Active Support for Clone Refactoring: A Perspective

Norihiro Yoshida
Nara Institute of Science and Technology, Japan
yoshida@is.naist.jp

Eunjong Choi Katsuro Inoue
Osaka University, Japan
{ejchoi, inoue}@ist.osaka-u.ac.jp

Abstract

Clone refactoring (merging duplicate code) is a promising solution to improve the maintainability of source code. In this position paper, we discuss directions towards the advancement of clone refactoring, and show a perspective of active support based on online analysis of code modification on an editor of IDE.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming

Keywords Refactoring, Code clone, IDE

1. Introduction

Clone refactoring is a series of the code transformation to merge similar parts of source code into a single program unit (e.g., Java method, C++ function). It is aimed at improving the maintainability of the source code.

Several tools for non-clone refactoring monitor code modification on the fly, and then dynamically utilize the monitoring result for refactoring support. For example, BeneFactor [8] and WitchDoctor [6] detect the beginning of refactoring, and proactively recommend a series of code transformation to complete the refactoring. On the other hand, in the case of clone refactoring, dynamic support based on the online analysis of code modification is very limited despite the complexity of its process. In clone refactoring, a developer detects all of clones that are similar to each other, and then determine whether those clones should be refactored according to maintainability impact and the difficulty of the refactoring. Finally, he/she merges those clones into a single module if possible and appropriate.

In this position paper, we argue research objectives and directions towards the advancement of clone refactoring from the perspective of dynamic support based on online analysis of code modification.

WRT '13, October 27, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2541348.2541352

2. Related work

Several approaches have been proposed on the identification and the categorization of clone refactoring opportunities in source code. Balazinska et al. [1] proposed an approach for supporting clone refactoring by categorizing code clones based on the differences of them. Baxter et al. [2] have developed a clone detection tool CloneDR based on AST similarity. CloneDR derives only syntactically-complete clones that can be easily refactored. Hotta et al. [12] focused on Form Template Method refactoring pattern [7], and proposed a specialized approach to identifying its opportunities. For the prioritization of clone refactoring opportunities, Higo et al. [11] and Choi et al. [5] proposed metric-based approaches respectively. Also, search-based approaches have been proposed for scheduling clone refactoring based on its benefit and effort [3, 17, 24].

A few approaches have been proposed on automatic code transformation for clone refactoring. Tairas and Gray [21] have developed an Eclipse plugin CEDAR that realizes automatic code transformation for clone refactoring using the refactoring feature of Eclipse. For Form Template Method refactoring, Juillerat and Hirsbrunner [14] proposed an automatic code transformation approach based on AST comparison.

An Eclipse plug-in JSync [18] will be invoked by explicit requests or SVN commits of developers, and then report potential inconsistent modifications to clones. JSync is one of the notable tools that support clone maintenance dynamically. However, it is unresponsive to code modification on an editor of IDE, so still differs from the online analysis of code modification.

To our knowledge, there is no tool that supports clone refactoring based on the online analysis of code modification. In next section, we discuss challenges in the tool development for clone refactoring from the perspective of dynamic support based on the online analysis of code modification.

