An Analysis of Developer Metrics for Fault Prediction

Shinsuke Matsumoto
Graduate School of Engineering,
Kobe University
Kobe, Japan
shinsuke@cs.kobe-u.ac.jp

Yasutaka Kamei
School of Computing
Queen's University
Kingston, Canada
kamei@cs.queensu.ca

Akito Monden
Graduate School of Information Science,
Nara Institute of Science and Technology
Nara, Japan
akito-m@is.naist.jp

Ken-ichi Matsumoto
Graduate School of Information Science,
Nara Institute of Science and Technology
Nara, Japan
matumoto@is.naist.jp

Masahide Nakamura
Graduate School of Engineering,
Kobe University
Kobe, Japan
masa-n@cs.kobe-u.ac.jp

ABSTRACT

Background: Software product metrics have been widely used as independent variables for constructing a fault prediction model. However, fault injection depends not only on characteristics of the products themselves, but also on characteristics of developers involved in the project. **Aims**: The goal of this paper is to study the effects of developer features on software reliability. **Method**: This paper proposes developer metrics such as the number of code churns made by each developer, the number of commitments made by each developer and the number of developers for each module. By using the eclipse project dataset, we experimentally analyzed the relationship between the number of faults and developer metrics. Second, the effective of developer metrics for performance improvements of fault prediction models were evaluated. **Results**: The result revealed that the modules touched by more developer contained more faults. Compared with conventional fault prediction models, developer metrics improved the prediction performance.**Conclusions**: We conclude that developer metrics are good predictor of faults and we must consider the human factors for improving the software reliability.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product metrics, Process metrics*; D.4.8 [**Performance**]: Measurements, Modeling and Prediction

General Terms

Measurement, Human Factors, Experimentation,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PROMISE2010, Sep 12-13, 2010. Timisoara, Romania
Copyright 2010 ACM ISBN 978-1-4503-0404-7...$10.00.

Keywords

Developer metrics, fault prediction, human factor...

1. INTRODUCTION

Over decades, many metrics for software products (i.e., *product metrics*) have been proposed for software quality management. Typical product metrics include, source lines of code (SLOC), McCabe's cyclomatic complexity and object-oriented metrics [3]. They are widely used for characterizing some static aspects of source code. It is known that the SLOC and McCabe's complexity have a positive correlation with the number of faults in the source code [15, 17]. These metrics are often used for software quality improvement activities such as test effort allocation. More and more product metrics are proposed in recent years. The change metrics [16] are calculated from the change history of the source code. The network structure metrics [27] are derived from dependencies among modules.

We have investigated many open source projects to see how and why faults were injected. There, we observed that "*software reliability depends not only on characteristics of the products themselves, but also on characteristics of developers involved in the project*". For example, inexperienced developers are likely to inject more faults than experts. Or, a module modified by many developers tends to contain faults, since different thoughts and coding styles are mixed within the module. These observations are not necessarily captured by the SLOC and McCabe's complexity, which motivated us to consider "developer metrics" for fault prediction.

Many studies have been reported on relationships between developer skills and coding speed or debugging efficiency [20, 6]. There is also a system that measures developer's activities[12, 26]. On the other hand, there exist criticisms for "human measurement" [1, 4]. Austin [1] pointed out that "measuring developers' activities can lead to developer evaluation unintentionally, so the measurement and the analysis based on the measurement should be avoided". In recent huge and complex software development, however, it is also

true that that assuring high reliability with low cost and short time is strongly required. Therefore, we believe it essential to understand human factors declining reliability, based on correct and fair analysis on empirical data.

In this paper, we therefore conduct an empirical study to analyze relationships between faults in source code and some developer metrics that can be derived from a software repository. The developer metrics include; $NoDRM(m)$ – the number of developers revising a module m, and $LoCRD(m, d)$ – the lines of code revised in a module m that are modified by a developer d. The empirical analysis is performed based on the following four hypotheses:

- H_0: The average number of faults injected per commitment (ANFIC) varies among different individual developers.
- H_1: Providing that H_0 is supported, the ANFIC can be characterized by human factors quantified by some developers metrics.
- H_2: A module modified by many developers tends to contain more faults.
- H_3: Developer metrics are good predictor of fault prediction.

In the experiment, we have used three datasets derived from software repositories of Eclipse Platform Project [5]. The data sets contain more than 5,000 modules (Java classes). More than 60 people are participated in the project.

2. PRELIMINARIES

2.1 Software Development Activity

We start with the model of software development assumed in this paper. A *software application* consists of *packages*, each of which contains *modules*. In the development, the modules and the packages are created or updated by multiple developers. We suppose that a certain software repository with the version control system (e.g., cvs or svn) is deployed for the project, so that the multiple developers can create, delete and update the modules concurrently.

