論文
4-4-23

JML : Java 프로그램의 정보 표현을 위한 XML 분석 (JML : XML Analysis for Information Representation of Java Program)

장 근 실(Geun-Sil Jang)[1] 유 철 중(Cheol-Jung Yoo)[2] 장 옥 배(Ok-Bae Chang)[3]

요 약

XML은 그 자체가 갖는 장점으로 인해 많은 분야에서 정보를 기술할 수 있는 메타언어로서 널리 이용되며 웹 상에서 정보 표현의 표준언어로 널리 인정되고 있다. 본 논문에서는 Java로 작성된 원시 프로그램을 분석하고 재사용하고 유지보수하는 등의 일련의 소프트웨어공학 활동에서 발생할 수 있는 많은 어려운 사항들에 대하여 논하고, 기존의 문서화와 관련된 연구들이 이러한 사항들을 효과적으로 지원하는지 여부를 기술하며, 이러한 사항들을 효과적으로 지원하는 XML 기반의 JML(Java Markup Language)을 제안하고, 이를 이용하여 실제 프로그램을 분석하고 적용한 결과를 제시한다. JML은 Java로 작성된 소프트웨어의 정보를 기술할 수 있는 DTD이다. 또한 JML을 구성하는 각 요소들의 의미를 자세하게 기술하고, 각 요소들을 이용하여 Java로 작성된 소프트웨어의 정보들을 표현하는 방법들에 대해 설명한다.

ABSTRACT

Using the self-contained characteristics of XML, in various areas the XML is widely used to a meta language which can describe information and is recognized to a standard language to present information on the web.

In this paper, we point out many difficult problems when we are going to a serial activity of software engineering like that we analyze, reuse, and maintenance source programs. Then we describe whether the existed studies about documentation provide the solutions for the problems above mentioned or not. Finally, we propose the JML(Java Markup Language) that can effectively support solutions about these difficulties. Then we analyze the sample program and present the generated JML docuement using the results. explain the result that The JML is an XML DTD to describe software information written by Java language.

Also, we describe the meaning of elements that are parts of whole JML, and explain how to represent the information of Java source codes using each element.

1) 정회원 : 광양보건대학 컴퓨터정보과 조교수
2) 정회원 : 전북대학교 자연과학대학 컴퓨터과학과 조교수
3) 정회원 : 전북대학교 공과대학 전자,정보공학부 교수

논문심사 : 2003. 7. 4.
심사완료 : 2003. 7. 18.

1. 서론

소프트웨어 시스템이 점차 복잡해지고 대형화되고 있는 가운데 하드웨어의 급속한 발전 추세를 따라가지 못하는 소프트웨어의 진보는 소프트웨어 개발자나 관리자 및 공학자 등을 비롯한 모든 관련인들에게 커다란 부담이 되고 있다[1, 2]. 또한 하드웨어의 이런 변화에 따른 개발환경의 변화는 기존의 고정된 사무실 환경에서 네트워크와 인터넷과 같은 개별적이고 분산화된 환경 및 이와 더불어 파생되는 다양한 문제점들을 유발하고 있다. 특히 개발팀원들이나 시스템 분석가나 유지보수자 등의 측면에서 정보 공유의 어려움은 별 다른 해결책이 없는 매우 중요한 문제로 인식되고 있다. 최근의 연구 및 통계에 의하면 전체 프로젝트 비용의 60%에서 80%사이의 비용이 유지보수에 소비되는 것으로 알려져 있다. 이와 같은 소프트웨어 유지보수에는 시스템이나 프로그램의 이해와 프로그램의 변경 및 이로 인한 파급효과 등이 포함된다[3, 4, 5].

본 논문에서는 객체지향 언어로서 폭넓은 지지를 받고 있는 프로그램 언어인 Java를 구현언어로 하는 소프트웨어나 프로젝트에 있어서 효과적이고 효율적인 정보 공유 및 제공을 위한 방법을 XML(eXtended Markup Language)을 기반으로 연구한다. XML은 HTML (Hyper Text Markup Language)이 갖는 장점들은 수용하고 SGML (Standard Generalized Markup Language)의 문제점인 학습 및 이용의 어려움을 해결한 마크업 언어로 사용자 정의 태그와 구조적인 정보의 표현 및 확장성이 용이하며, 이런 장점들로 인해 많은 분야에서 정보를 기술할 수 있는 메타언어로서 널리 이용되고 있으며 웹상에서 정보 표현의 표준언어로 널리 인정되고 있다[2, 3, 4].

본 논문에서 제안하는 JML(Java Markup Language)은 Java로 작성된 소프트웨어의 정보를 기술하기 위한 다양한 정보들을 정의하는 XML DTD(Document Type Definition)이다. 원시 코드를 분석하고 재사용하고 유지보수하는 등의 일련의 소프트웨어 공학활동에서 발생할 수 있는 많은 어려운 사항들을 지적하고, 기존의 문서화와 관련된 연구들이 이런 사항들을 효과적으로 지원하는지 여부를 기술하며, 이러한 사항들을 효과적으로 지원하는 정보들로 이루어진다. JML을 제안하기 위해 본 연구에서 유념한 부분은 크게 3가지로 나뉘어 진다. 첫째는 원시 코드가 존재하지 않는 경우 이용자들에게 최대한의 정보를 제공할 수 있는 부분이고, 둘째는 생성된 JML 문서가 존재하는 경우 이 문서로부터 해당 Java 원시 코드의 뼈대코드를 구축할 수 있는 정보를 제공하는 부분이며, 마지막 세 번째는 JML 문서를 생성하는 데 있어서 해당 Java 원시 코드가 존재할 때 문서화에 소비되는 시간과 비용을 절감하기 위해 문서 구축자의 개입을 최소화할 수 있는 JML 문서 자동 생성에 대한 부분이다. 이와 같은 사항들로 인해 제안된 JML은 원시 코드에 나타나는 정보들이 포함되어 있다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 본 연구에서 적용한 DTD를 비롯한 유사한 관련연구들을 기술하고, 비교한다. 3장에서는 원시 코드를 이해할 때 어려움을 주는 사항들을 설명하고, 이를 해결하기 위해 필요한 정보들을 제안하며, 이를 기반으로 Java 원시 코드의 각 모듈에 대해서 다양한 소프트웨어 공학 활동에 필요한 정보들을 제안하고, 이로부터 JML을 설계한다. 4장에서는 제안된 JML을 JDK 1.3 버전에 포함된 예제 프로그램을 대상으로 적용하고, 그 결과를 이용하여 원시 코드의 이해에 필요한 정보들을 제시한다. 마지막으로 5장에서는 결론 및 앞으로의 연구 분야에 대해서 언급한다.

