DOI: 10.3745/KIPSTD.2008.15-D.6.825

OCL을 이용한 자동화된 코드스멜 탐지와 리팩토링

김 태 웅[†] · 김 태 공[††]

요 약

리팩토링은 내부적으로는 시스템의 품질을 개선하고, 외부적으로는 시스템의 기능을 유지하는 일종의 소프트웨어를 변경하는 과정이다. 이러한 리팩토링을 적용하여 기존 소스코드를 개선하기 위해서는 개선할 사항이 무엇인지를 아는 것이 우선이다. 이를 위해 Martin Fowler와 Kent Beck은 코드속의 나쁜 냄새(코드스멜)를 식별할 수 있는 방법을 제시 하였다. 또한 코드스멜을 탐지하고 어디에 어떤 리팩토링을 적용할 것인가를 결정하는 문제와 관련된 몇몇 연구가 발표되었다. 그러나 이러한 연구들은 코드스멜에 대한 명확한 표현이 부족하거나 한정된 코드스멜만을 탐지하는 단점이 있다. 그리고 리팩토링을 적용할 경우 행위보존을 위한 선행조건들의 표현방법이 리팩토링 절차에 포함되어 있거나 정형화되지 않아 행위보존의 모호함이 발생되는 단점을 가지고 있다. 이에 본 논문에서는 OCL을 이용하여 코드스멜의 정보를 정확히 명세화하고, OCL 번역기를 통해 코드스멜을 자동으로 탐지하여 리팩토링하는 프레임워크를 제안한다. 또한 적용사례를 통하여 자바소스코드속의 코드스멜을 OCL로 명세화하여 자동탐지하고, 리팩토링을 적용해 봄으로써 활용성과 효용성을 검증해본다.

키워드 : 리팩토링, 코드스멜, OCL, AST, EMF, 자바

Automated Code Smell Detection and Refactoring using OCL

Kim Tae Woong[†] · Kim Tae Gong[††]

ABSTRACT

Refactoring is a kind of software modification process that improves system qualities internally but maintains system functions externally. What should be improved on the existing source codes should take precedence over the others in such a modification process using this refactoring. Martin Fowler and Kent Beck proposed a method that identifies code smells for this purpose. Also, some studies on determining what refactoring will be applied to which targets through detecting code smells in codes were presented. However, these studies have a lot of disadvantages that show a lack of precise description for such code smells and detect limited code smells only. In addition, these studies showed other disadvantages that generate ambiguity in behavior preservation due to the fact that a description method of pre-conditions for the behavior preservation is included in a refactoring process or unformalized. Thus, our study represents a precise specification of code smells using OCL and proposes a framework that performs a refactoring process through the automatic detection of code smells using an OCL interpreter. Furthermore, we perform the automatic detection in which the code smells are be specified by using OCL to the java program and verify its applicability and effectivity through applying a refactoring process.

Keywords : Refactoring, Code Smells, OCL(Object Constraint Language), AST(Abstract Syntax Tree), EMF(Eclipse Modeling Framework), Java

1. 서 론

소프트웨어 개발의 효율을 높이고 유지보수 비용을 낮추기 위한 많은 노력들이 이루어지고 있으며 그 중 대표적인 것이 리팩토링(Refactoring)이다. 리팩토링은 프로그램의 행위를 보존하면서 프로그램의 가독성, 구조, 성능, 유지보수성 등을 향상시키는 방법이다. 시스템의 기능을 유지하면서 시스템의 이해도를 높임과 동시에 유지보수를 보다 쉽게 할 수 있도록 내부 구조를 변경하는 것이다[1]. 이러한 리팩토링의 결과로 코드의 확장성, 모듈화, 재사용성, 유지보수성 같은 품질을 개선하여 개발의 속도를 높이고 코드 복잡도를 낮출 수 있다[2].

이미 최근의 소프트웨어 개발에서는 리팩토링이 일반적인 요소로서 활발하게 이용되고 있다. 이와 같은 리팩토링을 이용하여 기존 코드의 설계를 개선하기 위해서는 리팩토링 적용에 앞서 개선할 점이 무엇인지를 아는것이 우선이다[3]. 이를 위해 Martin Fowler와 Kent Beck은 코드속의 나쁜냄새(Bad Smells in Code, 코드스멜)에서 설계 문제를 식별할 수 있는 별도의 방법을 제시하였다[4]. 그들은 설계문제를 냄새에 비유했고, 특정 냄새를 제거하는데 어떤 리팩토링이 좋

† 정 회 원 : 인제대학교 컴퓨터공학부 연구교수
†† 정 회 원 : 인제대학교 컴퓨터공학부 교수
논문접수 : 2008년 10월 2일
수 정 일 : 1차 2008년 11월 18일
심사완료 : 2008년 11월 27일

은지 설명하였다.

코드스멜을 탐지하여 어디에 어떤 리팩토링을 적용할 것인가를 결정하는 문제와 관련된 몇몇 연구가 발표되었다[5,6,7,8]. 그러나 이러한 연구들은 코드스멜에 대한 명확한 표현이 부족하거나 한정된 코드스멜만을 탐지하는 단점을 가지고 있다. 그리고 리팩토링을 적용할 경우 행위보존(behavior preservation)을 위한 조건(pre, post-condition)들의 표현방법이 리팩토링 절차에 포함되어 있거나 정형화되지 않아 행위보존의 모호함이 발생되는 단점을 가지고 있다. 코드스멜에 대한 명확한 표현의 부족은 향후 리팩토링 적용시 시스템의 기능 보존과 같은 또 다른 문제점을 야기시킨다.

이에 본 논문에서는 OCL[9]을 이용하여 코드스멜을 정의하고, 이를 자동탐지에 이용한다. 그리고 탐지된 소스코드에 리팩토링 절차를 거쳐 수정된 새로운 소스코드를 산출하는 프레임워크를 제안한다. 이를 위하여 소스코드를 AST[10]메타모델을 확장한 EAST(Extended Abstract Syntax Tree)메타모델 기반의 XMI(XML Metadata Interchange)[11]문서로 변환하고, 코드스멜에 대한 정의는 OCL을 이용하여 작성한다. 작성된 OCL은 Eclipse 기반의 OCL 번역기(OCL Interpreter)[12]를 통하여 실행하고 자동탐지 한다. 탐지된 코드스멜은 XMI문서를 조작하기 위해 구현한 EMF[13] 기반의 JAS(Java Abstract Syntax) API와 기초리팩토링(primitive refactoring) API를 이용하여 수행하며, 최종적으로 수정된 소스코드를 얻는다.

본 논문은 2장에서 관련연구로 기존 연구들에 대해 분석하고 문제점을 지적한다. 3장에서는 코드스멜을 탐지하기 위한 방법과 리팩토링하는 방법에 대해 기술한다. 그리고 4장에서는 본 연구의 효용성과 활용성을 검증하기 위한 적용사례에 대해 다루고, 끝으로 5장에서는 향후 연구과제를 포함하여 결론을 맺는다.

2. 관련 연구

이번 장에서는 관련연구로서 코드스멜의 정의 및 종류에 대해 알아보고, 본 논문에서 제안한 코드스멜 탐지 및 리팩토링과 관련된 연구들에 대해 살펴본다.

2.1 코드속의 나쁜냄새(Bad Smells in Code)

코드스멜은 소스코드에서 문제(Problem)가 될 수 있는 부분을 나타낸다. 그러나 코드스멜은 컴파일시에 나타나는 문법적인 오류나 경고와는 다르다. 예를 들어 나쁜 소프트웨어 디자인 또는 나쁜 프로그래밍 습관으로 인해 발생하는 소스코드를 말한다. 이러한 잘못된 프로그래밍 습관이나 디자인은 소프트웨어에 새로운 기능이 추가되거나 플랫폼의 변경시 소프트웨어 개발 비용이 증가되는 악영향을 가져온다. 또한 잘못된 프로그래밍 습관으로 시스템 전체의 성능 저하와 같은 문제점도 발생할 가능성이 있다.

코드스멜은 크게 다음의 2가지로 분류하여 정의되며, <표 1>과 같은 세부항목으로 나누어질 수 있다[5].

- 단순 코드스멜(primitive smell) : 하나의 클래스에서 직접적으로 알 수 있는 단순냄새
- 유도된 코드스멜(derived smell) : 여러 클래스들의 관계에 의해 알 수 있는 유도된 냄새

<표 1>과 같이 단순 코드스멜은 하나의 클래스에서 유추가능한 반면 유도된 코드스멜은 여러 클래스간의 관계(상속관계, 인스턴스, 메소드의 사용 및 필드의 사용 등)에서 정보를 추출하여 도출 가능하다. 이러한 코드스멜을 탐지하기 위한 여러 가지 연구들이 있으며, 그 장단점에 대해 알아본다.

2.1.1 Van Emden의 연구[14]

자바소스코드에서 코드스멜을 탐지하고 코딩의 표준을 제시하는 도구인 JCosmo에 대한 연구이며, 자바의 문맥을 검사

〈표 1〉 코드스멜의 유형과 종류

유 형	종 류	설 명
primitive smell	Long Parameter List	메소드의 파라미터 개수가 너무 많다
	Switch Statements	지나치게 많은 case를 사용하는 switch 문장은 코드 중복의 신호
	Too Few Fields	필드의 수가 너무 적다
	Too Many Fields	필드의 수가 너무 많다
	Too Few Method	메소드의 수가 너무 적다
	Too Many Method	메소드의 수가 너무 많다
	Long Method	메소드의 내부가 너무 길다
	Message Chain	특정 개체를 사용하기 위해 지나치게 많은 클래스들을 거쳐야 한다
derived smell	Refused Bequest	상위 클래스의 속성과 메소드가 하위 클래스에서 사용되지 않는다
	Feature Envy	메소드를 수행할 때 주로 다른 클래스로부터 데이터를 얻어 와서 기능을 수행한다
	Shotgun Surgery	특정 클래스를 수정하면 그때마다 관련된 여러 클래스들내에서 자잘한 변경을 해야 한다
	Parallel Inheritance Hierarchies	연관 있는 여러 클래스 계층도가 지나치게 많이 생겨 중복을 유발

하여 코드스멜을 탐지하고 있다. 이는 자바의 구조에 대한 인식 및 유도된 코드스멜을 탐지할 수 없으며, 코드스멜이 추가될 경우 소프트웨어를 다시 개발해야 하는 단점을 가진다.