When a developer changes some modules, the developer *commits* the changes using the version control system. Thus, the application is revised and evolved by a sequence of commitments. When an iteration of revisions is finished, the application is *released* with a *version number*. Usually, the revision continues after the release, in order to develop a new version of the application.

2.2 Hypotheses

Several studies have been reported on individual difference in coding speed [20] and debugging efficiency [6]. Analogous to these, we believe that there must exist individual difference in *injecting faults*. To validate the belief empirically, the hypothesis H_0 is presented.

H_0: *The average number of faults injected per commitment (ANFIC) varies among different individual developers.*

The metric ANFIC is supposed to characterize *likelihood* that a developer injects faults in the product. A *commitment* for the version management system is often used in analysis of software evolution [9] and developer activities [10], since it well captures a unit of cohesive activities of a developer creating or modifying software products. Counting the average number of faults injected per a commitment can be an indicator of the fault injection of a developer.

If H_0 holds, our interest is then to check if the ANFIC can be *predicted* by some developer metrics recorded within or derived from development activities.

H_1: *Providing that H_0 is supported, ANFIC can be characterized by human factors quantified by some developers metrics.*

Generally, it is hard during the development to obtain the exact value of ANFIC. If this hypothesis is supported, ANFIC can be estimated by alternative metrics that are easier to measure.

The next hypothesis was derived from our observation that modules revised by many developers often contained different design thoughts and coding styles By contrast, a module created by a single developer tends to contain little faults, because the module is maintained by the developer only.

H_2: *A module modified by many developers tends to contain more faults.*

Some researchers have empirically shown that source code metrics such as SLOC and McCabe's cyclomatic complexity have a positive correlation with the number of faults [13, 18]. This fact is often used for the fault prediction, where a prediction model is constructed from large datasets measured in the previous projects [11, 18]. However, there are little reports that explicitly take developer metrics for the fault prediction. The final hypothesis below is to validate that introducing developer metrics can add more confidence to the existing fault prediction model.

H_3: *Developer metrics are good predictor of fault prediction.*

3. PROPOSED DEVELOPER METRICS

3.1 Overview

In this paper, we use the term "developer metrics" to refer to any metrics characterizing software development activities performed by developers. Although a variety of metrics can be considered, in this paper we especially focus on the following two types of developers metrics:

(Type 1) Developer metrics characterizing developer's activities.

(Type 2) Developer metrics characterizing modules revised by developers.

We use (Type 1) metrics to validate Hypotheses H_0 and H_1, since the hypotheses focus on the individual developers. On the other hand, (Type 2) metrics are used for Hypotheses H_2 and H_3 as they are interested in individual modules.

3.2 (Type 1) Characterizing Developer's Activity

To validate hypotheses H_0 and H_1, we propose (Type 1) metrics in this section. The metrics aim to characterize activities of individual developer in the software development.

Definition 1. Personal Commit Sequence
Let d be a developer, and App_v be v-th release version of an application. Let $seq = [c_1, c_2, ..., c_N]$ be a sequence of commitments to develop App_v. A personal commit sequence of d for App_v is defined by:

$$PCseq(d, App_v) = [c_1^d, c_2^d, ..., c_n^d]$$

where $PCseq(d, App_v)$ is obtained from seq by choosing only the commitments c_i's relevant with d.

$PCseq(d, App_v)$ is a projection of the sequence of commitments by all users onto a certain developer d, which is a primary attribute to characterize d's development activities. Using this, we define the following four metrics, NoC, NoLR, NoUMR, NoUPR, each of which characterizes certain activities of a developer d.

Definition 2. Number of Commitments
Let $PCseq(d, App_v) = [c_1^d, c_2^d, ..., c_n^d]$ be d's personal commit sequence. Then the number of commitments by d for App_v, denoted by $NoC(d, App_v)$, is defined by:

$$NoC(d, App_v) = n$$

Definition 3. Number of Lines Revised
Let $PCseq(d, App_v) = [c_1^d, c_2^d, ..., c_n^d]$ be d's personal commit sequence. For each c_i^d, let $line(c_i^d)$ represent the number of lines revised by d in the commitment c_i^d. Then the total number of lines revised by d for App_v is defined by:

$$NoLR(d, App_v) = \Sigma_i \ line(c_i^d)$$

Definition 4. Number of Unique Modules Revised
Let $PCseq(d, App_v) = [c_1^d, c_2^d, ..., c_n^d]$ be d's personal commit sequence. For each c_i^d, let $Mod(c_i^d)$ be a set of modules revised by d in the commitment c_i^d. Then the number of unique modules revised by d for App_v is defined by:

$$NoUMR(d, App_v) = | \cup_i \ Mod(c_i^d)|$$

Definition 5. Number of Unique Packages Revised
Let $PCseq(d, App_v) = [c_1^d, c_2^d, ..., c_n^d]$ be d's personal commit sequence. For each c_i^d, let $Pkg(c_i^d)$ be a set of packages revised by d in the commitment c_i^d. Then the number of unique packages revised by d for App_v is defined by:

$$NoUPR(d, App_v) = | \cup_i \ Pkg(c_i^d)|$$

All of the above metrics are derived from d's personal commit sequence, which reflects how d works in the development. First, $NoC(d, App_v)$ represents how *often* d revised the application, which may characterize d's enthusiasm to the development. However, frequent commitments don't necessarily come to the large development effort. So $NoLR(d, App_v)$ captures the effort in term of *lines* that are actually contributed by d. Next, $NoUMR(d, App_v)$ can measure the scope of responsibility (or interest) maintained by d. Finally, $NoUPR(d, App_v)$ also captures the scope of responsibility with coarser granularity.

Now, we give a definition of ANFIC which is our primary concern in hypotheses H_0 and H_1.

Definition 6. Average Number of Faults Injected By Commit
Let $PCseq(d, App_v) = [c_1^d, c_2^d, ..., c_n^d]$ be d's personal commit sequence. For each c_i^d, let $bug(c_i^d)$ be the number of faults injected by d in the commitment c_i^d. Then the average number of faults injected per commit by d for App_v is defined by:

$$ANFIC(d, App_v) = \Sigma_i \ (bug(c_i^d))/n$$

As $ANFIC(d, App_v)$ represents the likelihood of fault injection by d, it is useful for estimating defective modules. However, it is quite difficult (or almost impossible) during the development to calculate exact value of $ANFIC(d, App_v)$, since $bug(c_i^d)$ cannot be obtained instantly. Therefore, we try to estimate $ANFIC(d, App_v)$ by NoC, $NoLR$, $NoUMR$, $NoUPR$. Thus, the validation of hypotheses H_0 and H_1 is performed indirectly using the four metrics.

3.3 (Type 2) Characterizing Modules by Developers

To validate the hypotheses H_2 and H_3, we define (Type 2) metrics in this section. While (Type 1) characterizes the developers, (Type 2) measures the *modules* with attributes of developers. In this paper, we propose the following two metrics.

Definition 7. Number of Developers Revising Module
Let d be a developer, m be a module, App_v be v-th version of the application. Then, the number of developers revising module m of App_v, denoted by $NoDRM(m, App_v)$, is defined as the number of developers who revised m.

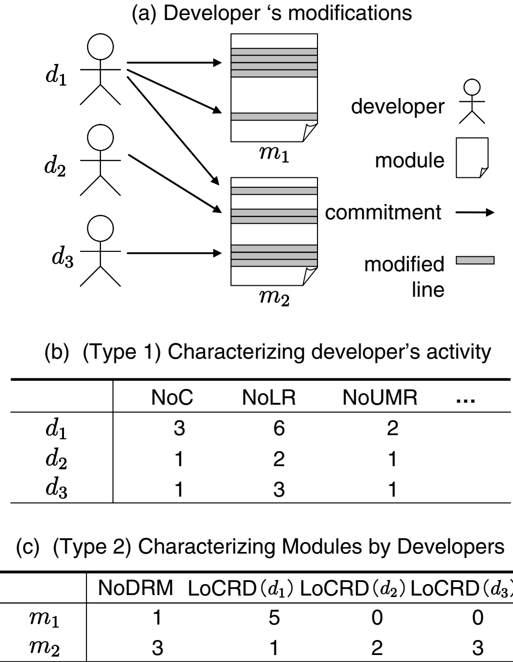
Definition 8. Lines of Code Revised by a Developer
Let $PCseq(d, App_v) = [c_1^d, c_2^d, ..., c_n^d]$ be d's personal commit sequence. Let $loc(m, c_i^d)$ be the lines of code of m revised by d in the commitment c_i^d. Then, the lines of code in m of App_v revised by d is defined by:

$$LoCRD(m, d, App_v) = \Sigma_i \ loc(m, c_i^d)$$

$NoDRM(m, App_v)$ represents how many developers revised a module. Therefore, it can be directly used for validating the hypothesis H_2. Using this together with $LoCRD(m, d, App_v)$, we try to validate the hypothesis H_3.

3.4 Example

We show an example of the proposed metrics using Figure 1. Figure 1 (a) depicts a software development process, where



(b) (Type 1) Characterizing developer's activity

	NoC	NoLR	NoUMR	...
d_1	3	6	2	
d_2	1	2	1	
d_3	1	3	1	

(c) (Type 2) Characterizing Modules by Developers

	NoDRM	LoCRD (d_1)	LoCRD (d_2)	LoCRD (d_3)
m_1	1	5	0	0
m_2	3	1	2	3

Figure 1: An example of developer metrics.

three developers d_1, d_2 and d_3 have modified two modules m_1 and m_2. Gray boxes in the module represent a line modified by a developer, and an arrow represents a commitment for the modified lines. Figure 1 (b) shows examples of (Type 1) metrics derived from Figure 1 (a). For each d_i of developer, we can count NoC, NoLR, NoUMR, etc. Figure 1 (c) shows examples of (Type 2) metrics. For each module, we can count NoDRM and LoCRD of each developer.

