기존 자바 파서 확장 기반의 코드 정적 분석기 구현

박지훈*, 박보경, 이근상, 김영철
홍익대학교 소프트웨어공학연구실
e-mail: {pjh, park, yi, bob}@selab.hongik.ac.kr

Implementing A Code Static Analysis based on the Java Parser

Jihoon Park*, Bokyung Park, Keunsang Yi, R. Youngchul Kim
SE Lab, Hongik University

요 약

현재 많은 테스팅 기법으로도 생산되는 결과물들의 잠재적 오류 발생을 예측하기 힘들다. 기존 오픈 소스 정적 분석 도구들(Source Navigator)은 불충분한 정보를 제공하여 원하는 내부 정보를 축출하기 어렵다. 이를 해결 위해, 기존 오픈 소스의 자바 파서의 개선을 통해, 코드 내부 품질 측정을 고려하고자 한다. 즉 기존 자바 파서 개선 기반의 “추상구문트리로 변환된 코드”에서 “직접 코드 정보 추출” 방안의 구현이다. 이를 통해, 기존의 SNDB보다 더 많은 코드 정보 추출로 코드 내부 품질 측정이 더 수월할 것을 기대한다.

1. 서론

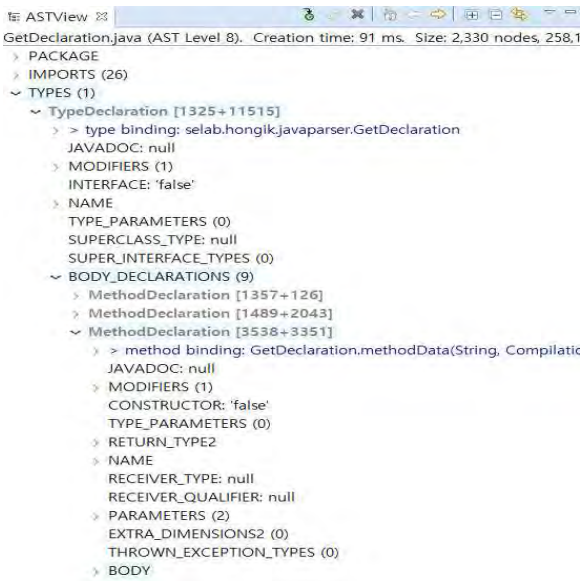
모든 분야의 공학에서는 각 분야별로 테스팅 기법을 가지고 있다. 제대로 테스팅 되지 않은 결과물들은 언제 어떤 문제가 발생할지 예측하기 힘들어진다. 2013년 12월, 네덜란드의 암스테르담 시는 1억 7천만 유로의 손해를 입었다. 문제는 180만 유로의 주택 보조금 지급하는 소프트웨어의 단위가 100배나 크게 설정된 사실을 파악 못했기 때문이다[4]. 소프트웨어를 사용 전에 테스트만 잘 수행해도 이런 큰 손해를 막을 수 있었다.

기존 테스팅은 크게 두 가지로 본다. 하나는 실행이 없이 테스트 방법인 ‘정적분석’이고 다른 하나는 실행하며 테스트 방법인 ‘동적분석’이다. 기존연구[1,2,3]에서는 정적분석도구중의 하나인 소스 내비게이터 (SNDB)를 이용하였다[6]. 하지만 소스 내비게이터가 코드에서 추출해낸 SNDB파일들의 정보로는 연구 목적인 Bad Smell 추출을 통한 코드의 품질 개선에 어려움이 있다. 예를 들면, 중복된 코드를 찾아낼 때 필요한 메서드안의 내용, 반복문, 조건문 등에 대한 정보가 부족하다. 본 논문은 기존연구의 정적분석 도구인 소스 내비게이터가 추출 못한 정보들을 자바 파서[7]로 추가적인 정보를 추출하였다. 기존 자바 파서가 Java 코드를 추상구문트리로 파싱한다. 파싱된 코드를 직접 트리를 순회하며 코드의 정보를 추출하도록 구현하였다. 자바 파서는 자동으로 정보를 추출하여 데이터베이스화 하지 못한다. 그래서 직접 추출된 정보를 데이터베이스로 만들었다. 기존의 SNDB 테이블을 자바 파서로 추출한 데이터 테이블로 대체하여 코드의 정보를 더 많이 파악할 수 있다.

