Predicting Fault-Prone Modules Based on Metrics Transitions

Yoshiki Higo[1] Kenji Murao[1] Shinji Kusumoto[1] Katsuro Inoue[1]
[1]Graduate School of Information Science and Technology, Osaka University
{higo,k-murao,kusumoto,inoue}@ist.osaka-u.ac.jp

ABSTRACT

This paper describe a method for identifying fault-prone modules. The method utilizes metrics transitions rather than raw metrics values. Metrics transitions are measured from the source code of consecutive versions, which is archived in software repositories. Metrics transitions should be an good indicator of software quality because they reflect how the software system has evolved. This paper exhibits a case study, which is a comparison between metrics transitions and CK metrics suite. In the case study, the metrics transitions could precisely identify fault-prone modules.

1. INTRODUCTION

Information about how a software system has evolved is useful for various activity on the system in the future. However, it is burdensome and costly to collect and analyze the information manually. Developers tend to get overwhelmed to process assigned works, they don't afford to do extra works for the future.

A software repository is a container that includes all products and histories of a software system. There are many software tools to handle software repositories. By mining software repositories, we can easily get any products of the software system at any given point in the past. Recently, mining software repositories has received much attention, and it is widely accepted that it provide beneficial information for developers or maintainers.

This paper describes a method to identify characteristic features of a software system by mining the software repository. The method measures how metrics of the modules included in the system have changed through the development and maintenance. For example, if the metrics of certain modules are up-and-down repeatedly, the modules may need more maintenance costs then other modules because the modules were changed substantially: we think that simple bug fixes or function additions don't yield up-and-down metrics transitions. The method lets us analyze a software system form various viewpoints, and we can easily identify its tendencies and attributes.

We believe that the method can be applied in the various contexts of software development and maintenance for getting useful information. In this paper, especially, we describes the method from the viewpoint of identifying fault-prone modules. Identification of fault-prone modules is an active research topic, and various kinds of techniques have been proposed for that [6, 9]. The identification lets us know which modules we should pay more attention than others, which leads to effective development and maintenance.

Section 2 describes some terms, and Section 3 explains statistics tools utilized in the proposal technique. Section 4 describes the procedure of metrics transitions from the viewpoint of identifying fault-prone modules. Section 5 gives a description of the results that we applied the method to an open source software system. Finally, we conclude our paper with future works in Section 7.

DEFECTS'08, July 20, 2008, Seattle, Washington, USA.

2. TERMS

Here, we define some terms in the context of the proposal technique.

Snapshot: A snapshot is a set of source files needed to construct the system just after a check-in is done by a developer. Hence, the number of snapshots is equal to the number of check-ins. In the method, snapshots are retrieved from repositories of version control systems (e.g., CVS [1], Subversion [3]). Also, the developer name and the date of the check-ins are retrieved as well as actual source code.

Fluctuation: A fluctuation is an indicator representing how a software system has evolved. In the proposal technique, fluctuations are calculated based on how the metrics of the software system have transited through all of the snapshots.

Characteristic Feature: A characteristic feature is a certain tendency or attribute of a software system. There are various kinds of characteristic features in single software system, and we believe that elucidating its characteristic features leads us to develop or maintain the system in more efficient way. For example, we think that it is possible to evaluate the modules, the design, and the developers of the software system by using the elucidated characteristic features. Especially, in this paper, we describe the proposal technique from the viewpoint of identifying fault-prone modules.

3. TOOLS FOR FLUCTUATION MEASUREMENT

We introduce several statistics tools for measuring fluctuations of software metrics. In this introduction, we assume the following situation:

$MO = \{mo_1, mo_2, \cdots, mo_\alpha\}$**:** the set of the modules included in the target software system where α is the number of them.

$ME = \{me_1, me_2, \cdots, me_\beta\}$**:** the set of the measured metrics where β is the number of them.

