Java 코드 분석기법을 이용한 UML 클래스 다이어그램 생성 방법

한무희[0] 김경수 김현수
충남대학교 정보통신공학부 컴퓨터공학 전공
(mhhan, kskim, hskim)@ce.cnu.ac.kr

Building a UML class diagram using Java code analysis techniques

Moo-Hee Han[0], Kyoung-soo Kim, Hyeon-soo Kim
Dept. of Computer Engineering, Chungnam National University

요 약

본 연구에서는 자바 코드로부터 UML 클래스 다이어그램을 추출하는 역공학방법을 제시하였다. 파서를 이용하여 자바 코드로부터 AST를 생성하고 이를 순회하면서 클래스다이어그램 생성에 필요한 정보를 추출하였다. 이를 위해 구조정보와 관계정보를 정의하였는데, 구조정보에서는 클래스 몸체를 구성하는 정보를 표현하였다. 관계정보에서는 클래스들 간의 연관관계를 결정하기 위해 필요한 정보를 표현하였으며, 얻어진 관계정보를 통해 연관관계를 유추하는 방법을 제시하였다. 특히 클래스들간의 연관관계를 추출하기 위한 규칙들을 정의하고, 이를 통해 얻어진 관계정보를 이용하여 연관관계를 유출하는 과정을 설명하였다.

1. 서론

1980년대 중반 이후 소프트웨어 개발과정에 도입된 객체지향 개념은 1990년대 중반 Java언어와 UML 이란 객체지향 모델링 언어가 등장하면서 개발자들로부터 엄청난 관심을 받으면서 널리 사용되게 되었다. 이를 계기로 기존의 절차중심의 개발방법이 객체지향 중심 개발형태로 상당수 변화되었으며, 절차중심 개발방법의 단점이던 소프트웨어에 대한 수정 및 확장의 어려움을 극복할 수 있으리란 기대를 하였다.

그러나 소프트웨어의 기능 확장이나 변경작업은 여전히 소스코드에 대한 이해를 필요로 하지만, 많은 경우 도큐먼트와의 불일치로 인해 이는 쉽지 않았다. 이를 극복하기 위한 방법으로 등장한 것이 순환공학이다. 이는 역공학을 통해 기존의 레거시 코드로부터 소프트웨어 시스템의 구조를 얻어내고 이에 대한 수정을 재공학과정을 통해 소스코드에 반영하는 것이다.

본 연구에서는 자바 소스코드로부터 UML 클래스 다이어그램을 추출하는 역공학 방법을 제시하고자 한다. 파서를 이용하여 자바 소스코드로부터 AST(abstract syntax tree)를 얻어낼 수 있고, 이를 순회하면 UML 다이어그램을 표현하기 위해 필요한 정보를 추출해낼 수 있다. 이때 필요한 정보들이 무엇인지 결정할 필요가 있는데, 자바가 클래스를 기본단위로 하는 객체지향 프로그래밍 언어이기 때문에 시스템의 정적인 정보를 표현하는 클래스 다이어그램을 그리는 것을 가장 첫 번째 단계로 보았다.

특히 클래스 다이어그램의 요소들 중 가장 유추하기 어려운 것이 클래스들 간의 연관관계를 결정하는 일인데 이는 자바 코드로부터 명시적으로 표현되지 않기 때문이며 이를 추출하기 위한 방법들에 대해 이 논문의 초점을 두었다.

2. 선행연구

Fujaba 프로젝트에서는 기존의 그래프 모델을 확장한 story diagram이라는 메타모델 형태의 그래프 재작성 언어를 정의하여 UML을 지원하는 순환공학 CASE 도구를 구현하였다[1]. 여기서 클래스 다이어그램을 추출하는 방법은 파서를 통해 얻어진 AST로부터 클래스와 클래스 멤버들을 찾아낸다. 메소드 몸체를 분석하는 과정에서 to-many관계를 얻기 위해 JFC 라이브러리에서 생성되는 컨테이너 클래스에서 미리 정의된 클래스들을 활용하거나, 양방향 연관관계를 얻기 위해서 'annotation engine'이라는 코드 클리쉐를 사용한다. 이를 위해서 fuzzy reasoning 기술을 적용하여 유사한 패턴에 매칭시켜 역공학을 구현한다. 그러나 자바에서 컨테이너 클래스는 포함된 개체들의 타입정보를 알지 못하기 때문에 클래스에 대한 접근 메소드의 의미를 사용자가 미리 알아야 한다.

