SW 가시화 기반 리펙토링 기법 적용을 통한 정적 코드 복잡도 개선

강건희*, 손현승, 김영수**, 박용범***, 김영철*
*홍익대학교 소프트웨어공학연구실
**정보통신진흥원 소프트웨어공학센터
***단국대학교 컴퓨터과학과
e-mail:kang@selab.hongik.ac.kr*, ysgold@nipa.kr**, ybpark@dankook.ac.kr***

Improving Static Code Complexity with Refactoring technique based on SW visualization.

Geon-hee Kang*, HyunSeung Son, Youngso*o Kim**, Yong B. Park***
, R. Young Chul Kim*
*SE Lab, Dept. of Computer Information Communication, Hongik University
**National IT Industry Promotion Agency
***Dept, of. Computer Science, Dankook University

요 약

기존의 소프트웨어 개발은 SW품질을 중요시 하지만, 고품질에 대한 문제가 아직도 존재한다. 또한 기존 레가시 시스템는 개발자나 설계의 부재 경우가 많고, 코드의 내부 복잡도와 모듈간의 결합도가 높을 가능성이 높다. 따라서 코드 가시화를 통한 복잡도 개선은 고품질화과 더불어 코드 모듈의 재사용과 유지보수등과 직접적 관련성이 있다. 본 논문은 기존 SW가시화용 자동 Tool Chain[3] 기반에서 여러 리펙토링 방법 절차 적용으로 복잡도 개선을 제안 한다. 이런 코드 가시화가 결과적으로 타깃의 결합도를 줄일 수 있다. 기존의 레가시 코드에 자동 Tool chain적용은 고품질 적용이 충분히 예상된다.

1. 서론

현재 우리나라의 소프트웨어 개발 환경에서는 소프트웨어의 품질을 중요시 하지만 대부분의 중소개발기업에서는 구현위주의 개발로 인해 고품질 소프트웨어와 동떨어져 있다[1]. 구현 위주의 개발을 하게 되면 소프트웨어의 특성상 가시화가 되지 않기 때문에 시스템 전체의 구조가 복잡도가 높아지고 일회용성의 소프트웨어가 만들어지기 쉽다. 일회용성 소프트웨어를 재사용이 가능하고 유지보수를 쉽게 하려면 전체적인 소프트웨어의 아키텍쳐를 알아야 한다. 복잡도가 심하다면 복잡도를 줄이고 재사용을 하고자 하는 모듈을 정하여 모듈간의 결합도를 줄여 모듈간의 독립성을 만들어 주는 것이 중요하다. 그래서 소프트웨어 아키텍쳐의 가시화가 중요하다. 소프트웨어의 아키텍쳐가 가시화됨으로서 전체적인 복잡도와 결합도의 판단이 가능하다.

본 논문에서는 소프트웨어 가시화를 통한 결합도의 리펙토링 방법을 제안한다. 2장에서는 SW 아키텍쳐의 가시화 와 결합도에 대해서 언급한다. 3장에서는 SW가시화 과정에 대해서 설명하고 4장에서는 결합도 리펙토링 과정에 대해서 설명한다. 5장에서는 타깃 프로그램에 대한 적용사례를 보여준다. 6장에서는 결론 및 향후연구에 대해서 언급한다.

2. 관련연구

2.1 SW 아키텍쳐 가시화

성공적인 소프트웨어개발, 품질관리를 하려면 소스 코드와 소프트웨어 개발 프로세스에 대한 관리가 필요하다. 소프트웨어공학 프로세스는 이러한 관리를 위한 전통적인 방법이다. 하지만 소프트웨어공학 프로세스에 의한 소프트웨어개발 품질관리를 수행하기는 전문화된 인력과 자본이 부족하다. 그렇기 때문에 부족한 인력과 자본을 메우기 위해서는 소프트웨어의 아키텍쳐 가시화가 중요하다.

소프트웨어의 아키텍쳐 가시화는 소프트웨어 개발의 가장 어려운 점인 소프트웨어 비가시성을 극복함으로서 전체 소프트웨어개발의 과정을 파악 할 수 있다. 그리고 소프트웨어개발의 과정과 구조를 파악함으로써 전문적이지 않더라도 고품질을 위한 소프트웨어 개발과 품질 관리가 가능하다[1].

