KIPS Trans. Softw. and Data Eng.
Vol.8, No.12 pp.465~472 pISSN: 2287-5905

https://doi.org/10.3745/KTSDE.2019.8.12.465

Building a Code Visualization Process to Extract Bad Smell Codes

Jihoon Park† · Bo Kyung Park†† · Ki Du Kim††† · R. Young Chul Kim††††

ABSTRACT

Today, in many area the rise of software necessity there has been increasing the issue of the impotance of Good Software. Our reality in software industrial world has been happening to frequently change requirements at any stage of software life cycle. Furthermore this frequent changing will be increasing the design complexity, which will result in being the lower quality of software against our purpose the original design goals. To solve this problem, we suggest how to improve software design through refactoring based on reverse engineering. This is our way of diverse approaches to visually identify bad smell patterns in source code. We expect to improve software quality through refactoring on even frequently changing requirements.

Keywords : Refactoring, Bad Smell, Visualization, Process, JavaParser

배드 스멜 코드 추출을 위한 코드 가시화 프로세스 구축

박 지 훈† · 박 보 경†† · 김 기 두††† · 김 영 철††††

요 약

오늘날 많은 영역에서 소프트웨어의 사용 범위가 넓어짐에 따라 좋은 소프트웨어 (Good Software)의 개발은 매우 중요하다. 하지만 현실은 소프트웨어 개발의 모든 단계에서 요구사항의 변경이 자주 발생한다. 또한 빈번한 변경으로 인해 설계 복잡성이 증가하여 원래의 설계 목표보다 소프트웨어 품질이 저하될 수 있다. 이러한 문제 해결을 위해, 배드 스멜(Bad Smell) 코드 추출을 위한 코드 가시화 프로세스를 제안한다. 이 방법은 마틴 파울러의 배드 스멜을 기반으로 소스 코드를 식별하여 리팩토링 영역을 가시화 한다. 잦은 요구사항의 변경에도 리팩토링을 통한 소프트웨어의 설계 개선을 기대한다.

키워드 : 리팩토링, 배드 스멜, 가시화, 프로세스, 자바 파서

1. 서 론

오늘날 많은 영역에서 소프트웨어의 사용 범위가 넓어지면서 다양한 사용자의 요구사항들이 생겨나고 완성된 소프트웨어도 유지보수가 필요하다. 소프트웨어는 새로운 요구사항에 맞춰 수정되고 향상될수록 코드는 점점 복잡해진다. 그로 인해 원래 디자인에서 벗어나 소프트웨어의 품질이 저하되는 상황이 발생한다. 소프트웨어 품질 개선 방법 중 하나인 리팩토링은 소프트웨어의 복잡성을 개선할 수 있다. 하지만 리팩토링을 어떤 부분에 어떻게 적용해야 할지에 대한 기준이 없다.

※ 본 논문은 2019년도 산업통상자원부의 '창의산업융합 특성화 인재양성사업'(과제번호 N0000717)과 2017년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(NRF-2017R1D1A3B03035421).
† 준 회 원 : 한국정보통신기술협회 전임연구원
†† 준 회 원 : 홍익대학교 소프트웨어공학전공 박사과정
††† 정 회 원 : 한국정보통신기술협회 책임연구원
†††† 정 회 원 : 홍익대학교 소프트웨어융합학과 교수
Manuscript Received : October 26, 2018
First Revision: July 17, 2019
Second Revision: September 3, 2019
Accepted : October 4, 2019
* Corresponding Author : R. Young Chul Kim(bob@hongik.ac.kr)

본 논문에서는 배드 스멜 코드 추출을 위한 코드 가시화 프로세스를 제안한다. 이 방법은 역공학을 통하여 소스 코드의 어느 부분에 리팩토링이 필요한지 가시화한다[1-4]. 리팩토링 가시화를 위해 마틴 파울러의 배드 스멜을 적용하였다[5]. 기존 연구에서는 총 22개의 배드 스멜 중 5개 패턴을 식별할 수 있었다. 또한 지속적인 연구를 통해 본 논문에서는 13개 패턴을 추가로 식별할 수 있었다[6-9]. 개발자들은 배드 스멜 코드의 가시화를 통해 소프트웨어 설계를 개선할 수 있다.

제안한 방법의 장점은 리팩토링 가시화의 자동화이다. 국내 중소기업의 경우 인력 및 비용이 부족하기 때문에, 쉽게 적용 가능하도록 오픈 소스 기반의 자동화된 가시화 툴체인을 구축하였다. 가시화 툴체인은 소스 코드를 분석하는 파서와 분석된 데이터를 저장하는 데이터베이스, 분석된 데이터를 이용하여 그래프를 그리는 GraphViz로 이루어져 있다. 이 도구들을 배드 스멜 코드 추출을 위한 코드 가시화에 접목시켜 다음과 같은 기대효과를 얻을 수 있다[10]. 1) 개발 상태를 실시간으로 파악할 수 있다. 2) 객관적, 정량적 분석을 통해 개발의 투명성을 보장할 수 있다. 3) 자동화를 통한 소

프트웨어 관리의 편이성 증가로 소프트웨어 개발 관리 문서화 작업의 간소화가 가능하다. 소프트웨어 개발 프로세스를 가시화함으로써 요구사항부터 테스트까지 상호 추적성을 확보할 수 있는 환경을 구축할 수 있다[11]. 이로 인해, 소프트웨어의 고품질화와 유지보수성 향상을 기대할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로 소프트웨어 가시화 프로세스와 리팩토링을 설명한다. 3장에서는 배드 스멜 코드 추출을 위한 코드 가시화 시스템을 설명한다. 4장에서는 배드 스멜 가시화 패턴과 리팩토링을 통해 개선된 결과를 설명한다. 마지막으로 5장은 결론 및 향후 연구를 언급한다.