본 논문의 순서는 다음과 같다. 1장에서는 서론, 2장은 AST(추상구문트리)에 대한 설명이다. 3장은 소스 내비게이터 DB와 자바 파서 DB의 차이점이다. 4장은 결론 및 향후 연구이다.

2. 추상화 구문 트리 (Abstract Syntax Tree)

추상 구문 트리는 프로그래밍 언어로 작성된 소스 코드의 추상 구문 구조를 표현한 것이다.

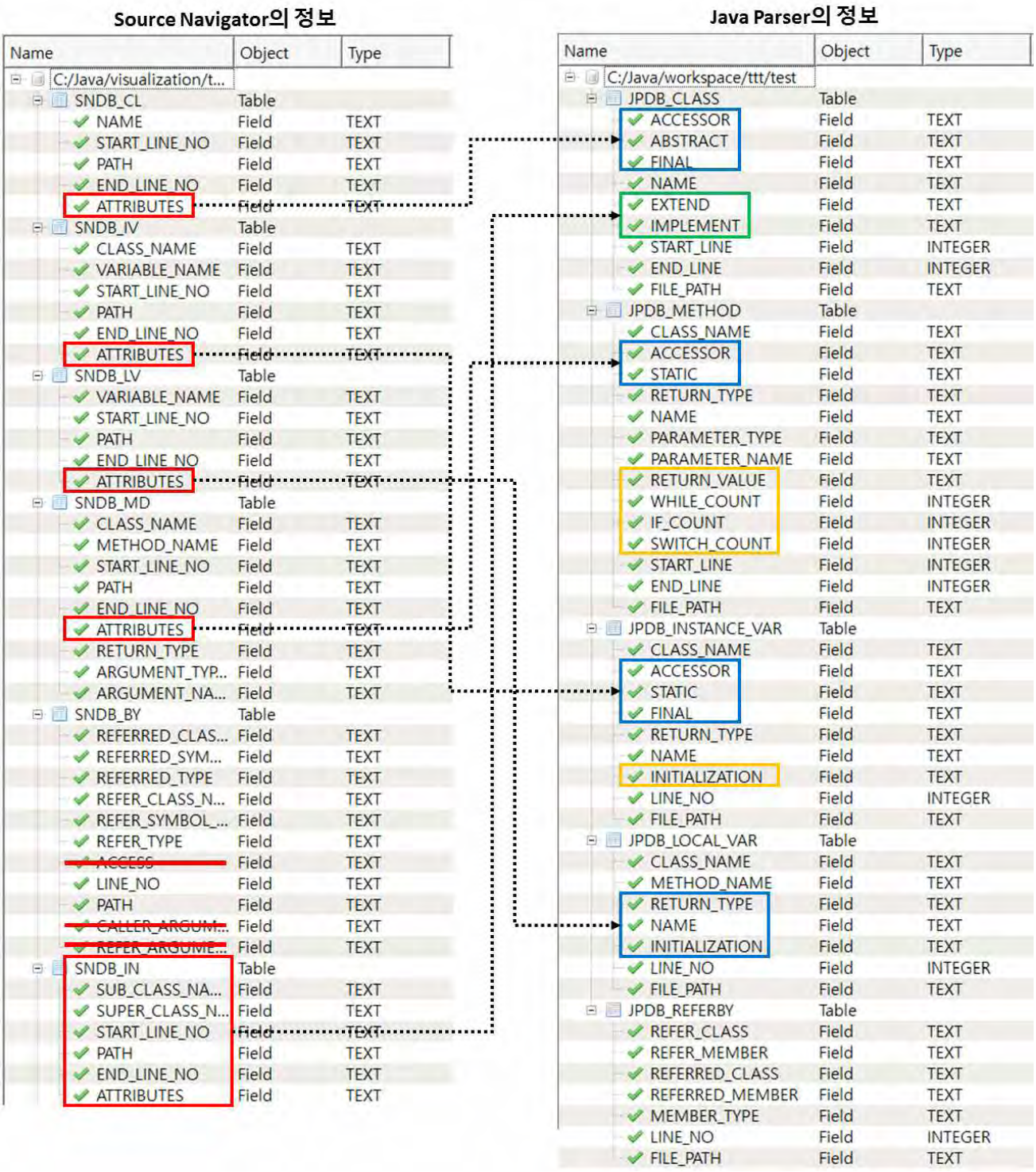


(그림 1) 추상구문트리로 변환된 Java 코드

그림 1은 ASTView[8]라는 이클립스의 플러그인으로 확인한 Java 코드의 추상구문트리이다. 자바 파서는 토큰 배열을 프로그램의 문법 구조를 반영하여 중첩 원소를 갖는 트리 형태로 바꾸는 과정이다[5]. 대부분의 경우 소스 코드는 특정 문법에 따라 쓰여진 하나의 문자열이다. 상대적으로 사람이 이해하기 쉽지만 기계에게는 다소 불편하다. 추상구문트리에서 대체적으로 괄호로 구분된다.