4. DESIGNING EXPERIMENT

4.1 Outline of Experiment

The objective of the experiment is to empirically validate the four hypothesis with datasets taken from a practical project. The experiment is performed by the following five steps.

Step 1 (Gathering Developer Information)
Mining the source code repository, we first collect information of developers from the change history of source code taken by the version control system.

Step 2 (Gathering Fault Information)
From the developer information and the bug tracking system, we collect information of all faults injected and corrected during the development.

Step 3 (Measuring Proposed Developer Metrics)
Using the developer information, we measure the proposed developers metrics.

Step 4 (Measuring Conventional Metrics)
To evaluate the effectiveness of the proposed metrics, we also measure the conventional static metrics [3] and change metric [16] from the source code repository.

Step 5 (Validating Hypotheses)
Using the fault information and all metrics, we validate the hypotheses by statistical analysis.

4.2 Step 1 (Gathering Developer Information)

We collected developer information from modification histories of a source code repository. In the modification history, who and why commit which lines of a module for all commit sequence (*seq*) were written. We create some scripts that parse the modification history for collecting developer information.

4.3 Step 2 (Gathering Fault Information)

From the bug tracking system, we collect information of all faults injected and corrected using SZZ algorithm[23]. The SZZ algorithm specifies when and who inject a fault to which module. Specifically, at first, the SZZ algorithm characterizes which lines in a module were modified when a fault was corrected within the module by bug modification histories of a bug tracking system. Next, which developer finally modified the lines was identified. Lastly, SZZ algorithm assumes the fault was injected by the developer.

4.4 Step 3 (Measuring Developer Metrics)

Using the developer information, we measure the proposed developers metrics. Each of collected developer metrics were defined in Section 3.

4.5 Step 4 (Measuring Conventional Metrics)

To evaluate the effectiveness of the proposed metrics, we also measure the conventional static metrics [3] and change metric [16] from the source code repository. These metrics were shown in Table 1. As static metrics, 15 well-known metrics were collected using Eclipse Metrics Plugin[1]. The change metrics contains seven metrics which collected from the change history of source code.

Table 1: Conventional metrics used in validating H_2 and H_3.

	Name	Definition
Static metrics	TLOC	Total lines of code.
	MLOC	Method lines of code.
	PAR	# of parameters
	NOF	# of attributes
	NOM	# of methods
	NORM	# of overridden methods
	NSC	# of children
	NSF	# of static attributes
	NSM	# of static methods
	NBD	Nested block depth
	VG	Cyclomatic complexity
	DIT	Depth of inheritance tree
	LCOM	Lack of cohesion
	WMC	Weighted methods per class
	SIX	Specialization index
Change metrics	Codechurn	Sum of modified lines (added lines + deleted lines)
	LOCAdded	total added lines
	LOCDeleted	total deleted lines
	Revisions	# of revisions
	Age	age of a file in weeks
	BugFixes	# of bug-fixing
	Refactorings	# of refactoring

4.6 Step 5 (Validating Hypotheses)

Using the fault information and all metrics, we validate the four hypotheses by following approach.

[1]http://eclipse-metrics.sourceforge.net

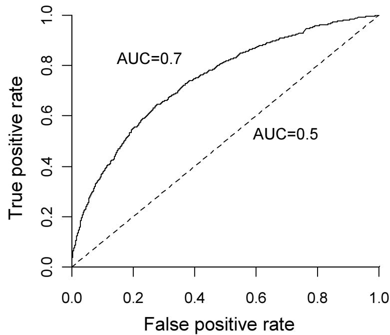


Figure 2: An example of ROC curve and its AUC value.

4.6.1 Validating Hypothesis H_0

We confirm a variety of ANFIC with only focus on some developers who has similar ANFIC in all three versions. If developer d committed a few times, ANFIC(d) sensitivity changes by small number of faults injected by d. To avoid misleading a validating H_0 by the sensitivity changing, we focus on the developers who has stable ANFIC values.

4.6.2 Validating Hypothesis H_1

When hypothesis H_0 is supported, we analyze the relationship between ANFIC and some developer activities. For the validating, we test single correlation coefficients between ANFIC and four (Type 1) developer metrics shown in section 3.2. In addition to four developer metrics, we use previous version of ANFIC ($ANFIC(App_{v-1})$ because $ANFIC$ (App_{v-1}) may already known at current version's development.