두 번째는 그래프에 적용한 방법이다[2]. 클래스, 인터페이스, 메소드를 노드집합의 요소로 생각하고, 이들간의 확장, 구현, 참조, 소유, 호출관계 등을 노드간의 에지로 간주하는 그래프 형태로 소스코드 정보를 표현하고 이들간의 관계를 조합하여 클래스 다이어그램에 필요한 정보들을 추출 하였다. 배열타입의 어트리뷰트나, 백터 또는 해쉬테이블 같은 어트리뷰트를 통해 다중성 정보를 얻어낸다. 에지들의 조합을 통해 호출관계 같은 기본적인 패턴을 정의하고 연관관계에서 집합연관을 자동적으로 분류할 수 있다. 그러나 소스코드가 커질 경우 그래프가 담는 정보가 지나치게 커질 수 있어서 정보를 추출하고 관리하는데 시스템의 부하가 많이 걸린다는 단점이 있다.

세 번째 방법은 객체지향 데이터베이스를 위한 ODMG 표준을 메

타모델로 삼았다[3]. 자바 코드로부터 ODMG 표준에서 제공하는 형식으로 클래스 관계를 기술하고, 이를 다시 UML 다이어그램으로 변환하여 표현하는 과정을 보여준다. 클래스 다이어그램의 몇몇 특징들이 ODMG 표준에 적용되는 특징들과 유사한 점에 착안하여 특히 연관관계에 대해 자세히 설명하고 있다.

위 방법들은 패턴이나 코드 클리쉐 같은 특정 도메인에 대한 사전 지식이 필요하며, 그래프나 ODMG같은 중간 메타모델을 두기 때문에 실제 구현 시 시스템의 성능저하를 가져올 수 있다. 그러나 본 연구에서는 자바 소스코드로부터 얻어진 AST를 한번 순회하면서 다이어그램을 표현하는데 필요한 모든 정보를 추출하고 미리 정해진 규칙에 따라 정보들을 정제하여 클래스다이어그램을 표현한다. 그러므로 특정 도메인에 대한 지식이나 도큐먼트 없이 순수하게 자바 소스코드로부터 얻어진 AST에 담긴 정보만을 이용하여 역공학을 구현, 클래스다이어그램을 표현할 수 있다.

3. 클래스 다이어그램 생성을 위한 규칙

이 장에서는 클래스 다이어그램을 그리기 위해 필요한 정보들과 소스코드로부터 얻을 수 있는 정보들을 알아보고 실제 코드에서 정보를 얻기 위한 규칙들을 정의한다.

3.1 클래스 다이어그램을 표기하기 위해 필요한 정보

UML 클래스 다이어그램을 표기하기 위해 필요한 정보들은 클래스 몸체를 구성하는 정보들과 클래스들간의 관계를 표현하는 정보들로 구성된다. 그러나 개념, 명세서, 구현 관점에 따라 표현되는 정보의 수준이 달라질 수 있다. 클래스 몸체는 다시 이름(name)과, 동작(operation), 속성(attribute)에 대한 부분들로 나뉘어 정보를 표현한다. 관계에 대한 정보는 연관관계와 이와 관련된 각종 장식들로 구성되는데 세부 요소들은 4장에서 정의되는 구조정보와 관계정보를 통해 알 수 있으므로 이 장에서는 클래스들간의 연관관계 표현에 대해 살펴보려 한다.

3.2 연관관계: 클래스 다이어그램과 소스코드 관점

UML 클래스다이어그램에서 연관(association)은 클래스의 인스턴스간의 관계를 보여주는 것으로서 각 연관에는 두 개의 연관 끝(association end)이 존재한다. 각 연관 끝에는 역할 이름(role name)과 다중성(multiplicity)을 표기할 수 있다. 다중성이란 참여객체의 하한과 상한 범위이다. 또한 화살표를 통해 운항성(navigability)을 표기할 수 있다. 한정(qualified)연관은 프로그래밍에서 다양하게 알고 있는 연관 배열이나 맵, 사전 등의 개념들에 대한 UML식 표현이다. 집합연관(aggregation)은 전체/부분관계를 말하며 연관과의 차이를 구분하는 것은 쉽지 않다[4].