2.2 결합도

결합도는 프로그램에서 모듈간의 상호의존 혹은 연관관계를 의미한다. 그렇기 때문에 강한 결합도가 모듈간의 의

* 이 논문은 2014년도 정부(미래창조과학부)의 재원으로 한국연구재단-차세대정보 · 컴퓨팅기술개발사업(No. 2012M3C4A7033348)과 2014년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(NRF-2013R1A1A2011601).

존성을 높여 변경, 유지보수 더 나아가 모듈의 재사용성과 유지보수의 엄청난 영향을 끼치게 된다. 결합도는 자료, 스탬프, 제어, 외부, 공유, 내용 총 6가지의 결합도가 있다. 자료 결합도에서 내용 결합도로 갈수록 모듈간의 상호의존성이 높아진다 그렇기 때문에 소프트웨어 공학적으로 품질이 저하된다. 그리고 코드의 재사용성과 유지보수의 들어가는 시간과 자본이 많이 든다 [2].

본 논문에서는 각 결합도별로 어떠한 코딩 규칙이 있는지 그리고 그 코딩 규칙에 대해서 어떻게 리펙토링 해야 낮은 결합도를 가질 수 있는지 알아본다. 그림 1은 정량적 결합도 측정지표의 점수이다[3].

그림 1 결합도 측정점수

3. SW 아키텍쳐 가시화 과정

SW가시화를 위해서는 소스 분석, 데이터베이스 저장, 구조 분석, 시각화 총 4가지의 단계가 필요하다.

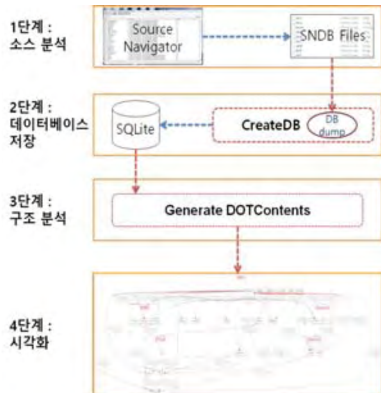

그림 2 SW 아키텍쳐 가시화

그림 2는 소프트웨어 가시화의 전 과정을 보여준다.

○ 1단계 : 소스 분석

소스 분석 단계는 기존 레가시 코드를 분석기을 통해 분석한다. 분석된 코드의 정보는 각 파서에 맞게 파일로 추출이 된다. 현재는 기존 오픈소스 Source Navigator를 사용한다.

○ 2단계 : 데이터베이스 저장

데이터베이스 저장 단계는 파서를 통해 분석한 정보를 저장한다. 이렇게 저장된 정보는 가시화 과정에서 필요한 정보만 효과적인 추출 과정이 포함된다. 또한 데이터베이스는 SQLite를 사용한다. Source Navigator를 이용해 추출된 파일(SNDB)이 바이너리로 구성이 되어서 DBdump 파일을 읽는 정보로 바꾼 뒤 직접 개발한 CreateDB프로그램을 사용하여 DB 내 입력을 하게 된다.

○ 3단계 : 구조분석

구조분석 단계에서는 미리 정한 모듈의 정의에 따라서 DB에 분류된 정보를 재해석한다. 2단계에서 저장된 정보로서 모듈 정보를 먼저 불러온다. 그리고 모듈과 그 외 요소들 간의 관계정보를 통해 구조를 GenerateDotContents라는 프로그램으로 분석한 뒤, 시각화를 위한 DotScript를 생성한다. 이 단계에서 미리 정한 품질지표의 측정치를 위한 정보도 같이 추출한다. 이 품질지표는 결합도를 우선으로 정하여 강~약 결합도 검출 정보를 불러온다.

○ 4단계 : 가시화

가시화 단계에서는 3단계에서 재해석한 모듈의 정보와 모듈간의 관계정보, 품질지표 정보의 의미를 이미지화 한다. Dot Script를 통하여 이미지화 한다[3-4].