2. 관련연구

2.1 DTD

DTD는 SGML DTD를 간소화한 버전으로 특정 범주를 지원하는 사용자 정의 태그와 문서의 구조 정보를 표현할 수 있는 정의[6, 7, 8]를 의미하며 정의하는 구문의 이용 목적이나 대상에 의해 속성 기반(Attribute-Centric)의 DTD와 요소 기반(Element-Centric)의 DTD로 나누어 볼 수 있다[9]. 속성 기반 DTD는 이용자 측면에서 효과적인 가독성을 제공하는 반면에 응용프로그램 기반의 적용(처리, 조회)은 어려운 특징이 있고, 반대로 요소 기반 DTD는 가독성은 낮지만 프로그램 적용이 용이한 특징이 있다. 본 연구에서는 데이터 수집과 가공 및 처리에 용이하도록 JML에 요소 기반 DTD를 적용하였다.

2.2 UXF와 기타 Markup Language

XML을 이용한 정보 제공 및 공유 방법에 대한 연구는 다양한 응용 분야를 중심으로 연구가 되어 왔고, 그 중에는 실제로 이용되고 있는 분야도 있다.

문헌 [10]의 UXF(UML eXchange Format)는 UML(Unified Modeling Language)을 표현할 수 있도록 개발한 XML의 서브셋으로 분산된 장소에서의 정보 공유 및 UML 문서에 포함되어 있는 논리적인 정보를 표현할 수 있도록 개발되었다. 현재 UML의 의미정보만을 표현할 수 있고, 다이어그램의 위치 정보나 도형 정보는 표현할 수 없다.

이 외에도 화학식을 표현하도록 개발된 CML (Chemical Markup Language), 수학식을 지원할 수 있도록 개발된 MathML(Mathematical Markup Language)[11], 푸쉬 기술(Push Technology)에 적용되는 CDF(Channel Data Format), 경영 및 재정과 관련된 정보를 표현할 수 있도록 개발된 OFX(Open Financial eXchange)[12] 등이 있다.

2.3 Javadoc 및 Doclet

원시 코드를 기반으로 주석의 내용을 문서화하는 연구물들은 다양한 형식의 문서 포맷들(Text 형식, Postscript 형식, Unix의 매뉴얼 형식 및 Windows 계열의 도움말 형식, HTML 등)을 지원하고 있지만, 웹과 같은 분산된 환경에서는 인터넷에서의 이용가능성에 중심을 두어야 한다. 인터넷 환경을 고려하면 문서의 형식은 웹에서의 HTML과 근래의 인터넷 표준 언어로 인정받는 XML 형식을 들 수 있다. 대부분의 기존 연구물들은 코드가 완성된 후 사용자나 개발자 측면에서 개발에 이용할 수 있는 API 형식의 문서를 HTML 형식으로 제공하는 것이 대부분이다. 기존의 연구물들 중에서 가장 널리 사용되는 것은 Java에 포함되어 배포되는 Javadoc와 문서화 API인 Doclet를 들 수 있다. Javadoc는 실행파일 형태로 제공되며, 생성된 원시 코드 내에 포함되어 있는 주석정보를 기반으로 메소드와 클래스에 대한 프로그램 코딩 측면의 API를 HTML 형식의 문서로 생성한다[13]. Doclet는 HTML 이외의 다른 형식의 문서를 생성할 때 이용할 수 있는 API이다[14].

이들 두 가지는 이용관점이나 시각에 따라 동일하게 간주되는데, 그 이유는 Javadoc를 이용하여 API HTML 문서를 생성할 때 명령행 인수 -doclet를 생략하면 "standard" Doclet이 적용된다. 만일 이용자(개발자나 최종사용자) 입장에서 다른 형식의 API를 생성할 때에는 각각 다른 프로그램을 작성해야 한다. 여기서 주목해야 할 것은 서로 다른 문서형식들이라 하더라도, 결국 내용은 HTML 문서와 동일한 API라는 데 있다. 즉, HTML 문서를 해당 포맷으로 변환한 것에 불과하다. 그리고 사용자 정의 태그(Javadoc가 인식하는 표준 태그 외의 태그들)를 지원하기 위해서는 Doclet 자체를 수정하여 자신만의 Doclet 파일을 생성해야 하는데, 이 과정에서 DTD를 이용할 수 없다. DTD는 XML의 가장 큰 장점인 정보의 구조와 항목의 의미를 지원하는데 반드시 필요한 필

수적인 요소이다. 결국, DTD가 존재하지 않는 상황에서 XML 형식의 문서를 생성한다는 것은 정보의 구조를 지원할 수 없음을 의미하며, 이와 같은 상황으로 인해 Javadoc를 이용하여 많은 양의 정보들이 다양한 계층구조 및 관계성으로 이루어져 있는 XML 문서를 생성하는 것은 제약사항이 많고, 개발자나 최종 사용자들에게 필요한 정보를 효과적으로 제공할 수 없다는 문제점을 갖게 된다. 〈표 1〉은 Javadoc 및 Doclet과 비교하여 본 연구가 갖는 특징들을 보여준다.

〈표 1〉 Javadoc/Doclet와 JML의 비교
〈Table 1〉 Compare of Javadoc/Doclet and JML

항목	차이점
문서의 내용 (Contents)	JML에 대한 본 연구는 API 형식의 문서와 API 형식 외에 논문에서 주장한 다양한 정보들을 복합적으로 포함한다.
문서의 포맷 (Format)	XML 형식의 문서를 만들기 위해 복잡한 프로그램 개발이 요구된다. 또한 정보의 구조를 나타내는 DTD를 적용시킬 수 없으므로 XML의 적용으로 인한 장점을 이용할 수 없다.
범용성 (Generality) 및 재사용성 (Reusability)	HTML 형식의 문서에 포함된 내용이 단순히 API 참조 매뉴얼과 같은 성격으로 HTML이 갖는 형식상의 제한사항으로 인해 문서들의 재사용 가능성 및 문서 정보의 구조 분석이나 내용의 가공에 어려움을 가지게 된다. 하지만, 본 연구에서 제안하는 XML 형식의 문서는 HTML의 어려운 사항들을 자체적으로 해결할 수 있으므로, 문서에 포함된 정보의 재사용이나 정보의 구조 분석 등이 용이하다.

3. Java 원시 코드의 모듈 정보 및 관련 DTD

3.1 원시 코드의 이해를 어렵게 하는 요인들

원시 코드를 이해하는 것은 역공학이나 재사용 및 유지보수와 같은 중요한 소프트웨어 공학 활동에 있어서 기본적이면서도 매우 중요한 과정이다[1, 2, 15]. 하지만, 다른 개발자에 의해 개발된 원시 코드나 자신이 개발한 원시 코드라 하더라도 원시 코드의 복잡한 내용과 구조를 완전하게 이해하는 과정은 매우 어렵고 많은 시간이 소비된다. 다음은 이와 같은 어려움의 원인이 되는 요인들이다.

