2012 4th International Conference on Computer Research and Development
IPCSIT vol.39 (2012) © (2012) IACSIT Press, Singapore

Identifying Core Asset Candidates in Existing Products Source Code

Tanit Reantragoon and Pornsiri Muenchaisri [+]

Center of Excellence in Software Engineering, Department of Computer Engineering
Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand

Abstract. Since late 1990s, software product line (SPL) was proposed as an approach to manage a group of related software systematically to reduce cost and effort in software development. The key activities of SPL are developing reusable artifacts as assets and reusing them to assemble and tailor to meet products individual needs. To develop assets, analysis of commonality and variability of products is an important part in core asset identification. However, a domain expert may face difficulty in identifying and extracting common parts from existing products as core assets if there are some differences within those similar parts. In this paper, we propose an approach to find commonality from existing products source-code using code clone detection technique, and then recommend how to extract them into core asset candidates. The proposed approach is evaluated with several versions of software. The result of the evaluation is presented.

Keywords: software product line, code clone detection, design pattern

1. Introduction

In attention of developers to increase productivity, improve quality of systems and reduce cost, software reuse is one of the best practical choices to achieve those goals. Software Product Line (SPL) [6] is a promising approach of software development with strategic reuse limited only for the specific group of software, called "product line" and managing them systematically. As features are scoped out by family of products, common and specific artifacts, called "core asset", are managed to satisfy individual products' requirements. Due to essence of core assets, identifying commonality of existing products is needed to develop them, especially in reactive SPL. Many researches related to this topic mostly focused on architecture-level or process oriented [14], [15].

Another basic code reuse practice which is usually used in development of software is code cloning [8], [9], [10]. Code clone is a code fragment that is copied from one place to another place to reuse it, with or without modification. Code clone detection is applied in several application fields [3], and also in SPL [11], [12]. In core asset development, some researches apply code clone detection to find commonality in existing products [1], [13]. They use code clone detection for similarity measure calculation in order to indicate the need of pruning in each part. However, developers usually modify copied code fragments, which make them different. Yoshiki et al. propose a classification scheme of multiple code clone detection techniques [19]. They focus on code clones with some modification or "gap" [18], as they may introduce bugs if they're not modified properly.

In this paper, we propose an approach to identify core asset candidates of existing products by finding commonality in code-level using a code clone detection technique. Detection results are classified by clone types [19] with NiCad [7] detection tool. Type-2 and type-3 code clones are formed into core assets by refactoring them into template method [21]. The remaining of this paper is organized as follow. Section 2 describes an approach in details while results of experiment are discussed in section 3. Lastly, Section 4 concludes the paper and suggests future works.

[+] Corresponding author. *E-mail address*: Tanit.Re@student.chula.ac.th, Pornsiri.Mu@chula.ac.th

2. Identifying Core Asset Candidates

As described in [2], [4], [5], code clones are roughly categorized into 4 groups without considering in comments and formatting layouts. Type-1 clones are identical code fragments, while type-2 clones are similar to type-1 clones but some identifiers have renamed. Copied code fragments which have some statements added, removed or changed are called type-3 clones. Type-4 clones are code fragments which have the same behaviour or output, from different implementations. We focus only type-1, type-2 and type-3 clones in this research. Detected code clones with some differences such as modified identifiers or statements, make extracting them to core asset more complicated. In Fig. 1(a), *action()* method in system *A* and *B* have similar sequences of *X1* and *X2* fragments but some differences in *diffA* and *diffB* code fragment. To extract common part from *action()* method, developers have to make a decision how to deal with these differences. Our approach aims to identify core asset candidates and also take differences into account. There are two main steps. Firstly, we detect code clones and classify them into type of clones by using a multiple detection scheme [19], and NiCad detection tool with various parameters [7]. Secondly, classified clones are transformed into template method design pattern [20], [21].

