메트릭을 이용한 리팩토링의 적용

이숙희[O] 채흥석 권용래
한국과학기술원 전자전산학과
{shlee[O], hschae, kwon}@salmosa.kaist.ac.kr

The Application of Refactoring Based on Metrics

Suk-Hee Lee[0] Heung-Seok Chae Yong-Rae Kwon
Dept. of Electronic Engineering and Computer Science, KAIST

요 약

리팩토링은 외부로 나타나는 행위의 변화없이 내부구조를 바꾸어 소프트웨어의 품질을 높인다. 이러한 리팩토링을 언제, 어디에 적용하는가에 대한 결정은 유지보수자(maintainer)의 주관적인 직관에 따르기 때문에 이를 체계적으로 결정하는 것은 어렵다. 본 논문에서는 메트릭을 이용하여 객체지향 시스템에서 언제, 어떤 리팩토링을 적용할 것인가를 체계적으로 결정하는 방안에 대해서 논의한다. 리팩토링의 최종 결정권은 유지보수자가 가지고 있으므로 리팩토링에 대한 결정을 보조하는 방안에 초점을 맞추었다. 그리고 본 논문에서 제안하는 방법을 6가지 대표적인 리팩토링에 적용시켜보았다.

1. 서론

리팩토링은 외부 동작을 바꾸지 않으면서 내부구조를 개선하는 방안으로 소프트웨어를 보다 이해하기 쉽고, 유지 보수 기간 동안의 변화에 따라 수정하기 쉽도록 만든다. 리팩토링 과정은 다음과 같이 3단계로 나눌 수 있다.[1]

(1) 언제 리팩토링을 해야하는지 결정한다
(2) 어디에, 어떤 리팩토링을 적용할지 결정한다
(3) 위에서 결정한 리팩토링을 실행한다.

현재, 이 3단계를 모두 자동화하는 연구는 없다. 언제, 어떤 리팩토링을 적용하는가를 결정하는 문제는 유지보수자의 주관적인 직관에 따라서 결정되기 때문에 이를 체계적으로 결정하기가 어렵다. 따라서 현 시점에서는 마지막 단계에만 자동화도구가 존재한다. 본 논문에서는 언제, 어떤 리팩토링을 적용하는가를 결정하는 문제를 체계적으로 해결하기 위한 방법으로 메트릭을 사용한다. 즉, 리팩토링을 하려는 시스템에 대해 메트릭 값을 측정하고 그 측정값을 분석하여 적절한 리팩토링을 제안한다. 이러한 해결방법은 리팩토링의 최고 결정자가 유지보수자라는 사실 때문에 유지보수자의 직관을 보조하는데 초점을 맞추고 있지만 보다 효율적이고 체계적으로 문제를 해결할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 다룰 몇몇 대표적인 리팩토링에 대해서 설명한다. 3장에서는 리팩토링이 필요한 구조를 찾는데 사용할 메트릭을 제시한다. 4장에서는 3장에서 설명한 메트릭을 적용하여 리팩토링이 필요한 구조를 찾는 방법에 대해서 기술한다. 5장에서는 간단한 예제를 보여준다. 6장에서는 관련 연구를 간단하게 소개한다. 마지막으로 7장에서는 결론 및 향후 연구에 대해서 기술한다.

2. 리팩토링

본 논문에서는 6개의 대표적인 리팩토링들만 다룬다. 이 리팩토링들은 모두 클래스의 멤버들과 관련되어 있다. 따라서 이 리팩토링들을 적용하여 내부구조를 바꾸는 것은 클래스가 너무 많은 기능을 가지고 방대해지는 것을 막고 같은 목적을 가진 멤버들을 한 클래스에 두어서 클래스의 캡슐화를 높여 클래스 디자인의 품질을 개선한다.