2.1.2 Richard C. Holt의 연구[15]

소스코드로부터 추출된 정보들을 기반으로 관계대수 도구인 Grok이라는 스크립트언어를 사용하여 코드스멜을 탐지하고 있다. 이는 소스코드가 가지고 있는 다양한 정보들로부터 사실(Facts)를 수집하고, 이를 저장소(Facts Repository)에 저장한 후, 관계대수를 이용하여 탐지한다. 이 논문에서는 소스코드의 변경시 사실들을 다시 수집하고, 저장소를 다시 구축해야 하는 단점을 가지고 있다.

2.1.3 Stefan Slinger의 연구[5]

코드스멜 탐지 도구를 이클립스 기반의 플러그인으로 개발하였다. 새로운 단순 코드스멜을 추가할 경우 제안한 5단계를 이용하여 프로그램을 작성해야 한다. 특히 클래스들간의 정보를 이용하여 탐지할 수 있는 유도된 코드스멜인 경우 Richard C. Holt의 연구와 같이 클래스들간의 정보로부터 사실을 수집한 후 Grok 스크립트 언어를 이용하여 탐지하고 있다. 이는 새로운 코드스멜을 정의할 경우 자바언어를 이용하여 복잡한 플러그인을 추가로 개발하여야 하며, 저장소(Facts Repository)를 다시 구축하여야 하는 단점을 가지고 있다.

위에서 살펴본 바와 같이 코드스멜을 탐지하고자 하는 연구들이 발표되었다. 코드스멜은 언제든지 새롭게 정의될 수 있으며, 소프트웨어 성격과 도메인에 따라 다르게 정의될 수 있다. 그러나 이러한 연구들은 새로운 코드스멜의 추가나 수정에 대한 방법으로 새로운 소프트웨어를 개발해야 하는 단점을 가지고 있다. 또한 코드스멜에 대한 명세가 정확히 이루어지지 않아 코드스멜에 대한 정의가 모호하다.

2.2 리팩토링

2.2.1 Narbor C. Mendonca의 연구[16]

소스코드를 리팩토링하기 위해 소스코드를 JavaML[17] XML문서로 표현하고 있으며, XQuery[18]를 이용하여 리팩토링을 수행한다. 리팩토링 후의 시스템의 행위보존(Behavior Preservation)을 위한 선행조건(pre-condition)은 AFS(Analysis Functions)[19]을 사용하여 XQuery에 포함시켜 수행하고 있다. 이는 리팩토링을 수행하는 부분에 선행조건이 포함되어 있어 시스템의 종류에 따라 달라질 수 있는 선행조건을 대체하기 어렵다는 단점을 가지고 있다.

2.2.2 Mel O Cinneide, B.Sc. M.Sc.의 연구[20]

디자인패턴을 위한 리팩토링을 제안하고 있으며 리팩토링을 적용한 후의 행위보존에 대해 기술하고 있다. 리팩토링을 미니리팩토링(Mini-Refactoring)으로 정의하고, 이들의 조합으로 리팩토링을 수행하는 절차를 가지고 있다. 행위보존을 위해 각 미니리팩토링 요소마다 선행조건을 부여하고 있지만 정확한 명세언어를 이용하여 선행조건을 명세하지 않아 선행조건의 모호함이 발생한다.

2.2.3 Raul Marticorena의 연구[21]

리팩토링을 하기 위한 방법론을 제시하였으며, 리팩토링의 요소를 재사용하기 위해 작은 리팩토링으로 소형화 하였으며(MOON, Minimal Object Oriented Notation), 선행조건에 대한 명세를 클래스로 구현하여 정의하였다. 이는 이미 컴파일된 선행조건을 이용함으로써 효율성이 떨어진다.

이러한 연구들은 리팩토링을 위한 여러 가지 방법과 프레임워크를 제시하였다. 그러나 소스코드를 표현하기 위한 명확한 메타모델이 정의되지 않고, 리팩토링 적용 후의 행위보존을 위한 선행조건의 표현방법이 정확하지 않다. 또한 선행조건이 컴파일된 형태로 존재하거나 리팩토링의 절차에 포함되어 있어 효율성이 떨어진다는 단점이 있다.

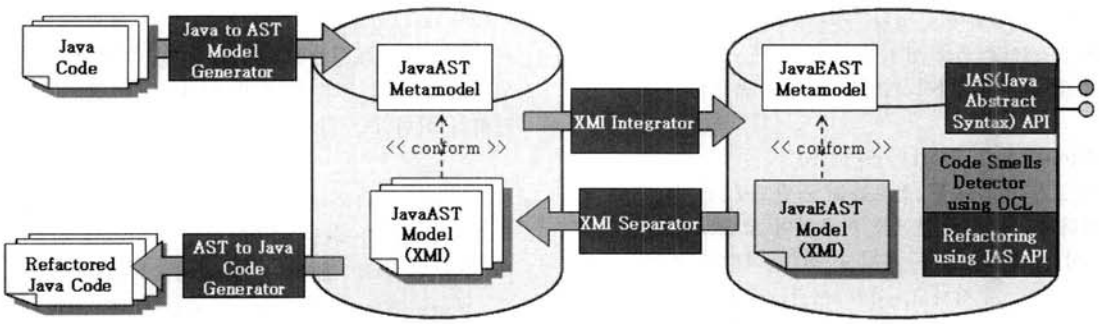
3. 코드스멜 탐지 및 리팩토링

본 장에서는 코드스멜을 자동으로 탐지하기 위한 전체 프레임워크를 (그림 1)과 같이 제안한다. 코드스멜을 자동으로 탐지하고 리팩토링하기 위해서는 <표 2>와 같은 다양한 요구사항을 만족해야 한다. <표 2>는 자동화된 코드스멜 탐지와 리팩토링을 위한 프레임워크가 가져야 할 요구사항과 이를 해결하기 위한 방법이다.

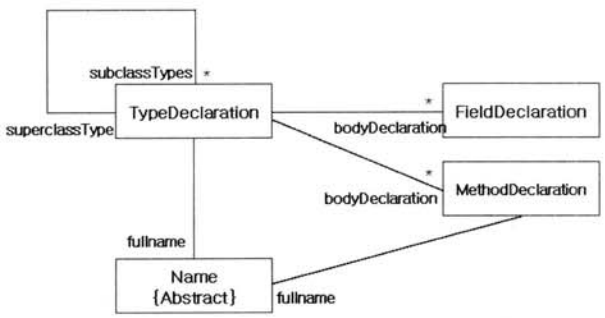
(그림 1)은 <표 2>에서의 요구사항을 만족하는 프레임워

〈표 2〉 프레임워크 요구사항 및 해결방법

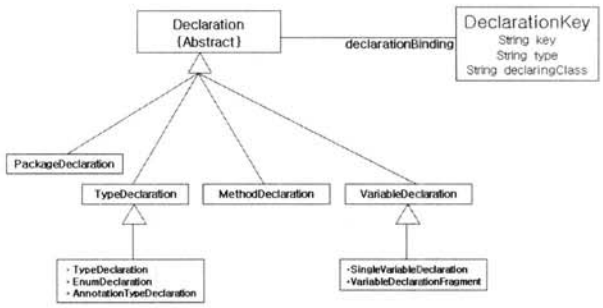
요구사항	해결방법
자바 언어의 모든 문법적 정보를 표현할 수 있도록 소스코드를 표현해야 한다.	AST 메타모델을 기반으로 자바소스 코드를 표현한다.
표현된 소스코드 모델에 쉽게 접근하여 의미적 정보를 추출할 수 있어야 한다.	OCL을 이용하여 소스코드의 의미적 정보에 접근한다.
OCL을 이용하여 대상모델에 쉽게 접근하고, 클래스간의 관계에 대한 정보를 추출할 수 있어야 한다.	JavaAST기반의 모델을 JavaEAST기반의 메타모델로 변환하여 클래스간의 관계정보에 쉽게 접근하도록 한다.
리팩토링을 쉽게 적용할 수 있어야 한다.	리팩토링 요소를 미니리팩토링 요소로 분해하여 API로 구현한다.
코드스멜을 탐지하기 위한 방법이 간편해야 한다.	코드스멜을 탐지하기 위해서는 OCL을 이용하여 코드스멜을 명세하고, 이를 탐지하는데 이용한다.
프레임워크를 구성하는 각 모델간의 변환이 잘 이루어져야 한다.	Java Source Code, JavaAST, JavaEAST간의 변환이 원활하도록 AST Generator, Integrator, Separator, Code Generator를 구현한다.



(그림 1) 코드스멜 자동탐지와 리팩토링을 위한 전체 프레임워크



(그림 2) 패키지를 포함한 클래스이름 정보와 상속관계를 표현하기 위한 메타모델



(그림 3) 바인딩 정보를 표현하기 위한 메타모델 (선언부분)

크이다. 자바로 작성된 소스코드를 MoDisco[10]에서 제안한 JavaAST(Java Abstract Syntax Tree)로 나타내고(Java to AST Model Generator), 이를 본 논문에서 확장한 JavaEAST (Java Extended Abstract Syntax Tree)기반의 XMI문서로 변환한다(XMI Integrator). 여기에 OCL을 적용하여 코드스멜을 탐지하고, JAS API를 통하여 리팩토링을 적용하여 변경된 JavaEAST기반의 XMI문서를 산출한다. 이를 다시 JavaAST 기반의 XMI문서로 변환(XMI Separator)하고, 최종적으로 수정된 소스코드를 얻는다(AST to Java Code Generator).

