韓國 컴퓨터情報學會 論文誌
第13卷 第3號, 2008. 5.

2008-13-3-1-1

자바 클래스 파일로부터 객체 다이어그램 얻기 : 실제적인 방법

양 창 모*

Generating Object Diagrams from Java Class Files : A Practical Approach

Yang, Changmo *

요 약

Gestwicki 등은 객체 그래프의 더 좋은 그림을 얻기 위하여 클래스 다이어그램으로부터 객체 다이어그램의 구조를 탐지하고 예측하는 기술을 제안하였다. 그들의 방법은 클래스 다이어그램으로부터 두 가지 예측 가능한 구조 - 되부름 덩어리(recursive cluster)와 잎 덩어리(leaf cluster)를 찾는다. 하지만 이 방법은 출발점이 클래스 다이어그램이므로 실제 프로그램에 적용하기에는 실제적이지 않다.

본 연구는 자바 클래스 파일에 적용하기 위하여 그들의 방법을 확장하고 구현한다. 자바 클래스 파일로부터 추출한 클래스들과 덩어리들의 구조와 관련성을 얻는다. 이 정보는 JIVE와 같은 자바 프로그램 실행 시각화 도구에게 제공되어 객체 다이어그램의 틀로 사용될 수 있다.

Abstract

Gestwicki et. al. proposed the technique that detects and predicts the structure of object diagrams from class diagrams to get the improved drawing of object graphs. Their approach finds two predictable structures - recursive clusters and leaf clusters from class diagrams. Their approach is not practical to be applied to real programs, because the starting point is class diagram.

In this work, we improve and implement their technique to apply to Java classes. We obtain the structure and relationships of classes and clusters extracted from class files. This information can be provided and used as a template of object graphs for Java execution visualization tools like JIVE.

▸ Keyword : 자바클래스파일, UML 다이어그램, 객체 다이어그램(Object Diagram), 역공학(Reverse Engineering)

• 제1저자 : 양창모
• 접수일 : 2008. 2. 15, 심사일 : 2008. 3. 25, 심사완료일 : 2008. 5. 24.
* 청주교육대학교 컴퓨터교육과

I. 서 론

JIVE(Java Interactive Visualization Environment)는 UML 클래스 다이어그램과 순차 다이어그램(sequence diagram)을 사용하여 자바 프로그램이 실행되는 동안 객체의 구조를 보여주어 사용자가 프로그램의 실행 상태를 보다 쉽게 이해할 수 있도록 도와주는 훌륭한 시각화 도구이다[1, 2, 3,4]. JIVE를 포함한 대부분의 시각화 도구는 프로그램의 정적인 구조를 고려하지 않고 객체와 그들 사이의 관련성을 방향 그래프로 간주하여 전통적인 그래프 그리기 기술[5]을 사용한다. 전통적인 그리기 방법에 클래스들의 특성을 적용하면 좀 더 효율적인 그림을 얻을 수 있다는 점에 착안하여 Gestwicki 등은 사용자들이 마음속에 그리는 객체들의 모습과 비슷하게 객체 다이어그램을 그릴 수 있도록 클래스 다이어그램으로부터 객체 다이어그램의 구조를 예측하는 방법을 제안하였다[1,4].

이 방법은 일반화(generalization), 연관(association), 집단화(aggregation)라는 세 가지 이진 객체 관련성의 정의[6,7]를 사용하여 되부름 덩어리와 잎 덩어리를 찾는다. 되부름 덩어리는 리스트, 트리, 그래프 등과 같이 되부름 타입(recursive type)으로 정의된 타입으로 구성되며, 잎 덩어리는 상대적으로 간단한 구조로 다른 객체를 참조하지 않는 잎 객체(leaf class)들과 그들을 유일하게 직접 참조하는 잎 모으개(aggregator)로 구성된다. 이 방법은 독창적이며 언어 독립적인 방법이기는 하지만 클래스 다이어그램을 대상으로 하였기에 자바와 같은 실제 프로그램에 적용하기에는 실질적이지 않으며 그러한 이유로 아직 구현되지 않았다.

본 연구에서는 [1,4]의 방법을 확장하여 자바 클래스 파일들로부터 되부름 덩어리와 잎 덩어리를 추출하는 방법을 구현하고자 한다.

[1,4]의 연구는 언어 독립적인 정형화 방법(formalization)을 사용하여 실제 프로그램에는 직접적으로 적용할 수 없다.

[1,4]의 연구에서 객체의 집단화 관련성만을 사용하여 객체 그래프에서의 되부름 타입을 추출하지만 본 연구에서는 특정 객체의 어트리뷰트가 아닌 객체(메서드 안에서 생성되는 객체)까지 고려하여 되부름 타입을 추출한다.

본 연구는 자바 프로그램을 분석하여 클래스들 사이의 관련성을 추출하고 잎 덩어리와 되부름 덩어리를 식별해낸다. 분석 결과는 클래스들의 구조와 관련성 정보를 포함하는 XML 형식의 문서와 텍스트 형식의 문서이며, XML 형식의 문서를 그림으로 보여주기 위하여 Graphviz[8]의 DOT 파일을 추가적으로 생성한다. 이 정보는 JIVE와 같은 자바 프로그램 실행 시각화 도구에게 제공되어 객체 다이어그램의 틀로 사용될 수 있다. 본 연구에서 제안하는 시스템의 개략적인 구조는 <그림 1>과 같다.

본 논문의 구성은 다음과 같다. II 절에는 자바 프로그램으로부터 클래스 관련성을 출하는 관련 연구에 대하여 소개한다. III 절에서는 클래스 다이어그램으로부터 관련성을 추출하는 기존의 방법과 한계점에 대하여 설명하고 IV 절에서는 III 절에서 제시한 문제점을 해결하고 자바 클래스 파일로부터 관련성을 추출하는 방법에 대하여 설명한다. V 절에서는 IV 절에서 얻은 클래스들의 관련성을 사용하여 클래스들의 덩어리를 만드는 방법을 설명한다. VI 절에서 본 논문에서 제안한 방법을 구현한 결과를 보여주고 VII 절에서 결론을 내리고 향후 연구에 대하여 기술한다.

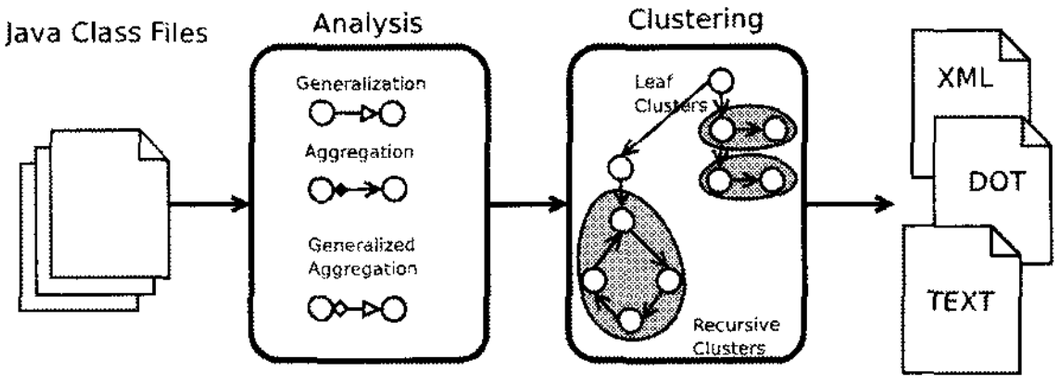

그림 1. 시스템의 개요
Fig 1. Overview of System

II. 관련연구

자바 프로그램으로부터 클래스 관련성을 추출하는 연구는 주로 역공학(reverse engineering) 분야에서 자바 프로그램으로부터 UML 다이어그램을 얻기 위한 연구들이다.

Jackson과 Waingold는 자바 바이트코드로부터 부분타입부여(subtyping)관계와 연관 관련성을 추출하고 이 연관성을 이용하여 객체 모델을 시각화하였다[9]. 이 연구의 부분 클래스 관계는 본 연구의 일반화와 비슷한 개념이며, 연관은 본 연구와 Guéhéneuc 등의 연구의 집단화와 비슷하다.

