객체 추적성을 위한 Object Mapping Structure를 활용한 코드 가시화

정세준[1*], 박보경[2], 장우성[3], 서채연[4], 김영철[5]
홍익대학교 소프트웨어융합학과, 유원대학교
{jung[1*], jang[3], chaeyun[4], bob[5]}@selab.hongik.ac.kr, parkse[2]@u1.ac.kr

Code Visualization with Object-Oriented Mapping Structure for Object Traceability

Jung Sejun, BoKyung Park, WooSung Jang, Ch.Y. Seo, R. Young Chul Kim
SE Lab, Dept. of Software and Comm. Engineering, Hongik University
Dept of Smart IT, U1 University

요 약

소프트웨어의 중요성이 높아지는 상황에서 고품질의 소프트웨어개발을 위해 역공학과 가시화 기술이 사용될 수 있다. 객체지향 코드를 역공학 하여 코드의 복잡도와 설계도를 추출하는 Tool-Chain 기술이 있다. 본 연구는 기존의 Tool-Chain이 추적성 및 연결성을 검색하는데 발생하는 문제점을 개선한다. 이를 기반으로 1) 새로운 Tool-Chain을 제시하고 2) Object Mapping Structure가 작동하는 방식과 생성과정을 설명하며 저장된 정보를 기반으로 설계 다이어그램을 자동 생성한다.

1. 서 론

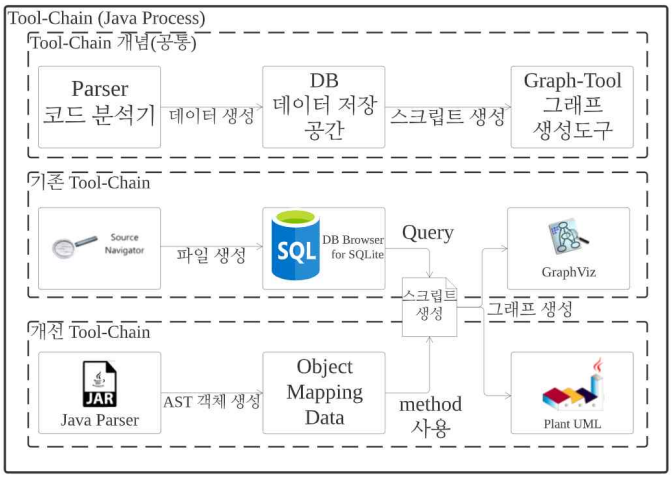
4차 산업혁명으로 다양한 분야에서 소프트웨어의 중요성과 제품들이 많아지고 있다. 하지만 소프트웨어의 고품질과 재사용성, 복잡도 등에 관한 이슈들은 소프트웨어의 성장 속도를 따라가지 못하고 있으며, 인공지능이나 오픈소스의 사용 또한 결과론적인 관점으로 사용되기도 한다. 소프트웨어는 개발자에 습관에 따라 형식이 다를 수 있고 다양한 코드에 같은 결과가 나오는 등 불확실성 문제가 큰 이슈이다. 따라서 고품질의 소프트웨어를 위해서 코딩 규칙을 정의하며 소스 코드를 분석하는 정적분석기가 필요하다. 이러한 문제를 해결하기 위해 고품질 소프트웨어의 품질을 높이기 위한 연구가 진행되고 있다[1][2][3]. 그 중, 소스 코드 가시화 기법으로 Tool-Chain[4][5]이 있다.

기존의 Tool-Chain은 Source-Navigator와 DB를 활용하여 정보를 저장하고 쿼리문을 통해 정보를 추출한다. 그러나 추적성 분석에서 쿼리를 반복적으로 사용하기 때문에 재사용성 떨어진다. 또한, 복합적인 정보를 가져올 때 쿼리문이 복잡해지므로 Tool-Chain의 품질이 낮아진다. 본 논문은 이러한 문제점을 개선하기 위해 데이터베이스를 대체할 Object Mapping Structure를 활용한 코드 가시화를 제안한다. 제안한 방법을 Tool-chain에 적용하여 소스 코드 가시화 및 분석을 진행한다. 제안한 방법을 통해 데이터의 중복성과 검색 복잡도를 줄이고 데이터 분석 속도 향상 기대한다.