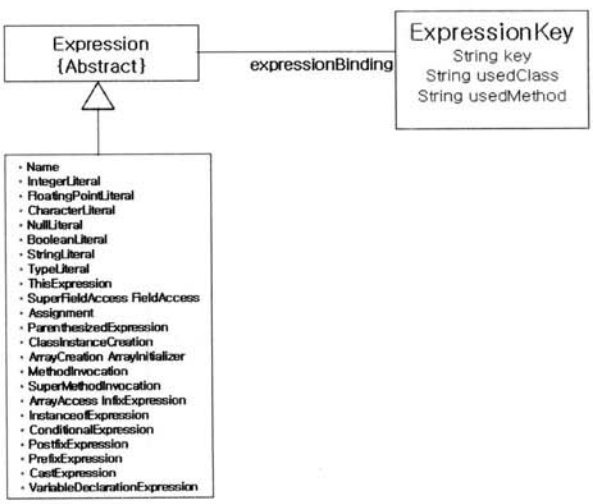
3.1 JavaEAST(Java Extended Abstract Syntax Tree)

자바로 작성된 소스코드를 JavaAST로 변환하면 소스코드의 문법 정보가 트리구조의 형태로 저장되며, 하나의 자바 소스당 하나의 XMI문서가 산출된다. 하지만 이는 소스코드의 문법적인 정보만을 가지고 있으며, 특히 여러 개의 클래스들간의 관계를 분석해야 비로소 가능한 코드속의 유도된 코드스멜을 탐지하는데 문제점이 있다. 따라서 본 논문에서 구현한 XMI Integrator엔진을 통해 기존의 JavaAST모델을 JavaEAST기반의 모델로 변환 및 통합한다. 이 모델은 코드속의 유도된 코드스멜을 탐지 가능하도록 필드의 바인딩 정보를 포함하고, OCL로 자유로운 탐색(Navigation)이 가능하도록 그림 2, 3, 4와 같은 모델을 추가한 EMF[13]기반의 메타모델이다.

(그림 2)는 클래스 선언부인 'TypeDeclaration'에 'fullname'을 가지게 하고, 속성으로서 패키지를 포함한 클래스의 이름을 표현하고 있다. 또한 'superClassType', 'subClassTypes'를 가지게 함으로써 클래스의 상속관계에 대한 정보를 표현하고 있다.

(그림 3)은 JavaAST모델의 모든 선언부가 'declarationBinding' 항목(element)을 가지고 속성으로서 key, type, declaringClass를 두어, 변수 또는 메소드가 선언될 때의 바인딩 정보를 표현하고 있다.

(그림 4)는 자바언어가 표현할 수 있는 모든 형태의 변수 또는 메소드 참조 요소에 'expressionBinding'을 가지게 하고, 속성으로서 key, usedClass, usedMethod를 두어 변수 또는 메소드를 참조하는 시점에서의 바인딩 정보를 표현하고 있다. 이러한 바인딩에 필요한 정보는 자바소스코드를



(그림 4) 바인딩 정보를 표현하기 위한 메타모델 (참조부분)

〈표 3〉 바인딩 정보를 표현하기 위한 확장된 메타모델의 속성 정보

분 류	속 성	설 명
DeclarationKey	key	선언된 필드 또는 메소드의 유일한 키값
	type	선언된 필드 또는 메소드의 타입
	declaringClass	선언된 클래스
ExpressionKey	key	선언된 필드 또는 메소드의 유일한 키값
	usedClass	필드나 메소드가 사용된 곳(클래스)
	usedMethod	필드나 메소드가 사용된 곳(메소드)

컴파일한 바이트코드로부터 얻어온다.

<표 3>은 바인딩 정보를 표현하는 항목에 대한 설명이다.

3.2 OCL을 이용한 코드스멜 명세

단순 코드스멜은 하나의 클래스에서 직접적으로 발견 가능한 것으로, 예시로 단순 코드스멜을 OCL로 명세하면 <표 4>와 같다.

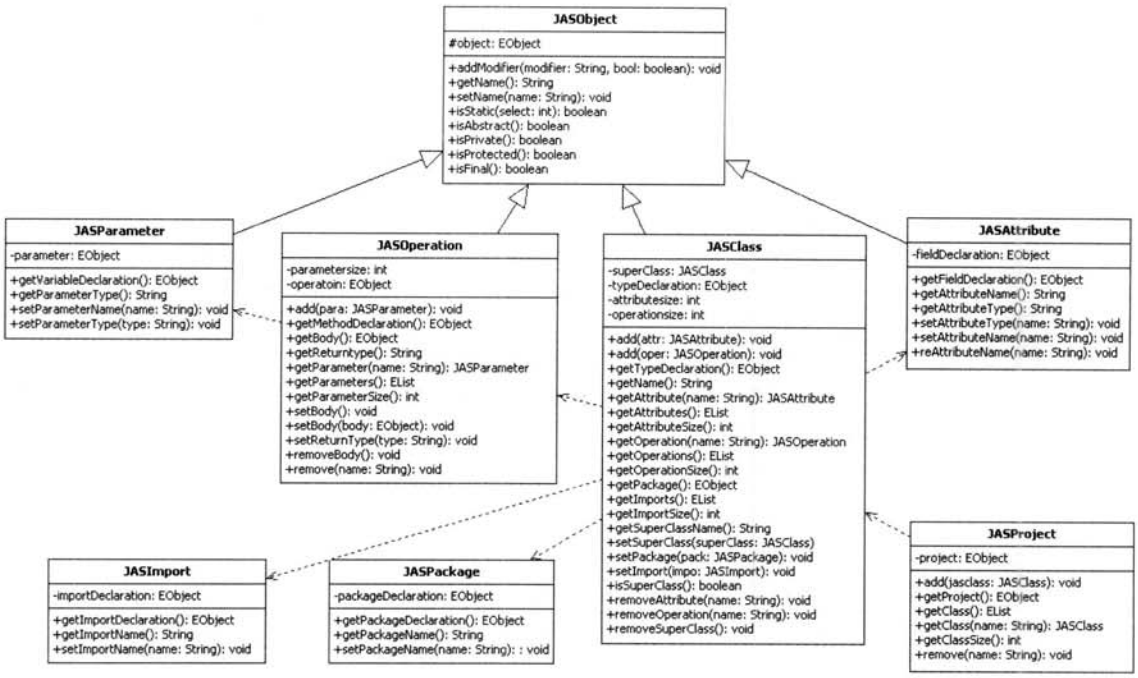
하지만 유도된 코드스멜은 단순 코드스멜과는 달리 여러 클래스들간의 관계를 분석함으로써 얻어진다. 이는 JavaAST 메타모델 기반으로 산출된 XMI문서로는 분석이 어려우며, 좀 더 복잡한 명세를 요구한다. 예를들어, 유도된 코드스멜 중 'Refused Bequest'를 탐지하기 위해서는 상속관계에 있는 클래스의 추출 뿐만 아니라, 클래스에는 어떠한 속성과 메소드가 선언되고 정의되어 있는지, 또 그 속성과 메소드는 어떤 클래스와 메소드에서 사용되는지에 대한 정보를 분석해야 한다. 이러한 클래스간의 관계에서 도출할 수 있는 정보를 OCL을 사용하여 명세화 한다<표 5>. 이러한 OCL명세는 본 논문의 3.1절에서 제시한 JavaEAST 메타모델을 기반으로 하고 있으며, 향후 특정한 코드속의 유도된 코드스

〈표 4〉 OCL을 이용한 단순 코드스멜 명세

단순 코드스멜 종류	설명	OCL 명세
Long Parameter List	메소드의 인자값의 개수가 지정해준 값보다 클 경우	self.parameters->notEmpty() self.parameters->size() > maximum
Too Many Fields Too Few Fields	클래스에 선언된 필드의 수가 지정해준 값보다 크거나 작을 경우	self.bodyDeclarations ->exists(oclIsTypeOf(FieldDeclaration)) self.bodyDeclarations ->select(oclIsTypeOf(FieldDeclaration)) ->size() > maximum(or < minimum)

〈표 5〉 클래스간의 관계로부터 도출 가능한 정보 (OCL을 이용하여 명세)

OCL 구문	의 미
context Project def:getSuperClasses() :Set(TypeDeclaration) =self.compilationUnits.types->collect(oclAsType(TypeDeclaration))->select(subclassTypes->notEmpty())->select(bodyDeclarations->select(oclIsTypeOf(FieldDeclaration))->notEmpty())->asSet()	현재 클래스의 상위 클래스를 가져온다
context Project def:getSubclassTypeKey(sType : TypeDeclaration):Set(String) =sType.subclassTypes->collect(oclAsType(SimpleType)).name.fullyQualifiedName->collect(subType|self.getTypeDeclarationByName(subType).declarationBinding.key)->asSet()	현재 클래스를 상속받는 클래스의 바인딩키값을 가져온다
context TypeDeclaration def:getFieldDeclarationKey():Set(String) =self.bodyDeclarations->select(oclIsTypeOf(FieldDeclaration)) ->collect(oclAsType(FieldDeclaration)).fragments.declarationBinding.key->asSet()	현재 클래스에서 선언된 필드들의 바인딩키값을 가져온다
context TypeDeclaration def:getFieldNames():Set(String) =self.bodyDeclarations->select(oclIsTypeOf(FieldDeclaration))->collect(oclAsType(FieldDeclaration)).fragments.name.fullyQualifiedName->asSet()	현재 클래스에서 선언된 필드들의 이름을 가져온다
context Project def: getUsedClassOfSimpleName(fKey : String) : Set(String) = SimpleName.allInstances()->select(expressionBinding->notEmpty()) ->select(expressionBinding.key = fKey).expressionBinding.usedClass->asSet()	변수의 키값으로 변수가 사용된 클래스의 이름을 가져온다.
context Project def : getTypeOfQualifiedName(fKey : String) : Set(String) = QualifiedName.allInstances()->select(expressionBinding->notEmpty()) ->select(expressionBinding.key = fKey).qualifier ->select(oclIsTypeOf(SimpleName))->collect(oclAsType(SimpleName)).expressionBinding.key->asSet()->collect(ebKey | self.getPositionOfVariable(ebKey))->asSet()	변수의 키값으로 그 변수의 타입을 가져온다



(그림 5) JAS(Java Abstract Syntax) API 클래스 다이어그램

멜을 정의하기 위한 조각들로 사용된다.

