DOI: 10.3745/KIPSTD.2011.18D.1.023

선행조건이 명세되어 있지 않은 복합 리팩토링에 대한 조립성 검사

김 경 민[†] · 장 필 재[††] · 김 태 공[†††]

요 약

최근의 소프트웨어 개발에서는 리팩토링이 활발하게 이용되고 있다. 리팩토링에 대한 관심이 높아지면서 요소 리팩토링을 정의해서 이들의 조립을 통해 더 큰 종류의 복합 리팩토링을 처리하려는 연구들이 많이 진행되고 있다. 복합 리팩토링은 조합된 요소 리팩토링의 순서대로 처리되기 때문에 프로그램에 실제 적용하기에 앞서 적용 가능성을 판단하는 것이 중요하다. 기존 연구들에서는 요소 리팩토링들의 선·후행조건들로부터 복합 리팩토링에 대한 선행조건을 계산하여 적용 가능성을 판단하고 있다. 이것은 선행조건이 정의되어 있는 복합 리팩토링만 판별할 수 있으며 선행조건이 정의되어 있지 않은 복합 리팩토링에 대해서는 적용 가능성을 판단할 수 없는 문제점이 있다.

본 연구에서는 복합 리팩토링의 명세를 추가로 정의 하지 않고 요소 리팩토링만으로 조립 가능 여부를 판단하고자 한다. 이를 위해 요소 리팩토링을 명세하는 방법과 조립성 검사 방법을 제안한다. 이를 기반으로 프로토타입 도구를 개발해 보고, 적용사례를 통해 선행조건이 명세되어 있지 않은 복합 리팩토링들의 조립성 검사를 확인해봄으로써 본 연구의 효용성을 검토해본다.

키워드 : 요소 리팩토링, 복합 리팩토링, 조립성 검사, 선행조건, 후행조건 명세, OCL, EMF

A Composition Check of Composite Refactorings Not Having a Specification of Precondition

Kim Kyung Min[†] · Jang Pil Jae[††] · Kim Tae Gong[†††]

ABSTRACT

Refactoring has been actively used in recent software developments. Many studies on the processing of more large scaled composite refactorings have been conducted through the composition of elementary refactorings. It is important to verify the possibility of composition before the refactoring is performed, because the composite refactorings are processed to the sequence of composed elementary refactorings. In conventional studies, they verify the possibility of composition using the precondition of composite refactorings which are computed from the precondition and postcondition of elementary refactorings. They can not verify the possibility of composition in case which composite refactorings do not have a specification of precondition.

Thus, we plan to verify the possibility of composition by using the elementary refactorings only without any additional definitions of the preconditions of composite refactorings. To achieve this goal, we proposes a specification method of elementary refactorings and a method for the composition check of refactorings. Then, we develop a prototype tool based on these methods. In addition, we verify the efficiency of our methods through case studies.

Keywords : Elementary Refactoring, Composite Refactoring, Composition Check, Precondition, Postcondition, OCL, EMF

1. 서 론

최근 리팩토링이 소프트웨어 개발에서 활발하게 이용되고 있다. 리팩토링은 프로그램의 행위를 보존하면서 프로그램의 가독성, 구조, 성능, 유지보수성 등을 향상시키는 방법이다. 시스템의 기능을 유지하면서 시스템의 이해도를 높임과

† 준 회 원 : 동의과학대학 컴퓨터정보계열 겸임교수
†† 정 회 원 : 태광엠티씨(주) R&D팀 주임
††† 정 회 원 : 인제대학교 컴퓨터공학부 교수
논문접수 : 2010년 5월 24일
수 정 일 : 1차 2010년 8월 2일
심사완료 : 2010년 9월 1일

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].

이처럼 리팩토링에 대한 관심이 높아지면서 기존 요소 리팩토링을 이용하여 이들의 조립을 통해 복잡하고 다양한 더 큰 종류의 복합 리팩토링을 처리하기 위한 많은 연구들[3,4,5,6,7]이 진행되고 있다. 리팩토링을 조립하기 위해서는 먼저 리팩토링 적용 전에 적용 가능성을 판단하는 것은 중요하다. 특히 여러 개의 요소 리팩토링들로 구성되어 있는 복합 리팩토링의 경우 임의의 요소 리팩토링 어느 하나라도 선행조건이 만족되지 않으면 지금까지 적용된 요소 리팩토링들을 철회하고 처음 상태로 되돌아가야 하기 때문이다.

기존 연구들[8,9,10]에서의 조립성 검사 방법은 복합 리팩토링에 대한 선행조건을 이용하여 조립성 검사를 하고 있다. 이를 위해 요소 리팩토링들의 선·후행조건을 1차 술어 논리 기반의 오퍼레이션으로 명세하고 이를 이용하여 복합 리팩토링에 대한 선·후행조건을 계산한다. 이전 요소 리팩토링의 후행조건을 반영시키면서 선행조건들의 논리곱이나 논리합으로 처리하고 있다. 이것은 복합 리팩토링에 대한 선행조건만으로 조립성 검사가 가능하게 되지만, 복합 리팩토링에 대한 선행조건이 계산되어 있지 않은 경우에는 조립성 검사를 처리할 수 없게 된다. 즉 선행조건이 정의되어 있는 복합 리팩토링만 판별할 수 있으며 임의의 요소 리팩토링들로 구성되는 다양한 복합 리팩토링에 대해서는 적용 가능성을 판단할 수 없다.

본 연구에서는 복합 리팩토링에 대한 명세를 추가로 정의하지 않고 요소 리팩토링만으로 조립 가능 여부를 판단하고자 한다. 이를 위해 요소 리팩토링을 명세하는 새로운 방법을 제안하며 이를 기반으로 요소 리팩토링들을 직렬화한다. 그리고 임의의 요소 리팩토링들로 구성된 복합 리팩토링의 조립성 검사를 위해 내부적으로 프로그램의 상태를 관리함으로써 조립성을 검사하는 방법을 제안한다. 이를 기반으로 프로토타입으로 도구를 개발해보고 적용사례를 통해 선행조건이 명세되어 있지 않은 복합 리팩토링들의 조립성 검사를 확인해봄으로써 본 연구의 효용성을 검토해본다.

2. 관련연구

Roberts의 연구[8]에서는 실제 리팩토링 사용에 있어서 빠르고 신뢰할 수 있는 리팩토링 툴을 만드는 방법에 대해 제안하고 있다. 이를 위해 'Primitive Analysis Functions', 'Derived Analysis Functions'을 정의하고, 이를 이용하여 리팩토링의 선·후행 조건에 중점을 둔 1차 술어 논리를 기반으로 새로운 리팩토링 명세를 제안한다. 그리고 Smalltalk를 위한 리팩토링 툴인 'Refactoring Browser'를 개발하여 연구에서 제안한 방법들을 검증해보고 있다. Roberts의 연구에서의 리팩토링 조립은 리팩토링들 간의 의존 관계를 고려하여 조립된 복합 리팩토링의 선행 조건을 도출하고 있다. 순서적인 리팩토링들의 'chain' 내에서 요소 리팩토링의 선행조건이 이전 요소 리팩토링의 후행 조건에 의존되어 어떻게 유도 되는지를 기술하고 있다.