2. 관련 연구

2.1 Tool-Chain 구성

소프트웨어 가시화를 위한 Tool-Chain 기술은 Parser(정적분석기) 와 DB, 결과물 생성을 위한 Graph-Tool로 구성된다. 그림 1은 소프트웨어 가시화 Tool-Chain 구성을 나타낸다.



(그림 1) Tool-Chain

기존의 Tool-Chain은 Parser로 Source-Navigator를 사용하고 DB로는 SQLite, Graph-Tool로 Graphviz를 사용한다. Tool-Chain을 통한 소스 코드 분석은 다음과 같다.

Step 1. Source-Navigator가 타겟 프로젝트를 분석한다. 분석결과로 소스 코드 구조정보가 저장된 파일이 생성된다.

Step 2. 추출된 파일들을 Source-Navigator의 dbdump를 이용하여 DB에 저장한다.

Step 3. DB에 저장된 정보를 쿼리문을 활용하여 필요한 정보를 추출한다.

Step 4. GraphViz로 UML Diagram 및 복잡도를 생성하

기 위해 추출한 정보로 스크립트를 작성하고 결과물을 생성한다.

2.2 개선한 Tool-chain

개선한 Tool-Chain은 Parser로 Java Parser, 정보저장을 위한 도구로 DB를 사용하지 않고 본 논문에서 새롭게 제안한 객체구조인 Object Mapping Data로 대체한다. 또한, Graph-Tool로는 GrapViz를 PlantUml로 대체한다. 개선된 Tool-Chain과정은 아래와 같다.

Step 1. 소스코드분석은 Java Parser을 사용하여 AST(Abstarct Syntax Tree) 구조인 ComplilationUnit 객체를 생성한다.

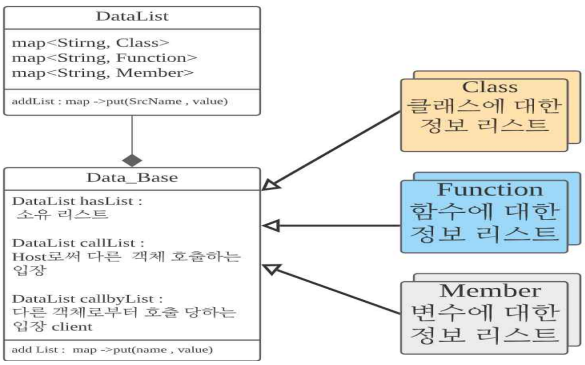
Step 2. 생성된 AST를 탐색하는 Visitor를 사용하여 정보를 DataList 객체에 저장한다.

Step 3. 생성된 정보는 DataList의 map 구조를 이용하여 중복을 방지하고 분석된 정보가 지속해서 갱신되며 연결한다.

Step 4. DataList 클래스의 메서드를 활용하여 plantUML 스크립트 생성하고 plantUML을 통해 그림으로 나타낸다.

3. Object Mapping Structure

Object Mapping Structure에서 정보는 객체 구조로 저장된다. 모든 정보는 has, call, callby 관계로 정의한다. has는 소유 관계로 클래스가 소유하는 메서드, 변수와 같은 관계이다. 메서드는 지역변수를 소유할 수 있으며 메서드라는 가상 함수를 소유할 수도 있다. Call은 호출 관계이다. 예를 들어 Barista의 makeCoffee 함수가 Coffee 객체를 생산한다. 이 경우 makeCoffee 함수는 Coffee 클래스를 호출한다. makeCoffee함수를 소유하는 Barista 클래스 또한 Coffee 클래스를 호출하는 구조를 가지게 된다. Callby관계는 call의 반대 개념이다. 따라서 Coffee 클래스는 Barista 클래스에 callby관계이다. 이렇게 객체 지향의 핵심 요소로 클래스, 함수, 맴버(변수)를 정의하였고 이를 바탕으로 정보들을 상속시켜 정의하였다. 클래스에 대한 정보는 Class, method에 대한 정보는 Function, 변수에 대한 정보는 Member에 저장된다. 그리고 모든 객체를 3가지의 관계로 Mapping하는 것이 핵심적인 구조이다.