4.6.3 Validating Hypothesis H_2

We can analyze the effectiveness of each metric for number of faults by comparing standard partial coefficients which calculated by constructing a regression model. In this paper, we construct a regression model that using a number of faults as a dependent variable. Independent variables for the model construction include the NoDRM and two conventional line-oriented metrics (e.g., SLOC and Codechurn). By constructing the regression model with two types of line metrics, we can validate H_2 based on *fault density*. Fault density which means the number of faults normalized by SLOC is widely used as reliability criteria in general development process.

4.6.4 Validating Hypothesis H_3

We evaluate the prediction performance of seven combinations of three types of metrics (i.e., developer metrics, static metrics and change metrics). As model constriction methods, we used three well-known discriminant models which are linear regression model (LDA), logistic regression model (LRA) and classification tree (CT). Each of prediction models is constructed by metrics datasets of v-th version, and evaluated by metrics datasets of $(v+1)$-th version.

As a evaluation criteria, we use AUC (Area Under the Curve) value of ROC (Receiver Operating Characteristics) curve according to prediction framework proposed by Lessmann et al. [14]. ROC curve shows the relationship of false positive rate (x-axis) and true positive rate (y-axis). The AUC value of ROC is a widely used measure of performance of for prediction models.An example of AUC-ROC shows in Figure 2. The AUC-ROC value normalized into [0-1], and the value takes about 0.5 when evaluating a random prediction model.

Table 2: Summary of datasets.

	ver.3.00	ver.3.10	ver.3.20
Branch tag	R3_0	R3_1	R3_2
Created date	25-Jun-04	28-Jun-05	30-Jun-06
Total developers	69	66	72
Total modules	8,313	9,663	11,525
# of modified modules	7,080	9,428	8,950
# of faulty modules	2,986	3,302	2,506
# of commitments	61,366	53,302	45,441
# of bugs	6,351	7,667	4,772
% of bugs/commitments	10.7%	17.4%	14.5%

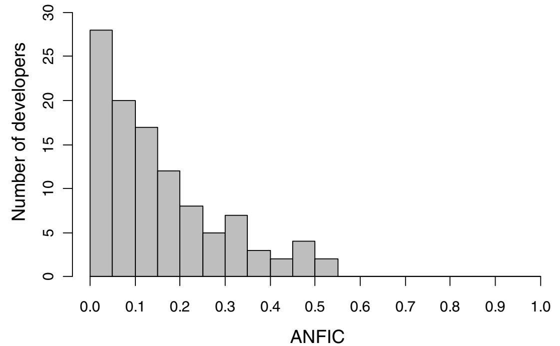


Figure 3: Histogram of ANFIC.

5. CASE STUDY

5.1 Eclipse Platform Project

As a case study, we used datasets from Eclipse Platform Project [5] since it is large and practical enough. Also, the project is well managed by a version control system and a bug tracking system, which allowed us to gather reliable empirical data. In the experiment, we investigated three versions (ver.3.00, ver.3.10 and ver.3.20) of Eclipse 3.

5.2 Dataset Summary

The summary of the collected datasets through step 1 to step 4 explained in Section 4 is shown in Table 2. In the experiment, we consider that a single Java class corresponds to a self-contained module.

5.3 Validation of Hypothesis H_0

H_0: *Average the Number of Faults Injectioned by per Commitment (ANFIC) varies among different individual developers.*

A histogram of ANFIC for all 108 developers in all versions are shown in Figure 3. The average ANFIC for whole modules is 17.4%. Half of developers distributed in 0% to 20%. Meanwhile, 17% developers had over 30% of ANFIC, and two developers had 50%. This result indicates there exist individual differences of ANFIC.

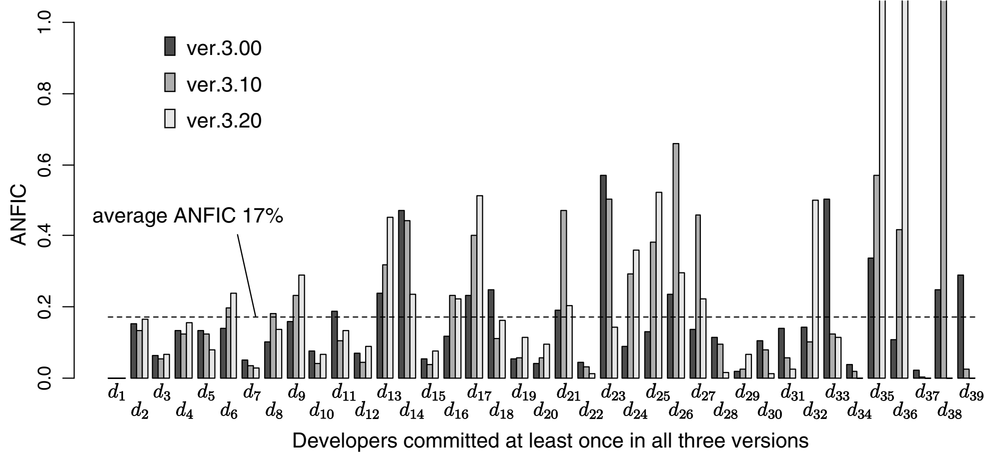


Figure 4: ANFIC for each developer on each version.