·클래스 관계성(Class Relationship) : 대부분의 클래스는 다른 클래스들과의 관계에 의해 기능을 수행한다. 문헌 [16]를 살펴보면 클래스들 사이에는 일반화(Generalization), 특수화(Specialization), 집약(Aggregation), 복합(Composition) 및 연관(Association)과 같은 다양한 관계성이 존재한다. 일반화와 특수화 관계성은 Java에서 implements 구문에 해당하며, 상속관계에서는 부모 클래스와 자식 클래스를 의미한다. 또한 집약, 복합, 연관 관계성은 클래스 몸체 내에서 호출되거나 몸체를 구성하는 다른 클래스들의 객체 또는 멤버들로 사상된다. 이와 같은 상속 및 재사용으로 인한 클래스들 사이의 관계성이나 객체의 빈번한 호출은 코드 이해의 어려움을 가중시키는 중요한 요인이다.

- 사용자 정의 메소드(User Defined Method) : Java는 많은 메소드들을 제공하지만, 개발에 필요한 모든 메소드를 제공하지 못한다. 사용자들은 이와 같은 메소드들을 이용하여 필요한 기능을 수행하는 메소드를 정의하는데, 이와 같은 메소드를 사용자 정의 메소드라 한다. 클래스 내부에 존재하는 사용자 정의 메소드는 객체지향 개념이 제공하는 연산자 오버로딩이나 오버라이딩과 같은 기능들로 인해 레퍼런스 라이브러리에 이미 존재하는 메소드와 혼동을 유발할 수 있고, 이 외에도 그 자신만의 특수한 기능을 수행하기 때문에 자세한 정보가 필요하다.
- 메소드 오버라이딩(Method Overriding) 과 오버로딩(Overloading) : 메소드 오버라이딩은 부모클래스에서 구현된 메소드를 자식클래스에서 구현된 메소드로 대체한다는 의미로 메소드의 이름 및 매개 변수의 개수와 형이 동일한 메소드가 부모와 자식 클래스에 모두 존재하는 경우 자식 클래스에 정의된 메소드가 우선한다는 개념이다. 오버로딩은 메소드 이름이 동일한 상황에서 매개 변수의 개수와 형이

서로 다른 경우를 의미한다. 메소드 오버라이딩은 주로 상속관계에서 존재하며, 오버로딩은 사용자 정의 메소드를 중심으로 동일 클래스 내부에서 존재한다. 이와 같은 메소드 오버라이딩과 오버로딩은 코드를 이해하는데 어려움을 가중시키는 중요한 요인으로, 함수의 원형 정보 및 클래스들의 계층구조에 대한 정보를 이용하여 어려움을 감소할 수 있다.

- 위에서 언급한 요인들 외에도 제어문 내에 중첩된 제어문은 구조 자체의 이해를 어렵게 한다. 이와 같은 제어문은 해당 루틴에서 선택이나 반복과 같은 역할을 주로 수행하므로 주석의 형식을 빌어 역할의 내용을 간략하게 기술하면 이해하는데 많은 도움이 된다.

3.2 모듈별 정보

Java는 객체지향 언어이지만 다른 객체지향 언어인 C++와는 구분되는 다른 점들이 있다. 그 중에서 객체지향 측면에서 볼 때 가장 큰 차이점은 클래스의 존재 여부에 있다. C++는 기존의 절차지향 언어인 C에 객체지향 개념을 추가시킨 언어이므로 프로그램이나 프로젝트의 상황에 따라 클래스가 존재하지 않을 수도 있지만, Java는 반드시 하나 이상의 클래스를 포함해야 한다.

본 연구에서는 Java를 구성하는 주요 구성요소를 위와 같은 특징으로 인해 클래스와 클래스를 구성하는 요소인 데이터 멤버 및 메소드로 구분한다. 그리고 여기에 클래스들의 관계성을 추가한다. 기존의 Javadoc와 같은 문서화 도구는 Java 원시 코드 내부에 필요한 정보를 시스템에서 지원하는 특정 플래그를 이용하여 정의하였다. 그리고, 이렇게 플래그화된 정보를 필요에 따라서 관련 HTML 문서로 생성해준다. 더불어 근래에 제작되거나 배포되는 대부분의 소프트웨어 시스템들은 사용자 지침서나 관련 정보를 HTML 문서의 형식을 빌어 제공하고 있다. 하지만, 이와 같은 HTML 문서들의 경우는 서론 및 관련연구에서 이미 언급한 것처럼 제공자가 제공하는 단 하나의 형식으로만 정보를 공유할 수 있다. 표현하는 정보의 내용은 동일하지만, 레이아웃이나 디스플레이와 같이 형식을 달리하는 경우에는 전혀 새로운 문서를 만들어야만 하는 상황이 된다.

일반적으로 원시 코드 내에 기술되는 정보들은 주석의 형식을 빌어 제공되며, 주석으로 제공되는 정보들은 원시 코드에 표현되는 정보를 기술하는 커멘트 정보(Comment Information)와 원시 코드의 기반이 되는 알고리즘이나 모듈들 사이의 관계와 같은 논리적인 정보를 기술하는 어노테이션 정보(Annotation Information)로 나눌 수 있다. 커멘트 정보는 일반적으로 코드 상에서 구문분석기 등을 이용하여 추출할 수 있는 정보들이 대부분이며, 어노테이션 정보의 경우는 구문분석기로는 추출이 불가능한 정보로 실제로 원시 코드를 읽고 이해할 때 많은 시간을 소비하게 하는 요인들에 대한 정보라 할 수 있다[15].

〈표 2〉, 〈표 3〉, 〈표 4〉는 문헌 [14, 17, 18, 19]를 기반으로 문서화에 표현되어야 할 정보들을 클래스, 메소드, 데이터 멤버(필드) 모듈로 나누어 정리한 내용을 나타낸다. 이 내용들은 본 연구 및 기존의 다양한 연구물들을 통해 제안된 것으로, 각 모듈에 포함된 의미나, 목적, 제한사항, 사전/사후조건 등의 어노테이션 정보들은 코드의 이해나 재사용 및 유지보수와 같은 다양한 소프트웨어 공학 활동에 중요한 역할을 하는 것으로 보고되며, 인정되고 있다.