$CT = \{ct_1, ct_2, \cdots, ct_\gamma\}$**:** the set of times when check-ins of any of the source files were done in the past where γ is the number of the total check-ins. ct_1 is the first check-in on the repository, and ct_γ is the last check-in.

$v(i, j, k)$**:** the value of metric me_j on module mo_i in time ct_k. If module mo_i doesn't exist in time ct_k, we assume that $v(i, j, k) = null$.

3.1 Entropy

Entropy is an indicator to represent the degree of uncertainty [7]. Given that fluctuation is uncertainty of metrics, we can use Entropy as an indicator of metrics fluctuations. Entropy is measured on every module and every metric, and it is defined as:

$$H(i,j) = -\sum_{l=1}^{\gamma'} p_l log_2 p_l$$

where:

- γ' is the number of different values $\{v'_1, v'_2, \cdots, v'_{\gamma'}\}$ of metric me_j on module mo_i ($1 \leq \gamma' \leq \gamma$),
- p_l is the probability that metric me_j is v'_l.

If all values of metric me_j on module mo_i ($v(i, j, 1)$, $v(i, j, 2)$, $\cdots$, $v(i, j, \gamma)$) are different from each other, Entropy $H(i, j)$ becomes the maximum value $-log_2 \frac{1}{\gamma}$. Meanwhile, if all metric values are the same, $H(i, j)$ becomes the minimum value $-log_2 1 = 0$.

3.2 Normalized Entropy

The maximum value of Entropy depends on the number of check-ins, so that it is difficult to compare Entropies between different software systems. To solve this problem, we define **Normalized Entropy**, H'.

Normalized Entropy is measured on every module and every metric because it is derived from Entropy. If we use the same assumption of Entropy, Normalized Entropy is defined as:

$$H'(i,j) = \frac{H(i,j)}{-log_2 \frac{1}{\gamma}}$$

3.3 Quartile Deviation

In the field of statistics, **Quartile Deviation** is utilized for representing how data spread. Given that fluctuation is spread of metrics, we can utilize Quartile Deviation[1] as an indicator of metrics fluctuations.

We assume that, $q_1(i, j)$/$q_3(i, j)$ is first/third quartile of the sorted set of values of metric me_j on module mo_i, Quartile Deviation, Q_{mo_i, me_j}, is defined as:

$$Q(i,j) = \frac{q_3(i,j) - q_1(i,j)}{2}$$

3.4 Quartile Dispersion Coefficient

Quartile Deviation is greatly affected by the difference of scales between metrics because it is an absolute value. Thus, it is difficult to compare Quartile Deviation between different metrics. To solve this problem, we uses **Quartile Dispersion Coefficient**, Q'.

[1]Spread of metrics is not normally-distributed. Quartile Deviation can be utilized to evaluate nonnormal distribution.

Quartile Dispersion Coefficient is measured on every module and every metric as well as Quartile Deviation because it is derived from Quartile Deviation. If we use the same assumption of Quartile Deviation and $m(i, j)$ is the median value, Quartile Dispersion Coefficient is defined as[2]:

$$Q'(i,j) = \frac{Q(i,j)}{m(i,j)}$$

3.5 Hamming Distance

In this research, **Hamming Distance** is measured on consecutive two snapshots, it is defined as:

$$HD(i,k) = \sum_{j=1}^{\beta} \text{diff}(v(i,j,k-1), v(i,j,k))$$

where:

$$\text{diff} = \begin{cases} 1 & (v(i,j,k-1) \neq v(i,j,k)) \\ 0 & (v(i,j,k-1) = v(i,j,k)) \end{cases}$$

$HD(i, k)$ represents the number of metrics changed between consecutive two check-ins, ct_{k-1} and ct_k, on module mo_i.