2. 관련 연구

2.1 가시화 프로세스

가시화 프로세스는 대규모 소프트웨어 개발과정의 어려움을 해결하기 위한 방법이다. 전체 소프트웨어 개발 과정이 관리가 되어야 좋은 소프트웨어를 개발할 수 있다. 하지만 국내 중소기업의 경우 인력, 비용 등의 부족으로 인하여 관리가 힘들다. 가시화 프로세스는 소프트웨어의 시각화 및 문서화를 통해 부담을 최소화하여 국내 중소기업 등의 소프트웨어 품질 관리를 도와준다. 가시화 프로세스는 소프트웨어의 비가시성을 극복함으로써 전체 개발 과정을 파악할 수 있다. 문서화는 내부 인력간의 업무 이해도 향상과 의사소통을 도와준다. 가시화 프로세스는 요구사항, 분석/설계, 구현, 테스트, 유지보수 과정의 구축을 통해 고객이 요청하는 품질수준의 소프트웨어를 만들 수 있다.

2.2 리팩토링

일반적으로 소프트웨어의 개발단계는 요구사항, 설계, 구현, 테스팅, 유지보수 순으로 이루어진다. 이상적인 개발 방법은 구현하기 전에 완벽히 설계를 마치는 것이다. 설계가 완벽하지 않다면 소프트웨어 오류 및 수정으로 인한 재작업이 발생하게 된다. 따라서 재작업은 유지보수 비용 지출의 원인이 된다. 소프트웨어는 개발 기간 동안 요구사항이 빈번하게 바뀐다. 그러므로 개발자는 예상 가능한 변경에 대해서 융통성 있는 설계를 고려해야 한다. 하지만 융통성 있는 솔루션은 단순한 솔루션보다 복잡하기 때문에 융통성의 비용이 크다[5]. 사전 설계(upfront design)로 리팩토링을 사용한다면 앞의 설계에 소모하는 비용을 최소화할 수 있다. 또한 설계를 전혀 하지 않고 코딩하더라도 리팩토링을 통해 체계적인 구조를 갖출 수 있다.

Mauricio A. Saca [12]는 리팩토링으로 인해 설계 프로세스가 훨씬 쉬워지고 설계비용을 절감 할 수 있다고 언급한다. 그러나 리팩토링은 프로그램의 성능에 항상 긍정적인 영향을 미치지는 않는다. 리팩토링은 프로그램 이해를 돕기 위해 수정하는 과정이다. 따라서 리팩토링은 때로는 프로그램을 느리게 할 수 있다. 하지만 리팩토링으로 인하여 코드가 잘 분배되어 있다면 성능 튜닝에 집중할 수 있는 시간이 늘어난다. 코드가 이해하기 쉬우므로 프로파일러로 성능 분석을 할 때 튜닝을 어떻게 해야 제대로 동작할 것인지 파악하기 쉬워진다. 그러나 리팩토링 후에는 더 나은 설계로 인해 코드를 이해하는 것이 쉬워진다.

Mehmet Kaya et al. [13] 소프트웨어의 리팩토링은 전문 지식의 성숙도를 필요로 한다고 언급한다. 특히 리팩토링은 전문 소프트웨어 엔지니어의 감독 하에 수행되어야 한다. 그렇지 않으면 소프트웨어 초보 엔지니어가 코드 부분을 평가하지 못할 수 있다. 결과적으로 초보자는 리팩토링이 필요한 부분을 찾을 수 있는 경험이 필요하다. 본 논문에서는 전문 지식이 부족한 경우에도 시각화를 통해 리팩토링이 필요한 코드 영역을 자동으로 찾는 방법을 제안한다.

3. 자동 배드 스멜 코드 추출 시스템

3.1 툴체인

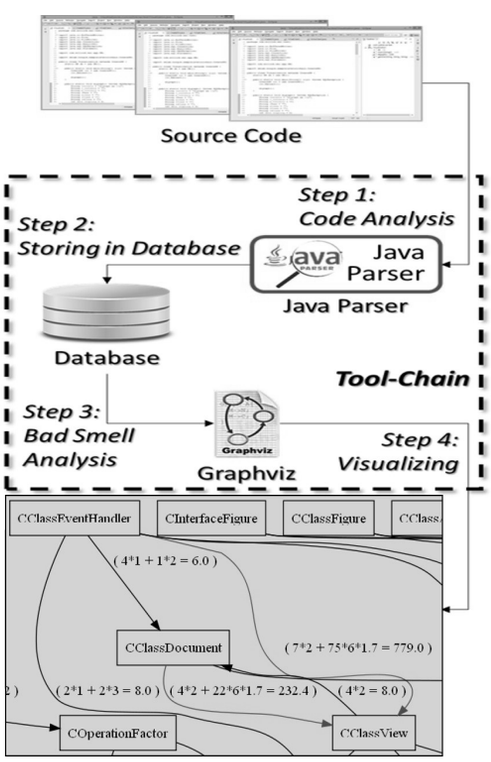

Fig. 1. Bad Smell Code Extraction Automation System Diagram

Fig. 1은 배드 스멜 코드 추출을 위한 자동화 시스템(툴체인)이다[6-9]. 툴체인은 총 4단계로 이루어져 있다. 1단계에서는 정적 분석기로 소스 코드를 분석한다. 2단계에서는 분석된 소스 코드의 정보를 저장할 데이터베이스를 구성한다. 3단계에서는 저장된 데이터를 이용하여 그래프를 그릴 수 있도록 DotScript를 생성한다. 4단계에서는 이미지 생성기로

