Refactoring Patterns Study in Code Clones during Software Evolution

Jaweria Kanwal
Quaid-i-Azam University
Islamabad, Pakistan
kjaweria09@yahoo.com

Katsuro Inoue
Osaka University
Osaka, Japan
inoue@ist.osaka-u.ac.jp

Onaiza Maqbool
Quaid-i-Azam University
Islamabad, Pakistan
onaiza@qau.edu.pk

***Abstract*—To investigate how code clones are handled by developers when they perform refactorings during software releases, we performed a longitudinal study on different versions of five Java systems. Our results show that a small proportion of code clones are refactored during the releases and code clones of same clone class are refactored consistently.**

***Index Terms*—Software refactoring, code clones, software maintenance and evolution.**

I. Introduction

Code clones—identical code fragments in software—are the result of developer's copy-paste-modify activity during software development. Sometimes clones cause an additional maintenance effort such as a change in one clone fragment may cause change in other clone fragments [1]. For this reason clones need to be detected for better software maintenance [2]. One of the maintenance tasks is to remove the clones from the system through refactoring. Refactoring is a widely used technique during software maintenance to improve code quality. It is the process of improving the internal structure of a software system without affecting its overall behavior [3]. There are different refactoring solutions for different types of code smells. Most well-known refactoring patterns have been proposed by Fowler [3]. These are 65 refactoring patterns e.g. move_method, extract_method, extract_interface and add_parameter.

Refactoring on code clone instances may/may not remove them from the system. For example, if the refactoring task is extract method, which takes the code fragment from the method and puts it in another method, then clones may be removed from the system as a result of this refactoring task but if only add_parameter task is performed on a clone instance, (which only changes the function signature) then it may improve the design of system but does not remove the clones from the system.

In literature, code clone evolution has been studied to investigate their change behavior e.g. how long clones remain in the system, whether they change consistently or inconsistently. Kim et al. [4] concluded that all clones are not refactorable and hence remain in the system till last release of the software. Among the disappearing clones, most of them disappear within few check-ins, thus reducing the need of extensive refactoring of code clones. But there is no work in literature to study the clone evolution in terms of actual refactorings performed on clones between software releases.

In this paper, we perform a longitudinal study to investigate code clone evolution in terms of refactoring tasks performed in the subsequent versions of software. Investigation of actual refactorings performed on code clones gives historical evidence of how code clones are treated by developers when they perform refactorings in software. In this paper, we focus on following two research questions:

1) How often are code clones refactored in software?
2) How often are clones refactored consistently?

Inspection of these research questions gives historical background of clone refactorings between software releases which will help software maintainers in taking code clone refactoring decisions in future releases. To the best of our knowledge there is no work on the historical study of clones in terms of actual refactorings performed on them. Our work is novel in this regard.

II. Experimental Setup and Results

In this section we describe the study setup for addressing each research question and analyze the results. For clone detection, we used CloneMiner [5] tool which uses a token based clone detection technique. We set the similarity threshold as 30 tokens for code clones. In order to identify refactorings performed between two versions, we use Ref-Finder [6] tool which extracts well-known refactoring patterns [3] between consecutive versions of software. Refactoring tasks detected by Ref-Finder tool are reported at method level, class level or interface level, referred to as refactored entities in this paper. The methods and files where clones reside are referred as clone entities. We developed an application for mapping clone entities with refactored entities.

For experiments, we selected five Java systems i.e. JHotDraw, Guava, Jabref, JFreeChart and Xerces_J. These are well known systems and have been studied previously for clone research. We selected some latest versions of these systems. Starting version number of each system is different for different systems, so for discussing the experimental results, we used the general notation of representing each version as V_n and next version as V_{n+1}. For JHotDraw starting version (V_n) is 7.1, for Guava 14.0.1, for Jabref 2.11.1, for JFreeChart, 1.0.12 and for Xerces_J, 2.5.

A. How Often Are Code Clones Refactored in Software?

Table 1 represents the percentage of refactored clones in each version of five Java systems. Average number of refactored clone instances in each system is also presented.

Table 1 shows that number of refactored clones is different in different versions. In some versions of JHotDraw, Guava and Xerces_J, number of refactored clones are greater than other

versions. This indicates that developers are concerned with the quality of software and paying attention to clones.

For example, in Xerces_j, there is a great variation in refactored clones. In version V_{n+3}, refactored clones are 34% but in next version they are 2%. This shows that there is no consistency of refactoring code clones between consecutive versions. This variation in clone refactorings depends upon the number of actual refactorings performed in these versions. In Xerces_J, in version V_{n+3} actual refactorings performed are 496 whereas in next version number of refactorings is only seven. This shows that number of refactorings performed in versions varies widely. One possible reason of this variation may be the release time duration between the versions. Release duration between version V_{n+2} and V_{n+3} is four months and release duration between version V_{n+3} and V_{n+4} is only one month. Developers may not be able to perform refactorings in such a short time period between the two releases.

