Journal of The Korea Society of Computer and Information
Vol. 21 No. 5, pp. 119-126, May 2016

www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2016.21.5.119

Extracting of Features in Code Changes of Existing System for Reengineering to Product Line

Seonghye Yoon*, Sooyong Park**, Mansoo Hwang***

Abstract

Software maintenance becomes extremely difficult, especially caused by multiple versions in project-based or customer-oriented software development methodology. For reducing the maintenance cost, reengineering to software product line can be a solution to the software which either is a family of products nevertheless little different functionalities or are customized for each different customer's requirement. At an initial stage of the reengineering, the most important activity in software product line is feature extraction with respect to commonality and variability from the existing system due to verifying functional coverage. Several researchers have studied to extract features. They considered only a single version in a single product. However, this is an obstacle to classify the commonality and variability of features. Therefore, we propose a method for systematically extracting features from source code and its change history considering several versions of the existing system. It enables us to represent functionalities reflecting developer's intention, and to clarify the rationale of variation.

▸Keyword: Reengineering, Software product line, Feature extraction, Change history, Software evolution

I. Introduction

프로젝트나 고객 중심의 개발 방법론은 소스 코드의 브랜치를 생성하여 멀티 버전을 유지하는 형태의 소프트웨어 개발 방식이다. 이 개발 방식은 제품의 공통(commonality) 기능을 가진 메인(trunk) 버전을 기준으로 특화된 버전(branch)을 생성하여 가변적인(variability) 기능을 추가한다. 그러므로 각 고객사나 프로젝트별로 발 빠른 대응이 가능하며 가변적인 기능들을 분리한다는 장점을 가진다. 그러나 공통성과 가변성을 기술적인 메커니즘으로 분리하지 않고 코드 분리(branch)를 통해 관리 측면의 메커니즘으로만 지원하기 때문에 빈번한 애드혹(ad-hoc) 재사용의 요인이 된다. 이는 코드 중복이나 불일치와 같은 소스 코드 내의 나쁜 냄새로 인해 유지보수 비용이 급격하게 증가하는 요인이 된다[1]. 이를 해결하기 위해 제품의 공통성과 가변성을 고려하여 제품을 설계하는 방식인 소프트웨어 프로덕트 라인(SPL)[2]으로 재공학(reengineering)하는 것이 필요하다[3].

기존 시스템을 SPL로 재공학하는 첫걸음은 해당 시스템의 모든 기능을 반영할 수 있는 휘처 모델을 생성하는 것이다. 이를 위해서는 우선 기존 시스템에 대한 이해가 필수적이나, 기존 시스템을 이해하는 것에는 다음과 같은 어려움이 따른다. 1) 소스 코드와 동기화되어 유지보수 되는 요구사항이나 산출물의 부족[4], 2) 고객의 임의적인 기능 추가나 변경 요청[2]. 특히, 2)와 같은 고객 커스터마이징의 경우, 기능의 지속적인 사용 여부와는 상관없이 고객과의 계약 종료 시점까지 소스 코드에 대한 유지보수가 필요하다. 그러나 실제적으로는 이런 기능들의 상당수는 해당 소스 코드의 관리 부재와 산출물 미비로 인하여, 유령 코드로 존재하고 있다. 그러므로 수작업으로 모든 휘처를 추출하고 해당 휘처에

• First Author: Seonghye Yoon, Corresponding Author: Mansoo Hwang
*Seonghye Yoon (yoonsh@kookmin.ac.kr), Dept. of Computer Science & Engineering, Sogang University and Dept. of Computer Science, Kookmin University
**Sooyong Park (sypark@sogang.ac.kr), Dept. of Computer Science & Engineering, Sogang University
***Mansoo Hwang (mshwang@shinhan.ac.kr), School of IT Convergence Engineering, Shinhan University
• Received: 2016. 03. 02, Revised: 2016. 04. 16, Accepted: 2016. 05. 12.

대한 이력을 추적하는 데는 한계가 따르게 된다.

이 문제를 해결하기 위해, 기존 시스템의 기능을 휘처로 추출하여 정련된 휘처 모델을 생성할 수 있게 지원하는 것이 필요하다. 이를 위한 여러 연구들[5-9] 은 휘처 클러스터링이나 휘처 맵핑과 같은 역공학(Reverse Engineering) 메커니즘을 통해 휘처를 추출하는 방법들을 제안하였다. 주로 소스 코드를 기반을 두어 분석하고 있으며, 몇몇 연구들은 동적 분석이나 변경 분석을 통해 휘처를 식별할 방안을 제시하고 있다. 그러나 이 연구들은 한 버전에 국한된 소스 코드를 대상으로 하고 있어 휘처 모델링[10]의 주요 개념 중 하나인 휘처의 공통성과 가변성을 분류하지 못하고 있다. [11] 연구에서는 멀티 버전의 소스 코드를 분석하여 휘처의 공통성 및 가변성을 복구하는 방안을 제안하고 있지만 휘처의 클러스터링 없이 소스코드의 유사도를 비교하여 공통성과 가변성을 나타내고 있다.