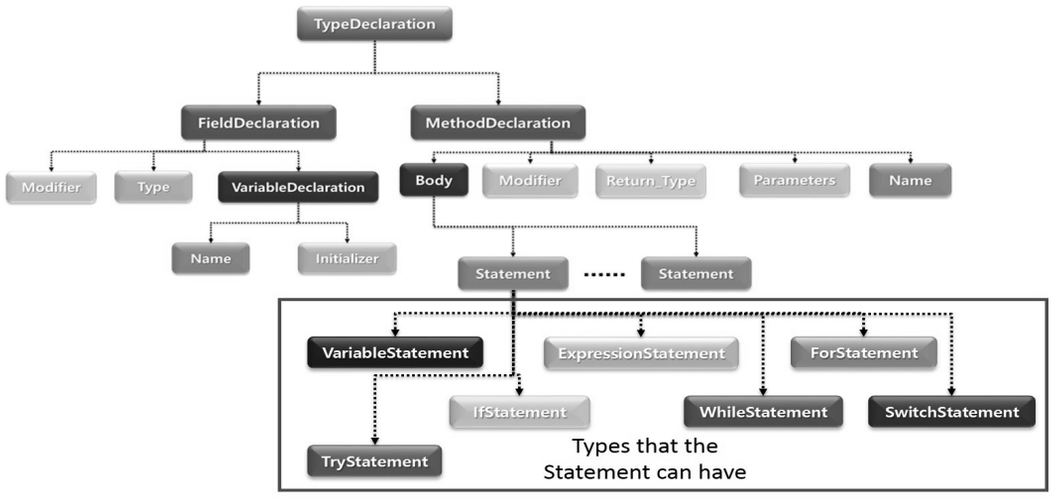

Fig. 2. AST Structure of Java Code

그래프를 생성한다.

툴체인의 구성 도구는 다음과 같다. 정적 분석기로는 JavaParser를 사용하였고, 데이터베이스로는 SQLite, 이미지 생성기로는 GraphViz를 사용하였다. 코드의 정보 분석을 위해, 본 논문에서는 JavaParser를 이용해 AST(Abstract Syntax Tree) 기반의 분석을 수행하였다. AST를 이용하여 모든 소스 코드를 분석했기 때문에 품질 지표에 필요한 모든 정보를 데이터베이스 구성에 따라 자유롭게 사용할 수 있다.

3.2 Java 코드의 AST 구조 파싱 방법

Fig. 2는 Java 코드를 AST로 표현한 것이다. JavaParser를 이용하여 AST 구조로 파싱하였다[14]. Java를 AST 구조로 표현하면 훨씬 더 많은 구조로 표현된다. 본 논문에서는 Fig. 2의 구조를 통해 배드 스멜을 가시화한다. AST 구조에서, TypeDeclaration은 하나의 클래스형태이고 인스턴스 변수인 FieldDeclaration(FD)과 메서드인 MethodDeclaration (MD)을 포함한다. FD는 Modifier(접근자), Type(타입), VariableDeclaration (VD, 변수 선언)으로 구성되며,, VD는 Name(이름), Initializer (초기 값)를 가질 수 있다. MD는 Modifier(접근자), Parameters(파라미터), Name(메서드이름), Return_Type(리턴 값), Body(메서드 내용)로 구성된다. Body는 메서드가 실제로 실행하는 내용이며, 여러 가지의 Statement들을 가질 수 있지만 본 논문에서는 총 7개의 Statement에 대해 설명한다. VariableStatement는 로컬 변수에 대한 내용이며, Expression Statement는 함수 호출 등의 호출 정보를 가지고 있다. 나머지 요소들은 이름처럼 각각 for문, if문, while문, switch문, try문에 대한 내용을 가지고 있다. 또한 이 요소들은 각각 Body를 가지고 있어 여러 개의 Statement를 가질 수 있다. 이처럼 Java 코드의 AST 구조를 root(TypeDeclaration)부터 DFS(Depth First Search)로 탐색함으로써 배드 스멜 자동 가시화를 위한 데이터베이스를 생성할 수 있다.

4. 배드 스멜 코드 추출을 위한 코드 가시화

4.1 배드 스멜 추출 쿼리 프로그램

마틴 파울러는 리팩토링이 필요할 것으로 보이는 징후들을 배드 스멜로 정의하였다. 리팩토링 수행 시작과 끝을 결정하는 것은 어떤 절차에 따라 수행해야 하는지가 중요하다. Table 1은 배드 스멜 추출에 사용되는 데이터베이스의 컬럼 값이다.

