JOURNAL OF SOFTWARE ENGINEERING & INTELLIGENT SYSTEMS
ISSN 2518-8739
30th April 2018, Volume 3, Issue 1, JSEIS, CAOMEI Copyright © 2016-2018
www.jseis.org

Using peer comparison approaches to measure software stability

[1] Liguo Yu, [2] Yingmei Li, [3] Srini Ramaswamy
[1]Indiana University South Bend, South Bend, Indiana 46634, USA
[2]Harbin Normal University, Harbin, Heilongjiang 150080, China
[3]ABB Inc., Cleveland, Ohio 44125, USA
Email: [1]ligyu@iusb.edu, [2]yingmei_li2013@163.com, [3]srini@acm.org

ABSTRACT

Software systems must change to adapt to new functional requirements and new nonfunctional requirements. This is called software revision. However, not all the modules within the system need to be changed during each revision. In this paper, we study how frequently each module is modified. Our study is performed through comparing the stability of peer software modules. The study is performed on six open-source Java projects: Ant, Flow4j, Jena, Lucence, Struct, and Xalan, in which classes are identified as basic software modules. Our study shows (1) about half of the total classes never changed; (2) frequent changes occur to small number of classes; and (3) the number of changed classes between current release and next release has no significant relations with the time duration between current release and next release.

Keywords: software evolution; software revision; software stability; class stability; open-source project; Java class;

1. INTRODUCTION

Software systems must continually evolve to fix bugs or adapt to new requirements or new environments. The changes made to an existing system would generate a new version of the system. This process is called revision. During the software revision process, some modules within the system are modified and some other modules are unchanged. The ability that a software module remains unchanged is called stability. Stability is an important measure of software modules and software systems. It is commonly agreed on that software stability could affect software quality [1]. For example, if more frequent and dramatic changes are made to a software module, it is more likely that errors will be introduced into the code, and accordingly the quality of the module and the quality of the product could be compromised. It is also commonly agreed on that less stable modules are more difficult to maintain than stable modules [2, 3]. For example, regression faults could be introduced in software maintenance. Maintaining module stable is also important for software product line, where the stability of core assets is essential for software reuse [4−6].

Software stability is an area that is under extensive research. For example, Fayad and Altman described a Software Stability Model (SSM) [7, 8], which has been applied to software product lines to bring multiple benefits to software architecture, design, and development [9]. Xavier & Naganathan presented a probabilistic model to enhance the stability of enterprise computing applications, which allows software systems to easily accommodate changes under different business policies [10, 11]. In Wang et al.'s research, stability is used as a measure to support component design [12].

Identification of stable and unstable software components is one of the important tasks in this area of research. For example, Hamza applied a formal concept analysis method to identify stable software modules [13]. Grosser et al. utilized case-based reasoning to predict class stability in object-oriented systems [14]. Bevan & Whitehead mined software evolution history to identify unstable software modules [15].

This paper studies software stability through comparing peer modules. The study is performed on six open-source Java projects. The remaining of the paper is organized as follows. Section 2 reviews the current available measurements of software stability. Section 3 presents our research method and introduces our new measurement of software stability. Section 4 describes the data source used in this study. Section 5 presents the results and the analysis of the case studies. Conclusions and future work are illustrated in Section 6.

2. LITERATURE REVIEW

A review of literature could find out that there are basically two ways to measure software module stability, static measurement and dynamic measurement. Static measurement is to analyze the interdependencies between software modules in order to study their probability of co-evolutions: changes made to one module could require corresponding changes to another module [16, 17]. If a software module has weak dependencies on other modules, change propagation is less likely to happen and the module is more stable. If a software module has strong dependencies on other modules, change propagation is more likely to happen and the module is less stable. This measurement is related with module coupling, which represents the architecture of the system. Because this measurement examines the interactions of different modules of one version of a software system, it is called static measurement. The concept of static measurement is illustrated in Figure 1(a).