3.2.1 클래스 및 인터페이스 관련 정보

클래스는 Java 언어의 핵심 모듈로 관련된 데이터 멤버와 메소드를 포함한다. 인터페이스는 객체지향의 특성인 다중 상속을 지원하지 못하는 Java 언어의 특징을 해결하기 위한 메커니즘으로 인터페이스를 정의하는 과정은 클래스와 거의 유사하다. 각각의 모듈을 구현하는 과정에서 Java의 예약어인 class와 interface로 구분되기 때문에 본 연구에서는 클래스와 인터페이스를 동일한 모듈로 간주한다. 클래스의 목적은 클래스의 목적

이나 정의 또는 적용가능한 분야 등을 포함하며, 현재 클래스의 사용방법 등을 포함한다. 여기서 현재 클래스의 사용방법은 해당 클래스가 다른 클래스의 일부분이나 부품으로서 사용될 때를 기술하는 정보이다. 클래스의 정의에 포함된 정보들은 대부분 코드로부터 추출될 수 있는 커멘트 정보들로 이루어진다. 식별자는 추상 클래스인지 가상 클래스인지 등의 정보를 나타내는 해당 예약어들이 해당되고, 이는 클래스의 엑세스 모드로 세분화된다. 상속구조로는 그 대상이 클래스인가 인터페이스인가에 따라 예약어로서 구분이 가능하다.

Java API에 존재하는 인터페이스나 클래스와는 달리 개발자가 작성한 클래스의 경우 위에 나타나는 다양한 정보들은 해당 클래스의 이해와 재사용에서 이용자에게 큰 도움을 준다. 〈표 2〉는 클래스 모듈에 포함되어야 할 정보들을 보여주고 있다.

〈표 2〉 클래스/인터페이스 관련 정보
〈Table 2〉 Class/Interface Related Information

항목	세부사항
· 클래스의 목적	·목적, 정의, 유용한 부분 ·사용방법(서브클래싱, 인스턴싱) ·작업방식
· 클래스 정의	·식별자 : abstract, final, public, ... ·상속구조(부모클래스) ·인터페이스 상속구조
· 클래스 인스턴스화	·셋업과정을 포함하는 클래스의 인스턴스 방법 ·결과로 자동 생성되는 다른 클래스들
· 서브클래스화	·서브클래스화되는 시점 ·오버라이드 메소드
· 메모리 해제	·해제되어야 할 인스턴스에 대한 참조

3.2.2 메소드 정보

메소드에 대한 정보는 〈표 3〉에 기술되어 있다. 이들 중 유의해야 할 것들은 메소드들의 정의와 호출 및 오버라이딩에 대한 정보이다. 이는 여러 개의 클래스들과 그들의 다양한 객체들 사이의 관련성을 갖는 프로그램의 경우 메소드들 사이의 호출관계나 오버라이딩 또는 오버로딩과 같은 기능들이 사용자들에게 혼란을 줄 수 있기 때문이다. XML을 이용한 정보 공유의 목적이 코드의 이해에 필요한 정보를 제공하는 것이기 때문에 현재의 메소드와 호출이나 피호출 관계에 있는 다른 메소드들에 대한 정보들을 제시하는 것과 메소드와 관련된 인수의 유효범위나 반환값의 유효범위 등의 정보는 사용자들에게 많은 도움이 될 수 있다.

메소드의 정의부분은 원시 코드에 존재하는 예약어 중심의 정보들로 이루어진다. 클래스 메소드인지 객체 메소드인가를 구분하는 예약어 static과 메소드의 사용가능한 범주를 지시하는 public, protected 및 private 등의 식별자, 메소드의 시그너처(signature)에 해당하는 인수들의 개수 및 각각의 자료형 및 반환값의 형 또는 반환값 등의 정보를 포함하며, 이 외에도 상수로서의 사용유무를 나타내는 예약어 final이나 추상 메소드를 나타내는 정보를 포함한다. 또한 메소드의 기능이나 목적, 호출방법 및 각 인수의 유효범위나 기능, 반환값의 유효범위, 메소드의 부작용 등의 정보는 전제조건이나 제한사항 등으로 표현된다.

〈표 3〉 메소드 관련 정보
〈Table 3〉 Method Related Information

항목	세부사항
· 메소드의 목적	·적용분야, 사용법, 효과 ·반환값, 부작용 또는 두가지 모두에 대한 효과
· 메소드의 정의	·static (class method/instance method) ·처리모드(public/protected/friendly/private) ·인수들의 개수 및 형 ·반환값의 형 ·abstract (abstract/concrete) ·final (unredefinable/redefinable) ·synchronized ·native ·throws (exceptions)
· 메소드 호출	·메소드 호출방법 ·각 인수의 유효범위 ·반환값의 유효범위 ·메소드의 부작용
· 메소드 오버라이딩	·부모/자식 클래스에서의 오버라이딩

3.2.3 데이터 멤버(필드)

데이터 멤버에 대한 정보들은 〈표 4〉에 나타난다. 제어문의 경우 데이터 멤버나 변수의 값에 의해 제어의 흐름이 많은 영향을 받으므로 주석에 기술된 데이터 멤버나 기타 다른 변수들의 기능은 원시 코드 복잡한 제어구조를 이해하는데 큰 도움이 된다. 또한 개발자 입장에서 특히 중요하게 다루어야 할 부분은 특정 "블록" 이나 "행" 의 기능에 대한 부분이다. 특정 블록의 경우는 개발자의 알고리즘이나 쉽게 이해하기 어려운 코딩 테크닉 등이 나타나는 부분으로서 코드를 이해하려는 이용자들이나 심지어 개발자 자신까지도, 많은 시간이 흐른 뒤의 유지보수나 이해에서 어려움을 느낄 수 있기 때문에 서술적인 표현을 이용한 정보의 기술이 요구된다[4, 20, 21].

〈표 4〉의 필드 정의 부분에 나타나는 정보들 중에서 read/write는 필드의 사용 범주를 나타내는 항목이고, 클래스 변수인지 객체 변수인지를 나타내기 위하여 static 정보를 포함하고 있다. 필드의 형은 언어 차원에서 제공되는 원시 자료형과 사용자가 정의한 클래스로부터 생성되는 객체형으로 구분이 가능하다.

〈표 4〉 데이터 멤버 관련 정보
〈Table 4〉 Data Member Related Information

항목	세부사항
· 필드의 목적, 적용	·루틴에서의 필드의 역할 ·제어변수의 변경
· 필드 정의	·static (class variable / instance variable) ·필드의 형 ·final (constant / variable) ·transient (transient / persistent) ·read / write

3.3 JML 제안 및 설계

이 절에서는 위에서 열거한 모듈들 중에서 클래스 모듈과 프로젝트 모듈에 대한 정보들의 정의와 생성 규칙을 기술하고, 이에 대응하여 생성된 JML 내용을 기술한다.

대부분의 프로젝트들은 여러 개의 파일들로 구성되며, 이로 인하여 여러 개의 클래스들이 존재할 수 있다. 이러한 프로젝트들의 특성을 DTD의 계층구조 특성과 접목시키기 위해 본 논문에서 제안하는 JML은 최상위 요소로서 Project를 지정하고, Project의 자식 요소들로 클래스와 메소드 및 데이터 멤버를 정의한다. 하나의 프로젝트 상에서 복수개의 클래스들이 존재할 수 있고, 각각의 클래스들은 다시 복수개의 메소드와 데이터 멤버들을 포함할 수 있다. Project 모듈을 구성하는 규칙들은 다음의 규칙 1에 나타난다.

