Indian Journal of Science and Technology, *Vol 8(S9), 420–425 , May 2015*

ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645
DOI: 10.17485/ijst/2015/v8iS9/68324

Aspect Refactoring Techniques for System Optimization

Seung-Hyung Lee[1] and Sungho Sim[2*]

[1]Department of Computer Engineering, Kyung Hee University, Korea; shlee7@khu.ac.kr
[2]Department of Liberal Education, Semyung University, Korea; shshim@semyung.ac.kr

Abstract

The primary goal is to improve readability and maintainability through reconfiguration of source code without changing function for system refactoring. In refactoring, which is designed for system optimization, the ability to extract duplicated code from scattered code is essential. Such extraction from duplicated code can be used for crosscutting concern, and system optimization can be done through crosscutting concern within Aspect-Oriented Programming. This research paper is for proposal of aspect refactoring for system optimization. To extract duplicated code, which is main target of refactoring, architectural-relation factor is to be extracted through system dependency graph and data dependency relation. By comparing node relation orders from program dependency graph, it is possible to satisfy crosscutting concern for refactoring. After testing system optimization by applying aspect refactoring techniques to Bison 1.25, length, and volume of original module has reduced by 8% and 6% each.

Keywords: Aspect, Refactoring, System Dependency Graph, System Optimization

1. Introduction

Structured/Object-Oriented Programming is most commonly used method for system development due to its multiple advantages. However, it is very costly to design while satisfying requirements and functionality at the same time. Such disadvantage is due to scattered functionality and confusion of code. Established programming is certainly beneficial for modulizing primary concern of each module, but it is inefficient to implement module of crosscutting concern which is associated with several modules[1].

Refactoring is the process which alters code structure to upgrade inner structure and reusability without behavioral change. Existing systems today are built by modularity of primary concern; therefore, there exists duplication of source code dispersed everywhere[2]. At design phase, concerns can be categorized as independent. However, at implementation phase, crosscutting concern will be built into module as well.

In order to do system optimization, refactoring method, which automatically eliminates duplicated code by crosscutting concern, is needed. This essay proposes way to reassemble encapsulated modules through data and control dependency relation. Extracting crosscutting concern starts from analysis of complete system. By analysis, analyzed elements will be given index. With the given index, analysis of data and control dependency will be processed, and then system dependency graph will be created. Next, duplicated crosscutting concern will be extracted with the graph. The concern is capable of optimization through weaving procedure from Aspect-Oriented Programming[3,4].

2. System Dependency Graph

System dependency graph[7,8], which is for describing dependency on one procedure, is basic work unit to show dependency on whole system. For each procedure, it can be combined with program dependency graph

*Author for correspondence

to get system dependency. It is the combination of data dependency and control dependency graph. It has links from both graphs. Therefore, it is capable of knowing information about running time actions without executing the program. It is program expression mode that is resourcefully used for compiler optimization and system slicing[10,11].

2.1 Control Dependency Relation

Control Dependency relation is to define sequential procedure of execution[5]. If the relation is understood, basic information can be comprehended, and it is certainly irreplaceable for optimization. Within the process where dependent nodes are connected to links in graph, if B code is triggered by A code, then B is dependent on A. In Figure 1, line 4 and 6 can be executed by line 3. Therefore, 4 and 5 will be dependent on 3 and 3 will be dependent on 4 and 5.

2.2 Data Dependency Relation

Data dependency relation expresses relation among variables affected by control flow. It can be used to study parameter implementation process, so it is widely used for program or system analysis tool[6]. Within control flow relation, dependency relation from data can combined to express control flow graph. With execution sequence, data flow from each execution is expressed as link. In Table 1, line 7 is determined by line 4 and 6, so line 7 is dependent on line 4 and 6.

3. Aspect Refactoring Process

The primary object of System refactoring is to discover crosscutting area from fully-developed system, and to optimize system by applying it to Aspect-Oriented Programming. Only the related targets must be separated from system components in order to process the task. If text-standardization method is taken for extraction of crosscutting section, sections which have identical structure or different identifier cannot be processed. In the case of (「register int i, j;」 , 「register int i register int j;」), where codes have identical context but different structure, process is unable to be processed. To deal with such issue, System refactoring process, which is based on Abstract Syntax Analysis, is proposed and shown in Figure 1.