3.3 리팩토링을 위한 JAS API(Java Abstract Syntax API)

리팩토링은 내부적으로는 시스템의 품질을 개선하고, 외부적으로는 시스템의 기능을 유지하는 일종의 소프트웨어를 변경하는 과정이다. 이러한 리팩토링을 자동화하기 위해서는 소스코드에 대한 접근 및 조작이 간편해야 하며, 다양하고 새로운 리팩토링 요소를 쉽게 추가 가능해야 한다. 이는 소스코드를 어떠한 모델로 기술하느냐?, 어떠한 방법으로 해당 모델에 접근할 것인가? 가 매우 중요하다.

```
package jas.refactor;

import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.util.EcoreUtil;
import jas.metamodel.xmiIntegrator.XMIChangeUtil;

public class JASClass extends JASObject{
    private EObject compilation;
    private EObject typeDeclaration;
    private int attrsize;
    private int opersize;

    public JASClass(String name,boolean isAbstract,String modifier){
        super(JASUtil.createTypeDeclaration());
        this.compilation = JASUtil.createCompilationUnit();
        this.typeDeclaration = super.object;
        super.addModifier("abstract", isAbstract);
        super.addModifier(modifier, true);
        this.setName(name);
        JASUtil.eListGet(compilation, "types").add(this.typeDeclaration);
    }
    public JASClass(EObject type) {
        super(type);
        this.typeDeclaration = type;
    }
```

(그림 6) JAS API중 JASClass의 구현부분

본 논문에서는 JavaAST메타모델을 확장한 JavaEAST메타모델을 기반으로 소스코드를 XMI형태로 표현하고 있다. 따라서 JavaEAST모델에 접근 가능한 JAS(Java Abstract Syntax) API를 EMF기반으로 구현하여 리팩토링을 한다. (그림 5)와 (그림 6)은 본 논문에서 구현한 JavaEAST기반에서 생성된 XMI문서에 접근할 수 있는 API구조를 나타낸 클래스다이어그램과 소스코드의 일부이다.

3.4 미니리팩토링(Mini-Refactoring) 요소 정의

이번 절에서는 리팩토링을 적용하기 위한 방법으로 <표 6>과 같이 미니리팩토링(Mini-Refactoring)을 정의하여, 이들의

〈표 6〉 미니리팩토링(Mini-Refactoring) 요소 정의

분류	항목
Class	createClass(name,isAbstract,modifier)
	renameClass(oldName,newName)
	removeClass(name)
Method	createMethod(className,name,returnType,isAbstract,modifier)
	renameMethod(className,oldName,newName)
	removeMethod(className,name)
	extractMethod(className,name)
Field	createField(className,type,modifier,name)
	renameField(className,oldName,newName)
	removeField(className,name);
Parameter	addParameter(className,methodName,name,type)
	removeParameter(className,methodName,name)

조합으로 리팩토링이 이루어지도록 한다. 이는 리팩토링을 하기 위한 과정을 작은 단위의 리팩토링 요소로 쪼개어 정의함으로써 향후 새로운 리팩토링을 추가할 경우 작은 리팩토링 요소의 결합으로 새롭게 정의할 수 있어 재사용의 효율을 높일 수 있다.

3.5 OCL을 이용한 미니리팩토링 선행조건(pre-condition) 정의

3.4에서 정의된 미니리팩토링을 실제 적용하기 위해서는 리팩토링을 실행하기 위한 선행조건이 필요하다. 이는 리팩토링을 적용하는데 필요한 조건을 명세함으로써 리팩토링이 안전하게 진행될 수 있도록 하는데 있다. 본 논문에서는 이러한 조건들을 OCL을 사용하여 명세하였다. 이는 리팩토링 절차와 선행조건을 분리하여 정의함으로써 서로 독립적으로 사용가능하고 재사용의 효율을 높일 수 있다. 또한 작성된 OCL명세는 정의에만 그치지 않고 OCL번역기를 통하여 선행조건 검사에 사용된다. <표 7>은 미니리팩토링에 사용될 선행조건들을 OCL을 이용한 명세의 일부분이다.

3.4에서 정의된 미니리팩토링 요소에 3.5에서 명세된 선행조건을 적용하기 위한 소스코드는 (그림 7)과 같으며, 별도의 파일로 작성된 OCL을 읽어들여 미니리팩토링을 적용하는 과정이다.

4. 적용 사례

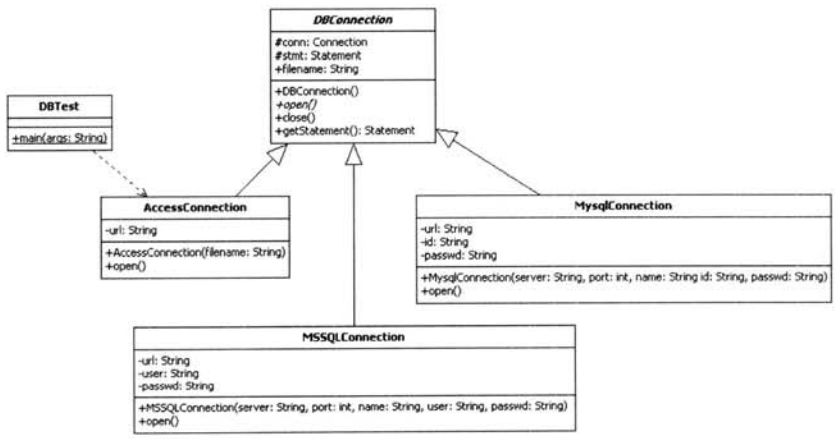
이번 장에서는 적용사례로서 본 논문에서 제안한 OCL을 이용한 자동화된 코드스멜 탐지와 리팩토링을 실제 적용하고, 수정된 소스코드를 얻는 절차 및 방법에 대해 기술한다. 적용할 대상은 (그림 8)과 같은 다양한 데이터베이스 연결자를 구성하는 예제 프로그램이다.

이 예제 프로그램은 (그림 8), 나)의 밑줄 부분과 같이 'DBConnection'에서 선언된 필드인 'filename'은 하위 클래스인 'AccessConnection'과 'DBTest' 클래스에서 사용되고 있다. 'AccessConnection' 클래스에서는 상위의 필드를 참조하는 형태이며, 'DBTest' 클래스에서는 'AccessConnection'을 인스턴스 받아 필드의 값을 참조하고 있다. 이와 같은 경우 실질적으로 'filename'이란 필드는 'AccessConnection' 클래스에서만 사용된다고 볼 수 있다. 즉 상위에서 선언된 필드가 오직 하나의 하위 클래스에서 사용되는 경우이다. 이와

〈표 7〉 미니리팩토링 요소에 사용될 선행조건(pre-condition)

OCL 정의	설 명
context Project def : hasSuperClass(cName : String) : Boolean =self.getClass(cName).superclassType->notEmpty()	명시된 클래스의 상위클래스가 존재하는가?
context Project def : hasClass(cName : String) : Boolean =self.getClassName()->includes(cName)	명시된 클래스가 이미 존재하는가?
context Project def : hasFieldName(cName : String, fName : String) : Boolean =self.getFieldsName(cName)->includes(fName)	해당되는 필드이름이 이미 존재 하는가?
context Project def: hasMethodsName(cName : String,mName : String) : Boolean =self.getMethodsName(cName)->includes(mName)	해당되는 메소드가 이미 존재하는가?
context Project def : hasParameterName(cName : String, mName : String, pName:String, pType : String) : Boolean =self.getParametersName(cName,mName,pType)->includes(pName)	해당되는 인자의 이름이 이미 존재하는가?
context Project def : getSuperClassName(cName : String) : String =self.getClass(cName).superclassType->select(oclIsTypeOf(SimpleType)).oclAsType(SimpleType).fullname.fullyQualifiedName->asSequence()->first()	상위 클래스의 이름을 얻어옴
context Project def : getFieldsName(cName : String) : Sequence(String) =self.getClass(cName).bodyDeclarations->select(oclIsTypeOf(FieldDeclaration)).oclAsType(FieldDeclaration).fragments.name.fullyQualifiedName->asSequence()	명시된 클래스의 필드이름들을 얻어옴
context Project def : getMethodsName(cName : String) : Sequence(String) =self.getClass(cName).bodyDeclarations->select(oclIsTypeOf(MethodDeclaration)).oclAsType(MethodDeclaration).fullname.fullyQualifiedName	명시된 클래스에 선언된 메소드의 이름을 얻어옴
context Project def : getParametersName(cName : String,mName : String,pType : String) : Sequence(String) =self.getClass(cName).getMethod(mName).parameters->select(type->select(oclIsTypeOf(SimpleType)).oclAsType(SimpleType).name.fullyQualifiedName->includes(pType)).name.fullyQualifiedName	명시된 메소드의 인자값의 이름을 얻어옴

(그림 7) 미니리팩토링요소에 OCL로 작성된 선행조건을 적용하기 위한 소스코드



가) 클래스 다이어그램

```
package Example;
import java.sql.Connection;
import java.sql.Statement;
abstract class DBConnection {
  protected Connection conn;
  protected Statement stmt;
  public String filename;
  public DBConnection() {}
  public abstract void open() throws Exception;
  public void close() throws Exception {
    if(stmt!=null) stmt.close();
    if(conn!=null) conn.close();
  }
  public Statement getStatement() throws Exception {
    return stmt;
  }
}

package Example;
import java.sql.*;
import sun.jdbc.odbc.*;
class AccessConnection  extends DBConnection {
  private String url;
  public AccessConnection(String filename) {
    this.filename = filename;
    url="jdbc:odbc:"+filename;
  }
  public void open() throws Exception {
   DriverManager.registerDriver(new JdbcOdbcDriver());
   conn=DriverManager.getConnection(url);
   stmt=conn.createStatement();
  }
}

package Example;
public class DBTest {
  public static void main(String args[]){
    AccessConnection access
    = new AccessConnection("javaaccess");
    System.out.println(access.filename);
  }
}
```

나) 소스코드

(그림 8) 데이터베이스 연결자 예제 프로그램

같은 경우 상위 클래스에 선언된 필드를 해당 하위 클래스로 옮기는(Push Down Field) 리팩토링을 적용한다.