Gogolla와 Kollmann은 역 연관(inverse association)과 컬렉션(collection) 타입을 중심으로 자바 원시 코드에서 연관, 집단화, 관련성의 다중도(multiplicity)를 정적으로 계산하여 UML 클래스 다이어그램으로 변환하는 방법을 제안하였다[10].

Barowski와 Cross II는 자바 바이트코드로부터 상속과 인터페이스 구현 측면의 클래스 간 종속성을 추출하여 UML

로 보여주는 jGRAPS라는 도구를 구현하였다[11]. 이 연구에서 종속성은 UML에서 일반화와 구체화(realization)에 해당한다.

Guéhéneuc 등은 의미가 모호한 UML의 이진 클래스 관련성인 연관, 집단화, 조합(composition) 관련성을 명확히 정의하기 위하여 자바의 언어 구조를 사용하였고 자바 프로그램으로부터 이진 클래스 관련성을 찾는 알고리즘을 제시하고 실험하였다[6,7].

Keschenau는 자바 바이트코드로부터 클래스들의 연관과 다중도를 계산하는 방법을 제시하였고 class2uml이라는 도구를 구현하였다[12].

이상의 연구들은 자바 프로그램으로부터 UML의 일반화, 연관, 집단화, 다중도를 추출하여 객체 다이어그램의 구조를 추측하려는 본 연구와 비슷한 부분은 있지만 전체적으로는 큰 차이점이 있다. [9,11,12]는 자바 바이트코드로부터 UML의 관련성을 추출한다는 면에서 본 연구와 비슷하지만 본 연구에서 필요한 정보를 모두 얻지는 못하였다. 자바 프로그램으로부터 추출한 관련성을 사용하여 실행시간에 나타나는 객체의 구조를 분석하려는 본 연구와는 달리 [6,7]은 객체들의 관련성을 보여주는 것으로 마무리하였다.

III. 클래스 다이어그램으로부터 관련성 추출

Getswicki 등은 [6,7]에서 제안한 일반화, 연관, 집단화의 개념을 사용하여 실행시간에 그려지는 객체 다이어그램의 구조를 예측할 수 있는 클래스의 중요한 특성인 되부름 덩어리와 잎 덩어리를 정형적으로 정의하고, 클래스 다이어그램에서 되부름 덩어리와 잎 덩어리를 찾는다. 그 과정을 다음과 같이 요약할 수 있다[1,4].

클래스 다이어그램으로부터 일반화 그래프와 집단화 그래프를 구성하고 이것들을 바탕으로 일반화된 집단화 경로를 구성한다.

일반화된 집단화 경로를 이용하여 되부름 타입을 찾는다. 되부름 타입은 일반화된 집단화 경로 상에서 자기 자신에게 돌아오는 경로가 존재하는 되부름 클래스들로 구성된다.

한 클래스에서 다른 클래스로의 일반화된 집단화 관련성이 없는 클래스들을 잎 클래스로 간주한다.

잎 클래스들과 잎 클래스들을 유일하게 집단화하는 잎 모으개를 모아 잎 덩어리를 얻는다.

되부름 덩어리는 동일한 되부름 타입을 갖는 클래스의 모임이다. 모으개가 되부름 타입에 속하면 되부름 덩어리는 잎 덩어리를 포함할 수도 있다.

서론에서 언급한 것과 같이 본 연구와 Gestwicki 등의 연구[1,4]와의 차이점은 집단화 관련성을 찾는 방법에 있다. 본 연구에서는 두 종류의 관련성을 집단화 관련성으로 사용한다. 하나는 전체-부분 관련성에 해당하는 관련성으로 명시적 집단화 관련성이라 한다. Gestwicki 등의 연구에서 사용한 집단화 관련성은 연관 관련성과 전체-부분 관련성을 모두 만족해야하는 관련성으로 너무 엄격하여 클래스들 사이의 관련성을 모두 포함할 수 없다. 예를 들어 〈그림 14〉의 이진 검색 트리의 예에서 클래스 BST와 클래스 Data는 전체-부분 관계를 갖지만 BST에서 Data의 인스턴스를 호출하지 않기 때문에 연관 관련성을 갖지 않는다. 연관 관련성을 갖기 위하여 메서드 호출이외에 어트리뷰트에 대한 접근까지 고려한다면 이 정의를 사용할 수 있다. 이 경우에도 연관 관련성과 집단화 관련성 모두를 만족하는 것은 더 많은 관련성을 추출하기에 제약으로 동작한다. 이러한 문제를 해결하기 위하여 본 연구에서는 엄격한 집단화 관련성 대신 전체-부분 관련성을 집단화 관련성으로 사용하고 명시적 집단화 관련성이라 부른다.

본 연구에서 사용하는 두 번째 집단화 관련성은 어트리뷰트로 선언되지는 않았으나 객체 생성 명령에 의하여 생성되는 객체의 클래스와 그 객체를 생성하는 객체의 클래스 사이의 관련성이다. 이러한 관련성을 묵시적 집단화 관련성이라 부른다. 클래스 A에서 클래스 B로의 묵시적 집단화 관련성이 있다는 것은 A 타입의 객체가 B 타입의 객체를 생성한다는 것을 의미한다. 이러한 관련성이 필요한 이유를 설명하기 위하여 팩토리알을 구하는 〈그림 2〉의 프로그램을 보면 클래스 Factorial은 클래스 FactMain에서 어트리뷰트로 선언되지 않았으므로 FactMain과 Factorial은 연관 관련성만을 갖는다. 이 두 클래스들은 집단화 관련성을 갖지 않으므로 [1,4]의 방법을 적용하면 〈그림 3(a)〉와 같은 관련성 그래프를 얻는다. 하지만 이 프로그램은 실행되는 동안 Factorial 객체가 자신을 반복적으로 생성하여 되부름 구조를 갖기 때문에 〈그림 3(b)〉와 같은 관련성을 가져야 한다. 본 연구에서는 이러한 관련성을 묵시적 집단화 관련성이라 부르고 묵시적 집단화 관련성을 추출하기 위하여 어떤 클래스의 메서드 내에서 어떤 객체를 생성하는지 분석한다.

```
public class FactMain {
  public static void main(String args[]) {
    int num = 5;
    Factorial factorial = new Factorial();
    int fact = factorial.compute(num);
  }
}
class Factorial {
  public int compute(int f) {
    int result = 0;
    if (f>2) {
      Factorial factorial = new Factorial();
      result = f * factorial.compute(f-1);
    } else {
      result = f;
    }
    return f;
  }
}
```

그림 2. 자바 팩토리알 프로그램
Fig 2. Java Factorial Program

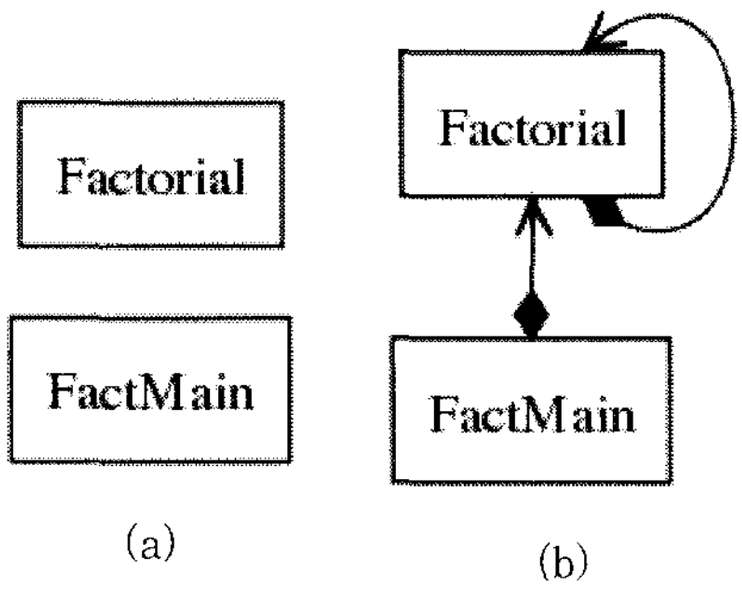

그림 3. 그림 2의 관련성 그래프
Fig 3. Dependency Graph of Fig 2.