4. 결합도 리펙토링 방법

4.1 자료 결합도, 스탬프 결합도

자료 결합도는 모듈 간의 인터페이스가 자료 요소로만 구성될 때의 결합도이다. 어떤 모듈이 다른 모듈을 호출하면 매개변수나 인수로 데이터를 넘겨주고 받은 상태이다. 그리고 스탬프 결합도는 모듈 간의 인터페이스가 기본 자료 요소 이외에 배열 혹은 레코드, 자료구조 등이 전달될 때 의 결합도이다[2]. 결합도 가운데서는 낮은 결합도이기 때문에 굳이 리펙토링을 하지 않아도 된다. 물론 스탬프 결합도를 자료 결합도로 바꿀 수는 있지만 그렇게 되면 롱 파라미터가 코드의 나쁜 냄새를 유발 할 수 있다.

4.2 제어 결합도

```
for(int i=0; i < OBJ_LENGTH; i++)
{
    if(i%3 == 2)
    {
        if(m_motorList[i].Pick(m_device,&vPickRayOrig,&vPickRayDir,FAR_Z-NEAR_Z))
        {
            m_nCurrentObject = i;
            return;
        }
    }
}
for(int i=0; i < OBJ_LENGTH; i++)
{
    if(i%3 == 1)
    {
        if(m_motorList[i].Pick(m_device,&vPickRayOrig,&vPickRayDir,FAR_Z-NEAR_Z))
        {
            m_nCurrentObject = i;
            return;
        }
    }
}
for(int i=0; i < OBJ_LENGTH; i++)
{
    if(i%3 == 0)
    {
        if(m_motorList[i].Pick(m_device,&vPickRayOrig,&vPickRayDir,FAR_Z-NEAR_Z))
        {
            m_nCurrentObject = i;
            return;
        }
    }
}
```

그림 3 중복사용 제어문

그림 3은 중복사용 제어문이다. 제어 결합도는 한 모듈에서 다른 모듈로 논리적인 흐름을 제어하는데 사용하는 제어 요소(Function Code, Switch, Tag, Flag)가 전달 될 때의 결합도이다. 대부분의 조건식에 들어가는 논리 값이 함수형태로 일 때 제어 결합도가 발생한다[2]. 이 결합도 제거는 약 결합도(자료, 스탬프)로 낮출 수 없다. 하지만 중복된 패턴을 찾아 제어 결합도의 수를 줄일 수 있다.

```
    int n1 = CheckPick(2, &vPickRayOrig, &vPickRayDir);
    int n2 = CheckPick(1, &vPickRayOrig, &vPickRayDir);
    int n3 = CheckPick(0, &vPickRayOrig, &vPickRayDir);
    if( (n1==-1) && (n2==-1) && (n3==-1) )
        m_nCurrentObject = -1;
}

int CSimultationView::CheckPick(int degree, D3DXVECTOR3 *pvNear, D3DXVECTOR3 *pvDir)
{
    for(int i=0; i < OBJ_LENGTH; i++)
    {
        if(i%3 == degree)
        {
            if(m_motorList[i].Pick(m_device,pvNear,pvDir,FAR_Z-NEAR_Z))
            {
                m_nCurrentObject = i;
                return i;
            }
        }
    }
    return -1;
}
```

그림 4 메소드화된 제어문

그림 4는 메소드화 된 제어문이다. 조건 변경 시, 중복된 모든 조건을 변경하므로 비효율적이다. 그림 4처럼 하나의 메소드(함수)화 하여 제거하면 모듈간의 제어 결합도가 감소한다.

4.3 외부 결합도

외부 결합도는 어떤 모듈이 외부로부터 선언된 데이터(변수)를 다른 모듈에서 참조할 때 발생하는 결합도이다[2].

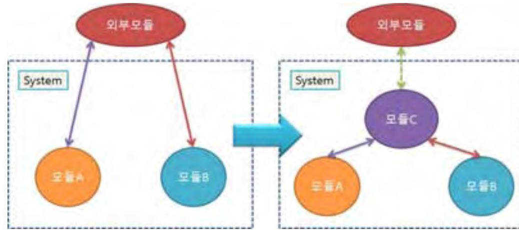

그림 5 외부결합도의 리펙토링 과정

그림 5는 외부 결합도의 리펙토링 과정을 설명이다. 외부 결합도를 줄이려면 외부의 데이터를 직접 참조하지 않고 외부의 데이터를 참조하여 받아오는 모듈을 하나 만든 다음 그 모듈을 참조한다면 데이터만 받아오는 것이 되기 때문에 외부 결합도는 줄고 자료 혹은 스탬프 결합도가 생기게 된다.