[규칙 1] Project 모듈의 구성

Project = Project_관련 정보, Import_구문*, Package 구문*, Class*, Method*, Field*
Project_관련 정보 = 프로젝트_이름?, 저작자 이름?, 버전_정보?, 날짜?, 전자우편?, 전화?
Import_구문 = 해당_원시_코드에_존재하는 Import_정보
Package_구문 = 해당_원시_코드에_존재하는_Package_정보
*: 해당 정보가 0번 이상 반복되어 나타날 수 있음.
+: 해당 정보가 1번 이상 반복되어 나타날 수 있음.
?: 해당 정보가 0 또는 1번 나타날 수 있음(즉, 생략가능함).
, : 각 요소들의 순서

위에서 정의된 규칙을 적용하여 생성된 JML의 DTD는 다음과 같다.

```
〈!ELEMENT Project (Name?, Author?, Version?, Date?, Email?, Tel?, Import*, Package*, Class*, Method*, Field*)〉
〈!ELEMENT Name (#PCDATA)〉
〈!ELEMENT Author (#PCDATA)〉
〈!ELEMENT Version (#PCDATA)〉
〈!ELEMENT Date (#PCDATA)〉
〈!ELEMENT Email (#PCDATA)〉
〈!ELEMENT Tel (#PCDATA)〉
〈!ELEMENT Import (#PCDATA)〉
〈!ELEMENT Package (#PCDATA)〉
```

문헌 [22]에 의하면, 클래스 선언(Class Declaration)이란 새로운 이름을 갖는 참조형을 명

시하는 것으로 설명하고 있으며, 다음과 같은 구문을 이용하여 클래스를 선언한다.

ClassDeclaration:

ClassModifiersopt class Identifier Superopt Interfacesopt ClassBody

위 구문에서 *ClassModifiers*는 클래스 식별자를 기술하는 부분으로 아래첨자 형식의 접미사 opt는 이 부분이 생략될 수 있음을 의미한다. class는 루틴이 클래스임을 암시하는 예약어이고, 그 다음에 나타나는 *Identifier*는 클래스의 이름을 지정한다. class위치에 예약어 interface를 기술하여 인터페이스를 선언할 수 있다. *Super*는 상속관계를 기술할 때 부모클래스의 이름을 명시하는 부분으로 예약어 extends를 이용한다. *Interfaces*는 다중상속을 지원하는 방법으로 인터페이스로부터 메소드나 필드들을 상속받을 때 해당 인터페이스를 명시하는 부분으로 예약어 implements를 이용하여 기술한다. 마지막으로 *ClassBody*부분은 클래스의 몸체를 기술하는 부분으로 다양한 메소드와 필드들을 선언하고 정의하는 역할을 한다.

이상에서 기술한 정보들을 기반으로 클래스 모듈이 포함해야할 정보들은 클래스의 이름과 클래스의 종류, 식별자의 범주, 부모 클래스의 이름과 현재 클래스를 부모 클래스로 하는 자식 클래스들의 이름, 관련성이 있는 인터페이스들의 이름과 이벤트 처리에 주로 이용되는 클래스 내부의 클래스인 내부 클래스(inner class)에 대한 정보, 클래스에 포함된 사용자 정의 메소드들의 정보와 필드들의 정보, 관련된 객체들의 정보와 static 요소들로 정의할 수 있다. 또한 어노테이션 정보에 포함되는 Purpose, Note, Precondition, Constraint, Exception 등을 포함할 수 있다.

위에 언급된 내용들을 기반으로 Class 모듈을 구성하는 규칙들은 다음의 규칙 2와 같다.

[규칙 2] Class 모듈의 구성

```
Class = Class+
Class = Class_이름, 클래스의_종류?, 식별자(Access_Mode)?, 부모_클래스?, 자식_클래스들*, 인터페이스_관련_정보*, 내부_클래스_관련_정보*, 메소드들의_정보*, 포함된_인스턴스들의_정보*, 필드들의_정보*, 정적멤버들의_정보*, 클래스_모듈의_목적/기능?, 이해에_도움이_되는_기타_정보들?, 클래스에_필요한_사전조건?, 클래스에_필요한_제한사항/제약사항들의_정보?, 예외사항에_대한_정보?
내부_클래스_관련_정보 = (클래스명, 클래스형식)*
```

또한, 규칙 2를 기반으로 생성된 Class 모듈에 대한 DTD와 DTD의 각 요소에 해당하는 항목들의 의미는 다음과 같이 정의된다.

```
<!ELEMENT Class (Name, Type?, Access?, Parent?, Childs*, Interfaces*, InnerClass*, Methods*, Instances*, Fields*, StaticMembers*, Purpose?, Note?, Precondition?, Constraint?, Exception?)+>
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Type (#PCDATA)>
  <!ELEMENT Access (#PCDATA)>
  <!ELEMENT InnerClass (Name, Type)*>
  <!ELEMENT Parent (#PCDATA)>
  <!ELEMENT Childs (#PCDATA)>
  <!ELEMENT Interfaces (#PCDATA)>
  <!ELEMENT Methods (#PCDATA)>
  <!ELEMENT Instances (#PCDATA)>
  <!ELEMENT Fields (#PCDATA)>
  <!ELEMENT StaticMembers (#PCDATA)>
  <!ELEMENT Purpose (#PCDATA)>
  <!ELEMENT Note (#PCDATA)>
  <!ELEMENT Precondition (#PCDATA)>
  <!ELEMENT Constraint (#PCDATA)>
  <!ELEMENT Exception (#PCDATA)>
```

문헌 [22]에 의하면, 메소드를 선언하는 구문은 다음과 같다.

① *MethodDeclaration:*
MethodHeader MethodBody
② *MethodHeader:*
MethodModifiersopt ResultType
MethodDeclarator Throwsopt
③ *ResultType:*
Type | void
④ *MethodDeclarator:*
Identifer (FormalParameterListopt)

위 구문에 의하면 메소드 선언부분은 ①에 해당한다. *MethodHeader*와 *MethodBody*로 구성된다. *MethodHeader*는 메소드의 호출관계나 메시지 전달에 필요한 정보들을 포함하는 부분으로 원형정보인 프로토타입을 지정하는 부분이다(②). 메소드 식별자(*MethodModifiers*) 와 반환형(ResultType) 및 메소드의 이름(*Method-Declarator*)로 구성된다. 메소드 식별자는 메소드들의 처리방법을 나타내는 부분으로 상속관계에 필요한 정보들을 기술한다. ③은 반환형에 대한 정보를 나타내는 부분으로 반환형이 없음을 의미하는 키워드 void와 다른 객체의 형이나 원시 자료형(Primitive Data Type: 정수, 실수, 문자형 등)을 기술하여 메소드의 반환값에 대한 정보를 지정한다. ④는 메소드 선언자로 메소드의 이름과 메소드의 기능을 수행하는 데 필요한 인수들의 목록들로 구성된다. 인수들의 목록은 데이터형과 변수명의 쌍으로 구성된다. 메소드 바디는 메소드의 실제적인 처리과정을 기술하는 부분이다. 위에 언급한 정보들과 3.2.2절에 기술한 정보를 기반으로 정의한 규칙은 다음의 규칙 3과 같다.