Ⅳ. 클래스 파일로부터 관련성 추출

본 연구에서 사용하는 방법은 특정 코드 패턴을 찾아 클래스들 사이의 관련성을 추출하기 위하여 클래스 파일을 분석하는 코드 패턴 기반이다. 클래스 관련성 분석은 자바 클래스 파일 단위로 이루어진다. 사용자가 지정한 클래스 파일로부터 분석을 시작하여 분석된 클래스들과 관련이 있는 다른 클래스 파일들로 분석을 진행한다. java.lang, java.util 등의 API 그리고 Integer와 같은 싸개 클래스(wrapper class)들을 무시한다. 클래스들의 관련성을 표현하기 위하여 노드가 자바 클래스이고 연결선이 클래스들 사이의 연결선을 의미하는 방향 그래프를 사용한다.

4.1. 자바 클래스 파일

자바 클래스 파일은 자바 클래스에 대응하는 실행 파일이다. 클래스 파일 형식은 자바 가상 기계 명세[13]에서 잘 정의되어 있다. 모든 자바 컴파일러는 이 명세에 따라 클래스 파일을 만들어야 하므로 클래스 파일들은 사용된 특정 컴파일러에 관계없이 분석에 필요한 데이터를 동일하게 제공한다. 자바 클래스 파일은 수퍼클래스의 이름, 인터페이스의 이름, 필드의 선언, 메서드 안의 바이트코드 정보를 제공한다.

자바 클래스 파일을 사용하는 중요한 장점은 자바 프로그램의 복잡한 컴파일 과정을 검증된 자바 컴파일러가 수행한 결과를 사용할 수 있다는 것과 자바 원시 프로그램을 사용할 수 없는 경우에도 분석이 가능하다는 것이다[11].

4.2. 일반화 관련성

자바 클래스 파일의 수퍼클래스와 인터페이스에 대한 정보를 사용하여 클래스간의 일반화 관련성을 〈정의 1〉과 같이 계산할 수 있다. 자바 클래스들의 일반화 관련성을 나타내는 일반화 그래프를 〈정의 2〉와 같이 정의할 수 있다.

정의 1. 일반화 관련성. 다음 중 하나의 조건을 만족하면 클래스 C_1에서 클래스 C_2로 일반화 관련성이 존재한다고 한다고 하고 $C_1 \triangleright C_2$로 표시한다.

클래스 C_1이 클래스 C_2로부터 상속받는다.

클래스 C_1이 클래스 C_2를 구현한다. □

정의 2. 일반화 그래프. 주어진 자바 프로그램 P가 클래스 $C_1, \cdots, C_n$으로 구성될 때, 일반화 그래프 $G=(V,E)$는 다음을 만족하는 그래프이다.

$V=\{C_1, C_2, \cdots, C_n\}$

$E=\{(C_i, C_j) \mid C_i, C_j \in V \wedge C_i \triangleright C_j\}$ □

〈그림 4〉의 프로그램에서 클래스 A는 B로부터 상속받고 I를 구현하므로 A에서 B로, A에서 I로 일반화 관련성이 존재한다. 따라서 〈그림 5〉와 같은 일반화 그래프를 얻을 수 있다.

```
class A extends B implements I { ... }
```

그림 4. 일반화 관련성을 갖는 클래스들
Fig 4. Classes with Generalization

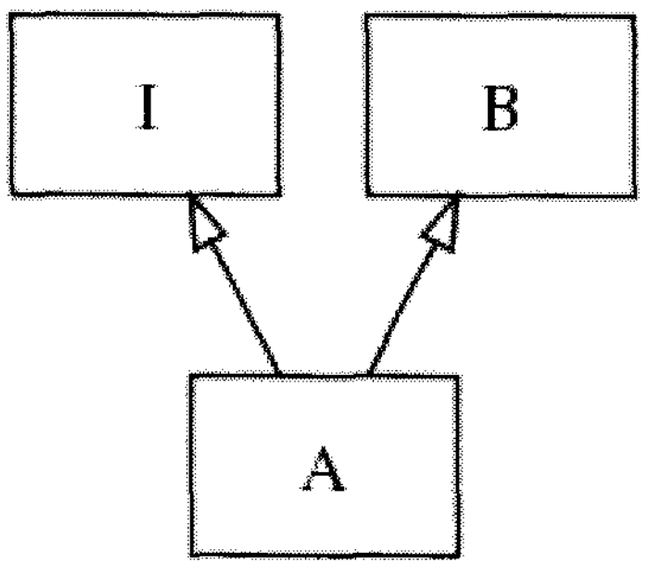

그림 5. 클래스 A, B, I의 일반화 그래프
Fig 5. Generalization Graph of A, B, and I

〈알고리즘 1〉은 클래스 파일들로부터 일반화 그래프를 얻는 알고리즘이다. 이 알고리즘은 입력으로 주어진 이름의 클래스 파일로부터 종속된 모든 클래스 파일을 읽어가면서 클래스 파일의 수퍼클래스와 인터페이스를 찾아 일반화 관련성을 추출하여 일반화 그래프를 만드는 알고리즘이다. 이 알고리즘과 이어지는 알고리즘들을 위하여 다음과 같은 메서드를 가정한다.

readClassFile() 클래스 파일의 내용을 읽는다.
getThisClass() 클래스의 이름을 얻는다.
getSuperClass() 수퍼클래스의 이름을 얻는다.
getInterfaces() 인터페이스들의 이름을 얻는다.
getElement() 집합에서 하나의 원소를 얻는다.
getDependents() 클래스 파일의 상수 풀을 검사하여 종속된 클래스들의 이름을 얻는다.

알고리즘 1. 일반화 그래프 생성 알고리즘
Algorithm 1. Algorithm to Generate Generalization Graph

Input : 클래스 파일 이름 $className$
Output: 일반화 그래프 $G=(V,E)$
Method generateGenailzationGraph($className$)
  $javaClass \leftarrow readClassFile(className)$
  $thisClass \leftarrow javaClass.getThisClass()$
  $V \leftarrow \{thisClass\}$ $E \leftarrow \varnothing$ $N \leftarrow \{thisClass\}$
  while $N \neq \varnothing$ **do**
    $superClass \leftarrow javaClass.getSuperClass()$
    $V \leftarrow \{superClass\}$
    $E \leftarrow E \cup \{(thisClass, superClass)\}$
    $interfaces \leftarrow javaClass.getInterfaces()$
    for all $interface \in interfaces$ **do**
      $V \leftarrow V \cup \{interface\}$
      $E \leftarrow E \cup \{(thisClass, interface)\}$
    end for
    $dependents \leftarrow javaClass.getDependents()$
    $N \leftarrow N \cup dependents - \{thisClass\}$
    $className \leftarrow N.getElement()$
    $javaClass \leftarrow readClassFile(className)$
    $thisClass \leftarrow javaClass.getThisClass()$
  end while
  return G
end Method

현재 클래스(thisClass)에서 수퍼클래스(superClass)와 인터페이스(interface)로의 연결선을 일반화 그래프에 추가하여 일반화 그래프를 완성한다.

4.3. 연관 관련성

클래스 A에서 클래스 B로의 연관 관련성이 있다는 것은 A의 인스턴스가 B의 인스턴스를 참조하는 것을 의미한다. 자바 바이트코드에서는 메서드 호출 명령으로 인스턴스의 참조를 구현한다. 클래스 A에 클래스 B로의 인스턴스의 메서드를 호출하는 명령이 하나 이상 존재한다면 A에서 B로의 연관 관련성이 존재한다. 연관 관련성은 〈정의 3〉과 같이 정의된다.