본 논문에서는 소스 코드와 변경 이력 분석을 기반으로 하여 멀티 버전의 특성을 고려한 휘처를 식별하고 휘처의 공통성 및 가변성을 분류하는 방법을 제안하고자 한다. 먼저, 각 버전의 소스 코드의 문법적 정보에 기반을 두어 횡단 관심사나 유틸성 함수들을 고려하여 태스크를 분석한다. 다음으로는 추출된 태스크들을 변경 이력을 기반으로 논리적으로 하나의 기능을 표현하는 태스크들을 연결하여 휘처를 추출한다. 마지막으로, 버전별로 발생하는 소스 코드의 스냅샷 비교를 통해 공통성과 가변성을 식별한다. 본 논문에서 제안한 방법을 검증하기 위해, 오픈 소스 프로젝트인 Subclipse[12]의 공통 휘처와 가변 휘처를 분석하는 과정을 보여준다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 관련된 연구를 소개하고 3장에서는 본 연구에서 제안하는 휘처 추출 방법에 대하여 언급한다. 4장에서는 subclipse에 제안한 연구를 적용한 사례를 소개한다. 마지막으로 5장에서는 결론을 기술한다.

II. Related works

휘처 클러스터링이나 휘처 맵핑을 위한 역공학 메커니즘은 여러 연구들[5-9]에서 제안되어 왔다. 이 연구들은 소스 코드의 한 스냅샷을 이용하여 여러 태스크에 의해서 호출되는 공유 휘처를 추출하고 있다. 다시 말해, 여러 버전의 시스템으로부터 공통 휘처를 추출하는 것이 아니라 한 버전 내에서 기능적 휘처와 비기능적 휘처를 추출하는데 적합하다고 볼 수 있다.

M. P. Robillard et al. [5] 는 개발자들이 흩어져있는 관심사(concern)들을 이해하고 재구성하는 데 도움을 주기 위해, 관심사 그래프(conren graph)를 제안하였다. 관심사를 구현하는 구조적 정보의 키를 기록하여 다른 관심사와의 관계를 명확히 문서화하는 방식을 사용함으로써, 소프트웨어가 진화하는 동안에도 지속해서 관심사에 대한 모니터링이 가능하다. M. P. Robillard [6]은 태스크를 수정할 때, 영향을 미칠 수 있는 태스크를 식별하는 비용을 줄이기 위해 프로그램의 구조적 의존관계에 기반을 두어 잠재적 관계가 있는 태스크들을 분석하였다. 개발자들이 각기 맡은 태스크들 단위로 관련 구성요소(interesting elements)를 제공하면 이를 바탕으로 잠재적으로 관련이 있을 수 있는 구성 요소들을 추천하는 방식이다. K. Kobayashi et al. [7]은 소프트웨어를 이해하기 위해 중요한 클러스터링을 하는 기법으로 SArF 알고리즘을 제안하였다. 이 알고리즘은 소스 코드의 의존성을 분석하여 클래스 간의 외부 멤버 접근을 고려한 Delication 점수로 가중치를 계산하고, 이를 활용해 공통 휘처를 공유하는 모듈들을 식별한다. T. Eisenbarth et al. [8]은 시스템의 기능을 이해하기 위해서 기능을 구성하고 있는 유닛들을 맵핑하고 추적할 수 있어야 하는데, 이를 위해 소스 코드의 호출 관계와 시나리오에 명세한 동적 실행 타임 호출 정보를 이용하고 있다. 동적 정보를 사용한 이유는 소스 코드에서 확인할 수 없는 휘처들간의 연관성을 추적하기 위함이다. B. Adams et al. [9]은 기존 시스템의 횡단 관심사를 추출하기 위해 fan-in을 강조한 Determining methods를 제안하고 있다. 높은 fan-in을 가진 함수들을 선별한 후, 유틸성 함수들을 필터링한다.

멀티 버전의 소스 코드로부터 공통 휘처를 추출하는 연구로는 C. Nunes et al. [11] 연구가 있다. 이 연구는 프로그램이 다양한 버전으로 진화할 때 원활한 유지보수를 위해 공통 휘처와 가변 휘처를 구별할 필요가 있으므로, 여러 버전의 소스 코드를 비교하여 휘처를 분리하는 기법을 제안한다. 그러나 이 연구에서는 기존 시스템이 이미 프로덕트 패밀리임을 전제하므로 휘처 클러스터링은 제외하고, 버전별 소스 코드의 유사도를 비교하여 구현 레벨에서 휘처의 공통성과 가변성만을 식별하고 있다.

본 연구에서는 기존의 시스템이 잘 정련된 휘처에 대한 정보를 가지지 않은 멀티 버전의 시스템일 때, 휘처를 추출하고 이들의 공통성과 가변성을 구별할 수 있는 기법을 제안한다. 또한, 식별된 휘처들이 버전별로 다르게 진화해온 근거를 추적할 수 있게 하여, 효율적인 휘처 모델링을 돕고자 한다.