Cinneide의 연구[9]는 디자인 패턴을 위한 리팩토링을 제안하고 있으며 리팩토링을 적용한 후의 행위보존에 대해 기술하고 있다. 이를 위해 'Analysis Functions', 'Helper Functions', 'Primitive Refactorings'들을 정의하고, 이를 이용하여 리팩토링 조립을 통해 6종류의 'Minipatterns'들을 정의하고 있다. 'Minipatterns' 이용하여 디자인 패턴의 자동 변환을 처리하고 있다. 그리고 프로토타입의 소프트웨어 툴인 'DPT(Design Pattern Tool)'를 개발하여 제안한 방법들을 검증해보고 있다. Cinneide의 연구에서의 리팩토링 조립은 요소 리팩토링의 선·후행 조건을 이용하여 조립된 복합 리팩토링의 선·후행 조건을 도출하고 있다. 복합 리팩토링의 선행 조건은 요소 리팩토링의 선행 조건들의 논리곱(AND), 후행 조건은 요소 리팩토링들의 논리합(OR)으로 계산하며, 이 때 두 번째 요소 리팩토링의 선행 조건은 첫 번째 요소 리팩토링의 후행 조건이 반영되도록 처리하고 있다. 그리고 리팩토링 조립이 순서적으로 적용되는 'Chaining'과 반복적으로 적용되는 'Set iteration' 두 가지로 리팩토링 조립 방법을 제안하고 있다.

Kniesel의 연구[10]에서는 사용자가 리팩토링들을 만들고 수정하고 조립할 수 있는 'Refactoring Editor'라는 새로운 종류의 리팩토링 툴을 제안하고 있다. 특히 존재하는 요소 리팩토링들을 이용하여 더 큰 종류의 복합 리팩토링으로 조립하는 기능에 대해 초점을 두고 있다. 이를 위해 조립이 가능한 리팩토링들을 선언하고 프로그램 독립적으로 조립될 수 있도록 정형화된 모델을 제안하고 있다. 정형화된 모델에서는 'Conditional Transformations', 'Joint Precondition', 'Backward Transformation Description', 'Program-independent Composition' 등을 정의하여 리팩토링 조립을 설명하고 있다. 그리고 자바 프로그램을 위한 리팩토링 프레임워크인 'jConditioner'를 기반으로 'ConTraCT(Conditional Transformation Composition Tool)'라는 리팩토링 툴을 개발하여 제안하는 방법들을 실제 검증해보고 있다. Kniesel의 연구에서의 리팩토링 조립은 요소 리팩토링의 선행 조건과 'Backward Transformation Description'을 이용하여 복합 리팩토링의 선행 조건인 'Joint Precondition'을 도출하고 있다. 'Joint Precondition'은 요소 리팩토링의 선행 조건들의 논리곱으로 계산되며, Roberts나 Cinneide의 연구처럼 이전 요소 리팩토링들의 기능이 반영되도록 처리하고 있다.

이처럼 기존 연구들에서는 요소 리팩토링들의 선·후행조건들로 부터 복합 리팩토링에 대한 선행조건을 계산하여 리팩토링의 조립 가능성을 판단하고 있다. 이것은 복합 리팩

토링에 대한 선·후행조건이 명세되어 있지 않은 경우에는 조립성 검사를 확인할 수 없게 된다. 이에 본 연구에서는 복합 리팩토링의 선·후행조건을 추가로 정의 하지 않고 요소 리팩토링만으로 조립 가능 여부를 판단하고자 한다. 이를 위해 요소 리팩토링을 명세하는 방법과 리팩토링의 조립성 검사 방법을 제안한다.

3. 리팩토링의 명세

3.1 리팩토링의 명세 방법

본 연구에서는 복합 리팩토링에 대한 추가적인 선·후행조건을 정의하지 않고 조립성 검사를 처리하기 위해 리팩토링을 명세하는 새로운 방법을 제안한다.

기존 연구들[8,9,10]에서는 조립성 검사를 위해 요소 리팩토링의 명세를 1차 술어 논리 기반의 기본 조건들로 정의하고 있다. 때문에 새로운 술어(predicate)가 추가되면 기존에 명세되어 있던 모든 리팩토링의 선·후행조건이 업데이트되어야 하는 문제점이 있다.

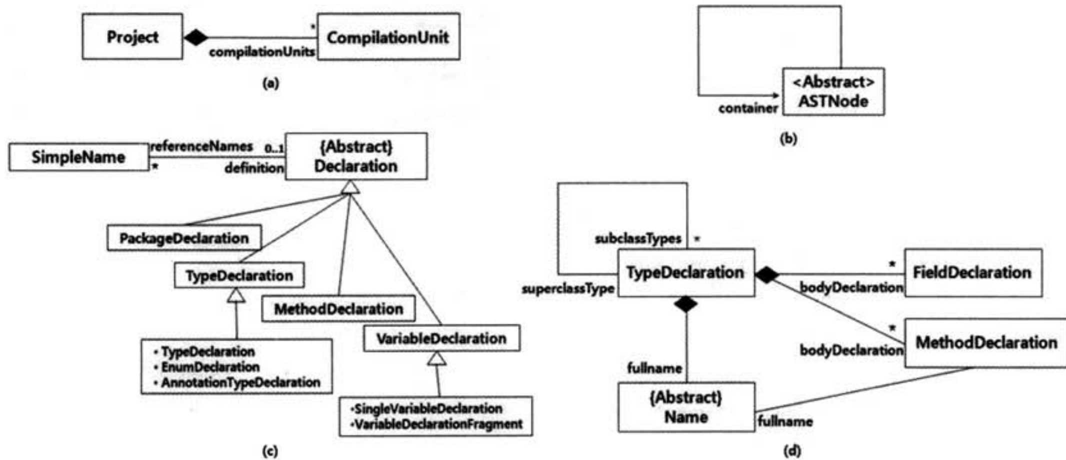
본 연구에서는 JavaEAST 메타모델을 기반으로 OCL[11]을 이용하여 리팩토링을 명세하고자 한다. 메타모델에 정의되어 있는 모델 프로퍼티를 이용하여 조건들을 정의하고 복잡한 조건들의 경우는 OCL의 'def' 기능을 이용하여 정의된 피처(Additional Feature)로 정의하여 사용한다. 기본적으로 메타모델의 모델 프로퍼티만을 이용하여 명세하기 때문에 새로운 조건이 추가 되더라도 기존에 명세되어 있던 다른 리팩토링들에는 전혀 영향을 미치지 않는다. 그리고 모델 제약사항을 체계적으로 기술해주는 표준 언어이며 정형적 언어인 OCL을 이용하여 정의하기 때문에 보다 명확하다고 할 수 있다[12].

리팩토링 효과는 실제로 리팩토링이 적용된 것과 같은 동일한 효과가 반영되어 처리될 수 있도록 '@post'라는 키워드를 사용하여 델타(Delta) 명세로 기술한다. 델타 명세는 리팩토링의 적용 후 변경되어야 하는 모든 모델 프로퍼티들을 정의하는 것으로 리팩토링의 적용 전을 기준으로 하는 '@post' 키워드를 이용해서 적용 후 변경되어야 하는 값을 표현한다. 리팩토링 적용 전을 기준으로 델타 명세를 기술하면 리팩토링 적용 후를 기준으로 했을 때 보다 '@pre'의 사용빈도가 낮아져 명세가 단순해진다. 이러한 델타 명세는 모두 OCL 기반으로 쉽게 정의할 수 있다. <표 1>은 본 연구에서 제안하는 리팩토링 선행조건과 델타 명세의 기술 방법을 정리한 것이다.