In dynamic measurement, software evolution history is used to measure module stability, where stability is represented as differences between two versions of an evolving software product. The differences between two versions of an evolving software module could be measured with program metrics difference, such as number of variables, number of methods, etc. [18, 19]. Because this measurement examines the differences of two versions of one module, it is called dynamic measurement. The concept of dynamic measurement is illustrated in Figure 1(b).

In Threm et al.'s latest work, information-level metrics based on Kolmogorov complexity are used to measure the difference between several versions of software products [20]. Using normalized compression distance, various evolutionary stability metrics of software artifacts are defined, including version stability, branch stability, structure stability, and aggregate stability. Again, this is a dynamic measurement based on information entropy, which also belongs to Figure 1(b).

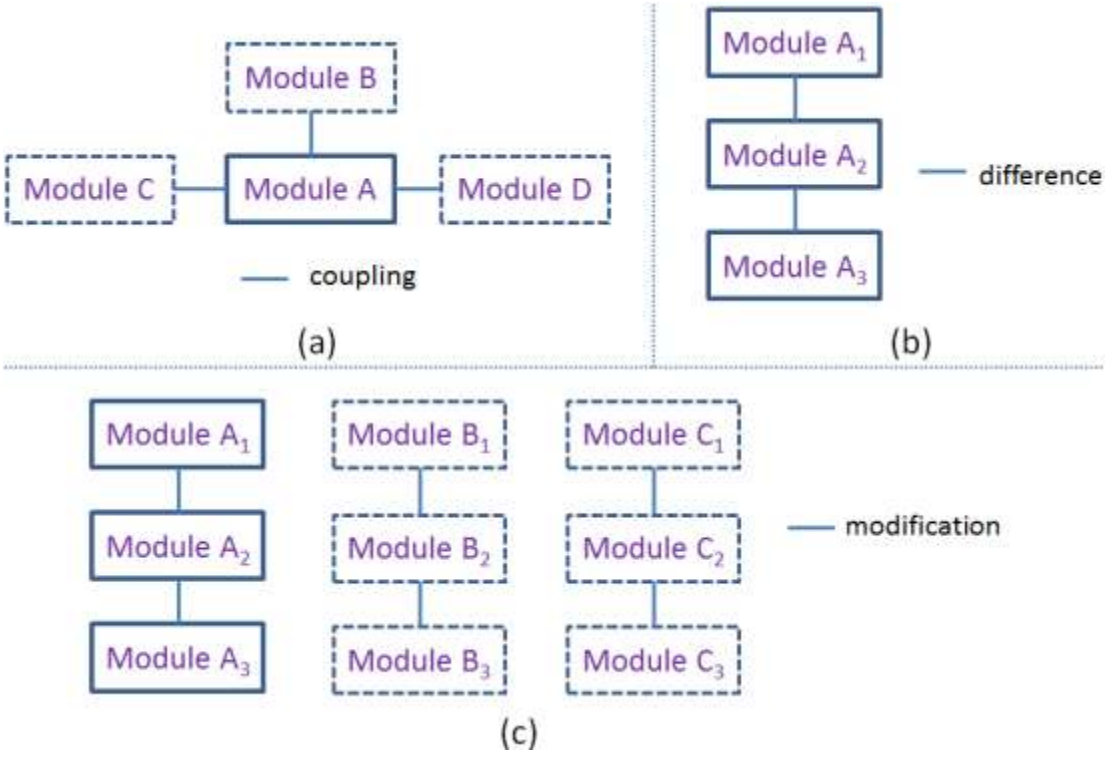

Figure. 1 Three measurements of module stability: (a) static measurement; (b) dynamic measurement; and (c) peer comparisons.

3. RESEARCH METHOD

In this study, we analyze module stability through comparing peer modules in one system. Instead of looking at interactions between modules (static measurement) and difference between versions of one module (dynamic

measurement), we compare the change frequencies of different modules. We call our approach peer comparisons. The concept of peer comparisons is illustrated in Figure 1(c). In this measurement, the stability of a module is measured through comparing its frequency of changes with other modules' frequencies of changes. For example, in Figure 1(c), if Module A is modified 1 time, Module B is modified 2 times, and Module C is modified 3 times, in three revisions, we can say Module A is more stable than Module B, and Module B is more stable than Module C.