(그림 2) Object Mapping Structure 구조

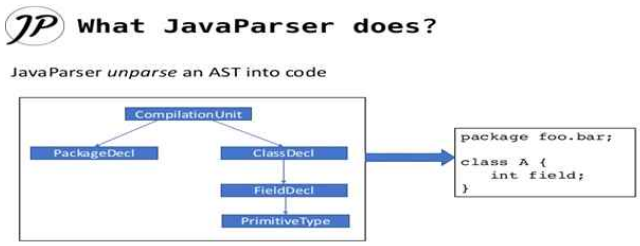
따라서 하나의 프로젝트에 대한(타겟) DataList에는 프로젝트에 존재하는 모든 Class, Function, Member가 Map 구조에 저장된다. 또한, DataList를 3가지의 관계에도 사용한다. 따라서 Class, Fumction, Memeber는 각각의 관계에 대한 datalist를 소유하고 있다. 소유 관계의 hasList, 호출관계의 callList, 호출당하는 관계의 callbyList에 객체들을 매핑시켜 연결 구조를 생성한다. 이러한 공통적인 관계 구조와 기본적인 정보로 객체를 추상 클래스로 선언했다. 따라서 DB를 대체하는 의미로 Data_Base 클래스를 정의하고 Class, Function, Member를 상속시켰다. 따라서 동일한 다형성을 통해 재사용성을 높이고 코드를 간결화하여 코드의 품질을 높일 수 있다.

3.1 Parsing

Tool-Chain의 Parser 역할인 기존 Source-Navigator를 대체하여 Java Parser을 사용한다. Java Parser은 타겟 프로젝트가 자바 언어로 제한되지만, Java로 짜인 소스 코드의 대부분 정보를 추출할 수 있는 오픈소스 도구이다.

Java Parser 객체가 추상화 구문트리(AST)를 생성하여 객체로 반환해준다. AST는 모든 정보가 트리구조로 저장되어 있다. 클래스 선언에서, 메소드, return 구문처럼 하단의 노드일수록 단순한 정보가 저장되고 상단 노드일수록 복합적인 정보가 저장된다. 생성된 AST는 특정 node를 root로 놓고 내부를 전부 탐색하는 Visitor 객체를 이용하여 탐색할 수 있다. 탐색 시에는 visit 함수를 사용한다. visi 함수는 특정 노드를 기점으로 오버 로딩하여 특정 문법에 필요한 작업을 진행할 수 있다.

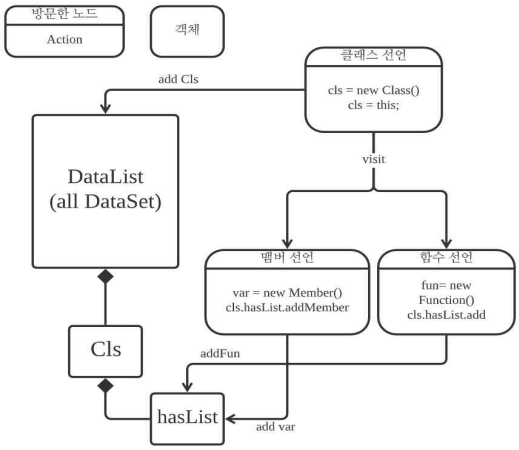
즉, visit 함수를 이용하여 각 노드에 맞게 정보를 추출하여 Object Mapping Structure에 정보를 추가하고 객체를 생성한다.