〈표 1〉 리팩토링의 선행조건과 델타 명세 기술 방법

Precondition	JavaEAST 메타모델 기반으로 OCL을 이용하여 리팩토링 전에 만족해야할 조건을 명세
Delta 명세	JavaEAST 메타모델 기반으로 변경되어야 할 모델 프로퍼티에 대해 리팩토링 적용 후 변화될 값으로 명세

본 연구에서 제안하는 델타 명세는 일반적인 모델 오퍼레이션에 대한 후행조건 명세 방법과 다음과 같은 차이점을 가진다. 일반적인 모델 오퍼레이션의 후행조건은 연산이 수행된 후에 상태가 어떤 조건을 만족해야 하는 가를 나타낸 것으로 상태가 어떻게 변해야 하는 가를 기술하는 것은 아니다. 하지만 델타 명세는 리팩토링을 적용하기 이전의 모델 프로퍼티가 리팩토링을 적용한 후에 그 모델 프로퍼티가 어떻게 변해야 하는 가를 나타내는 것이므로 리팩토링의 효과를 정확히 기술할 수 있게 된다. 그리고 일반적인 모델 오퍼레이션의 후행조건 명세나 본 논문에서 제안하고 있는 델타 명세는 모두 OCL을 형식 언어로 사용하고 있다. 하지



(그림 1) JavaEAST Metamodel에 확장된 요소들

만 모델 오퍼레이션의 후행조건 명세는 OCL을 제약(constraint) 언어로써 사용하고 있으나, 델타 명세는 OCL을 질의(query) 언어로 사용하고 있다.

3.2 요소 리팩토링의 명세

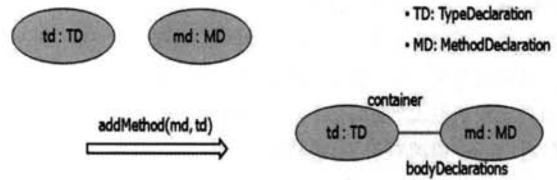
3.1절에서 제안한 리팩토링의 명세 방법을 이용하여 JavaEAST 메타모델 기반으로 'addMethod' 요소 리팩토링을 명세해본다. 이를 위해 먼저 MoDisco[13]에서 제안한 JavaAST(Java Abstract Syntax Tree)를 확장하여 'Java EAST Metamodel'을 정의한다. JavaAST는 소스코드의 문법적인 정보만을 가지고 있고, 'SimpleName'들에 대한 바인딩 정보가 없기 때문에 JavaAST를 확장했다. (그림 1)은 JavaEAST 메타모델에 확장된 요소들이며 <표 2>에서 간단히 설명한다.

〈표 2〉 JavaEAST Metamodel의 모델 요소 설명

모델 요소 이름	설 명
(a)의 Project	JavaEAST Metamodel의 CompilationUnit의 상위 엘리먼트
(b)의 container	ASTNode에서 상위 요소를 탐색 할 수 있는 프로퍼티
(c)의 definition	SimpleName이 어떤 Declaration에서 선언되었는지를 나타내는 프로퍼티
(c)의 referenceNames	Declaration이 어떤 SimpleName들에서 사용되고 있는지를 나타내는 프로퍼티
(d)의 subclassTypes	상위 클래스에서 하위 클래스를 알 수 있는 프로퍼티
(d)의 fullname	클래스와 메소드의 이름 중복을 해결하기 위한 프로퍼티

(그림 2)는 클래스에 메서드를 연결시켜주는 'addMethod' 요소 리팩토링에 대한 명세다. 선행조건은 프로그램내의 TypeDeclaration 'td'가 가지고 있는 MethodDeclaration중에 MethodDeclaration 'md'와 동일한 시그니처가 포함되어 있으면 안 됨을 의미한다. 델타 명세는 추가된 Method Declaration 'md'의 'container' 값이 TypeDeclaration 'td'로 변경되고, TypeDeclaration 'td'의 'bodyDeclarations'에는 MethodDeclaration 'md'가 포함되어야 함을 의미한다.

이처럼 본 연구에서 명세한 요소 리팩토링에는 addClass, addMethod, addField, removeClass, removeMethod, remove Field, removeFieldFragment, renameClass, renameMethod, renameFieldFragment, pullUpField들이 있으며 정의한 선행조건과 델타 명세는 부록에 표로 정리했다.



- Precondition :
  - not (td.methods→exists(m | m.isSameSignature(md))) and md.container.oclIsUndefined()
- Deltas :
  - delta : md.container@post = td
  - delta : td.bodyDeclarations@post = td.bodyDeclarations→including(md)
- Deltas in OCL :
  - context ASTNode def : container1 : ASTNode = if (self = md) then td else self.container endif
  - context ATD def : bodyDeclarations1 : Set(bodyDeclaration) = if (self = td) then td.bodyDeclarations→including(md) else self.bodyDeclarations endif

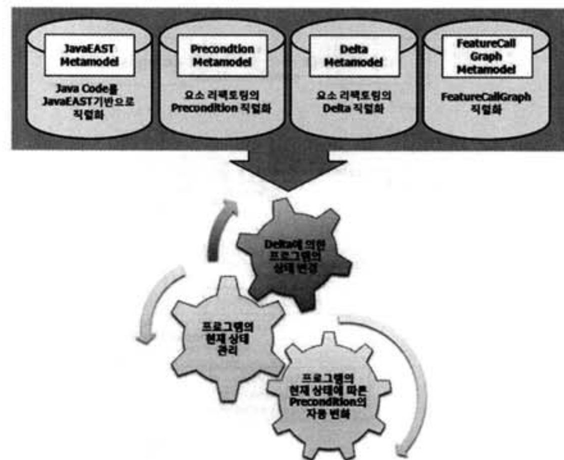
(그림 2) addMethod(md:MethodDeclaration, td:TypeDeclaration)의 선행조건과 델타 명세

4. 리팩토링의 조립성 검사

4.1 리팩토링의 조립성 검사 방법

두 개의 요소 리팩토링들이 순차로 구성된 복합 리팩토링에 대한 조립성 검사를 하기위해서는 첫 번째 요소 리팩토링에 의해 프로그램의 상태가 변하기 때문에 두 번째 요소 리팩토링의 선행조건을 그대로 사용할 수 없게 된다. 그래서 기존 연구들에서는 복합 리팩토링마다 선행조건을 미리 계산해서 해결하고 있다.

본 연구에서는 복합 리팩토링의 선행조건을 미리 정의하지 않고 요소 리팩토링만으로 조립 가능 여부를 판단할 수 있는 방안을 제안한다.



