Assessing refactoring instances and the maintainability benefits of them from version archives

Kenji Fujiwara[1], Kyohei Fushida[2], Norihiro Yoshida[1], and Hajimu Iida[1]

[1] Nara Institute of Science and Technology,
8916-5 Takayama, Ikoma, Nara Japan
{kenji-f,yoshida}@is.naist.jp,iida@itc.naist.jp
[2] NTT DATA Corporation
Toyosu Center Bldg. Annex, 3-8, Toyosu 3-chome, Koto-ku, Tokyo, Japan
fushidak@nttdata.co.jp

Abstract. For the development of high quality software, process quality assessment should be applied into development organization. So far, several process quality assessment methodologies are proposed and applied into a lot of organizations. In this paper, we propose an approach to assess instances of refactoring that is one of the key processes for quality improvement. The proposed approach can be done semi-automatically by investigating version archives in a configuration management system. We applied our proposed method to the Columba project which is an open source software project. The result of our preliminary case study shows that the frequency of defect introduction tends to decrease in the term after frequent refactoring.

Keywords: Refactoring, Mining software repositories, Process assessment

1 Introduction

So far, our research group has proposed several reasonable approaches for fine-grained assessment of the development process from version archives [6, 8]. These approaches can assess ongoing or past development processes and compare their expectations by analyzing version archives of configuration management systems (e.g., CVS, Subversion). Refactoring is a sort of processes, which is defined as the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure[4]. Because the appropriateness of refactoring affects product quality, as well as other sort of processes, project managers must consider the following:

– Did the previous refactoring instances improve the quality of the product?
– Were the previous refactoring instances performed based on the original intention of the developers?
– Was any additional unintentional refactoring performed?

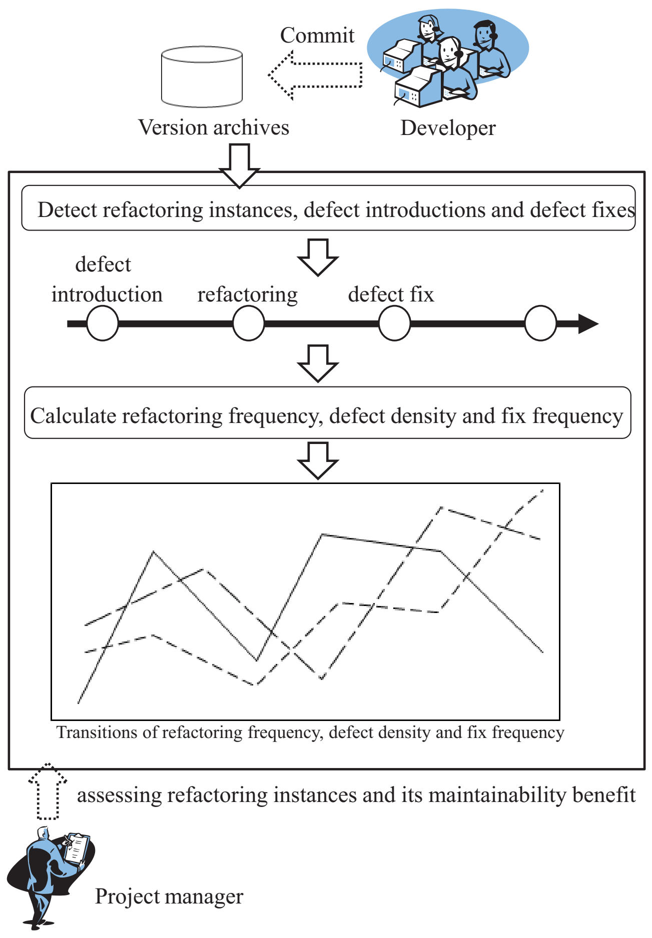
Commit
Version archives
Developer
Detect refactoring instances, defect introductions and defect fixes
defect introduction
refactoring
defect fix
Calculate refactoring frequency, defect density and fix frequency
Transitions of refactoring frequency, defect density and fix frequency
assessing refactoring instances and its maintainability benefit
Project manager

Fig. 1. An overview of our research

However, manual finding of version archives (e.g., commit logs, source code changes) is time consuming for developers, especially project managers who have a little knowledge of the source code of the project.