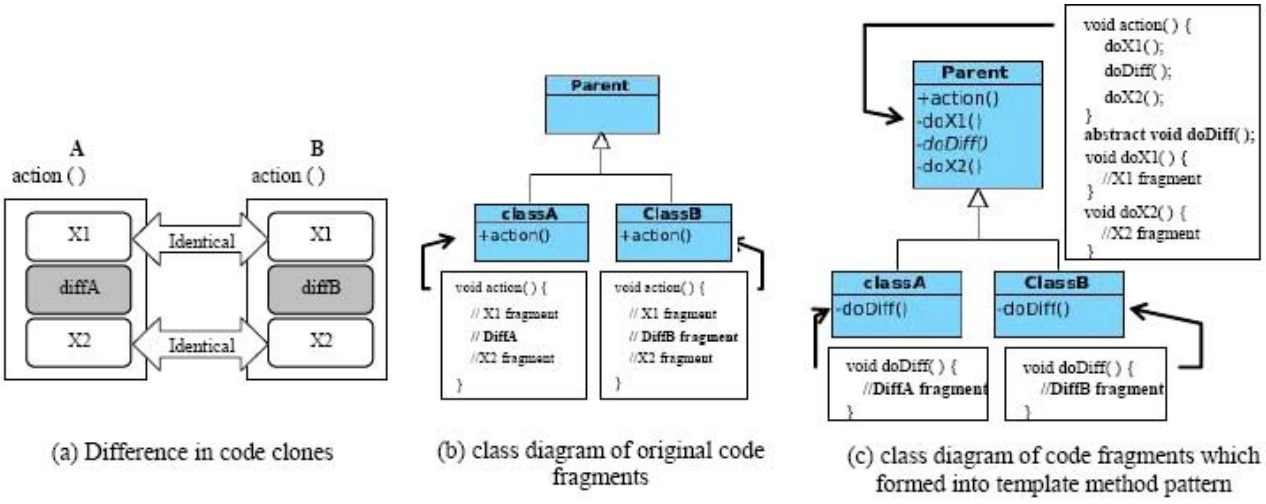
2.1. Code Clone Detection and Classification

Due to space limitation, the approach mainly focuses on constructing core assets from type-3 clones in this paper. NiCad [7], a tool used to identify type-3 clones, and it has three main steps. Firstly, source codes are normalized to remove effect of differences in whitespace, formatting layout and comments. Secondly, preprocessing is a step in order to improve detection result in specific condition by replacing some part of code. Thirdly, code fragments are compared to find code clones. We use renaming and threshold to classify code clone results.

Renaming is preprocessed normalization used to replace all identifiers with the same name. As shown in Table 1, we set the threshold of 0.0 with renaming in *B* detection method in order to detect only type-2 clones.

Threshold is a maximum percentage of difference between code fragments that acceptable in clone result. *A* detection method can detect only type-1 clones while *C* detection method can detect any code clones which have less than 10% differences between their pairs. Clones detected by this method may be type-1, type-2 or type-3 clones. Note that we use threshold of 0.1 in *C* detection method to avoid false negative.

In Table 1., given S_A, S_B and S_C are sets of detected code clone in a small box by detection method *A*, *B* and *C*. Detection method *A* can detect type-1 clone and *B* can detect both type-1 and type-2 clones. We can obtain only type-2 clones from the difference of results from using both methods; *A* and *B*.



(a) Difference in code clones

(b) class diagram of original code fragments

(c) class diagram of code fragments which formed into template method pattern

Fig. 1: Example of type-3 code clone fragment and core asset using template method.

Table 1: Relationship of parameters set and detected results

Detection method	Parameters set		Type of code clones		
	Threshold	*Renaming*	*Type-1*	*Type-2*	*Type-3*
A	0.0		√		
B	0.0	√	√	√	
C	0.1		√	√	√

$S_{type\text{-}1} = S_A$

$S_{type\text{-}2} = S_B - S_A$

$S_{type\text{-}3} = S_C - S_B$

2.2. Forming Core Asset Candidates

Identifying code clones of type-3 is described in the previous step. So we can form them into core asset candidates using template method pattern. In this design pattern, sequences of similar methods are broken into new methods, so differences are put into other new methods as hook methods. A hook method is declared as an abstract method. In this way, it is called by a template method and has to be implemented in each subclass.

To form a hook method, differences have to be identified. Code clone detection tools usually concern only similarity between code fragments and omit differences between them. In order to find differences among them, we compare pairs of type-2 and type-3 clones by using Unix *diff* utility [16], [17]. Those identified statements are moved into concrete methods of individual subclass. However, not every statement can be moved, as it can be either expression statement, declaration statement or control flow statement.