III. A method of feature extraction in code changes

본 논문에서는 구현 모델(정적 코드 분석 결과)과 변경 이력 모델(버전 관리 시스템의 변경 로그)로부터 공통성과 가변성을 고려한 휘처를 추출하는 역공학 방법을 제안한다.

분석에 앞서, 분석 대상이 되는 소스 코드를 선정한다. 일반적으로, 메인 버전과 주요 고객사의 브랜치 버전들의 마지막 리비전들이 선택될 수 있다. 선택된 소스 코드 버전들을 대상으로

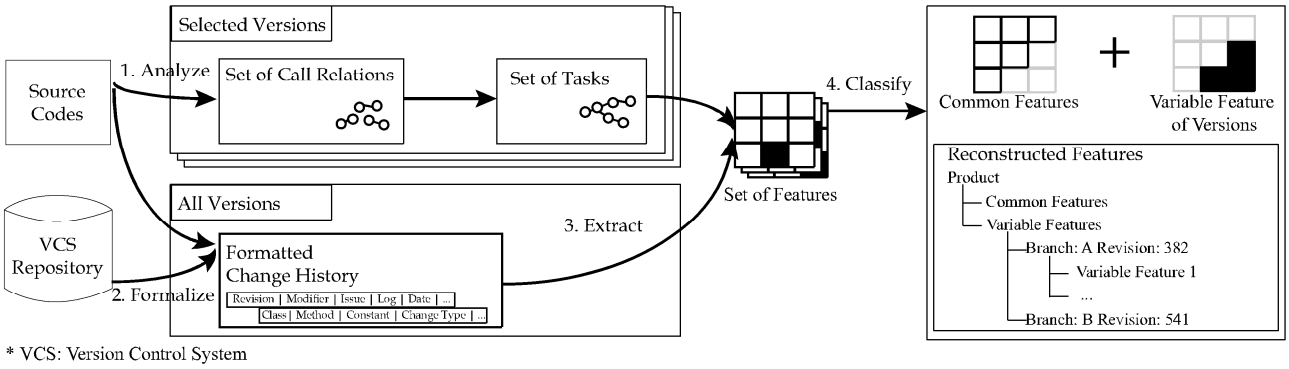


Fig. 1. Overall Process

그림 1과 같이 4가지 단계에 의해 휘처를 추출한다: 1) 선택된 버전의 소스코드 내의 태스크 분석, 2) 모든 버전의 변경 이력 정형화, 3) 휘처 추출 및 재구성, 4) 휘처의 공통성과 가변성 분류

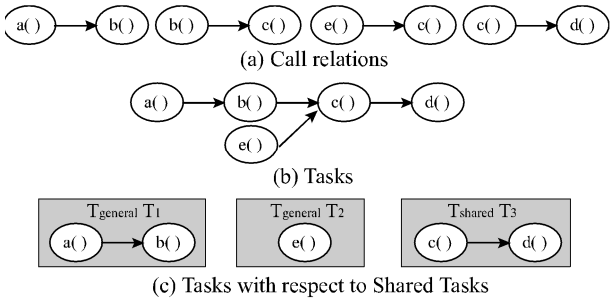
1. Task analysis in selected versions

소스 코드의 문법적(syntax) 관계인 함수(method: M)의 호출관계(call relation: CR)를 고려하여 한 번에 수행되는 트랜잭션을 의미하는 태스크(task: T)를 찾는 단계이다. 태스크는 일반 태스크(general task: $T_{general}$)와 여러 태스크에 의해 함께 사용되는 태스크인 공유 태스크(shared task: T_{shared})가 있으며, 이들은 각각 다음과 같이 정의한다.

- $CR = < M_{caller}, M_{callee} >$
- $T_{general} = CR_n = < M_{caller}, none >$ 이거나 $count(< M_{callee_{CR_n}}, * >) > 1$을 만족하는 $< CR_1, CR_2, ..., CR_n >$
- $T_{shared} = count(< *, M_{caller_{CR_1}} >) > 1$을 만족하는 $< CR_1, CR_2, ..., CR_n >$
- $T = T_{general} \cup T_{shared}$

$T_{general}$는 호출 관계인 CR_i의 순차적 집합이며, CR은 호출하는 함수와 호출되는 함수의 쌍으로 구성된다. $T_{general}$가 종료되는 조건은 CR_n이 호출되는 함수가 존재하지 않는 호출관계인 $< M_{caller}, none >$이거나, T_{shared}를 호출하여 CR_{n+1}의 M_{caller}가 다른 CR의 M_{callee}로 두 번 이상 포함되는 경우이다.