4.4 공유 결합도

공유 결합도는 공유되는 공통 데이터영역을 여러 가지 모듈이 사용할 때의 결합도이다[2]. 그렇기 때문에 공유되는 공통 데이터 영역을 사용하는 모든 모듈에 영향을 미치게 되어 모듈의 독립성을 약하게 만든다. 본 논문에서는 가시화 하는 과정에서 같은 전역변수를 참조하는 것만 공유결합도로 추출을 한다. 그래서 전역변수에 대한 리펙토링을 설명한다. 아래 그림 6은 전역변수에 대한 소스코드이다.

```
dWorldID                         m_world;
dSpaceID                         m_space;
static dGeomID                            m_ground;
static dJointGroupID                      m_contactgroup;

ID3DXFont*              g_pFont = NULL;
LPD3DXSPRITE            m_pTextSprite                   = NULL;
```

그림 6 전역변수

그림 6과 같은 전역변수들을 한 클래스에 멤버변수로 넣어준 뒤 그림 7과 같이 전역변수 였던 변수들의 getter를 생성하여 주면 getter를 통해 데이터를 받기 때문에 공유 결합도가 없어진다.

```
public:
    dWorldID        static GetWorld()          {   return m_world; };
    dJointGroupID   static GetContactGroup()   {   return m_contactgroup; };
private:
    static dWorldID                      m_world;
    static dSpaceID                      m_space;
    static dJointGroupID                 m_contactgroup;

    ID3DXFont*             g_pFont;
    LPD3DXSPRITE           m_pTextSprite;
```

그림 7 getter생성

그림 7은 전역변수를 getter로 생성한 것이다.

4.5 내용 결합도

내용 결합도는 한 모듈이 다른 모듈의 내부 기능 및 그 내부 자료를 직접 참조 하거나 수정 할 때의 결합도이다[2]. 내용 결합도를 제거하려면 내부 자료의 직접 참조와 수정을 막아주면 된다.

```
BOX_OBJECT *pBox = (BOX_OBJECT *)m_view->m_boxObjectList.GetNext(pos);
if(pBox->bTracking)
{
    data.m_nCommand = COM_SENSOR_RECV_DEGREE;
    data.m_nCode = pBox->theta;
    data.m_nUserNum = 0;
    SendUser(USER_ALL,&data);

    data.m_nCommand = COM_SENSOR_RECV_
    data.m_nCode = pBox->distance;
    data.m_nUserNum = 0;
    SendUser(USER_ALL,&data);
    break;
}
```

```
typedef struct tagPacket
{
    int m_nUserNum;
    int m_nCommand;
    int m_nCode;
}PACKET;
```

그림 8 내부자료의 직접 참조

그림 8은 내부자료의 직접 참조의 예제이다. tagPacket이라는 구조체의 데이터를 직접 참조, 수정하는 소스코드와 tagPacket 의 구조이다. 이렇게 직접 참조 수정하던 소스코드를 그림 9과 같이 CData라는 클래스를 만들어서 tagPacket에 있던 멤버변수를 CData의 멤버변수로 만든 뒤 Getter와 Setter를 통하여 값을 받아와 수정하는 방식으로 하면 내용 결합도를 자료, 스탬프 결합도로 낮춘다.

```
BOX_OBJECT *pBox = (BOX_OBJECT *)m_view->m_boxObjectList.GetNext(pos);
if(pBox->bTracking)
{
    data.SetData(0, COM_SENSOR_RECV_DEGREE, pBox->theta);
    SendUser(USER_ALL,&data);

    data.SetData(0, COM_SENSOR_RECV_RANGE, pBox->distance);
    SendUser(USER_ALL,&data);
    break;
}
```

```
class CData : public CObject
{
public:
    CData();
    virtual ~CData();
private:
    int m_nUserNum;
    int m_nCommand;
    int m_nCode;
```

그림 9 내용결합도 리펙토링

그림 9는 내용결합도 리펙토링의 예제다.