(그림 3) AST 구조

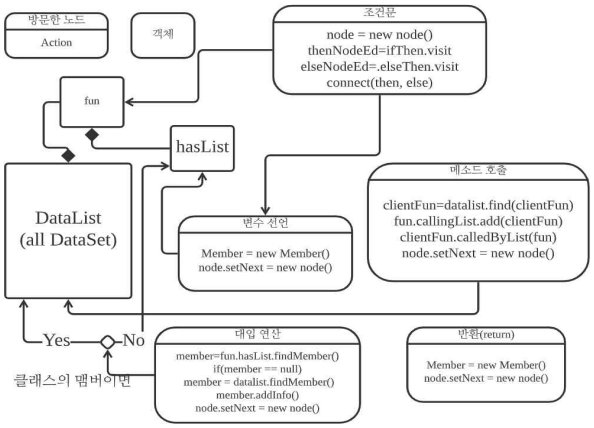
3.2 The Construction of Object Mapping Structure

AST를 순환할 visitor는 특정 구문의 visit함수에서 내부를 추가로 경유(visit)할지를 선택할 수 있다. 이를 활용하면 특정 목적에 따라 다른 Visitor를 작성하여 사용하는 것이 가능하다. 본 연구에서는 기본적인 정보와 has 관계를 먼저 전부 구성한다. (그림 4) 그 후, 메소드 내부를 탐색하여 call관계와 callby관계를 구성한다. (그림 5)



(그림 4) 작성된 visitor 알고리즘 1

그림 4는 설계한 첫 번째 visitor의 작동 방식을 나타낸다. AST의 클래스와 클래스 내의 필드(변수), 메서드(함수) 등을 모두 DataList의 Map을 통해 추가한다.



(그림 5) 작성된 visitor 알고리즘 2

그림 5는 메서드 내부의 특정 구문의 순서와 정보를 추가하기 위해 생성한 두 번째 Visitor를 나타낸다. 예를 들어, return x = a.fun(); 코드는 return, 대입 연산자, 메소드 호출 순으로 방문(visit)한다. 또한, 호출식의 경우 dataList의 Map을 통해 해당하는 객체를 가져와 정보를 추가한다. 변수들의 경우는 변수 선언 시 반드시 지역변수가 되므로 변수정보를 함수의 hasList에 추가한다. 변수가 사용될 때에는 먼저, hasList에서 검색을 하여 해당 변수의 정보를 업데이트 한다. hasList에 없다면 객체의 필드값이 사용된 것이다. 따라서 datalist 혹은 클래스 객체로부터 해당 정보를 가져온다.

그림6은 MethodCallExpr node에 대한 vistor 함수다. 그림5, 6의 알고리즘과 같이 수행하면 작성된 vistor들을 순환시켜 데이터를 생성한다. 이와 같이 사용자의 요구에 따라 visitor를 작성 및 변경하여 원하는 정보를 추출하고 타겟 정보들이 포함된 데이터객체를 생성할 수 있다.

```
public void visit(MethodCallExpr n, StringBuilder arg)
{
    FunctionInfo hostFun = datalist.getFunction(arg.toString());    //null일 수가 없음. 메소드 주인.

    ResolvedMethodDeclaration resolve = n.resolve();    //해당 노드의 resolve

    //호출하는 함수의 소유자의 클래스 정보 접근
    ResolvedClassDeclaration clientClassResolve = resolve.declaringType().asClass();
    String clientSrcClassName = clientClassResolve.getQualifiedName(); //호출자 Src 이름
    ClassInfo clientClass = datalist.getClass(clientSrcClassName);  //datalist에서 해당 클래스 찾기

    String clientFunSrc = resolve.getQualifiedSignature();  //호출하는 함수의 src
    FunctionInfo clientFun = datalist.getFunction(clientFunSrc); //호출하는 함수 datalist에서 가져오기
```