정의 3 연관 관련성. 클래스 C_1의 인스턴스가 클래스 C_2의 인스턴스를 호출한다면 C_1에서 C_2로 연관 관련성 $C_1 \rightarrow C_2$이 존재한다고 한다고 한다. □

정의 4 일반화 그래프. 주어진 자바 프로그램 P가 클래스 $C_1, \cdots, C_n$으로 구성될 때, 연관 그래프 $G=(V,E)$는 다음을 만족하는 그래프이다.
$V=\{C_1, C_2, \cdots, C_n\}$
$E=\{(C_i, C_j) \mid C_i, C_j \in V \wedge C_i \rightarrow C_j\}$ □

```
class C {                    public class C {
  void method(C c) {           void method() {
    c.method();                }
  }                            }
}
```

그림 6. 연관 관련성을 갖는 클래스들
Fig 6. Classes with Association

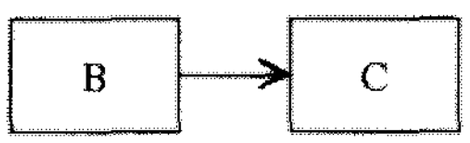

그림 7. 클래스 B와 C의 연관 그래프
Fig 7. Association Graph of Class B and C

〈그림 6〉은 연관 관련성을 갖는 클래스들의 예이다. B와 C 사이의 연관 관련성은 메서드 호출 c.method()로 인하여 존재한다. 〈그림 7〉은 클래스 B와 C의 연관 그래프이다.

알고리즘 2. 연관 그래프 생성 알고리즘
Algorithm 2. Algorithm to Generate Association Graph

```
Input : 클래스 파일 이름 className
Output: 연관 그래프 G = (V, E)
Method generateAssociationGraph(className)
  javaClass←readClassFile(className)
  thisClass←javaClass.getThisClass()
  V← {thisClass} E← ∅ N← {thisClass}
  Call←{INVOKEINTERFACE,INVOKESTATIC,INVOKEVIRTUAL}
  while N ≠ do
    Methods←javaClass.getMethods()
    for all method∈Methods do
     while method.hasNextInstruction() do
      inst←method.getNextInstruction()
      opcode←inst.getOpcode()
      if opcode∈CALL then
       operand← inst.getOperand()
       classType←operand.getClass()
       V← V∪ {classType}
       E ← E∪ {(thisClass.classType)}
       N←N∪ {classType}
      end if
     end while
    end for
    dependents←javaClass.getDependents()
    N ← N ∪ dependents − {thisClass}
    className ← N.getElement()
    javaClass←readClassFile(className)
    thisClass←javaClass.getThisClass()
  end while
  return G
end Method
```

〈알고리즘 1〉과 비슷한 방법으로 〈알고리즘 2〉는 입력으로 주어진 클래스 파일이 종속되는 클래스 파일들을 순방하면서 메서드가 호출되는 클래스를 찾아 연관 그래프를 만든다. 이 알고리즘은 다음과 같은 메서드들을 가정한다.

getMethods() 클래스의 메서드들을 읽는다.

hasNextInstruction() 현재 명령어가 현재 메서드에 있는 바이트코드의 마지막 명령어라면 참, 그렇지 않으면 거짓을 돌려준다.

getNextInstruction() 현재 명령어가 현재 메서드에 있는 바이트코드의 마지막 명령어가 아니라면 다음 명령어를 돌려준다.

getOpcode() 명령어의 연산코드를 돌려준다.

getOperand() 명령어의 피연산자를 돌려준다. 메서드 호출 명령어의 피연산자는 클래스의 이름과 메서드의 이름의 쌍이다.

getClass() 메서드 호출 명령어의 피연산자의 클래스 이름을 돌려준다.

〈알고리즘 2〉는 클래스 파일의 모든 메서드를 읽으며 명령어가 INVOKEINTERFACE, INVOKESTATIC, INVOKEVIRTUAL 명령어 가운데 하나인지 검사한다. 그렇다면 피연산자의 클래스 이름이 노드의 집합에 추가되고 현재 클래스에서 피연산자의 클래스 이름으로의 연결선이 연결선의 집합에 추가된다.

4.4. 집단화 관련성

집단화 관련성은 클래스들의 구조적 종속을 표현한다. 구조적 종속 관계에 따라 객체 그래프를 그리므로 이 관련성은 객체 그래프 그리기에서 중요한 역할을 차지한다.

클래스 A에서 클래스 B로의 명시적 집단화 관련성이 존재한다는 것은 A가 타입이 클래스 B인 어트리뷰트를 하나 이상 갖는다는 것을 의미한다. 타입 A의 객체가 타입 B의 객체를 만든다면 A에서 B로의 묵시적 집단화 관련성이 존재한다. 메서드의 바이트코드에서 NEW, NEWARRAY, MULTIANEWARRAY, ANEWARRAY와 같은 객체 생성 명령어를 찾음으로써 묵시적 집단화 관련성을 추출한다.

집단화 관련성은 〈정의 5〉와 같다. 첫 번째 항목은 명시적 집단화 관련성에 해당되며, 두 번째 관련성은 묵시적 집단화 관련성에 해당한다. 이 정의로부터 집단화 그래프를 〈정의 6〉과 같이 구성한다.

정의 5 집단화 관련성. 다음과 같은 성질 가운데 하나가 만족되면 $C_1 \diamond\!\rightarrow C_2$로 표현되는 C_1에서 C_2로 집단화 관련성이 존재한다고 한다고 한다.
클래스 C_1이 타입이 C_2인 어트리뷰트를 갖는다.
클래스 C_1의 하나 이상의 메서드에서 타입이 C_2인 객체를 만든다. □

정의 6 집단화 그래프. 주어진 자바 프로그램 P가 클래스 $C_1, \cdots, C_n$으로 구성될 때, 집단화 그래프 $G=(V,E)$는 다음을 만족하는 그래프이다.

$V=\{C_1, C_2, \cdots, C_n\}$

$E=\{(C_i, C_j) \mid C_i, C_j \in V \wedge C_i \diamond\!\rightarrow C_j\}$ □

〈그림 8〉의 코드는 집단화 관련성을 갖는 클래스들의 예이다. 명시적 집단화 관련성은 어트리뷰트 선언 public D d로 인한 것이며, 묵시적 집단화 관련성은 클래스 A에서 C c =

new C()로 인하여 객체를 생성함으로써 발생한다. 〈그림 9〉는 클래스 C와 D의 집단화 그래프이다. 두 종류의 집단화 관련성을 구분하기 위하여 다른 모양의 연결선을 사용하였다.

```
class A {                    public class B {
  void method(C c) {           public C c;
    C c = new C();           }
  }                          public class C { }
}
```

그림 8. 집단화 관련성을 갖는 클래스들
Fig 8. Classes with Aggregation

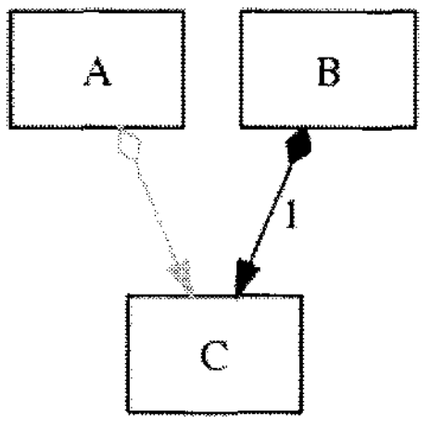

그림 9. 클래스 C와 D의 집단화 그래프
Fig 9. Aggregation Graph of C and D

〈알고리즘 3〉도 입력으로 주어진 클래스 파일에 종속되는 클래스 파일들을 순방하면서 클래스들이 어떤 타입의 필드를 갖는지 메서드에서 어떤 객체가 만들어지는지를 조사한다. 이 알고리즘은 다음과 같은 메서드를 사용한다.

getFields() 클래스의 필드를 가져온다.

hasClassType() 필드가 주어진 클래스 타입을 갖는다면 참을, 갖지 않는다면 거짓을 돌려준다.

getType() 필드의 타입을 돌려준다.

이 알고리즘에서 명시적 집단화 관련성을 첫 번째 forall 구조에서, 추출하며 묵시적 집단화 관련성을 두 번째 forall 구조에서 추출된다.

4.5. 일반화된 집단화 관련성

클래스들로부터 되부름 덩어리를 추출하려면 자신과 집단화 관련성을 갖는 클래스들을 찾아야 한다. 이러한 클래스들의 모임을 되부름 타입이라 부른다. 집단화 관련성이 상속되기 때문에 일반화 관련성도 고려하여야 한다[1,4].