Next, we describe about the result of analyzing ANFIC for individual developers. Figure 4 shows ANFIC for each developer and each version. This figure was extracted only 39 developers who committed at least once for all three versions. The dashed line depicts the average ANFIC (17.4%). The sort criteria is ascending order of variation coefficient of three version's ANFIC. This means the left side developers has similar values for three versions. Note that the names of developers were assigned by sort ordering for unidentifying individuals.

We can show low ANFIC developers (d_1ĄCd_3ĄCd_7ĄCd_{10}ĄCd_{12}) and high ANFIC developers (d_{13-14}ĄCd_{17}) from Figure 4. Although, d_{12} and d_{14} committed more than 1,000 times in all three versions, their ANFIC had five times different (ANFIC of d_{12} was 0.07 and d_{14} was 0.38). By Friedman test, these differences has significant differences (P<0.01). So, we conclude the H_0 is supported.

5.4 Validation of Hypothesis H1

H_1: *Providing that H_0 is supported, the ANFIC can be characterized by human factors quantified by some developers metrics.*

Table 3 shows single correlation coefficients between ANFIC and five developer features. The "*" indicates significantly correlated pairs (P<0.05). The single coefficient of ANFIC(App_{v-1}) in ver.3.00 could not calculate because ver.3.00 is the first released version in ver.3 series.

$ANFIC(App_{v-1})$ had positively significant correlations with $ANFIC(App_v)$. In other words, a developer who injected many faults in previous version tends to inject many faults in next version. On the other hand, other developer's activity metrics had no significant correlations. These result indicates $ANFIC(App_v)$ can not characterized by metrics calculated by developer activities in current version, but can only characterized from $ANFIC(App_{v-1})$. Therefore, we conclude hypothesis H_1 was supported.

5.5 Validation of Hypothesis H2

H_2: *A module modified by many developers tends to contain more faults.*

Table 3: Single correlation coefficients between $ANFIC(App_v)$ and (Type 1) developer metrics.

	ver.3.00	ver.3.10	ver.3.20
NoC	-0.07	-0.19	-0.18
NoLR	-0.05	0.09	-0.14
NoUMR	0.00	-0.14	-0.22
NoUPR	0.35*	-0.06	-0.08
$ANFIC(App_{v-1})$	——	0.47*	0.32*

*: significantly correlated pairs (P < 0.05)

Table 4: Standard partial regression coefficients of linear regression models.

	ver.3.00	ver.3.10	ver.3.20
NoDRM	0.101	0.164	0.124
SLOC	0.279	0.091	0.141
Codechurn	0.276	0.537	0.383

5.5.1 Result by regression model

Table 4 shows standard coefficients of three metrics for each version. The result shows codechurn was mostly related with faults in all versions, and NoDRM little contributed for the regression model. Especially, in ver.3.10, coefficient of NoDRM was higher than SLOC. This means the number of developers was strongly related with increase in faults than SLOC in ver.3.10. Therefore, we conclude hypothesis H_2 is supported.

5.5.2 Focus on Modules

Although hypothesis H_2 was supported in section 5.5.1, we explored number of faults for every number of developers in more detail. Figure 5 indicates percentages of faulty modules which include at least one more fault, for every number of developers. Figure 5 shows over than 70% of modules modified by more than six developers were contain faults. Moreover, boxplots of number of faults for each modules are shown in Figure 6.About 80% of modules modified by one to five developers contain less than three faults. Meanwhile modules modified by over six developers contains at least 4.2 faults in average. The result of ver.3.10 and ver.3.20 were same as ver.3.00.

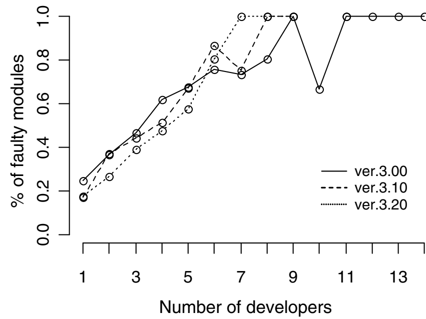


Figure 5: Percentage of faulty modules for every each number of developers.

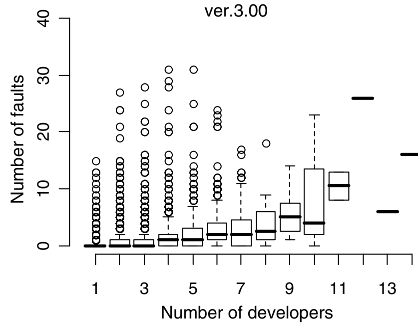


Figure 6: Number of bugs for each number of developers (ver.3.00).