(그림 6) MethodCallExpr visit 함수 일부

3.3 PlantUML

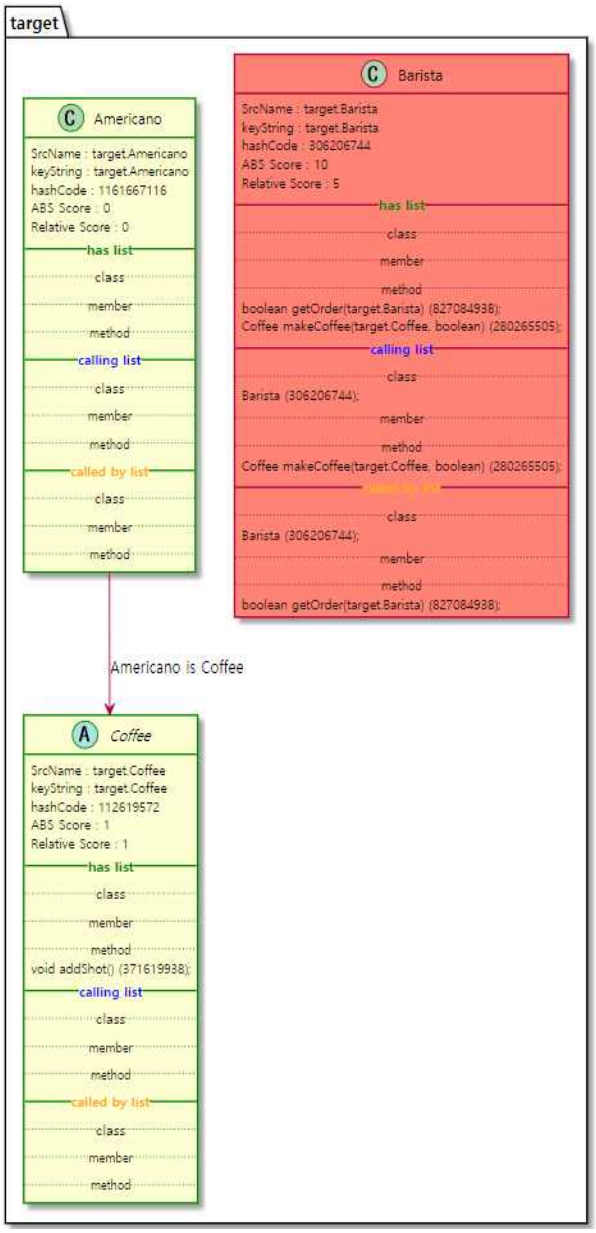
마지막으로 PlantUML을 사용하여 스크립트를 작성하여 결과물을 출력한다. 스크립트에 들어가는 내용은 이전까지의 단계로 생성된 DataList(Object Mapping structure)에서 가져온다. 따라서 해당 객체들의 메소드를 사용하여 목적에 따라 정보를 추출하고 문법에 맞게 문자열을 작성한다. 그리고 현재 저장된 Object Mapping Structure의 클래스들을 출력한다. 이러한 과정은 생성된 DataList의 구조를 확인하고 각 객체의 연결 관계를 체크하는 것도 가능하다. 아래는 파라미터로 받은 datalist에 연결된 객체들의 이름을 문법에 맞게 작성하는 함수의 예이다.

```
private static void addListInfo(Datalist datalist, StringBuilder str)
{
    //add classList
    str.append("..class..\n");

    Map<String, ClassInfo> map = datalist.getClassMap();
    for( ClassInfo target : map.values() )
    {
        str.append( target.getSrcName() +";\n" );
    }

    // add MemberList
    str.append("..member..\n");
    for (MemberInfo target : datalist.getMemberMap().values())
    {
        str.append("\t " + target.getName() + ";\n");
    }

    // add FunctionList
    str.append("..method..\n");
    for (FunctionInfo target : datalist.getFunctionMap().values())
    {
        str.append("\t" + target.getSig() + ";\n");
    }

}
```