In this paper, we propose an approach to help a project manager who assess refactoring instances from version archives in a configuration management system (see Figure 1). In our approach, first refactoring instances are detected by the UMLDiff algorithm, then, frequencies of refactorings, defect introductions, and defect fixes are computed for a certain period. Finally, a comparison is performed among the evolutions of those computed frequencies. In addition, we show a preliminary case study of an open source software project Columba to confirm the appropriateness of our approach.

The results of our preliminary case study show that once refactoring instances appear frequently, the frequency of defect introduction tends to decrease.

2 Background

2.1 Refactoring

Refactoring is a technique for improving software design. It is defined as the process of changing the structure of a program without changing its behavior[4]. Various typical refactoring patterns have been cataloged by Fowler[3], who describes that one of refactoring effects is the decrease of software defects.

2.2 Refactoring Detection

We need to investigate when and how refactoring was performed in software development. Approaches to detect refactoring activities are categorized into four types[7]:

Type (a) Using log messages recorded in version control systems.
Type (b) Analyzing source code changes.
Type (c) Observing developers' activity.
Type (d) Tracking usage of refactoring tools.

A type (a) approach detects evidences of refactorings by searching for the word "refactor" and possibly for related words such as "rename" or "extract" from log messages recorded in version control systems. The method assumes developers write performed refactorings into log messages in the version control system. Type (b) analyzes difference of source code between two versions. As a result, evidences of refactorings such as "Rename Field" or "Extract Method" are detected. The method allows us to detect more refactorings than type (a) because it can recover undocumented refactorings. On type (c), researchers observe how developers perform refactoring. Its range of application is narrow because it needs human resources or observation tools. In contrast, it allows us to collect detailed information of refactorings. Type (d) tracks when developers use a refactoring support feature and what part of the source code is changed.

Types (a) and (b), are possible to be applied to existing software projects that adopt a version control systems. However, their results are not complete because they are estimation from development histories. On the other hand, type (c) and (d) cannot be applied to existing projects because they require preparation. For which, we consider the adoption of these types of refactoring detection techniques is highly costly for software development organizations.

3 Investigation Method

3.1 Metrics

In our investigation method, we introduce three metrics:

- ***refactoring frequency***: the number of performing of the refactoring within a certain period

– ***defect density***: the number of introductions of the defect within a certain period
– ***fix frequency***: the number of fixes of the defect within a certain period

In order to define these metrics, we denote the set of revisions recorded in the version control system by V and each revision by v_i. Therefore, V is denoted by $V = [v_1, v_2, \cdots, v_n]$. Then, we let op_i be a source code change from revision v_i to v_{i+1}. In addition, we define the function $r(op_i)$ which returns whether op_i contains a refactoring. $r(op_i)$ returns the value one if op_i contains a refactoring, zero otherwise. Furthermore, we define *refactoring frequency* to be $f_r(j,k)$ as follow:

$$r(op_i) = \begin{cases} 1 & (\text{ if } op_i \text{ contains a refactoring }) \\ 0 & (\text{ otherwise }) \end{cases}$$

$$f_r(j,k) = \frac{\sum_{i=j}^{k-1} r(op_i)}{k-j} \qquad (j < k, \quad v_j, v_k \in V)$$

Similarly, we define the function $d(op_i)$ which returns whether op_i contains a defect introduction and defect density to be $f_d(j,k)$, we also define the function $f(op_i)$ which returns whether op_i contains a fix of the defect and fix frequency to be $f_f(j,k)$ as follow:

$$d(op_i) = \begin{cases} 1 & (\text{if defects are introduced at } v_{i+1}) \\ 0 & (\text{otherwise}) \end{cases}$$

$$f_d(j,k) = \frac{\sum_{i=j}^{k-1} d(op_i)}{k-j} \qquad (j < k, \quad v_j, v_k \in V)$$

$$f(op_i) = \begin{cases} 1 & (\text{if defects are fixed at } v_{i+1}) \\ 0 & (\text{otherwise}) \end{cases}$$

$$f_d(j,k) = \frac{\sum_{i=j}^{k-1} f(op_i)}{k-j} \qquad (j < k, \quad v_j, v_k \in V)$$