4. DATA SOURCE

The data used in this study are retrieved from Helix - Software Evolution Data Set [21]. The evolution data of six open-source Java projects are downloaded and analyzed. They are Ant, Flow4j, Jena, Lucence, Struct, and Xalan. Table 1 shows the general information about six Java projects. Please note (1) The release months are shown in mm/yy format; (2) The first release date and the last release date are referring to the data collected by Helix, and are not necessarily representing the available data on the project web site; and (3) Number of classes is counted on the last release of each project as specified in the table.

Table. 1 The general information of six Java projects

	Ant	Flow4j	Jena	Lucence	Struct	Xalan
Num. of releases	18	29	25	19	18	15
First release	07/00	10/03	09/01	06/02	09/00	03/00
Last release	04/10	08/05	06/10	06/10	09/09	11/07
Duration (days)	3573	660	3168	2931	3288	2803
Num. of classes	561	274	915	398	910	1198

5. ANALYSIS AND RESULTS

First, we study the retirement rate of classes. During the evolution of an object-oriented software product, some new classes could be added to the project and some existing classes could be removed from the project. The removal of a class from a software system is called retirement of the class. The definition of class retirement rate is given below.

Definition 1. Class retirement rate of a project is the ratio of the number of retired classes over the number of total classes ever existed in the project.

Class retirement rate measures the stability of a software system as a whole: higher class retirement rate indicates lower stability of the system and lower class retirement rate indicates higher stability of the system. Table 2 shows the class retirement rates of six Java projects studied in this research. It is worth noting that Row 2 shows the total number of ever existed classes in each project. From Table 2, we can see that the class retirement rates are in the range of 6.5%, which is for Ant, and 52.2%, which is for Jena. Based on the data in Table 2, it is fair to say that Ant is more stable than Jena. Class retirement rate represents the stability of the entire system, but not individual classes (modules). To study the stability of individual classes (modules), we need to examine them in more detail.

For all the current classes in six Java projects, if a class has never been changed during the revisions, it is called an *unchanged class*; if a class has ever been changed during the revisions, it is called a *changed class*. Figure 2 illustrates the percentage of changed classes and the percentage of unchanged classes in all six Java projects, in which Lucence has the largest percentage of changed classes and Struct has the largest percentage of unchanged classes.

Table 2. The class retirement rate of six Java projects

	Ant	Flow4j	Jena	Lucence	Struct	Xalan
Num. of classes	600	403	1916	478	1498	1585
Current classes	561	274	915	398	910	1198
Retired classes	39	129	1001	80	588	387
Retirement rate	6.5%	32.0%	52.2%	16.7%	39.3%	24.4%

