객체 지향 패러다임에서의 코드 재사용을 위한 응집도 레벨 식별 모범 사례

변은영[*1], 박보경[*], 장우성[*], 김영철[*]
[*]홍익대학교 소프트웨어공학연구실
{eybyun[1], bk, jang, bob}@selab.hongik.ac.kr

Best Practice on identifying the level of cohesion for reusing source code in object-oriented paradigm

Eun-Young Byun[*1], Bo-Kyung Park[*], Woo-Sung Jang[*], Young-Chul Kim[*]
[*]SE Lab., Hongik University

요 약

소프트웨어의 재사용은 소프트웨어 개발의 품질과 생산성을 높이고 개발 비용을 절감할 수 있다. 소프트웨어 재사용을 위해서 가장 중요한 것은 소스 코드에서 재사용성이 높은 모듈을 추출하기 위해 모듈화에 적합한 소스 코드를 식별해야 한다. 이를 위해서 우리는 코드 가시화를 적용한다. 정량적 지표인 응집도 지표와 추출하여 코드의 복잡도와 재사용성을 판단한다. 본 논문에서는 객체 지향 패러다임에서 응집도를 재정의 하여 제안하고 모듈 단위를 메소드로 정의하여 모듈의 응집도를 추출한다. 이를 통해 모듈화가 가능한 코드의 재사용과 복잡한 코드의 리팩토링이 가능하도록 한다.

1. 서론

현재 기술이 급속하게 발전하고 소비자 요구가 증가하고 있다. 따라서 기존 기능 보다 다양한 기능을 가진 소프트웨어들이 개발되고 있다. 이로 인해 소프트웨어 시스템의 규모가 크고 복잡한 경우가 지속적으로 증가하고 있다. 이러한 소프트웨어를 개발하는 데는 많은 시간과 비용이 소비된다[1].

소프트웨어와는 다르게 하드웨어 분야에서는 기본 구성 단위인 칩이 모듈화 되어 있다. 새로운 하드웨어 구성 시, 모듈화 된 칩들 중에서 필요한 기능을 수행하는 칩을 재사용하여 하드웨어를 구성한다. 이러한 방법으로 하드웨어 분야에서는 개발 시간과 비용을 절감해 가는 추세이다. 그러므로 소프트웨어 분야에서도 이를 절감하기 위해 재사용이 이루어져야 한다.

소프트웨어 재사용이 적용되는 대상 중에 제일 직접적인 영향을 주는 대상은 소프트웨어 아키텍처인 소스 코드이다. 소스 코드를 재사용하기 위해서는 추상적인 코드를 모듈화 해야 한다. 모듈화를 위한 구체적인 지표로 응집도(Cohesion)를 사용하여 재사용이 가능한 모듈을 추출한다. 그리고 복잡도를 판단하여 리팩토링 할 수 있다.

본 논문의 구성은 다음과 같다. 2장 관련연구는 재사용과 코드 복잡도에 대해 설명한다. 3장은 기존의 응집도 개념을 통해 객체 지향 관점의 응집도를 재정의 한다. 4장은 추출 가시화로 추출한 결과에 대해 기술한다. 마지막으로 5장에서는 결론 및 향후 연구를 언급한다.

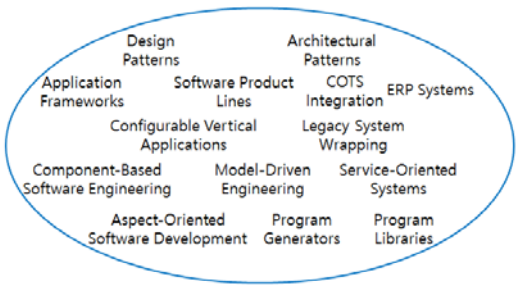
2. 관련연구

2.1 재사용

소프트웨어 재사용은 기존에 개발한 소프트웨어의 여러 가지 요소들을 새로운 소프트웨어 개발이나 유지보수에 재적용하는 것이다. 소프트웨어 재사용을 통해서 점점 커지고 복잡해지는 소프트웨어의 개발 품질과 생산성을 높일 수 있다.

소프트웨어 재사용에는 컴포넌트를 재사용하는 컴포넌트 기반 개발 방법(CBD : Component based development), 설계 지식을 재사용하는 디자인 패턴, 소프트웨어의 상위 설계 및 기본 구조를 재사용하는 소프트웨어 아키텍처, 제품군에 대응할 수 있는 소프트웨어 프로덕트 라인(Software product line) 등이 있다. 그림 1은 소프트웨어 재사용의 적용 범위를 보였다. 소프트웨어 재사용의 적용 범위는 점점 넓어지는 것을 볼 수 있다[2,3].