3.6 Euclidean Distance

Hamming Distance counts only whether each metric value changed or not while **Euclidean Distance** counts how much the values changed. In this research, Euclidean Distance is a distance in β-dimensional Euclidean space, and it is defined as:

$$ED(i,k) = \sqrt{\overrightarrow{\mathrm{V(i,k)}}^T \overrightarrow{\mathrm{V(i,k)}}}$$

where:

- $\overrightarrow{\mathrm{v(i,k)}} = [v(i,1,k), v(i,2,k), \cdots, v(i,\beta,k)]$, which is a vector representation of the metrics values on module mo_i at time ct_k,
- $\overrightarrow{\mathrm{V(i,k)}} = [\overrightarrow{\mathrm{v(i,k)}} - \overrightarrow{\mathrm{v(i,k-1)}}]$.

3.7 Mahalanobis Distance

Fluctuations measured by Euclidean Distance are valid under assumption that there is no correlation and no scale difference between metrics. However, in practice, there are often some correlations and scale differences between them. Thus, the proposal technique utilizes **Mahalanobis Distance**, which is valid under existences of correlations and scale differences.

If we use the same assumption of Euclidean Distance, and Σ is the covariance matrix of the whole of the system, Mahalanobis Distance is defined as:

$$MD(i,k) = \sqrt{\overrightarrow{\mathrm{V(i,k)}}^T \Sigma^{-1} \overrightarrow{\mathrm{V(i,k)}}}$$

4. MEASUREMENT PROCEDURE

Figure 1 illustrates an overview of the proposal technique. The method can be applied to only the software systems maintained with a revision control system. The process of the method is the follows:

STEP1 :Retrieves all of the snapshots

In this step, all versions of the snapshots are retrieved from the software repository to measure metrics. A snapshot is a set of all

[2]The definition is based on the assumption that all metrics values are greater than 0. If some values are negative, the median value may be 0, or very close to 0. In such case, we have to consider another definition.

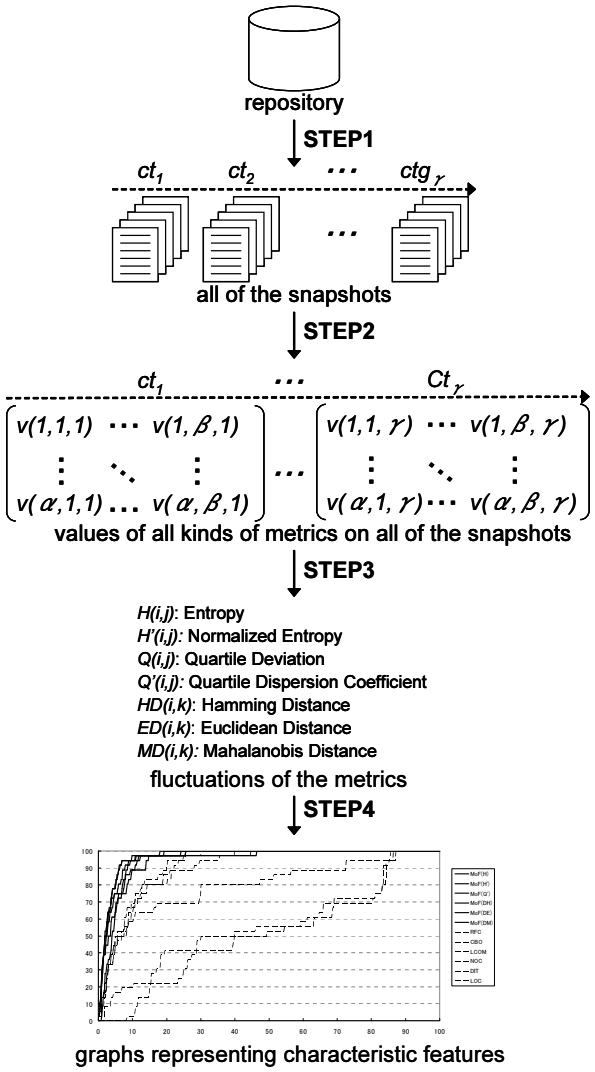

Figure 1: Overview of the method

source files just after at least one source files was updated by a check-in. For example, if there is a software repository including change history represented in Figure 2. The method retrieves sets of the source files at four times ct_1, ct_2, ct_3, and ct_4.