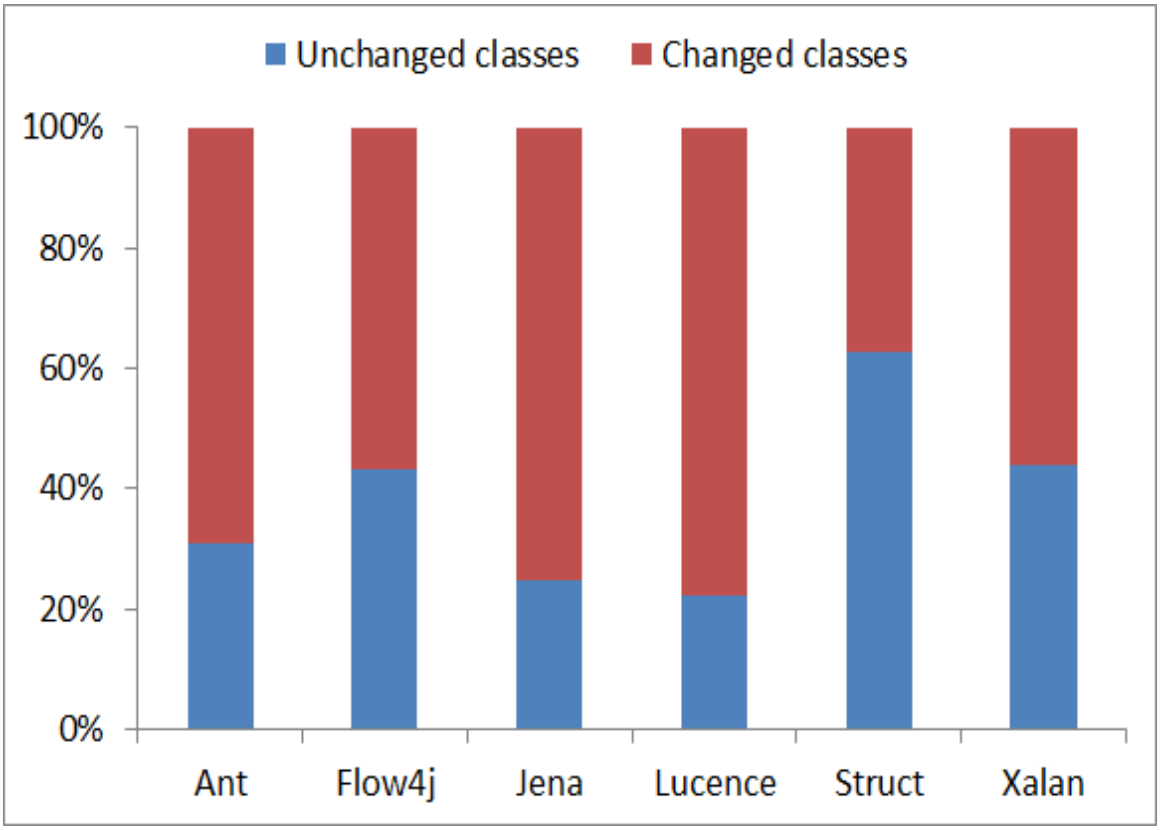

Figure. 2 The percentages of unchanged and changed classes in each system.

For changed classes, Figure 3 shows the frequency of the number of times a class is changed in all the revisions. It can be seen that more classes are changed fewer times and fewer classes are changed more times.

Some classes are not introduced in the first version of the product. Instead, they are added later during the revisions. Therefore, these later added classes did not experience the full evolution lifetime and accordingly, the number of changes made to them could not accurately represent their stability. To account for the shortcomings of using the number of changes as a direct measure of class stability, we introduce a new metric, revision rate.

Definition 2. Revision rate of a class is the ratio of the number of times this class is changed over the total number of revisions this class experienced.

For example, if a class experienced 4 revisions and changes are made in 1 revision, the revision rate of this class is 1/4 (25%). We can tell from the definition that revision rate is in the range of [0%, 100%]. Figure 4 shows the frequency of classes with different revision rates in their lifetime of evolution. It should be noted here that Figure 4 includes both changed classes and unchanged classes as illustrated in Figure 2.

From Figure 4, we can see that more classes have low revision rates while less classes have high revision rates. For example, in Ant, Lucence, and Xalan, some classes have about 90% possibilities of being changed in a revision.