Table 1. Meaning of Database Column Values

Column	Meaning	Column	Meaning
ACCESSOR	accessor	PARAMETER_NAME	name of parameter
CLASS_NAME	name of class	PARAMETER_TYPE	type of parameter
END_LINE	end line number	REFER_CLASS	referencing class
EXTEND	extend Class	REFER_MEMBER	referencing member
FILE_PATH	file path	REFERRED_CLASS	referenced class
FINAL	whether final or not	REFERRED_MEMBER	referenced member
IF_COUNT	number of if statements	RETURN_TYPE	type of return value
IMPLEMENT	implement Class	RETURN_VALUE	return value
INITIALIZATION	initial value	START_LINE	start line number
LINE_NO	line number	STATIC	whethe static or not
MEMBER_TYPE	type of member	SWITCH_COUNT	number of switch statements
METHOD_NAME	name of method	TYPE	type
NAME	name	WHILE_COUNT	Number of while statements

Table 1의 총 26개 컬럼들을 이용하여 배드 스멜을 추출한다. 하지만 마틴 파울러는 경험적 기준으로 리팩토링 수행 시기를 추정했다. 경험적 기준은 추상적이기 때문에 정적 분석된 코드 정보의 배드 스멜 패턴을 찾기는 쉽지 않다. 이를 해결하기 위해, 본 논문에서는 각 배드 스멜에서 사용되는 리팩토링 패턴을 코드 패턴으로 정의하였다. Table 2는 본 논문에서 확인 가능한 배드 스멜 항목들이다. 범위는 메서드에 해당하는 6개의 배드 스멜과 클래스에 해당하는 7개의 배드 스멜로 분류된다.

Table 2. Items Extracted by Bad Smell

Range	item	definition
Method	Switch Statements	Too many Switch statements
	Long Method	Too long method line
	Long Parameter List	Too many parameters
	Feature Envy	Too many using attributes of another class
	Message Chains	Too complex call link of the method
	Middle Man	Delegation of too many classes
Class	Data Clump	Parameter groups of the same structure
	Shotgun Surgery	Several other functions must be modified to modify a target function.
	Comments	Too many comments.
	Lazy Class	A class does nothing.
	Data Class	A class contains only getters and setters.
	Large Class	Too many instance variables.
	Refused Bequest	Not utilizing all inherited objects.

다음은 13개의 배드 스멜들을 이용하여 어떤 방식으로 추출하는지 각 항목별로 설명한다.

- Middle Man은 어떤 메서드가 다른 클래스에게 너무 많은 위임을 하는 경우이다. 메서드의 길이가 5줄 이하이고, 리턴 값이 다른 클래스에서도 호출하는 경우를 Middle Man으로 정의한다.
- Long Method는 메서드의 길이가 너무 긴 경우이다. 시작 길이와 끝나는 길이를 측정하여 일정 길이 이상의 메서드를 추출하는 것으로 정의한다.
- Long Parameter List는 파라미터의 개수가 너무 많은 경우이다. 메서드의 파라미터 개수를 측정한 후 일정 개수 이상일 경우 Long Parameter List를 추출한다.
- Data Clump는 같은 구조의 파라미터 그룹들이 있는 경우이다. Data Clump는 메서드의 파라미터 리스트를 추출한 후 일정한 패턴의 파라미터 그룹을 추출한다.
- Message Chains는 메서드의 호출 연결고리가 너무 복잡한 경우이다. 메서드를 통해 어떤 값을 얻고자 할 때 연속적으로 거쳐 가는 클래스의 개수가 일정 개수 이상일 경우 추출한다.
- Shotgun Surgery는 클래스가 하는 일이 다른 클래스에게 너무 간섭을 받는 경우이다. 즉, 어떤 메서드가 호출당하는 개수보다 다른 메서드를 호출하는 개수가 더 많은 경우 Shoygun Surgety를 추출한다.
- Feature Envy는 다른 클래스의 속성을 너무 많이 사용하는 경우이다. 즉, 다른 클래스의 get/set메서드를 일정 개수 이상 호출하는 경우에 추출한다.
- Switch Statements는 스위치문의 개수가 많은 경우 다

Table 3. Query Program Table to extract Bad Smell

Middle Man

```
ResultSet mm = statement.excuteQuery(
        "select count(RETURN_TYPE) from JPDB_METHOD where
                RETURN_TYPE in (select REFERRED_CLASS
                from JPDB_REFERBY where REFER_CLASS in
            (select [name] from JPDB_CLASS) and LINE_NO < 5)");
while (mm.next()) {
   return mm.getString("CNT");
}
```

Long Method

```
ResultSet lm = statement.excuteQuery(
            "select START_LINE, END_LINE from JPDB_METHOD
                                where NAME = [name]");
while (lm.next()) {
   int max = max(ss.getString("END_LINE");
   int min = min(ss.getString("START_LINE");
   int line = max - min;
   return line;
}
```

Long Parameter List

```
ResultSet lpl = statement.excuteQuery(
                "select PARAMETER_TYPE from JPDB_METHOD
                                where NAME = [name]");
while (lpl.next()) {
   String[] argument = lpl.getString("PARAMETER_TYPE");
   return argument.length;
}
```

Data Clump

```
ResultSet dcp = statement.excuteQuery(
                "select PARAMETER_TYPE, PARAMETER_NAME
                from JPDB_METHOD where NAME = [name]");
while (dcp.next()) {
   dataClump.put(dcp.getString("PARAMETER_TYPE +
                                PARAMETER_NAME"));
   if(dataClump.count() > count) {
      return count;
   }
}
```