3. 기존연구와의 파싱 도구 비교

기존연구에서는 소스 코드를 가시화하는 자동화 도구에서 파싱부분을 오픈소스인 소스 내비게이터로 사용하였다. 그림 2는 소스 내비게이터가 자동으로 추출해주는 소스코드의 데이터베이스와 자바 파서로 소스코드의 정보를 직접 추출한 데이터베이스를 비교한 그림이다. 파란색은 기존의 값을 한 단계 더 해석한 것이고, 노란색은 새로 추가된 값, 초록색은 다른 테이블에서 옮겨온 값들이다. 소스 내비게이

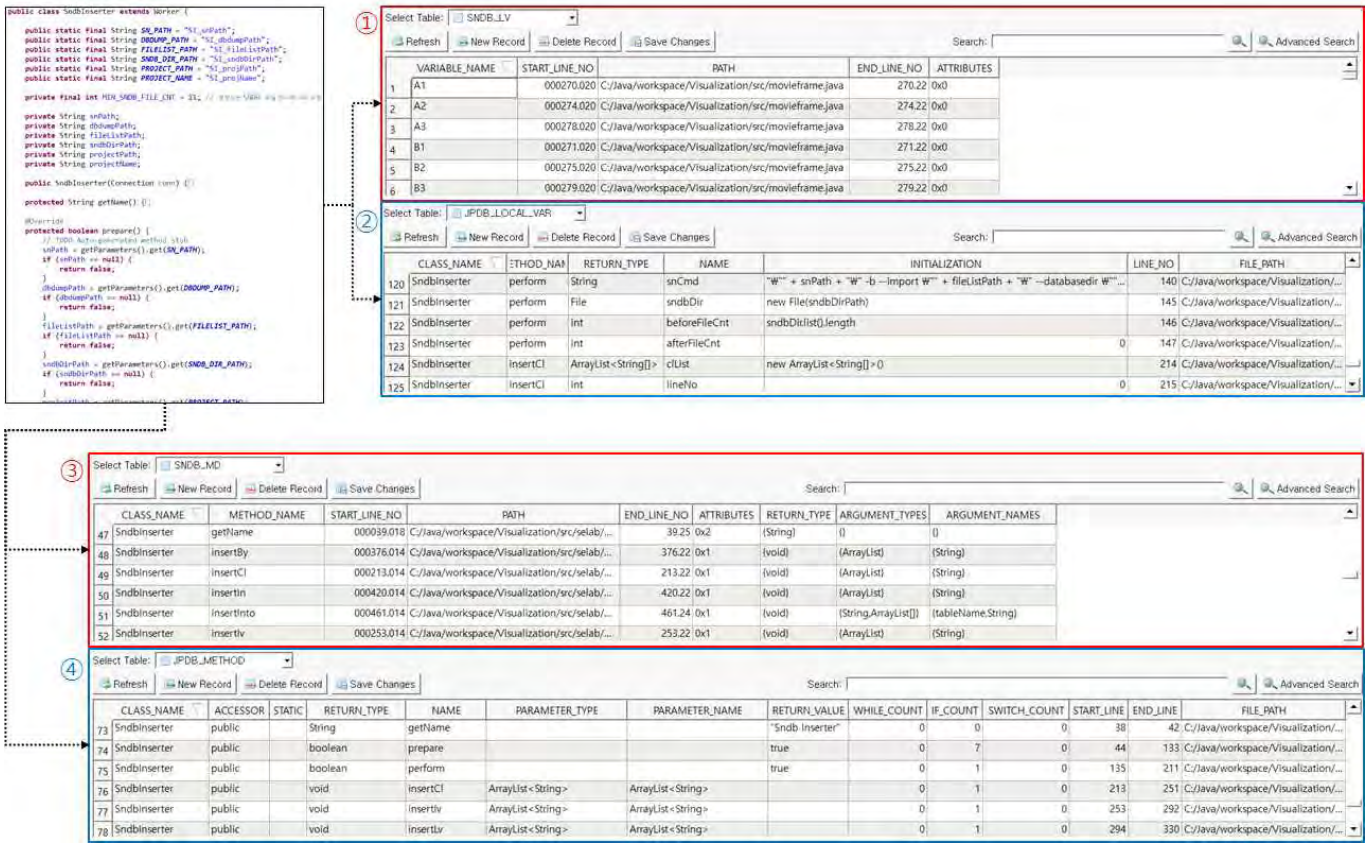


(그림 2) 소스 내비게이터와 자바 파서의 데이터베이스 비교

터에서 자동으로 생성된 SNDB의 소스코드 데이터는 그림 2의 왼쪽 데이터이다. 데이터를 보면 소스 내비게이터에서 파싱하여 추출해낸 코드의 정보는 총 6개의 테이블에 저장되어 있다. 그리고 새로 구성한 자바 파서를 이용하여 직접 추출한 소스코드 데이터는 그림 2의 오른쪽 데이터와 같다. 데이터를 보면 자바 파서로 파싱하여 얻은 AST데이터들은 5개의 테이블에 정제한 것을 볼 수 있다.

먼저 소스 내비게이터의 테이블 중 SNDB_CL 테이블에는 클래스의 이름, 시작 라인 넘버, 파일의 경로, 마지막 라인 넘버, 접근자의 정보가 들어있다. SNDB_IV 테이블에는 인스턴스 변수가 속해있는 클래스, 인스턴스 변수의 이름, 시작 라인 넘버, 파일의 경로, 마지막 라인 넘버, 접근자의 정보가 들어있다. SNDB_LV 테이블에는 로컬변수의 이름, 시작 라인 넘버, 파일의 경로, 마지막 라인 넘버, 접근자의 정보가 들어있다. SNDB_MD 테이블에는 메서드가 속해있는 클래스, 메서드의 이름, 시작 라인 넘버, 파일 경로, 마지막 라인 넘버, 접근자, 리턴 타입, 파라미터의 타입, 마라미터의 이름이 들어있다. SNDB_BY 테이블에는 호출당하는 메서드나 변수의 이름과 클래스, 타입, 호출하는 메서드의 이름과 클래스, 타입, 액세스 정보, 라인 넘버, 파일 경로, 호출당하는 메서드의 파리미터의 타입이 들어있다. SNDB_IN 테이블에는 서브클래스의 이름, 슈퍼클래스의 이름, 시작 라인 넘버, 파일 경로, 마지막 라인 넘버, 접근자의 정보가 들어있다.