3.2 Locating refactoring changes

In order to measure ***refactoring frequency***, we need to know when refactorings were performed to the software. Our method uses the refactoring detection method based on the UMLDiff algorithm[12] designed by Xing et al[13]. The UMLDiff algorithm takes two revisions of the source code and derives the difference among two revisions. It recovers design information from each revision of the source code to derive the difference. In detail, it firstly extracts source code entities such as classes and methods from each revision. Then, it derives the relationships among entities. Finally, it detects the differences such as method/class additions and method/class renames.

Xing et al. also designed the refactoring detection method based on the UMLDiff algorithm[13]. It detects refactorings which performed on the changes from one revision to another one by analyzing the output of the UMLDiff algorithm. The method takes two revisions as input of the UMLDiff algorithm and applies the algorithm. Then, it analyzes the output to search the set of differences which is a candidate for an evidence of refactoring. It is promising to derive only refactoring instances and ignore non-refactoring changes because the detection mechanism in the UMLDiff algorithm follows refactoring patterns written by Fowler[4].

In this paper, we locate changes which contain refactoring instances by the following steps:

Step 1. Divide all revisions by every n revisions into k groups.
Step 2. For each k group, use the first revision and the final revision as inputs for the UMLDiff algorithm and apply the algorithm.
Step 3. Locate the refactoring change which contains refactoring instances detected in the previous step.

3.3 Locating fix and fix-inducing changes

Our method measures ***defect density*** (which is the number of introductions of defect within a certain period). In order to measure ***defect density*** and ***fix frequency***, we need to know when defects were introduced to the software and when defects were fixed. Śliwerski et al. designed the SZZ algorithm which is an algorithm to extract fix-inducing changes from version control systems[10]. Their algorithm links information recorded in the issue tracking system and the version control system. We consider fix-inducing changes as introductions of defects.

The SZZ algorithm links the issue and commit by following steps:

Step 1. Firstly, the SZZ algorithm searches candidates of the fix commits by using commit messages. For instance, it search the keyword which related to the fixes such as "fixes", "closed" and so on. If the commit message contains the keywords, SZZ searches defect ID from the commit message.
Step 2. Then, it filter out suspected candidates. SZZ use the criterion of filtering as follow.

- Does the defect ID exist in the issue tracking system?
- Has the fix completely done?

As a result of this step, we got the link of defects and commits.
Step 2. Finally, SZZ trace the changes to detect when the fix-inducing change was introduced by using the version control system.

3.4 Requirements of Target Projects

The target project of our method requires to be adopted the version control system and the issue tracking system. In addition, to apply the SZZ algorithm, the project needs to collaborate with the version control system and the issue tracking system.

4 Case Study

We applied our proposed method to the Columba[3] project which is an open source software project. Table 1 shows an overview of the Columba project. The Columba project uses Subversion[4] which is a version control system and an issue tracking system provided by SourceForge.net.

Table 1. An over view of the Columba project

category	development period	#revisions	LOC of the final revision
Mail Client	2006/7/9 – 2011/7/11	458	192,941

4.1 Study Procedure

We measured three the metrics described in Section 3 from the development history of the Columba project by the following steps:

Step 1. Located refactoring changes by the UMLDiff based refactoring detection method.
Step 2. Located fix and fix-inducing changes by the SZZ algorithm.
Step 3. Calculated three metrics from the results of the Step 1 and Step 2. Each metric was calculated every 25 revisions. For instance, we calculated a refactoring frequency from revision 25 to 50.

Step 1. Locating refactoring changes Using the procedures described in Section 3.2, we detected refactoring changes from the archive of the Columba. When using the UMLDiff algorithm, we used the following approaches to realize precise detection of refactoring instances.