STEP2: Measures metrics of all of the snapshots

In this step, the method measures metrics of all versions of the retrieved snapshots. It is necessary to select appropriate software metrics fitted for the purpose: if the unit of module is class/method, we should utilize class/method metrics; if we focus on the coupling/cohesion of the software system, coupling/cohesion metrics should be utilized. In the case study of this paper, we utilizes CK metrics suite [5] mainly.

STEP3: Computes fluctuation of the metrics

The method computes fluctuation of the metrics measured in the previous step. As of now, seven kinds of the fluctuations described in Section 3 are computed.

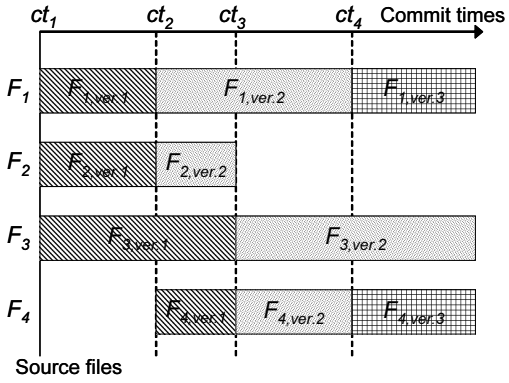

Figure 2: Example of change history

STEP4: Analyzes characteristic features

The fluctuations calculated in the previous step have 2-dimensions: for example, Hamming Distance has module and time dimensions. By cutting a dimension of the two, we can compute the fluctuation of the other dimension. Thus, we can compute various fluctuations of modules, metrics, and times. Especially, in this paper, we describe about fluctuations of modules for identifying fault-prone modules.

Module Fluctuation is an indicator how modules have evolved through its lifecycle[3]. The fluctuation of module mo_i utilizing Entropy (H) is defined as:

$$MoF(H(i,j)) = \sum_{j=1}^{\beta} H(i,j)$$

Module fluctuations utilizing other statistics tools introduced in Section 3 can be defined as well as the module fluctuation utilizing Entropy:

- $MoF(H(i,j)') = \sum_{j=1}^{\beta} H'(i,j)$,
- $MoF(Q(i,j)) = \sum_{j=1}^{\beta} Q(i,j)$,
- $MoF(Q(i,j)') = \sum_{j=1}^{\beta} Q'(i,j)$,
- $MoF(HD(i,k)) = \sum_{k=1}^{\gamma} HD(i,k)$,
- $MoF(ED(i,k)) = \sum_{k=1}^{\gamma} ED(i,k)$,
- $MoF(MD(i,k)) = \sum_{k=1}^{\gamma} MD(i,k)$.

[3]We also defined **Metric Fluctuation** and **Change Fluctuation**. However we don't describes about them in this paper because they are out of scope of this workshop.

After calculating fluctuations, Graph representations are utilized for visualizing them. These fluctuations indicate various characteristic features of the software system, and they are useful to get rich knowledge of the system.

5. CASE STUDY

5.1 Objective

The objective is to confirm that the method provides useful information for software development and maintenance. Especially, in this case study, we applied the method for confirming that classes having fluctuations tend to be more fault-prone than other classes.

5.2 Target

In this case study, the target is an open source software system, FreeMind. FreeMind is a mind-mapping software written in Java language [2]. Table 1 represents the overview of FreeMind. At the first snapshot (ct_1), the LOC is 3,882, and the number of source files is 67. At the last snapshot (ct_{225}), the LOC is 39,350, and the number of source files is 221.

5.3 Utilized Metrics

In this case study, we utilized six metrics. Five of them are RFC, CBO, LCOM, NOC, DIT, which are members of CK metrics suite [5]. The last metric is LOC, which is the simplest and most widely utilized metric.