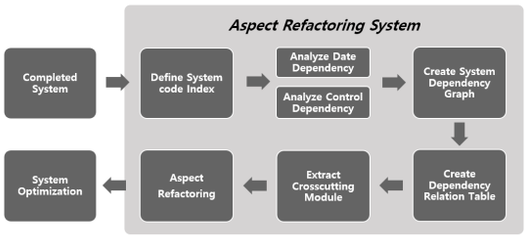

Figure 1. Aspect Refactoring Process for System Optimization.

Table 1. Sample Code for Dependency Relation Analysis

Line Number	Code
1	void control (int a)
2	{
3	**if(a = 2)**
4	**a++;**
5	else
6	**a--;**
7	int b = a;
8	}

To optimize fully-developed system, the first step is to define unique index for data and control dependency of analyzed code. Lines which can be expressed with same pattern will be assigned the index mark. Second step is to analyze data and control dependency relation with given index, and to make system dependency graph. Third, dependency relation table needs to be compiled. For each module, it will be made uniquely based on kind of dependency. But, based on relation graph equation, dependent node will be noted as node (N) and relation in between as edge (E).

$$e_1 = (n_1, n_2),\ e_2 = (n_3, n_4),\ e_1, e_2 \in E \qquad (1)$$

Fourth is to extract duplication by edge comparison. It can be done as follow: Edges of Module A and B are defined as $A_e = \{e_1, e_2, \ldots, e_n\}$, $B_e = \{e_1, e_2, \ldots, e_n\}$, and index of node $N_d = \forall n:\ n \in E_d$ from duplicated edge $\{E_d \mid E_d \in A_e,\ E_d \in B_e\}$ can be combined with index-mapped source to do the work. Fifth step is to select crosscutting concern for refactoring. Extracted crosscutting concern is applicable target for refactoring, and it selects crosscutting concern based on priority. Sixth, refactoring is to be processed with chosen concern applied to Aspect-Oriented Programming. On basis of aspect, functions for crosscutting must be defined in order to execute weaving.

4. Refactoring's Detail for Crosscutting Module Extraction

In order to optimize preexisting system, refactoring needs to be processed through system of bison 1.25, or YACC-compatible Parser Generator. Figure 2 is the sample code of refactoring process with crosscutting concern extract. Sample code is about class Bison_OutputA and Bison_OutputB with duplication from reader.c and lex.c.

4.1 Define System Code Index

Text-based extraction is to standardize blank or certain symbols by replacement. It maintains original form of texts to for comparison while eliminating unnecessary texts. In case of token-based standardization, it is probable that it will contain different textual factors. Therefore, in consideration of refactoring convenience, analysis is taken place with standardization. In addition, two classes, which are refactoring targets, will be defined in their textual patterns, and then unique indices will be given to all defined patterns. Example is shown in Table 2.

By combining index and source code of each text, control and data dependency relations are made as shown in Figure 3. Within the source code for graph compilation, system relational data is to be parsed through CodeSurfer API of Grammatech. Next, for line dependencies of parsed code, they are used to compile graph by forming GDL (Graph Description Language) with indices of textual pattern.

In dependency graph of module Bison_OutputA, and Bison_OutputB, relation of each node is expressed as dependency edge. Each edge is noted as either data or control dependency graph. Each node is index of textual pattern mapped with line of source code. Data/control dependency of compiled dependency graph is set up as follows in Table 3.

This is definition of duplicated edges using module's dependency table. With control dependency edge $A_{ec} = \{(AI, AJ), (AI, AK), (AK, AL), (AK, AM), (AK, AN), (AI, AT)\}$ of Bison_OutputA and control dependency edge $B_{ec} = \{(AI, AJ), (AI, AK), (AK, AL), (AK, AL), (AK, AM), (AK, AN), (AI, AT)\}$ of Bison_OutputB, duplicated edge $\{E_d | E_d \in A_{ec},\ E_d \in B_{ec}\}$ are to be extracted. It applies the same way from data dependency edge as well. Extracted edges are shown in Table 4.

Using dependency relation (Figure 4.) each module, extract node $\{E_d \mid E_d \in A_e, E_d, \in B_e\}$ from duplicated edge. Map source code from extracted node {AH, AI, AJ, AK, AL, AM, LN}. Extract aspect refactoring candidate (crosscutting concern).

Using extracted candidate, structure of aspect-oriented programming for system optimization is defined as Figure 5.