이러한 Push Down Field 코드스멜 탐지와 리팩토링을 본 논문에서 제안한 방법을 이용하여 다음과 같은 절차로 진행한다.

① 소스코드를 JavaAST기반의 XMI문서로 변환한다. (4.1절)
② 산출된 각 클래스마다의 XMI문서에 바인딩 정보와 통합된 JavaEAST기반의 XMI문서로 변환한다. (4.1절)
③ OCL을 이용하여 Push Down Field 코드스멜을 명세한다. (4.2절)
④ OCL Interpreter을 이용하여 자동으로 코드스멜을 탐지한다. (4.2절)
⑤ 탐지된 정보를 이용하여 리팩토링을 적용한다. (4.3절)
⑥ XMI문서를 입력으로 하여 최종적으로 변경된 자바소

스코드를 산출한다. (4.3절)

4.1 JavaEAST기반의 XMI문서 산출

본 논문에서 구현한 Java to AST Model Generator(XMI Generator)를 이용하여 JavaAST기반의 XMI문서를 산출하고 XMI Integrator를 이용하여 산출된 문서를 통합한다. 여기에 소스코드를 컴파일한 결과인 바이트코드(.class 파일)로부터 바인딩 정보를 가져와 최종적으로 (그림 9)와 같이 하나의 XMI문서를 산출한다. (그림 9)의 밑줄 부분은 'DBConnection'에 선언된 'filename'필드를 어떤 클래스의 어떤 메소드에서 사용되며, 또한 어떤 형태로 사용되고 있는지를 나타내고 있다.

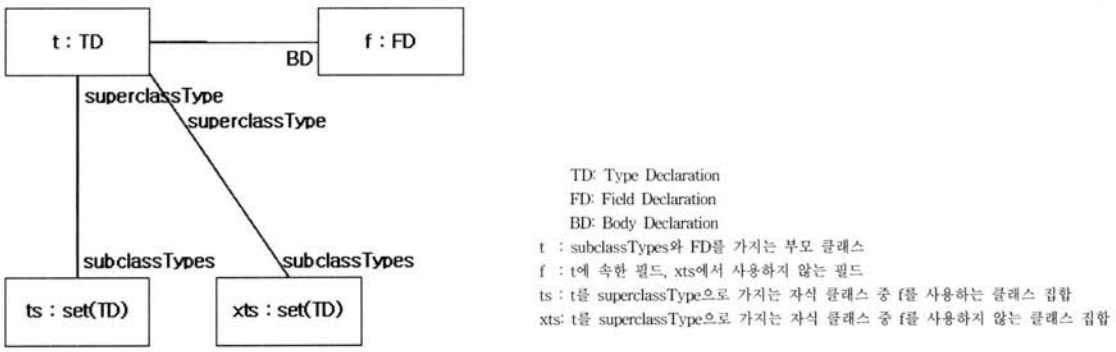
4.2 OCL을 이용한 Push Down Field 코드스멜의 명세

Push Down Field 코드스멜을 정의하기 위해서는 (그림 10)과 같이 클래스와 필드간의 관계가 성립되어야 한다.

(그림 10)에서 표현한 내용을 기반으로 Push Down Field

```
<?xml version="1.0" encoding="ASCII"?>
<Project xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="JavaEAbstractSyntax5" xsi:schemaLocation="JavaEAbstractSyntax5 JavaEAbstractSyntax5.ecore">
  <compilationUnits>
    <types xsi:type="TypeDeclaration" packageMemberTypeDeclaration="true">
                                        ......
      <bodyDeclarations xsi:type="FieldDeclaration">
        <modifiers xsi:type="Modifier" public="true"/>
        <fragments>
          <name fullyQualifiedName="filename" identifier="filename" declaration="true"/>
          <declarationBinding key="LExample/DBConnection;.filename)Ljava/lang/String;"
          type="Ljava/lang/String;;" declarationClass="LExample/DBConnection;"/>
        </fragments>
        <type xsi:type="SimpleType">
          <name xsi:type="SimpleName" fullyQualifiedName="String" identifier="String"/>
        </type>
      </bodyDeclarations>

                                        ......
      <name fullyQualifiedName="DBConnection" identifier="DBConnection" declaration="true"/>
      <fullname xsi:type="SimpleName" fullyQualifiedName="Example.DBConnection" identifier="Example.DBConnection"/>
    </types>
  </compilationUnits>

                                        ......

  <compilationUnits>
    <types xsi:type="TypeDeclaration" packageMemberTypeDeclaration="true">
                                        ......
      <leftHandSide xsi:type="FieldAccess">
        <expression xsi:type="ThisExpression"/>
        <name fullyQualifiedName="filename" identifier="filename">
          <expressionBinding key="LExample/DBConnection;.filename)Ljava/lang/String;"
              usedClass="LExample/AccessConnection;" usedMethod="LExample/AccessConnection;.()V"/>
        </name>
      </leftHandSide>
                                        ......
      <name fullyQualifiedName="AccessConnection" identifier="AccessConnection" declaration="true"/>
      <fullname xsi:type="SimpleName" fullyQualifiedName="Example.AccessConnection" identifier="Example.AccessConnection"/>
    </types>
  </compilationUnits>

                                        ......

  <compilationUnits>
    <types xsi:type="TypeDeclaration" packageMemberTypeDeclaration="true">
                                        ......
      <arguments xsi:type="QualifiedName" fullyQualifiedName="access.filename">
        <name fullyQualifiedName="filename" identifier="filename"/>
        <qualifier xsi:type="SimpleName" fullyQualifiedName="access" identifier="access">
          <expressionBinding key="LExample/DBTest;.main([Ljava/lang/String;)V#access"
              usedClass="LExample/DBTest;" usedMethod="LExample/DBTest;.main([Ljava/lang/String;)V"/>
        </qualifier>
        <expressionBinding key="LExample/DBConnection;.filename)Ljava/lang/String;"
              usedClass="LExample/DBTest;" usedMethod="LExample/DBTest;.main([Ljava/lang/String;)V"/>
      </arguments>
                                        ......
      <name fullyQualifiedName="DBTest" identifier="DBTest" declaration="true"/>
      <fullname xsi:type="SimpleName" fullyQualifiedName="DBTest" identifier="DBTest"/>
    </types>
  </compilationUnits>
</Project>
```

(그림 9) 소스코드와 바이트코드로부터 산출된 JavaEAST기반의 XMI문서



(그림 10) Push Down Field 코드스멜 명세

```
context Project
def : getPushDownFieldDetection()
:Sequence(Tuple(superClass : String, field : String, subClass : String))
=self.getSuperClasses()->collect(sType |
sType.getFieldDeclarationKey()->collect( sFieldKey |
if self.getUsedClassOfSimpleName(sFieldKey)->includes(sType.getTypeKey) then
  null
else if self.getTypeOfQualifiedName(sFieldKey)->includes(sType.getTypeKey) then
  null
  else if
 self.getUsedClassOfSimpleName(sFieldKey)->includesAll(self.getSubclassTypeKey(sType)) then
      null
      else
 (self.getUsedClassOfSimpleName(sFieldKey)->intersection(self.getSubclassTypeKey(sType))
  ->union(self.getTypeOfQualifiedName(sFieldKey)) -
      self.getSubclassTypes(sType)->iterate(acc : TypeDeclaration ;
      result : Set(String) = Set{} |
      if acc.getFieldNames()->includes(self.getVariableName(sFieldKey)) then
          acc.getTypeKey->union(result)
      else
          result
      endif
       ))->collect(target |
       Tuple{superClass = sType.getClassName(), field = self.getVariableName(sFieldKey),
       subClass = self.getClassNameByKey(target)}
       )
       endif
   endif
endif
))->excluding(null)->asSet()->asSequence()
```

(그림 11) OCL을 이용한 Push Down Field 코드스멜 명세

코드스멜을 찾기 위해서는 다음과 같은 조건에 만족하는 클래스 t와 필드 f를 찾는다.