〈그림 10〉에서 AbsTree는 Element을 집단화하고, StrElm은 Element를 일반화하며, Tree는 AbsTree를 일반화한다. Tree에서 AbsTree로의 일반화 관련성과 StrElm에서 Element로의 일반화 관련성으로 인하여 Tree는 Element를 집단화하고 AbsTree는 StrElm을 집단화한다.

알고리즘 3. 집단화 그래프 생성 알고리즘
Algorithm 3. Algorithm to Generate Aggregation Graph

Input : 클래스 파일 이름 $className$
Output: 집단화 그래프 $G=(V,E)$
Method generateAggregationGraph($className$)
 $javaClass \leftarrow readClassFile(className)$
 $thisClass \leftarrow javaClass.getThisClass()$
 $V \leftarrow \{thisClass\}\ E \leftarrow \varnothing\ N \leftarrow \{thisClass\}$
 while $N \neq$ **do**
  $Fields \leftarrow javaClass.getFields()$
  for all $field \in Fields$ **do**
   if $field.hasClassType()$ **then**
    $fieldType \leftarrow field.getType()$
    $V \leftarrow V \cup \{fieldType\}$
    $E \leftarrow E \cup \{(thisClass, fieldType)\}$
    $N \leftarrow N \cup \{fieldType\}$
   end if
  end for
  $Methods \leftarrow javaClass.getMethods()$
  for all $method \in Methods$ **do**
   while $method.hasNextInstruction()$ **do**
    $inst \leftarrow method.getNextInstruction()$
    $opcode \leftarrow inst.getOpcode()$
    if $opcode \in$
     $\{NEW, NEWARRAY, ANEWARRAY, MLTIANEAARAY\}$
    then
     $operand \leftarrow inst.getOperand()$
     $classType \leftarrow operand.getClass()$
     $V \leftarrow V \cup \{classType\}$
     $E \leftarrow E \cup \{(thisClass, classType)\}$
     $N \leftarrow N \cup \{classType\}$
    end if
   end while
  end for
  $dependents \leftarrow javaClass.getDependents()$
  $N \leftarrow N \cup dependents - \{thisClass\}$
  $className \leftarrow N.getElement()$
  $javaClass \leftarrow readClassFile(className)$
  $thisClass \leftarrow javaClass.getThisClass()$
 end while
 return G
end Method

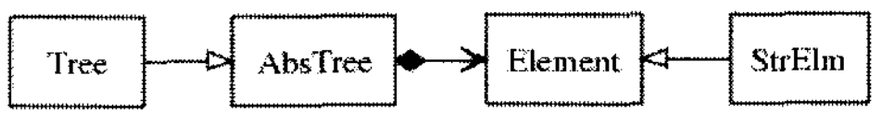

그림 10. 일반화된 집단화 관련성
Fig 10. Generalized Aggregation

집단화 관련성의 일반화를 얻기 위하여 일반화된 집단화 관련성(generalized aggregation relationship)을 〈정의 7〉과 같이 정의한다.

정의 7 일반화된 집단화 관련성. 다음 조건들 가운데 하나를 만족하면 클래스 C_i, C_j, C_k는 집단화된 일반화 관련성을 갖는다. C_i에서 C_j로의 일반화된 집단화 관련성을 $C_i \square\!\!\rightarrow C_j$ 로 표현한다.

(i) $C_i \diamond\!\!\rightarrow C_j$이면 $C_i \square\!\!\rightarrow C_j$이다.

(ii) $C_i \rhd C_k \wedge C_k \diamond\!\!\rightarrow C_j$이면 $C_i \square\!\!\rightarrow C_k \square\!\!\rightarrow C_j$

(iii) $C_i \diamond\!\!\rightarrow C_k \wedge C_j \rhd C_k$이면 $C_i \square\!\!\rightarrow C_k \square\!\!\rightarrow C_j \square$

일반화된 집단화 경로는 집단화 관련성을 갖는 클래스들의 연속으로 〈정의 8〉과 같다. 그리고 클래스의 일반화된 집단화 사이클(generalized aggregation cycle)은 원천과 종점이 같은 일반화된 집단화 경로이다. 이것은 〈정의 9〉와 같다.

정의 8 일반화된 집단화 경로. C에서 C'로의 일반화된 집단화 경로 $path(C,C')$는 다음과 같이 정의된다.

$$path(C,C') = \{[C_0,...,C_m] | m \geq 0 \wedge C_0 = C \wedge C_m = C' \wedge \forall i < m, C_i \square\!\!\rightarrow C_{i+1}\} \ \square$$

정의 9 일반화된 집단화 사이클. C에서 자신으로의 일반화된 집단화 사이클 $cycle(C)$는 다음과 같이 정의된다.

$$cycle(C) = \{[C_0,...,C_m] | m \geq 0 \wedge C_0 = C \wedge C_m = C \wedge \forall i < m, C_i \square\!\!\rightarrow C_{i+1}\} \ \square$$

〈그림 11〉의 일반화된 집단화 관련성을 〈그림 10〉으로부터 얻을 수 있다. 이 관련성의 일반화된 집단화 경로는 (Tree, AbsTree, Element, StrElm)이다.

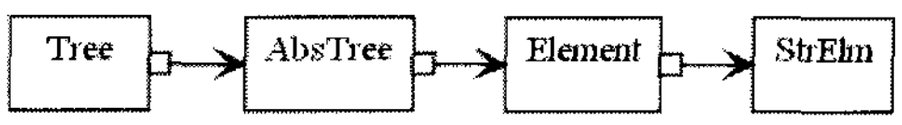

그림 11. 그림 10의 일반화된 집단화 경로
Fig 11. Generalized Aggregation Path of Fig.10

〈알고리즘 4〉는 〈정의 7〉에 따라 일반화 그래프 $G_1 = (V, E_1)$과 집단화 그래프 $G_2 = (V, E_2)$를 사용하여 일반화된 집단화 경로 $P = (V, E_p)$를 생성한다. 〈정의 7〉의 (i) 항목을 만족하는 클래스들의 쌍을 얻기 위하여 집단화 그래프의 노드를 일반화된 집단화 그래프의 노드로 추가한다 ($E \leftarrow E_2$). 다음, (ii)를 만족하는 클래스의 쌍을 첫 번째 내부 forall 구조에서 얻는다. 첫 번째 내부 forall 구조는 (iii)을 만족하는 클래스의 쌍을 찾는다. 마지막으로 일반화된 집단화 경로 $P = G.genPath()$를 얻는다.

알고리즘 4. 일반화된 집단화 경로 생성 알고리즘
Algorithm 4. Algorithm to Generate Generalized Aggregation Path

```
Input : 일반화 그래프 G1 = (V, E1) 집단화 그래프 G2 = (V, E2)
Output: 일반화된 집단화 경로 P = (V, Ep)
Method generateGenailzedAggregationPath(G1, G2)
  E <-- E2 where G = (V, E)
  for all (u,v) ∈ E1 where u,v ∈ V do
    for all (v,w) ∈ E2 where v,w ∈ V do
      E ← E ∪ {(v,w),(u,v)}
    end for
    for all (w,v) ∈ E2 where v,w ∈ V do
      E ← E ∪ {(w,v),(v,u)}
    end for
  end for
  P = G.genPath()
  return P
end Method
```

V. 덩어리 만들기

이 절에서는 Ⅳ 절의 알고리즘들로부터 얻은 관련성을 사용하여 잎 덩어리와 되부름 덩어리를 찾는 방법을 설명한다.

5.1. 잎 덩어리 찾기

잎 클래스들은 어트리뷰트들의 원시 타입 또는 싸개 클래스이어서 다른 클래스를 일반화하거나 집단화하지 않는 클래스이다. 그리고 잎 클래스를 집단화하는 유일한 클래스를 잎 덩어리라고 부른다.