그에 비해 자바 파서로 추출한 테이블 중 JPDB_CLASS 테이블에는 SNDB의 0x404처럼 ASCII 코가 아닌 스트링문자열로 public, protected, private, static, final을 나타내었다. 이는 아스키코드를 스트링 값으로 변환한 후 한 번 더 스플릿하여 접근자를 정제해야 했던 단계를 줄여준다. 그리고 SNDB_IN 테이블에 떨어져 있던 상속정보를 extend와 implement칼럼에 저장함으로써 데이터 정제가 더 간단해졌다. JPDB_METHOD 테이블에도 마찬가지로 접근자의 정보를 스트링 문자열로 가시화하였고 SNDB_MD 테이블에서는 메서드의 리턴 타입은 있었지만 메서드의 리턴 값은 나와 있지 않았다. 하지만 JPDB_METHOD 테이블에서는 메서드의 리턴 타입과 리턴 값까지 저장되어, 리턴 값에 있는 변수이름을 이용하여 어떠한 값이 리턴되는지도 확인할 수 있다. 추가적으로 메서드 안의 while, if, switch 문의 개수를 파악하여 이용할 수 있도록 하였다. JPDB_INSTANCE_VAR 테이블도 마찬가지로 접근자의 정보를 스트링 문자열로 가시화하였다. 그리고 SNDB_IV 테이블에서는 인스턴스 변수의 초기화 값들을 알 수 없었지만 JPDB_INSTANCE_VAR 테이블에서는 각 인스턴스 변수들의 초기화 값을 알 수 있다. JPDB_LOCAL_VAR 테이블은 로컬변수의 정보가 들어있다. 로컬변수는 접근자를 가지고 있지 않기 때문에 기존SNDB_LV의 Attribute값을 삭제하



(그림 3) 소스 내비게이터보다 더 많은 정보를 추출한 자바 파서

고 리턴 타입과 초기화 값을 추가하였다. JPDB_REFERBY 테이블에서는 각 멤버간의 호출정보를 나타낸다. 호출하는 클래스와 변수 또는 메서드의 이름, 호출당하는 클래스와 변수 또는 메서드의 이름, 호출하는 멤버의 타입, 라인 넘버, 파일 경로가 들어있다. 기존의 SNDB_BY 테이블의 Access값과 Argument값은 사용 안 되어 삭제하였다.