T_{shared}는 CR_1의 M_{caller}가 다른 CR의 M_{callee}로 두 번 이상 포함되는 경우에 해당하는 M_{callee}를 기준으로 시작되는 태스크이다. 예를 들어, 태스크 C가 태스크 A와 태스크 B에 의해 공유된다면, 태스크 A와 태스크 B는 실행 시에 태스크 C를 호출한다는 의미이다. 이를 식별하는 이유는 하나의 역할을 가진 잘 쪼개진 (fine-grained) 태스크로 구성하여 분석의 복잡도를 낮추기 위함이다. 그림 2는 태스크를 분석하는 예제를 보여주고 있다. 그림 2 (a)와 같이 함수 a()와 b(), b()와 c(), e()와 c(), c()와 d()가 호출 관계를 맺고 있다면, 기본적으로 태스크는 그림 2 (b)와 같이 구성된다. 이를 공유 태스크를 고려하여 쪼개게 되면 그림 2 (c)와 같이 두 개의 일반 태스크(T_1, T_2)와 하나의 공유 태스크(T_3)로 구성된다.



ASTFig. 2. Example of Tasks Analysis

2. Change history formalization

분석 대상이 되는 소스 코드와 관련된 변경 이력을 버전 관리 시스템에서 조회하여 분석이 쉽도록 정형화하는 단계이다. 각 리비전마다 변경된 함수 및 변경 타입(추가, 수정, 삭제)을 추출한다.

우선 비교 대상이 되는 두 개의 리비전의 변경된 파일들에 대한 AST(Abstract Syntax Tree) 를 구성한다. 구성된

의 노드들을 비교 분석함으로써 변경된 함수 및 클래스 정보를 추출한다. AST는 컴파일러에서 분석된 내용을 기반으로 하므로 파일의 라인 단위가 아닌 코드 문법 단위의 비교가 가능하다는 이점이 있다. 두 리비전 간의 AST들을 비교하여 얻어지는 정보를 다음과 같이 정의한다.

- $r = \{r_x | x \in$ 탐색해야하는리비전번호들의집합$\}$
- $C(r_x) = \{C_y(r_x) | C_y(r_x) \in$ 리비전r_x에서 변경된C_y클래스들의AST들$\}$
- $C_y(r_x) = \{C_y(r_x).M_z | M_z \in C_y(r_x)$를 만족하는 함수들을 표현하는$C_y(r_x).M_z \in subtree$들$\}$
- $UPDATED =$ 수정된 함수들의 집합, $C_n(r_x).M_z$와$C_m(r_{x-1}).M_z$가동일한루트노드를 가졌지만부분적으로두함수의문장이다른경우
- $INSETED = C_n(r_x).M - C_m(r_{x-1}) - UPDATED$

- $DELETED = C_n(r_{x-1}).M - C_m(r_x).M - UPDATED$
- $CHANGED = UPDATED \cup INSERTED \cup DELETED$

탐색해야 하는 리비전들에서 변경된 파일마다 각각 AST를 생성한다. 변경된 함수들의 집합인 *CHANGED*는 하나의 리비전(r_x)에서 분석된 AST들의 함수 집합과 그 직전 리비전(r_{x-1})에서 분석된 AST들의 함수 집합을 비교하여 구성한다. 이 집합의 함수들은 1) r_{x-1}에서는 존재하지 않았던 함수가 r_x에는 존재하는 경우(*INSERTED*), 2) r_{x-1}에서는 존재하였으나, r_x에는 존재하지 않는 경우(*DELETED*), 3) r_{x-1}와 r_x에 모두 존재하나, 함수의 바디인 문장의 구성이 달라진 경우(*UPDATED*)에 해당하는 함수들이다.

*CHANGED*에 속한 함수 가운데, 공유 태스크의 시작함수가 있고, 이 함수의 시그니처 변경으로 인해 다른 함수들과 함께 변경된 경우 *CHANGED*를 공집합으로 초기화한다. 이 경우는 주로 함수 이름 변경 등의 리팩토링이 발생한 리비전에서 나타난다. 리팩토링은 패턴이 유사한 경우를 일률적으로 작업하므로, 의미적으로 유사해서 함께 변경되었다고 보기 어렵다.

3. Feature extraction

다음 단계는 분석한 태스크들과 정형화된 변경 이력을 토대로 태스크를 그룹핑하여 휘처를 생성하는 단계이다. 태스크는 문법 정보에 기반하여 호출 관계가 있는 함수들의 모음이라고 한다면, 휘처는 의미적 정보에 기반하여 동일 *CCM* (co-committed methods) 에 속하는 함수들이 속한 태스크들의 모음이다. 이는 동적 바인딩이나 다형성과 같은 특성에 의해 하나의 행위가 여러 행위로 분리되는 문제와 하나의 기능이 여러 개의 태스크로 구성될 수 있는 태스크의 불완전성을 해결하기 위한 과정이다.

이를 해결하기 위해서, 한 리비전 내에서 함께 커밋 되었던 함수들의 의미를 확장하여 특정 리비전의 변경된 함수 이력에 속한 함수가 다른 리비전에도 속해있다면, 해당 리비전들의 함수들 역시 *CCM*에 포함한다. 이는 다음과 같이 정의한다.