5.6 Validation of Hypothesis H3

H_3: *Developer metrics are good predictor of fault prediction.*

The result of ROC-AUC values from fault prediction experiments for seven metrics combination by using three types of metrics are shown in Table 5. The "o" indicates the case of using the metrics type. The bottom line shows the case of using all of metrics which described in Table 1. The "**" and "*" represent the best performance combination and the second best combination compared with other combinations respectively. Average rank of each metrics combinations are shown in the rightmost column.

When constructed (and predicted) by only single type of metrics, a prediction model constructed by only change metrics has the best prediction performance (row 1), developer metrics is the second (row 3), and static metrics is the lowest (row 2). Compared with whether prediction model was constructed with developer metrics or not: average rank of static metrics improves from 6.7 (row 1) to 3.3 (row 5), change metrics improves from 3.5 (row 2) to 2.0 (row 6), static and change metrics improves from 3.5 (row 4) to 2.0 (row 7). Because prediction performance was improved by adding developer metrics, we conclude the H_2 is supported.

6. DISCUSSION

This paper analyzed the relationship between developer metrics and fault injection for understand human factors for software development. However *personnel evaluation* based on the developer metrics and on the analyzing results are inadequacy approaches. Some researchers pointed out that the measuring about individuals especially knowledge worker must be avoided [1, 4] because individual measurements (i.e., productivity) cannot correctly reflect individual performance.

For example, d_{14} in Figure 4 had higher ANFIC than averages in all three versions. However, there is a possibility that modules touched by d_{14} were complex and complicated. In addition, d_{14} committed over 1,000 times in all three versions. Raymond [19] pointed out that developers should release the software rapidly without regard to faults for tested by much of beta-testers. From this perspective, d_{14} made great contribution for Eclipse project.

The contributions of this paper is not to personnel evaluation but to understand following human factors for software development based on correct and fair analysis on empirical data.

- ANFIC varies among individual developers as with coding speed and debugging efficiency
- ANFIC positively correlated with ANFIC in previous version.
- The number of developers positively correlated with the number of faults as with SLOC and McCabe's complexity.
- Developer metrics are good predictor of fault prediction.

We suggest following reliability assessment activities based on these results.

- Allocate more testing efforts for modules which modified by high ANFIC developers in previous version. Note that ANFIC must be carefully used by only quality assurance teams which independent from developer teams.
- Reconsider developer assignments and/or allocate the testing efforts for the modules for modules which modified by many developers.
- When conducting a fault prediction, developer metrics should be added as predictors as with change metrics and static metrics.

7. RELATED WORK

Many studies have been reported on relationship between software product metrics and software reliability [2, 7, 8, 13, 18, 21, 24]. Gaffney [8] reported larger modules tend to contain more faults. Koru et al. [13] studied although larger modules contain more faults, smaller modules had higher fault density. These analysis were based on software product features, however there are a little studies focus on developers.

Schröter et al. [22] investigated empirical study based on developer's activities. This study described a data mining method from a bug tracking system (BTS). The result shows

Table 5: Result of fault-prone module detection (AUC of ROC curve).

#	metrics static	metrics change	metrics developer	constructed by ver.3.00 predict ver.3.10 LDA	LRA	CT	constructed by ver.3.10 predict ver.3.20 LDA	LRA	CT	average rank
1	o			0.732	0.739	0.697	0.720	0.722	0.650	6.7
2		o		0.818	0.838	0.738	0.893**	0.894*	0.771	3.5
3			o	0.832	0.846	0.657	0.815	0.833	0.660	5.2
4	o	o		0.825	0.849	0.653	0.818	0.842	0.713	4.8
5	o		o	0.820	0.834	0.738	0.888*	0.894**	0.771	3.3
6		o	o	0.861**	0.876**	0.767**	0.883	0.887	0.774**	2.0
7	o	o	o	0.853*	0.872*	0.767**	0.887	0.890	0.774**	2.0

LDA: linear regression, LRA: logistic regression, CT: classification tree
**: the best combination, *: the second best combination

number of pre-released failures and number of post-released failures for each developers varies among individuals as H_0 in this paper. Additionally they reported there is no significant relationship between number of failures and number of files changed as H_1. While the results used a single version data, we analyzed using three versions data and studying the relationship with not only number of changed modules, but also with lines modified and changed packages.

Some researchers have been studied about fault prediction. Moser et al. [16] proposed change metrics calculated by source code change histories and conducted fault prediction experiments using the change metrics. The change metrics used in this paper are based on the research. Although the change metrics proposed by Moser et al. contains number of developers, it is not clear that a module modified by many developers tends to contain more faults (H_2) because the number of developers and all of other 17 change metrics (e.g., modified modules, number of commitments) were used a single set of metrics as “change metrics” in the experiments. Moreover whether developer metrics are good predictor or not (H_3) is also unknown.