Message Chains

```
ResultSet mc = statement.excuteQuery(
                "select REFERRED_MEMBER, REFERRED_CLASS
                                from JPDB_REFERBY
            where REFERRED_MEMBER LIKE 'get%' and NAME in
                            (select [name] from JPDB_CLASS)");
while (mc.next()) {
   String rdsName = mc.getString("REFERRED_MEMBER");
   String rdcName = mc.getString("REFERRED_CLASS");
   count = re_getMC(rdsName, rdcName, count);
}
```

Shotgun Surgery

```
ResultSet sgs = statement.excuteQuery(
                                "select count(REFER_CLASS) as refer,
          count(REFERRED_CLASS) as referred from JPDB_REFERBY
                                  where REFERRED_NAME = [name]");
while (sgs.next()) {
    int refer = sgs.getString("refer");
    int referred = sgs.getString("referred");
    int shotgunSurgery = refer - referred;
    if (shotgunSurgery > count) {
        return count;
    }
```

Feature Envy

```
    ResultSet fe = statement.excuteQuery(
              "select count(*) as CNT from JPDB_REFERBY where
    REFERRED_MEMBER LIKE 'get%' and REFERRED_CLASS in
    (select [name] from JPDB_CLASS)");
    while (fe.next()) {
        int cnt = Integer.parseInt(fe.getString("CNT"));
        return cnt;
    }
```

Switch Statements

```
    ResultSet ss = statement.excuteQuery(
    "select SWITCH_COUNT from JPDB_METHOD
    where NAME = [name]");
    while (ss.next()) {
        count++;
    }
    return count;
```

Comments

```
ResultSet cmt = statement.excuteQuery(
        "select COMMENT from JPDB_CLASS where NAME = [name]");
while (cmt.next()) {
    int comment = cmt.getString("COMMENT");
    return comment;
}
```

Lazy Class

```
ResultSet lzc = statement.excuteQuery(
                        "select count(*) as CNT  from JPDB_REFERBY
                                  where REFERRED_CLASS = [name]");
while (lzc.next()) {
    int count = lzc.getString("CNT");
    return count;
}
```

Data Class

```
ResultSet dc = statement.excuteQuery(
              "select NAME from JPDB_METHOD where CLASS_NAME
                        LIKE = 'get%' or CLASS_NAME LIKE = 'set%'
                                                and NAME = [name]");
while (dc.next()) {
    int count = dc.getString("CNT");
    return count; }
```

Large Class

```
ResultSet lgc = statement.excuteQuery(
                    "select count(*) as CNT from JPDB_INSTANCE_VAR
          where CLASS_NAME in (select [name] from JPDB_CLASS)");
while (lgc.next()) {
    int count = Integer.parseInt("CNT");
    return count;
}
```

Refused Bequest

```
ResultSet rb = statement.excuteQuery(
        "select count(*) from JPDB_REFERBY where REFERRED_CLASS
                                    and REFERRED_CLASS = [name]
         select count(*) from JPDB_METHOD where CLASS_NAME in
        (select EXTEND from JPDB_CLASS where NAME = [name])");
while (rb.next()) {
    int count = rb.getString("CNT");
    return count;
}
```

형성으로 해결할 수 있으므로 스위치 문을 추출한다.

- Comments는 주석의 길이가 너무 긴 경우이다.
- Lazy Class는 클래스가 하는 일이 없는 경우이다. Lazy Class는 어떠한 메서드에서도 객체 생성이 되지 않고 메서드 호출을 한 번도 당하지 않는 경우이다.
- Data Class는 클래스의 메서드가 get/set메서드로만 이루어진 경우 추출한다.
- Large Class는 클래스의 인스턴스 변수가 너무 많은 경우이다. 클래스의 인스턴스 변수가 일정 개수 이상일 경우 추출한다.
- Refused Bequest는 상속받은 것들을 모두 사용하지 않는 경우이다. 상속받은 메서드 중에 사용되지 않는 메서드가 하나라도 있다면 Refused Bequest를 추출한다.

Table 2에서처럼, 배드 스멜 추출 방법을 정의한 다음 배드 스멜을 추출할 수 있는 쿼리문을 작성해야 한다. 이 쿼리문을 통해서 배드 스멜의 가시화 그래프를 추출할 수 있다. Table 3은 배드 스멜을 추출하는 쿼리 프로그램의 예이다.

4.2 배드 스멜 패턴 가시화와 리팩토링을 통한 개선 결과

배드 스맬 패턴 가시화를 위한 샘플코드를 적용하였다. 이 코드는 학생의 전화번호를 출력해주는 프로그램으로, [15]에 있는 예제코드이다. 이 프로그램에서 학생의 전화번호 데이터를 획득하려면 데이터베이스에 접근하여 승인이 되어야 한다. 이 경우에, 프로그램은 학생의 전화번호를 사용자에게 출력해준다.