본 논문은 소프트웨어의 재사용을 위해서 소프트웨어 아키텍처인 소스 코드를 대상으로 한다.



(그림 1) 소프트웨어 재사용의 적용 범위

2.2 코드 복잡도

소프트웨어 복잡도는 크게 2가지로 구분된다. 요구 사항 자체가 복잡하기 때문에 소프트웨어가 복잡해지는 본질적 복잡성(essential complexity)과 여러 이유로 개발자가 스스로 초래한 돌발적 복잡성(accidental complexity)이 있다. 소프트웨어는 요구사항에 의해 설계되기 때문에 요구사항의 영향을 받는 본질적 복잡성은 줄일 수가 없다. 반면에 돌발적 복잡성은 레거시 코드, 플랫폼 버그, 툴의 사용성 등 다양한 원인으로 발생한다[4].

돌발적 복잡성에 가장 직접적으로 영향을 주는 원인은 레거시 코드라고 할 수 있다. 레거시 코드는 소프트웨어 프로젝트의 효율성에 큰 영향을 준다. 레거시 코드의 복잡도의 정량적 지표로 대표적으로 사용되는 방법에는 순환복잡도(Cyclomatic Complexity), 결합도(Coupling), 응집도 등이 있다. 이 논문에서는 소스 코드에서 ASTM(Abstract Syntax Tree Metamodel)을 추출하고 코드 복잡도를 판단하는 여러 가지 지표 중 응집도를 적용하여 응집도에 따라 재사용에 적합한 모듈을 판단할 수 있다[5,8].

2.3 응집도

응집도는 한 모듈 내에 필요한 구성 요소들의 친화력을 측정하는 척도이다. 한 모듈의 응집도가 높을수록 모듈에 필요한 구성 요소들이 그 모듈 내에 존재할 확률이 높으며, 응집도가 낮을수록 모듈 내에 서로 관련 없는 구성 요소들이 존재한다. 응집도는 정보 은닉 개념을 확장한 것으로서, 응집도가 높은 모듈은 프로그램의 다른 부분을 수행하는 다른 모듈과의 상호작용이 거의 없이 한 모듈 내에서 하나의 임무를 수행할 수 있다.

<표 1> 기존의 응집도 정의

응집도	정의
Functional Cohesion	모듈 내 모든 구성요소들이 한 가지 기능과 연관이 있는 경우
Sequential Cohesion	모듈 내 한 구성요소의 출력이 다른 구성 요소의 입력이 되는 경우
Communicational Cohesion	모듈이 여러 가지 기능을 수행하며 모듈 내 구성 요소들이 같은 입력 자료를 이용하거나 동일한 출력 데이터를 만들어 내는 경우
Procedural Cohesion	모듈 내 구성 요소들이 연관되어 있고 특정 순서에 의해 수행되어야 하는 경우
Temporal Cohesion	모듈 내 구성 요소들이 서로 다른 기능을 같은 시간대에 실행하는 경우
Logical Cohesion	모듈 내 구성 요소들이 논리적으로 연관된 임무나 비슷한 기능을 수행하는 경우
Coincidental Cohesion	모듈 내 구성 요소들이 서로 관련이 없는 경우

기존의 응집도의 종류는 기능적, 순차적, 교환적, 절차적, 시간적, 논리적, 우연적 응집도로 나누어진다. 표 1은 기존 절차 지향 패러다임의 응집도 정의를 보인다.[6]

3. 객체 지향 관점의 응집도

그림 2는 응집도에 따른 소프트웨어 재사용 품질이다. 응집도 종류의 각 정의에 따라 그림 2와 같이 함수적 응집도로 갈수록 응집력이 높아 코드 복잡도를 낮출 수 있고 소프트웨어 품질이 좋다. 따라서 재사용에 적합한 모듈이라고 판단할 수 있다. 반대로 우연적 응집도로 갈수록 응집력이 낮아 코드 복잡도를 높이고 소프트웨어 품질이 나쁘다. 따라서 재사용에 부적합한 모듈이라고 판단할 수 있다[6].

