity metrics. For each system and block of metrics, when popularity metrics augment the results of other metrics, we put in bold the highest value reached.

Analyzing the results of the sole popularity metrics, we notice that, in terms of correlation, Equinox and Maven still present a strong correlation, *i.e.,* higher than .40, while Lucene is less correlated. The popularity metrics alone are not sufficient for per-

forming predictions in the Jackrabbit system. Looking at the results obtained by using other metrics, we first note that Jackrabbit's results are much lower if compared to those reached in other systems, especially for the R^2, and partly for the R_{spm}. Only the R_{spm} reached with change metrics reach good results in this system.

Going back to the other systems, the R^2 adjusted values are always increased and the best results are achieved when using all popularity metrics together. The increase with respect to the other metrics varies from 2%, when other metrics already reach high values, up to 107%. Spearman's coefficients also increase by using the information given by popularity metrics: Their values augment, on average, more than fifteen percent. However, there is not a single popularity metric that outperforms the others, and their union does not give the best results.

4 Discussion

Popularity of software components do correlate with software defects. Three software systems out of four show a strong rank correlation, *i.e.,* coefficients ranging from .42 to .53, between defects of software components and their popularity in e-mail discussions. Only Jackrabbit is less rank correlated with a coefficient of .23.

Popularity can predict software defects, but without major improvements over previously established techniques. In the second part of our results, consistently with the correlation analysis, the quality of predictions done by Jackrabbit using popularity metrics are extremely low, both for the R^2 adjusted values and for the Spearman's correlation coefficients. On the contrary, our popularity metrics applied to the other three systems lead to different results: Popularity metrics are able to predict defects. However, if used alone, they do not compete with the results obtained through other metrics. The best average results are shown by the Change Metrics, corroborating previous works stating the quality of such predictors [7,8].

Popularity metrics do improve prediction performances of existing defect prediction techniques. The strongest results are obtained integrating the popularity information into other techniques. This creates more reliable and complete predictors that significantly increase the overall results: The improvements on correlation coefficients are, on average, more than fifteen percent higher, with peaks over 30% and reaching the top value of 72%, to those obtained without popularity metrics. This corroborate our initial assumption that popularity metrics measure an aspect of the development process that is different from those captured by other techniques.

Results put in evidence that, given the considerable difference of the prediction performance across different software projects, bug prediction techniques that exploit popularity metrics should not be applied in a "black box" way. As suggested by Nagappan *et al.* [5], the prediction approach should be first validated on the history of a software project, to see which metrics work best for predictions for the system.

5 Threats to validity

Threats to construct validity regard the relationship between theory and observation, *i.e.,* measured variables may not actually measure conceptual variables. A first construct validity threat concerns the way we link bugs with versioning system files and subsequently with classes. In fact, the pattern matching technique we use to detect bug references in commit comments does not guarantee that all the links are retrieved. We also made the assumption that commit comments do contain bug fixing information, which limits the application of our bug linking approach only to software projects where this convention is used. Finally, a commit that is referring to a bug can also contain modifications to files that are unrelated to the bug. However, this technique represents the current state of the art in linking bugs to versioning system files and is widely used in literature [18].

Another threat concerns the procedure for linking e-mails to discussed classes. We use linking techniques whose effectiveness was measured [14], and it is known that they cannot produce a perfect linking. The enriched object-oriented model can contain wrongly reported links or miss connections that are present. We alleviated this problem manually inspecting all the classes that showed an uncommon number of links, *i.e.,* outliers, and, whenever necessary, adjusted the regular expressions composing the linking techniques to correctly handle such unexpected situations.

Threats to statistical conclusion validity concern the relationship between the treatment and the outcome. In our experiments all the Spearman correlation coefficients and all the regression models were significant at the 99% level.

Threats to external validity concern the generalization of the findings. In our approach we analyze only open-source software projects, however the development in industrial environment may differ and conduct to different comportments in the developers, thus to different results. Another external validity threat concerns the language: all the software systems are developed in Java. Although this alleviates parsing bias, communities using other languages could have different developer cultures and the style of e-mails can vary. To obtain a better generalization of the results, in our future work, we plan to apply our approach to industrial systems and other object-oriented languages.