5. 적용사례

코드가시화의 적용사례로 직접 개발된 6족 로봇 시뮬레이터를 적용타깃으로 사용한다[5]. 6족 로봇 시뮬레이터에서는 다관절 로봇의 동작제어를 도구 내의 컨트롤러로 쉽게 할 수 있다. 그리고 실제로 로봇개발 전에 도구 내의 시뮬레이션을 통하여 실제 로봇의 동작을 예측이 가능하다. 즉, 시뮬레이션 로봇이 동작할 환경을 구축한 후 다관절 로봇의 모션을 입력하면 해당 환경에서 동작을 한다. 모델링 도구인 HiMEM[6]과 네트워크로 연결이 가능하다. 이를 통해, Pre-Testing(선-테스팅) 방법을 제안했다[5]. 그림 10은 시뮬레이터의 실행화면이다.

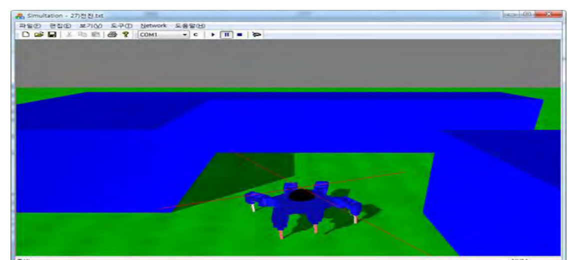
그림 10 6족로봇 시뮬레이션 실행화면

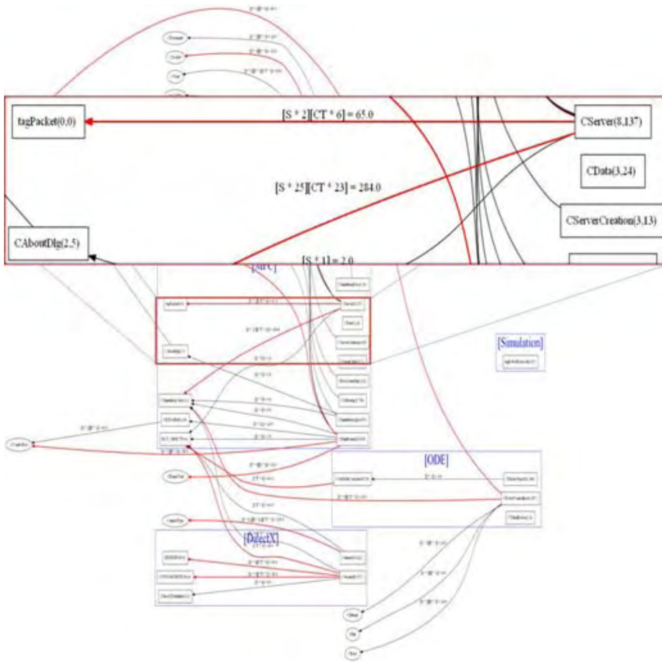

그림 11 리펙토링 전 가시화 한 이미지

그림 11은 기존의 타깃 프로그램을 리펙토링 전 가시화 한 이미지이다. 그림 11과 같은 아키텍처를 제어, 공유, 내용 결합도 총 3단계에 리펙토링을 실행하였다. 그래서 그림 12처럼 아키텍처 이미지를 얻는다. 그림 11은 tagPacket모듈과 CServer모듈간의 결합도를 보여준다.