- Move Method
  메소드가 자신이 정의된 클래스의 기능보다 다른 클래스의 기능을 더 많이 사용하거나 다른 클래스에서 더 많이 사용되어진다면 이 메소드를 다른 클래스로 옮긴다. 클래스가 너무 많은 기능을 가지거나 다른 클래스와 결합이 클 때에도 메소드를 옮긴다.
- Move Attribute
  속성이 자신이 정의된 클래스보다 다른 클래스에 의해서 더 많이 사용되고 있다면 속성을 다른 클래스에 옮긴다.
- Extract Class
  두 개 이상의 클래스가 해야 할 일을 하나의 클래스가 하고 있는 경우 새로운 클래스를 만들어서 관련있는 속성과 메소드를 예전 클래스에서 새로운 클래스로 옮긴다. 이런 클래스들은 많은 메소드와 데이터를 가지는 경우가 많다.
- Inline Class
  클래스가 하는 일이 많지 않은 경우에 그 클래스에 있는 모든 변수와 메소드를 다른 클래스로 옮기고 그 클래스를 제거한다. 주로 move method/attribute 리팩토링의 결과로 클래스에 거의 멤버가 없을 때 inline class 리팩토링을 한다.
- Push Down Method
  수퍼클래스에 있는 메소드가 서브클래스중 일부에만 관련되어 있다면 그 메소드를 관련된 서브클래스로 옮긴다.
- Push Down Attribute
  속성이 일부 서브클래스에 의해서만 사용되고 있다면 그 속성을 관련된 서브클래스로 옮긴다.

3. 메트릭

본 논문에서 다루고 있는 리팩토링은 " 밀접한 관계에 있는 것들은 같은 클래스 안에 있어야 한다" 라는 원리에 어긋나는 부분에 적용되는 것들이다. Bieman과 Kang은 응집도(cohesion) 란 한 클래스 내의 멤버들 사이의 서로 연관된 정도라고 정의하였다[2]. 따라서 응집도 메트릭을 이용하여 위에서 소개한 리팩토링이 필요한 구조를 찾아낼 수 있다. 현재 응집도 메트릭에 관련된 많은 논문들이 발표되어 있다(e.g. [2],[3],[4]). 그 중 [4]에서 처음으로 소개된 응집도 메트릭 CBMC는 객체지향 클래스의 특징을 고려하여 클래스 멤버들간의 연결정도에 따라 응집도를 측정한다.

임의의 클래스 c에 대하여 참조그래프 (reference graph)를 Gc라고 하자. 이 때 참조그래프 Gc의 응집도 값은

CBMC(Cohesion based member connectivity)
= Fc(Gc) X Fs(Gc)

여기서

$$Fc(\text{Connectivity factor}) = \frac{|Mg(Gc)|}{|Nm(Gc)|},$$

Fs(Structure factor) = 1/n ∑CBMC(G') 이다.

(Mg(Gc), Nm(Gc)는 각각 glue methods의 집합, normal methods의 집합, G' 는 Gc의 응집도 컴포넌트를 나타낸다.)따라서 클래스 c의 응집도는 다음과 같이 정의된다.

Cohesion (c) = CBMC(Gc)

CBMC는 클래스의 특성을 고려할 뿐 아니라 클래스가 단 하나의 기능을 가지도록 설계되었는지 알 수 있게 해 주고 만약 하나 이상의 기능을 가지다면 분리하는 방법까지 제시한다.

CBMC 외에 리팩토링이 필요한 구조를 찾는데 필요한 몇가지 메트릭들을 정의하여 표 1에 정리하였다

<표 1 : 리팩토링에 이용할 메트릭>

이름	정의
NMC	클래스의 공용 메소드(public methods)의 개수. *NMC=|M(c)|*
NAC	클래스의 속성들(상속된 속성 포함)의 개수 *NAC=|V(c)|*
NUA	속성을 참조하는 메소드의 개수 *NUA = |R(a)|*
RNM	메소드의 바디(body)에서 사용한 모든 기능에 대한 다른 클래스의 기능의 비율 *RNM=|La(p)+Lm(p)|/|Ua(p)+Um(p)|*
NUM	메소드를 호출하는 자신 클래스의 메소드의 수 *NUM = |I(m)|*
RNA	속성을 사용하는 메소드에 대한 다른 클래스의 메소드의 비율 *RNA = |Ur(a) ∩ R(a)| / |Ur(a)|*
NIM	메소드를 호출하는 다른 클래스에서 정의된 메소드의 개수 *NIM = |UI(m)|*