(a) Give the first and the last revisions to the UMLDiff algorithm for input.
(b) Pick-up every five revisions from the archive, and then give each pair of the picked-up revisions next to each other.

In order to locate refactoring changes from the result of the UMLDIff algorithm, we manually analyzed source code archives by using the browsing differences feature of Subversion. At that time, we removed false positives. For instance, an Extract Method refactoring was detected by the UMLDiff algorithm, but it was not an actual refactoring because the name of the extracted method was similar to an the existing method, but its body was quite different.

[3] http://sourceforge.net/projects/columba/
[4] http://subversion.apache.org/

Table 2. List of detected refactorings

Kind of refactoring	#Refactorings Approach A	Approach B	total
Convert top level to inner	1	0	1
Die-hard/legacy classes	1	1	1
Downcast type parameter	3	0	3
Encapsulate field (get)	1	0	1
Extract class	2	1	2
Extract method	7	2	7
Extract subsystem/package	3	2	3
Extract super interface	0	6	6
Generalize type (method)	10	9	10
Generalize type (field)	2	2	2
Generalize type (parameter)	42	40	42
Information hiding	6	0	6
Inline subsystem/package	1	2	2
Move method/field/behavior	0	12	12
Move subsystem/package/class	0	12	12
Pull-up method/field/behaivior	5	4	5
Push-down method/field/behaivior	1	0	1
total	85	93	116

Step 2. Locating fix and fix-inducing changes In order to locate fix and fix-inducing changes, we used a tool implemented the SZZ algorithm. The columba project has a rule of commit log strictly. For a bug fix commit, the tag word [**bug**] or [**fix**] is added to the log. Therefore, we recognized a commit whose log contains those tag words as a fix change. Then, we located the fix-inducing change from those fix changes.

4.2 Result

Table 2 shows the result of the UMLDiff based refactoring detection method. The result does not include false positives. The columns of *Approach A* and *B* correspond to the approaches described in the previous section. *Total* column represents the union of the result of approaches A and B. 14 kinds of refactorings were extracted, and the total number of extracted refactoring was 116. As a result of the SZZ algorithm, 322 fix changes and 243 fix-inducing changes were extracted.

Figure 2 shows the transition of the ***refactoring frequency***, ***defect density*** and ***fix frequency*** of the Columba project. X-axis indicates the revision number, and y-axis indicates the value of three metrics. As described in the previous section, each value of three metrics was calculated and plotted every 25 revisions. For instance, the value of the ***fix frequency*** from revision 250 to revision 275 is 0.2.

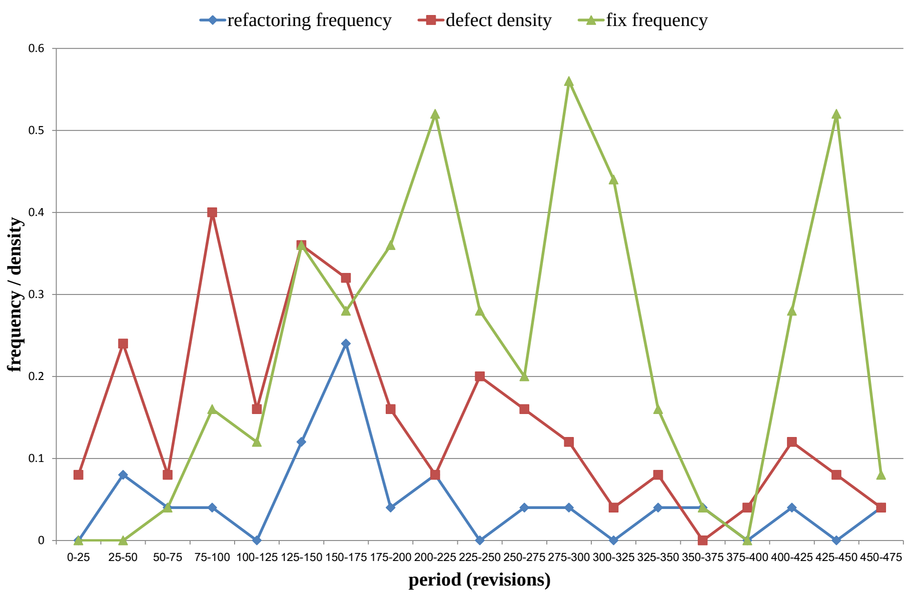

Fig. 2. Transition of the refactoring frequency, defect density and fix frequency in the Columba

5 Discussion

We confirmed following two characteristics from Figure 2.