그림 3은 실제 Tool-Chain 코드를 소스 내비게이터와 자바 파서를 이용하여 데이터를 추출한 그림이다. 그림 3의 ①,③번은 SNDB, ②,④번은 JPDB이다. 둘 다 메서드안의 로컬 변수에 대한 테이블이다. ①번에서는 메서드의 이름이 나와있지 않고 접근자가 아스키코드로 표현되었다. 하지만 ②번에서는 로컬변수가 속해있는 메서드의 이름과 리턴 타입, 초기화 값이 표현되어있다. ①번은 로컬변수가 객체를 몇 번이나 생성하는지 알 수 없지만 ②번은 초기화 값을 이용하여 객체가 몇 번 생성되는지 찾아낼 수 있다. ③번에서는 메서드의 시작라인 넘버만 나와있지만 ④번에서는 메서드의 전체길이를 확인할 수 있다. ③번은 메서드가 void가 아닐 때 어떤 변수를 리턴하는지 확인할 수 없지만 ④번은 RETURN_VALUE값으로 다른 객체에 얼마나 많은 위임을 시도하는지 확인할 수 있다.

4. 결론 및 향후계획

정적 분석은 대상 소프트웨어를 실제적으로 실행하지 않는 상태에서 수행하는 것이다. 그래서 실질적인 오류를 발견하기엔 적합하지 않다. 하지만 보통 리뷰와 마찬가지로 장애(Failures)보다는 결함(Defects)를 발견하는데 중점을 둘 수 있다. 동적 테스팅에서 발견하기 어려운 문제점들을 찾아낼 수 있다. 또한 코드가 개발 중에 수정이 가능하다. 설계와 유지보수성을 향상시킬 수 있음으로써 소프트웨어의 품질 향상에 기여한다.

기존연구에서 사용하던 오픈소스인 정적분석도구 소스 내비게이터로 데이터 추출에 한계를 느꼈다. 그래서 본 논문에서는 소스 내비게이터 부분을 또 다른 오픈 소스인 자바 파서로 교체하였다. 하지만 자바 파서는 소스 내비게이터 와 같이 정적분석한 결과를 자동으로 데이터베이스화 해주지 않는다. 그래서 자바 파서 내의 자바 코딩 파싱 기능을 이용하여 직접 데이터베이스를 자동 생성할 수 있는 정적 분석 도구를 구현하였다. 즉 자바 파서는 소스코드의 모든 부분을 추상구문트리로 변환할 수 있다. 소스 내비게이터 가 찾아내지 못하는 소스코드의 더 깊은 정보를 추출할 수 있다. 그러므로 기존연구의 소스 내비게이터 기반 Tool-Chain[1,2,3]보다 더 다양한 소스 코드의 복잡도를 찾아낼 수 있다.

참고문헌

[1] 박지훈, 권하은, 강건희, 이근상, 김영철, “코드 가시화를 통한 나쁜 코딩 습관 개선 방안 연구”, 제 13권, 제 1호, 한국인터넷방송통신학회, 2015

[2] 박지훈, 김영철, “나쁜 코딩 습관 개선을 위한 코드 가시화 연구”, 제 23권, 제 2호, 한국정보처리학회, 2016

[3] 박지훈, 장우성, 이진협, 손현승, 김영철, “확장된 나쁜 코딩 습관 식별 구현”, 한국정보과학회, 2016

[4] “Software and staff to blame for Amsterdam’s €188m benefit error“, Dutch News, 14 Jan 2014.

[5] Kyle Simpson "You Don’t know JS SCOPE & CLOSURES" Hanbit’s Book

[6] http://sourcenav.sourceforge.net/

[7] http://javaparser.org/

[8] http://www.eclipse.org/jdt/ui/astview/