Bison_OuputA	Bison_OuputB

public class Bison_OuputA
{
public void token_actions() {
int i; int j; int k;
actrow = NEW2(ntokens, 2);
k = action_row(0);
System.out.println("nstatic const short yydefact[] = " + new String(k));
save_row(0);
j = 10;
for (i = 1; i < nstates; i++){
System.out.print(",");
if (j > = 10){System.out.print("\n"); j = 1;}else{ j++;}
k = action_row(i);
System.out.println(new String(k));
save_row(i);
}
System.out.print("\n};\n");
}
};
	public class Bison_OuputB
	{
	public void output_stos(){
	int i; int j;
	System.out.println("nstatic const short yystos [] = {0");
	j = 10;
	for (i = 1; i < nstates; i++){
	System.out.print(",");
	if (j > = 10){System.out.print("\n"); j = 1;}else{ j++;}
	System.out.println(new String(accessing_symbol[i]));
	}
	System.out.print("\n};\n");
	}
	}

Figure 2. Convert the code in Java, GNU bison-1.25.

Table 2. Index defined for the source code

Index	Source Code
AA	int i;
AB	int j;
AC	int k;
AD	actrow = NEW2(ntokens, 2);
AE	k = action_row(0);
AF	System.out.println("nstatic const short yydefact[] ...
AG	save_row(0);
AH	j = 10;
AI	for (i = 1; i < nstates; i++)
AJ	System.out.print(",");
AK	if (j > = 10)
AL	System.out.print("\n");
AM	j = 1;
AN	j++;
AO	k = action_row(i);
AP	System.out.println(new String(k));
AQ	save_row(i);
AR	System.out.print("\n};\n");
AS	System.out.println("nstatic const short yystos[] …
AT	System.out.println(new String(accessing_symbol[i]));

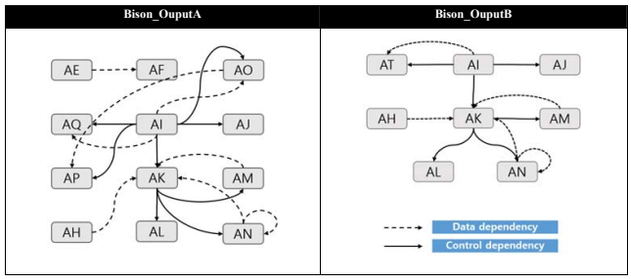

Figure 3. Dependency graph of the class Bison_OuputA/ Bison_OuputB.

Table 3. Dependency of class Bison_OutputA, Bison_OutputB

Bison_OuputA		Bison_OuputB	
Control dependen`cy	Data dependency	Control dependency	Data dependency
(AI,AJ)c	(AE,AF)d	(AI,AJ)c	(AH,AK)d
(AI,AK)c	(AH,AK)d	(AI,AK)c	(AM,AK)d
(AK,AL)c	(AM,AK)d	(AK,AL)c	(AN,AK)d
(AK,AM)c	(AN,AK)d	(AK,AM)c	(AI,AT)d
(AK,AN)c	(AI,AO)d	(AK,AN)c	(AN,AN)d
(AI,AP)c	(AO,AP)d	(AI,AT)c	
	(AI,AQ)d		
	(AN,AN)d		

Table 4. Extracted dependency table

Control dependency	Data dependency
(AI,AJ)c	(AH,AK)d
(AI,AK)c	(AM,AK)d
(AK,AL)c	(AN,AK)d
(AK,AM)c	(AN,AN)d
(AK,AN)c	

```
j = 10;
for (i = 1; i < nstates; i++)
System.out.print(",");
if (j > = 10)
System.out.print("\n");
j = 1;
j++;
```

Figure 4. Extracted refactoring candidate.

```
Aspect_Bison_output(int p){
System.out.print(",");
if (j > = 10){System.out.print("\n"); j = 1;}
else{ j++;}
return j;
}
```

Figure 5. **AOP structure for system optimization.**

```
public aspect New_Bison_output {
…
pointcut Aspect_Bison_output(int j)
: call(* Aspect_Bison_output(int)) andand args(int);
around(int j) : Aspect_Bison_output(j) {
…
}
}
```

Figure 6. Crosscutting functionality specification for refactoring.

If aspect is element in method, interface {j = Aspedt_Bison_output(j)} needs to be defined and node corresponding to aspect is to be switched into interface. In addition, Aspect_Bison_output() within source code needs to be defined as well for crosscutting.