정의 10 잎 클래스. 클래스 $C, C_1, ..., C_n$이 주어졌을 때 만일,

$$\{C_i | 1 \leq i \leq n \wedge (C \rhd C_i \vee C \diamond\!\!\rightarrow C_i)\} = \varnothing \wedge |\{C_i | 1 \leq i \leq n \wedge C_i \diamond\!\!\rightarrow C\}| = 1$$

를 만족하면 클래스 C는 잎 클래스이다. $\square$

정의 11 잎 모으개. 잎 클래스 C_l의 잎 모으개는 $C \diamond\!\!\rightarrow C_l$인 클래스 C이다. $\square$

〈알고리즘 5〉는 일반화 그래프와 집단화 그래프로부터 잎 클래스를 찾는다. 이 알고리즘에서 메서드 $\in \text{degree}()$와

$outdegree()$는 주어진 노드의 진입차수와 진출차수를 각각 돌려준다. $\in \mathrm{degree}(u)=0$이면 u로의 관련성이 없다는 것이며, $outdegree(u)=0$이면 u로부터의 관련성이 없음을 나타낸다.

알고리즘 5. 잎 클래스 찾기 알고리즘
Algorithm 5. Algorithm to Find Leaf Classes

```
Input : 일반화 그래프 G1 = (V, E1) 집단화 그래프 G2 = (V, E2)
Output: 잎 클래스의 집합 L
Method findLeaveClasses(G1, G2)
  L ← ∅
  for all u ∈ V do
    if G1.outdegree(u) = 0 ∧ G2.outdegree(u) = 0
       ∧ G2.indegree(u) = 1 then
      L ← L ∪ {u}
    end if
  end for
  return L
end Method
```

〈알고리즘 6〉은 잎 클래스를 집단화하는 모으개를 찾으며 〈알고리즘 7〉은 잎 모으개 a와 a가 집단화하는 잎 클래스들의 집합 $\{l_1,...,l_n\}$의 쌍인 $(a,\{l_1,...,l_n\})$으로 구성되는 잎 덩어리를 찾는다. 내부 forall 구조에서 잎 클래스들의 집합 $\{l_1,...,l_n\}$을 모으고, 내부 forall 구조 다음 문장에서 잎 덩어리를 구성한다. 잎 덩어리는 〈알고리즘 9〉에서 되부름 덩어리를 찾기 위하여 사용된다.

알고리즘 6. 잎 모으개 찾기 알고리즘
Algorithm 6. Algorithm to Find Leaf Aggregators

```
Input : 집단화 그래프 G = (V, E), 잎 클래스의 집합 L
Output: 잎 모으개의 집합 A
Method findLeaveAggregators(G, L)
  A ← ∅
  for all v ∈ L do
    if (u, v) ∈ E then A ← A ∪ {u} end if
  end for
  return A
end Method
```

5.2. 되부름 덩어리 찾기

되부름 덩어리는 되부름 타입과 되부름 타입이 모으개인 잎 덩어리로 구성된다. 클래스가 일반화된 집단화 사이클을 갖는다면 클래스는 되부름이다. 되부름 타입은 동일한 일반화된 집단화 사이클을 갖는 클래스의 모임이다.

알고리즘 7. 잎 덩어리 찾기 알고리즘
Algorithm 7. Algorithm to Find Leaf Clusters

```
Input : 집단화 그래프 G = (V, E), 잎 모으개의 집합 A,
        잎 클래스의 집합 L
Output: 잎 덩어리의 집합 LC
Method findLeaveClusters(G, A, L)
  LC ← ∅
  for all a ∈ A do
    C ← ∅
    for all (a, l) ∈ E do
      C ← C ∪ {l}
    end for
    LC ← LC ∪ {(a, C)}
  end for
  return LC
end Method
```

정의 12 되부름 타입. 되부름 타입은 $\forall C_i, C_j, 1 \le i,j \le n \wedge cycle(C_i) = cycle(C_j)$인 클래스의 집합 $\{C_i,...,C_n\}$이다. □

알고리즘 8. 되부름 타입 찾기 알고리즘
Algorithm 8. Algorithm to Find Recursive Types

```
Input : 일반화된 집단화 경로 P = (V, E)
Output: 되부름 타입의 집합 RT
Method findRecursiveTypes(P)
  RT ← ∅
  for all (u, u) ∈ E do
    T ← {u}
    for all (u, v) ∈ E ∧ cycle(u) = cycle(v) do
      T ← T ∪ {v}
    end for
    RT ← RT ∪ {T}
  end for
  return RT
end Method
```

일반화된 집단화 경로를 사용하여, 〈알고리즘 8〉은 되부름 타입의 집합 $R=\{T_1,...,T_n\}$을 돌려준다. $\forall T_i, T_j \in R$에 대하여 $T_i \cap T_j = \varnothing$이고, $\forall j,k \le m, cycle(C_i^j) = cycle(C_i^k)$에 대하여 $T_i = \{C_i^1,...,C_i^m\}$이다. 즉, 되부름 타입은 어떠한 클래스도 공유하지 않고 되부름 타입의 모든 클래스는 동일한 사이클을 갖는다.

〈알고리즘 9〉는 되부름 타입의 집합과 잎 덩어리의 집합을 사용하여 되부름 덩어리의 집합을 찾는다. 되부름 덩어리는 $(R,\{LC_1,...,LC_n\})$의 형태이다. 여기에서 R은 되부름 타입이고 $LC_1,...,LC_n$은 모으개가 R의 클래스인 잎 덩어리이다 (즉, $\forall 1 \le i \le n, LC_i = (a,\{l_i^1,...,l_i^m\}) \wedge a_i \in RT$).

알고리즘 9. 되부름 덩어리 찾기 알고리즘
Algorithm 9. Algorithm to Find Recursive Clusters

Input : 되부름 타입의 집합 RT, 잎 덩어리의 집합 LC
Output: 되부름 덩어리의 집합 RC
Method findRecursiveClusters(RT, LC)
  $RC \leftarrow \varnothing$
  for all $R \in RT$ **do**
    $rc \leftarrow \varnothing$
    for all $lc = (a, \{l_1, \cdots, l_n\}) \in LC$ **do**
      if $a \in R$ **then** $rc \leftarrow rc \cup \{lc\}$ **end if**
    end for
    $RC \leftarrow RC \cup \{(R, rc)\}$
  end for
  return RC
end Method

VI. 실험과 예

jCAT은 본 논문에서 제시한 알고리즘을 구현한 자바 클래스 관련성 추출 시스템으로 Eclipse Europa[14]에서 자바로 구현하였다. jCAT의 실행 화면은 〈그림 12〉와 같다. 〈그림 12〉의 왼쪽 화면은 자바 원시 프로그램이며 오른쪽 화면은 왼쪽 화면 프로그램을 시작 클래스로 하는 클래스 파일들의 관련성을 분석한 결과를 텍스트로 보여준다.

〈그림 15〉와 〈그림 18〉은 자바 프로그램(〈그림 14〉와 〈그림 17〉)[1,3,5]의 클래스와 덩어리들의 구조와 관련성을 본 논문에서 구현한 시스템이 생성한 DOT 파일을 Graphviz[8]를 사용하여 얻은 것이다. 〈그림 16〉와 〈그림 19〉은 결과 XML 문서이다.

〈그림 20〉은 GUI 이진 검색 트리 프로그램[1,5]의 결과이다. 원시 프로그램과 결과 XML 문서는 길이가 너무 길어 생략한다.