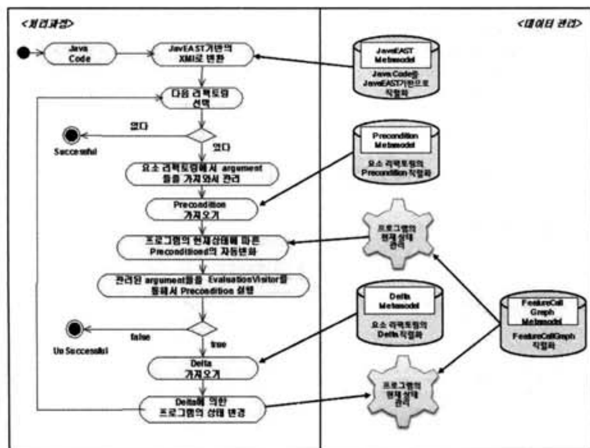
(그림 3) 조립성 검사를 위한 해결 방안

(그림 3)은 조립성 검사를 위해 필요한 정보와 연산들을 종합적으로 나타낸 것이다. 어떤 프로그램에 대해 복합 리팩토링이 적용 가능한지를 판단하기 위해서는 그 복합 리팩토링을 구성하는 요소 리팩토링들을 순서대로 적용해가면서 변하는 프로그램의 상태를 유지 관리해야 한다. 프로그램은 'JavaEAST Metamodel' 기반의 XMI[14] 문서로 직렬화되며, 프로그램의 상태는 모델의 프로퍼티와 정의된 피처들로 추상화된다. 복합 리팩토링이 적용가능하기 위해서는 복합 리팩토링을 구성하고 있는 모든 요소 리팩토링이 구성되어 있는 순서대로 적용가능해야 하며 만약 어느 하나라도 적용 가능하지 않으면 복합 리팩토링은 적용할 수 없게 된다. 복합 리팩토링을 구성하는 하나의 요소 리팩토링이 적용 가능한지는 그 요소 리팩토링의 선행조건을 프로그램의 현재 상태에서 해석될 수 있도록 변경시킨 후 선행 조건을 판단해야 한다. 선행 조건이 만족되면 요소 리팩토링의 델타 명세를 이용하여 프로그램의 현재 상태를 다음 상태로 변경하며, 선행 조건이 만족되지 않으면 조립성 검사는 즉시 종료하며 복합 리팩토링을 적용할 수 없는 것으로 판단한다.

조립성 검사가 진행되면서 필요한 정보들은 직렬화되어 있는 각각의 XMI 문서에서 가져 온다. 프로그램의 상태 정보는 'JavaEAST Metamodel' 기반의 XMI 문서에서 가져오고, 요소 리팩토링의 선행 조건과 델타 명세는 요소 리팩토링마다 직렬화해 놓은 XMI 문서에서 가져 온다. 모델 프로퍼티가 변경되면 이에 의존하고 있는 모든 정의된 피처들도 변화해야 하는데 이에 대한 정보는 직렬화된 'Feature CallGraph Metamodel' 기반의 XMI 문서에서 가져 온다.

4.2 조립성 검사 과정

리팩토링의 조립성을 검사하기 위해서는 내부적으로 프로그램의 상태를 관리하면서 조립을 원하는 요소 리팩토링의 선행조건을 현재의 상태가 반영되도록 변경시키고 조건 만족 시 델타 명세를 이용하여 내부적으로 관리되고 있는 프로그램의 상태를 변경해야 한다.



(그림 4) 조립성 검사를 위한 처리과정

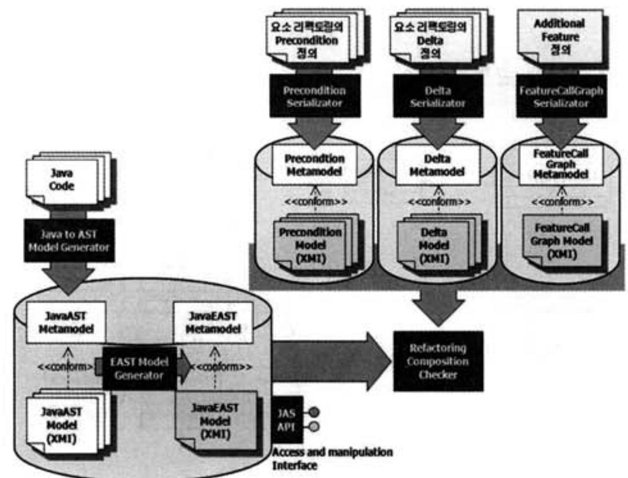
임의의 요소 리팩토링들로 순서화되어 있는 복합 리팩토링에 대한 조립성 검사를 처리하기 위한 과정은 그림 4와 같다. 이를 단계별로 설명하면 다음과 같다.

① 자바 프로그램을 'JavaEAST Metamodel' 기반으로 직렬화한다.
② 다음 요소 리팩토링이 있으면 선택하고, 없으면 복합 리팩토링을 적용할 수 있다고 판단한다.
③ 'Precondition Metamodel' 기반으로 관리되고 있는 선행조건들 중에 선택된 요소 리팩토링의 선행조건을 가져온다.
④ 선행조건을 프로그램의 현재 상태가 반영되도록 변환시킨다.
⑤ 변환된 선행조건을 판단한다.
⑥ 선행조건 검사가 참이면 'Delta Metamodel' 기반으로 직렬화되어 있는 선택된 요소 리팩토링의 델타 명세를 가져온다. 만약 거짓이면 복합 리팩토링을 적용할 수 없다고 판단하다.
⑦ 델타 명세를 이용하여 리팩토링이 실제 적용된 것과 동일한 효과가 반영되도록 프로그램의 상태를 변경시킨다.
⑧ 조립을 원하는 요소 리팩토링들은 ②~⑦까지 반복 처리된다.

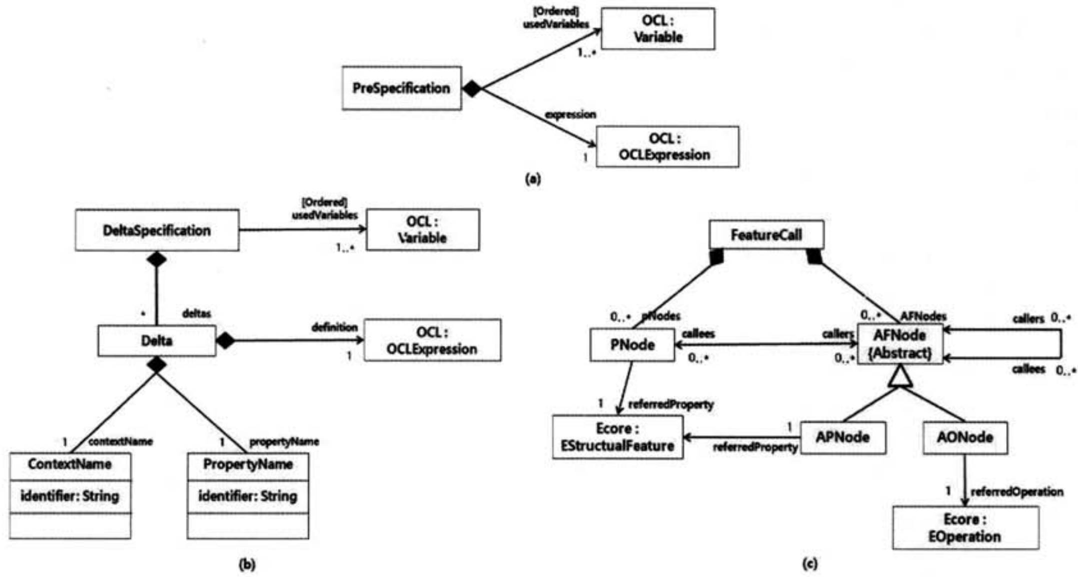
5. 적용사례

5.1 조립성 검사를 위한 도구의 프로토타입 개발

본 연구에서는 선행조건이 명세되어 있지 않은 임의의 복합 리팩토링들의 조립 가능성을 판단하기 위해 3, 4장에서 제안한 리팩토링의 명세 방법과 조립성 검사 방법을 기반으로 리팩토링의 조립성 검사 도구를 프로토타입으로 개발했다. (그림 5)는 리팩토링의 조립성 검사를 위한 도구의 전체 구조이며 Eclipse[15]의 Modeling Framework[16]와 OCL Interpreter[17]를 이용하여 개발했다.