(a) ***Defect density*** tends to decrease from the most refactored period (from 150 revision to 175 revision).
(b) ***Fix frequency*** tends to increase after the refactoring frequency is high through out the project.

The average of the ***defect density*** from revision zero to revision 175 is 0.23. In contrast, the average from revision 175 to revision 458 is 0.09. Furthermore, commit logs in the most refactored period (from revision 150 to revision 175) include evidence of big refactoring[5]. We consider that the decrease of the defect density after a big refactoring is due to improvement of the source code by performing refactoring. On the other hand, we also consider that defect density decreased because the Columba project has entered to the maintenance phase.

[5] The commit logs from revision 151 to revision 155 were exactly same. These were *[intern] big refactoring of the filter stuff. There was a interface created and all the required interfaces moved to core-api. After that all classes which using the filter stuff now refactored to use the interfaces instead the implementation. Now the mail-api compiles without problems with maven2.*

However, the version 1.4 of the Columba was released in the revision after about 100 revisions from the point a big refactoring occurred.

We consider that the performing of refactoring improved source code readability and then the developers could find and fix defects easily, however, we also consider developers performed refactoring in order to be source code more flexible and then performed fixes. In contrast, there is a possibility these fixes are due to failure of refactorings.

6 Threats to Validity

Preciseness of the refactoring detection method Our proposed method relies on preciseness of the refactoring detection method. The preciseness depends on the granularity of the divisions of the revisions. In fact, the result of *Approach A* and *B* were different. Thus, we probably left out some refactoring changes (e.g., file addition after revision r_n, delection before revision r_m where (r_n, r_m) denotes the input of the UMLDiff Algorithm). However, we believe that our result will not change dramatically if we can add detect ignored refactorings.

Preciseness of the SZZ algorithm Our proposed method also relies on preciseness of the fix and fix-inducing detection method. In our case study, that preciseness depends on the accuracy of the developers record activities because we located fix changes based on the commit logs of the Columba project. Some studies report the accuracy of existing detection methods for locating fix and fix-inducing changes based on commit logs depend on the quality of commit log and data in bug tracking systems[1, 2]. Hence, they also report that these methods cannot recover all fix and fix-inducing changes completely. For locating fix-inducing changes, the SZZ algorithm traces back changes of the source code by using version control systems. Thus, the algorithm does not support certain type of changes which consist in only deletions. However, our result does not suffer from those changes because such cases are rare.

Granularity of refactorings, defects and fixes Our proposed method focuses on whether a refactoring, a defect introduction and a fix are performed in each change. Therefore, our method treats a single refactoring change and a multiple refactoring change in the same manner. We consider that treating the number of refactoring in our proposed method is not appropriate because the kinds of refactoring detected by the UMLDiff based method has much granularity. Similarly, for fix and fix-introducing changes, we focused on whether those changes were performed or not.

7 Related Work

Ratzinger et al. analyzed the relationship between refactorings and defects[9]. They used log messages in a version control system to detect refactoring. They reported that when the number of refactorings is high over a certain period of the software project, defect introducing changes would decrease over successive period. Therefore, they concluded that refactoring reduces defect introduction. We investigate details of time relation between refactorings and defects.

Kim et al. investigated the role of refactorings among three open source software[5]. They extracted refactoring histories from the software development history, and then evaluated the role from the following viewpoints:

– Are there more bug fixes after refactorings?
– Do refactorings improve developer productivity?
– Do refactorings facilitate bug fixes?
– How many refactorings are performed before major releases?

