프로그램 의존성 그래프를 이용한 어스팩트 리팩토링에 관한 연구

조병현, 이승형, 송영재
경희대학교 컴퓨터공학과
e-mail : keane@khu.ac.kr

A Study on Aspect Refactoring using Program Dependency Graph

Byoung-Hyoun Cho, Seung-Hyung Lee, Young-Jae Song
Dept. of Computer Engineering, KyungHee University

요 약

리팩토링은 시스템의 기능 변경 없이 코드 구조를 재조정하여 가독성을 높이고 유지보수성을 향상하기 위함이다. 기존의 어스팩트 리팩토링은 프로그램의 특정 부분을 어스팩트로 정의하여 리팩토링하거나 구현된 어스팩트 명세를 재구성하는 방식으로, 객체지향 프로그램에 적용하는데 어려움이 있다. 본 논문은 객체지향 리팩토링에 어스팩트 개념을 적용하기 위한 구체화된 접근방법을 제시하는 것이 목적이며 이를 위해 프로그램 의존성 그래프를 이용한다. 리팩토링의 주요 어스팩트인 중복 코드는 프로그램 의존 그래프에서 노드 사이의 순서관계를 비교하여, 리팩토링을 위한 어스팩트 후보로 변환하며 이를 근거로 재조합 함으로써 캡슐화된 객체 내부의 리팩토링 요소를 편리하게 처리할 수 있다.

1. 서론

리팩토링은 프로그램의 기능, 행위의 변경 없이 내부 구조와 재사용성을 향상하기 위하여 코드의 구조를 변경하는 프로세스이다. 기존의 객체지향 방법은 핵심 관심사의 모듈화에는 효율적이나 횡단 관심사를 모듈화할 수 없다. 설계 시점에는 횡단 관심사를 독립적인 관심사로 분리할 수 있으나 구현 시점에는 한 개의 모듈에 횡단 관심사가 혼합하여 들어가게 된다.

구현된 프로그램을 리팩토링 하는 과정에서 횡단 관심은, 이미 캡슐화된 객체들의 일부를 재조합하는 방법을 사용해야 하는데 이는 객체지향의 원칙에 어긋나는 방법으로, 고려 대상에서 제외될 수 밖에 없었다. 그러나 객체지향 리팩토링에 어스팩트 개념을 도입할 경우 각 모듈 내에 산재해 있는 리팩토링 대상을 횡단 관심으로 정의하여 쉽게 분리 제거할 수 있으며 이는 어스팩트에서 메소드 호출을 분리함으로써 크로스커팅 기능을 캡슐화 하여 이루어진다.

기존의 어스팩트 리팩토링[1]은 어스팩트 지향 프로그래밍[2] 방법으로 구현된 프로그램의 각 어스팩트 명세 내의 리팩토링 요소를 고려하여 재구성하거나 객체지향 프로그램의 일부를 어스팩트 명세로 정의하여 특정 요소를 분리하는 것으로, 객체지향으로 설계된 프로그램에 대한 구체적인 어스팩트 리팩토링 방법으로는 부족하다.

어스팩트 리팩토링에서는 기존 프로그램의 리팩토링 요소를 어스팩트로 변환하는 기술이 필수적이며 이러한 변환을 지원하는 기술에 대한 연구가 필요하다. 본 논문은 리팩토링 시 캡슐화된 객체를 재조합하기 위해 어스팩트 개념을 도입하며 리팩토링 요소를 프로그램 의존성 그래프를 이용하여 어스팩트로 변환하고 이를 적용하는 구체화된 접근 방법을 단계별로 정의하였다.

2. 관련연구

2.1 어스팩트 리팩토링

어스팩트는 기능의 분산과 코드의 혼란을 해결하기 위한 크로스커팅 개념의 구현방법이다. 일반적인 어스팩트 리팩토링은 어스팩트 지향 프로그래밍 방법으로 구현된 프로그램의 어스팩트 명세 중 리팩토링을 위한 요소를 찾고 이를 처리하기 위한 명세를 새롭게 작성하여 어스팩트를 최적화[3]하는 것을 의미한다. 또한 객체지향 프로그램에서 캡슐화된 객체 내의 메소드를 치환하거나 분리[4]하여 리팩토링을 구현할 경우 객체에 접근하기 위한 방법으로 어스팩트 지향 프로그램밍을 사용하고 이를 어스팩트 리팩토링으로 정의한다.