(그림 5) 리팩토링 조립성 검사를 위한 도구의 전체 구조

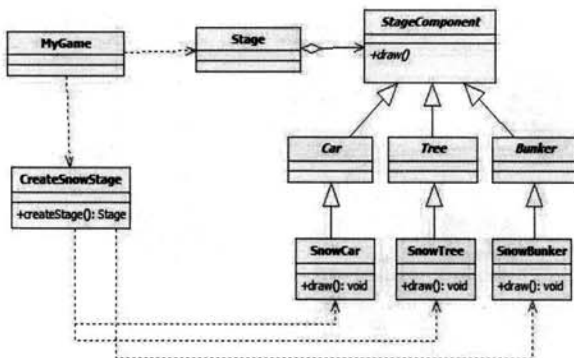


(그림 6) (a)Precondition Metamodel, (b)Delta Metamodel, (c)FeatureCallGraph Metamodel

리팩토링 조립성 검사를 하기 위해서는 (그림 5)에서처럼 요소 리팩토링의 선행조건과 델타 명세, 피처 의존성 정보들이 직렬화되어야 한다. 이를 위해 (그림 6)과 같이 메타모델을 정의하여 이를 기반으로 프로토타입 도구를 개발했다.

(그림 6)의 (a)는 리팩토링의 선행조건을 직렬화하기 위해 정의한 메타모델이고 (b)는 델타 명세를 직렬화하기 위해 정의한 메타모델이다. 그리고 프로그램의 상태를 변경하기 위해서는 어떤 모델 프로퍼티가 변경되면 반드시 같이 변경되어야 하는 정의된 피처들을 알아야 한다. 이것은 모델 프로퍼티와 정의된 피처들간의 의존관계를 알면 가능하다. 따라서 리팩토링 조립성 검사 과정에서 모델 프로퍼티와 정의된 피처들간의 의존관계를 파악해 프로그램의 상태를 변경할 수 있도록 하기 위해 피처 의존성을 직렬화했다. (c)는 정의된 피처를 직렬화하기 위해 정의한 메타모델이다.

5.2 예제 프로그램

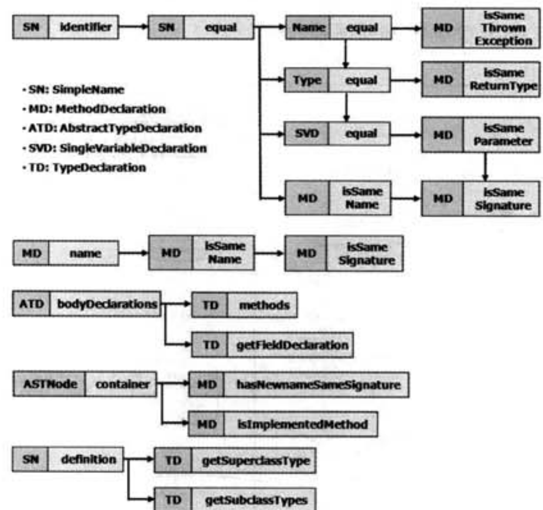


(그림 7) 눈싸움게임 예제 프로그램

(그림 7)은 리팩토링의 조립성 검사를 확인해 볼 예제 프로그램의 구조이다. 이는 참고문헌 [18]에 나와있는 눈싸움 게임의 일부분으로 화면을 구성하는 부분이다.

5.3 조립성 검사

4개의 요소 리팩토링 'addMethod', 'renameMethod', 'pullUpField', 'addField'를 이용하여 3종류의 적용사례에서 조립성 검사를 확인해보겠다. 조립성 검사에서 적용할 요소 리팩토링의 선행조건과 델타 명세는 부록의 표에 정의되어 있으며, (그림 8)은 적용할 요소 리팩토링에 대해 5.1절에서 정의한 피처들의 의존성을 가시화한 것이다. 이 의존성 정보를 이용해서 프로그램의 상태를 변경시킨다.



(그림 8) Feature들의 의존성 가시화 예

5.3.1 사례 1

'Stage' 클래스에 스테이지를 지우는 'deleteStage()' 메서드와 스테이지를 재설정하는 'resetStage()' 메서드를 추가하고 'deleteStage()' 메서드 이름을 "deleteStageComponent"로 변경하는 요소 리팩토링들에 대해 조립성 검사를 확인해보겠다.

▸ addMethod(deleteStage, Stage) ; addMethod(resetStage, Stage) ; renameMethod (deleteStage, "deleteStage Component")

[1 단계] addMethod 요소 리팩토링의 선행조건을 가져온다. 프로그램의 현재 상태는 아직 어떠한 리팩토링도 적용되지 않았기 때문에 프로그램의 시작 상태에 있다. 따라서 선행조건에서 참조하고 있는 모델 프로퍼티와 정의된 피처들은 변경될 필요가 없다. 다만 addMethod(deleteStage, Stage)를 통해 전달받은 파라미터 값으로 선행조건을 바인딩하면 다음과 같이 재해석된다. 재해석된 선행조건을 판단하면 True가 된다.

```
not(Stage.methods->exists(m | m.isSameSignature(deleteStage)))
and deleteStage.container.oclIsUndefined()
```

[2 단계] addMethod 요소 리팩토링의 델타 명세를 가져온다. 델타 명세에서 정의하고 있는 모델 프로퍼티인 'container'와 'bodyDelarations'에 대해 (그림 10)의 의존성이 있는 피처들 모두 재정의하여 프로그램 상태를 다음과 같이 변경한다. 이때 델타 명세에서 참조하고 있는 모델 프로퍼티와 정의된 피처들은 변화된 프로그램의 상태를 반영하도록 변경한다. 'getFieldDeclaration1'은 5.3.3 사례 3의 [2단계]와 동일하므로 생략하겠다. 여기서 언급이 안 된 피처들은 이전의 프로그램 상태와 동일하다.

```
context ASTNode
def: container1 : ASTNode =
if (self=md) then td
else self.container endif

context AbstractTypeDeclaration
def : bodyDeclarations1 : Set(BodyDeclaration) =
if (self=td) then td.bodyDeclarations->including(md)
else self.bodyDeclarations endif

context MethodDeclaration
def : hasNewnameSameSignature1(mdname : String) : Boolean =
self.container1.oclAsType(TypeDeclaration).getSuperClasses
->union(self.container1.oclAsType(TypeDeclaration).getSubClasses)
->union(self.container1.oclAsType(TypeDeclaration)->asSet())
->collect(getMethods)->select( not ( modifiers
->select(oclIsTypeOf(Modifier))->collect(oclAsType(Modifier))
->exists(m | m.private=true)) )
->exists(m | m.name.identifier=mdname and m.isSameParameter(self))

context MethodDeclaration
def : isImplementedMethod1 : Boolean =
self.container1.oclAsType(TypeDeclaration).getSuperclassType->asSet()
->union(self.container1.oclAsType(TypeDeclaration).getSuperInterfaces)
->collect(getMethods)->select(m | m.body = null)
->exists(m| m.isSameSignature(self))

context TypeDeclaration
def : methods1 : Set(MethodDeclaration) =
self.bodyDeclarations1
->select(oclIsTypeOf(MethodDeclaration)).oclAsType(MethodDeclaration)
->select(method | not(method->exists(constructor)) and
method.modifiers.oclAsType(Modifier)->exists(public) and
not(method.modifiers.oclAsType(Modifier)->exists(static)))
->asSet()
```