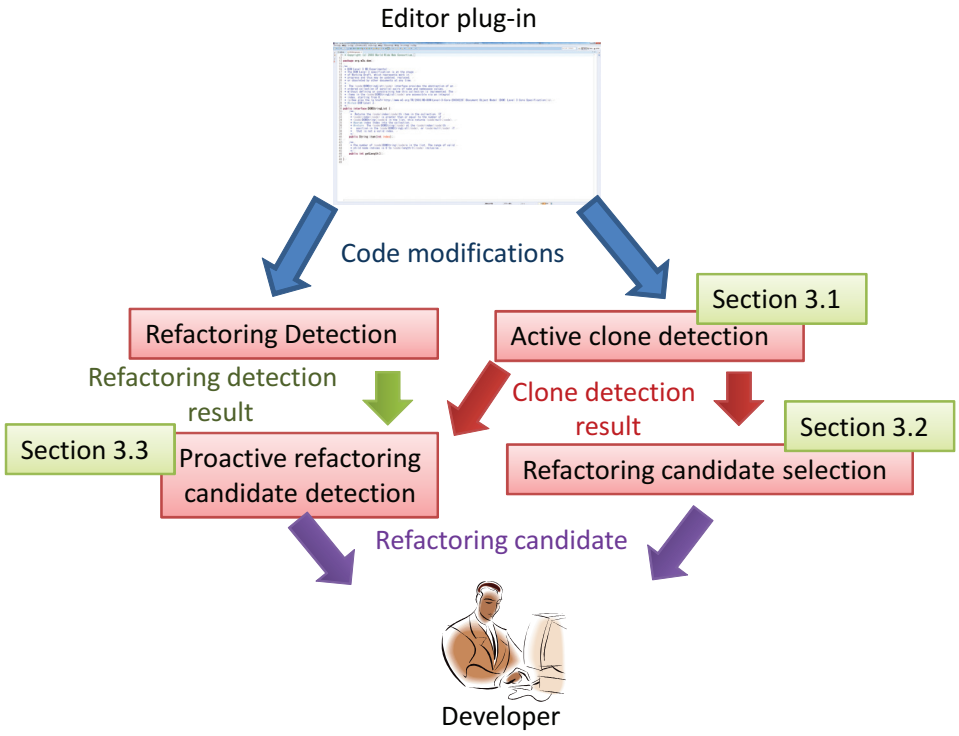


Figure 1. An overview of the research directions that we argue

3. Research Directions

Figure 1 illustrates an overview of the research directions that we argue in this section. In section 3.1, we argue approaches that perform clone detection incrementally when a developer modifies source code on an editor plug-in of an IDE (e.g., Eclipse). And then, section 3.2 presents approaches that fiter out clones that should be detected from the output of section 3.1. Finally, in section 3.3, we argue approaches to detecting refactoring based on the analysis of code modifications on a code editor, and then recommending to refactor clones of the refactored code at one time.

3.1 Active clone detection

According to our experience in the application of clone related tools into industry, developers are interested in only newly-appeared clones because of the following reasons [23]:

- Previously-existing clones are often manually verified and recognized as harmless for the maintainability of the source code.
- Editing previously-existing clones often leads reperformance of large-scale test (e.g., integration/system test) that consumes much time.

Therefore, it is necessary to develop a tool that performs clone detection incrementally when a developer modifies source code on an editor plug-in of an IDE. Although several tools have been developed on incremental clone detection that identifies newly-changed clones by comparing current and previous revisions [9, 13], the challenges still remains on when and how to catch code modification, how to reflect it to incremental clone detection implemented in IDE.

Another challenge is the development of a near real time clone detection technique that can be invoked by modification in a source code editor, and then can finish the detection at by the timing of next modification. Also, the detection technique is required to cover various types of clones. This is based on the practice that clones are frequently merged even if not only variations exist in identifiers, literals, and types but also modifications are made such as changed, added or removed statements [4, 7, 20].

The output of this step is passed to the steps described in section 3.2 and 3.3 respectively.

3.2 Refactoring Candidate Selection

Several studies pointed out that clones are not always refactoring candidates [10, 15, 16] even in the case of newly-appeared one [23]. For precise refactoring candidate selection, the following two types of clones should be filtered out from the output of the clone detection described in the section 3.1 [11, 23].

At first, syntactically incomplete clones should be not be detected [11, 23]. According to the experience in applying clone detection to industrial software development [23],

clones include whole parts of loop or branch statements were considered as ones that should be merged. Meanwhile, developers rarely recognize clones include only parts of loop or branch statements as ones should be merged because it is difficult to merge syntactically incomplete clones.

Also, according to the industrial experience [23], in the case of clones that were newly-appeared by adding new code, developers frequently recognize them as ones should be merged. On the other hand, clones are sometimes accidentally created by only the replacement or the deletion of statements. In other words, even if no line is added to a code fragment, it sometimes becomes a code clone together with other code fragments when one (or greater than one) character is changed in it. In such case, developers mostly decide to leave those clones as it is.

One of the challenges is discovering further heuristics for eliminating clones that should not be detected. Also, another challenge is that how to visualize the refactoring candidates derived in this step. A straightforward visualization is listing them in a view of IDE, and updating the list once a developer modifies source code.

3.3 Proactive Refactoring Candidate Detection

When a developer applies a well-defined refactoring transformation to a code fragment, it implies that he/she should consider whether or not to refactor the clones of the code fragment. For example, when a developer extracts a code fragment as a method, he/she should consider whether or not to extract the clones of the code fragment, and then merge them into the same method at one time.

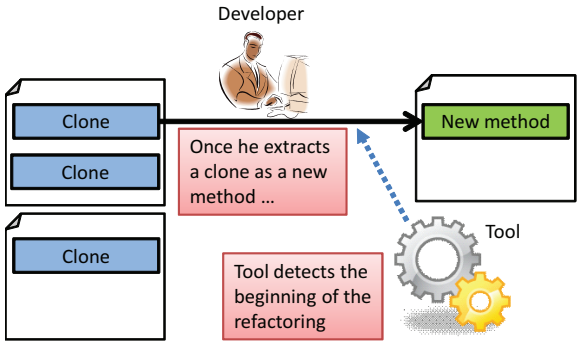
Fortunately, much research has been done on the detection of refactoring between previous and current revisions [19, 22]. Those refactoring detection techniques are promising to identify the behaviors of a developer who refactor a code fragment.