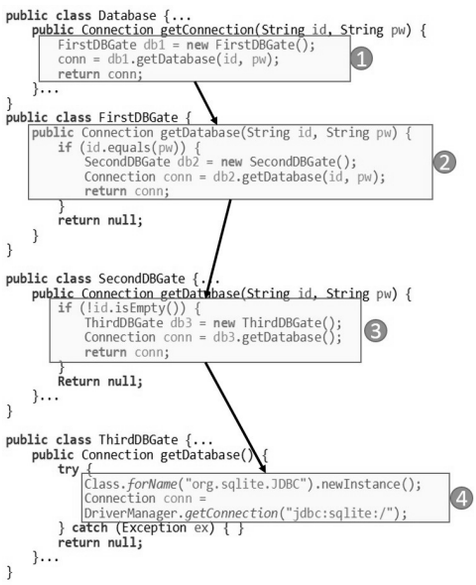

Fig. 3. Database Connection Part of Sample Code

Fig. 3은 예제 프로그램 중에 데이터베이스 승인을 얻어 접속정보를 얻어오는 부분이다. 코드를 보면 Database 클래스에서 Connection을 얻기 위해 ① FirstDBGate 클래스의 getDatabase() 메서드를 호출하고 있다. FirstDBGate의 getDatabase()메서

```
public class Phone {
   private String unformattedNumber;
   public Phone(Connection conn) throws SQLException {
      Statement stat = conn.createStatement();
      ResultSet rs = stat.executeQuery("");
      while(rs.next()) {
         String number = rs.getString("");
         unformattedNumber = number;
      }
   }
   public String getAreaCode() {
      return unformattedNumber.substring(0, 3);
   }
   public String getPrefix() {
      return unformattedNumber.substring(3, 6);
   }
   public String getNumber() {
      return unformattedNumber.substring(6, 10);
   }
}
public class Student {
   public String getMobilePhoneNumber(Phone mobilePhone) {
      String phoneNumber = "(" + mobilePhone.getAreaCode() +
        ") " + mobilePhone.getPrefix() +
        "-" + mobilePhone.getNumber();
      return phoneNumber;
   }
}
```

Fig. 4. The Part of the Sample Code that Receives the Student's Phone Number

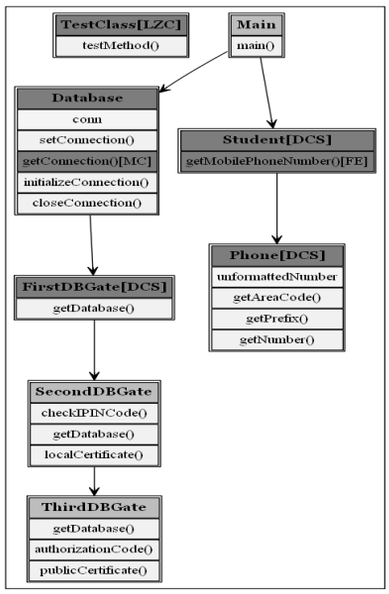

Fig. 5. Sample Code Before Refactoring

드에서 parameter값으로 id와 pw 정보를 받는다. ② id와 pw가 같을 경우 SecondDBGate 클래스의 getDatabase()메서드를 호출한다. ③ id가 빈값이 아닐 경우 SecondDBGate의 getDatabase()에는 ThirdDBGate의 getDatabase()를 호출한다. ④ ThirdDBGate의 getDatabase의 메서드에서 데이터베이스 연결정보를 얻는 것을 확인할 수 있다. 이 과정에서 Database 클래스는 입력 받은 id와 pw를 이용하여 연결정보를 생성할 수도 있었지만 불필요하게 여러 클래스의 get메서드를 거쳐 연결 정보를 받아오고 있다. 이런 경우 배드 스멜 중 Message Chains에 해당된다.

Fig. 4는 샘플 코드 중 핸드폰 객체를 통해 학생의 전화번호를 가져오는 부분이다. 코드를 보면 Phone 클래스는 생성자로 데이터베이스에 연결하여 정형화되지 않은 학생의 전화번호 정보를 가지고 있다. 이 Phone 객체를 받은 Student 클래스의 getMobilePhoneNumber() 메서드는 Phone 클래스의 get-메서드들을 호출하여 phoneNumber 변수로 형식을 맞추어 반환해준다. 하지만 Phone 클래스에서 처음부터 형식을 정형화하여 반환해주는 메서드가 존재하였다면 Student 클래스에서 Phone 클래스의 get-메서드들을 여러 번 호출할 필요가 없었을 것이다. 이런 경우 배드 스멜 중 Feature Envy에 해당된다.

Fig. 5는 전체 샘플 코드를 그래프로 가시화한 결과이다. 가시화 결과를 보면 위에서 언급하였던 Message Chains와 Feature Envy가 표시되었음을 확인할 수 있다. 또한 추가적으로 Data Class와 Lazy Class가 표시되었다. Table 2에서 언급하였듯이 get-메서드로만 이루어져 있는 FirstDBGate, Student, Phone클래스에 [DCS]가 표시되어 있다. Main 클래스에서 호출하고 있지 않는 TestClass가 [LZC]가 표시되어 있다. DCS는 Data Class이고, LZC는 Lazy Class이다. 이를 모두 리팩토링을 통하여 개선한다.