As a result, they reported that the bug fix rate of 5 revisions before refactoring is around 26.1% for Eclipse JDT, and refactorings decrease time of bug fixes. In addition, they reported refactorings tend to be performed together with bug fixes and there are many refactoring before major releases. Their research focused on the effect of the refactorings on the bug fixes. On the other hand, our approach focused on the effect on the defect introducing changes.

In order to detect refactorings, Weißgerber and Diehl presented a method by using syntactical and signature information of the source code[11]. Their method extracts refactoring candidates by that information, and then ranks them based on code clone detection technique. Compared with the method based on the UMLDiff algorithm, their method can detect less kinds of refactoring(10 kinds). In our study, we use the method based on the UMLDiff algorithm because the method can detect 33 kinds of refactoring.

8 Conclusion and Future Work

We proposed an approach to assess refactoring instances from version archives. Our proposed method detects refactoring instances using a refactoring detection method based on the UMLDiff algorithm. It also detects defect introductions and defect fixes using the SZZ algorithm.

As a result of our preliminary case study using the Columba project, our method is promising to support the assessment of the assessing refactoring instances and its maintainability benefit.

As a future work, we plan to apply our method to another open source software project. However, in the case study, the run-time for the UMLDiff algorithm based refactoring detection was approximately 19 hours. The computation time of the detection method is roughly proportional to the scale of the target project. Therefore, in order to apply our method to the project which is larger than the Columba project, we need to develop a refactoring detection tool which detect refactoring instances among two versions more quickly.

9 Acknowledgments

This research was supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (No.22500027).

References

1. Bachmann, A., Bird, C., Rahman, F., Devanbu, P., Bernstein, A.: The missing links: bugs and bug-fix commits. In: Proc. the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering(FSE 2010). p. 97 (2010)
2. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced?: bias in bug-fix datasets. In: Proc. the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE2009). pp. 121–130 (2009)
3. Fowler, M.: Refactoring home page. http://refactoring.com/
4. Fowler, M.: Refactoring: improving the design of exsiting code. Addison Wesley (1999)
5. Kim, M., Cai, D., Kim, S.: An empirical investigation into the role of api-level refactorings during software evolution. In: Proc. the 33rd International Conference on Software Engineering(ICSE 2011). pp. 151–160 (2011)
6. Kula, R.G., Fushida, K., Kawaguchi, S., Iida, H.: Analysis of bug fixing processes using program slicing metrics. Proc. the 11th Product-Focused Software Process Improvement(PROFES 2010) LNCS 6156, 032–046 (6 2010)
7. Murphy-Hill, E., Black, A.P., Dig, D., Parnin, C.: Gathering refactoring data: a comparison of four methods. In: Proc. the 2nd ACM Workshop on Refactoring Tools(WRT 2008). pp. 1–5 (2008)
8. Ohkura, K., Goto, K., Hanakawa, N., Kawaguchi, S., Iida, H.: Project replayer with email analysis – revealing contexts in software development. In: Proc. the 13th Asia Pacific Software Engineering Conference (APSEC 2006). pp. 453–460 (2006)
9. Ratzinger, J., Sigmund, T., Gall, H.C.: On the relation of refactorings and software defect prediction. In: Proc. the 5th Working Conference on Mining Software Repositories(MSR 2008). pp. 35–38 (2008)
10. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proc. the 2nd International Workshop on Mining Software Repositories(MSR 2005). pp. 1–5 (2005)
11. Weißgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: Proc. the 21st IEEE/ACM International Conference on Automated Software Engineering(ASE 2006). pp. 231–240 (2006)
12. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: Proc. the 20th IEEE/ACM International Conference on Automated Software Engineering(ASE 2005). pp. 54–65 (2005)
13. Xing, Z., Stroulia, E.: Refactoring Detection based on UMLDiff Change-Facts Queries. In: Proc. the 13th Working Conference on Reverse Engineering(WCRE 2006). pp. 263–274 (2006)