[3 단계] addMethod 요소 리팩토링의 선행조건을 가져온다. 선행조건에서 참조하고 있는 모델 프로퍼티와 정의된 피처들은 변화된 프로그램의 상태를 반영하도록 변경한다. 그리고 addMethod(resetStage, Stage)를 통해 전달받은 파라미터 값으로 선행조건을 바인딩하면 다음과 같이 재해석된다. 재해석된 선행조건을 판단하면 True가 된다.

```
not(Stage.methods1->exists(m | m.isSameSignature(resetStage)))
and resetStage.container1.oclIsUndefined()
```

[4 단계] addMethod 요소 리팩토링의 델타 명세를 가져온다. 델타 명세에서 정의하고 있는 모델 프로퍼티인 'container'와 'bodyDelarations'에 대해 (그림 10)의 의존성이 있는 피처들 모두 재정의하여 프로그램 상태를 관리한다. 이때 델타 명세에서 참조하고 있는 모델 프로퍼티와 정의된 피처들은 변화된 프로그램의 상태를 반영하도록 변경한다. 여기서 언급이 안 된 피처들은 이전의 프로그램 상태와 동일하다.

```
context ASTNode
def: container2 : ASTNode =
if (self=md) then td
else self.container1 endif

context AbstractTypeDeclaration
def : bodyDeclarations2 : Set(BodyDeclaration) =
if (self=td) then td.bodyDeclarations1->including(md)
else self.bodyDeclarations1 endif

context MethodDeclaration
def : hasNewnameSameSignature2(mdname : String) : Boolean =
```

```
self.container2.oclAsType(TypeDeclaration).getSuperClasses
->union(self.container2.oclAsType(TypeDeclaration).getSubClasses)
->union(self.container2.oclAsType(TypeDeclaration)->asSet())
->collect(getMethods)->select( not ( modifiers
->select(oclIsTypeOf(Modifier))->collect(oclAsType(Modifier))
->exists(m | m.private=true)) )
->exists(m | m.name.identifier=mdname and m.isSameParameter(self))

context MethodDeclaration
def : isImplementedMethod2 : Boolean =
self.container2.oclAsType(TypeDeclaration).getSuperclassType->asSet()
->union(self.container2.oclAsType(TypeDeclaration).getSuperInterfaces)
->collect(getMethods)->select(m | m.body = null)
->exists(m | m.isSameSignature(self))

context TypeDeclaration
def : methods2 : Set(MethodDeclaration) =
self.bodyDeclarations2
->select(oclIsTypeOf(MethodDeclaration)).oclAsType(MethodDeclaration)
->select(method | not(method->exists(constructor)) and
method.modifiers.oclAsType(Modifier)->exists(public) and
not(method.modifiers.oclAsType(Modifier)->exists(static)))
->asSet()
```

[5 단계] renameMethod 요소 리팩토링의 선행조건을 가져온다. 선행조건에서 참조하고 있는 모델 프로퍼티와 정의된 피처들은 변화된 프로그램의 상태를 반영하도록 변경한다. 그리고 renameMethod(deleteStage, "deleteStage Component")를 통해 전달받은 파라미터 값으로 선행조건을 바인딩하면 다음과 같이 재해석된다.

```
not(deleteStage.constructor)
and not(deleteStage.isImplementedMethod2)
and not(hasNewnameSameSigature2(deleteStageComponent))
```

마지막 요소 리팩토링의 재해석된 선행조건이 True가 나옴에 따라 요소 리팩토링을 순서대로 적용 가능함을 알 수 있다.

5.3.2 사례 2

'Stage' 클래스에 스테이지를 지우는 'deleteStage()' 메서드를 추가하고 다시 동일한 'deleteStage()' 메서드를 추가하는 요소 리팩토링에 대해 조립성 검사를 확인해보겠다.

▸ addMethod(deleteStage, Stage) ; addMethod(delete Stage, Stage)

[1 단계]부터 [2 단계]까지는 사례 1의 경우와 동일하므로 생략하겠다.

[3 단계] addMethod 요소 리팩토링의 선행조건을 가져온다. 선행조건에서 참조하고 있는 모델 프로퍼티와 정의된 피처들은 변화된 프로그램의 상태를 반영하도록 변경한다. 그리고 addMethod(deleteStage, Stage)를 통해 전달받은 파라미터 값으로 선행조건을 바인딩하면 다음과 같이 재해석된다.

```
not(Stage.methods1->exists(m | m.isSameSignature(deleteStage)))
and deleteStage.container1.oclIsUndefined()
```

두 번째 요소 리팩토링의 재해석된 선행조건은 False가 나온다. 이는 변화된 'methods1'에 의해 현재 추가하고자 하는 메서드와 동일한 시그니처의 메서드가 존재하기 때문이다. 따라서 사례 2의 경우 요소 리팩토링을 순서대로 적용이 불가능하다고 판단할 수 있다.

5.3.3 사례 3

'Tree' 클래스의 'stageName' 필드를 부모 클래스로 Pullup하고 'StageComponent' 클래스에 'stageName' 필드와 이름이 동일한 새로운 'newFD' 필드를 추가하는 요소 리팩토링들에 대해 조립성 검사를 확인해보겠다.

▸ pullUpField(stageName) ; addField(newFD, StageComponent)

[1 단계] pullUpField 요소 리팩토링의 선행조건을 가져온다. 프로그램의 현재 상태는 아직 어떠한 리팩토링도 적용되지 않았기 때문에 프로그램의 시작 상태에 있다. 따라서 선행조건에서 참조하고 있는 모델 프로퍼티와 정의된 피처들은 변경될 필요가 없다. 다만 pullUpField(stageName)를 통해 전달받은 파라미터 값으로 선행조건을 바인딩하면 다음과 같이 재해석된다. 재해석된 선행조건을 판단하면 True가 된다.

```
stageName.container.oclAsType(TypeDeclaration).superclassType <> null
    and not(hasSameFieldDeclarationInSuperclasses(stageName)
    and stageName.fragments->size() = 1 and
    not(stageName.modifiers.oclAsType(Modifier)->exists(m | m.private=true))
    and stageName.modifiers.oclAsType(Modifier)->exists(m | m.final=true)
    implies stageName.fragments.initializer->notEmpty()
```