- $CCM(M_z) = \{M_z \in CHANGED(r_x)$을 만족하는 $CHANGED(r_x)$의 합집합$\}$
- $F = \{ T_x | T_x$의 함수들 $\in CCM(M_z)\}$

그림 3 (a)에서 나타내는 것과 같이, 기존 시스템에 일반 태스크 T_1, T_2, T_4와 공유 태스크 T_3가 있다고 가정한다. 그림 3 (b)와 같이, 리비전 100에서 함수 a(), b()가 변경되었고, 리비전 122에서 함수 b(), c()가 변경되었다면, 함수 a(), c()는 하나의 리비전 내에서 변경된 적이 없더라도 함수 b()에 의해 하나의 *CCM*에 포함되는 것으로 본다. 리비전 124에서 e() 함수는 T_3에서 이름이 수정되고, b() 함수와 g() 함수는 e()함수의 변경된 이름을 반영하기 위하여 수정이 되었다. 그러므로, 리비전 124는 분석 대상에서 제외 되어, *CCM*은 a(), b(), c() 이다. 이를 토대로 태스크를 그룹핑하여 휘처는 그림 3 (c)와 같이 구성 된다. 만약, 리비전 124가 제외되지 않았다면, a(), b(), c(), e(), g()가 하나의 *CCM*의 범주에 속하게 되어 T_1, T_2, T_3, T_4가 모두 하나의 휘처로 구성된다.

4. Feature classification and reconstruction

이전 단계에서 각 버전(v)마다 추출된 휘처는 다음 정의에 의해 두 카테고리로 분류한다: 1) 공통된 휘처(common feature: $F_{commonality}$), 2) 가변적 휘처(variable feature: $F_{variability}$)

$$F_{commonality} = \{ \bigcap_{i=1}^{N} features(v_i) \}$$

$$F_{variability} = \{ \bigcup_{i=1}^{N} [features(v_i) - F_{commonality} \}$$

공통된 휘처는 분석대상이 되는 N개의 버전에 공통적으로 존재하는 휘처이다. 가변적 휘처는 개의 버전 중 하나 이상의 버전에 존재하는 휘처이다. 이 가변적 휘처는 가변성(variability)이 발생한 지점에 따라 휘처, 태스크, 함수의 세 가지 레벨로 다시 분류될 수 있다.

함수 레벨인 경우, 휘처는 가변점(variation point)를 가지며 이는 if-else 구문을 삽입하여 공통 휘처에 위치시킨다. 함수 레벨 외의 경우 변화의 폭이 크므로 각각 프로덕트의 가변 휘처에 위치시킨다. 위치가 변경된 휘처들을 위해 기존 소스 코드의 패키지 구조 및 소스 코드의 위치를 자동 변환한다. 또한 그림 5와 같이 해당 휘처에 변경이 발생했을 시점으로부터 로그 내용을 소스 코드에 주석을 첨부함으로써 개발자가 리팩토링에 대한 근거를 추적할 수 있도록 지원한다.

그림 4은 가변성에 대한 세 가지 레벨을 고려하여 휘처를 분류한 예제를 보여주고 있다. 그림 4 (a)는 이전 단계에서 추출한 각 버전별 휘처이다. 브랜치 A는 메인 버전으로부터 브랜치된 후에 휘처 F_3에 태스크 T_3를 추가하면서 진화하였다. 브랜치 B 역시, 메인 버전으로부터 브랜치된 후에 휘처 F_6이 추가되고, 휘처 F_4의 태스크 T_3에 함수 c()가 추가되면서 변경되었다. 각 버전들(메인 버전, 브랜치 A, 브랜치 B)의 휘처들의 유사도를 비교하여 분류한 결과는 그림 4 (b)와 같다. 공통 휘처에서 휘처 F_4의 b() 함수는 가변점을 고려하여 수정되었다.

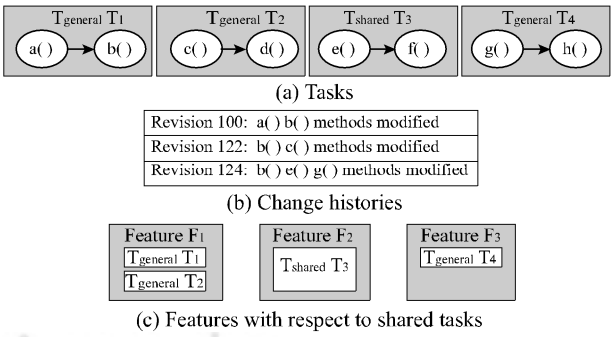


Fig. 3. Example of Feature Extraction

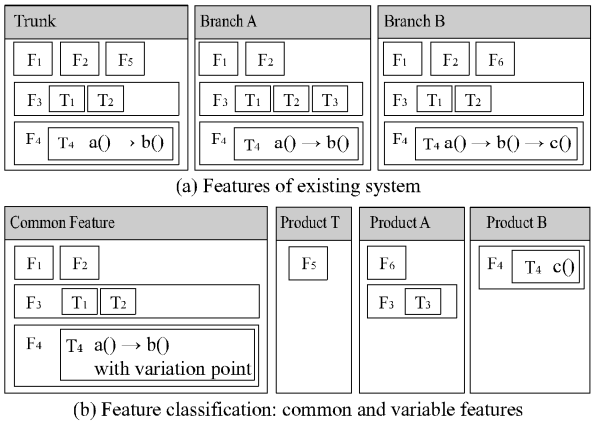


Fig. 4. Example of Classification

```
/** ######### The related revisions of this feature #########
 *  date - revision - author -  log
 *  example)
 * 2001-06-30 - r1234 - Jane.kim - move method(from ... )
 * 2001-07-02 - r1256 - Bob.Lee - modify method by req #1
 * ....
 * */
```