[규칙 3] Method 모듈의 구성

```
Method = Method+
Method = 메소드_이름, 메소드_형식, 메소드_반환형?,
         Access_Mode?, Static Mode?, 메소드_인수 정
         보?, 다형성_정보?, 메소드를 포함하는_클래스
         명*, 메소드_초기치*, 메소드에_포함된 멤버변
         수_정보*, 메시지 정보*, 이해에_도움이 되는_
         기타_정보들?, 메소드에_필요한 사전조건?, 메
         소드에_필요한_제한사항/제약사항들의 정보?,
         예외사항에_대한_정보?
메소드_인수_정보 = (인수형, 인수명)*
다형성 정보 = (형식, 해당_클래스)*
메소드_초기치 = (메소드명, 메소드초기값)
메소드_멤버 = (멤버형*, 멤버명*, 멤버값*)*
메시지_정보 = (송신자*, 수신자*)
  송신자|수신자 = (관련_클래스명, 관련_메소드명)
```

또한, 규칙 3을 기반으로 생성된 Method 모듈에 대한 DTD와 각 요소에 해당하는 항목들은 다음과 같은 구문으로 정의된다.

```
<!ELEMENT Method (Name, Type, ReturnValue?,
Access?, Static?, Parameter?, Polymorphism?,
IncludeClass*, MethodInitValue*, Members*,
Messages*, Purpose?, Note?, Precondition?,
Constraint?, Exception?)+>
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Type (#PCDATA)>
  <!ELEMENT ReturnValue (#PCDATA)>
  <!ELEMENT Access (#PCDATA)>
  <!ELEMENT Static (#PCDATA)>
  <!ELEMENT Parameter (ParaType, ParaName)*>
  <!ELEMENT ParaType (#PCDATA)>
  <!ELEMENT ParaName (#PCDATA)>
  <!ELEMENT Polymorphism (Type, Class)*>
  <!ELEMENT IncludeClass (#PCDATA)>
  <!ELEMENT MethodInitValue (MethodName,
  InitValue)>
  <!ELEMENT MethodName (#PCDATA)>
  <!ELEMENT InitValue (#PCDATA)>
  <!ELEMENT Members (MemberType*,
  MemberName*, MemberValue*)*>
```

〈!ELEMENT MemberType (#PCDATA)〉
〈!ELEMENT MemberName (#PCDATA)〉
〈!ELEMENT MemberValue (#PCDATA)〉
〈!ELEMENT Messages (Sender*, Receiver*)〉
〈!ELEMENT Sender (ClassName, MethodName)〉
〈!ELEMENT Receiver (ClassName, MethodName)〉
〈!ELEMENT ClassName (#PCDATA)〉
〈!ELEMENT Purpose (#PCDATA)〉
〈!ELEMENT Note (#PCDATA)〉
〈!ELEMENT Precondition (#PCDATA)〉
〈!ELEMENT Constraint (#PCDATA)〉
〈!ELEMENT Exception (#PCDATA)〉

마지막으로 필드 모듈에 대한 정보들은 3.2.3절에 기술한 정보와 이를 정리한 〈표 4〉를 기반으로 다음의 규칙 4에 나타난다.

[규칙 4] 데이터멤버(field) 모듈의 구성

```
Field = Field*
Field = 이름, (데이터_형식 | 객체_형식), Access_Mode?,
        Static_Mode?, Final_Mode?, 초기값?, 해당_클래
        스*, 역할이나_목적?, 부가정보?)+
```

위에서 정의된 규칙을 적용하여 생성된 JML의 DTD는 다음과 같다.

〈!ELEMENT Field (Name?, (DataType | AttributeType), Access?, Static?, Final?, InitValue?, UseClass*, Purpose?, Note?)+〉
〈!ELEMENT Name (#PCDATA)〉
〈!ELEMENT DataTyper (#PCDATA)〉
〈!ELEMENT AttributeType (#PCDATA)〉
〈!ELEMENT Access (#PCDATA)〉
〈!ELEMENT Static (#PCDATA)〉
〈!ELEMENT Final (#PCDATA)〉
〈!ELEMENT UseClass (#PCDATA)〉
〈!ELEMENT Purpose (#PCDATA)〉
〈!ELEMENT Note (#PCDATA)〉

이상과 같은 과정을 통하여 생성된 JML을 적용하여 저작된 문서로부터 3.1절에서 언급한 원시 코드의 이해에 필요한 다양한 정보들을 획득하는 과정 및 결과는 다음 장에서 상세하게 설명한다.

4. 적용사례 및 분석

이 장에서는 JML을 구성하는 각 요소 및 속성들을 이용하여 실제 정보를 획득하는 과정과 실제 원시 코드에 적용시켜 생성된 각종 정보 및 이들의 표현을 기술한다.

본 연구의 테스트는 JDK 1.3 SE 버전에 포함되어 설치되는 데모 파일을 대상으로 실시하였다. 설치된 데모 폴더는 여러 개의 서브 폴더들을 포함하고 있는데, 이들 중에서 applet 폴더는 다시 각각의 프로젝트 별로 개별적인 폴더로 구성되고 그 중에서 ArcTest 프로젝트를 대상으로 데이터 활용 정도와 활용 방법을 기술한다. 〈표 5〉는 ArcTest에 대한 개괄적인 정보로 전체 라인수(주석 포함), 포함된 클래스의 개수 및 이름, 개별적인 클래스 별로 포함된 필드의 개수와 메소드의 개수 및 전체의 개수를 나타내고, mLOC는 변경된 라인수를 의미한다. mLOC의 변경된 라인수의 의미는 ArcTest.java에 포함된 주석정보 중에서 저작권에 관련된 정보를 제외한 것을 나타낸다. ArcTest.java에는 총 4개의 주석정보가 제공되는데, 그 중 2개는 "/* */"를 이용한 다중라인 주석으로 각각 저작권에 대한 정보와 ArcTest 클래스의 정의에 앞서 제공되는 실행에 관련된 정보이다. 나머지 2개는 ArcTest 클래스에서 이용되는 두 개의 객체형에 대한 정보로 "//"를 이용한 단일라인 주석 형식으로 정의된다.