Fig. 6은 리팩토링을 통한 개선 과정 중 Database 클래스의 예이다. Fig. 6을 보면 Message Chains를 없애기 위해 데이터베이스 연결 정보를 모두 Database 클래스에서 해결하도록 리팩토링 하였다. 원래 코드에서는 데이터베이스의 연결정보를 얻기 위하여 FirstDBGate, SecondDBGate, ThirdDBGate를 통하여 가능 여부 승인 후 연결이 완료되었다. 하지만 이 모든 과정을 Database 클래스 한 군데에서 해결한다면 불필요한 여러 객체를 통한 접근을 방지할 수 있다. 결과적으로 데이터베이스에 접근하기 위해서는 앞으로 getConnection 메서드만 호출한다면 완료할 수 있다.

```
public Connection getConnection(String id, String pw) {
    FirstDBGate db1 = new FirstDBGate();
    conn = db1.getDatabase(id, pw);
    if (id.equals(pw)) {
        if (!id.isEmpty()) {
            try {
                Class.forName("org.sqlite.JDBC").newInstance();
                Connection conn = DriverManager.getConnection("jdbc:sqlite:/");
            } catch (Exception ex) { }
        }
    }
    return conn;
}
```

Fig. 6. Database Class with getConnection() Refactored

Fig. 7은 Feature Envy를 없애기 위해 Phone 클래스를 리팩토링한 코드이다. 본래 Student 클래스에서 Phone 클래스 객체를 호출하여 get-메서드를 여러 개 호출하는 Feature Envy가 발생하였다. 하지만 리팩토링을 통하여 Phone 클래스 안에서 해결하여 불필요한 객체 생성을 막을 수 있도록 toFormattedString() 메서드를 생성하였다. 이를 통하여 Student 클래스에서는 기존의 3번 호출을 수행하던 것을 toFormattedString() 호출 한 번으로 개선할 수 있었다. 마지막으로 이 프로그램에서 사용되지 않는 TestClass, FirstDBGate, SecondDBGate, ThirdDBGate를 삭제하였다.

```
public class Phone {...
	public String getAreaCode() {
		return unformattedNumber.substring(0, 3);
	}
	public String getPrefix() {
		return unformattedNumber.substring(3, 6);
	}
	public String getNumber() {
		return unformattedNumber.substring(6, 10);
	}
	public String toFormatedString() {
		String phoneNumber = "(" + getAreaCode() +
				") " + getPrefix() +
				"-" + getNumber();
		return phoneNumber;
	}
}
```

Fig. 7. Phone Class with toFormattedString () Added Due to Refactoring

리팩토링을 통하여 Message Chains와 Feature Envy, Lazy Class, Data Class등을 개선하였다. Fig. 8은 리팩토링 수행 결과이다. Fig. 6으로 인하여 Database 클래스의 Message Chains가 사라졌고 Fig. 7로 인하여 Student 클래스의 Feature Envy, Phone 클래스의 Data Class, Lazy Class의 사용되지 않는 클래스들이 사라진 것을 확인할 수 있다. 이처럼 배드 스멜이 발견된 클래스나 메서드들을 빨간색(①)으로 표시하고, 발견되지 않을 경우를 하늘색(②)으로 표시하여 소프트웨어 안의 배드 스멜 패턴들을 쉽게 확인할 수 있다.

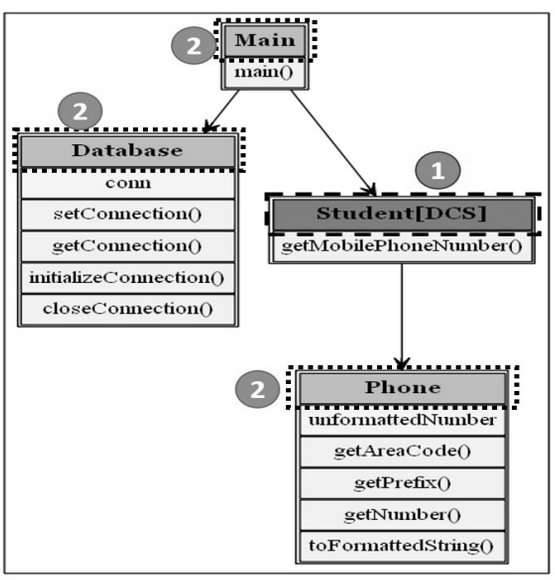

Fig. 8. Improved Code After Refactoring

Table 4는 리팩토링 수행 전후를 비교한 결과이다. 개선 전 배드 스멜 패턴 추출 결과는 총 6개의 코드, 4개의 배드 스멜 패턴을 추출하였다. 이 중 FirstDBGate, Phone, Student 클래스는 Data Class Pattern이다. 리팩토링을 수행한 결과, 총 6개의 배드 스멜 패턴 중 5개 패턴을 개선할 수 있었다.

Table 4. Improvement Result

Before Refactoring		After Refactoring	
Bad Smell Pattern	Target Code	Bad Smell Pattern	Target Code
LZC(Lazy Class)	TestClass	DCS (Data Class)	Student
MC(Message Chains)	getConnection()		
DCS(Data Class)	FirstDBGate		
	Phone		
	Student		
FE(Feature Envy)	getMobilePhoneNumber()		

5. 결론 및 향후 연구

본 논문은 소프트웨어의 품질 개선을 위해 리팩토링을 통한 배드 스멜 추출 방법을 제안하였다. 기존의 소프트웨어는 소프트웨어의 비가시성 때문에 소프트웨어 구조 분석이 어렵다. 제안한 방법은 코드 가시화를 통해 리팩토링이 필요한 부분을 식별할 수 있다. 이를 통해 개발자들이 어느 부분에 배드 스멜이 있는지 확인할 수 있다. 즉, 개발자들이 직접 리팩토링을 수행함으로써 소프트웨어 품질 개선에 도움을 줄 수 있다.