Fig. 5. Example of affected change logs

IV. Case study

본 장에서는, 이클립스 통합툴에서 서브버전(Subversion)을 지원하는 오픈 소스 소프트웨어인 서브클립스(Subclipse)에 제안한 방안을 적용하였다. 서브클립스는 안정화까지 기능이 꾸준히 추가되기도 하였지만, 서브버전의 버전업에 따라 기능들이 변경이 되었다. 이전 서브버전에 사용 가능한 서브클립스는 안정화되기 이전의 것을 사용해야 하는 상황이다. 이에 여러 서브버전에 적용 가능한 서브클립스로 리엔지니어링을 할 수 있도록 분석하였다.

본 연구에서 제안한 기법의 효용성을 검증하기 위해, 자바 개발 경력 8년차, 10년차의 두 전문가가 직접 추출한 휘처들과 제안한 기법으로 추출한 휘처들을 비교하였다. 우선, 표 1은 서브클립스가 실제 본 연구를 적용해보기에 적합한지를 검증하기 위해 릴리즈 노트를 기준으로 릴리즈 정보를 분석한 결과이다. 분석한 정보는 각 리비전마다 수정된 휘처나 버그 (Features or bugs:FB) 수와 각 FB당 변경된 파일 수이다. 이 수치들은 각 리비전마다 얼마나 많은 변경이 영향을 주고 있는지를 의미한다. 분석 결과, 각 버전들마다 기능적 차이가 커서 연구의 효용성을 검증하기에 적합한 것으로 볼 수 있다. 또한, 최근보다는 초기에 변경이 많았음을 알 수 있다. 효율적인 분석을 위해, 임의적으로 초기 릴리즈 중에서 두 리비전을 선택해, 해당 리비전을 기준으로 두 개의 버전이라고 가정하였다. 그리고 이들의 차이를 분석하여 두 버전을 모두 지원하는 하나의 버전을 만들고자 한다.

1. Task analysis in selected versions

선택된 두 버전(리비전 2151, 리비전 2269)의 소스 코드로부터 호출 관계를 분석하여 표 2와 같이 태스크가 추출되었다. 분석 결과, 리비전 2151 의 경우, 전체 호출 관계는 총 10910개, 연관된 호출 관계를 고려하여 추출한 태스크의 수는 1507개이다. 리비전 2269의 경우, 전체 호출 관계는 총 11618, 추출한 태스크의 수는 1574개이다.

Table 2. The selected revisions

Revision	# of call relations	# of tasks
2151	10910	1507
2269	11618	1574

2. Change history formalization

다음으로, 시스템의 변경 이력을 수집하는 단계이다. 원래 수집 범위는 전체 변경 이력을 포함해야 하나, 리비전의 단위(granularity)가 적정한지를 수작업으로 검증하는데 있어 어려움이 있다. 그래서 본 사례 연구에서는 리비전 1989에서 리비전 3493까지를 범위로 하고 각 리비전이 잘 쪼개진(fine-grained) 변경인지를 수작업으로 검증하였다.

분석 결과, *CCM*을 구성하게 하는 리비전 그룹들은 각각 다음과 같이 식별되었다. 동일 그룹에 속한 리비전에서 변경된 함수들은 *CCM*의 구성 요소가 된다.

- 리비전 2034, 리비전 2412
- 리비전 2037, 리비전 2243
- 리비전 2150, 리비전 2181, 리비전 2182, 리비전 2303
- 리비전 2151, 리비전 2250
- 리비전 2263, 리비전 2268

Table 3. The Identified Variable Features

Added methods	133
Modified methods	160
Deleted methods	89
Changed tasks	164
Variability features	27

Table 1. Change information of the system

Releases	# of features or bugs(FB)	# of changed revisions	# of changed files	# of revisions per FB	# of files per FB
0.9.103~1.0.6	9.38	8.23	174.69	4.08	18.62
1.3.8~1.4.8	8.53	19.94	46.35	2.34	5.43
1.8.0~1.8.18	5.47	7.63	16.11	1.39	2.95
Avg.	7.79	21.93	79.05	2.60	9.00