Figure 2 illustrates an example of proactive refactoring candidate detection using refactoring detection technique. At first, once a developer extracts a clone as a new method, the tool for proactive refactoring candidate detection identifies the refactoring (Figure 2(a)) , and then discovers the clones corresponding to the refactored clone (Figure 2(b)) according to the output of the clone detection described in section 3.1. Finally, the tool recommends him to merge those clones into the new method (Figure 2(c)). In Figure 2(c), a brown dashed arrow means that a clone is replaced by a call of the new method.

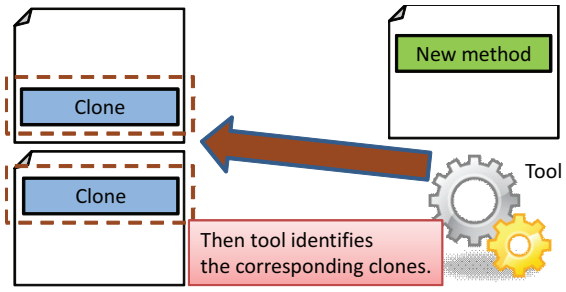
UI for recommending refactoring candidates is a challenge as well as the step in section 3.2.

4. Summary

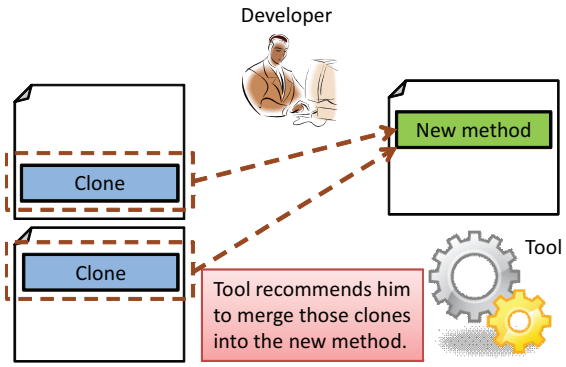
In this position paper, we reviewed studies related with clone refactoring, and argued research objectives and directions towards the advancement of clone refactoring from the perspective of dynamic support based on online analysis of code modification.



(a) Refactoring detection



(b) Identification of the corresponding clones



(c) Recommendation of refactoring candidates

Figure 2. Proactive Refactoring Candidate Detection

Acknowledgments

We express our great thanks to Dr. Zhenchang Xing at Nanyang Technological University for helpful comments. This work was supported by JSPS KAKENHI Grant Numbers 21240002 and 25220003.

References

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. of METRICS '99*, pages 292–303, 1999.

[2] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of ICSM '98*, pages 368–377, 1998.

[3] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *GECCO*, pages 1885–1892, 2006.

[4] E. Choi, N. Yoshida, and K. Inoue. What kind of and how clones are refactored?: A case study of three OSS projects. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 1–7, 2012.

[5] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano. Extracting code clones for refactoring using combinations of clone metrics. In *IWSC*, pages 7–13, 2011.

[6] S. R. Foster, W. G. Griswold, and S. Lerne. WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings. In *Proc. of ICSE*, pages 222–232, 2012.

[7] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[8] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proc. of ICSE*, pages 211–221, 2012.

[9] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proc. of ICSE*, pages 311–320, 2011.

[10] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Information & Software Technology*, 49(9-10):985–998, 2007.

[11] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *Journal of Software Maintenance and Evolution*, 20(6):435–461, 2008.

[12] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *CSMR*, pages 53–62, 2012.

[13] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Proc. of ICSM*, pages 1–9, 2010.

[14] N. Juillerat and B. Hirsbrunner. Toward an implementation of the "form template method" refactoring. In *SCAM*, pages 81–90, 2007.

[15] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[16] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *ESEC/SIGSOFT FSE*, pages 187–196, 2005.

[17] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Software: Practice and Experience*, 41(5), 2011.

[18] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Trans. Softw. Eng.*, 38(5):1008–1026, 2012.

[19] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proc. of ICSM*, 2010.

[20] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[21] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information & Software Technology*, 54(12):1297–1307, 2012.

[22] Z. Xing and E. Stroulia. Refactoring detection based on umldiff change-facts queries. In *Proc. of WCRE*, pages 263–274, 2006.

[23] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano. Applying clone change notification system into an industrial development process. In *ICPC*, pages 199–206, 2013.

[24] M. F. Zibran and C. K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *SCAM*, pages 105–114, 2011.