There is an empirical study [25] that uses developer features for constructing a fault prediction model. This research studies relationship between number of developers and faults as H_3. Weyuker et al. reports that the number of developers has no significant relationship with faults by reason that fault injection depends on other causes. However, the result derived from the observation that prediction performances were not improved since adding developer metrics as predictors. This explanation is a non-straightforward way for understanding hypothesis H_2 because performance of prediction model depends on other product metrics. On the other hand, in this paper, we calculated single partial coefficients with considering the effect of other metrics. Furthermore, experiment of hypothesis H_3 used multiple versions data and multiple combinations of three type of metrics.

8. CONCLUSION

This paper conducted an empirical study to analyze relationships between faults in source code and some developer metrics that can be derived from a software repository. The result from Eclipse Platform Project datasets shows that fault injection rate varies among different developers and the modules touched by more developer contained more faults. Furthermore, compared with conventional fault prediction models, developer metrics improved the prediction performance.

However, the effect of human factors strongly depends on each of its development environments. We need more empirical experiments using other project datasets for ensuring validity of the results. Although this paper shows significant positive correlation between faults and developers, understanding the causal relationship between developers and fault injection is our future work.

9. ACKNOWLEDGEMENTS

This research was partially supported by the Japan Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Young Scientists (B) (No.21700077).

10. REFERENCES

[1] R. D. Austin. *Measuring and Managing Performance in Organizations.* Dorset House Publishing Company, 1996.

[2] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *J. Syst. Softw.*, 51(2):245–273, 2000.

[3] S. G. Crawford, A. A. McIntosh, and D. Pregibon. An analysis of static metrics and faults in C software. *J. Syst. Softw.*, 5(1):37–48, 1985.

[4] T. Demarco and T. Lister. *Peopleware: Productive Projects and Teams, 2nd Ed.* Dorset House Publishing Company, February 1999.

[5] Eclipse Platform Project. http://www.eclipse.org/projects/project_summary.php?projectid=eclipse.platform.

[6] D. E. Egan. Individual differences in human-computer interaction. In *Handbook of Human-Computer Interaction*, pages 543–568. Elsevier Science, 1988.

[7] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, 2000.

[8] J. E. Gaffney. Estimating the number of faults in code. *IEEE Trans. Softw. Eng.*, 10(3):584–585, 1984.

[9] D. M. German. Using software trails to reconstruct the evolution of software: Research articles. *J. Softw. Maintenance and Evolution*, 16(6):367–384, 2004.

[10] D. M. German and A. Hindle. Visualizing the evolution of software using softchange. *Int'l J. Softw. Eng. and Knowledge Eng.*, 16(1):5–22, 2006.

[11] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. Int'l Conf. on Softw. Eng. (ICSE '09)*, pages 16–24, 2009.

[12] P. M. Johnson, H. Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, S. Zhen, and W. E. Doane.

Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proc. Int'l Conf. on Softw. Eng. (ICSE '03)*, pages 641–646, 2003.

[13] A. G. Koru, D. Zhang, K. E. Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Trans. Softw. Eng.*, 35(2):293–304, 2009.

[14] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, 2008.

[15] K. S. Lew, T. S. Dillon, and K. E. Forward. Software complexity and its impact on software reliability. *IEEE Trans. Softw. Eng.*, 14(11):1645–1655, 1988.

[16] 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. (ICSE '08)*, pages 181–190, 2008.

[17] J. C. Munson and T. M. Khoshgoftaar. *Software Metrics for Reliability Assessment.* McGraw-Hill, Inc., 1996.

[18] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk. Preliminary results on using static analysis tools for software inspection. In *Proc. Int'l Symp. on Softw. Reliability Eng. (ISSRE '04)*, pages 429–439. IEEE Computer Society, 2004.

[19] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly and Associates, 1999.

[20] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Commun. ACM*, 11(1):3–11, 1968.

[21] N. F. Schneidewind and H.-M. Hoffmann. An experiment in software error data collection and analysis. *IEEE Trans. Softw. Eng.*, 5(3):276–286, 1979.

[22] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk... In *Proc. Int'l Symp. on Empirical Softw. Eng. (ISESE '06)*, pages 18–20, 2006.

[23] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int'l Conf. on Mining Softw. Repositories (MSR '05)*, pages 1–5, 2005.

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

[25] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Softw. Eng.*, 13(5):539–559, 2008.

[26] S. Yamada and S. Osaki. Software reliability growth modeling: Models and applications. *IEEE Trans. Softw. Eng.*, 11:1431–1437, 1985.

[27] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proc. Int'l Conf. on Softw. Eng. (ICSE '08)*, pages 531–540, 2008.