순공학적인 방법을 따라서 쓰임새를 통한 요구사항 분석 단계를 거친 후 설계단계에서 클래스 다이어그램을 표현하는 것과 달리, 본 논문의 초점이 되는 역공학적인 방법에 의한 소스 코드로부터 클래스 다이어그램을 추출하는 것은 동일한 표기법을 적용하기 어렵다. 특히 역할이름을 부여하는 것과 연관관계를 추출하고 이들 중에서 집합연관을 찾고 분류하는 일은 기계적으로 결정할 수 있는 작업은 아니다. 물론 그래프에 적용시킨 [2]에서는 연관관계에서 집합연관을 자동적으로 분류하려는 시도가 있었지만, 이를 구분하는 것은 사용자의 판단에 맡기는 것이 더 옳다는 판단이 들었다. 따라서 역할이름을 부여하는 일과, 연관관계로부터 집합연관을 구분하는 작업은 사용자에게 맡기도록 한다.

3.3 소스코드로부터 연관관계 유추

다른 클래스의 객체를 참조하는 관계는 일반적으로 호출관계를 통해 얻을 수 있으므로, 선행연구[2]에서는 호출관계를 몇 가지 문맥에 따라 보다 세부적으로 다음과 같이 분류하였다.

- this: 동일한 객체를 호출
- super: 자신의 조상을 호출
- attribute: 어트리뷰트로 선언된 객체 호출
- local: 메소드의 지역 변수로 선언된 객체 호출
- result: 메소드 결과 객체 참조
- static: 정적 메소드를 호출

이 방법에서는 연관관계 생성에 도움이 되지 않는 관계들을 제거하였는데 이를테면 this의 경우 동일한 객체에 대한 메소드 호출은 매우 흔하며 따라서 많은 정보를 얻지 못하므로 제거된다. super 역시 동일한 조상을 호출한 경우, 상속 받는 메소드는 재정의 되기 때문에 이런 호출은 제거된다. static의 경우 자바에서 정적 메소드에 대한 호출은 흔히 전역적으로 가용한 System이나 Math같은 표준 클래스들을 잡고 있기 때문에 역시 제거된다.

그러나 위와 같이 단순한 기준으로 호출관계를 제거하는 것은 문제가 있다. 왜냐하면 자신의 타입을 갖는 객체를 참조하는 재귀연관(recursive association)관계를 표현하기 위해 this관계도 사용되어야 하며, static관계 역시 사용자 정의 경우에는 제외되어서는 안 된다. 비록 클래스 다이어그램이 시스템의 정적인 구조를 표현하기 위한 것이지만, 클래스간 연관관계는 실제로 해당 타입의 객체를 통해 표현되므로, 단순히 호출관계만 고려하기 보다는 코드 내에서 실제로 객체가 생성되거나 참조 또는 호출되는 정보를 모두 포함하여 연관관계를 유추하는 것이 더 바람직하다.

이런 관점에서 연관관계를 유추하기 위한 규칙을 재정의 하면 다음과 같다. 다음에 열거하는 각각의 경우에는 연관관계가 있음을 유추할 수 있다.

1. 클래스A의 어트리뷰트나 혹은 메소드 내에서 사용자 정의된 클래스B 타입의 객체를 선언하고 이 객체의 어트리뷰트를 참조하거나, A에서 B 객체의 메소드를 호출하는 경우

예를 들어 그림1과 같이 Teacher 클래스와 참조된 Course 클래스간에 관계가 있음을 유추할 수 있다.

```
public class Course { ...
   static int addHours() { ... }
   public int setCourseName(Course c) { ... }
}
public class Teacher { ...
   public void createCourse(String cName) {
          Course c = new Course();
          c.setCourseName(cName);
... }}
```

그림 1. Course 객체의 생성 및 메소드 호출

2. 클래스A에 속한 메소드의 아규먼트로서 클래스B 타입의 객체

를 전달하는 경우

예를 들어 그림2에서 Student 클래스와 아규먼트로 넘겨지는 Course 클래스와 관계가 있음을 유추 할 수 있다.

```
public class Test {
public static void main(String args[]) {
        Teacher t = new Teacher("Harry");
        Course c = t.createCourse("Math", 10);
        Student s = new Student();
        s.refer(c);
} }
```

그림 2. 메소드의 아규먼트로 객체 전달

3. 전역적으로 참조가 가능하도록 선언된 클래스의 메소드나, 어트리뷰트를 직접 참조하는 경우

예를 들어 그림3에서 Student 클래스에서 Course 클래스 타입의 객체를 선언하지 않고 hours 메소드를 직접 참조한다. 역시 두 클래스간에 어떠한 관계가 있음을 유추할 수 있다.

```
class Student {
   int hours;
   public int calcHours() {
      this.hours = Course.addHours();
   ... } ... }
```

그림 3. 전역 메소드 참조

4. 클래스 다이어그램 생성

4.1 구조정보 및 관계정보의 정의

클래스 다이어그램에 표현되는 정보들은 클래스 이름, 메소드 정보, 필드 정보 등 클래스 자체에 대한 정보와, 상속, 인터페이스의 구현, 의존, 연관관계 등 클래스들간의 관계를 표현하는 정보들로 구분할 수 있다. 클래스 자체에 대한 정보를 '구조정보', 클래스들 간의 관계들에 대한 정보를 '관계정보'라고 정의한다.