C - 임의의 클래스
V(C) - C의 모든 속성들의 집합. 상속된 속성포함
M(C) - C의 모든 메소드들의 집합. 공용으로 상속된 메소드 포함
m->a - 메소드 m이 속성 a를 참조한다
m=>p - 메소드 m이 메소드p를 호출한다
R(a) ={m | m∈M(C) and a∈V(C) and m->a}
La(p)=Ua(p) - {a | p∈M(C) and p->a and a∈V(C)}
Ua(p)={a | p∈M(C) and p->a}
Lm(p)=Um(p)-{m | p,m∈M(C) and p=>m}
Um(p)={m | p∈M(C) and p=>m}
I(m)={p | m,p∈M(C) and p=>m}
Ur(a)={m | a∈V(C) and m->a}
UI(m)={p | m∈V(C) and p=>m} ∩ I(m)

4. 메트릭을 이용한 리팩토링

본 논문에서 설명한 6가지 리팩토링은 모두 사용관계(use relations)에 근거를 둔다: 서로 밀접하게 사용되어지는 것들은 같은 클래스에 있어야 한다. 이런 관점에서 봤을 때 리팩토링이 필요한 구조를 찾는 것은 응집도 값이 낮은 클래스를 찾는 문제로 줄여질 수 있다. 리팩토링이 필요한 구조를 찾는 순서는 다음과 같다.

(1) 리팩토링을 할 시스템의 각각의 클래스에 대해 응집도 메트릭 값을 측정한다
(2) 모든 클래스의 응집도 값을 분석하여 낮은 값을 가지는 클래스를 찾는다
(3) (2)에서 찾은 클래스에 대해서 나머지 메트릭 값을 측정한다
(4) 측정값을 분석하여 적절한 리팩토링을 제안한다

본 논문에서 다루는 리팩토링 자체가 명확한 속성을 가지지않고 메트릭 측정값만으로 디자인의 옳고 그름을 단정짓기에는 이에 대한 명확한 정의가 없기 때문에 메트릭값을 분석하는 정확한 기준값을 제시하지는 않는다. 적절한 리팩토링을 찾기위해서 메트릭 측정값 분석을 아래와 같이 분류하여 표 2에 정리하였다.

- Extract Class
  NMC와 NAC는 클래스의 크기를 측정하는데 사용한다. 따라서 CBMC값이 작고, NMC, NAC 값이 크다는 것은 클래스가 하나 이상의 기능을 가진다는 것을 의미한다.
- Inline Class
  CBMC, NMC, NAC 값이 작다는 것은 클래스가 관련 없는 소수의 멤버들로 이루어졌다는 것을 의미한다.
- Move method
  NUM, NIM은 각각 메소드가 자신이 정의된 클래스에서 사용되어진 횟수, 다른 클래스에서 사용되어진 횟수를 측정하고 RNM은 메소드가 사용한 기능중 다른 클래스에서 정의되어진 기능의 비율을 측정한다. 따라서 CBMC, NUM 값이 작고, RNM 값이 크다는 것은 메소드가 자신이 정의된 클래스의 속성보다 다른 클래스의 속성을 더 많이 사용한다는 것을 의미한다. 또, CBMC, NUM 값이 작고 NIM 값이 크다는 것은 그 메소드를 자신이 정의된 클래스에서보다 다른 클래스에서 더 많이 사용된다는 것을 나타낸다.
- Move attribute
  RNA는 속성을 사용하는 것들 중 다른 클래스에서 정의된 것들의 비율을 측정한다. 따라서 CBMC, NUA 값이 작고, RNA 값이 크다는 것은 속성이 정의된 클래스에서보다 다른 클래스에서 그 속성을 더 많이 사용하고 있다는 것을 의미한다.
- Push down method
  CBMC, RNM, NUM, NIM 값이 모두 작다는 것은 그 메소드가 거의 사용되지 않거나 서브 클래스에서만 사용되어진다는 것을 의미한다.
- Push down attribute
  CBMC, NUA, RNA 값이 작다는 것은 속성이 사용되지 않거나 서브 클래스에서만 사용되어진다는 것을 의미한다.