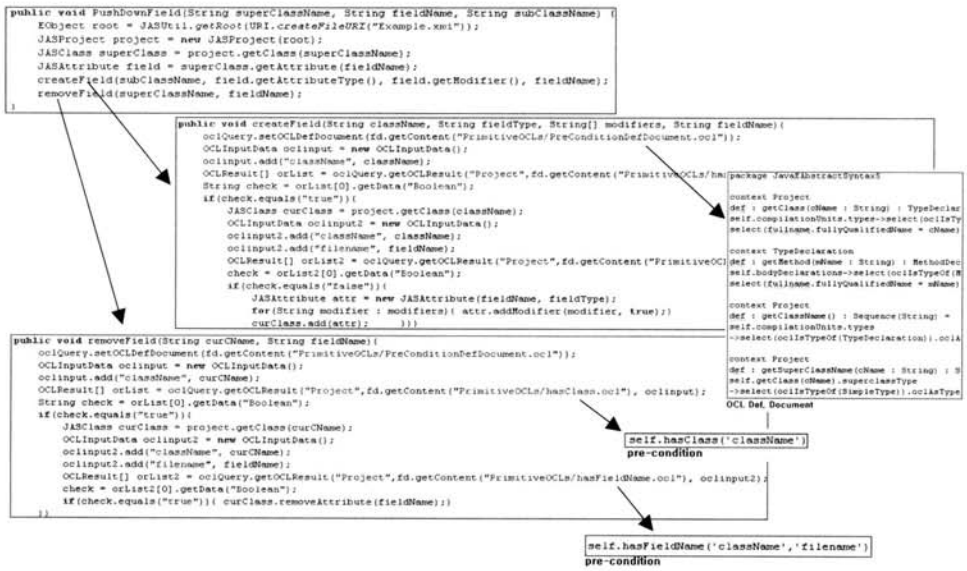
① 자식클래스와 필드를 가지는 수퍼 클래스 t를 찾는다.
② t가 가지는 필드 중 다음의 조건을 만족하는 f를 찾는다.
  가) t에서 사용하지 않는 필드이어야 한다.
  나) xts에서 사용하지 않는 필드이어야 한다.
  다) t를 통하여 사용하지 않는 필드이어야 한다.
  라) xts를 통하여 사용하지 않은 필드이어야 한다.
  마) ts에서 재정의(overriding)하지 않아야 한다.

산출된 XMI문서를 모델로하여 이와 같은 조건을 OCL로 작성하면 (그림 11)과 같이 정의될 수 있다. 여기에서 사용된 OCL 함수들은 본 논문의 3.2절에서 이미 정의한 클래스들간의 관계에서 도출 가능한 OCL 함수들이며, 이런 함수들의 조합으로 Push Down Field 코드스멜을 탐지하기 위한 OCL을 정의한다.

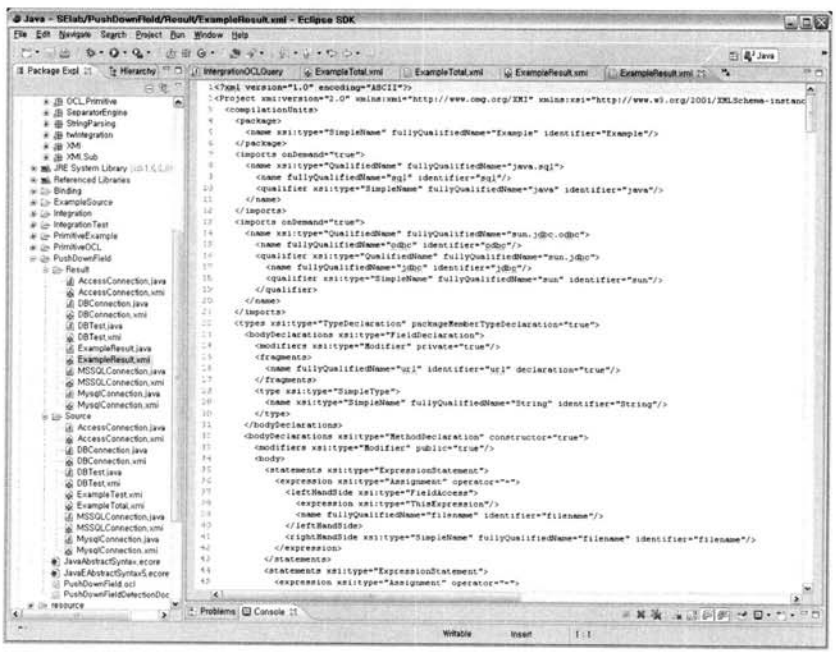
(그림 11)과 같이 정의된 OCL 문장을 OCL 번역기를 거쳐 실행을 하면 밑줄 부분과 같이 수퍼클래스 이름, 필드이름, 옮길 하위 클래스 이름을 튜플로하여 시퀀스 타입으로 리턴 받는다. 따라서 적용대상인 데이터베이스 연결자 예제 프로그램을 대상으로 실행하면 ("Example.DBConnection", "filename", "Example.AccessConnection")라는 결과값을 얻는다.

4.3 리팩토링 및 소스코드 산출

4.2절에서 명세된 OCL의 실행결과로 얻은 결과값으로 실제 리팩토링을 적용하기 위해서는 3.4절에서 정의된 미니리팩토링 요소를 조합하여 (그림 12)와 같은 PushDownField(String



(그림 12) Push Down Field 리팩토링 구현(미니리팩토링 요소의 사용과 OCL로 작성된 선행조건 사용)



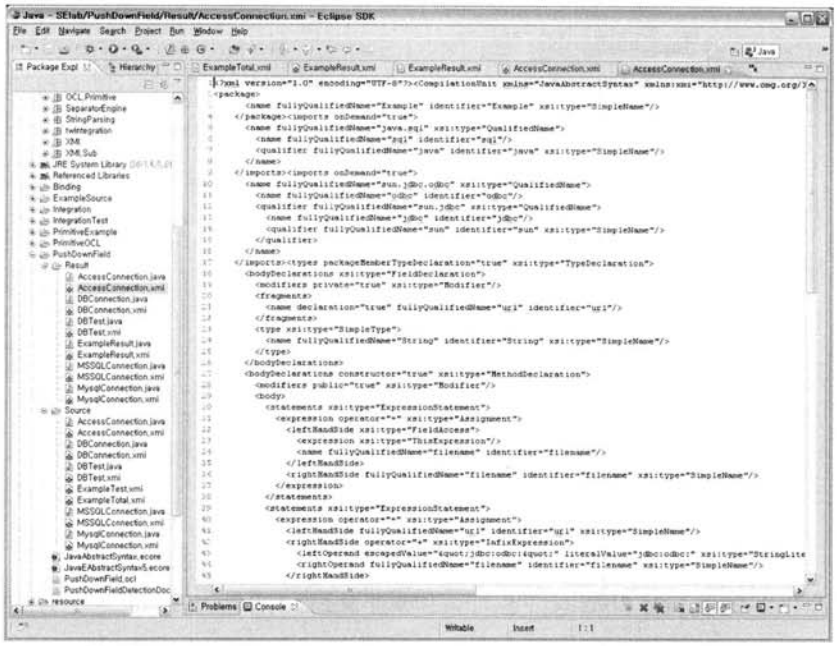
(그림 13) 리팩토링을 적용한 후의 JavaEAST기반의 XMI문서

superClassName, String fieldName, String subClassName) 메소드를 구현하여 적용한다. 미니리팩토링 요소에는 해당 클래스와 필드가 이미 존재하는지를 검사하는 OCL 문장을 포함하여 구현한다.

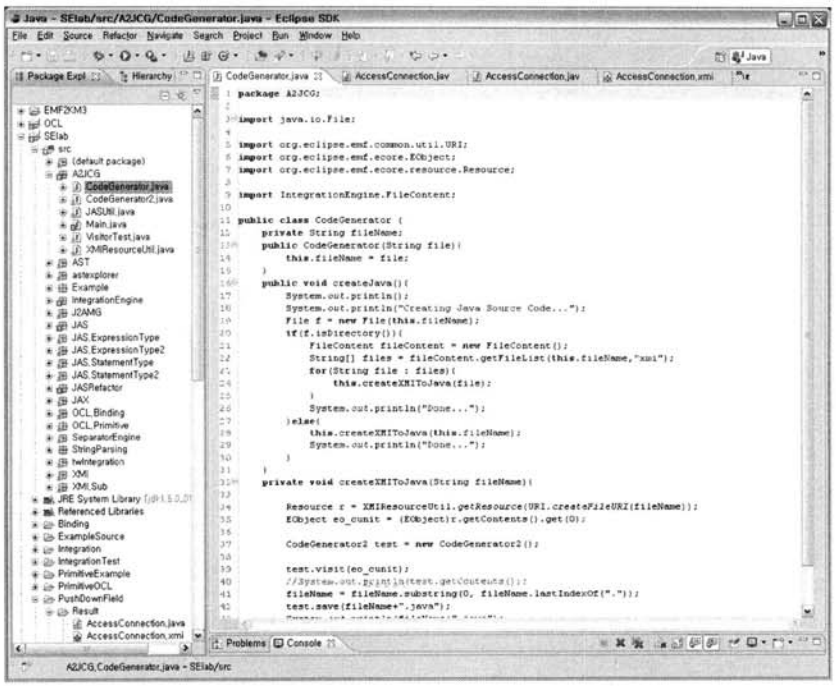
3.4절에서 정의된 미니리팩토링 요소는 JavaEAST기반으로 작성된 XMI문서에 접근하는 JAS API를 이용하여 구현되었다. 따라서 (그림 12)와 같이 구현된 메소드를 실행하면 (그림 13)과 같이 변경된 JavaEAST기반의 XMI문서가 산출되며, 이를 XMI Separator를 통해 JavaAST기반의 XMI문서로 변환한다(그림 14). 그리고 코드변환기(Code Generator, 그림 15)를 통해 최종적으로 수정된 소스코드를 얻는다.

(그림 16)은 다음과 같은 단계를 본 논문에서 구현한 클래스들을 이용하여 작성한 소스코드이다. 최종 산출된 결과물은 리팩토링이 적용된 후의 수정된 자바소스코드로 (그림 17)의 나)와 같다. 밑줄 부분인 'public String filename'이란 필드가 'DBConnection'클래스에서 'AccessConnection'으로 옮겨진(Push Down Field) 모습이다.