6 Related work

6.1 Mining Data From E-Mail Archives

Li *et al.* first introduced the idea of using the information stored in the mailing lists as an additional predictor for finding defects in software systems [25]. They conducted a case study on a single software system, used a number of previously known predictors and defined new mailing list predictors. Mainly such predictors counted the number of messages to different mailing lists during the development of software releases. One predictor *TechMailing*, based on number of messages to the technical mailing list during development, was evaluated to be the most highly rank correlated predictor with the number of defects, among all the predictors evaluated. Our works differs in genre and granularity of defects we predict: We concentrate on defects on small source code units that can be easily reviewed, analyzed, and improved. Also Li *et al.* did not remove the

noise from the mailing lists, focusing only on source code related messages. Pattison *et al.* were the first to introduce the idea of studying software entity (function, class etc.) names in emails [13]. They used a linking based on simple name matching, and found a high correlation between the amount of discussions about entities and the number of changes in the source code. However, Pattison *et al.* did not validate the quality of their links between e-mails and source code. To our knowledge, our work [14] was the first to measure the effectiveness of linking techniques for e-mails and source code. This research is the first work that uses information from development mailing lists at class granularity to predict and to find correlation with source code defects. Other works also analyzed development mailing lists but extracting a different kind of information: social structures [26], developers participation [27] and inter-projects migration [28], and emotional content [29].

6.2 Defect Prediction

Change Log Approaches use information extracted from the versioning system to perform defect prediction. They are based on the heuristic that recently or frequently changed files are the most probable source of future bugs. Indeed, it is only by changing the behavior of the program that one can introduce new defects. Nagappan and Ball performed a study on the influence of code churn (*i.e.,* the amount of change to the sytem) on the defect density in Windows Server 2003 [6]. Hassan introduced a measure of the complexity of code changes [30] and used it as defect predictor on 6 open-source systems. Moser *et al.* used a set of change metrics to predict the presence/absence of bugs in files of Eclipse [7]. Ostrand *et al.* predict faults on two industrial systems, using change and previous defect data [9]. The approach by Bernstein *et al.* uses bug and change information in non-linear prediction models [8].

Single-version approaches employ the heuristic that the current design and behavior of the program influence the presence of future defects, assuming that changing a part of the program that is hard to understand is inherently more risky than changing a part with a simpler design. Basili *et al.* used the CK metrics on 8 medium-sized information management systems [1]. Ohlsson *et al.* used several graph metrics including the McCabe cyclomatic complexity on a telecom system [2]. Subramanyam *et al.* used the CK metrics on a commercial C++/Java case study [3], while Gyimothy *et al.* performed a similar analysis on Mozilla [4]. Nagappan *et al.* used a catalog of source code metrics to predict post release defects at the module level on five Microsoft systems, and found that it was possible to build predictors for one individual project, but that no predictor would perform well on all the projects [5]. Zimmermann *et al.* applied a number of code metrics on the Eclipse IDE [10].

Other Approaches. Zimmermann and Nagappan used dependencies between binaries to predict defect [24]. Marcus *et al.* used a cohesion measurement based on LSI for defect prediction on C++ systems [31]. Neuhaus *et al.* used a variety of features of Mozilla to detect vulnerabilities, a subset of bugs with security risks [32].

The main difference between our work and the mentioned approaches is that our approach is the first one which exploits e-mail archives data for defect prediction.

7 Conclusion

We have presented a novel approach to correlate popularity of source code artifacts within e-mail archives to software defects. We also investigated whether such metrics could be used to predict post-release defects. We showed that, while there is a significant correlation, popularity metrics by themselves do not outperform source code and change metrics in terms of prediction power. However, we demonstrated that, in conjunction with source code and change metrics, popularity metrics increase both the explanative and predictive power of existing defect prediction techniques.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA" (SNF Project No. 118063).