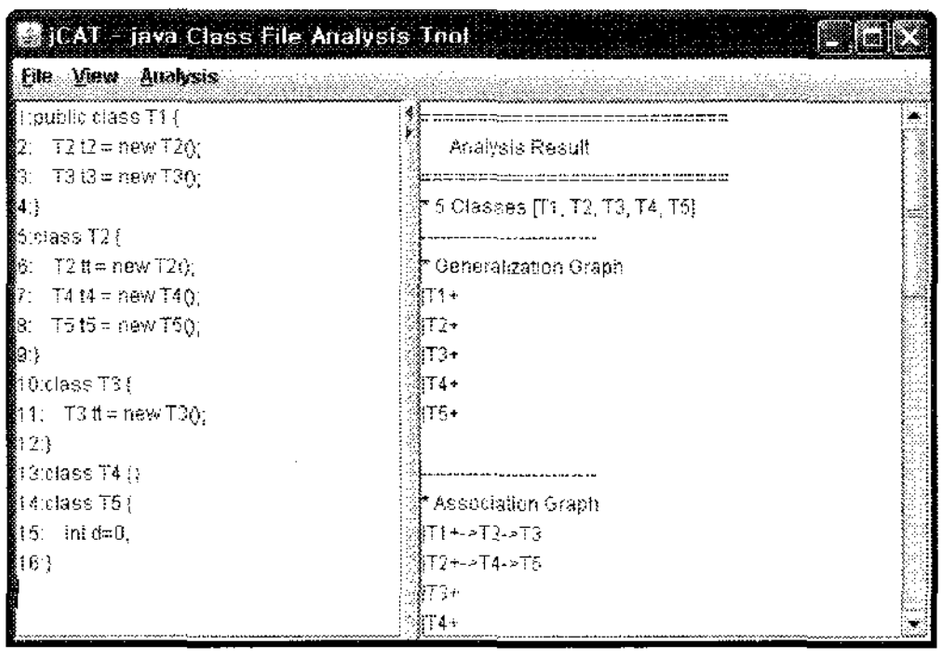

그림 12. jCAT 실행 화면
Fig 12. Screen Shot of jCAT

〈그림 13〉은 결과 XML 문서를 위한 DTD이다.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT RESULT
(CLASSES, RELATIONSHIPS, CLUSTERS)>
<!ELEMENT RELATIONSHIPS
(GENERALIZATION, ASSOCIATION,
AGGREGATION)>
<!ELEMENT RECURSIVE_TYPES (TYPE+)>
<!ELEMENT RECURSIVE_CLUSTERS
(RECURSIVE_CLUSTER+)>
<!ELEMENT RECURSIVE_CLUSTER
(RECURSIVE_TYPES,
RECURSIVE_AGGREGATORS)>
<!ELEMENT RECURSIVE_AGGREGATORS
(AGGREGATOR, LEAF+)*>
<!ELEMENT LEAF_CLUSTERS (LEAF_CLUSTER)*>
<!ELEMENT LEAF_CLUSTER
(AGGREGATOR, LEAF+)*>
<!ELEMENT LEAF (#PCDATA)>
<!ELEMENT GENERALIZATION (FROM, TO+)*>
<!ELEMENT CLUSTERS
(LEAF_CLUSTERS, RECURSIVE_CLUSTERS)>
<!ELEMENT CLASSES ((CLASS+))>
<!ELEMENT CLASS (#PCDATA)>
<!ELEMENT ASSOCIATION (FROM, TO+)*>
<!ELEMENT AGGREGATOR (#PCDATA)>
<!ELEMENT AGGREGATION (FROM, TO+)*>
<!ELEMENT TYPE (#PCDATA)>
<!ELEMENT FROM (#PCDATA)>
<!ELEMENT TO (#PCDATA)>
```

그림 13. 결과 XML 문서의 DTD
Fig 13. DTD for Result XML Document

```
public class BST {
        public static void main (String args[]) {
                int i;    Tree tr;
                tr = new Tree(100);
        tr.insert(50);    tr.insert(150);    tr.insert(25);
                }
                }
                class Tree {
public Tree(int n) { value.i = n; left = null; right = null;};
                public void insert(int n) {
                if (value.i == n) return;
                if (value.i < n)
                if (right == null) right = new Tree(n);
                else right.insert(n);
        else if (left == null) left = new Tree(n);
                else left.insert(n);
                }
        protected Data value;
        protected Tree left;
        protected Tree right;
                }
        public class Data {
                static int i;
                }
```

그림 14. 이진 검색 트리 프로그램
Fig 14. Binary Search Tree Program

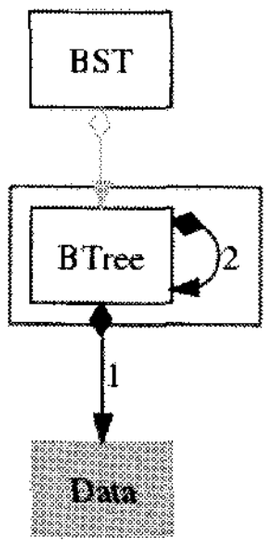

그림 15. 이진 검색 트리 프로그램의 클래스 구조와 관련성
Fig 15. Classes and Relationship of Binary Search Tree Program

```
<RESULT>
  <CLASSES>
                    <CLASS>BST</CLASS>
                   <CLASS>BTree</CLASS>
                   <CLASS>Data</CLASS>
                     </CLASSES>
                    <RELATIONSHIPS>
                    <GENERALIZATION />
                      <ASSOCIATION>
                 <FROM>BST</FROM><TO>BTree</TO>
                      </ASSOCIATION>
                      <AGGREGATION>
                 <FROM>BST</FROM><TO>BTree</TO>
          <FROM>BTree</FROM><TO>BTree</TO><TO>Data</TO>
                      </AGGREGATION>
                     </RELATIONSHIPS>
                      <CLUSTERS>
                    <LEAF_CLUSTERS>
                     <LEAF_CLUSTER>
                <AGGREGATOR>BTree</AGGREGATOR>
                     <LEAF>Data</LEAF>
                     </LEAF_CLUSTER>
                    </LEAF_CLUSTERS>
                  <RECURSIVE_CLUSTERS>
                   <RECURSIVE_CLUSTER>
              <RECURSIVE_TYPES><TYPE>BTree</TYPE>
                   </RECURSIVE_TYPES>
                 <RECURSIVE_AGGREGATORS>
                <AGGREGATOR>BTree</AGGREGATOR>
                     <LEAF>Data</LEAF>
                 </RECURSIVE_AGGREGATORS>
                   </RECURSIVE_CLUSTER>
                  </RECURSIVE_CLUSTERS>
                    </CLUSTERS>
                    </RESULT>
```

그림 16. 이진 검색 트리의 결과 XML 문서
Fig 16. Resulting XML Document of Binary Search Tree

```
public class T1 {
  T2 t2 = new T2();
  T3 t3 = new T3();
      }
   class T2 {
  T2 tt = new T2();
  T4 t4 = new T4();
  T5 t5 = new T5();
      }
  class T3 {
 T3 tt = new T3();
      }
 class T4 {}
 class T5 {
    int d=0;
      }
```

그림 17. 되부름 덩어리를 갖는 프로그램
Fig 17. Program with Recursive Clusters

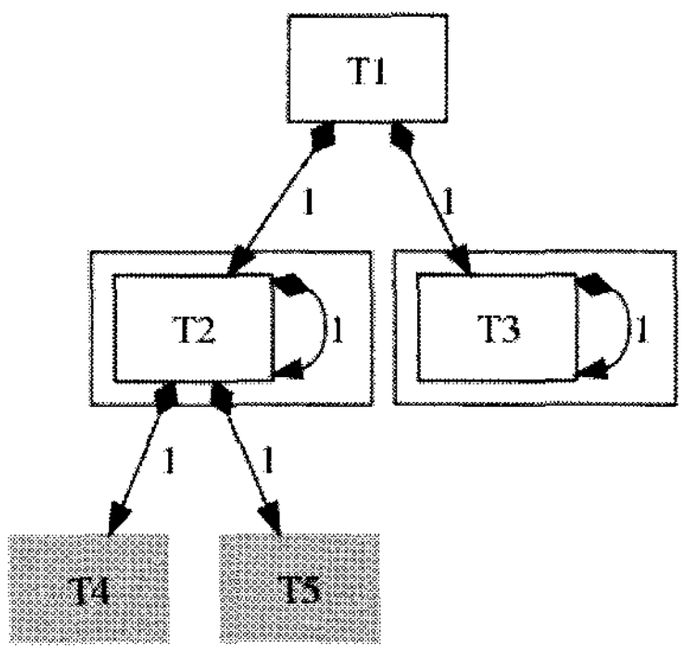

그림 18. 그림 17의 클래스 구조와 관련성
Fig. 18. Classes and Relationship of Fig 17