- Expression statement: A statement which contains assignment expressions or method calls.
- Declaration statement: Declaration of variables or objects
- Control flow statement: Any statements related to decision making or looping.

As control flow statement such as *return* or *break*, can exit code block, we focus on only expression and declaration statements in this paper. For each of assignment statements, we assume that variables or methods invocated in its expression are accessible in new created methods. So statements are moved in whole-line.

Fig 1(c). shows code fragments of *action()* method's code fragment in Fig 1(b) that is separated into *doX1()*, *doDiff()*, and *doX2()* methods. A hook method; *doDiff()* is declared in a parent class as an abstract method, and then implemented in *classA* and *classB* with *diffA* and *diffB* code fragments respectively.

3. Experiment

To evaluate an approach described in section 2, an open source project, JabRef [22] is used as subjects of experiment and its details are shown in Table 2(a). A scheme of code clone detections is performed to detect and classify clones, as shown in Table 2(b). Then, type-2 and type-3 clones are used to form core asset candidates.

Table 2: Related information of experiment

(a) Detail of subject software in experiment

Version of JabRef	Characteristics		
	SLOC	*No. of files*	*No. of methods*
1.8	38680	311	2686
2.0	44444	360	3310
2.4	68953	537	4774
2.7	79212	597	5492

(b) Number of clone methods classified by type of clones

Version of JabRef	*2.0*	*2.4*	*2.7*
1.8	Type-1 = 986 Type-2 = 28 Type-3 = 165	Type-1 = 508 Type-2 = 19 Type-3 = 175	Type-1 = 478 Type-2 = 18 Type-3 = 166
2.0		Type-1 = 720 Type-2 = 20 Type-3 = 231	Type-1 = 661 Type-2 = 17 Type-3 = 218
2.4			Type-1 = 1960 Type-2 = 28 Type-3 = 235

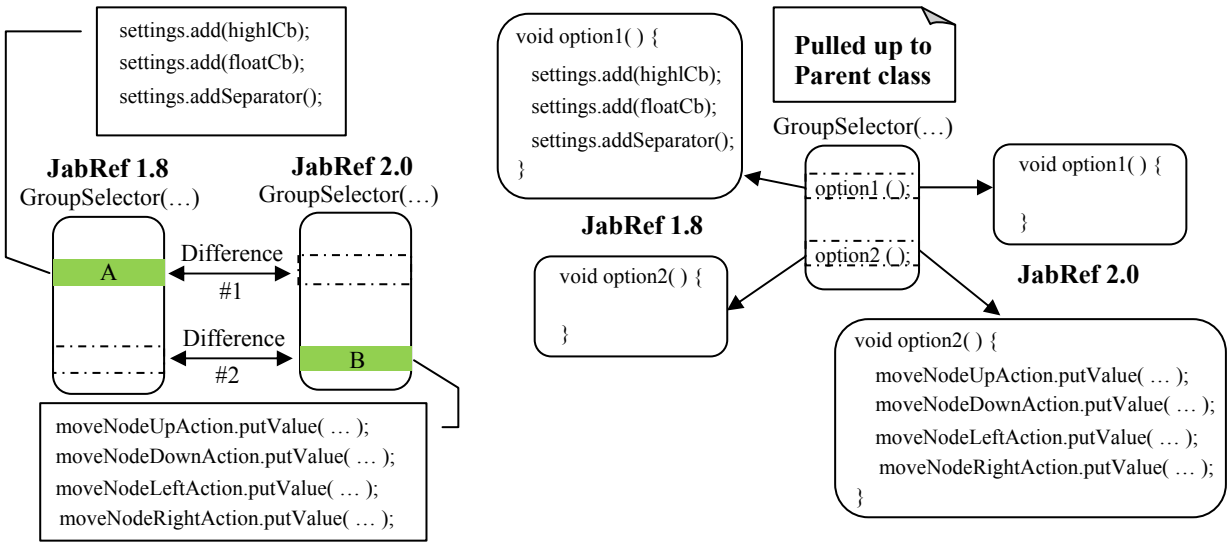


Fig. 2: "GroupSelector" constructor method from JabRef in version 1.8 and 2.0 (GroupSelector.java)