2.2 제어 종속성 그래프

제어 종속성 그래프는 프로그램 실행 시의 순차적 진행 과정을 정의하기 위한 것으로 프로그램 내의 종속 관계를 조사하면 코드 스케줄링을 위한 기본 정보를 파악할 수 있으며 주로 컴파일러에 사용되고 프로

그램 최적화에 필수적이다. 각 노드가 실행되는 과정에서 종속성이 있는 노드에 링크를 연결시킨 그래프로, B 코드의 실행 여부는 A 코드에 따라 결정된다고 할 때, B는 A에 종속성이 있다고 한다.

2.3 데이터 종속성 그래프

데이터 종속성 그래프[5]는 제어 흐름에 영향을 받는 변수들간의 관계를 나타내며 이를 이용하여 메소드 호출에 따른 파라메터 이행 과정에 대한 정보를 파악할 수 있어 시스템 및 프로그램 분석 툴에 널리 사용되고 있다. 제어 흐름 그래프에서 데이터 부분의 종속성 연결이 더해진 형태의 그래프이다. 제어 흐름 그래프에서 만들어진 실행순서에 각 실행문에서 사용된 데이터의 흐름을 링크로 표현한 것이다.

2.4 프로그램 의존성 그래프

프로그램 의존성 그래프[6,7]는 하나의 프로시저 에 대해 의존성을 표시하는 것으로서 전체 프로그램의 의존성을 나타내기 위한 기본 작업 단위라 할 수 있다. 프로시저 각각에 대하여 프로그램 의존성 그래프를 생성하고 이것을 적절히 조합하면 시스템 의존성 그래프를 얻을 수 있다. 데이터 종속성 그래프와 제어 종속성 그래프가 섞인 형태의 그래프이며 데이터 종속성 그래프와 제어 종속성 그래프의 링크를 모두 가지고 있으므로 두 가지 그래프의 특징인 제어 종속성 링크와 데이터 종속성 링크를 이용하여 프로그램을 실제 실행하지 않고 프로그램의 실행 시간 동작에 대한 정보를 유추할 수 있어, 컴파일러 최적화 분야에 유용하게 사용되고 있으며 프로그램 자르기 (Program slicing) 분야 등 활용가치를 높여가고 있는 프로그램 표현 방식이다.

3. 어스팩트 리팩토링 방법

캡슐화된 객체의 내부 요소 중 리팩토링 대상만을 분리하여 재조합하기 위해 기존의 리팩토링에 어스팩트 개념을 도입하며, 의존성 그래프를 이용하여 리팩토링 대상인 중복코드를 어스팩트로 변환한다. 이에 대한 과정을 다음과 같이 정의하였다.

1 단계: 소스 코드로부터 의존성 그래프 작성을 위한 인덱스 정의 - 프로그램 의존성 그래프로 표현될 소스 코드의 각 행에 대한 고유 기호를 정의한다. 의존성 그래프 상에 동일한 패턴으로 표시될 수 있는 행은 같은 기호를 부여한다.

2 단계: 프로그램 의존성 그래프 작성 - 객체 단위로 소스 코드에 대한 데이터 종속성, 제어 종속성 관계를 방향성 그래프로 표현한다. 각 행에 대한 표시는 부여한 기호로 하며 종속의 종류를 구분할 수 있도록 구성한다.

3 단계: 의존 관계 테이블 작성 - 각 객체에 대해 종속의 종류별로 작성하되, 방향성 그래프 표현 식을 근거로 종속성을 지닌 행은 노드(N)로, 각 노드 간의 관계는 에지(E)로 정의한다.

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

4 단계: 순서 관계(에지) 비교를 통한 중복 코드 추출 - 객체 A, B 의 에지는 $A_e=\{e_1,e_2 \cdots e_n\}$, $B_e=\{e_1,e_2 \cdots e_n\}$로 정의되며, 중복된 에지 $\{E_d | E_d \in A_e, E_d \in B_e\}$에 해당되는 각 노드 $N_d=\forall n : n \in E_d$ 의 인덱스와 맵핑된 소스를 조합하여 중복코드를 추출한다.

5 단계: 리팩토링을 위한 어스팩트 선정 - 추출된 중복코드는 리팩토링에 적용 가능한 어스팩트 후보이며 이 중 리팩토링 우선 순위에 따라 어스팩트를 선정한다.

6 단계: 어스팩트를 적용한 리팩토링 - 선정된 어스팩트가 실제 크로스커팅에 적용될 수 있도록 재구성하고 크로스 커팅을 위한 포인트 컷, 어드바이스 등을 명세한 후 위빙을 실시한다.