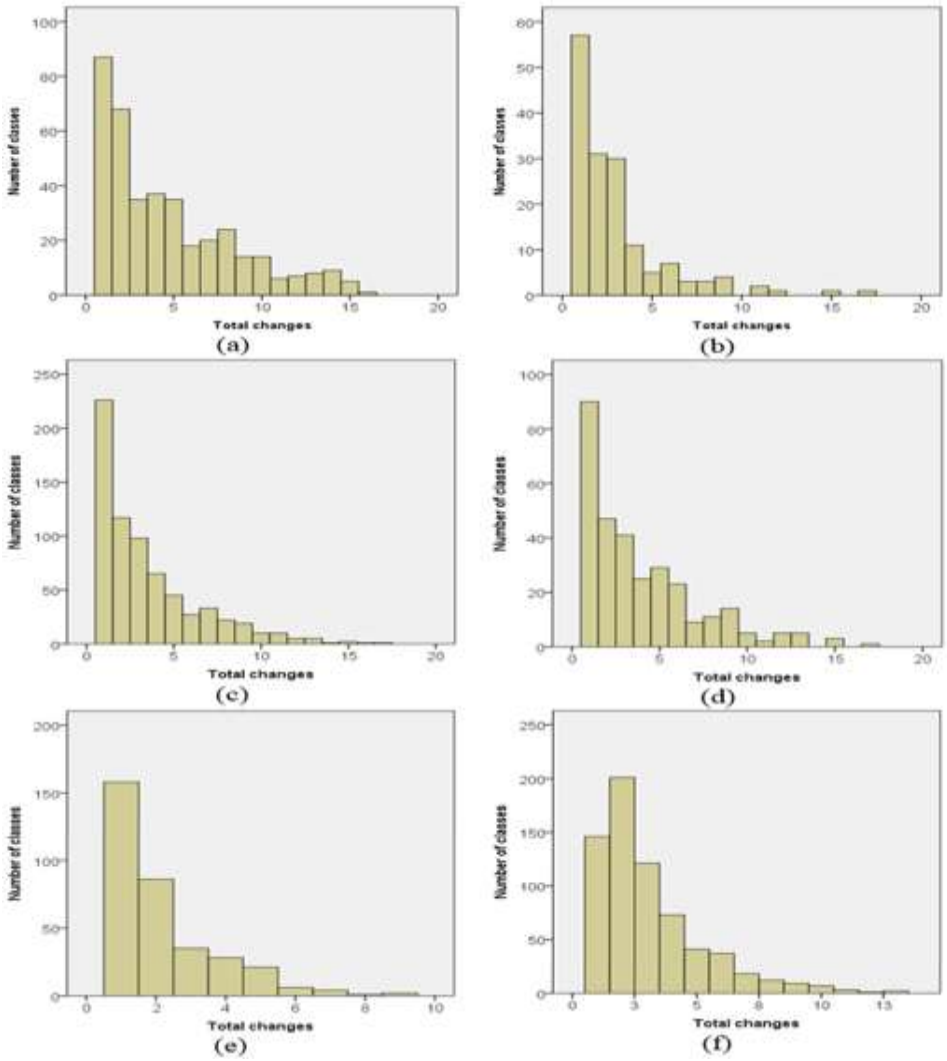

Figure. 3 For changed classes, the frequency of the number of times a class is changed in all the revisions: (a) Ant; (b) Flow4j; (c) Jena; (d) Lucence; (e) Struct; and (f) Xalan.

As described in Section 3, peer comparisons are used in this study to evaluate module stability. Figure 3 and Figure 4 show that different classes in one system have different number of changes and different possibility of being changed in a revision. Accordingly, classes with high possibility of being changed (revision rate) in a revision is considered more unstable than classes with low possibility of being changed (revision rate).

Next, we study the relationship between amount of changes and the duration of each revision. If a revision takes longer time, it is more likely that major changes are being made and more classes are being modified; if a revision takes shorter time, it is more likely that minor changes are being made and fewer classes are being made. To see if this analysis is correct, we study the correlation between the percentage of modified classes made on previous release and the duration between previous release and current release in each revision. Table 3 shows the results of Spearman's rank correlation test, where significance at the 0.05 level is bolded. Figure 5 illustrates the scatter plots of percentage of classes modified and the duration between previous release and current release.