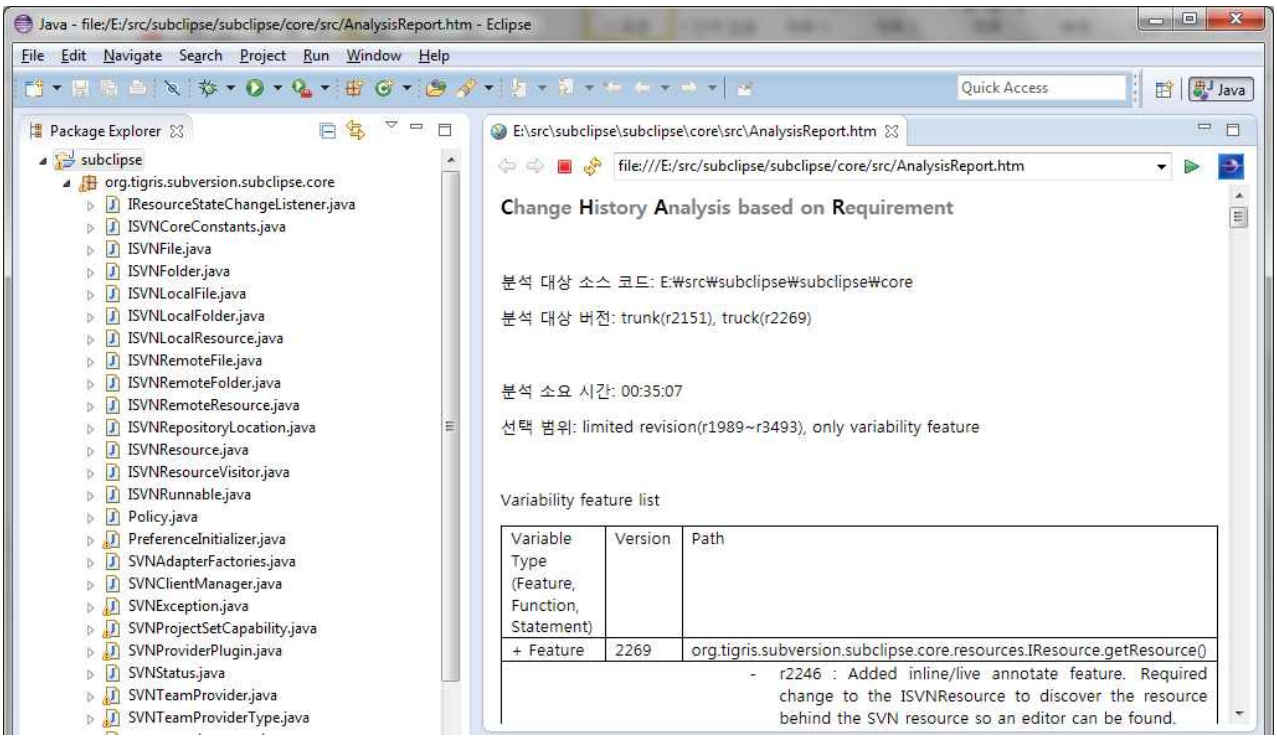


Fig 6. The analysis report of the subclipse

3. Feature extraction

함께 변경된 *CCM*을 토대로 태스크들을 그룹핑하여 버전별 휘처를 추출하였다. 추출된 휘처들은 전문가들이 변경 로그들을 비교하면서 검증하였다. 예를 들어, 리비전 2263과 리비전 2268의 태스크는 함께 그룹핑되었다. 이 리비전들은 Export/Import Preferences 위저드 기능을 변경한 리비전이였다. 리비전 2263은 처음으로 기능이 추가되면서 10개의 태스크가 수정되어 커밋 되었고, 리비전 2268은 버그를 수정 과정에서 2개의 태스크가 수정되었다. 변경 정보에 따르면, 리비전 2263은 의미적으로나 문법적으로 리비전 2268과 관계가 있다. 하나의 리비전에서 변경된 태스크의 숫자는 기능을 추가하거나 버그를 수정하기 위해서 여러 태스크들이 변경 된다는 것을 의미한다.

표 3에서 설명하고 있는 것과 같이, 두 개의 버전을 비교한 결과, 추가된 함수 133개, 수정된 함수 160개, 삭제된 함수 89개가 있었다. 이 함수들은 164개의 태스크들에 영향을 미치고 있었으며, *CCM*을 고려하여 그룹핑하였더니 27개의 가변 휘처들이 식별 되었다. 전문가들이 수작업으로 분석하였을 때, 26개의 가변 휘처가 추출되었다.

수작업과 비교했을 때, 본 연구가 식별하지 못한 1개의 가변 휘처는 리비전 2154때문이다. 리비전 2154는 정적 분석 측면에서의 호출 관계를 가지지 않는 상속 관계에 의한 코드가 수정이 되었다. 그렇기 때문에 전문가의 추출 과정에서는 관계가 있다고 분석되었으나, 제안한 방안에서는 다른 태스크로 인식되었다.

4. Feature classification and reconstruction

앞서 식별된 가변 휘처를 고려하여 재구성한 소스 코드와 간략한 분석 리포트를 그림 6과 같이 제공한다. 분석 리포트는 분석에 소요된 시간 및 분석 대상과 가변 휘처로 분석된 휘처의 리스트를 포함하고 있어, 소스 코드 내에서 이들의 위치를 쉽게 식별할 수 있도록 하였다.