[2 단계] pullUpField 요소 리팩토링의 델타 명세를 가져온다(부록참고). 델타 명세에서 정의하고 있는 모델 프로퍼티인 'container'와 'bodyDelarations', 'referenceNames', 'definition'에 대해 (그림 7)의 의존성이 있는 피처들 모두 재정의하여 프로그램 상태를 변경한다. 이때 델타 명세에서 참조하고 있는 모델 프로퍼티와 정의된 피처들은 변화된 프로그램의 상태를 반영하도록 변경한다. 'hasNewnameSame Signature1', 'isImplementedMethod1', 'methods1'은 5.3.1 사례 1의 [2 단계]와 동일하므로 생략하겠다. 여기서 언급이 안 된 피처들은 이전의 프로그램 상태와 동일하다.

```
context ASTNode
def : container1 : ASTNode = ...

context TypeDeclaration
def : bodyDeclarations1 : Set(BodyDeclaration) = ...

context VariableDeclarationFragment
def : referenceNames1 : Set(SimpleName) = ...

context SimpleName
def : definition1: Declaration = ...

context TypeDeclaration
def : getFieldDeclaration1 : Set(FieldDeclaration) =
self.bodyDeclarations1
->select(oclIsTypeOf(FieldDeclaration)).oclAsType(FieldDeclaration)
->asSet()

context TypeDeclaration
def : getSuperclassType1 : TypeDeclaration =
self.superclassType.oclAsType(SimpleType).name.oclAsType(SimpleName)
.definition1.oclAsType(TypeDeclaration)

context TypeDeclaration
def : getSubclassTypes1 : Set(TypeDeclaration) =
self.subclassTypes.oclAsType(SimpleType).name.oclAsType(SimpleName)
.definition1.oclAsType(TypeDeclaration)->asSet()
```

[3 단계] addField 요소 리팩토링의 선행조건을 가져온다. 선행조건에서 참조하고 있는 모델 프로퍼티와 정의된 피처들은 변화된 프로그램의 상태를 반영하도록 변경한다. 그리고 addField(newFD, StageComponent)를 통해 전달받은 파라미터 값으로 선행조건을 바인딩하면 다음과 같이 재해석된다.

```
not(StageComponent.getFieldDeclaration1
->exists(f|f.fragments.name.identifier=newFD.fragments.name.identifier))
and newFD.container1.oclIsUndefined()
```

두 번째 요소 리팩토링의 재해석된 선행조건은 False가 나온다. 이는 변화된 'getFieldDeclaration1'에 의해 현재 추가하고자 하는 필드와 동일한 이름의 필드가 'Stage Component' 클래스에 존재하기 때문이다. 따라서 사례 3의 경우 요소 리팩토링을 순서대로 적용이 불가능하다고 판단할 수 있다.

6. 결 론

본 연구에서는 복합 리팩토링에 대한 선행조건이 명세되어 있지 않더라도 구성된 요소 리팩토링만으로 조립 가능 여부를 판단할 수 있도록 리팩토링의 명세 방법과 조립성 검사 방법을 제안하였고, 이를 기반으로 프로토타입 도구를 개발하여 실제 사례에 적용시켜 그 가능성을 확인하였다.

본 연구에서 제안한 델타 명세는 일반적인 모델 오퍼레이션에 대한 후행조건과 달리 변경되어야할 모델 프로퍼티에 대해 리팩토링 적용 후 변화될 값으로 명세하는 방법이기 때문에 프로그램의 상태를 명확하게 정의함으로써 선행조건이 명세되어 있지 않은 복합 리팩토링의 조립성 검사를 가능하게 한다. 메타모델의 모델 프로퍼티만을 이용하여 명세하기 때문에 새로운 술어가 추가되더라도 기존에 명세되어 있던 다른 리팩토링들에는 전혀 영향을 미치지 않는다. 그리고 JavaEAST 메타모델을 기반으로 OCL을 이용하여 구조화되고 정형화되어 있기 때문에 작성이 쉬우며 체계적이다.

본 연구에서 프로토타입으로 개발된 조립성 검사 도구는 자바 기반으로 개발되어 있으며 자바로 구현되어 있는 적용사례에 대해서만 조립성 검사를 확인할 수 있다. 하지만 본 연구에서 제안하는 리팩토링의 명세 방법과 조립성 검사 방법은 다른 언어에서도 동일하게 적용될 수 있을 것이다.

현재 프로토타입의 조립성 검사 도구에서는 복합 리팩토링을 구성하는 요소 리팩토링들이 순서화되어 있는 경우에만 처리하고 있지만 'if'나 'loop' 등과 같은 다양한 방법으로 요소 리팩토링을 조립할 수 있다면 요소 리팩토링의 재사용성을 향상시킬 수 있을 것이다. 앞으로 이와 같은 조립 메커니즘을 정의하고 이를 지원해주는 Composition Language에 대한 연구가 필요하다. 그리고 복합 리팩토링에 사용할 수 있는 요소 리팩토링들을 라이브러리화 할 필요가 있다.

참 고 문 헌

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code," Addison Wesley, 1999.

[2] T. Mens, T. Tourwe, "A Survey of Software Refactoring," IEEE Transactions on Software Engineering, Vol. XX, No. Y, Month 2004.

[3] William F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D thesis, University of Illinois at Urbana-Champaign, 1992.

[4] Martin Kuhlemann, Don Batory, Christian Kastner, "Safe Composition of Refactoring Feature Modules," Technical Report, School of Computer Science, University of Magdeburg, 2009.

[5] Ethan Hadar, Irit Hadar, "The Composition Refactoring Triangle(CRT) Practical Toolkit: From Spaghetti to Lasagna," ACM OOPSLA'06, 2006

[6] 김태웅, 김태공, "OCL을 이용한 자동화된 코드스멜 탐지와 리팩토링," 한국정보처리학회 논문지 D, Vol. 15-D, No.06, pp.0825-0840, 2008. 12.

[7] Javier Perez, Yania Crespo, Berthold Hoffmann, Tom Mens, "A case study to evaluate the suitability of graph

transformation tools for program refactoring," International Journal on Software Tools for Technology Transfer(STTT), Vol. 12, No. 3-4, pp.183-199, 2010. 7.
[8] Donald Bradley Roberts, "Practical Analysis for Refactoring," Ph.D thesis, University of Illinois at Urbana-Champaign, 1999.
[9] Mel O Cinneide, "Automated Application of Design Patterns : A Refactoring Approach," Ph.D thesis, University of Dublin, Trinity College, October 2000.
[10] G. Kniesel, H. Koch, "Static Composition of Refactorings," In R. Lammel, editor, Science in Computer Programming; Special issue on program transformation. Elsevier Science, 2004.
[11] O. M. G., "UML 2.0 OCL specification," OMG Adopted Specification(ptc/03-10-14), 2003.
[12] J. Warmer, A. Kleppe, "The Object Constraint Language Second Edition: getting your models ready for MDA," Addison Wesley, 2003.
[13] MoDisco, "MoDisco Tool - Java Abstract Syntax Discovery Tool," http://www.eclipse.org/gmt/modisco/toolBox/Java Abstract Syntax/
[14] W3C, "XML Schema 1.1," http://www.w3.org/XML/Schema#dev
[15] Eclipse Consortium, "Eclipse Version 3.4.1," http://www.eclipse.org
[16] Eclipse, "Eclipse Modeling Framework Project," http://www.eclipse.org/modeling/emf- /?project=emf
[17] Eclipse, "OCL Library," http://www.cs.kent.ac.uk/projects/ocl/tools.html#documentation
[18] 박지훈, "자바 디자인 패턴과 리팩토링," 한빛미디어(주), 2003.