In Fig. 2(a), an example of type-3 clones is presented. Constructor methods of *GroupSelector* class in JabRef 1.8 and 2.0 are demonstrated. There are two differences of this method between versions. Additional statements in version 1.8 and 2.0 are marked by shaded box *A* and *B*, respectively. As shown in Fig. 2(b), a template method is formed in the parent class. *GroupSelector* method calls hook methods for both differences as *option1()* and *option2()*, respectively. *option1()* in version 1.8 is implemented with code fragments in shaded area *A*, while one of version 2.0 is implemented as a NOP method. Likewise, *option2()* is implemented with fragment in shaded area B for version 2.0. In this way, both *option1()* and *option2()* are overridden for each version of software. As experiment results, we found that type-2 and type-3 clones are able to form core asset by template method pattern. However, to optimize hook method, it should be analyzed in more details if differences are spitted into many fragments.

4. Conclusion

In SPL, identifying commonality of existing products is important for core asset extraction, especially source code assets. However, common parts with some differences are not easy to extract. In this paper, we propose an approach to identify core asset candidates by code clone detection. With multiple detections, various parameters are used to classify results with differences depending on types of code clones. Then, template method pattern can be applied to extract differences and form core asset candidates.

We evaluate our approach with several versions of Java open source project. We find that differences in clone results can be formed into separated methods for individual products with template method. However, differences of type of object or expression in some cases are complicated and need more consideration, which is left for future work.

5. References

[1] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection," European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2008, pp. 163-172.

[2] C.K. Roy and J.R. Cordy, "A Survey on Software Clone Detection Research," SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY, vol. 115, 2007, p. 115.

[3] R. Koschke, "Survey of Research on Software Clones," R. Koschke, E. Merlo, and A. Walenstein, eds., Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[4] 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., vol. 74, 2009, pp. 470-495.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," IEEE Transactions on Software Engineering, vol. 33, 2007, pp. 591, 577.

[6] C. Clements and L.M. Northrop, Software product lines: practices and patterns, Addison-Wesley Professional, 2001.

[7] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, 2008, pp. 172–181.

[8] B. S. Baker, "On finding duplication and near-duplication in large software systems," in Reverse Engineering, 1995., Proceedings of 2nd Working Conference on, 1995, pp. 86–95.

[9] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems," Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, no. 2, pp. 61-82, Mar. 2006.

[10] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," 1996, p. 244.

[11] I. D. Baxter and D. Churchett, "Using clone detection to manage a product line," 2002.

[12] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A Case Study in Refactoring a Legacy Component for Reuse in a Product Line," 21st IEEE International Conference on Software Maintenance (ICSM'05), pp. 369-378, 2005.

[13] D. Faust and C. Verhoef, "Software product line migration and deployment," Software: Practice and Experience, vol. 33, no. 10, pp. 933-955, Aug. 2003.

[14] K. Kim, H. Kim, and W. Kim, "Building Software Product Line from the Legacy Systems 'Experience in the Digital Audio and Video Domain'," 11th International Software Product Line Conference (SPLC 2007), pp. 171-180, Sep. 2007.

[15] H. P. Breivold, S. Larsson, and R. Land, "Migrating Industrial Systems towards Software Product Lines: Experiences and Observations through Case Studies," 2008 34th Euromicro Conference Software Engineering and Advanced Applications, pp. 232-239, Sep. 2008.

[16] J. W. Hunt and M. MacIlroy, An algorithm for differential file comparison. Bell Laboratories, 1976.

[17] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," Communications of the ACM, vol. 20, no. 5, pp. 350-353, May 1977.

[18] Y. Ueda, T. Kamiya, and S. Kusumoto, "On detection of gapped code clones using gap locations," Ninth Asia-Pacific Software Engineering Conference, 2002, pp. 327-336, 2002.

[19] Y. Higo, K. Sawa, and S. Kusumoto, "Problematic Code Clones Identification Using Multiple Detection Results," 2009, pp. 365-372.

[20] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," in Reverse Engineering, 2000. Proceedings. Seventh Working Conference on, 2000, pp. 98–107.

[21] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999, p. 464.

[22] SourceForge. Website, 2011. http://sourceforge.net.