(그림 2) 응집도에 따른 소프트웨어 품질

표 2는 객체 지향 지향 응집도의 재정의이다. 모듈의 단위가 메소드이기 때문에 절차 지향 관점과 유사한 부분이 많다. 하지만 절차적 응집도에서는 객체 지향의 특성에 따라 절차 지향 관점과 차이가 있다. 객체 지향에서의 절차적 응집도는 하나의 클래스에 있는 메소드들이 여러 개 호출되는 경우로 정의한다. 즉, 하나의 객체에 대한 작업이 순서대로 이루어진다는 것을 의미한다. 클래스라는 개념이 존재하지 않는 절차 지향 관점과 차이가 있는 것을 확인할 수 있다.

<표 2> 객체지향 관점의 응집도 재정의

응집도	정의
Functional Cohesion	• 대입 되는 변수가 공통적으로 사용되는 경우로 정의한다. • 메소드 호출이 한 번 일어난 경우로 정의한다.
Sequential Cohesion	• 메소드의 리턴 된 결과가 다음 메소드의 입력 파라미터로 쓰이는 경우로 정의한다.
Communicational Cohesion	• 메소드 호출에 공통된 파라미터가 입력되는 경우로 정의한다.
Procedural Cohesion	• 하나의 클래스에 있는 메소드들을 여러 개 호출하는 경우로 정의한다.
Temporal Cohesion	• 메소드 호출이 일어나지 않고 변수의 초기화만 실행된 경우로 정의한다.
Logical Cohesion	• switch문이 쓰여 case에 따라 비슷하지만 다른 작업을 수행하는 경우로 정의한다.
Coincidental Cohesion	• 위의 경우가 모두 아닌 경우로 정의한다.

다음은 객체지향 관점의 응집도 샘플이다. 표 1에서 나오는 응집도의 정의에서 모듈을 객체 지향에서의 메소드로 정의하고 객체 지향 언어인 Java를 통해서 응집도의 예시를 구현하고 구조를 살펴본다.

- Functional Cohesion Sample Code

```
public class Functional {
    double computePay(float hours, float wage){
        double pay;
        pay = wage * 40.0;
        pay = …;
        return pay;                    }
    double squareRoot(double rear){
        double result = Math.sqrt(rear);
        return result;                 }
    int getSum(int num){
        int i, sum = 0;
        for(i=0;i<=num;i++) sum += i;
        return sum;                    }}
```

- Sequential Cohesion Sample Code

```
public class Sequential {
    seqGrade grade;
    void processGrade(){
        …
        numberGrade = grade.getGrade();
        letterGrade = grade.computeLetter(numberGrade);
        grade.displayLetter(letterGrade);          }
    comMatrix matrix;
    void Process_Matrix(){
        …
        inverse_matrix = matrix.inverse(aMatrix);
        matrix.printMatrix(inverse_matrix);   }}
```

- Communicational Cohesion Sample Code

```
public class Communicational {
    comMatrix marix;
    void Compute_Matrix(… ){
        int[][] aMatrix={ {0},{0} };
        …
        transform_matrix = marix.trans(aMatrix);
        inverse_matrix = marix.inverse(aMatrix); }}
```

- Procedural Cohesion Sample Code

```
public class Procedural extends Letter{
    void sendLetter(){
        writerBody();
        writerSalutation();
        send();                        }}
```

- Temporal Cohesion Sample Code

```
public class Temporal {
    int no_student;
    int no_department;
    String university_name;

    void Init_Variables(){
        no_student = 0;
        no_department = 0;
        university_name = "Hongik University";}}
```

- Logical Cohesion Sample Code

```
public class Logical {
    long solve_equation(int no_equ, long x){
        long y = 0;
        switch(no_equ){
        case 1 : y = 5 * x * x;
                 break;
        case 2 : y = 6 * x * x * x;
                 break;
        case 3 : y = 7 * x * x * x * x;
                 break;                }
        y = y+6 * x+4;
        return y;                      }}
```

- Coincidental Cohesion Sample Code

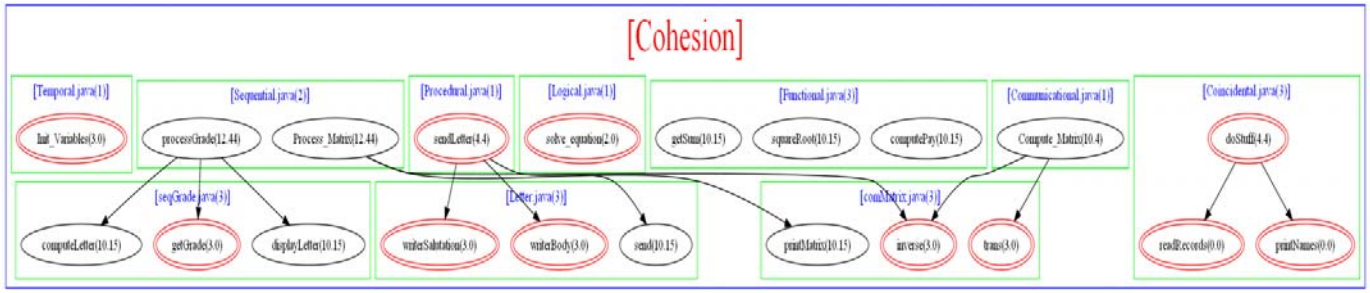
```
public class Coincidental extends RecordProcess{
    int average, totalScore;
    boolean done;
    Letter l;

    void doStuff(){
        readRecords();
        average = totalScore / 10;
        l.printNames();
        done = true;
        return;                        }}
```