구조정보는

- 클래스: 이름
- 생성자: 이름, 아규먼트
- 메소드: 가시성(visibility), 이름, 아규먼트, 리턴타입
- 어트리뷰트: 가시성, 타입, 이름, 초기값

들로서 생성자는 사용자에 의해 정의된 것을 말한다. 이 정보들은 AST를 순회하면서 바로 얻을 수 있다.

관계정보는

- 상속 / 구현관계
- 연관관계: 연관(association), 집합연관(aggregation)

들이다.

4.2 클래스 다이어그램 생성

자바 코드로부터 클래스 다이어그램을 그리기 위한 정보를 추출하는 방법은 다음과 같다.

1. 구조정보를 얻기 위해 AST를 순회하면서 해당정보를 저장한다.
   - 이름: 클래스이름
   - 동작: 생성자, 메소드 정보
   - 속성: 어트리뷰트 정보
2. 관계정보 중에서 상속 및 인터페이스 구현관계를 찾는다. 'extends'란 예약어 다음에 나오는 클래스로부터 상속된 것이며 인터페이스 구현은 'implements'란 예약어 다음에 나오는 인터페이스로부터 구현 된 것이므로, 해당 예약어를 검색하고 있으면 그 후에 나오는 클래스와 인터페이스 이름을 저장한다.
3. 관계정보 중에서 유추할 수 있는 정보를 추출한다.

메소드 내부를 순회하는 도중 다른 클래스 타입의 객체를 생성, 참조(호출), 수정, 제거하는 작업이 발견될 경우, 대상이 되는 객체 이름과 이들의 타입이 되는 클래스 이름, 이 객체에 대해 메소드가 수행한 작업을 메소드 이름과 함께 테이블 형태로 저장한다. 예를 들어 Student 클래스를 순회하는 동안 두 메소드에서 Course 클래스를 각각 참조하였다면 그림4와 같이 표기할 수 있으며, 이를 통해 두 클래스가 연관관계가 있음을 유추할 수 있다.

참조되는 객체이름	-	c
참조되는 어트리뷰트	-	-
참조되는 메소드	addHours	getGrade
참조되는 클래스 타입	Course	Course

그림 4. 연관정보 테이블

4. 연관관계 정보를 얻는다.

모든 메소드에 대한 순회를 마치면 구조정보를 통해 클래스 몸체를 생성하고, 연관정보 테이블을 통해 자신 클래스와 연관이 있는 클래스들을 알 수 있게 된다. 테이블에 저장된 클래스들이 이 클래스와 연관관계가 있는 것들이며, 다중성정보는 한 클래스타입으로 new 연산에 의해 생성되는 객체의 숫자를 통해 파악할 수 있다.

5. 결론

본 논문에서는 자바 코드로부터 얻은 AST 정보를 순회하여 클래스 다이어그램을 표현하기 위해 필요한 정보를 추출하고 이를 정제하여 다이어그램을 표현하는 과정을 설명하였다.

그러나 단순하게 연관관계가 있는 클래스들과 다중성 정보만을 추출할 수 있으므로 여기서 집합연관관계의 후보가 되는 클래스들을 추출하고 한정연관, 연관 클래스 등 연관관계를 보다 세부적으로 표현할 수 있는 정보추출에 대한 연구를 더 진행해야 하리라 본다.

6. 참고문헌

[1] J. Niere, J. Wadsack and A. Zündorf: Recovering UML Diagrams from Java Code using Patterns. *Proc. of 2nd Workshop on Soft Computing Applied to Software Engineering*, LNCS. Springer, 2001

[2] J. Seeman and J. Wolff, "Pattern-based design recovery of Java software," *Proc. of the 6th ACM Sigsoft Int'n Symp. on Foundations of Software Engineering*, 1998

[3] R. Kollmann and M. Gogolla, "Application of UML associations and their adornments in design recovery," *Proc. Of the 8th working conference on reverse engineering*, IEEE, 2002

[4] M. Fowler and K. Scott, UML distilled 2nd edition ch4, ch6, Addison Wesley, 2000