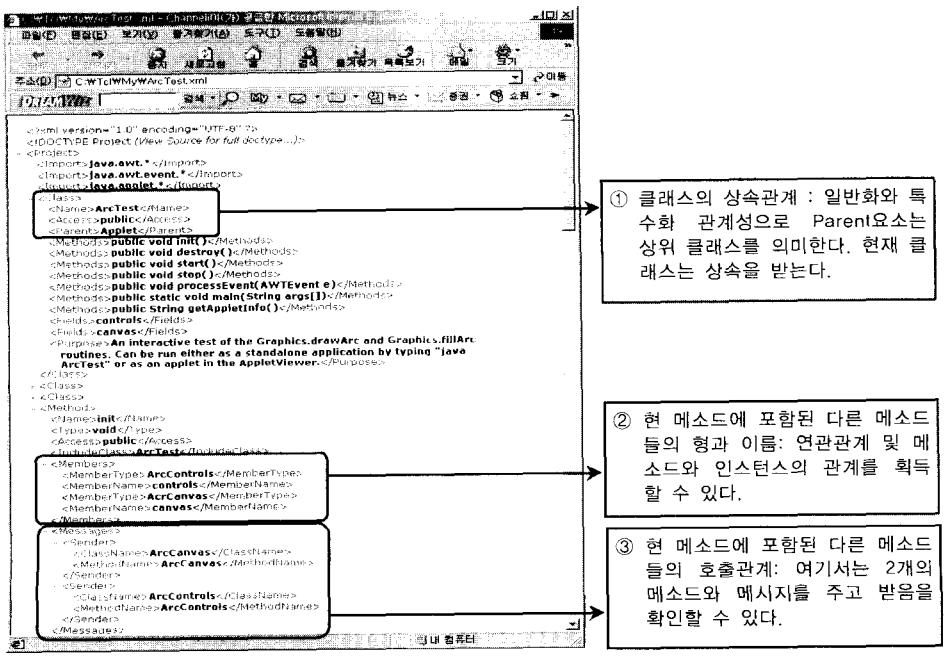
〈표 5〉 예제 프로그램의 개괄적인 정보
〈Table 5〉 General Information of Sample Program

프로젝트명	파일명	LOC	mLOC	클래스명	필드 개수	메소드 개수
ArcTest	ArcTest	159	130	ArcTest	2	7
				ArcCanvas	4	2
				ArcControls	3	2
총계	1			3	9	11

[그림 1]은 ArcTest 파일로부터 생성한 JML 문서를 인터넷 익스플로러를 통해 브라우징한 화면으로 Project라는 루트 요소로부터 출발하여 ArcTest 클래스를 중심으로 Class 요소 및 Method 요소 등의 정보를 디스플레이하고 있다.

먼저 3장에서 언급한 각 컴포넌트별 정보와 이를 적용한 JML을 중심으로 원시 코드의 이해나 재사용에 필요한 정보를 획득할 때 클래스들 사이의 관계나 객체들 사이의 관계 및 메소드 사이의 호출관계 및 다형성과 같은 정보들을 이용자가 획득하는 방법을 기술한다. 이미 언급한 것처럼 일반화와 특수화 관계성은 JML에서 각 Class 요소의 자식 요소인 Parent 요소를 이용하여 확인이 가능하다[그림 1의 ①]. 각 Class 요소에 존재하는 Interfaces 요소는 다중상속을 지원하는 Java의 객체지향 메커니즘으로 현재의 클래스에서 상속받는 인터페이스를 의미한다. 또한 현재 클래스를 부모 클래스로 하는 상속관계를 갖는 자식 클래스들은 Childs 요소를 통해 확인할 수 있다.

또한 연관관계에 포함되는 집약과 복합의 관계성은 어떤 클래스 내부에서 클래스의 일부로 구성되는 다른 클래스들과 그들의 인스턴스로 표현되며, 이들의 구분은 클래스나 객체들 상호간의 관계가 어느 정도의 밀접성으로 유지되는가에 달려있다. 즉, 객체와 클래스의 관계에서 생명주기가 동일할 경우 밀접성이 높다고 판단되므로 이러한



[그림 1] 예제 JML 파일
[Fig. 1] Sample JML File

관계성을 복합이라 한다. 하지만, 코드 내부에서 이들을 구분하는 것은 설계나 분석 단계와는 다른 구현 단계 혹은 최종 결과물 상에서는 큰 의미가 없으므로 본 연구에서는 이 둘의 의미를 구분하지 않고, 연관관계로 통일한다.

클래스들의 연관관계를 JML을 이용하여 구성하는 방법은 다음과 같다. JML에서 Class 요소는 여러 개의 Fields 요소를 가질 수 있다. 이들은 해당 클래스 내부에 존재하는 일반 원시 자료형들과 인스턴스들로 구성되며, 이들 중에서 클래스 내부에 존재하는 인스턴스들이 위에 언급한 연관관계에 해당하는 클래스들이 된다.

메소드들 사이의 호출 관계의 경우는 JML에 존재하는 Method 요소에 존재하는 Polymorphism 요소와 IncludeClass 요소 및 Messages 요소들을 이용하여 획득할 수 있다. 먼저 Polymorphism 요소는 메소드의 형식과 클래스의 이름을 요소들로 가지기 때문에 해당 클래스 자료구조의 메소드 정보와 형식을 추적하여 다형성 정보를 획득할 수 있고, IncludeClass 요소는 현 메소드를 포함하는 클래스의 이름을 제공한다. 마지막 요소인 Messages는 메시지 호출에 관한 정보를 제공하기 위해 Sender 요소와 Receiver 요소로 구성되고, 이들 두 요소는 각각 클래스 이름과 메소드 이름을 그 값으로 취한다. 이중에서 Sender 요소는 현재 메소드에 메시지를 전달하는 메소드와 이를 포함하는 클래스를 나타내고, Receiver 요소는 현재 메소드가 메시지를 전달하는 메소드와 이를 포함하는 클래스를 나타내기 때문에 이들 *Messages* 요소에 포함된 Sender와 Receiver 요소의 값들을 추적하여 메소드들의 호출관계를 획득할 수 있다[그림 1의 ③].

또한 메소드 내부에서 포함된 객체 및 클래스는 각 Method 요소의 Members 요소를 통해 확인할 수 있다. Members 요소는 각각 클래스의 이름과 객체의 이름을 나타내는 MemberType과 MemberName의 쌍들을 포함할 수 있는데 이 정보를 이용하여 메소드와 인스턴스의 관계성 및 연관관계를 획득한다[그림 1의 ②].