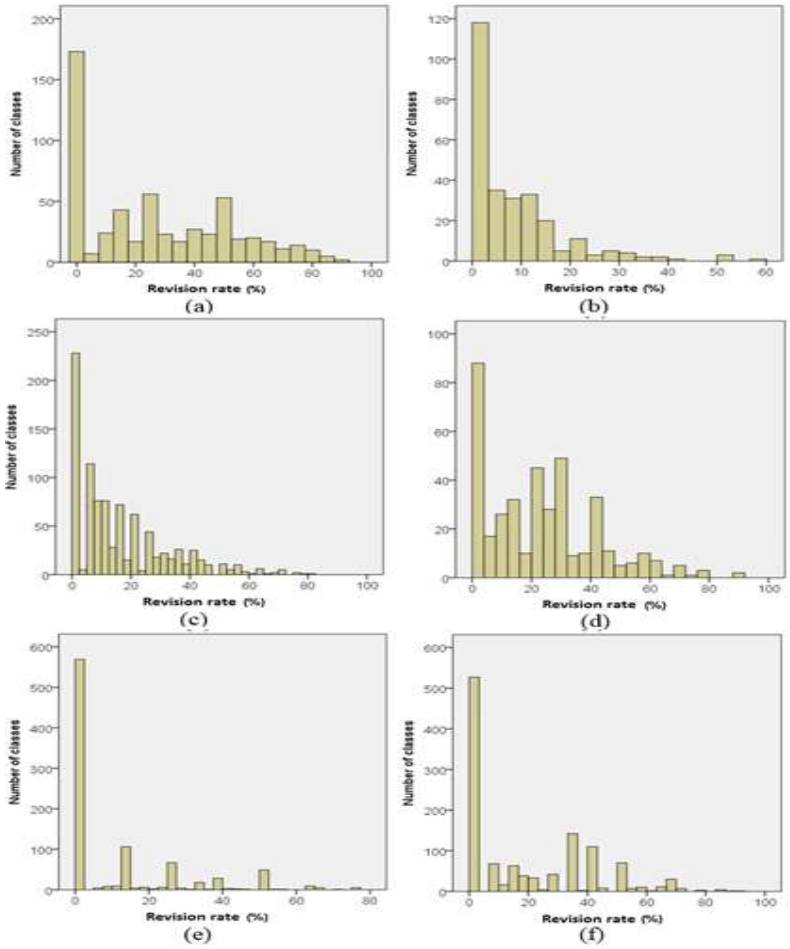

Figure. 4 The frequency of classes with different revision rates: (a) Ant; (b) Flow4j; (c) Jena; (d) Lucence; (e) Struct; and (f) Xalan.

From Table 3 we can see that 5 out of the 6 correlations are positive and 3 out of the 5 positive correlations are at the 0.05 level. Based on this result, we could not conclude that the amount of changes made to classes are correlated with the duration of each revision. In other words, the amount of changes depends on revision activities. On the other hand, the duration of each revision is also related with the maintenance activity. Accordingly, amount of changes and the duration of each revision are indirectly correlated, but with no causal relations.

Table 3: Spearman's correlation test on percentage of classes modified and duration between previous release and current release

	Ant	Flow4j	Jena	Lucence	Struct	Xalan
Num. of datasets	17	28	24	18	17	14
Correlation (r)	0.554	0.175	-0.170	0.498	0.583	0.363
Significance (p)	**0.02**	0.37	0.43	**0.04**	**0.01**	0.20