On average, the number of refactored clones in different versions in these five systems ranges from 2% to 24%. This shows that a very small proportion of code clones are refactored between consecutive versions.

TABLE 1: REFACTORED CLONES IN JAVA SYSTEMS

Versions	JHotDraw	Guava	Jabref	JFreeChart	Xerces_J
V_n	27. 4%	47.0%	6.0%	3.8%	32.0%
V_{n+1}	19.1%	0.8%	9.0%	4.5%	24.8%
V_{n+2}	21.1%	0.5%	4.8%	1.3%	34.0%
V_{n+3}	21.4%	0.8%	17.0%	0.5%	34.2%
V_{n+4}	33.3%	6.2%	6.7%	1.2%	2.0%
V_{n+5}	-	8.2%	0.4%	0.6%	10.5%
V_{n+6}	-	-	3.4%	-	7.1%
V_{n+7}	-	-	-	-	8.2%
Average	24.2%	10.5%	6.7%	2.0%	19.1%

B. How Often Are Clones Refactored Consistently?

Refactored clones are analyzed further to know whether these are consistent or inconsistent clone refactorings. If *all instances* of a clone class are refactored and same refactoring task e.g. add_parameter is applied on them then it will be a consistent refactoring. Consistent clone refactoring represents that developers are aware of clone entities in the software. We can call consistent refactoring as clone-aware refactoring. If only a fraction of clone instances is refactored, then it is called in-consistent refactoring. One possible reason of this inconsistent refactoring may be unawareness of developers of exact clone information in the system. Or it may be a design issue or programming language limitation that other two instances cannot be refactored similarly.

To determine how often clones are refactored consistently in software, we measured the number of clone classes that are consistently refactored in different versions.

Table 2 shows the percentage of clones that are refactored consistently among the total refactored clones in different versions of five Java systems. In some versions 100% clones are refactored consistently whereas in some versions, only 10% clone refactorings are consistent. In JHotDraw, more than 52% clones are refactored consistently in each version.

Table 2 also shows the average number of consistently refactored clones for each system. Highest average of consistently refactored clones among the five systems is 62 in JHotDraw and lowest is 38% in JFreeChart. In JHotDraw, Guava and Xerces_J, consistent refactorings of clones are more than 52% which shows that developers of these systems are aware of the presence of clones while performing refactoring tasks.

TABLE 2: PERCENTAGE OF CONSISTENT REFACTORINGS AMONG TOTAL CLONE REFACTORINGS

Versions	JHotDraw	Guava	Jabref	JFreeChart	Xerces_J
V_n	62.7%	62.0%	50.0%	32.0%	53.1%
V_{n+1}	56.7%	20.0%	80.0%	42.0%	40.0%
V_{n+2}	73.5%	100%	42.8%	32.0%	47.1%
V_{n+3}	52.0%	45.4%	70.4%	42.1%	49.0%
V_{n+4}	66.8%	45.0%	32.3%	44.1%	100%
V_{n+5}	-	48.0%	10.0%	36.0%	34.2%
V_{n+6}	-	-	45.1%	-	44.4%
V_{n+7}	-	-	-	-	52.3%
Average	62.2%	53.2%	47.2%	38.0%	52.0%

III. CONCLUSIONS

In this paper, we studied code clone evolution by investigating the refactoring patterns applied on code clones. Our results showed that a small portion of code clones are refactored during the releases. More than 40% clones are refactored consistently in most of the versions. Consistent refactoring of clones represents that in many cases developers are aware of cloning in the system and that they intentionally use this copy paste approach.

In the future we will investigate variation in the frequency of clone refactorings, utility of clone refactorings on software maintenance and assessing their impact in next versions.

ACKNOWLEDGMENT

This work was supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) JP25220003, and by Osaka University Program for Promoting International Joint Research.

This research was also supported by Higher Education Commission, Pakistan.

REFERENCES

[1] C. Kapser and M. Godfrey. ""Cloning considered harmful" considered harmful: patterns of cloning in software," Journal of Empirical Software Engineering, vol. 13, pp. 645-692, 2008, Springer.

[2] T. Kamiya, S. Kusumoto and K. Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, vol. 28, pp. 654-670, 2002.

[3] M. Fowler, "Refactoring: Improving the Design of Existing Programs," Addison-Wesley, 1999.

[4] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. "An Empirical Study of Code Clone Genealogies," ACM SIGSOFT Software Engineering Notes, vol. 30, pp. 187-196, 2005.

[5] H. Basit, and S. Jarzabek. "A data mining approach for detecting higher-level clones in software," IEEE Transactions on Software Engineering, vol. 35, pp. 497-514, 2009.

[6] M. Kim, M. Gee, A. Loh and N. Rachatasumrit. "Ref-Finder: a refactoring reconstruction tool based on logic query templates," Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, pp. 371-372, ACM, 2010.