향후 연구로는 배드 스멜을 추출하는 것만 아니라, 다양한 문제 코드들을 추출되는 사례별로 확장하는 솔루션을 제공할 것이다. 또한 아직 남아있는 저전력 배드 스멜을 모두 추출하여 리팩토링하기 위한 품질 지표를 만들 것이다.

References

[1] Geon-hee Kang, HyunSeung Son, Youngsoo Kim, Young B. Park, and R. Youngchul Kim, “Improving Static Code Complexity with Refactoring Technique Based on SW Visualization,” *Fall Conference of the KIPS*, Vol.21, No.2, pp.646-649, 2014.

[2] Haeun Kwon, Bokyung Park, Keunsang Yi, Young B. Park, Youngsoo Kim, and R. Youngchul Kim, “Applying Reverse Engineering Through Extracting Models from Code Visualization,” *Fall Conference of the KIPS*, Vol.21, No.2, pp.650-653, 2014.

[3] Jihoon Park, Haeun Kwon, Geon-hee Kang, Keunsang Yi, and R. Youngchul Kim, “A Study on Improving Bad Smell through Code Visualization,” *IIBC Domestic Conference*, No.13, No.1, pp.47-48, 2015.

[4] Jihoon Park and R. Youngchul Kim, “A Study on Software Visualization for Bad Smell Coding Improvement,” *The KIPS Fall Conference*, Vol.23, No.2, pp.497-500, 2016.

[5] Martin Fowler, “Refactoring: Improving The Design of

Existing Code," Addison-Wesley, 2002.
[6] Jihoon Park, Woo sung Jang, Jin Hyub Lee, HyunSeung Son, and R. Youngchul Kim, "Implementing the Automatic Identification of the Bad Smell Coding Patterns," *KIISE Winter Conference*, pp.401-404, 2016.
[7] Jihoon Park, HyunSeung Son, Bokyung Park, Jin Hyub Lee, and R. Youngchul Kim, "Tailoring an Automatic Priority Based on Quality Metrics of an Organization for Effective Refactoring," *KCSE 2017*, Vol.19, No.1, pp.175-178, 2017.
[8] Jihoon Park, Bokyung Park, Keunsang Yi, and R. Youngchul Kim, "Implementing A Code Static Analysis Based on the Java Parser," *The KIPS Spring Conference*, Vol.24, No.1, pp.641-644, 2017.
[9] Jihoon Park and R. Youngchul Kim, "Implementing a Visualization Tool for refactoring Procedural Style within Object-Oriented Code," *ICT Platform*, Vol.5, No.3, pp.28-32, 2017.
[10] NIPA SW Engineering Center, "SW Development Quality Management Manual(SW Visualization)," 2013.
[11] Jihoon Park, HyunSeung Son, and R. Youngchul Kim, "A Study on the Test Case Traceability based on Abstract Test Case Maturity Model," *KIISE*, pp.563-565, 2017.
[12] Mauricio A. Saca, "Refactoring Improving The Design Of Existing Code," *IEEE 37th Central America and Panama Convention (CONCAPAN XXXVII)*, pp.1-3, 2017.
[13] M. Kaya, S. Conley, and Z. S. Othman, "Effective Software Refactoring Process," *6th International Symposium on Digital Forensic and Security (ISDFS)*, pp.205-210, 2018.
[14] JavaParser, Getting started, [Online], http://javaparser.org/getting-started/
[15] JaeHong Lee, "Go for the Really Impatient," Gilbut, 2015.

박 지 훈
https://orcid.org/0000-0002-1053-0350
e-mail : jh91082@tta.or.kr
2010년 ~ 2016년 홍익대학교 컴퓨터정보통신공학과(학사)
2017년 ~ 2019년 홍익대학교 소프트웨어공학전공 석사과정
2019년 ~ 현 재 한국정보통신기술협회 전임연구원
관심분야 : 역공학, Go language, 정적분석, 블록체인 가시화

박 보 경
https://orcid.org/0000-0002-7007-852X
e-mail : park@selab.hongik.ac.kr
2002년 ~ 2008년 홍익대학교 컴퓨터정보통신공학과(학사)
2010년 ~ 2012년 홍익대학교 소프트웨어전공(석사)
2013년 ~ 현 재 홍익대학교 소프트웨어공학전공 박사과정
관심분야 : 요구공학, 역공학, 테스트 성숙도 모델, 블록체인 가시화 프로세스, 소프트웨어 품질

김 기 두
https://orcid.org/0000-0002-6723-5950
e-mail : kdkim@tta.or.kr
2013년 홍익대학교 컴퓨터정보통신(학사)
2005년 홍익대학교 소프트웨어공학(석사)
2014년 홍익대학교 소프트웨어공학(박사)
2005년 ~ 현 재 한국정보통신기술협회 책임연구원
관심분야 : 테스트 성숙도 모델, 테스트 프로세스, SW 신뢰성 테스트

김 영 철
https://orcid.org/0000-0002-2147-5713
e-mail : bob@hongik.ac.kr
2000년 Illinois Institute of Technology(IIT)(공학박사)
2000년~2001년 LG산전 중앙연구소 Embedded System 부장
2001년 ~ 현 재 홍익대학교 소프트웨어융합학과 교수
관심분야 : 역공학, 모델 기반 테스팅, 정적분석, 블록체인 가시화, 테스트 성숙도 모델, 소프트웨어 품질, 소프트웨어 가시화