4. 의존성 그래프를 이용한 어스팩트 리팩토링

```
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");
      }
}
```

(그림 1) GNU bison-1.25 의 코드를 Java 로 변환

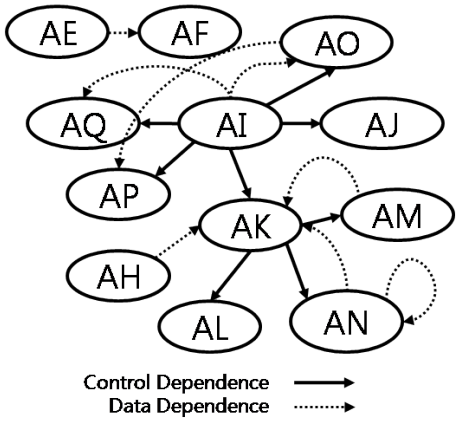
(그림 1)은 두 개의 클래스로 구성되며 각 클래스의 중복 코드를 리팩토링한다.

(표 1) 소스 코드에 대한 인덱스 정의

인덱스	소스 코드
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]));

(표 1)은 프로그램 의존성 그래프의 노드를 소스 코드와 맵핑하기 위한 인덱스이다.



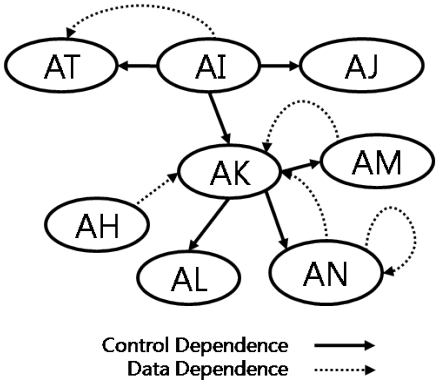
(그림 2) 클래스 Bison_OuputA 의 의존성 그래프

클래스 Bison_OutputA 의 제어 종속성과 데이터 종속성을 (그림 2)과 같이 도식한다.

(표 2) 클래스 Bison_OutputA 의 의존 관계

제어 종속성	데이터 종속성
(AI,AJ)c	(AE,AF)d
(AI,AK)c	(AH,AK)d
(AK,AL)c	(AM,AK)d
(AK,AM)c	(AN,AK)d
(AK,AN)c	(AI,AO)d
(AI,AP)c	(AO,AP)d
(AI,AQ)c	(AI,AQ)d
(AI,AO)c	(AN,AN)d

클래스 Bison_OutputA 의 의존성 그래프에서 각 노드의 관계를 (표 2) 와 같이 제어 종속성, 데이터 종속성 에지로 표현한다.



(그림 3) 클래스 Bison_OuputB 의 의존성 그래프

클래스 Bison_OutputB 의 제어 종속성과 데이터 종속성을 (그림 3)과 같이 도식한다.

(표 3) 클래스 Bison_OutputB 의 의존 관계

제어 종속성	데이터 종속성
(AI,AJ)c	(AH,AK)d
(AI,AK)c	(AM,AK)d
(AK,AL)c	(AN,AK)d
(AK,AM)c	(AI,AT)d
(AK,AN)c	(AN,AN)d
(AI,AT)c	

클래스 Bison_OutputB 의 의존성 그래프에서 각 노드의 관계를 (표 3) 과 같이 제어 종속성, 데이터 종속성 에지로 표현한다.

각 클래스의 의존 관계표를 이용하여 중복된 에지를 정의한다. Bison_OutputA 의 제어 종속성 에지 A_{ec}={(AI,AJ), (AI,AK), (AK,AL), (AK,AM), (AK,AN), (AI,AP),(AI,AQ),(AI,AO)} 와 Bison_OutputB 의 제어 종속성 에지 B_{ec}={(AI,AJ), (AI,AK), (AK,AL), (AK,AM), (AK,AN), (AI,AT)} 에서 중복에지 $\{E_d | E_d \in A_{ec}, E_d \in B_{ec}\}$를 추출한다. 데이터 종속성 에지에서도 동일한 방법으로 중복에지를 추출한다. 추출된 에지는 (표 4)와 같다.

(표 4) 추출된 의존 관계

제어 종속성	데이터 종속성
(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	

중복된 에지에서 노드 $\forall$ n:n$\in E_d$ 를 추출한 후 {AH,AI,AJ,AK,AL,AM,AN} 인덱스의 소스 코드와 맵핑하여 리팩토링을 위한 어스팩트 후보를 (그림 4)와 같이 정의한다.