① 자바소스코드를 JavaAST기반의 XMI문서를 산출한다.
② JavaAST기반의 문서를 JavaEAST기반의 XMI문서로



(그림 14) XMI Separator를 이용하여 변환된 JavaAST기반의 XMI 문서



(그림 15) XMI문서를 자바소스코드로 변환하는 Code Generator

산출한다.

③ 명세된 OCL을 이용하여 코드스멜을 탐지한다.

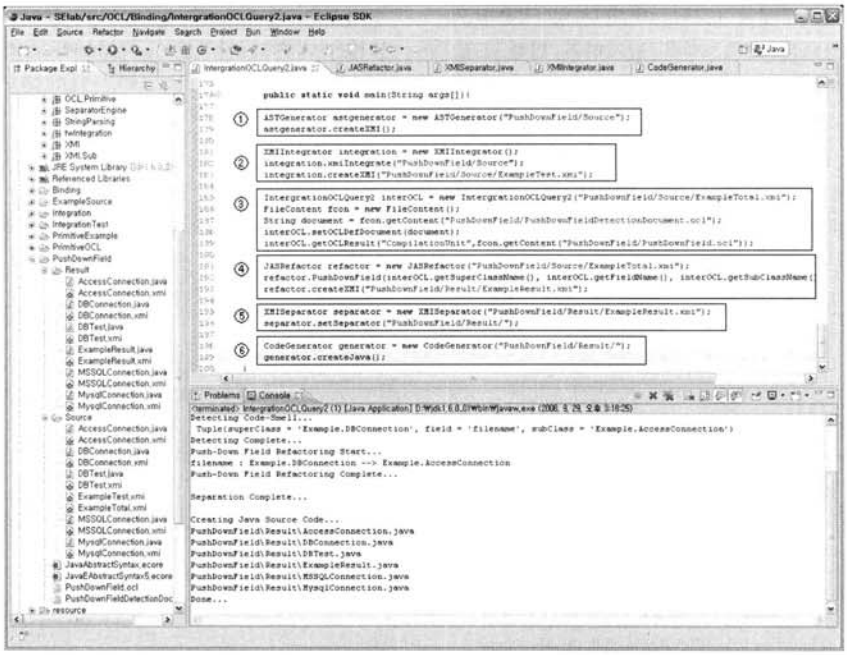
④ 탐지된 정보를 이용하여 리팩토링을 적용한다.

⑤ JavaEAST기반의 문서를 JavaAST기반의 문서로 변환한다.

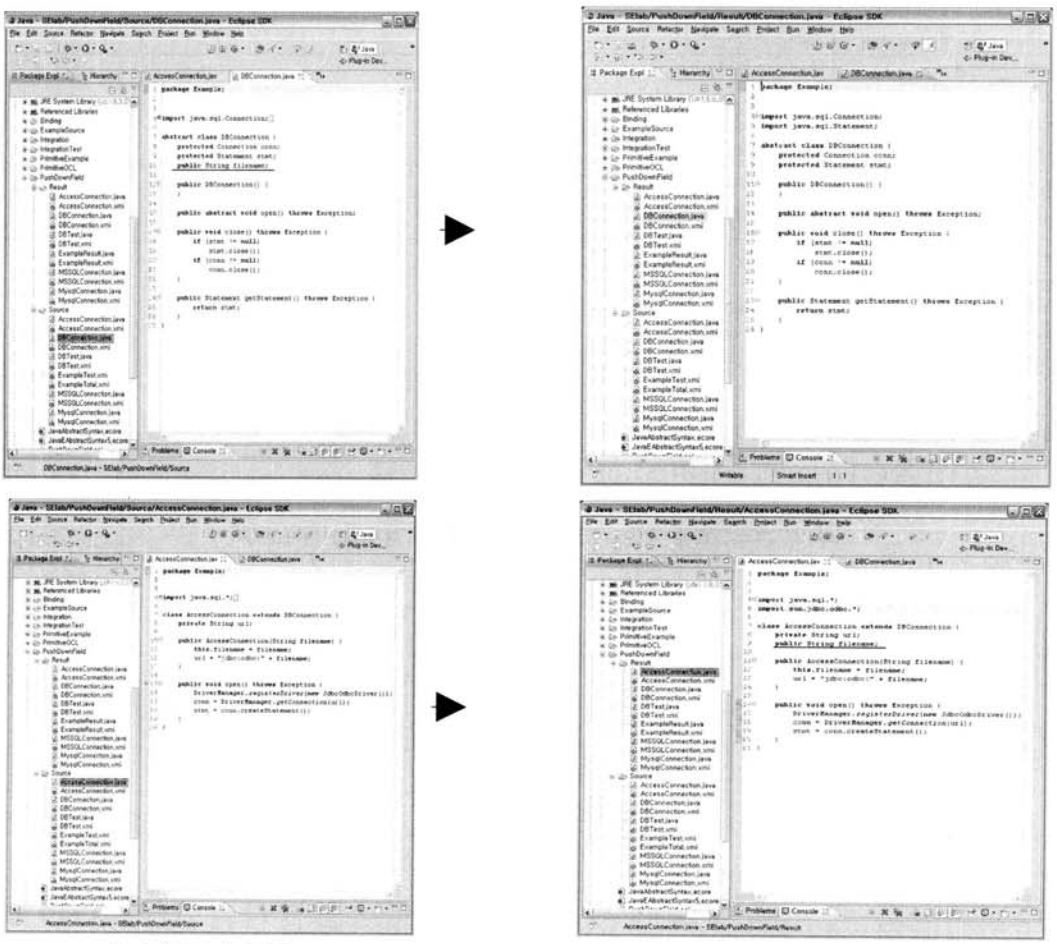
⑥ JavaAST기반의 문서를 자바 소스코드로 변환한다.

이번 장에서는 본 논문에서 제안한 프레임워크를 기반으로 하여 코드스멜 자동 탐지와 리팩토링을 적용해 보았다. 적용대상은 (그림 8)과 같은 데이터베이스 연결자 예제 프로그램으로 Push Down Field 코드스멜을 자동으로 탐지하고, 리팩토링하는 과정과 절차를 6단계로 나누어 적용하였다. 이와 같이 OCL로 코드스멜을 정의하고, OCL번역기를 통한 코드스멜의 자동탐지, 그리고 리팩토링을 자동화함으로써 수정된 소스코드를 산출할 수 있다.

자동으로 코드스멜을 탐지하기 위해서는 (그림 11)과 같은



(그림 16) Push Down Field 코드스멜 탐지와 리팩토링 실행화면



가) 리팩토링 적용전의 소스코드 나) 리팩토링 적용후의 소스코드

(그림 17) 리팩토링 전후의 소스코드 변화

OCL로 작성된 코드스멜 명세를 (그림 16-③)처럼 적용함으로써 가능하다. 이는 새로운 코드스멜을 탐지하기 위해 프로그램 개발단계 없이 OCL만을 이용하여 명세화함으로써 탐지 가능하다는 장점을 가진다.

<표 8>은 Remove Field에 대한 코드스멜을 OCL로 정의하였으며, 이러한 명세만을 이용하여 어느 곳에서도 참조하지 않는 필드를 자동탐지 할 수 있다.

이러한 결과로 본 논문에서 제안한 프레임워크가 가지는 장점은 기존의 연구들과 비교하여 <표 9>, <표 10>과 같이 요약할 수 있다.

5. 결 론

리팩토링은 시스템의 기능을 유지하며 소스코드의 확장

〈표 8〉 Remove Field 코드스멜 명세

OCL 명세	의미
context Project def : allTD : Set(TypeDeclaration) = self.compilationUnits.types->collect(oclAsType(TypeDeclaration)) ->select(bodyDeclarations->select(oclIsTypeOf(FieldDeclaration)) ->notEmpty())->asSet()	필드를 가진 모든 클래스를 가져온다
context Project def : getVariableName(fKey : String) : String = VariableDeclarationFragment.allInstances()->select(declarationBinding ->notEmpty()) ->select(declarationBinding.key->includes(fKey)).name.fullyQualifiedName ->asSequence()->first()	클래스에 선언된 필드의 이름을 가져온다
context Project def: getUsedClassOfSimpleName() : Set(String) = SimpleName.allInstances()->select(expressionBinding ->notEmpty()).expressionBinding.key->asSet()	모든 expressionBiding값을 가져온다
context TypeDeclaration def : getFieldDeclarationKey() : Set(String) = self.bodyDeclarations->select(oclIsTypeOf(FieldDeclaration)) .oclAsType(FieldDeclaration).fragments.declarationBinding.key->asSet()	클래스에 선언된 필드의 키값을 가져온다
context TypeDeclaration def : getClassName() : String =self.fullname.fullyQualifiedName	클래스의 full-name을 가져온다
context Project def : getRemoveFieldDection() : Sequence(Tuple(className : String,fieldName : String)) = self.allTD->collect(td | td.getFieldDeclarationKey()->collect(fkey | if self.getUsedClassOfSimpleName()->includes(fkey) then null else Tuple{className = td.getClassName(),fieldName = self.getVariableName(fkey)} endif))->excluding(null)->asSequence()	삭제될 필드가 선언된 클래스이름과 필드의 이름을 가져온다