```
<RESULT>
<CLASSES>
<CLASS>T1</CLASS><CLASS>T2</CLASS>
<CLASS>T3</CLASS><CLASS>T4</CLASS>
<CLASS>T5</CLASS>
</CLASSES>
<RELATIONSHIPS>
<GENERALIZATION />
<ASSOCIATION>
<FROM>T1</FROM><TO>T2</TO><TO>T3</TO>
<FROM>T2</FROM><TO>T4</TO><TO>T5</TO>
</ASSOCIATION>
<AGGREGATION>
<FROM>T1</FROM><TO>T2</TO><TO>T3</TO>
<FROM>T2</FROM><TO>T2</TO><TO>T4</TO><TO>
T5</TO>
<FROM>T3</FROM><TO>T3</TO>
</AGGREGATION>
</RELATIONSHIPS>
```

```
<CLUSTERS>
<LEAF_CLUSTERS>
<LEAF_CLUSTER>
<AGGREGATOR>T2</AGGREGATOR>
<LEAF>T4</LEAF><LEAF>T5</LEAF>
</LEAF_CLUSTER>
</LEAF_CLUSTERS>
<RECURSIVE_CLUSTERS>
<RECURSIVE_CLUSTER>
<RECURSIVE_TYPES>
<TYPE>T3</TYPE></RECURSIVE_TYPES>
<RECURSIVE_AGGREGATORS />
</RECURSIVE_CLUSTER>
<RECURSIVE_CLUSTER>
<RECURSIVE_TYPES><TYPE>T2</TYPE>
</RECURSIVE_TYPES>
<RECURSIVE_AGGREGATORS>
<AGGREGATOR>T2</AGGREGATOR>
<LEAF>T4</LEAF><LEAF>T5</LEAF>
</RECURSIVE_AGGREGATORS>
</RECURSIVE_CLUSTER>
</RECURSIVE_CLUSTERS>
</CLUSTERS>
</RESULT>
```

그림 19. 그림 17의 결과 XML 문서
Fig 19. Resulting XML Document of Fig 17

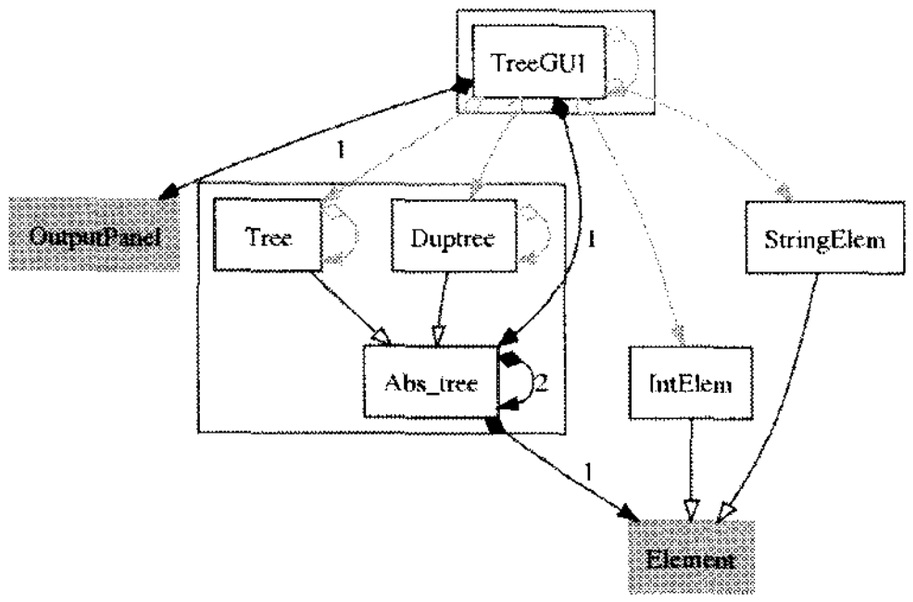

그림 20. TreeGUI 프로그램의 구조와 관련성
Fig 20. Resulting Graph of TreeGUI Program

VII. 결론 및 향후 계획

본 연구에서 좀 더 나은 객체 그래프를 그리기 위하여 자바 클래스 파일로부터 잎 덩어리와 되부름 덩어리를 찾기 위한 알고리즘을 제시하고 구현하였다. 본 논문의 알고리즘은 [1,4]의 정의를 개선한 집단화 관련성 정의에 기반하고 있다. 본 연구에서 정의한 명시적 집단화 관련성과 묵시적 집단화 관련성을 통하여 실행시간에 나타날 수 있는 객체에 가까운 클래스 관련성을 얻을 수 있다.

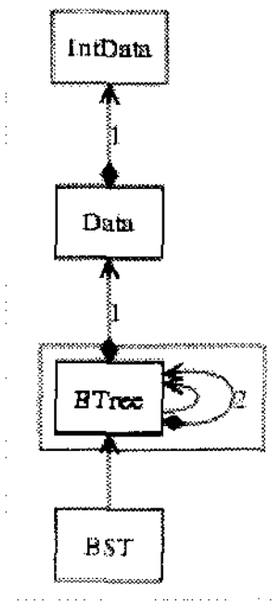

그림 21. 변형된 이진 검색 트리 프로그램
Fig 21. Modified Binary Search Tree Program

〈그림 21〉의 그래프는 변형된 이진 검색 트리 프로그램의 클래스 다이어그램이다. 클래스 BST는 Data를 집단화하고 Data는 IntData를 집단화한다. BTree, Data, IntData를 하나의 덩어리를 모으는 것이 합리적이다. 이러한 덩어리를 구성하려면 많은 프로그램의 실행을 관측하여 모으개로부터 잎까지의 적절한 거리를 결정하는 것이 필요하다.

현재 실행시간에 나타나는 자바 객체의 모양이 선형 구조인지, 트리 구조인지, DAG인지, 사이클 그래프인지를 예측하기 위하여 자바 바이트코드를 분석하는 연구를 진행하고 있다. 이러한 정보를 실행시간 전에 얻는다면 이 정보는 객체 그래프 그리기를 향상하기 위한 그리기 알고리즘을 선택하는데 결정적인 역할을 하게 될 것이다.

참고문헌

[1] P. Gestwicki, Interactive Visualization of Object-Oriented Languages, Ph.D. Thesis, Dept. of Computer Science and Engineering, SUNY at Buffalo, 2005.

[2] http://www.cse.buffalo.edu/LRG/JIVE

[3] J. K. Czyz and B. Jayaraman, "Declarative and Visual Debugging in Eclipse", Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange, pp.31-35, 2007.

[4] G. D. Battisa, P. Eades, R. Tamassia, and I. G. Tollis, Graph Drawing : Algorithms for the Visualization of Graphs, Prentice Hall, 1999.

[5] P. Gestwicki, B. Jayaraman, and A. Garg, From class diagram to object diagrams : A systematic approach, Technical Report CSETR-2004-21, Dept. of Computer Science and Engineering, SUNY at Buffalo, 2004.

[6] Y. -G. Guéhéneuc and H. Albin-Amiot, Recovering binary class relationships: Putting icing on the UML cake, Proceedings of the 19th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '04), 2004.

[7] Y. -G. Guéhéneuc, H. Albin-Amiot, R. Douence, and P. Cointe, Bridging the gap between modeling and programming languages, Technical Report, École des Mines de Nantes, 2002..

[8] http://www.graphviz.org

[9] D. Jackson and A. Waingold, Lightweight extraction of obejct models from bytecode, Proceedings of International Conference on Software Engineering, pp.194-202, 1999.

[10] M. Gogolla and R. Kollman, Redocumentation of Java with UML class diagrams, Proceedings of the 7th Reengineering Forum, 2000.

[11] L. A. Barowski and J. H. Cross II, Extraction and use of class dependency information for Java, Proc.eedings of the 9th Working Conference on Reverse Engineering, pp. 309-315, 2002.

[12] M. Keschenau, Reverse engineering of UML specifications from Java programs, Proceedings of the 19th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '04), pp.326-327, 2004.

[13] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 2nd edition, 1999.

[14] http://www.eclipse.org/

저 자 소 개

양 창 모

1997년 8월 : 인하대학교 컴퓨터공학과 박사

1998년 3월 ~ 현재: 청주교육대학교 부교수

〈관심분야〉 프로그래밍 언어, 정보 보안, 컴퓨터교육