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

(그림 4) 추출된 어스팩트 후보

어스팩트 후보를 이용하여 리팩토링을 위한 어스팩트의 구조를 (그림 5)와 같이 정의한다.

```
Aspect_Bison_ouput(int j){
      System.out.print(",");
      if (j >= 10){System.out.print("Wn"); j = 1;}
      else{ j++;}
      return j;
}
```

(그림 5) 어스팩트 구조

어스팩트가 메소드의 엘리먼트일 경우 어스팩트의 인터페이스{j=Aspect_Bison_ouput(j)}를 정의하고 어스팩트에 해당되는 노드를 인터페이스로 치환한다.

```
public aspect New_Bison_ouput {
   …
   pointcut Aspect_Bison_ouput(int j)
      : call(* Aspect_Bison_ouput(int)) && args(int);
   around(int j) : Aspect_Bison_ouput(j) {
   …
   }
}
```

(그림 6) 크로스커팅 기능 명세

각 클래스에 적용될 메소드를 선행 정의한 후 이를 포인트컷으로 설정하고, 조인포인트에서의 실행은 메소드 호출 전후에 사용자 지정 행위를 수행하는 어드바이스인 around()를 사용하여 어스팩트 본문으로 일괄 변환한다. 이를 크로스커팅 기능 명세로 (그림 6)과 같이 작성한 후 위빙을 실시하여 리팩토링을 완성한다.

5. 결론

객체지향 프로그래밍에서는 캡슐화된 객체 내의 특정 요소만을 크로스커팅 후 재조합 하는 것이 불가능하였다. 따라서 각 객체 내의 공통적인 리팩토링 요소인 중복코드 처리에 있어서는 많은 난점을 지니고 있었다. 이를 해결하기 위한 방안으로 어스팩트 개념을 도입할 수 있으나 어스팩트 자체가 객체지향 리팩토링 요소에 대한 구체적이고 객관화된 접근방법을 가지고 있지는 못하다.

본 논문은 객체지향 프로그램의 리팩토링 요소에 대해 어스팩트를 적용하는 과정을 개선하였다. 객체지향 리팩토링에 주요한 관심 요소인 메소드 내의 파라메터 사이에 존재하는 중복 코드에 대한 처리는 AspectJ, AspectC 등의 어스팩트 지원 프레임워크에서 조자 직접적인 명세가 불가능하고 어스팩트 지원 프레임워크의 핵심인 횡단관심에 대한 정의가 모호하여 사용목적에 따라 이를 특정 지을 수 있는 접근 방법이 필요하였다. 리팩토링 시 어스팩트의 모호성을 구체화 할 수 있는 방안으로 프로그램 의존성 그래프를 이용한 6 단계의 정형화된 모델을 제시하였으며, 이는 어스팩트 지원 프레임워크의 종류에 관계 없이 일관된 결과를 얻을 수 있다.

어스팩트 개념으로 리팩토링에 접근하여 캡슐화된 객체의 재조합 문제를 해결할 수 있었으며, 어스팩트 리팩토링의 난제인 어스팩트 추출 또한 프로그램 의존성 그래프와 방향성 그래프 속성을 응용하여 처리할 수 있었다.

참고문헌

[1] M. Marin, L. Moonen, and A. van Deursen.,” An approach to aspect refactoring based on crosscutting concern types.”, In Proceedings of the 2005 workshop on Modeling and analysis of concerns in software, pages 1- 5, 2005.

[2] The AspectJ Team, “The AspectJ Programming Guide.”, Palo Alto Research Center.Version 1.2, 2003.

[3] S. Apel, C. Kastner, and D. Batory.,” Program refactoring using functional aspects.”,In Proceedings of the 7th International Conference on Generative Programming and Component Engineering. ACM Press, Pages 161-170, 2008.

[4] S. Hanenberg, C. Oberschulte, and R. Unland.,"Refactoring of Aspect-Oriented Software.", In Proceedings of the 4th International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World,pages 19-35,2003.

[5] A. Orso, S. Sinha, and M.J. Harrold, “Effects of pointers on data dependences”, In Proceedings of the 9th International Workshop on Program Comprehension, pages 39-49,2001.

[6] F. Balmas.,” Displaying dependence graphs: a hierarchical approach.”, In Proceedings of the 8th Working Conference on Reverse Engineering,pages 261- 270, 2001.

[7] C. Liu, C. Chen, J. Han and P. Yu.,"detection of software plagiarism by program dependence graph analysis", In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining,pages 872 - 881,2006.