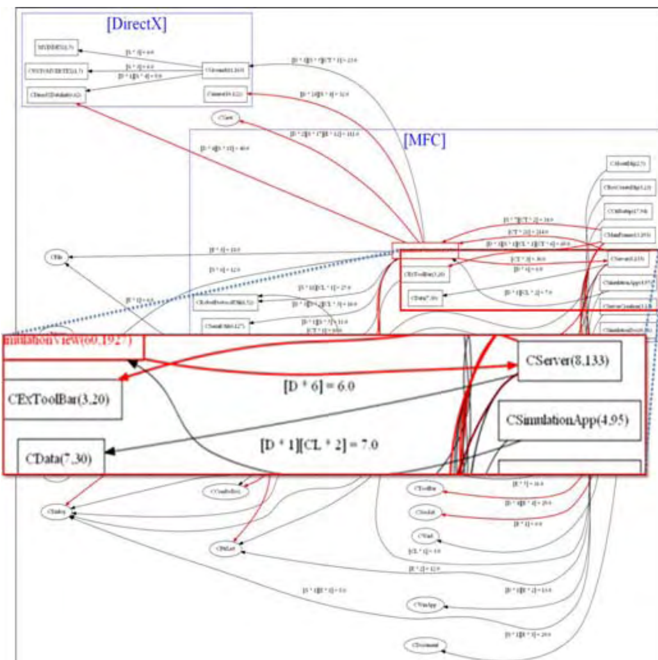

그림 12 리펙토링 이후의 가시화 이미지

그림 12는 리펙토링 이후의 가시화 이미지이다. 공유결합도와 내용결합도와 같은 악성 결합도는 대부분 제거하였고 제어 결합도는 중복된 조건문을 하나의 메소드로 만들어 줄였다. 제어 결합도를 제거하여 5228.9인 결합도가 내용 결합도를 제거하여 2867.2로 약46% 감소한 것으로 결과를 보인다.

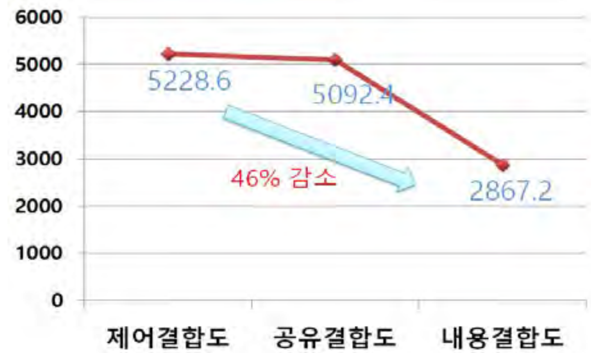

그림 13 리펙토링에 따른 결합도 변화

그림 13은 리펙토링 이후에 결합도 개선변화를 보인다.

6. 결론 및 향후연구

이 논문은 고품질의 소프트웨어를 위해서, 기존의 레카시 코드의 아키텍쳐 가시화를 통하여 레가시 코드의 구조와 결합도에 대한 코드 패턴을 조사·분석한다. 해당 코드 패턴에 대한 리펙토링 방법으로 코드 복잡도 개선을 제안한다. 결과적으로 타깃 프로그램에 대한 리펙토링 과정에서 제어, 공유, 내용 결합도를 감소시킨다. 이를 통해 소프트웨어의 개발 및 레가시 코드를 SW의 구조와 결합도에 대해 실시간으로 정량적인 수치개선을 확인한다. 즉 개발자의 코드 습관을 낮은 결합도 측면으로 개발 할 수 있게 개선이 가능하다. 현재 모든 가능한 결합도에 대한 경우의 수를 가시화하여, 전부는 아니지만 찾을 수 있었다. 향후 최대한 많은 경우의 수를 찾을 것이다. 또한 응집도에 대한 정량적인 품질 측정을 연구하고, 다양한 사례에 적용하고자 한다.

참고문헌

[1] NIPA SW Engineering Center “SW Development Quality Management Manual(SW Visualization)” 2013. 12.

[2] B.Bruegge, A.Dutoit, “Object-Oriented Software Engineering Using UML, Patterns and Java Third Edition”, Pearson, 2010.

[3] Geon-Hee Kang, Keunsang Yi, DongHo Kim, Junsun Hwang, Youngsoo Kim, Young B. Park, R. Young Chul Kim “ A Practical Study on Tool Chain for Code Static Analysis on Procedural Language”, KCC2014, pp.559-561, 2014.

[4] Bokyung Park, Haeun Kwon, Hyeoseok Yang, Soyoung Moon, Youngsoo Kim, R. Youngchul Kim “ A Study on Tool-Chain for statically analyzing Object Oriented Code” KCC2014 pp.463-465 2014.

[5] JaeSoo Kim “embedded System For Small heterogeneous Moving Object based on Modeling & Simulation” (Doctoral dissertation, Hongik Univ., Seoul, Republic of Korea) 2009

[6] Woo Yeol Kim, Hyun Seung Son, R. Young Chul Kim C. R. Carlson “MDD based CASE Tool for Modeling Heterogeneous Multi-Jointed Robots” CSIE 2009 pp775-779 2009