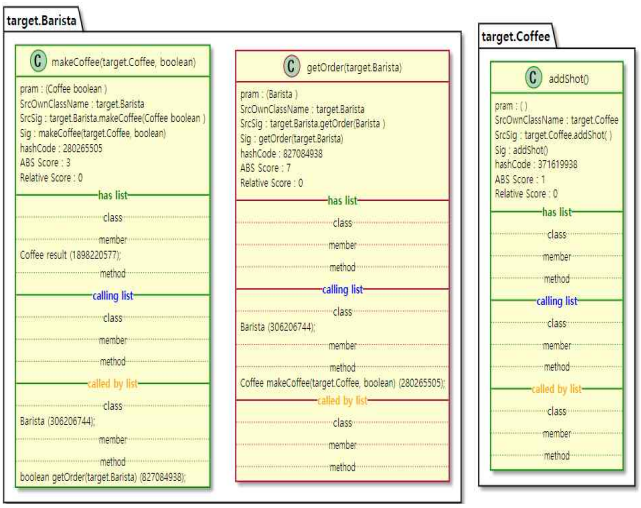
(그림 7) addListInfo 함수

파라미터로 받은 StringBuider str은 최종적으로 작성하는 스크립트 문이다. 따라서 해당 문자열을 최종적으로 완성한 내용을 파일로 작성하여 PlantUML로 실행시키면 그림이 완성된다.

3.4 결과



(그림 8) Class 객체들의 클래스 다이어그램



(그림 9) Function 객체들의 클래스 다이어그램

그림 8에서 각각의 클래스는 고유한 hashcode를 가진다. 또한, has, calling 관계에서 각각의 클래스나 메서드에 괄호로 hashcode를 확인(Barista(306206744))할 수 있다. 동일한 hashcode 값으로 데이터의 중복 없이 서로 참조 되고 있음을 확인할 수 있다. 또한, 클래스의 경우 상속에 관련된 정보와 abstract에 대한 여부가 위처럼 표시되고 있다. 그림 8을 보면 패키지 명과 클래스 명으로 구분되어 Function 객체들이 묶여 있으며 파라미터 정보, 소유 클래스 정보, signature 정보 등이 저장되어 있다. 마찬가지로 고유한 hashcode를 확인할 수 있으며 중복이 없는 것을 확인할 수 있다.

4. 결론

본 논문은 Object Mapping Structure를 활용한 코드 가시화 Tool-Chain을 제안한다. 제안하는 Tool-Chain으로 DB 사용 없이 기존의 Tool-Chain을 대체할 수 있다. 따라서 DB를 사용하지 않기 때문에 기존방법보다 재사용성을 높이고 복잡도를 줄여 Tool-Chain의 품질을 고도화한다.

향후, 프로젝트에서 DB의 사용한 방법과 제안한 모델링의 성능 비교연구를 융합한 연구를 할 예정이다. 또한, 객체구조를 활용한 다양한 데이터축적에 관한 연구와 더불어 소스 코드를 트리구조로 변환해서 여러 품질지표에 대한 가시화 연구를 할 예정이다. 연결성과 객체의 상태를 연구하여 Use-Case와 State-Diagram을 복원하는 역공학 관련 연구를 진행할 예정이다. 이를 기반으로 test-case를 추출하고 요구사항들의 연관성을 추적하는 연구를 할 예정이다.

Acknowledgment

이 논문은 2020년도 정부(교육부)의 재원으로 한국 연구재단의 지원을 받아 수행된 기초 연구 사업 (NRF-2020R1I1A1A01072928)임.

참 고 문 헌

[1] 문소영 “역공학 기반 코드 가시화를 통한 기능점수의 비용 추정 검증 방안”, 『제 20회 한국 소프트웨어공학 학술대회 논문집』 제20권 제1호, 2018. pp.182-183.

[2] 권하은 “역공학을 통한 소스 코드로부터 유스케이스 설계 추출” 『2016 춘계학술대회 논문집』 제5권 제1호, 2016. pp.289-291.

[3] 이원영 “역공학 기반 UML 설계 복원을 위한 소프트웨어 가시화”『2016 춘계학술대회 논문집』 제21권 제2호, 2020. pp.211-212.

[4] 강건희 “SW 가시화 기반 리펙토링 기법 적용을 통한 정적 코드 복잡도 개선”『2014년 추계학술발표대회』 제21권 제2호, 2014. pp.646-649.

[5] 강건희 “SW 가시화 기반 리펙토링 기법 적용을 통한 정적 코드 복잡도 개선”『제 18회 한국 소프트웨어공학 학술대회 논문집』 제18권 제1호, 2016. pp.395-398.