Table 5. The measured complexity of the system module

Module	org_source			CCFinder			SDG(Our)		
	length	volume	level	length	volume	level	length	volume	level
LR0.c	1885	13320	0.005882	1805	12754	0.006168	1805	12754	0.006168
conflicts.c	2413	17153	0.004401	2303	16322	0.004525	1957	13828	0.005362
lalr.c	2503	18372	0.006036	2460	18012	0.00603	2427	17770	0.006125
output.c	4043	31870	0.004906	3919	30771	0.005056	3811	29923	0.005202
reader.c	6002	49331	0.004019	5852	48013	0.004191	5629	51345	0.007419

Methods applicable for each class needs to be defined and configured to pointcut. Execution within JoinPoint will be turned into optimized system code using around (), which is user-defined advice, before and after method call. Such process is written as crosscutting definition as Figure 6, and system optimization is completed by weaving from aspect-oriented programming.

5. Evaluation

Proposed process efficiently perform system optimization was performed for quantitative evaluation. Using original source code, crosscutting concern module extracted applying ccFinder and proposed SDG. And System optimization was performed. In order to evaluate the performance results refactoring, System complexity was measured. Based on operator and operand code used in the system to calculate the length, volume, level of the system. If length/volume is lower and level is higher, means that complexity is low.

After crosscutting concern extraction through the CCFinder and proposed process, the system optimization was performed. After optimizing the system, the measurement of the system complexity is shown in Table 5.

- CCFinder Official Site12.

As a result of applying Bison 1.25 into system, the efficiency of optimization is certainly improved. Compare to original source, the average length, and volume of 5 modules have decreased by 92.4% (improve 7.6%) and 94.2% (improve 5.8%) each. Also, the average level is increased by 124.8% (improve 24.8%). Compare to ccFinder, the average length, and volume of 5 modules have decreased by 95.4% (improve 4.6%) and 97.5% (improve 2.5%) each. Also, the average level is increased by 120.0% (improve 20.0%).

6. Conclusion

In Object-oriented programming, reusability and maintenance become the issue whenever there is change in features or application of new data. Also, it is impossible to reassemble certain encapsulated features within object after crosscutting. Therefore, it is very troublesome to process commonly duplicated codes from several objects. Unfortunately, there is no detailed and objective approach to solve such issues with object-oriented refactoring features.

This research proposes a method to extract refactoring features in both Structured Programming (SP) and Object-Oriented Programming (OOP). The control and data dependency is used to take reusable module. First, it defines index of source code. Second, based on defined index, program dependency graph is established. With established graph, dependency relation table is set up. Duplicated modules are extracted through comparison of order relation from dependency relation table, and system optimization is enabled by weaving of AOP. With the method proposed in this essay, reassembling of encapsulated objects in OOP, and SP is capacitated, and therefore performance is able to be upgraded through system optimization.

7. References

1. Marin M, Moonen L, van Deursen A. An approach to aspect refactoring based on crosscutting concern types. Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software; 2005. p. 1–5.
2. The Aspect J Team. The Aspect J Programming Guide. Palo Alto Research Center. Version 1.2. 2003.
3. Apel S, Kastner C, Batory D. Program refactoring using functional aspects. Proceedings of the 7th International Conference on Generative Programming and Component Engineering. ACM Press; 2008. p. 161–70.

4. Hanenberg S, Oberschulte C, Unland R. Refactoring of aspect-oriented software. Proceedings of the 4th International Conference on Object-Oriented and Internet-based Technologies Concepts, and Applications for a Networked World; 2003. p. 19–35.
5. Matsumoto T, Saito H, Fujita M. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. Proceedings of the 7th International Symposium on Quality Electronic Design; 2006. p. 370–5.
6. Orso A, Sinha S, Harrold MJ. Effects of pointers on data dependences. Proceedings of the 9th International Workshop on Program Comprehension; 2001. p. 39–49.
7. Balmas F. Displaying dependence graphs: a hierarchical approach. Proceedings of the 8th Working Conference on Reverse Engineering; 2001. p. 261–70.
8. Liu C, Chen C, Han J, Yu P. Detection of software plagiarism by program dependence graph analysis. Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2006. p. 872–81.
9. Harman M, Binkley D, Gallagher K, Gold N, Krinke J. Dependence clusters in source code. ACM Transactions on Programming Languages and Systems (TOPLAS). 2009; 32(1):1–33.
10. Krinke J. Effects of context on program slicing. J Syst Software. 2006; 79(9):1249–60.
11. Binkley D, Danicic S, Gyimothy T, Harman M, Kiss A, Korel B. A formalisation of the relationship between forms of program slicing. Special Issue on Source Code Analysis and Manipulation (SCAM 2005); 2006. p. 228–52.
12. Archive of CCFinder Official Site. Available from: http://www.ccfinder.net/