4. 추출 가시화

응집도 추출의 대상이 될 코드는 3장에서 구현한 Java 기반의 코드이다. 이 코드는 응집도 종류별 추출이 가능하도록 간단하게 구현된 코드이다.

그림 3에서 가시화의 전체 결과를 확인할 수 있다. 빨간색 이중 노드로 표시된 노드는 응집도가 낮아 재사용에 부적합한 모듈임을 확인할 수 있고 검정색 노드로 표시된 노드는 응집도가 높아 재사용에 적합한 모듈임을 확인할 수 있다. 따라서 모듈화가 가능한 코드의 재사용이 가능하고 응집도가 낮은 모듈은 복잡한 코드로 판단하여 리팩토링이 가능하다.

표 3은 가시화 결과를 응집도 각 종류별로 정리한 것 이다. 단, 순차적 응집도와 교환적 응집도에서는 절차적 응집도를 포함하고 있음으로 해당 응집도의 점수보다 높은 점수가 나온다. 제일 밖에 위치한 파란 서브 그래프는 패키지, 그 안에 초록색 서브 그래프는 클래스를 의미한다. 클래스 안에 타원형 노드는 메소드를 의미한다. 화살표는 메소드 간의 호출을 의미한다. 응집도가 높은 메소드는 빨간색인 이중 타원으로 표현된다.



(그림 3) 가시화 그래프

5. 결론 및 향후 연구

현재의 소프트웨어가 점점 커지고 복잡해지는 시점에 코드의 재사용성은 품질과 생산성에 결정적인 역할을 한다. 본 논문은 재사용에 적합한 모듈의 판단을 위해 객체지향 기반의 코드를 분석하여 응집도를 추출해 보았다. 예시 코드에 따라 응집도가 추출되는 것을 확인할 수 있다. 지금까지 연구로는 응집도의 개념이 너무 추상적이기 때문에 구체적인 응집도의 예시를 찾아내는데 어려움이 있었다. 향후에는 구체적인 응집도의 경우를 추가적으로 검출해 조금 더 정확한 의미의 응집도 개념을 세우고 Java 기반의 코드를 자동화 추출할 수 있는 연구를 진행 할 예정이다.

ACKNOWLEDGEMENT

이 논문은 2015년 교육부와 한국연구재단의 지역혁신창의인력양성사업의 지원을 받아 수행된 연구임(NRF-2015H1C1A1035548).

참고문헌

[1] Tomas A. Standish, “An Essay on Software Reuse”, IEEE Transactions on Software Engineering, vol.10

[2] Ivar Jacobson, “Software reuse:Architecture, Process and Organization for Business Success”

[3] Ian Sommerville, “Software Enginerring” 3rd Ed. Paperback

[4] Frederick P. Brooks Jr. “The Mythical Man-Month: Essays on Software Engineering ” 2rd Ed. Anniversary Edition

[5] Hyun Seong Son , So young Moon and R. Young Chul Kim, “Replacing Source navigator with Abstract Syntax Tree Metamodel(ASTM) on the open source oriented tool chains SW Visualization,″ The 5th International Conference on Convergence Technology 2015, vol. 5, no. 1, pp. 366-367, June. 2015.

[6] Stephen R. Schach, “Software Engineering” 2rd Ed.

[7] 강건희, 손현승, 김영수, 박용범, 김영철, “SW 가시화 기반 리팩토링 기법 적용을 통한 정적 코드 복잡도 개선”, 제21권, 제2호, 한국정보처리학회, 2014. 11

[8] 권하은, 박보경, 이근상, 박용범, 김영수, 김영철, “코드 가시화부터 모델링 추출을 통한 역공학 적용”, 제21권, 제2호, 한국정보처리학회, 2014. 11