References

1. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering **22**(10) (1996) 751–761
2. Ohlsson, N., Alberg, H.: Predicting fault-prone software modules in telephone switches. IEEE Transactions on Software Engineering **22**(12) (1996) 886–894
3. Subramanyam, R., Krishnan, M.S.: Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Transactions on Software Engineering **29**(4) (2003) 297–310
4. Gyimóthy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software Engineering **31**(10) (2005) 897–910
5. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the ICSE 2006 (28th International Conference on Software Engineering), ACM (2006) 452–461
6. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of ICSE 2005 (27th International Conference on Software Engineering), ACM (2005) 284–292
7. Moser, R., Pedrycz, W., Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of ICSE 2008 (30th International Conference on Software Engineering). (2008) 181–190
8. Bernstein, A., Ekanayake, J., Pinzger, M.: Improving defect prediction using temporal features and non linear models. In: Proceedings of the International Workshop on Principles of Software Evolution, Dubrovnik, Croatia, IEEE CS Press (2007) 11–18
9. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering **31**(4) (2005) 340–355
10. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Proceedings of PROMISE 2007, IEEE CS (2007) 76
11. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E.: Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering **28**(10) (2002) 970–983
12. Lucia, A.D., Fasano, F., Grieco, C., Tortora, G.: Recovering design rationale from email repositories. In: Proceedings of ICSM 2009 (25th IEEE International Conference on Software Maintenance), IEEE CS Press (2009)
13. Pattison, D., Bird, C., Devanbu, P.: Talk and Work: a Preliminary Report. In: Proceedings of the Fifth International Working Conference on Mining Software Repositories, ACM (2008) 113–116

14. Bacchelli, A., D'Ambros, M., Lanza, M., Robbes, R.: Benchmarking lightweight techniques to link e-mails and source code. In: Proceedings of WCRE 2009 (16th IEEE Working Conference on Reverse Engineering), IEEE CS Press (2009) xxx – xxx
15. Demeyer, S., Tichelaar, S., Ducasse, S.: FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern (2001)
16. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Software Eng. **20**(6) (1994) 476–493
17. Emam, K.E., Melo, W., Machado, J.C.: The prediction of faulty classes using object-oriented design metrics. Journal of Systems and Software **56**(1) (2001) 63–75
18. Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE CS Press (2003) 23–32
19. Kollmann, R., Selonen, P., Stroulia, E.: A study on the current state of the art in toolsupported uml-based static reverse engineering. In: Proceedings of the Ninth Working Conference on Reverse Engineering. (2002) 22–32
20. Triola, M.: Elementary Statistics. 10th edn. Addison-Wesley (2006)
21. Ostrand, T.J., Weyuker, E.J.: The distribution of faults in a large industrial software system. SIGSOFT Software Engineering Notes **27**(4) (2002) 55–64
22. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Where the bugs are. In: Proceedings of ISSTA 2004 (ACM SIGSOFT International Symposium on Software testing and analysis), ACM (2004) 86–96
23. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Automating algorithms for the identification of fault-prone files. In: Proceedings of ISSTA 2007 (ACM SIGSOFT International Symposium on Software testing and analysis, ACM (2007) 219–227
24. Zimmermann, T., Nagappan, N.: Predicting defects using network analysis on dependency graphs. In: Proceedings of ICSE 2008 (30th International Conference on Software Engineering). (2008)
25. Li, P.L., Herbsleb, J., Shaw, M.: Finding predictors of field defects for open source software systems in commonly available data sources: A case study of openbsd. In: METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium, IEEE Computer Society (2005) 32
26. Bird, C., Gourley, A., Devanbu, P., Gertz, M., Swaminathan, A.: Mining Email Social Networks. In: Proceedings of MSR 2006 (3rd International Workshop on Mining software repositories), ACM (2006) 137–143
27. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology **11**(3) (2002) 309–346
28. Bird, C., Gourley, A., Devanbu, P., Swaminathan, A., Hsu, G.: Open Borders? Immigration in Open Source Projects. In: Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories), IEEE Computer Society (2007) 6
29. Rigby, P.C., Hassan, A.E.: What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list. In: Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories), IEEE Computer Society (2007) 23
30. Hassan, A.E.: Predicting faults using the complexity of code changes. In: Proceedings of ICSE 2009 (31st International Conference on Software Engineering). (2009) 78–88
31. Marcus, A., Poshyvanyk, D., Ferenc, R.: Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Transactions on Software Engineering **34**(2) (2008) 287–300
32. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of CCS 2007 (14th ACM Conference on Computer and Communications Security), ACM (2007) 529–540