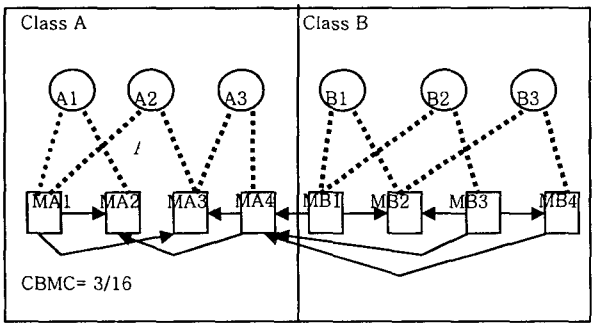
<표 2: 가능한 리팩토링의 조건>

리팩토링	CBMC	NMC	NAC	NUA	RNM	NUM	RNA	NIM
Extract class	↓	↑	↑					
Inline Class	↓	↓	↓					
Move method	↓ ↓				↑	↓ ↓		↑
Move attribute	↓			↓			↑	
Push down method	↓				↓	↓		↓
Push down attribute	↓			↓			↓	

(표 2에서 ↓ 는 작은 값을 , ↑ 는 큰 값을 의미한다. 예를 들어 표 2의 2행 Extract class는 CBMC 값이 작고, NMC, NAC 값이 크면 Extract class 리팩토링을 적용가능하다는 것을 나타내고 있다.)

5. 예제 (Examples)

간단한 예제를 가지고 위의 메트릭 값을 측정하여 적절한 리팩토링을 제안하는 과정을 설명한다. 그림 1,2에서 원으로 표시된 것은 속성을, 사각형으로 표시된 것은 메소드를 나타낸다. 그리고 점선으로 연결된 선은 참조관계를 나타내고 화살표로 연결된 선은 호출 관계를 나타낸다. 예를 들어 그림1에서 메소드 MA1은 A1을 참조하고, 메소드 MA1은 메소드 MA2를 호출한다. 아래 그림에서 메소드 MA4를 살펴보면 클래스 A의 멤버만을 사용하고 있지만 클래스 A에서보다 클래스 B에서 더 많이 사용되고 있어 클래스 A와 클래스 B의 결합이 복잡하다. 따라서 메소드 MA4를 클래스 B로 옮기는 것이 타당해보인다. (리팩토링의 최고 결정자는 유지보수자이다. 따라서 리팩토링을 하지 않고 이 디자인을 그대로 사용할 수도 있다.)



<그림 1 : Move method 리팩토링 예제>

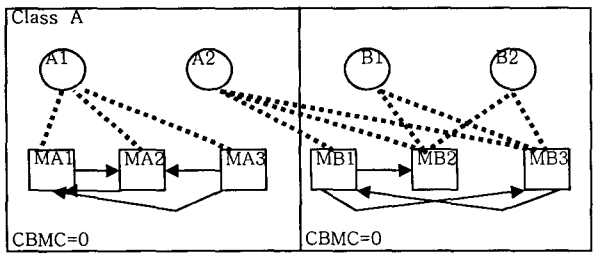
클래스 A에 대해서 위의 메트릭 값을 측정하여 표 3과 표 4에 정리하였다. 표4에서 MA4의 값을 보면 NUM, RNM은 각각 0으로 값이 작고, NIM은 3이므로 비교적 크다. 따라서 메소드 MA4에 move method 리팩토링을 적용가능하다는 것을 찾아낼 수 있다.