6. 결론

본 연구에서는 최근의 소프트웨어 환경의 변화와 소프트웨어 요구사항의 복잡화 및 하드웨어 환경 등의 변화로 인해 유발되는 소프트웨어 유지보수의 어려움을 들었다. 그리고, 이와 같은 어려움을 해결하는데 도움이 되는 정보들을 분산된 프로젝트 구현 환경에서 팀 구성원들 사이에 존재하는 다양한 정보들로 추출하고 정리하였다. 이들 정보들은 객체지향 설계 및 분석 과정을 통해 추출되는 다양한 정보들을 포함한다. 또한 이러한 정보들이 개발자들 사이에 원만한 정보의 표현 및 공유와 재사용에 이용될 수 있도록 웹 환경에서 표준 언어로 인식되고 있는 XML을 이용하여 제안하였다.

본 연구와 기존의 연구물들과의 가장 커다란 차이점으로는 XML을 통해 표현되는 정보들은 기존의 HTML이나 다른 문서 형식으로 표현되는 다양한 연구물들에 비해 정보의 추출 및 가공이 용이하다는 점을 들 수 있다. 이와 같은 XML 형식의 문서는 문서화의 목적인 문서 생성자와 문서의 사용자들 사이의 의사소통을 용이하게 하고, 설계 단계의 의도나 적용사례들 및 잠재적인 문제점 등과 같은 많은 정보들에 대한 통찰력을 제공해 줄 수 있다.

향후 지속적인 연구가 요구되는 분야로는 Java 원시 코드에 대한 DTD를 지속적으로 보완 발전시키는 것이 급선무이다. 또한 이와 병행하여 JML 문서를 분석하고, 에러 유무를 판단하기 위한 파서에 대한 연구가 필요하다. 그리고 서론에서 언급한 JML 문서를 최소한의 사용자 개입으로 완성할 수 있는 GUI 인터페이스를 기반으로 하는 JML 문서 생성기에 대한 연구와 생성된 JML 문서로부터 해당 Java 원시 코드의 골격 코드를 생성하는 Java 골격 코드 생성기에 대한 연구가 필요하다.

※ 참고문헌

[1] 장옥배, 유철중, 이병걸, 김지홍, 양해술, 김병기, '소프트웨어공학 이론과 실제', pp.495, 도서출판 한산, 2001.

[2] Frank, M., and Gail, K., "Software Engineering in the Internet Age", IEEE Internet Computing, pp.22-24, Sept.-Oct. 1998.

[3] Canfora, G., and Cimitile, A. 'Software Maintenance', Handbook of Software Engineering and Knowledge Engineering, Skokie, IL: Knowledge Systems Institute, Vol.1, pp.91-120, 2001.

[4] Bennett, K. H. 'An Overview of Maintenance And Reverse Engineering, The REDO Compendium', John Wiley & Sons, 1993.

[5] Yau, S.S., and Colleferro, J.S., "Some Stability Measures for Software Maintenance", IEEE Transactions on Software Engineering SE-6, No.6, pp.545-552, 1980.

[6] W3C, Extensible Markup Language(XML) 1.0, 2nd Ed., W3C Recommendation, Oct. 2000, http://www.w3.org/TR/REC-XML.

[7] Mark Johnson, "XML for the absolute beginner", JavaWorld, April 1999.

[8] Richard Light, "Presenting XML", Sams.net, 1997.

[9] Michael, D., "Are Elements and Attributes Interchangeable?", XML Journal, Vol.2, Is.7, pp.42-47, July 2001.

[10] Junichi Suzuki, Yoshikazu Yamamoto, "Managing the Software Design Document with XML", ACM SIGDOC, 1998.

[11] Ion P, et.al., "Mathematical Markup Language", W3C.

[12] CheckFree Corp., "Open Financial Exchange Specification 1.0.2", Open Financial Exchange.

[13] Lisa Friendly, "The Design of Distributed Hyperlinked Programming Documentation", Sun Microsystems Inc., 1996.

[14] Javadoc, http://java.sun.com/javadoc.

[15] 장근실, 문양선, 유철중, 장옥배, "C++ 프로그램의 유지보수 지원 시스템 개발", 정보처리논문지, 제5권 제7호, pp.1759-1773, 1998.

[16] Grady, B., and et. al., 'The Unified Modeling Language User Guide', Addison Wesley, 1999.

[17] Jang, G. S., Yoo, C. J., and Chang, O. B., "Information Sharing of Java Program Using XML", ACIS 1st Int. Conf., SNPD 00, pp.384-391, May 2000.

[18] Doc++, http://www.zib.de/Visual/software/doc++/index.html.

[19] 김재웅, 유철중, 장옥배, "Java 프로그램에 대한 복잡도 척도들의 실험적 검증", 정보과학회논문지: 소프트웨어 및 응용, 27권 12호, pp.1141-1154, Dec. 2000.

[20] Pankaj, K. G., and Walt, S., "A Hypertext System to Manage Software Life-Cycle Documents", IEEE Software, pp.90-98, May 1990.

[21] Marry, H., "Using Documentation as a Life-Cycle Tool", Software Magazine, Dec. 1992.

[22] J. Gosling, B. Joy, G. Steele, G. Bracha, 'The JavaTM Language Specification', 2nd Ed., Addison Wesley, 2000.

장 근 실

1995년 전북호원대학교
전자계산학과 졸업(이학사)
1997년 전북대학교 대학원
전산통계학과 졸업(이학석사)
1999년 전북대학교 대학원 전산
통계학과 수료(박사과정)
1996년 9월 ~ 1997년 3월
전북대학교 전자계산소 조교
1997년 3월 ~ 현재
광양보건대학 컴퓨터정보과
조교수
관심분야 : 소프트웨어공학,
컴포넌트기술, 웹공학

유 철 중

1982년 전북대학교
전산통계학과 졸업(이학사)
1985년 전남대학교 대학원
계산통계학과 졸업(이학석사)
1994년 전북대학교 대학원
전산통계학과 졸업(이학박사)
1982년 9월 ~ 1985년 3월
전북대학교 전자계산소 조교
1985년 4월 ~ 1996년 12월
전주기전여자대학
전자계산과 전임강사-부교수
1997년 1월 ~ 현재
전북대학교 자연과학대학
컴퓨터과학과
전임강사-조교수
관심분야 : 소프트웨어공학,
에이전트공학, 컴포넌트기술,
분산객체기술, GNSS(GPS),
GIS, 멀티미디어, 인지과학

장 옥 배

1966년 고려대학교 수학과 졸업
(이학사)
1973년 고려대학교 교육대학원
(교육학석사)
1980년 죠지아 주립대
(박사과정수료)
1987년 산타바바라대학교 졸업
(Ph.D)
1990년~1991년 영국 에딘버러
대학교 객원교수
1980년 4월~현재 : 전북대학교
공과대학 전자정보공학부
교수
관심분야 : 소프트웨어공학,
전산교육, 수치해석,
인공지능 등