<부록 표1> 요소 리팩토링의 선행조건과 델타 명세

요소 리팩토링의 원형	
선행 조건	**델타**
void addClass(td:TypeDeclaration, pro:Project)	

not(pro.hasTDNames(td.name.identifier))
and
td.container.oclIsUndefined()
	context ASTNode def : container1 : ASTNode =
	if (self=td.container) then pro else self.container endif
	
	context Project def : compilationUnits1 : Set(CompilationUnit) =
	if (self=pro) then pro.compilationUnits->including(cu.container)
	else self.compilationUnits endif

void addMethod(md:MethodDeclaration, td:TypeDeclaration)	

not(td.methods
->exists(m | m.isSameSignature(md)))
and
md.container.oclIsUndefined()
	context ASTNode def: container1:ASTNode =
	if (self=md) then td else self.container endif
	
	context AbstractTypeDeclaration def: bodyDeclarations1:Set(BodyDeclaration) =
	if (self=td) then td.bodyDeclarations->including(md)
	else self.bodyDeclarations endif

void addField(fd:FieldDeclaration, td:TypeDeclaration)	

not(td.getFieldDeclaration
->exists(f | f.fragments.name.identifier=
fd.fragments.name.identifier))
and
fd.container.oclIsUndefined()
	context ASTNode def : container1 : ASTNode =
	if (self=fd) then td else self.container endif
	
	context AbstractTypeDeclaration def : bodyDeclarations1 : Set(BodyDeclaration) =
	if (self=td) then td.bodyDeclarations->including(fd)
	else self.bodyDeclarations endif

void removeClass(td:TypeDeclaration)	

td.referenceNames->isEmpty()	context ASTNode def : container1 : ASTNode =
	if (self=td.container) then null else self.container endif
	
	context Project def : compilationUnits1 : Set(CompilationUnit) =
	if (self=td.container.container) then
	td.container.container.oclAsType(Project).compilationUnits
	->excluding(td.container.oclAsType(CompilationUnit))
	else self.compilationUnits endif

<table>
<tr><th colspan="2">void removeMethod(md:MethodDeclaration)</th></tr>
<tr><td>not(md.isImplementedMethod)
and
md.referenceNames
->forAll(sn|sn.containingBody(sn)=
md.body)</td><td>context ASTNode def : container1 : ASTNode =
if (self=md) then null else self.container endif

context AbstractTypeDeclaration def : bodyDeclarations1 : Set(BodyDeclaration) =
if (self=md.container) then
md.container.oclAsType(TypeDeclaration).bodyDeclarations->excluding(md)
else self.bodyDeclarations endif</td></tr>
<tr><th colspan="2">void removeField(fd:FieldDeclaration)</th></tr>
<tr><td>fd.fragments
->collect(f|f.referenceNames)->isEmpty()</td><td>context ASTNode def : container1 : ASTNode =
if (self=fd) then null else self.container endif

context TypeDeclaration def : bodyDeclarations1: Set(BodyDeclaration) =
if (self=fd.container) then
fd.container.oclAsType(TypeDeclaration).bodyDeclarations->excluding(fd)
else self.bodyDeclarations endif</td></tr>
<tr><th colspan="2">void removeFieldFragment(vdf:VariableDeclarationFragment)</th></tr>
<tr><td>vdf.referenceNames->isEmpty()</td><td>context ASTNode def : container1 : ASTNode =
if (self=vdf) then null else self.container endif

context FieldDeclaration def : fragments1 : Set(VariableDeclatationFragment) =
if (self=vdf.container) then
vdf.container.oclAsType(FieldDeclaration).fragments->excluding(vdf)
else self.fragments endif</td></tr>
<tr><th colspan="2">void renameClass(td:TypeDeclaration, tdname:String)</th></tr>
<tr><td>not
(TypeDeclaration.allInstances()
->exists(t|t.name.identifier=tdname))</td><td>context SimpleName def : identifier1 : String =
if (self=td.name) or (td.referenceNames->includes(self)) or
(td.getConstructorMethodSNs->includes(self)) then tdname
else self.identifier endif</td></tr>
<tr><th colspan="2">void renameMethod(md:MethodDeclaration, mdname:String)</th></tr>
<tr><td>not(md.constructor) and
not(md.isImplementedMethod) and
not(hasNewnameSameSignature(mdname))</td><td>context SimpleName def: identifier1 : String =
if (self=md.name) or (md.referenceNames->includes(self)) then mdname
else self.identifier endif</td></tr>
<tr><th colspan="2">void renameFieldFragment(vdf:VariableDeclarationFragment, fdname:String)</th></tr>
<tr><td>not(hasNewnameSameFragment(fdname))</td><td>context SimpleName def : identifier1 : String =
if (self=vdf.name) or (vdf.referenceNames->includes(self)) then vdfname
else self.identifier endif</td></tr>
<tr><th colspan="2">void pullUpField(fd:FieldDeclaration)</th></tr>
<tr><td>fd.container.oclAsType(TypeDeclaration)
.superclassType <> null
and
not(hasSameFieldDeclarationInSuperclasses
(fd))</td><td>context ASTNode def : container1 : ASTNode =
if (self=fd) then fd.container.oclAsType(TypeDeclaration).superclassType
else if (fd.getSameFieldsInSiblingClass->includes(self)) then null
else self.container endif endif

context TypeDeclaration def : bodyDeclarations1: Set(BodyDeclaration) =
if (self=fd.container.oclAsType(TypeDeclaration).superclassType) then
fd.container.oclAsType(TypeDeclaration).superclassType.bodyDeclarations
->including(fd)
else if (self=fd.container) then
fd.container.oclAsType(TypeDeclaration).bodyDeclarations->excluding(fd)
else if (fd.getSameFieldsInSiblingClass
->collect(f|f.container.oclAsType(TypeDeclaration))->includes(self))
then self.bodyDeclarations - fd.getSameFieldsInSiblingClass</td></tr>
</table>

and
fd.fragments->size() = 1
and
not(fd.modifiers.oclAsType(Modifier)
->exists(m | m.private=true))
and
fd.modifiers.oclAsType(Modifier)
->exists(m | m.final=true) implies
fd.fragments.initializer->notEmpty()
	else self.bodyDeclarations endif endif endif
	
	context VariableDeclarationFragment def : referenceNames1 : Set(SimpleName) =
	if (fd.getSameFieldsInSiblingClass
	->includes(self.container.oclAsType(FieldDeclaration)))
	then fd.fragments.referenceNames->asSet()->union(self.referenceNames)
	else self.referenceNames endif
	
	context SimpleName def : definition1: Declaration =
	if (fd.getSameFieldsInSiblingClass->collect(f | f.fragments.referenceNames)
	->includes(self)) then fd.fragments->first()
	else self.definition endif

김 경 민

e-mail : kmkim@cs.inje.ac.kr
2002년 인제대학교 전산학과(이학사)
2004년 인제대학교 전산학과(이학석사)
2006년 인제대학교 전산학과(박사과정 수료)
2008년~현 재 동의과학대학 컴퓨터정보계열 겸임교수
관심분야 : 소프트웨어 공학, Model Engineering, Code Smell Detection, Refactoring, Design Pattern 등

장 필 재

e-mail : pj.jang@tk.t2group.co.kr
2008년 인제대학교 컴퓨터공학과(공학사)
2010년 인제대학교 전산학과(전산학석사)
2010년~현 재 태광엠티씨(주) R&D팀 주임
관심분야 : 소프트웨어 공학, Refactoring, Code Smell Detection, Design Pattern, OCL 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