<표 3: 속성에 대한 측정값>

	NUA	RNA
A1	2	0
A2	2	0
A3	2	0

<표 4: 메소드에 대한 측정값>

	NUM	RNM	NIM
MA1	0	0	0
MA2	2	0	0
MA3	2	0	0
MA4	0	0	3



<그림 2 : Move attribute 리팩토링 예제>

그림 2에서 보면 속성 A2는 클래스 B에서만 사용되고 있다. 따라서 속성 A2를 클래스 B로 옮기는 것이 타당해보인다. 클래스 A의 속성들에 대해서 메트릭값을 측정해 표 5에 정리하였다. 속성 A2에 대해서 살펴보면 NUA=0(최저값), RNA=1(최대값)이므로 move attribute를 적용가능하다는 것을 찾아낼 수 있다.

6. 관련연구

최근에 언제, 어떤 리팩토링을 적용할 것인가를 결정하는 문제와 관련된 몇몇 연구가 발표되었다([5],[6],[7]).

[7]은 메트릭을 사용하여 리팩토링이 필요한 곳을 찾는다는 점에서는 본 논문의 접근 방법과 비슷하다. 하지만 [7]에서는 멤버들 사이의 거리를 측정하는 응집도 메트릭을 사용하여 Move method/attribute, Extract/inline class 4가지 리팩토링에 대해서 다루고 있다. 그리고 move method 리팩토링이 필요한 구조를 찾을 때 메소드가 다른 클래스에서 정의된 멤버를 더 많이 사용하는 경우만 고려하기 때문에 그림 1과 같은 경우에는 적절한 리팩토링을 제안하기 어렵다.

7. 결론 및 향후 연구

본 논문에서는 리팩토링이 필요한 구조를 체계적으로 결정하기 위해 메트릭을 사용하였다. 특히 6가지 리팩토링의 결정을 보조하기 위해서 응집도 메트릭을 선택하고 이를 보조하는 다른 메트릭을 정의하였다. 그리고 측정한 메트릭값을 분석하여 적절한 리팩토링을 제안하는 방법을 설명하였다.

본 논문에서 사용한 메트릭은 코드를 정적으로 분석하여 비교적 쉽게 측정할 수 있는 장점이 있다. 하지만 적절한 리팩토링을 제안하기 위해서 분석할 때 정확한 기준값이 존재하지 않기 때문에 분석이 어렵다. 이를 해결하기 위해서는 많은 실제 산업 시스템에 적용시켜서 측정값 분석을 위한 휴리스틱(heuristic)을 개발하여야 한다. 현재 위에서 제안한 메트릭을 실제 시스템에 적용시켜 결과를 분석하여 메트릭의 효율성을 검증하는 작업을 진행중에 있다.

<표 5 클래스 A 속성>

	NUA	RNA
A1	3	0
A2	0	1

Refernece

[1] M. Fowler, *Refactoring: improving the design of code*, Addison-Wesley, New York, 1999

[2] James M. Bieman and Byung-Kyoo Kang, " Measuring Design-Level Cohesion" , *IEEE Transactions on Software Engineering, Vol 24, No.2, February 1998*

[3] Briand LC, Daly JW, and W□ st J., "A unified framework for cohesion measurement in object-oriented system", *Empirical software Engineering Journal 1998;3(1):65-117*

[4] Chae HS, Kwon YR, and Bae DH, "A cohesion measure for object-oriented classes", *Software-Practice and Experience 2000;30(12):1405-1431*

[5] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin, "Automated Support for Program Refactoring using Invariants", *In Proc. Int. Conf. On Software Maintenance, pages 736-743,2001*

[6] St□phane Ducasse, Matthias Rieger, and Serge Demeyer, "A language independent approach for detecting duplicated code", In Hongji Yang and Lee white, editors, *Proc. Int'l Conf. Software Maintenance, pages 109-118. IEEE Computer Society Press, September 1999*

[7] Frank Simon, Frank Steinbru□ckner, and Clause Lewerent, "Metrics Based Refactoring", In Proc. 5th European Conference on Software Maintenance and Reengineering, pages 30-38. IEEE Computer Society Press, 2001