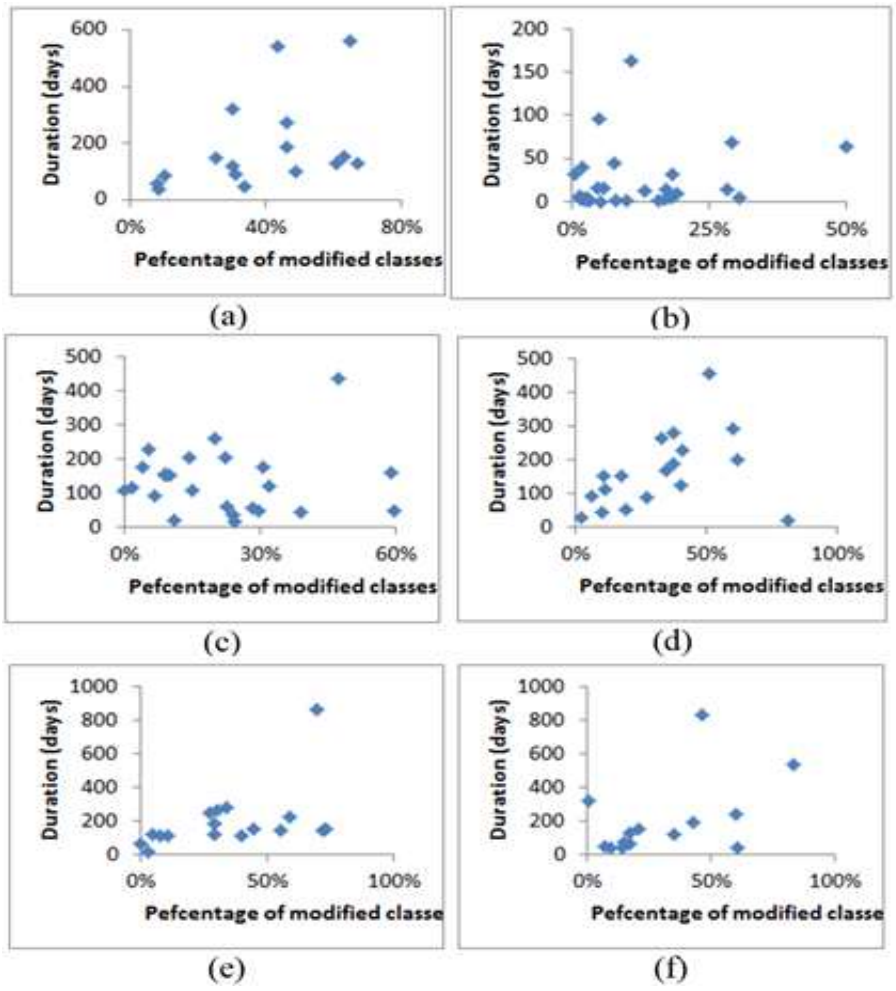

Figure. 5 The scatter plots between percentage of classes modified and the duration to previous release: (a) Ant; (b) Flow4j; (c) Jena; (d) Lucence; (e) Struct; and (f) Xalan.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a new method to measure the stability of software systems and the stability of software modules. Using this approach, we studied the class stability of 6 open-source Java projects. We compared system stability of these 6 Java systems and measured revision rate of each class, which represents the possibility a class could be changed during a revision. Our study found (1) about half of the total classes never changed in the studied lifetime of software evolution; (2) frequent changes occur to small number of classes; and (3) the number of changed classes between current release and next release has no significant relations with the time duration between current release and next release.

In practice, our proposed approach can be used to identify stable and unstable modules, which can help us improve software design quality and reusability. In addition, our proposed metrics, such as retirement rate and revision rate can be easily implemented in a version control and configuration management system, such as Subversion and Git. In our future research, we will create a software tool integrated with GitHub so that it can be used to measure project stability and module stability of any projects on GitHub.

Other similar research has been done to study change sets, which are characterized as the architecture features of the program [22]. Our definition of stable and unstable modules can be used to evaluate these studies. In addition, our

findings about the relation between amount of changes and release time could be further validated with other tools and on other projects.