It was experimentally evaluated that CK metrics suite is a good indicator to predict fault-prone classes [4, 8]. Hence, it is beneficial that the fluctuations of CK metrics suite are compared with its raw values.

5.4 Configuration

We performed this case study in the following procedure.

1. Devides the development history into anterior half and posterior half.

2. Calculates fluctuations from the anterior half history. In this case study, we didn't utilize quartile deviation because there are scale differences between each of CK metrics and LOC.

3. Measures the six metrics from the last snapshot of the anterior half history.

4. Identifies bug fixes in the posterior half and counts the number of them. In this case study, we identifies the bug fixes based on commit log messages. In the FreeMind project, prefix "Bug fix:" is attached to the commit log message if the commit is a bug fix. We regarded commits whose log message includes both "bug" and "fix" as bug fixes. Also, in this study, we regarded bugs in a source file as bugs in the public class defined in the source file.

5. Sorts FreeMind's classes in the order of fluctuations and raw metrics values. Also, bug coverages are calculated based on the orders.

Table 1: Overview of FreeMind

Software	FreeMind
Language	Java
# of Developers	12
# of snapshots (γ)	225
first commit time (ct_1)	2000/08/01 19:56:09
last commit time (ct_γ)	2008/01/13 20:55:35
# of source files of ct_1	67
# of source files of ct_γ	221
LOC of ct_1	3,882
LOC of ct_γ	39,350

5.5 Results

The case study was performed on a single PC workstation[4]. It took about 18 minutes to calculate the fluctuations from the snapshots of the posterior half [5].

Figure 3 illustrates the result of bug coverage comparison between the class (module) fluctuations and raw metrics values. X axis is ranking coverage (%), and Y axis is bug coverage (%). Ranking coverage means sorted files in descending order of their class fluctuations, and bug coverage means bugs ratio included in the files of top xx%, which is specified by the ranking coverage. We can see that the ranking based on the class fluctuations could identify fault-prone class more precisely than the ranking based on the raw metrics values. At the top 20% of all classes, ranking based on class fluctuations includes 95% to 100% bugs meanwhile ranking based on the raw metrics values includes 22% to 89% bugs.

5.6 Discussion

The result of this case study shows that class fluctuations can be a good indicator to predict fault-prone classes as well or better than raw software metrics. To confirm that this is true in other software systems, we applied the proposal technique to other two open source software systems, JHotDraw and HelpSetMaker. In the case of JHotDraw (HelpSetMaker), ranking based on class fluctuations includes 44% (60%) to 59% (75%) bugs meanwhile ranking based on the raw metrics values includes 10% (28%) to 48% (63%) bugs at the top 20% of all classes. In the applications of the both systems, the class fluctuations had better bug coverages than raw metrics values as well as FreeMind. However, calculating module fluctuations requires much more time than measuring metrics from a single version of the source code. In this case study, it took requires 18 minutes to calculate class fluctuations from sets of snapshots of FreeMind, which was stored in the local storage in advance.

6. RELATED WORKS

Williams et al. reported that checking return values of methods is an effective way to predict fault occurrences in the future based on their experiments [9]. Also, they applied the technique to the source code of the latest version with and without the information stored in the CVS repository of the software system, and compared the results. The results showed that the checking with the CVS information can predict fault occurrences more precisely than the checking without the CVS information.

Ostrand et al. predicts the location and number of faults occurred in the next release based on information stored in CVS repository and bug database [6]. They constructed a negative binomial regression model from the information, and experimentally evaluated that the model can predict more precisely than the prediction based on LOC. Also, they reported that it is possible to suggest which source files should be paid attention in the test phase.

[4]CPU: Core2Duo 1.86GHz, Memory: 2GB

[5]Source files of all snapshots were extracted in the local storage of the workstation in advance.