〈표 9〉 코드스멜에 대한 기존 연구들과 제안한 프레임워크의 비교

항목	기존의 연구들			제안한 프레임워크에서의 해결방법
	Emden의 연구[14]	Holt의 연구[15]	Slinger의 연구[5]	
단순 코드스멜 탐지	자바의 문맥을 검사하여 탐지	-	제안한 5단계로 플러그인을 개발하여 탐지	OCL을 이용하여 명세하고, OCL Component를 이용하여 자동 탐지
유도된 코드스멜 탐지	-	클래스간의 관계정보를 Fact로 기술	클래스간의 관계정보를 Fact로 기술	
클래스간의 관계정보 추출 방법	-	Facts Repository를 구축하여 Grok 스크립트를 이용하여 정보추출	Fact Repository를 구축하여 Grok 스크립트를 이용하여 정보추출	JavaEAST기반의 소스코드모델에서 OCL을 이용하여 정보추출
탐지대상 소스코드의 수정에 따른 처리 방법	-	Facts Repository를 업데이트하여 새로운 정보를 추출	Facts Repository를 업데이트하여 새로운 정보를 추출	저장소를 사용하지 않고 JavaEAST기반으로 표현된 소스코드에서 직접적으로 새로운 정보 추출
새로운 코드스멜 요소 추가 및 수정에 따른 처리 방법	새로운 코드스멜 추가 및 수정 불가	Grok스크립트를 이용하여 코드스멜을 작성하고, Fact들을 분석	플러그인 개발 또는 Grok 스크립트를 이용하여 Fact 분석	OCL만을 이용하여 코드스멜을 추가 및 수정

〈표 10〉 리팩토링에 대한 기존 연구들과 제안한 프레임워크의 비교

항목	기존의 연구들			제안한 프레임워크에서의 해결방법
	Medonca의 연구[16]	Cinneide의 연구[20]	Marticorena의 연구[21]	
안전한 리팩토링을 위한 방법	AFs를 사용하여 표현	자연어로 선행조건을 표현	선행조건을 클래스의 메소드로 구현	OCL을 이용하여 안전한 리팩토링을 위한 선행조건 명세
리팩토링 수행방법	XQuery를 이용한 XML문서의 수정	미니 리팩토링요소로 정의하고, 이를 조합하여 리팩토링 수행	미니 리팩토링 요소를 소형화하여, 이를 재사용	미니 리팩토링에 대한 API구현, 이를 조합하여 리팩토링 수행
리팩토링 수행과 선행조건의 결합	리팩토링 수행과 종속적 XQuery에 포함	-	이미 컴파일된 클래스를 참조하여 수행	미니 리팩토링요소에 별도로 작성된 OCL을 읽어 수행

성, 모듈화, 재사용성, 유지보수성과 같은 품질을 개선하는 과정으로 일종의 소프트웨어를 변경하는 작업이다. 이러한 리팩토링을 적용하여 기존 소스코드를 개선하기 위해서는 개선할 사항이 무엇인지를 아는것이 우선이다. 이를 위해 Martin Fowler와 Kent Beck은 코드스멜을 식별할 수 있는 방법을 제시 하였다. 또한 코드스멜을 탐지해 내어, 어디에 어떤 리팩토링을 적용할 것인가를 결정하는 문제와 관련된 몇몇 연구가 발표되었다. 그러나 이러한 연구들은 코드스멜에 대한 명확한 표현이 부족하고, 한정된 코드스멜만을 탐지하는 단점이 있다. 그리고 리팩토링을 적용할 경우 행위보존을 위한 선행조건들의 표현방법이 리팩토링 절차에 포함되어 있거나 정형화되지 않아 행위보존의 모호함이 발생되는 단점을 가지고 있다.

이에 본 논문에서는 OCL을 이용하여 코드스멜의 정보를 정확히 명세화하고, OCL번역기를 통해 코드스멜을 자동으로 탐지하여 리팩토링하는 프레임워크를 제안하였다.

이를 위하여 클래스들간의 관계, 속성 및 메소드에 대한 바인딩 정보를 표현하기 위한 JavaEAST모델을 제안하고, 이를 기반으로 코드스멜을 OCL로 정의하였다. 이는 나쁜 냄새에 대한 명확한 명세가 이루어진다는 장점 뿐만 아니라, 향후 새로운 코드스멜에 대한 추가 및 수정사항 발생시 OCL만을 정의하여 가능하게 함으로써 생산성 향상을 가져온다. 특히 유도된 코드스멜의 명세는 이미 정의된 클래스들간의 정보를 추출하는 OCL정의를 조합하여 생성되므로 소프트웨어 재사용이라는 장점을 가진다. 이러한 과정을 통하여 탐지된 코드스멜은 리팩토링 절차를 통하여 소스코드가 수정된다. 리팩토링 절차는 본 논문에선 제안하고 구현한 EMF 기반의 JAS API를 통하여 이루어진다. 또한 리팩토링은 미니 리팩토링의 조합으로 가능하므로 리팩토링 요소를 재 사용할 수 있으며, 새로운 리팩토링의 추가 또는 변경시 효율적이다.

본 논문에서 제안하고 구현한 OCL을 이용한 코드스멜 자동탐지와 리팩토링을 위한 프레임워크는 OCL을 이용하기 때문에 소프트웨어의 재사용과 생산성 향상이라는 장점을 가진다. 이는 보다 유연한 리팩토링 자동화 도구의 개발에 필요한 중요한 선행연구가 될 것이다. 향후 다양한 코드스멜에 대한 정의와 리팩토링을 적용해 봄으로써 메타모델의 검증과 더불어 보다 향상된 프레임워크를 개발해야 하겠다.

참 고 문 헌

[1] Fowler, M. Refactoring. "Improving the Design of Existing Programs," Addison-Wesley. 1999.

[2] Mens, T., Tourwe, T. "A Survey of Software Refactoring," IEEE Transactions on Software Engineering, February, Vol.30, No.2. pp.126-139, 2004,

[3] Joshua Kerievsky, "Refactoring to Patterns," Addison Wesley, 2005.

[4] Martin Fowler, "Refactoring:Improving the Design Existing Code," Addison Wesley, 1999.

[5] Stefan Slinger, "Code Smell Detection in Eclipse," Delft University of Technology, March, 2005.

[6] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin, "Automated Support for Program Refactoring using Invariants," In Proc. Int. Conf. On Software Maintenance, pp.736-743, 2001.

[7] Stephane Ducasse, Matthias Rieger, and Serge Demeyer, "A language independent approach for detecting duplicated code," In Hongji Yang and Lee white, editors, Proc. Int'l Conf. Software Maintenance, IEEE Computer Society Press, pp.109-118, September, 1999.

[8] Frank Simon, Frank SteinbruUckner, and Clause Lewerent, "Metrics Based Refactoring," In Proc. 5th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, pp.30-38, 2001.

[9] Object Management Group, "Object Constraint Language Specification, Version 2.0," http://www.omg.org/technology/documents/formal/ocl.htm.

[10] MoDisco, "MoDisco Tool- Java Abstract Syntax Discovery Tool," http://www.eclipse.org/gmt/modisco/toolBox/JavaAbstractSyntax/.

[11] Object Management Group, "MOF 2.0/XMI Mapping Specification, V2.1.1," http://www.omg.org/technology/documents/formal/xmi.htm.

[12] Eclipse, "OCL for EMF," http://www.eclipseplugincentral.com/Web_Links-index-req-viewlink-cid-200.html#, updated 2004.

[13] Eclipse, "Eclipse Modeling Framework Project," http://www.eclipse.org/modeling/emf/?project=emf.

[14] E. van Emden and L.Moonen, "Java Quality Assurance

by Detecting Code Smells," In Proc. 9th Working Conference on Reverse Engineering IEEE Computer Society Press, October, 2002.

[15] R.C. Holt, "Structural Manipulations of Software Architecture using Tarski Relational Algebra," In Proc. 5th Working Conference on Reverse Engineering (WCRE'98), pp.210-219, 1998.

[16] Narbor C. Mendonca, Paulo Henrique M. Maia, Leonardo A. Fonesca, Rossana M.C. Andrade, "RefaX : A Refactoring Framework Based on XML," Proceedings of the 20th IEEE International Conference on Software Maintenamce (ICSM'04), 2004.

[17] Mammas, E. and Kontogiannis, C., "Towards Portable Source Code Representation using XML," in Proc. of the 7th Working Conference on Reverse Engineering (WCRE, 00), Brisbane, Australia, pp.172-180, November, 2000.

[18] W3C, "XQuery 1.0 : An XML Query Language," W3C Recommendation Draft 12 November 2003, Available at http://www.w3c.org/TR/xquery/, 2003.

[19] Roberts, D.B, "Practical Analysis for Refactoring," Ph.D. Thesis, Unversity of Illinois at Urbana-Chanpaign, 1999

[20] Mel O Cinneide, Automated Application of Design Patterns : A Refactoring Approach, Department of Computer Science, Trinity College Dublin, October, 2000.

[21] Raul Marticorena, Carlos Lopez, Yania Crespo, Francisco Javier Perez, "Reuse based Refactoring Tools," Proceedings of 1st Workshop Refactorign Tools(WRT'07), pp.21-22, July, 2007.

김 태 웅

e-mail : twkim@cs.inje.ac.kr
1994년 인제대학교 전산학과(이학사)
1998년 인제대학교 전산학과(이학석사)
2002년 인제대학교 전산학과(박사수료)
1999년~2006년 동의과학대학 겸임교수
2007년~현 재 인제대학교 컴퓨터공학부 연구교수
관심분야 : 소프트웨어 공학, 소프트웨어 개발, Refactoring, Code Smell Detection 등

김 태 공

e-mail : ktg@cs.inje.ac.kr
1983년 서울대학교 계산통계학과(학사)
1985년 서울대학교대학원 계산통계학과 전산과학전공(이학석사)
1994년 서울대학교대학원 계산통계학과 전산과학전공(이학박사)
2003년~2004년 The University of Texas at Dallas 방문 교수
1990년~현 재 인제대학교 컴퓨터공학부 교수
관심분야 : 소프트웨어 공학, Model Engineering, AOSD, Generative Programming, Code Smell Detection, Refactoring