본 사례 연구에서, 한정된 리비전의 제약으로 공통 휘처를 제외한 가변 휘처만이 추출되었다. 기존 연구[11]와 같이 변경 이력을 기반으로 한 *CCM*과 공유 태스크를 고려하지 않은 경우에는 가변 휘처가 20개로 식별되어, 74%의 정확률을 보였다. 그렇지만 제안한 변경 이력과 공유 태스크를 고려한 방안을 적용한 경우, 96%의 정확률을 보임으로서 휘처를 추출하는 것이 보다 효율적임을 보여주고 있다.

V. Conclusion

멀티 버전 관리로 유지 보수가 어려워진 소프트웨어를 프로덕트 라인으로 재공학하기 위해서는 휘처 모델링이 중요하다. 휘처 모델링은 공통 휘처와 가변 휘처를 식별하는 것이 핵심인데, 본 연구에서는 기존 시스템의 소스 코드와 변경 이력을 분석하여 이를 추출하는 기법을 제안하였다. 이를 위해 소스 코드의 정적 호출 관계를 분석하여 태스크를 추출하고, 소스 코드가

변경해온 이력을 통해 휘처를 그룹핑하였다. 본 연구의 기여는 다음과 같다. 첫 번째, 소스 코드로 분석되지 않는 태스크 간의 관계를 개발자의 의도가 담긴 변경 이력을 통해 찾는다. 이는 앞선 연구들[6][8]에서 지적된 한계인 개발자의 공수(ex: 관련 구성 요소 연결, 시나리오 작성) 문제를 해소할 수 있는 방안이 될 수 있다. 두 번째, 추출된 휘처의 변경 이력을 손쉽게 참고할 수 있도록 변경에 대한 근거를 주석으로 제공한다.

향후, 변경 이력의 단위(granularity)에 의해 일어나는 문제를 개선하고자 한다. 변경 이력은 개발자의 습관이나 성향에 따라 단위가 달라지게 되는데, 이로 인해 휘처가 거대하게 그룹핑 될 수 있는 문제를 가지고 있다.

REFERENCES

[1] W. Codenie, N. Gonz´alez-Deleito, J. Deleu, V. Blagojevic, P. Kuvaja, and J. Simil¨a, “Managing Flexibility and Variability: A Road to Competitive Advantage.“ Taylor and Francis, pp. 269–313, 2009

[2] C. W. Krueger, “Easing the transition to software mass customization,” in PFE ’01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering. London, UK: Springer, pp. 282–293, 2001

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz, Object-Oriented Reengineering Patterns - Version of 2009-09-28. Square Bracket Associates, 2009.

[4] R. Glass, Facts and Fallacies of Software Engineering. Addison-Wesley, 2003.

[5] M. P. Robillard and G. C. Murphy, “Representing concerns in source code,” ACM Trans. Softw. Eng. Methodol., vol. 16, no. 1, Feb. 2007.

[6] M. P. Robillard, “Topology analysis of software dependencies,” ACM Trans. Softw. Eng. Methodol., vol. 17, no. 4, pp. 18:1–18:36, Aug. 2008.

[7] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, “Feature-gathering dependency-based software clustering using dedication and modularity,” in ICSM, 2012, pp. 462–471, 2012

[8] T. Eisenbarth, R. Koschke, and D. Simon, “Locating features in source code,” IEEE Trans. Softw. Eng., vol. 29, no. 3, pp. 210–224, Mar. 2003.

[9] B. Adams, Z. M. Jiang, and A. E. Hassan, “Identifying crosscutting concerns using historical code changes,” in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ser. ICSE ’10, pp. 305–314, 2010

[10] Kang, K.C. and Cohen, S.G. and Hess, J.A. and Novak, W.E. and Peterson, A.S., "Feature-oriented domain analysis (FODA) feasibility study", Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, November 1990

[11] C. Nunes, A. Garcia, C. Lucena, and J. Lee, “History-sensitive heuristics for recovery of features in code of evolving program families,” in Proceedings of the 16th International Software Product Line Conference - Volume 1, ser. SPLC ’12, pp. 136–145, 2012.

[12] Subclipse. http://subclipse.tigris.org/

Authors

Seonghye Yoon is a visiting professor of Department of Computer Engineering at Kookmin University and Ph.D. candidate in Department of Computer Science and Engineering at Sogang University.
She received the MS in information processing from Sogang University in 2012, and the BS in Information & Computer Engineering from Inje University in 2003. She worked as a software engineer at WareValley from 2007 until 2011 and also at Spectra from 2002 until 2006. Her research focuses on software evolution, software reuse, software product line engineering and secure software engineering.

Sooyong Park is a Professor of the Department of Computer Science and Engineering at Sogang University. He received his B.S degree in computer science and engineering from Sogang University in 1986 and the M.S. Degree from Florida States University.
And he received his Ph.D. Degree in Information Technology from George Mason University. His research focuses on requirement engineering, financial technology and inter of things.

Mansoo Hwang is a professor in the School of IT Convergence Engineering, Shinhan University, Korea. He received the M.S degree in computer science from Chungang University in 1986 and Ph.D in Computer Engineering from Soongsil University, Korea in 2001.
His main researches are Software Requirement Engineering and Software Quality Management