REFERENCES

1. Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. 2001. "*Does code decay? assessing the evidence from change management data*". IEEE Transactions on Software Engineering, 27(1), 1–12.
2. Mohagheghi, P., Conradi, R., Killi, O. M., & Schwarz, H. 2004. "*An empirical study of software reuse vs. defect-density and stability*". In Proceedings. 26th International Conference on Software Engineering (pp. 282–291). IEEE.
3. Menzies, T., Williams, S., Boehm, B., & Hihn, J. 2009. "*How to avoid drastic software process change (using stochastic stability)*". In Proceedings of the 31st International Conference on Software Engineering (pp. 540–550). IEEE Computer Society.
4. Leavens, G. T., & Sitaraman, M. 2000. "*Foundations of component-based systems*". Cambridge University Press.
5. Dantas, F. 2011. "*Reuse vs. maintainability: revealing the impact of composition code properties*". In Proceedings of the 33rd International Conference on Software Engineering (pp. 1082–1085). ACM.
6. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F. and Dantas, F. 2008. "*Evolving software product lines with aspects*". In Proceedings of the 30th ACM/IEEE International Conference on Software Engineering (pp. 261–270). IEEE.
7. Fayad, M. E., & Altman, A. 2001. "*Thinking objectively: an introduction to software stability*". Communications of the ACM, 44(9), 95–98.
8. Fayad, M. (2002). "*Accomplishing software stability*". Communications of the ACM, 45(1), 111–115.
9. Fayad, M. E., & Singh, S. K. 2010. Software stability model: software product line engineering overhauled. In Proceedings of the 2010 Workshop on Knowledge-Oriented Product Line Engineering (p. 4). ACM.
10. Xavier, P. E., & Naganathan, E. R. 2009. "*Productivity improvement in software projects using 2-dimensional probabilistic software stability model (PSSM)*". ACM SIGSOFT Software Engineering Notes, 34(5), 1–3.
11. Naganathan, E. R., & Eugene, X. P. 2009. "*Architecting autonomic computing systems through probabilistic software stability model (PSSM)*". In Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (pp. 643–648). ACM.
12. Wang, Z. J., Zhan, D. C., & Xu, X. F. 2006. *STCIM: a dynamic granularity oriented and stability based component identification method*". ACM SIGSOFT Software Engineering Notes, 31(3), 1–14.
13. Hamza, H. S. 2005. "*Separation of concerns for evolving systems: a stability-driven approach*". In ACM SIGSOFT Software Engineering Notes (Vol. 30, No. 4, pp. 1–5). ACM.
14. Grosser, D., Sahraoui, H. A., & Valtchev, P. 2002. "*Predicting software stability using case-based reasoning*". In Proceedings of the 17th IEEE International Conference on Automated Software Engineering (pp. 295–298). IEEE.
15. Bevan, J., & Whitehead Jr, E. J. 2003. "*Identification of Software Instabilities*". In WCRE (Vol. 3, p. 134–145).
16. Yau, S. S., & Collofello, J. S. 1980. "*Some stability measures for software maintenance*". IEEE Transactions on Software Engineering, (6), 545–552.
17. Yau, S. S., & Collofello, J. S. 1985. "*Design stability measures for software maintenance*". IEEE Transactions on Software Engineering, (9), 849–856.
18. Kelly, D. 2006. "*A study of design characteristics in evolving software using stability as a criterion*". IEEE Transactions on Software Engineering, 32(5), 315–329.
19. Yu, L., & Ramaswamy, S. 2009. "*Measuring the evolutionary stability of software systems: case studies of Linux and FreeBSD*". IET software, 3(1), 26–36.
20. Threm, D., Yu, L., Ramaswamy, S., & Sudarsan, S. D. 2015. "*Using normalized compression distance to measure the evolutionary stability of software systems*". In Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (pp. 112–120). IEEE.
21. Vasa, R., Lumpe, M., & Jones, A. 2010. "*Helix-Software Evolution Data Set*".

22. Wong, S., Cai, Y., Valetto, G., Simeonov, G., & Sethi, K. 2009. "*Design rule hierarchies and parallelism in software development tasks*". In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (pp. 197–208). IEEE Computer Society.

AUTHORS PROFILE

Dr. Liguo Yu received his PH.D degree in Computer Science from Vanderbilt University. He received his BS degree in Physics from Jilin University. He is currently an associate professor in Computer Science Department, Indiana University South Bend, USA. His research interests are in software engineering, information technology, complexity system, and computer education.

Prof. Yingmei Li is a full professor at Harbin Normal University, China. Her research interest is in software engineering and computer science education.

Dr. Srini Ramaswamy received his PH.D degree from University of Louisiana at Lafayette. His specialty area includes technical/engineering management (software and systems), program/project management, new technology scouting and evaluation, vision / strategy development and execution, software development process adaptation and improvement, industrial automation systems, energy, power & water systems, healthcare systems, healthcare IT, big data, and cloud computing. He is currently a software project manager at ABB Inc.