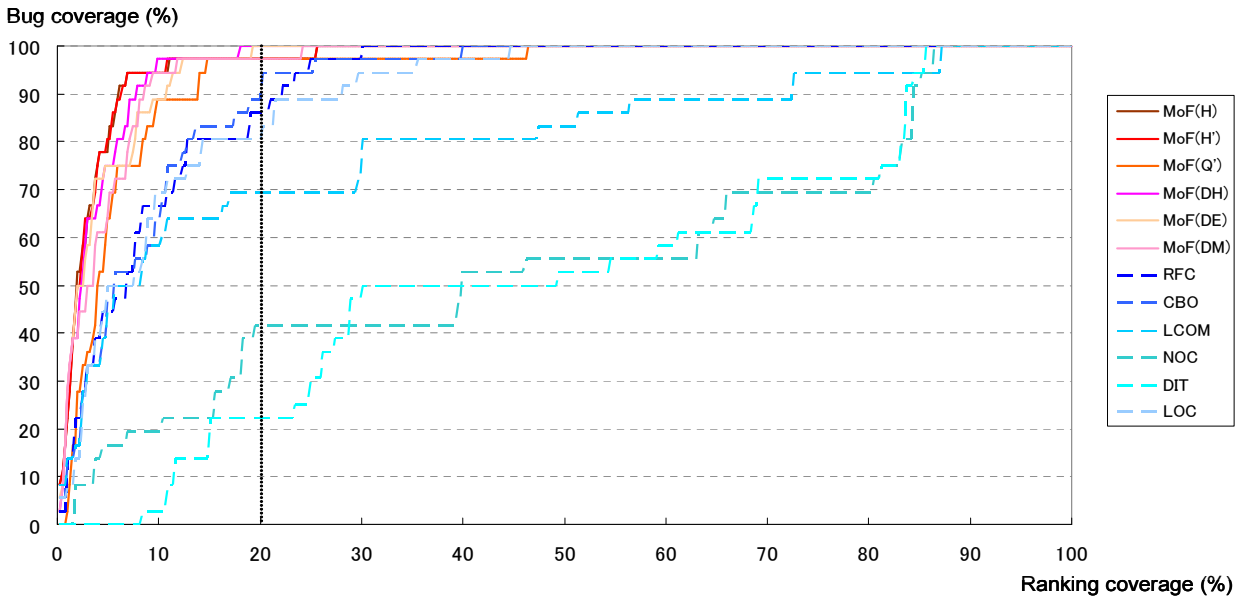

Figure 3: Bug Coverage Comparison between Class (Module) Fluctuations and Raw Metric Values

7. CONCLUSION

In this paper, we proposed a method for analyzing characteristic features of a software system based on metrics transitions. Also, this paper describes the result of identification of fault-prone modules. This research is work-in-progress, and we have many things to do, the followings are the some of them: (1)we have to define details of the method, some heuristics may be introduced to the method; (2)we have to investigate attributes of the fluctuations; We think that the fluctuations are very much correlated with the number of changed or number of lines changes of the source code over revisions; (3)we are going to enhance the tool to handle other programming languages, and apply the method to various software systems; (4)we are going to apply the method to analyze other feathers in the different contexts; (5)it may be interesting to combine multiple fluctuations using machine learning or linear regression algorithms to predicut faults.

Acknowledgment

This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology.

8. REFERENCES

[1] CVS. `http://ximbiot.com/cvs/wiki/`.

[2] FreeMind. `http://freemind.sourceforge.net/wiki/index.php/Main_Page`.

[3] Subversion. `http://subversion.tigris.org/`.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct 1996.

[5] S. Chidamber and C. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 25(5):476–493, Jun 1994.

[6] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, Apr 2005.

[7] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[8] R. Subramanyam and M. S. Krishnan. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, Apr 2003.

[9] C. C. Williams and J. K. Hollingworth. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, Jun 2005.